FlexPDE 6

Version 6.40
9/20/2016

Copyright © 2016 PDE Solutions Inc.

FlexPDE 6

Copyright © 2016 PDE Solutions Inc.

Complying with all copyright laws is the responsibility of the user. Without limiting the rights under copyright, no part of this document may be reproduced, stored in or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, or otherwise) without the express written permission of PDE Solutions Inc.

PDE Solutions Inc. may have patents, patent applications, trademarks, and copyrights or other intellectual property rights covering subject matter in this document. Except as provided in any written license agreement from PDE Solutions Inc., the furnishing of this document does not give you any license to these patents, trademarks, copyrights or other intellectual property.

PDE Solutions, and FlexPDE are either registered trademarks or trademarks of PDE Solutions Inc. in the United States of America and/or other countries.

Products that are referred to in this document may be either trademarks and/or registered trademarks of the respective owners. The publisher and the author make no claim to these trademarks.

While every precaution has been taken in the preparation of this document, the publisher and the author assume no responsibility for errors or omissions, or for damages resulting from the use of information contained in this document or from the use of programs and source code that may accompany it. In no event shall the publisher and the author be liable for any loss of profit or any other commercial damage caused or alleged to have been caused directly or indirectly by this document.

Note :

This version of this manual is current as of the indicated release date. Electronic versions of this manual together with subsequent release notices in the FlexPDE documentation are available online at www.pdesolutions.com. Electronic versions are updated more frequently than printed versions, and may reflect recent developments in FlexPDE more accurately.

Table of Contents

Part I Getting Started 2

1 Installation 2
2 Starting FlexPDE 2
3 FlexPDE Working Files 3
4 The Main Menu Bar 4
- The File Menu 6
- The Controls Menu 7
- The Stop Menu 8

5 The Tool Bar 10
6 Editing Descriptor Files 10
7 Domain Review 12
8 While the Problem Runs 14
9 When the Problem Finishes 18
10 Viewing Saved Graphic Files 19
11 Example Problems 20
12 Registering FlexPDE 21
- The Register Dialog 22
- Internet Key Registration 23
- Dongle Registration 24
- Network Dongle Registration 25
- Software Key Registration 26

Part II User Guide 30

1 Overview 30
- What Is FlexPDE? 30
- What Can FlexPDE Do? 31
- How Does It Do It? 31
- Who Can Use FlexPDE? 32
- What Does A Script Look Like? 33
- What About Boundary Conditions? 34

2 Basic Usage 34
- How Do I Set Up My Problem? 34
- Problem Setup Guidelines 35
- Notation 36

Variables and Equations 36
Mapping the Domain 37
An Example Problem 38
Generating A Mesh 39
Defining Material Parameters 40
Setting the Boundary Conditions 41
Requesting Graphical Output 41
Putting It All Together 42
Interpreting a Script 45
3 Some Common Variations 45
Controlling Accuracy 45
Computing Integrals 46
Reporting Numerical Results 47
Summarizing Numerical Results 47
Parameter Studies Using STAGES 48
Cylindrical Geometry 49
Integrals In Cylindrical Geometry 50
A Cylindrical Example 50
Time Dependence 52
Bad Things To Do In Time Dependent Problems 54
Eigenvalues and Modal Analysis 55
The Eigenvalue Summary 58
4 Addressing More Difficult Problems 58
Nonlinear Coefficients and Equations 58
Complications Associated with Nonlinear Problems 60
Natural Boundary Conditions 61
Some Typical Cases 62
An Example of a Flux Boundary Condition 63
Discontinuous Variables 64
Contact Resistance 65
Decoupling 66
Using JUMP in problems with many variables 67
5 Using FlexPDE in One-Dimensional Problems 68
6 Using FlexPDE in Three-Dimensional Problems 69
The Concept of Extrusion 69
Extrusion Notation in FlexPDE 70
Layering 71
Setting Material Properties by Region and Layer 72
Void Compartments 74

Limited Regions 74

Specifying Plots on Cut Planes 75

The Complete 3D Canister 76

Setting Boundary Conditions in 3D 78

Shaped Layer Interfaces 81

Surface-Generating Functions 83

Integrals in Three Dimensions 84

More Advanced Plot Controls 87

7 Complex Variables 89

The Time-Sinusoidal Heat 90

Interpreting Time-Sinusoidal Results 92

8 Vector Variables 94

Curvilinear Coordinates 95

Magnetic Vector Potential 95

9 Variables Inactive in Some Regions 97

A Chemical Beaker 98

10 Moving Meshes 100

Mesh Balancing 101

The Pulsating Blob 102

11 Controlling Mesh Density 103

12 Post-processing with FlexPDE 105

13 Exporting Data to Other Applications 106

14 Importing Data from Other Applications 108

15 Using ARRAYS and MATRICES 109

16 Solving Nonlinear Problems 110

17 Using Multiple Processors 112

18 Running FlexPDE from the Command Line 113

19 Running FlexPDE Without A Graphical Interface 114

20 Getting Help 114

Part III Problem Descriptor Reference 116

1 Introduction 116

Preparing a Descriptor File 117

File Names and Extensions 117

Problem Descriptor Structure 117

Problem Descriptor Format 118

Case Sensitivity 118

"Include" Files 119
A Simple Example 119
2 The Elements of a Descriptor 121
Comments 121
Reserved Words and Symbols 121
Separators 124
Literal Strings 125
Numeric Constants 125
Built-in Functions 125
Analytic Functions 126
Non-Analytic Functions 126
Unit Functions 128
String Functions 128
The FIT Function 129
The LUMP Function 130
The RAMP Function 130
The SAVE Function 131
The SUM Function 131
The SWAGE Function 132
The VAL and EVAL functions 132
Boundary Search Functions 133
Operators 133
Arithmetic Operators 133
Complex Operators 134
Differential Operators 135
Integral Operators 136
Time Integrals 136
Line Integrals 136
2D Surface Integrals 137
3D Surface Integrals 138
2D Volume Integrals 138
3D Volume Integrals 139
Relational Operators 139
String Operators 140
Vector Operators 140
Tensor Operators 141
Predefined Elements 142
Expressions 143
Repeated Text 144
3 The Sections of a Descriptor 145
Title 145
Select 145
Mesh Generation Controls 146

Solution Controls 147
Global Graphics Controls 150
Coordinates 153
Variables 154
The THRESHOLD Clause 155
Complex Variables 156
Moving Meshes 156
Variable Arrays 156
Vector Variables 157
Global Variables 157
Definitions 158
ARRAY Definitions 159
MATRIX Definitions 161
Function Definitions 163
STAGED Definitions 164
POINT Definitions 165
TABLE Import Definitions 165
The TABLE Input function 166
The TABLEDEF input statement 167
TABLE Modifiers 167
TABLE File format 168
TABULATE definitions 169
TRANSFER Import Definitions 169
TRANSFER File format 170
The PASSIVE Modifier 172
Mesh Control Parameters 173
Initial Values 174
Equations 174
Association between Equations, Variables and Boundary Conditions 175
Sequencing of Equations 175
Modal Analysis and Associated Equations 176
Moving Meshes 177
Constraints 178
Extrusion 179
Boundaries 180
Points 181
Boundary Paths 181
Regions 183
Reassigning Regional Parameters 184
Regions in One Dimension 184
Regions in Three Dimensions 185
Regional Parameter Values in 3D 185
Limited Regions in 3D 186
Empty Layers in 3D 186

Excludes ... 187
Features ... 187
Node Points ... 187
Ordering Regions ... 188
Numbering Regions ... 188
Fillets and Bevels ... 189
Boundary Conditions ... 189
Syntax of Boundary Condition Statements ... 190
Point Boundary Conditions ... 190
Boundary conditions in 1D ... 191
Boundary Conditions in 3D ... 191
Jump Boundaries ... 192
Periodic Boundaries ... 193
Complex and Vector Boundary Conditions ... 194
Front ... 194
Resolve ... 195
Time ... 196
Monitors and Plots ... 197
Graphics Display and Data Export Specifications ... 197
Graphic Display Modifiers ... 200
Controlling the Plot Domain ... 206
Reports ... 208
The ERROR Variable ... 209
Window Tiling ... 209
Monitors in Steady State Problems ... 209
Monitors and Plots in Time Dependent Problems ... 209
Hardcopy ... 210
Graphics Export ... 210
Examples ... 210
Histories ... 211
End ... 212
4 Batch Processing ... 212
Part IV Electromagnetic Applications 214
1 Introduction ... 214
Finite Element Methods ... 214
Principles ... 214
Boundary Conditions ... 215
Integration by Parts and Natural Boundary Conditions ... 216
Adaptive Mesh Refinement ... 217
Time Integration ... 217
Summary ... 218
2 Electrostatics ... 218

Electrostatic Fields in 2D ... 219
Electrostatics in 3D ... 223
Capacitance per Unit Length in 2D Geometry ... 225
3 Magnetostatics ... 230
A Magnet Coil in 2D Cylindrical Coordinates ... 231
Nonlinear Permeability in 2D ... 234
Divergence Form ... 238
Boundary Conditions ... 239
Magnetic Materials in 3D ... 239
4 Waveguides ... 245
Homogeneous Waveguides ... 246
TE and TM Modes ... 247
Non-Homogeneous Waveguides ... 251
Boundary Conditions ... 252
Material Interfaces ... 253
5 References ... 257

Part V Technical Notes 260

1 Natural Boundary Conditions ... 260
2 Solving Nonlinear Problems ... 261
3 Eigenvalues and Modal Analysis ... 263
4 Avoid Discontinuities! ... 263
5 Importing DXF Files ... 265
6 Extrusions in 3D ... 265
7 Applications in Electromagnetics ... 270
8 Smoothing Operators in PDE's ... 277
9 3D Mesh Generation ... 279
10 Interpreting Error Estimates ... 280
11 Coordinate Scaling ... 282
12 Making Movies ... 284
13 Converting from Version 4 to Version 5 ... 284
14 Converting from Version 5 to Version 6 ... 285

Part VI Sample Problems 288

1 applications ... 288
chemistry ... 288
chemburn ... 288
melting ... 290

reaction ... 291
control ... 293
control_steady ... 293
control_transient ... 294
electricity ... 295
3d_capacitor ... 295
3d_capacitor_check ... 296
3d_dielectric ... 298
capacitance ... 299
dielectric ... 300
fieldmap ... 300
plate_capacitor ... 301
space_charge ... 302
fluids ... 303
1d_eulerian_shock ... 303
1d_lagrangian_shock ... 304
2d_eulerian_shock ... 305
2d_piston_movingmesh ... 306
3d_flowbox ... 308
3d_vector_flowbox ... 309
airfoil ... 311
black_oil ... 312
buoyant+time ... 313
buoyant ... 315
channel ... 317
contaminant_transport ... 318
coupled_contaminant ... 319
flowslab ... 321
geoflow ... 322
hyperbolic ... 323
lowvisc ... 324
swirl ... 325
vector_swirl ... 327
viscous ... 329
groundwater ... 330
porous ... 330
richards ... 331
water ... 332
heatflow ... 333
1d_float_zone ... 333
3d_bricks+time ... 333
3d_bricks ... 335
axisymmetric_heat ... 336
float_zone ... 337

heat_boundary 338
radiation_flow 340
radiative_boundary 341
slider 341
lasers 343
laser_heatflow 343
self_focus 344
magnetism 346
3d_magnetron 346
3d_vector_magnetron 347
helmholtz_coil 349
magnet_coil 350
permanent_magnet 352
saturation 352
vector_helmholtz_coil 354
vector_magnet_coil 356
misc 357
diffusion 357
minimal_surface 358
surface_fit 359
stress 360
3d_bimetal 360
anisotropic_stress 362
axisymmetric_stress 364
bentbar 366
elasticity 367
fixed_plate 370
free_plate 371
harmonic 373
prestube 375
tension 376
vibrate 378
2 usage 380
2d_integrals 380
fillet 381
fit+weight 382
function_definition 382
ifthen 383
lump 384
polar_coordinates 384
repeat 385
save 386

spacetime1 387
spacetime2 388
spline_boundary 389
staged_geometry 389
stages 390
stage_vs 391
standard_functions 391
sum 392
swage_pulse 393
swage_test 393
tabulate 394
tintegral 395
two_histories 396
unit_functions 397
vector_functions 397
1D 398
- 1d_cylinder 398
- 1d_cylinder_transient 399
- 1d_float_zone 400
- 1d_slab 400
- 1d_sphere 401

3D_domains 401
- 2d_sphere_in_cylinder 401
- 3d_box_in_sphere 402
- 3d_cocktail 403
- 3d_cylspec 404
- 3d_ellipsoid 405
- 3d_ellipsoid_shell 405
- 3d_extrusion_spec 406
- 3d_fillet 407
- 3d_helix_layered 408
- 3d_helix_wrapped 410
- 3d_integrals 412
- 3d_lenses 413
- 3d_limited_region 414
- 3d_pinchout 415
- 3d_planespec 416
- 3d_pyramid 417
- 3d_shell 418
- 3d_shells 419
- 3d_sphere 421
- 3d_spherebox 422

3d_spherespec 422
3d_spool 423
3d_thermocouple 424
3d_toggle 425
3d_torus 427
3d_torus_tube 427
3d_twist 429
3d_void 431
regional_surfaces 432
tabular_surfaces 433
two_spheres 434
twoz_direct 434
twoz_export 435
twoz_import 436
twoz_planar 438
two_spheres 439
accuracy 440
forever 440
gaus1d 441
gaus2d 442
gaus3d 442
sine1d 443
sine2d 444
sine3d 445
arrays+matrices 446
arrays 446
array_boundary 446
matrices 447
matrix_boundary 448
complex_variables 449
complex+time 449
complex_emw21 449
complex_variables 450
sinusoidal_heat 450
constraints 451
3d_constraint 451
3d_surf_constraint 453
boundary_constraint 454
constraint 454
coordinate_scaling 455
scaled_z 455
unscaled_z 456
discontinuous_variables 457
3d_contact 457
3d_contact_region 459

contact_resistance_heating 460
thermal_contact_resistance 461
transient_contact_resistance_heating 462
eigenvalues 463
3d_oildrum 463
3d_plate 464
drumhead 465
drumhole 466
filledguide 467
shiftguide 468
vibar 469
waveguide 471
waveguide20 472
import-export 473
3d_mesh_export 473
3d_mesh_import 473
3d_post_processing 474
3d_surf_export 475
blocktable 476
export 477
export_format 477
export_history 478
mesh_export 479
mesh_import 480
post_processing 481
splinetable 482
table 482
tabledef 483
table_export 484
table_import 484
transfer_export 485
transfer_import 485
mesh_control 487
3d_curvature 487
boundary_density 488
boundary_spacing 488
front 489
mesh_density 490
mesh_spacing 490
resolve 491
moving_mesh 492
1d_stretchx 492
2d_Lagrangian_shock 493
2D_movepoint 494

2d_position_blob ... 495
2d_stretch_x ... 496
2d_stretch_xy ... 497
2d_velocity_blob ... 498
3d_position_blob ... 499
3d_velocity_blob ... 501
ode ... 502
linearode ... 502
nonlinode ... 503
second_order_time ... 504
periodicity ... 504
3d_antiperiodic ... 504
3d_xperiodic ... 505
3d_zperiodic ... 507
antiperiodic ... 507
azimuthal_periodic ... 508
periodic+time ... 509
periodic ... 510
two-way_periodic ... 511
plotting ... 512
3d_ploton ... 512
plot_on_grid ... 513
plot_test ... 514
print_test ... 515
regional_variables ... 516
regional_variables ... 516
sequenced_equations ... 517
theneq+time ... 517
theneq ... 518
stop+restart ... 519
restart_export ... 519
restart_import ... 520
variable_arrays ... 521
array_variables ... 521
vector_variables ... 522
vector+time ... 522
vector_lowvisc ... 522
vector_variables ... 523
Index 525

Part I

Getting Started

1 Getting Started

This section presents an overview of how to install and interact with FlexPDE on your computer. It does not address the issues of how to pose a partial differential equations problem in the scripting language of FlexPDE. These issues are addressed in the sections User Guide 30 and Problem Descriptor Reference 116.

1.1 Installation

The general principles of installation for FlexPDE are the same across all platforms: the set of installation files must be extracted from the compressed distribution archive and placed in the system file hierarchy. The details of how this is done vary with computer platform.

There are two media options for FlexPDE installation:

- **Installation from CDROM.** Your documentation package should include printed Installation instructions. An electronic version of these instructions will be found in the individual operating system folders on the CDROM.

- **Installation from Internet download.** Click the file name of the desired version, and store the downloaded file at a convenient place in your file system. For more information, click the "Installation" link next to the version download you have chosen.

1.2 Starting FlexPDE

Windows

The FlexPDE installation program will place a FlexPDE icon on your desktop. You can start FlexPDE merely by double-clicking this icon. Alternatively, you can use the File Manager to navigate to the folder where FlexPDE was installed, and then double-click on the FlexPDE executable.

The installation program will also create an association of the "**.pde"** extension with the installed FlexPDE executable, so that FlexPDE can be started merely by double-clicking a script file in the file manager.

Mac OSX

FlexPDE is installed in the "Applications | FlexPDE6" folder by default, but you can choose to install it in any location you wish. Navigate to this folder and open the flexpde6 application.

The installation program will also create an association of the "**.pde"** extension with the installed FlexPDE executable, so that FlexPDE can be opened merely by double-clicking a script file in the Finder.

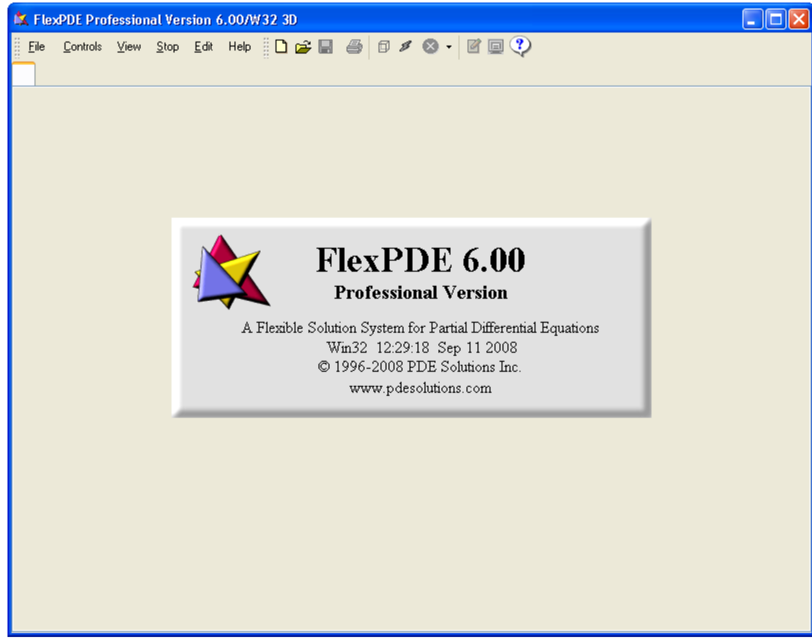
Linux

FlexPDE is installed in the directory you choose when extracting the archive. You can start FlexPDE by typing a command line in a console window, or from the file manager by navigating to the installation directory and opening the flexpde6 application.

Association of the **".pde"** extension with flexpde6 can be made manually using the standard procedures of the operating system. You can also place a FlexPDE icon on your desktop using the "fpde6icon.png" file included in the installation files.

The Sign-On Screen

Whatever method you use to invoke FlexPDE, you will see a screen like this:



The display banner reports "FlexPDE", the version number and date of creation of the running version of FlexPDE.

Whenever a license has been acquired*, the display banner will show the class of the user's license (Student or Professional). The window caption bar will also report the platform version and license level, with "1D", "2D" or 3D", depending on the licensing level of the running program. Temporary licenses will display the time remaining in the license.

The window presents a standard menu bar and a tool bar, most items of which at this point are disabled.

----

* Note: Software keys and dongle licenses are read at invocation of FlexPDE. Network licenses are not read until a problem is run; at that time, a license of the required level, 1D, 2D or 3D will be requested from the network.

1.3 FlexPDE Working Files

FlexPDE works with an assortment of files differing in the file extension. All have the structure <problem name>.<extension>, where <problem name> is the unique identifier for the model being run. The meaning of the most commonly used extensions are described below. Other file extensions can be created and used in other circumstances as described later in the documentation.

Input

.PDE

FlexPDE reads a model description from a script file with the extension ".pde". This file is created by the user and contains the full description of the model to be run. The name of this file establishes the

<problem name> used by the other files. This is an ordinary text file and can be opened with any text editor. This file should not be modified by formatting editors like Word as they may insert illegal characters.

Output

.PG6

FlexPDE writes primary graphical output into a file with the extension ".pg6". This file can be viewed later and used to print or export graphical data to various other formats. The format of this file is unique to FlexPDE and cannot be read by other programs.

.LOG

FlexPDE writes a summary of the progress of each run into a file with the extension ".log". This file contains information about time steps, error estimates, memory use and other data. This is an ordinary text file and can be opened with any text editor.

.DBG

FlexPDE writes a more elaborate summary of each run into a file with the extension ".dbg". This file is sometimes useful in determining errors or locating trouble spots in the domain. This is an ordinary text file and can be opened with any text editor.

.EIG

In eigenvalue problems, FlexPDE writes a summary of final system eigenvalues into a file with the extension ".eig". This is an ordinary text file and can be opened with any text editor.

Note: By default Windows hides the file name extensions, relying on distinctive icons to indicate file type. Windows can be configured to show file extensions and we encourage users to do this. FlexPDE has unique icons for ".pde" and ".pg6" files, but not for the other files.

The DELETE Selector

DELETE (extension, ...) included in the SELECT 145 section will cause FlexPDE to delete the specified files <problem name>.extension when the ".pde" file is closed. For example, DELETE (dbg, log) will delete the associated ".dbg" and ".log" files.

1.4 The Main Menu Bar

The items of the main menu present many of the conventional functions of graphical applications. The availability and precise meaning of these menu items depends on the current state of processing of the problem. We summarize the menu items here, and describe them in more detail in the following sections.

File

The "File" menu item allows you to begin operation by opening a problem descriptor file, importing a DXF file, or viewing previously stored graphical output from a FlexPDE run. It also allows you to save your work or exit the application. These operations are performed using standard dialogs of the computer operating system. (See "The File Menu" 6);

Controls

This menu contains an assortment of functions that may be performed during the generation and running of a problem descriptor, such as running the script or switching between edit and plot modes. (See "The Controls Menu" 7)

View

When a stored FlexPDE graphics file has been opened, the View menu item will present a menu of options for controlling the display of the stored images. (See "Viewing Saved Graphic Files" 19)

Stop

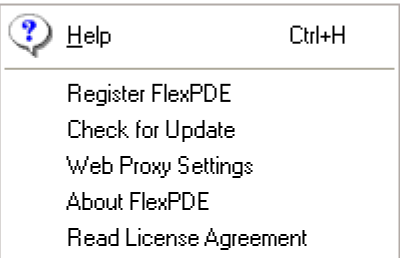
While a problem is being run, the Stop menu item will display a selection of termination strategies of various levels of urgency. (See "The Stop Menu" 8)

Edit

When a descriptor is being edited, this menu provides standard editing commands. (See "Editing Descriptor Files" 10)

Help

The Help menu contains six items as shown below:



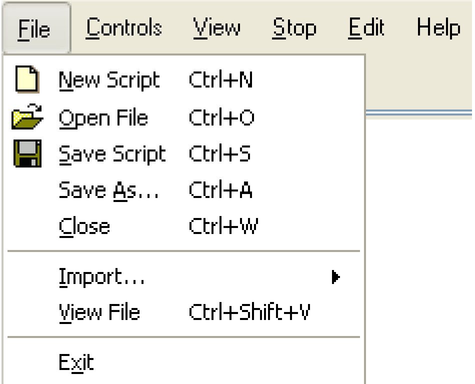
On Windows, the "Help" sub-item will initiate the help system.
On Mac and Linux, you must manually initiate your browser and direct it to "Help | Html | Index.html" in the FlexPDE installation directory.

- The "Register FlexPDE" sub-item allows you to inspect or modify the FlexPDE license registration. (See "Registering FlexPDE" 21)
- The "Check for Update" sub-item will contact the PDE Solutions website and determine whether later updates are available. Updates will not be automatically downloaded or installed. This check is performed automatically on a random basis when you run FlexPDE (approximately 5% of the time.) To bypass this auto check, manually modify the "flexpde6.ini" file with "[UPDATECHECK] 0". The file can be found in the user's "flexpde6user" directory.
- The "Web Proxy Settings" sub-item allows you to set relevant information about your Proxy Server, if you have one.
- The "About FlexPDE" sub-item redisplays the sign-on screen. Note that on Mac this item appears in the FlexPDE "Application" menu.
- The "Read License Agreement" sub-item displays the End-User Licence Agreement.

Note: *On Windows and Linux, the menu bar can be detached and moved to a different part of the screen.*

1.4.1 The File Menu

The File Menu allows the creation of new files, opening existing files, saving and closing active problems, importing DXF files and viewing saved graphics:



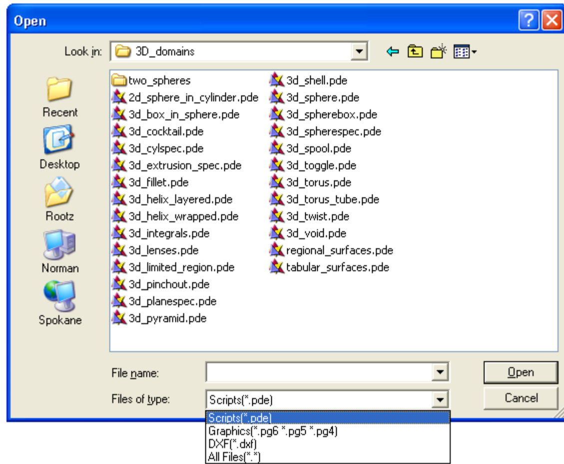
The menu items have the following functions:

New Script

Use this menu item to create a new problem descriptor file (or "script"). FlexPDE will initialize the descriptor with the most common section headings. In most cases, it will be more convenient to create a new descriptor by editing an existing one which is close in function to the new problem.

Open File

This menu item can be used to open an existing descriptor file (either to modify it or to run the problem), to open a stored graphics file for viewing, or to open a DXF file for import. A standard Open_File dialog will appear. Navigate to the folder which contains the descriptor you wish to open. For example, navigating to the standard samples folder "Samples | Usage | 3D_domains" will display the following screen:



(If your system is configured to hide file extensions, you will not see the ".pde" part of the filenames, but you can still recognize the FlexPDE icon.)

The default display shows script files (.pde extension). You can select other file types using the dropdown "Files of Type" list. (On Macintosh or Linux, the selection of alternate file types is slightly different, but follows the customary methods for the operating system.)

Double-click on the file of your choice, or single-click and click Open. See the following section "Editing Descriptor Files" 10 for the rest of the story.

A new tab will be displayed, showing the name of the selected problem file. You can switch between tabs at will.

You can open as many descriptors as you wish, and any number of them can be running at the same time.

Save Script

Use this menu item to save a descriptor which you have modified. The currently displayed file is saved in place of the original file. This function is automatically activated when a problem is Run.

Save As

Use this menu item to save to a *new file name* a descriptor which you have modified. The original source file will remain unchanged.

Close

Use this menu item to remove the currently displayed problem and disconnect from the associated files.

Import

Use this menu item to import descriptors from other formats. The only option available at this time is "DXF", which will import a descriptor from AutoCad version R14. See the Technical Note "Importing DXF Files" 265 for more information. (This function is the same as "Open File" with the DXF file type selected.)

View

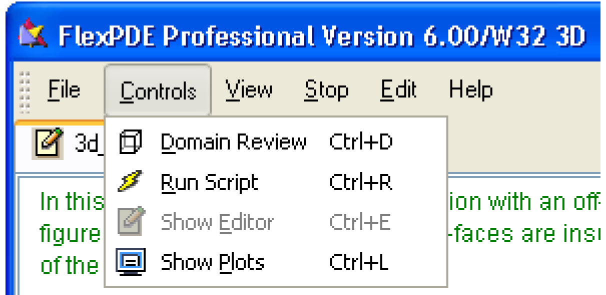
Use this menu item to open a file of saved graphical output from a FlexPDE problem which was run and completed at an earlier time. A standard Open_File dialog will appear. Navigate the folder containing the desired ".pg6" file. Double-click on the file of your choice, or single-click and click Open. See the following section "Viewing Saved Graphics Files" 19 for more information. You may View more than one saved problem, and you may open files for viewing while other descriptors are open, but you should not open the same problem for simultaneous viewing and running, since file access conflicts may occur. (This function is the same as "Open File" with the "Graphics" file type selected.)

Exit

Click here to terminate your FlexPDE session. All open descriptors and Views will be closed. If changes have been made and not saved, you will be prompted.

1.4.2 The Controls Menu

The Controls menu presents several optional functions for processing descriptors.



FlexPDE has two different operating modes, Edit and Plot. When in edit mode, the text of the current discriptor is displayed for editing. When in Plot mode, graphics are displayed, either the monitors and plots being constructed as a problem runs, or the final state of plots when a run is completed.

Domain Review

This is a modified form of the "Run" item. When FlexPDE is in Edit mode, the Domain Review menu item will begin processing the displayed problem descriptor, halting at various stages of the mesh generation to display the current state of the mesh construction. This is an aid to constructing problem domains. (See topic "Domain Review" 12 below.)

Run

When FlexPDE is in Edit mode, the Run menu item will begin processing of the displayed problem descriptor. Execution will proceed without interruption through the mesh generation, execution and graphic display phases. (See topic "While the Problem Runs" 14 below.)

Show Editor

When a problem is in Plot mode with graphics being displayed, the Show Editor menu item will enter Edit mode and display the current problem text. (See topic "Editing Scripts" 10 below.) If the problem is stopped or has not yet been run, the tab will show the icon. If the problem is running while the editor is displayed, the icon will display on the problem tab.

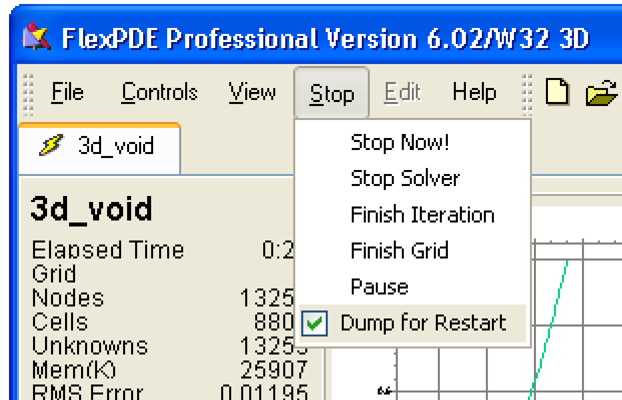
Show Plots

When a problem is in the Edit mode, the Show Plots menu item will switch to Plot mode and display the current state of the problem graphics. (See topic "While the Problem Runs" 14 below.)

1.4.3 The Stop Menu

When a problem is running, it is sometimes necessary to request an abnormal termination of the solution process. This may be because the user has discovered an error in his problem setup and wishes to modify it and restart, or because the solution is satisfactory for his needs and additional computation would be unnecessary.

The Stop menu provides several ways to do this, with the most imperative controls at the top, descending to less immediate terminations:



The contents of this menu will depend on the type of problem that is being run. Below are the most common.

Stop Now!

This is a panic stop that causes processing to be interrupted as soon as possible. No attempt is made to complete processing or write output. You will be given a chance to change your mind:

If you click "No", the "Stop Now!" will be ignored.

Finish Iteration

At the conclusion of the current iteration phase, the processing will be completed as if convergence had been achieved. Final plots will be written, and FlexPDE will halt in Plot mode.

Finish Grid

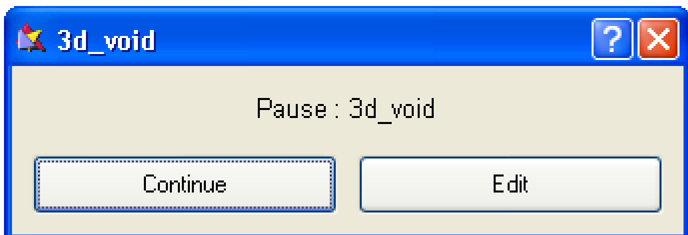
Processing will continue until convergence requirements have been met for the current mesh. No additional adaptive mesh refinement will be attempted, and the problem will terminate as if final convergence had been achieved. Final plots will be written, and FlexPDE will halt in Plot mode.

Finish Stage

In a "Staged" 48 problem (q.v.), the current stage will be completed, including any necessary mesh refinement. Final plots will be written for the current stage, but no more stages will be begun. FlexPDE will halt in Plot mode.

Pause

FlexPDE will stop processing and go into an idle state waiting for a mouse click response to the displayed dialog:



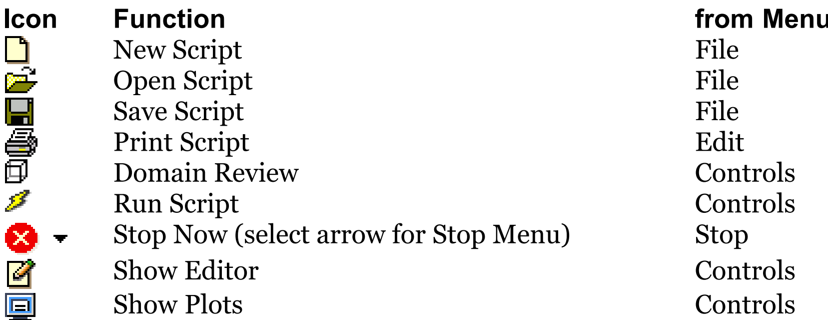
"Continue" will resume processing at the point where it was interrupted. "Edit" will terminate processing as if "Stop Now!" had been clicked. This function can be used to temporarily free computer resources for a more important task without terminating the FlexPDE run.

Dump for Restart

Selecting this checkbox will cause FlexPDE to save a TRANSFER 169 file after another entry in the Stop menu is selected. See the example "Restart_Export.pde" 519. Note: TRANSFER files do not save the state of HISTORY plots, so restarted problems will have fragmented Histories.

1.5 The Tool Bar

The buttons in the tool bar replicate some of the common entries in the various menus:



Icon	Function	from Menu
	New Script	File
	Open Script	File
	Save Script	File
	Print Script	Edit
	Domain Review	Controls
	Run Script	Controls
	Stop Now (select arrow for Stop Menu)	Stop
	Show Editor	Controls
	Show Plots	Controls

The tool bar icons also appear on the menu bar entries with corresponding function.

Note: *The tool bar can be detached and moved to another part of the screen.*

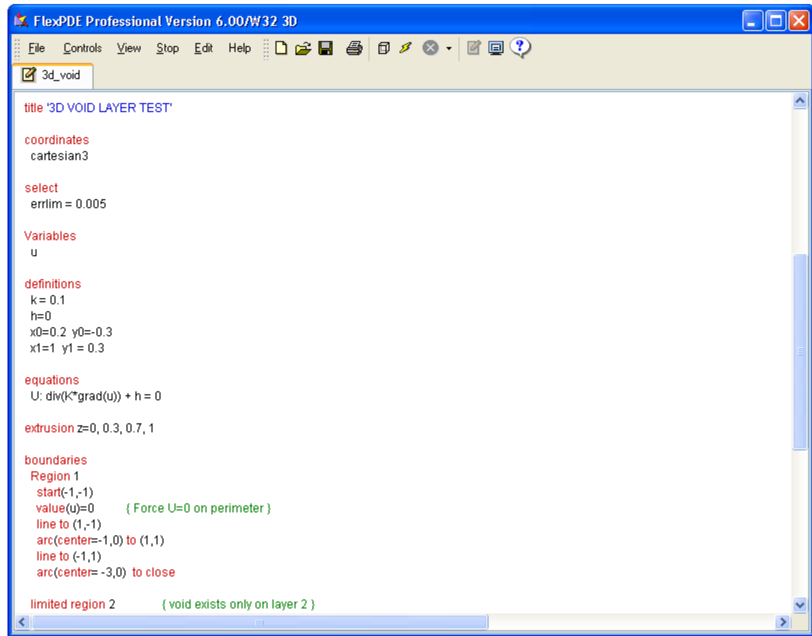
1.6 Editing Descriptor Files

A FlexPDE problem descriptor file is a complete description of the PDE modeling problem. It describes the system of partial differential equations, the parameters and boundary conditions used in the solution, the domain of the problem, and the graphical output to generate. See the section "User Guide 30" for a tutorial on the use of FlexPDE problem descriptors. See the section "Problem Descriptor Reference 116" for a complete description of the format and content of the descriptor file.

You can open a descriptor file in either of two ways: 1) by running FlexPDE from the desktop icon or from your file manager program, and then following the "File|Open" menu sequence; or 2) if an association of FlexPDE with the ".pde" extension has been made, either automatically in Windows or manually in other operating systems, you can double-click on the .pde file in your file manager. In either case, the descriptor file will be opened, a new tab will be created, and an edit window will appear.

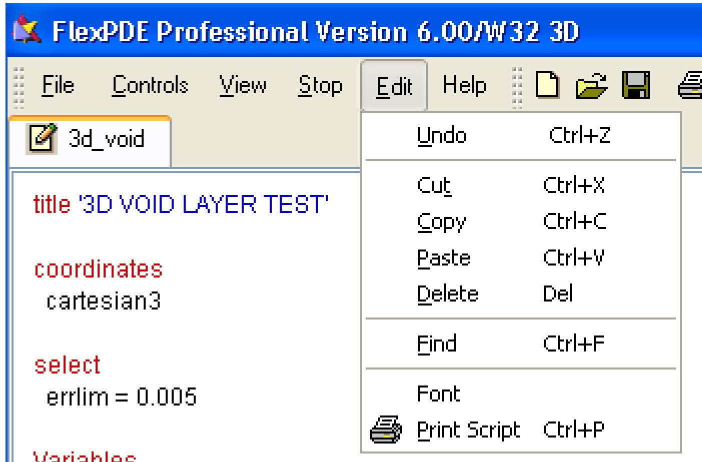
For example, suppose we follow the "Open" sequence to the "Samples | Misc | 3D_domains" folder and select "3d_void.pde". The newly opened problem file will be recorded in a tab along the top of the window, allowing it to be selected if a number of scripts are open simultaneously.

The Edit window appears as follows:



This is a standard NOTEPAD-type editing window, showing the contents of the selected descriptor. You can scroll and edit in the usual way. FlexPDE keywords are highlighted in red, comments in green, and text strings in blue.

The "Edit" item in the main menu contains the editing functions:



The menu items have the conventional meanings, and the control key equivalents are shown.
The Find, Font and Print items have the following use:

Find

This item allows you to search the file for occurrences of a string. The search will find imbedded patterns, not just full words.

Font

This item allows you to select the display font for the editor. Your selection will be recorded and used in subsequent FlexPDE sessions.

Print Script
Prints the descriptor file to a configured printer.

In addition to the main menu Edit item, you can *right-click* the text window to bring up the same editing menu.

At any time, you can click "File | Save" or "File |Save_As..." in the main menu to save your work before proceeding.

Now click "Domain Review" or "Run Script" in the Controls menu, and your problem will begin execution. The file will be automatically saved in the currently open file, so if you wish to retain the unedited file, you must use "Save_As" before "Run".

> ***Note**: The FlexPDE script editor is a "programming" editor, not a word processor. There are no sophisticated facilities for text manipulation.*

1.7 Domain Review

The "Domain Review" menu item is provided in the Controls menu as a way to validate your problem domain before continuing with the analysis.

2D Problems

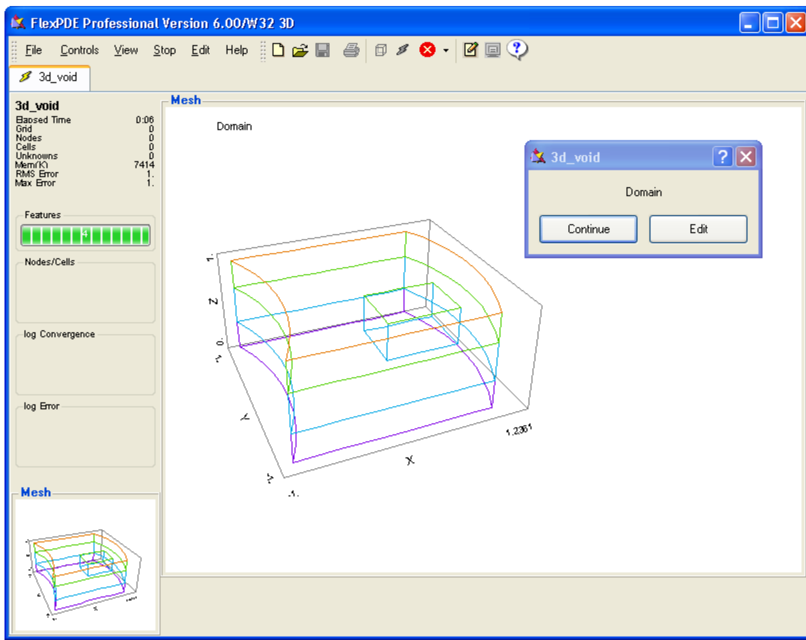
When you click "Domain Review", the descriptor file will be saved to disk, and the domain construction phase will begin. FlexPDE will halt with a display of the domain boundaries specified in the descriptor. If these are as you intended, click "Continue". If they are not correct, click "Edit", and you will be returned to the edit phase to correct the domain definition. If you continue, the mesh generation process will be activated, and FlexPDE will halt again to display the final mesh. Again, you can continue or return to the editor.

3D Problems

The 3D domain review is more extensive. Echoing the mesh generation process used in FlexPDE, the review will halt after each of the following stages:

- A domain plot showing the boundaries of each extrusion surface and layer in order from lower to higher Z coordinate. The surface plots show the boundaries that exist in the surface. The layer plot shows the boundaries that extend through the layer and therefore form material compartments. If at any point you detect an error, you can click "Edit" to return to the editor and correct the error.

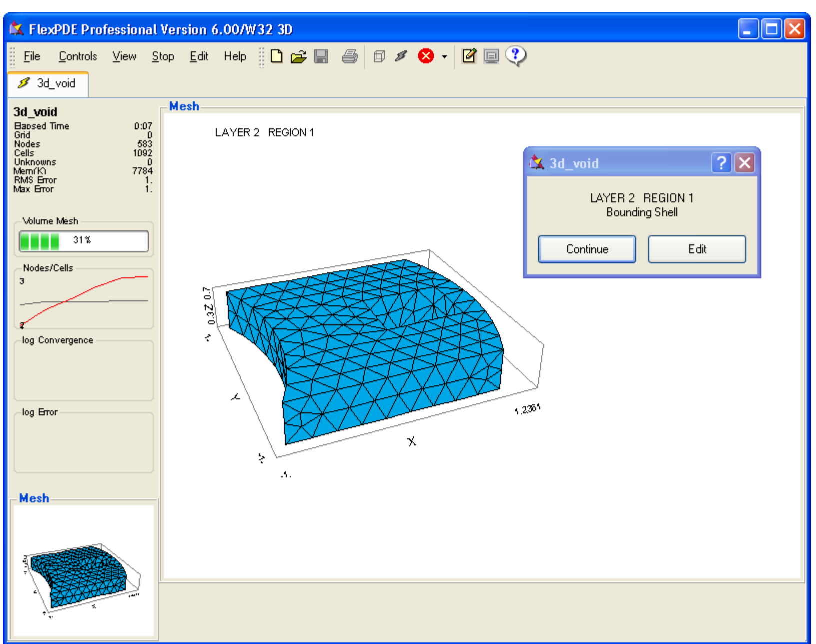
- After the display of individual surfaces and layers, you will be presented a composite view of all the boundaries of the domain, which might look like this:



- Once the domain boundaries are correct, FlexPDE will proceed to the generation of the 2D finite element mesh for each extrusion surface. These will be displayed in order from lower to upper surfaces. You can return to "Edit" after any surface.

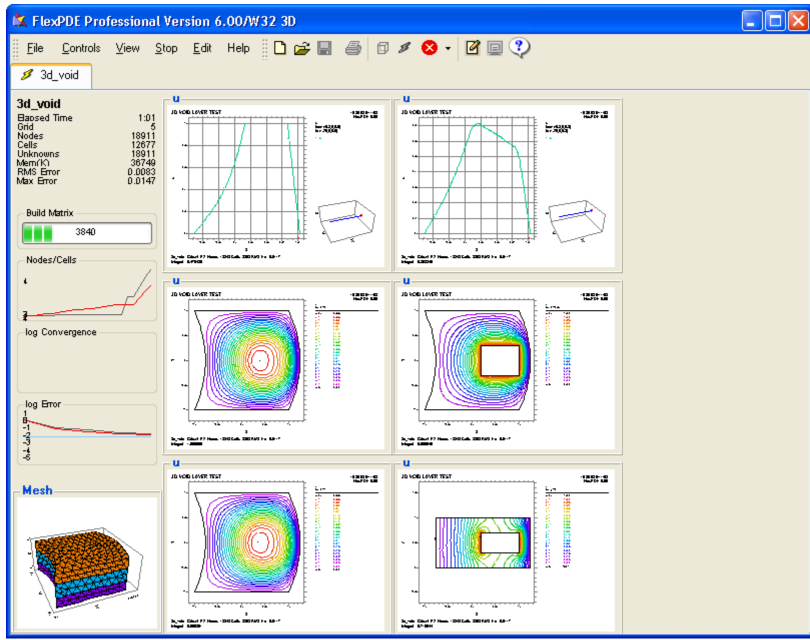
- Once the surface meshes are correct, FlexPDE will proceed to the generation of the 3D finite element mesh. Each subregion of the first layer will be displayed and meshed. When the layer is complete, the full layer will be displayed. When all layers are complete, the full 3D mesh will be displayed. You can return to "Edit" at any point.

A 3D "Domain Review" plot might look like this:



1.8 While the Problem Runs

Whether you click "Run" or proceed through the "Domain Review", once the problem begins running, the icon on the problem tab will change from the edit icon ( ) to the Run icon ( ). The screen will look something like this:



The STATUS Panel

On the left is the "Status Panel", which presents an active report of the state of the problem execution. It contains a text based report, a progress bar for the current operation, several history plots summarizing the activity, and a "Thumbnail" window of the current computational grid.

The history plots are new in version 6. They summarize the number of nodes/cells in the mesh, the convergence of the current solver, the error estimates for the solution, and the current time step (in the case of time dependent problems).

The format of the printed data will depend upon the kind of problem, but the common features will be:

- The elapsed computer time charged to this problem.
- The current regrid number.
- The number of computation Nodes.
- The number of Finite Element Cells.
- The number of Unknowns (nodes times variables).
- The amount of memory allocated for working storage (in KiloBytes).
- The current estimate of RMS (root-mean-square) spatial error.
- The current estimate of Maximum spatial error in any cell.

Other items which may appear are:

- The current problem time and timestep

- The stage number
- The RMS Solution error for the most recent iteration
- The iteration count
- The convergence status of the current iteration
- A report of the current activity

The PLOT Windows

On the right side of the screen are separate "Thumbnail" windows for each of the PLOTS or MONITORS requested by the descriptor.

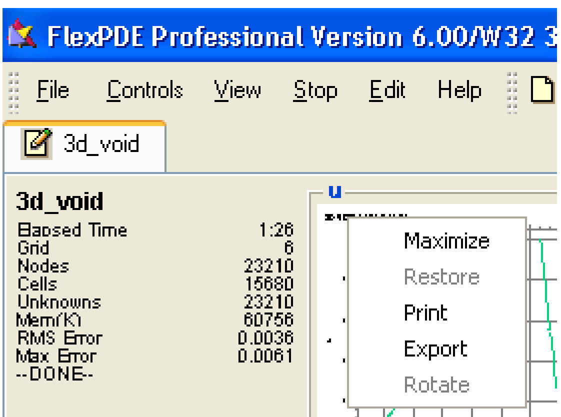
In steady-state problems, only MONITORS will be displayed during the run. They will be replaced by PLOTS when the solution is complete.

In time-dependent problems, all MONITORS and PLOTS will be displayed simultaneously, and updated as the sequencing specifications of the descriptor dictate.

PLOTS will be sent to the ".pg6" graphic record on disk for later recovery. MONITORS will not.

In eigenvalue problems, there will be one set of MONITORS or PLOTS for each requested mode. In other respects, eigenvalue problems behave as steady-state problems.

A right-click in any "thumbnail" plot brings up a menu from which several options can be selected:



The menu items are:

Maximize

Causes the selected plot to be expanded to fill the display panel. You can also maximize a thumbnail by double-clicking in the selected plot.

Restore

Causes a maximized plot to be returned to thumbnail size.

Print

Sends the window to the printer using a standard Print dialog.

Export

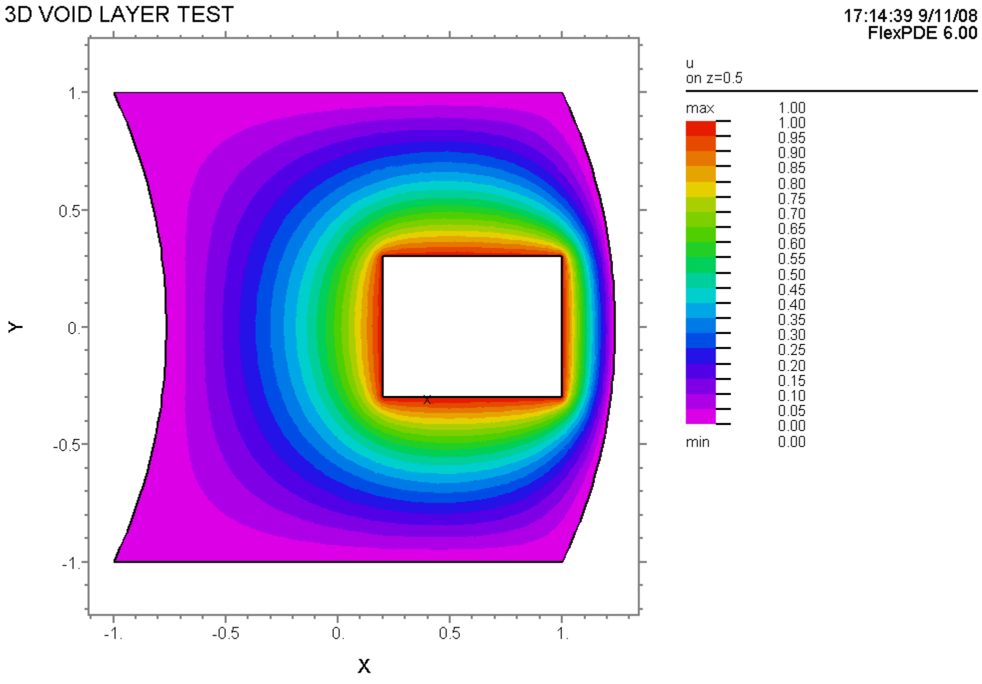
Invokes a dialog which allows the selection of a format for exporting the plot in standard format to other processes. Currently, the options are BMP, EMF, EPS, PNG, PPG and XPG. For bitmap formats (BMP, PNG, PPG and XPG) the dialog allows the selection of the drawing linewidth and resolution of the bitmap, independent of the resolution of the screen. For vector formats (EMF, EPS) no resolution is necessary (FlexPDE uses a fixed resolution of 5400x7200). EPS produces an 8.5x11 inch landscape mode PostScript file suitable for printing.

Rotate

3D plots can be rotated in polar and azimuthal angle.

Plot Labeling

A typical CONTOUR plot might appear as follows:



At the top of the display the "Title" field from the problem descriptor appears, with the time and date of problem execution at the right corner, along with the version of FlexPDE which performed the computation.

At the bottom of the page is a summary of the problem statistics, similar to that shown in the Status Window:

- The problem name
- The number of gridding cycles performed so far
- The polynomial order of the Finite-Element basis (p2 = quadratic, p3 = cubic)
- The number of computation nodes
- The number of computation cells

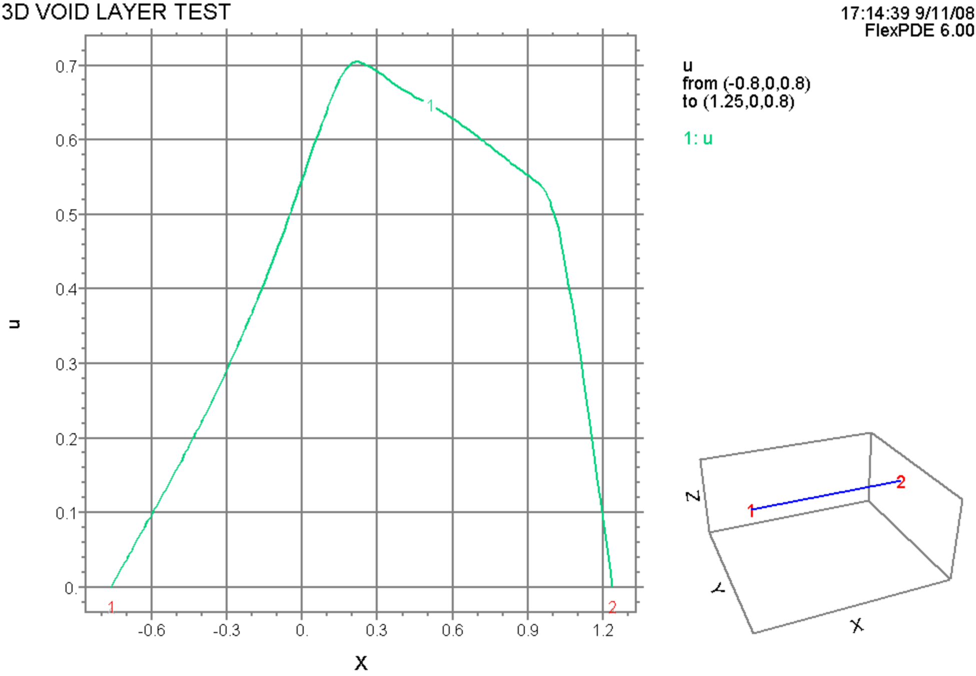
- The estimated RMS value of the relative error in the variables

In staged problems, the stage number will be reported.
In eigenvalue problems, the mode number will be reported.
In time dependent problems, the current problem time and timestep will be reported.

By default, FlexPDE computes the integral under the displayed curve, and this value is reported as "Integral".

Any requested REPORTS will appear in the bottom line.

A typical ELEVATION plot might appear as follows:



Here all the labeling of the contour plot appears, as well as a thumbnail plot of the problem domain, showing the position of the elevation in the figure. For boundary plots, the joints of the boundary are numbered on the thumbnail. The numbers also appear along the baseline of the elevation plot for positional reference.

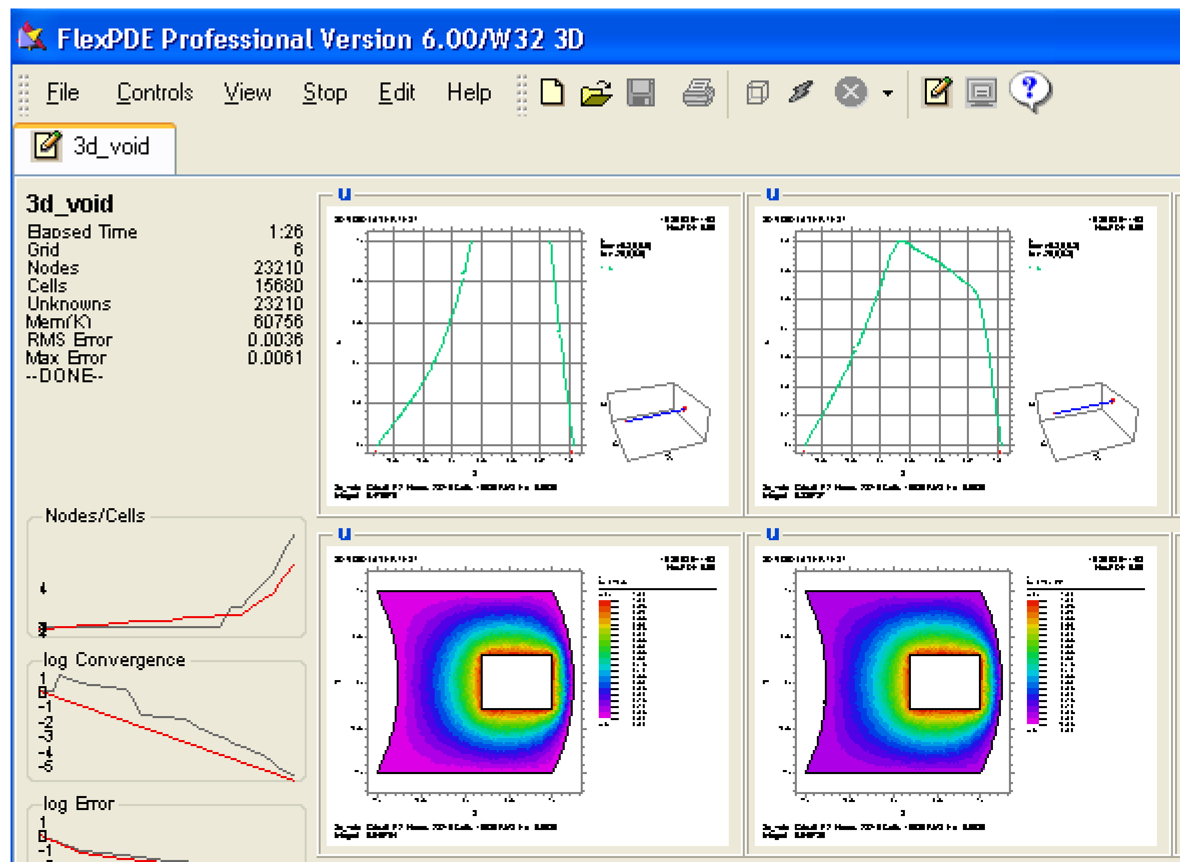
Editing While Running

While the problem is running, you can return the display panel to the editor mode by clicking the Edit Script tool () or the Show Editor item in the Controls menu. The Run icon ( ) will continue to be displayed in the problem tab as long as the problem is running. When the problem terminates, the problem tab will again display the Edit icon ( ).

You can return to the graphic display panel by clicking the Show Plots tool ( ) or the Show Plots item in the Controls menu.

1.9 When the Problem Finishes

When FlexPDE completes the solution of the current problem, it will leave the displays requested in the PLOTS section of the descriptor displayed on the screen. The problem tab will display the Edit icon ( ).



At this point you have several options:

Edit or Save the Script

Click "Controls|Show Editor" (or the Tool) to switch the display into Edit mode, allowing you to change the problem and run again.

From Edit mode, you can click "Controls|Show Plots" (or the Tool) to redisplay the plots.

You can also click "File|Save" (or the Tool) to save the file, "File|Save_As" to save with a new name, or "File|Close" to close the problem.

Switch to Another Problem

Each currently open problem is represented by a named tab on the tab bar. You can switch back and forth among open problems by selecting any tab.

Open a New File

Click "File|Open" (or the Tool) to open another problem script without closing the current problem.

1.10 Viewing Saved Graphic Files

Whenever a problem is run by FlexPDE 6, the graphical output selected by the PLOTS section of the descriptor is written to a file with the extension ".pg6". These files can later be viewed by FlexPDE without re-running the job. (FlexPDE 6 can also open output files from versions 4 and 5.) You can open these files from the "File | View File" or the "View | View FIle" menu items on the main FlexPDE menu, or from the "File | Open File" menu using suffix selection. A standard "Open_File" dialog will appear, from which you may select from the available files on your system. Once a file is selected, the first block of plots will be displayed.

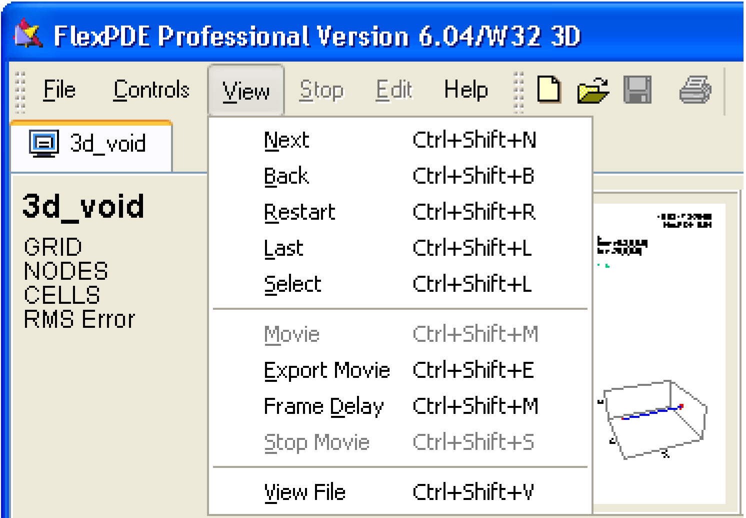
On the left is a "Status" window, much like the one that appears during the run. Not all the runtime information will appear here, but only those items necessary to identify the current group of plots.

In steady-state problems, all the PLOT windows will be displayed. If the problem is staged, then each stage will appear in a separate group.

In time-dependent problems, each plot time group specified in the PLOTS section of the descriptor will form a display group.

The Problem Tab shows the icon to indicate that this is a "View" file, not a PDE problem.

You can use the "View" item in the main menu to control the viewing of these stored graphics:



Thumbnail Plot Displays

In the normal thumbnail display, all the plots of a group are displayed simultaneously. In this case, the "View" menu items have the following meanings:

Next
Use this item to advance to the next group of plots in the file. If there are no more groups, a message box will appear.

Back
Causes FlexPDE to back up and redisplay the previous group. If there are no earlier groups, a

message box will appear.

Restart
Returns to the beginning of the file and displays the first group.

Last
Scans to the end of the file and displays the last group.

Select
Displays a list of plot times that can be viewed. Double-clicking an entry views the selected plot group.

Movie
This item is active only for time-dependent or staged problems. It will cause all groups to be displayed sequentially, with a default delay of 500 milliseconds between groups (plus the file reading time).

Frame Delay
Allows redefining of the delay time between movie frames.

Stop
During the display of a movie, you can use Stop to halt the display.

View File
Selects a new graphics file to be opened in a new tab.

Maximized Plot Windows

When a selected View plot is maximized, either by the right-click menu or by double-click, the behavior of some of the View menu items is modified:

Next
Advances to the next instance of the currently maximized plot. If there are no more instances, a message box will appear.

Back
Backs up and redisplays the previous instance of the currently maximized plot. If there are no earlier instances, a message box will appear.

Movie
This item is active only for time-dependent or staged problems. It will cause all instances of the current plot to be displayed sequentially, separated by the currently active Frame Delay time (plus the file read time).

Export Movie
An export parameters dialog will appear, allowing you to select the file format and resolution. A movie will then be displayed as with "Movie". Each frame of the movie will be exported to a file of the selected type and resolution. The files will be numbered sequentially, and can be subsequently imported into an animation program such as "Animation Shop" to create animations.

1.11 Example Problems

The standard distribution of FlexPDE includes over one hundred example problems, showing the application of FlexPDE to many areas of study. These problem scripts are installed by the standard installation procedure, and are located in a tree structure headed by the "Samples" folder in the installation directory. Modifying a copy of an existing descriptor is frequently the most efficient way to start building a

descriptor for a new problem.

Also included in the distribution, in the "Backstrom_Books" folder, are many samples from books written by Prof. Gunnar Backstrom showing the use of FlexPDE in an academic environment. See professor Backstrom's website at http://learnbyprogramming.com/fields.htm.

Since the example problem scripts are installed in the same folder as the FlexPDE executable file, it may be necessary to copy the sample files to another directory before running or modifying, to avoid file permission problems in your environment.

1.12 Registering FlexPDE

The standard distribution of FlexPDE will run demonstration problems as provided by PDE Solutions Inc, or view stored graphics files from FlexPDE runs without need for license registration. Any other use requires a license, which may be purchased from PDE Solutions Inc in one of many forms.

Internet Key

The standard method of licensing FlexPDE Professional Version is by Internet activation. This mode of licensing generates a text key that locks the execution of FlexPDE to a specific computer. Access to the internet is required on a periodic basis to validate the key. The key can be released from one machine and reactivated on another without difficulty. If you need to use a proxy server for internet access, you can set this information on the "Help | Web Proxy Settings" 4 menu.

Dongle

On request, Professional configurations can be licensed by use of a portable hardware license key (or dongle). You should receive this device in your FlexPDE distribution kit. The standard dongle for use with FlexPDE 6 is a single-machine USB device. You may request a parallel port dongle or network dongle at the time of your order.

In order for FlexPDE to find the dongle, you must
1) Run the appropriate dongle driver install program to load it into your system.
2) Install the dongle in an appropriate USB connector or hub.
3) Start FlexPDE and go to the "Help | Register FlexPDE" menu to inform FlexPDE that it is to look for a dongle license. (See "The Register Dialog" 22)

Network License

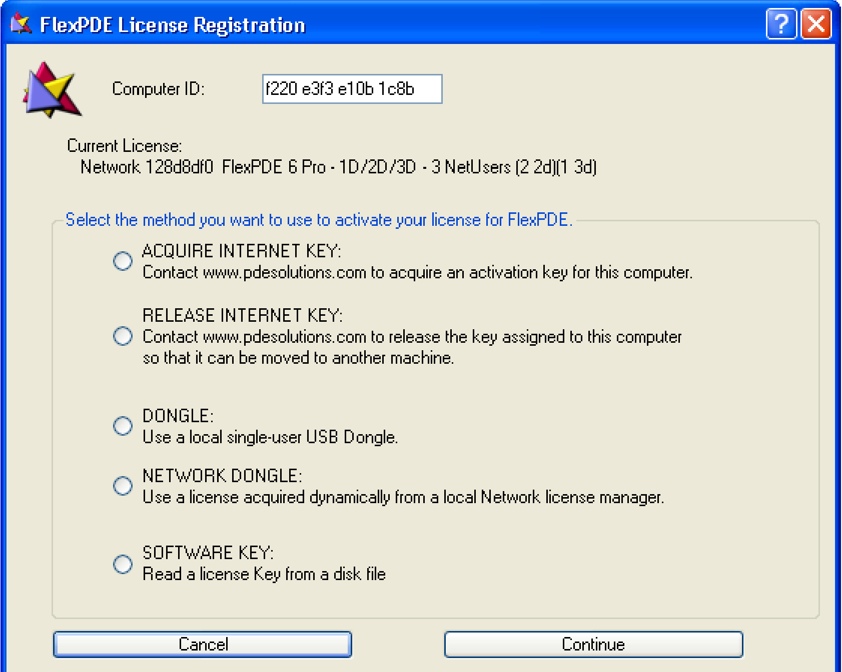
FlexPDE can be licensed over a network, in which case one selected machine in the network runs the license manager service (with a network dongle attached), and client computers on the network can request licenses to run FlexPDE on a first-come first-served basis up to the limit of the licensed number of stations. To tell FlexPDE to search the network for a license server, go to the "Help | Register FlexPDE" menu and select "Network Dongle" licensing (See "The Register Dialog" 22). A network version of the dongle is required.

Software Key

On request, Professional configurations can be licensed in the form of a text key that locks the execution of FlexPDE to a specific computer CPU. If you prefer a software license key, you must first download and install the software and record the computer ID from the sign-on screen or "Help | Register" screen. Include the computer ID on the license application form. Your software key will be sent to you by Email. Copy this key to the FlexPDE installation directory (you may need administrator privileges to do this).

1.12.1 The Register Dialog

To open the license registration dialog, click "Help" on the main menu bar, then click "Register". The following dialog will appear:



Computer ID

This text is the unique identification of your computer. It may be used to request a software key or Evaluation license for FlexPDE Professional.

Current License

If your license has already been registered, this text will display the details of that registration. In the case displayed,
- the license method is by network USB dongle;
- the dongle serial number is #128d8df0;
- the network dongle has three total licenses - all three licenses can run 1D problems, two of the three can run 2D problems, and only one can run 3D problems.

Select a License Method

This section allows you to choose the form of licensing you will use. You can select one of the four options:
- Acquire Internet Key,
- Dongle,
- Network,
- Software Key.

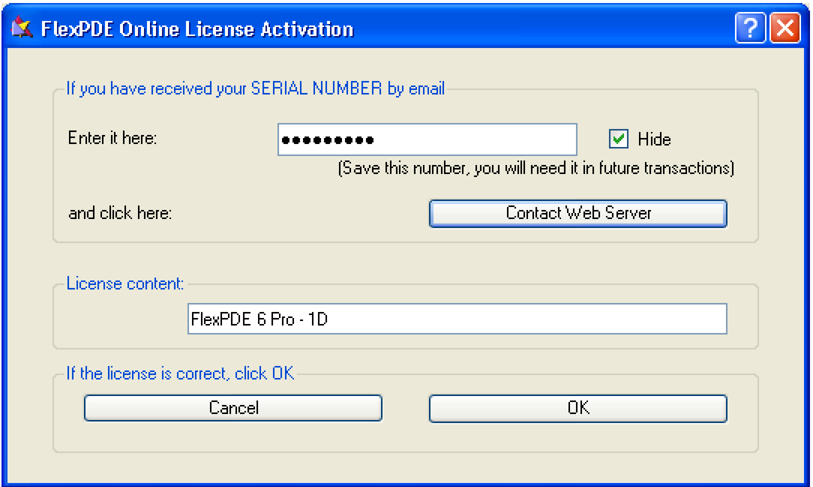
Choose an option and click the "Continue" button.

- Release Internet Key can be used to move the license to another machine.

1.12.2 Internet Key Registration

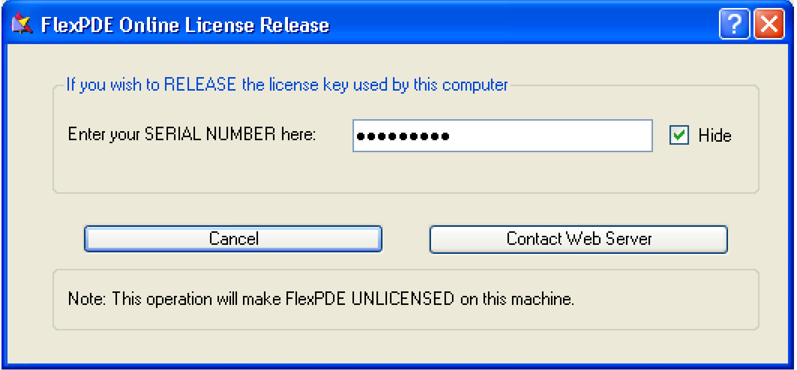
Activation

Enter your serial number into text field and click "Contact Web Server". If successful, the license contents will be displayed. If not, FlexPDE will report an error. Click "OK" to finish the registration. (If this activation is performed in public places, you can choose to "Hide" the Serial Number.)



Deactivation

Enter your serial number into the text field and click "Contact Web Server". If successful, FlexPDE will release the license on the local machine. If not, it will report an error.



Initially, the license must be deactivated from the same machine that is currently activate. However, in an attempt to make switching the license between two machines more convenient, FlexPDE will allow deactivation of the license from either of the last two machines that have been successfully registered.

Notes :

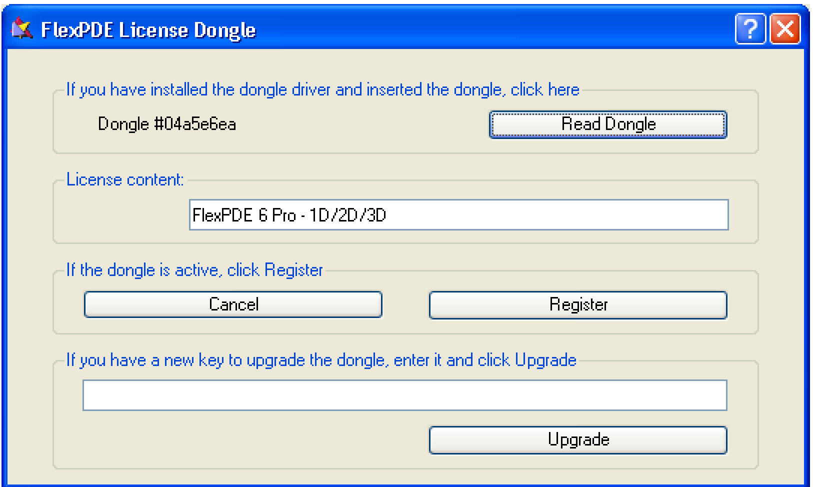
A computer's identification is constructed in part from it's MAC address and the operating system's report of a unique identifier for that installation. Sometimes the MAC address can change (usually on laptops connecting to different networks or when connected by Wi-Fi instead of a wired connection). If this

happens after the machine is licensed, FlexPDE will issue an error telling the user that the license authorizes a different computer. When that happens, the user can simply release and reacquire the license in order to resolve the issue.

If you need to use a proxy server for internet access, you can set this information on the "Help | Web Proxy Settings" 4 menu.

1.12.3 Dongle Registration

If your license is to be read from a locally attached dongle, click the "Dongle" button in the Register Dialog then click "Continue". The following dialog will appear:



The "Read Dongle" button

This button will read the contents of the dongle without installing it as the selected license method. FlexPDE will search the USB and Parallel ports for an appropriate license dongle. If a dongle is found, the ID number and the license contents are displayed. You must select the "Register" button to activate the dongle as the license method.
If no dongle is found, or if the dongle driver has not been installed, the search will fail, and FlexPDE will report an error.

License Content

This line displays the characteristics encoded in the license identified by the previous selections. In the case displayed above, the license encodes a FlexPDE Professional license for 1D, 2D or 3D problems.

The "Register" button

Click "Register" to install the dongle as the active license method. Subsequently, every time you start FlexPDE it will search the USB and Parallel ports for the dongle.

The "Cancel" button

Click "Cancel" to abort without changing the active license method.

Upgrading a Dongle

You can use the Register dialog to field-upgrade a dongle (including Network dongles). If you have previously been issued a FlexPDE version 4, version 5 or version 6 dongle, and subsequently purchase an upgrade, you will be sent a software key which encodes the upgrade. Dongles issued with FlexPDE version 2 or version 3 cannot be upgraded to version 6. You will be sent a new version 6 dongle when upgrading from these versions.

Type or paste your upgrade key in the field provided, and click "Upgrade". Your dongle will be updated with the information encoded in the key. Note that the dongle upgrade facility will rewrite the dongle only if the serial number of the dongle matches the serial number encoded in the upgrade key. Click "Register" finish the upgrade.

1.12.4 Network Dongle Registration

If you select "Network", then "Continue", from the Registration Dialog, FlexPDE will search your network for a running license manager, and return the status of that license.

Unlike local dongle registration, Network Dongle registration automatically installs the network dongle as the active registration method. You will not be given the option of registering the dongle. This success or failure of Network Dongle registration depends only on the presence or absence of a valid license facility on the network. It does not examine the available licensed capabilities.

In the future, every time you start FlexPDE, it will expect to find a network license manager to grant licenses. In fact, the request for a network license will not be made until you actually "Run" a problem. At that time, a license of the appropriate class, 1D, 2D or 3D will be requested from the network. The acquired license will be held until the current invocation of FlexPDE is terminated. In this way, networks of FlexPDE users can get optimal use out of the mix of 1D, 2D and 3D licenses that have been purchased.

When you "Run" a problem with the network licensing method, if the license manager finds that all available licenses are in use, you will be given the option of waiting for an available license or running in demonstration mode.

License Manager Installation

In order to use the network dongle, one must first install a license manager service on the machine that the dongle is physically connected to. The license manager installer must be downloaded from the dongle vendor's website. A URL link to the download is provided along with the dongle driver on your FlexPDE installation CD or from our website at www.pdesolutions.com/sdmenu6.html .

To set up the license manager :

1) Choose a computer on the local network that you want to be the "server". This machine will have the dongle physically connected to it.
2) Install the dongle driver on the server.
3) Install the license manager as a "service" on the server.
4) Plug the network dongle into the server.

At this point, any machine on the local network should be able to run FlexPDE and find the network dongle.

Modifying How FlexPDE Accesses The License Manager

***Note:** The search parameters for finding a running license manager can be controlled by editing the "nethasp.ini" file in the FlexPDE installation folder. See comments in the file for a complete list and explanation of the settings.*

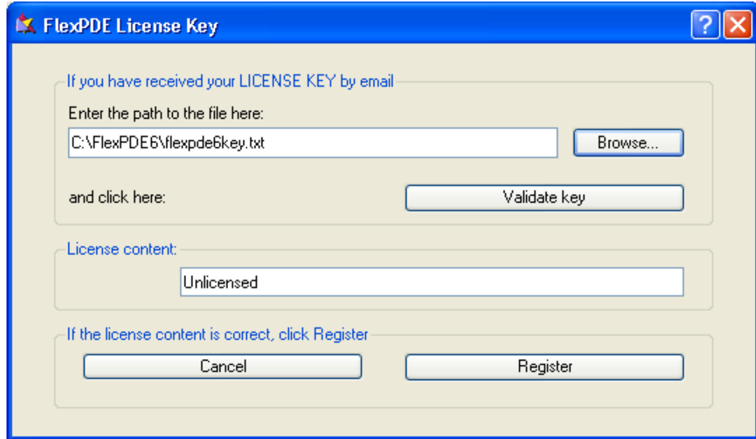
In order to access the license manager from outside the local network, the "nethasp.ini" file will have to be modified. Under the appropriate protocol section (IPX, NETBIOS or TCPIP), enter the server name or IP address (and/or port number). An example is shown here :

[NH_TCPIP]
NH_SERVER_ADDR = 12.123.456.789
NH_SERVER_NAME = lmserver
NH_PORT_NUMBER = 999

If experiencing frequent timeouts of FlexPDE communicating with the license manager, the timeout length can be modified by changing the value of NH_SEND_RCV . This number is the number of seconds that FlexPDE will wait for a response from the license manager before displaying a dialog notifying the user that connection has been lost.

1.12.5 Software Key Registration

If your license is to be read from a software key file, click the "Manual" button in the Register Dialog then click "Continue". The following dialog will appear:



Browse to the location of the software key file and select "Validate Key".

The "Validate Key" button

This button will read the contents of the license file without installing it as the selected license method. FlexPDE will validate the license file entered and display the contents. You must select the "Register" button to set this as the license method.

License Content

This line displays the characteristics encoded in the license identified by the previous selections. In the case displayed above, there is no license file and FlexPDE is in Evaluation mode.

The "Register" button

Click "Register" to install this as the active license method. FlexPDE will copy this information into the installation folder. In future, every time you start FlexPDE, it will look for the license key file in the installation folder.

The "Cancel" button

Click "Cancel" to abort without changing the active license method.

Part II

User Guide

2 User Guide

This section introduces the reader gradually to the use of FlexPDE in the solution of systems of partial differential equations.

We begin with a discussion of the basic character of FlexPDE. We then construct a simple model problem and proceed to add features to the model.

The result is a description of the most common features of FlexPDE in what we hope is a meaningful and understandable evolution that will allow users to very quickly become accustomed to the use of FlexPDE and to use it in their own work.

No attempt is made in this manual to present a complete description of each command or circumstance which can arise. Detailed descriptions of each command are presented in the Problem Descriptor Reference 116 section.

2.1 Overview

2.1.1 What Is FlexPDE?

FlexPDE is a "scripted finite element model builder and numerical solver".

By this we mean that from a script written by the user, FlexPDE performs the operations necessary to turn a description of a partial differential equations system into a finite element model, solve the system, and present graphical and tabular output of the results.

FlexPDE is also a "problem solving environment".

It performs the entire range of functions necessary to solve partial differential equation systems: an editor for preparing scripts, a mesh generator for building finite element meshes, a finite element solver to find solutions, and a graphics system to plot results. The user can edit the script, run the problem and observe the output, then re-edit and re-run repeatedly without leaving the FlexPDE application environment.

FlexPDE has no pre-defined problem domain or equation list.

The choice of partial differential equations is totally up to the user.

The FlexPDE scripting language is a "natural" language.

It allows the user to describe the mathematics of his partial differential equations system and the geometry of his problem domain in a format similar to the way he might describe it to a co-worker.

For instance, there is an EQUATIONS section in the script, in which Laplace's equation would be presented as

```
Div(grad(u)) = 0.
```

Similarly, there is a BOUNDARIES section in the script, where the geometric boundaries of a two-dimensional problem domain are presented merely by walking around the perimeter:

```
Start(x1,y1) line to (x2,y1) to (x2,y2) to (x1,y2) to close
```

This scripted form has many advantages

- The script completely describes the equation system and problem domain, so there is no uncertainty

about what equations are being solved, as might be the case with a fixed-application program.
- New variables, new equations or new terms may be added at will, so there is never a case of the software being unable to represent a different loss term, or a different physical effect.
- Many different problems can be solved with the same software, so there is not a new learning curve for each problem

There is also a corollary requirement with the scripted model:

- The user must be able to pose his problem in mathematical form.

In an educational environment, this is good. It's what the student wants to learn.

In an industrial environment, a single knowledgeable user can prepare scripts which can be used and modified by less skilled workers. And a library of application scripts can show how it is done.

2.1.2 What Can FlexPDE Do?

- FlexPDE can solve systems of first or second order partial differential equations in one, two or three-dimensional Cartesian geometry, in one-dimensional spherical or cylindrical geometry, or in axi-symmetric two-dimensional geometry. (Other geometries can be supported by including the proper terms in the PDE.)

- The system may be steady-state or time-dependent, or alternatively FlexPDE can solve eigenvalue problems. Steady-state and time-dependent equations can be mixed in a single problem.

- Any number of simultaneous equations can be solved, subject to the limitations of the computer on which FlexPDE is run.

- The equations can be linear or nonlinear. (FlexPDE automatically applies a modified Newton-Raphson iteration process in nonlinear systems.)

- Any number of regions of different material properties may be defined.

- Modeled variables are assumed to be continuous across material interfaces. Jump conditions on derivatives follow from the statement of the PDE system. (CONTACT boundary conditions can handle discontinuous variables.)

- FlexPDE can be extremely easy to use, and this feature recommends it for use in education. But FlexPDE is not a toy. By full use of its power, it can be applied successfully to extremely difficult problems.

2.1.3 How Does It Do It?

FlexPDE is a fully integrated PDE solver, combining several internal facilities to provide a complete problem solving system:

- **A script editing facility** with syntax highlighting provides a full text editing facility and a graphical domain preview.

- **A symbolic equation analyzer** expands defined parameters and equations, performs spatial differentiation, and symbolically applies integration by parts to reduce second order terms to create symbolic Galerkin equations. It then symbolically differentiates these equations to form the Jacobian

coupling matrix.

- **A mesh generation facility** constructs a triangular or tetrahedral finite element mesh over a two or three-dimensional problem domain. In two dimensions, an arbitrary domain is filled with an unstructured triangular mesh. In three-dimensional problems, an arbitrary two-dimensional domain is extruded into a the third dimension and cut by arbitrary dividing surfaces. The resulting three-dimensional figure is filled with an unstructured tetrahedral mesh.

- **A Finite Element numerical analysis facility** selects an appropriate solution scheme for steady-state, time-dependent or eigenvalue problems, with separate procedures for linear and nonlinear systems. The finite element basis may be linear, quadratic or cubic.

- **An adaptive mesh refinement procedure** measures the adequacy of the mesh and refines the mesh wherever the error is large. The system iterates the mesh refinement and solution until a user-defined error tolerance is achieved.

- **A dynamic timestep control procedure** measures the curvature of the solution in time and adapts the time integration step to maintain accuracy.

- **A graphical output facility** accepts arbitrary algebraic functions of the solution and plots contour, surface, vector or elevation plots.

- **A data export facility** can write text reports in many formats, including simple tables, full finite element mesh data, CDF, VTK or TecPlot compatible files.

2.1.4 Who Can Use FlexPDE?

Most of physics and engineering is described at one level or another in terms of partial differential equations. This means that a scripted solver like FlexPDE can be applied to *virtually any* area of engineering or science.

- **Researchers** in many fields can use FlexPDE to model their experiments or apparatus, make predictions or test the importance of various effects. Parameter variations or dependencies are not limited by a library of forms, but can be programmed at will.

- **Engineers** can use FlexPDE to do design optimization studies, feasibility studies and conceptual analyses. The same software can be used to model all aspects of a design -- no need for a separate tool for each effect.

- **Application developers** can use FlexPDE as the core of a special-purpose applications that need finite element modeling of partial differential equation systems.

- **Educators** can use FlexPDE to teach physics or engineering. A single software tool can be used to examine the full range of systems of interest in a discipline.

- **Students** see the actual equations, and can experiment interactively with the effects of modifying terms or domains.

2.1.5 What Does A Script Look Like?

A problem description script is a readable text file. The contents of the file consist of a number of sections, each identified by a header. The fundamental sections are:

- TITLE — a descriptive label for the output.
- SELECT — user controls that override the default behavior of FlexPDE.
- VARIABLES — here the dependent variables are named.
- DEFINITIONS — useful parameters, relationships or functions are defined.
- EQUATIONS — each variable is associated with a partial differential equation.
- BOUNDARIES — the geometry is described by walking the perimeter of the domain, stringing together line or arc segments to bound the figure.
- MONITORS and PLOTS — desired graphical output is listed, including any combination of CONTOUR, SURFACE, ELEVATION or VECTOR plots.
- END — completes the script.

Note: There are several other optional sections for describing special aspects of the problem. Some of these will be introduced later in this document. Detailed rules for all sections are presented in the FlexPDE Problem Descriptor Reference chapter "The Sections of a Descriptor 145*".*

COMMENTS can be placed anywhere in a script to describe or clarify the work. Two forms of comment are available:

- { Anything inside curly brackets is a comment. }
- ! from an exclamation to the end of the line is a comment.

Example:

A simple diffusion equation on a square might look like this:

```
TITLE 'Simple diffusion equation'
{ this problem lacks sources and boundary conditions }
VARIABLES
        u
DEFINITIONS
        k=3        { conductivity }
EQUATIONS
        div(k*grad(u)) =0
BOUNDARIES
        REGION 1
        START(0,0)
            LINE TO (1,0) TO (1,1) TO (0,1) TO CLOSE
PLOTS
        CONTOUR(u)
        VECTOR(k*grad(u))
END
```

Later on, we will show detailed examples of the development of a problem script.

2.1.6 What About Boundary Conditions?

Proper specification of boundary conditions is crucial to the solution of a PDE system.

In a FlexPDE script, boundary conditions are presented as the boundary is being described.

The primary types of boundary condition are VALUE and NATURAL.

The VALUE (or Dirichlet) boundary condition specifies the value that a variable must take on at the boundary of the domain.

The NATURAL boundary condition specifies a flux at the boundary of the domain. (The precise meaning of the NATURAL boundary condition depends on the PDE for which the boundary condition is being specified. Details are discussed in the Chapter "Natural Boundary Conditions 61")

In the diffusion problem presented above, for example, we may add fixed values on the bottom and top edges, and zero-flux conditions on the sides as follows:

```
...
BOUNDARIES
    REGION 1
    START(0,0)
        VALUE(u) = 0     LINE TO (1,0) { fixed value on bottom }
        NATURAL(u)=0     LINE TO (1,1) { insulated right side }
        VALUE(u)=1              LINE TO (0,1) { fixed value on top }
        NATURAL(u)=0     LINE TO CLOSE        { insulated left side }
...
```

Notice that a VALUE or NATURAL statement declares a condition which will apply to the subsequent boundary segments until the declaration is changed.

2.2 Basic Usage

2.2.1 How Do I Set Up My Problem?

FlexPDE reads a text script that describes in readable language the characteristics of the problem to be solved. In simple applications, the script can be very simple. Complex applications may require much more familiarity with the abilities of FlexPDE.

In the following discussion, we will begin with the simpler features of FlexPDE and gradually introduce more complex features as we proceed.

FlexPDE has a built-in editor with which you can construct your problem script. You can edit the script, run it, edit it some more, and run it again until the result satisfies your needs. You can save the script for later use or as a base for later modifications.

The easiest way to begin a problem setup is to copy a similar problem that already exists.

Whether you start fresh or copy an existing file, there are four basic parts to be defined:

- Define the variables and equations
- Define the domain
- Define the material parameters
- Define the boundary conditions
- Specify the graphical output.

These steps will be described in the following sections. We will use a simple 2D heatflow problem as an example, and start by building the script from the most basic elements of FlexPDE. In later sections, we will elaborate the script, and address the more advanced capabilities of FlexPDE in an evolutionary manner. 3D applications rely heavily on 2D concepts, and will be discussed in a separate chapter.

> ***Note:*** *We will make no attempt in the following to describe all the options that are available to the user at any point, but try to keep the concept clear by illustrating the most common forms. The full range of options is detailed in the FlexPDE Problem Descriptor Reference. Many will also be addressed in subsequent topics.*

2.2.2 Problem Setup Guidelines

In posing any problem for FlexPDE, there are some guidelines that should be followed.

- **Start with a fundamental statement of the physical system.** Descriptions of basic conservation principles usually work better than the heavily massaged pseudo-analytic "simplifications" which frequently appear in textbooks.

- **Start with a simple model, preferably one for which you know the answer.** This allows you both to validate your presentation of the problem, and to increase your confidence in the reliability of FlexPDE. (One useful technique is to assume an analytic answer and plug it into the PDE to generate the source terms necessary to produce that solution. Be sure to take into account the appropriate boundary conditions.)

- **Use simple material parameters at first.** Don't worry about the exact form of nonlinear coefficients or material properties at first. Try to get a simple problem to work, and add the complexities later.

- **Map out the domain.** Draw the outer boundary first, placing boundary conditions as you go. Then overlay the other material regions. Later regions will overlay and replace anything under them, so you don't have to replicate a lot of complicated interfaces.

- **Use MONITORS** of anything that might help you see what is happening in the solution. Don't just plot the final value you want to see and then wonder why it's wrong. Get feedback! That's what the MONITORS section is there for.

- **Annotate your script** with frequent comments. Later you will want to know just what it was you were thinking when you wrote the script. Include references to sources of the equations or notes on the derivation.

- **Save your work.** FlexPDE will write the script to disk whenever you click "Domain Review" or "Run Script". But if you are doing a lot of typing, use "Save" or "Save_as" to protect your work from

unforeseen interruptions.

2.2.3 Notation

In most cases, FlexPDE notation is simple text as in a programming language.

- Differentiation, such as du/dx, is denoted by the form `dx(u)`. All active coordinate names are recognized, as are second derivatives like `dxx(u)` and differential operators `Div`, `Grad` and `Curl`.
- Names are NOT case sensitive. "F" is the same as "f".
- Comments can be placed liberally in the text. Use { } to enclose comments or ! to ignore the remainder of the line.

***Note**: See the Problem Descriptor Reference chapter on Elements* 121 *for a full description of FlexPDE notation.*

2.2.4 Variables and Equations

The two primary things that FlexPDE needs to know are:

- what are the variables that you want to analyze?
- what are the partial differential equations that define them?

The VARIABLES and EQUATIONS sections of a problem script supply this information. The two are closely linked, since you must have one equation for each variable in a properly posed system.

In a simple problem, you may have only a single variable, like voltage or temperature. In this case, you can simply state the variable and equation:

```
VARIABLES
    Phi
EQUATIONS
    Div(grad(Phi))  = 0
```

In a more complex case, there may be many variables and many equations. FlexPDE will want to know how to associate equations with variables, because some of the details of constructing the model will depend on this association.

Each equation must be labeled with the variable to which it is associated (name and colon), as indicated below:

```
VARIABLES
    A,B
EQUATIONS
A:   Div(grad(A))  = 0
B:   Div(grad(B))  = 0
```

Later, when we specify boundary conditions, these labels will be used to associate boundary conditions with the appropriate equation.

2.2.5 Mapping the Domain

Two-Dimensional Domain Description

A two-dimensional problem domain is described in the BOUNDARIES section, and is made up of REGIONS, each assumed to contain unique material properties. A REGION may contain many closed loops or islands, but they are all assumed to have the same material properties.

- A REGION specification begins with the statement REGION <number> (or REGION "name") and all loops following the header are included in the region.
- REGIONs occurring later in the script overlay and cover up parts of earlier REGIONs.
- The first REGION should contain the entire domain. This is an unenforced convention that makes the attachment of boundary conditions easier.

Region shapes are described by walking the perimeter, stepping from one joint to another with LINE, SPLINE or ARC segments. Each segment assumes that it will continue from the end of the previous segment, and the START clause gets things rolling. You can make a segment return to the beginning with the word CLOSE (or TO CLOSE).

- A rectangular region, for example, is made up of four line segments:

```
START(x1,y1)
    LINE TO(x2,y1)
    TO (x2,y2)
    TO (x1,y2)
    TO CLOSE
```

  (Of course, any quadrilateral figure can be made with the same structure, merely by changing the coordinates. And any polygonal figure can be constructed by adding more points.)

- Arcs can be built in several ways, the simplest of which is by specifying a center and an angle:

```
START(r,0)
ARC(CENTER=0,0) ANGLE=360
```

- Arcs can also be built by specifying a center and an end point:

```
START(r,0)
ARC(CENTER=0,0) TO (0,r)     { a 90 degree arc }
```

  An elliptical arc will be built if the distance from the center to the endpoint is different than the distance from the center to the beginning point. The axes of the ellipse will extend along the horizontal and vertical coordinate axes. The axes can be rotated with the ROTATE=degrees command.

- Loops can be named for use in later references, as in:

```
START "Name" (...)
```

The prototype form of The BOUNDARIES section is then:

```
BOUNDARIES
    REGION 1
```

```
<closed loops around the domain>
REGION 2
<closed loops around overlays of the second material>
...
```

You can build your domain a little at a time, using the "domain review" menu button to preview a drawing of what you have created so far.

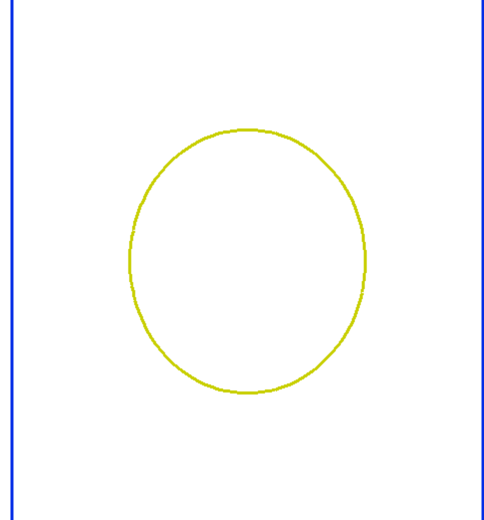
The "Save" and "Save_As" menu buttons allow you to frequently save your work, just in case.

2.2.6 An Example Problem

Let us build as an example a circular inclusion between two plates. We will simply treat the plates as the top and bottom surfaces of a square enclosure, with the circle centered between them. Using the statements above and adding the required control labels, we get:

```
BOUNDARIES
    REGION 1  'box'      { the bounding box }
    START(-1,-1)
        LINE TO(1,-1)
        TO (1,1)
        TO (-1,1)
        TO CLOSE
        REGION 2  'blob'  { the embedded circular 'blob' }
    START 'ring' (1/2,0)
        ARC(CENTER=0,0) ANGLE=360 TO CLOSE
```

The resulting figure displayed by the "domain review" button is this:

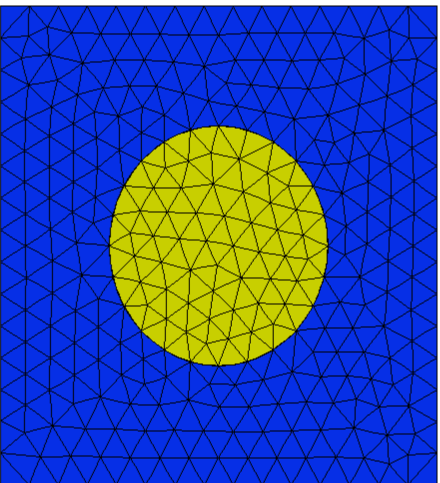


***Note**: The detailed Rules for constructing domain boundaries is included in the Reference chapter " Sections | Boundaries* 180*".*

2.2.7 Generating A Mesh

When you select "Run Script" from the Controls menu (or the button), FlexPDE will begin execution by automatically creating a finite element mesh to fit the domain you have described. In the automatic mesh, cell sizes will be determined by the spacing between explicit points in the domain boundary, by the curvature of arcs, or by explicit user density controls.

In our example, the automatic mesh looks like this:



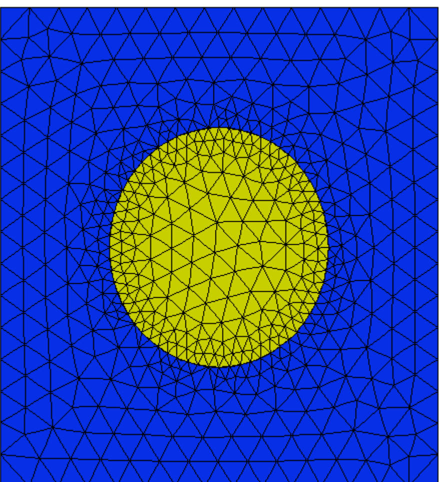
Notice that the circular boundary of region 2 is mapped onto cell legs.

There are several controls that the user can apply to change the behavior of the automatic mesh. These are described in detail in the chapter "Controlling Mesh Density" 103 below.

As an example, we can cause the circular boundary of region 2 to be gridded more densely by using the modifier MESH_SPACING:

```
REGION 2 'blob'           { the embedded 'blob' }
    START(1/2,0)
        MESH_SPACING = 0.05
        ARC(CENTER=0,0) ANGLE=360
```

The resulting mesh looks like this:



In most cases, it is not necessary to intervene in the mesh generation, because as we will see later, FlexPDE will adaptively refine the mesh wherever it detects that there are strong curvatures in the solution.

2.2.8 Defining Material Parameters

Much of the complexity of real problems comes in the fact that the coefficients that enter into the partial differential equation system take on different values in the various materials of which a structure is composed.

This is handled in FlexPDE by two facilities. First, the material parameters are given names and default values in the DEFINITIONS section. Second, the material parameters are given regional values within the domain REGIONS.

So far, it has been of little consequence whether our test problem was heat flow or electrostatics or something else entirely. However, for concreteness in what follows, let us assume it is a heat equation, describing an insulator imbedded in a conductor between to heat reservoirs. We will give the circular insulator a conductivity of 0.001 and the surrounding conductor a conductivity of 1.

First, we define the name of the constant and give it a default value in the definitions section:

```
DEFINITIONS
    k = 1
```

This default value will be used as the value of "k" in every REGION of the problem, unless specifically redefined in a region.

Now we introduce the constant into the equation:

```
EQUATIONS
    Div(-k*grad(phi))  = 0
```

Then we specify the regional value in region 2:

```
...
REGION 2 'blob'  { the embedded blob }
    k = 0.001
    START(1/2,0)
        ARC(CENTER=0,0) ANGLE=360
```

We could also define the parameter k=1 for the conductor in REGION 1, if it seemed useful for clarity.

2.2.9 Setting the Boundary Conditions

Boundary conditions are specified as modifiers during the walk of the perimeter of the domain.

The primary types of boundary condition are VALUE and NATURAL.

The VALUE (or Dirichlet) boundary condition specifies the value that a variable must take on at the boundary of the domain. Values may be any legal arithmetic expression, including nonlinear dependences on variables.

The NATURAL boundary condition specifies a flux at the boundary of the domain. Definitions may be any legal arithmetic expression, including nonlinear dependence on variables. With Laplace's equation, the NATURAL boundary condition is equivalent to the Neumann or normal derivative boundary condition.

> ***Note:*** *The precise meaning of the NATURAL boundary condition depends on the PDE for which the boundary condition is being specified. Details are discussed in the Chapter "Natural Boundary Conditions* 61*."*

Each boundary condition statement takes as an argument the name of a variable. This name associates the boundary condition with one of the listed equations, for it is in reality the equation that is modified by the boundary condition. The equation modified by VALUE(u)=0, for example, is the equation which has previously been identified as defining u. NATURAL(u)=0 will depend for its meaning on the form of the equation which defines u.

In our sample problem, suppose we wish to define a zero temperature along the bottom edge, an insulating boundary on the right side, a temperature of 1 on the top edge, and an insulating boundary on the left. We can do this with these commands:

```
...
REGION 1 'box'   { the bounding box }
    START(-1,-1)
        { Phi=0 on base line: }
        VALUE(Phi)=0   LINE TO(1,-1)
        { normal derivative =0 on right side: }
        NATURAL(Phi)=0       LINE TO (1,1)
        { Phi = 1 on top: }
        VALUE(Phi)=1   LINE TO (-1,1)
        { normal derivative =0 on left side: }
        NATURAL(Phi)=0 LINE TO CLOSE
```

Notice that a VALUE or NATURAL statement declares a condition which will apply to the subsequent boundary segments until the declaration is changed.

Notice also that the segment shape (Line or Arc) must be restated after a change of boundary condition.

> ***Note****: Other boundary condition forms are allowed. See the Reference chapter "Sections | Boundaries* 180*".*

2.2.10 Requesting Graphical Output

The MONITORS and PLOTS sections contain requests for graphical output.

MONITORS are used to get ongoing information about the progress of the solution.

PLOTS are used to specify final output, and these graphics will be saved in a disk file for later viewing.

FlexPDE recognizes several forms of output commands, but the primary forms are:

- CONTOUR a plot of contours of the argument; it may be color-filled
- SURFACE a 3D surface of the argument
- VECTOR a field of arrows
- ELEVATION a "lineout" along a defined path
- SUMMARY text-only reports

Any number of plots may be requested, and the values plotted may be any consistent algebraic combination of variables, coordinates and defined parameters.

In our example, we will request a contour of the temperature, a vector map of the heat flux, k*grad(Phi), an elevation of temperature along the center line, and an elevation of the normal heat flux on the surface of the blob:

```
PLOTS
    CONTOUR(Phi)
    VECTOR(-k*grad(Phi))
    ELEVATION(Phi) FROM (0,-1) TO (0,1)
    ELEVATION(Normal(-k*grad(Phi))) ON 'ring'
```

Output requested in the PLOTS section is produced when FlexPDE has finished the process of solving and regridding, and is satisfied that all cells are within tolerance. An alternative section, identical in form to the PLOTS section but named MONITORS, will produce transitory output at more frequent intervals, as an ongoing report of the progress of the solution.

A record of all PLOTS is written in a file with suffix .PG6 and the name of the .PDE script file. These recorded plots may be viewed at a later time by invoking the VIEW item in the FlexPDE main menu.

MONITORS are not recorded in the .PG6 file. It is strongly recommended that MONITORS be used liberally during script development to determine that the problem has been properly set up and that the solution is proceeding as expected.

> ***Note**: FlexPDE accepts other forms of plot command, including* GRID *plots and* HISTORIES*. See the Reference chapter "Sections | Monitors and Plots* 197*".*

2.2.11 Putting It All Together

In the previous sections, we have gradually built up a problem specification.

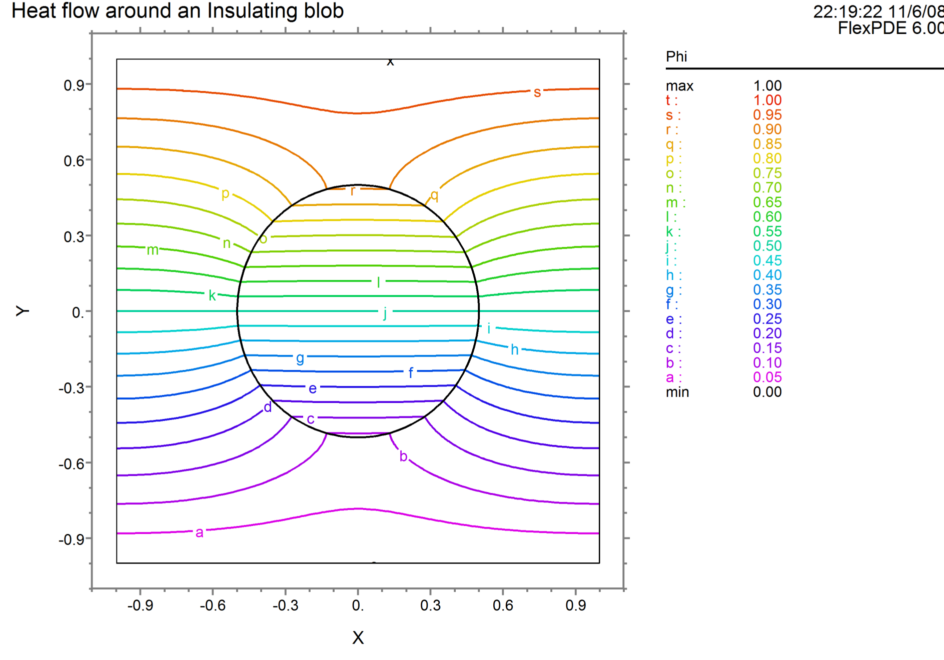
Putting it all together and adding a TITLE, it looks like this:

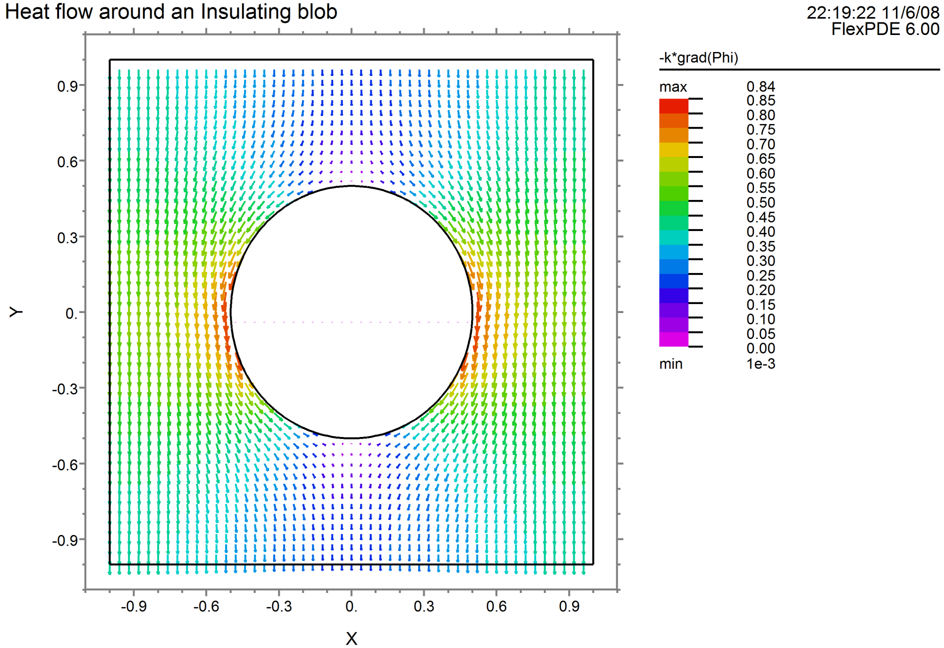
```
TITLE 'Heat flow around an Insulating blob'
VARIABLES
    Phi             { the temperature }
DEFINITIONS
    K = 1           { default conductivity }
    R = 0.5                 { blob radius }
EQUATIONS
    Div(-k*grad(phi)) = 0
```

```
BOUNDARIES
    REGION 1 'box'
    START(-1,-1)
        VALUE(Phi)=0     LINE TO (1,-1)
        NATURAL(Phi)=0 LINE TO (1,1)
        VALUE(Phi)=1     LINE TO (-1,1)
        NATURAL(Phi)=0 LINE TO CLOSE
    REGION 2  'blob'       { the embedded blob }
    k = 0.001
    START 'ring' (R,0)
        ARC(CENTER=0,0) ANGLE=360 TO CLOSE
PLOTS
    CONTOUR(Phi)
    VECTOR(-k*grad(Phi))
    ELEVATION(Phi) FROM (0,-1) to (0,1)
    ELEVATION(Normal(-k*grad(Phi))) ON 'ring'
END
```

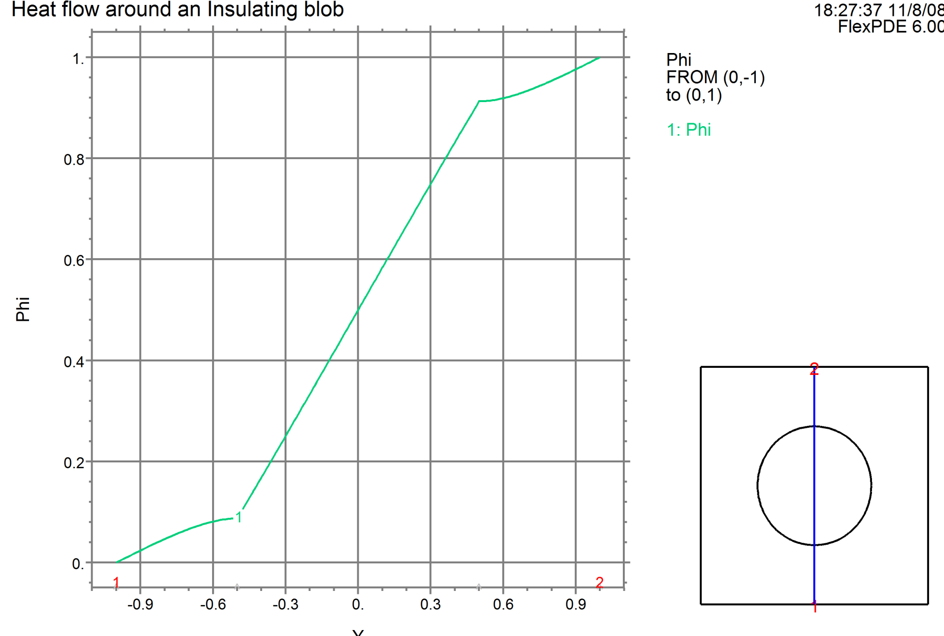
We have defined a complete and meaningful problem in twenty-three readable lines.

The output from this script looks like this:

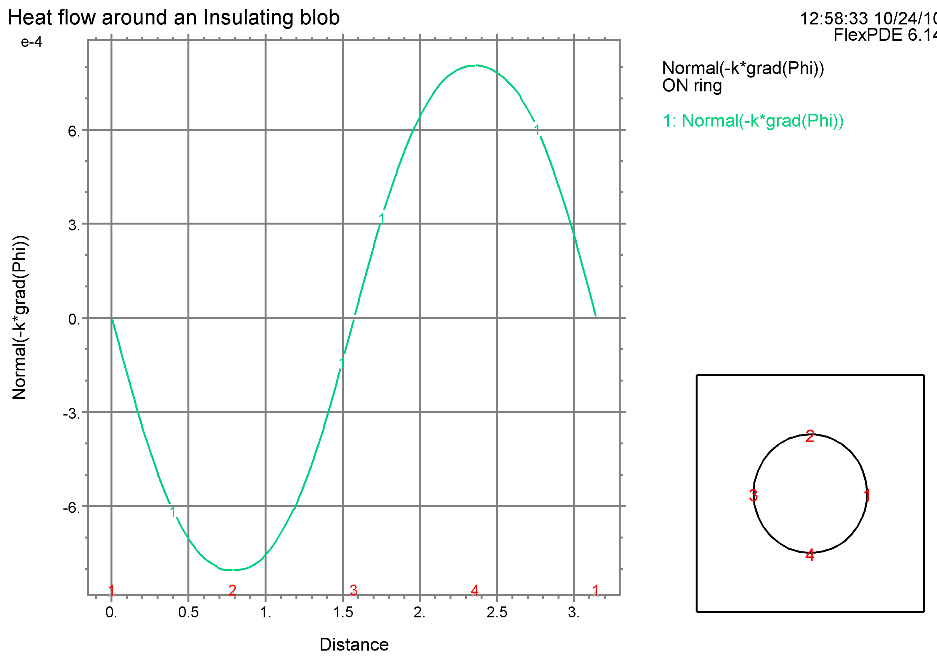




Heat flow around an Insulating blob
22:19:22 11/6/08
FlexPDE 6.00
-k*grad(Phi)
max 0.84
0.85
0.80
0.75
0.70
0.65
0.60
0.55
0.50
0.45
0.40
0.35
0.30
0.25
0.20
0.15
0.10
0.05
0.00
min 1e-3
Y
X
ex1: Grid#4 P2 Nodes=7727 Cells=3806 RMS Err= 5.4e-6



Heat flow around an Insulating blob
18:27:37 11/8/08
FlexPDE 6.00
Phi
FROM (0,-1)
to (0,1)
1: Phi
Phi
Y
ex1: Grid#4 P2 Nodes=7727 Cells=3806 RMS Err= 5.4e-6
Integral= 0.999959



2.2.12 Interpreting a Script

It is important to understand that a FlexPDE script is not a procedural description of the steps to be performed in solving the PDE system. It is instead a description of the dependencies between various elements of the model.

A parameter defined as `P = 10` means that whenever `P` is used in the script, it represents the constant value `10`.

If the parameter is defined as `P = 10*X`, then whenever `P` is used in the script, it represents `10` times the value of X at each point of the domain at which the value of `P` is needed for the solution of the system. In other words, `P` will have a distribution of values throughout the domain.

If the parameter is defined as `P = 10*U`, where `U` has been declared as a `VARIABLE`, then whenever **`P`** is used in the script, it represents 10 times the current value of `U` at each point of the domain, and at each stage of the solution process. That is, the single definition `P = 10*U` implies repeated re-evaluation as necessary throughout the computation.

2.3 Some Common Variations

2.3.1 Controlling Accuracy

FlexPDE applies a consistency check to integrals of the PDE's over the mesh cells. From this it estimates the relative uncertainty in the solution variables and compares this to an accuracy tolerance. If any mesh cell exceeds the tolerance, that cell is split, and the solution is recomputed.

The error tolerance is called `ERRLIM`, and can be set in the `SELECT` section of the script.

The default value of `ERRLIM` is 0.002, which means that FlexPDE will refine the mesh until the estimated error in any variable (relative to the variable range) is less than 0.2% over every cell of the mesh.

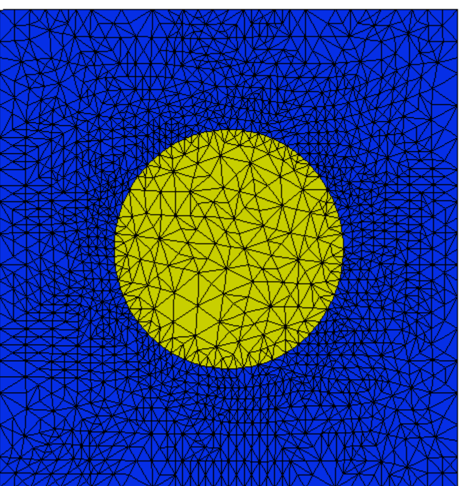
> ***Note**: This does not mean that FlexPDE can guarantee that the solutions is accurate to 0.2% over the domain. Individual cell errors may cancel or accumulate in ways that are hard to predict.*

In our sample problem, we can insert the statement

```
SELECT ERRLIM=1e-5
```

as a new section. This tells FlexPDE to split any cell in which the consistency check implies an error of more than 0.001% over the cell.

FlexPDE refines the mesh twice, and completes with a mesh that looks like this:



In this particular case, the result plots are not noticeably different from the default case.

> ***Note**: In time-dependent problems, spatial and temporal errors are both set by ERRLIM, but they can also be independently controlled by* XERRLIM *and* TERRLIM. *See the Problem Descriptor Reference* 147.

2.3.2 Computing Integrals

In many cases, it is an integral of some function that is of interest in the solution of a PDE problem. FlexPDE has an extensive repertoire of integration facilities, including volume integrals, surface integrals on bounding surfaces and line integrals on bounding lines. The two-dimensional forms are

- Result = LINE_INTEGRAL(expression, boundary_name)

  Computes the integral of expression over the named boundary.
  Note: BINTEGRAL is a pseudonym for LINE_INTEGRAL.

- Result = VOL_INTEGRAL(expression, region_name)

  Computes the integral of expression over the named region.
  If region_name is omitted, the integral is over the entire domain.

> ***Note**:* INTEGRAL *is a pseudonym for* VOL_INTEGRAL.

> ***Note**: In 2D Cartesian geometry,* AREA_INTEGRAL *is also the same as* VOL_INTEGRAL*, since the*

domain is assumed to have a unit thickness in Z.

In our example problem, we might define

```
DEFINITIONS
    { the total flux across 'ring':
        (recall that 'ring' is the name of the boundary of 'blob') }
    Tflux = LINE_INTEGRAL(NORMAL(-k*grad(Phi)), 'ring')
    { the total heat energy in 'blob': }
    Tenergy = VOL_INTEGRAL(Phi, 'blob')
```

In the case of internal boundaries, there is sometimes a different value of the integral on the two sides of the boundary. The two values can be distinguished by further specifying the region in which the integral is to be evaluated:

```
    { the total flux across 'ring': }
    Tflux = LINE_INTEGRAL(NORMAL(-k*grad(Phi)), 'ring', 'box')
    { evaluated on the 'box' side of the boundary }
```

Note*: Three-dimensional integral forms will be addressed in a later section. A full description of integral operators is presented in the Problem Descriptor Reference section "Elements | Operators | Integral Operators*136*".*

2.3.3 Reporting Numerical Results

In many cases, there are numerical quantities of interest in evaluating or classifying output plots. Any plot command can be followed by the REPORT statement:

```
REPORT value AS "title"
```

Or just

```
REPORT value
```

Any number of REPORTs can be requested following any plot, subject to the constraint that the values are printed on a single line at the bottom of the plot, and too many reports will run off and be lost.

For instance, we might modify the contour plot of our example plot to say

```
CONTOUR(Phi)  REPORT(k)  REPORT(INTEGRAL(Phi, 'blob'))
```

On running the problem, we might see something like this at the bottom of the plot:

```
ex1:  Grid#1  p2  Nodes=1121 Cells=530 RMS Err= 5.e-5
k=  1.000000  INTEGRAL(Phi, 'blob')=  0.392695  Integral=  1.999999
```

2.3.4 Summarizing Numerical Results

A special form of plot command is the SUMMARY. This plot command does not generate any pictorial output, but instead creates a page for the placement of numerous REPORTs.

SUMMARY may be given a text argument, which will be printed as a header.

For example,

```
SUMMARY
    REPORT(k)
    REPORT(INTEGRAL(Phi,'blob')) as "Heat energy in blob"
    REPORT('no more to say')
```

In our sample, we will see a separate report page with the following instead of a graphic:

```
SUMMARY

k= 1.000000
Heat energy in blob= 0.392695
no more to say
```

2.3.5 Parameter Studies Using STAGES

FlexPDE supports a facility for performing parameter studies within a single invocation. This facility is referred to as "staging". Using staging, a problem can be solved repeatedly, with a range of values for a single parameter or a group of parameters.

The fundamental form for invoking a staged run is to define one or more parameters as STAGED:

```
DEFINITIONS
    Name = STAGED(value1, value2, ….)
```

The problem will be re-run as many times as there are values in the value list, with "Name" taking on consecutive values from the list in successive runs.

If the STAGED parameter does not affect the domain dimensions, then each successive run will use the result and mesh from the previous run as a starting condition.

> ***Note**: This technique can also be used to approach the solution of a strongly nonlinear problem, by starting with a linear system and gradually increasing the weight on a nonlinear component.*

If the STAGED parameter is used as a dimension in the domain definition, then each successive run will be restarted from the domain definition, and there will be no carry-over of solutions from one run to the next.

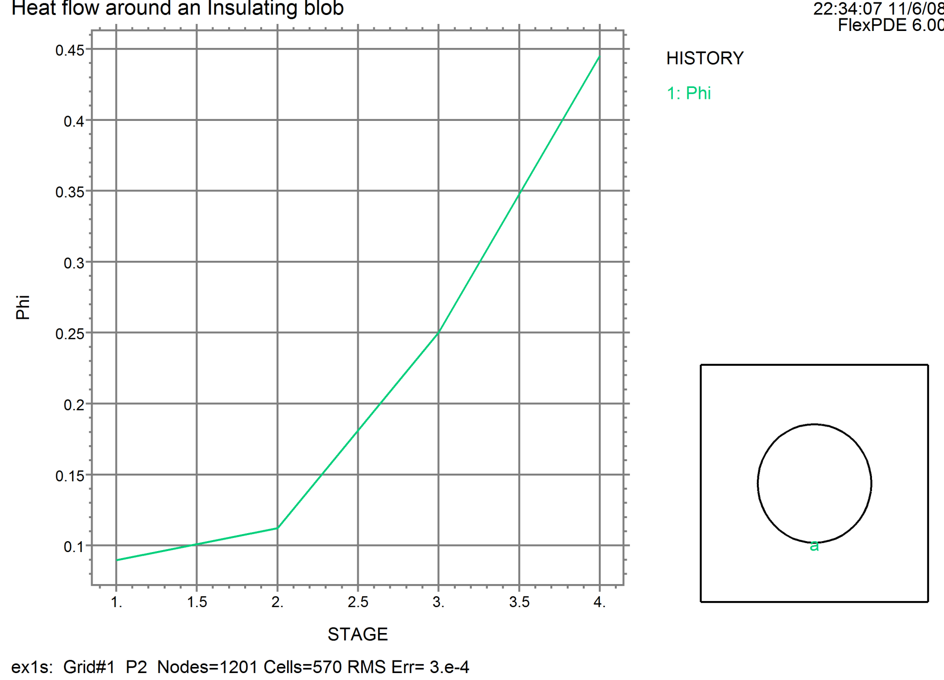
As for time-dependent problems (which we will discuss later), variation of arbitrary quantities across the stages of a problem can be displayed by HISTORY plots. In staged runs the history is plotted against stage number.

As an example, let us run our sample heat flow problem for a range of conductivities and plot a history of the top edge temperature.

We will modify the definition of **K** in the insulator as follows:

```
DEFINITIONS
    Kins = STAGED(0.01, 0.1, 1, 10)
    { Notice that the STAGED specification must appear at the initial declaration of a
    name.  It cannot be used in a regional redefinition. }
...
    REGION 2     'blob'   { the embedded blob }
        K = Kins
        START(R,0) ARC(CENTER=0,0) ANGLE=360
...
HISTORY(Phi) AT (0,-R)
```

When this modified descriptor is run, the history plot produces the following:



In a staged run, all PLOTS and MONITORS requested will be presented for each stage of the run.

Other Staging Controls

- The global selector STAGES can be used to control the number of stages to run. If this selector appears, it overrides any STAGED lists in the DEFINITIONS section (lists shorter than STAGES will report an error). It also defines the global name STAGE, which can be used subsequently in arithmetic expressions. See the Problem Descriptor Reference 164 for details.

- The default action is to proceed at once from one stage to the next, but you can cause FlexPDE to pause while you examine the plots by placing the command AUTOSTAGE=OFF in the SELECT section of the script.

Note: *The STAGE facility can only be used on steady-state problems. It cannot be used with time dependent problems.*

2.3.6 Cylindrical Geometry

In addition to two-dimensional Cartesian geometry, FlexPDE can solve problems in axisymmetric cylindrical coordinates, (r,z) or (z,r).

Cylindrical coordinates are invoked in the COORDINATES section of the script. Two forms are available, XCYLINDER and YCYLINDER. The distinction between the two is merely in the orientation of the graphical displays.

- XCYLINDER places the rotation axis of the cylinder, the Z coordinate, along the abscissa (or "x"-axis) of

the plot, with radius along the ordinate. Coordinates in this system are (Z,R)
- YCYLINDER places the rotation axis of the cylinder, the Z coordinate, along the ordinate (or "y" axis) of the plot, with radial position along the abscissa. Coordinates in this system are (R,Z)

Either form may optionally be followed by a parenthesized renaming of the coordinates. Renaming cannot be used to change the geometric character of the coordinate. Radius remains radius, even if you rename it "Z".

The default names are

XCYLINDER implies XCYLINDER('Z','R').
YCYLINDER implies YCYLINDER('R','Z').

2.3.6.1 Integrals In Cylindrical Geometry

The VOL_INTEGRAL (alias INTEGRAL) operator in Cylindrical geometry is weighted by 2*PI*R, representing the fact that the equations are solved in a revolution around the axis.

An integral over the cross-sectional area of a region may be requested by the operator AREA_INTEGRAL. This form differs from VOL_INTEGRAL in that the 2*PI*R weighting is absent.

Similarly, the operator SURF_INTEGRAL will form the integral over a boundary, analogous to the LINE_INTEGRAL operator, but with an area weight of 2*PI*R.

2.3.6.2 A Cylindrical Example

Let us now convert our Cartesian test problem into a cylindrical one. If we rotate the box and blob around the left boundary, we will form a torus between two circular plates (like a donut in a round box).

These changes will be required:

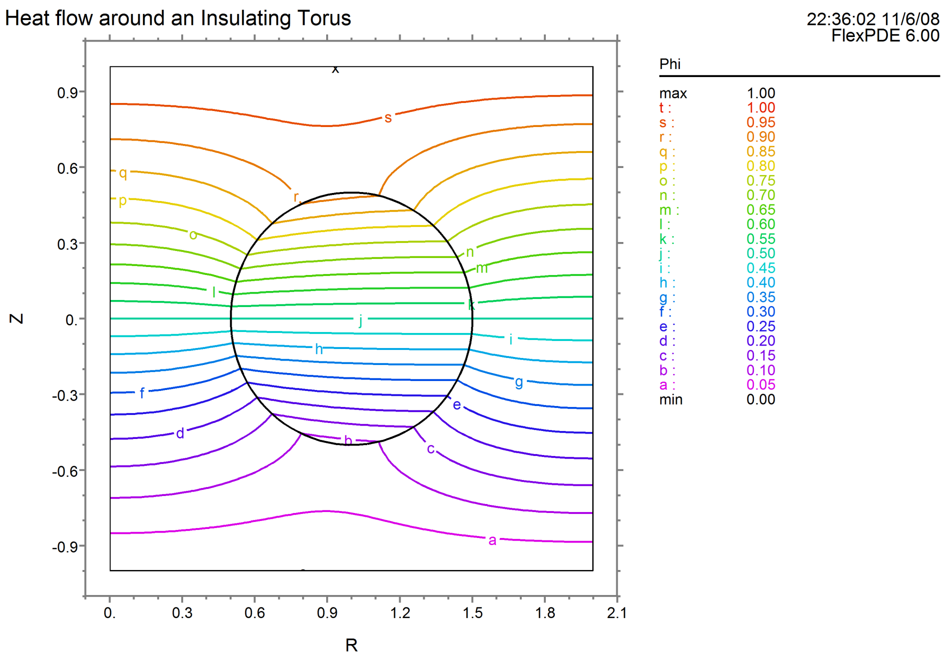
- We must offset the coordinates, so the left boundary becomes R=0.
- Since we want the rotation axis in the Y-direction, we must use YCYLINDER coordinates.
- Since 'R' is now a coordinate name, we must rename the 'R' used for the blob radius.

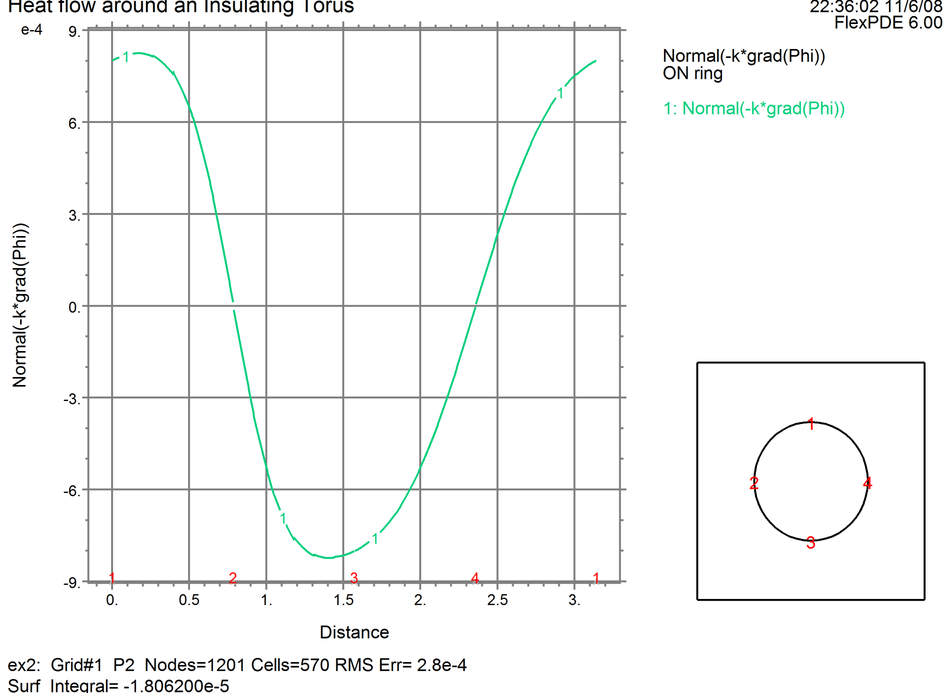
The full script, converted to cylindrical coordinates is then:

```
TITLE 'Heat flow around an Insulating Torus'
COORDINATES
    YCYLINDER
VARIABLES
    Phi          { the temperature }
DEFINITIONS
    K = 1        { default conductivity }
    Rad = 0.5    { blob radius (renamed)}
EQUATIONS
    Div(-k*grad(phi)) = 0
BOUNDARIES
    REGION 1 'box'
    START(0,-1)
        VALUE(Phi)=0     LINE TO (2,-1)
        NATURAL(Phi)=0 LINE TO (2,1)
        VALUE(Phi)=1     LINE TO (0,1)
        NATURAL(Phi)=0 LINE TO CLOSE
```

```
    REGION 2      'blob'   { the embedded blob }
    k = 0.001
    START 'ring' (1,Rad)
        ARC(CENTER=1,0) ANGLE=360 TO CLOSE
PLOTS
    CONTOUR(Phi)
    VECTOR(-k*grad(Phi))
    ELEVATION(Phi) FROM (1,-1) to (1,1)
    ELEVATION(Normal(-k*grad(Phi))) ON 'ring'
END
```

The resulting contour and boundary plot look like this:





2.3.7 Time Dependence

Unless otherwise defined, FlexPDE recognizes the name "T" (or "t") as representing time. If references to time appear in the definitions or equations, FlexPDE will invoke a solution method appropriate to initial-value problems.

FlexPDE will apply a heuristic control on the timestep used to track the evolution of the system. Initially, this will be based on the time derivatives of the variables, and later it will be chosen so that the time behavior of the variables is nearly quadratic. This is done by shortening or lengthening the time intervals so that the cubic term in a Taylor expansion of the variables in time is below the value of the global selector ERRLIM.

In time dependent problems, several new things must be specified:

- The THRESHOLD of meaningful values for each variable (if not apparent from initial values).
- The time-dependent PDE's
- The time range of interest,
- The times at which plots should be produced
- Any history plots that may be desired

> ***Note**: FlexPDE can treat only first derivatives in time. Equations that are second-order in time must be split into two equations by defining an intermediate variable.*

The time range is specified by a new script section

```
TIME start TO finish
```

Plot times are specified by preceding any block of plot commands by a time control, in which specific times may be listed, or intervals and end times, or a mixture of both:

```
FOR  T = t1,  t2 BY step TO t3 ....
```

We can convert our heat flow problem to a time dependent one by including a time term in the heat equation:

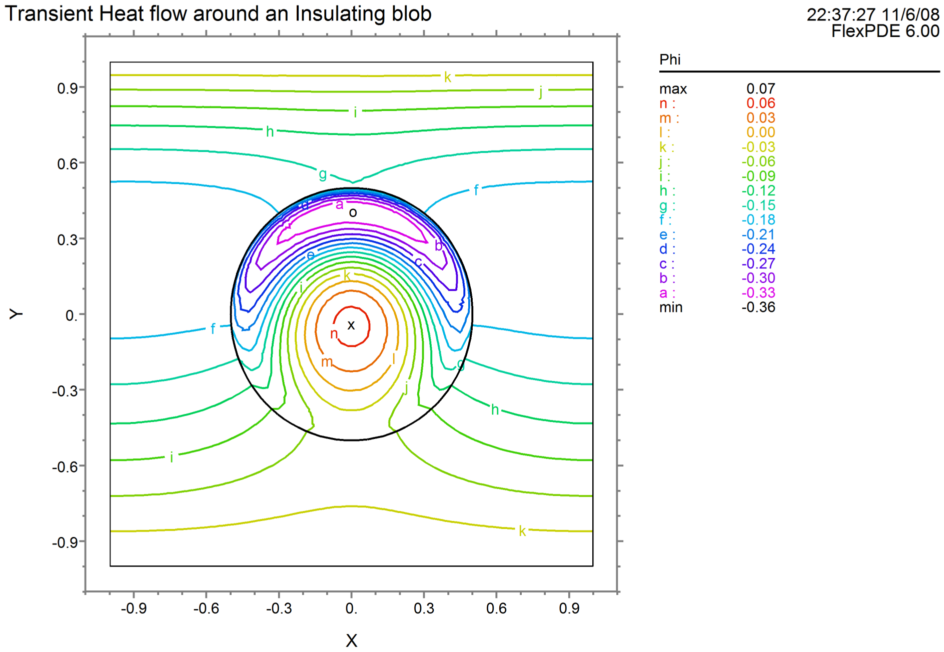
```
Div(k*grad(Phi)) = c*dt(Phi)
```

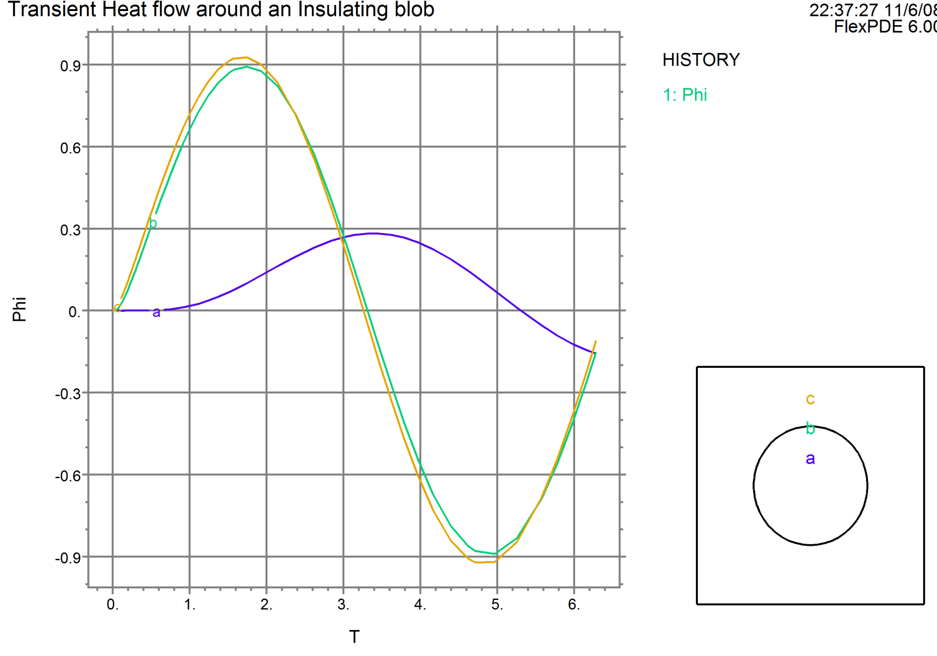
To make things interesting, we will impose a sinusoidal driving temperature at the top plate, and present a history plot of the temperature at several internal points.

The whole script with pertinent modifications now looks like this:

```
TITLE 'Transient Heat flow around an Insulating blob'
VARIABLES
    Phi (threshold=0.01)      { the temperature }
DEFINITIONS
    K = 1          { default conductivity }
    C = 1          { default heat capacity }
    R = 1/2
EQUATIONS
    Div(-K*grad(phi)) + C*dt(Phi) = 0
BOUNDARIES
    REGION 1 'box'
    START(-1,-1)
        VALUE(Phi)=0          LINE TO (1,-1)
        NATURAL(Phi)=0        LINE TO (1,1)
        VALUE(Phi)=sin(t)     LINE TO (-1,1)
        NATURAL(Phi)=0        LINE TO CLOSE
    REGION 2     'blob'   { the embedded blob }
    K = 0.001
    C = 0.1
    START(R,0)
        ARC(CENTER=0,0) ANGLE=360
TIME 0 TO 2*pi
PLOTS
    FOR T = pi/2 BY pi/2 TO 2*pi
        CONTOUR(Phi)
        VECTOR(-K*grad(Phi))
        ELEVATION(Phi) FROM (0,-1) to (0,1)
HISTORIES
    HISTORY(Phi) AT (0,r/2) (0,r) (0,3*r/2)
END
```

At the end of the run (t=2*pi), the contour and history look like this:





2.3.7.1 Bad Things To Do In Time Dependent Problems

Inconsistent Initial Conditions and Instantaneous Switching

If you start off a time-dependent calculation with initial conditions that are inconsistent, or turn on

boundary values instantaneously at the start time (or some later time), you induce strong transient signals in the system. This will cause the time step, and probably the mesh size as well, to be cut to tiny values to track the transients.

Unless it is specifically the details of these transients that you want to know, you should start with initial conditions that are a consistent solution to a steady problem, and then turn on the boundary values, sources or driving fluxes over a time interval that is meaningful in your problem.

It is a common mistake to think that simply turning on a source is a smooth operation. It is not. Mathematically, the turn-on time is significantly less that a femtosecond (zero, in fact), with attendant terahertz transients. If that's the problem you pose, then that's the problem FlexPDE will try to solve. More realistically, you should turn on your sources over a finite time. Electrical switches take milliseconds, solid state switches take microseconds. But if you only want to see what happens after a second or two, then fuzz the turn-on.

Turning on a driving flux or a volume source is somewhat more gentle than a boundary value, because it implies a finite time to raise the boundary value to a given level. But there is still a meaningful time interval over which to turn it on.

2.3.8 Eigenvalues and Modal Analysis

FlexPDE can also compute the eigenvalues and eigenfunctions of a PDE system.

Consider the homogeneous time-dependent heat equation as in our example above,

$$C\frac{\partial\phi}{\partial t} - \nabla \cdot K\nabla\phi = 0$$

together with homogeneous boundary conditions

$$\phi = 0$$

and/or

$$\frac{\partial\phi}{\partial n} + \alpha\phi = 0$$

on the boundary.

If we wish to solve for steady oscillatory solutions to this equation, we may assert

$$\phi(x,y,t) = \psi(x,y)\exp(-\beta t)$$

The PDE then becomes

$$\nabla \cdot K\nabla\psi + \lambda\psi = 0$$

$$\lambda = -C\beta$$

The values of λ and ψ for which this equation has nontrivial solutions are known as the eigenvalues and eigenfunctions of the system, respectively. All steady oscillatory solutions to the PDE can be made up of combinations of the various eigenfunctions, together with a particular solution that satisfies any non-homogeneous boundary conditions.

Two modifications are necessary to our basic steady-state script for the sample problem to cause FlexPDE

to solve the eigenvalue problem.

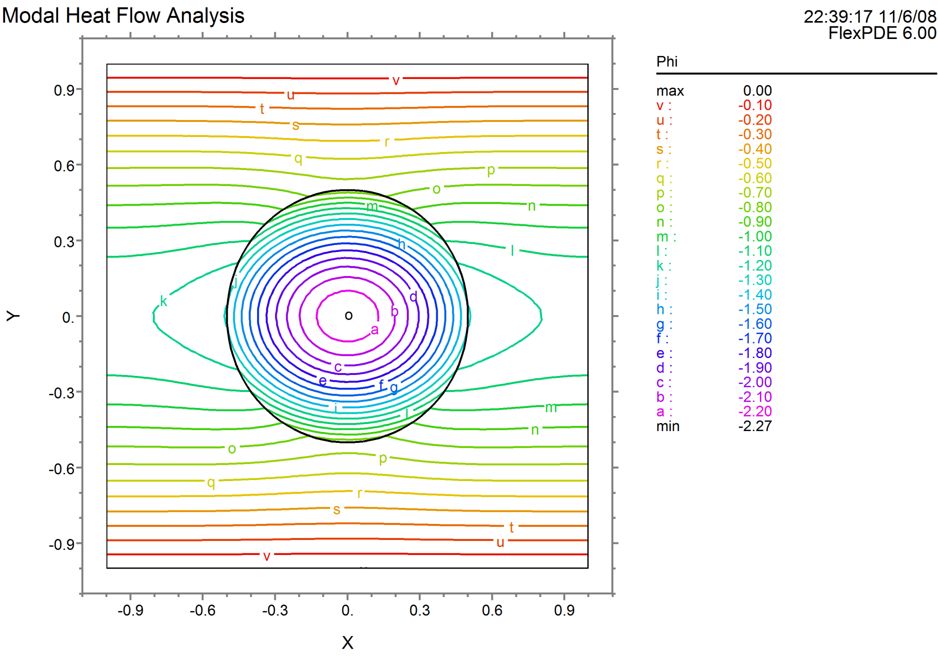
- A value must be given to the MODES parameter in the SELECT section. This number determines the number of distinct values of λ that will be calculated. The values reported will be those with lowest magnitude.
- The equation must be written using the reserved name LAMBDA for the eigenvalue.
- The equation should be written so that values of LAMBDA are positive, or problems with the ordering during solution will result. The full descriptor for the eigenvalue problem is then:

```
TITLE 'Modal Heat Flow Analysis'
SELECT
    modes=4
VARIABLES
    Phi             { the temperature }
DEFINITIONS
    K = 1           { default conductivity }
    R = 0.5         { blob radius }
EQUATIONS
    Div(k*grad(Phi)) + LAMBDA*Phi = 0
BOUNDARIES
    REGION 1 'box'
    START(-1,-1)
        VALUE(Phi)=0     LINE TO (1,-1)
        NATURAL(Phi)=0 LINE TO (1,1)
        VALUE(Phi)=0     LINE TO (-1,1)
        NATURAL(Phi)=0 LINE TO CLOSE
    REGION 2       'blob'   { the embedded blob }
    k = 0.2                 { This value makes more interesting pictures }
    START 'ring' (R,0)
        ARC(CENTER=0,0) ANGLE=360 TO CLOSE
PLOTS
    CONTOUR(Phi)
    VECTOR(-k*grad(Phi))
    ELEVATION(Phi) FROM (0,-1) to (0,1)
    ELEVATION(Normal(-k*grad(Phi))) ON 'ring'
END
```

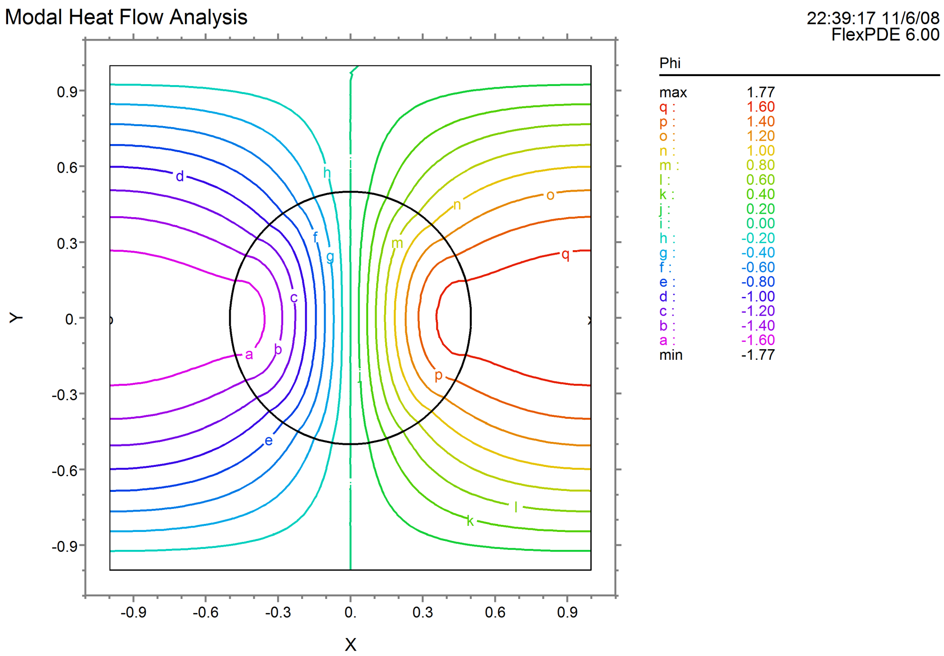
The solution presented by FlexPDE will have the following characteristics:

- The full set of PLOTS will be produced for each of the requested modes.
- An additional plot page will be produced listing the eigenvalues.
- The mode number and eigenvalue will be reported on each plot.
- LAMBDA is available as a defined name for use in arithmetic expressions.

The first two contours are as follows:



ex4: Grid#1 P2 Nodes=1201 Cells=570 RMS Err= 3.2e-4
Mode 1 Lambda= 2.0761 Integral= -3.396896



ex4: Grid#1 P2 Nodes=1201 Cells=570 RMS Err= 3.2e-4
Mode 2 Lambda= 3.4320 Integral= -3.761614e-5

2.3.8.1 The Eigenvalue Summary

When running an Eigenvalue problem, FlexPDE automatically produces an additional plot displaying a summary of the computed eigenvalues.

If the user specifies a SUMMARY plot, then this plot will supplant the automatic summary, allowing the user to add reports to the eigenvalue listing.

For example, we can add to our previous descriptor the plot specification:

```
SUMMARY
    REPORT(lambda)
    REPORT(integral(phi))
```

This produces the following report on the summary page:

```
Modal Heat Flow Analysis                    22:15:55 5/23/05
                                              FlexPDE 5.0.0

   Eigenvalues:

   Mode   1: lambda= 2.076144  integral(phi)=-3.408079
   Mode   2: lambda= 3.431960  integral(phi)=-4.340801e-6
   Mode   3: lambda= 5.704378  integral(phi)=-1.050399
   Mode   4: lambda= 6.752271  integral(phi)= 9.194491e-4
```

2.4 Addressing More Difficult Problems

If heat flow on a square were all we wanted to do, then there would probably be no need for FlexPDE. The power of the FlexPDE system comes from the fact that almost any functional form may be specified for the material parameters, the equation terms, or the output functions. The geometries may be enormously complex, and the output specification is concise and powerful.

In the following sections, we will address some of the common situations that arise in real problems, and show how they may be treated in FlexPDE.

2.4.1 Nonlinear Coefficients and Equations

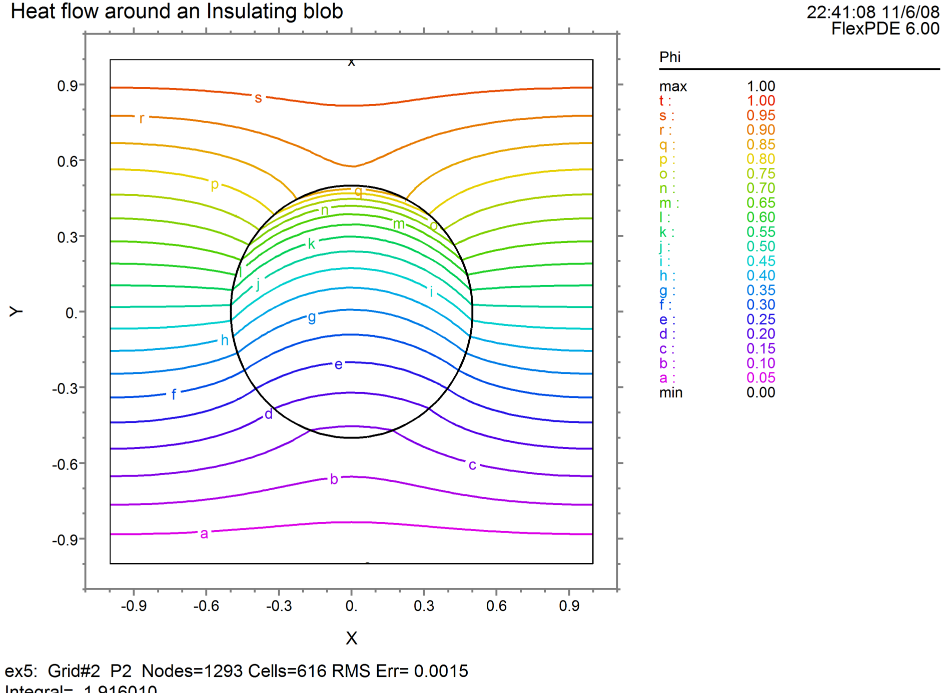
One common complication that arises is that either the terms of the equation or the material properties are complicated functions of the system variables. FlexPDE understands this, and has made full provision for handling such systems.

Suppose, for example, that the conductivity in the 'blob' of our example problem were in fact a strong function of the temperature. Say, for example, that K=exp(-5*phi). The solution couldn't be simpler. Just define it the way you want it and click "run":

```
...
REGION 2 'blob'  { the embedded blob }
    k = exp(-5*phi)
...
```

The appearance of a nonlinear dependence will automatically activate the nonlinear solver, and all the dependency details will be handled by FlexPDE.

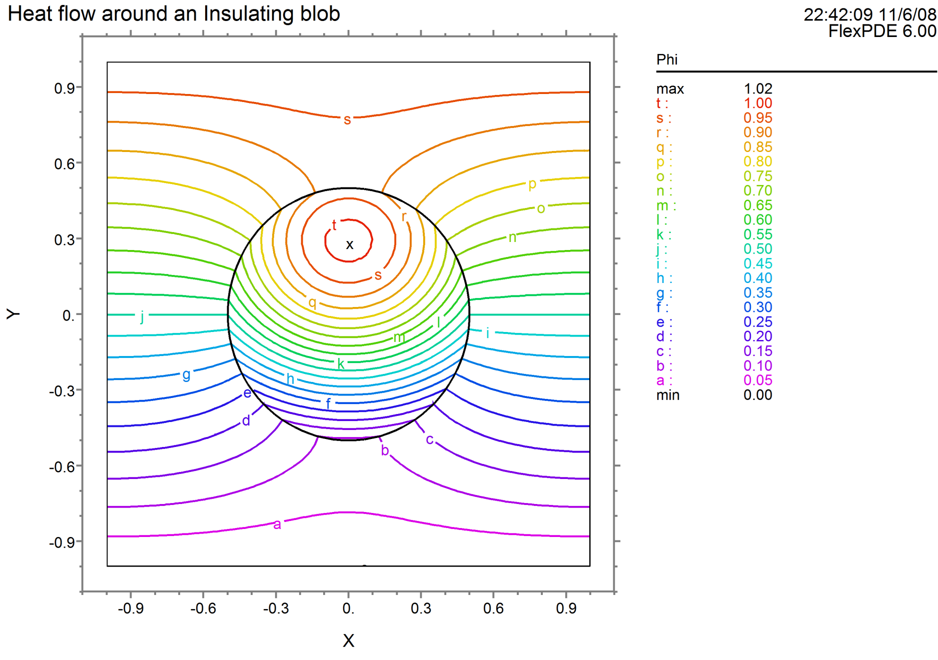
The modified result appears immediately:



Nonlinear terms in the equation are just as easy. If our system has a nonlinear sinusoidal source, for example, we may type:

```
EQUATIONS
    Div(k*grad(phi)) + 0.01*phi*sin(phi) = 0
```

Click "run", and the solution appears:



2.4.1.1 Complications Associated with Nonlinear Problems

Actually, nonlinear problems are frequently more difficult than we have implied above, for several reasons.

- Nonlinear problems can have more than one solution.
- A nonlinear problem may not, in fact, have a solution at all.

FlexPDE uses a Newton-Raphson iteration process to solve nonlinear systems. This technique can be very sensitive to the initial estimate of the solution. If the starting conditions are too far from the actual solution, it may be impossible to find the answer, even though it might be quite simple from a different starting value.

There are several things that can be done to help a nonlinear problem find a solution:

- Provide as good an initial value as you can, using the INITIAL VALUES section of the script.
- Ensure that the boundary conditions are consistent.
- Use STAGES to progress from a linear to a nonlinear system, allowing the linear solution to provide initial conditions for the nonlinear one.
- Pose the problem as a time-dependent one, with time as an artificial relaxation dimension.
- Use SELECT CHANGELIM to limit the excursion at each step and force FlexPDE to creep toward a solution.
- Use MONITORS to display useful aspects of the solution, to help identify troublesome terms.

We will return in a later section 110 to the question of intransigent nonlinear problems.

2.4.2 Natural Boundary Conditions

The term "natural boundary condition" usually arises in the calculus of variations, and since the finite element method is fundamentally one of minimization of an error functional, the term arises also in this context.

The term has a much more intuitive interpretation, however, and it is this which we will try to present.

Consider a **Laplace equation**,

$$\nabla \cdot \nabla u = 0$$

The **Divergence Theorem** says that the integral of this equation over all space is equal merely to the integral over the bounding surface of the normal component of the flux,

$$\iint_A div(grad(u))dA = \oint_S n \cdot grad(u)dl$$

(we have presented the equation in two dimensions, but it is valid in three dimensions as well).

The surface value of $n \cdot grad(u)$ is in fact the "natural boundary condition" for the Laplace (and Poisson) equation. It is the way in which the system *inside* interacts with the system *outside*. It is the (negative of the) flux of the quantity u that crosses the system boundary.

The **Divergence Theorem** is a particular manifestation of the more general process of **Integration by Parts**. You will remember the basic rule,

$$\int_a^b udv = uv\Big|_a^b - \int_a^b vdu$$

The term uv is evaluated at the ends of the integration interval and gives rise to surface terms. Applied to the integration of a divergence, integration by parts produces the Divergence Theorem.

FlexPDE applies integration by parts to all terms of the partial differential equations that contain second-order derivatives of the system variables. In the Laplace equation, of course, this means the only term that appears.

In order for a solution of the Laplace equation (for example) to be achieved, one must specify at all points of the boundary either the value of the variable (in this case, u) or the value of $n \cdot grad(u)$.

In the notation of FlexPDE,

`VALUE(u)=u1`	supplies the former, and
`NATURAL(u)=F`	supplies the latter.

In other words,

The NATURAL boundary condition statement in FlexPDE supplies the value of the surface flux, as that flux is defined by the integration of the second-order terms of the PDE by parts. The default boundary condition for FlexPDE is NATURAL(VARIABLE)=0.

Note: On an internal boundary the NATURAL defines the difference in flux between the two adjacent regions, producing a source or sink at that boundary.

Consistent with our discussion of nonlinear equations, the value given for the surface flux may be a nonlinear value.

The radiation loss from a hot body, for example, is proportional to the fourth power of temperature, and the statement

```
NATURAL(u) = -k*u^4
```

is a perfectly legal boundary condition for the Laplace equation in FlexPDE.

2.4.2.1 Some Typical Cases

Since **integration by parts** is a fundamental mathematical operation, it will come as no surprise that its application can lead to many of the fundamental rules of physics, such as Ampere's Law.

For this reason, the Natural boundary condition is frequently a statement of very fundamental conservation laws in many applications.

But it is not always obvious at first what its meaning might be in equations which are more elaborate than the Laplace equation.

So let us first list some basic terms and their associated natural boundary condition contributions (we present these rules for two-dimensional geometry, but the three-dimensional extensions are readily seen).

- Applied to the term $\partial f(u)/\partial x$, integration by parts yields

  $$\iint \frac{\partial f(u)}{\partial x} dxdy = \oint f(u)dy = \oint f(u)\alpha dl$$

  Here α is the x-direction cosine of the surface normal and dl is the differential path length. Since FlexPDE applies integration by parts only to second order terms, this rule is applied only if the function $f(u)$ contains further derivatives of u. Similar rules apply to derivatives with respect to other coordinates.

- Applied to the term $\partial^2 f(u)/\partial x^2$, integration by parts yields

  $$\iint \frac{\partial^2 f(u)}{\partial x^2} dxdy = \oint \frac{\partial f(u)}{\partial x} dy = \oint \frac{\partial f(u)}{\partial x} \alpha dl$$

  Since this term is second order, it will always result in a contribution to the natural boundary condition.

- Applied to the term $\nabla \cdot \vec{F}(u)$, integration by parts yields the Divergence Theorem

  $$\iint \nabla \cdot \vec{F}(u) dxdy = \oint \vec{F}(u) \cdot \hat{n} dl$$

  Here $\hat{n}$ is the outward surface normal unit vector.

  As with the x-derivative case, integration by parts will not be applied unless the vector $\vec{F}$ itself contains further derivatives of u.

- Applied to the term $\nabla \times \vec{F}(u)$, integration by parts yields the Curl Theorem

  $$\iint \nabla \times \vec{F}(u) dxdy = \oint \hat{n} \times \vec{F}(u) dl$$

Using these formulas, we can examine what the natural boundary condition means in several common cases:

The Heat Equation

Div(-k*grad(Temp)) + Source = 0

Natural(Temp) = normal(-k*grad(Temp)) { outward surface-normal flux }

(Notice that we have written the PDE in terms of heat flux with the negative sign imbedded in the equation. If the sign is left out, the sign of the Natural is reversed as well.)

One-dimensional heat equation

dx(-k*dx(Temp)) + Source = 0
Natural(Temp) = outward surface-normal component of flux = (-k*dx(temp)*nx),

where nx is the x-direction cosine of the surface normal.

Similar forms apply for other coordinates.

Magnetic Field Equation

curl(curl(A)/mu) = J
Natural(A) = tangential component of H = tangential(curl(A)/mu)

Convection Equation

dx(u)-dy(u)=0
Natural(u) is undefined, because there are no second-order terms.

See the section "Hyperbolic systems" for further discussion.

2.4.2.2 An Example of a Flux Boundary Condition

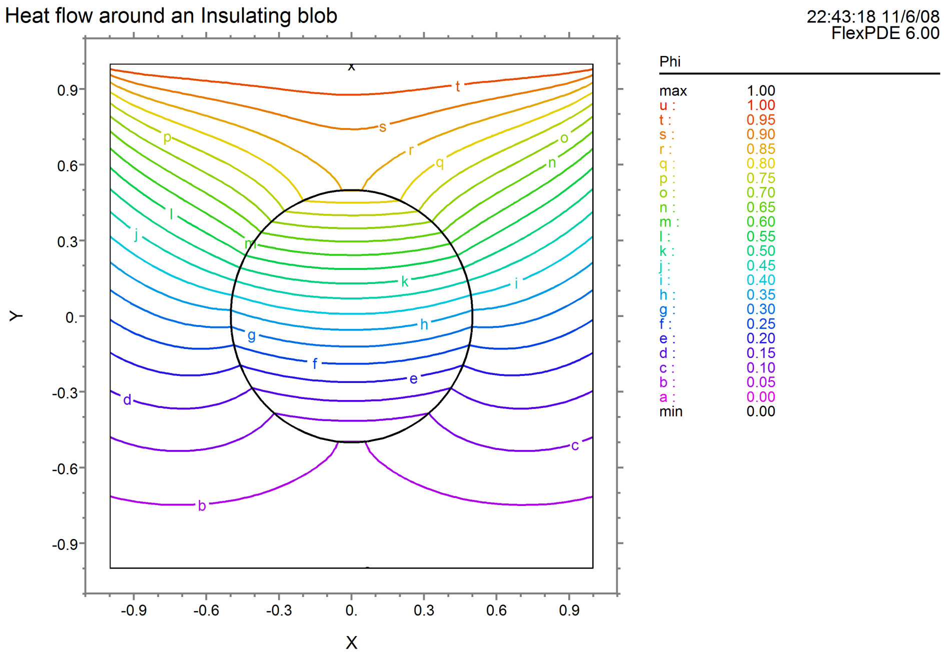
Let us return again to our heat flow test problem and investigate the effect of the Natural boundary condition. As originally posed, we specified Natural(Phi)=0 on both sidewalls. This corresponds to zero flux at the boundary. Alternatively, a convective cooling loss at the boundary would correspond to a flux

```
Flux = -K*grad(Phi) = Phi - Phi0
```

where Phi0 is a reference cooling temperature. With convectively cooled sides, our boundary specification looks like this (assuming Phi0=0):

```
REGION 1 'box'
    START(-1,-1)
        VALUE(Phi)=0          LINE TO (1,-1)
        NATURAL(Phi)=Phi      LINE TO (1,1)
        VALUE(Phi)=1          LINE TO (-1,1)
        NATURAL(Phi)=Phi      LINE TO CLOSE
```

The result of this modification is that the isotherms curve upward:



2.4.3 Discontinuous Variables

The default behavior of FlexPDE is to consider all variables to be continuous across material interfaces. This arises naturally from the finite element model, which populates the interface with nodes that are shared by the material on both sides.

FlexPDE supports discontinuities in variables at material interfaces by use of the words CONTACT and JUMP in the script language.

CONTACT(V) is a special form of NATURAL boundary condition which also causes the affected variable to be stored in duplicate nodes at the interface, capable of representing a double value.

JUMP(v) means the instantaneous change in the value of variable "v" when moving outward across an interface from inside a given material. At an interface between materials '1' and '2', JUMP(V) means (V2-V1) in material '1', and (V1-V2) in material '2'.

The expected use of JUMP is in a CONTACT Boundary Condition statement on an interior boundary. The combination of CONTACT and JUMP causes a line or surface source to be generated proportional to the difference between the two values.

JUMP may also be used in other boundary condition statements, but it is assumed that the argument of the JUMP is a variable for which a CONTACT boundary condition has been specified. See the example "Samples | Usage | Discontinuous_Variables | Contact_Resistance_Heating.pde" for an example of this kind of use.

The interpretation of the JUMP operator follows the model of contact resistance, as explained in the next section.

2.4.3.1 Contact Resistance

The problem of contact resistance between two conductors is a typical one requiring discontinuity of the modeled variable.

In this problem, a very thin resistive layer causes a jump in the temperature or voltage on the two sides of an interface. The magnitude of the jump is proportional to the heat flux or electric current flowing across the resistive film. In microscopic analysis, of course, there is a physical extent to the resistive material. But its dimensions are such as to make true modelling of the thickness inconvenient in a finite element simulation.

In the contact resistance case, the heat flux across a resistive interface between materials '1' and '2' as seen from side '1' is given by

```
F1 = -K1*dn(T) = -(T2-T1)/R
```

where F1 is the value of the outward heat flux, K1 is the heat conductivity, dn(T) is the outward normal derivative of T, R is the resistance of the interface film, and T1 and T2 are the two values of the temperature at the interface.

As seen from material '2',

```
F2 = -K2*dn(T) = -(T1-T2)/R = -F1
```

Here the normal has reversed sign, so that the outflow from '2' is the negative of the outflow from '1', imposing energy conservation.

The Natural Boundary Condition for the heat equation

```
div(-K*grad(T)) = H
```

is given by the divergence theorem as

```
Natural(T) = -K*dn(T),
```

representing the outward heat flux.

This flux can be related to a discontinuous variable by use of the CONTACT boundary condition in place of the NATURAL.

The FlexPDE expression JUMP(T) is defined as (T2-T1) in material '1' and (T1-T2) in material '2'.

The representation of the contact resistance boundary condition is therefore

```
CONTACT(T) = -JUMP(T)/R
```

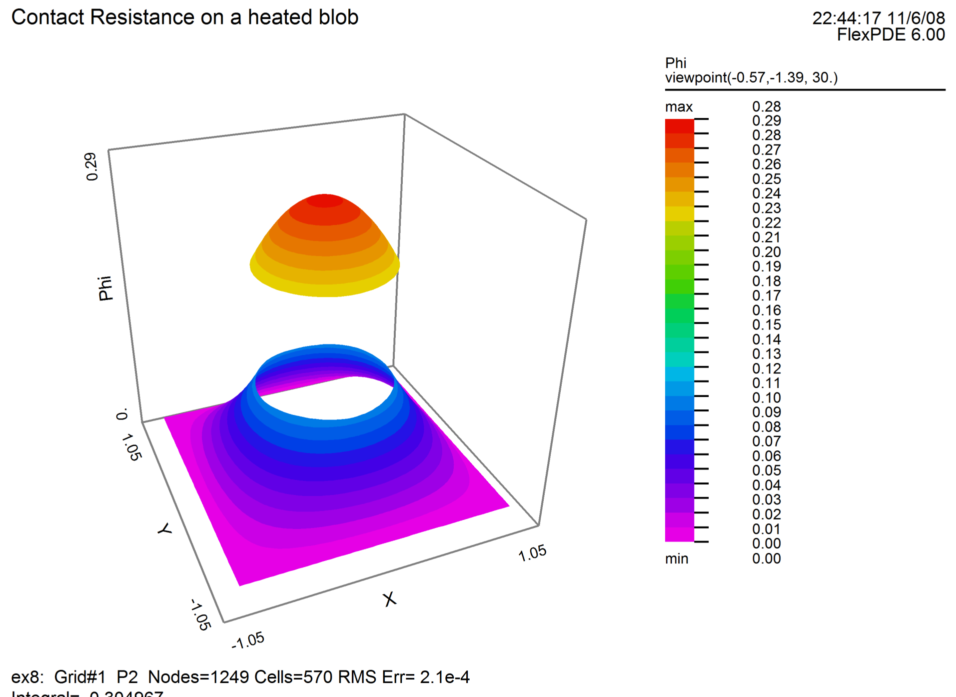
This statement means the same thing in both of the materials sharing the interface. [Notice that the sign applied to the JUMP reflects the sign of the divergence term.]

We can modify our previous example problem to demonstrate this, by adding a heat source to drive the jump, and cooling the sidewalls. The restated script is:

```
TITLE 'Contact Resistance on a heated blob'
VARIABLES
    Phi             { the temperature }
DEFINITIONS
    K = 1           { default conductivity }
    R = 0.5                 { blob radius }
    H = 0           { internal heat source }
```

```
    Res = 0.5     { contact resistance }
EQUATIONS
    Div(-k*grad(phi)) = H

BOUNDARIES
    REGION 1 'box'
        START(-1,-1)
            VALUE(Phi)=0{ cold outer walls }
            LINE TO (1,-1) TO (1,1) TO (-1,1) TO CLOSE
    REGION 2      'blob'   { the embedded blob }
        H = 1       { heat generation in the blob }
        START 'ring' (R,0)
            CONTACT(phi) = -JUMP(phi)/Res
            ARC(CENTER=0,0) ANGLE=360 TO CLOSE
PLOTS
    CONTOUR(Phi)
    SURFACE(Phi)
    VECTOR(-k*grad(Phi))
    ELEVATION(Phi) FROM (0,-1) to (0,1)
    ELEVATION(Normal(-k*grad(Phi))) ON 'ring'
END
```

The surface plot generated by running this problem shows the discontinuity in temperature:



2.4.3.2 Decoupling

Using the Contact Resistance model, one can effectively decouple the values of a given variable in two adjacent regions. In the previous example, if we replace the jump boundary condition with the statement

CONTACT(phi) = 0*JUMP(phi)

the contact resistance is infinite, and no flux can pass between the regions.

> ***Note**: The JUMP statement is recognized as a special form. Even though the apparent value of the right hand side here is zero, it is not removed by the arithmetic expression simplifier.*

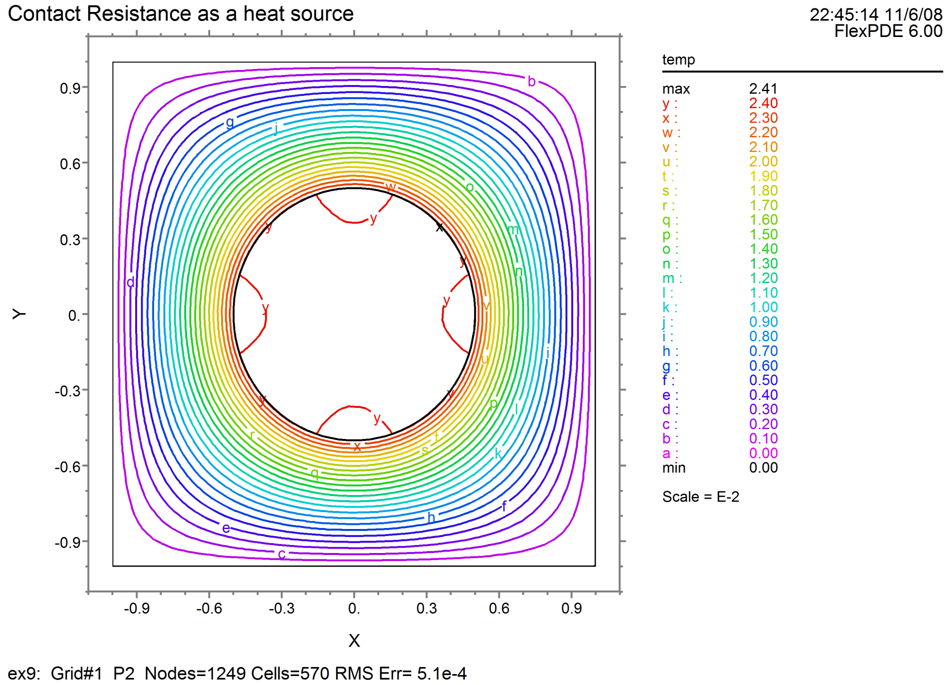
2.4.3.3 Using JUMP in problems with many variables

An expression JUMP(V) may appear in any boundary condition statement on a boundary for which the argument variable V has been given a CONTACT boundary condition.

In an electrical resistance case, for example, the voltage undergoes a jump across a contact resistance, and the current through this contact is a source of heat for a heatflow equation. The following example, though not strictly realizable physically, diagrams the technique. Notice that the JUMP of Phi appears as a source term in the Natural boundary condition for Temp. Phi, having appeared in a CONTACT boundary condition definition, is stored as a double-valued quantity, whose JUMP is available to the boundary condition for Temp. Temp, which does not appear in a CONTACT boundary condition statement, is a single-valued variable at the interface.

```
TITLE 'Contact Resistance as a heat source'
VARIABLES
    Phi             { the voltage }
    Temp            { the temperature }
DEFINITIONS
    Kd = 1          { dielectric constant }
    Kt = 1          { thermal conductivity }
    R = 0.5                 { blob radius }
    Q = 0           { space charge density }
    Res = 0.5       { contact resistance }
EQUATIONS
    Phi:    Div(-kd*grad(phi)) = Q
    Temp:Div(-kt*grad(temp)) = 0
BOUNDARIES
    REGION 1 'box'
        START(-1,-1)
            VALUE(Phi)=0{ grounded outer walls }
            VALUE(Temp)=0       { cold outer walls }
            LINE TO (1,-1) TO (1,1) TO (-1,1) TO CLOSE
    REGION 2       'blob'   { the embedded blob }
        Q = 1       { space charge in the blob }
        START 'ring' (R,0)
            CONTACT(phi) = -JUMP(phi)/Res
            { the heat source is the voltage difference times the current }
            NATURAL(temp) = -JUMP(Phi)^2/Res
            ARC(CENTER=0,0) ANGLE=360 TO CLOSE
PLOTS
    CONTOUR(Phi)        SURFACE(Phi)
    CONTOUR(temp)       SURFACE(temp)
END
```

The temperature shows the effect of the surface source:



2.5 Using FlexPDE in One-Dimensional Problems

FlexPDE treats problems in one space dimension as a degenerate case of two dimensional problems.

The construction of a problem descriptor follows the principles laid out in previous sections, with the following specializations:

- The COORDINATES specification must be CARTESIAN1, CYLINDER1 or SPHERE1
- Coordinate positions are given by one dimensional points, as in
  START(0) LINE TO (5)
- The boundary path is in fact the domain, so the boundary must not CLOSE on itself.
- Since the boundary path is the domain, boundary conditions are not specified along the path. Instead we use the existing syntax of POINT VALUE and POINT LOAD to specify boundary conditions at the end points of the domain:
  START(0) POINT VALUE(u)=0 LINE TO (5) POINT LOAD(u)=1
- Only ELEVATION and HISTORY are meaningful plots in one dimension.

Our basic example problem does not have a one-dimensional analog, but we can adapt it to an insulating spherical shell between two spherical reservoirs as follows:

```
TITLE 'Heat flow through an Insulating shell'
COORDINATES
    Sphere1
VARIABLES
    Phi             { the temperature }
DEFINITIONS
    K = 1           { default conductivity }
    R1 = 1          { the inner reservoir }
    Ra = 2          { the insulator inner radius }
```

```
    Rb = 3          { the insulator outer radius }
    R2 = 4          { the outer reservoir }
EQUATIONS
    Div(-k*grad(phi)) = 0
BOUNDARIES
    REGION 1                { the total domain }
    START(R1)    POINT VALUE(Phi)=0
        LINE TO (R2)      POINT VALUE(Phi)=1
        { note: no 'Close'! }
    REGION 2      'blob'   { the embedded layer }
    k = 0.001
    START (Ra) LINE TO (Rb)
PLOTS
    ELEVATION(Phi) FROM (R1) to (R2)
END
```

2.6 Using FlexPDE in Three-Dimensional Problems

First, a caveat:

Three-dimensional computations are not simple. We have tried to make FlexPDE as easy as possible to use, but the setup and interpretation of 3D problems relies heavily on the concepts explained in 2D applications of FlexPDE. Please do not try to jump in here without reading the preceding 2D discussion.

Extrusion:

FlexPDE constructs a three-dimensional domain by extruding a two-dimensional domain into a third dimension. This third dimension can be divided into layers, possibly with differing material properties and boundary conditions in each layer. The interface surfaces which separate the layers need not be planar, but there are some restrictions placed on the shapes that can be defined in this way.

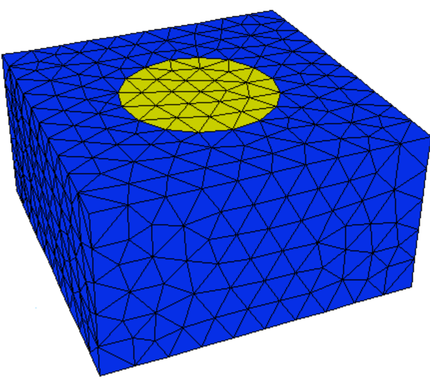
The finite element model constructed by FlexPDE in three-dimensional domains is fully general. The domain definition process is not.

2.6.1 The Concept of Extrusion

The fundamental idea of extrusion is quite simple; a square extruded into a third dimension becomes a cube; a circle becomes a cylinder. Given spherical layer surfaces, the circle can also become a sphere.

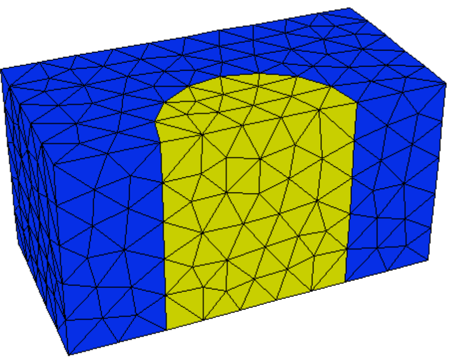
> ***Note**: It is important to consider carefully the characteristics of any given problem, to determine the orientation most amenable to extrusion.*

What happens if we extrude our simple 2D heat flow problem into a third dimension? Setting the extrusion distance to half the plate spacing, we get a cylinder imbedded in a brick, as we see in the following figure:



A cross-section at any value of Z returns the original 2D figure.

A cross-section cut at Y=0 shows the extruded structure:



2.6.2 Extrusion Notation in FlexPDE

Performing the extrusion above requires three basic changes in the 2D script:

- The COORDINATES section must specify CARTESIAN3.
- A new EXTRUSION section must be added to specify the layering of the extrusion.
- PLOTS and MONITORS must be modified to specify any cut planes or surfaces on which the display is to be computed.

There are two forms for the EXTRUSION section, the elaborate form and the shorthand form. In both cases, the layers of the model are built up in order from small to large Z.

In the elaborate form, the dividing SURFACES and the intervening LAYERS are each named explicitly, with algebraic formulas given for each dividing surface.

> ***Note**: With this usage, we have overloaded the word* SURFACE*. As a plot command, it can mean a form of graphic output in which the data are presented as a three-dimensional surface; or, in this new case, it can mean a dividing surface between extrusion layers. The distinction between the two uses should be clear from the context.*

In the simple case of our extruded cylinder in a square, it looks like this:

```
EXTRUSION
    SURFACE 'Bottom'  z=0
        LAYER 'Everything'
    SURFACE 'Top'         z=1
```

The bottom and top surfaces are named, and given simple planar shapes.

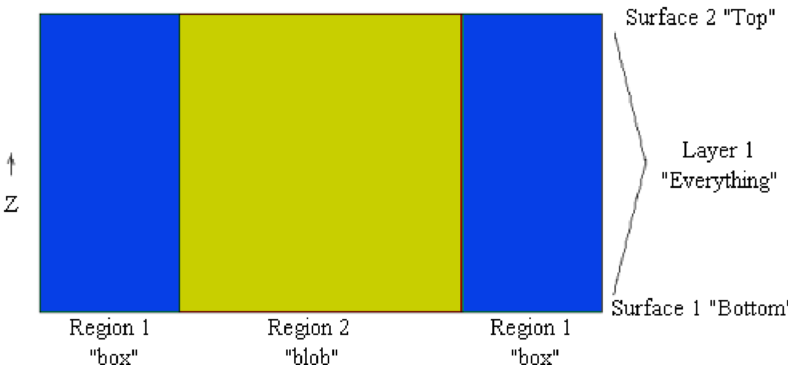
The layer between these two surfaces comprises everything in the domain, so we can name it 'Everything'.

In the shorthand form, we merely state the Z-formulas:

```
EXTRUSION      z = 0, 1
```

In this case, the layers and surfaces must later be referred to by number. The first surface, z=0, is identified as surface 1. The second surface, z=1, as surface 2.

Notice that there is no distinction, as far as the layer definition is concerned, between the parts of the layer which are in the cylinder and the parts of the layer which are outside the cylinder. This distinction is made by combining the LAYER concept with the REGION concept of the 2D base plane representation. In a vertical cross-section we can label the parts as follows:



Notice that the cylinder can be uniquely identified as the intersection of the 'blob' region of the base plane with the 'Everything' layer of the extrusion.

2.6.3 Layering

Now suppose that we wish to model a canister rather than a full length cylinder. This requires that we break up the material stack above region 2 into three parts, the canister and the continuation of the box material above and below it.

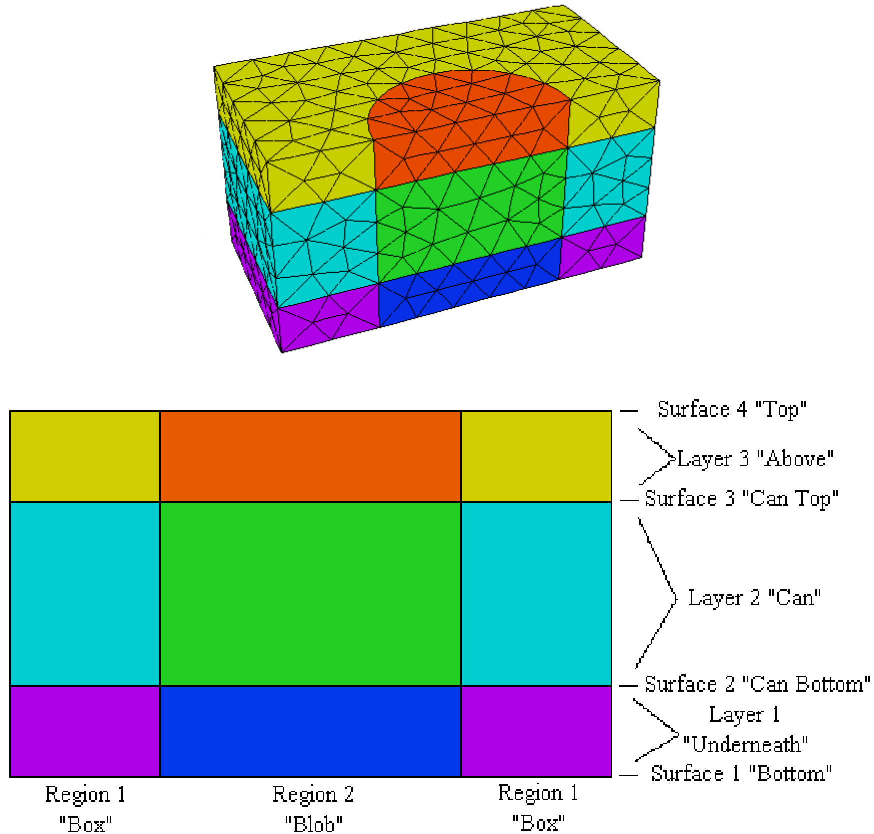
We do this by specifying three layers (and four interface surfaces):

```
EXTRUSION
    SURFACE "Bottom"  z=-1/2
        LAYER "Underneath"
    SURFACE "Can Bottom"  z=-1/4
        LAYER "Can"
    SURFACE "Can Top"  z=1/4
        LAYER "Above"
    SURFACE "Top"       z=1/2
```

We have now divided the 3D figure into six logical compartments: three layers above each of two base regions.

Each of these compartments can be assigned unique material properties, and if necessary, unique boundary conditions.

The cross section now looks like this:



It would seem that we have nine compartments, but recall that region 1 completely surrounds the cylinder, so the left and right parts of region 1 above are joined above and below the plane of the paper. This results in six 3D volumes, denoted by the six colors in the figure.

We stress at this point that it is neither necessary nor correct to try to specify each compartment as a separate entity. You do not need a separate layer and region specification for each material compartment, and repetition of identical regions in the base plane or layers in the extrusion will cause confusion.

The compartment structure is fully specified by the two coordinates REGION and LAYER, and any compartment is identified by the intersection of the REGION in the base plane with the LAYER in the extrusion.

2.6.4 Setting Material Properties by Region and Layer

In our 2D problem, we specified the conductivity of the blob inside the REGION definition for the blob, and that continues to be the technique in 3D.

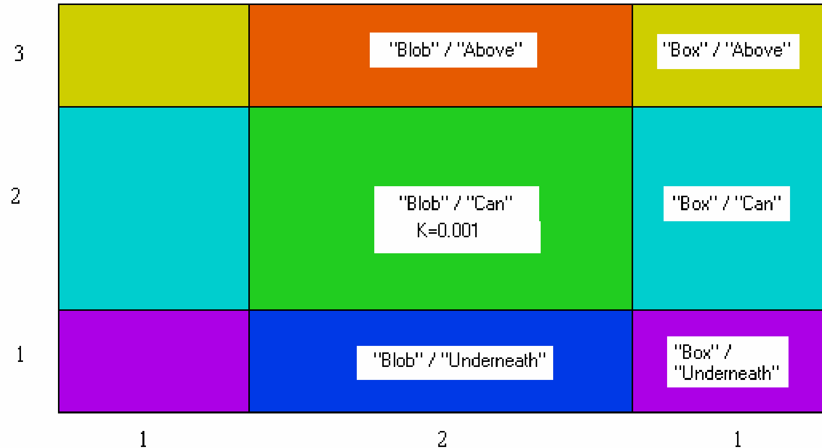
The difference now is that we must also specify the LAYER to which the definition applies. We do this with a LAYER qualification clause:

```
REGION 2 'blob'  { the embedded blob }
```

```
LAYER 'Can'  K = 0.001
START 'ring' (R,0)
    ARC(CENTER=0,0) ANGLE=360
```

Without the LAYER qualification clause, the definition would apply to all layers lying above region 2 of the base plane. Here, the presence of the parameter definition inside a REGION and qualified by a LAYER selects a specific 3D compartment to which the specification applies.

In the following diagram, we have labeled each of the six distinct compartments with a (region,layer) coordinate.



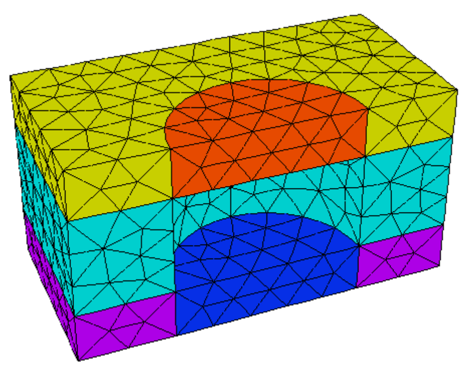
The comprehensive logical structure of parameter redefinitions in the BOUNDARIES section with the location of parameter redefinition specifications in this grid can be described for the general case as follows:

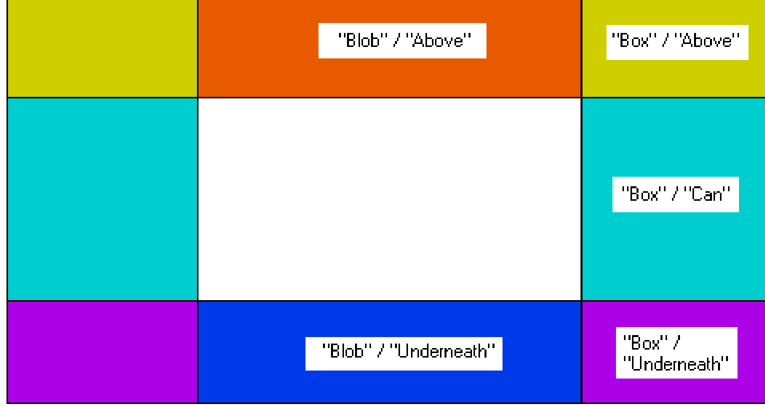
```
BOUNDARIES

    REGION 1
    params(1,all) { parameter redefinitions for all layers of region 1 }
    LAYER 1
        params(1,1){ parameter redefinitions restricted to layer 1 of region 1 }
    LAYER 2
        params(1,2){ parameter redefinitions restricted to layer 2 of region 1 }
    LAYER 3
        params(1,3){ parameter redefinitions restricted to layer 3 of region 1 }
    START(,)  ....TO CLOSE     { trace the perimeter }

    REGION 2
        params(2,all) { parameter redefinitions for all layers of region 2 }
    LAYER 1
        params(2,1)   { parameter redefinitions restricted to layer 1 of region 2 }
    LAYER 2
        params(2,2)   { parameter redefinitions restricted to layer 2 of region 2 }
    LAYER 3
        params(2,3)   { parameter redefinitions restricted to layer 3 of region 2 }
    START(,)  ....TO CLOSE     { trace the perimeter }

    { ... and so forth for all regions }
```

2.6.5 Void Compartments

The reserved word VOID is treated syntactically the same as a parameter redefinition. If this word appears in any of the LAYER-qualified positions above, then that (region,layer) compartment will be excluded from the domain.

```
REGION 2      'blob'   { the embedded blob }
    LAYER 'Can'  VOID
    START 'ring' (R,0)
        ARC(CENTER=0,0) ANGLE=360
```





The example problem "Samples | Usage | 3D_Domains | 3D_Void.pde 431" demonstrates this usage.

2.6.6 Limited Regions

In what we have discussed so far, the region structure specified in the 2D base plane has been propagated unchanged throughout the extrusion dimension. FlexPDE uses the specifier LIMITED REGION to restrict the defined region to a specified set of layers and/or surfaces.

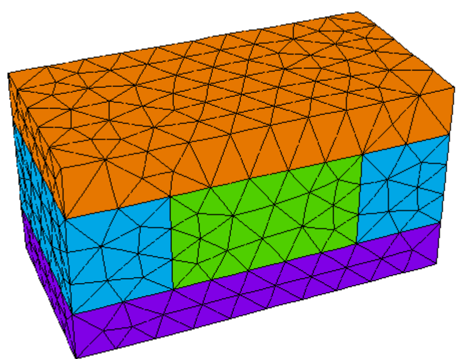
Instead of propagating throughout the extrusion dimension, a LIMITED REGION exists only in the layers

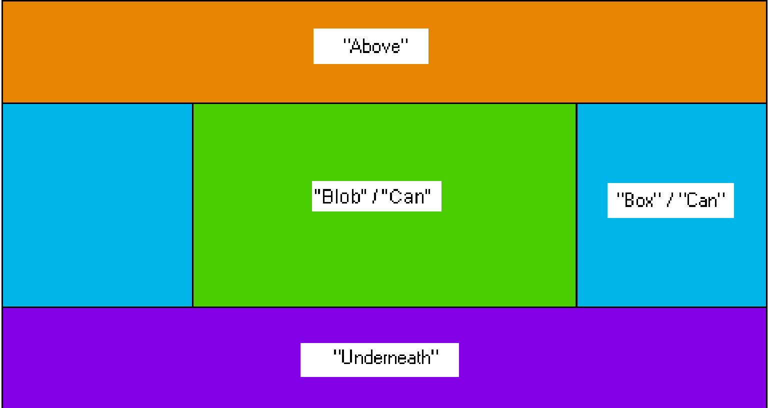
and surfaces explicitly referenced in the declarations within the region. Mention of a layer causes the LIMITED REGION to exist in the specified layer and in its bounding surfaces. Mention of a surface causes the LIMITED REGION to exist in the specified surface.

In our ongoing example problem, we can specify:

```
LIMITED REGION 2      'blob'   { the embedded blob }
    LAYER 'Can'  K = 0.001
    START 'ring' (R,0)
        ARC(CENTER=0,0) ANGLE=360 TO CLOSE
```

In this form, the canister is not propagated through the "Above" and "Underneath" layers:





2.6.7 Specifying Plots on Cut Planes

In two-dimensional problems, the CONTOUR, SURFACE, VECTOR, GRID output forms display data values on the computation plane.

In three dimensions, the same displays are available on any cut plane through the 3D figure. The specification of this cut plane is made by simply appending the equation of a plane to the plot command, qualified by 'ON':

```
PLOTS
    CONTOUR(Phi) ON x=0
```

***Note**: More uses of the ON clause, including plots on extrusion surfaces, will be discussed later* 87.

We can also request plots of the computation grid (and by implication the domain structure) with the **GRID** command:

```
GRID(x,z)  ON y=0
```

This command will draw a picture of the intersection of the plot plane with the tetrahedral mesh structure currently being used by FlexPDE. The plot will be painted with colors representing the distinct material properties present in the cross-section. 3D compartments with identical properties will appear in the same color. The arguments of the GRID plot are the values to be displayed as the abscissa and ordinate positions. Deformed grids can be displayed merely by modifying the arguments.

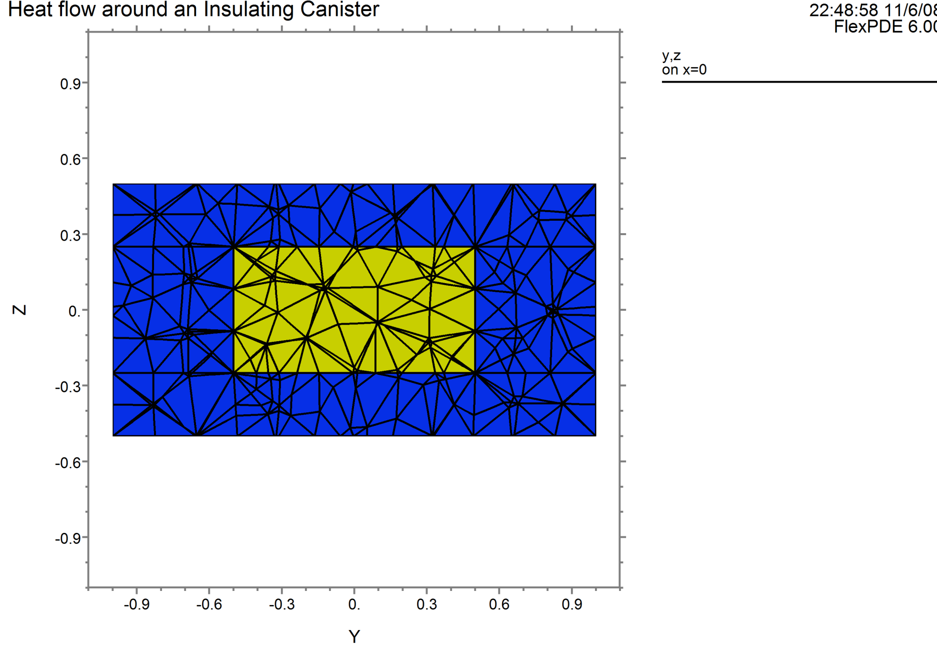
2.6.8 The Complete 3D Canister

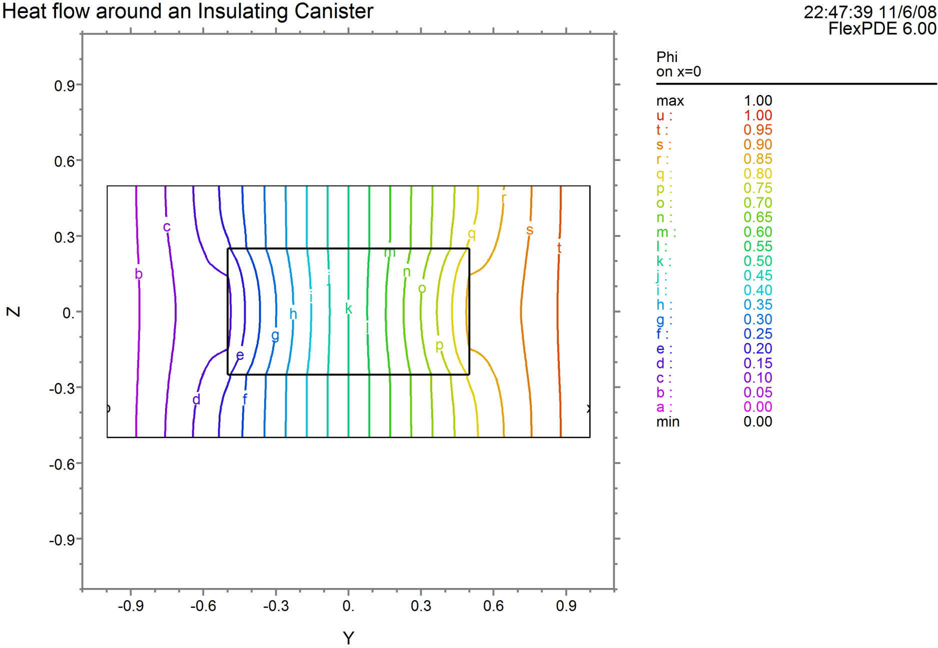
With all the described modifications installed, the full script for the 3D canister problem is as follows:

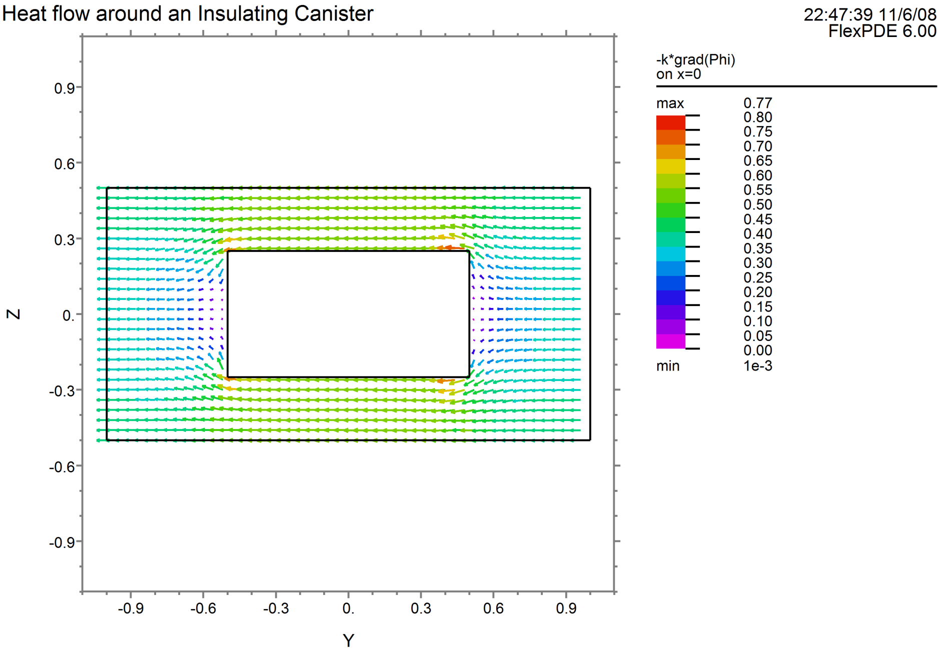
```
TITLE 'Heat flow around an Insulating Canister'
COORDINATES
    Cartesian3
VARIABLES
    Phi             { the temperature }
DEFINITIONS
    K = 1           { default conductivity }
    R = 0.5         { blob radius }
EQUATIONS
    Div(-k*grad(phi)) = 0
EXTRUSION
    SURFACE 'Bottom'  z=-1/2
        LAYER 'underneath'
    SURFACE 'Can Bottom'  z=-1/4
        LAYER 'Can'
    SURFACE 'Can Top'  z=1/4
        LAYER 'above'
    SURFACE 'Top'      z=1/2
BOUNDARIES
    REGION 1 'box'
        START(-1,-1)
            VALUE(Phi)=0LINE TO (1,-1)
            NATURAL(Phi)=0      LINE TO (1,1)
            VALUE(Phi)=1LINE TO (-1,1)
            NATURAL(Phi)=0      LINE TO CLOSE
    LIMITED REGION 2  'blob'  { the embedded blob }
        LAYER 2 k = 0.001             { the canister only }
        START 'ring' (R,0)
            ARC(CENTER=0,0) ANGLE=360 TO CLOSE
PLOTS
    GRID(y,z) ON x=0
    CONTOUR(Phi) ON x=0
    VECTOR(-k*grad(Phi)) ON x=0
    ELEVATION(Phi) FROM (0,-1,0) to (0,1,0) { note 3D coordinates }
END
```

Since we have specified no boundary conditions on the top and bottom extrusion surfaces, they default to zero flux. This is the standard default, for reasons explained in an earlier section.

The first three of the requested PLOTS are:





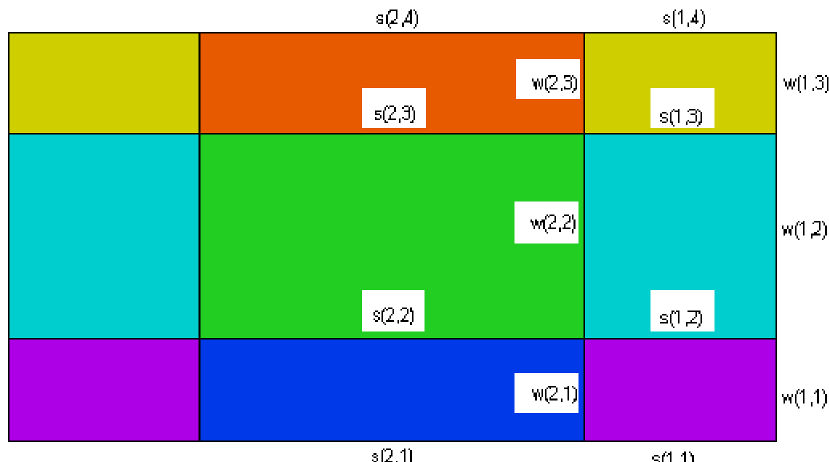


2.6.9 Setting Boundary Conditions in 3D

The specification of boundary conditions in 3D problems is an extension of the techniques used in 2D.

- Boundary condition specifications that in 2D applied to a bounding curve are applied in 3D to the extruded sidewalls generated by that curve.
- The qualifier LAYER number or LAYER "name" may be applied to such a sidewall boundary condition to restrict its application to a specific layer of the sidewall.
- Boundary conditions for extrusion surfaces are constructed as if they were parameter redefinitions over a REGION or over the entire 2D domain. In these cases, the qualifier SURFACE number or SURFACE "name" must precede the boundary condition definition.

In the following figure, we have labeled the various surfaces which can be assigned distinct boundary conditions. Layer interface surfaces have been labeled with an "s", while sidewall surfaces have been labeled with "w". We have shown only a single sidewall intersection in our cross-sectional picture, but in fact each segment of the bounding trace in the base plane can specify a distinct "w" type wall boundary condition.



The comprehensive logical structure of the **BOUNDARIES** section with the locations of the boundary condition specifications in 3D can be diagrammed as follows:

```
BOUNDARIES
SURFACE 1
    s(all, 1)   { BC's on surface 1 over full domain }
SURFACE 2
    s(all, 2)   { BC's on surface 2 over full domain }
{...other surfaces }
REGION 1
    SURFACE 1
        s(1,1) { BC's on surface 1, restricted to region 1 }
    SURFACE 2
        s(1,2) { BC's on surface 2, restricted to region 1 }
    ...
    START(,)    { -- begin the perimeter of region m }
        w(1,..) { BC's on following segments of sidewall of region 1 on all layers }
        LAYER 1
            w(1,1) { BC's on following segments of sidewall of region 1, restricted to layer
            1 }
        LAYER 2
            w(1,2) { BC's on following segments of sidewall of region 1, restricted to layer
            2 }
        ...
        LINE TO ....
        { segments of the base plane boundary with above BC's }
        LAYER 1
            w(1,1) { new BC's on following segments of sidewall of region 1, restricted to
            layer 1 }
        ...
        LINE TO ....
        {  continue the perimeter of region 1 with modified boundary conditions }
        TO CLOSE
REGION 2
    SURFACE 1
        s(2,1) { BC's on surface 1, restricted to region 2 }
```

```
SURFACE 2
    s(2,2) { BC's on surface 2, restricted to region 2 }
...
START(,)    { -- begin the perimeter of region m }
    w(2,..) { BC's on following segments of sidewall of region 2 on all layers }
    LAYER 1
        w(2,1){ BC's on following segments of sidewall of region 2, restricted to layer
        1 }
    LAYER 2
        w(2,2){ BC's on following segments of sidewall of region 2, restricted to layer
        2 }
    ...
    LINE TO ....
    { segments of the base plane boundary with above BC's }
        LAYER 1
            w(2,1) { new BC's on following segments of sidewall of region 2, restricted
            to layer 1 }
    ...
    LINE TO ...
    {  continue the perimeter of region 2 with modified boundary conditions }
    TO CLOSE
```

Remember that, as in 2D, REGIONS appearing later in the script will overlay and cover up portions of earlier regions in the base plane. So the real extent of REGION 1 is that part of the base plane within the perimeter of REGION 1 which is not contained in any later REGION.

For an example of how this works, suppose we want to apply a fixed temperature "Tcan" to the surface of the canister of our previous example. The canister portion of the domain has three surfaces, the bottom, the top, and the sidewall.

The layer dividing SURFACES that define the bottom and top of the canister are named 'Can Bottom' and 'Can Top'. The part we want to assign is that part of the surfaces which lies above region 2 of the base plane. We therefore put a boundary condition statement inside of the region 2 definition, together with a SURFACE qualifier.

The sidewall of the canister is the extrusion of the bounding line of REGION 2, restricted to that part contained in the layer named 'Can'. So we add a boundary condition to the bounding curve of REGION 2, with a LAYER qualifier.

The modified BOUNDARIES section then looks like this:

```
BOUNDARIES
    REGION 1 'box'
        START(-1,-1)
            VALUE(Phi)=0LINE TO (1,-1)
            NATURAL(Phi)=0      LINE TO (1,1)
            VALUE(Phi)=1LINE TO (-1,1)
            NATURAL(Phi)=0      LINE TO CLOSE
    REGION 2      'blob'   { the embedded blob }
        SURFACE 'Can Bottom' VALUE(Phi)=Tcan
        SURFACE 'Can Top' VALUE(Phi)=Tcan
        { parameter redefinition in the 'Can' layer only: }
        LAYER 2 k = 0.001
        START 'ring' (R,0)
            { boundary condition in the 'Can' layer only: }
```

```
        LAYER 'Can' VALUE(Phi)=Tcan
        ARC(CENTER=0,0) ANGLE=360 TO CLOSE
```

2.6.10 Shaped Layer Interfaces

We have stated that the layer interfaces need not be planar. But FlexPDE makes some assumptions about the layer interfaces, which places some restrictions on the possible figures.

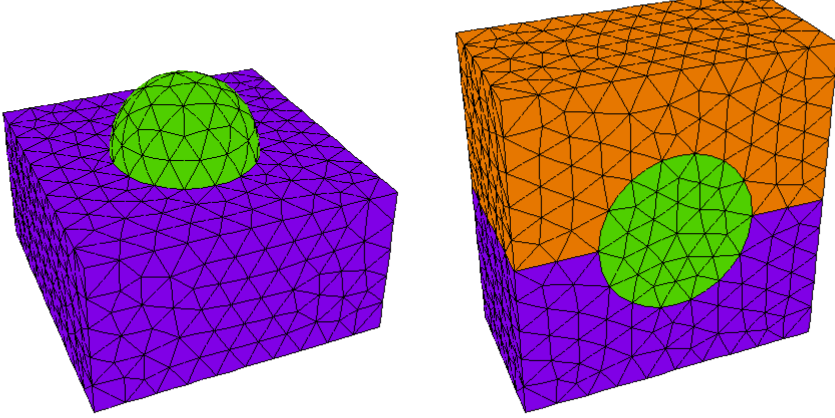
- Figures must maintain an extruded shape, with sidewalls and layer interfaces (the sidewalls cannot grow or shrink)
- Layer interface surfaces must be continuous across region boundaries. If a surface has a vertical jump, it must be divided into layers, with a region interface at the jump boundary and a layer spanning the jump. (Not this: but this:)
- Layer interface surfaces may merge, but may not invert. Use a MAX or MIN function in the surface definition to block inversion.
  Using these rules, we can convert the canister of our example into a sphere by placing spherical caps on the cylinder.
  The equation of a spherical end cap is
  Z = Zcenter + sqrt(R^2 - x^2 - y^2)
  Or,
  Z = Ztop - R + sqrt(R^2 - x^2 - y^2)
- To avoid grazing contact of this new sphere with the top and bottom of our former box, we will extend the extrusion from –1 to 1.
- To avoid arithmetic errors, we will prevent negative arguments of the sqrt.

Our modified script now looks like this:

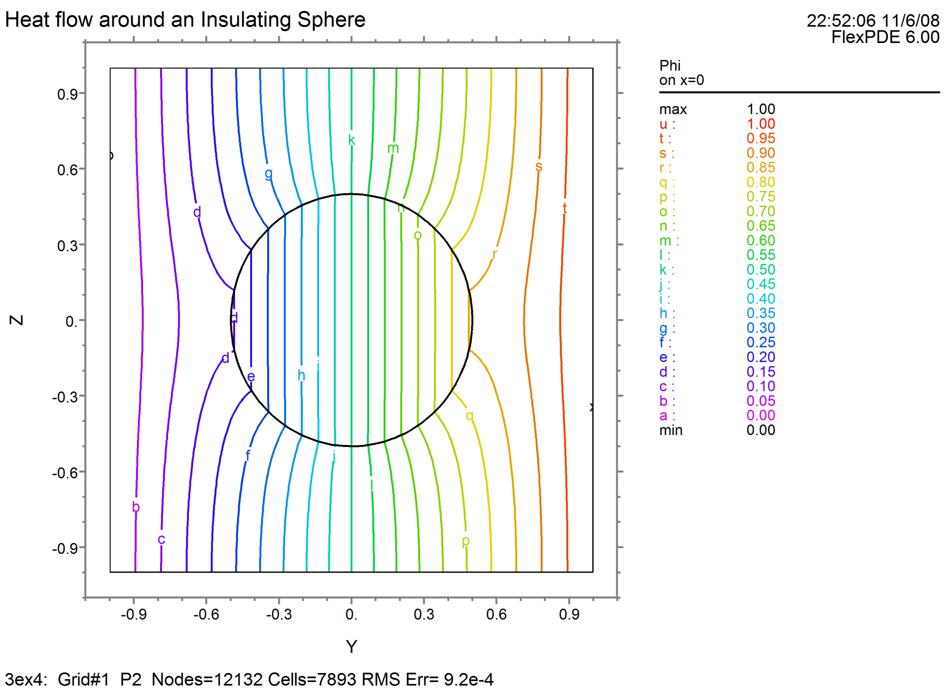
```
TITLE 'Heat flow around an Insulating Sphere'
COORDINATES
    Cartesian3
VARIABLES
    Phi             { the temperature }
DEFINITIONS
    K = 1           { default conductivity }
    R = 0.5         { sphere radius }
    { shape of hemispherical cap: }
    Zsphere = sqrt(max(R^2-x^2-y^2,0))
EQUATIONS
    Div(-k*grad(phi)) = 0
EXTRUSION
    SURFACE 'Bottom'  z=-1
        LAYER 'underneath'
    SURFACE 'Sphere Bottom'  z = -max(Zsphere,0)
        LAYER 'Can'
    SURFACE 'Sphere Top'  z = max(Zsphere,0)
        LAYER 'above'
    SURFACE 'Top'         z=1
BOUNDARIES
    REGION 1 'box'
        START(-1,-1)
            VALUE(Phi)=0LINE TO (1,-1)
            NATURAL(Phi)=0      LINE TO (1,1)
```

```
                VALUE(Phi)=1LINE TO (-1,1)
                NATURAL(Phi)=0      LINE TO CLOSE
        LIMITED REGION 2   'blob'  { the embedded blob }
            LAYER 2 K = 0.001
            START 'ring' (RSphere,0)
                ARC(CENTER=0,0) ANGLE=360
                TO CLOSE
    PLOTS
        GRID(y,z) on x=0
        CONTOUR(Phi) on x=0
        VECTOR(-k*grad(Phi)) on x=0
        ELEVATION(Phi) FROM (0,-1,0) to (0,1,0)
    END
```

Cut-away and cross-section images of the **LAYER x REGION** compartment structure of this layout looks like this:



The contour plot looks like this:



Notice that because of the symmetry of the 3D figure, this plot looks like a rotation of the 2D contour plot in "Putting It All Together".

2.6.11 Surface-Generating Functions

FlexPDE version 6 includes three surface-generation functions (PLANE, CYLINDER and SPHERE) to simplify the construction of 3D domains (See Surface Functions 179 in the Problem Descriptor Reference)

With the SPHERE command, for example, we could modify the Zsphere definition above as

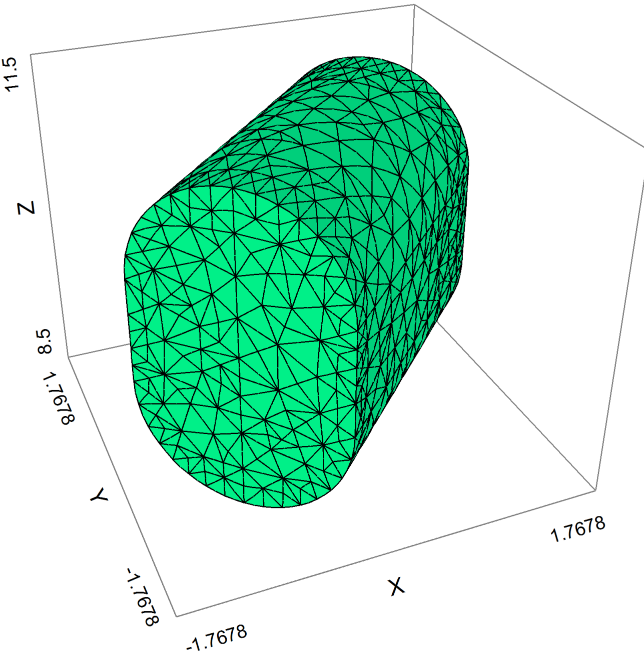
```
{ shape of hemispherical cap: }
Zsphere = SPHERE( (0,0,0), R)
```

We can also build a duct with cylindrical top and bottom surfaces using the following script fragments:

```
DEFINITIONS
    R0 = 1                  { cylinder radius }
    Len = 3                         { cylinder length }
    theta = 45                      { axis direction in degrees }
    c = cos(theta degrees)  { direction cosines of the axis direction }
    s = sin(theta degrees)
    x0 = -(len/2)*c                 { beginning point of the cylinder axis }
    y0 = -(len/2)*s
    zoff = 10               { a z-direction offset for the entire figure }

{ The cylinder function constructs the top surface of a cylinder with azis along z=0.5. The
positive and negative values of this surface will be separated by a distance of one unit at
the diameter. }
```

```
zs = CYLINDER((x0,y0,0.5), (x0+Len*c,y0+Len*s, 0.5), R0)

EXTRUSION
    SURFACE z = zoff-zs     { the bottom half-surface }
    SURFACE z = zoff+zs      { the top half-surface }

BOUNDARIES
    REGION 1
        START  (x0,y0)
            LINE TO (x0+R0*c,y0-R0*s)
            TO (x0+Len*c+R0*c,y0+Len*s-R0*s)
            TO (x0+Len*c-R0*c,y0+Len*s+R0*s)
            TO (x0-R0*c,y0+R0*s)
            TO CLOSE
```

The constructed figure looks like this:



See the example problem "Samples | Usage | 3D_Domains | 3D_Cylspec.pde" for the complete cylinder script.

2.6.12 Integrals in Three Dimensions

In three-dimensional problems, volume integrals may be computed over volume compartments selected by region and layer.

- **Result = VOL_INTEGRAL(<integrand>)**
  Computes the integral of the integrand over the entire domain.

- **Result = VOL_INTEGRAL(<integrand>, <region name>)**
  Computes the integral of the integrand over all layers of the specified region.

- **Result = VOL_INTEGRAL(<integrand>, <layer name>)**

Computes the integral of the integrand over all regions of the specified layer.

- **Result = VOL_INTEGRAL(<integrand>, <region name>, <layer name>)**
  Computes the integral of the integrand over the compartment specified by the region and layer names.

- **Result = VOL_INTEGRAL(<integrand>, <region number>, <layer number>)**
  Computes the integral of the integrand over the compartment specified by the region and layer numbers.

Surface integrals may be computed over selected surfaces. From the classification of various qualifying names, FlexPDE tries to infer what surfaces are implied in a surface integral statement. In the case of non-planar surfaces, integrals are weighted by the actual surface area.

- **Result = SURF_INTEGRAL(<integrand>)**
  Computes the integral of the integrand over the outer bounding surface of the total domain.

- **Result = SURF_INTEGRAL(<integrand>, <surface name> {, <layer_name>})**
  Computes the integral of the integrand over all regions of the named extrusion surface. If the optional <layer_name> appears, it will dictate the layer in which the computation is performed.

- **Result = SURF_INTEGRAL(<integrand>, <surface name>, <region name> {, <layer_name>})**
  Computes the integral of the integrand over the named extrusion surface, restricted to the named region. If the optional <layer_name> appears, it will dictate the layer in which the computation is performed.

- **Result = SURF_INTEGRAL(<integrand>, <region name>, <layer name>)**
  Computes the integral of the integrand over all surfaces of the compartment specified by the region and layer names. Evaluation will be made inside the named compartment.

- **Result = SURF_INTEGRAL(<integrand>, <boundary name> {, <region_name>})**

  Computes the integral of the integrand over all layers of the sidewall generated by the extrusion of the named base-plane curve. If the optional <region name> argument appears, it controls on which side of the surface the integral is evaluated. Portions of the surface that do not adjoin the named layer will not be computed.

- **Result = SURF_INTEGRAL(<integrand>, <boundary name>, <layer name> {, <region_name>})**
  Computes the integral of the integrand over the sidewall generated by the extrusion of the named base-plane curve, restricted to the named layer. If the optional <region name> argument appears, it controls on which side of the surface the integral is evaluated. Portions of the surface that do not adjoin the named layer will not be computed.

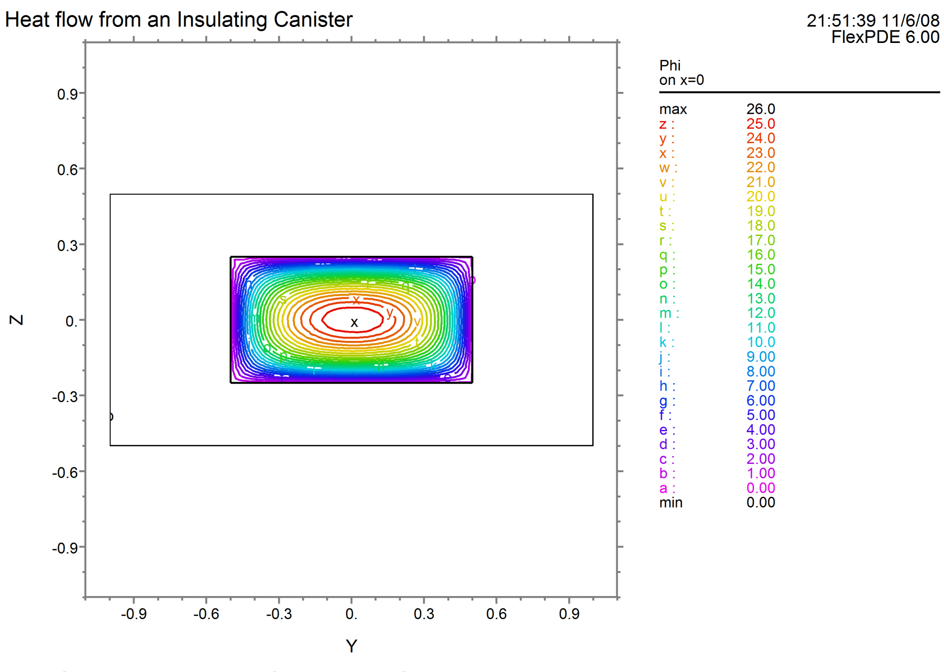
***Note**: The example problem "Samples | Usage | 3D_Integrals.pde* 412*" demonstrates several forms of integral in a three-dimensional problem.*

Let us modify our Canister problem to contain a heat source, and compare the volume integral of the source with the surface integral of the flux, as checks on the accuracy of the solution:

```
TITLE 'Heat flow from an Insulating Canister'
```

```
COORDINATES
    Cartesian3
VARIABLES
    Phi              { the temperature }
DEFINITIONS
    K = 1            { default conductivity }
    R = 0.5          { blob radius }
    S = 0
EQUATIONS
    Div(-k*grad(phi)) = S
EXTRUSION
    SURFACE 'Bottom'  z=-1/2
        LAYER 'underneath'
    SURFACE 'Can Bottom'  z=-1/4
        LAYER 'Can'
    SURFACE 'Can Top'  z=1/4
        LAYER 'above'
    SURFACE 'Top'         z=1/2
BOUNDARIES
    REGION 1 'box'
        START(-1,-1)
            VALUE(Phi)=0LINE TO (1,-1)
            NATURAL(Phi)=0      LINE TO (1,1)
            VALUE(Phi)=1LINE TO (-1,1)
            NATURAL(Phi)=0      LINE TO CLOSE
    REGION 2      'blob'  { option: could be LIMITED }
        LAYER 2 k = 0.001         { the canister only }
        S = 1                     { still the canister }
        START 'ring' (R,0)
            ARC(CENTER=0,0) ANGLE=360 TO CLOSE
PLOTS
    GRID(y,z) on x=0
    CONTOUR(Phi) on x=0
    VECTOR(-k*grad(Phi)) on x=0
    ELEVATION(Phi) FROM (0,-1,0) to (0,1,0)
    SUMMARY
        REPORT(Vol_Integral(S,'blob','can')) AS 'Source Integral'
        REPORT(Surf_Integral(NORMAL(-k*grad(Phi)),'blob','can'))
            AS 'Can Heat Loss'
        REPORT(Surf_Integral(NORMAL(-k*grad(Phi))))
            AS 'Box Heat Loss'
        REPORT(Vol_Integral(S,'blob','can'
        )-Surf_Integral(NORMAL(-k*grad(Phi))))
            AS 'Energy Error'
END
```

The contour plot is as follows:



The summary page shows the integral reports:

SUMMARY

Source Integral= 0.392690
Can Heat Loss= 0.387963
Box Heat Loss= 0.394317
Energy Error= -1.626284e-3

***Note**: The "Integral" reported at the bottom of the contour plot is the default Area_Integral(Phi) reported by the plot processor.*

2.6.13 More Advanced Plot Controls

We have discussed the specification of plots on cut planes in 3D. You can, if you want, apply restrictions to the range of such plots, much like the restrictions of integrals.

You can also specify plots on extrusion SURFACES (layer interface surfaces), even though these surfaces may not be planar.

The basic control mechanism for plots is the ON <thing> statement.

For example, the statement

```
CONTOUR(Phi) ON 'Sphere Top' ON 'Blob'
```

requests a contour plot of the potential Phi on the extrusion surface named 'Sphere Top', restricted to the region 'Blob'.

```
CONTOUR(NORMAL(-K*GRAD(Phi))) ON 'Sphere Top' ON 'Blob' ON 'Can'
```

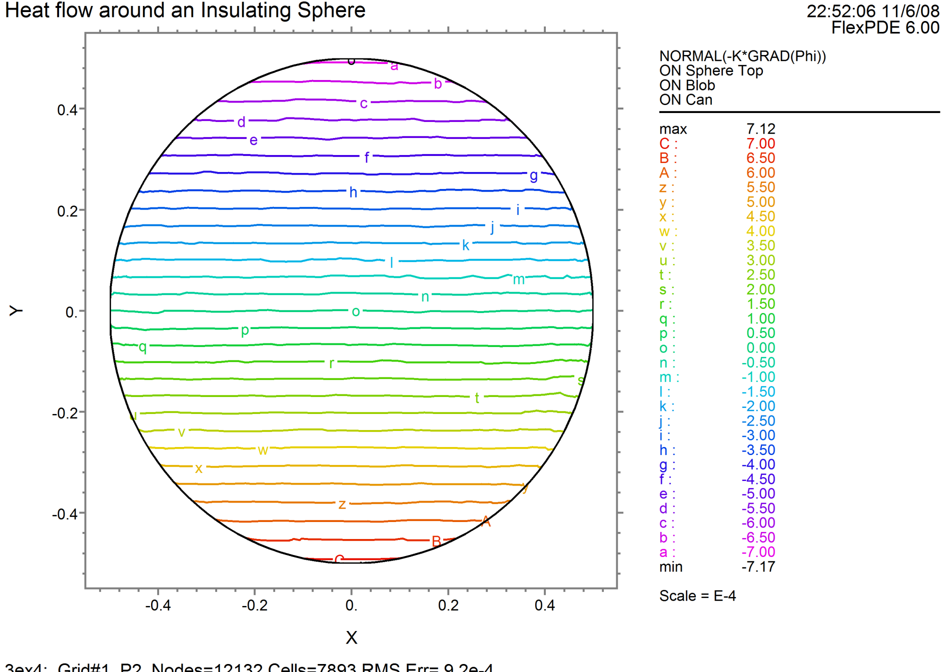
requests a contour plot of the normal component of the heat flux on the top part of the sphere, with evaluation to be made within layer 'Can', i.e., inside the sphere.

- In general, the qualifier ON <name> will request a localization of the plot, depending on the type of object named by <name>.
- The qualifier ON REGION <number> selects a region by number, rather than by name.
- The qualifier ON SURFACE <number> selects a layer interface surface by number, rather than by name.
- The qualifier ON LAYER <number> selects a layer by number, rather than by name.

As an example, let us request a plot of the heat flux on the top of the sphere, as shown above. We will add this command to the PLOTS section, and also request an integral over the same surface, as a cross check. The plot generator will automatically compute the integral over the plot grid. This computation should give the same result as the SURF_INTEGRAL, which uses a quadrature on the computation mesh.

```
CONTOUR(NORMAL(-K*GRAD(Phi))) ON 'Sphere Top' ON 'Blob' ON 'Can'
    REPORT(SURF_INTEGRAL(NORMAL(-k*GRAD(Phi)),'Sphere Top','Blob','Can'))
        AS 'Surface Flux'
```

The result looks like this:



Since in this case the integral is a cancellation of values as large as 7e-4, the reported "Surface Flux" value of -5.2e-8 is well within the default error target of ERRLIM=0.002. The automatically generated plot grid integral, "Surf_Integral", shows greater error at -6.49e-5, due to poorer resolution of integrating the area-weighted function in the plot plane.

2.7 Complex Variables

In previous versions of FlexPDE, it has been possible to treat complex variables and equations by declaring each component as a VARIABLE and writing a real PDE for each complex component.

In version 6, FlexPDE understands complex variables and makes provision for treating them conveniently.

The process starts by declaring a variable to be COMPLEX, and naming its components:

```
VARIABLES
    C = COMPLEX(Cr,Ci)
```

Subsequently, the complex variable **C** can be referenced by name, or its components can be accessed independently by their names.

In the EQUATIONS section, each complex variable can be given an equation, which will be interpreted as dealing with complex quantities. The complex equation will be processed by FlexPDE and reduced to two real component equations, by taking the real and imaginary parts of the resulting complex equation.

For example, the time-harmonic representation of the heat equation can be presented as

```
EQUATIONS
C:   DIV(k*GRAD(C)) - COMPLEX(0,1)*C = 0
```

Alternatively, the individual components can be given real equations:

```
EQUATIONS
Cr:  DIV(k*GRAD(Cr)) + Ci = 0
Ci:  DIV(k*GRAD(Ci)) - Cr = 0
```

In a similar way, boundary conditions may be assigned either to the complex equation or to each component equation individually:

`VALUE(C) = COMPLEX(1,0)`   assigns 1 to the real part and 0 to the imaginary part of C

or

`VALUE(Cr) = 0   NATURAL(Ci) = 0`

Any parameter definition in the DEFINITIONS section may be declared COMPLEX as well:

```
DEFINITIONS
    complexname = COMPLEX(realpart, imaginarypart)
```

FlexPDE recognizes several fundamental complex operators 134:

REAL (complex)	Extracts the real part of the complex number.
IMAG (complex)	Extracts the imaginary part of the complex number.
CARG (complex)	Computes the Argument (or angular component) of the complex number, implemented as CARG(complex(x,y)) = Atan2(y,x).
CONJ (complex)	Returns the complex conjugate of the complex number.
CEXP (complex)	Computes the complex exponential of the complex number, given by CEXP(complex(x,y)) = exp(x+iy) = exp(x)*(cos(y)+i*sin(y)).

COMPLEX quantities can be the arguments of PLOT commands, as well. Occurrence of a complex quantity in a PLOT statement will be interpreted as if the real and imaginary parts had been entered separately in the **PLOT** command.

```
ELEVATION(C) FROM A TO B
```

will produce a plot with two traces, the real and imaginary parts of C.

2.7.1 The Time-Sinusoidal Heat

Suppose we wish to discover the time-dependent behavior of our example Cartesian blob 42 due to the application of a time-sinusoidal applied temperature.

The time-dependent heat equation is `Div(K*Grad(Phi)) = Cp*dt(Phi)`

If we assume that the boundary values and solutions can be represented as

```
Phi(x,y,t) = Cphi(x,y)*exp(i*omega*t)
```

Substituting in the heat equation and dividing out the exponential term, we are left with a complex equation

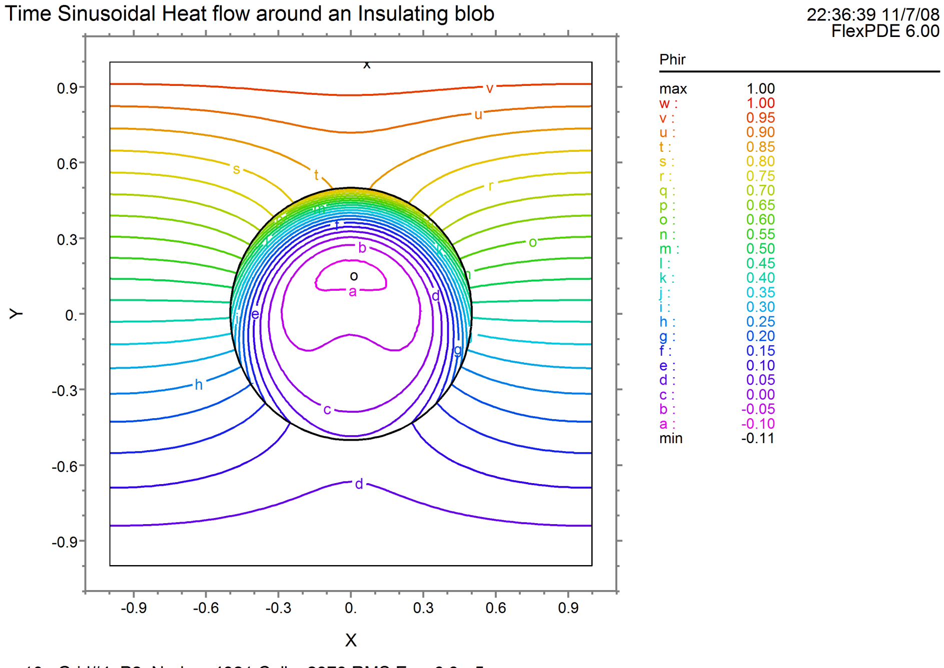
```
Div(K*Grad(Cphi)) - Complex(0,1)*Cphi = 0
```

The time-varying temperature Phi can be recovered from the complex Cphi simply by multiplying by the appropriate time exponential and taking the real part of the result.

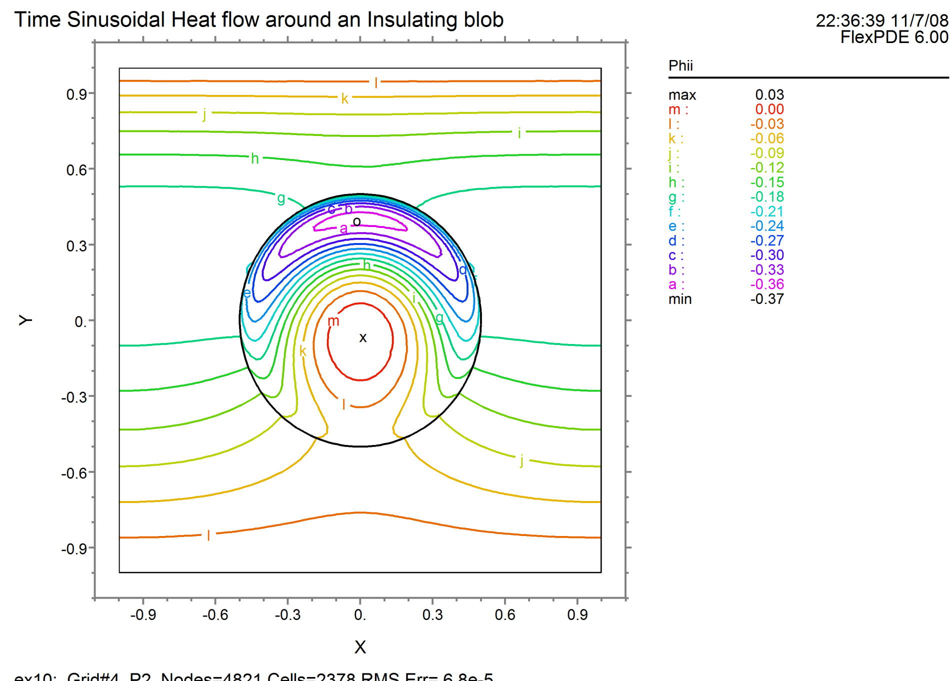
The modified script becomes:

```
TITLE 'Heat flow around an Insulating blob'
VARIABLES
    Phi = Complex(Phir,Phii)      { the complex temperature amplitude }
DEFINITIONS
    K = 1             { default conductivity }
    R = 0.5                   { blob radius }
EQUATIONS
    Phi:  Div(-k*grad(phi)) - Complex(0,1)*Phi = 0
BOUNDARIES
    REGION 1 'box'
        START(-1,-1)
            VALUE(Phi)=Complex(0,0)   LINE TO (1,-1)
            NATURAL(Phi)=Complex(0,0)        LINE TO (1,1)
            VALUE(Phi)=Complex(1,0)   LINE TO (-1,1)
            NATURAL(Phi)=Complex(0,0)        LINE TO CLOSE
    REGION 2  'blob'       { the embedded blob }
        k = 0.01    { change K for prettier pictures }
        START 'ring' (R,0)
            ARC(CENTER=0,0) ANGLE=360 TO CLOSE
PLOTS
    CONTOUR(Phir)  CONTOUR(Phii)
    VECTOR(-k*grad(Phir))
    ELEVATION(Phi) FROM (0,-1) to (0,1)
    ELEVATION(Normal(-k*grad(Phir))) ON 'ring'
END
```

Running this script produces the following results for the real and imaginary components:

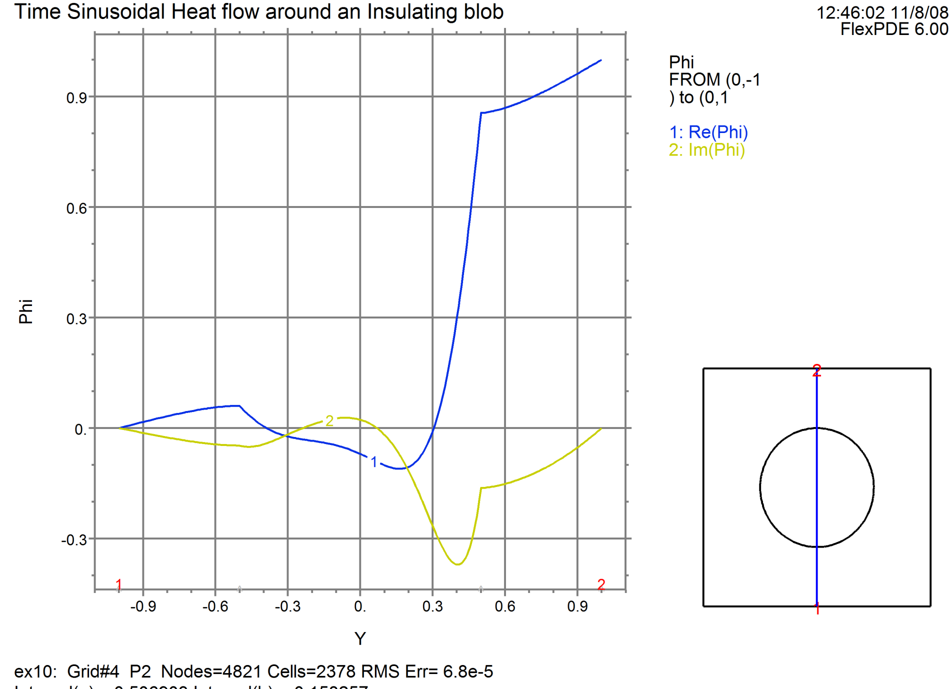


ex10: Grid#4 P2 Nodes=4821 Cells=2378 RMS Err= 6.8e-5
k= 1.000000 INTEGRAL(Phir, 'blob')= 0.068905 Integral= 1.533706



ex10: Grid#4 P2 Nodes=4821 Cells=2378 RMS Err= 6.8e-5
k= 1.000000 INTEGRAL(Phii, 'blob')= -0.115794 Integral= -0.473407

The ELEVATION trace through the center shows:



2.7.2 Interpreting Time-Sinusoidal Results

Knowledge of the real and imaginary parts of the complex amplitude function is not very informative. What we really want to know is what the time behavior of the temperature is. We can investigate this with the help of some other facilities of FlexPDE 6.

We can examine distributions of the reconstructed temperature at selected times using a REPEAT statement

```
PLOTS
    REPEAT tx=0 BY pi/2 TO 2*pi
        SURFACE(Phir*cos(tx)+Phii*sin(tx)) as "Phi at t="+$[4]tx
    ENDREPEAT
```

We can also reconstruct the time history at selected positions using ARRAYS 159. The ARRAY facility allows us to declare arbitrary arrays of values, manipulate them and plot them.

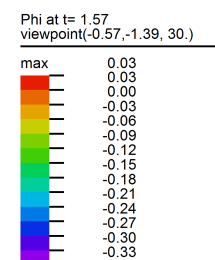
We will declare an array of time points at which we wish to evaluate the temperature, and compute the sin and cos factors at those times. We also define an ARRAY-valued function to return the time history at a point:

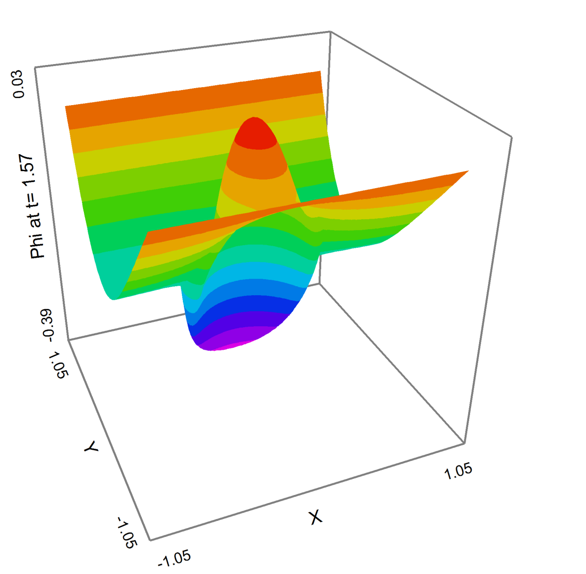
```
DEFINITIONS
    ts = ARRAY (0 BY pi/20 TO 2*pi)    { An array of 40 times }
    fr = cos(ts)     fi = sin(ts)          { the arrays of trigonometric factors }
    poft(px, py) = EVAL(phir,px,py)*fr + EVAL(phii,px,py)*fi
PLOTS
    ELEVATION(poft(0,0), poft(0,0.2), poft(0,0.4), poft(0,0.5)) VS ts
        AS "Histories"
```

Two of the plots produced by the addition of these script lines are:

Time Sinusoidal Heat flow around an Insulating blob

12:46:02 11/8/08
FlexPDE 6.00

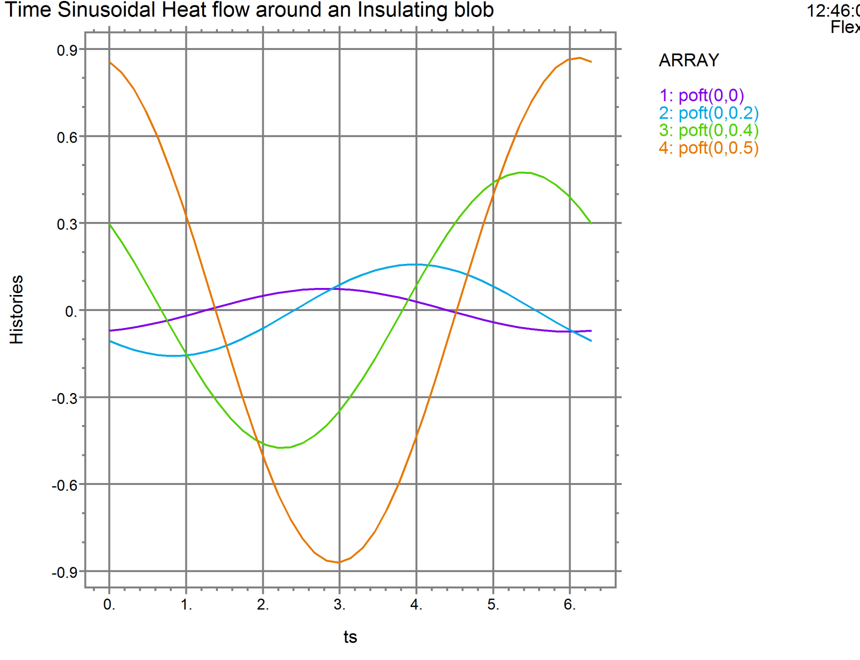




ex10: Grid#4 P2 Nodes=4821 Cells=2378 RMS Err= 6.8e-5
Integral= -0.473407

Time Sinusoidal Heat flow around an Insulating blob

12:46:02 11/8/08
FlexPDE 6.00



ex10: Grid#4 P2 Nodes=4821 Cells=2378 RMS Err= 6.8e-5
Integral(a)= 2.775558e-17 Integral(b)= 1.144917e-16 Integral(c)= -6.938894e-17 Integral(d)= -1.110223e-16

2.8 Vector Variables

FlexPDE version 6 supports the definition of VECTOR variables. Each VECTOR variable is assumed to have a component in each of the three spatial coordinates implied by the COORDINATES 153 section in the script, regardless of the number of dimensions represented in the computation domain.

For example, you can construct a one-dimensional spherical model of three vector directions. Values will be assumed to vary only in the radial direction, but they can have components in the polar and azimuthal directions, as well.

The use of VECTOR variables begins by declaring a variable to be a VECTOR 157, and naming its components:

```
VARIABLES
    V = VECTOR(Vx,Vy,Vz)
```

The component directions are associated by position with the directions implicit in the selected COORDINATES. In YCYLINDER (R,Z,Phi) coordinates, the vector components will be (Vr,Vz,Vphi).

Components may be omitted from the right, in which case the missing components will be assumed to have zero value.

A component may be explicitly declared to have zero value, by writing a 0 in its component position, as in

```
    V = VECTOR(0,0,Vphi)
```

This will construct a one-variable model, in which the variable is the azimuthal vector component.

Subsequently, the vector variable V can be referenced by name, or its components can be accessed independently by their names.

In the EQUATIONS section, each vector variable can be given an equation, which will be interpreted as dealing with vector quantities. The vector equation will be processed by FlexPDE and reduced to as many real component equations as are named in the declaration, by taking the corresponding parts of the resulting vector equation.

For example, the three dimensional cartesian representation of the Navier-Stokes equations can be presented as

```
EQUATIONS
V:  dens*DOT(V,GRAD(V)) + GRAD(p) - visc*DIV(GRAD(V)) = 0
```

Alternatively, the individual components can be given real equations:

```
EQUATIONS
Vx:  dens*(Vx*DX(Vx)+Vy*DY(Vx)+Vz*DZ(Vx)) + DX(p) - visc*DIV(GRAD(Vx)) = 0
Vy:  dens*(Vx*DX(Vy)+Vy*DY(Vy)+Vz*DZ(Vy)) + DY(p) - visc*DIV(GRAD(Vy)) = 0
Vz:  dens*(Vx*DX(Vz)+Vy*DY(Vz)+Vz*DZ(Vz)) + DZ(p) - visc*DIV(GRAD(Vz)) = 0
```

In a similar way, boundary conditions may be assigned either to the complex equation or to each component equation individually:

```
VALUE(V) = VECTOR(1,0,0)
```

or

```
VALUE(Vx) = 0   NATURAL(Vy) = 0
```

Any parameter definition in the DEFINITIONS section may be declared VECTOR as well:

```
DEFINITIONS
    vectorname = VECTOR(xpart,ypart,zpart)
```

VECTOR quantities can be the arguments of PLOT commands, as well. Occurrence of a vector quantity in a PLOT statement will be interpreted as if the component parts had been entered separately in the PLOT command.

```
ELEVATION(V) FROM A TO B
```

will produce a plot with as many traces as are active in the COORDINATES definition.

Examples:
Samples | Usage | Vector_Variables | Vector_Variables.pde 523

2.8.1 Curvilinear Coordinates

An aspect of vector variables in curvilinear coordinates that is sometimes overlooked is that the derivative of a vector is not necessarily the same as the vector of derivatives of the components. This is because in differentiating a vector, the unit vectors in the coordinate space must also be differentiated.

In cylindrical (R,Phi,Z) coordinates, for example, the radial component of the Laplacian of a vector V is

```
DEL2(Vr) - Vr/R^2 - 2*DPHI(Vphi)/R^2
```

The extra 1/R^2 terms have arisen from the differentiation of the unit vectors.

FlexPDE performs the correct expansion of the differential operators in all supported coordinate systems.

2.8.2 Magnetic Vector Potential

Our Cylindrical torus problem 50 can easily be converted to a model of a current-carrying torus inside a box.

The geometry is unchanged, but we now solve for the magnetic vector potential A. We will also move the location slightly outward in radius to avoid the singularity at R=0.

Maxwell's equation for the magnetic field can be expressed in terms of the magnetic vector potential as

```
Curl(Curl(A)/mu) = J
```

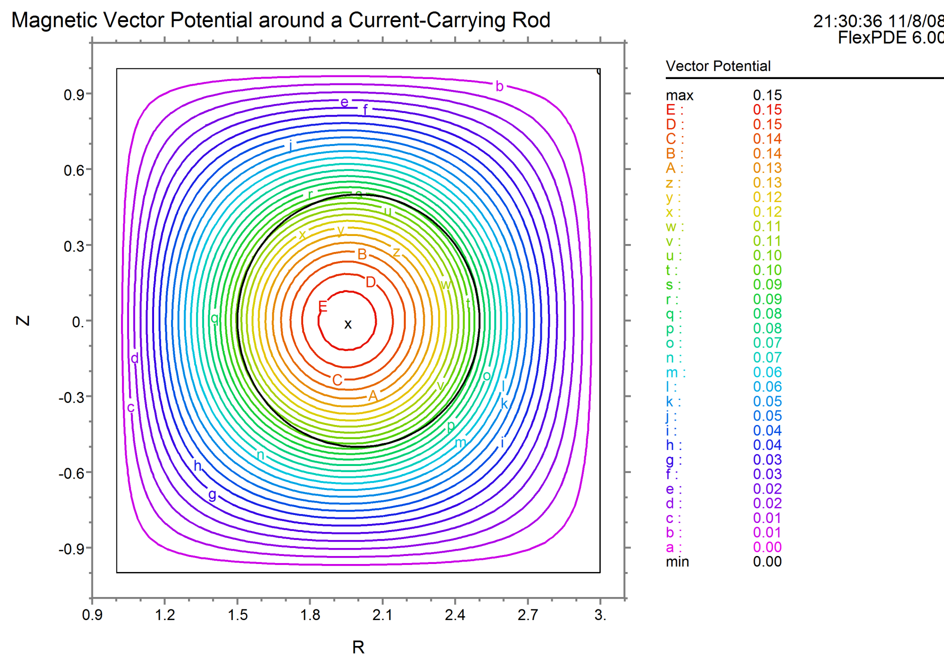
Here J is the vector current density and mu is the magnetic permeability.

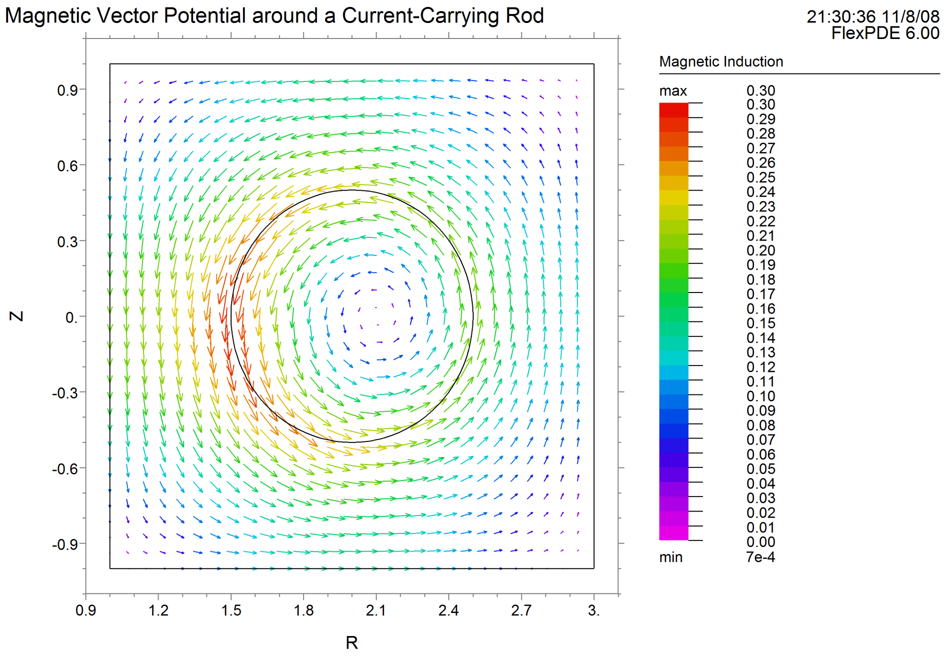
The script becomes

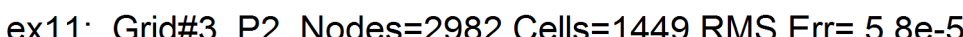
```
TITLE 'Magnetic Field around a Current-Carrying Torus'
COORDINATES YCYLINDER              { implicitly R,Z,Phi }
VARIABLES
    A = VECTOR(0,0,Aphi)
DEFINITIONS
    J = VECTOR(0,0,0)   { default current density }
    mu = 1
    Rad = 0.5                { blob radius (renamed)}
EQUATIONS
```

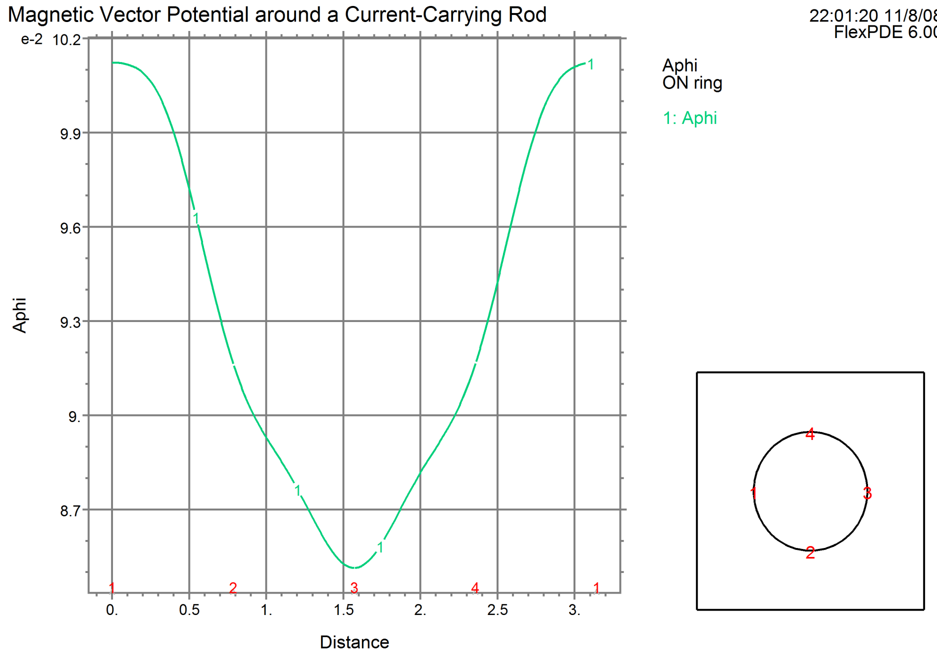
```
    A:  CURL(CURL(A)/mu)) = J
BOUNDARIES
    REGION 1 'box'
        START(1,-1)
            VALUE(A)=VECTOR(0,0,0)
            LINE TO (3,-1) TO (3,1) TO (1,1) TO CLOSE
    REGION 2      'blob'   { the torus }
        J = VECTOR(0,0,1)        { current in the torus }
        START 'ring' (2,Rad)
            ARC(CENTER=2,0) ANGLE=360 TO CLOSE
PLOTS
    CONTOUR(Aphi)  as "Vector Potential"
    VECTOR(CURL(A)) as "Magnetic Induction"
    ELEVATION(Aphi) ON 'ring'
END
```

The resulting plots are:









ex11: Grid#3 P2 Nodes=2982 Cells=1449 RMS Err= 5.8e-5
Surf_Integral= 3.630098

2.9 Variables Inactive in Some Regions

FlexPDE 6 supports the ability to restrict some variables and equations to act only in specified REGIONS. This feature is controlled by declaring variables to be INACTIVE in some regions.

```
VARIABLES
   var1, var2 {,...}
BOUNDARIES
  REGION 1
   INACTIVE(var1, var2 {,...} )
```

In solving the EQUATIONS for these variables, it will be as if the INACTIVE regions had not been included in the domain definition. Boundaries between regions in which the variables are active and those in which they are inactive will be treated as exterior boundaries for these variables. Boundary conditions may be placed on these boundaries as if they were the exterior boundary of the system.

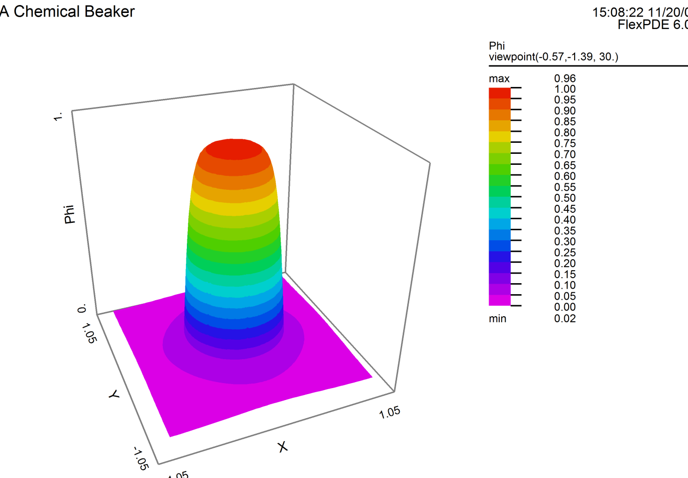
2.9.1 A Chemical Beaker

As an example of Regionally Inactive Variables, let us use the Cartesian Blob 42 test problem, and modify it to represent a chemical beaker immersed in a cooling bath.

Inside the beaker we will place chemicals A and B that react to produce heat. Temperature will be allowed to diffuse throughout the beaker and into the cooling bath, but the chemical reactions will be confined to the beaker. The cooling bath itself is insulated on the outer wall, so no heat escapes the system. The modified script is as follows:

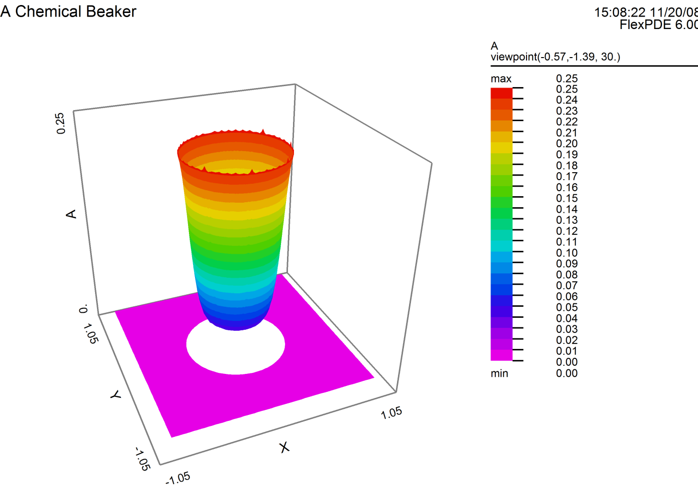
```
TITLE "A Chemical Beaker"
VARIABLES
    Phi(0.1)                    { the temperature }
    A(0.1), B(0.1)          { the chemical components }
DEFINITIONS
    Kphi = 1                    { default thermal conductivity }
    Ka = 0.01      Kb = 0.001   { chemical diffusivities }
    H = 1                       { Heat of reaction }
    Kr = 1+exp(3*Phi)           { temperature dependent reaction rate }
    Cp = 1                  { heat capacity of mixture }
    R = 0.5                     { blob radius }
    A0 = 1     B0 = 2           { initial quantities of chemicals }
INITIAL VALUES
    A = A0
    B = B0
EQUATIONS
   Phi:     Div(kphi*grad(phi)) + H*kr*A*B = Cp*dt(phi)
   A:   Div(ka*grad(A)) - kr*A*B = dt(A)
   B:   Div(kb*grad(B)) - kr*A*B = dt(B)
BOUNDARIES
    REGION 1 'box'
        INACTIVE(A,B)           { inactivate chemicals in the outer region }
        START(-1,-1)
            NATURAL(Phi)=0
            LINE TO (1,-1) TO (1,1) TO (-1,1) TO CLOSE
    REGION 2  'blob'        { the embedded blob }
        kphi = 0.02
        START 'ring' (R,0)
            ARC(CENTER=0,0) ANGLE=360 TO CLOSE
TIME 0 TO 40
PLOTS
```

```
    FOR t=0.1, 0.2, 0.3, 0.5, 1, 2, 5, 10, 20, ENDTIME
        SURFACE(Phi)
        SURFACE(A)
        HISTORY(Phi) AT (0,0) (0,0.4) (0,0.49) (0,0.6)
        REPORT integral(Phi)/integral(1) AS "Average Phi"
        REPORT integral(B,'blob')/integral(1,'blob') as "Residual B"
END
```



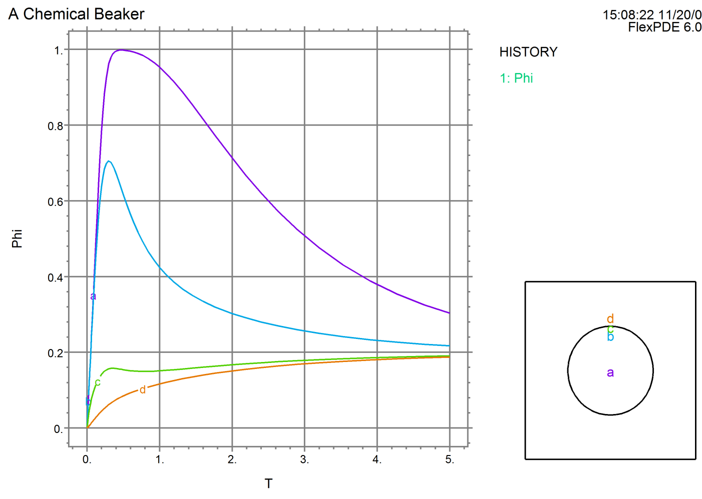
ex12: Cycle=46 Time= 0.3000 dt= 0.0267 P2 Nodes=2297 Cells=1118 RMS Err= 4.5e-4
Integral= 0.704216

This plot of temperature shows diffusion beyond the boundaries of the beaker.



ex12: Cycle=46 Time= 0.3000 dt= 0.0267 P2 Nodes=2297 Cells=1118 RMS Err= 4.5e-4
Integral= 0.079722

This plot of concentration A shows depression in the center where higher temperature increases the reaction rate. No chemical diffuses beyond the beaker boundary.



This plot of temperature history shows an average value of 0.196348. This agrees favorably with the energy conservation value of H*pi*Rad^2/(Cp*Box^2) = 0.196350. The residual quantity of B is correct at 1.0.

2.10 Moving Meshes

FlexPDE supports methods for moving the domain boundaries and computation mesh during the course of a problem run.

The mechanisms for specifying this capability are simple extensions of the existing script language. There are three parts to the definition of a moving mesh:

- Declare a surrogate variable for each coordinate you wish to move:

```
VARIABLES
    Xm = MOVE(x)
```

- Write equations for the surrogate variables:

```
EQUATIONS
    dt(xm) = umesh
```

- Write boundary conditions for the surrogate variables:

```
BOUNDARIES
    START (0,0) VELOCITY(xm) = umesh
```

The specification of ordinary equations is unaffected by the motion of the boundaries or mesh. EQUATIONS are assumed to be presented in Eulerian (Laboratory) form. FlexPDE symbolically applies motion correction terms to the equations. The result of this approach is an Arbitrary Lagrange/Eulerian (ALE) model, in which user has the choice of mesh velocities:

- Locking the mesh velocity to a fluid velocity results in a Lagrangian model. (FlexPDE has no mechanism for reconnecting twisted meshes, so this model is discouraged in cases of violent motion).
- Specifying a mesh velocity different from the fluid velocity preserves mesh integrity while still allowing deformation of the bounding surfaces or following bulk motion of a fluid.
- If no mesh motion is specified, the result is an Eulerian model, which has been the default in previous versions of FlexPDE.

EULERIAN and LAGRANGIAN EQUATIONS

The EQUATIONS section is assumed to present equations in the Eulerian (Laboratory) frame.

The EQUATIONS section can optionally labeled LAGRANGIAN EQUATIONS, in which case FlexPDE will apply no motion corrections to the equations. The user must then provide equations that are appropriate to the moving nodes.

For clarity, the section label EULERIAN EQUATIONS can be used to specify that the equations are appropriate to the laboratory reference frame. This is the default interpretation.

2.10.1 Mesh Balancing

A convenient method for distributing the computation mesh smoothly within a moving domain boundary is simply to diffuse the coordinates or the mesh velocities.

For example, suppose we change our basic example problem to model a sphere of oscillating size Rm=0.5 + 0.25*cos(t).

Diffusing Mesh Coordinates

We define surrogate coordinates for X and Y:

```
VARIABLES
    Phi
    Xm = MOVE(x)
    Ym = MOVE(y)
```

For the EQUATIONS of the mesh coordinates, we will use simple diffusion equations to distribute the positions smoothly in the interior, expecting the actual motions to be driven by boundary conditions:

```
Div(Grad(Xm)) = 0
Div(Grad(Ym)) = 0
```

We can apply the boundary velocities directly to the mesh coordinates on the blob surface using the time derivative of R and geometric rules:

```
VELOCITY(Xm) = -0.25*sin(t)*x/r
VELOCITY(Ym) = -0.25*sin(t)*y/r
```

Diffusing Mesh Velocities

Alternatively, we can define mesh velocity variables as well as the surrogate coordinates for X and Y:

```
VARIABLES
    Phi
    Xm = MOVE(x)
    Ym = MOVE(y)
    Um
    Vm
```

The EQUATIONS for the mesh coordinates are simply the velocity relations:

```
dt(Xm) = Um
dt(Ym) = Vm
```

For the mesh velocities we will use a diffusion equation to distribute the velocities smoothly in the interior:

```
div(grad(Um)) = 0
div(grad(Vm)) = 0
```

The boundary conditions for mesh velocity on the blob are as above:

```
VALUE(Um) = -0.25*sin(t)*x/r
VALUE(Vm) = -0.25*sin(t)*y/r
```

Since the finite element equations applied at the boundary nodes are averages over the cells, we must also apply the hard equivalence of velocity to the mesh coordinates on the blob boundary

```
VELOCITY(Xm) = Um
VELOCITY(Ym) = Vm
```

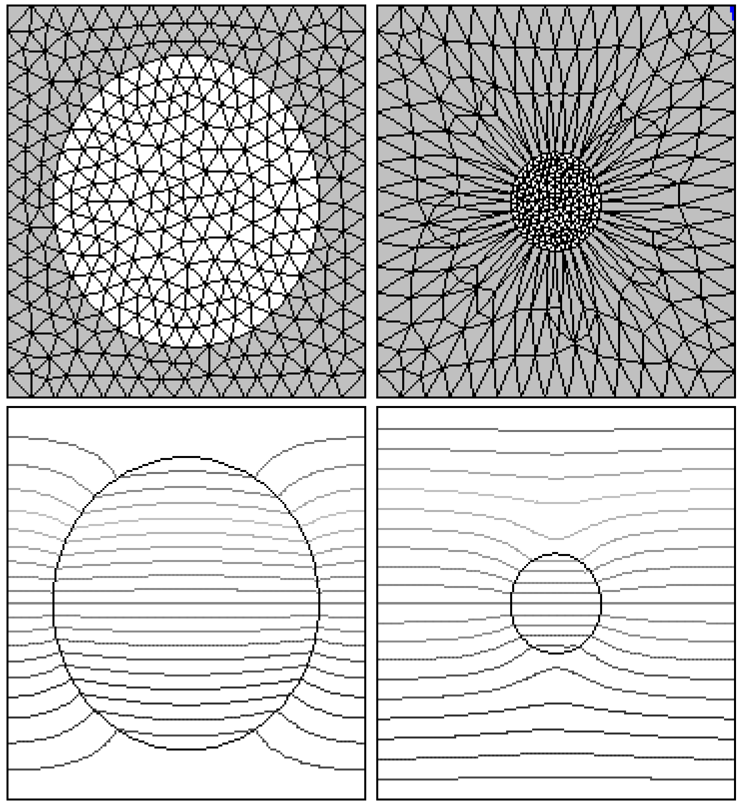
2.10.2 The Pulsating Blob

Using the position balancing form from the preceding paragraph, the modified script for our example problem becomes:

```
TITLE 'Heat flow around an Insulating blob'
VARIABLES
    Phi             { the temperature }
    Xm = MOVE(x)        { surrogate X }
    Ym = MOVE(y)        { surrogate Y }
DEFINITIONS
    K = 1           { default conductivity }
    R0 = 0.75               { initial blob radius }
EQUATIONS
    Phi:    Div(-k*grad(phi)) = 0
    Xm:     div(grad(Xm)) = 0
    Ym:     div(grad(Ym)) = 0
BOUNDARIES
    REGION 1 'box'
        START(-1,-1)
            VALUE(Phi)=0 VELOCITY(Xm)=0 VELOCITY(Ym)=0
            LINE TO (1,-1)
            NATURAL(Phi)=0      LINE TO (1,1)
            VALUE(Phi)=1        LINE TO (-1,1)
            NATURAL(Phi)=0      LINE TO CLOSE
    REGION 2      'blob'  { the embedded blob }
        k = 0.001
        START 'ring' (R,0)
            VELOCITY(Xm) = -0.25*sin(t)*x/r
            VELOCITY(Ym) = -0.25*sin(t)*y/r
            ARC(CENTER=0,0) ANGLE=360 TO CLOSE
TIME 0 TO 2*pi
PLOTS
    FOR T = pi/2 BY pi/2 TO 2*pi
        GRID(x,y)
        CONTOUR(Phi)
        VECTOR(-k*grad(Phi))
```

```
        ELEVATION(Phi) FROM (0,-1) to (0,1)
        ELEVATION(Normal(-k*grad(Phi))) ON 'ring'
    END
```

The extremes of motion of this problem are shown below. See Help system or online documentation for an animation.



The position and velocity forms of this problem can be seen in the following examples:

Samples | Usage | Moving_Mesh | 2D_Position_Blob.pde 495

Samples | Usage | Moving_Mesh | 2D_Velocity_Blob.pde 498

Three-dimensional forms of the problem can be seen in the following examples:

Samples | Usage | Moving_Mesh | 3D_Position_Blob.pde 499

Samples | Usage | Moving_Mesh | 3D_Velocity_Blob.pde 501

2.11 Controlling Mesh Density

There are several mechanisms available for controlling the cell density in the mesh created by FlexPDE.

Implicit Density

The cell density of the created mesh will follow the spacing of points in the bounding segments. A very small segment in the boundary will cause a region of small cells in the vicinity of the segment.

Maximum Density

The global command

```
SELECT NGRID = <number>
```

controls the maximum cell size. The mesh will be generated with approximately **NGRID** cells in the largest dimension, and corresponding size in the smaller dimension, subject to smaller size requirements from other criteria.

Explicit Density Control

Cell density in the initial mesh may be controlled with the parameters MESH_SPACING 173 and MESH_DENSITY 173. MESH_SPACING controls the maximum cell dimension, while MESH_DENSITY is its inverse, controlling the minimum number of cells per unit distance. The mesh generator examines many competing effects controlling cell size, and accepts the smallest of these effects as the size of a cell. The MESH_SPACING and MESH_DENSITY controls therefore have effect only if they are the smallest of the competing influences, and a large spacing request is effectively ignored.

The MESH_SPACING and MESH_DENSITY controls can be used with the syntax of either defined parameters or boundary conditions.

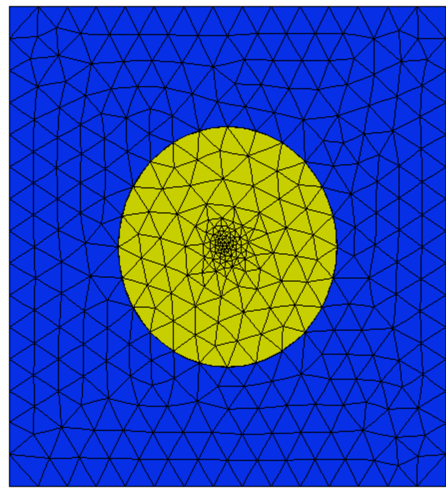
Used as defined parameters, these controls may appear in the DEFINITONS section or may be redefined in subsequent regional redefinition sections. In this use, the controls specify the volume or area mesh density over a region or over the entire domain.

For controlling the cell density along boundary segments, the controls MESH_SPACING and MESH_DENSITY may be used with the syntax of boundary conditions, and may appear wherever a boundary condition statement may appear. In this usage, the controls specify the cell spacing on the boundary curve or surface.

The value assigned to MESH_SPACING or MESH_DENSITY controls may be functions of spatial coordinate. In the example of the chapter "Generating a Mesh" 39, we could write:

```
REGION 2 'blob'          { the embedded 'blob' }
   MESH_DENSITY = 50*EXP(-50*(x^2+y^2))
   START(1/2,0)
      ARC(CENTER=0,0) ANGLE=360
```

This results in the following initial mesh:



See also the example problems

"Samples | Usage | Mesh_Control | Mesh_Spacing.pde" 490

"Samples | Usage | Mesh_Control | Mesh_Density.pde" 490

Adaptive Mesh Refinement

Once the initial mesh is constructed, FlexPDE will continue to estimate the solution error, and will refine the mesh as necessary to meet the target accuracy. In time dependent problems, an adaptive refinement process will also be applied to the initial values of the variables, to refine the mesh where the variables undergo rapid change. Whereas cells created by this adaptive refinement process can later be re-merged, cells created by the initial explicit density controls are permanent, and cannot be un-refined.

> ***Note**: The adaptive refinement process relies on evaluation of the various sources and derivatives at discrete points within the existing mesh. Sources or other effects which are of extremely small extent, such as thin bands or point-like functions, may not be discernible in this discrete model. Any effects of small extent should be brought to the attention of the gridder by explicitly placing gridding features at these locations. Use* REGIONS *or* FEATURES 187 *wherever something interesting is known to take place in the problem.*

See also the FRONT 194 and RESOLVE 195 statements for additional controls.

2.12 Post-processing with FlexPDE

FlexPDE can be used to import both data and mesh structure from a previous run's TRANSFER 169 and perform post-processing without gridding or solving any equations.

This is easily accomplished in a step-wise process:

- Make a copy of the script that generated the exported data. This will ensure that you have the same domain structure in your post-processing script as you did in the exporting script.
- Remove the VARIABLES 154 and EQUATIONS 174 sections. This is how FlexPDE will know not to try and solve any equations.
- Remove any boundary conditions stated in the BOUNDARIES 180 section. Since the variables have been removed, any boundary condition statements will generate a parse error.
- Add the TRANSFERMESH 169 statement in the DEFINITIONS 158 section. This will read in the exported mesh and data.
- Add any new PLOTs 197 that you desire. Now you can easily add plots that were not requested in the initial run, without having to rerun the original script. This is especially useful when you have a computation that takes a lot of time.

***Note**: The domain structure must exactly match that of the exporting problem.*

Examples:

"Samples | Usage | Import-Export | Post_Processing.pde" 481

"Samples | Usage | Import-Export | 3D_Post_Processing.pde" 474

2.13 Exporting Data to Other Applications

FlexPDE supports several mechanisms for exporting data to other applications or visualization software.

The EXPORT Qualifier

The simplest method is to append the modifier "EXPORT" (or "PRINT") to a plot command. In this case, the plot data will be written to a text file in a predefined format suitable for importing to another FlexPDE problem using the TABLE input function. For ELEVATIONS or HISTORIES, the output will consist of a list of the times or X-, Y- or Z- coordinates of the data followed by a list of the data values (see the description of the TABLE input function). For 2D plots, a regular rectangular grid will be constructed, and the data written in TABLE input format.

The FORMAT String

The format of the text file created by the EXPORT modifier may be controlled by the inclusion of the modifier FORMAT "string".
If this modifier appears together with the EXPORT or PRINT modifier, then the file will contain one text line for each data point in the grid. The contents of the line will be exactly that specified by the **string**.

- All characters except "#" will be copied literally into the output line.
- "#" will be interpreted as an escape character, and various options will be selected by the character following the "#":
- #x, #y, #z and #t will print the value of the spatial coordinates or time of the data point;
- #1 through #9 will print the value of the corresponding element of the plot function list;
- #b will write a tab character;
- #r will cause the remainder of the format string to be repeated for each plot function in the plot list;
- #i inside a repeated string will print the value of the current element of the plot function list.

In all cases of FORMATTED export, a header will be written containing descriptive information about the origin of the file. This header will be delimited by "{" and "}". In 2D grids, table points which are outside the problem domain will also be bracketed by "{" and "}" and marked as "exterior". If these commenting forms are unacceptable to the importing application, then the data files must be manually edited before import.

TABLE Output

The TABLE 166 plot command may also be used to generate tabular export. This command is identical to a CONTOUR command with an EXPORT qualifier, except that no graphical output is generated. The FORMAT "string" qualifier may also be used with TABLE output.

Transferring Data to another FlexPDE problem

FlexPDE supports the capability of direct transfer of data defined on the Finite Element mesh. The TRANSFER output function writes the current mesh structure and the requested data values to an ASCII text file. Another FlexPDE problem can read this file with the **TRANSFER** input function. The transferred data will be interpolated on the output mesh with the Finite Element basis of the creating problem. The TRANSFER input mesh need not be the same as the computation mesh, as long as it spans the necessary area.

The data format of the TRANSFER file is similar to the TECPLOT file described below. The TRANSFER file, however, maintains the quadratic or cubic basis of the computation, while the **TECPLOT** format is

converted to linear basis. Since this is an ASCII text file, it can also be used for data transfer to user-written applications. The format of the TRANSFER file is described in the Problem Descriptor Reference chapter "Transfer File Format 170"

Output to Visualization Software

FlexPDE can export solution data to third-party visualization software. Data export is requested by what is syntactically a PLOT command, with the type of plot (such as CONTOUR) replaced by the format selector. Two formats are currently supported, CDF and TECPLOT.

CDF

CDF(arg1 [,arg2,...]) selects output in netCDF version 3 format. CDF stands for "common data format", and is supported by several software products including SlicerDicer (www.visualogic.com). Information about CDF, including a list of software packages supporting it, can be viewed at the website www.unidata.ucar.edu/packages/netcdf .

CDF data are constrained to be on a regular rectangular mesh, but in the case of irregular domains, parts of the rectangle can be absent. This regularity implies some loss of definition of material interfaces, so consider using a ZOOMed domain to resolve small features.

The CDF "plot" statement can be qualified by ZOOM or "ON SURFACE" modifiers, and its density can be controlled by the POINTS modifier. For global control of the grid size, use the statement "SELECT CDFGRID=n", which sets all dimensions to n. The default gridsize is 50.

Any number of arguments can be given, and all will be exported in the same file. The output file is by default "<problem>_<sequence>.cdf", but specific filenames can be selected with the FILE modifier.

TECPLOT

TECPLOT(arg1 [,arg2,...]) selects output in TecPlot format. TecPlot is a visualization package which supports finite element data format, and so preserves the material interfaces as defined in FlexPDE. No ZOOM or POINTS control can be imposed. The full computation mesh is exported, grouped by material number. TecPlot can selectively enable or disable these groups. Any number of arguments can be given, and all will be exported in the same file. The output file is by default "<problem>_<sequence>.dat", but specific filenames can be selected with the FILE modifier.

Information about TecPlot can be viewed at www.amtec.com .

VTK

VTK(arg1 [,arg2,...]) selects output in Visual Tool Kit format. VTK is a freely available library of visualization software, which is beginning to be used as the basis of many visualization packages. The file format can also be read by some visualization packages that are not based on VTK, such as VisIt (www.llnl.gov/visit). The format preserves the mesh structure of the finite element method, and so preserves the material interfaces as defined in FlexPDE. No ZOOM or POINTS control can be imposed. The full computation mesh is exported. Particular characteristics of the visualization system are outside the control of FlexPE. Any number of arguments can be given, and all will be exported in the same file. The output file is by default "<problem>_<sequence>.vtk", but specific filenames can be selected with the FILE modifier.

The VTK format supports quadratic finite element basis directly, but not cubic. To export from cubic-basis

computations, use VTKLIN.

VTKLIN(arg1 [,arg2,...]) produces a VTK format file in which the native cells of the FlexPDE computation have been converted to a set of linear-basis finite element cells.

Information about VTK can be viewed at public.kitware.com/VTK/.

Examples:

Samples | Usage | Import-Export | Export.pde 477

Samples | Usage | Import-Export | Export_Format.pde 477

Samples | Usage | Import-Export | Export_History.pde 478

Samples | Usage | Import-Export | Transfer_Export.pde 485

Samples | Usage | Import-Export | Transfer_Import.pde 485

Samples | Usage | Import-Export | Table.pde 482

Note:

Reference to products from other suppliers does not constitute an endorsement by PDE Solutions Inc.

2.14 Importing Data from Other Applications

The TABLE 165 facility can be used to import data from other applications or from manually created data lists.

Suppose that in our example problem 42 we wish to define a thermal conductivity that varies with temperature (called "Phi" in the example script). We could simply define a temperature-dependent function for the conductivity. But if the dependency is derived from observation, there may be no simple analytic relationship. In this case, we can use a TABLE to describe the dependency.

A table file describing conductivity vs temperature might look like this:

```
{ Conductivity vs temperature }
Phi  6
1 2 10 22 67 101
Data
0.01  0.02  0.05  0.11  0.26  3.8
```

Supposing that we have named this file "conductivity.tbl", our script will simply include the following definition:

```
K = TABLE("conductivity.tbl")
```

Notice that within the table file, the name Phi is declared as the table coordinate. When FlexPDE reads the table file, this name is compared to the names of defined quantities in the script, and the connection is made between the data in the table and the value of "Phi" at any point in the computation where a value of "K" is required.

If the table file had defined the table coordinate as, say, "Temp", we could still use the table in our example by over-riding the table file definition with a new dependency coordinate:

```
K = TABLE("conductivity.tbl", Phi)
```

This statement would cause FlexPDE to ignore the name given in the file itself and associate the table coordinate with the local script value "Phi".

Other forms of TABLE command are available. See the Problem Descriptor Reference chapter "Table Import Definitions" 165 for more information.

2.15 Using ARRAYS and MATRICES

FlexPDE version 6 includes expanded capabilities for using ARRAYS and MATRICES.

ARRAYS 159 and MATRICES 161 differ from other objects in FlexPDE, such as VARIABLES or VECTORS, in that no assumptions are made about associations between the ARRAY or MATRIX and the geometry or mesh structure of the PDE model. ARRAYS and MATRICES are simply lists of numbers which can be defined, manipulated and accessed independently of any domain definition or coordinate geometry. Typically, an ARRAY is created and filled with data using one of the available declaration statements, e.g.,

```
A = array(1,2,3,4,5,6,7,8,9,10)
B = array for x(0 by 0.1 to 10)  : sin(x)+1.1
```

New ARRAYS can be created by performing arithmetic operations on existing arrays:

```
C = exp(A)        { each element of C is the exponential of the corresponding element of
A }
D = C+A  { each element of D is the sum of the corresponding elements of C and A }
E = 100*B{ each element of E is 100 times the corresponding element of B }
```

Elements can be accessed individually by indexing operations:

```
E[12] = B[3]+9
```

ARRAYS can be used in PLOT statements:

```
ELEVATION (D) VS A
```

Similarly, MATRICES can be created and filled with data using one of the available declaration statements, e.g.,

```
M = MATRIX((1,2,3),(4,5,6),(7,8,9))
N = MATRIX FOR x(0 BY 0.1 TO 10)
               FOR y(0 BY 0.1 TO 10) : sin(x)*sin(y)+1.1
```

New ARRAYS or MATRICES can be created by performing *element-by-element* arithmetic operations on existing ARRAYS and MATRICES:

```
P = 1/M   { each element of matrix P is the reciprocal of the corresponding element of M
}
Q = P+M
```

The special operators ****** and **//** are defined for specifying conventional matrix-array arithmetic:

```
R = N**B { R is an ARRAY representing the conventional matrix-array multiplication of B
by N }
S = B//N  { S is the solution of the equation N**S=B (i.e., S is the result of multiplying
B by the inverse of N) }
```

Elements of MATRICES can be accessed individually by indexing operations:

```
U = N[3,9]
```

ARRAYS and MATRICES may also be used to define domain boundaries. See "Boundary Paths" 182 in the Problem Descriptor Reference.

All operations on ARRAYS and MATRICES are checked for compatible sizes, and incompatibilities will be reported as errors.

> ***Note**: You must remember that the FlexPDE script is not a procedural program. Objects in the script describe the dependencies of quantities, and are not "current state" records of values that can be explicitly modified by subsequent redefinition or looping.*

Examples:

See the example folder "Samples | Usage | Arrays+Matrices" 446

2.16 Solving Nonlinear Problems

FlexPDE automatically recognizes when a problem is nonlinear and modifies its strategy accordingly. The solution method used by FlexPDE is a modified Newton-Raphson iteration procedure. This is a "descent" method, which tries to fall down the gradient of an energy functional until minimum energy is achieved (i.e. the gradient of the functional goes to zero). If the functional is nearly quadratic, as it is in simple diffusion problems, then the method converges quadratically (the relative error is squared on each iteration). The default strategy implemented in FlexPDE is frequently sufficient to determine a solution without user intervention. But in cases of strong nonlinearities, it may be necessary for the user to help guide FlexPDE to a valid solution. There are several techniques that can be used to help the solution process.

Time-Dependent Problems

In nonlinear time-dependent problems, the default behavior is to take a single Newton step at each timestep, on the assumption that any nonlinearities will be sensed by the timestep controller, and that timestep adjustments will guarantee an accurate evolution of the system from the given initial conditions. In this mode, the derivatives of the solution with respect to the variables is computed once at the beginning of the timestep, and are not updated.

Steady-State Problems

In the case of nonlinear steady-state problems, the situation is somewhat more complicated. We are not guaranteed that the system will have a unique solution, and even if it does, we are not guaranteed that FlexPDE will be able to find it.

- **Start with a Good Initial Value**
  Providing an initial value which is near the correct solution will aid enormously in finding a solution. Be particularly careful that the initial value matches the boundary conditions. If it does not, serious excursions may be excited in the trial solution, leading to solution difficulties.

- **Use STAGES to Gradually Activate the Nonlinear Terms**
  You can use the staging facility of FlexPDE to gradually increase the strength of the nonlinear terms. Start with a linear or nearly linear system, and allow FlexPDE to find a solution which is consistent with the boundary conditions. Then use this solution as a starting point for a more strongly nonlinear system. By judicious use of staging, you can creep up on a solution to very nasty problems.

- **Use artificial diffusion to stabilize solutions**
  Gibbs phenomena are observed in signal processing when a discontinuous signal is reconstructed from its Fourier components. The characteristic of this phenomenon is ringing, with overshoots and undershoots in the recovered signal. Similar phenomena can be observed in finite element models when a sharp transition is modeled with an insufficient density of mesh cells. In signal processing, the signal can be smoothed by use of a "window function". This is essentially a low-pass filter that removes the high frequency components of the signal. In partial differential equations, the diffusion operator `Div(grad(u))` is a low-pass filter that can be used to smooth oscillations in the solution. See the Technical Note "Smoothing Operators in PDE's 277" for technical discussion of this operator. In brief, you can use a term `eps*Div(Grad(u))` in a PDE to smooth oscillations of spatial extent D by setting `eps=D^2/pi^2` in steady state or `eps=2*D*c/pi` in time dependence (where c is the signal propagation velocity). The term should also be scaled as necessary to provide dimensional consistency with the rest of the terms of the equation. Use of such a term merely limits the spatial frequency components of the solution to those which can be adequately resolved by the finite element mesh.

- **Use `CHANGELIM` to Control Modifications**
  The selector `CHANGELIM` 147 limits the amount by which any nodal value in a problem may be modified on each Newton-Raphson step. As in a one-dimensional Newton iteration, if the trial solution is near a local maximum of the functional, then shooting down the gradient will try to step an enormous distance to the next trial solution. FlexPDE applies a backtracking algorithm to try to find the step size of optimal residual reductions, but it also limits the size of each nodal change to be less than `CHANGELIM` times the average value of the variable. The default value for `CHANGELIM` is `0.5`, but if the initial value (or any intermediate trial solution) is sufficiently far from the true solution, this value may allow wild excursions from which FlexPDE is unable to recover. Try cutting `CHANGELIM` to `0.1`, or in severe cases even `0.01`, to force FlexPDE to creep toward a valid solution. In combination with a reasonable initial value, even `CHANGELIM=0.01` can converge in a surprisingly short time. Since `CHANGELIM` multiplies the RMS average value, not each local value, its effect disappears when a solution is reached, and quadratic final convergence is still achieved.

- **Watch Out for Negative Values**
  FlexPDE uses piecewise polynomials to approximate the solution. In cases of rapid variation of the solution over a single cell, you will almost certainly see severe under-shoot in early stages. If you are assuming that the value of your variable will remain positive, don't. If your equations lose validity in the presence of negative values, perhaps you should recast the equations in terms of the logarithm of the variable. In this case, even though the logarithm may go negative, the implied value of your actual variable will remain positive.

- **Recast the Problem in a Time-Dependent Form**
  Any steady-state problem can be viewed as the infinite-time limit of a time-dependent problem. Rewrite your PDE's to have a time derivative term which will push the value in the direction of decreasing deviation from solution of the steady-state PDE. (A good model to follow is the time-dependent diffusion equation `DIV(K*GRAD(U)) = DT(U)`. A negative value of the divergence indicates a local maximum in the solution, and results in driving the value downward.) In this case, "time" is a fictitious variable analogous to the "iteration count" in the steady-state N-R iteration, but the time-dependent formulation allows the timestep controller to guide the evolution of the solution.

2.17 Using Multiple Processors

FlexPDE version 6 uses multi-threaded computation to support modern multi-core and multi-processor hardware configurations. Only shared-memory multi-processors are supported, not clusters.

Each opened problem runs in its own computation thread, and can use up to eight additional computation threads. A single main thread controls the graphic interface and screen display.

Matrix construction, residual calculations and linear system solvers are all multi-threaded. Mesh generation and plot functions are not, although graphics load is shared between the problem thread and the main graphics thread.

Individual Problem Control
Each individual script can declare the number of worker threads to be used in the computation:

SELECT THREADS = <number>

requests that <number> worker threads be used, in addition to the main graphics thread and the individual problem thread.

Setting the Default
The default number of worker threads can be set by manually editing the configuration file "flexpde6.ini" in the "flexpde6user" folder. This folder resides in the "My Documents" folder under Windows, and the user's "home" folder under Linux and MacOSX. Edit the line:

[THREADS] 1

to reflect the desired default number of worker threads.

Command-Line Control
If you run FlexPDE6 from a command line and include the switch -T<number>, the default thread count will be set to <number>. For example, the command line

flexpde6 -T4 problem

will set the default to 4 threads and load the script file "problem.pde". The selected thread count will be written to the flexpde6.ini file on conclusion of the flexpde6 session.

Speed Effects of Multiple Processors
There are many factors that will influence the timing of a multi-thread run.

- The dominant factor is the memory bandwidth. If the memory cannot keep up with the processor speed, then more threads will run slower due to the overhead of constructing and synchronizing threads and merging data.
- The size of the problem will also affect the speedup, because with a larger problem a smaller proportion of data can be held in cache memory. The memory bandwidth limitation will therefore be greater with a larger problem.
- Graphics *construction* is not multi-threaded in FlexPDE V6. Too many complex plots will therefore drive the performance to 1-thread levels. (Graphic *redraw* is handled in a separate thread).

The following chart shows our experience with speeds in versions 5 and 6. These tests were run on a 4-core AMD Phenom with 667 MHz 128-bit memory. Notice that the Black_Oil problem is significantly faster in version 6, even though it is taking many more timesteps. This timestep count indicates that the timestep control in V6 is more pessimistic than V5. The speedup with V6 1 thread is partly due to the fact that graphic redraws are run in a separate thread in V6 but not in V5.

Notice that in this machine, the memory saturates at 3 threads, so that the fourth thread produces no significant speed improvement (and in fact may be slower).

		Black_Oil.pde		3D_FlowBox.pde
Version	Threads	CPU time	timesteps	CPU time
5	1	14:37	534	8:15
5	2	12:17	540	6:09
6	1	10:21	688	8:06
6	2	6:58	684	4:14
6	3	6:16	696	3:30
6	4	7:13	703	3:22

2.18 Running FlexPDE from the Command Line

When FlexPDE is run from a command line or as a subtask from another application, there are some command-line switches that can be used to control its behavior:

-R	Run the file which is named on the command line. Do not enter edit mode.
-V	View the file which is named on the command line. Do not enter edit mode.
-X	Exit FlexPDE when the problem completes.
-M	Run in "minimized" mode (reduced to an icon).
-Q	Run "quietly". Combines -R -X -M.
-S	Run "silently". -Q with all error reports suppressed.
-T	Set the default thread count. Append the number : -T6 will use six threads.
-L	License FlexPDE. For Internet Key, append A for activate, R for release, then serial number : -LA668668886. For local or network dongle, append D or N : -LD or -LN.

For example, the command line

flexpde6 -R problem

will load and run the script file "problem.pde".

2.19 Running FlexPDE Without A Graphical Interface

Starting in version 6.30, there is a FlexPDE executable that does not use any graphical interface. This is necessary for users to run FlexPDE on systems that do not provide interactive graphics. The executable is suffixed with 'n' (for "no graphics") to distinguish it from the graphical version.

The graphics-less FlexPDE must be run from a command line. For example, the command line

flexpde6n problem

will load and run the script file "problem.pde".

The run can be interrupted by typing 'Q'. The user is then prompted whether to interrupt or not. Type 'Y' to complete the interrupt.

2.20 Getting Help

We're here to help.

Of course, we would rather answer questions about how to use FlexPDE than about how to do the mathematical formulation of your problem.

FlexPDE is applicable to a wide range of problems, and we cannot be experts in all of them.

If you have what appears to be a malfunction of FlexPDE, or if it is doing something you don't understand or seems wrong,

- Send us an Email describing the problem.
- Attach a descriptor file that demonstrates the difficulty, and explain clearly what you think is wrong.
- The more concise you can make your question, the more promptly we will be able to answer.
- Tell us what version of FlexPDE you are using; your problem may have been solved in a later release.

Send your enquiry to support@pdesolutions.com and we will answer as soon as we can, usually within a day or two.

Part III

Problem Descriptor Reference

3 Problem Descriptor Reference

This section presents a detailed description of the components of FlexPDE problem descriptors. No attempt is made here to give tutorial explanations of the use of these components. See Part I Getting Started 2 for user interface information and Part II User Guide 30 for tutorial guidance in the use of FlexPDE.

3.1 Introduction

FlexPDE is a script-driven system. It reads a description of the equations, domain, auxiliary definitions and graphical output requests from a text file referred to as a "problem descriptor" or "script".

The problem descriptor file can be created either with the editor facility in FlexPDE, or with any other ASCII text editor. A word processor can be used only if there is an optional "pure text" output, in which formatting codes have been stripped from the file.

Problem descriptors use an easy to learn natural language originally developed by Robert G. Nelson for use in the PDS2 system at Lawrence Livermore National Lab and later in the PDEase2 system from Macsyma, Inc. The language is also described in Dr. Gunnar Backstrom's book, "Simple Fields of Physics by Finite Element Analysis".

As FlexPDE has evolved, a number of extensions have been added to extend its processing capabilities. The language as currently implemented in FlexPDE is described in this document.

While similar in some ways to a computer programming language, FlexPDE scripting language is more natural, and is oriented to the description of PDE systems. Most intermediate level college students, engineers, and scientists who have had at least an introductory course in partial differential equations can quickly master the language well enough to prepare simple problem descriptor files and begin solving problems of their own devising.

The FlexPDE problem descriptor language can be viewed as a shorthand language for creating Finite Element models. The statements of the descriptor provide the information necessary for FlexPDE to assemble a numerical process to solve the problem.

It is important to understand that the language of FlexPDE problem descriptors is not a *procedural* one. The user describes how the various components of the system relate to one another. He does *not* describe a sequence of steps to be followed in forming the solution, as would be done in a procedural programming language such as C or FORTRAN. Based on the relations between problem elements, FlexPDE decides on the sequence of steps needed in finding the solution.

FlexPDE makes various assumptions about the elements of the problem descriptor.

For example, if a variable is named in the VARIABLES section, it is assumed that:

- the variable is a scalar or vector field which takes on values over the domain of the problem,
- it will be approximated by a finite element interpolation between the nodes of a computation mesh,
- the values of the variable are continuous over the domain, and
- a partial differential equation will be defined describing the behavior of the variable.

If a definition appears in the DEFINITIONS section, it is assumed that the named quantity

- is ancillary to the PDE system,
- may be discontinuous over the domain,
- does not (necessarily) obey any PDE.

In the chapters that follow, we describe in detail the rules for constructing problem descriptors.

3.1.1 Preparing a Descriptor File

Problem descriptor files for use with FlexPDE are most easily prepared and edited using FlexPDE's built-in editor, which uses syntax highlighting to enhance the readability of the user's script. Recognized grammatical keywords are displayed in red, comments in green, and text strings in blue.

To begin a new descriptor file, simply click "File | New Script" from the FlexPDE main menu bar.

To edit an existing descriptor, click "File | Open Script" instead.

A convenient way to create a new descriptor is to start with a copy of an existing descriptor for a similar problem and to modify it to suit the new problem conditions.

FlexPDE's built-in editor is similar to the Windows Notepad editor and produces a pure ASCII text file without any imbedded formatting characters. Descriptor files can also be prepared using any ASCII text editor or any editor capable of exporting a pure ASCII text file. Descriptor files prepared with word processors that embed formatting characters in the text will cause FlexPDE to report parsing errors.

3.1.2 File Names and Extensions

A problem descriptor file can have any name which is consistent with the host operating system. Even though permitted by some operating systems, names with imbedded blank characters should be avoided. It is best to choose a name that is descriptive of the problem.

Problem descriptor files must have the extension '.pde'. When saving a file using the built-in editor, FlexPDE will automatically add the extension '.pde'. When using a separate or off-line editor, be sure to give the file a '.pde' extension instead of the default extension.

Windows operating systems by default hide the file name extension. FlexPDE script files can still be recognized by the icon. Alternatively, Windows can be configured to display file extensions.

See also "FlexPDE Working Files". 3

3.1.3 Problem Descriptor Structure

Problem descriptors organize a problem by breaking it into sections of related items.

Each section is headed by a proper name followed by one or more statements which define the problem.

The permitted section names are:

TITLE	- defines the problem title
SELECT	- sets various options and controls
COORDINATES	- defines the coordinate system
VARIABLES	- names the problem variables
DEFINITIONS	- defines ancillary quantities and parameters
INTIAL VALUES	- sets initial values of variables

EQUATIONS	- defines the partial differential equation system
CONSTRAINTS	- defines optional integral constraints
EXTRUSION	- extends the domain to three dimensions
BOUNDARIES	- describes the 2D or projected 3D domain
RESOLVE	- optionally supplements mesh refinement control
FRONT	- optionally supplements mesh refinement control for advancing fronts
TIME	- defines the time domain
MONITORS	- selects interim graphic display
PLOTS	- selects final graphic display
HISTORIES	- selects time-summary displays
END	- identifies the end of the descriptor

The number of sections used in a particular problem descriptor can vary, subject only to the requirement that all files must contain a BOUNDARIES section and an END section.

While some flexibility exists in the placement of these sections, it is suggested that the user adhere to the ordering described above.

DEFINITIONS and SELECT can appear more than once.

Because descriptors are dynamically processed from top to bottom, they cannot contain forward references. Definitions may refer to variables and other defined names, provided these variables and names have been defined in a preceding section or previously in the same section.

3.1.4 Problem Descriptor Format

While not strictly required, we suggest use of the following indentation pattern for all problem descriptors:

```
section 1
      statement
section 2
      statement 1
      statement 2
            *
            *
section 3
      statement 1
      statement 2
            *
            *
```

This format is easy for both the person preparing the file and for others to read and understand.

3.1.5 Case Sensitivity

With the exception of quoted character strings, which are reproduced exactly as they appear in a problem descriptor, words, characters and other text items used in problem descriptors are NOT case sensitive.

Upper case letters and lower case letters are equivalent.

The text items variables, VARIABLES, Variables and mixed case text like VaRiAbles are all equivalent.

Judicious use of capitalization can improve the readability of the script.

3.1.6 "Include" Files

FlexPDE supports the C-language mechanism of including external files in the problem descriptor. The statement

#INCLUDE "filename"

will cause the named file to be included bodily in the descriptor in place of the #INCLUDE "filename" statement.

If the file does not reside in the same folder as the descriptor, the full path to the file must be given.

An include statement can be placed anywhere in the descriptor, but for readability it should be placed on its own line.

This facility can be used to insert common definition groups in several descriptors.

> ***Note**: Although FlexPDE is not case sensitive, the operating system which is being asked for the included file may be case sensitive. The quoted file name must conform to the usage of the operating system.*

3.1.7 A Simple Example

As a preview example to give the flavor of a FlexPDE descriptor file, we will construct a model of heatflow on a square domain.

The heatflow equation is

```
div(K*grad(Temp)) + Source = 0
```

If K is constant and Source = 4*K, the heat equation will be satisfied by the function

```
Temp = Const - x^2 - y^2 .
```

We define a square region of material of conductivity K = 1, with a uniform heat source of 4 heat units per unit area.

We further specify the boundary value

```
Temp = 1 - x^2 - y^2
```

Since we know the analytic solution, we can compare the accuracy of the FlexPDE solution.

The text of the descriptor is as follows:

```
{ ******************************************************************
  SIMPLE.PDE

   This sample demonstrates the simplest application of FlexPDE to
   heatflow problems.
****************************************************************** }

 TITLE "Simple Heatflow"

 VARIABLES
   temp                { Identify "Temp" as the system variable }

 DEFINITIONS
   k = 1               { declare and define the conductivity }
   source = 4          { declare and define the source }
   texact = 1-x^2-y^2  { exact solution for reference }

 INITIAL VALUES
   temp = 0     { unimportant in linear steady-state problems,
                but necessary for time-dependent or nonlinear
                systems }

 EQUATIONS      { define the heatflow equation :}
   div(k*grad(temp)) + source = 0


 BOUNDARIES   { define the problem domain: }
   REGION 1            { ... only one region }
     { specify Dirichlet boundary at exact solution: }
     VALUE(temp)=texact
     START(-1,-1)      { specify the starting point }
     LINE TO (1,-1)    { walk the boundary }
       TO (1,1)
       TO (-1,1)
       TO CLOSE        { bring boundary back to starting point }

 MONITORS
   CONTOUR(temp)       { show the Temperature during solution }

 PLOTS          { write these plots to disk at completion: }

   CONTOUR(temp)       { show the solution }
   SURFACE(temp)       { show a surface plot as well }
                       { display the solution error :}
   CONTOUR(temp-texact) AS "Error"
                       { show a vector flow plot: }
   VECTOR(-dx(temp),-dy(temp)) AS "Heat Flow"

 END                   { end of descriptor file }
```

3.2 The Elements of a Descriptor

The problem descriptors or 'scripts' which describe the characteristics of a problem to FlexPDE are made up of a number of basic elements, such as names and symbols, reserved words, numeric constants, etc. These elements are described in the sections that follow.

3.2.1 Comments

Problem descriptors can be annotated by adding comments.

Multi-line comments can be placed anywhere in the file. Multi-line comments are formed by enclosing the desired comments in either curly brackets **{** and **}** or the paired symbols **/*** and ***/**. Comments can be nested, but comments that begin with a curly bracket must end with a curly bracket and comments that begin with '/*' must end with '*/'.

Example:

```
{ this is a comment
so is this.
}
```

End-of-line comments are introduced by the exclamation mark **!**. End-of-line comments extend from the **!** to the end of the line on which they occur. Placing the line comment symbol **!** at the beginning of a line effectively removes the whole line from the active portion of the problem descriptor, in a manner similar to 'rem' at the beginning of a line in a DOS batch file or "//" in C++.

Example:

```
! this is a comment
this is not
```

Comments can be used liberally during script development to temporarily remove lines from a problem descriptor. This aids in localizing errors or focusing on specific aspects of a problem.

3.2.2 Reserved Words and Symbols

FlexPDE assigns specific meanings and uses to a number of predefined 'reserved' words and symbols in descriptors.
Except when they are included as part of a comment or a literal string, these words may only be used for their assigned purpose.

The following parser keywords are highlighted by the FlexPDE editor:

ACUMESH	**ALIAS**	**ALIGN_MESH**
AND	**ANGLE**	**ANTIPERIODIC**
ARC	**ARRAY**	**AS**
AT		
BATCH	**BEVEL**	**BLOCK**
BOUNDARIES	**BY**	
CDF	**CENTER**	**CLOSE**
COMPLEX	**CONST**	**CONSTRAINTS**
CONTACT	**CONTOUR**	**COORDINATES**

CYLINDER

DEBUG **DEFINITIONS** **DEGREES**
DELAY **DELTAT** **DIR**
DIRECTION

ELEVATION **ELSE** **END**
ENDLABEL **ENDREPEAT** **EQUATION**
EQUATIONS **ERRWEIGHT** **EULERIAN**
EVAL **EXCLUDE** **EXPORT**
EXTRUSION

FEATURE **FILE** **FILLET**
FINALLY **FINISH** **FIXED**
FOR **FORMAT** **FRAME**
FREEZE **FROM** **FRONT**

GLOBAL **GLOBALMAX** **GLOBALMAX_X**
GLOBALMAX_Y **GLOBALMAX_Z** **GLOBALMIN**
GLOBALMIN_X **GLOBALMIN_Y** **GLOBALMIN_Z**
GRID

HALT **HISTORIES** **HISTORY**

IF **INACTIVE** **INITIAL**

JUMP

LABEL **LAGRANGIAN** **LAMBDA**
LAYER **LAYERED** **LAYERS**
LEVELS **LIMIT** **LIMITED**
LINE **LIST** **LOAD**

MAP **MATRIX** **MERGE**
MESH_DENSITY **MESH_SPACING** **MODE**
MONITORS **MOVE**

NATURAL **NEUMANN** **NOBC**
NODE **NOT**

OFF **ON** **OR**

PERIODIC **PLANE** **PLOTS**
POINT **POINT_LOAD** **POINT_NATURAL**
POINT_VALUE **POINT_VELOCITY** **POINTS**
PRINT **PRINTONLY**

RADIANS **RADIUS** **RANGE**
REGION **REGIONS** **REPEAT**
REPORT **RESOLVE** **ROTATE**

SCALAR **SELECT** **SIMPLEX**
SIZEOF **SMOOTH** **SPHERE**

SPLINE **SPLINETABLE** **SPLINETABLEDEF**
STAGE **STAGED** **START**
SUM **SUMMARY** **SURFACE**

TABLE **TABLEDEF** **TABULATE**
TECPLOT **TENSOR** **THEN**
TIME **TITLE** **THRESHOLD**
TO **TRANSFER** **TRANSFERMESH**
TRANSFERMESHTIME

UNORMAL **USE**

VAL **VALUE** **VALUES**
VARIABLES **VECTOR** **VELOCITY**
VERSUS **VIEWANGLE** **VIEWPOINT**
VOID **VOLJ** **VS**
VTK **VTKLIN**

WINDOW

ZOOM

The following names of built-in functions are not recognized by the FlexPDE editor's syntax highlighter, but may be used only for their assigned purpose:

ABS **AINTEGRAL** **ARCCOS**
ARCSIN **ARCTAN** **AREA_INTEGRAL**
ATAN2

BESSI **BESSJ** **BESSK**
BESSY **BINTEGRAL**

CARG **CEXP** **CLOG**
CONJ **COS** **COSH**
CROSS **CURL**

DEL2 **DIFF** **DIV**
DOT

ENDTIME **ERF** **ERFC**
EXPINT **EXP**

FEATURE_INDUCTION **FIT**

GAMMAF **GLOBALMAX** **GLOBALMAX_X**
GLOBALMAX_Y **GLOBALMAX_Z** **GLOBALMIN**
GLOBALMIN_X **GLOBALMIN_Y** **GLOBALMIN_Z**
GRAD

IMAG **INTEGRAL** **INTEGRATE**

JACOBIAN

LINE_INTEGRAL	**LN**	**LOG10**
LUMP		
MAGNITUDE	**MAX**	**MIN**
MOD		
NORMAL		
PARTS	**PASSIVE**	**PI**
RAMP	**RANDOM**	**REAL**
SAVE	**SIGN**	**SIN**
SINH	**SINTEGRAL**	**SQRT**
SURF_INTEGRAL	**SWAGE**	
TAN	**TANGENTIAL**	**TANH**
TIME_INTEGRAL	**TIME_MAX**	**TIME_MIN**
TIMEMAX	**TIMEMAX_T**	**TIMEMIN**
TIMEMIN_T	**TINTEGRAL**	
UPULSE	**UPWIND**	**URAMP**
USTEP		
VINTEGRAL	**VOL_INTEGRAL**	
XBOUNDARY	**XCOMP**	**XXCOMP**
XYCOMP	**XZCOMP**	**YBOUNDARY**
YCOMP	**YXCOMP**	**YYCOMP**
YZCOMP	**ZBOUNDARY**	**ZCOMP**
ZXCOMP	**ZYCOMP**	**ZZCOMP**

3.2.3 Separators

White Space

Spaces, tabs, and new lines, frequently referred to as "white space", are treated as separators and may be used freely in problem descriptors to increase readability. Multiple white spaces are treated by FlexPDE as a single white space.

Commas

Commas are used to separate items in a list, and should be used only where explicitly required by the descriptor syntax.

Semicolons

Semicolons are not significant in the FlexPDE grammar. They are treated as equivalent to commas.

3.2.4 Literal Strings

Literal strings are used in problem descriptors to provide optional user defined labels, which will appear on softcopy and hardcopy outputs.

The label that results from a literal string is reproduced on the output exactly (including case) as entered in the corresponding literal string.

Literal strings are formed by enclosing the desired label in either single or double quote marks . Literal strings that begin with a double quote mark must end in a double quote mark, and literal strings that begin with a single quote mark must end in a single quote mark.

A literal string may consist of any combination of alphanumeric characters, separators, reserved words, and/or symbols including quote marks, provided only that strings that begin with a double quote mark may contain only single quote marks and strings that begin with a single quote mark may contain only double quote marks.

Example:

```
TITLE "This is a literal 'string' used as a problem title"
```

3.2.5 Numeric Constants

Integers

Integers must be of the form XXXXXX where X is any decimal digit from 0 to 9. Integer constants can contain up to 9 digits.

Decimal Numbers

Decimal numbers must be of the form XXXXX.XXX where X is any decimal digit from 0 to 9 and '.' is the decimal separator. Decimal numbers must not include commas ','. Using the European convention of a comma ',' as a decimal separator will result in an error. Commas are reserved as item separators. Decimal numbers may include zero to nine digits to the left of the decimal separator and up to a total of 308 digits total. FlexPDE considers only the first fifteen digits as significant.

Engineering Notation Numbers

Engineering notation numbers must be of the form XXXXXEsYYY where X is any digit from 0 to 9 or the decimal separator '.', Y is any digit from 0 to 9, E is the exponent separator, and s is an optional sign operator. Engineering notation numbers must not include commas ','. Using the European convention of a comma ',' as a decimal separator will result in an error. Commas are reserved as item separators. The number to the left of the exponent separator is treated as a decimal number and the number to the right of the exponent separator is treated as an integer and may not contain a decimal separator or more than 3 digits. The range of permitted engineering notation numbers is 1e-307 to 1e308.

3.2.6 Built-in Functions

Functions and Arguments

All function references must include at least one argument. Arguments can be either numerical constants or expressions that evaluate to numerical values. The following functions are supported in problem descriptors:

3.2.6.1 Analytic Functions

The following analytic functions are supported by FlexPDE:

Function	Comments
ABS(x)	Absolute value
ARCCOS(x)	Inverse cosine (returns radians)
ARCSIN(x)	Inverse sine (returns radians)
ARCTAN(x)	Inverse tangent (returns radians)
ATAN2(y,x)	Arctan(y/x) with numerically safe implementation
BESSI(order,x)	Modified Bessel function I for real x
BESSJ(order,x)	Bessel Function J
BESSK(order,x)	Modified Bessel function K for real x
BESSY(order,x)	Bessel Function Y
COS(x)	cosine of x (angle in radians)*
COSH(x)	Hyperbolic cosine
ERF(x)	Error Function
ERFC(x)	Complementary Error Function
EXP(x)	Exponential function
EXPINT(x)	Exponential Integral Ei(x) for real x>0 **
EXPINT(n,x)	Exponential Integral En(x) for n>=0, real x>0 **
GAMMAF(x)	Gamma function for real x>0
GAMMAF(a,x)	Incomplete gamma function for real a>0, x>0
LOG10(x)	Base-10 logarithm
LN(x)	Natural logarithm
SIN(x)	sine of x (angle in radians)*
SINH(x)	Hyperbolic sine
SQRT(x)	Square Root
TAN(x)	tangent of x (angle in radians)*
TANH(x)	Hyperbolic tangent

* Use for example COS(x DEGREES) to convert arguments to radians.
** as defined in Abramowitz & Stegun, "Handbook of Mathematical Functions".

Examples:
Samples | Usage | Standard_Functions.pde 391

3.2.6.2 Non-Analytic Functions

The following non-analytic functions are supported in FlexPDE:

MAX(arg1,arg2)

The maximum function requires two arguments. MAX is evaluated on a point by point basis and is equal to the larger of the two arguments at each point.

MIN(arg1,arg2)

The minimum function requires two arguments. MIN is evaluated on a point by point basis and is equal to the lessor of the two arguments at each point.

MOD(arg1,arg2)

The modulo function requires two arguments. MOD is evaluated on a point by point basis and is equal to the remainder of (arg1/arg2) at each point.

GLOBALMAX(arg)

The global maximum function requires one argument. GLOBALMAX is equal to the largest value of the argument over the problem domain. GLOBALMAX is tabulated, and is re-evaluated only when components of the argument change.

GLOBALMAX_X(arg)
GLOBALMAX_Y(arg)
GLOBALMAX_Z(arg)

Returns the specified coordinate of the associated GLOBALMAX. Global searches are tabulated by argument expression, and repeated calls to GLOBALMAX and its related coordinates do not cause repeated evaluation.

GLOBALMIN(arg)

The global minimum function requires one argument. GLOBALMIN is equal to the smallest value of the argument over the problem domain. GLOBALMIN is tabulated, and is re-evaluated only when components of the argument change.

GLOBALMIN_X(arg)
GLOBALMIN_Y(arg)
GLOBALMIN_Z(arg)

Returns the specified coordinate of the associated GLOBALMIN. Global searches are tabulated by argument expression, and repeated calls to GLOBALMIN and its related coordinates do not cause repeated evaluation.

RANDOM(arg)

The random function requires one argument. The result is a pseudo-random number uniformly distributed in (0,arg). The only reasonable application of the RANDOM function is in initial values. Use in other contexts will probably result in convergence failure.

SIGN(arg)

The sign function requires one argument. SIGN is equal to 1 if the argument is positive and -1 if the argument is negative.

TIMEMAX(arg)

The time maximum function requires one argument. TIMEMAX is equal to the largest value of the argument over the time span of the problem. TIMEMAX is tabulated, and is re-evaluated only when components of the argument change.

TIMEMAX_T(arg)

Returns the time at which the associated TIMEMAX of the argument occurs. Time searches are tabulated by argument expression, and repeated calls to TIMEMAX and its related times do not cause repeated evaluation.

TIMEMIN(arg)

The time minimum function requires one argument. TIMEMIN is equal to the smallest value of the argument over the time span of the problem. TIMEMIN is tabulated, and is re-evaluated only when components of the argument change.

TIMEMIN_T(arg)

Returns the time at which the associated TIMEMIN of the argument occurs. Time searches are tabulated by argument expression, and repeated calls to TIMEMIN and its related times do not cause repeated evaluation.

3.2.6.3 Unit Functions

The following unit-valued functions are supported in FlexPDE:

USTEP(arg)

The unit step function requires one argument. USTEP is 1 where the argument is positive and 0 where the argument is negative. For example, USTEP(x-x0) is a step function at x=x0.

UPULSE(arg1,arg2)

The unit pulse function requires two arguments. UPULSE is 1 where arg1 is positive and arg2 is negative and 0 everywhere else. UPULSE(t-t0, t-t1) is a pulse from t0 to t1 if t1>t0. [Note: because instantaneous switches cause serious trouble in time dependent problems, the UPULSE function automatically ramps the rise and fall over 1% of the total pulse width.]

URAMP(arg1,arg2)

The unit ramp function requires two arguments. URAMP is like UPULSE, except it builds a ramp instead of a rectangle. URAMP is 1 where arg1 and arg2 are both positive, linearly interpolated between 0 and 1 when arg1 is positive and arg2 is negative, and 0 everywhere else.

Examples:

Samples | Usage | Unit_Functions.pde 397

3.2.6.4 String Functions

FlexPDE provides support for dynamically constructing text strings.

$number (i.e. <dollar> number)

This function returns a text string representing the integer value of **number**. **number** may be a literal value, a name or a parenthesized expression. If **number** has integral value, the string will have integer format. Otherwise, the string will be formatted as a real number with a default length of 6 characters.

$[width]number

This form acts as the form above, except that the string size will be width.

These functions may be used in conjunction with the concatenation operator "+" to build boundary or region names or plot labels. For example

```
REPEAT i=1 to 4 do
  START  "LOOP"+$i  (x,y)
  { path_info ... }
ENDREPEAT
```

This is equivalent to

```
START "LOOP1" (x,y) <path_info> ...
START "LOOP2" (x,y) <path_info> ...
START "LOOP3" (x,y) <path_info> ...
START "LOOP4" (x,y) <path_info> ...
```

Example:

See "Samples | Usage | Repeat.pde"

3.2.6.5 The FIT Function

The following two forms may be used to compute a finite-element interpolation of an arbitrary argument:

result = FIT(expression)

computes a Finite Element fit of the given expression using the current computational mesh and basis. Nodal values are computed to return the correct integral over each mesh cell.

result = FIT(expression, weight)

as with FIT(expression), but with a smoothing diffusion with coefficient equal to **weight** (try 0.1 or 1.0 , and modify to suit).

weight may be an arbitrary expression, involving spatial coordinates, time, or variables of the computation. In this way it can be used to selectively smooth portions of the mesh. The value of **weight** has a well-defined meaning: it is the spatial wavelength over which variations are damped: spatial variations with wavelength much smaller than **weight** will be smoothed, while spatial variations with wavelength much greater than weight will be relatively unmodified.

> ***Note***: *FIT() builds a continuous representation of the data across the entire domain, and connot preserve discontinuities in the fitted data. In some cases, multiplying the data by an appropriate material parameter can result in a continuous function appropriate for fitting. An exception to this rule is in the case of CONTACT boundaries, where the mesh nodes are duplicated, and discontinuities can be preserved in FIT functions.*

FIT() may be used to smooth noisy data, to block ill-behaved functions from differentiation in the derivative computation for Newton's method, or to avoid expensive re-computation of complex functions.

See also the **SAVE** 131 function, in which nodal values are directly computed.

Example:

Samples | Usage | fit+weight.pde 382

3.2.6.6 The LUMP Function

The LUMP function creates a field on the finite element mesh, and saves a single value of the argument expression in each cell of the finite element mesh. The value stored for each cell is the average value of the argument expression over the cell, and is treated as a constant over the cell.

The LUMP function may be used to block ill-behaved functions from differentiation in the derivative computation for Newton's method, or to avoid expensive re-computation of complex functions.

The normal use for LUMP is in the DEFINITIONS section, as in

name = LUMP (expression)

Note*: This definition of* LUMP(F) *is NOT the same as the "lumped parameters" frequently referred to in finite element literature.*

Example:

Samples | Usage | Lump.pde 384

3.2.6.7 The RAMP Function

The RAMP function is a modification of the URAMP 128 function, intended to make the usage more nearly like an IF..THEN 143 statement.

It has been introduced to provide an alternative to discontinuous functions like USTEP 128 and the discontinuous IF..THEN 143 construct.

Discontinuous switching can cause serious difficulties, especially in time dependent problems, and is strongly discouraged. FlexPDE is an adaptive system. Its procedures are based on the assumption that by making timesteps and/or cell sizes smaller, a scale can be found at which the behavior of the solution is representable by polynomials. Discontinuities do not satisfy this assumption. A discontinuity is a discontinuity, no matter how close you look. Instantaneous turn-on or turn-off introduces high-frequency spatial or temporal components into the solution, including those which are far beyond the physical limits of real systems to respond. This makes the computation slow and possibly physically meaningless.

The RAMP function generates a smooth transition from one value to another, with the transition taking place as "expression" changes by and amount "width". It can be thought of as a "fuzzy IF", and has a usage very similar to an IF.. THEN, but without the harsh switching characteristics.

The form is:

value = RAMP(expression, left_value, right_value, width)

This expression is logically equivalent to

```
value = IF expression < 0 THEN left_value ELSE right_value
```

except that the transition will be linear over width. If the left and right values are functions, then you may not get a straight line as the ramp. The result will be a linear combination of the two functions.

See the SWAGE 132 function for a similar function with both smooth value and derivative.

Example:

see "Samples | Usage | Swage_test.pde" 393 for a picture of the SWAGE and RAMP transitions and their derivatives.

3.2.6.8 The SAVE Function

The SAVE function creates a field on the finite element mesh, and saves the values of the argument expression at the nodal points for subsequent interpolation. SAVE builds a continuous representation of the data within each material region, and can preserve discontinuities in the saved data.

The SAVE function may be used to block ill-behaved functions from differentiation in the derivative computation for Newton's method, or to avoid expensive re-computation of complex functions.

The normal use for SAVE is in the DEFINITIONS section, as in

name = SAVE (expression)

***Note**: SAVE() builds a continuous representation of the data across the entire domain, and cannot preserve discontinuities in the fitted data. In some cases, multiplying the data by an appropriate material parameter can result in a continuous function appropriate for saving. An exception to this rule is in the case of CONTACT boundaries, where the mesh nodes are duplicated, and discontinuities can be preserved in SAVE functions.*

Example:

"Samples | Usage | Save.pde" 386

See the FIT() 129 function for a similar function with integral conservation and variable smoothing capabilities.

3.2.6.9 The SUM Function

The SUM function produces the sum of repetitive terms. The form is:

value = SUM(name, initial, final, expression)

The expression argument is evaluated and summed for name = initial, initial+1, initial+2,...final.

For example, the statement:

```
source = SUM(i,1,10,exp(-i))
```

forms the sum of the exponentials exp(-1)+exp(-2)+...+exp(-10).

The SUM function may be used with data ARRAYs, as in

```
DEFINITIONS
   A = ARRAY(1,2,3,4,5,6,7,8,9,10)
   source = SUM(i,1,10,A[i])
```

Example:
Samples | Usage | Sum.pde 392

3.2.6.10 The SWAGE Function

The SWAGE function has been introduced to provide an alternative to discontinuous functions like USTEP 128 and the discontinuous IF..THEN 143 construct. Discontinuous switching can cause serious difficulties, especially in time dependent problems, and is strongly discouraged.

FlexPDE is an adaptive system. Its procedures are based on the assumption that by making timesteps and/or cell sizes smaller, a scale can be found at which the behavior of the solution is representable by polynomials. Discontinuities do not satisfy this assumption. A discontinuity is a discontinuity, no matter how close you look. Instantaneous turn-on or turn-off introduces high frequency spatial or temporal components into the solution, including those which are far beyond the physical limits of real systems to respond. This makes the computation slow and possibly physically meaningless.

The SWAGE function generates a smooth transition from one value to another. The slope at the center of the transition is the same as a RAMP 130 of the given width, but the curve extends to five times the given width on each side, approaching the end values asymptotically. It also has smooth derivatives. It can be thought of as a "fuzzy IF", and has a usage very similar to an IF.. THEN, but without the harsh switching characteristics.

The form is:

value = SWAGE(expression, left_value, right_value, width)

This expression is logically equivalent to

value = IF expression < 0 THEN left_value ELSE right_value

except that the transition will be smeared over width.

See the RAMP 130 function for a similar function which is smooth in value, but not in derivative.

Example:
see "Samples | Usage | Swage_test.pde" 393 for a picture of the SWAGE and RAMP transitions and their derivatives.

Wiktionary:
swage 1.(noun) A tool, variously shaped or grooved on the end or face, used by blacksmiths and other workers in metals, for shaping their work. 2.(verb)To bend or shape using a swage.

3.2.6.11 The VAL and EVAL functions

There are two ways to evaluate an arbitrary expression at selected coordinates, VAL and EVAL.

value = VAL(expression, x, y)
value = VAL(expression, x, y, z)

The value of expression is computed at the specified coordinates. *The coordinates must be constants*. The value is computed and stored at each phase of the solution process, allowing efficient reference in

many computations.

FlexPDE maintains a "scoreboard" of dependencies and re-evaluates the expression whenever the dependency changes. If the expression depends on a variable, it will also create an implicit coupling between the expression and its point of use, causing the value to be solved simultaneously during the solution phase.

Expression can include derivative terms, but the VAL itself cannot be differentiated.

value = EVAL(expression, x, y)
value = EVAL(expression, x, y, z)

The value of expression is computed at the specified coordinates. *The coordinates may be dynamically variable*. The value is recomputed at each reference, possibly leading to increased run time.

This form does NOT allow FlexPDE to compute implicit couplings between computation nodes referencing and evaluating the value.

Derivative operators applied to EVAL will be passed through and applied to expression.

***Note**: The value returned from these functions must be scalar.*

3.2.6.12 Boundary Search Functions

The functions XBOUNDARY, YBOUNDARY and ZBOUNDARY allow the user to search for the position of a system boundary from an evaluation point:

XBOUNDARY("boundary name")
YBOUNDARY("boundary name")
ZBOUNDARY("surface name")
ZBOUNDARY(surface_number)

In each case, the function returns the X,Y or Z coordinate of the named boundary at the (Y,Z), (X,Z) or (X,Y) coordinates of the current evaluation.

3.2.7 Operators

3.2.7.1 Arithmetic Operators

The following customary symbols can be use in arithmetic expressions:

Operator	Action
-	Unary negate, Forms the negative of a single operand
+	Binary add, Forms the sum of two operands
-	Binary subtract, Forms the difference of two operands
*	Binary multiply, Forms the product of two operands
/	Binary divide, Divides the first operand by the second
^	Binary power, Raises the first operand to the power of the second

These operators can be applied to scalars, arrays or matrices. When used with arrays or matrices, the operations are applied element-by-element.

Special operators are defined to designate conventional matrix and array operations.

Operator	Action
**	Binary MATRIX multiply. Forms the product of two matrices or the product of a MATRIX and an ARRAY. Applied to tensors, the result is the same as the DOT operator.
//	Matrix "division". A1 = A2 // M produces the ARRAY A1 satisfying the equation A2 = M**A1.

3.2.7.2 Complex Operators

The following operators perform various transformations on complex quantities.

REAL (complex)

Extracts the real part of the complex number.

IMAG (complex)

Extracts the imaginary part of the complex number.

CABS (complex)

Computes the magnitude of the complex number, given by
CABS(complex(x,y)) = sqrt(x^2 + y^2).

CARG (complex)

Computes the Argument (or angular component) of the complex number, implemented as
CARG(complex(x,y)) = Atan2(y,x).

CEXP (complex)

Computes the complex exponential of the complex number, given by
CEXP(complex(x,y)) = exp(x + iy) = exp(x)*(cos(y) + i*sin(y)).

CLOG (complex)

Computes the natural logarithm of the complex number, given by
CLOG(complex(x,y)) = ln(x + iy) = ln(sqrt(x^2 + y^2)) + i*arctan(y/x).

CONJ (complex)

Returns the complex conjugate of the complex number.

CSQRT (complex)

Computes the complex square root of the complex number, given by
CSQRT(complex(x,y)) = complex(sqrt((r + x)/2), sign(y)*sqrt((r - x)/2))
where r = CABS(x,y).

3.2.7.3 Differential Operators

Differential operator names are constructed from the coordinate names for the problem, either as defined by the user, or as default names.

First derivative operators are of the form "D<name>", where <name> is the name of the coordinate. Second-derivative operators are of the form "D<name1><name2>".

In the default 2D Cartesian case, the defined operators are "DX", "DY", "DXX", "DXY", and "DYY".

All differential operators are expanded internally into the proper forms for the active coordinate system of the problem.

D<n> (arg)

First order partial derivative of arg with respect to coordinate <n>, eg. DX(arg).

D<n><m> (arg)

Second order partial derivative of arg with respect to coordinates <n> and <m>, eg. DXY(arg).

DIV (vector_arg)

Divergence of vector argument. Produces a scalar result.

DIV (argx, argy {, argz })

Divergence of the vector whose components are argx and argy (and possibly argz in 3D). This is the same as DIV(vector(argx,argy,argz), and is provided for convenience.

DIV (tensor_arg)

Divergence of tensor argument. Produces a vector result. In curvilinear geometry, DIV(GRAD(vector)) is NOT the same as the Laplacian of the components of the vector, because differentiation of the unit vectors introduces additional terms. FlexPDE handles these expansions correctly in all supported geometries.

GRAD (scalar_arg)

Gradient of scalar argument. Produces a vector result.

GRAD (vector_arg)

Gradient of vector argument. This operation produces a **tensor** result. In curvilinear geometry, this creates additional terms due to the differentiation of the unit vectors. It is NOT equivalent to the gradient of the vector components except in Cartesian geometry. FlexPDE handles these expansions correctly in all supported geometries.

CURL (vector_arg)

Returns the vector result of applying the curl operator to vector_arg.

CURL (scalar_arg)

Curl of a scalar_arg (2D only). Assumes arg to be the magnitude of a vector normal to the computation plane, and returns a vector result in the computation plane.

CURL (argx, argy {, argz })

Curl of a vector whose components in the computation plane are argx and argy (and possibly argz in 3D). This is the same as CURL(vector(argx,argy,argz)), and is provided for convenience.

DEL2 (scalar_arg)

Laplacian of scalar_arg. Equivalent to DIV(GRAD(scalar_arg)).

DEL2 (vector_arg)

Laplacian of vector_arg. Equivalent to DIV(GRAD(vector_arg)).

3.2.7.4 Integral Operators

Integrals may be formed over volumes, surfaces or lines. The specific interpretation of the integral operators depends on the coordinate system of the current problem. Integral operators can treat only scalar functions as arguments. You cannot integrate a vector field.

Examples

Samples | Applications | Heatflow | Heat_Boundary.pde 338
Samples | Usage | 3d_Domains | 3D_Integrals.pde 412
Samples | Usage | Constraints | Boundary_Constraint.pde 454
Samples | Usage | Constraints | 3D_Constraint.pde 451
Samples | Usage | Constraints | 3D_Surf_Constraint.pde 453
Samples | Usage | Tintegral.pde 395

3.2.7.4.1 Time Integrals

The operators TINTEGRAL and TIME_INTEGRAL are synonymous, and perform explicit time integration of arbitrary *scalar* values from the problem start time to the current time:

TINTEGRAL (integrand)
TIME_INTEGRAL (integrand)

***Note**: This operator cannot be used to create implicit linkage between variables. Use a* GLOBAL VARIABLE *instead.*

3.2.7.4.2 Line Integrals

The operators BINTEGRAL and LINE_INTEGRAL are synonymous, and perform line integrations of scalar integrands.

The integral is always taken with respect to distance along the line or curve.

The basic form of the LINE_INTEGRAL operator is:

BINTEGRAL (integrand, named_boundary)
LINE_INTEGRAL (integrand, named_boundary)

The boundary specification may be omitted, in which case the entire outer boundary is implied.

2D Line Integrals

In 2D Cartesian geometry, LINE_INTEGRAL is the same as SURF_INTEGRAL.

In 2D cylindrical geometry, SURF_INTEGRAL will contain the 2*pi*r weighting, while LINE_INTEGRAL will not.

2D Line integrals may be further qualified by specifying the region in which the evaluation is to be made:

LINE_INTEGRAL (integrand, named_boundary, named_region)

named_region must be one of the regions bounded by the selected boundary.

3D Line Integrals

3D Line integrals may be computed only on extrusion surfaces of the 3D domain.

LINE_INTEGRAL (integrand, named_boundary, surface_number)
LINE_INTEGRAL (integrand, named_boundary, named_surface)

The named_boundary must exist in the named_surface (ie, it must not have been excluded by LIMITED REGION commands).

3.2.7.4.3 2D Surface Integrals

The synonymous prototype forms of surface integral functions in 2D are:

SINTEGRAL (integrand, named_boundary)
SURF_INTEGRAL (integrand, named_boundary)

Here named_boundary may be specified *by name*, or it can be omitted, in which case the entire outer boundary of the domain is implied.

In two-dimensional Cartesian problems, the surface element is formed by extending the two-dimensional line element a single unit in the Z-direction, so that the surface element is dl*1. In this case, the surface integral is the same as the line integral.

In two-dimensional cylindrical problems, the surface element is formed as 2*pi*r*dl, so the surface integral is NOT the same as the line integral.

The region in which the evaluation is made can be controlled by providing a third argument, as in

SURF_INTEGRAL (integrand, named_boundary, named_region)

named_region must be one of the regions bounded by the selected surface.

3.2.7.4.4 3D Surface Integrals

In three-dimensional problems, there are several forms for the surface integral:

1. Integrals over extrusion surfaces are selected by surface name or number and qualifying region name or number:

SINTEGRAL (integrand, surface, region)
SURF_INTEGRAL (integrand, surface, region)

If region is omitted, the integral is taken over all regions of the specified surface.
If both surface and region are omitted, the integral is taken over the entire outer surface of the domain.

Integrals of this type may be further qualified by selecting the layer in which the evaluation is to be made:

SURF_INTEGRAL (integrand, surface, region, layer)

layer must be one of the layers bounded by the selected surface.

2. Integrals over "sidewall" surfaces are selected by boundary name and qualifying layer name:

SINTEGRAL (integrand, named_boundary, named_layer)
SURF_INTEGRAL (integrand, named_boundary, named_layer)

If layer is omitted, the integral is taken over all layers of the specified surface.

Integrals of this type may be further qualified by selecting the region in which the evaluation is to be made:

SURF_INTEGRAL(integrand, named_boundary, named_layer, named_region)

named_region must be one of the regions bounded by the selected surface.

3. Integrals over entire bounding surfaces of selected subregions are selected by region name and layer name, as with volume integrals:

SINTEGRAL (integrand, named_region, named_layer)
SURF_INTEGRAL (integrand, named_region, named_layer)

If named_layer is omitted, the integral is taken over all layers of the specified surface.

3.2.7.4.5 2D Volume Integrals

The synonymous prototype forms of volume integral functions in 2D are:

INTEGRAL (integrand, region)
VOL_INTEGRAL (integrand, region)

Here region can be specified by number or name, or it can be omitted, in which case the entire domain is implied.

In two-dimensional Cartesian problems, the volume element is formed by extending the two-dimensional cell a single unit in the Z-direction, so that the volume integral is the same as the area integral in the coordinate plane.

In two-dimensional cylindrical problems, the volume element is formed as 2*pi*r*dr*dz, so that the volume integral is NOT the same as the area integral in the coordinate plane. For the special case of 2D cylindrical geometry, the additional operator

AREA_INTEGRAL (integrand, region)

computes the area integral of the integrand over the indicated region (or the entire domain) without the 2*pi*r weighting.

3.2.7.4.6 3D Volume Integrals

The synonymous prototype forms of volume integral functions in 3D are:

INTEGRAL (integrand, region, layer)
VOL_INTEGRAL (integrand, region, layer)

Here layer can be specified by number or name, or it can be omitted, in which case the entire layer stack is implied.

region can also be specified by number or name, or it can be omitted, in which case the entire projection plane is implied.

If region is omitted, then layer must be specified *by name* or omitted. If both region and layer are omitted, the entire domain is implied.

For example,
INTEGRAL(integrand, region, layer) means the integral over the subregion contained in the selected region and layer.
INTEGRAL(integrand, named_layer) means the integral over all regions of the named layer.
INTEGRAL(integrand, region) means the integral over all layers of the selected region.
INTEGRAL(integrand) means the integral over the entire domain.

3.2.7.5 Relational Operators

The following operators may be used in constructing conditional expressions:

Relational Operators

Operator	Definition
=	Equal to

<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
<>	Not equal to

Relational Combinations

Operator	Definition
AND	Both conditions true
OR	Either condition true
NOT	(Unary) reverses condition

Assignment Operator

In addition to its use as an equal operator, problem descriptors use the '=' symbol to assign (associate) values functions and expressions with defined names.

3.2.7.6 String Operators

The following operators can be used in expressions that construct string constants:

Operator	Action
+	Binary add, Forms the catenation of two text-string operands

3.2.7.7 Vector Operators

The following operators perform various transformations on vector quantities.

Vector quantities are assumed to have one component in each of the three coordinate directions implied by the COORDINATES selection, whether the selected model geometry is one, two or three dimensional. For example, a Vector can have a Z-component in a two-dimensional X,Y geometry. The restricted geometry simply means that there is no computable variation of the solution in the missing directions. In the explicit construction of Vectors, the third component may be omitted, in which case it is assigned a value of zero.

CROSS (vector1, vector2)

Forms the cross product of two vectors and returns the resulting vector. In 2D geometries, the CROSS product of two vectors lying in the computation plane returns a vector with a nonzero component only in the direction normal to the problem plane. Where appropriate, FlexPDE will interpret this vector as a scalar, suitable for arithmetic combination with other scalars.

DOT (vector1, vector2)

Forms the dot product of two vectors and returns a scalar value equal to the magnitude of the vector dot product.

MAGNITUDE (vector)

Returns a scalar equal to the magnitude of a vector argument.

MAGNITUDE (argx, argy {, argz })*

Returns a scalar equal to the magnitude of a vector whose components are argx and argy (and

possibly argz).

NORMAL (vector)
NORMAL (argx, argy {, argz})*

Returns a scalar equal to the component of a vector argument normal to a boundary. This operator may be used only in boundary condition definitions or in boundary plots or integrals, where the reference surface is clear from the context of the statement. (See also UNORMAL below).

TANGENTIAL(vector)
TANGENTIAL (argx, argy {, argz })*

Returns a scalar equal to the component of a vector argument tangential to a boundary. This operator may be used only in boundary condition definitions or in boundary plots or integrals, where the reference surface is clear from the context of the statement.

VECTOR (argx {, argy {, argz }})*

Constructs a vector whose components are the scalar arguments. Omitted arguments are assumed zero.

XCOMP (vector)

Returns a scalar whose value is the *first* component of the vector argument (regardless of the names of the coordinates).

YCOMP (vector)

Returns a scalar whose value is the *second* component of the vector argument (regardless of the names of the coordinates).

ZCOMP (vector)

Returns a scalar whose value is the *third* component of the vector argument, if it exists (regardless of the names of the coordinates).

The Special Function UNORMAL

UNORMAL is a built-in function which returns the unit-normal vector at the location of evaluation. It's use is valid only in expressions computed on a system boundary. UNORMAL takes no arguments, as it's arguments are implicitly the coordinates at the point of evaluation.

----
* ***Note***: *arguments in brackets {} are optional.*

3.2.7.8 Tensor Operators

FlexPDE supports limited use of TENSOR quantities, to parallel the results of GRAD(vector).

A TENSOR is a vector of vectors, potentially 3 x 3 components.

TENSOR((T11, T12, T13) , (T21, T22, T23) , (T31, T32, T33))

This operator returns a TENSOR object with the indicated components. Each of the Tij may be any scalar expression.

DOT(vector, tensor)

This operator returns a VECTOR with components ((V1*T11+V2*T21+V3*T31), (V1*T12 +V2*T22+V3*T32), (V1*T13+V2*T23+V3*T33)).

DOT(tensor, vector)

This operator returns a VECTOR with components ((T11*V1+T12*V2+T13*V3), (T21*V1 +T22*V2+T23*V3), (T31*V1+T32*V2+T33*V3)).

DOT(tensor, tensor)

This operator returns a TENSOR representing the matrix product of the tensors. The operator ** can be used to produce the same result (i.e. tensor**tensor).

DIV(tensor)

This operator returns a VECTOR value whose components depend on the metric coefficients of the selected problem geometry. In Cartesian geometry, the result is a VECTOR made up of the divergences of the tensor columns.

TRANSPOSE(tensor)

This operator returns a TENSOR which is the transpose of the argument tensor.

vector * vector

produces a tensor of all combinations of component products.

XXCOMP (tensor)
XYCOMP (tensor)
XZCOMP (tensor)
YXCOMP (tensor)
YYCOMP (tensor)
YZCOMP (tensor)
ZXCOMP (tensor)
ZYCOMP (tensor)
ZZCOMP (tensor)

These operators returns a scalar whose value is the indicated component of the tensor argument (X indicates the first coordinate component, Y the second and Z the third, regardless of the actual assigned names of the coordinates).

3.2.8 Predefined Elements

The problem descriptor language predefines the following element:

PI	3.14159265358979

For Cartesian coordinates in which **'R'** is not specified as a coordinate name or a defined name, the problem descriptor language predefines the following elements:

R	R=SQRT(x^2 + y^2)	radius vector length in 2D

R=SQRT(x^2 + y^2 + z^2) radius vector length in 3D

THETA THETA = ARCTAN(y/x) azimuthal angle in 2D or 3D

Note: *If "R" or "Theta" appear on the left side of a definition before any use in an expression, then the new definition will become the meaning of the name, and the predefined meaning will be hidden.*

In staged problems where "STAGES = integer" is declared in the SELECT section,

STAGE an internally declared index which increments from 1 to integer.

In modal analysis (eigenvalue and eigenfunction) problems where "MODES = integer" is declared in the SELECT section,

LAMBDA an internally declared name which represents the various eigenvalues.

In time-dependent problems, the current timestep interval is available:

DELTAT an internally declared name which returns the size of the current timestep.

3.2.9 Expressions

Value Expressions

Problem descriptors are composed predominantly of arithmetic expressions made of one or more operators, variables, defined values and pairs of parentheses that evaluate to numerical values. In evaluating value expressions, FlexPDE follows the algebraic rules of precedence in which unary operators are evaluated first, followed by binary operators in the following order:

power
multiplication and division
addition and subtraction
relational operators (<, <=, =, <>, >=, >)
relational combinations (AND, OR)

When included in expressions, subexpressions enclosed in pairs of parentheses are evaluated first, without regard to the precedence of any operators which precede or follow them. Parentheses may be nested to any level, with inner subexpressions being evaluated first and proceeding outward. Parentheses must always be used in pairs.

Examples:

```
a = b*(c+d)
div(k*grad(u))
```

Conditional-Value Expressions

Problem descriptors can contain conditional expressions of the form

```
IF condition THEN subexpression ELSE subexpression .
```

This form selects one of the two alternative values as the value of the expression. It is used in expressions like

```
y = IF a THEN b ELSE c
```

analogous to the expression "y = a ? b : c" in the C programming language.

It is ***not*** the procedural alternative construct

```
IF a THEN y=b ELSE y=c{ Wrong ! }
```

familiar in procedural programming languages.

The THEN or ELSE subexpressions my contain nested IF...THEN...ELSE expressions. Each ELSE will bind to the nearest previous IF.

Conditional expressions used in material parameters can cause numerical trouble in the solution of a PDE system, because they imply an instantaneous change in the result value. This instantaneous change violates assumptions of continuity upon which the solver algorithms are based.

See URAMP 128, RAMP 130 and SWAGE 132 for switching functions that transition smoothly between alternative values.

3.2.10 Repeated Text

The REPEAT..ENDREPEAT construct allows the repetition of sections of input text.

The syntax looks like a FOR loop in procedural languages, but we emphasize that in FlexPDE this feature constitutes a *textual* repetition, not a procedural repetition.

The form of a repeat clause is

REPEAT name = initial TO final
REPEAT name = initial BY delta TO final

These statements specify that the following lines of descriptor text should be repeated a number of times. The given name is defined as if it had appeared in the DEFINITIONS section, and is given the value specified by initial.

The repeated section of text is terminated by the statement

ENDREPEAT

At this point, the value of name is incremented by delta (or by one, if no delta is given). If the new value is not greater than final, the repeated text is scanned again with the new value in place of name. If delta is negative, the value of name is decremented and the termination test is modified accordingly.

The REPEAT statement can appear in the following locations:

- in BATCH file lists
- in VARIABLE lists
- in EXTRUSION lists
- in INITIAL VALUE lists
- anywhere the REGION, START or LINE keywords are legal.
- around any plot command or group of plot commands.
- around any DEFINITION or group of DEFINITIONS.
- around any REPORT command or group of REPORT commands.
- around AT points in a HISTORY list

Use of ARRAYS and the $integer string function can extend the power of the REPEAT loop.

Examples:

```
REPEAT xc=1/4 by 1/4 to 7/4
   REPEAT yc=1/4 by 1/4 to 7/4
      START(xc+rad,yc)  ARC(CENTER=xc,yc) ANGLE=360 CLOSE
   ENDREPEAT
ENDREPEAT
```

This double loop constructs a 7 x 7 array of circles, all part of the same REGION.

See the sample problems:

Samples | Usage | Repeat.pde 385

***Note**: REPEAT..ENDREPEAT replaces the older FOR..ENDFOR facility used in earlier versions of FlexPDE. The older facility is no longer supported, and will produce parsing errors.*

3.3 The Sections of a Descriptor

The SECTIONS of a descriptor were outlined in the introduction. In the following pages we present a detailed description of the function and content of each section.

3.3.1 Title

The optional TITLE section can contain one literal string.

When a TITLE is used, the literal string it contains is used as a title label for all MONITORS and PLOTS.

If TITLE is not specified, the plots will not have a title label.

Example:

```
TITLE "this is my first model"
```

3.3.2 Select

The SELECT section, which is optional, is used when it is necessary to override some of the default selectors internal to the program.

Selectors are used to control the flow of the process used to solve a problem.

The SELECT section may contain one or more selectors and their associated values. The default selectors have been chosen to optimize how FlexPDE handles the widest range of problems.

The SELECT section should be used only when the default behavior of FlexPDE is somehow inadequate.

Unlike the other elements used in program descriptors, the proper names used for the selectors are not part of the standard language, are not reserved words, and are not meaningful in other descriptor sections.

The selectors implemented in FlexPDE are specific to a version of FlexPDE, and may not correspond to those available in previous versions of FlexPDE or in other applications using the FlexPDE descriptor language.

3.3.2.1 Mesh Generation Controls

The following controls can be used in the SELECT section to modify the behavior of the mesh generator.

- Logical selectors can be turned on by selector = ON, or merely mentioning the selector
- Logical selectors can be turned off by selector = OFF.
- Numeric selectors are set by selector = number.

ASPECT type: Numeric default: 2.0
Maximum cell aspect ratio for mesh generation in 2D problems and 3D surface meshes. Cells may be stretched to this limit of edge-size ratio.

CURVEGRID type: Logical default: On
If ON, cells will be bent to follow curved boundaries, and a 3D mesh will be refined to resolve surface curvature.
If OFF, neither of these modifications will be attempted, and the computation will proceed with straight-sided triangles or flat-sided tetrahedra. (It may be necessary to turn this option OFF when surfaces are defined by TABLES, because the curvature is infinite at table breaks.)

FEATURE_INDUCTION type: Numeric default: 2
In the initial domain layout, FlexPDE attempts to discover cell sizes necessary to resolve domain elements, iterating to propagate the influence of small features. In complex domains this can become expensive. If feature sizes are relatively uniform, or if the user controls the cell size manually, the iteration can be bypassed by setting FEATURE_INDUCTION to 0.

GRIDARC type: Numeric default: 30 degrees
Arcs will be gridded with no cell exceeding this angle. Other factors may cause the sizes to be smaller.

GRIDLIMIT type: Numeric default: 8
Maximum number of regrids before a warning is issued. Batch runs stop at this limit.

INITGRIDLIMIT type: Numeric default: 5
Maximum number of regridding passes in the initial refinement to define initial values.
INITGRIDLIMIT=0 suppresses initial refinement.

MERGE type: Logical default: On
Allows merging of low-error mesh cells. Only cells which have previously been split can be merged.

MERGEDIST type: Numeric default: Automatic
In the initial domain layout, points closer than MERGEDIST will be coalesced into a single point. This helps overcome the effects of roundoff and input number precision in generation of domains. A default merge distance is computed during initial layout. MERGEDIST will over-ride this default value. Individual values for X, Y and Z coordinates can be set with XMERGEDIST, YMERGEDIST and ZMERGEDIST respectively. (These controls should be used only in unusual cases, when the default value performs incorrectly.)

NGRID type: Numeric default: See below
Specifies the number of mesh rows in each dimension. Use this control to set the maximum cell size in open areas. This is a convenient way to control the overall mesh density in a problem. Default values are shown below:

	1D	2D	3D
Professional	100	15	10
Student	50	10	5

NODELIMIT type: Numeric default: See below

Specifies the maximum node count. If mesh refinement tries to create more nodes than the limit, the cell-merge threshold will be raised to try to balance errors across a mesh of the specified size. This control cannot be used to reduce the size if the initial mesh construction, which is dictated by NGRID, user density controls, and domain boundary feature sizes. Default values are shown below, although these limits will likely not be reachable within the resources of most computers:

	1D	2D	3D
Professional	1,000,000	10,000,000	50,000,000
Student	100	800	1600

REGRID type: Logical default: On
By default, FlexPDE implements adaptive mesh refinement. This selector can be used to turn it off and proceed with a fixed mesh.

SMOOTHINIT type: Logical default: On
Implements a mild initial-value smoothing for time dependent problems, to help ameliorate discontinuous initial conditions.

STAGEGRID type: Logical default: Off
Forces regeneration of mesh with each stage of a staged problem. FlexPDE attempts to detect stage dependencies in the domain and regenerate the mesh, but this selector may be used to override the automatic detection.

XMERGEDIST type: Numeric default: Automatic
See MERGEDIST.

YMERGEDIST type: Numeric default: Automatic
See MERGEDIST.

ZMERGEDIST type: Numeric default: Automatic
See MERGEDIST.

***Note**: See the "Mesh Control Parameters* 173*" section in this manual and the "Controlling Mesh Density* 103*" section in the User Guide for more discussion of mesh control.*

3.3.2.2 Solution Controls

The following controls can be used in the SELECT section to modify the solution methods of FlexPDE.

- Logical selectors can be turned on by selector = ON, or merely mentioning the selector.
- Logical selectors can be turned off by selector = OFF.
- Numeric selectors are set by selector = number.

AUTOSTAGE type: Logical default: On
In STAGED problems, this selector causes all stages to be run consecutively without pause. Turning this selector OFF causes FlexPDE to pause at the end of each stage, so that results can be examined before proceeding.

CHANGELIM type: Numeric default: 0.5(steady state), 0.1(time dependent)
Steady state: Specifies the maximum change in any nodal variable allowed on any Newton iteration step (measured relative to the variable norm). In severely nonlinear problems, it may be necessary to force a slow progress toward the solution in order to avoid pathological behavior of the nonlinear functions.
Time dependent: Specifies the maximum change in one timestep of any nodal variable derived from a

steady-state equation. Changes larger than this amount will cause the timestep to be cut.

CUBIC type: Logical default: Off
Use cubic Finite Element basis (same as ORDER=3). The default is quadratic (ORDER=2). Cubic basis creates a larger number of nodes, and sometimes makes the system more ill-conditioned.

ERRLIM type: Numeric default: 0.002
This is the primary accuracy control. Both the spatial error control XERRLIM the temporal error control TERRLIM are set to this value unless over-ridden by explicit declaration.
[Note: ERRLIM is an *estimate* of the relative error in the dependent variables. The solution is not guaranteed to lie within this error. It may be necessary to adjust ERRLIM or manually force greater mesh density to achieve the desired solution accuracy.]

FIRSTPARTS type: Logical default: Off
By default, FlexPDE integrates all second-order terms by parts, creating the surface terms represented by the Natural boundary condition. This selector causes first-order terms to be integrated by parts as well. Use of this option may require adding terms to Natural boundary condition statements.

FIXDT type: Logical default: Off
Disables the automatic timestep control. The timestep is fixed at the value given in the TIME section. (In most cases, this is not advisable, as it is difficult to choose a single timestep value that is both accurate and efficient over the entire time range of a problem. Consider modifying the ERRLIM control instead.)

HYSTERESIS type: Numeric default: 0.5
Introduces a hysteresis in the decay of spatial error estimates in time-dependent problems. The effective error estimate includes this fraction of the previous effective estimate added into the current instantaneous estimate. This effect produces more stable regridding in most cases.

ICCG type: Logical default: On
Use Incomplete Choleski Conjugate-Gradient in symmetric problems. This method usually converges much more quickly. If ICCG=OFF or the factorization fails, then the Orthomin method will be used.

ITERATE type: Numeric default: 1000 (steady-state)
default: 500(time-dependent)
Primary conjugate gradient iteration limit. This is the count at which convergence-coercion techniques begin to be applied. The actual hard maximum iteration count is 4*ITERATE.

LINUPDATE type: Numeric default: 5
In linear steady-state problems, FlexPDE repeats the linear system solution until the computed residuals are below tolerance, up to a maximum of LINUPDATE passes.

MODES type: Numeric default: 0
Selects the Eigenvalue solver and specifies the desired number of modes. The default is *not* to run an Eigenvalue problem.

NEWTON type: Numeric default: (5/changelim)+40 (steady_state)
default: 1 (time-dependent)
Overrides the default maximum Newton iteration limit.

NONLINEAR type: Logical default: Automatic
Selects the nonlinear (Newton-Raphson) solver, even if the automatic detection process does not want it.

NONSYMMETRIC type: Logical default: Automatic
Selects the nonsymmetric Lanczos conjugate gradient solver, even if the automatic detection process does

not want it.

NOTIFY_DONE type: Logical default: Off
Requests that FlexPDE emit a beep and a "DONE" message at completion of the run.

NRMINSTEP type: Numeric default: 0.009
Sets the minimum fraction of the computed stepsize which will be applied during Newton-Raphson backtracking. This number only comes into play in difficult nonlinear systems. Usually the computed step is unmodified.

NRSLOPE type: Numeric default: 0.1
Sets the minimum acceptable residual improvement in Newton-Raphson backtracking of steady-state solutions.

ORDER type: Numeric default: 2
Selects the order of finite element interpolation (2 or 3). The selectors QUADRATIC and CUBIC are equivalent to ORDER=2 and ORDER=3, respectively.

OVERSHOOT type: Numeric default: 0.0005
Sub-iteration convergence control. Conjugate-Gradient solutions will iterate to a tolerance of OVERSHOOT*ERRLIM. (Some solution methods may apply additional multipliers.

PRECONDITION type: Logical default: On
Use matrix preconditioning in conjugate-gradient solutions. The default preconditioner is the diagonal-block inverse matrix.

PREFER_SPEED type: Logical default: On
This selector chooses parameters for nonlinear time-dependent problems that result in greatest solution speed for well-behaved problems. Equivalent to NEWTON=1, REMATRIX=Off.

PREFER_STABILITY type: Logical default: Off
This selector chooses parameters for nonlinear time-dependent problems that result in greatest solution stability in ill-behaved problems. Equivalent to NEWTON=5, REMATRIX=On.

QUADRATIC type: Logical default: On
Selects use of quadratic Finite Element basis. Equivalent to ORDER=2.

RANDOM_SEED type: Numeric default: random
Specifies the seed for random number generation. May be used to create repeatable solution of problems using random numbers.

REINITIALIZE type: Logical default: Off
Causes each Stage of a STAGED problem to be reinitialized with the INITIAL VALUES specifications, instead of preserving the results of the previous stage.

REMATRIX type: Logical default: Off
Forces a re-calculation of the Jacobian matrix for each step of the Newton-Raphson iteration in nonlinear problems. The matrix is also recomputed whenever the solution changes appreciably, or when the residual is large. Replaces NRMATRIX in previous version.

STAGES type: Numeric default: 1
Parameter-studies may be run automatically by selecting a number of Stages 164. Unless the geometric domain parameters change with stage, the mesh and solution of one stage are used as a starting point for the next.

SUBSPACE type: Numeric default: MIN(2*modes,modes+8)
If MODES has been set to select an eigenvalue problem, this selector sets the dimension of the subspace used to calculate eigenvalues.

TERRLIM type: Numeric default: 0.002
This is the primary temporal accuracy control. In time dependent problems, the timestep will be cut if the estimated relative error in time integration exceeds this value. The timestep will be increased if the estimated temporal error is smaller than this value. TERRLIM is automatically set by the ERRLIM control.
***Note**: TERRLIM is an estimate of the relative error in the dependent variables. The solution is not guaranteed to lie within this error. It may be necessary to adjust TERRLIM to achieve the desired solution accuracy.*

THREADS type: Numeric default: 1
Selects the number of worker threads to use during the computation. This control is useful in increasing computation speed on computers with multiple shared-memory processors. FlexPDE does not support clusters. See "Using Multiple Processors" 112 for more information.

TNORM type: Numeric default: 4
Error averaging method for time-dependent problems. Timestep control is based on summed (2^TNORM) power of nodal errors. Allowable values are 1-4. Use larger TNORM in problems with localized activity in large mesh.

UPFACTOR type: Numeric default: 1
Multiplier on upwind diffusion terms. Larger values can sometimes stabilize a marginal hyperbolic system.

UPWIND type: Logical default: On
"Upwind" convection terms in the primary equation variable. In the presence of convection terms, this adds a diffusion term along the flow direction to stabilize the computation.

VANDENBERG type: Logical default: Off
Use Vandenberg Conjugate-Gradient iteration (useful if hyperbolic systems fail to converge). This method essentially solves (AtA)x = (At)b instead of Ax=b. This squares the condition number and slows convergence, but it makes all the eigenvalues positive when the standard CG methods fail.

XERRLIM type: Numeric default: 0.002
This is the primary spatial accuracy control. Any cell in which the estimated relative spatial error in the dependent variables exceeds this value will be split (unless NODELIMIT is exceeded). XERRLIM is set automatically by the ERRLIM selector.
***Note**: XERRLIM is an estimate of the relative error in the dependent variables. The solution is not guaranteed to lie within this error. It may be necessary to adjust XERRLIM or manually force greater mesh density to achieve the desired solution accuracy.*

3.3.2.3 Global Graphics Controls

The following controls can be used in the SELECT section to modify the behavior of the graphics subsystem.

- Logical selectors can be turned on by selector = ON, or merely mentioning the selector.
- Logical selectors can be turned off by selector = OFF.
- Numeric selectors are set by selector = number.

In the usual case, these selectors can be over-ridden by specific controls in individual plot commands (see

Graphic Display Modifiers 200).

ALIAS (coord) type: string default: Coordinate name
Defines an alternate label for the plot axes. Example: ALIAS(x)="distance".

AUTOHIST type: Logical default: On
Causes history plots to be updated when any other plot is drawn.

BLACK type: Logical default: Off
Draw all graphic output in black only. Use GRAY to select grayscale output.

CDFGRID type: Numeric default: 51
Specifies the default size of CDF output grid (ie, 51x51).

CONTOURGRID type: Numeric default: 51
Resolution specification for contour plots, in terms of the number of plot points along the longest plot dimension. The actual plot grid will follow the computation mesh, with subdivision if the cell size is greater than that implied by the CONTOURGRID control.

CONTOURS type: Numeric default: 15
Target number of contour levels. Contours are selected to give "nice" numbers, and the number of contours may not be exactly as specified here.

ELEVATIONGRID type: Numeric default: 401
Elevation plot grid size used by From..To elevation plots. The actual plot grid will follow the computation mesh, with subdivision if the cell size is greater than that implied by the EVATIONGRID control. Elevations on boundaries ignore this number and use the actual mesh points.

FEATUREPLOT type: Logical default: Off
If this selector is ON, FEATURE boundaries will be plotted in gray.

FONT type: Numeric default: 2
Font=1 selects sans-serif font. Font=2 selects serif font.

GRAY type: Logical default: Off
Draws all plots with a gray scale instead of the default color palette.

HARDMONITOR type: Logical default: Off
Causes MONITORS to be written to the hardcopy (.pg6) file.

LOGLIMIT type: Numeric default: 15
The range of data in logarithmic plots is limited to LOGLIMIT decades below the maximum data value. This is a global control which may be overridden by the local LOG(number) qualifier on the plot command.

NOMINMAX type: Logical default: Off
Deletes "o" and "x" marks at min and max values on all contour plots.

NOTAGS type: Logical default: Off
Suppresses level identifying tags on all contour and elevation plots.

NOTIPS type: Logical default: Off
Plot arrows in vector plots without arrowheads. Useful for bi-directional stress plots.

PAINTED type: Logical default: Off
Draw color-filled contour plots. Plots can be painted individually by selecting PAINT in the plot modifiers.

PAINTGRID type: Logical default: On
Draw color-filled grid plots. Colors represent distinct materials, as defined by parameter matching.

PAINTMATERIALS type: Logical default: On
Synonymous with PAINTGRID, included for symmetry with individual PLOT modifiers.

PAINTREGIONS type: Logical default: Off
Sets PAINTGRID, but selects a different coloring scheme. Colors represent logical regions in 2D, or logical (region x layer) compartments in 3D, instead of distinct material parameters.

PENWIDTH type: Numeric default: 0
Sets the on-screen pen width for all plots. Value is an integer (0,1,2,3,...) which specifies the width of the drawn lines, in thousandths of the pixel width (0 means thin).

PLOTINTEGRATE type: Logical default: On
Integrate all spatial plots. Default is volume and surface integrals, using 2*pi*r weighting in cylindrical geometry. Histories are not automatically integrated, and must be explicitly integrated.

PRINTMERGE type: Logical default: Off
Send all stages or plot times of each EXPORT statement to a single file. By default, EXPORTS create a separate file for each time or stage. Individual EXPORTS can be controlled by the plot modifier MERGE.

SPECTRAL_COLORS type: Logical default: Off
Sets the order of colors used in labeling plots. ON puts red at the bottom (lowest spectral color). OFF puts red at the top (hot). This selector is the reverse of THERMAL_COLORS.

SURFACEGRID type: Numeric default: 51
Selects the minimum resolution for Surface plots, in terms of the number of plot points along the longest plot dimension. The actual plot grid will follow the computation mesh, with subdivision if the cell size is greater than that implied by the SURFACEGRID control.

TEXTSIZE type: Numeric default: 35
Controls size of text on plot output. Value is number of *lines per page*, so larger numbers mean smaller text.

THERMAL_COLORS type: Logical default: On
Sets the order of colors used in labeling plots. ON puts red at the top (hot). OFF puts red at the bottom (lowest spectral color). This selector is the reverse of SPECTRAL_COLORS.

VECTORGRID type: Numeric default: 41
Sets resolution of Vector plots. Arrows are placed on a regular grid with the selected number of points along the longest plot dimension.

VIEWPOINT (x, y, angle) default: negative X&Y, 30
Defines default viewpoint for SURFACE plots and 3D GRID plots. Angle is in degrees. (In 3D cut plane plots, this specifies a position in the cut plane coordinates)

3.3.3 Coordinates

The optional COORDINATES section defines the coordinate geometry of the problem.

Each geometry selection has an implied three-dimensional coordinate structure. In 2D and 1D geometries, the solution if the PDE system is assumed to have no variation in one or two of the coordinate directions. The finite element mesh is therefore constructed in the remaining space, and derivatives in the absent coordinates are assumed to be zero.

In 3D geometry the X & Y coordinates are the projection plane in which a figure is constructed, and the Z coordinate is the direction of extrusion.

The first coordinate in the order of listing is used as the horizontal axis in graphical output, while the second is used as the vertical axis.

The basic form of the COORDINATES section is:

COORDINATES geometry

where geometry may be any of the following:

Name	Coordinate system	Modeled Coordinates
CARTESIAN1	Cartesian (X,Y,Z)	X
CYLINDER1	Cylindrical (R,Phi,Z)	R
SPHERE1	Spherical (R,Theta,Phi)	R
CARTESIAN2	Cartesian (X,Y,Z)	X,Y
XCYLINDER	Cylindrical (Z,R,Phi)	Z,R
YCYLINDER	Cylindrical (R,Z,Phi)	R,Z
CARTESIAN3	Cartesian (X,Y,Z)	X,Y,Z

If no COORDINATES section is specified, a CARTESIAN2 coordinate system is assumed.

Renaming Coordinates

A second form of the COORDINATES section allows renaming (aliasing) of the coordinates:

COORDINATES geometry ('Xname' [,'Yname' [,'Zname']])

In this case, the 'Xname' argument renames the coordinate lying along the horizontal plot axis, and 'Yname' renames the coordinate lying along the vertical plot axis. 'Zname' renames the extrusion coordinate. Names may be quoted strings or unquoted names. Renaming coordinates does not change the fundamental nature of the coordinate system. In cylindrical geometries, for example, the radial coordinate will continue to be the radial coordinate, even if you name it "Z".

In time-dependent problems, the time coordinate may be renamed using TIME ('Tname') in the COORDINATES section :

COORDINATES geometry TIME ('Tname')

This may be used in conjunction with the renaming of spatial coordinates.

Differential Operators

Renaming coordinates causes a redefinition of the differential operators. DX becomes D<Xname>, etc.

The DIV, GRAD, and CURL operators are expanded correctly for the designated geometry. Use of these operators in the EQUATIONS section can considerably simplify problem specification.

Other Geometries

Since FlexPDE accepts arbitrary mathematical forms for equations, it is always possible to construct equations appropriate to an arbitrary geometry.

For example, using the CARTESIAN2 coordinate system and renaming coordinates, one can write the heat equation for cylindrical geometry as

```
COORDINATES cartesian2("R","Z")
VARIABLES u
...
EQUATIONS
u:  dr(k*r*dr(u)) + r*dz(k*dz(u)) + r*source = 0
```

This equation derives from expanding the DIV and GRAD operators in cylindrical coordinates and multiplying by the volume weighting factor "r", and is the same as the equation that FlexPDE itself will construct in XCYLINDER geometry.

Coordinate Transformations

The function definition facility of FlexPDE can be used to simplify the transformation of arbitrary coordinates to Cartesian (X,Y,Z) coordinates.

The example problem "Samples | Usage | polar_coordinates.pde" 384 uses this facility to pose equations in polar coordinates:

```
DEFINITIONS
    dr(f) = (x/r)*dx(f) + (y/r)*dy(f)        { functional definition of polar derivatives... }
    dphi(f) = (-y)*dx(f) + x*dy(f)           {... in cartesian coordinates }

EQUATIONS                            { equation expressed in polar coordinates }
                                     { (and Multiplied by r^2 to clear the r=0 singularity) }
U:   r*dr(r*dr(u)) + dphi(dphi(u)) + r*r*s = 0
```

Graphic output using this procedure is always mapped to the fundamental Cartesian coordinate system.

3.3.4 Variables

The VARIABLES section is used to define and assign names to all the primary dependent variables used in a problem descriptor. The form of this section is

VARIABLES   variable_name_1 , variable_name_2 ,...

All names appearing in the VARIABLES section will be represented by a finite element approximation over

the problem mesh. Each variable is assumed to define a continuous field over the problem domain. It is further assumed that each variable will be accompanied by a partial differential equation listed in the EQUATIONS section.

Each variable_name may be followed by various qualifiers, which will be described in subsequent sections. These qualifiers allow you to control mesh motion, declare complex and vector variables, declare arrays of variables, and control some of the ways FlexPDE treats the variable.

In assigning names to the dependent variables, the following rules apply:

- Variable names must begin with an alphabetic character. They may not begin with a number or symbol.
- Variable names may be a single character other than the single character "t", which is reserved for the time variable.
- Variable names may be of any length and any combination of characters, numbers and/or symbols other than reserved words.
- Variable names may not contain any separators. Compound names can be formed with the '_' symbol (e.g. temperature_celsius).
- Variable names may not contain the character '-' which is reserved for the minus sign.

Example:

```
VARIABLES
    U,V
```

3.3.4.1 The THRESHOLD Clause

An optional THRESHOLD clause may be associated with a variable name.

The THRESHOLD value determines the *minimum* range of values of the variable for which FlexPDE must try to maintain the requested ERRLIM accuracy. In other words, THRESHOLD defines the level of variation at which the user begins to lose interest in the details of the solution.

Error estimates are scaled to the *greater* of the THRESHOLD value or the observed range of the variable, so the THRESHOLD value becomes meaningless once the observed variation of a variable in the problem domain exceeds the stated THRESHOLD. If you make the THRESHOLD too large, the accuracy of the solution will be degraded. If you make it too small, you will waste a lot of time computing precision you don't need. So if you provide a THRESHOLD, make it a modest fraction of the expected range (max minus min) of the variable.

The THRESHOLD clause has two alternative forms:

variable_name (THRESHOLD = number)
variable_name (number)

Note*: In most cases, the use of* THRESHOLD *is meaningful only in time-dependent or nonlinear steady-state problems with uniform initial values, or that ultimately reach a solution of uniform value.*

3.3.4.2 Complex Variables

You may declare that a VARIABLE name represents a complex quantity. The format of a complex declaration is:

variable_name = COMPLEX (real_name , imaginary_name)

This declaration tells FlexPDE that variable_name represents a complex quantity, and assigns the real_name and imaginary_name to the real and imaginary parts of variable_name. You may subsequently assign EQUATIONS and boundary conditions either to the variable_name, or to its components individually. Similarly, you can perform arithmetic operations or request graphical output of either the variable_name itself, or its components individually.

Example:

```
VARIABLES
   U,V
   C = COMPLEX(Cr,Ci)
```

3.3.4.3 Moving Meshes

FlexPDE can be configured to move the finite element mesh in time-dependent problems.

In order to do this, you must assign a VARIABLE as a surrogate for each coordinate you wish to modify. This specification uses the form

variable_name = MOVE (coordinate_name)

This declaration assigns variable_name as a surrogate variable for the coordinate_name. You may subsequently assign EQUATIONS and boundary conditions to the surrogate variable in the normal way, and these equations and boundary conditions will be imposed on the values of the selected mesh coordinate at the computation nodes.

Example:

```
VARIABLES
   U,V
   Xm = MOVE(X)
```

See Moving Meshes 177 later in this document and the Moving Meshes chapter in the User Guide 100.

3.3.4.4 Variable Arrays

You may declare that a VARIABLE name represents an array of variables. The format of a variable array declaration is:

variable_name = ARRAY [number]

This declaration tells FlexPDE that variable_name represents an array of variable quantities, each one a scalar field on the problem domain. FlexPDE creates internal names for the elements of the array by subscripting variable_name with "_" and the element number (e.g. U_7). You can access the components either by this internal name or by an indexed reference variable_name[index].

You may subsequently assign EQUATIONS and boundary conditions either to the individual components, or in a REPEAT loop by indexed reference. Similarly, you can perform arithmetic operations or request graphical output of either the indexed array name, or by the individual component names.

Example:

```
VARIABLES
   A = ARRAY[10]      { declares ten variables A_1 through A_10 }
                      { also accessible as  A[1] through A[10] }
```

See example problems:
Samples | Usage | Variable_Arrays | array_variables.pde 521

3.3.4.5 Vector Variables

You may declare that a VARIABLE name represents a vector quantity. The format of a vector declaration is:

variable_name = VECTOR (component1)
variable_name = VECTOR (component1 , component2)
variable_name = VECTOR (component1 , component2 ,component3)

This declaration tells FlexPDE that variable_name represents a vector quantity, and assigns the component names to the geometric components of variable_name. You may subsequently assign EQUATIONS and boundary conditions either to the variable_name, or to its components individually. Similarly, you can perform arithmetic operations or request graphical output of either the variable_name itself, or its components individually.

The three component names correspond to the coordinate directions as implied in the COORDINATES section of the problem descriptor. You can declare any or all of the three component directions, even if the model domain treats only one or two.

Any of the component names can be replace by "0" to indicate that this component of the vector is not to be modeled by FlexPDE, but is to be assumed zero. Similarly, omitted names cause the corresponding vector components to be assumed zero.

Example:
In XCYLINDER geometry, which has coordinates (Z,R,Phi), you can tell FlexPDE to model only the Phi component of a vector quantity as follows:

```
VARIABLES
   A = Vector(0,0,Aphi)
```

See example problems:
Samples | Usage | Vector_Variables 522
Samples | Applications | Fluids | 3d_Vector_Flowbox.pde 309
Samples | Applications | Fluids | Vector_Swirl.pde 327
Samples | Applications | Magnetism | 3D_Vector_Magnetron.pde 347
Samples | Applications | Magnetism | Vector_Magnet_Coil.pde 356

3.3.5 Global Variables

The **GLOBAL VARIABLES** section is used to define auxiliary or summary values which are intricately linked to the field variables.

Each GLOBAL VARIABLE takes on a single value over the entire domain, as opposed to the nodal finite element field representing a VARIABLE.

GLOBAL VARIABLES differ from simple DEFINITIONS in that DEFINITIONS are algebraically substituted in place of their references, while GLOBAL VARIABLES represent stored values which are assigned a row and column in the master coupling matrix and are solved simultaneously with the finite element equations.

The GLOBAL VARIABLES section must follow immediately after the VARIABLES section.

Rules for declaring GLOBAL VARIABLES are the same as for VARIABLES, and a GLOBAL VARIABLE may have a THRESHOLD, and may be declared to be COMPLEX, VECTOR or ARRAY, as with VARIABLES.

Each GLOBAL VARIABLE will be associated with an entry in the EQUATIONS section, with rules identical to those for VARIABLES.

GLOBAL VARIABLES do not have boundary conditions. They may be either steady-state or time-dependent, and may be defined in terms of integrals over the domain, or by point values of other functions.

Examples:

Samples | Applications | Control | Control_Steady.pde 293
Samples | Applications | Control | Control_Transient.pde 294

***Note**: In previous versions of FlexPDE, Global Variables were referred to as SCALAR VARIABLES. This usage is still allowed for compatibility, but the newer terminology is preferred.*

3.3.6 Definitions

The **DEFINTIONS** section is used to declare and assign names to special numerical constants, coefficients, and functions used in a problem descriptor.

In assigning names to the definitions, the following rules apply:

- Must begin with an alphabetic character. May *not* begin with a number or symbol.
- May be a single character other than the single character t, which is reserved for the time variable.
- May be of any length and any combination of characters, numbers, and symbols other than reserved words, coordinate names or variable names.
- May *not* contain any separators. Compound names can be formed with the '_' symbol (e.g. temperature_celsius).
- May *not* contain the '-' which is reserved for the minus sign.

Normally, when a definition is declared it is assigned a value by following it with the assignment operator '=' and either a value or an expression. Definitions are dynamic elements and when a value is assigned, it will be the initial value only and will be updated, if necessary, by the problem solution.

Example:

```
Viscosity = 3.02e-4*exp(-5*Temp)
```

Definitions are expanded inline in the partial differential equations of the EQUATIONS section. They are not represented by a finite element approximation over the mesh, but are calculated as needed at various times and locations.

Redefining Regional Parameters

Names defined in the DEFINITIONS section may be given overriding definitions in some or all of the REGIONS of the BOUNDARIES section. In this case, the quantity may take on different region-specific values. Quantities which are completely specified in subsequent REGIONS may be stated in the DEFINITIONS section without a value.

> ***Note:*** *See the User Guide section "Setting Material Properties by Region"* 72 *for examples of redefined regional parameters.*

Defining Constant Values

Normally, DEFINITIONS are stored as the defining formulas, and are recomputed as needed. In rare cases (as with RANDOM elements), this is inappropriate. The qualifier CONST() can be used to force the storage of numeric values instead of defining formulas. Values will be computed when the script is parsed, and will not be recomputed.

name = CONST (expression)

Note: *Scripts with staged geometry* 164 *will reparse the script file and regenerate any CONST values.*

3.3.6.1 ARRAY Definitions

Names may be defined as representing arrays or lists of values. ARRAY definition can take several forms:

name = ARRAY (value_1 , value_2 ... value_n)
defines name to be an n-element array of values value_1 ... value_n.

name = ARRAY [number]
defines name to be an array of number elements. Values are as yet undefined, and must be supplied later in the script.

name = ARRAY [number] (value_1 , value_2 ... value_number)
defines name to be an array of number elements, whose values are value_1, value_2, etc.

name = ARRAY FOR param (initial BY step TO final) : expression
defines name to be an array of values generated by evaluating expression with param set to initial, initial + step, initial + 2*step, and so forth up to param = final.

name = ARRAY FOR param (P1 , P2 { , P3 ...}) : expression
defines name to be an array of values generated by evaluating expression with param set to P1, P2, and so forth up to the end of the listed parameters.

The values assigned to ARRAY elements must evaluate to scalar numbers. They may contain coordinate or variable dependencies, but must not be VECTOR, COMPLEX or TENSOR quantities.

Examples:

```
v = array(0,1,2,3,4,5,6,7,8,9,10)
w = array(0 by 0.1 to 10)
alpha =array for x(0 by 0.1 to 10)  : sin(x)+1.
```

Referencing ARRAY values

Within the body of the descriptor, ARRAY values may be referenced by the form

name [index]

The value of the selected ARRAY element is computed and used as though it were entered literally in the text.

ARRAY elements that have not been previously assigned may be given values individually by conventional assignment syntax:

name [index] = expression

Arithmetic Operations on ARRAYS

Arithmetic operations may be performed on ARRAYS as with scalar values. Names defined as the result of ARRAY arithmetic will be implicitly defined as ARRAYS. Arithmetic operations and functions on ARRAYS are applied element-by-element.

ARRAYS may also be operated on by MATRICES 161 (q.v.)

Example:

```
beta = sin(w)+1.1       { beta is an ARRAY with the same data as alpha }
gamma = sin(v)+0.1      { gamma is an ARRAY with the dimension of v }
```

The SIZEOF operator

The operator SIZEOF may be used to retrieve the allocated size of an ARRAY.

Example:

```
n = SIZEOF(v)           { returns 11, the allocates size of the example array "v" above }
```

ARRAYS of Constant Values

Normally, ARRAYS are stored as the defining formulas for the elements, and are recomputed as needed.

In rare cases (as with RANDOM elements), this is inappropriate. The qualifier CONST can be prepended to the ARRAY definition to force the storage of numeric values instead of defining formulas. Elements will be computed when the script is parsed, and will not be recomputed. For example:

name = CONST ARRAY (value_1 , value_2 ... value_n)

Note: *Scripts with staged geometry* 164 *will reparse the script file and regenerate any CONST* 158 *values.*

See Also: "Using ARRAYS and MATRICES" 109

3.3.6.2 MATRIX Definitions

Names may be defined as representing matrices or tables of values. MATRIX definition can take several forms:

name = MATRIX ((value_11 , value_12 ... value_1m) ,
... (value_n1 , value_n2 ... value_nm))
defines name to be a matrix of values with n rows and m columns.

name = MATRIX [rows , columns]
defines name to be an matrix of elements with the stated dimensions. Values are as yet undefined, and must be supplied later in the script.

name = MATRIX [n , m] ((value_11 , value_12 ... value_1m) ,
... (value_n1 , value_n2 ... value_nm))
defines name to be an array of number elements, whose values are as listed.

name = MATRIX FOR param1 (initial1 BY step1 TO final1)
FOR param2 (initial2 BY step2 TO final2) : expression
defines name to be a matrix of values generated by evaluating expression with param1 and param2 set to the indicated range of values. param2 is cycled to create columns, and param1 is cycled to create rows.

name = MATRIX FOR param1 (P11 , P12 { , P13 ...})
FOR param1 (P21 , P22 { , P23 ...}) : expression
defines name to be a matrix of values generated by evaluating expression with param1 and param2 set to the indicated range of values. param2 is cycled to create columns, and param1 is cycled to create rows.

The values assigned to MATRIX elements must evaluate to scalar numbers. They may contain coordinate or variable dependencies, but must not be VECTOR, COMPLEX or TENSOR quantities.

Examples:

```
m1 = matrix((1,2,3),(4,5,6),(7,8,9))

m2 = matrix for x(0.1 by 0.1 to 5*pi/2)     { a 79x79 diagonal matrix of amplitude 10
}
             for y(0.1 by 0.1 to 5*pi/2)  :  if(x=y) then 10 else 0
```

```
m3 = matrix for x(0.1 by 0.1 to 5*pi/2)     { a 79x79 matrix of sin products }
            for y(0.1 by 0.1 to 5*pi/2)     :   sin(x)*sin(y) +1
```

Referencing MATRIX values

Within the body of the descriptor, MATRIX values may be referenced by the form

name [row_index , column_index]

The value of the selected MATRIX element is computed and used as though it were entered literally in the text.

MATRIX elements that have not been previously assigned may be given values individually by conventional assignment syntax:

name [row_index , column_index] = expression

Arithmetic Operations on MATRICES

Arithmetic operations may be performed on MATRICES. Names defined as the result of MATRIX arithmetic will be implicitly defined as MATRICES or ARRAYS, as appropriate to the operation.

- Standard arithmetic operations and functions on MATRICES are applied element-by-element.
- The special operator ** is defined for conventional matrix multiplication

Examples:

N = M1 * M2	{ N is a MATRIX, each element of which is the product of corresponding elements in M1 and M2 }
S = sin(M)	{ S is a MATRIX, each element of which is the sine of the corresponding element of M }
N = M1 ** M2	{ N is a MATRIX, each element of which is the dot product of corresponding row in M1 and column in M2 (ie, conventional matrix multiplication) }

Arithmetic Operations of MATRICES on ARRAYS

Arithmetic operations may be performed by MATRICES on ARRAYS. Names defined as the result of these operations will be implicitly defined as ARRAYS, as appropriate to the operation. The MATRIX and ARRAY appearing in such operations must agree in dimensions or the operation will be rejected.

- The special operator ****** is defined for conventional (matrix x vector) multiplication, in which each element of the result vector is the dot product of the corresponding matrix row with the argument vector.
- The special operator **//** is defined for (vector / matrix) division. This operation is defined as multiplication of the vector by the inverse of the argument matrix.

Examples:

V2 = M ** V1 { V2 is an ARRAY, each element of which is the dot product of the corresponding row of M with the ARRAY V1 }

V2 = V1 // M { V2 is an ARRAY that satisfies the equation M**V2 = V1 }

The TRANSPOSE operator

The operator TRANSPOSE may be used to create the tranpose of a MATRIX.

The SIZEOF operator

The operator SIZEOF may be used to retrieve the allocated size of a MATRIX.

Example:

```
n = SIZEOF(v)          { returns 11, the allocates size of the example array "v" above }
```

MATRICES of Constant Values

Normally, MATRICES are stored as the defining formulas for the elements, and are recomputed as needed. In rare cases (as with RANDOM elements), this is inappropriate. The qualifier CONST can be prepended to the MATRIX definition to force the storage of numeric values instead of defining formulas. Elements will be computed when the script is parsed, and will not be recomputed. For example:

```
name = CONST MATRIX ( ( value_11 , value_12 ... value_1m ) ,
                  ... ( value_n1 , value_n2 ... value_nm) )
```

See Also: "Using ARRAYS and MATRICES" 109

3.3.6.3 Function Definitions

Definitions can be made to depend on one to three explicit arguments, much as with a Function definition in a procedural language. The syntax of the parameterized definition is

```
name ( argname ) = expression
name ( argname1 , argname2 ) = expression
name ( argname1 , argname2 , argname3 ) = expression
```

The construct is only meaningful if expression contains references to the argnames. Names defined in this way can later be used by supplying actual values for the arguments. As with other definitions in FlexPDE, these actual parameters may be any valid expression with coordinate or variable dependences. The argnames used in the definition are local to the definition and are undefined outside the scope of the defining expression.

Note that it is never necessary to pass known definitions, such as coordinate names, variable names, or other parameters as arguments to a parameterized definition, because they are always globally known and are evaluated in the proper context. Use the parameterized definition facility when you want to pass values

that are not globally known.

> ***Note****: This construct is implemented by textual expansion of the definitions in place of the function reference. It is not a run-time call, as in a procedural language.*

Example:

```
DEFINITIONS
  sq(arg) = arg*arg
  ...
EQUATIONS
  div(a*grad(u)) + sq(u+1)*dx(u) +4 = 0;
```

In this case, the equation will expand to

```
  div(a*grad(u)) + (u+1)*(u+1)*dx(u) + 4 = 0.
```

See also "Samples | Usage | Function_Definition.pde" 382

3.3.6.4 STAGED Definitions

FlexPDE can perform automated parameter studies through use of the STAGE facility. In this mode, FlexPDE will run the problem a number of times, with differing parameters in each run. Each STAGE begins with the solution and mesh of the previous STAGE as initial conditions.

HISTORY 209 plots can be used to show the variation of scalar values as the STAGES proceed.

Note: *The STAGE facility can only be used on steady-state problems. It cannot be used with time dependent problems.*

The STAGES Selector

In the SELECT section, the statement

STAGES = number

specifies that the problem will be run number times. A parameter named STAGE is defined, which takes on the sequence count of the staged run. Other definitions may use this value to vary parameter values, as for example:

```
Voltage = 100*stage
```

STAGED Definitions

A parameter definition may also take the form:

param = STAGED (value_1, value_2, ... value_n)

In this case, the parameter param takes on value_1 in stage 1, value_2 in stage 2, etc.
If STAGED parameters are defined, the STAGES selector is optional. If the STAGES selector is not defined, the length of the STAGED list will be used as the number of stages. If the STAGES selector is defined, it overrides the length of the STAGED list. Commas are optional.

See the example "Samples | Usage | Stages.pde" 390.

STAGED Definitions by incrementation

Any **value** in the **STAGED** form above may be replaced by the incrementation form

value_i BY increment TO value_j

STAGED Geometry

If the geometric domain definition contains references to staged quantities, then the solution and mesh will not be retained, but the mesh will be regenerated for the new geometry. History plots can still be displayed for staged geometries.

See the example "Samples | Usage | Staged_Geometry.pde" 389.

FlexPDE attempts to detect stage dependence in the geometrical domain definition and automatically regenerate the mesh. If for any reason these dependencies are undetected, the global selector STAGEGRID can be used to force grid staging.

Note: *Scripts with staged geometry will reparse the script file and regenerate any CONST* 158 *values.*

3.3.6.5 POINT Definitions

A name may be associated with a coordinate point by the construct

point_name = POINT(a,b)

Here a and b must be computable constants at the time the definition is made. They may not depend on variables or coordinates. They may depend on stage number.

The name of the point can subsequently appear in any context in which the literal point **(a,b)** could appear.

Individual coordinates of a named point can be extracted using the XCOMP, YCOMP or ZCOMP functions.

Movable Points

Named points that are used in boundary definitions in moving-mesh problems become locked to the mesh, and will move as the mesh moves.

Such points can be used in "AT" selectors for histories to track values at points that move with the mesh.

3.3.6.6 TABLE Import Definitions

FlexPDE supports the import of tabular data in several script commands. In each case, the model assumes that a text file contains data defining one or more functions of one, two or three coordinates. The coordinates may be associated with any quantity known to FlexPDE, such as a spatial coordinate, a variable, or any defined quantity. At each point of evaluation, whether of a plot or a quadrature

computation of coupling matrix, or any other context, the values of the declared coordinates of the table are computed and used as lookup parameters to interpolate data from the table.

This feature is useful for modeling systems where experimental data is available and for interfacing with other software programs.

The names of quantities to be used as table coordinates may be declared inside the table file, or they may be imposed by the TABLE input statement itself.

Table coordinates must be in monotonic increasing order.

TABLE data are defined on a rectangular grid, and interpolated with linear, bilinear or trilinear interpolation. Modifiers can be prepended to table definitions to create spline interpolation or histogram interpretation, or to smooth the imported data.

Table import files are ASCII text files, and can be generated with any ASCII text editor, by user programs designed to generate tables, or by FlexPDE itself, using the EXPORT plot modifier or the TABLE output statement (see MONITORS and PLOTS 197).

See TABLE File Format 168 for a definition of the table file format.
See Importing Data from other applications 108 for a discussion of TABLE usage.

3.3.6.6.1 The TABLE Input function

A single imported data function may be declared by one of the forms:

name = TABLE ('filename')
name = TABLE ('filename', coord1 [,coord2...])

Both forms import a data table from the named file and associate the data with the defined **name**.

In the first form, the coordinates of the table must be named in the file.
In the second form, the coordinates are named explicitly in the command.
In either case, the declared coordinates must be names known to FlexPDE at the time of reading the file.

The format of the TABLE file describes a function of one, two or three coordinates. The

The TABLE statement must appear in a parameter definition (in the DEFINITIONS section or as a regional parameter definition in a REGION clause), and the table data are associated with the given name.

> ***Note**: Unlike previous versions of FlexPDE, version 6 does not allow TABLE to be used directly in arithmetic expressions.*

When the parameter name is used in subsequent computations, the current values of the table coordinates will be used to interpolate the value. For instance, if the table coordinates are the spatial coordinates X and Y, then during computations or plotting, the named parameter will take on a spatial distribution corresponding to the table data spread over the problem domain.

> ***Note**: The SPLINETABLE function used in previous versions of FlexPDE is still supported, but is deprecated. Use the SPLINE modifier 167 instead.*

Examples:

Samples | Usage | Import-Export | Table.pde 482

3.3.6.6.2 The TABLEDEF input statement

The TABLEDEF input statement is similar to the TABLE 166 input function, but can be used to directly define one or several parameters from a multi-valued table file.

The format is

TABLEDEF('filename' , name1 { , name2 , ... })

Whereas in the TABLE statement the additional arguments are coordinate reassignments, in the TABLEDEF statement the additional arguments are the names to be defined and associated with the table data. The TABLEDEF statement is not able to redefine the names of the table coordinates, and the names in the table file must be those of values known to FlexPDE at the time of reading the table.

The TABLEDEF statement is syntactically parallel to the TRANSFER statement.

TABLEDEF may optionally be preceded by TABLE modifiers 167.

3.3.6.6.3 TABLE Modifiers

The default interpolation for table data is linear (or bilinear or trilinear) within the table cells. Alternative treatments of the data can be specified by prefixes attached to the **TABLE** statement.

Modifier	Effect
SPLINE	A cubic spline is fit to the table data (one- and two-dimensional tables only)
BLOCK	Data points are assumed to denote the beginning of a histogram level. The data value at a given point will apply uniformly to the coordinate interval ending at the next coordinate point. A ramped transition will be applied to the interpolation, transitioning from one level to the next in 1/10 of the combined table cell widths.
BLOCK(fraction)	Data are interpreted as with BLOCK, but fraction is used as the transition width factor in place of the default 1/10.
SMOOTH (wavelength)	A diffusive smoothing is applied to the TABLE data, in such a way that the integral of the data is preserved, but sharp transitions are blurred. This can result in more efficient solution times if the data are used as sources or parameters in time-dependent problems. Fourier components with spatial wavelengths less than wavelength will be damped. (See Technical Note: Smoothing Operators in PDE's 277).

Examples:

```
Data = SMOOTH(0.1) TABLE("input_file")
Data = SPLINE TABLE("input_file")
```

3.3.6.6.4 TABLE File format

Data files for use in TABLE or TABLEDEF input must have the following form:

```
{ comments }
name_coord1   datacount1
  value1_coord1   value2_coord1   value3_coord1 ...
name_coord2   datacount2
  value1_coord2   value2_coord2   value3_coord1 ...
name_coord3   datacount3
  value1_coord3   value2_coord3   value3_coord3 ...
data { comments }
data111   data211   data311 ...
data121   data221   data321 ...
data131   data231   data331 ...
 ...       ...       ...
 ...       ...       ...
data112   data 212   data312 ...
data122   data 222   data322 ...
data132   data 232   data 332 ...
 ...       ...       ...
 ...       ...       ...
```

where

name_coordN	is the coordinate name in the N direction. Names must match defined names in the importing script unless table coordinate redefinition is used.
valueN_coordM	is the Nth value of the Mth coordinate. These must be in monotonic increasing order.
datacountN	is the number of data points in the N direction.
DataJKL	is the data at coordinate point (J,K,L)
...	ellipses indicate extended data lists, which may be continued over multiple lines.

Note that in presenting data, coord1 is cycled first, then coord1, then coord3.
Coordinate lists and data lists are free-format, and may be arbitrarily spaced, indented or divided into lines.

Example:

```
{ this is an example table.  }
x   6
   -0.01 2 4 6 8 10.01
y   6
   -0.01 2 4 6 8 10.01
data
   1.1    2.1     3.1     4.1     5.1     6.1
   1.2    2.2     3.2     4.2     5.2     6.2
   1.3    2.3     3.3     4.3     5.3     6.3
   1.4    2.4     3.4     4.4     5.4     6.4
```

```
1.5    2.5    3.5    4.5    5.5    6.5
1.6    2.6    3.6    4.6    5.6    6.6
```

3.3.6.7 TABULATE definitions

The TABULATE statement can be used to generate a TABLE internally from arithmetic expressions. The result is a TABLE identical to one produced externally and read by the TABLE or TABLEDEF statements.

This facility can be used to tabulate parameters that are very expensive to compute, resulting in an improvement in the efficiency of the system solution.

The TABULATE statement has a syntax identical to that of ARRAY and MATRIX definition, with the addition of a possible third table dimension.

name = TABULATE FOR param1 (first1 BY step1 TO final1) : expression

name = TABULATE FOR param1 (first1 BY step1 TO final1)
FOR param2 (first2 BY step2 TO final2) : expression

name = TABULATE FOR param1 (first1 BY step1 TO final1)
FOR param2 (first2 BY step2 TO final2)
FOR param3 (first3 BY step3 TO final3) : expression

These statements define name to be a TABLE of values generated by evaluating expression at all combinations of the specified parameters. param1, param2 and param3 must be names already defined in the script, and they become the coordinate values of the table.

As with MATRICES and ARRAYS, table points can be stated explicitly

name = TABULATE FOR param1 (p11 , p12 { , p13 ...}) : expression

The two forms of coordinate definition can be mixed at will, as in

name = TABULATE FOR param1 (p1 , p2 , p3 BY step TO final , pN) : expression

Interpretation of the resulting table can be modified as with the TABLE statement, by prefixing the TABULATE clause by the modifiers SPLINE, BLOCK or SMOOTH.

3.3.6.8 TRANSFER Import Definitions

FlexPDE supports a TRANSFER facility for exchanging data between FlexPDE problem runs. The format is unique to FlexPDE, and is not supported by other software products.

A TRANSFER file contains data defined on the same unstructured triangle or tetrahedral mesh as used in the creating FlexPDE computation, and maintains the full information content of the original computation. It also contains a description of the problem domain definition of the creating run.

The TRANSFER input statement has three forms

TRANSFER ('filename' , name1 { , name2 , ... })
TRANSFERMESH ('filename', name1 { ,name2,.. })
TRANSFERMESHTIME ('filename', name1 { ,name2,.. })

The file specified in the transfer input function must have been written by FlexPDE using the TRANSFER output function. The names listed in the input function will become defined as if they had appeared in a "name=" definition statement. The names will be positionally correlated with the data fields in the referenced output file.

With the TRANSFER form, the mesh structure of the imported file is stored independently from the computation mesh, and is not influenced by refinement or merging of the computation mesh.

The TRANSFERMESH input statement not only imports data definitions stored on disk, but also IMPOSES THE FINITE ELEMENT MESH STRUCTURE of the imported file onto the current problem, bypassing the normal mesh generation process. In order for this imposition to work, the importing descriptor file must have EXACTLY the same domain definition structure as the exporting file. Be sure to use a copy of the exporting domain definition in your importing descriptor. You may change the boundary conditions, but not the boundary positions and ordering.

The TRANSFERMESHTIME statement acts precisely as the TRANSFERMESH statement, except that the problem time is imported from the transfer file as well as the mesh. This statement can be used to resume a time-dependent problem from the state recorded in the transfer file.

Note: TRANSFER import does not restore the state of HISTORY plots.

Examples:

Samples | Usage | Import-Export | Transfer_Export.pde 485
Samples | Usage | Import-Export | Transfer_Import.pde 485
Samples | Usage | Import-Export | Mesh_export.pde 479
Samples | Usage | Import-Export | Mesh_import.pde 480
Samples | Usage | Stop+Restart | Restart_export.pde 519
Samples | Usage | Stop+Restart | Restart_import.pde 520

3.3.6.8.1 TRANSFER File format

The format of a TRANSFER file is dictated by the TRANSFER output format, and contains the following data.

The Header Section

1) A header containing an identifying section listing the FlexPDE version, generating problem name and run time, and plotted variable name or function equation. This header is enclosed in comment brackets, { ... }.

2) A file identifier "FlexPDE transfer file", and the problem title.

3) The number of geometric dimensions and their names.

4) The finite element basis identifier from 4 to 10, meaning:

- 4 = linear triangle (3 points per cell)
- 5 = quadratic triangle (6 points per cell)
- 6 = cubic triangle (9 points per cell)
- 7 = cubic triangle (10 points per cell)
- 8 = linear tetrahedron (4 points per cell)
- 9 = quadratic tetrahedron (10 points per cell)
- 10 = cubic tetrahedron (20 points per cell)

5) The number of degrees of freedom (points per cell as above).

6) Current problem time and timestep (time-dependent problems only).

7) The number of output variables and their names

8) The number of domain joints (boundary break points) and their descriptions, including
- Joint number
- Periodic image joint (or 0)
- Associated global node number
- Extrusion surface (or 0)
- Active flag

9) The number of domain edges and their descriptions, including
- Edge number
- Associated base plane edge number
- Beginning joint number
- Ending joint number
- Periodic image edge (or 0)
- Extrusion surface (or 0)
- Extrusion layer (or 0)
- Active, Feature and Contact flags
- Edge name

10) The number of 3D domain faces and their descriptions, including
- Face number
- Associated base plane face number
- Left adjoining Region number
- Right adjoining Region number
- Periodic image face (or 0)
- Shape selector
- Layer or surface number
- Active and Contact flags
- Face name

11) The number of domain regions and their descriptions, including
- Region number
- Associated base plane region number
- Layer (or 0)
- Material number
- Active flag
- Region name

The Data Section

Each distinct material type in the exported problem is represented by a separate section in the TRANSFER file. Material types are defined by matching parameter definitions. Each data section consists of:

1) The number of nodes

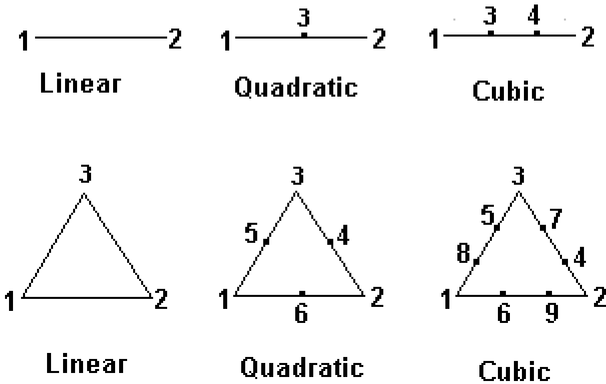
2) The nodal data, containing one line for each node with the following format:
- two or three coordinates and as many data values as specified in (7).
- a colon (:)

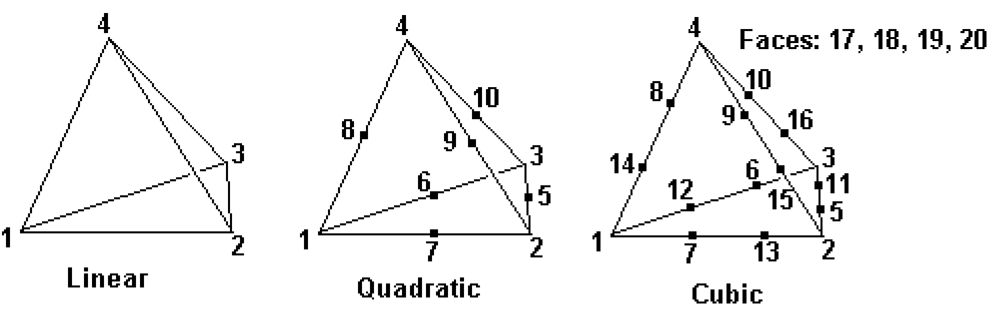
- the global node index
- the node type (0=interior; 1=joint; 2=edge; 3=face; 4=exterior)
- the type qualifier (region number, joint number, edge number or face number)
- the periodic node index

3) The number of cells.

4) The cell connectivity data, one line per cell, listing the following:
- the geometric basis (as in Header 4)
- the node numbers (local to the current material block) which comprise the cell. The count of these node numbers is controlled by (Header 5).
- a colon (:)
- the global cell number
- the logical region number
- the material number

The node numbers are presented in the following order:





3.3.6.9 The PASSIVE Modifier

Definitions may be specified as **PASSIVE**, in which case they will be blocked from differentiation with respect to system variables in the formation of the global Jacobian matrix. In strongly nonlinear systems, this sometimes prevents pathological behavior, at the expense of slower convergence.

Example:

```
Viscosity = Passive(3.02*exp(-5*Temp))
```

The derivative of Viscosity with respect to Temp will be forced to zero, instead of the true value (-5)* 3.02*exp(-5*Temp).

3.3.6.10 Mesh Control Parameters

FlexPDE uses an adaptive initial mesh generation procedure. Cell sizes are generated to conform with local boundary feature sizes, and cell sizes will grow gradually from locales of small cell size to locales of large cell size. Cells sides always match everywhere, and there is never a mismatch between adjacent cells.

It is possible, however, to override the default cell size logic by use of the controls MESH_SPACING and MESH_DENSITY. These parameters have special meaning in controlling the initial mesh layout. They may appear in the context of a parameter definition or redefinition (ie, in the DEFINITIONS section or in a REGION), or in the context of a boundary condition. There may be more than one control active in any locale, and the control (default or explicit) resulting in the smallest mesh cells will dominate.

MESH_SPACING dictates the desired spacing between mesh nodes.

MESH_DENSITY is the reciprocal of MESH_SPACING, and dictates the desired number of mesh nodes per unit distance.

Appearing in the DEFINITIONS section, these parameters specify a global default mesh density function in the volume of the domain.

Appearing in a REGION, these parameters specify a mesh density function in the volume of the current region (in 3D they may be qualified by LAYER or SURFACE).

Appearing in the context of a boundary condition (ie, inside a *path*) they dictate the mesh density along the curve or sidewall surface currently being defined. In 3D they may be qualified by LAYER or SURFACE to restrict the application of the density function.

MESH_SPACING and MESH_DENSITY specifications may be any function of spatial coordinates (but not of VARIABLES).

Examples:

```
MESH_DENSITY = exp(-(x^2+y^2+z^2)
```

This will create a Gaussian density distribution around (0,0,0), with spacing ultimately overridden by the size limit implied by NGRID.

See the User Guide section "Controlling Mesh Density 103" for more information.
See also
"Samples | Usage | Mesh_Control | Mesh_Density.pde" 490
"Samples | Usage | Mesh_Control | Mesh_Spacing.pde" 490
"Samples | Usage | Mesh_Control | Boundary_Density.pde" 488
"Samples | Usage | Mesh_Control | Boundary_Spacing.pde" 488

3.3.7 Initial Values

The **INITIAL VALUES** section is used to initialize the dependent variables.

When not specifically initialized, the dependent variables are initialized to zero.

For steady state problems the INITIAL VALUES section is optional.

For time dependent problems, the INITIAL VALUES section should include a value assignment statement for each dependent variable.

Initial value statements are formed by following the dependent variable name with the assignment operator '=' and either a constant, function, expression or previously defined definition.

Example:

```
INITIAL VALUES
   U = 1.0-x
```

Setting Initial Values from an imported table:

For syntactic reasons, initial values cannot be set directly from TABLE 165 or TRANSFER 169.
An intermediate name must be defined by the TABLE or TRANSFER command, and then assigned to the initial value:

```
DEFINITIONS
   TRANSFER("initial_U.xfr",U0)
INITIAL VALUES
   U = U0
```

3.3.8 Equations

The **EQUATIONS** section is used to list the partial differential equations that define the dependent variables of the problem.

There must be one equation for each dependent variable listed in the VARIABLES and GLOBAL VARIABLES sections.

Each equation must be prefixed by variable_name: in order to associate the equation with a variable and with boundary condition declarations. (If there is only a single equation, the prefix may be omitted.)

Equations are entered into a problem descriptor in much the same way as they are written on paper. In their simplest form they can be written using the DIV (divergence), GRAD (gradient), CURL and DEL2 (Laplacian) operators. FlexPDE will correctly expand these operators in the coordinate system specified in the COORDINATES section.

When it is necessary to enter partial differential terms, differential operators of the form D<name> or D<name1><name2> may be used. Here <name> represents a coordinate name, such as X, Y or Z (or other names chosen by the user in the COORDINATES section).

In the default 2D Cartesian geometry, the operators DX, DY, DXX, DXY, DYX and DYY are defined.

Similarly, in the default cylindrical geometries (XCYLINDER and YCYLINDER), the operators DR, DZ, DRR, DRZ, DZR and DZZ are defined.

In 3D Cartesian geometry, the operators DZ, DZZ, DXZ, and DYZ are also defined.

Example:

```
EQUATIONS
u:  div(k*grad(u)) + u*dx(u) = 0
```

Complex and Vector Variables

Equations can be written using COMPLEX or VECTOR variables. In each case, FlexPDE will expand the stated equation into the appropriate number of scalar equations for computing the components of the COMPLEX or VECTOR variable.

Example:

```
VARIABLES
     U = COMPLEX(Ur,Ui)
EQUATIONS
U:   DIV(k*GRAD(U)) + COMPLEX(Ui,Ur) = 0
```

Third Order and Higher Order Derivatives

Equation definitions may contain spatial derivatives of only first or second order. Problems such as the biharmonic equation which require the use of higher order derivatives must be rewritten using an intermediate variable and equation so that each equation contains only first or second order derivatives.

3.3.8.1 Association between Equations, Variables and Boundary Conditions

In problems with a single variable, there is no ambiguity about the assignment of boundary conditions to the equations.

In problems with more than one variable, FlexPDE requires that equations be explicitly associated with variables by tagging each equation with a variable name. This process also allows optimal ordering of the equations in the coupling matrix.

Example:

```
U:   div(k*grad(u))+u*dx(u)= 0 { associates this equation with the variable U }
```

Boundary conditions are defined in the BOUNDARIES 180 section, and are associated with equations by use of the variable name, which selects an equation through the association tag. VALUE(U)=0, for example, will cause the nodal equations for the equation tagged U: to be replaced by the equation u=0 along the selected boundary .

Natural boundary conditions must be written with a sign corresponding to the sign of the generating terms when they are moved to the left side of the equal sign. *We suggest that all second-order terms should be written on the left of the equal sign, to avoid confusion regarding the sign of the applied natural boundary condition.*

3.3.8.2 Sequencing of Equations

New in version 6 is the ability to sequence sets of equations.

The sets are defined using the THEN and FINALLY sections following the EQUATIONS section.

```
EQUATIONS
   <set A>
THEN
   <set B>
{ THEN
   <set C> ... }
{ FINALLY
   <set D> }
```

Any number of THEN equation sets may be designated and these sets along with the main EQUATIONS section will be run sequentially and repetitively (including regrids) until the solution meets the normal error criteria. Once the EQUATIONS and THEN sets are finished, the last set defined in the FINALLY section will be solved.

Each set of equations is solved for the variables defined by the equations of that set, with the other variables held constant at their current values. Solutions of the EQUATIONS set will be held constant during the solution of the first THEN set, etc.

Each VARIABLE may be defined only once in the complete list of equations.

In time-dependent problems, the full set of equations is solved once during each timestep. The FINALLY clause is ignored in time-dependent problems.

> ***Note**: This facility finds its greatest utility in steady-state problems and time-dependent problems with one-way coupling. In time-dependent problems with two-way coupling, use of sequenced equations may falsify propagation speeds, or lead to instability.*

Example:

```
EQUATIONS
  u:  div(grad(u)) + s = 0
THEN
  v:  div(grad(v)) + u = 0
```

Examples:

Samples | Usage | Sequenced_Equations | Theneq.pde
Samples | Usage | Sequenced_Equations | Theneq+time.pde

3.3.8.3 Modal Analysis and Associated Equations

When modal analysis is desired, it must be declared in the SELECT section with the selector

```
MODES = integer
```

where integer is the number of modes to be analyzed.

The equation should then be written in the form

```
F(V) +LAMBDA*G(V) = H(X,Y)
```

Where F(V) and G(V) are the appropriate terms containing the dependent variable, and H(X,Y) is a driving source term.

The name LAMBDA is automatically declared by FlexPDE to mean the eigenvalue, and should not be declared in the DEFINITIONS section.

3.3.8.4 Moving Meshes

FlexPDE can support moving computation meshes in time-dependent problems. Use of this capability requires:

- The assignment of a surrogate variable 100 for each coordinate to be moved
- Definition of an EQUATION of motion for each such surrogate coordinate
- Suitable Boundary Conditions on the surrogate coordinate.

In some problems, the mesh positions may be driven directly. In others, there will be a variable defining the mesh velocity. This may be the same as the fluid velocity, in which case the model is purely Lagrangian, or it may be some other better-behaved motion, in which case the model is mixed Lagrange/Eulerian (ALE).

FlexPDE 6 contains no provisions for re-connecting distorted meshes. Except in well-behaved problems, pure Lagrangian computations are therefore discouraged, as severe mesh corruption may result.

Alternative Declaration Forms

EQUATIONS are always assumed to refer to the stationary Eulerian (Laboratory) reference frame. FlexPDE automatically computes the required correction terms for mesh motion. .

Alternatively, the user can declare LAGRANGIAN EQUATIONS, and FlexPDE will not modify the user's stated equations. In this case, the equations must be written correctly for the values at the moving nodes.

The declaration EULERIAN EQUATIONS can also be used for clarity, although this is equivalent to the default EQUATIONS declaration.

Internal Mesh Redistribution

When the mesh is not tied directly to a fluid velocity, a convenient technique for maintaining mesh integrity is to diffuse either the mesh coordinates or the mesh velocities in the problem interior.

For direct coordinate diffusion, we apply the diffusion equation to the surrogate coordinates:

```
DIV(GRAD(x_surrogate)) = 0
```

and apply the motion conditions to the coordinate boundary conditions with either VALUE or VELOCITY conditions:

```
VELOCITY(x_surrogate) = x_velocity
```
or
```
VALUE(x_surrogate) = moving_positions
```

If the mesh is driven by a mesh velocity variable, we apply the diffusion equation to the velocity variables:

```
DIV(GRAD(x_velocity_variable)) = 0
DT(x_coordinate) = x_velocity_variable
```

At the boundaries, we apply the driving motions to the velocity variables and lock the surrogate coordinate variable to its associated velocity

```
VALUE(x_velocity_variable) = x_velocity
VELOCITY(x_surrogate) = x_velocity
```

***Note**: See the User Guide section on Moving Meshes* 100 *and the example problems in the "Samples | Moving_Mesh" folder.*

Effect of Mesh Motion on EQUATION Specifications

EQUATIONS are always written in the Eulerian (Laboratory) reference frame, regardless of whether the mesh moves or not. FlexPDE automatically computes the required correction terms for mesh motion.

3.3.9 Constraints

The **CONSTRAINTS** section, which is optional, is used to apply integral constraints to the system. These constraints can be used to eliminate ambiguities that would otherwise occur in steady state systems, such as mechanical and chemical reaction systems, or when only derivative boundary conditions are specified.

The CONSTRAINTS section, when used, normally contains one or more statements of the form

INTEGRAL (argument) = expression

CONSTRAINTS should not be used with steady state systems which are unambiguously defined by their boundary conditions, or in time-dependent systems.

A CONSTRAINT creates a new auxiliary functional which is minimized during the solution process. If there is a conflict between the requirements of the CONSTRAINT and those of the PDE system or boundary conditions, then the final solution will be a *compromise* between these requirements, and may not strictly satisfy either one.

CONSTRAINTS can be applied to any of the INTEGRAL operators 136.

CONSTRAINTS *cannot* be used to enforce local requirements, such as positivity, to nodal variables.

Examples:

Samples | Usage | Constraints | Constraint.pde 454
Samples | Usage | Constraints | Boundary_Constraint.pde 454
Samples | Usage | Constraints | 3D_Constraint.pde 451
Samples | Usage | Constraints | 3D_Surf_Constraint.pde 453
Samples | Applications | Chemistry | Reaction.pde 291

3.3.10 Extrusion

The layer structure of a three-dimensional problem is specified bottom-up to FlexPDE in the EXTRUSION Section:

```
EXTRUSION
  SURFACE   "Surface_name_1"      Z = expression_1
    LAYER   "Layer_name_1"
  SURFACE   "Surface_name_2"      Z = expression_2
    LAYER   "Layer_name_2"
  . . .
  SURFACE   "Surface_name_n"      Z = expression_n
```

The specification must start with a SURFACE and end with a SURFACE.

LAYERS correspond to the space between the SURFACES.

The Layer_names and Surface_names in these specifications are optional. The LAYER specifications may be omitted if a name is not needed to refer to them.

- Surfaces need not be planar, and they may merge, but they must not cross. expression_1 is assumed to be everywhere less than or equal to expression_2, and so on. Use a MIN or MAX function when there is a possibility of crossover.
- Surface expressions can refer to regionally defined parameters, so that the surface takes on different definitions in different regions. The disjoint expressions must, however, be continuous across region interfaces. (see example "Samples | Usage | 3d_Domains | Regional_surfaces.pde" 432)
- If surface expressions contain conditional values (IF...THEN or MIN, MAX, etc), then the base plane domain should include FEATURES to delineate the breaks, so they can be resolved by the gridder.
- Surfaces must be everywhere continuous, including across material interfaces. Use of conditionals or regional definitions must guarantee surface continuity.
- Surface expressions can refer to tabular input data (see example "Samples | Usage | 3D_Domains | Tabular_surfaces.pde" 433).

See the User Guide chapter Using FlexPDE in Three-Dimensional Problems 69 for more information on 3D extrusions.

Shorthand form

Stripped of labels, the EXTRUSION specification may be written:

```
EXTRUSION Z = expression_1, expression_2 {, ...}
```

In this form layers and surfaces must subsequently be referred to by numbers, with surface numbers running from 1 to n and layer numbers from 1 to (n-1). SURFACE #1 is Z=expression_1, and LAYER #1 is between SURFACE #1 and SURFACE #2.

Built-In Surface Generators

FlexPDE version 6 defines three surface generation functions

PLANE (point1 , point2 , point3)	Defines a plane surface containing the three stated points.

CYLINDER (point1 , point2 , radius)	Defines the top surface of a cylinder with axis along the line from **point1** to **point2** and with the given radius (see note below). **point1** and **point2** must be at the same z coordinate. Z-Tilted cylinders are not supported.
SPHERE (point , radius)	Defines the top surface of a sphere of the given **radius** with center at the specified center **point** (see note below).

Each point specification is a parenthesized coordinate double (xn , yn) or triple (xn , yn , zn). If zn is omitted, it is assumed zero.

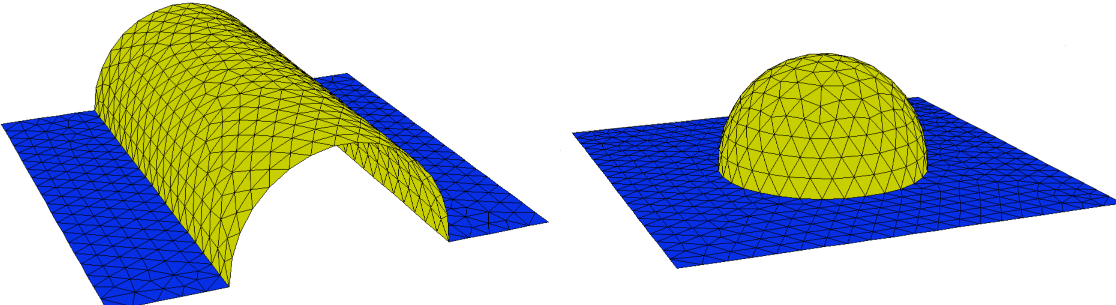
These functions can be used to simplify the layout of extrusion surfaces.

CYLINDER and SPHERE construct the top surface of the specified figure (see note below). To generate both the upper and lower halves of the CYLINDER and SPHERE, simply construct the figure at Z=0 and add and subtract the surface function from the desired Z coordinate of the center or axis.

Example:

```
DEFINITIONS
  Zsphere = SPHERE((0,0,0), 10)
EXTRUSION
  Zcenter-Zsphere, Zcenter+Zsphere
```

***Note**: These functions generate surfaces defined throughout X,Y space.* CYLINDER *and* SPHERE *include Z=constant skirts to extend the surface definitions. The diameters of the* CYLINDER *and* SPHERE*, as well as the extent of the* CYLINDER *along its axis and of the* PLANE *must be provided by* REGION BOUNDARIES *or* FEATURES.



3.3.11 Boundaries

The **BOUNDARIES** section is used to describe the problem domain over which the specified equation system is to be solved, and to specify boundary conditions along the outer surfaces of this domain.

Because of the history of FlexPDE, the discussion of boundaries has a strong two-dimensional orientation. Three-dimensional figures are made up by extruding a two-dimensional domain into the third dimension. One-dimensional domains are constructed by specializations of 2D techniques.

Every problem descriptor must have a BOUNDARIES section.

Problem BOUNDARIES are made up by walking the periphery of each material region on boundary paths through a 2D Cartesian space.

In this way, the physical domain is broken down into REGION, FEATURE and EXCLUDE subsections.

Every problem descriptor must have at least one REGION subsection. FEATURE and EXLUDE subsections are optional.

For concrete examples of the constructs described here, refer to the sample problems distributed with the FlexPDE software.

3.3.11.1 Points

The fundamental unit used in building problem domains is the geometric POINT. POINTS in a FlexPDE script are expressed as a parenthesized list of coordinate values, as in the two dimensional point (2.4, 3.72).

Since two- and three- dimensional domain definitions both begin with a two-dimensional layout, the use for three-dimensional points is generally limited to ELEVATION PLOTS.

In one-dimensional systems, a POINT degenerates to a single parenthesized coordinate, such as (2.4).

3.3.11.2 Boundary Paths

A boundary path has the general form

START(a,b) segment TO (c,d) ...

where (a,b) and (c,d) are the physical coordinates of the ends of the segment, and segment is either LINE, SPLINE or ARC.

The path continues with a connected series of segments, each of which moves the segment to a new point. The end point of one segment becomes the start point of the next segment.

A path ends whenever the next input item cannot be construed as a segment, or when it is closed by returning to the start point. The closing segment may simply end at the start point, or it can explicitly reference CLOSE, which will cause the current path to be continued to meet the starting point:

... segment TO CLOSE.

or

... segment CLOSE.

__Line Segments__

Line segments take the form

LINE TO (x,y)

When successive LINE segments are used, the reserved word LINE does not have to be repeated, as in the following:

```
LINE TO (x1,y1) TO (x2,y2) TO (x3,y3) TO ...
```

Spline Segments

Spline segments are syntactically similar to Line segments

SPLINE TO (x,y) TO (x2,y2) TO (x3,y3) TO ...

A cubic spline will be fit to the listed points. The first point of the spline will be either the START point or the ending point of the previous segment. The last point of the spline will be the last point stated in the chain of TO(,) points.
The fitted spline will have zero curvature at the end points, so it is a good idea to begin and end with closely spaced points to establish the proper endpoint directions.

Arc Segments

Arc segments create either circular or elliptical arcs, and take one of the following the forms:

ARC TO (x1,y1) to (x2,y2)
ARC (RADIUS = R) to (x,y)
ARC (CENTER = x1,y1) to (x2,y2)
ARC (CENTER = x1,y1) ANGLE=angle

Here angle is an angle *measured in degrees*, and follows the convention that positive angles rotate counter-clockwise and negative angles rotate clockwise. The coordinate point at the end of the arc is determined by the radius swept out by the angle. To specify the angle in radians, follow the radian value by the qualifier RADIANS.

Elliptical Segments

When the form ARC (CENTER = x1,y1) to (x2,y2) is used and the center (x1,y1) is not equidistant from the start and end points, an elliptical arc segment is generated with major and minor axes along the X and Y coordinate directions.

The orientation of the major and minor axes can be rotated with the **ROTATE** qualifier.
ARC (CENTER = x1,y1 ROTATE = 30) TO (x2,y2)
The rotation angle is defined in degrees unless followed by the qualifier RADIANS.

The end point is not rotated by this command, and must be stated correctly to intercept the rotated ellipse.

Named Paths

Names can be assigned to paths. When names are assigned to paths they take the form of a quoted string and must be placed immediately after the reserved word START:

START "namedpath" (<x> , <y>)

Assigned path names are useful when boundary or line-related integrals are desired or for establishing paths over which ELEVATION plots are desired.

Names can be assigned to portions of a path by entering a new START clause, or by overlaying a portion of the boundary path by an independently declared FEATURE 187.

Paths Defined by ARRAYS and MATRICES

Paths may be defined by ARRAYS or MATRICES.

In the case of ARRAYS, two arrays of equal dimension are used to specify the coordinates in a LIST

boundary:

LINE LIST(Ax,Ay)
SPLINE LIST(Ax,Ay)

Here Ax and Ay are ARRAYS listing the X- and Y- coordinates of the path.

A 2-by-N MATRIX may also be used to specify a LINE or SPLINE LIST, with the syntax:

LINE LIST(Mxy)
SPLINE LIST(Mxy)

Examples:
Samples|Usage|Arrays+Matrices|Array_Boundary.pde 446
Samples|Usage|Arrays+Matrices|Matrix_Boundary.pde 448

3.3.11.3 Regions

A **REGION** is a portion of a two-dimensional problem domain (or of the projection of a 3D problem domain), bounded by *boundary paths* 181, that encloses an area and contains a single material (but see Regions in One Dimension 184 for exceptions).

Each material property in the REGION has a single definition, although this definition may be arbitrarily elaborate.

A REGION may consist of many disjoint areas.

Example:

```
REGION 1        { an outer box }
  START(0,0)
  LINE TO (10,0) TO (10,10) TO (0,10) TO CLOSE

REGION 2        { two embedded boxes }
  START(1,1)
  LINE TO (2,1) TO (2,2) TO (1,2) TO CLOSE
  START(5,5)
  LINE TO (6,5) TO (6,6) TO (5,6) TO CLOSE
```

Overlaying regions:

RULE:
REGIONS DEFINED LATER OVERLAY AND OBSCURE REGIONS DEFINED EARLIER.
AREAS COMMON TO TWO REGIONS BECOME PART OF THE LATER DEFINED REGION.

So, in the example above, the two smaller boxes overlay the large box. The material parameters assigned to the large box pertain only to the part of the large box not overlaid by the small boxes.

It is customary to make the first region define the entire outer boundary of the problem domain, and then to overlay the parts of the domain which differ in parameters from this default region. If you overlay all parts of the outer domain with subregions, then the outer region definition becomes invisible. It may be useful to do this in some cases, since it allows a localization of boundary condition specifications. Nevertheless, one of the subregions is superfluous, because it could be the default.

3.3.11.3.1 Reassigning Regional Parameters

Names previously defined in the DEFINITIONS section can be assigned a new value within a REGION by adding one or more assignments of the form

```
name = new_expression
```

immediately following the reserved word REGION.

When definitions are reassigned new values in this manner, the new value applies only to the region in which the reassignment occurs.

Example:

```
DEFINITIONS
  K = 1     { the default value }
REGION 1  { assumes default, since no override is given }
  START(0,0) LINE TO (10,0) TO (10,10) TO (0,10) TO CLOSE
REGION 2
  K = 2     { both sub-boxes are assigned K=2 }
  START(1,1) LINE TO (2,1) TO (2,2) TO (1,2) TO CLOSE
  START(5,5) LINE TO (6,5) TO (6,6) TO (5,6) TO CLOSE
REGION 3  { again assumes the default }
  START(3,3) LINE TO (4,3) TO (4,4) TO (3,4) TO CLOSE
```

3.3.11.3.2 Regions in One Dimension

In one-dimensional domains, the concept that a REGION bounds a finite area by closing on itself is no longer true. In one dimension, it is sufficient to define a path from the start of a material region to its finish. (Referencing CLOSE in a 1D bounding path will cause serious troubles, because the path will retrace itself.)

For example, the statements

```
REGION 1
  START(0) LINE TO (5)
```

are sufficient to define a region of material extending from location 0 to location 5 in the 1D coordinate system.

In order to maintain grammatical consistency with two- and three- dimensional constructs, omitting the parentheses is *not* permitted.

Other general characteristics of REGIONS remain in force in one-dimensional domains:
Later REGIONS overlay earlier REGIONS, material properties are defined following the REGION keyword, and so forth.

3.3.11.3.3 Regions in Three Dimensions

The concept of a REGION in 3D domains retains the same character as for 2D domains.

The REGION is a partition of the 2D projection of the figure, and is extruded into the third dimension according to the EXTRUSION specification.

A material **compartment** in 3D is uniquely defined by the REGION of the projection which bounds it, and the LAYER of the extrusion in which it resides.

Extrusion of each 2D REGION therefore creates a stack of layers above it, each with possibly unique material properties.

A question then arises as to when a component that exists in a given layer of the domain must be divided into multiple regions. The rule can be stated as follows:

Rule: When two points in the projection plane see different stacks of materials above them in the extrusion direction, then these two points must reside in different REGIONS of the domain layout.

In the presence of **LIMITED REGIONS** 186, the above rule can be interpreted to consider only the two layers adjoining a given extrusion surface. If the materials above and below the surface differ between two points, then there must be a REGION boundary separating the two points *in the subject extrusion surface.* REGION boundaries are induced in surfaces by the presence of a REGION boundary in either adjoining LAYER (subject to the overlay rule 183).

See the User Guide chapter Using FlexPDE in Three-Dimensional Problems 69 for further discussion of the construction of 3D domains.

3.3.11.3.4 Regional Parameter Values in 3D

In three-dimensional problems, a redefinition of a parameter inside a REGION causes the parameter to be redefined in all layers of the layer stack above the region. To cause the parameter to be redefined only in a selected layer, use the LAYER qualifier, as in

LAYER number name = new_expression
LAYER "layer_name" name = new_expression

The LAYER qualifier acts on all subsequent parameter redefinitions, until a new LAYER qualifier or a functionally distinct clause breaks the group of redefinitions.

Example:

The following descriptor fragment shows the redefinition of a parameter K in various contexts:

```
DEFINITIONS
  K=1                        { defines the default value }

BOUNDARIES
  LAYER 1 K=2 { (valid only in 3D) defines the value in layer 1 of all regions }
  REGION 1
    K=3          { redefines the value in region 1 only, in all layers of a 3D domain
}
    LAYER 2 K=4           { (valid only in 3D) defines the value in layer 2 of region
1 only }
    START(0,0) LINE TO ....
```

3.3.11.3.5 Limited Regions in 3D

In three dimensional problems, many figures to not fit readily into the extrusion model. In particular, there are frequently features that in reality exist only at very restricted positions in the extrusion dimension, and which create poor meshes when extruded throughout the domain.

FlexPDE implements the concept of **LIMITED REGIONS** to accommodate this situation.

A LIMITED REGION is defined as one that is considered to exist only in specified layers or surfaces of the domain, and is absent in all other layers and surfaces.

The LIMITED REGION will be constructed only in layers and surfaces specifically stated in the body of the REGION definition.

An example of this type of structure might be a transistor, where the junction structure of the device is present only in a very thin layer of the domain, while the substrate occupies the majority of the volume.

In earlier versions of FlexPDE, the shape of the junction structure was propagated and meshed throughout the extrusion dimension. Since version 4, the structure can be restricted, or LIMITED, to a single layer or a few layers.

For example, the following descriptor fragment defines a 3-unit cube with a 0.2-unit cubical structure in the center. The small structure is present in the layer 2 mesh only.

```
EXTRUSION  Z=0, 1.4, 1.6, 3
BOUNDARIES
   REGION 1
      START(0,0) LINE TO (3,0) TO (3,3) TO (3,0) TO CLOSE
   LIMITED REGION 2
      LAYER 2 K=9
      START(1.4,1.4)
      LINE TO (1.6,1.4) TO (1.6,1.6) TO (1.4,1.4) TO CLOSE
```

See the User Guide section "Limited Regions 74" for a graphical example of this facility.

Examples:

Samples | Usage | 3D_Domains | 3D_Limited_Region.pde 414

3.3.11.3.6 Empty Layers in 3D

In three dimensional problems, it is sometimes necessary to define holes or excluded regions in the extruded domain. This may be done using the **VOID** qualifier. VOID has the syntax of a parameter redefinition.

For example, the following descriptor fragment defines a 3-unit cube with a 1-unit cubical hole in the center:

```
EXTRUSION  Z=0,1,2,3
```

```
BOUNDARIES
  REGION 1
    START(0,0) LINE TO (3,0) TO (3,3) TO (3,0) TO CLOSE
  REGION 2
    LAYER 2 VOID
    START(1,1) LINE TO (2,1) TO (2,2) TO (1,2) TO CLOSE
```

Examples:

Samples | Usage | 3D_Domains | 3D_Void.pde 431

3.3.11.4 Excludes

EXCLUDE subsections are used to describe closed domains which overlay parts of one or more REGION subsections. The domain described by an exclude subsection is excluded from the system. EXCLUDE subsections must follow the REGION subsections which they overlay

EXCLUDE subsections are formed in the same manner as REGION subsections and can use all the same LINE and ARC segments.

3.3.11.5 Features

FEATURE subsections are used to describe non-closed entities which do not enclose a subdomain with definable material parameters.

FEATURE subsections are formed in the same manner as REGION 183 subsections and can use all the same LINE and ARC segments.

FEATURE subsections do not end with the reserve word CLOSE.

A FEATURE will be explicitly represented by nodes and cell sides.

FEATURE subsections are used when a problem has internal line sources; when it is desirable to calculate integrals along an irregular path; or when explicit control of the grid is required.

In 3D problems, FEATURES should be used to delineate any sharp breaks in the slope of extrusion surfaces. Unless mesh lines lie along the surface breaks, the surface modeling will be crude.

Example:

```
REGION 1        { an outer box }
  START(0,0) LINE TO (10,0) TO (10,10) TO (0,10) TO CLOSE

FEATURE         { with a diagonal gridding line }
  START(0,0) LINE TO (10,10)
```

3.3.11.6 Node Points

FlexPDE supports the ability to place mesh nodes at specific points in the problem geometry. This is done with the statements

NODE POINT (x_value , y_value)
NODE POINT (x_value , y_value , z_value)

A mesh node will be placed at the specified location, and linked into the computation mesh.

NODE POINTS can be used to place POINT VALUE 190 or POINT LOAD 190 boundary conditions (see Caveat 190).

In moving mesh problems, NODE POINTS will move with the mesh; they will not be locked to the specified location unless appropriate POINT VALUE boundary conditions are used to freeze the point.

In 3D geometries, specification of only two coordinates will cause a vertical meshing line to be placed throughout the Z-coordinate range of the domain. A three-coordinate point will specify a single node. Placing NODE POINTS in coincidence with EXTRUSION surfaces will have undefined effects, and may lead to mesh generation failure.

An alternative way of forcing nodes is to run a FEATURE or REGION boundary to and through the desired point.

3.3.11.7 Ordering Regions

While not strictly enforced, it is recommended that all REGION subsections be listed before any EXCLUDE or FEATURE subsections and that all EXCLUDE subsections be listed before any FEATURE subsections.

It is further recommended that the first REGION subsection be formed by walking the outside boundary of the problem thereby enclosing the entire domain of the problem.

Rule:

REGIONS *defined later are assumed to overlay any previously listed* REGIONs*, and any properties assigned to a* REGION *will override properties previously assigned to the domains they overlay.*

Regions in 3D Domains

In 3D domains, the above rule is applied in each extrusion surface.

3.3.11.8 Numbering Regions

REGION,EXCLUDE and FEATURE subsections can be assigned numbers and/or names.

When numbers are assigned they should be in ascending sequential order beginning with one. It is recommended that numbers always be assigned.

When names are assigned they must take the form of a quoted string and must be placed immediately after either the reserved word REGION, EXCLUDE, or FEATURE or any number assigned to the REGION, EXCLUDE, or FEATURE. Assigned names must be unique to the REGION, EXCLUDE or FEATURE that they name.

Assigned region names are useful when region-restricted plots or volume integrals are desired.

Example:

```
REGION 2 'Thing'
```

```
  {...}
PLOTS
  contour(u) on 'Thing'
```

3.3.11.9 Fillets and Bevels

Any point in a path may be followed by one of the specifications

FILLET(radius)
BEVEL(length)

The point will be replaced by a circular arc of the specified radius, or by a bevel of the specified length. FILLETS and BEVELS should not be applied to points which are the intersection of several segments, or confusion may ensue.

Example:

```
LINE TO (1,1)  FILLET(0.01)
```

Example problem:

"Samples | Usage | Fillet.pde 381"

3.3.11.10 Boundary Conditions

The following forms of boundary condition specification may be applied to boundary segments:

VALUE (variable) = expression
NATURAL (variable) = expression
LOAD (variable) = expression
CONTACT (variable) = expression
VELOCITY (variable) = expression
NOBC (variable)

The variable designated in the boundary condition specification identifies (by explicit association) the equation to which this boundary condition is to be applied.

Dirichlet (Value) Boundary Conditions

A **VALUE** segment boundary condition forces the solution of the equation for the associated variable to the value of expression on a continuous series of one or more boundary segments. The expression may be an explicit specification of value, involving only constants and coordinates, or it may be an implicit relation involving values and derivatives of system variables.

Generalized Flux (Natural) Boundary Conditions

NATURAL and **LOAD** segment boundary conditions are synonymous. They represent a generalized flux boundary condition derived from the divergence theorem. The expression may be an explicit specification, involving only constants and coordinates, or it may be an implicit relation involving values and derivatives of system variables. The Natural boundary condition reduces to the Neumann boundary condition in the special case of the Poisson equation. See the User Guide chapter Natural Boundary Conditions 61 for information on the implementation of Natural boundary conditions.

Contact Resistance (Discontinuous Variable) Boundary Conditions

Interior boundaries can be defined to have a contact resistance using the **CONTACT(variable)**

boundary condition. See "Jump Boundaries 192" in the next section.

Velocity (Time Derivative) Boundary Conditions

This boundary condition imposes a specified time derivative on a boundary value (time-dependent problems only). This condition is especially useful in specifying moving boundaries, by applying it to the surrogate coordinate variable. If you have declared a velocity variable which is applied to a coordinate, then you should lock the surrogate coordinate to the mesh velocity variable at the boundary using a **VELOCITY()** boundary condition.

Terminating the current BC

Boundary conditions, once stated, remain in effect until explicitly changed or until the end of the path. NOBC(VARIABLE) can be used to turn off a previously specified boundary condition on the current path. It is equivalent in effect to NATURAL(VARIABLE)=0 (the default boundary condition), except that it will not lead to "Multiple Boundary Condition Specification" diagnostics.

Default Boundary Conditions

The default boundary condition for FlexPDE is NATURAL(VARIABLE)=0.

***Note**: The NEUMANN, DNORMAL and DTANGENTIAL boundary conditions supported in earlier versions have been deleted due to unreliable behavior. They may be restored in later versions. In most cases, derivative boundary conditions are more appropriately applied through the NATURAL boundary condition facility.*

3.3.11.10.1 Syntax of Boundary Condition Statements

Segment boundary conditions are added to the problem descriptor by placing them in the BOUNDARIES section.

Segment boundary conditions must immediately precede one of the reserved words LINE or ARC and cannot precede the reserved word TO.

A top-down system is used for applying segment boundary conditions to the equations. Following the START point specification in each path definition, a segment boundary condition is set up for each variable/equation. It is recommended that a boundary condition be specified for each variable/equation. If no other segment boundary condition is specified no error will occur and a NATURAL(VARIABLE) = 0 segment boundary condition is assumed.

Under the top-down system, as boundary segments occur, the previously specified segment boundary condition for a variable will continue to hold until a new boundary condition is specified for that variable.

If the recommendation is followed that REGION 1 be formed by walking the outside boundary of the problem, thereby enclosing the entire domain of the problem, then for most problems segment boundary conditions need only be specified for the segments in REGION 1.

3.3.11.10.2 Point Boundary Conditions

POINT VALUE boundary conditions can be added by placing

POINT VALUE (variable) = expression

following a coordinate specification. The stated value will be imposed at the coordinate point immediately

preceding the specification.

POINT LOAD boundary conditions can be added by placing

POINT LOAD (variable) = expression

following a coordinate specification. The stated load will be imposed as a lumped source on the coordinate point immediately preceding the specification.

A Caveat:

The results achieved by use of these specifications are frequently disappointing.

A diffusion equation, for example, div(grad(u))+s=0, can support solutions of the form u=A-Br-Cr^2, where r is the distance from the point value and A, B and C are arbitrary constants. By the superposition principle, FlexPDE is free to add such shapes to the computed solution in the vicinity of the point value, without violating the PDE. A POINT VALUE condition usually leads to a sharp spike in the solution, pulling the value up to that specified, but otherwise leaving the solution unmodified.

The POINT LOAD is not subject to this same argument, but since it is a load without scale, it will frequently produce a dense mesh refinement around the point.

A better solution is to use a distributed load or an extended value boundary segment, ring or box.

3.3.11.10.3 Boundary conditions in 1D

The idea that a boundary condition applies along the length of a boundary segment, while meaningful in two and three dimensions, is meaningless in one dimension, since it is the value along the segment that is the object of the computation.

In one dimensional problems, therefore, it is necessary to use the Point boundary condition described in the previous section for all boundary condition specifications.

Example:

```
BOUNDARIES
    REGION 1
        START(0)        POINT VALUE(u)=1
        LINE TO (5)     POINT LOAD(u)=4
```

The node at coordinate 0 will have value 1, while that at coordinate 5 will have a load of 4.

3.3.11.10.4 Boundary Conditions in 3D

In three-dimensional problems, an assignment of a segment boundary condition to a region boundary causes that boundary condition to be applied to the "side walls" of all layers of the layer stack above the region. To selectively apply a boundary condition to the "side walls" of only one layer, use the LAYER qualifier, as in

LAYER number VALUE(variable) = expression
LAYER "layer_name" VALUE(variable) = expression

The LAYER qualifier applies to all subsequent boundary condition specifications until a new LAYER qualifier is encountered, or the segment geometry (LINE or ARC) statements begin.

The boundary conditions on the extrusion surfaces themselves (the slicing surfaces) can be specified by the SURFACE qualifier preceding the boundary condition specification.

Consider a simple cube. The EXTRUSION and BOUNDARIES sections might look like this:

```
EXTRUSION  z = 0,1
BOUNDARIES
  SURFACE 1 VALUE(U)=0               { 1 }
  REGION 1
    SURFACE 2 VALUE(U)=1    { 2 }
    START(0,0)
    NATURAL(U)=0            { 3 }
      LINE TO (1,0)
    LAYER 1 NATURAL(U)=1    { 4 }
      LINE TO (1,1)
    NATURAL(U)=0            { 5 }
      LINE TO (0,1) TO CLOSE
```

Line { 1 } specifies a fixed value of 0 for the variable U over the entire surface 1 (ie. the Z=0 plane).
Line { 2 } specifies a value of 1 for the variable U on the top surface *in* REGION 1 *only*.
Line { 3 } specifies an insulating boundary on the Y=0 side wall of the cube.
Line { 4 } specifies a flux (whose meaning will depend on the PDE) on the X=1 side wall *in* LAYER 1 *only*.
Line { 5 } returns to an insulating boundary on the Y=1 and X=0 side walls.

[Of course, in this example the restriction to region 1 or layer 1 is meaningless, because there is only one of each.]

3.3.11.10.5 Jump Boundaries

In the default case, FlexPDE assumes that all variables are continuous across internal material interfaces. This is a consequence of the positioning of mesh nodes along the interface which are shared by the cells on both sides of the interface.

FlexPDE supports the option of making variables discontinuous at material interfaces (see the "Discontinuous Variables 64" in the User Guide for tutorial information).

This capability can be used to model such things as contact resistance, or to completely decouple the variables in adjacent regions.

The key words in employing this facility are **CONTACT** and **JUMP**.

The conceptual model is that of contact resistance, where the difference in voltage V across the interface (the JUMP) is given by

```
V2 - V1 = R*current
```

In the general case, the role of "current" is played by the generalized flux, or Natural boundary condition 190. (See the User Guide for further discussion of Natural Boundary Conditions 61.) The CONTACT boundary condition is a special form of NATURAL, which defines a flux but also specifies that FlexPDE should model a double-valued boundary.

So the method of specifying a discontinuity is

```
CONTACT(V) = (1/R)*JUMP(V)
```

"CONTACT(V)", like "NATURAL(V)", means the outward normal component of the generalized flux as seen from any cell. So from any cell, the meaning of "JUMP(V)" is the difference between the interior and exterior values of V at a point on the boundary. Two cells sharing a boundary will then see JUMP values and outward normal fluxes of opposite sign. "Flux" is automatically conserved, since the same numeric value is used for the flux in both cells.

Specifying a CONTACT boundary condition at an internal boundary causes duplicate mesh nodes to be generated along the boundary, and to be coupled according to the JUMP boundary condition statement.

Specifying a very small (1/R) value effectively decouples the variable across the interface.

Example Problems:

"Samples | Usage | Discontinuous_Variables | Thermal_Contact_Resistance.pde" 461
"Samples | Usage | Discontinuous_Variables | Contact_Resistance_Heating.pde" 460
"Samples | Usage | Discontinuous_Variables | Transient_Contact_Resistance_Heating.pde" 462

3.3.11.10.6 Periodic Boundaries

FlexPDE supports periodic and antiperiodic boundary conditions in one, two or three dimensions.

Periodicity in the X-Y Plane

Periodicity in a two-dimensional problem, or in the extrusion walls of a three-dimensional problem, is invoked by the PERIODIC or ANTIPERIODIC statement.

The PERIODIC statement appears in the position of a boundary condition, but the syntax is slightly different, and the requirements and implications are more extensive.

The syntax is:

PERIODIC (X_mapping, Y_mapping)
ANTIPERIODIC (X_mapping, Y_mapping)

The mapping expressions specify the arithmetic required to convert a point (X,Y) in the immediate boundary to a point (X',Y') on a remote boundary. The mapping expressions must result in each point on the immediate boundary being mapped to a point on the remote boundary. Segment endpoints must map to segment endpoints. The transformation must be invertible; do not specify constants as mapped coordinates, as this will create a singular transformation.

The periodic boundary statement terminates any boundary conditions in effect, and instead imposes equality of all variables on the two boundaries. It is still possible to state a boundary condition on the remote boundary, but in most cases this would be inappropriate.

The periodic statement affects only the next following LINE or ARC path. These paths may contain more than one segment, but the next appearing LINE or ARC statement terminates the periodic condition unless the periodic statement is repeated.

Periodicity in 1D

Periodicity in a one-dimensional problem is invoked by the POINT PERIODIC or POINT ANTIPERIODIC statement. All other aspects are similar to the description above for X-Y periodicity.

Periodicity in the Z-Dimension

Periodicity In the extruded dimension is invoked by the modifier PERIODIC or ANTIPERIODIC before the EXTRUSION statement, for example,

```
PERIODIC EXTRUSION  Z=0,1,2
```

In this case, the top and bottom extrusion surfaces are assumed to be conformable, and the values are forced equal (or sign-reversed) along these surfaces.

Restrictions

Each node in the finite element mesh can have at most one periodic image. This means that two-way or three-way periodicity cannot be directly implemented. Usually it is sufficient to introduce a small gap in the periodic boundaries, so that each corner is periodic with only one other corner of the mesh.

Example Problems:

"Samples | Usage | Periodicity | periodic.pde" 510
"Samples | Usage | Periodicity | azimuthal_periodic.pde" 508
"Samples | Usage | Periodicity | antiperiodic.pde" 507
"Samples | Usage | Periodicity | 3d_xperiodic.pde" 505
"Samples | Usage | Periodicity | 3d_zperiodic.pde" 507
"Samples | Usage | Periodicity | 3d_antiperiodic.pde" 504

3.3.11.10.7 Complex and Vector Boundary Conditions

Boundary conditions for COMPLEX or VECTOR VARIABLES may be declared for the complex or vector variable directly, or for the individual components.

If C is a COMPLEX VARIABLE with components Cr and Ci, the following boundary condition declarations are equivalent:

```
VALUE(C) = Complex(a,b)
VALUE(Cr) = a    VALUE(Ci) = b
```

If V is a VECTOR VARIABLE with components Vx and Vy, the following boundary condition declarations are equivalent:

```
NATURAL(V) = Vector(a,b)
NATURAL(Vx) = a    NATURAL(Vy) = b
```

The component form allows the application of different boundary condition forms (VALUE or NATURAL) to the components, while the root variable form does not.

3.3.12 Front

The **FRONT** section is used to define additional criteria for use by the adaptive regridder. In the normal case, FlexPDE repeatedly refines the computational mesh until the estimated error in the approximation of the PDE's is less than the declared or default value of ERRLIM. In some cases, where meaningful activity is confined to some kind of a propagating front, it may be desirable to enforce greater refinement near the

front. In the FRONT section, the user may declare the parameters of such a refinement.

The FRONT section has the form:

FRONT (criterion, delta)

The stated criterion will be evaluated at each node of the mesh. Cells will be split if the values at the nodes span a range greater than (-delta/2, delta/2) around zero.

That is, the grid will be forced to resolve the criterion to within delta as it passes through zero.

Example:

Samples | Usage | Mesh_Control | Front.pde 489

3.3.13 Resolve

The **RESOLVE** section is used to define additional criteria for use by the adaptive regridder. In the normal case, FlexPDE repeatedly refines the computational mesh until the estimated error in the approximation of the PDE's is less than the declared or default value of ERRLIM. In some cases, this can be achieved with a much less dense mesh than is necessary to make pleasing graphical presentation of derived quantities, such as derivatives of the system variables, which are much less smooth than the variables themselves. In the RESOLVE section, the user may declare one or more additional functions whose detailed resolution is important. The section has the form:

RESOLVE (spec1) , (spec2) , (spec3) {...}

Here, each spec may be either an expression, such as "(shear_stress)", or an expression followed by a weighting function, as in "(shear_stress, x^2)".

In the simplest form, only the expressions of interest need be presented. In this case, for each stated function, FlexPDE will

- form a Finite Element interpolation of the stated function over the computational mesh
- find the deviation of the interpolation from the exact function
- split any cell where this deviation exceeds ERRLIM times the global RMS value of the function.

Because the finite element interpolation thus formed assumes continuous functions, application of RESOLVE to a discontinuous argument will result in dense gridding at the discontinuity. An exception to this is at **CONTACT** 192 boundaries, where the finite element representation is double valued.

In the weighted form, an importance-weighting function is defined, possibly to restrict the effective domain of resolution. The splitting operation described above is modified to multiply the deviation at each point by the weight function at that point. Areas where the weight is small are therefore subjected to a less stringent accuracy requirement.

Example:

Samples | Usage | Mesh_Control | Resolve.pde 491

3.3.14 Time

The TIME section is used in time dependent problem descriptors to specify a time range over which the problem is to be solved. It supports the following alternative forms:

FROM time1 TO time2
FROM time1 BY increment TO time2
FROM time1 TO time2 BY increment

Where:

time1	is the beginning time
time2	is the ending time.
increment	is an optional specification of the initial time step for the solution. (the default initial time step is 1e-4*(time2-time1)).

All time dependent problem descriptors must include statements which define the time range.
While the problem descriptor language supports alternate methods of specifying a time range, it is recommended that all time dependent problems include the TIME section to specify the total time domain of the problem.

Halting Execution
The time range specification may optionally be followed by a HALT statement:

HALT minimum
HALT = minimum

This statement will cause the computation to halt if the automatically controlled timestep drops below minimum. This facility is useful when inconsistencies in data or discontinuities in parameters cause the timestep controller to become confused.

HALT condition

Here the condition can be any relational operation, such as globalmax(myvariable) < 204. If the condition is met on any timestep, the computation will be halted.

Limiting the maximum timestep
The time range specification may optionally be followed by a LIMIT statement:

LIMIT maximum
LIMIT = maximum

This statement will prevent the timestep controller from increasing the computation timestep beyond the stated maximum.
maximum may be any constant arithmetic expression.

Critical Times
The time range specification may optionally be followed by a CRITICAL statement:

CRITICAL time1 {, time2, time3 ...}

This will ensure that each of the times in the list will fall at the end of some timestep interval.
Times may be separated by commas or spaces.
An #include statement can be used to read the times from a disk file.

3.3.15 Monitors and Plots

The **MONITORS** section, which is optional, is used to list the graphic displays desired at intermediate steps while a problem is being solved.

The **PLOTS** section, which is optional, is used to list the graphic displays desired on completion of a problem or stage, or at selected problem times.

PLOTS differ from MONITORS in that they are written to the permanent .PG6 record for viewing after the run is completed.

(For debugging purposes the global selector HARDMONITOR can be used to force MONITORS to be written to the .pg6 file.)

Plot statements and Monitor statements have the same form and function.

The basic form of a PLOT or MONITOR statement is:

display_specification (plot_data) display_modifiers

display_specification must be one of the known plot types, as described in the next section.
In some cases, multiple **plot_data** arguments may be provided.
There may be any number of **display_modifiers**, with meanings determined by the display_specification.
The various **display_modifiers** supported by FlexPDE are listed in the "Graphic Display Modifiers 200" section.

An Exhortation:

The MONITORS facility has been provided to allow users to see immediate feedback on the progress of their computation, and to display any and all data that will help diagnose failure or misunderstanding. Please use MONITORS extensively, especially in the early phases of model development! Since they do not write to the .pg6 storage file, they can be used liberally without causing disk file bloat. After the model is performing successfully, you can remove them or comment them out. Many user pleas for help recieved by PDE Solutions could be avoided if the user had included enough MONITORs to identify the cause of trouble.

Examples:

Samples | Usage | Plotting | Plot_test.pde 514

Note*: All example problems contain PLOTS and MONITORS.*

3.3.15.1 Graphics Display and Data Export Specifications

The MONITORS or PLOTS sections can contain one or more display specifications of the following types:

CDF (arg1 [,arg2,...])

- Requests the export of the listed values in netCDF version 3 format.
- The output will be two or three dimensional, following the current coordinate system or subsequent ON SURFACE 206 modifiers.
- The included domain can be zoomed.

- If the FILE 202 modifier does not follow, then the output will be written to a file "<problem>_<sequence>.cdf".
- Staged, eigenvalue and time-dependent problems will stack subsequent outputs in the same file, consistent with netCDF conventions.
- CDF uses a regular rectangular grid, so interface definition may be ragged.
- Use ZOOM 206 to show details.

CONTOUR (arg)

- Requests a two dimensional contour map of the argument, with levels at uniform intervals of the argument.

CONTOUR (arg1, arg2)

- Requests a two dimensional contour map of both arg1 and arg2, each with levels at independent uniform intervals.
- A level table is displayed for both arg1 and arg2.

ELEVATION (arg1, [arg2,...]) path

- Requests a two dimensional display (some times called a line-out) which displays the value of its argument(s) vertically and the value of its path horizontally.
- Each ELEVATION listed must have at least one argument and may have multiple arguments separated by commas.
- path can be either a line segment specified using the forms FROM 206 (X1,Y1) TO (X2,Y2) or ON 206 name, where name is a literal string selecting a path named in the BOUNDARIES 180 section.

GRID (arg1, arg2)

- Requests a two dimensional plot of the computation grid, with nodal coordinates defined by the two arguments.
- Grids are especially useful for displaying material deformations.
- In 3D problems, a two-argument GRID plot will show a cut-plane, and must be followed by an ON 206 specification.
- 3D cut plane grid plots do not necessarily accurately represent the computational grid.

GRID (arg1, arg2, arg3)

- Requests a three dimensional plot of the computation grid, with nodal coordinates defined by the three arguments.
- Only the outer surface of the grid will be drawn.
- This plot can be interactively rotated, as with SURFACE 199 plots.

MODE_SUMMARY

- In eigenvalue problems, this produces a SUMMARY page for each mode (comparable to the version 5 SUMMARY).

SUMMARY

- This plot type defines a text page on which only REPORT 208 items may appear.
- A SUMMARY page can be EXPORT 201 ed to produce text reports of scalar values.

SUMMARY ('string')

- If a string argument is given with a SUMMARY command, it will appear as a page header on the

summary page.

SURFACE (arg)

- A quasi three dimensional surface which displays its argument vertically.
- If no VIEWPOINT 205 clause is used, the viewing azimuth defaults to 216 degrees, the distance to three times the size, and the viewing elevation to 30 degrees.

TABLE (arg1 [,arg2,...])

- Requests the export of the listed values in tabular ASCII format.
- The output will be two or three dimensional, following the current coordinate system or subsequent ON 204 modifiers.
- The included domain can be zoomed.
- If the FILE 202 modifier does not follow, then the output will be written to a file "<problem>_<sequence>.tbl".
- Staged, eigenvalue and time-dependent problems will create separate files for each stage or mode, with additional sequencing numbers in the name.
- TABLE output uses a regular rectangular grid, so interface definition may be lost.
- Use ZOOM 206 to show details.

TECPLOT (arg1 [,arg2,...])

- Requests the export of the listed values to a file readable by the TecPlot visualization system.
- The output will be two or three dimensional, following the current coordinate system.
- The entire mesh is exported.
- If the FILE 202 modifier does not follow, then the output will be written to a file "<problem>_<sequence>.dat".
- Staged, eigenvalue and time-dependent problems will stack subsequent outputs in the same file, consistent with TecPlot conventions.
- TecPlot uses the actual triangular or tetrahedral computation mesh (subdivided to linear basis), so material interfaces are preserved.

TRANSFER (arg1 [,arg2,...])

- Requests the export of the listed values and finite element mesh data in a file readable by FlexPDE using the TRANSFER or TRANSFERMESH 169 input command. This method of data transfer between FlexPDE problems retains the full accuracy of the computation, without the error introduced by the rectangular mesh of the TABLE function.
- The exported domain cannot be zoomed.
- If the FILE 202 modifier does not follow, then the output will be written to a file "<problem>_<sequence>.dat". This export format uses the actual computation mesh, so material interfaces are preserved.
- The full computation mesh is exported.
- When used in Staged, Time dependent or Eigenvalue problems, each output file will be identified by appending a sequence number to the file name.
- Note: TRANSFER files do not record the state of HISTORY plots. Problems restarted from a TRANSFER file will have fragmented HISTORY plots.

VECTOR (vector)

- Requests a two dimensional display of directed arrows in which the direction and magnitude of the arrows is set by the **vector** argument.
- The origin of each arrow is placed at its reference point.

VECTOR (arg1, arg2)

- Requests a two dimensional display of directed arrows in which the horizontal and vertical components of the arrows are given by arg1 and arg2.
- The origin of each arrow is placed at its reference point.

VTK (arg1 [,arg2,...])

- Requests the export of the listed values to a file in VTK (Visualization Tool Kit) format for display by visualization systems such as VisIt.
- The output will be two or three dimensional, following the current coordinate system.
- The entire mesh is exported.
- If the FILE modifier does not follow, then the output will be written to a file "<problem>_<sequence>.vtk".
- Staged, eigenvalue and time-dependent problems will produce a family of files distinguished by the sequence number.
- VTK format uses the actual triangular or tetrahedral computation mesh, so material interfaces are preserved.
- The VTK format supports quadratic finite element basis directly, but not cubic. To export from cubic-basis computations, use VTKLIN.

VTKLIN (arg1 [,arg2,...])

- Produces a VTK format file in which the native cells of the FlexPDE computation have been converted to a set of linear-basis finite element cells.
- This command may be used to export to VTK visualization tools from cubic-basis FlexPDE computations, or in cases where the visualization tool does not support quadratic basis.

For all commands, the argument(s) can be any valid expression.

3.3.15.2 Graphic Display Modifiers

The appearance of any display can be modified by adding one or more of the following clauses:

AREA_INTEGRATE

- Causes CONTOUR and SURFACE plots in cylindrical geometry to be integrated with dr*dz element, rather than default 2*pi*r*dr*dz volume element.
- See also: LINE_INTEGRATE 203

AS 'string'

- Changes the label on the display from the evaluated expression to **string**.

BLACK

- Draws current plot in black color only.

BMP
BMP (pixels)
BMP (pixels, penwidth)

- Selects automatic creation of a graphic export file in BMP format.

- pixels is the horizontal pixel count, which defaults to 1024 if omitted.
- penwidth is an integer (0,1,2 or 3) which specifies the width of the drawn lines, in thousandths of the drawing width (0 means thin).
- The export file name is the problem name with plot number and sequence number appended.
- The file name cannot be altered.

CONTOURS = number

- Selects the number of contour lines for CONTOUR plots. This is a local control equivalent to the global CONTOURS control, but applying only to a single plot.

DROPOUT

- Marks EXPORT and TABLE output points which fall outside the problem domain as "external". This modifier affects only EXPORTS and TABLES with FORMAT strings (see below).

EMF
EMF (pixels)
EMF (pixels, penwidth)

- Windows version only. Produces a Microsoft Windows Enhanced Metafile output.
- pixels is the horizontal pixel count of the reference window, which defaults to 1024 if omitted.
- penwidth is an integer (0,1,2 or 3) which specifies the width of drawn lines, in thousandths of the drawing width (0 means thin).
- The export file name is the problem name with plot number and sequence number appended.
- The file name cannot be altered.
- **Warning:** FlexPDE uses Windows rotated fonts to plot Y-labels and axis labels on surface plots. Microsoft Word can read and resize these pictures, but its picture editor cannot handle them, and immediately "rectifies" them to horizontal.

EPS

- Produces an Encapsulated PostScript output.
- The graphic is a 10x7.5 inch landscape-mode format with 7200x5400 resolution.

EXPORT

- Causes a disk file to be written containing the data represented by the associated MONITOR or PLOT .
- A regular rectangular grid will be constructed, and the data will be printed in a format suitable for reading by the FlexPDE TABLE function.
- The dimension of the grid will be determined by the plot grid density appropriate to the type of plot.
- The format of EXPORTED data may be controlled by the FORMAT modifier (see below).
- (This is a renaming of the older PRINT modifier)

EXPORT (n)

- Modifies the EXPORT command by specifying the dimension of the printed data grid.
- For two- or three-dimensional plots, the grid will be (n x n) or (n x n x n).

EXPORT (nx, ny)
EXPORT (nx, ny, nz)

- Modifies the EXPORT command by specifying the dimension of the printed data grid.

FILE 'string'

- Overrides the default naming convention for files created by the EXPORT or PRINT modifiers, and writes the file named 'string' instead.

FIXED RANGE (arg1, arg2)

- Changes the dynamically set range used for the variable axis to a minimum value of arg1 and a maximum of arg2. Data outside this range is not plotted.
- See also: RANGE 205

FORMAT 'string'

- This modifier replaces the default format of the EXPORT or PRINT modifiers, or of the TABLE output command. When this modifier appears, the output will consist of one line for each point in the export grid.
- The contents of this line will be completely controlled by the format string as follows:

1. all characters except "#" will be copied literally to the output line.
2. "#" will be interpreted as an escape character, and various options will be selected by the character following the "#":
   - #x, #y, #z and #t will print the value of the spatial coordinates or time of the data point;
   - #1 through #9 will print the value of the corresponding element of the plot function list;
   - #b will write a taB character;
   - #r will cause the remainder of the format string to be repeated for each plot function in the plot list;
   - #i inside a repeated string will print the value of the current element of the plot function list.

- See the example problems "export_format" and "export_history".

FRAME (X, Y, Wide, High)

- Forces the plot frame to the specified coordinates, regardless of the size of the problem domain.
- The plot frame will be forced to a 1:1 aspect ratio using the largest of the width and height values.
- This allows the creation of consistently-sized plots in moving-mesh problems.
- See "Samples | Moving_Mesh | Piston.pde".
- See also: ZOOM 206

GRAY

- Draws current plot with a 32-level gray scale instead of the default color palette.

INTEGRATE

- Causes a report of the integral under the plotted function.
- For CONTOUR and SURFACE plots, this is a volume integral (with Cartesian element dx*dy*1 or cylindrical element 2*pi*r*dr*dz).
- For ELEVATIONS, it is a surface integral (with Cartesian element dl*1 and cylindrical element 2*pi*r*dl). (See also AREA_INTEGRATE, LINE_INTEGRATE).
- This integral differs from a REPORT(INTEGRAL(...)) in that this command will integrate on the plot grid, while the REPORT will integrate on the computation grid.
- This modifier can be globally imposed by use of PLOTINTEGRATE in the SELECT section.

LEVELS = l1, l2, l3.....

- Explicitly defines the contour levels for CONTOUR plots.

LINE_INTEGRATE

- Causes ELEVATIONS in cylindrical geometry to be integrated with dl element, rather than default 2*pi*r*dl element.
- See also: AREA_INTEGRATE 200

LOG
LINLOG
LOGLIN
LOGLOG

- Changes the default linear scales used to those specified by the scaling command.
- LOG is the same as LINLOG, and specifies logarithmic scaling in the data coordinate.

<lx><ly><lz>

- Changes the default linear scales used to those specified by the scaling command.
- Each of **<lx>**, **<ly>** and **<lz>** can be either LIN or LOG, and controls the scaling in the associated dimension.

LOG (number)
...combinations as above

- Limits the number of decades of data displayed to number.
- This effect can also be achieved globally by the Selector LOGLIMIT.

MERGE

- Sends EXPORT output for all stages or plot times to a single output file.
- This is the default for TECPLOT output.
- This option can be set globally by SELECT PRINTMERGE.

MESH

- In SURFACE plots, causes the surface to be displayed as a hidden-line drawing of the meshed surface.
- This display is more suitable on some hardcopy devices.

NOHEADER

- Deletes the problem-identification header from EXPORT output.

NOLINES

- Suppresses mesh lines in grid plot.

NOMERGE

- Sends EXPORT output for each stage or plot time to a separate output file.
- This is the default for EXPORT output.

NOMINMAX

- Deletes "o" and "x" marks at min and max values on contour plot.

NORM

- In VECTOR plots, causes all vectors to be drawn with the same length. Only the color identifies different magnitudes.

NOTAGS

- Suppresses labelling tags on contour or elevation plot.
- This can be applied globally with SELECT NOTAGS.

NOTIPS

- Plots VECTORS as line segments without heads.
- The line segment will be centered on the reference point.

ON <control>

- Selects region, surface or layer restrictions of plot domain. See "Controlling the Plot Domain 206".

PAINTED

- Fills areas between contour lines with color. (This is slower than conventional contour lines.)

PAINTMATERIALS
PAINTREGIONS

- Draw color-filled grid plot.
- These local flags are equivalent to and override the corresponding global flags set in the SELECT section. They affect only the current plot.

PENWIDTH = n

- Sets the on-screen pen width for the current plot.
- n is an integer (0,1,2,3,...) which specifies the width of the drawn lines, in thousandths of the pixel width (0 means thin).
- See also : Global Graphics Controls 152.

PNG
PNG (pixels)
PNG (pixels, penwidth)

- Selects automatic creation of a graphic export file in PNG format.
- pixels is the horizontal pixel count, which defaults to 1024 if omitted.
- penwidth is an integer (0,1,2 or 3) which specifies the width of the drawn lines, in thousandths of the pixel width (0 means thin).
- The export file name is the problem name with plot number and sequence number appended.
- The file name cannot be altered.

POINTS = n
POINTS = (nx , ny)
POINTS = (nx, ny, nz)

- Overrides the default plot grid size and uses n instead.
- Two and three dimensional exports will use n in all dimensions.
- For two-dimensional export commands, the two-dimensional grid can be explicitly controlled.
- For three-dimensional exports, the three-dimensional grid can be explicitly controlled.

PPM
PPM (pixels)
PPM (pixels, penwidth)

- Selects automatic creation of a graphic export file in PPM format.
- pixels is the horizontal pixel count, which defaults to 1024 if omitted.
- penwidth is an integer (0,1,2 or 3) which specifies the width of the drawn lines, in thousandths of the pixel width (0 means thin).
- The export file name is the problem name with plot number and sequence number appended.
- The file name cannot be altered.

PRINT
PRINT (n)
PRINT (nx, ny)
PRINT (nx, ny, nz)

- Equivalent to EXPORT, EXPORT(n), EXPORT(nx,ny) and EXPORT(nx,ny,nz), respectively.

PRINTONLY

- Supresses graphical output. Use with PRINT or EXPORT to create text output only.

RANGE (arg1, arg2)

- Changes the dynamically set range used for the variable axis to a minimum value of arg1 and a maximum of arg2.
- If the calculated value of the variable falls outside of the range argument, the range argument is ignored and the dynamically calculated value is used.
- See also: FIXED RANGE 202

VIEWPOINT(X, Y, angle)

- With SURFACE plots, the VIEWPOINT modifier sets the viewing azimuth and perspective distance and the viewing elevation angle.

VOL_INTEGRATE

- Causes CONTOURS and SURFACE plots in cylindrical geometry to be integrated with 2*pi*r*dr*dz element.
- This is the default, and is equivalent to INTEGRATE.
- See also: INTEGRATE 202, AREA_INTEGRATE 200

XPM
XPM (pixels)
XPM (pixels, penwidth)

- Selects automatic creation of a graphic export file in XPM format.
- pixels is the horizontal pixel count, which defaults to 1024 if omitted.
- penwidth is an integer (0,1,2 or 3) which specifies the width of the drawn lines, in thousandths of the pixel width (0 means thin).
- The export file name is the problem name with plot number and sequence number appended.
- The file name cannot be altered.

ZOOM (X, Y, Wide, High)

- Expands (zooms) a selected area of the display or export, with (X,Y) defining the lower left hand corner of the area and (Wide,High) defining the extent of the expanded area.
- In 3D cut planes, the X and Y coordinates refer to the horizontal and vertical dimensions in the cut plane.
- See also: FRAME 202

ZOOM (X, Y, Z, Xsize, Ysize, Zsize)

- Expands (zooms) a selected volume of an export, with (X,Y,Z) defining the lowest corner of the volume and (Xsize,Ysize,Zsize) defining the extent of the included volume.

3.3.15.3 Controlling the Plot Domain

"ON" selectors

The primary mechanism for controlling the domain over which plot data are constructed is the "ON" statement, which has many forms:

ON "name"
ON REGION "name"
ON REGIONS "name1" , "name2" { , ... }
ON REGION number
ON REGIONS number1 , number2 { , ... }
ON GRID(Xposition,Yposition)

In three-dimensional problems, the following are also meaningful:

ON LAYER "name"
ON LAYERS "name1" , "name2" { , ... }
ON LAYER number
ON LAYERS number1 , number2 { , ... }
ON SURFACE "name"
ON SURFACE number
ON equation

The first listed form selects a boundary path, region, layer or surface depending on the definition of the "name". (It is actually redundant to specify SURFACE "name", etc, since the fact that a surface is being specified should be clear from the "name" itself. Nevertheless, the forms are acceptable.)

The multiple REGIONS and LAYERS forms allow grouping REGIONS and LAYERS to select the portion of the domain over which to display the plot.

In many cases, particularly in 3D, more than one "ON" clause can be used for a single plot, since each "ON" clause adds a restriction to those already in effect. There is a direct correspondence between the "ON" clauses of a plot statement and the arguments of the various INTEGRAL 136 operators, although some of the allowable integral selections do not have valid corresponding plot options.

In two dimensional geometries, area plots which are not otherwise restricted are assumed to be taken over the entire problem domain.

Contours, Surface Plots, Grid Plots and Vector Plots

Contours, "surfaces" (3D topographic displays), grid plots and vector plots must be taken on some kind of two dimensional data surface, so in 3D problems these plot commands are incomplete without at least one "ON" clause. This can be an extrusion-surface name, or a cut-plane equation (it cannot be a

projection-plane boundary path). For example, in a 3D problem,

CONTOUR(...) ON SURFACE 2
requests a contour plot of data evaluated on the second extrusion surface.
CONTOUR(...) ON SURFACE "top"
requests a contour plot of data evaluated on the extrusion surface named "top".
CONTOUR(...) ON X=Y
requests a contour plot of data evaluated on the cut plane where x=y.

In addition to a basic definition of the data surface, "ON" clauses may be used to restrict the display to an arbitrary REGION or LAYER. In 2D, a REGION restriction will display only that part of the domain which is in the stated region:

CONTOUR(...) ON REGION 2
requests a contour plot of data evaluated on REGION 2.

Similarly, in 3D,

CONTOUR(...) ON SURFACE 2 ON REGION 2
requests a contour plot of data evaluated on extrusion surface 2, restricted to that part of the surface lying above REGION 2 of the baseplane projection.

CONTOUR(...) ON SURFACE 2 ON REGION 2 ON LAYER 3
requests a contour plot of data evaluated on extrusion surface 2, restricted to that part of the surface lying above REGION 2 of the baseplane projection, and with the evaluation taken in LAYER 3, which is assumed to be bounded by the selected surface.

.

Cut Planes in 3D

Contours, surface plots and vector plots can also be specified on cut planes by giving the general formula of the cutting plane:

CONTOUR(...) ON X = expression
requests a contour plot of data evaluated on the Y-Z plane where X is the specified value.

Cut planes need not be simple coordinate planes:

CONTOUR(...) ON X=Y
requests a contour plot of data evaluated on the plane containing the z-axis and the 45 degrees line in the XY plane.

The coordinates displayed in oblique cut planes have their origin at the point of closest approach to the origin of the domain coordinates. The axes are chosen to be aligned with the nearest domain coordinate axes.

Elevation Plots

Elevation plots can be specified by endpoints of a line:

ELEVATION(...) FROM (x1,y1) TO (x2,y2)
ELEVATION(...) FROM (x1,y1,z1) TO (x2,y2,z2)

The plot will be displayed on the straight line connecting the specified endpoints. These endpoints might span only a small part of the problem domain, or they might exceed the domain dimensions somewhat, in which case the plot line will be truncated to the interior portion.

In 2D geometry only, an elevation plot may be specified by the name of a boundary path, as in

ELEVATION(...) ON "outer_boundary"

These boundary-path elevations can be additionally restricted as to the region in which the evaluation is to be made:

ELEVATION(...) ON "inner_boundary" ON REGION "core"

This form requests that the evaluation of the plot function be made in the region named "core", with the assumption that "core" is one of the regions adjoining the "inner_boundary" path.

Plots on Deformed Grids

In fixed-mesh problems with implied deformation, such as "Samples | Applications | Stress | Bentbar.pde", CONTOUR, SURFACE and VECTOR plots can be displayed on the deformed domain shape. The syntax combines the forms of CONTOUR and GRID plots:

CONTOUR(...) ON GRID(Xposition,Yposition)

See "Samples | Usage | Plotting | Plot_on_grid.pde" 513 for an example.
(This feature is new in version 6.03)

Sign of Vector Components

In many cases, boundary-path elevations present normal or tangential components of vectors. For these applications, the sense of the direction is the same as the sense of the NATURAL boundary condition:
The positive normal is outward from the evaluation region.
The positive tangent is counter-clockwise with respect to the evaluation region.

Plots of the normal components of vectors on extrusion surfaces in 3D follows the same rule:
The positive normal is outward from the evaluation region.

3.3.15.4 Reports

Any display specification can be followed by one or more of the following clauses to add report quantities to the plot page:

REPORT expression
Adds to the bottom of a display the text 'text of **expression**=value of **expression**', where **expression** is any valid expression, possibly including integrals. Multiple **REPORT** clauses may be used. **REPORT** is especially useful for reporting boundary and area integrals and functions thereof.

REPORT expression AS 'string'
A labeled REPORT of the form 'string=value_of_expression'.

REPORT('string')
REPORT 'string'
Inserts 'string' into the REPORT sequence.

3.3.15.5 The ERROR Variable

The reserved word ERROR can be used to display the current state of spatial error estimates over the mesh, as for example:

```
CONTOUR(ERROR)
```

3.3.15.6 Window Tiling

When multiple MONITORS or PLOTS are listed, FlexPDE displays each one in a separate window and automatically adjusts the window sizes to tile all the windows on the screen. Individual windows cannot be independently resized or iconized. Any plot window can be maximized by double-clicking, or by right-clicking to bring up a menu.

In steady-state and eigenvalue problems, MONITORS are displayed during solution, and are replaced by **PLOTS** on completion.

In time-dependent problems, MONITORS, PLOTS and HISTORIES are displayed at all times.

3.3.15.7 Monitors in Steady State Problems

In steady state problems the listed MONITORS are displayed after each regrid. In addition, after each Newton-Raphson iteration of a nonlinear problem or after each residual iteration of a linear problem, if sufficient time has elapsed since the last monitor display, an interim set of monitors will be displayed.

3.3.15.8 Monitors and Plots in Time Dependent Problems

In time dependent problems the display specifications must be preceded by a display-time declaration statement. The display-time declaration statement may be either of the form

FOR CYCLE = number

in which case the displays will be refreshed every number time steps, or

FOR T = time1 [timeset ...]

Where each **timeset** may be one of the following :

time2
BY delta TO time2

In this case the displays will be refreshed at times specified by the **timeset** values.

Any number of plot commands can follow a display-time declaration, and the specification will apply to all of them. It is not necessary to give a display-time specification for each plot.

Multiple display time declaration statements can be used. When multiple display time statements are used

each applies to all subsequent display commands until a new time declaration is encountered or the MONITORS or PLOTS section ends.

Examples:

"Samples | Applications | Heatflow | Float_Zone.pde" 337
"Samples | Applications | Chemistry | Melting.pde" 290

3.3.15.9 Hardcopy

A right-click on any plot window, whether tiled or maximized, will bring up a menu from which the plot may be printed or exported (or rotated, if this is meaningful for the plot).

Text listings of plotted values can be written to disk by the plot modifier EXPORT (aka PRINT) in the descriptor.

3.3.15.10 Graphics Export

Bitmaps

A right-click in any displayed plot window brings up a menu, one item of which is "Export". Clicking this item brings up a dialog for exporting bitmap forms of the displayed plot. Current options are BMP, PNG, PPM and XPM. See the "Getting Started" section for more information.
All these formats can also be selected automatically as graphic display modifiers 200.

Retained Graphics

All displays in the PLOTS section are written in compressed form to a disk file with the extension ".PG6".
These files may be redisplayed at a later time by use of the "View" menu item in the "File" menu. On some systems, this may be accomplished simply by double-clicking the ".PG6" file in the system file manager.
See the "Getting Started" section for more information.

Screen Grabs

Any display may also be pasted into other windows programs by using a screen capture facility such as that provided with PaintShopPro by JASC (www.jasc.com).

Export Files

The plot types CDF, TABLE, TECPLOT and VTK 197 can be used to export data to other applications for external processing. TRANSFER 197 can be used to transfer data to another FlexPDE run for postprocessing.
See Graphics Display and Data Export 197 or Exporting Data to Other Applications 106 for more information.

Examples

See the following sample problems for examples of exporting plot data:
Samples | Usage |Plotting | Print_test.pde 515
Samples | Usage |Import-Export|Export.pde 477
Samples | Usage |Import-Export|Export_Format.pde 477
Samples | Usage |Import-Export|Export_History.pde 478

3.3.15.11 Examples

See the sample problem Samples | Usage | Plotting | Plot_test.pde 514 for examples of PLOTS and MONITORS.

See the sample problem Samples | Usage | Plotting | Print_test.pde 515 for examples of exporting plot data.

See the sample problem Samples | Usage | Import-Export | Export.pde 477 for examples of exports without display.

3.3.16 Histories

The **HISTORIES** section, which is optional, specifies values for which a time history is desired. While multiple **HISTORY** statements can be listed they must all be of the form:

HISTORY (arg1 [,arg2,...])
HISTORY (arg1 [,arg2,...]) AT (X1,Y1) [(X2,Y2)...]

The coordinates specify locations in the problem at which the history is to be recorded. If no coordinate is given, the arg must evaluate to a scalar.

The modifiers and reports available to PLOTS and MONITORS may also be applied to HISTORY statements.

The display of HISTORIES is controlled by the AUTOHIST select switch, which defaults to ON. With the default setting all HISTORIES are automatically refreshed and displayed with the update of any MONITORS or PLOTS.

If desired, HISTORY statements can be included directly in the MONITORS section or PLOTS section.

Histories in Staged Problems

HISTORY statements may be used in STAGED problems as well as in time-dependent problems.
In this case, the default abscissa will be stage number. You can select a different value for the abscissa quantity by appending the clause

VERSUS expression

In this case, the values of the given expression in the various stages will be used as the plot axis.

Windowing History Plots

HISTORY plots by default display the total time range of the problem run. Specific time ranges can be specified in several ways. A global window specifier can be set in the SELECT section:

SELECT HISTORY_WINDOW = time

This command causes all histories to display only the most recent **time** interval of the data.

Individual HISTORY plots can be windowed by the two plot qualifier forms:

WINDOW = time	selects a moving window containing the most recent **time** interval
WINDOW (time1 , time2)	selects a fixed time range, plotting the time between **time1** and **time2**

See the sample problem "Samples | Usage | Two_Histories.pde" 396 for an example.

3.3.17 End

All problem descriptors must have an END section.

With the exception of a numeric enabling key used in special demonstration files prepared by PDE Solutions Inc., anything appearing after the reserved word end is ignored by FlexPDE and treated as a comment.

Problem notes can be conveniently placed after the reserved word END.

3.4 Batch Processing

A special form of descriptor is used to specify a group of problems to be run in batch mode.

A single "section" introduced by the word BATCH identifies a descriptor as a batch control file. Following this header, a sequence of names appears, each name enclosed in quote marks. Commas may optionally be used to separate the names. Any number of names may appear on each line of the descriptor. Each name is the name of a problem descriptor to be run. Names may include directory paths, which are assumed to originate in the directory containing the batch descriptor. The ".pde" extension is not required, and will be assumed if omitted. The list should be closed with an END statement.

Example:

```
BATCH
    { FlexPDE will accept either \ or / as a separator }
    "misc\table", "steady_state\heat_flow\slider"
    "steady_state/stress/3d_bimetal"
END
```

The entire problem list is examined immediately, and any syntax errors in the names are reported. All files named in the list are located, and missing files are reported before any processing begins.

Each problem named in the list is run to completion in sequence. As the problems run, status information is written to a log file in the directory containing the batch descriptor. This file has the same name as the batch descriptor, with the extension ".log", and all problems in the list are summarized in this single file. Graphical output from each problem is written as usual to a unique ".pg6" file in the directory with the specific descriptor. After the run is completed, this graphic output may be reviewed by restarting FlexPDE and using the VIEW 19 menu item.

Simple names may be listed without the quotes, but in this case embedded spaces, path separators, reserved words and numeric initials will all cause error diagnostics.

An optional **DELAY** value may be set immediately following the BATCH identifier. This delay value specifies the number of seconds to wait prior to starting the next problem in the sequence.

For example,

```
BATCH
  DELAY = 3
  ...
END
```

Part IV

Electromagnetic Applications

4 Electromagnetic Applications

4.1 Introduction

FlexPDE is a software tool for finding numerical solutions to systems of linear or non-linear partial differential equations using the methods of finite element analysis. The systems may represent static boundary value, time dependent initial/boundary value, or eigenvalue problems. Rather than addressing the solution of specific equations related to a given area of application, FlexPDE provides a framework for treating partial differential equation systems in general. It gives users a straightforward method of defining the equations, domains and boundary conditions appropriate to their application. From this description it creates a finite element solution process tailored to the problem. Within quite broad limits, then, FlexPDE is able to construct a numerical solution to a wide range of applications, without itself having any built-in knowledge of any of them.

The goal of this book is not to provide a discussion of the specific grammatical rules of writing scripts for FlexPDE, nor to describe the operation of the graphical user interface. Those topics are covered in other volumes of the FlexPDE documentation, the Getting Started guide, the User Guide tutorial, and the Problem Descriptor Reference.

In this book we will address several fields of physics in which FlexPDE finds fruitful application, describing the various problems, the mathematical statement of the partial differential equation system, and the ultimate posing of the problem to FlexPDE. The volume is accompanied by the text of all the examples, which the user can submit to FlexPDE to see the solution in progress or use as a foundation for problems of his own.

This manual is emphatically not a compendium of the problems FlexPDE "knows how to solve". It is rather a group of examples showing ways in which the power of FlexPDE can be applied to partial differential equations systems in many fields. The true range of applicability of FlexPDE can be demonstrated only by the full range of ingenuity of users with insight into the mathematics of their own special fields.

Nor does this manual attempt to present textbook coverage of the theory of the topics addressed. The range of applications addressable by FlexPDE would make such an attempt impossible, even if we were capable of such an endeavor. Instead, we have presented enough of the theory of each topic to allow those practitioners who are familiar with the subject to see how the material has been analyzed and presented to FlexPDE. Users who are unfamiliar with the various fields of application should consult standard textbooks to find the full theoretical development of the subjects.

4.1.1 Finite Element Methods

It is not our intent to provide an elaborate discussion of finite element methods. One goal of FlexPDE has been to allow users in the various fields of science and engineering to begin reaping the benefits of applying finite element analysis to their individual work without becoming programmers and numerical analysts. There are hundreds of books in print detailing the method and its variants in many fields, and the interested student can find a wealth of material to keep him busy. If we have been successful in our endeavors, he won't have to.

Nevertheless, a familiarity with some of the concepts of finite element analysis can be of benefit in understanding how FlexPDE works, and why it sometimes does not. Hence this brief overview.

4.1.2 Principles

Partial differential equations generally arise as a mathematical expression of some conservation principle such as a conservation of energy, momentum or mass. Partial differential equations by their very nature deal with continuous functions -- a derivative is the result of the limiting process of observing differences

at an infinitesimal scale. A temperature distribution in a material, for example, is assumed to vary smoothly between one extreme and another, so that as we look ever more closely at the differences between neighboring points, the values become ever closer until at "zero" separation, they are the same.

Computers, on the other hand, apply arithmetic operations to discrete numbers, of which only a limited number can be stored or processed in finite time. A computer cannot analyze an infinitude of values. How then can we use a computer to solve a real problem?

Many approaches have been devised for using computers to approximate the behavior of real systems. The finite element method is one of them. It has achieved considerable success in its few decades of existence, first in structural mechanics, and later in other fields. Part of its success lies in the fact that it approaches the analysis in the framework of integrals over small patches of the total domain, thus enforcing aggregate correctness even in the presence of microscopic error. The techniques applied are little dependent on shapes of objects, and are therefore applicable in real problems of complex configuration.

The fundamental assumption is that no matter what the shape of a solution might be over the entire domain of a problem, at some scale each local patch of the solution can be well approximated by a low-order polynomial. This is closely related to the well-known Taylor series expansion, which expresses the local behavior of a function in a few polynomial terms.

In a two-dimensional heat flow problem, for example, we assume that if we divide the domain up into a large number of triangular patches, then in each patch the temperature can be well represented by, let us say, paraboloidal surfaces. Stitching the patches together, we get a Harlequin surface that obeys the differential limiting assumption of continuity for the solution value—but perhaps not for its derivatives. The patchwork of triangles is referred to as the computation "mesh", and the sample points at vertices or elsewhere are referred to as the "nodes" of the mesh.

In three dimensions, the process is analogous, using a tetrahedral subdivision of the domain.

How do we determine the shape of the approximating patches?

1. Assign a sample value to each vertex of the triangular or tetrahedral subdivision of the domain. Then each vertex value is shared by several triangles (tetrahedra).

2. Substitute the approximating functions into the partial differential equation.

3. Multiply the result by an importance-weighting function and integrate over the triangles surrounding each vertex.

4. Solve for the vertex values which minimize the error in each integral.

This process, known as a "weighted residual" method, effectively converts the continuous PDE problem into a discrete minimization problem on the vertex values. This is usually known as a "weak form" of the equation, because it does not strictly enforce the PDE at all points of the domain, but is instead correct in an integral sense relative to the triangular subdivision of the domain.

The locations and number of sample values is different for different interpolation systems. In FlexPDE, we use either quadratic interpolation (with sample values at vertices and midsides of the triangular cells), or cubic interpolation (with values at vertices and two points along each side). Other configurations are possible, which gives rise to various "flavors" of finite element methods.

4.1.3 Boundary Conditions

A fundamental component of any partial differential equation system is the set of boundary conditions, which alone make the solution unique. The boundary conditions are analogous to the integration constants that arise in integral calculus. We say $\int x^2 dx = \frac{1}{3}x^3 + C$, where C is any constant. If we differentiate the right hand side, we recover the integrand, regardless of the value of C.

In a similar way, to solve the equation $\frac{\partial^2 u}{\partial x^2} = 0$, we must integrate twice. The first integration gives $\frac{\partial u}{\partial x} + C_1$, and the second gives $C_1 x + C_2$. These integration constants must be supplied by the boundary conditions of the problem statement.

It is clear from this example that there are as many integration constants as there are nested differentiations in the PDE. In the general case, these constants can be provided by a value at each end of an interval, a value and a derivative at one end, etc. In practice, the most common usage is to provide either a value or a derivative at each end of the domain interval. In two or three dimensions, a value or derivative condition applied over the entire bounding curve or surface provides one condition at each end of any coordinate integration path.

4.1.4 Integration by Parts and Natural Boundary Conditions

A fundamental technique applied by FlexPDE in treating the finite element equations is "integration by parts", which reduces the order of a derivative integrand, and also leads immediately to a formulation of derivative boundary conditions for the PDE system.

In its usual form, integration by parts is given as

$$\int_a^b u dv = (uv)\Big|_a^b - \int_a^b v du .$$

Application of integration by parts to a vector divergence in a two- or three-dimensional domain, for example, results in the Divergence Theorem, given in 2D as

$$\iint_A \nabla \bullet \vec{F} \, dA = \oint_l \vec{F} \bullet \hat{n} \, dl .$$

This equation relates the integral inside the area to the flux crossing the outer boundary ($\hat{n}$ referring to the outward surface-normal unit vector).

As we shall see, the use of integration by parts has a wide impact on the way FlexPDE interprets and solves PDE systems.

Applied to the weighted residual method, this process dictates the flux conservation characteristics of the finite element approximation at boundaries between the triangular approximation cells, and also provides a method for defining the interaction of the system with the outside world, by specifying the value of the surface integrand.

The values of the surface integrands are the "Natural" boundary conditions of the PDE system, a term which also arises in a similar context in variational calculus.

FlexPDE uses the term "Natural" boundary condition to specify the boundary flux terms arising from the integration by parts of all second-order terms in the PDE system.

For example, in a heat equation, $\nabla \bullet (-k \nabla \varphi) + S = 0$, the divergence term will be integrated by parts, resulting in

$$\iint_A \nabla \bullet (-k \nabla \varphi) \, dA = \oint_l (-k \nabla \varphi) \bullet \hat{n} \, dl \quad (0.1)$$

The right hand side is the heat flux crossing the outer boundary, and the value of $-k \nabla \varphi$ must be provided

by the user in a Natural boundary condition statement (unless a value BC is applied instead).

At an interface between two materials, $-k_1\left(\nabla\varphi\right)_1\bullet\hat{n}_1$ represents the heat energy leaving material 1 at a point on the interface. Likewise, $-k_2\left(\nabla\varphi\right)_2\bullet\hat{n}_2$ represents the heat energy leaving material 2 at the same point. Since the outward normal from material 1 is the negative of the outward normal from material 2, the sum of the fluxes at the boundary is $\left[k_2\left(\nabla\varphi\right)_2 - k_1\left(\nabla\varphi\right)_1\right]\bullet\hat{n}_1$, and this becomes the Natural boundary condition at the interface. In this application, we want energy to be conserved, so that the two flux terms must sum to zero. Thus the internal Natural BC is zero at the interface, and this is the default value applied by FlexPDE.

Useful Integral Rules

(0.2) $$\iiint_V \nabla f dV = \iint_S (\vec{n} f) dS$$ (Gradient Theorem)

(0.3) $$\iiint_V \nabla\bullet F dV = \oiint_S (\vec{n}\bullet\vec{F}) dS$$ (Divergence Theorem)

(0.4) $$\iiint_V \varphi\, \nabla\bullet F dV = \oiint_S \varphi\, (\vec{n}\bullet\vec{F}) dS - \iiint_V \left(\nabla\varphi\right)\bullet\vec{F} dV$$

(0.5) $$\iiint_V \nabla\times\vec{F} dV = \oiint_S (\vec{n}\times\vec{F}) dS$$ (Curl Theorem)

4.1.5 Adaptive Mesh Refinement

We have said that at “some scale“, the solution can be adequately approximated by a set of low-order polynomials. But it is not always obvious where the mesh must be dense and where a coarse mesh will suffice. In order to address this issue, FlexPDE uses a method of “adaptive mesh refinement“. The problem domain presented by the user is divided into a triangular mesh dictated by the feature sizes of the domain and the input controls provided by the user. The problem is then constructed and solved, and the cell integrals of the weighted residual method are crosschecked to estimate their accuracy. In locations where the integrals are deemed to be of questionable accuracy, the triangles are subdivided to give a new denser mesh, and the problem is solved again. This process continues until FlexPDE is satisfied that the approximation is locally accurate to the tolerance assigned by the user. Acceptable local accuracy does not necessarily guarantee absolute accuracy, however. Depending on how errors accumulate or cancel, the global accuracy could be better or worse than the local accuracy condition implies.

4.1.6 Time Integration

The finite element method described above is most successful in treating boundary value problems. When addressing initial value problems, while the finite element method could be applied (and sometimes is), other techniques are frequently preferable. FlexPDE uses a variable-order implicit backward difference method (BDM) as introduced by C.W. Gear. In most cases, second order gives the best tradeoff between stability, smoothness and speed, and this is the default configuration for FlexPDE. This method fits a quadratic in time to each nodal value, using two known values and one future (unknown) value. It then solves the coupled equations for the array of nodal values at the new time. By looking backward one additional step, it is possible to infer the size of the cubic term in a four-point expansion of the time behavior of each nodal value. If these cubic contributions are large, the timestep is reduced, and if extreme, the current step repeated.

4.1.7 Summary

With this very cursory examination of finite element methods, we are ready to start applying FlexPDE to the solution of PDE systems of interest in real scientific and engineering work.

Disclaimer

We have tried to make these notes as accurate as possible, but because we are not experts in all the fields addressed, it is possible that errors have crept in. We invite readers to comment freely on the material presented here, and to take us to task if we have erred.

4.2 Electrostatics

Perhaps the most important of all partial differential equations is the simple form

(1.1) $\nabla\bullet(k\nabla\varphi)+q=0$

It is encountered in virtually all branches of science and engineering, and describes the diffusion of a quantity φ with diffusivity k and volume source q. With k=1 it is referred to as Poisson's equation, $\nabla^2\varphi+q=0$. With k=1 and q=0, it is referred to as Laplace's equation, $\nabla^2\varphi=0$.

If φ is electric potential, k is permittivity and q is charge density, then (1.1) is the electrostatic field equation.

If φ is temperature, k is thermal conductivity and q is heat source, then (1.1) is the heat equation.

If we identify derivatives of φ with fluid velocities,

$$u=\frac{\partial\varphi}{\partial x},\quad v=\frac{\partial\varphi}{\partial y}$$

then (1.1) is the potential flow equation.

In most cases, we can identify $-k\nabla\varphi$ with the flux of some quantity such as heat, mass or a chemical. (1.1) then says that the variation of the rate of transfer of the relevant quantity is equal to the local source (or sink) of the quantity.

If we integrate the divergence term by parts (or equivalently, apply the divergence theorem), we get

(1.2) $$\iiint_V \nabla\bullet(k\nabla\varphi)dV=\oint\!\!\oint_S \vec{n}\bullet(k\nabla\varphi)dS=-\iiint_V q\,dV$$

That is, the total interior source is equal to the net flow across the outer boundary.

In a FlexPDE script, the equation (1.1) is represented simply as

```
Div(k*grad(phi)) + q = 0
```

The boundary flow $\vec{n}\bullet(k\nabla\varphi)$ is represented in FlexPDE by the Natural boundary condition,

```
Natural(phi) = <boundary flux>
```

The simplest form of the natural boundary condition is the insulating or "no flow" boundary,

```
Natural(phi) = 0.
```

4.2.1 Electrostatic Fields in 2D

Let us as a first example construct the electrostatic field equation for an irregularly shaped block of high-dielectric material suspended in a low-dielectric material between two charged plates.

First we must present a title:

```
title
  'Electrostatic Potential'
```

Next, we must name the variables in our problem:

```
variables
  V
```

We will need the value of the permittivity:

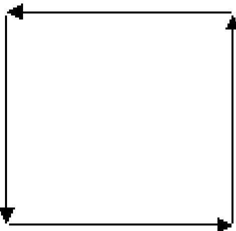
```
definitions
  eps = 1
```

The equation is as presented above, using the div and grad operators in place of $\nabla\bullet$ and ∇ :

```
equations
  div(eps*grad(V)) = 0
```

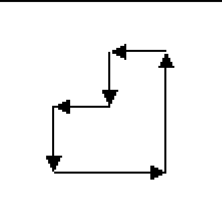
The domain will consist of two regions; the bounding box containing the entire space of the problem, with charged plates top and bottom:

```
boundaries
  region 1
    start (0,0)
    value(V) = 0
        line to (1,0)
    natural(V) = 0
        line to (1,1)
    value(V) = 100
        line to (0,1)
    natural(V) = 0
        line to close
```



and the imbedded dielectric:

```
region 2
 eps = 50
 start (0.4,0.4)
 line to (0.8,0.4)
      to (0.8,0.8)
       to (0.6,0.8)
       to (0.6,0.6)
       to (0.4,0.6)
       to close
```



Notice that we have used the insulating form of the natural boundary condition on the sides of the bounding box, with specified potentials top (100) and bottom (0).

We have specified a permittivity of 50 in the imbedded region. (Since we are free to multiply through the equation by the free-space permittivity ε_0, we can interpret the value as relative permittivity or dielectric constant.)

What will happen at the boundary between the dielectric and the air? If we apply equation (1.2) and integrate around the dielectric body, we get

$$\oint_l \vec{n} \bullet (k \nabla \varphi) dl = \iint_A q dA = 0$$

If we perform this integration just inside the boundary of the dielectric, we must use k = 50, whereas just outside the boundary, we must use k = 1. Yet both integrals must yield the same result. It therefore follows that the interface condition at the boundary of the dielectric is

$$\vec{n} \bullet (k \nabla \varphi)_{inside} = \vec{n} \bullet (k \nabla \varphi)_{outside} .$$

Since the electric field vector is $\vec{E} = \nabla \varphi$ and the electric displacement is $\vec{D} = \varepsilon \vec{E}$, we have the condition that the normal component of the electric displacement is continuous across the interface, as required by Maxwell's equations.

We want to see what is happening while the problem is being solved, so we add a monitor of the potential:

```
monitors
    contour(V) as 'Potential'
```

At the end of the problem we would like to save as graphical output the computation mesh, a contour plot of the potential, and a vector plot of the electric field:

```
plots
  grid(x,y)
  contour(V) as 'Potential'
  vector(-dx(V),-dy(V)) as 'Electric Field'
```

The problem specification is complete, so we end the script:

```
end
```

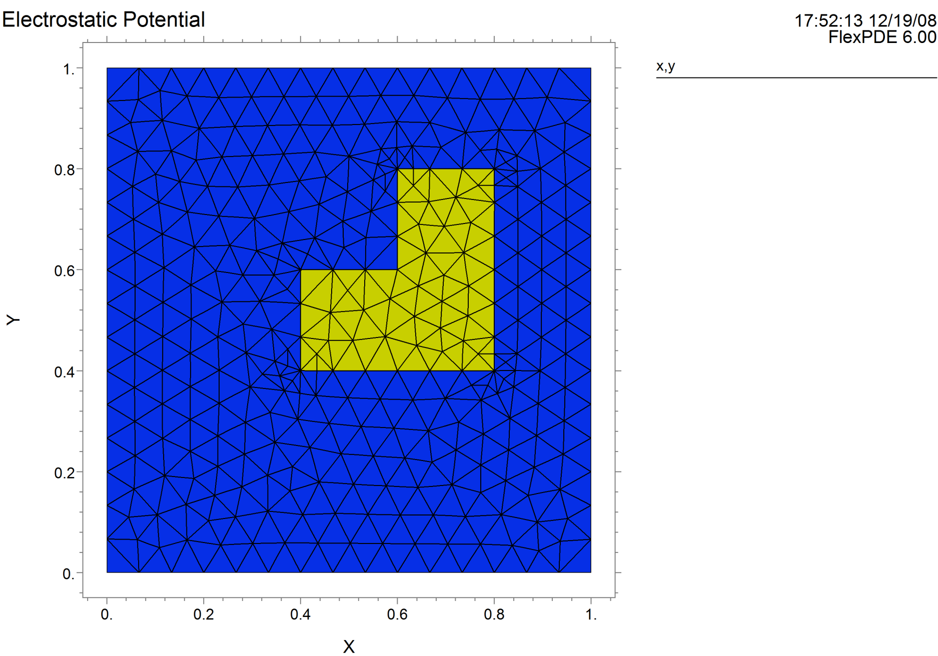
Putting all these sections together, we have the complete script for the dielectric problem:

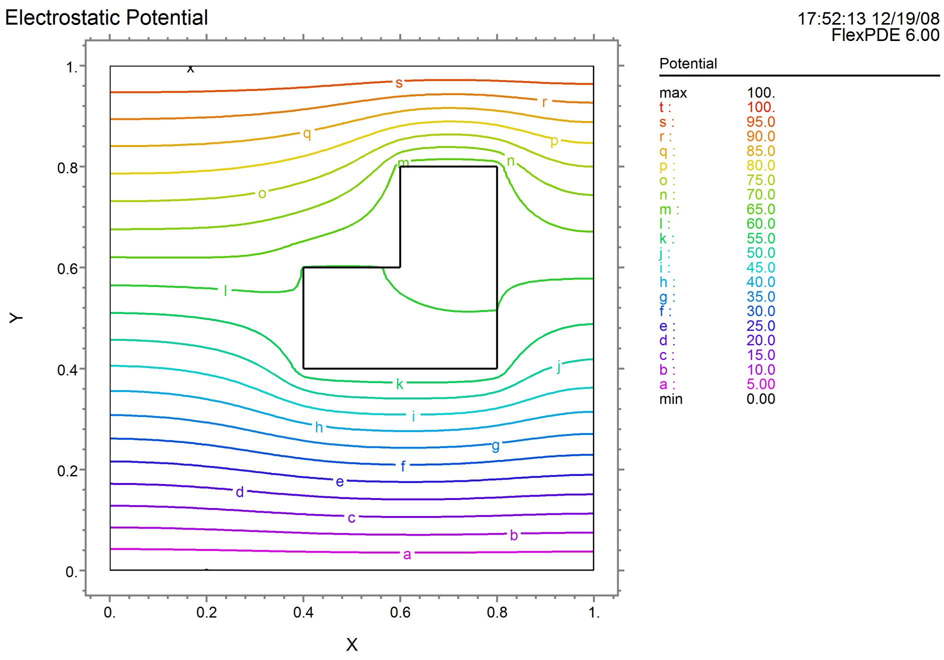
See also "Samples | Applications | Electricity | Dielectric.pde" 300
See also "Samples | Applications | Electricity | Fieldmap.pde" 300

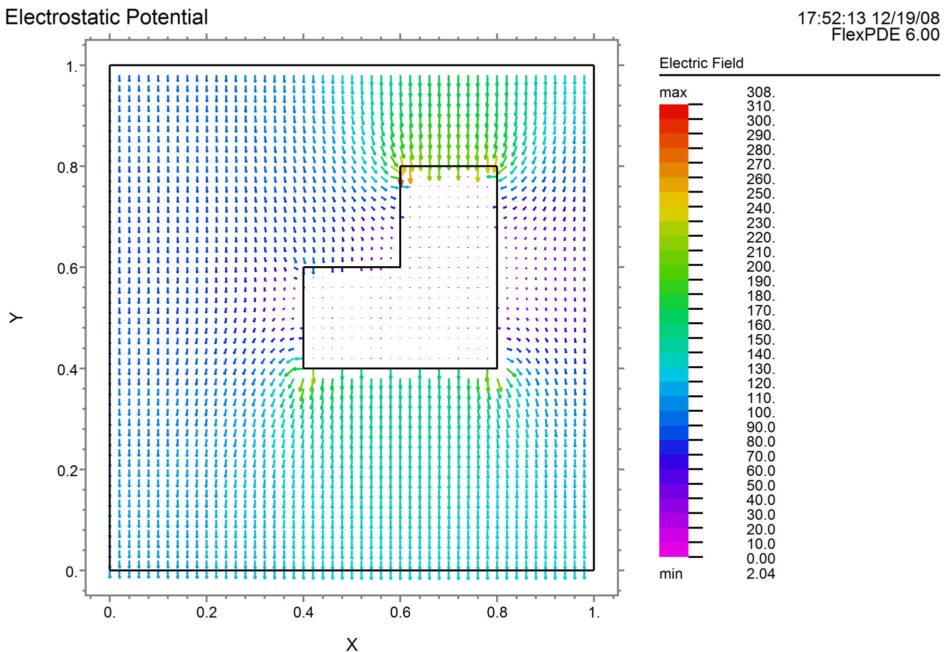
Descriptor 1.1: Dielectric.pde

```
title
  'Electrostatic Potential'
variables
  V
definitions
  eps = 1
equations
  div(eps*grad(V)) = 0
boundaries
  region 1
    start (0,0)
    value(V) = 0             line to (1,0)
    natural(V) = 0           line to (1,1)
    value(V) = 100           line to (0,1)
    natural(V) = 0           line to close
  region 2
    eps = 50
    start (0.4,0.4)
    line to (0.8,0.4) to (0.8,0.8)
       to (0.6,0.8) to (0.6,0.6)
       to (0.4,0.6) to close
monitors
  contour(V) as 'Potential'
plots
  grid(x,y)
  contour(V) as 'Potential'
  vector(-dx(V),-dy(V)) as 'Electric Field'
end
```

The output plots from running this script are as follows:


Electrostatic Potential
17:52:13 12/19/08
FlexPDE 6.00
x,y
Y
X
dielectric: Grid#2 P2 Nodes=1313 Cells=626 RMS Err= 9.5e-4


Electrostatic Potential
17:52:13 12/19/08
FlexPDE 6.00
Potential
max 100.
t : 100.
s : 95.0
r : 90.0
q : 85.0
p : 80.0
o : 75.0
n : 70.0
m : 65.0
l : 60.0
k : 55.0
j : 50.0
i : 45.0
h : 40.0
g : 35.0
f : 30.0
e : 25.0
d : 20.0
c : 15.0
b : 10.0
a : 5.00
min 0.00
Y
X
dielectric: Grid#2 P2 Nodes=1313 Cells=626 RMS Err= 9.5e-4
Integral= 52.85127



4.2.2 Electrostatics in 3D

We can convert this example quite simply to a three dimensional calculation. The modifications that must be made are:

- Specify cartesian3 coordinates.
- Add an extrusion section listing the dividing surfaces.
- Provide boundary conditions for the end faces.
- Qualify plot commands with the cut plane in which the plot is to be computed.

In the following descriptor, we have divided the extrusion into three layers. The dielectric constant in the first and third layer are left at the default of k=1, while layer 2 is given a dielectric constant of 50 in the dielectric region only.

A contour plot of the potential in the plane x=0 has been added, to show the resulting vertical cross section. The plots in the z=0.15 plane reproduce the plots shown above for the 2D case.

Modifications to the 2D descriptor are shown in red.

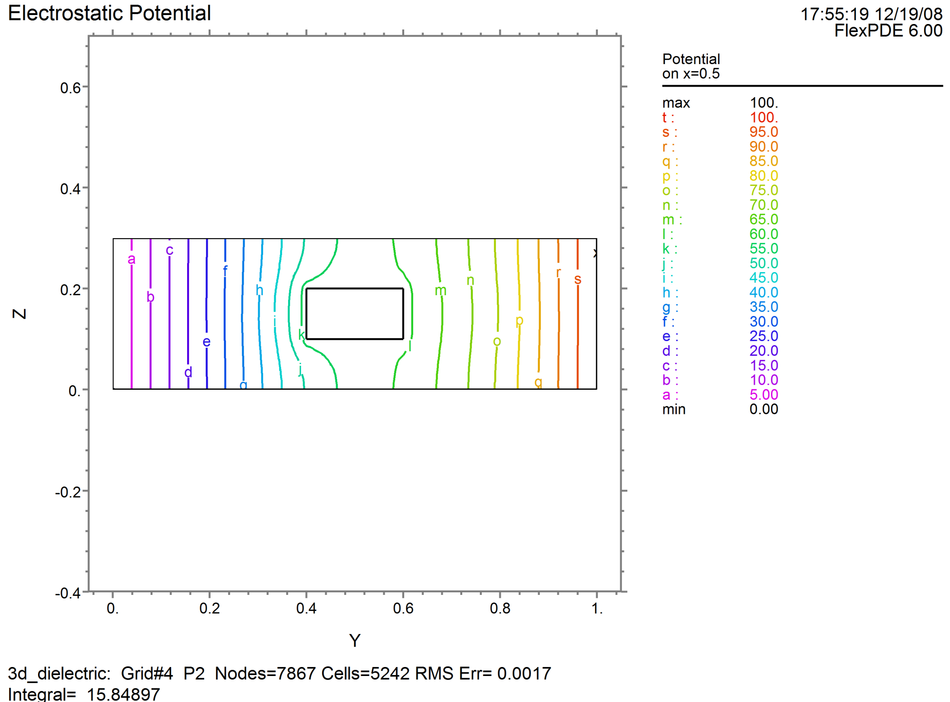
See also"Samples | Applications | Electricity | 3D_Dielectric.pde" 298

Descriptor 1.2: 3D Dielectric.pde

```
title
  'Electrostatic Potential'

coordinates
  cartesian3

variables
```

```
  V

definitions
  eps = 1

equations
  div(eps*grad(V)) = 0

extrusion
  surface "bottom"  z=0
  surface "dielectric_bottom" z=0.1
    layer "dielectric"
  surface "dielectric_top"  z=0.2
  surface "top"  z=0.3

boundaries

  surface "bottom" natural(V)=0
  surface "top" natural(V)=0

  region 1
    start (0,0)
    value(V) = 0       line to (1,0)
    natural(V) = 0     line to (1,1)
    value(V) = 100     line to (0,1)
    natural(V) = 0     line to close

  region 2
    layer "dielectric"  eps = 50
      start (0.4,0.4)
      line to (0.8,0.4) to (0.8,0.8)
          to (0.6,0.8) to (0.6,0.6)
          to (0.4,0.6) to close

monitors
  contour(V) on z=0.15 as 'Potential'

plots
  contour(V) on z=0.15 as 'Potential'
  vector(-dx(V),-dy(V)) on z=0.15 as 'Electric Field'
  contour(V) on x=0.5 as 'Potential'

end
```

The following potential plot on x=0 shows the vertical cross section of the extruded domain. Notice that the potential pattern is not symmetric, due to the influence of the extended leg of the dielectric in the y direction.



4.2.3 Capacitance per Unit Length in 2D Geometry

—Submitted by J.B. Trenholme

This problem illustrates the calculation of capacitance per unit length in a 2D X-Y geometry extended indefinitely in the Z direction. The capacitance is that between a conductor enclosed in a dielectric sheath and a surrounding conductive enclosure. In addition to these elements, there is also another conductor (also with a dielectric sheath) that is "free floating" so that it maintains zero net charge and assumes a potential that is consistent with that uncharged state.

We use the potential V as the system variable, from which we can calculate the electric field $\vec{E} = \nabla V$ and displacement $\vec{D} = \varepsilon \vec{E}$, where ε is the local permittivity and may vary with position.

In steady state, in charge-free regions, Maxwell's equation then becomes

$$\nabla \bullet \vec{D} = \nabla \bullet (\varepsilon \vec{E}) = \nabla \bullet (\varepsilon \nabla V) = 0 \, .$$

We impose value boundary conditions on V at the surfaces of the two conductors, so that we do not have to deal with regions that contain charge.

The metal in the floating conductor is "faked" with a fairly high permittivity, which has the effect of driving the interior field and field energy to near zero. The imposition of (default) natural boundary conditions then keeps the field normal to the surface of the conductor, as Maxwell requires. Thus we get a good answer without having to solve for the charge on the floating conductor, which would be a real pain due to its localization on the surface of the conductor.

The capacitance can be found in two ways. If we know the charge Q on the conductor at fixed potential V,

we solve

$Q = CV$ to get $C = Q/V$. We know V because it is imposed as a boundary condition, and we can find Q from the fact that

$$\oint_s \vec{n} \bullet \vec{D} = Q$$

where the integral is taken over a surface enclosing a volume and Q is the charge in the volume.

Alternatively, we can use the energy relation $W = \frac{1}{2} C V^2$ to get $C = 2W / V^2$. We find the energy W by integrating the energy density $\frac{1}{2} \vec{E} \bullet \vec{D}$ over the area of the problem.

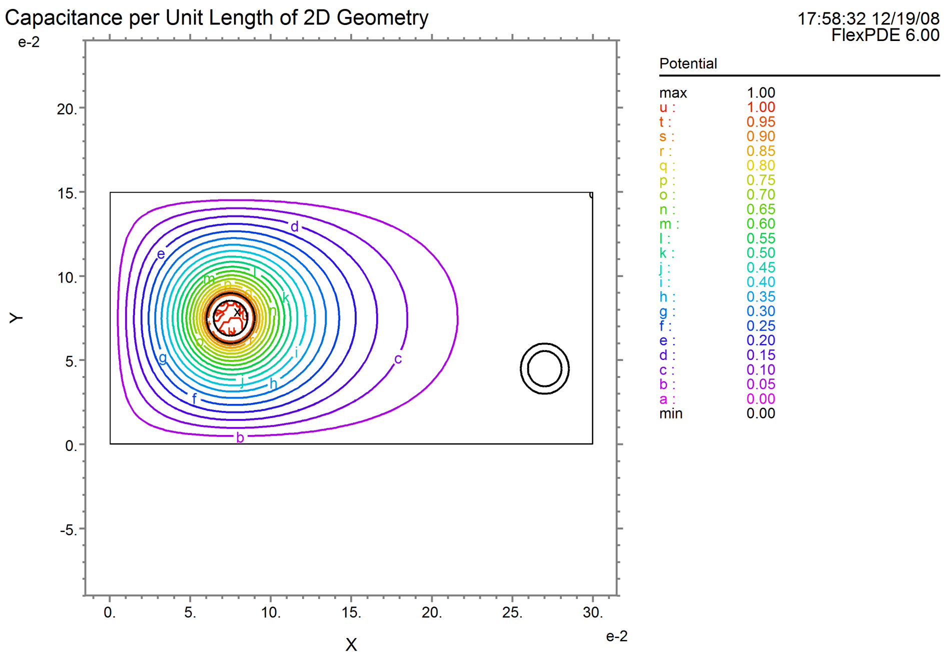
See also "Samples | Applications | Electricity | Capacitance.pde" 299

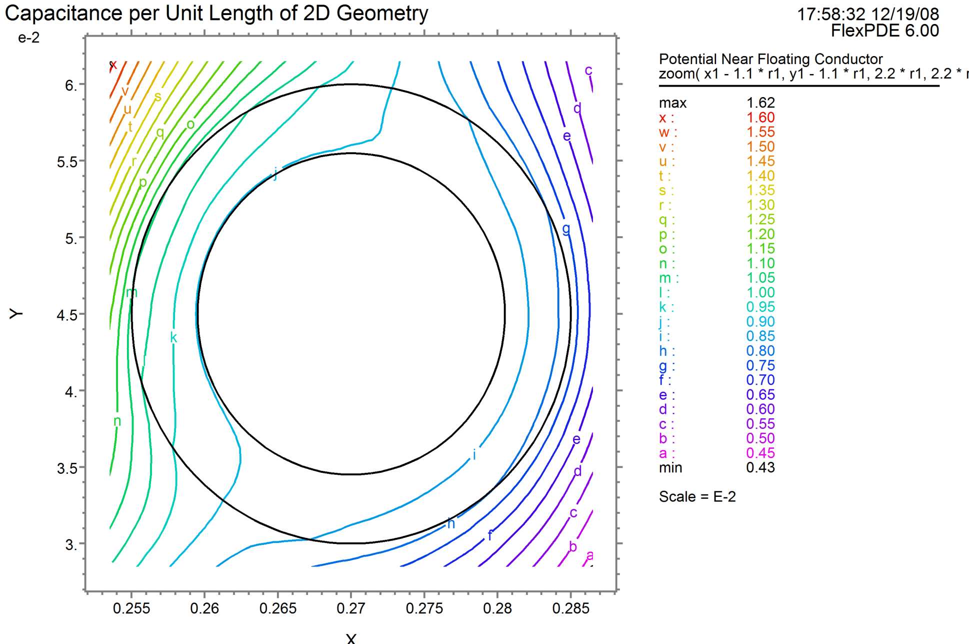
Descriptor 1.3: Capacitance.pde

```
TITLE 'Capacitance per Unit Length of 2D Geometry'
{ 17 Nov 2000 by John Trenholme }

SELECT
  errlim 1e-4
  thermal_colors on
  plotintegrate off

VARIABLES
  V

DEFINITIONS
  mm = 0.001             ! meters per millimeter
  Lx = 300 * mm          ! enclosing box dimensions
  Ly = 150 * mm
  b = 0.7                ! fractional radius of conductor
  ! position and size of cable at fixed potential:
  x0 = 0.25 * Lx
  y0 = 0.5 * Ly
  r0 = 15 * mm
  x1 = 0.9 * Lx
  y1 = 0.3 * Ly
  r1 = r0
  epsr                   ! relative permittivity
  epsd = 3               ! epsr of cable dielectric
  epsmetal = 1000        ! fake metallic conductor
  eps0 = 8.854e-12       ! permittivity of free space
  eps = epsr * eps0
  v0 = 1                 ! fixed potential of the cable

  ! field energy density:
  energyDensity = dot( eps * grad( v), grad( v) )/2
```

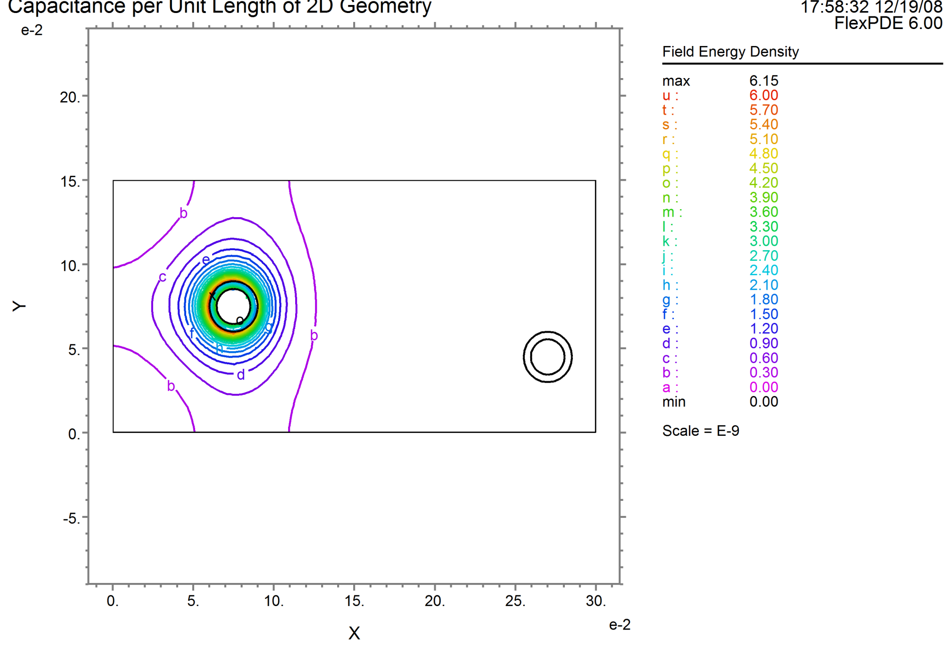
```
EQUATIONS
  div( eps * grad( v) ) = 0

BOUNDARIES
  region 1  'inside'  epsr = 1
    start  'outer'  ( 0, 0)  value( v) = 0
    line to (Lx,0) to (Lx,Ly) to (0,Ly) to close
  region 2  'diel0'  epsr = epsd
    start  'dieb0'  (x0+r0, y0)
    arc ( center = x0, y0) angle = 360
  region 3  'cond0'  epsr = 1
    start  'conb0'  (x0+b*r0, y0)  value(v) = v0
    arc ( center = x0, y0) angle = 360
  region 4  'diel1'  epsr = epsd
    start  'dieb1'  ( x1+r1, y1)
    arc ( center = x1, y1) angle = 360
  region 5  'cond1'  epsr = epsmetal
    start  'conb1'  ( x1+b*r1, y1)
    arc ( center = x1, y1) angle = 360

PLOTS
  contour( v) as 'Potential'
  contour( v) as 'Potential Near Driven Conductor'
    zoom(x0-1.1*r0, y0-1.1*r0, 2.2*r0, 2.2*r0)
  contour( v) as 'Potential Near Floating Conductor'
    zoom(x1-1.1*r1, y1-1.1*r1, 2.2*r1, 2.2*r1)
  elevation( v) from ( 0,y0) to ( x0, y0)
   as 'Potential from Wall to Driven Conductor'
  elevation( v) from ( x0, y0) to ( x1, y1)
   as 'Potential from Driven to Floating Conductor'
  vector( grad( v)) as 'Field'
  contour( energyDensity) as 'Field Energy Density'
  contour( energyDensity)
     zoom( x1-1.2*r1, y1-1.2*r1, 2.4*r1, 2.4*r1)
     as 'Field Energy Density Near Floating Conductor'
  elevation( energyDensity)
     from (x1-2*r1, y1) to ( x1+2*r1, y1)
    as 'Field Energy Density Near Floating Conductor'
  contour( epsr) paint on "inside"
     as 'Definition of Inside'

SUMMARY
  report sintegral(normal(eps*grad(v)),'conb0', 'diel0')
     as 'Driven charge'
  report sintegral(normal(eps*grad(v)),'outer','inside')
     as 'Outer charge'
  report sintegral(normal(eps*grad(v)),'conb1','diel1')
     as 'Floating charge'
  report sintegral(normal(eps*grad(v)),'conb0','diel0')/v0
     as 'Capacitance (f/m)'
  report integral( energyDensity, 'inside')
     as 'Energy (J/m)'
```

```
  report 2 * integral( energyDensity, 'inside') / v0^2
    as 'Capacitance (f/m)'
  report 2 * integral(energyDensity)/(v0*
    sintegral( normal(eps*grad(v)), 'conb0', 'diel0'))
    as 'cap_by_energy / cap_by_charge'

END
```





capacitance: Grid#4 P2 Nodes=4365 Cells=2134 RMS Err= 7.7e-5



capacitance: Grid#4 P2 Nodes=4365 Cells=2134 RMS Err= 7.7e-5

Capacitance per Unit Length of 2D Geometry

18:06:17 12/19/08
FlexPDE 6.00

SUMMARY

Driven charge= 2.942077e-11
Outer charge= -2.951385e-11
Floating charge= -2.146545e-15
Capacitance (f/m)= 2.942077e-11
Energy (J/m)= 1.384088e-11
Capacitance (f/m)= 2.768177e-11
cap_by_energy / cap_by_charge= 1.004412

capacitance: Grid#4 P2 Nodes=4365 Cells=2134 RMS Err= 7.7e-5

4.3 Magnetostatics

From Maxwell's equations in a steady-state form we have

(2.1)
$$\nabla \times \vec{H} = \vec{J}$$
$$\nabla \bullet \vec{B} = 0$$
$$\nabla \bullet \vec{J} = 0$$

where $\vec{H}$ is the magnetic field intensity, $\vec{B} = \mu\vec{H}$ is the magnetic induction, μ is the magnetic permeability and $\vec{J}$ is the current density.

The conditions required by Maxwell's equations at a material interface are

$$\vec{n} \times \vec{H}_1 = \vec{n} \times \vec{H}_2$$
$$\vec{n} \bullet \vec{B}_1 = \vec{n} \bullet \vec{B}_2$$
(2.2)

It is sometimes fruitful to use the magnetic field quantities directly as variables in a model. However, eq. (2.2) shows that the tangential components of $\vec{H}$ are continuous across an interface, while the normal components of $\vec{B}$ are continuous.

The finite element method used by FlexPDE has a single value of each variable on an interface, and therefore requires that the quantities chosen for system variables must be continuous across the interface.

In special cases, it may be possible to choose components of $\vec{B}$ or $\vec{H}$ which satisfy this continuity requirement. We could, for example model B_x in a problem where material interfaces are normal to x. In the general case, however, meeting the continuity requirements can be impossible.

It is common in Magnetostatics to use instead of the field quantities the magnetic vector potential $\vec{A}$, defined as

(2.3) $$\vec{B} = \nabla \times \vec{A}.$$

This definition automatically enforces $\nabla \bullet \vec{B} = 0$. Furthermore, $\vec{A}$ can be shown to be continuous everywhere in the domain, and can represent the conditions (2.2) correctly.

$\vec{A}$ can be derived from Ampere's Law, and shown to be the integrated effect at each point of all the current loops active in the domain. In this derivation, $\vec{A}$ will have components parallel to the components of $\vec{J}$, so that it can be determined a priori which components of $\vec{A}$ must be represented.

Eq. (2.3) alone is not sufficient to uniquely define $\vec{A}$. It must be supplemented by a definition of $\nabla \bullet \vec{A}$ to be unique. This definition (the "gauge condition") is usually taken to be $\nabla \bullet \vec{A} = 0$ ("Coulomb gauge"), a definition consistent with the derivation of $\vec{A}$ from Ampere's Law. Other definitions are useful in some applications. It is not important what the qauge condition is; in all cases $\nabla \times \vec{A}$, and therefore the field quantities, remain the same.

Combining eq. (2.1) with (2.3) gives

(2.4) $$\nabla \times ((\nabla \times \vec{A}) / \mu) = \vec{J}$$

In cases with multiple materials, where μ can take on different values, it is important to keep the μ inside the curl operator, because it is the integration of this term by parts that gives the correct jump conditions at the material interface.

Applying eq. (0.5) we have

(2.5) $$\iiint_V \nabla \times \left(\left(\nabla \times \vec{A} \right) / \mu \right) dV = \iiint_V \nabla \times \vec{H} dV = \oiint_S \vec{n} \times \vec{H} dS,$$

so that the Natural boundary condition defines $\vec{n} \times \vec{H}$ on external boundaries, and $\vec{n} \times \vec{H}$ is assumed continuous across internal boundaries, consistent with Maxwell's equations.

4.3.1 A Magnet Coil in 2D Cylindrical Coordinates

As a first example, we will calculate the magnetic field created by a coil, using 2D cylindrical (r,z) geometry. We will apply current only in the azimuthal direction, so the only nonzero component of $\vec{A}$ will be the azimuthal component A_ϕ. With only a single component normal to the computational plane, the gauge

condition is automatically satisfied, since $\nabla \bullet \vec{A} = \frac{1}{r}\frac{\partial A_\phi}{\partial \phi} = 0$

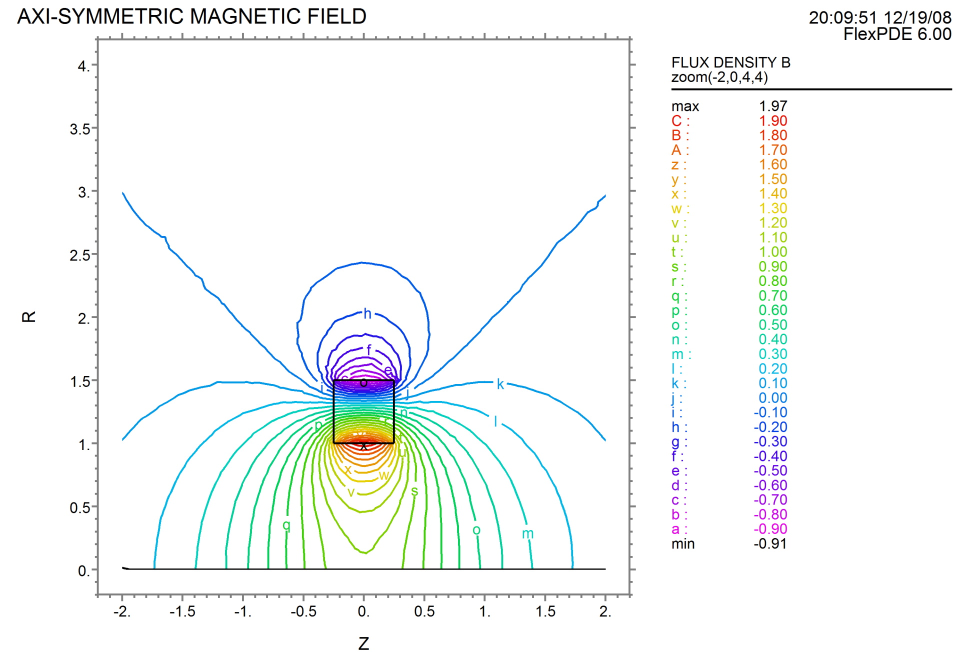
In the descriptor which follows, note that we have chosen to align the cylindrical axis with the horizontal plot axis. FlexPDE uses a right-hand coordinate system, so in this case positive J_ϕ is outward from the plot page.

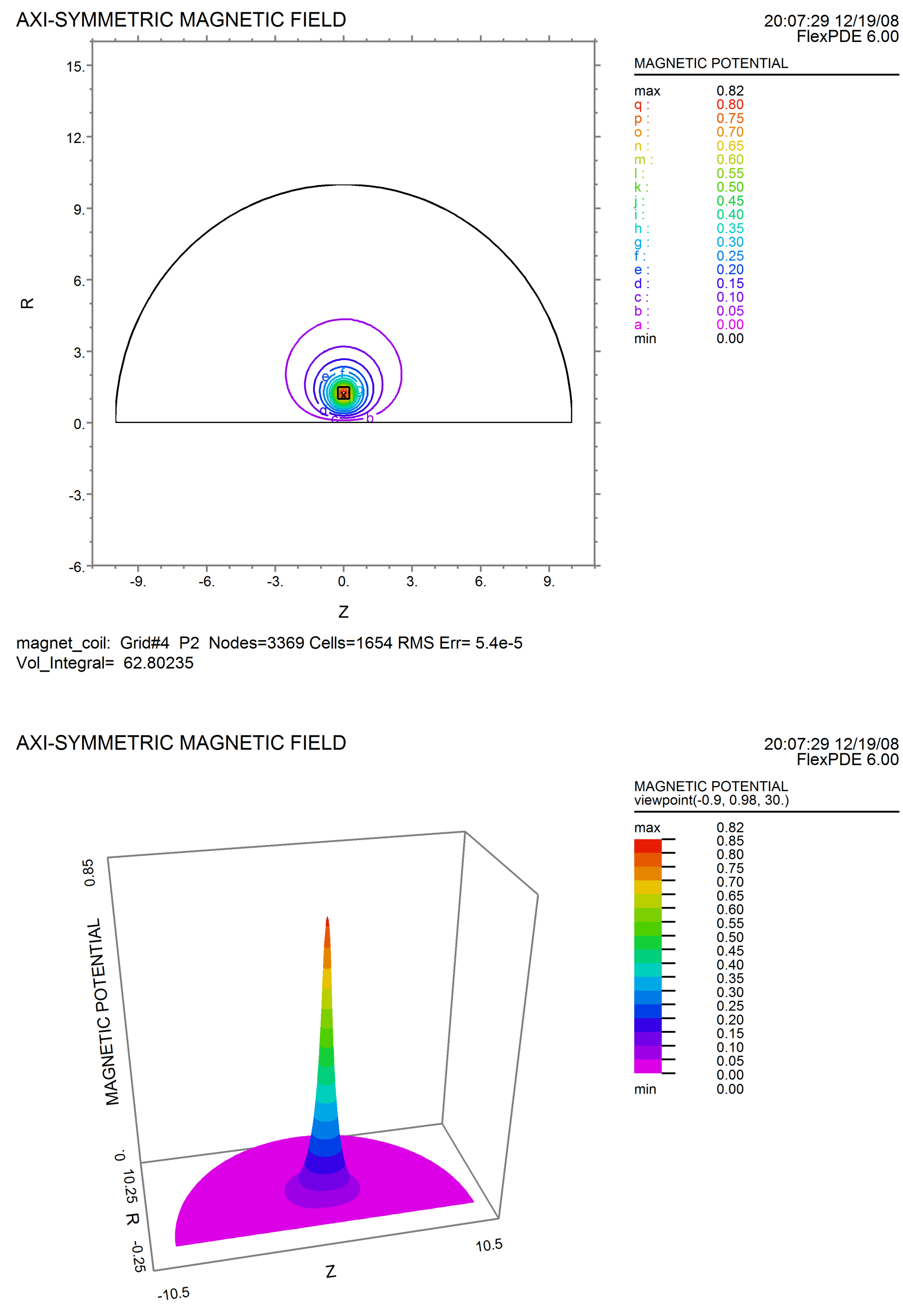
See also "Samples | Applications | Magnetism | Magnet_Coil.pde" 350

Descriptor 2.1: Magnet_Coil.pde

```
Title 'AXI-SYMMETRIC MAGNETIC FIELD'

Coordinates
  xcylinder(Z,R)

Variables
  Aphi    { azimuthal component of the vector potential }

Definitions
  mu = 1              { the permeability }
  J = 0               { global source term defaults to zero }
  current = 10        { the source value in the coil }
  Br = -dz(Aphi)      { definitions for plots }
  Bz = dr(r*Aphi)/r

Equations
  Curl(curl(Aphi)/mu) = J

Boundaries
  Region 1
    start(-10,0)
    value(Aphi) = 0     { specify A=0 along axis }
    line to (10,0)
    value(Aphi) = 0     { H x n = 0 on distant sphere }
    arc(center=0,0) angle 180 to close
  Region 2
    J = current         { redefine source value }
    start (-0.25,1)
    line to (0.25,1) to (0.25,1.5)
         to (-0.25,1.5) to close

Monitors
  contour(Bz) zoom(-2,0,4,4) as 'FLUX DENSITY B'
  contour(Aphi) as 'Potential'

Plots
  grid(z,r)
  contour(Bz)  as 'FLUX DENSITY B'
  contour(Bz) zoom(-2,0,4,4)  as 'FLUX DENSITY B'
```

```
elevation(Aphi, dr(Aphi), Aphi/r, Bz)
  from (0,0) to (0,1) as 'Near Axis'
vector(Bz,Br) as 'FLUX DENSITY B'
vector(Bz,Br) zoom(-2,0,4,4) as 'FLUX DENSITY B'
contour(Aphi) as 'MAGNETIC POTENTIAL'

contour(Aphi) zoom(-2,0,4,4)  as 'MAGNETIC POTENTIAL'
surface(Aphi)  as 'MAGNETIC POTENTIAL'
  viewpoint (-1,1,30)

End
```





4.3.2 Nonlinear Permeability in 2D

In the following 2D Cartesian example, a current-carrying copper coil is surrounded by a ferromagnetic

core with an air gap. Current flows in the coil in the Z direction (out of the computation plane), and only the Z component of the magnetic vector potential is nonzero. The Coulomb gauge condition is again satisfied automatically. We assume a symmetry plane along the X-axis, and impose $A_z = 0$ along the remaining sides. The relative permeability is $\mu = 1$ in the air and the coil, while in the core it is given by

$$\mu = \frac{\mu_{max}}{1 + C(\nabla A_z)^2} + \mu_{min},$$

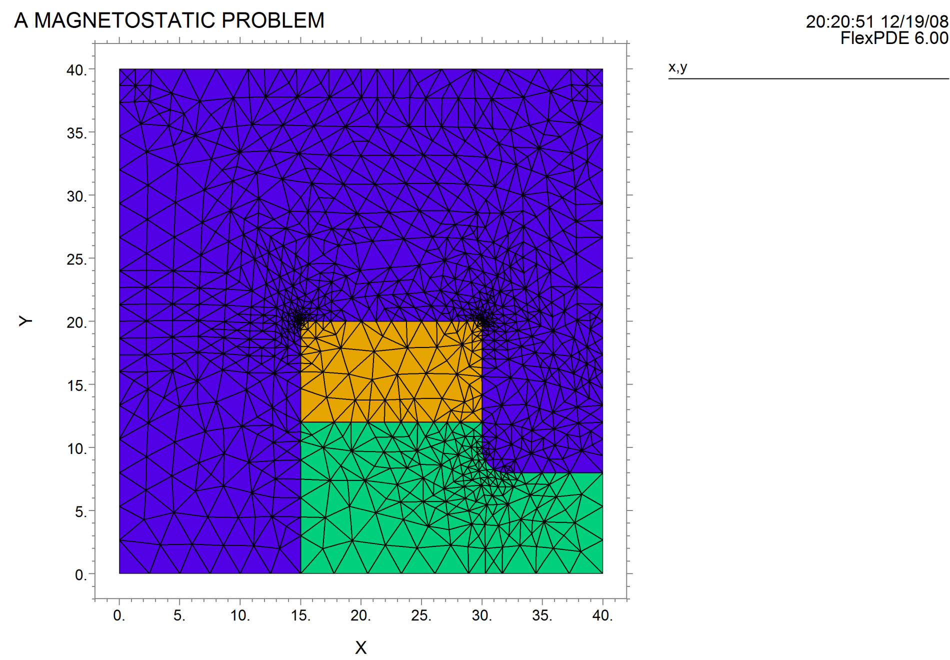
with parameters giving a behavior similar to transformer steel.

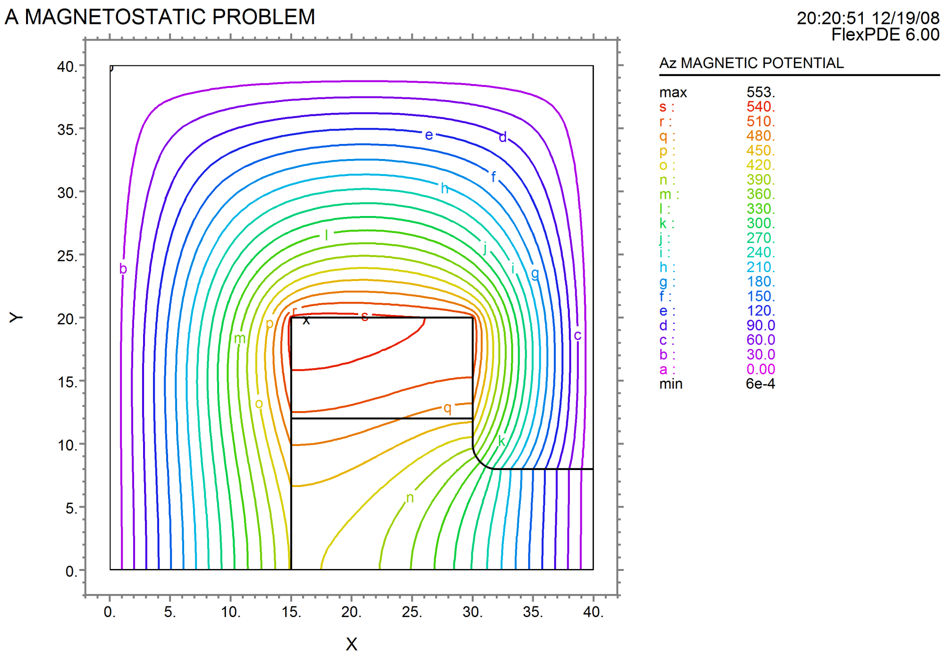
See also "Samples | Applications | Magnetism | Saturation.pde" 352

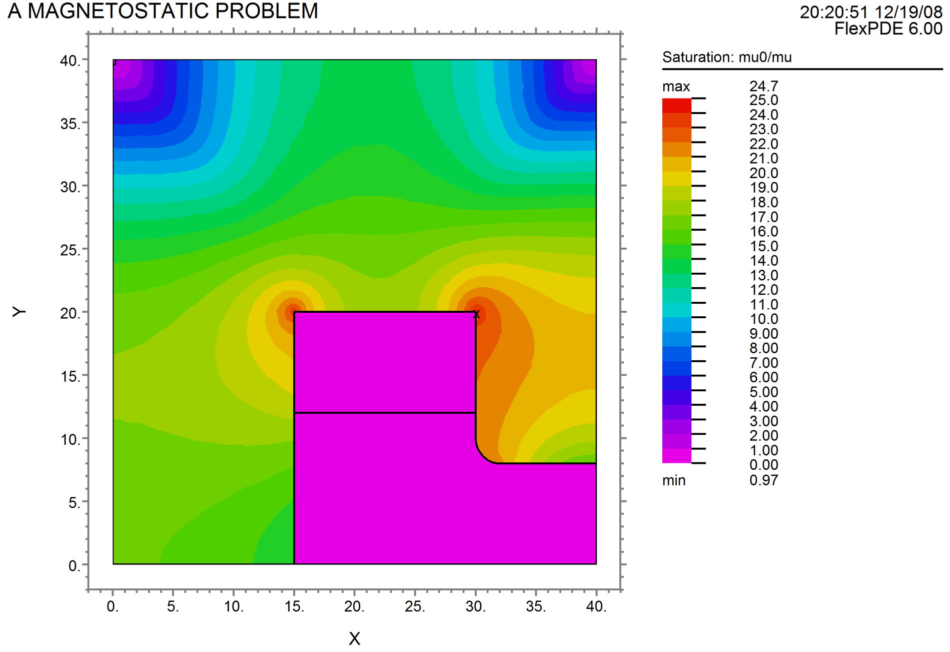
Descriptor 2.2: Saturation.pde

```
Title "A MAGNETOSTATIC PROBLEM"

Select
  errlim = 1e-4

Variables
  A

Definitions
  mu = 1              { default to air}
  mu0 = 1             { for saturation plot }
  mu_max = 5000
  mu_min = 200
  mucore = mu_max/(1+0.05*grad(A)^2) + mu_min
  S = 0
  current = 2
  y0 = 8

Equations
   curl(curl(A)/mu) = S

Boundaries
  Region 1           { The IRON core }
    mu = mucore
    mu0 = mu_max
    start(0,0)
    natural(A) = 0   line to (40,0)
    value(A) = 0     line to (40,40) to (0,40) to close

  Region 2           { The AIR gap }
    mu = 1
    start (15,0)
        line to (40,0) to (40,y0) to (32,y0)
        arc (center=32,y0+2) to (30,y0+2)
        line to (30,20) to (15,20) to close

  Region 3           { The COIL }
    S = current
```

```
    mu = 1
    start (15,12)
        line to (30,12) to (30,20) to (15,20) to close

Monitors
  contour(A)

Plots
  grid(x,y)
  vector(dy(A),-dx(A)) as "FLUX DENSITY B"
  vector(dy(A)/mu, -dx(A)/mu) as "MAGNETIC FIELD H"
  contour(A)  as "Az MAGNETIC POTENTIAL"
  surface(A)  as "Az MAGNETIC POTENTIAL"
  contour(mu0/mu) painted as "Saturation: mu0/mu"

End
```




A MAGNETOSTATIC PROBLEM
20:20:51 12/19/08
FlexPDE 6.00
Az MAGNETIC POTENTIAL
max 553.
s : 540.
r : 510.
q : 480.
p : 450.
o : 420.
n : 390.
m : 360.
l : 330.
k : 300.
j : 270.
i : 240.
h : 210.
g : 180.
f : 150.
e : 120.
d : 90.0
c : 60.0
b : 30.0
a : 0.00
min 6e-4
Y
X
saturation: Grid#5 P2 Nodes=4069 Cells=1994 RMS Err= 9.7e-5
Integral= 380334.7


A MAGNETOSTATIC PROBLEM
20:20:51 12/19/08
FlexPDE 6.00
Saturation: mu0/mu
max 24.7
25.0
24.0
23.0
22.0
21.0
20.0
19.0
18.0
17.0
16.0
15.0
14.0
13.0
12.0
11.0
10.0
9.00
8.00
7.00
6.00
5.00
4.00
3.00
2.00
1.00
0.00
min 0.97
Y
X
saturation: Grid#5 P2 Nodes=4069 Cells=1994 RMS Err= 9.7e-5
Integral= 18686.64

4.3.3 Divergence Form

In two dimensional geometry with a single nonzero component of $\vec{A}$, the gauge condition $\nabla \bullet \vec{A} = 0$ is automatically satisfied. Direct application of eq. (2.4) is therefore well posed, and we can proceed without further modification.

In 3D, however, direct implementation of eq. (2.4) does not impose a gauge condition, and is therefore ill-posed in many cases. One way to address this problem is to convert the equation to divergence form using the vector identity

(2.6) $$\nabla \times (\nabla \times \vec{A}) = \nabla(\nabla \bullet \vec{A}) - \nabla^2 \vec{A}.$$

As long as μ is piecewise constant we can apply (2.6) together with the Coulomb gauge $\nabla \bullet \vec{A} = 0$ to rewrite (2.4) as

(2.7) $$\nabla \bullet \left(\frac{\nabla \vec{A}}{\mu} \right) + \vec{J} = 0$$

If μ is variable, we can generalize eq. (2.6) to the relation

(2.8) $$\nabla \times \left(\frac{\nabla \times \vec{A}}{\mu} \right) = \nabla \bullet \left(\frac{\nabla \vec{A}}{\mu} \right)^T - \nabla \bullet \left(\frac{\nabla \vec{A}}{\mu} \right)$$

We assert without proof that there exists a gauge condition $\nabla \bullet \vec{A} = F(x, y, z)$ which forces

(2.9) $$\nabla \bullet \left(\frac{\nabla \vec{A}}{\mu} \right)^T = 0.$$

The equations governing F can be stated as

$$\frac{\partial}{\partial x}\left(\frac{F}{\mu}\right) = \frac{\partial}{\partial x}\left(\frac{1}{\mu}\frac{\partial A_y}{\partial y} + \frac{1}{\mu}\frac{\partial A_z}{\partial z}\right) - \frac{\partial}{\partial y}\left(\frac{1}{\mu}\frac{\partial A_y}{\partial x}\right) - \frac{\partial}{\partial z}\left(\frac{1}{\mu}\frac{\partial A_z}{\partial x}\right)$$

$$\frac{\partial}{\partial y}\left(\frac{F}{\mu}\right) = \frac{\partial}{\partial y}\left(\frac{1}{\mu}\frac{\partial A_x}{\partial x} + \frac{1}{\mu}\frac{\partial A_z}{\partial z}\right) - \frac{\partial}{\partial x}\left(\frac{1}{\mu}\frac{\partial A_x}{\partial y}\right) - \frac{\partial}{\partial z}\left(\frac{1}{\mu}\frac{\partial A_z}{\partial y}\right)$$

$$\frac{\partial}{\partial z}\left(\frac{F}{\mu}\right) = \frac{\partial}{\partial z}\left(\frac{1}{\mu}\frac{\partial A_x}{\partial x} + \frac{1}{\mu}\frac{\partial A_y}{\partial y}\right) - \frac{\partial}{\partial x}\left(\frac{1}{\mu}\frac{\partial A_x}{\partial z}\right) - \frac{\partial}{\partial y}\left(\frac{1}{\mu}\frac{\partial A_y}{\partial z}\right)$$

It is not necessary to solve these equations; we show them merely to indicate that F embodies the commutation characteristics of the system. The value of F is implied by the assertion (2.9). Clearly, when μ is constant, the equations reduce to $\nabla F = 0$, for which $F = 0$ is a solution.

Using the definition (2.9) we can again write the divergence form

(2.10) $$\nabla\bullet\left(\nabla\vec{A}/\mu\right)+J=0$$.

4.3.4 Boundary Conditions

In converting the equation to a divergence, we have modified the interface conditions. The natural boundary condition for each component equation of (2.10) is now the normal component of the argument of the divergence:

$$Natural(A_x)=\vec{n}\bullet\nabla A_x/\mu$$

$$Natural(A_y)=\vec{n}\bullet\nabla A_y/\mu$$

(2.11) $$Natural(A_z)=\vec{n}\bullet\nabla A_z/\mu$$

The default interior interface condition assumes component-wise continuity of the surface terms across the interface.

Of the conditions (2.2) required by Maxwell's equations at an interface, the first describes the tangential components of $\vec{H}$, which by (2.3) involve the normal components of $\nabla\vec{A}$. Eq. (2.11) shows that these components scale by $1/\mu$, satisfying the tangential condition on $\vec{H}$.

The second condition is satisfied by the fact that the variables A_x, A_y, A_z have only a single representation on the boundary, requiring that their tangential derivatives, and therefore the normal component of $\vec{B}$, will be continuous across the interface.

In all cases it is important to keep the μ attached to the $\nabla\vec{A}$ term to preserve the correct interface jump conditions.

4.3.5 Magnetic Materials in 3D

In magnetic materials, we can modify the definition of $\vec{H}$ to include magnetization and write

(2.12) $$\vec{H}=\vec{B}/\mu-\vec{M}$$

We can still apply the divergence form in cases where $\vec{M}\neq 0$, but we must treat the magnetization terms specially.

The equation becomes:

(2.13) $$\nabla\bullet\left(\frac{\nabla\vec{A}}{\mu}\right)+\nabla\times\vec{M}+\vec{J}=0$$

FlexPDE does not integrate constant source terms by parts, and if $\vec{M}$ is piecewise constant the magnetization term will disappear in equation analysis. It is necessary to reformulate the magnetic term so that it can be incorporated into the divergence. We have from (2.5)

(2.14) $$\iiint_V \nabla\times\vec{M}dV = \oiint_S \vec{n}\times\vec{M}dS$$.

Magnetic terms that will obey

(2.15) $$\vec{n}\times\vec{M} = \vec{n}\bullet\overleftrightarrow{N}$$

can be formed by defining $\overleftrightarrow{N}$ as the antisymmetric dyadic

$$\overleftrightarrow{N} = \begin{pmatrix} 0 & M_z & -M_y \\ -M_z & 0 & M_x \\ M_y & -M_x & 0 \end{pmatrix}$$

Using this relation, we can write eq. (2.13) as

(2.16) $$\nabla\bullet\left(\frac{\nabla\vec{A}}{\mu}+\overleftrightarrow{N}\right)+\vec{J}=0$$

This follows because integration by parts will produce surface terms $\vec{n}\bullet\overleftrightarrow{N}$, which are equivalent to the required surface terms $\vec{n}\times\vec{M}$.

Expanded in Cartesian coordinates, this results in the three equations

$$\nabla\bullet\left(\frac{\nabla A_x}{\mu}+\vec{N}_x\right)+J_x=0$$

$$\nabla\bullet\left(\frac{\nabla A_y}{\mu}+\vec{N}_y\right)+J_y=0$$

(2.17) $$\nabla\bullet\left(\frac{\nabla A_z}{\mu}+\vec{N}_z\right)+J_z=0$$

where the $\vec{N}_i$ are the rows of $\overleftrightarrow{N}$.

In this formulation, the Natural boundary condition will be defined as the value of the normal component of the argument of the divergence, eg.

(2.18) $$natural(A_x)=\vec{n}\bullet\left(\frac{\nabla A_x}{\mu}+\vec{N}_x\right)$$.

As an example, we will compute the magnetic field in a generic magnetron. In this case, only M_z is

applied by the magnets, and as a result A_z will be zero. We will therefore delete A_z from the analysis. The outer and inner magnets are in reversed orientation, so the applied M_z is reversed in sign.

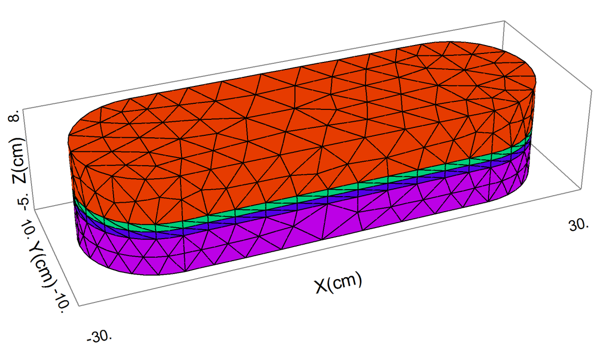
See also "Samples | Applications | Magnetism | 3D_Magnetron.pde" 346

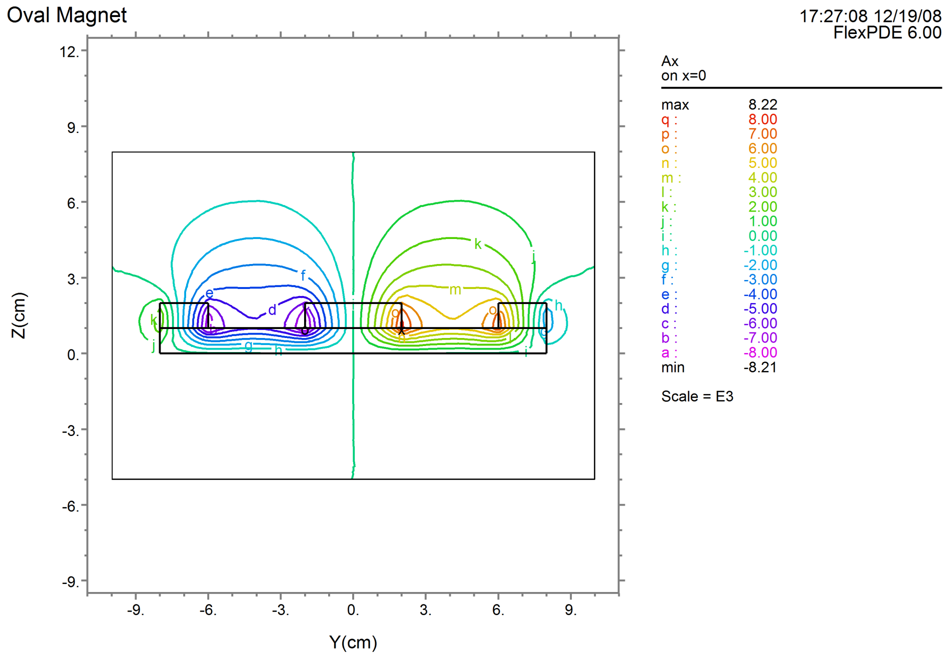
Descriptor 2.3: 3D_Magnetron.pde

```
TITLE 'Oval Magnet'

COORDINATES
  CARTESIAN3

SELECT
   alias(x) = "X(cm)"
   alias(y) = "Y(cm)"
   alias(z) = "Z(cm)"
   nodelimit = 40000
   errlim=1e-4

VARIABLES
  Ax,Ay          { assume Az is zero! }

DEFINITIONS
  MuMag=1.0                { Permeabilities: }
  MuAir=1.0
  MuSST=1000
  MuTarget=1.0
  Mu=MuAir                 { default to Air }

  MzMag = 10000            { permanent magnet strength }
  Mz = 0
  Nx = vector(0,Mz,0)
  Ny = vector(-Mz,0,0)

  B = curl(Ax,Ay,0)        { magnetic flux density }
  Bxx= -dz(Ay)
  Byy= dz(Ax)              { "By" is a reserved word. }
  Bzz= dx(Ay)-dy(Ax)

EQUATIONS
  Ax: div(grad(Ax)/mu + Nx) = 0
  Ay: div(grad(Ay)/mu + Ny) = 0

EXTRUSION
  SURFACE "Boundary Bottom"       Z=-5
  SURFACE "Magnet Plate Bottom"  Z=0
     LAYER "Magnet Plate"
  SURFACE "Magnet Plate Top"     Z=1
     LAYER "Magnet"
  SURFACE "Magnet Top"           Z=2
```

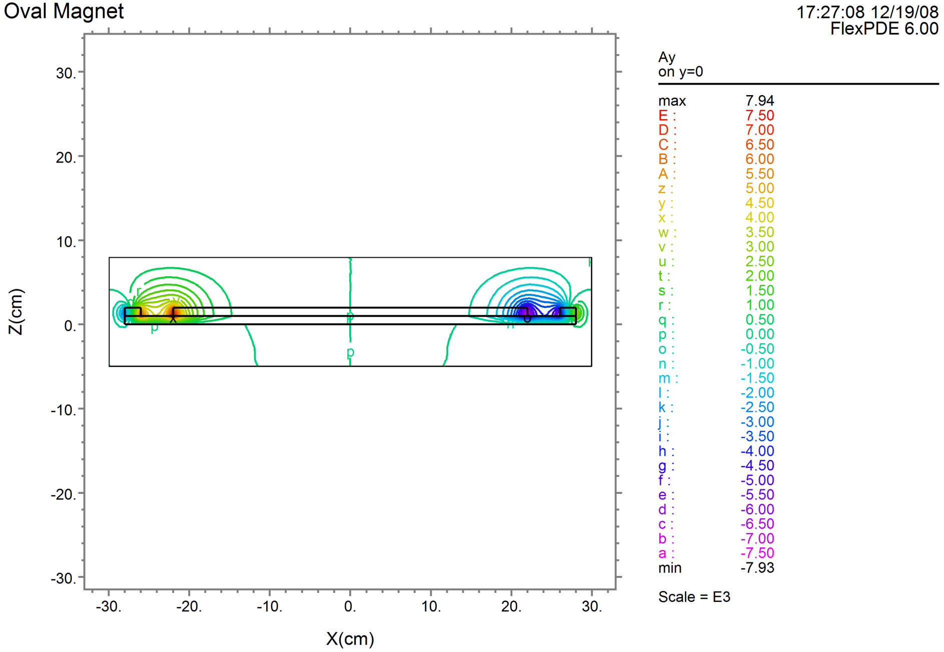
```
  SURFACE "Boundary Top"          Z=8

BOUNDARIES
  Surface "boundary bottom"
        value (Ax)=0  value(Ay)=0
  Surface "boundary top"
        value (Ax)=0 value(Ay)=0

  REGION 1     { Air bounded by conductive box }
  START (20,-10)
     value(Ax)=0  value(Ay)=0
     arc(center=20,0) angle=180
     Line TO (-20,10)
     arc(center=-20,0) angle=180
     LINE TO CLOSE

  REGION 2   { Magnet Plate Perimeter and outer magnet }
    LAYER "Magnet Plate"
     Mu=MuSST
    LAYER "Magnet"
     Mu=MuMag
     Mz=MzMag
     START (20,-8)
      arc(center=20,0) angle=180
      Line TO (-20,8)
      arc(center=-20,0) angle=180
      LINE TO CLOSE

  REGION 3     { Air }
    LAYER "Magnet Plate"
      Mu=MuSST
    START (20,-6)
      arc(center=20,0) angle=180
     Line TO (-20,6)
      arc(center=-20,0) angle=180
     LINE TO CLOSE

  REGION 4     { Inner Magnet }
    LAYER "Magnet Plate"
        Mu=MuSST
    LAYER "Magnet"
        Mu=MuMag
        Mz=-MzMag
    START (20,-2)
        arc(center=20,0) angle=180
      Line TO (-20,2)
        arc(center=-20,0) angle=180
      LINE TO CLOSE

MONITORS
  grid(x,z) on  y=0
  grid(x,y) on  z=1.01
  grid(x,z) on  y=1
```

```
PLOTS
  grid(x,y) on z=1.01
  grid(y,z) on x=0
  grid(x,z) on y=0
  contour(Ax) on x=0
  contour(Ay) on y=0
  vector(Bxx,Byy) on z=2.01 norm
  vector(Byy,Bzz) on x=0 norm
  vector(Bxx,Bzz) on y=4 norm
  contour(magnitude(Bxx,Byy,Bzz)) on z=2.01 LOG

END
```





3d_magnetron: Grid#7 P2 Nodes=184381 Cells=134358 RMS Err= 0.001
Integral= 46.91325



3d_magnetron: Grid#7 P2 Nodes=184381 Cells=134358 RMS Err= 0.001
Integral= -46.71416

4.4 Waveguides

A waveguide is any of several kinds of structure intended to direct the propagation of high-frequency electromagnetic energy along specific paths. While the analysis of bends and terminations in such a system is an essentially three-dimensional problem, the propagation in long straight segments of the guide can be reduced to a two-dimensional analysis. In this case, we assume that the guide is of uniform cross-section in the (X,Y) plane, unvarying in the Z-dimension of the propagation direction. In this configuration, we can make the assumption that the fields inside the guide may be represented as a sinusoidal oscillation in time and space, and write

$$\vec{E}(x,y,z,t) = \vec{\mathcal{E}}(x,y)\exp(i\omega t - i\gamma z)$$
$$\vec{H}(x,y,z,t) = \vec{\mathcal{H}}(x,y)\exp(i\omega t - i\gamma z) \quad (3.1)$$

It is easy to see that these expressions describe a traveling wave, since the imaginary exponential generates sines and cosines, and the value of the exponential will be the same wherever $\gamma z = \omega t$. A purely real γ implies an unattenuated propagating mode with wavelength $\lambda = 2\pi / \gamma$ along the z direction.

We start from the time-dependent form of Maxwell's equations

$$\nabla \times \vec{H} = \vec{J} + \frac{\partial \vec{D}}{\partial t} = \vec{J} + \varepsilon \frac{\partial \vec{E}}{\partial t}$$
$$\nabla \bullet \vec{B} = \nabla \bullet \left(\mu \vec{H} \right) = 0$$
$$\nabla \times \vec{E} = -\frac{\partial \vec{B}}{\partial t} = -\mu \frac{\partial \vec{H}}{\partial t}$$
$$\nabla \bullet \vec{D} = \nabla \bullet \left(\varepsilon \vec{E} \right) = \rho \quad (3.2)$$

Assume then that $\vec{J} = 0$ and $\rho = 0$, and apply (3.1) :

$$\nabla \times \vec{\mathcal{H}} = i\omega\varepsilon\vec{\mathcal{E}} \qquad \nabla \bullet \left(\mu \vec{\mathcal{H}} \right) = 0$$
$$\nabla \times \vec{\mathcal{E}} = -i\omega\mu\vec{\mathcal{H}} \qquad \nabla \bullet \left(\varepsilon \vec{\mathcal{E}} \right) = 0 \quad (3.3)$$

Taking the curl of each curl equation in (3.3) and substituting gives

$$\nabla\times\left(\frac{\nabla\times\vec{\mathcal{H}}}{\varepsilon}\right)=\omega^2\mu\vec{\mathcal{H}}$$

$$\nabla\times\left(\frac{\nabla\times\vec{\mathcal{E}}}{\mu}\right)=\omega^2\varepsilon\vec{\mathcal{E}} \tag{3.4}$$

In view of (3.1), we can write

$$\begin{aligned}\nabla &= \vec{1}_x\frac{\partial}{\partial x}+\vec{1}_y\frac{\partial}{\partial y}-\vec{1}_z i\gamma\\ &=\nabla_T-\vec{1}_z i\gamma\end{aligned} \tag{3.5}$$

with ∇_T denoting the operator in the transverse plane.

4.4.1 Homogeneous Waveguides

In many cases, the waveguide under analysis consists of a metal casing, either empty or filled homogeneously with an isotropic dielectric. In these cases, the analysis can be simplified.

Eq. (3.3) can be expanded using (3.5) and rearranged to express the transverse x and y components in terms of the axial z components $\mathcal{H}_z$ and $\mathcal{E}_z$.

$$\begin{aligned}
\left(\omega^2\mu\varepsilon-\gamma^2\right)\mathcal{E}_x &= -i\left(\omega\mu\frac{\partial\mathcal{H}_z}{\partial y}+\gamma\frac{\partial\mathcal{E}_z}{\partial x}\right)\\
\left(\omega^2\mu\varepsilon-\gamma^2\right)\mathcal{E}_y &= i\left(\omega\mu\frac{\partial\mathcal{H}_z}{\partial x}-\gamma\frac{\partial\mathcal{E}_z}{\partial y}\right)\\
\left(\omega^2\mu\varepsilon-\gamma^2\right)\mathcal{H}_x &= i\left(\omega\varepsilon\frac{\partial\mathcal{E}_z}{\partial y}-\gamma\frac{\partial\mathcal{H}_z}{\partial x}\right)\\
\left(\omega^2\mu\varepsilon-\gamma^2\right)\mathcal{H}_y &= -i\left(\omega\varepsilon\frac{\partial\mathcal{E}_z}{\partial x}+\gamma\frac{\partial\mathcal{H}_z}{\partial y}\right)
\end{aligned} \tag{3.6}$$

The i in the right hand side corresponds to a phase shift of $\pi/2$ in the expansion (3.1).

Applying, the divergence equations of (3.3) become

$$i\gamma \mathcal{H}_z = \frac{\partial \mathcal{H}_x}{\partial x} + \frac{\partial \mathcal{H}_y}{\partial y}$$

$$i\gamma \mathcal{E}_z = \frac{\partial \mathcal{E}_x}{\partial x} + \frac{\partial \mathcal{E}_y}{\partial y} \quad (3.7)$$

so the z component equations of (3.4) are

$$\nabla_T \bullet (\nabla_T \mathcal{H}_z) + (\omega^2 \mu\varepsilon - \gamma^2)\mathcal{H}_z = 0$$

$$\nabla_T \bullet (\nabla_T \mathcal{E}_z) + (\omega^2 \mu\varepsilon - \gamma^2)\mathcal{E}_z = 0 \quad (3.8)$$

These are eigenvalue equations in $\mathcal{E}_z$ and $\mathcal{H}_z$, and the values of $(\omega^2 \mu\varepsilon - \gamma^2)$ for which solutions exist constitute the propagation constants of the unattenuated propagation modes that can be supported in the guide under analysis. For any eigenvalue, there are an infinite number of combinations of $\omega, \varepsilon, \mu, \gamma$ which can excite this mode, and the exact determination will depend on the materials and the driving frequency.

4.4.2 TE and TM Modes

In a homogeneously filled waveguide, there exist two sets of distinct modes. One set of modes has no magnetic field component in the propagation direction, and are referred to as Transverse Magnetic, or TM, modes. The other set has no electric field component in the propagation direction, and are referred to as Transverse Electric, or TE, modes. In either case, one member of (3.8) vanishes, leaving only a single variable and a single equation. Correspondingly, equations (3.6) are simplified by the absence of one or the other field component.

In the TE case, we have $\mathcal{E}_z = 0$, and the first of (3.8)

$$\nabla_T \bullet (\nabla_T \mathcal{H}_z) + (\omega^2 \mu\varepsilon - \gamma^2)\mathcal{H}_z = 0 \quad (3.9)$$

The boundary condition at an electrically conducting wall is $\hat{n}\bullet\vec{H} = 0$. Through (3.6), this implies $\hat{n}\bullet\nabla_T \mathcal{H}_z = 0$,which is the Natural boundary condition of (3.9).

In the TM case, we have $\mathcal{H}_z = 0$, and the second of (3.8)

$$\nabla_T \bullet (\nabla_T \mathcal{E}_z) + (\omega^2 \mu\varepsilon - \gamma^2)\mathcal{E}_z = 0 \quad (3.10)$$

The boundary condition at a metallic wall is $\hat{n}\times\vec{E} = 0$, which requires that tangential components of $\vec{\mathcal{E}}$ be zero in the wall. Since $\mathcal{E}_z$ is always tangential to the wall, the boundary condition is the Dirichlet condition $\mathcal{E}_z = 0$.

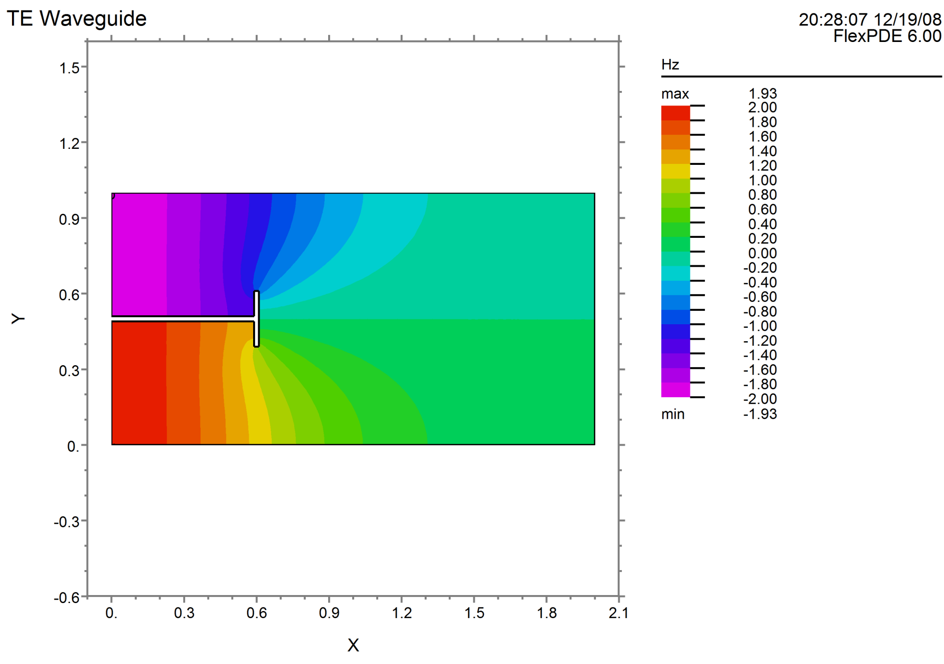
In the following example, we compute the first few TE modes of a waveguide of complex cross-section. The natural boundary condition allows an infinite number of solutions, differing only by a constant offset in the eigenfunction, so we add an integral constraint to center the eigenfunctions around zero. Since all the material parameters are contained in the eigenvalue, it is unnecessary to concern ourselves with their values. Likewise, the computation of the transverse field components are scaled by constants, but the shapes are unaffected.

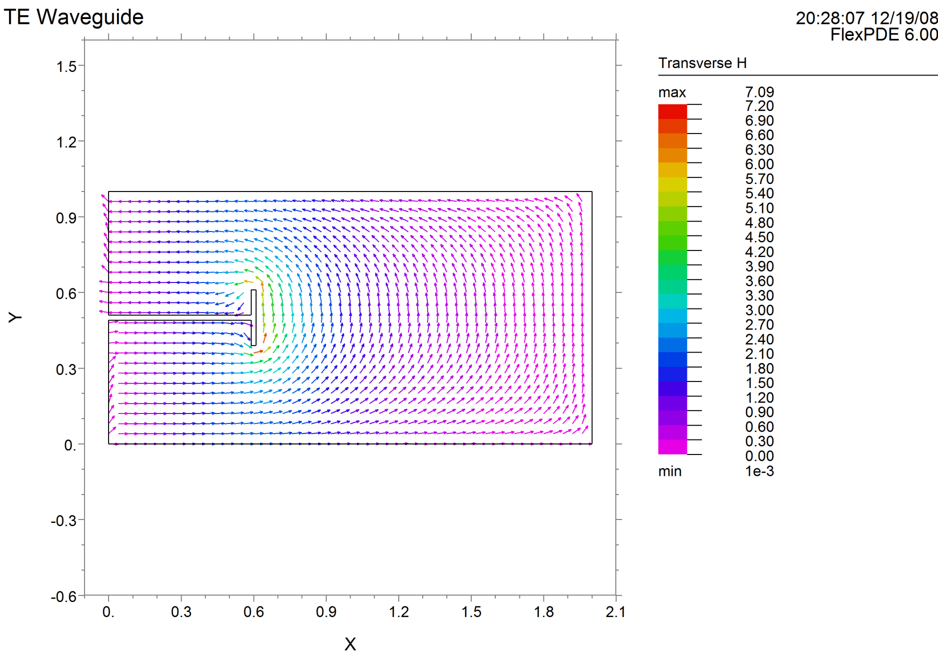
See also "Samples | Usage | Eigenvalues | Waveguide.pde" 471

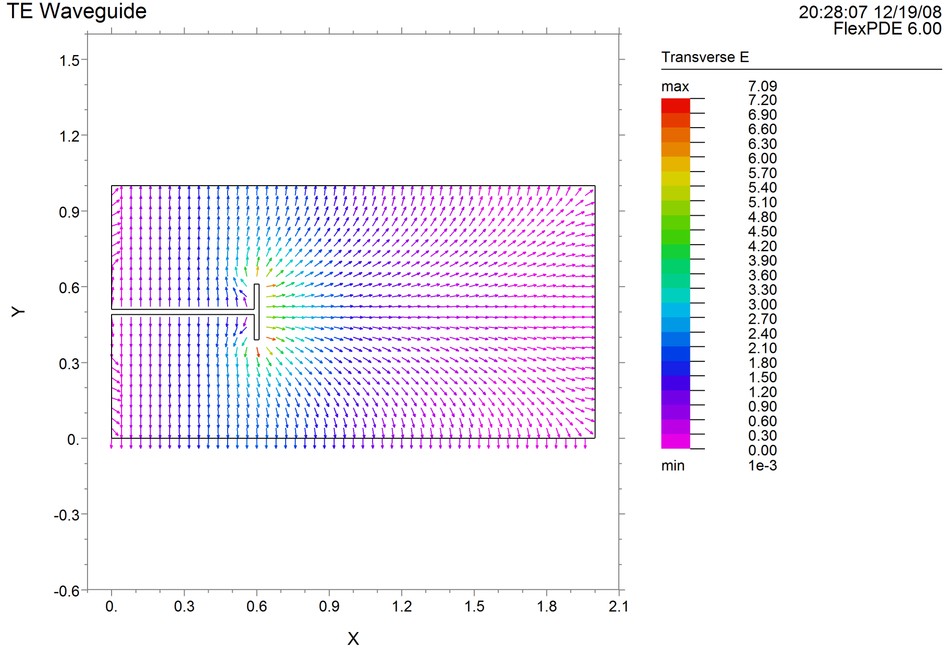
Descriptor 3.1 Waveguide.pde

```
title "TE Waveguide"

select
  modes = 4        { This is the number of Eigenvalues desired. }

variables
  Hz

definitions
  L = 2
  h = 0.5            ! half box height
  g = 0.01           ! half-guage of wall
  s = 0.3*L          ! septum depth
  tang = 0.1         ! half-width of tang
  Hx = -dx(Hz)
  Hy = -dy(Hz)
  Ex = Hy
  Ey = -Hx

equations
  div(grad(Hz)) + lambda*Hz = 0

constraints   { since Hz has only natural boundary conditions,
                        we need to constrain the answer }
  integral(Hz) = 0

boundaries
  region 1
    start(0,0)
    natural(Hz) = 0
    line to (L,0) to (L,1) to (0,1)  to (0,h+g)
    natural(Hz) = 0
    line to (s-g,h+g) to (s-g,h+g+tang) to (s+g,h+g+tang)
         to (s+g,h-g-tang) to (s-g,h-g-tang)
         to (s-g,h-g) to (0,h-g)
         to close

monitors
   contour(Hz)

plots
```

```
    contour(Hz) painted
    vector(Hx,Hy)  as "Transverse H" norm
    vector(Ex,Ey)  as "Transverse E" norm

  end
```




TE Waveguide
20:28:07 12/19/08
FlexPDE 6.00
Transverse H
max 7.09
7.20
6.90
6.60
6.30
6.00
5.70
5.40
5.10
4.80
4.50
4.20
3.90
3.60
3.30
3.00
2.70
2.40
2.10
1.80
1.50
1.20
0.90
0.60
0.30
0.00
min 1e-3
Y
X
waveguide: Grid#2 P2 Nodes=863 Cells=398 RMS Err= 8.3e-4
Mode 2 Lambda= 2.6444


TE Waveguide
20:28:07 12/19/08
FlexPDE 6.00
Transverse E
max 7.09
7.20
6.90
6.60
6.30
6.00
5.70
5.40
5.10
4.80
4.50
4.20
3.90
3.60
3.30
3.00
2.70
2.40
2.10
1.80
1.50
1.20
0.90
0.60
0.30
0.00
min 1e-3
Y
X
waveguide: Grid#2 P2 Nodes=863 Cells=398 RMS Err= 8.3e-4
Mode 2 Lambda= 2.6444

4.4.3 Non-Homogeneous Waveguides

Note: *The development given here follows that of Fernandez and Lu, "Microwave and Optical Waveguide Analysis", and of Silvester and Ferrari, "Finite Elements for Electrical Engineers".*

In many applications, a waveguide is partially or inhomogeneously filled with dielectric material. In this case, pure TE and TM modes do not exist. Both $\mathcal{E}_z$ and $\mathcal{H}_z$ exist simultaneously, and the propagation modes are hybrid in nature.

It is possible to address a simultaneous solution of equations (3.4) in a manner similar to (3.8). However, care must be taken to keep the ε parameter inside of some of the derivatives, and problems arise with the simplifications implicit in (3.7). This approach also has been plagued with spurious solution modes. It is claimed that these spurious modes arise because the axial field model does not explicitly impose $\nabla \bullet \vec{B} = 0$, and that the spurious modes are those for which this condition is violated.

An alternative approach seeks to reduce the equations (3.4) to a pair of equations in the transverse components of the magnetic field, $\mathcal{H}_T = \mathcal{H}_x \hat{1}_x + \mathcal{H}_y \hat{1}_y$. In the process, the condition $\nabla \bullet \vec{B} = 0$ is explicitly imposed, and it is claimed that no spurious modes arise.

In the development that follows, we continue to treat μ as a constant (invalidating use where magnetic materials are present), but we exercise more care in the treatment of ε.

For notational convenience, we will denote the components of $\vec{\mathcal{H}}$ as $\vec{\mathcal{H}} = a\hat{1}_x + b\hat{1}_y + c\hat{1}_z$ and use subscripts to denote differentiation. The first equation of (3.4) can then be expanded with (3.5) to give

$$\begin{aligned}
&\left(b_x / \varepsilon\right)_y - \left(a_y / \varepsilon\right)_y - i\gamma c_x / \varepsilon + \gamma^2 a / \varepsilon = \omega^2 \mu a \\
&\left(a_y / \varepsilon\right)_x - \left(b_x / \varepsilon\right)_x - i\gamma c_y / \varepsilon + \gamma^2 b / \varepsilon = \omega^2 \mu b \\
&-\left(c_x / \varepsilon\right)_x - \left(c_y / \varepsilon\right)_y - i\gamma \left(a / \varepsilon\right)_x - i\gamma \left(b / \varepsilon\right)_y = \omega^2 \mu c
\end{aligned} \tag{3.11}$$

The condition $\nabla \bullet \vec{B} = 0$ allows us to replace

$$i\gamma c = a_x + b_y \tag{3.12}$$

and to eliminate the third equation. We can also define $\varepsilon = \varepsilon_r \varepsilon_0$ and $\mu = \mu_0$ and multiply through by ε_0 leaving

$$\begin{aligned}
&\left(b_x / \varepsilon_r\right)_y - \left(a_y / \varepsilon_r\right)_y - \left(a_x + b_y\right)_x / \varepsilon_r + \gamma^2 a / \varepsilon_r = \omega^2 \mu_0 \varepsilon_0 a \\
&\left(a_y / \varepsilon_r\right)_x - \left(b_x / \varepsilon_r\right)_x - \left(a_x + b_y\right)_y / \varepsilon_r + \gamma^2 b / \varepsilon_r = \omega^2 \mu_0 \varepsilon_0 b
\end{aligned} \tag{3.13}$$

In vector form we can write this as

$$\nabla_T \times \left(\frac{\nabla_T \times \vec{\mathcal{H}}_T}{\varepsilon_r} \right) - \frac{\nabla_T \left(\nabla_T \bullet \vec{\mathcal{H}}_T \right)}{\varepsilon_r} + \frac{\gamma^2 \vec{\mathcal{H}}_T}{\varepsilon_r} = \omega^2 \mu_0 \varepsilon_0 \vec{\mathcal{H}}_T$$

(3.14)

The equation pair (3.13) is an eigenvalue problem in γ^2. We can no longer bundle the ω^2 and γ^2 terms inside the eigenvalue, because the ε_r dividing γ^2 is now variable across the domain. Given a driving frequency ω, we can compute the axial wave numbers γ for which propagating modes exist.

4.4.4 Boundary Conditions

To see what the natural boundary conditions imply, integrate the second order terms of (3.13) by parts:

$$\iint_T \left[\left(b_x / \varepsilon_r \right)_y - \left(a_y / \varepsilon_r \right)_y - \left(a_x + b_y \right)_x / \varepsilon_r \right] dxdy$$

$$\longrightarrow \oint \left[n_y \left(b_x - a_y \right) / \varepsilon_r - n_x \left(a_x + b_y \right) / \varepsilon_r \right] dl$$

$$\iint_T \left[\left(a_y / \varepsilon_r \right)_x - \left(b_x / \varepsilon_r \right)_x - \left(a_x + b_y \right)_y / \varepsilon_r \right] dxdy$$

$$\longrightarrow \oint \left[n_x \left(a_y - b_x \right) / \varepsilon_r - n_y \left(a_x + b_y \right) / \varepsilon_r \right] dl$$

(3.15)

We have shown only the contour integrals arising from the integration, and suppressed the area integral correcting for varying ε. This term will be correctly added by FlexPDE, and does not contribute to the boundary condition.

The integrand of the contour integrals is the value represented by the natural boundary condition statement in FlexPDE.

The boundary conditions which must be satisfied at an electrically conducting wall are

$$\hat{n} \bullet \vec{\mathcal{H}} = 0$$

$$\hat{n} \times \vec{\mathcal{E}} = 0$$

(3.16)

The first condition requires that $n_x \mathcal{H}_x + n_y \mathcal{H}_y + n_z \mathcal{H}_z = 0$. At a vertical wall, $n_y = n_z = 0$, and the condition becomes simply $\mathcal{H}_x = 0$. Similarly, at a horizontal wall, it is $\mathcal{H}_y = 0$. Both are easily expressed as Value boundary conditions. At an oblique wall, the condition can be expressed as an implicit value boundary condition for one of the components.

The second condition requires that the tangential components of $\vec{\mathcal{E}}$ must vanish in the wall. In particular, $\mathcal{E}_z$ is always tangential and must therefore be zero. From (3.3) we can derive $i\omega\varepsilon\mathcal{E}_z = (b_x - a_y)$. But this is just the first term of the integrands in (3.15), so at a vertical wall we can set Natural($\mathcal{H}_y$)=0, and at a

horizontal wall we can use Natural($\mathcal{H}_x$)=0. These are the reverse assignments from the value conditions above, so the two form a complementary set and completely specify the boundary conditions for (3.13). Similar arguments can be used at a magnetic wall, resulting in a reversed assignment of value and natural boundary conditions.

4.4.5 Material Interfaces

At a material interface, Maxwell's equations require that the tangential components of $\vec{\mathcal{E}}$ and $\vec{\mathcal{H}}$ and the normal components of $\varepsilon\vec{\mathcal{E}}$ and $\mu\vec{\mathcal{H}}$ must be continuous.

The tangential continuity of components $\mathcal{H}_x = a$ and $\mathcal{H}_y = b$ is automatically satisfied, because FlexPDE stores only a single value of variables at the interface. Continuity of $\mathcal{H}_z = c$, which is always tangential, requires, using (3.12), $\left(a_x + b_y\right)_1 = \left(a_x + b_y\right)_2$. Continuity of $\mathcal{E}_z$ requires $\left(b_x - a_y\right)_1 = \left(b_x - a_y\right)_2$.

Now consider the integrals (3.15) to be taken over each material independently. Each specifies in a general sense the "flux" of some quantity outward from the region. The sum of the two integrands, taking into account the reversed sign of the outward normal, specifies the conservation of the "flux". In the usual case, the sum is zero, representing "flux" conservation. In our case, we must specify a jump in the flux in order to satisfy the requirements of Maxwell's equations.

For the $\mathcal{H}_x$ component equation we have, using the outward normals from region 1,

$$integrand_1 + integrand_2 =$$
$$n_y\left[\left(\frac{b_x - a_y}{\varepsilon_r}\right)_1 - \left(\frac{b_x - a_y}{\varepsilon_r}\right)_2\right] - n_x\left[\left(\frac{a_x + b_y}{\varepsilon_r}\right)_1 - \left(\frac{a_x + b_y}{\varepsilon_r}\right)_2\right]$$

But the continuity requirements above dictate that the numerators be continuous, so the internal natural boundary condition becomes

$$integrand_1 + integrand_2 =$$
$$\left[n_y\left(b_x - a_y\right) - n_x\left(a_x + b_y\right)\right]\left[\left(\frac{1}{\varepsilon_r}\right)_1 - \left(\frac{1}{\varepsilon_r}\right)_2\right]$$

By a similar argument, the internal natural boundary condition for the $\mathcal{H}_y$ component equation is

$$integrand_1 + integrand_2 =$$
$$\left[n_x\left(a_x - b_y\right) - n_y\left(a_x + b_y\right)\right]\left[\left(\frac{1}{\varepsilon_r}\right)_1 - \left(\frac{1}{\varepsilon_r}\right)_2\right]$$

Clearly, at an internal interface where ε_r is continuous, the internal natural boundary condition reduces to zero, which is the default condition.

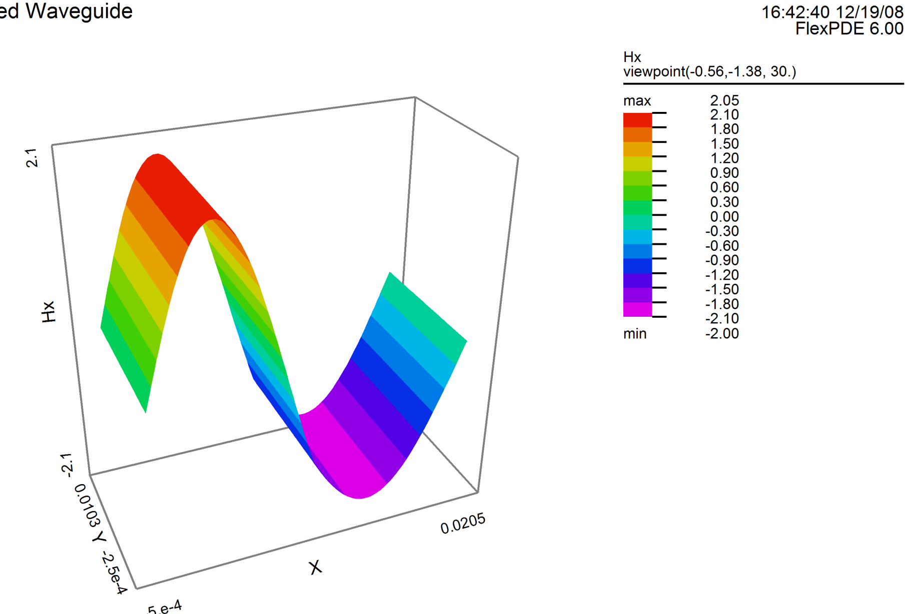
In the example which follows, we consider a simple 2x1 metal box with dielectric material in the left half. Note that FlexPDE will compute the eigenvalues with lowest magnitude, regardless of sign, while negative eigenvalues correspond to modes with propagation constants below cutoff, and are therefore not physically realizable.

See also "Samples | Usage | Eigenvalues | Filledguide.pde" 467

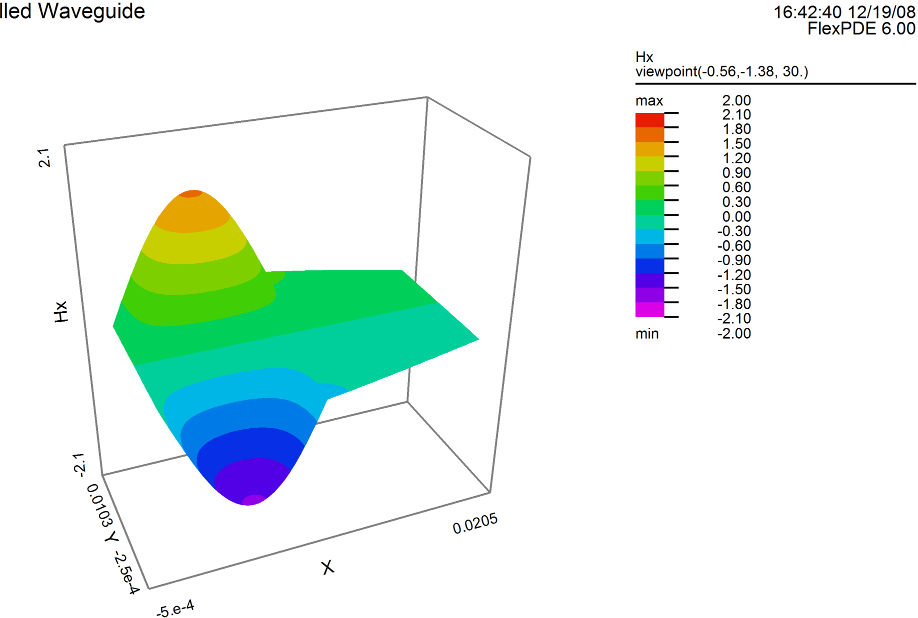
Descriptor 3.2 Filledguide.pde

```
title "Filled Waveguide"

select
  modes = 8      { This is the number of Eigenvalues desired. }

variables
  Hx,Hy

definitions
  cm = 0.01              ! conversion from cm to meters
  b = 1*cm               ! box height
  L = 2*b                ! box width
  epsr
  epsr1=1       epsr2=1.5
  ejump = 1/epsr2-1/epsr1      ! the boundary jump parameter
  eps0 = 8.85e-12
  mu0 = 4e-7*pi
  c =  1/sqrt(mu0*eps0) ! light speed
  k0b = 4
  k0 = k0b/b
  k02 = k0^2             ! k0^2=omega^2*mu0*eps0

  curlh = dx(Hy)-dy(Hx) ! terms used in equations and BC's
  divh = dx(Hx)+dy(Hy)

equations
  Hx: dx(divh)/epsr - dy(curlh/epsr) + k02*Hx - lambda*Hx/epsr = 0
  Hy: dx(curlh/epsr) + dy(divh)/epsr + k02*Hy - lambda*Hy/epsr = 0

boundaries
  region 1  epsr=epsr1
    start(0,0)
```

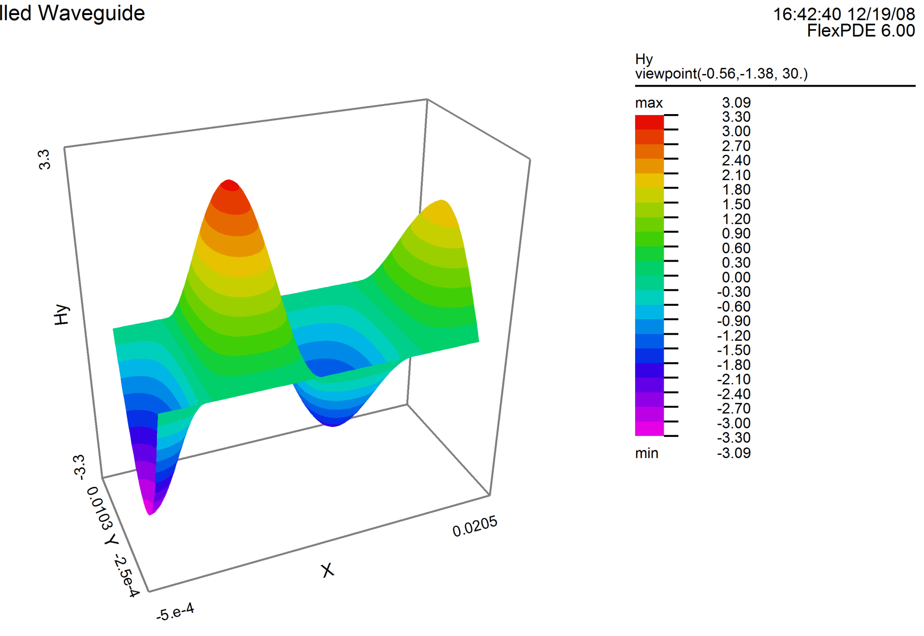
```
  natural(Hx) = 0  value(Hy)=0
  line to (L,0)
  value(Hx) = 0  value(Hy)=0  natural(Hy)=0
  line to (L,b)
  natural(Hx) = 0 value(Hy)=0
  line to (0,b)
  value(Hx) = 0  natural(Hy)=0
  line to close

region 2  epsr=epsr2
  start(b,b)
  line to (0,b) to (0,0) to (b,0)
  natural(Hx) = normal(-ejump*divh,ejump*curlh)
  natural(Hy) = normal(-ejump*curlh,-ejump*divh)
  line to close

monitors
   contour(Hx) range=(-3,3)
   contour(Hy) range=(-3,3)

plots
   contour(Hx) range=(-3,3) report(k0b)
            report(sqrt(abs(lambda))/k0)
   surface(Hx) range=(-3,3) report(k0b)
            report(sqrt(abs(lambda))/k0)
   contour(Hy) range=(-3,3) report(k0b)
            report(sqrt(abs(lambda))/k0)
   surface(Hy) range=(-3,3) report(k0b)
            report(sqrt(abs(lambda))/k0)

summary  export
   report(k0b)
   report lambda
   report(sqrt(abs(lambda))/k0)

end
```



filledguide: Grid#1 P2 Nodes=2235 Cells=1072 RMS Err= 0.0062
Mode 6 Lambda= 98613. k0b= 4.000000 sqrt(abs(lambda))/k0= 0.785066 Integral= -4.020731e-5



filledguide: Grid#1 P2 Nodes=2235 Cells=1072 RMS Err= 0.0062
Mode 4 Lambda= 56149. k0b= 4.000000 sqrt(abs(lambda))/k0= 0.592397 Integral= -6.871203e-10



filledguide: Grid#1 P2 Nodes=2235 Cells=1072 RMS Err= 0.0062
Mode 7 Lambda=-1.1429e+5 k0b= 4.000000 sqrt(abs(lambda))/k0= 0.845175 Integral= 9.158979e-6

4.5 References

N. J. Cronin, “Microwave and Optical Waveguides”, London, Institute of Physics Publishing, 1995.

F. Anibal Fernandez and Yilong Lu, “Microwave and Optical Waveguide Analysis”, Somerset,UK, Research Studies Press, Ltd. 1996.

S. R. H. Hoole, “Computer-Aided Analysis and Design of Electromagnetic Devices”, New York, Elsevier, 1989.

Nathan Ida and Joao P.A. Bastos “Electromagnetics and Calculation of Fields”, New York, Springer-Verlag, 1992.

J. D. Jackson, “Classical Electrodynamics”, Second Edition, New York, John Wiley & Sons, 1975.

Jianming Jin, “The Finite Element Method in Electromagnetics”, New York, John Wiley & Sons, Inc, 1993.

Peter P. Silvester and Ronald L. Ferrari, “Finite Elements for Electrical Engineers”, Third Edition, Cambridge University Press, 1996.

C. T. Tai, “Generalized Vector and Dyadic Analysis”, New York, IEEE Press, 1992.

Part V

Technical Notes

5 Technical Notes

5.1 Natural Boundary Conditions

The NATURAL boundary condition 61 is a generalization of the concept of a flux boundary condition. In diffusion equations, it is in fact the outward flux of the diffusing quantity. In stress equations, it is the surface load. In other equations, it can be less intuitive.

FlexPDE uses integration by parts to reduce the order of second derivative terms in the system equations. Application of this technique over a two-dimensional computation cell produces an interior area integral term and a boundary line integral term. Forming the same integral in two adjacent computation cells produces the same boundary integral at their interface, except that the direction of integration is opposite in the two cells. If the integrals are added together to form the total integral, the shared boundary integrals cancel.

- Applied to the term dx(f), where f is an expression containing further derivative terms, integration by parts yields
  Integral(dx(f)*dV) = Integral(f*c*dS),
  where c denotes the x-component of the outward surface-normal unit vector and dS is the differential surface element.
  (Y- and Z- derivative terms are handled similarly, with c replaced by the appropriate unit-vector component.)

- Applied to the term dxx(f), where f denotes a scalar quantity, integration by parts yields
  Integral(dxx(f)*dV) = Integral(dx(f)*c*dS),
  where c denotes the x-component of the outward surface-normal unit vector and dS is the differential surface element.
  (Y- and Z- derivative terms are handled similarly, with c replaced by the appropriate unit-vector component.)

- Applied to the term DIV(**F**), where **F** denotes a vector quantity containing further derivative terms, integration by parts is equivalent to the divergence theorem,
  Integral(DIV(**F**)dV) = Integral(**F** . n dS),
  where **n** denotes the outward surface-normal unit vector and dS is the differential surface element.

- Applied to the term CURL(**F**), where **F** denotes a vector quantity containing further derivative terms, integration by parts is equivalent to the curl theorem,
  Integral(CURL(**F**) dV) = Integral(**n x F** dS),
  where again **n** denotes the outward surface-normal unit vector and d*S* is the differential surface element.

- FlexPDE performs these integrations in 3 dimensions, including the volume and surface elements appropriate to the geometry. In 2D Cartesian geometry, the volume cell is extended one unit in the Z direction; in 2D cylindrical geometry, the volume cell is r*dr*dtheta.

This technique forms the basis of the treatment of exterior boundary conditions and interior material interface behavior in FlexPDE.

- All boundary integral terms are assumed to vanish at internal cell interfaces.
- All boundary integral terms are assumed to vanish at internal and external boundaries, *unless* a NATURAL boundary condition statement provides an independent evaluation of the boundary integrand.

There are several ramifications of this treatment:

In divergence equations, such as DIV(k **F**) = 0,

- the quantity (k **F . n**) will be continuous across interior material interfaces.
- The NATURAL boundary condition specifies the value of (k **F . n**) on the boundary.
    - If (k **F**) is heat flux (k **F** = -k Grad(T)), then energy will be conserved across material discontinuities, and the NATURAL boundary condition defines outward heat flux.
    - If (k **F**) is electric displacement (D = -eps Grad(V)) or magnetic induction (B = Curl(A)), then the material interface conditions dictated by Maxwell's equations will be satisfied, and in the electric case the NATURAL boundary condition will define the surface charge density.

In curl equations, such as CURL(k **F**) = 0,

- the quantity (k **n x F**) will be continuous across interior material interfaces.
- The NATURAL boundary condition specifies the value of (k **n x F**) on the boundary.
- If (k **F**) is magnetic field (**H** = (1/mu) Curl(**A**)) or electric field (**E** = -Grad(V)), then the material interface conditions dictated by Maxwell's equations will be satisfied, and in the magnetic case the NATURAL boundary condition will define the surface current density.

Note that it is not necessary to write the equations explicitly with the DIV or CURL operators for these conditions to be met. Any valid differential equivalent in the coordinate system of the problem will be treated the same way.

Note also that the NATURAL boundary condition and the PDE are intimately related.

- If a differential operator has an argument that itself contains a differential operator, then that argument becomes the object of integration by parts, and generates a corresponding component of the NATURAL boundary condition.
- If the PDE is multiplied by some factor, then the associated NATURAL boundary condition must be multiplied by the same factor.
- The NATURAL boundary condition must have a sign consistent with the sign of the associated PDE terms *when moved to the left side of the equation*.
- The NATURAL boundary condition statement specifies to FlexPDE the integrand of the surface integral generated by the integration by parts, which is otherwise assumed to be zero.

5.2 Solving Nonlinear Problems

FlexPDE automatically recognizes when a problem is nonlinear and modifies its strategy accordingly.

In nonlinear systems, we are not guaranteed that the system will have a unique solution, and even if it does, we are not guaranteed that FlexPDE will be able to find it. The solution method used by FlexPDE is a modified Newton-Raphson iteration procedure. This is a "descent" method, which tries to fall down the gradient of an energy functional until minimum energy is achieved (i.e. the gradient of the functional goes to zero). If the functional is nearly quadratic, as it is in simple diffusion problems, then the method converges quadratically (the relative error is squared on each iteration). The default strategy implemented in FlexPDE is usually sufficient to determine a solution without user intervention.

Time-Dependent Problems

In nonlinear time-dependent problems, the default behavior is to compute the Jacobian matrix (the "slope" of the functional) and take a single Newton step at each timestep, on the assumption that any nonlinearities will be sensed by the timestep controller, and that timestep adjustments will guarantee an accurate evolution of the system from the given initial conditions.

Several selectors are provided to enable more robust (but more expensive) treatment in difficult cases. The "NEWTON=number" selector can be used to increase the maximum number of Newton iterations performed on each timestep. In this case, FlexPDE will examine the change in the system variables and recompute the Jacobian matrix whenever it seems warranted. The Selector REMATRIX=On will force the Jacobian matrix to be re-evaluated at each Newton step.

The PREFER_SPEED selector is equivalent to the default behavior, setting NEWTON=1 and REMATRIX=Off.
The PREFER_STABILITY selector resets the values of NEWTON=5 and REMATRIX=On.

Steady-State Problems

In the case of nonlinear steady-state problems, the situation is somewhat more complicated. The default controls are usually sufficient to achieve a solution. The Newton iteration is allowed to run a large number of iterations, and the Jacobian matrix is recomputed whenever the change in the solution values seem to warrant it. The Selector REMATRIX=On may be used to force re-computation of the Jacobian matirx on each Newton step.

In cases of strong nonlinearities, it may be necessary for the user to help guide FlexPDE to a valid solution. There are several techniques that can be used to help the solution process.

Start with a Good Initial Value
Providing an initial value which is near the correct solution will aid enormously in finding a solution. Be particularly careful that the initial value matches the boundary conditions. If it does not, serious excursions may be excited in the trial solution, leading to solution difficulties.

Use STAGES to Gradually Activate the Nonlinear Terms
You can use the staging facility of FlexPDE to gradually increase the strength of the nonlinear terms. Start with a linear (or nearly linear) system, and allow FlexPDE to find a solution which is consistent with the boundary conditions. Then use this solution as a starting point for a more strongly nonlinear system. By judicious use of staging, you can creep up on a solution to very nasty problems.

Use CHANGELIM to Control Modifications
The selector CHANGELIM limits the amount by which any nodal value in a problem may be modified on each Newton-Raphson step. As in a one-dimensional Newton iteration, if the trial solution is near a local maximum of the functional, then shooting down the gradient will try to step an enormous distance to the next trial solution. FlexPDE limits the size of each nodal change to be less than CHANGELIM times the average value of the variable. The default value for CHANGELIM is 0.5, but if the initial value (or any intermediate trial solution) is sufficiently far from the true solution, this value may allow wild excursions from which FlexPDE is unable to recover. Try cutting CHANGELIM to 0.1, or in severe cases even 0.01, to force FlexPDE to creep toward a valid solution. In combination with a reasonable initial value, even CHANGELIM=0.01 can converge in a surprisingly short time. Since CHANGELIM multiplies the RMS average value, not each local value, its effect disappears when a solution is reached, and quadratic final convergence is still achieved.

Watch Out for Negative Values
FlexPDE uses piecewise polynomials to approximate the solution. In cases of rapid variation of the solution over a single cell, you will almost certainly see severe under-shoot in early stages. If you are assuming that the value of your variable will remain positive, don't. If your equations lose validity in the presence of negative values, perhaps you should recast the equations in terms of the logarithm of the variable. In this case, even though the logarithm may go negative, the implied value of your actual variable will remain positive.

Recast the Problem in a Time-Dependent Form
Any steady-state problem can be viewed as the infinite-time limit of a time-dependent problem. Rewrite your PDE's to have a time derivative term which will push the value in the direction of decreasing deviation

from solution of the steady-state PDE. (A good model to follow is the time-dependent diffusion equation DIV(K*GRAD(U)) = DT(U). A negative value of the divergence indicates a local maximum in the solution, and results in driving the value downward.) In this case, "time" is a fictitious variable analogous to the "iteration count" in the steady-state N-R iteration, but the time-dependent formulation allows the timestep controller to guide the evolution of the solution.

5.3 Eigenvalues and Modal Analysis

FlexPDE can solve eigenvalue problems involving an arbitrary number of equations. This type of problem is identified by the appearance of the selector MODES=<number> in the select section, and by use of the reserved word LAMBDA in the equations section. The MODES selector tells FlexPDE how many modes to calculate, and LAMBDA in the equations stands for the eigenvalue.

FlexPDE uses the method of subspace iteration (see Bathe and Wilson, "Numerical Methods in Finite Element Analysis", Prentice-Hall, 1976) to solve for a selected number of eigenvalues of lowest magnitude. In this method, the full problem is projected onto a subspace of much smaller dimension, and the eigenvalues and eigenvectors of this reduced system are found. This process is repeated until convergence of the eigenvalues is achieved. The eigenvectors of the full system are then recovered from expansion of the eigenvectors of the reduced system. As in a power-series expansion, there is some loss of accuracy in the higher modes due to truncation error. For this reason, FlexPDE solves a subspace of dimension min(n+8,2*n), where n is the number of requested modes.

See the eigenvalue examples 463 for demonstrations of this use of FlexPDE.

Eigenvalue Shifting

It is possible to examine eigenmodes which do not correspond to eigenvalues of the smallest magnitude by the technique of eigenvalue shifting. Consider the two systems

```
L(u) + lambda*u = 0
```

And

```
L(u) + lambda*u + shift*u = 0.
```

These systems will have the same eigenvectors, but the eigenvalues will differ by the value of "shift". Given the latter problem, FlexPDE will find a set of eigenvalues corresponding to the eigenvalues closest above "shift" in the spectrum of the former problem. The sum "lambda+shift" will correspond to the eigenvalue of the former system.

Eigenvalue shifting is demonstrated in the examples "Samples | Usage | Eigenvalues | Waveguide20.pde" 472 and "Samples | Usage | Eigenvalues | Shiftguide.pde" 468.

5.4 Avoid Discontinuities!

Discontinuities can cause serious numerical difficulty. This is most glaringly true in time-dependent problems, but can be a factor in steady-state problems as well.

Steady-State

The nodal finite element model used in FlexPDE assumes that all variables are continuous throughout the problem domain. This follows from the fact that the mesh nodes that sample the values of the variables are shared between the cells that they adjoin. Internally, the solution variables are interpolated by low-order polynomials over each cell of the finite element mesh. A discontinuous change in boundary conditions along the boundary path, particularly between differing VALUE conditions, will require intense mesh refinement to resolve the transition.

Whenever possible, use RAMP 130, URAMP 128, SWAGE 132, part of a sine or supergaussian, or some other smooth function to make a transition in value conditions over a physically meaningful distance.

If the quantity you have chosen as a system variable is in fact expected to be discontinuous at an interface, consider choosing a different variable which is continuous, and from which the real variable can be computed.

Time-Dependent

It is a common tendency in posing problems for numerical solution to specify initial conditions or boundary conditions as discontinuous functions, such as "at time=2 seconds, the boundary temperature is raised instantaneously to 200 degrees." A little thought will reveal that such statements are totally artificial. They violate the constraints of physics, and they pose impossible conditions for numerical solution. Not quite so obvious is the case where a boundary condition is applied at the start of the problem which is inconsistent with the initial values. This is in fact a statement that "at time=0 the boundary temperature is raised instantaneously to a new value", and so is the same as the statement above.

To raise a temperature "instantaneously" requires an infinite heat flux. To move a material position "instantaneously" requires an infinite force. In the real world, nothing happens "instantaneously". Viscosity diffuses velocity gradients, elastic deformation softens displacement velocities, thermal diffusion smoothes temperature changes. At some scale, all changes in nature are smooth.

In the mathematical view, the Fourier transform of a step function is (1/frequency). This means that a discontinuity excites an infinite spectrum of spatial and temporal frequencies, with weights that diminish quite slowly at higher frequencies. An "accurate" numerical model of such a system would require an infinite number of nodes and infinitesimal time steps, to satisfy sampling requirements of two samples per cycle. Any frequency components for which the sampling requirement is not met will be modeled wrong, and will cause oscillations or inaccuracies in the solution.

How then have numerical solutions been achieved to these problems over the decades? The answer is that artificial numerical diffusion processes have secretly filtered the frequency spectrum of the solution to include only low-frequency components. Or the answers have been wrong. Right enough to satisfy the user, and wrong enough to satisfy the calculation.

It is useful in this context to note that the effect of a diffusion term D*div(grad(U)) is to apply an attenuation of 1/(1+D*K*K) to the K-th frequency component of U. Conversely, any side effect of a numerical approximation which damps high frequency components is similar to a diffusion operator in the PDE.

We have attempted in FlexPDE to eliminate as many sources of artificial solution behavior as possible. Automatic timestep control and adaptive gridding are mechanisms which try to follow accurately the solution of the posed PDE. Discontinuities cannot be accurately modeled, and are therefore, strictly speaking, ill-posed problems. They cause tiny timesteps and intense mesh refinement in the early phases, causing long running times.

What can be done?

- Start your problem with initial conditions which are self-consistent; this means the values should correspond to a steady state solution with some set of boundary conditions. If you cannot by inspection determine these values, use a steady-state FlexPDE run with TRANSFER to precompute the initial values.
- Use RAMP, URAMP, SWAGE or other smooth function of time to turn the source value on over a meaningful interval of time.
- Whenever possible, instead of an instantaneously applied value condition, use a flux boundary condition which reflects the maximum physical initial flux that could arise from such a step condition (see the sample problem SAMPLES|Applications|Misc|Diffusion.pde for an example).

- Volume source functions and Natural boundary conditions are not as sensitive as direct conditions on the variables, because they appear in the numerical solution as integrals over some interval, and are thus somewhat "smoothed".

It may seem like an imposition that we should require such adulteration of your pure PDE, but the alternative is that we apply these adulterations behind your back, in unknown quantities and with unknown affect on your solution. At least this way, you're in control.

5.5 Importing DXF Files

FlexPDE supports the import of DXF files, allowing you to use AutoCAD to prepare your FlexPDE problem descriptor files.

To prepare the problem in AutoCAD, use the following rules:

- On layer 0, enter as text the entire body of the problem description, *excluding* the BOUNDARIES section.
- Use one layer for each region of the problem. Draw on each layer the boundaries pertaining to that region. Enter as text on each layer the necessary regional definitions for that region. For boundaries that are shared between regions, be sure that the points are recognizably the same (within 1e-6*domain size). Snap-to-grid is advised.
- Enter as text the necessary boundary conditions. Place the text so that the insertion point is near the boundary to which the boundary condition applies.
- Export the drawing as a DXF file in R14 format.

To run the problem in FlexPDE, do the following:

- Select the "Import->DXF 2D" item from the "File" menu.
- Select the DXF file to import and click "open".
- Enter a minimum merge distance. This is the distance at which two points will be considered the same, and merged.
- FlexPDE will read the DXF file and translate it into a corresponding .PDE file. This file will be displayed in the FlexPDE editor and also written to disk as a .PDE file for later use.
- Examine the translated file for errors, then proceed as for a standard .PDE file.

You may chose to modify the translated .PDE file, or to continue to update the .DXF file, whichever is most convenient for your needs.

Examples:

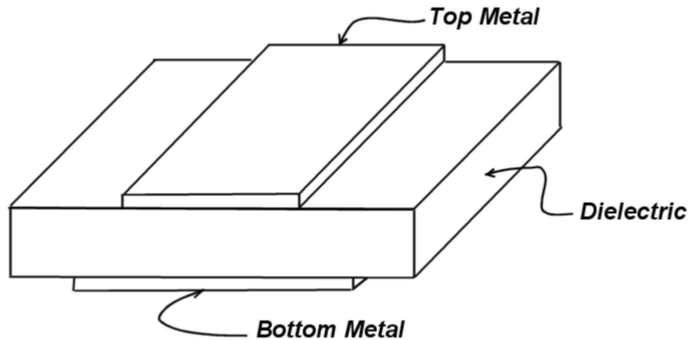
See the sample problem "Samples | Usage | Import-Export | AcadSample.dxf" and its associated drawing file AcadSample.dwg.

5.6 Extrusions in 3D

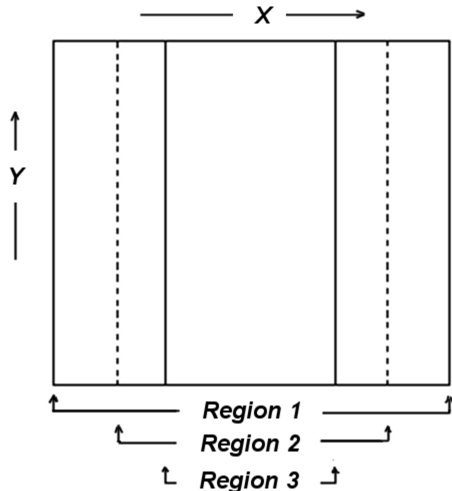
The specification of three-dimensional geometries as extrusions in FlexPDE is based on the decomposition of the object into two parts:

- The projection of the object onto the base X-Y plane.
- The division of the extrusion of this projection into layers in the Z dimension.

Let us take as a model a sandwich formed by a layer of dielectric material with two rectangular conductive patches, top and bottom, with differing dimensions. We wish to model the dielectric, the conductive patches and the surrounding air.



First, we form the projection of this figure onto the X-Y plane, showing all relevant interfaces:

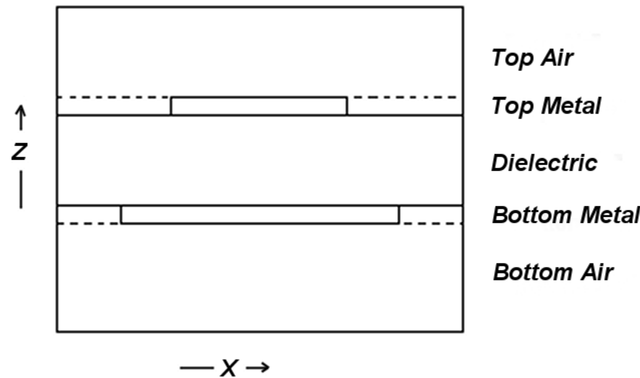


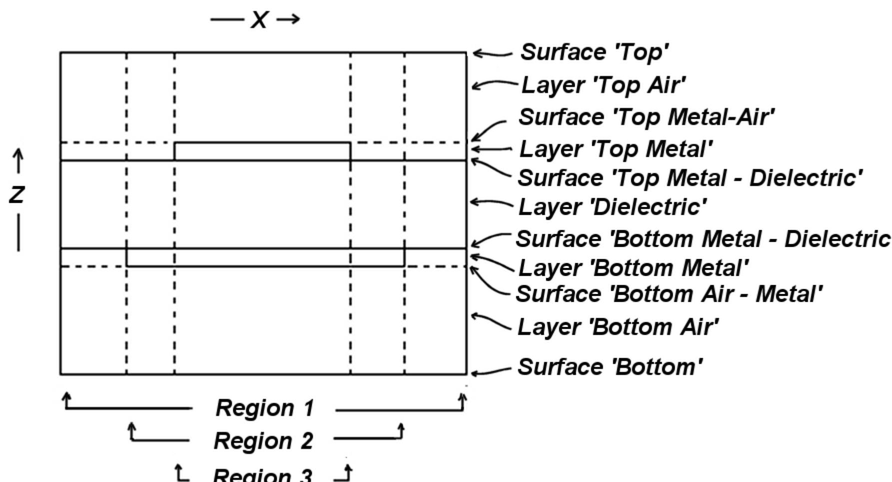
The geometry is specified to FlexPDE primarily in terms of this projection. A preliminary description of the 2D base figure is then:

```
BOUNDARIES
    REGION 1      {this is the outer boundary of the system}
        START(0,0)
            LINE TO (5,0) TO (5,5) TO (0,5) TO CLOSE
    REGION 2      {this region overrides region 1 and describes the larger plate}
        START(1,0)
            LINE TO (4,0) TO (4,5) TO (1,5) TO CLOSE
    REGION 3      {this region overrides region 1 & 2 and describes the smaller plate}
        START(2,0)
            LINE TO (3,0) TO (3,5) TO (5,3) TO CLOSE
```

Note that any part of the projection which will have a different stack of material properties above it must be represented by a region. All parts of the projection which will have the same stack of material properties may be included in a single region, even though they may be disjoint in the projection.

Next we view the X-Z cross-section of the sandwich:





The layer structure is specified bottom-up to FlexPDE in the EXTRUSION statement:

```
EXTRUSION
  SURFACE          "Bottom"                      Z=0
    LAYER  "Bottom Air"
  SURFACE          "Bottom Air - Metal"          Z=0.9
    LAYER  "Bottom Metal"
  SURFACE          "Bottom Metal - Dielectric"   Z=1
    LAYER  "dielectric"
  SURFACE          "Top Metal - Dielectric"              Z=2
    LAYER  "Top Metal"
  SURFACE          "Top Metal - Air"             Z=2.1
    LAYER  "Top Air"
  SURFACE          "top"                         Z=3
```

The LAYER statements are optional, as are the names of the surfaces. If surfaces and layers are not named, then they must subsequently be referred to by numbers, with surface numbers running in this case from 1 to 6 and layer numbers from 1 to 5. SURFACE 1 is Z=0, and LAYER 1 is between SURFACE 1 and SURFACE 2.

Note: a shorthand form to this specification is:

```
EXTRUSION Z=(0, 0.9, 1, 2, 2.1, 3)
```

In this form layers and surfaces must subsequently be referred to by number.

Assume that we have the following DEFINITIONS and EQUATIONS section:

```
DEFINITIONS
    K = Kair              {default the dielectric coefficient   to the value for air}
    Kdiel = 999    {replace 999 with problem value}
    Kmetal =  999    {replace 999 with problem value}
EQUATIONS
    DIV(K*GRAD(V))
```

We now modify the BOUNDARIES section to include layering information in the various regions:

```
BOUNDARIES
    REGION 1      {this is the outer boundary of the system}
        LAYER "Dielectric" K = Kdiel {all other layers default to Kair}
        START(0,0)
            LINE TO (5,0) TO (5,5) TO (0,5) TO CLOSE
    REGION 2      {this region overrides region 1 and describes the larger plate}
        LAYER "Bottom Metal" K = Kmetal
        LAYER "Dielectric" K = Kdiel
        START(1,0)
            LINE TO (4,0) TO (4,5) TO (1,5) TO CLOSE
    REGION 3      {this region overrides region 1 & 2 and describes the smaller plate}
        LAYER "Bottom Metal" K = Kmetal
        LAYER "Dielectric" K = Kdiel
        LAYER "Top Metal" Kmetal
        START(2,0)
            LINE TO (3,0) TO (3,5) TO (5,3) TO CLOSE
```

If layers are not named, then layer numbers must be used in place of the names above. The LAYER specifiers act as group headers, and all definitions following a LAYER specification refer to that layer, until the group is broken by SURFACE, LAYER or START. Definitions which apply to all layers of the region must appear before any LAYER specification.

The specification of boundary conditions proceeds in a similar way. As in the description of 2D problems in FlexPDE, the default boundary condition is always NATURAL(variable)=0. In the X-Y projection of our problem, which forms the basis of our 3D description, we have described the bounding lines of the regions. A boundary condition attached to any of these bounding lines will apply to all layers of the vertical surface formed by extruding the line. Boundary conditions along this surface may be specialized to a layer in the same way as the material properties are specialized to a layer. Assume that we wish to apply a potential of V0 to one end of the lower plate and V1 to the opposite end of the upper plate. We will modify the descriptor in the following way:

```
BOUNDARIES
    REGION 1      { this is the outer boundary of the system }
        LAYER "Dielectric" K = Kdiel      { all other layers default to Kair }
        START(0,0)
```

```
        LINE TO (5,0) TO (5,5) TO (0,5)  TO CLOSE
REGION 2      { this region overrides region 1, and describes the larger plate }
    LAYER "Bottom Metal" K = Kmetal
    LAYER "Dielectric" K = Kdiel
    START(1,0)
    LAYER "Bottom Metal" VALUE(V)=V0
        LINE TO (4,0)
    LAYER "Bottom Metal" NATURAL(V)=0
        LINE TO (4,5) TO (1,5) TO CLOSE
REGION 3      { this region overrides regions 1&2, and describes the smaller
plate}
    LAYER "Bottom Metal" K = Kmetal
    LAYER "Dielectric" K = Kdiel
    LAYER "Top Metal" K = Kmetal
    START(2,0)
        LINE TO (3,0) TO (3,5)
    LAYER "Top Metal" VALUE(V)=V1
        LINE TO (2,5)
    LAYER "Top Metal" NATURAL(V)=0
        LINE TO CLOSE
```

The final requirement for boundary condition specification is the attachment of boundary conditions to the X-Y end faces of the extruded figure. This is done by the SURFACE modifier. Suppose we wish to force the bottom surface to V=0 and the top to V=1. We would modify the descriptor as follows:

```
BOUNDARIES
    SURFACE "Bottom" VALUE(V)=0
    SURFACE "Top" VALUE(V)=1
    REGION 1      { this is the outer boundary of the system }
        LAYER "Dielectric" K = Kdiel      { all other layers default to Kair }
        START(0,0)
            LINE TO (5,0) TO (5,5) TO (0,5)  TO CLOSE
    REGION 2      { this region overrides region 1, and describes the larger plate }
        LAYER "Bottom Metal" K = Kmetal
        LAYER "Dielectric" K = Kdiel
        START(1,0)
        LAYER "Bottom Metal" VALUE(V)=V0
            LINE TO (4,0)
        LAYER "Bottom Metal" NATURAL(V)=0
            LINE TO (4,5) TO (1,5) TO CLOSE
    REGION 3      { this region overrides regions 1&2, and describes the smaller plate}
        LAYER "Bottom Metal" K = Kmetal
        LAYER "Dielectric" K = Kdiel
        LAYER "Top Metal" K = Kmetal
        START(2,0)
            LINE TO (3,0) TO (3,5)
        LAYER "Top Metal" VALUE(V)=V1
            LINE TO (2,5)
        LAYER "Top Metal" NATURAL(V)=0
            LINE TO CLOSE
```

Observe that since the SURFACE statements lie outside any REGION specification, they apply to all regions of the surface. To specialize the SURFACE statement to a specific region, it should be included

within the REGION definition.

In this example, we have used named surfaces and layers. The same effect can be achieved by omitting the layer names and specifying layers and surfaces by number:

```
BOUNDARIES
    SURFACE 1 VALUE(V)=0
    SURFACE 6 VALUE(V)=1
    REGION 1      { this is the outer boundary of the system }
        LAYER 3 K = Kdiel         { all other layers default to Kair }
        START(0,0)
            LINE TO (5,0) TO (5,5) TO (0,5)  TO CLOSE
    REGION 2      { this region overrides region 1, and describes the larger plate }
        LAYER 2 K = Kmetal
        LAYER 3 K = Kdiel
        START(1,0)
        LAYER 2 VALUE(V)=V0
            LINE TO (4,0)
        LAYER 2 NATURAL(V)=0
            LINE TO (4,5) TO (1,5) TO CLOSE
    REGION 3      { this region overrides regions 1&2, and describes the smaller plate}
        LAYER 2 K = Kmetal
        LAYER 3 K = Kdiel
        LAYER 4 K = Kmetal
        START(2,0)
            LINE TO (3,0) TO (3,5)
        LAYER 4 VALUE(V)=V1
            LINE TO (2,5)
        LAYER 4 NATURAL(V)=0
            LINE TO CLOSE
```

Remember that in our terminology a REGION refers to an area in the projected base plane, while a LAYER refers to a section of the Z-extrusion. A particular 3D chunk of the figure is uniquely identified by the intersection of a REGION and a LAYER.

A completed form of the descriptor outlined here may be found in the sample problem "Samples | Usage | 3D_Domains | 3D_Extrusion_spec.pde". A slightly more complex and interesting variation may be found in "Samples | Applications | Electricity | 3D_Capacitor.pde".

5.7 Applications in Electromagnetics

I. Maxwell's Equations

The purpose of this note is to develop formulations for the application of FlexPDE to various problems in electromagnetics. It is not our intention to give a tutorial on electromagnetics; we assume that the reader has some familiarity with the subject, and has access to standard references.

The starting point for our discussion is, as usual, Maxwell's equations. Posed in FlexPDE notation, these are

(1) Curl(**H**) = **J** + dt(**D**)
(2) Div(**B**) = 0

(3) Curl(**E**) = -dt(**B**)
(4) Div(**D**) = p (p = charge density, "rho")

To these we add the constitutive relations

(5) **D** = e**E** (e = permittivity, "epsilon")
(6) **B** = m**H** (m = permeability, "mu")
(7) **J** = s**E** (s = conductivity, "sigma")

(In isotropic materials, e, m and s are scalars, possibly nonlinear. In more complex materials, they may be tensors. In studies involving hysteresis or permanent magnets, modifications must be made to equation (6))

From these can be derived a convenient statement of charge conservation:

(8) Div(**J**) + dt(p) = 0

These equations form a very general framework for the study of electromagnetic fields, and admit of numerous combinations and permutations, depending on the characteristics of the problem at hand. Much confusion arises, in fact, from the tendency of textbooks to specialize the equations too soon, in order to simplify the exposition. This approach appears to present as a generally applicable formulation one which in reality embodies many assumptions about the problem being analyzed. We will discover that some alterations or substitutions that seem esthetically pleasing will not turn out to be wise computationally.

II. Choice of variables

A constraint due directly to the Finite Element model used in FlexPDE strikes us at the very outset. FlexPDE uses a continuous piecewise polynomial representation of all model variables. That is, at each computational node in the system it is assumed that each variable takes on a unique value, and that these nodal values can be connected in space by polynomial interpolations.

Application of the Divergence Theorem and Stokes' Theorem to Maxwell's equations yield the following boundary rules at material interfaces.

1) The tangential component of **E** is continuous; the normal component of **D**=(e**E**) is continuous (in the absence of surface charges).
2) The tangential component of **H** is continuous (in the absence of surface current); the normal component of **B**=(m**H**) is continuous.

These rules are in general inconsistent with the model assumptions of FlexPDE. This means that the field components themselves cannot be chosen as the model variables unless one of the following conditions applies:

1) There are no material property discontinuities in the domain,
2) The discontinuous components of the field are absent in the specific configuration being modeled.

For example, if we know that in a specific configuration that all the electric fields must be tangential to the material interfaces, we can use **E** as a model variable. If we know instead that all the electric fields are normal to the material interfaces, we can use **D** as a model variable.

The analysis of fields in terms of the field components comprises the bulk of textbook treatments, and we will not pursue the topic further here. We will instead turn our attention to a more generally applicable modeling approach. Nevertheless, despite the seemingly restrictive nature of these prohibitions, there is a large class of problems which can be analyzed successfully by FlexPDE in terms of field components.

III. Potentials

For any twice-differentiable vector **v**, the vector identity Div(Curl(**v**)) = 0 holds. This identity together with equation (2) implies that we can define a vector potential function **A**, the magnetic vector potential, such that

(9) Curl(**A**) = **B**

A theorem due to Helmholtz states that a vector field can be uniquely defined only by specifying both its curl and its divergence. We must remain aware, therefore, that at this point our vector potential is incompletely determined. The arbitrariness of Div(**A**) is frequently exploited to simplify the equations. In many cases, it is not necessary to explicitly specify Div(**A**), allowing the boundary conditions and the artifacts of the computational model to define it by default.

Substituting relation (9) into equation (3) gives Curl(**E** + dt(**A**)) = 0. Another vector identity states that Curl(Grad(f))=0 for any twice differentiable scalar f. This allows us to define a scalar potential function V such that

(10) **E** = -Grad(V) - dt(**A**).

In the absence of time variation, V is seen to be the electrostatic potential.

Application of Faraday's Law to a pillbox on a material interface shows that V must be continuous across material interfaces. Application of Stokes' Theorem to **A** shows that the tangential component of **A** must be continuous across material interfaces. All the conventional definitions of Div(A) also have the property that the normal component of **A** is continuous across material interfaces. Therefore, formulations in terms of V and **A** completely satisfy the modeling assumptions of FlexPDE.

Since the two definitions (9) and (10) satisfy equations (2) and (3), we are left with Maxwell's equations (1) and (4), which in terms of **A** and V are:

(11) Curl(Curl(**A**)/m) + s dt(**A**) + e dtt(**A**) + s Grad(V) + e dt(Grad(V)) = 0
(12) Div(e Grad(V)) + Div(e dt(**A**)) + p = 0

At this point, it is customary in the literature to apply vector identities to convert the Curl(Curl(**A**)/m) into a form containing Div(**A**), so that a complete definition of **A** can be achieved. In fact, these transformations require that m be continuous across material interfaces. We therefore defer this operation for discussion under the appropriate specializations to follow. We should also point out that in (11) we have substituted (7) **J** = s**E**, a substitution we may later wish to rescind.

IV. Boundary Conditions

FlexPDE uses the Divergence Theorem and its related Curl Theorem to reduce the order of second derivative terms, and assumes that the resulting surface integrals vanish at internal boundaries. Applied to (12), this process results in the continuity of the normal component of **D**, as required by boundary rule 1). Applied to (11), this process results in the continuity of the tangential component of **H**, as required by boundary rule 2).

At exterior boundaries, the Natural boundary condition specifies the value of the integrand of the surface integrals. For equation (11) this means the tangential component of **H**, while for equation (12) it means the normal component of **D**.

1. Symmetry planes

Following the above definition of the natural boundary condition, the specification "NATURAL(V)=0" for equation (12) means that the normal component of **D** is zero. This means that field lines must be parallel to the system boundary and that potential contours must be normal to the boundary. These are the conditions of a symmetry plane.

Similarly, if we specify "NATURAL(**A**)=0" for equation (11), we require that the tangential component of **H** be zero. This says that field lines and potential contours must be normal to the boundary, which is again the condition of a symmetry plane.

2. Perfect conductors

Since a perfect conductor cannot sustain a field, the boundary condition "VALUE(V)=constant" for equation (12) defines a perfectly conducting boundary. Note that since equation (12) contains only derivatives of V, an arbitrary constant value may be added to the solution without affecting the equation. In order for a numerical solution to succeed, there must be some point in the domain at which a value condition is prescribed, in order to make the potential solution unique.

Similarly, the specification "VALUE(**A**)=constant" for equation (11) forces the normal component of **H** to be zero. As with V, a value should be ascribed to **A** somewhere in the domain, in order to make the potential solution unique.

3. Distant Boundaries

Ampere's Law states that the integral of **H.dl** around a closed path is equal to the integral of **J.dS** over the enclosed surface, or just **I**, the enclosed current. Now, **H.dl** is the tangential component of **H**, which is exactly the quantity specified in equation (11) by the Natural boundary condition. In many cases, this fact can be used to construct meaningful terminating boundary conditions for otherwise open domains.

The differential form of Ampere's Law can also be used to derive a general rule for the value of **A:**

A(**R**) = integral [**J**(**R**')/|**R**-**R**'|] d_3**R**'

(In time-varying systems, **J** must refer to a current retarded in time by the propagation time from **R**' to **R**.)

This form has the property that Div(**A**) = 0. We may add to this definition the gradient of an arbitrary scalar function G without affecting the resulting fields.

At points distant from any currents we may write

A(**R**) -> **I**/|**R**|.

Note that here **I** is a vector which embodies the direction of the current, and that **A** has the direction of **I**.

V. Harmonic Analysis

Equations (11) and (12) describe a full time dependent model of the fields which can be extremely expensive to compute. In many cases of interest, the time dependence we desire to study is the stable oscillation caused by a sinusoidal excitation. In these cases it is convenient to make the assumption that each of the field components can be expressed in the complex form

P = ***P*** exp(iwt),

Where **P** is any of the field quantities, ***P*** is an associated complex amplitude (a function of space only),

w is the angular velocity, and i is the square root of minus 1. The observable field quantity is then the real part of **P**, Re(**P**).

With these assumptions, the time derivative terms in our equations reduce to simple forms:

$$dt(\mathbf{P}) = iw\mathbf{P}$$
$$dtt(\mathbf{P}) = -w^2\mathbf{P}$$

Applying these assumptions to equations (11) and (12) results in the harmonic equations

(13) $Curl(Curl(\boldsymbol{A})/m) + w(is-ew)\boldsymbol{A} + (s+iew)Grad(V) = 0$
(14) $Div(e\ Grad(V)) + iw\ Div(e\boldsymbol{A}) + p = 0$

These equations require solution in space only, and are thus much more economical than the full time dependent system (11), (12). We will return to these equations frequently in the sections which follow.

VI. Posing Equations for FlexPDE

We have been writing our equations in terms of vector fields, but in fact FlexPDE is not able to deal directly with vector fields; we must manually reduce the system to component equations. In a three dimensional space, equation (11) comprises three component equations while equation (12) is scalar. So we have a total of four equations in four unknowns, Ax, Ay, Az and V.

Equations (13)-(14) are more complicated, since each component has a real and an imaginary part, for a total of eight components. Each of these eight scalar variables must be represented by a separate component equation.

We will not expand the equations into their final form just yet, because in most of the specializations addressed subsequently the resulting forms are not nearly so frightening as the full equations.

VII. Specializations

In most problems of interest, the full generality of equations (11) and (12) or their harmonic equivalents (13) and (14) are not necessary. Analysis of the needs of the problem at hand can usually lead to considerable simplification. We will consider a few cases here.

1. Electrostatics

For fields which are constant in time, equation (12) decouples from equation (11), and the electric scalar potential may be found from the single equation

(15) $Div(e\ Grad(V)) + p = 0$

Since FlexPDE applies the divergence theorem over each computational cell, inclusion of e inside the divergence is sufficient to guarantee the correct behavior of the field quantities across material interfaces. The natural boundary condition on V becomes a specification of the normal derivative of (e Grad(V)).

Systems of this kind are addressed in the sample problems xxx.pde, etc.

2. Magnetostatics

For fields which are constant in time, equation (11) becomes

$$Curl(Curl(\mathbf{A})/m) + s\ Grad(V) = 0$$

Here the term s Grad(V) is in fact a representation of the current density **J**, which we will probably wish to specify directly as the driving current for the fields:

(16) Curl(Curl(**A**)/m) + **J** = 0

In the geometric interpretation of **A**, for which Div(**A**)=0, **A** has components parallel to the components of **J**, so if **J** is restricted to a single component, we may restrict **A** to only that component.

As discussed in section IV, the Natural boundary condition for **A** specifies the tangential component of **H**. Natural(**A**)=0 specifies a symmetry plane, and Value(**A**)=0 specifies a perfect conductor.

Systems of this kind are addressed in the sample problems xxx.pde, etc.

3. Non-magnetic Materials (constant m)

In the common case where m is constant, we can perform some simplification on equation (11). We can apply the vector identity

Curl(Curl(**A**)) = Grad(Div(**A**)) - Div(Grad(**A**))

To give

(17) (1/m)Div(Grad(**A**)) - s dt(**A**) - e dtt(**A**) = (1/m)Grad (Div(**A**)) + s Grad(V) + e dt(Grad(V)).

Since we now have an explicit Div(**A**), we are in a position to define it in any way we choose to generate a form appropriate to our needs. The definition of Div(A) is commonly known as the "Gauge Condition". The choice of gauge will be determined by what it is that we know about the problem at hand. Several common gauge conditions and the resulting forms of (11)-(12) are given below.

Note that this operation is not without consequences. The definition of the natural boundary condition has changed. It is no longer the boundary value of Curl(**A**)/m, but is now the boundary value of Grad(**A**)/m. Natural(**A**)=0 remains the condition for a symmetry plane, and Value(**A**)=0 still defines a perfect conductor boundary, but care must be taken if other values are assigned. In the case Div(**A**)=0, the two will be equivalent, in other choices of gauge they may not be.

Also note that because of typographical constraints we have written Div(Grad(**A**)) for the component-wise Laplacian of the vector **A**. This notation is not strictly correct in curvilinear coordinates, and a more careful derivation must be made in those cases.

Without making further assumptions about e or s, we can apply the Coulomb gauge condition,

Div(A)=0

With this assertion, equation (17) becomes

(18) Div(Grad(**A**)) - ms dt(**A**) - me dtt(**A**) = ms Grad(V) + me dt(Grad(V))
(19) Div(e Grad(V)) + Div(e dt(**A**)) + p = 0

Note that even though we have assumed Div(**A**)=0, we are not free to delete the Div(e dt(**A**)) from equation (18) unless e is also constant. Piecewise constancy of e is not sufficient, because Grad(e) is undefined at the interface and we have no way to apply a divergence theorem to convert it to a surface integral.

4. Non-magnetic Materials with constant e

In the special case where both m and e are constant, the scalar potential equation (19) with Coulomb gauge can be simplified to

(19') Div(Grad(V)) + p/e = 0.

Alternatively, we can use the "Diffusion" gauge condition:

Div(A) = -me dt(V)

We can reverse the order of differentiation and cause Div(**A**) to cancel the dt(Grad(V)) term in equation (11) and replace the dt(**A**) term in equation (12).

(20) Div(Grad(**A**)) - ms dt(**A**) - me dtt(**A**) = ms Grad(V)
(21) Div(Grad(V)) - me dtt(V) + p/e = 0

In some cases, s Grad(V) may be interpreted as the negative of the static current density, in which case the equations decouple and (20) may be eliminated.

5. Non-magnetic Materials with constant e and s

In the special case where m, e and s are all constant, we can use the Lorentz gauge condition:

Div(A) = -msV - me dt(V)

This allows all the V terms to cancel from equation (17) resulting in decoupled equations for **A** and V

(22) Div(Grad(**A**)) - ms dt(**A**) - me dtt(**A**) = 0
(23) Div(Grad(V)) - ms dt(V) - me dtt(V) + p/e = 0

The equations have now been decoupled, and may be solved separately. These forms are useful in the solution of wave propagation problems.

VIII. Specializations of the Harmonic Equations

1. Non-magnetic Materials

Equations (13) and (14) can also be specialized to the case of constant m. The basic form of equation (13) is

(24) (1/m)Div(Grad(***A***)) + w(ew-is)***A*** = (1/m)Grad(Div(***A***)) + (s + iwe) Grad(*V*).

Without making further assumptions about e or s we may apply the Coulomb gauge condition

Div(***A***)=0,

from which equations (13) and (14) become

(25) Div(Grad(***A***)) + mw(ew-is) ***A*** = m(s+iew) Grad(*V*)
(26) Div(e Grad(*V*)) + iw Div(e ***A***) + *p* = 0

Note that even though we have assumed Div(***A***)=0, we are not free to delete the Div(e ***A***) from equation (26) unless e is constant.

2. Non-magnetic Materials with constant e

In the special case where both m and e are constant, equation (26) with Coulomb gauge can be simplified to

(26') Div(Grad(*V*)) + *p*/e = 0.

Alternatively, we can use the diffusion gauge condition

Div(***A***) = -iwme *V* ,

from which we derive the equations

(27) Div(Grad(***A***)) + w(ew-is)***A*** = ms Grad(*V*)
(28) Div(Grad(*V*)) + w^2me *V* + *p*/e = 0

In some cases, s Grad(*V*) may be interpreted as the negative of the conduction current density, in which case the equations decouple and (28) may be eliminated.

3. Non-magnetic Materials with constant e and s

In the special case where m, e and s are all constant, we can use the Lorentz gauge condition, which in the harmonic approximation becomes

Div(***A***) = -m(s+iew) *V*

All the V terms vanish in equation (24), and the pair (13), (14) become

(29) Div(Grad(***A***)) + w(ew-is)***A*** = 0
(30) Div(Grad(*V*)) - m(s+iew)*V* + *p*/e = 0

The equations have now been decoupled, and may be solved separately. These forms are useful in the solution of wave propagation problems.

(to be continued...)

5.8 Smoothing Operators in PDE's

The Laplacian Operator as a Bandpass Filter Function

Assume that we have a function v(x) which we wish to smooth.
The Fourier expansion of this function is $v(x) = \sum V_k \exp(i\,k\,x)$.
Let the smoothed function be $u(x) = \sum U_k \exp(i\,k\,x)$, with k the angular velocity in radians per unit distance;
then the Laplacian of u is $\nabla^2 u = \sum (-k^2)\, U_k \exp(i\,k\,x)$.

We define u from the relation $u - \varepsilon \nabla^2 u = v$
then $\sum U_k \exp(i\,k\,x)\,(1 + \varepsilon k^2) = \sum V_k \exp(i\,k\,x)$.

Component by component, $U_k \exp(i\,k\,x)\,(1 + \varepsilon k^2) = V_k \exp(i\,k\,x)$

Or, $U_k = V_k / (1 + \varepsilon k^2)$

so that the kth frequency component is attenuated by a factor of $1/(1 + \varepsilon k^2)$.

The Sampling Theorem states (McGillem and Cooper, "Continuous and Discrete Signal and System Analysis", p 164):
" A band-limited signal can be uniquely represented by a set of samples taken at intervals spaced 1/2W seconds apart, where W is the signal bandwidth in Hz."

The sampled signal is the product of the input signal and the sampling function, and the spectrum of the sampled signal is the convolution of the two transforms. The spectrum of the sampling function is a series of impulses at the harmonics of the sampling frequency (2W), and the convolution leads to a replication of the signal spectrum around each of these harmonics. If the signal bandwidth exceeds the harmonic spacing 2W, then the harmonics will overlap, and aliasing will occur.

From this we infer that if spatial data are available at a spacing of D meters, then the maximum bandwidth in the defined signal will be $W = 1/(2D)$ cycles per meter, corresponding to $k = 2\pi W$ radians per meter.

Combining these two items, we wish to infer a value of ε that will damp components of U with frequencies above W. However, the Laplacian filter does not have a sharp cutoff at any frequency, so we have some latitude in assigning ε.

Let us find ε such that the frequency component at frequency W is attenuated by a factor N, ie.

$1/(1 + \varepsilon 4\pi^2 W^2) = 1/N$, with $1/(2W) = D$.

Then $\varepsilon = (N-1)/(4\pi^2 W^2) = D^2(N-1)/\pi 2$.

Arbitrarily choosing a frequency attenuation factor of N=2, we get $\varepsilon = D^2 / \pi^2$.

Smoothing Steady-state Solutions

In the solution of partial differential equation systems, it sometimes happens that auxiliary equations must be solved simultaneously with the PDE, and that these auxiliary equations have no spatial coupling, being point relations or other zero-order equations. In these cases, the finite element method works poorly, because the discretization is based on integrals over space, and oscillatory solutions can satisfy the integrals. In such systems, we are justified in adding to the equation a diffusion operator to impose a smoothing on the solution. If we have, for example,

$U = F(..)$

then we can replace this equation with

$U - (D^2/\pi^2)\nabla^2 U = F(..)$, with D the approximate spatial wavelength of acceptable oscillations.

Damping Time-dependent Systems

A similar analysis can be applied to time-dependent partial differential equations.

Suppose we have a system $\partial v/\partial t = f$, in which the discretized equations support high frequency solutions which destabilize the numerical solution process. We wish to damp high frequency components.

Assume that v can be expanded as $v(x,t) = \sum V_k \exp(i\,k\,(x-ct))$, where c is a propagation velocity.

Let the smoothed function be $u(x,t) = \sum U_k \exp(i\,k\,(x-ct))$,

then the Laplacian of u is $\nabla^2 u = \sum (-k^2) U_k \exp(i\,k\,(x-ct))$,

while the time derivative is $\partial u/\partial t = \sum (-ikc) U_k \exp(i\,k\,(x-ct))$.

We define u from the relation $\partial u/\partial t - \varepsilon \nabla^2 u = \partial v/\partial t$
then $\sum U_k \exp(ik(x-ct)) (-ikc + \varepsilon k^2) = \sum V_k \exp(i k (x-ct))(-ikc)$.

Component by component, $U_k = V_k (-ikc)/(\varepsilon k^2 - ikc)$

Or, $|U_k| = |V_k|/\mathrm{sqrt}(1 + \varepsilon^2 k^2/c^2)$

so that the kth frequency component is attenuated by a factor of

$$1/\mathrm{sqrt}(1 + \varepsilon^2 k^2/c^2).$$

Again defining W = 1/(2D) and seeking an attenuation factor of 2, we get

$$\varepsilon^2 = (N^2-1)c^2/(4\pi^2W^2) = D^2(N^2-1)c^2/\pi^2 = 3D^2c^2/\pi^2,$$

or approximately, $\varepsilon = 2Dc/\pi$.

We can now solve the equation $\partial u/\partial t - \varepsilon \nabla^2 u = f$, with the expectation that u will be a frequency-filtered representation of v.

Steady-state limits of Time-dependent Equations

In some cases, a steady-state limit of a known time-dependent system is desired, but while the time-dependent equation itself is stable, the steady-state equation which results from merely setting the time derivative to zero is not. In these cases, we can replace the time derivative by $-\varepsilon \nabla^2 u$, again with the expectation that u will be a frequency-filtered representation of v.

5.9 3D Mesh Generation

FlexPDE version 4.0 introduced an entirely new mesh generator for 3D problems. With support for LIMITED REGIONS, it offers users much more flexibility in the creation of 3D domains. It is also a much more complex computation, and is sometimes in need of some user assistance to successfully create a mesh for complex 3D problems.

The greatest challenge faced by the 3D mesh generator is the transition across wide ranges of feature sizes. Any help the user can give in easing this transition will be amply rewarded in a decreased incidence of mesh generation failure. We at PDE Solutions are also engaged in improving the intelligence of the mesh generator to also assist in reaching this goal.

DOMAIN REVIEW

The first facility that users should be aware of is the "Domain" item on the main menu bar. Selecting this item instead of "Run" will give the user a step-by-step review of the mesh generation process. This review reflects the order of operations performed by the mesh generator.

- The first sequence of displays shows the domain boundaries in the surfaces and layers of the extrusion. The first plot shows the domain boundaries present in the bottom surface; the next shows the boundaries which extend through the first layer; then the boundaries present in the second extrusion surface; and so on through entire domain, and ending with the top surface. You should examine each of these displays to determine that the structure is as you intended. Errors at this point can create serious trouble later.

- After the individual surfaces and layers are displayed, a composite 3D display is presented of the total domain, as represented by boundaries. This plot can be rotated to examine all aspects of the domain.
- The next sequence of displays shows the triangular surface meshes created for the extrusion surfaces. These meshes are created in 3D space, but are displayed as projections into the (X,Y) base plane. This presentation reflects the fact that the surfaces are first meshed as independent surfaces in space. Following initial surface mesh creation, the meshes are refined to create sufficient resolution of surface curvature. They are then analyzed for proximity, and coarser meshes are refined due to influence from nearby dense meshes.
- The next sequence of displays shows the creation of the tetrahedral 3D meshes for each of the regions and layers of the domain. Before a block is filled, the bounding surface is shown; after filling, the filled block is displayed (it looks the same). The sequence presents first the region blocks for layer 1, followed by a unified mesh of layer 1. This pattern is repeated through the layers of the domain, until finally a unified 3D mesh is displayed. At this point, the mesh is composed of linear (straight-sided) tetrahedra.
- Once the domain is filled with linear tetrahedra, the additional nodes needed for quadratic or cubic interpolation. Cells are also bent at this point to conform to curved boundaries. This curving can create troubles in thin curved shells. The 3D mesh generator is not yet smart enough to compute shell thickness and curvature and automatically adapt the size. You may have to do it manually. Sqrt(Radius*thickness) is a good rule of thumb.
- This completes the mesh generation process, and solution should proceed promptly.

DEALING WITH FAILURE

The most common cause of failure is inability to make the transition from very small to very large feature sizes without tangling. If the mesh generation fails, the user has several options, all involving some kind of manual mesh density control.

- The simplest way of dealing with mesh generation failure is simply to increase the NGRID selector. This causes the entire mesh to be more dense, and also more regular. In some cases, it may create a mesh which is simply too large for effective computing with the available computer resources.
- A second approach is to use the MESH_SPACING control to increase the overall density in a troublesome region, layer or surface. Remember that MESH_SPACING can be specified as arbitrary functions of spatial coordinate, allowing dense meshes in specific locales.
- The ASPECT control can be used to increase the cell sizes in thin components, thereby reducing the range of sizes that must be dealt with in surrounding media. Increasing ASPECT can create elongated cells in surrounding media, so you may need to balance its use by explicitly controlling MESH_SPACING in these regions.
- You can localize the problem areas by building your domain one layer at a time. Build the first layer and examine the regional meshes for compliance with your expectations. Then add the next layer. You might at this point want to delete the first layer, so you can deal with the second layer as an independent item.

5.10 Interpreting Error Estimates

FlexPDE uses estimates of the modeling error to control mesh refinement and timestep size. This note describes the methods used and the interpretation of the reports.

Spatial Error

The Galerkin Finite Element method uses integrals of the PDE's to form the discretized equations at the mesh nodes.

Each nodal equation requires that the weighted integral of the associated equation over the mesh cells surrounding the node be satisfied within a convergence tolerance. In FlexPDE this tolerance is taken to be

a relative error of (ERRLIM * OVERSHOOT) in the norm of the solution vector.

In a regular hexagonal 2D mesh, for example, the Galerkin method requires that each hexagonal set of six triangular mesh cells must produce a weighted integral residual of zero.

This method at no point imposes any conditions on the integral over a single mesh cell, and conceivably on could have cancelling errors in adjacent cells.

In FlexPDE, we choose to use the individual cell integrals as a measure of the mesh quality. If the aggregate (eg 6-cell) integral is correct but the individual cells show large error, then the mesh must be refined.

The fundamental system which is solved by FlexPDE can be indicated as R=G(U)=0, where R is the residual and G(U) is the Galerkin integral of the PDE for variable U. If the residual over an individual cell is R, we can write J*dU=R, where J is the Jacobian matrix of derivatives of the Galerkin integral with respect to the nodal values, and dU is the error in U which produces the residual R.

J is of course the coupling matrix which is solved to produce the solution U. We don't want to completely repeat the solution process just to get an error estimate, so we use D, the diagonal components of J, to produce the error estimate dU=Inv(D)*R.

The "RMS Error" reported by FlexPDE in the Status Panel is just the root-mean-square average of dU/range(U) over the cells of the problem, while the reported "MAX Error" is the largest error dU/range(U) seen in any cell.

Mesh cells for which dU/range(U) > ERRLIM are split in the mesh refinement pass.

Notice that the error measure is not a guarantee that the computed solution is "accurate" to within the stated error, that is, that the computed solution differs from the "true" solution by no more than the stated error. The error estimate is a local measure of how much variation of the solution would produce the computed error in the cell integral. Deviations from the "true" solution might accumulate over the domain of the problem, or they might cancel in neighboring regions.

Temporal Error

In time dependent problems, an estimate must also be formed of the error in integrating the equations in time.

FlexPDE integrates equations in time using a second-order implicit Backward Difference Formula (Gear method).

In order to measure temporal error, FlexPDE stores an additional timestep of values previous to the three points of the quadratic solution, and fits a cubic in time to the sequence at each node. The size of the cubic term implies the error in the quadratic solution, and is used to either increase or decrease the timestep in order to keep the RMS temporal error within the range specified by ERRLIM.

The three-point integration method requires an independent method to create data for the initial interval. FlexPDE uses a comparison of one-step and two-step trapezoidal rule integration to adapt the initial timestep to a range that produces acceptable error.

The temporal error estimate is not currently reported on the status panel.

FlexPDE Error Controls

There are several SELECT controls that can be used to alter the behavior of FlexPDE in regard to error measures.

The basic control is ERRLIM, which specifies the desired relative error in the solution variables, and controls both spatial and temporal measures. Smaller ERRLIM causes more mesh subdivision and smaller timesteps. Larger ERRLIM allows cruder meshes and, in principle, larger timesteps. However, a large ERRLIM can allow oscillations to develop, ultimately causing severe timestep cuts and a slower overall execution. It is rarely advisable to use an ERRLIM value *larger* than the default 0.001.

XERRLIM and TERRLIM are analogous to ERRLIM, but refer specifically to the spatial and temporal controls, allowing separate control of the two processes. If either of these controls is absent, it defaults to the value of ERRLIM.

5.11 Coordinate Scaling

FlexPDE treats all spatial coordinates on an equal footing, and tries to create meshes that are balanced in size in all coordinates.
Sometimes, though, there are problems in which one dimension is expected to have much less variation that the others, and fully meshing the domain with equilateral cells creates an enormous and expensive mesh. In these cases, it would be advantageous to scale the long dimension to bring the coordinate sizes into balance. Similarly, in semiconductor problems, for example, the structure is extremely thin, and would benefit from an expansion of the Z thickness coordinate.

It is possible that FlexPDE will eventually implement automatic coordinate scaling, but in the meantime, users can implement it manually.

Consider as an example the heat equation

```
div(k*grad(T))+Q = C*dt(T)
```

with k the conductivity, Q a source and C the heat capacity.

Define a coordinate transformation,

```
z = s*w
```

where w is the physical coordinate, z is the FlexPDE coordinate, and s is a scaling factor.
The expanded physical equation is then

```
dx(k*dx(T)) + dy(k*dy(T)) + dw(k*dw(T)) + Q = C*dt(T)
```

We can transform the heat equation using this transformation and observing that

```
dw(f) = (∂f/∂w) = (∂f/∂z)*(∂z/∂w) = s*dz(f)
```

The result is

(1) `dx(k*dx(T)) + dy(k*dy(T)) + s*dz(k*s*dz(T)) + Q = C*dt(T)`

Flux Conservation

In forming the finite element model for this equation, FlexPDE assumes continuity of the surface integrals generated by integration-by-parts of the second-order terms (equivalent in this case to the Divergence Theorem). This is the Natural Boundary Condition for the equation, as discussed elsewhere in the FlexPDE documentation.

The z-directed flux terms in the transformed equation therefore assume that s^2*k*dz(T) is continuous across cell interfaces. This is equivalent to flux conservation in the physical system as long as s is constant throughout the domain.

In order to guarantee conservation of flux in the presence of differing scale factors in layers, we must have the following equality across an interface between materials 1 and 2:

$k1*dw(T)_1 = k2*dw(T)_2$

or

$k1*s1*dz(T)_1 = k2*s2*dz(T)_2$

This will be satisfied if we divide our transformed equation by s :

(2) dx(k*dx(T))/s + dy(k*dy(T))/s + dz(k*s*dz(T)) + Q/s = C*dt(T)/s

where s is defined as s1 in material 1 and s2 in material 2.

Un-Scaling Fluxes

Fluxes appropriate to the unscaled system can be recovered by the same modifications as those made in the PDE:

- Fluxes in the scaled direction must be *multiplied* by the scale factor. Integrals of these fluxes need not be further modified, as they are integrated over surfaces in true coordinates.
- Fluxes in the unscaled directions are correctly computed in true coordinates, but when integrated over surfaces, they must be *divided* by the scale factor to account for the scaled area.

Flux integrals then appear in the same form as in the scaled PDE:

```
Total_Real_Flux = Surf_Integral(NORMAL(-k*dx(T)/s, -k*dy(T)/s, -k*dz(T)*s)
```

Natural Boundary Conditions

The natural boundary condition defines the argument of the outermost derivative operator (or the argument of the divergence). In the conservative equation (2):

- Components in the unscaled direction have been divided by s. Therefore the natural boundary conditions for these components must be divided by s. (e.g. NATURAL(T) = x_flux/s on x-normal surfaces.)
- In the scaled direction, the value defined by the natural is k*s*dz(T) which is in fact k*dw(T), the flux in the physical coordinate system. The natural in the scaled direction is therefore unmodified by the scaling. (e.g. NATURAL(T) = z_flux on z-normal surfaces.)

Examples

"Samples | Usage | Coordinate_Scaling | Scaled_Z.pde" 455 shows the implementation of this technique.
"Samples | Usage | Coordinate_Scaling | UnScaled_Z.pde" 456 provides an unscaled reference for comparison.

5.12 Making Movies

Since version 5, FlexPDE has had a simplified the process of creating movies from problem data.

1) Replaying a movie from a stored .PG6 file:
- Open a .PG6 file from the "View | View File" menu.
- You can use the "View | Frame Delay" menu item to set the delay between frames (default 500 ms).
- Double-click to maximize a selected frame in the thumbnail display
- Click "View | Movie" to replay all the instances of the selected frame.
- Click "View | Restart" whenever you wish to begin a new replay, to move the reader to the beginning of the file.

2) Exporting a Movie from a stored .PG6 file to graphic files on disk:
- Open a .PG6 file from the "View | View File" menu.
- Double-click a thumbnail to maximize a selected frame.
- Click "View | Export Movie". This will bring up a selection dialog to set the export parameters.
- The selected frame will be scanned as for Movie, and all files will be written according to the selected parameters.
- Use JASC AnimationShop to assemble the individual files into a GIF animation.
- Use GIF2SWF or other conversion program to create Flash animations.

See Viewing Saved Graphic Files 19 for more information.

5.13 Converting from Version 4 to Version 5

Several items have been changed in version 5 that may require some attention for users of FlexPDE version 4. In general, we have tried to make the transition as simple as possible.

- ERROR ESTIMATION: The algorithms used for error estimation have been changed in version 5. In most cases, the new measures are more pessimistic that those used in version 4, resulting in some cases in more intense mesh refinement and longer running. Nevertheless, we feel that the new algorithms provide an error measure closer to the actual disparity between the numerical and analytical solutions in test problems. In order to ameliorate the impact of this change, we have relaxed the default ERRLIM to 0.002 and allowed individual cells to exceed the ERRLIM specification, as long as a weighted average of errors is below ERRLIM. You may wish to adjust your ERRLIM specifications to reflect this new behavior.

- SMOOTHING INITIAL VALUES: Version 5 applies a smoothing procedure to initial conditions in time-dependent problems, to ameliorate the harsh behaviour caused by discontinuous initial conditions. In most cases, you will experience a much quicker startup, with no significant difference in solution. The smoothing operation is scaled to cell sizes, so you can recover accurate resolution of initial transients by merely specifying dense meshing at important initial discontinuities. The smoothing operation can be suppressed by SELECT SMOOTHINIT=OFF.

- CLOSE: The reserved word FINISH used in previous versions has been changed to CLOSE, to more accurately reflect its function. You will be warned once, after which FINISH will be accepted as in version 4. Except in cases where you want to run a problem on both versions, we suggest converting to the new format.

- GLOBAL: The designation SCALAR VARIABLES used in version 4 has been changed to GLOBAL VARIABLES, to more accurately reflect its function. You will be warned once, after which SCALAR VARIABLES will be accepted as in version 4. Except in cases where you want to run a problem on both

versions, we suggest converting to the new format.

5.14 Converting from Version 5 to Version 6

FlexPDE version 6 is almost totally backward-compatible with version 5.

In order to support the new features of version 6, however, we have had to make a few syntactic changes:

Parentheses
Parens "( )" are no longer interchangeable with square brackets "[ ]". In particular,

- Square brackets can no longer be used in expression grouping. They are reserved for array and matrix indexing.
- Parentheses can no longer be used for array indexing. Only square brackets will serve in this capacity.

Exponentiation
Double-asterisk "**" can no longer be used as an exponentiation operator. Double-asterisk is now the matrix multiply operator. Use the caret "^" for exponentiation.

Solution Controls

- Error estimation algorithms are somewhat different, and may result in somewhat shorter timesteps and longer running for time-dependent problems. These changes were made in the interest of more truthful reports of error.
- The selector NRMATRIX has been changed to an ON/OFF selector REMATRIX which selects recomputation of the Jacobian matrix on every Newton iteration. The default is OFF. Even without this selector, FlexPDE will recompute the Jacobian matrix whenever the variable changes are greater than an internal threshold.
- Nonlinear time-dependent problems default to one Newton step per timestep, with timestep controls to cut the timestep if convergence is not readily achieved. This is usually a more efficient scheme than other alternatives. The Selector NEWTON=number is available for specifying a more strenuous convergence policy. The Selector PREFER_STABILITY can be used to allow up to 5 Newton iterations per timestep, with full re-computation of the Jacobian matrix on each iteration. This is the most expensive option, but should provide the most stable operation.

Reserved Names
The names REAL and IMAG can no longer be used as user-defined values. They are now built-in component selectors for Complex data types. See the list of Reserved Words 121 for other changes.

Part VI

Sample Problems

6 Sample Problems

When FlexPDE is installed, a directory of sample problems is placed in the installation folder. These sample problems have been prepared by PDE Solutions staff and show various applications of FlexPDE, or illustrate features or techniques. Many of these problems contain commentary describing the derivation of the model. All are keyed for execution by the Evaluation version of FlexPDE.

6.1 applications

6.1.1 chemistry

6.1.1.1 chemburn

```
{  CHEMBURN.PDE

   This problem models an extremely nonlinear chemical reaction in an open tube
   reactor with a gas flowing through it. The problem illustrates the use of
   FlexPDE to solve mixed boundary value - initial value problems and involves
   the calculation of an extremely nonlinear chemical reaction.

   While the solutions sought are the 3D steady state solutions, the problems
   are mixed boundary value / initial value problems with vastly different
   phenomena dominating in the radial and axial direction.

   The equations model a cross-section of the reactor which flows with the
   gas down the tube.  There is therefore a one to one relation between the
   time variable used in the equations and distance down the tube given by
   z = v*t.

   The chemical reaction has a reaction rate which is exponential in
   temperature, and shows an explosive reaction completion, once an
   'ignition' temperature is reached.  The problem variable 'C' represents
   the fractional conversion (with 1 representing reaction completion).
   The reaction rate 'RC' is given by

        RC(C,Temp) = (1-C)*exp[gamma*(1-1/Temp)]

   where the parameter GAMMA is related to the activation energy of the
   reaction.

   The gas is initially at a temperature of 1, in our normalized units, with
   convective cooling at the tube surface coupled to a cooling bath at a
   temperature of 0.92.

   The problem is cylindrically symmetric about the tube axis. Because of
   the reaction the axis of the tube will remain hotter than the periphery,
   and eventually the reaction will ignite on the tube axis, sending
   completion and temperature fronts propagating out toward the wall. For
   small GAMMA, these fronts are gentle, but for GAMMA greater than about
   twelve the fronts becomes very steep and completion is reached rapidly
   and sharply creating very rapid transition from a very high reaction rate
   reaction rate to a zero reaction rate.  The adaptive gridding and
   adaptive evolution 'time' stepping capabilities of FlexPDE come into
   play in this extreme nonlinear and process nonisotropic problem,
   allowing a wave of dense gridding in time to accompany the completion
   and temperature fronts across the tube.

   In this problem we introduce a heating strip on the two vertical
   faces of the tube, for a width of ten degrees of arc.  These strips are
   held at a temperature of 1.2, not much above the initial gas temperature.
   The initial timesteps are held small while the abrupt temperature gradient
   at the heating strips diffuses into the gas.

   As the cross-section under study moves down the reactor, the heat generated
   by the reaction combines with the heat diffusing in from the strip heater to cause
   ignition at a point on the x-axis and cause the completion front and temperature
   front to progate from this point across the cross-section.

   We model only a quarter of the tube, with mirror planes on the X- and Y-axes.
   The calculation models a cross-section of the tube, and this cross-section
   flows with the gas down the tube.

   The "cycle=10" plots allow us to see the flame-front propagating across
```

```
  the volume, which happens very quickly, and would not be seen in a
  time-interval sampling.

  While the magnitudes of the numerical values used for the various
  constants including gamma are representative of those found with real
  reactions and real open tube reactors they are not meant to represent
  a particular reaction or reactor.

 }

title
  'Open Tube Chemical Reactor with Strip Heater'

select
   painted      { make color-filled contour plots }

variables
  Temp(threshold=0.1)
  C(threshold=0.1)

definitions
  Lz = 1
  r1=1
  heat=0
  gamma = 16
  beta = 0.2
  betap = 0.3
  BI = 1
  T0 = 1
  TW = 0.92
  { the very nasty reaction rate: }
  RC = (1-C)*exp(gamma-gamma/Temp)
  xev=0.96      { some plot points }
  yev=0.25

initial values
  Temp=T0
  C=0

equations
  Temp:     div(grad(Temp)) + heat + betap*RC = dt(Temp)
  C:        div(grad(C)) + beta*RC = dt(C)

boundaries
  region 1
    start (0,0)

    { a mirror plane on X-axis }
    natural(Temp) = 0
    natural(C) = 0
    line to (r1,0)

    { "Strip Heater" at fixed temperature }
    { ramp the boundary temp in time, because  discontinuity is costly to diffuse }
    value(Temp)=T0 + 0.2*uramp(t,t-0.05)

    natural(C)=0                { no mass flow on strip heater }
    arc(center=0,0) angle 5

    { convective cooling and no mass flow on outer arc }
    natural(Temp)=BI*(TW-Temp)
    natural(C)=0
    arc(center=0,0) angle 85

    { a mirror plane on Y-axis }
    natural(Temp) = 0
    natural(C) = 0
    line to (0,0) to close

time 0 to 1

plots
  for cycle=10                  { watch the fast events by cycle }
    grid(x,y)
    contour(Temp)
    contour(C) as "Completion"

  for t= 0.2 by 0.05 to 0.3     { show some surfaces during burn }
    surface(Temp)
```

```
    surface(C) as "Completion"

histories
  history(Temp) at (0,0) (xev/2,yev/2) (xev,yev) (yev/2,xev/2) (yev,xev)
  history(C) at (0,0)  (xev/2,yev/2) (xev,yev) (yev/2,xev/2) (yev,xev) as "Completion"

end
```

6.1.1.2 melting

```
{  MELTING.PDE

  This problem shows the application of FlexPDE to the melting of metal.

  We choose as our system variables the temperature, "temp", and the
  fraction of material which is solid at any point, "solid".

  The temperature is given by the heat equation,

        rho*cp*dt(temp) - div(lambda*grad(temp)) = Source

  where cp is the heat capacity, rho the density and lambda the conductivity.

  The latent heat, Qm, is an amount of heat released as "Solid" changes from
  zero to one.  We have Qm = integral[0,1]((dH/dSolid)*dSolid), or assuming
  dH/dSolid is constant, dH/dSolid = Qm.
  Then heat source from freezing is

        dH/dt = (dH/dSolid)*(dSolid/dt) = Qm*dt(Solid).

  We assume that the solid fraction can be represented by a linear ramp from
  one down to zero as the temperature passes from (Tm-T0/2) to (Tm+T0/2).

        solid = 1                   when  temp  <  Tm-T0
                (Tm+T0/2-temp)/T0   when  Tm-T0 <= temp <= Tm+T0
                0                   when  temp  >  Tm+T0

  where Tm is the melting temperature, and T0 is a temperature range over
  which the melting transition occurs.  Since there are no spatial derivatives
  in this equation, we introduce a diffusion term with small coefficient to act
  as a noise filter.

  The particular problem addressed here is a disk of cold solid material
  immersed in a bath of liquid.  The initial temperatures are such that material
  first freezes onto the disk, but after  equilibrium is reached all the material
  is liquid.    The outer boundary is insulated.

  Since the initial condition is a discontinuous distribution, we use a separate
  REGION 183 to define the cold initial disk, so that the grid lines will follow the
  shape.  We also add a  FEATURE 187 bounding the disk  to help the gridder
  define the abrupt transition.  SELECT 145 SMOOTHINIT 146 helps minimize
  interpolator overshoots.

}

TITLE
  'Melting Metal'

COORDINATES
  ycylinder('r','z')

SELECT
  smoothinit
  threads=2

VARIABLES
  temp(threshold=1)
  solid(threshold=0.01)

DEFINITIONS

  Qm= 225000          { latent heat }
  Tm=1850             { Melting temperature }
  T0= 20              { Melting interval +- T0 }
  temp_liq=2000       { initial liquid temperature }
  temp_sol=400        { initial solid temperature }
  Tinit
```

```
  sinit

  R_inf = 0.7        { Domain Radius m}

  { plate }
  d=0.05
  dd = d/5           { a defining layer around discontinuity }
  R_Plate=0.15


  lambda = 30+4.5e-5*(temp-1350)^2 { Conductivity }
  rho=2500                         { Density kg/m3 }
  cp = 700                         { heat capacity }

INITIAL VALUES
  temp=Tinit
  solid =  0.5*erfc((tinit-Tm)/T0)

EQUATIONS
  temp:  rho*cp*dt(temp) - div(lambda*grad(temp)) = Qm*dt(solid)
  solid: solid - 1e-6*div(grad(solid)) = RAMP((temp-Tm), 1, 0, T0)

BOUNDARIES

  region 'Outer'
     Tinit = temp_liq
     sinit = 0
     start 'outer' (0,-R_inf)
       value(temp)= temp_liq    arc(center=0,0) angle 180
       natural(temp)=0          line to close

  region 'Plate'
     Tinit = temp_sol
     sinit = 1
     start(0,0)
       mesh_spacing=dd
       line to (R_Plate,0) to (R_Plate,d) to (0,d) to close

TIME  0 by 1e-5 to 600

MONITORS
 for cycle=10
   grid(r,z) zoom (0,-0.1,0.25,0.25)
   elevation(temp)  from(0.1,-0.1) to (0.1,0.15) range=(0,2000)
   elevation(solid) from(0.1,-0.1) to (0.1,0.15) range=(0,1)

PLOTS
   for t= 0 1e-4 1e-3 1e-2 0.1 1 10 by 10 to 100 by 100 to 300 by 300 to endtime

   contour(temp)    range=(0,2000)
   contour(temp)    zoom (0,-0.2,0.45,0.45) range=(0,2000)
   elevation(temp)  from(0.1,-0.1) to (0.1,0.15) range=(0,2000)
   contour(solid)   range=(0,1)
   contour(solid)   zoom (0,-0.2,0.45,0.45) range=(0,1)
   surface(solid)   zoom (0,-0.2,0.45,0.45) range=(0,1) viewpoint(1,-1,30)
   elevation(solid) from(0.1,-0.1) to (0.1,0.15) range=(0,1)

HISTORIES

   history(temp)  at (0.051,d/2) at (0.075,d/2) at (R_plate,d/2)
   history(temp)  at (0.051,d)   at (0.075,d)   at (R_plate,d)
   history(solid) at (0.051,d/2) at (0.075,d/2) at (R_plate,d/2)
   history(solid) at (0.051,d)   at (0.075,d)   at (R_plate,d)
   history(integral(cp*temp+Qm*(1-solid))) as "Total Energy"


END
```

6.1.1.3 reaction

```
{   REACTION.PDE

    This example shows the application of FlexPDE to the solution
    of reaction-diffusion problems.

    We describe three chemical components, A,B and C, which react and
    diffuse, and a temperature, which is affected by the reactions.
```

```
        I) A combines with B to form C, liberating heat.
       II) C decomposes to A and B, absorbing heat. The decomposition rate
           is temperature dependent.
      III) A, B, C and Temperature diffuse with differing diffusion constants.

     The boundary of the vessel is held cold, and heat is applied
     to a circular exclusion patch near the center, intended to model an
     immersion heater.

     A, B and C cannot diffuse out the boundary.

     The complete equations including the Arrhenius terms that describe
     the system are:

      div(Kt*grad(Temp)) + heat + K1*exp(-H1/(Temp+273))*eabs*A*B
           - K2*exp(-H2/(Temp+273))*eabs*C*(Temp+273) = 0
      div(Ka*grad(A)) - K1*exp(-H1/(Temp+273))*A*B
           + K2*exp(-H2/(Temp+273))*C*(Temp+273) = 0
      div(Kb*grad(B)) - K1*exp(-H1/(Temp+273))*A*B
           + K2*exp(-H2/(Temp+273))*C*(Temp+273) = 0
      div(Kc*grad(C)) + K1*exp(-H1/(Temp+273))*A*B
           - K2*exp(-H2/(Temp+273))*C*(Temp+273) = 0

     where Kt,Ka,Kb and Kc are the diffusion constants, EABS is the heat
     liberated when A and B combine, and HEAT is any internal heat source.

     Notice that the system is non-linear, as it contains terms involving
     A*B and C*Temp.

     There are an infinite number of solutions to these equations, differing
     only in the total particle count.  In reality, since particles are
     conserved, the final solution is uniquely determined by the initial
     conditions.  But this fact is not embodied in the steady-state equations.
     The only way to impose this condition on the steady-state system
     is through an integral constraint equation, which describes the
     conservation of total particle number.

}

  title "Chemical Beaker"

  variables          { declare the system variables }
    temp,a,b,c

  definitions
    kt = 0.001      { define the diffusivities }
    ka = 0.005
    kb = 0.02
    kc = 0.01

    heat = 0        { define the volume heat source }
    eabs = 0.0025   { define the reaction energy }

    K1 = 1          { Reaction rate coef for  A + B -> C }
    H1 = 10         { Activation energy/K for A + B -> C }
    K2 = 0.0025     { Reaction rate coef for  C -> A + B }
    H2 = 200        { Activation energy/K for C -> A + B }

    a0 = 0.1        { define the initial distribution }
    b0 = 0.1        { (we will need this for the constraint) }
    c0 = 0.01

    tabs =  Temp+273
    tfac1 = K1*exp(-H1/tabs)
    tfac2 = K2*exp(-H2/tabs)

  initial values     { Initialize the variables }
    temp = 100*(1-x^2-y^2)
    a = a0
    b = b0
    c = c0

  equations          { define the equations }
    temp:  div(kt*grad(Temp)) + heat + tfac1*eabs*a*b - tfac2*eabs*c*tabs = 0
    a:     div(ka*grad(a)) - tfac1*a*b + tfac2*c*tabs = 0
    b:     div(kb*grad(b)) - tfac1*a*b + tfac2*c*tabs = 0
    c:     div(kc*grad(c)) + tfac1*a*b - tfac2*c*tabs = 0

  constraints        { demand particle conservation }
```

```
    integral(a+b+2*c) = integral(a0+b0+2*c0)

  boundaries
    Region 1
      { the cold outer boundary - impermeable to the chemicals }
      start(0,-1)
        value(temp)= 0
        natural(a) = 0
        natural(b) = 0
        natural(c) = 0
      arc to (1,0) to (0,1) to (-1,0) to close

      { the hot inner boundary - also impermeable to the chemicals }
       start(-0.2,0)
        value(temp)= 100
         natural(a) = 0
         natural(b) = 0
         natural(c) = 0
      arc(center=-0.2,-0.2) angle 360

  monitors
     contour(temp)
     contour(a)
     contour(b)
     contour(c)

  plots
     contour(temp)
     contour(a)
     contour(b)
     contour(c)
     surface(temp) as "temperature"
     surface(a) as "A-concentration"
     surface(b) as "B-concentration"
     surface(c) as "C-concentration"

  end
```

6.1.2 control

6.1.2.1 control_steady

```
{  CONTROL_STEADY.PDE

    This example shows the use of a GLOBAL VARIABLE[157] in a control application.
    We wish to find the required power input to a heater, such that the resulting
    average temperature over the domain is a specified value.

    Notice that the equation nominally defining power does not explicitly reference
    the power variable, but is coupled through the heat term in the temperature
    equation.

}

TITLE "steady-state Control test"

VARIABLES
  temp      { The temperature field }

GLOBAL VARIABLES
  power     { a single value for input power }

DEFINITIONS
  setpoint=700       { the desired average temperature }
  skintemp=325       { fixed outer boundary temperature }
  k=1                { conductivity }
  heat=0             { the heat function for the temperature.
                       it is non-zero only in the heater region }

  tcontrol=integral(temp)/integral(1)   { the control function, average temperature }
{  tcontrol=val(temp,0,0)     -- an alternative control method, unused here }

INITIAL VALUES
  temp = setpoint
  power= 100         { initial guess for power  }

EQUATIONS
```

```
  temp:   div(-k*grad(temp))-heat = 0   { diffusion of temperature field }
  power:  tcontrol = setpoint           { single equation defining power }

BOUNDARIES

  REGION 'Insulation'
    k=0.1
    heat=0
    start(-4,-4)
      value(temp)=skintemp
    line to (4,-4) to (4,4) to (-4,4) to close

  REGION 'Heater'
    k=50
    heat=power
    start(-1,-1) line to (1,-1) to (1,1) to (-1,1) to close

MONITORS
  contour(temp)
    report power
    report tcontrol

PLOTS
  contour(temp)
    report power
    report tcontrol as "Average Temp"
  elevation(temp) from(-4,0)  to (4,0)
  elevation(temp) from(-4,-4) to (4,4)

END
```

6.1.2.2 control_transient

```
{  CONTROL_TRANSIENT.PDE

    This example shows the use of a GLOBAL VARIABLE 157 in a control application.
    We wish to find the required power input to a heater, such that the resulting
    average temperature over the domain is a specified value.

    The temperature on the outer surface is prescribed, with a time-sinusoidal
    oscillation.

    The input power is driven by a time-relaxation equation.  The coefficient
    of the right hand side is the reciprocal of the response time of the power.

    This problem is a modification of CONTROL_STEADY.PDE 293, showing the use of
    time-dependent GLOBAL equations.

}

TITLE "Time-dependent Control test"

VARIABLES
  temp      { The temperature field }

GLOBAL VARIABLES
  power (threshold=0.1)     { a single value for input power }

DEFINITIONS

  setpoint = 700            { the desired average temperature }
  skintemp = 325+20*sin(t)  { oscillating outer boundary temperature }
  responsetime = 0.1        { response time of the power input }
  k=1       { conductivity }
  heat=0    { the heat function for the temperature.
                it is non-zero only in the heater region }

  { the control function, average temperature }
  tcontrol = integral(temp)/integral(1)
  {  tcontrol = val(temp,0,0)    -- an alternative control method, which tracks the
                                 temperature value at a specified point (unused here) }

  {initial guess for temperature distribution }
  tinit1=min(1767-400*abs(x), 1767-400*abs(y))

INITIAL VALUES
  temp=min(1500,tinit1)
```

```
  power=137     { initial guess for power  }

EQUATIONS
  temp:   div(-k*grad(temp))-heat = 0   { diffusion of temperature field }
  { single equation defining power.  response time is 1/100  }
  power:  dt(power) = (setpoint - tcontrol)/responsetime

BOUNDARIES

    REGION 'Insulation'
        k=0.1
        heat=0
        start(-4,-4)
            value(temp)=skintemp
        line to (4,-4) to (4,4) to (-4,4) to close

    REGION 'Heater'
        k=50
        heat = power
        start(-1,-1)  line to (1,-1) to (1,1) to (-1,1) to close

TIME 0 to 20 by 1e-4

PLOTS
 for cycle=10
  contour(temp)
    report power
    report tcontrol as "Avg Temp"

HISTORIES
  History(tcontrol-setpoint, skintemp-325) as "Skin Temperature and Error"
  History(tcontrol-setpoint) as "Controlled temperature error"
  History(power)

END
```

6.1.3 electricity

6.1.3.1 3d_capacitor

```
{  3D_CAPACITOR.PDE

  This problem is an extension of "3D_EXTRUSION_SPEC.PDE" 406 , and shows
  a capacitor formed by two metal strips of different size separated
  by a sheet of dielectric.

}

TITLE '3D Capacitor'

COORDINATES
  CARTESIAN3

SELECT
  { rename the axes }
  alias(x) = "X(mm)"
  alias(y) = "Y(mm)"
  alias(z) = "Z(mm)"
  { paint all contours }
  PAINTED

VARIABLES
  V

DEFINITIONS
  Kdiel= 6
  Kmetal=1e6
  Kair=1
  K = Kair  { default K to Kair - this will change in some layers/regions }
  V0 = 0
  V1 = 1
  Eps0 = 8.854e-12                { Farads/M }
  Eps0mm = 0.001*Eps0             { Farads/mm }
  W = integral(0.5*K*eps0mm*grad(V)^2)  { Stored Energy }
  C = 1.0e6*2*W/(V1-V0)^2         { Capacitance in microFarads }

EQUATIONS
```

```
  V : DIV(K*GRAD(V)) = 0

EXTRUSION
  SURFACE    "Bottom"                     Z=0
    LAYER    "Bottom Air"
  SURFACE    "Bottom Air - Metal"         Z=0.9
    LAYER    "Bottom Metal"
  SURFACE    "Bottom Metal - Dielectric" Z=1
    LAYER    "Dielectric"
  SURFACE    "Top Metal - Dielectric"    Z=2
    LAYER    "Top Metal"
  SURFACE    "Top Metal - Air"           Z=2.1
    LAYER    "Top Air"
  SURFACE    "Top"                        Z=3

BOUNDARIES
  SURFACE "Bottom" NATURAL(V)=0  { Insulators top and bottom }
  SURFACE "Top"    NATURAL(V)=0

  REGION 1  { this is the outer boundary of the system }
      LAYER "dielectric" K = Kdiel  { all other layers default to Kair }
      START(0,0)
      LINE TO (5,0)  TO (5,5)  TO(0,5)  to close

   LIMITED REGION 2 { the larger bottom plate }
      SURFACE "Bottom Air - Metal"         VALUE(V)=V0
      SURFACE "Bottom Metal - Dielectric" VALUE(V)=V0
      LAYER "Bottom Metal" K = Kmetal
      START(1,0)
      LAYER "Bottom Metal" VALUE(V)=V0
      LINE TO (4,0)
      LAYER "Bottom Metal" NATURAL(V)=0
      Line TO (4,4) TO (1,4) to close

   LIMITED REGION 3 { the smaller top plate}
      SURFACE "Top Metal - Dielectric" VALUE(V)=V1
      SURFACE "Top Metal - Air"        VALUE(V)=V1
      LAYER "Top Metal" K = Kmetal
      START(2,1)
      LINE TO (3,1) TO (3,5)
      LAYER "Top Metal" VALUE(V)=V1
      LINE TO (2,5)
      LAYER "Top Metal" NATURAL(V)=0
      LINE to close

MONITORS
  CONTOUR(V) ON Y=2.5

PLOTS
  GRID(X,Z)  ON Y=2.5
  CONTOUR(V) ON X=2.5 REPORT(C) as "Capacitance(uF)"
  CONTOUR(V) ON Y=2.5 REPORT(C) as "Capacitance(uF)"
  CONTOUR(V) ON Z=1.5 REPORT(C) as "Capacitance(uF)"
  CONTOUR(1/K) ON Y=2.5 as "Material"

END
```

6.1.3.2 3d_capacitor_check

```
{  3D_CAPACITOR_CHECK.PDE

  This problem shows a parallel-plate capacitor, and compares the computed
  capacitance to the ideal value.
}
```

```
TITLE '3D Capacitor validation'

COORDINATES
  CARTESIAN3

SELECT
  { rename the axes }
  alias(x) = "X(mm)"
  alias(y) = "Y(mm)"
  alias(z) = "Z(mm)"
  { paint all contours }
  PAINTED

VARIABLES
  V

DEFINITIONS
  Kmetal=1e6
  Kdiel = 88     { Water @ 0 C }
  Kair=1


  K = Kair       { default K to Kair - this will change in some layers/regions }
  V0 = 0
  V1 = 1

  X0 = 2     Xwid = 3    X1 = X0+Xwid    X2 = X1+X0      Xc = X2/2
  Y0 = 2     Ywid = 3    Y1 = Y0+Ywid    Y2 = Y1+Y0      Yc = Y2/2
  Z0 = 3     Zdist=0.1   Zthick=0.1      Zc = Z0+Zdist/2

  Eps0 = 8.854e-12                       { Farads/M }
  Eps0mm = 0.001*Eps0                    { Farads/mm }
  W = integral(0.5*K*eps0mm*grad(V)^2)   { Stored Energy }
  C = 1.0e6*2*W/(V1-V0)^2                { Capacitance in microFarads }
  C0 = 1.0e6*Kdiel*eps0mm*Xwid*Ywid/Zdist

EQUATIONS
  V : DIV(K*GRAD(V)) = 0

EXTRUSION
  SURFACE   "Bottom"                     Z=0
    LAYER   "Bottom Air"
  SURFACE   "Bottom Air - Metal"         Z=Z0-Zthick
    LAYER   "Bottom Metal"
  SURFACE   "Bottom Metal - Dielectric" Z=Z0
    LAYER   "Dielectric"
  SURFACE   "Top Metal - Dielectric"     Z=Z0+Zdist
    LAYER   "Top Metal"
  SURFACE   "Top Metal - Air"            Z=Z0+Zdist+Zthick
    LAYER   "Top Air"
  SURFACE   "Top"                       Z=Z0+Zthick+Zdist+Zthick+Z0

BOUNDARIES
  SURFACE "Bottom" natural(V)=0
  SURFACE "Top" natural(V)=0

  REGION 1  { this is the outer boundary of the system }
      START(0,0)
      LINE TO (X2,0)  TO (X2,Y2)  TO(0,Y2)  to close

  LIMITED REGION 2  { plates and dielectric }
      SURFACE "Bottom Air - Metal"         VALUE(V)=V0
      SURFACE "Bottom Metal - Dielectric" VALUE(V)=V0
      SURFACE "Top Metal - Dielectric"     VALUE(V)=V1
      SURFACE "Top Metal - Air"            VALUE(V)=V1
      LAYER "Bottom Metal" K = Kmetal
      LAYER "Dielectric" K = Kdiel
      LAYER "Top Metal" K = Kmetal
      START(X0,Y0)
        LAYER "Bottom Metal" VALUE(V)=V0
        LAYER "Top Metal" VALUE(V)=V1
      LINE TO (X1,Y0)  TO (X1,Y1) TO (X0,Y1) to close

MONITORS
  CONTOUR(V) ON Y=Yc
      REPORT(C) as "Capacitance(uF)"
      REPORT(C0) as "Cideal(uF)"
  CONTOUR(magnitude(grad(V))) ON Y=Yc as "Em"
      ZOOM(X0-Zthick,Z0-2*Zthick, 5*Zthick,5*Zthick)
```

```
PLOTS
  CONTOUR(V) ON X=Xc
      REPORT(C)  as "Capacitance(uF)"
      REPORT(C0) as "Cideal(uF)"
  CONTOUR(V) ON Y=Yc
      REPORT(C)  as "Capacitance(uF)"
      REPORT(C0) as "Cideal(uF)"
  CONTOUR(V) ON  Z=Zc
      REPORT(C)  as "Capacitance(uF)"
      REPORT(C0) as "Cideal(uF)"
  CONTOUR(V) ON Y=Yc
      ZOOM(X0-Zthick,Z0-2*Zthick, 5*Zthick,5*Zthick)
  GRID(X,Z) ON Y=Yc
  GRID(X,Y) ON Z=Zc
  CONTOUR(log10(K)) ON Y=Yc as "Material"

  SUMMARY
      REPORT(C)  as "Capacitance(uF)"
      REPORT(C0) as "Cideal(uF)"
      REPORT(W)  as "Stored Energy"

END
```

6.1.3.3 3d_dielectric

```
{  3D_DIELECTRIC.PDE

   This problem is a 3D extension of DIELECTRIC.PDE 300
}

title
  'Electrostatic Potential'

coordinates
  cartesian3

variables
  V

definitions
  eps = 1

equations
  div(eps*grad(V)) = 0  { Potential equation }

extrusion
  surface "bottom"  z=0
  surface "dielectric_bottom" z=0.1
 layer "dielectric"
  surface "dielectric_top"  z=0.2
  surface "top"  z=0.3

boundaries

  surface "bottom" natural(V)=0
  surface "top" natural(V)=0

  region 1
    start (0,0)
    value(V) = 0     line to (1,0)
    natural(V) = 0      line to (1,1)
    value(V) = 100      line to (0,1)
    natural(V) = 0      line to close

  region 2
    layer "dielectric"  eps = 50
      start (0.4,0.4)
      line to (0.8,0.4) to (0.8,0.8) to (0.6,0.8)
           to (0.6,0.6) to (0.4,0.6) to close

monitors
  contour(V) on z=0.15 as 'Potential'

plots
  contour(V) on z=0.15 as 'Potential'
  vector(-dx(V),-dy(V)) on z=0.15 as 'Electric Field'
```

```
  contour(V) on x=0.5 as 'Potential'

end
```

6.1.3.4 capacitance

```
{ CAPACITANCE.PDE

   See discussion in Help section "Electromagnetic Applications | Electrostatics" 218.
}

 TITLE 'Capacitance per Unit Length of 2D Geometry'
{ 17 Nov 2000 by John Trenholme }

SELECT
  errlim 1e-4
  thermal_colors on
  plotintegrate off

VARIABLES
  v

DEFINITIONS
  mm = 0.001                    ! meters per millimeter
  Lx = 300 * mm                    ! enclosing box dimensions
  Ly = 150 * mm
  b = 0.7                    ! radius of conductor / radius of entire cable
  x0 = 0.25 * Lx                    ! position and size of cable raised to fixed potential
  y0 = 0.5 * Ly
  r0 = 15 * mm
  x1 = 0.9 * Lx
  y1 = 0.3 * Ly
  r1 = r0
  epsr                           ! relative permittivity of any particular region
  epsd = 3                   ! relative permittivity of cable dielectric
  eps0 = 8.854e-12           ! permittivity of free space
  eps = epsr * eps0
  v0 = 1                   ! fixed potential of the cable

  energyDensity = dot( eps * grad( v), grad( v))/2          ! field energy density

EQUATIONS
  div( eps * grad( v)) = 0

BOUNDARIES
  region 1  'inside'  epsr = 1
    start  'outer'  ( 0, 0)  value( v) = 0
    line to ( Lx, 0) line to ( Lx, Ly) line to ( 0, Ly) line to close
  region 2  'diel0'  epsr = epsd
    start  'dieb0'  ( x0 + r0, y0)
    arc ( center = x0, y0) angle = 360
  region 3  'cond0'  epsr = 1
    start  'conb0'  ( x0 + b * r0, y0)  value( v) = v0
    arc ( center = x0, y0) angle = 360
  region 4  'diel1'  epsr = epsd
    start  'dieb1'  ( x1 + r1, y1)
    arc ( center = x1, y1) angle = 360
  region 5  'cond1'  epsr = 1000         ! fake metallic conductor
    start  'conb1'  ( x1 + b * r1, y1)
    arc ( center = x1, y1) angle = 360

PLOTS
  contour( v) as 'Potential'
  contour( v) as 'Potential Near Driven Conductor'
    zoom( x0 - 1.1 * r0, y0 - 1.1 * r0, 2.2 * r0, 2.2 * r0)
  contour( v) as 'Potential Near Floating Conductor'
    zoom( x1 - 1.1 * r1, y1 - 1.1 * r1, 2.2 * r1, 2.2 * r1)
  elevation( v) as 'Potential from Wall to Driven Conductor' from ( 0,y0) to ( x0, y0)
  elevation( v) as 'Potential from Driven to Floating Conductor' from ( x0, y0) to ( x1, y1)
  vector( grad( v)) as 'Field'
  contour( energyDensity) as 'Field Energy Density'  png(3072,2)
  contour( energyDensity) as 'Field Energy Density Near Floating Conductor'
    zoom( x1 - 1.2 * r1, y1 - 1.2 * r1, 2.4 * r1, 2.4 * r1)
  elevation( energyDensity) from ( x1 - 2 * r1, y1) to ( x1 + 2 * r1, y1)
    as 'Field Energy Density Near Floating Conductor'
  contour( epsr) paint on "inside" as 'Definition of Inside'
```

```
SUMMARY png(3072,2)
    report sintegral( normal( eps * grad( v)), 'conb0', 'diel0') as 'Driven charge'
    report sintegral( normal( eps * grad( v)), 'outer', 'inside') as 'Outer charge'
    report sintegral( normal( eps * grad( v)), 'conb1', 'diel1') as 'Floating charge'
    report sintegral( normal( eps * grad( v)), 'conb0', 'diel0') / v0 as 'Capacitance (f/m)'
    report integral( energyDensity, 'inside') as 'Energy (J/m)'
    report 2 * integral( energyDensity, 'inside') / v0^2 as 'Capacitance (f/m)'
    report 2 * integral( energyDensity, 'inside') / ( v0 * sintegral( normal( eps * grad( v)),
'conb0', 'diel0'))
      as 'cap_by_energy / cap_by_charge'

END
```

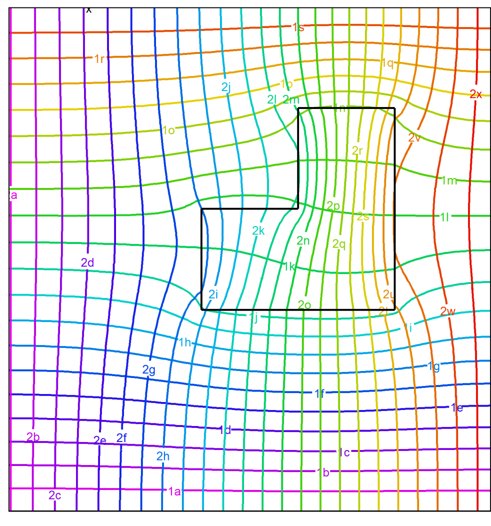
6.1.3.5 dielectric

```
{  DIELECTRIC.PDE

    This problem shows the electrostatic potential and the electric field
    in a rectangular domain with an internal region in which the dielectric
    constant is fifty times that of the surrounding material.
    The electric field E is -grad(V), where V is the electrostatic potential.

    See also FIELDMAP.PDE 300
}

title 'Electrostatic Potential'

variables V

definitions
  eps = 1

equations
  div(eps*grad(V)) = 0

boundaries
  region 1
    start (0,0)
    value(V) = 0    line to (1,0)
    natural(V) = 0  line to (1,1)
    value(V) = 100  line to (0,1)
    natural(V) = 0  line to close
  region 2
    eps = 50
    start (0.4,0.4)
    line to (0.8,0.4) to (0.8,0.8)
         to (0.6,0.8) to (0.6,0.6)
         to (0.4,0.6) to close

monitors
  contour(V) as 'Potential'

plots
  grid(x,y)
  contour(V) as 'Potential'
  vector(-dx(V),-dy(V)) as 'Electric Field'

end
```

6.1.3.6 fieldmap

```
{  FIELDMAP.PDE

    This example shows the use of the adjoint equation to display Electric field
    lines and to compare these to the vector plot of E.

    The problem shows the electrostatic potential and the electric field
    in a rectangular domain with an internal region in which the dielectric
    constant is five times that of the surrounding material.
    The electric field E is -grad(V), where V is the electrostatic potential.

    See also DIELECTRIC.PDE
}
```

```
title
  'Electrostatic Potential and Electric Field'

variables
  V
  Q

definitions
  eps = 1

equations
  { Potential equation }
  V:    div(eps*grad(V)) = 0
  { adjoint equation }
  Q:    div(grad(Q)/eps) = 0

boundaries
  region 1
    start (0,0)
      value(V) = 0
      natural(Q) = tangential(grad(V))
    line to (1,0)
      natural(V) = 0
      natural(Q) = tangential(grad(V))
    line to (1,1)
      value(V) = 100
      natural(Q) = tangential(grad(V))
    line to (0,1)
      natural(V) = 0
      natural(Q) = tangential(grad(V))
    line to close

  region 2
    eps = 5
    start (0.4,0.4)
    line to (0.8,0.4) to (0.8,0.8) to (0.6,0.8)
         to (0.6,0.6) to (0.4,0.6) to close

monitors
  contour(V) as 'Potential'
  contour(Q) as 'Field'

plots
  grid(x,y)
  contour(V) as 'Potential'
  contour(Q) as 'Field Lines'
  contour(V,Q) as 'Potential and Field Lines'
  vector(-dx(V),-dy(V)) as 'Electric Field'
  vector(-dx(V),-dy(V)) norm notips as 'Electric Field'

end
```



6.1.3.7 plate_capacitor

```
{  PLATE_CAPACITOR.PDE

  This problem computes the field around a plate capacitor.
  (adapted from "Fields of Physics on the PC" by Gunnar Backstrom)

}
```

```
title 'Plate capacitor'

variables
  u

definitions
  Lx=1      Ly=1
  delx=0.5  d=0.2
  ddy=0.2*d
  Ex=-dx(u)     Ey=-dy(u)
  Eabs=sqrt(Ex^2+Ey^2)
  eps0=8.854e-12
  eps
  DEx=eps*Ex    DEy=eps*Ey
  Dabs=sqrt(DEx^2+DEy^2)
  zero=1.e-15

equations
  u : div(-eps*grad(u)) = 0

boundaries
  Region 1
    eps=eps0
    start(-Lx,-Ly)
        Load(u)=0
    line to (Lx,-Ly) to (Lx,Ly) to (-LX,Ly) to close

    start(-delx/2,-d/2)
        value(u)=0
    line to (delx/2,-d/2) to (delx/2,-d/2-ddy) to (-delx/2,-d/2-ddy)
         to close

    start(-delx/2,d/2+ddy)
        value(u)=1
    line to (delx/2,d/2+ddy) to (delx/2,d/2) to(-delx/2,d/2)
         to close

  Region 2
    eps = 7.0*eps0
    start(-delx/2,-d/2)
    line to (delx/2,-d/2) to (delx/2,d/2) to(-delx/2,d/2)
         to close

monitors
   contour(u)

plots
   contour(u)
   surface(u)
   vector(dx(u),dy(u))

end
```

6.1.3.8 space_charge

```
{  SPACE_CHARGE.PDE

  This problem describes the electric field in an insulated cardioid-like
  chamber due to an electrode at the tip of the cardioid and a localized
  space charge near the center of the body.

  Adaptive grid refinement detects the space charge and refines
  the computation mesh to resolve its shape.

 }

title "Electrostatic Potential with probe and space charge"
```

```
select errlim = 1e-4

definitions
  bigr = 1
  smallr = 0.4
  x0 = sqrt(bigr^2/2)
  y0 = x0
  r = sqrt(x^2+y^2)
  { define the electrode center }
  xc = sqrt((bigr-smallr)^2/2)
  yc = xc
  { A space charge source at -xc }
  source = x/((x+xc)^2 + y^2 + 0.001)
  k=0.1

variables
  V

equations
  V : div(k*grad(V)) + source = 0

boundaries
  region 1
    start(xc,yc-smallr)
        natural(V) = 0      { -- insulated outer boundary }
        arc(center=xc,yc) to (x0,y0)
        arc(center=0,0) angle 270
        arc(center=xc,-yc) to (xc,smallr-yc)
        value(V)=1          { -- applied voltage = 1 on tip }
        arc(center=xc,0) angle -180 to close

plots
  grid(x,y)
  contour(V) as "Potential"
  contour(V) zoom(0.2,-0.2,0.4,0.4)
  surface(V) viewpoint (0,10,30)
  surface(V) zoom(-0.6,-0.2,0.4,0.4)
  surface(source) zoom(-0.6,-0.2,0.4,0.4)

end
```

6.1.4 fluids

6.1.4.1 1d_eulerian_shock

```
{  1D_EULERIAN_SHOCK.PDE

   Comparison with shock tube problem of G.A. Sod

   See 1D_LAGRANGIAN_SHOCK.PDE 304 for a Lagrangian model of the same problem.

   Ref: G.A. Sod, "A Survey of Several Finite Difference Methods for Systems of
   Nonlinear Hyperbolic Conservation Laws", J. Comp. Phys. 27, 1-31 (1978)

   See also Kershaw, Prasad and Shaw, "3D Unstructured ALE Hydrodynamics with the
   Upwind Discontinuous Finite Element Method", UCRL-JC-122104, Sept 1995.
}

TITLE "Sod's Shock Tube Problem - Eulerian"

COORDINATES
  cartesian1

SELECT
  ngrid=200     { increase the grid density }
  regrid=off    { disable the adaptive mesh refinement }
  errlim=1e-4   { lower the error limit }

VARIABLES
  rho(1)
  u(1)
  P(1)

DEFINITIONS
  len = 1
  gamma = 1.4
  smeardist = 0.001 { a damping term to kill unwanted oscillations }
```

```
  eps = sqrt(gamma)*smeardist   { ~ cspeed*dist }

INITIAL VALUES
  rho = 1.0 - 0.875*uramp(x-0.49, x-0.51)
  u = 0
  P = 1.0 - 0.9*uramp(x-0.49, x-0.51)

EQUATIONS
  rho:  dt(rho) + u*dx(rho) + rho*dx(u)  = eps*dxx(rho)
  u:    dt(u) + u*dx(u) + dx(P)/rho  = eps*dxx(u)
  P:    dt(P) + u*dx(P) + gamma*P*dx(u)  = eps*dxx(P)

BOUNDARIES
  REGION 1
    START(0)  point value(u)=0
    Line to (len) point value(u)=0

TIME 0 TO 0.375

MONITORS
  for cycle=5
    elevation(rho) from(0) to (len)
    elevation(u)   from(0) to (len)
    elevation(P)   from(0) to (len)
    history(rho) at (0.5)
    history(u)   at (0.48) (0.49) (0.5) (0.51) (0.52)
    history(p)   at (0.48) (0.49) (0.5) (0.51) (0.52)
    history(deltat)

PLOTS
  for t=0.143, 0.375
    elevation(rho) from(0) to (len)
    elevation(u)   from(0) to (len)
    elevation(P)   from(0) to (len)
    history(rho) at (0.48) (0.49) (0.5) (0.51) (0.52)
    history(u)   at (0.48) (0.49) (0.5) (0.51) (0.52)
    history(p)   at (0.48) (0.49) (0.5) (0.51) (0.52)

END
```

6.1.4.2 1d_lagrangian_shock

```
{  1D_LAGRANGIAN_SHOCK.PDE

   This example solves Sod's shock tube problem on a 1D moving mesh.
   Mesh nodes are given the local fluid velocity, so the model is fully Lagrangian.

   See 1D_EULERIAN_SHOCK.PDE 303 for an Eulerian model of the same problem.

   Ref: G.A. Sod, "A Survey of Several Finite Difference Methods for Systems of
   Nonlinear Hyperbolic Conservation Laws", J. Comp. Phys. 27, 1-31 (1978)

   See also Kershaw, Prasad and Shaw, "3D Unstructured ALE Hydrodynamics with the
   Upwind Discontinuous Finite Element Method", UCRL-JC-122104, Sept 1995.
}

TITLE "Sod's Shock Tube Problem - Lagrangian"

COORDINATES
  cartesian1

SELECT
  ngrid = 100   { increase the grid density }
  regrid = off  { disable the adaptive mesh refinement }
  errlim = 1e-4 { lower the error limit }

VARIABLES
  rho(1)
  u(1)
  P(1)
  xm=move(x)

DEFINITIONS
  len = 1
  gamma = 1.4
  smeardist = 0.001 { a damping term to kill unwanted oscillations }
  eps = sqrt(gamma)*smeardist  { ~ cspeed*dist }
```

```
  v = 0
  rho0 = 1.0 - 0.875*uramp(x-0.49, x-0.51)
  p0   = 1.0 - 0.9*uramp(x-0.49, x-0.51)

INITIAL VALUES
  rho = rho0
  u = 0
  P = p0

EULERIAN EQUATIONS
  { equations are stated as appropriate to the Eulerian (lab) frame.
    FlexPDE will convert to Lagrangian form for moving mesh }
  rho:  dt(rho) + u*dx(rho) + rho*dx(u)   = eps*dxx(rho)
  u:    dt(u) + u*dx(u) + dx(P)/rho   = eps*dxx(u)
  P:    dt(P) + u*dx(P) + gamma*P*dx(u)   = eps*dxx(P)
  xm:   dt(xm) = u

BOUNDARIES
  REGION 1
    START(0)        point value(u)=0    point value(xm)=0
    line to (len)   point value(u)=0    point value(xm)=len

TIME 0 TO 0.375

MONITORS
  for cycle=5
    elevation(rho) from(0) to (len) range (0,1)
    elevation(u)   from(0) to (len) range (0,1)
    elevation(P)   from(0) to (len) range (0,1)

PLOTS
  for t=0 by 0.02 to 0.143, 0.16 by 0.02 to 0.375
    elevation(rho) from(0) to (len) range (0,1)
    elevation(u)   from(0) to (len) range (0,1)
    elevation(P)   from(0) to (len) range (0,1)

END
```

6.1.4.3 2d_eulerian_shock

```
{  2D_EULERIAN_SHOCK.PDE

   Comparison with shock tube problem of G.A. Sod

   See 1D_EULERIAN_SHOCK.PDE 303 for a 1D model of the same problem.

   Ref: G.A. Sod, "A Survey of Several Finite Difference Methods for Systems of
   Nonlinear Hyperbolic Conservation Laws", J. Comp. Phys. 27, 1-31 (1978)

   See also Kershaw, Prasad and Shaw, "3D Unstructured ALE Hydrodynamics with the
   Upwind Discontinuous Finite Element Method", UCRL-JC-122104, Sept 1995.
}

TITLE "Sod's Shock Tube Problem - 2D Eulerian"

SELECT
  ngrid = 100    { increase the grid density }
  regrid = off   { disable the adaptive mesh refinement }
  errlim = 1e-4 { lower the error limit }

VARIABLES
  rho(1)
  u(1)
  P(1)

DEFINITIONS
  len = 1
  wid = 0.02
  gamma = 1.4
  eps = 0.001  {=4*(1/63)^2}

INITIAL VALUES
  rho = 1.0 - 0.875*uramp(x-0.49, x-0.51)
  u   = 0
  P   = 1.0 - 0.9*uramp(x-0.49, x-0.51)

EQUATIONS
  rho:  dt(rho)+u*dx(rho) = eps*div(grad(rho)) - rho*dx(u)
```

```
  u:    dt(u)+u*dx(u) = eps*div(grad(u)) - dx(P)/rho
  P:    dt(P)+u*dx(P) = eps*div(grad(P)) - gamma*P*dx(u)

BOUNDARIES
  REGION 1
    START(0,0)
    Line to (len,0)
    Value(u)=0 line to (len,wid)
    Natural(u)=0 line to (0,wid) to close

TIME 0 TO 0.375

MONITORS
  for cycle=5
    elevation(rho) from(0,wid/2) to (len,wid/2)
    elevation(u)   from(0,wid/2) to (len,wid/2)
    elevation(P)   from(0,wid/2) to (len,wid/2)
    history(rho) at (0.5,wid/2)
    history(u)   at (0.48,wid/2) (0.49,wid/2) (0.5,wid/2) (0.51,wid/2) (0.52,wid/2)
    history(p)   at (0.48,wid/2) (0.49,wid/2) (0.5,wid/2) (0.51,wid/2) (0.52,wid/2)
    history(deltat)

PLOTS
  for t=0.143, 0.375
    elevation(rho) from(0,wid/2) to (len,wid/2)
    elevation(u)   from(0,wid/2) to (len,wid/2)
    elevation(P)   from(0,wid/2) to (len,wid/2)
    history(rho) at (0.48,wid/2) (0.49,wid/2) (0.5,wid/2) (0.51,wid/2) (0.52,wid/2)
    history(u)   at (0.48,wid/2) (0.49,wid/2) (0.5,wid/2) (0.51,wid/2) (0.52,wid/2)
    history(p)   at (0.48,wid/2) (0.49,wid/2) (0.5,wid/2) (0.51,wid/2) (0.52,wid/2)

END
```

6.1.4.4 2d_piston_movingmesh

```
{  2D_PISTON_MOVINGMESH.PDE

  This problem models the flow of a perfect gas in a compressor cylinder.
  The initial gas pressure is chosen as 1e-4 Atm, to show interesting swirling.
  The boundaries of the domain are moved according to the oscillation of the piston,
  while the interior mesh is tensioned within the moving boundaries.
  This results in a mixed Lagrange/Eulerian model, in that the mesh is moving,
  but with different velocity than the fluid.
}

TITLE "Piston"

COORDINATES
  Ycylinder

SELECT
  regrid=off    { disable the adaptive mesh refinement }
  painted       { paint all contours }

DEFINITIONS
  my_ngrid = 30 { later applied to the NGRID control and smoother }
  stroke = 8    { cylinder stroke cm }
  rad = 4       { cylinder bore radius cm }
  zraise = 1        { raised height of piston center }
  rraise = 3*rad/4  { radius or raised piston center }
  gap = 2           { piston/head clearance }
  gamma = 1.4

  rho0 = 0.001
  P0 = 100      { initial pressure (dyne/cm2) = 1e-4 Atm }
  visc = 0.15   { kinematic viscosity, cm^2/sec }

  rpm = 1000        { compressor speed }
  period = 60/rpm   { seconds }
  vpeak = (pi*stroke/period)

  { velocity profile: }
  vprofile =vpeak*sin(2*pi*t/period)

  { the piston shape: }
  zpiston = if r<rraise then zraise else zraise*(rad-r)/(rad-rraise)

  { the time-dependent piston profile: }
```

```
  zprofile = zpiston+0.5*stroke*(1-cos(2*pi*t/period))
  ztop = stroke+gap+zraise  { maximum z postion }

VARIABLES
  rho(rho0/10)      { gas density }
  u(vpeak/10)       { radial velocity }
  v(vpeak/10)       { axial velocity }
  P(P0/10)          { pressure }
  vm(vpeak/10)      { mesh velocity }
  zm=move(z)        { mesh position }

DEFINITIONS
  { sound speed }
  cspeed = sqrt(gamma*P/rho)
  cspeed0 = sqrt(gamma*P0/rho0)

  { a smoothing coefficient: }
  smoother = cspeed0*(rad/my_ngrid)
  evisc = max(visc,smoother)

SELECT
  ngrid= my_ngrid

INITIAL VALUES
  rho = rho0
  u = 0
  v = 0
  P = P0

EULERIAN EQUATIONS
  { Eulerian gas equations: (FlexPDE will add motion terms) }
  rho: dt(rho) + dr(rho*u*r)/r + dz(rho*v) = smoother*div(grad(rho))
  u:   dt(u) + u*dr(u)+v*dz(u) + dr(P)/rho  = evisc*div(grad(u))-evisc*u/r^2
  v:   dt(v) + u*dr(v)+v*dz(v) + dz(P)/rho  = evisc*div(grad(v))
  P:   dt(P) + u*dr(P)+v*dz(P) + gamma*P*(dr(r*u)/r+dz(v)) = smoother*div(grad(P))
  vm:  dzz(vm)=0    { balance mesh velocities in z only }
  zm:  dt(zm)=vm    { node positions - move only in z }

BOUNDARIES
  { use a piston and compression chamber with beveled edge, to create a swirl }
  REGION 1
    START(0,zraise)
        value(u)=0 value(v)=vprofile value(vm)=vprofile dt(zm)=vprofile
    line to (rraise,zraise) to (rad,0)
        value(u)=0 nobc(v) nobc(vm) nobc(zm)
    line to (rad,stroke+gap)
        value(u)=0 value(v)=0 value(vm)=0 dt(zm)=0
    line to (rraise,ztop) to (0,ztop)
        value(u)=0 nobc(v) nobc(vm) nobc(zm)
    line to close

  { add a diagonal feature to help control cell shapes at upper corner }
  FEATURE start(rraise,zraise) line to (rad,stroke+gap)

TIME 0 TO 2*period by 1e-6

PLOTS
  for t=0 by period/120 to endtime
  { control the frame size and data scaling to create a useable movie
    ( the movie can be created by replaying the .PG5 file and selecting
      EXPORT MOVIE, or we could add PNG() commands here to create
      it directly) }
  grid(r,z) frame(0,0, rad,ztop)
  contour(rho) frame(0,0, rad,ztop) fixed range(0,0.01) contours=50  nominmax
  contour(u) frame(0,0, rad,ztop) fixed range(-500,500) contours=50
  contour(v) frame(0,0, rad,ztop) fixed range(-550,550) contours=50
  contour(P) frame(0,0, rad,ztop) fixed range(0,2300) contours=50  nominmax
  vector(u,v) frame(0,0, rad,ztop) fixed range(0,550)
  contour(cspeed) frame(0,0, rad,ztop) fixed range(0,600)
  contour(magnitude(u,v)/cspeed) frame(0,0, rad,ztop) fixed range(0,1.1)

  history(vprofile/vpeak,zprofile/stroke) range(-1,1)
        report(vpeak) report(stroke)
  history(globalmax(P), globalmin(P))
  history(integral(P))
  history(globalmax(rho), globalmin(rho))
  history(integral(rho))
  history(deltat)
```

```
END
```

6.1.4.5 3d_flowbox

```
{  3D_FLOWBOX.PDE

  This problem demonstrates the use of FlexPDE in 3D fluid flow.  It shows the flow of
  fluid through a plenum box with a circular inlet at the bottom and an offset circular
  outlet at the top. The inlet pressure is arbitrarily set at 0.05 units.

  The problem runs in two stages, first as a massless fluid to get an initial pressure and
  velocity distribution in a linear system, and then with momentum terms included.

  Adaptive mesh refinement is turned off for speed in demonstration.  In a real application,
  regridding could be used to better resolve the flow past the corners of the ducts.

  The solution uses a "penalty pressure", in which the pressure variable is used merely
  to guarantee mass conservation.

}

title '3D flow through a plenum'

coordinates
  cartesian3

variables
  vx(1e-6) vy(1e-6) vz(1e-6) p

select
  ngrid=20
  stages=2
  regrid=off

definitions
  long = 2
  wide = 1
  high = 1/2
  xin = -1   yin = 0
  xout = 1   yout = 0
  rc = 0.5
  duct = 0.2

  dens=staged(0,1)  { fluid density }
  visc= 0.01        { fluid viscosity }
  v=vector(vx,vy,vz)
  vm=magnitude(v)

  div_v = dx(vx) + dy(vy) + dz(vz)

  PENALTY = 1e4*visc/high^2

  Pin = 0.05
  Pout = 0

initial values
  vx=0
  vy=0
  vz=0
 p=Pin+(Pout-Pin)*(z+high+duct)/(2*high+2*duct)

equations
  vx:   dens*(vx*dx(vx) + vy*dy(vx) + vz*dz(vx)) + dx(p) -visc*div(grad(vx)) = 0
  vy:   dens*(vx*dx(vy) + vy*dy(vy) + vz*dz(vy)) + dy(p) -visc*div(grad(vy)) = 0
  vz:   dens*(vx*dx(vz) + vy*dy(vz) + vz*dz(vz)) + dz(p) -visc*div(grad(vz)) = 0
  p:    div(grad(p)) = PENALTY*div_v

extrusion z = -high-duct,-high,high,high+duct

boundaries

    Region 1    { plenum box }
      surface 2 value(vx)=0  value(vy)=0  value(vz)=0 natural(p)=0
      surface 3 value(vx)=0  value(vy)=0  value(vz)=0 natural(p)=0
      layer 1 void
      layer 3 void
      start(-long,-wide)
           value(vx)=0  value(vy)=0  value(vz)=0 natural(p)=0  { fix all side values }
```

```
        line to (long,-wide)
          to (long,wide)
          to (-long,wide)
          to close

    limited Region 2    { input hole }
      layer 1
      { input duct opening: }
      surface 1 natural(vx)=0  natural(vy)=0  natural(vz)=0 value(p)=Pin
      start(xin,yin-rc)
          { duct sidewall drag: }
          layer 1   value(vx)=0  value(vy)=0  value(vz)=0 natural(p)=0
        arc(center=xin,yin) angle=360

    limited Region 3    { exit hole }
      layer 3
      { output duct opening: }
      surface 4 natural(vx)=0  natural(vy)=0  natural(vz)=0 value(p)=Pout
      start(xout,yout-rc)
          { duct sidewall drag: }
          layer 3   value(vx)=0  value(vy)=0  value(vz)=0 natural(p)=0
        arc(center=xout,yout) angle=360


monitors
    contour(vx)  on x=0 report dens report pin
    contour(vx)  on y=0 report dens report pin
    contour(vz)  on y=0 report dens report pin
    vector(vx,vz)on y=0 report dens report pin
    contour(vx)  on z=0 report dens report pin
    contour(vy)  on z=0 report dens report pin
    contour(vz)  on z=0 report dens report pin
    vector(vx,vy)on z=0 report dens report pin
    contour(p)   on y=0 report dens report pin

plots
    contour(vx)  on x=0 report dens report pin
    contour(vx)  on y=0 report dens report pin
    contour(vz)  on y=0 report dens report pin
    vector(vx,vz)on y=0 report dens report pin
    contour(vx)  on z=0 report dens report pin
    contour(vy)  on z=0 report dens report pin
    contour(vz)  on z=0 report dens report pin
    vector(vx,vy)on z=0 report dens report pin
    contour(p)   on y=0 report dens report pin

end
```

6.1.4.6 3d_vector_flowbox

```
{  3D_VECTOR_FLOWBOX.PDE

  This is a modification of the example 3D_FLOWBOX.PDE 308 to use vector variables.

  This problem demonstrates the use of FlexPDE in 3D fluid flow.  It shows the flow of
  fluid through a plenum box with a circular inlet at the bottom and an offset circular
  outlet at the top. The inlet pressure is arbitrarily set at 0.05 units.

  The problem runs in two stages, first as a massless fluid to get an initial pressure
  and velocity distribution in a linear system, and then with momentum terms included.

  Adaptive mesh refinement is turned off for speed in demonstration.  In a real application,
  regridding could be used to better resolve the flow past the corners of the ducts.

  The solution uses a "penalty pressure", in which the pressure variable is used merely
  to guarantee mass conservation.

}

title '3D flow through a plenum'

coordinates
  cartesian3

variables
  v(1e-6) = vector(vx,vy,vz)
  p
```

```
select
  ngrid=20
  stages=2
  regrid=off

definitions
  long = 2
  wide = 1
  high = 1/2
  xin = -1   yin = 0
  xout = 1   yout = 0
  rc = 0.5
  duct = 0.2

  dens=staged(0,1)  { fluid density }
  visc= 0.01        { fluid viscosity }

  vm=magnitude(v)

  div_v = dx(vx) + dy(vy) + dz(vz)

  PENALTY = 1e4*visc/high^2

  Pin = 0.05
  Pout = 0

INITIAL VALUES
  v = vector(0,0,0)
  p=Pin+(Pout-Pin)*(z+high+duct)/(2*high+2*duct)

EQUATIONS
  v:    dens*dot(v,grad(v)) + grad(p) - visc*div(grad(v)) = 0
  p:    div(grad(p)) = PENALTY*div_v

extrusion z = -high-duct,-high,high,high+duct

boundaries

    Region 1    { plenum box }
       surface 2 value(v) = vector(0,0,0) natural(p)=0
       surface 3 value(v) = vector(0,0,0) natural(p)=0
       layer 1 void
       layer 3 void
       start(-long,-wide)
            value(v) = vector(0,0,0) natural(p)=0   { fix all side values }
         line to (long,-wide)
           to (long,wide)
           to (-long,wide)
           to close

    limited Region 2    { input hole }
      layer 1
      surface 1 natural(v) = vector(0,0,0) value(p)=Pin     { input duct opening }
      start(xin,yin-rc)
            layer 1   value(v) = vector(0,0,0) natural(p)=0 { duct sidewall drag }
        arc(center=xin,yin) angle=360

    limited Region 3    { exit hole }
      layer 3
      surface 4 natural(v) = vector(0,0,0) value(p)=Pout    { output duct opening }
      start(xout,yout-rc)
            layer 3   value(v) = vector(0,0,0) natural(p)=0 { duct sidewall drag }
        arc(center=xout,yout) angle=360


monitors
    contour(vx)  on x=0 report dens report pin
    contour(vx)  on y=0 report dens report pin
    contour(vz)  on y=0 report dens report pin
    vector(vx,vz)on y=0 report dens report pin
    contour(vx)  on z=0 report dens report pin
    contour(vy)  on z=0 report dens report pin
    contour(vz)  on z=0 report dens report pin
    vector(vx,vy)on z=0 report dens report pin
    contour(p)   on y=0 report dens report pin

plots
    contour(vx)  on x=0 report dens report pin
    contour(vx)  on y=0 report dens report pin
```

```
    contour(vz)  on y=0 report dens report pin
    vector(vx,vz)on y=0 report dens report pin
    contour(vx)  on z=0 report dens report pin
    contour(vy)  on z=0 report dens report pin
    contour(vz)  on z=0 report dens report pin
    vector(vx,vy)on z=0 report dens report pin
    contour(p)   on y=0 report dens report pin

end
```

6.1.4.7 airfoil

```
{  AIRFOIL.PDE

  This example considers the laminar flow of an incompressible, inviscid
  fluid past an obstruction.

  We assume that the flow can be represented by a stream function, PSI,
  such that the velocities, U in the x-direction and V in the y-direction,
  are given by:

        U = -dy(PSI)
        V = dx(PSI)

  The flow can then be described by the equation

        div(grad(PSI)) = 0.

  The contours of PSI describe the flow trajectories of the fluid.

  The problem presented here describes the flow past an airfoil-like figure.
  The left and right boundaries are held at PSI=y, so that U=-1, and V=0.

}

title "Stream Function Flow past an Airfoil"

variables
   { define PSI as the system variable }
   psi

definitions
   far = 5      { size of solution domain }
   psi_far = y  { solution at large x,y }

equations   { the equation of continuity: }
   psi : div(grad(psi)) = 0

boundaries
   region 1 { define the domain boundary }
      start(-far,-far)  { start at the lower left }
        { impose -dy(psi)=U=-1 (outward normal of psi) on the bottom boundary }
        natural(psi)= -1
      line to (far,-far)    { walk the boundary Counter-Clockwise }
        natural(psi)=0      { impose dx(psi)=0 on right }
      line  to (far,far)
        natural(psi)=1      { impose dy(psi)=-U=1 on top }
      line  to (-far,far)
        natural(psi)=0      { impose -dx(psi)=0 on left }
      line to close         { return to close }

      start(-0.5,-0.05) { start at lower left corner of airfoil }
        value(psi)=0    { specify no flow through the airfoil surface }
      arc to (0.0,0.02) to (0.5,0.05)       { specify a gentle arc by three points }
      arc (center=0.495,0.1) to (0.5,0.15)  { a tight arc by two points and center }
      arc to (0.075,0.105) to (-0.35,0)     { the top arc by three points }
      line to close     { finally a straight line to close the figure }

monitors{ monitor progress while running }
   contour(psi) zoom (-0.6,-0.4,1.4,1.2) as "stream lines"

plots   { write hardcopy files at termination }
   grid(x,y) zoom (-0.6,-0.4,1.4,1.2)
   contour(psi) zoom (-0.6,-0.4,1.4,1.2) as "stream lines"  painted
   { show the flow vectors: }
   vector(-dy(psi),dx(psi)) zoom (-0.6,-0.4,1.4,1.2)as "flow" norm
   surface(psi) zoom (-0.6,-0.4,1.4,1.2) as "stream lines"
```

```
end
```

6.1.4.8 black_oil

```
{  BLACK_OIL.PDE

  This example considers the transport of oil and water in soil.

  The model is given in Gelinas, et al, "Adaptive Forward-Inverse Modeling
  of Reservoir Fluids Away from Wellbores",  (Lawrence Livermore National
  Laboratory report UCRL-ID-126377) and in Saad & Zhang, " Adaptive Mesh for
  Two-Phase Flow in Porous Media" (in Recent Advances in Problems of Flow and
  Transport in Porous Media, Crolet and El Hatri, eds., Kluwer Academic Publishers,
  Boston, 1998).

  The saturation of water is represented by S, with the saturation of oil defined
  as 1-S.  The relative permeabilities of water and oil are assumed to be S^2 and
  (1-S)^2, respecitvely.  The total mobility M is defined as
        M = S^2/muw + (1-S)^2/muo,
  where muw and muo are the viscosities of water and oil.

  The total velocity, V, and the fractional flux, f, are defined as
        V = - K*M grad(P)
        f = [S^2/muw]/M,
  where K represents the saturation-independent permeability coefficient, and
  P is the pressure, assuming capillary to be zero and oil and water pressures
  equal.

  If the porosity Phi is taken as constant and gravity effects are negligible, the
  PDE's governing the system reduce to
        Phi*dt(S) + div(V*f) = 0
        div(V) = 0.

  Here we study the flow through a 30-meter box with an inlet pipe in the upper
  left and   an outlet pipe in the lower right.  The box is initially filled with oil,
  and water is pumped into the inlet pipe at a constant pressure.  Time is measured
  in seconds.

         --  Submitted by Said Doss, Lawrence Livermore National Laboratory.

 }

TITLE 'Black Oil Model'

SELECT
      smoothinit    { Smooth the initial conditions a little, to minimize
                        the time wasted tracking the initial discontinuity }

VARIABLES
      s, p          { Saturation and Pressure }

DEFINITIONS
      muo = 4.e-3                 { oil viscosity }
      muw = 1.e-3                 { water viscosity }
      K = 1.e-12                  { Saturation-independent permeability coefficient }
      Pin = 1.5e6                 { Inlet pressure }
      Pout = 1.e6                 { Outlet pressure }
      M = S^2/muw + (1-S)^2/muo { Total mobility }
      f = S^2/muw/M               { Fractional flux }
      krw = S^2/muw               { Relative permeability of water }
      phi =.206                   { porosity }

      xmax = 30                   { Box dimensions }
      ymax = xmax
      out_ctr = 8
      tfrac = 2*out_ctr
      diam = 2
      in_ctr = ymax-out_ctr
      rad = diam/5

      epsvisc = 1.e-6       { A little artificial diffusion helps smooth the solution }

      sint = integral(s)    { the total extraction integral }

      hour = 60*60
      day = hour*24         { seconds per day }

 INITIAL VALUES
```

```
      s = 0                         { start with all oil }
      p = Pin + (Pout-Pin)*x/xmax   { start with a rough approximation to the pressure }

EQUATIONS
      s:  phi*dt(s) - div(K*krw*grad(p)) - epsvisc*div(grad(s)) = 0
      p:  div(K*M*grad(p)) = 0

BOUNDARIES
   REGION 1
     { fillet the input pipe, and define
       no-flow boundaries of the box }
     start(-2*rad,in_ctr-diam)
       natural(p)=0 natural(s) = 0
       line to (0,in_ctr-diam) fillet(rad)
       line to (0,0) to (xmax,0)
            to (xmax,out_ctr-diam) fillet(rad)
       line to (xmax+2*rad,out_ctr-diam)

       { set constant outlet pressure, and
         "tautological" saturation flux }
       value(p) = Pout
       natural(s) =  -K*krw*dx(p)
       line to (xmax+2*rad,out_ctr+diam)

       { reset no-flow box boundaries }
       natural(p)=0  natural(s)=0
       line to (xmax,out_ctr+diam) fillet(rad)
       line to (xmax,ymax) to (0,ymax)
            to (0,in_ctr+diam) fillet(rad)
       line to (-2*rad,in_ctr+diam)

       { set constant inlet pressure and saturation }
       value(p) = Pin   value(s) = 1
       line to close

TIME   0 to 120*day  by 10

MONITORS
    for cycle=5
      contour(s) as "Saturation" range(0,1)
      contour(s) zoom(xmax-tfrac+2*rad,0, tfrac,tfrac) as "Outflow Saturation"
          range(0,1)
      contour(p) as "Pressure"
      vector(-K*M*grad(p)) norm as "Flow Velocity"

PLOTS
   for t = day by day  to 20*day
               by 10*day  to 120*day
      grid(x,y)
      contour(s) as "Saturation" range(0,1) painted
      surface(s) as "Saturation" range(0,1) painted  viewpoint(60,-120,30)
      contour(s) zoom(xmax-tfrac+2*rad,0,  tfrac,tfrac) as "Outflow Saturation"
          range(0,1)  painted
      contour(p) as "Pressure" painted
      vector(-K*M*grad(p)) norm as "Flow Velocity"
      contour(K*M*magnitude(grad(p))) norm as "Flow Speed" painted

HISTORIES
      history(sint) at (0,0) as "Extraction"

END
```

6.1.4.9 buoyant+time

```
{  BUOYANT+TIME.PDE

  This example is the time-dependent form of the steady-state example
  BUOYANT.PDE 315.

  Here we gradually ramp up the heat input to the level given in the
  steady-state problem.

  At early times, a single convection cell is established, but at later
  times the bottom of the bowl stagnates and establishes the two-cell
  flow pattern seen in the steady problem.

}
```

```
TITLE 'Buoyant Flow by Stream Function and Vorticity - no slip'

VARIABLES
   temp(100)
   psi(0.001)
   w(1)

DEFINITIONS
   Lx = 1   Ly = 0.5
   Rad = 0.5*(Lx^2+Ly^2)/Ly
   Gy = 980

   sigma_top = 0.01     { surface heat loss coefficient }
   sigma_bowl =  1      { bowl heat loss coefficient }
   k =  0.0004          { thermal conductivity }

   alpha = 0.001        { thermal expansion coefficient }
   visc = 1

   heatin = min(10,t)
   t0 = 50

   rho0 = 1
   rho = rho0*(1 - alpha*temp)
   cp = 1

   u = dy(psi)
   v = -dx(psi)

   penalty = 5000

EQUATIONS
   temp: div(k*grad(temp)) = rho0*cp*(dt(temp) + u*dx(temp) + v*dy(temp))
   psi:  div(grad(psi)) + w = 0
   w:    dt(w) + u*dx(w) + v*dy(w) = visc*div(grad(w)) - Gy*dx(rho)

BOUNDARIES
   region 1

    { on the arc of the bowl, set Psi=0, apply conduction loss to T,
        and apply penalty function to w to enforce no-slip condition. }
    start(0,0)
      natural(temp) = -sigma_bowl*temp
      value(psi) = 0
      natural(w)=penalty*tangential(u,v)
      arc (center=0,Rad) to (Lx,Ly)

      { on the top, continue the prior BC for Psi,
        but apply a heat input and loss to T.
        Apply natural=0 BC (no vorticity transport) for w }
      load(temp) = heatin*exp(-(10*x/Lx)^2) - sigma_top*temp
      natural(w)=0
      line to (0,Ly)

      { in the symmetry plane assert w=0, with a reflective BC for T }
      value(w)=0
      load(temp) = 0
      line to close

TIME 0 to 100

MONITORS
   for cycle=5  { watch what's happening }
   contour(temp) as "Temperature"
   contour(psi)  as "Stream Function"
   contour(w)    as "Vorticity"
   vector(curl(psi)) as "Flow Velocity" norm

PLOTS
   for t = 1 by 1 to 10 by 10 to endtime
   grid(x,y)
   contour(temp) as "Temperature"  painted
   contour(psi)  as "Stream Function"
   contour(w)    as "Vorticity"  painted
   vector(curl(psi)) as "Flow Velocity" norm
   contour(rho)  as "Density"  painted

HISTORIES
  history(temp) at (0.1*Lx,Ly) (0.2*Lx,Ly) (0.5*Lx,Ly) (0.8*Lx,Ly)
```

```
        (0.7*Lx,0.5*Ly) (0.04*Lx,0.1*ly) as "Temperature"
  history(u) at (0.1*Lx,Ly) (0.2*Lx,Ly) (0.5*Lx,Ly) (0.8*Lx,Ly)
        (0.7*Lx,0.5*Ly) (0.04*Lx,0.2*Ly) as "X-velocity"
  history(v) at  (0.04*Lx,0.1*ly) as "Y-velocity"


END
```

6.1.4.10 buoyant

```
{  BUOYANT.PDE

  This example addresses the problem of thermally driven buoyant flow
  of a viscous liquid in a vessel in two dimensions.

  In the Boussinesq approximation, we assume that the fluid is incompressible,
  except for thermal expansion effects which generate a buoyant force.

  The incompressible form of the Navier-Stokes equations for the flow of a fluid
  can be written
        dt(U) + U.grad(U) + grad(p) = nu*div(grad(U)) + F
        div(U) = 0
  where U represents the velocity vector,
        p is the pressure,
        nu is the kinematic viscosity
  and   F is the vector of body forces.

  The first equation expresses the conservation of momentum, while the second,
  or Continuity Equation, expresses the conservation of mass.
  If the flow is steady, we may drop the time derivative.

  If we take the curl of the (steady-state) momentum equation, we get
        curl(U.grad(U)) + curl(grad(p)) = nu*curl(div(grad(U)) + curl(F)

  Using div(U)=0 and div(curl(U))=0, and defining the vorticity W = curl(U),
  we get
        U.grad(W) = W.grad(U) + nu*div(grad(W)) + curl(F)

  W.grad(U) represents the effect of vortex stretching, and is zero in
  two-dimensional systems. Furthermore, in two dimensions the velocity has
  only two components, say u and v, and the vorticity has only one,
  which we shall write as w.

  Consider now the continuity equation.  If we define a scalar function psi
  such that
        u = dy(psi)    v = -dx(psi)
  then div(U) = dx(dy(psi))-dy(dx(psi)) = 0, and the continuity equation is
  satisfied exactly.  We may write
        div(grad(psi)) = -dx(v)+dy(u) = -w

  Using psi and w, we may write the final version of the Navier-Stokes
  equations as
        dy(psi)*dx(w) -dx(psi)*dy(w) = nu*div(grad(w)) + curl(F)
        div(grad(psi)) + w = 0

  If F is a gravitational force, then
        F = (0,-g*rho) and
        curl(F) = -g*dx(rho)
  where rho is the fluid density and g is the acceleration of gravity.

  The temperature of the system may be found from the heat equation
        rho*cp*[dt(T)+U.grad(T)] = div(k*grad(T)) + S

  Dropping the time derivative, approximating rho by rho0,
  and expanding U in terms of psi, we get
        div(k*grad(T)) + S = rho0*cp*[dy(psi)*dx(temp) - dx(psi)*dy(temp)]

  If we assume linear expansion of the fluid with temperature, then
        rho = rho0*(1+alpha*(T-T0)) and
        curl(F) = -g*rho0*alpha*dx(T)

  --------------------------

  In this problem, we define a trough filled with liquid, heated along a center
  strip by  an applied heat flux, and watch the convection pattern and the heat
  distribution.  We compute only half the trough, with a symmetry plane
  in the center.
```

```
  Along the symmetry plane, we assert w=0, since on this plane
  dx(v) = 0 and u=0, so dy(u) = 0.

  Applying the boundary condition psi=0 forces the stream lines to be parallel
  to the boundary, enforcing no flow through the boundary.

  On the surface of the bowl, we apply a penalty function to enforce a "no-slip"
  boundary condition.  We do this by using a natural BC to introduce a surface
  source of vorticity to counteract the tangential velocity. The penalty weight
  was arrived at by trial and error.  Larger weights can force the surface
  velocity closer to zero, but this has no perceptible effect on the temperature
  distribution.

  On the free surface, the proper boundary condition for the vorticity is
  problematic. We choose to apply NATURAL(w)=0, because this implies no vorticity
  transport across the free surface.    (11/16/99)

 }

TITLE 'Buoyant Flow by Stream Function and Vorticity - No Slip'

VARIABLES
   temp psi w

DEFINITIONS
   Lx = 1   Ly = 0.5
   Rad = 0.5*(Lx^2+Ly^2)/Ly
   Gy = 980

   { surface heat loss coefficient }
   sigma_top = 0.01
   { bowl heat loss coefficient }
   sigma_bowl = 1
   { thermal conductivity }
   k = 0.0004
   { thermal expansion coefficient }
   alpha = 0.001
   visc = 1
   rho0 = 1
   heatin = 10  { heat source }
   t0 = 50

   dens = rho0*(1 - alpha*temp)
   cp = 1

   penalty = 5000

   u = dy(psi)
   v = -dx(psi)

EQUATIONS

   temp: div(k*grad(temp)) = rho0*cp*(u*dx(temp) + v*dy(temp))
   psi:  div(grad(psi)) + w = 0
   w:    u*dx(w) + v*dy(w) = visc*div(grad(w)) - Gy*dx(dens)

BOUNDARIES
   region 1

    { on the arc of the bowl, set Psi=0, and apply a conductive loss to T.
        Apply a penalty function to w to force the tangential velocity to zero }
    start "outer" (0,0)
      natural(temp) = -sigma_bowl*temp
      value(psi) = 0
      natural(w)= penalty*tangential(curl(psi))
      arc (center=0,Rad) to (Lx,Ly)

      { on the top, continue the Psi=0 BC, but add the heat in put term to T,
        and apply a natural=0 BC for w }
      natural(w)=0
      load(temp) = heatin*exp(-(10*x/Lx)^2) - sigma_top*temp
      line to (0,Ly)

      { in the symmetry plane assert w=0, with a reflective BC for T }
      value(w)=0
      load(temp) = 0
      line to close

MONITORS
```

```
   contour(temp) as "Temperature"
   contour(psi)  as "Stream Function"
   contour(w)    as "Vorticity"
   vector(u,v)   as "Flow Velocity" norm

PLOTS
   grid(x,y)
   contour(temp) as "Temperature"  painted
   contour(psi)  as "Stream Function"
   contour(w)    as "Vorticity"  painted
   vector(u,v)   as "Flow Velocity" norm
   contour(dens) as "Density"  painted
   contour(magnitude(u,v)) as "Speed"  painted
   elevation(magnitude(u,v)) on "outer"
   elevation(temp) on "outer"

END
```

6.1.4.11 channel

```
{  CHANNEL.PDE

  This example is a modification of the LOWVISC.PDE 324 problem, in which the
  no-slip boundary has been placed at the bottom of the domain, with free flow
  at the top.

  The declared parameters in this problem are chosen for demonstration purposes,
  and are not intended to represent any real conditions.  The fluid is far more
  viscous than water.

}

title 'Flow in 2D channel'

select errlim = 0.005

variables
   u(0.1)
   v(0.01)
   p(1)

definitions
   Lx = 5        Ly = 1.5
   p0 = 1        { input pressure }
   speed2 = u^2+v^2
   speed = sqrt(speed2)
   dens = 1
   visc = 0.04
   vxx = (p0/(2*visc*(2*Lx)))*y^2   { open-channel x-velocity with drag at the bottom }

   rball=0.4
   cut = 0.1    { value for bevel at the corners of the obstruction }

   penalty = 100*visc/rball^2
   Re = globalmax(speed)*(Ly/2)/(visc/dens)

initial values
  { In nonlinear problems, Newton's method requires a good initial guess at the solution,
    or convergence may not be achieved.  You can use SELECT CHANGELIM=0.1 to
    force the solver to creep toward a solution from a bad guess.
    In our problem, the open channel velocity is a good place to start. }
   u = vxx  v = 0  p = p0*x/(2*Lx)

equations
   u:  visc*div(grad(u)) - dx(p) = dens*(u*dx(u) + v*dy(u))
   v:  visc*div(grad(v)) - dy(p)  = dens*(u*dx(v) + v*dy(v))
   p:  div(grad(p)) = penalty*(dx(u)+dy(v))

boundaries
   region 1
      start(-Lx,0)
      value(u) = 0   value(v) = 0   load(p) = 0
        line to (Lx/2-rball,0)
             to (Lx/2-rball,rball) bevel(cut)
             to (Lx/2+rball,rball) bevel(cut)
             to (Lx/2+rball,0)
             to (Lx,0)
```

```
        load(u) = 0  value(v) = 0  value(p) = p0
        mesh_spacing=Ly/20
          line to (Lx,Ly)

        mesh_spacing=100
        load(p) = 0
          line to (-Lx,Ly)

        value(p) = 0
          line to close

monitors
   contour(speed)  report(Re)

plots
   contour(u)  report(Re)
   contour(v)  report(Re)
   contour(speed) painted  report(Re)
   vector(u,v) as "flow"   report(Re)
   contour(p)  as "Pressure" painted
   contour(dx(u)+dy(v)) as "Continuity Error"
   elevation(u) from (-Lx,0) to (-Lx,Ly)
   elevation(u) from (0,0)   to (0,Ly)
   elevation(u) from (Lx/2,0)to (Lx/2,Ly)
   elevation(u) from (Lx,0)  to (Lx,Ly)

end
```

6.1.4.12 contaminant_transport

```
{ CONTAMINANT_TRANSPORT.PDE

  This example shows the use of sequenced equations in the calculation of steady-state
  contaminant transport  in which the fluid properties are independent of the contaminant
  concentration.

  Fluid equations are solved first on each grid refinement, then the contaminant
  concentration is updated.

  The problem is a modification of the example CHANNEL.PDE 317 .

}

title 'Contaminant transport in 2D channel'

select errlim = 0.005

variables
   u(0.1)
   v(0.01)
   p(1)
   c(0.01)

definitions
   Lx = 5       Ly = 1.5
   p0 = 2
   speed2 = u^2+v^2
   speed = sqrt(speed2)
   dens = 1
   visc = 0.04
   vxx = (p0/(2*visc*(2*Lx)))*y^2   { open-channel x-velocity }

   rball=0.4
   cut = 0.1    { value for bevel at the corners of the obstruction }

   penalty = 100*visc/rball^2
   Re = globalmax(speed)*(Ly/2)/(visc/dens)

   Kc = 0.01    { contaminant diffusivity }

initial values
   u =  vxx    v=0   p = p0*x/Lx

equations
   u:  visc*div(grad(u)) - dx(p) = dens*(u*dx(u) + v*dy(u))
   v:  visc*div(grad(v)) - dy(p) = dens*(u*dx(v) + v*dy(v))
   p:  div(grad(p)) = penalty*(dx(u)+dy(v))
then
```

```
   c:  u*dx(c) + v*dy(c) = div(Kc*grad(c))

boundaries
   region 1
      start(-Lx,0)
      value(u) = 0   value(v) = 0   load(p) = 0  natural(c)=0
        line to (Lx/2-rball,0)
             to (Lx/2-rball,rball) bevel(cut)
             to (Lx/2+rball,rball) bevel(cut)
             to (Lx/2+rball,0)
             to (Lx,0)

      mesh_spacing=Ly/20
      load(u) = 0  value(v) = 0  value(p) = p0  value(c) = Upulse(y,y-Ly/3)
        line to (Lx,Ly)

      mesh_spacing = 100
      load(p) = 0 natural(c)=0
        line to (-Lx,Ly)

      value(p) = 0
        line to close

monitors
     contour(speed)
     contour(c)

plots
   contour(c)  report(Re)
   contour(u)  report(Re)
   contour(v)  report(Re)
   contour(speed) painted  report(Re)
   vector(u,v) as "flow"   report(Re)
   contour(p)  as "Pressure" painted
   contour(dx(u)+dy(v)) as "Continuity Error"
   elevation(u) from (-Lx,0) to (-Lx,Ly)
   elevation(u) from (0,0)   to (0,Ly)
   elevation(u) from (Lx/2,0)to (Lx/2,Ly)
   elevation(u) from (Lx,0)  to (Lx,Ly)

end
```

6.1.4.13 coupled_contaminant

```
{  COUPLED_CONTAMINANT.PDE

  This example shows the use of FlexPDE in a contaminant transport calculation in
  which the fluid viscosity is strongly dependent on the contaminant concentration.

  The example LOWVISC_FULL.PDE  must first be solved to establish flow velocities.

  This time-dependent modification of that example then reads the initial values and
  computes the flow of a contaminant in the channel.

  Fluid equations are solved fully implicitly with the contaminant concentration.

}

title 'Contaminant transport in 2D channel, Re > 40'

variables
   u(0.1)
   v(0.01)
   p(1)
   c(0.01)

definitions
   Lx = 5       Ly = 1.5
   p0 = 2
   speed2 = u^2+v^2
   speed = sqrt(speed2)
   dens = 1
   visc = 0.04*(1+c)
   vxx = (p0/(2*visc*(2*Lx)))*(Ly-y)^2       { open-channel x-velocity }

   rball=0.4
   cut = 0.1        { value for bevel at the corners of the obstruction }
```

```
   penalty = 100*visc/rball^2
   Re = globalmax(speed)*(Ly/2)/(visc/dens)

   { program a contaminant pulse in space and time
        use SWAGE to eliminate discontinuous changes }
   swagepulse(f,a,b) = swage(f-a,0,1,0.1*(b-a))*swage(f-b,1,0,0.1*(b-a))
   cinput = swagepulse(y,-0.4,0.4 )*swagepulse(t,0,1)

   Kc = 0.002        { contaminant diffusivity }

   transfermesh("lowvisc_full_01.dat", uin, vin, pin)

initial values
   u = uin
   v = vin
   p = pin

equations
   u:  visc*div(grad(u)) - dx(p) = dens*dt(u) + dens*(u*dx(u) + v*dy(u))
   v:  visc*div(grad(v)) - dy(p) = dens*dt(v) + dens*(u*dx(v) + v*dy(v))
   p:  div(grad(p)) = penalty*(dx(u)+dy(v))
   c:  dt(c) + u*dx(c) + v*dy(c) = div(Kc*grad(c))

boundaries
   region 1
      start(-Lx,-Ly)
      value(u) = 0   value(v) = 0   load(p) = 0  load(c)=0
        line to (Lx,-Ly)

      load(u) = 0  value(v) = 0  value(p) = p0
      { Introduce a lump of contaminant: }
      value(c) = cinput
      mesh_spacing=Ly/20
        line to (Lx,Ly)

      mesh_spacing=100
      value(u)=0  value(v)=0  load(p)= 0  load(c)=0
        line to(-Lx,Ly)

      load(u) = 0  value(v) = 0  value(p) = 0
        line to close

     exclude
      start(Lx/2-rball,0)
      value(u)=0  value(v)=0  load(p)= 0  load(c)=0
        line to (Lx/2-rball,-rball) bevel(cut)
             to (Lx/2+rball,-rball) bevel(cut)
             to (Lx/2+rball,rball) bevel(cut)
             to (Lx/2-rball,rball) bevel(cut)
        line to close


time 0 to 10

monitors
   for cycle = 1
   contour(speed)  report(Re)
   contour(c) range(0,1)  report(Re)
   elevation(cinput) from (Lx,-Ly) to (Lx,Ly)


plots
   for t=0 by 0.05 to endtime
   contour(min(max(c,0),1))   range(0,1) painted nominmax report(Re)
   contour(u)  report(Re)
   contour(v)  report(Re)
   contour(speed) painted  report(Re)
   vector(u,v) as "flow"   report(Re)
   contour(p)  as "Pressure" painted
   contour(dx(u)+dy(v)) as "Continuity Error"

   history(integral(c))
   history(u) at (0,0.8) (2,0.8) (3,0.8) (4,0.8) (Lx,0)
   history(v) at (0,0.8) (2,0.8) (3,0.8) (4,0.8)

end
```

6.1.4.14 flowslab

```
{  FLOWSLAB.PDE

  This problem considers the laminar
  flow of an incompressible, inviscid
  fluid past an obstruction.

  We assume that the flow can be
  represented by a stream function, PSI,
  such that the velocities, U in the
  x-direction and V in the y-direction,
  are given by:
        U = -dy(PSI)
        V = dx(PSI)

  The flow can then be described by the
  equation
        div(grad(PSI)) = 0.

  The contours of PSI describe the flow
  trajectories of the fluid.

  The problem presented here describes
  the flow past a slab tilted at
  45 degrees to the flow direction. The
  left and right boundaries are held
  at PSI=y, so that U=-1, and V=0.

}
title "Stream Function Flow past 45-degree slab"

variables
   psi              { define PSI as the system variable }

definitions
   a = 3;  b = 3    { size of solution domain }
   len = 0.5        { projection of length/2 }
   wid = 0.1        { projection of width/2 }
   psi_far = y      { solution at large x,y }

equations           { the equation of continuity: }
   psi : div(grad(psi)) = 0

boundaries
   region 1                 { define the domain boundary }
      start(-a,-b)          { start at the lower left }
      value(psi)= psi_far   { impose U=-1 on the outer boundary }
      line to (a,-b)        { walk the boundary Counter-Clockwise }
           to (a,b)
           to (-a,b)
           to close         { return to close }

      start(-len-wid,len-wid)   { start at upper left corner of slab }
      value(psi)=0              { specify no flow on the slab surface }
      line to (-len+wid,len+wid){ walk around the slab CLOCKWISE for exclusion }
           to (len+wid,-len+wid)
           to (len-wid,-len-wid)
           to close             { return to close }

monitors
   contour(psi) { show the potential during solution }

plots           { write hardcopy files at termination }
   grid(x,y)                { show the final grid }
   grid(x,y)   zoom(-1,0,1,1)   { magnify gridding at corner }
   contour(psi) as "stream lines"      { show the stream function }
   vector(-dy(psi),dx(psi)) as "flow"    { show the flow vectors }
   vector(-dy(psi),dx(psi)) as "flow" zoom(-1,0,1,1)

end
```

6.1.4.15 geoflow

```
{ GEOFLOW.PDE

  In its simplest form, the nonlinear steady-state quasi-geostrophic equation
  is the coupled set:

            q  = eps*del2(psi) + y                    (1)

        J(psi,q) = F(x,y) - k*del2(psi)               (2)


  where psi     is the stream function
        q       is the absolute vorticity
        F       is a specified forcing function

        eps and k are specified parameters

        J       is the Jacobian operator:

                J(a,b) = dx(a)*dy(b) - dy(a)*dx(b)

  The single boundary condition is the one on psi stating that the closed
  boundary C of the 2D area should be streamline:

        psi = 0 on C.

  In this test, the term k*del2(psi) in (2) has been replaced by (k/eps)*(q-y),
  and a smoothing diffusion term damp*del2(q) has been added.

  Only the natural boundary condition is needed for Q.
}

title 'Quasi-Geostrophic Equation, square, eps=0.005'

variables
    psi
    q

definitions
    kappa = .05
    epsilon = 0.005
    koe = kappa/epsilon
    size = 1.0
    f = -sin(pi*x)*sin(pi*y)
    damp =  1.e-3*koe

initial values
    psi = 0.
    q   = y

equations
    psi: epsilon*del2(psi) - q = -y
    q:   dx(psi)*dy(q) - dy(psi)*dx(q) + koe*q - damp*del2(q) = koe*y + f

boundaries
    region 1
        start(0,0)
        value(psi)=0  natural(q)=0  line to (1,0)
        value(psi)=0  natural(q)=0  line to (1,1)
        value(psi)=0  natural(q)=0  line to (0,1)
        value(psi)=0  natural(q)=0  line to close

monitors
    contour(psi)
    contour(q)

plots
    contour(psi) as "Potential"
    contour(q)   as "Vorticity"
    surface(psi) as "Potential"
    surface(q)   as "Vorticity"
    vector(-dy(psi),dx(psi)) as "Flow"

end
```

6.1.4.16 hyperbolic

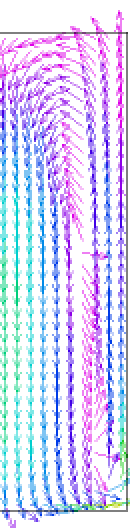
```
{ HYPERBOLIC.PDE

  This problem shows the capabilities of FlexPDE in hyperbolic systems.

  We analyze a single turn of a helical tube with a programmed flow velocity.
  A contaminant is introduced into the center of the flow on the input surface.
  Contaminant outflow is determined from the flow equations.
  The contaminant concentration should flow uniformly around the helix.
}

title 'Helical Flow: a hyperbolic system.'

select
  ngrid=30  regrid=off  { Fixed grid works better in hyperbolic systems }
  contourgrid=60    { increase plot grid density to resolve peak }
  surfacegrid=60

variables
  u

definitions
  Rin = 1
  Rout = 2
  R0 = 1.5
  dR = 0.3     { width of the input contaminant profile }
  gap = 10     { angular gap between input and output faces }
  gapr = gap*pi/180 { gap in radians }
  cg = cos(gapr)
  sg = sin(gapr)
  pin = point(Rin*cg,-Rin*sg)
  pout = point(Rout*cg,-Rout*sg)

  r = magnitude(x,y)
  v = 1
  vx = -v*y/r
  vy = v*x/r
  q = 0         { No Source }
  sink = 0      { No Sink }

equations
  u : div(vx*u, vy*u) + sink*u + q = 0

boundaries
  region 1
    start (Rout,0)
    value(u) = 0        { We know there should be no contaminant on walls  }
      arc(center=0,0) angle=360-gap    { positive angle on outside }

    nobc(u) { "No BC" on exit plane allows internal solution to dictate outflow }
      line to pin

    value(u)=0
      arc(center=0,0) angle=gap-360    { negative angle on inside }

    value(u)=exp(-((x-R0)/dR)^4)    { programmed inflow is supergaussian }
      line to (1.2,0) to (1.4,0) to (1.6,0) to (1.8,0) to close  { resolve shape }

monitors
  contour(u)

plots
  contour(u) painted
  surface(u)
  elevation(u) from (Rin,0.01) to (Rout,0.01)
  elevation(u) from (0,Rin) to (0,Rout)
  elevation(u) from (-Rin,0.01) to (-Rout,0.01)
  elevation(u) from (0,-Rin) to (0,-Rout)
  elevation(u) from pout to pin

end
```

6.1.4.17 lowvisc

```
{  LOWVISC.PDE

  This example is a modification of the VISCOUS.PDE[329] problem, in which the
  viscosity has been lowered to produce a Reynold's number of approximately
  40.  This seems to be the practical upper limit or Reynolds number for
  steady-state solutions of Navier-Stokes equations with FlexPDE.

  As the input pressure is raised, the disturbance in velocities propagates farther
  down the channel.  The channel must be long enough that the velocities
  have returned to the open-channel values, or the P=0 boundary condition
  at the outlet will be invalid and the solution will not succeed.

  The problem computes half of the domain, with a reflective boundary at the bottom.

  We have included four elevation plots of X-velocity, at the inlet, channel
  center, obstruction center and outlet of the channel.  The integrals presented
  on these plots show the consistency of mass transport across the channel.

  We have added a variable psi to compute the stream function for plotting stream lines.

}
title 'Viscous flow in 2D channel, Re > 40'


variables
   u(0.1)
   v(0.01)
   p(1)
   psi

definitions
   Lx = 5
   Ly = 1.5
   p0 = 2
   speed2 = u^2+v^2
   speed = sqrt(speed2)
   dens = 1
   visc = 0.04    vxx = (p0/(2*visc*(2*Lx)))*(Ly-y)^2  { open-channel x-velocity }

   rball=0.4
   cut = 0.1    { value for bevel at the corners of the obstruction }

   penalty = 100*visc/rball^2
   Re = globalmax(speed)*(Ly/2)/(visc/dens)

   w = zcomp(curl(u,v))  ! vorticity is the source for streamline equation

initial values
   u = 0.5*vxx  v = 0  p = p0*x/(2*Lx)

equations
   u:  visc*div(grad(u)) - dx(p) = dens*(u*dx(u) + v*dy(u))
   v:  visc*div(grad(v)) - dy(p) = dens*(u*dx(v) + v*dy(v))
   p:  div(grad(p)) = penalty*(dx(u)+dy(v))
then
   psi:  div(grad(psi)) + w = 0  ! solve streamline equation separately from velocities

boundaries
   region 1
      start(-Lx,0)
      load(u) = 0   value(v) = 0  load(p) = 0   value(psi) = 0
        line to (Lx/2-rball,0)

      value(u) = 0  value(v) = 0  load(p) = 0
      mesh_spacing = rball/10  ! dense mesh to resolve obstruction
        line to (Lx/2-rball,rball) bevel(cut)
             to (Lx/2+rball,rball) bevel(cut)
             to (Lx/2+rball,0)

      mesh_spacing = 10*rball  ! cancel dense mesh requirement
      load(u) = 0  value(v) = 0  load(p) = 0
        line to (Lx,0)

      load(u) = 0  value(v) = 0  value(p) = p0  natural(psi) = 0
        line to (Lx,Ly)
```

```
      value(u) = 0  value(v) = 0  load(p) = 0  natural(psi) = normal(-v,u)
        line to (-Lx,Ly)

      load(u) = 0  value(v) = 0  value(p) = 0  natural(psi) = 0
        line to close

monitors
   contour(speed)  report(Re)
   contour(psi) as "Streamlines"
   contour(max(psi,-0.003)) zoom(Lx/2-3*rball,0, 3*rball,3*rball)  as "Vortex Streamlines"
   vector(u,v) as "flow"    zoom(Lx/2-3*rball,0, 3*rball,3*rball) norm

plots
   contour(u)  report(Re)
   contour(v)  report(Re)
   contour(speed) painted  report(Re)
   vector(u,v) as "flow"   report(Re)
   contour(p)  as "Pressure" painted
   contour(dx(u)+dy(v)) as "Continuity Error"
   elevation(u) from (-Lx,0) to (-Lx,Ly)
   elevation(u) from (0,0) to (0,Ly)
   elevation(u) from (Lx/2,0) to (Lx/2,Ly)
   elevation(u) from (Lx,0) to (Lx,Ly)
   contour(psi) as "Streamlines"
   contour(max(psi,-0.003)) zoom(Lx/2-3*rball,0, 3*rball,3*rball)  as "Vortex Streamlines"
   vector(u,v) as "flow"    zoom(Lx/2-3*rball,0, 3*rball,3*rball) norm

end
```

6.1.4.18 swirl



```
{  SWIRL.PDE

  This problem addresses swirling flow in a cylindrical
  vessel driven by a bottom impeller.

  In two-dimensional cylindrical coordinates, we can
  represent three velocity components (radial, axial
  and tangential) as long as there is no variation of
  cross-section or velocity in the azimuthal coordinate.

  The Navier-Stokes equation for flow in an incompressible
  fluid with no body forces can be written in FlexPDE
  notation as
    dens*(dt(U) + dot(U,grad(u)) = -grad(p) + visc*del2(U)
  where U represents the vector fluid velocity, p is the
  pressure, dens is the density and visc is the viscosity
  of the fluid.  Here the pressure can be considered as
  the deviation from static pressure, because uniform static
  forces like gravity can be cancelled out of the equation.

  In two-dimensional steady-state axisymmetric form, this
  equation becomes three component equations, radial (vr),
  tangential (vt) and axial (vz):
    vr*dr(vr) - vr^2/r + vz*dz(vr) + dr(p)
               = visc*[div(grad(vr)) - vr/r^2]
    vr*dr(vt) + vr*vt/r + vz*dz(vt)
               = visc*[div(grad(vt)) - vt/r^2]
    vr*dr(vz) + vz*dz(vz) + dz(p)
               = visc*div(grad(vz))

  Notice that various strange terms arise, representing centrifugal and coriolis
  forces in cylindrical coordinates and derivatives of the unit vectors in the
  viscosity term. Notice also that there are no tangential derivatives, these
  having been assumed zero.

  In principle, these equations are supplemented by the equation of incompressible
  mass conservation:
    div(U) = 0
  but this equation contains no reference to the pressure, which is nominally the
  remaining variable to be determined.

  In practice, we choose to solve a "slightly compressible" system by defining a
  hypothetical equation of state
    p(dens) = p0 + L*(dens-dens0)
  where p0 and dens0 represent a reference density and pressure, and L is a large
  number representing a strong response of pressure to changes of density.  L is
  chosen large enough to enforce the near-incompressibility of the fluid, yet not
```

```
  so large as to erase the other terms of the equation in the finite precision
  of the computer arithmetic.

  The compressible form of the continuity equation is
    dt(dens) + div(dens*U) = 0
  which, together with the equation of state yields
    dt(p) = -L*dens0*div(U)

  In steady state, we can replace the dt(p) by -div(grad(p))
     [see Help | Tech Notes | Smoothing Operators in PDEs"] 277,
  resulting in the final pressure equation:
    div(grad(p)) = L*dens*div(U)

  In a real stirring vessel, the fluid is driven by an impeller bar in the bottom
  of the fluid. Since we cannot directly represent this geometry in an axisymmetric
  model, we approximate the effect of the impeller by a body force on the fluid
  in the lower segment of the domain.  This body force attempts to accelerate the
  fluid to the velocity of the stir bar, with an arbitrary partition of the
  velocity into vr, vt and vz.

}

TITLE 'Swirling cylindrical flow'

COORDINATES
  ycylinder ('r','z')

VARIABLES
  vr(0.001) { radial velocity with minimum expected range }
  vz(0.001) { axial velocity with minimum expected range  }
  vt (0.001){ tangential velocity with minimum expected range  }
  p(0.001)  { pressure, with linear interpolation and minimum expected range  }

DEFINITIONS
  rad=0.01  { vial radius }
  ht=0.035  { vial height }

  dens=1000     { fluid density }
  visc=0.001    { fluid viscosity }

  vm=magnitude(vr,vz,vt)

  div_v=  1/r*dr(r*vr)+dz(vz)   { velocity divergence }

  PENALTY = 1e4*visc/rad^2      { the phony equation of state coefficient }

  band = ht/20  { height of force band }
  bf = 1000     { arbitrary body-force scaling }
  f             { stirbar force - defined by region later }
  rpm =staged(50,100,150)   { several stirring speeds }
  v0 = 2*pi*r*rpm/60        { impeller velocity }
  vr0 = 0.2*V0  { arbitrary partition of stirring velocity }
  vt0 = 1.0*V0
  vz0 = 0.3*V0

  mass_balance = div_v/integral(1)

INITIAL VALUES
  vr=0
  vz=0
  vt=0
  p=0

EQUATIONS
  { Radial Momentum equation }
  vr: dens*(vr*dr(vr) - vt^2/r + vz*dz(vr)) + dr(p) - visc*(div(grad(vr))-vr/r^2)=F*(VR0-vr)
  { Axial Momentum equation }
  vz: dens*(vr*dr(vz) + vz*dz(vz)) + dz(p) - visc*(div(grad(vz)))=F*(vz0-vz)
  { Tangential ("Swirling") Momentum equation (Corrected 2/2/06)}
  vt: dens*(vr*dr(vt) + vr*vt/r + vz*dz(vt)) - visc*(div(grad(vt))-vt/r^2)=F*(vt0-vt)
  { Equation of state }
  p:  div(grad(p)) =  penalty*div_v

BOUNDARIES

  Region 'domain'
    F=0
    Start 'outer' (0,0)
      { mirror conditions on bottom boundary }
```

```
      natural(vr)=0 natural(vt)=0   value(vz)=0     natural(p)=0    line to (rad,0)
      { no slip on sides (ie, velocity=0)  }
      value(vr)=0   value(vt)=0     value(vz)=0     natural(p)=0    line to (rad,ht)
      { zero pressure and no z-flow on top, but free vr and vt }
      natural(vr)=0 natural(vt)=0   value(vz)=0     value(p)=0      line to (0,ht)
      { no radial or tangential velocity on spin axis }
      value(vr)=0   value(vt)=0     natural(vz)=0   natural(p)=0    line to close

  Region "impeller"
    F=bf
    Start(0,0) line to (0.9*rad,0) to (0.9*rad,band) to (0,band) to close

  { add a gridding feature to help resolve the shear layer at the wall }
  Feature start(0.95*rad,0) line to (0.95*rad,ht)

MONITORS
  contour(vr) as "Radial Velocity"  report(rpm)
  contour(vt) as "Swirling Velocity" report(rpm)
  contour(vz) as "Axial Velocity" report(rpm)
  elevation(vt,v0) from(0,0) to (rad,0)  as "Impeller Velocity" report(rpm)
  contour(p) as "Pressure"
  vector(vr,vz) as "R-Z Flow"

PLOTS
  contour(vr) as "Radial Velocity"  report(rpm)
  contour(vt) as "Swirling Velocity"  report(rpm)
  contour(vz) as "Axial Velocity" report(rpm)
  contour(vm) as "Velocity Magnitude" report(rpm)
  contour(p) as "Pressure" report(rpm)
  vector(vr,vz) norm as "R-Z Flow"  report(rpm)
  contour(mass_balance)  report(rpm)
  elevation(vr) from (0,0) to (rad,0) as "Radial Velocity"   report(rpm)
  elevation(vt) from (0,0) to (rad,0) as "Swirling Velocity" report(rpm)
  elevation(vz) from (0,0) to (rad,0) as "Axial Velocity"  report(rpm)
  elevation(vt,v0) from(0,0) to (rad,0)  as "Impeller Velocity" report(rpm)
  elevation(vm) from (0,0) to (rad,0) as "Velocity Magnitude" report(rpm)
  elevation(vr) from (0,ht/2) to (rad,ht/2) as "Radial Velocity"   report(rpm)
  elevation(vt) from (0,ht/2) to (rad,ht/2) as "Swirling Velocity" report(rpm)
  elevation(vz) from (0,ht/2) to (rad,ht/2) as "Axial Velocity"  report(rpm)
  elevation(vm) from (0,ht/2) to (rad,ht/2) as "Velocity Magnitude" report(rpm)
  elevation(vr) from (0,0.9*ht) to (rad,0.9*ht) as "Radial Velocity"   report(rpm)
  elevation(vt) from (0,0.9*ht) to (rad,0.9*ht) as "Swirling Velocity" report(rpm)
  elevation(vz) from (0,0.9*ht) to (rad,0.9*ht) as "Axial Velocity"  report(rpm)
  elevation(vm) from (0,0.9*ht) to (rad,0.9*ht) as "Velocity Magnitude" report(rpm)
  elevation(vm) from (rad/2,0) to (rad/2,ht) as "Velocity Magnitude" report(rpm)

END
```

6.1.4.19 vector_swirl

```
{  VECTOR_SWIRL.PDE

  This is a modification of the example SWIRL.PDE 325 to use vector variables.

}

TITLE 'Swirling cylindrical flow'

COORDINATES
  ycylinder ('r','z')

VARIABLES
  V(0.001)  = vector(vr, vz, vt)
  p(0.001)  { pressure, with linear interpolation and minimum expected range  }

DEFINITIONS
  rad=0.01    { vial radius }
  ht=0.035    { vial height }

  dens=1000     { fluid density }
  visc=0.001    { fluid viscosity }

  vm=magnitude(V)

  div_v=  div(V)    { velocity divergence }

  PENALTY = 1e4*visc/rad^2  { the phony equation of state coefficient }
```

```
  band = ht/20  { height of force band }
  bf = 1000     { arbitrary body-force scaling }
  f             { stirbar force - assigned by region later }
  rpm =staged(50,100,150)   { several stirring speeds }
  vimp = 2*pi*r*rpm/60      { impeller velocity }
  vr0 = 0.2*vimp    { arbitrary partition of stirring velocity }
  vt0 = 1.0*vimp
  vz0 = 0.3*vimp

  V0 = vector(vr0, vz0, vt0)

  mass_balance = div_v/integral(1)

INITIAL VALUES
  vr=0
  vz=0
  vt=0
  p=0

EQUATIONS
  V: dens*dot(V,grad(V)) + grad(p) - visc*div(grad(V)) = F*(V0-V)
  p: div(grad(p)) =  penalty*div_v

BOUNDARIES

  Region 'domain'
    F=0
    Start 'outer' (0,0)
      { mirror conditions on bottom boundary }
      natural(vr)=0 natural(vt)=0   value(vz)=0   natural(p)=0  line to (rad,0)
      { no slip on sides (ie, velocity=0)  }
      value(vr)=0   value(vt)=0     value(vz)=0   natural(p)=0  line to (rad,ht)
      { zero pressure and no z-flow on top, but free vr and vt }
      natural(vr)=0 natural(vt)=0   value(vz)=0   value(p)=0    line to (0,ht)
      { no radial or tangential velocity on spin axis }
      value(vr)=0   value(vt)=0     natural(vz)=0 natural(p)=0  line to close

  Region "impeller"
    F=bf
    Start(0,0) line to (0.9*rad,0) to (0.9*rad,band) to (0,band) to close

  { add a gridding feature to help resolve the shear layer at the wall }
  Feature start(0.95*rad,0) line to (0.95*rad,ht)


MONITORS
  contour(vr) as "Radial Velocity"  report(rpm)
  contour(vt) as "Swirling Velocity" report(rpm)
  contour(vz) as "Axial Velocity" report(rpm)
  elevation(vt,v0) from(0,0) to (rad,0)  as "Impeller Velocity" report(rpm)
  contour(p) as "Pressure"
  vector(vr,vz) as "R-Z Flow"

PLOTS
  contour(vr) as "Radial Velocity"  report(rpm)
  contour(vt) as "Swirling Velocity"  report(rpm)
  contour(vz) as "Axial Velocity" report(rpm)
  contour(vm) as "Velocity Magnitude" report(rpm)
  contour(p) as "Pressure" report(rpm)
  vector(vr,vz) norm as "R-Z Flow"  report(rpm)
  contour(mass_balance)  report(rpm)
  elevation(vr) from (0,0) to (rad,0) as "Radial Velocity"   report(rpm)
  elevation(vt) from (0,0) to (rad,0) as "Swirling Velocity" report(rpm)
  elevation(vz) from (0,0) to (rad,0) as "Axial Velocity"  report(rpm)
  elevation(vt,v0) from(0,0) to (rad,0)  as "Impeller Velocity" report(rpm)
  elevation(vm) from (0,0) to (rad,0) as "Velocity Magnitude" report(rpm)
  elevation(vr) from (0,ht/2) to (rad,ht/2) as "Radial Velocity"   report(rpm)
  elevation(vt) from (0,ht/2) to (rad,ht/2) as "Swirling Velocity" report(rpm)
  elevation(vz) from (0,ht/2) to (rad,ht/2) as "Axial Velocity"  report(rpm)
  elevation(vm) from (0,ht/2) to (rad,ht/2) as "Velocity Magnitude" report(rpm)
  elevation(vr) from (0,0.9*ht) to (rad,0.9*ht) as "Radial Velocity"   report(rpm)
  elevation(vt) from (0,0.9*ht) to (rad,0.9*ht) as "Swirling Velocity" report(rpm)
  elevation(vz) from (0,0.9*ht) to (rad,0.9*ht) as "Axial Velocity"  report(rpm)
  elevation(vm) from (0,0.9*ht) to (rad,0.9*ht) as "Velocity Magnitude" report(rpm)
  elevation(vm) from (rad/2,0) to (rad/2,ht) as "Velocity Magnitude" report(rpm)

END
```

6.1.4.20 viscous

```
{  VISCOUS.PDE

  This example shows the application of FlexPDE to problems in
  viscous flow.

  The Navier-Stokes equation for steady incompressible flow in two
  cartesian dimensions is
        dens*(dt(U) + U*dx(U) + V*dy(U)) = visc*del2(U) - dx(P) + dens*Fx
        dens*(dt(V) + U*dx(V) + V*dy(V)) = visc*del2(V) - dy(P) + dens*Fy
  together with the continuity equation
        div[U,V] = 0
  where
        U and V are the X- and Y- components of the flow velocity
        P is the fluid pressure
        dens is the fluid density
        visc is the fluid viscosity
        Fx and Fy are the X- and Y- components of the body force.

  In principle, the third equation enforces incompressible mass conservation,
  but the equation contains no reference to the pressure, which is nominally
  the remaining variable to be determined.

  In practice, we choose to solve a "slightly compressible" system by defining a
  hypothetical equation of state
    p(dens) = p0 + L*(dens-dens0)
  where p0 and dens0 represent a reference density and pressure, and L is a large
  number representing a strong response of pressure to changes of density.  L is
  chosen large enough to enforce the near-incompressibility of the fluid, yet not
  so large as to erase the other terms of the equation in the finite precision
  of the computer arithmetic.

  The compressible form of the continuity equation is
    dt(dens) + div(dens*U) = 0
  which, together with the equation of state yields
    dt(p) = -L*dens0*div(U)

  In steady state, we can replace the dt(p) by -div(grad(p))
     [see Help | Tech Notes | Smoothing Operators in PDEs"],
  resulting in the final pressure equation:
    div(grad(p)) = M*div(U)
  Here M has the dimensions of density/time or viscosity/distance^2.

  The problem posed here shows flow in a 2D channel blocked by a bar of square
  cross-section.  The channel is mirrored on the bottom face, and only the upper
  half is computed.

  We have chosen a "convenient" value of M, one that gives good
  accuracy in reasonable time.  The user can alter this value to find one
  which is satisfactory for his application.

  We have included three elevation plots of X-velocity, at the inlet, center
  and outlet of the channel.  The integrals presented on these plots show the
  consistency of mass transport across the channel.

 }

title 'Viscous flow in 2D channel, Re < 0.1'

variables
   u(0.1)
   v(0.01)
   p(1)

definitions
   Lx = 5         Ly = 1.5
   Gx = 0         Gy = 0
   p0 = 1
   speed2 = u^2+v^2
   speed = sqrt(speed2)
   dens = 1
   visc = 1

   vxx = (p0/(2*visc*Lx))*(Ly-y)^2  { open-channel x-velocity }

   rball=0.25
   cut = 0.05   { value for bevel at the corners of the obstruction }
```

```
   penalty = 100*visc/rball^2   { "equation of state" }
   Re = globalmax(speed)*(Ly/2)/(visc/dens)

initial values
   u = 0.5*vxx  v = 0  p = p0*x/Lx

equations
   u:  visc*div(grad(u)) - dx(p) = dens*(u*dx(u) + v*dy(u))
   v:  visc*div(grad(v)) - dy(p) = dens*(u*dx(v) + v*dy(v))
   p:  div(grad(p)) = penalty*(dx(u)+dy(v))

boundaries
   region 1
     start(0,0)
       { unspecified boundary conditions default to LOAD=0 }
       value(v)=0   line to (Lx/2-rball,0)
       value(u)=0   line to (Lx/2-rball,rball) bevel(cut)
                    line to (Lx/2+rball,rball) bevel(cut)
                    line to (Lx/2+rball,0)
       load(u)=0    line to (Lx,0)
       value(p)=p0  line to (Lx,Ly)
       value(u)=0   load(p)=0   line to (0,Ly)
       load(u)=0    value(p)=0  line to close

monitors
   contour(speed) report(Re)

plots
   grid(x,y)
   contour(u)  report(Re)
   contour(v)  report(Re)
   contour(speed) painted  report(Re)
   vector(u,v) as "flow"   report(Re)
   contour(p)  as "Pressure" painted
   contour(dx(u)+dy(v)) as "Continuity Error"
   contour(p) zoom(Lx/2,0,1,1) as "Pressure"
   elevation(u) from (0,0) to (0,Ly)
   elevation(u) from (Lx/2,0) to (Lx/2,Ly)
   elevation(u) from (Lx,0) to (Lx,Ly)

end
```

6.1.5 groundwater

6.1.5.1 porous

```
{  POROUS.PDE

  This problem describes the flow through an anisotropic porous foundation.
  It is taken from Zienkiewicz, "The Finite Element Method in Engineering Science",
  p. 305.

}

title 'Anisotropic Porous flow'

variables
  pressure

definitions
  ky = 1
  kx = 4

equations
  pressure : dx(kx*dx(pressure)) + dy(ky*dy(pressure)) = 0

boundaries
  region 1
      start(0,0)
        natural(pressure)=0 line to (5,0) to (5,5)
        value(pressure)=0   line to (2,2)
        natural(pressure)=0 line to (2.5,2) to (2.5,1.95) to (1.95,1.95)
        value(pressure)=100 line to close

monitors
  contour(pressure)
```

```
plots
  contour(pressure)
  surface(pressure)

end
```

6.1.5.2 richards

```
{  RICHARDS.PDE

  A solution of Richards' equation in 1D.
  Constant negative head at surface, unit gradient at bottom.
  This problem runs slowly, because the very steep wave front
  requires small cells and small timesteps to track accurately.

  submitted by Neil Soicher of University of Hawaii.
}

title "1-D Richard's equation"

coordinates
    cartesian1('y')

variables
    h (1)

definitions
    thr = 0.2
    ths = 0.58
    alpha = .08
    n = 1.412
    ks = 10
    {Using Van Genuchten parameters for water content (wc),
     water capacitance (C=d(wc)/dh), effecive saturation (se),
     and hydraulic Conductivity (k) }
    m  = 1-1/n
    wc = if h<0 then thr+(ths-thr)*(1+(abs(alpha*h))^n)^(-m) else ths
    C  = ((1-n)*abs(-alpha*h)^n*(1+abs(-alpha*h)^n)^((1/n)-2)*(ths-thr))/h
    se = (wc-thr)/(ths-thr)
    k  = ks*sqrt(se)*(1-(1-se^(1/m))^m)^2

initial values
    h = 199*exp(-(y-100)^2)-200

equations
    h : dy(k*(dy(h)+1)) = C*dt(h)

boundaries
    region 1
        start(0)
        line to (100) point value(h) = -1

front(h+150,1)

time 0 to 2

monitors
    for cycle=10
        elevation(c) from (100) to (0) as "capacitance"
        elevation(h) from (100) to (0) as "pressure"
        elevation(k) from (100) to (0) log as "conductivity"
        grid(y) from (100) to (0)

plots
     for t=0.001 0.01 0.1 1
        elevation(h) from (100) to (0) as "pressure"
        elevation(c) from (100) to (0) as "capacitance"
        elevation(k) from (100) to (0) log as "conductivity"
        grid(y) from (100) to (0)

end
```

6.1.5.3 water

```
{  WATER.PDE

    This problem shows the flow of water to two wells, through soil regions of
    differing porosity.  It also displays the ability of FlexPDE to grid features
    of widely varying size.

}

title 'Groundwater flow to two wells'

definitions
    k                    { no value is required, as long as it appears later }
    s = 0                { no volumetric source }
    k1 = 0.1             { high porosity value }
    k2 = 1.0e-7          { low  porosity value }
    sx1 = 0.7   sy1 = 0.4{ well 1 location }
    sx2 = 0.5   sy2 = 0.2{ well 2 location }
    srad = 0.001         { well radius = one thousandth of domain size }
    w = 0.05             { a zoom window size }

    px = 0.4    py = 0.4 { percolation pond center }
    pr = 0.025           { percolation pond radius }
    ps = 1e-4            { percolation rate }

variables
    h

equations
    h : div(k*grad(h)) + s = 0

boundaries

  region 1  { The domain boundary, held at constant pressure head }
    k=k1
    start(0,0)
    value(h)=0 line to (0.25,-0.1)
                    to (0.45,-0.1)
                    to (0.65,0)
                    to (0.95,0.1)
                    to (0.95,0.4)
                    to (0.75,0.6)
                    to (0.45,0.65)
                    to (0,0.4)
                    to close

    { Two wells, held at constant draw-down depths }
    start(sx1,sy1-srad)
    value(h) = -1        arc(center=sx1,sy1) angle=-360
    start(sx2,sy2-srad)
    value(h) = -2        arc(center=sx2,sy2) angle=-360

  region 2  { Some regions of low porosity }
    k=k2
    start(0,0) line to (0.25,-0.1)
                    to (0.45,-0.1)
                    to (0.45,0.05)
                    to (0,0.05)
                    to close

    start(0.95,0.1) line to (0.95,0.3)
                    to (0.65,0.3)
                    to (0.65,0)
                    to close

    start(0.3,0.3) line to (0.5,0.4)
                    to (0.5,0.6)
                    to (0.3,0.5)
                    to close

  region 3  { A percolation pond }
    k = k2
    s = ps  { percolation rate }
    start (px,py-pr) arc(center=px,py) angle=360

monitors
  contour(h)
```

```
plots
  grid(x,y)
  grid(x,y) zoom(sx1-w/2,sy1-w/2,w,w)
  grid(x,y) zoom(sx2-w/2,sy2-w/2,w,w)
  contour(h) as 'Head'
  contour(h) as 'Head' zoom(0.65,0.35,0.1,0.1)
  surface(h) as 'Head'

end
```

6.1.6 heatflow

6.1.6.1 1d_float_zone

```
{  1D_FLOAT_ZONE.PDE

   This is a version of the example FLOAT_ZONE.PDE 337 in 1D cartesian geometry.

}

title
  "Float Zone in 1D Cartesian geometry"

select
  nodelimit=100

coordinates
  cartesian1

variables
  temp(threshold=100)

definitions
  k = 10    {thermal conductivity}
  cp = 1    { heat capacity }
  long = 18
  H = 0.4   {free convection boundary coupling}
  Ta = 25   {ambient temperature}
  A = 4500  {amplitude}

  source = A*exp(-((x-1*t)/.5)^2)*(200/(t+199))

initial value
  temp = Ta

equations
  temp : div(k*grad(temp)) + source -H*(temp - Ta) = cp*dt(temp)

boundaries
  region 1
    start(0) point value(temp) = Ta
    line to (long) point value(temp) = Ta

time -0.5 to 19 by 0.01

monitors
  for t = -0.5 by 0.5 to (long + 1)
    elevation(temp) from (0) to (long) range=(0,1800) as "Surface Temp"

plots
  for t = -0.5 by 0.5 to (long + 1)
    elevation(temp) from (0) to (long) range=(0,1800) as "Axis Temp"
    elevation(source) from(0) to (long)
    elevation(-k*grad(temp)) from(0) to (long)

histories
  history(temp) at (0) (1) (2) (3) (4) (5) (6) (7) (8) (9)
                   (10) (11) (12) (13) (14) (15) (16) (17) (18)

end
```

6.1.6.2 3d_bricks+time

```
{  3D_BRICKS+TIME.PDE

 This problem demonstrates the application of FlexPDE to time-dependent
```

```
 three dimensional heat conduction.  An assembly of bricks of differing
 conductivities has a gaussian internal heat source, with all faces held
 at zero temperature.  After a time, the temperature reaches a stable
 distribution.
 This is the time-dependent analog of example problem 3D_BRICKS.PDE 335.

}

title 'time-dependent 3D heat conduction'

select
    regrid=off  { use fixed grid }
    ngrid=5     { smaller grid for quicker run }

coordinates
    cartesian3

variables
    Tp(threshold=0.1)   { the temperature variable, with approximate size  }

definitions
    long = 1
    wide = 1
    K   { thermal conductivity -- values supplied later }
    Q = 10*exp(-x^2-y^2-z^2)    { Thermal source }
    tmax = 6                    { plot range control }

initial values
    Tp = 0.

equations
    Tp : div(k*grad(Tp)) + Q = dt(Tp)   { the heat equation }

extrusion z = -long,0,long      { divide Z into two layers }

boundaries
    surface 1 value(Tp)=0       { fix bottom surface temp }
    surface 3 value(Tp)=0       { fix top surface temp }

    Region 1    { define full domain boundary in base plane }
       layer 1 k=1              { bottom right brick }
       layer 2 k=0.1            { top right brick }
       start(-wide,-wide)
         value(Tp) = 0          { fix all side temps }
         line to (wide,-wide)   { walk outer boundary in base plane }
              to (wide,wide)
              to (-wide,wide)
              to close

    Region 2    { overlay a second region in left half }
       layer 1 k=0.2            { bottom left brick }
       layer 2 k=0.4            { top left brick }
       start(-wide,-wide)
         line to (0,-wide)      { walk left half boundary in base plane }
              to (0,wide)
              to (-wide,wide)
              to close

time 0 to 3 by 0.01 { establish time range and initial timestep }

monitors
  for cycle=1
    contour(Tp) on z=0  as "XY Temp" range=(0,tmax)
    contour(Tp) on x=0  as "YZ Temp" range=(0,tmax)
    contour(Tp) on y=0  as "XZ Temp" range=(0,tmax)
    elevation(Tp) from (-wide,0,0) to (wide,0,0) as "X-Axis Temp" range=(0,tmax)
    elevation(Tp) from (0,-wide,0) to (0,wide,0) as "Y-Axis Temp" range=(0,tmax)
    elevation(Tp) from (0,0,-long) to (0,0,long) as "Z-Axis Temp" range=(0,tmax)

plots
  for t = endtime
    contour(Tp) on z=0  as "XY Temp" range=(0,tmax)
    contour(Tp) on x=0  as "YZ Temp" range=(0,tmax)
    contour(Tp) on y=0  as "XZ Temp" range=(0,tmax)

histories
    history(Tp) at (wide/2,-wide/2,-long/2)
                   (wide/2,wide/2,-long/2)
                   (-wide/2,wide/2,-long/2)
```

```
                    (-wide/2,-wide/2,-long/2)
                    (wide/2,-wide/2,long/2)
                    (wide/2,wide/2,long/2)
                    (-wide/2,wide/2,long/2)
                    (0,0,0)  range=(0,tmax)

end
```

6.1.6.3 3d_bricks

```
{  3D_BRICKS.PDE

 This problem demonstrates the application
 of FlexPDE to steady-state three dimensional
 heat conduction.  An assembly of four bricks
 of differing conductivities has a gaussian
 internal heat  source,  with all faces held
 at zero temperature.  After a time, the
 temperature reaches a stable distribution.

 This is the steady-state analog of problem
 3D_BRICKS+TIME.PDE 333

}
title 'steady-state 3D heat conduction'

select
    regrid=off  { use fixed grid }

coordinates
    cartesian3

variables
    Tp

definitions
    long = 1
    wide = 1
    K   { thermal conductivity -- values supplied later }
    Q = 10*exp(-x^2-y^2-z^2)    { Thermal source }

initial values
    Tp = 0.

equations
    Tp : div(k*grad(Tp)) + Q = 0    { the heat equation }

extrusion z = -long,0,long      { divide Z into two layers }

boundaries
    Surface 1 value(Tp)=0       { fix bottom surface temp }
    Surface 3 value(Tp)=0       { fix top surface temp }

    Region 1    { define full domain boundary in base plane }
      layer 1 k=1               { bottom right brick }
      layer 2 k=0.1             { top right brick }
      start(-wide,-wide)
        value(Tp) = 0           { fix all side temps }
        line to (wide,-wide)    { walk outer boundary in base plane }
          to (wide,wide)
          to (-wide,wide)
          to close

    Region 2    { overlay a second region in left half }
      layer 1 k=0.2             { bottom left brick }
      layer 2 k=0.4             { top left brick }
      start(-wide,-wide)
        line to (0,-wide)       { walk left half boundary in base plane }
          to (0,wide)
          to (-wide,wide)
          to close

monitors
    contour(Tp) on z=0  as "XY Temp"
    contour(Tp) on x=0  as "YZ Temp"
    contour(Tp) on y=0  as "ZX Temp"
```

```
    elevation(Tp) from (-wide,0,0) to (wide,0,0)  as "X-Axis Temp"
    elevation(Tp) from (0,-wide,0) to (0,wide,0)  as "Y-Axis Temp"
    elevation(Tp) from (0,0,-long) to (0,0,long)  as "Z-Axis Temp"

plots
    contour(Tp) on z=0  as "XY Temp"
    contour(Tp) on x=0  as "YZ Temp"
    contour(Tp) on y=0  as "ZX Temp"

end
```

6.1.6.4 axisymmetric_heat

```
{  AXISYMMETRIC_HEAT.PDE

  This example demonstrates axi-symmetric
  heatflow.

  The heat flow equation in any coordinate
  system is
       div(K*grad(T)) + Source = 0.

  The following problem is taken from
  Zienkiewicz, "The Finite Element Method
  in Engineering Science", p. 306 (where
  the solution is plotted, but no dimensions
  are given). It describes the flow of heat
  in a spherical vessel.
  The outer boundary is held at Temp=0,
  while the inner boundary is held at Temp=100.

}

title "Axi-symmetric Heatflow "

coordinates
  ycylinder("R","Z")    { select a cylindrical coordinate system, with
                           the rotational axis along the "Y" direction
                            and the coordinates named "R" and "Z" }
variables
  Temp              { Define Temp as the system variable }

definitions
  K = 1             { define the conductivity }
  source = 0        { define the source (this problem doesn't have one) }

Initial values
  Temp = 0          { unimportant in linear steady-state problems }

equations           { define the heatflow equation: }
  Temp : div(K*grad(Temp)) + Source = 0

boundaries          { define the problem domain }
  Region 1          { ... only one region }

    start(5,0)
    natural(Temp)=0 { define the bottom symmetry boundary condition }
    line to (6,0)

    value(Temp)=0   { fixed Temp=0 in outer boundary }
    line to (6,3)   { walk the funny stair-step outer boundary }
         to (5,3)
         to (5,4)
         to (4,4)
         to (4,5)
         to (3,5)
         to (3,6)
         to (0,6)

    natural(Temp)= 0    { define the left symmetry boundary }
    line to (0,5)

    value(Temp)=100     { define the fixed inner temperature }
    arc( center=0,0) to close   { walk an arc to the starting point }

monitors
  contour(Temp)     { show contour plots of solution in progress }
```

```
plots                { write these hardcopy files at completion }
  contour(Temp)      { show solution }
  surface(Temp)
  vector(-2*pi*r*K*grad(Temp)) as "Heat Flow"

end
```

6.1.6.5 float_zone

```
{ FLOAT_ZONE.PDE

  This example illustrates time-dependent axi-symmetric heat flow with a
  moving source.

  A rod of conductive material of unit radius and "long" units length
  is clamped to a heat sink at either end.  An RF coil passes the
  length of the rod, creating a moving heat source of gaussian profile.
  This produces a moving melt zone which carries impurities with it as it moves.
  A cam adjusts the source amplitude by 200/(t+199) to produce an approximately
  constant maximum temperature.

}

title
  "Float Zone"

coordinates
  xcylinder('Z','R')

select
  cubic     { Use Cubic Basis }

variables
  temp(threshold=100)

definitions
  k = 0.85      { thermal conductivity }
  cp = 1        { heat capacity }
  long = 18
  H = 0.4       { free convection boundary coupling }
  Ta = 25       { ambient temperature }
  A = 4500      { amplitude }

  source = A*exp(-((z-1*t)/.5)^2)*(200/(t+199))

initial value
  temp = Ta

equations
  temp : div(k*grad(temp)) + source = cp*dt(temp)

boundaries
  region 1
    start(0,0)
    natural(temp) = 0 line to (long,0)
    value(temp) = Ta  line to (long,1)
    natural(temp) = -H*(temp - Ta) line to (0,1)
    value(temp) = Ta  line to close
  feature
    start(0.01*long,0) line to (0.01*long,1)

time -0.5 to 19 by 0.01

monitors
  for t = -0.5 by 0.5 to (long + 1)
    elevation(temp) from (0,1) to (long,1) range=(0,1800) as "Surface Temp"
    contour(temp)

plots
  for t = -0.5 by 0.5 to (long + 1)
    elevation(temp) from (0,0) to (long,0) range=(0,1800) as "Axis Temp"

histories
  history(temp) at (0,0) (1,0) (2,0) (3,0) (4,0) (5,0) (6,0) (7,0) (8,0)
                   (9,0) (10,0) (11,0) (12,0) (13,0) (14,0) (15,0) (16,0)
                   (17,0) (18,0)

end
```

6.1.6.6 heat_boundary

```
{ HEAT_BOUNDARY.PDE

    This problem shows the use of natural boundary conditions to model
    insulation, reflection, and convective losses.

    The heatflow equation is
        div(K*grad(Temp)) + Source = 0

    The Natural boundary condition specifies the value of the surface-normal
    component of the argument of the divergence operator, ie:
        Natural Boundary Condition  =  normal <dot> K*grad(Temp)

    Insulating boundaries and symmetry boundaries therefore require the
    boundary condition:
        Natural(Temp) = 0

    At a convective boundary, the heat loss is proportional to the temperature
    difference between the surface and the coolant. Since the heat flux is
        F = -K*grad(Temp)  =  b*(Temp - Tcoolant)
    the appropriate boundary condition is
        Natural(Temp) = b*(Tcoolant - Temp).

    In this problem, we define a quarter of a circle, with reflective
    boundaries on the symmetry planes to model the full circle.  There is a
    uniform heat source of 4 units throughout the material.  The outer
    boundary is insulated, so the natural boundary condition is used to
    specify no heat flow.

    Centered in the quadrant is a cooling hole. The temperature of the
    coolant is Tzero, and the heat loss to the coolant is (Tzero - Temp)
    heat units per unit area.

    In order to illustrate the characteristics of the Finite Element model,
    we have selected output plots of the normal component of the heat flux
    along the system boundaries.  The F.E. method forms its equations based
    on the weighted average of the deviation of the approximate solution
    to the PDE over each cell.  There is no guarantee that on the outer
    boundary, for example, where the Natural(Temp) = 0, the point-by-point
    value of the normal derivative will necessarily be zero. But the integral
    of the PDE over each cell should be correct to within the requested
    accuracy.

    Here we have requested three solution stages, with successively tighter
    accuracy requirements of 1e-3, 1e-4 and 1e-5.

    Notice in plot 7 that while the pointwise values of the normal flux
    oscillate by ten percent in the first stage, they oscillate about the
    same solution as the later stages, and the integral of the heat loss is
    2.628, 2.642 and 2.6395 for the three stages.  Compare this with the
    analytic integral of the source (2.6389) and with the numerical integral
    of the source in plot 5 (all 2.6434).  The Divergence Theorem is
    therefore satisfied to 0.004, 0.001, and 0.0002 in the three stages.

    In plot 7,  "Integral" and "Bintegral" differ because they are the result
    of different quadrature rules applied to the data.
}
```

```
title "Coolant Pipe Heatflow"

select
    stages = 3
    errlim = staged(1e-3,1e-4,1e-5)
    autostage=off

variables
    Temp

definitions
    K = 1              { conductivity }
    source = 4         { source }
    Tzero = 0          { coolant temperature }
    flux = -K*grad(Temp)  { thermal flux vector }
initial values
    Temp = 0

equations
    Temp : div(K*grad(Temp)) + source = 0


boundaries                    { define the problem domain }
    Region 1                  { ... only one region }
        start "OUTER" (0,0) { start at the center }
        natural(Temp)=0      { define the bottom symmetry boundary condition }
        line to(1,0)         { walk to the surface }

        natural(Temp)=0      { define the "Zero Flow" boundary condition }
        arc (center=0,0) to (0,1)    { walk the outer arc }

        natural(Temp)=0      { define the Left symmetry B.C. }
        line to close        { return to close }

        start "INNER" (0.4,0.2) { define the excluded coolant hole }
        natural(Temp)=Tzero-Temp { "Temperature-difference" flow boundary.
                                   Negative value means negative K*grad(Temp)
                                     or POSITIVE heatflow INTO coolant hole }
        arc (center=0.4,0.4){ walk boundary CLOCKWISE for exclusion }
           to (0.6,0.4)
           to (0.4,0.6)
           to (0.2,0.4)
           to close

monitors
    contour(Temp)    { show contour plots of solution in progress }

plots              { write these hardcopy files at completion: }
    grid(x,y)                          { show the final grid }
    contour(Temp)                      { show the solution  }
    surface(Temp)
    vector(-K*dx(Temp),-K*dy(Temp)) as "Heat Flow"
    contour(source)                    { show the source to compare integral }
    elevation(normal(flux)) on "outer" range(-0.08,0.08)
       report(bintegral(normal(flux),"outer")) as "bintegral"
    elevation(normal(flux)) on "inner" range(1.95,2.3)
       report(bintegral(normal(flux),"inner")) as "bintegral"

histories
    history(bintegral(normal(flux),"inner"))

end
```

6.1.6.7 radiation_flow

```
{ RADIATION_FLOW.PDE

  This problem demonstrates the use of FlexPDE
  in the solution of problems in radiative transfer.

  Briefly summarized, we solve a Poisson equation
  for the radiation energy density, assuming that
  at every point in the domain the local
  temperature has come into equilibrium with the
  impinging radiation field.

  We further assume that the spectral character-
  istics of the radiation field are adequately
  described by three average cross-sections:
  the emission average, or "Planck Mean", sigmap;
  the absorption average, sigmaa; and the transport
  average, or "Rosseland Mean-Free-Path", lambda.
  These averages may, of course, differ in various
  regions, but they must be estimated by facilities
  outside the scope of FlexPDE.

  And finally, we assume that the radiation field
  is sufficiently isotropic that Fick's Law, that
  the flux is proportional to the gradient of the
  energy density, is valid.

  The problems shows a hot slab radiating across an
  air gap and heating a distant dense slab.
}

title 'Radiative Transfer'

variables
    erad    { Radiation Energy Density }

definitions
    source          { declare the parameters, values will follow }
    lambda          { Rosseland Mean Free Path }
    sigmap          { Planck Mean Emission cross-section }
    sigmaa          { absorption average cross-section }
    beta = 1/3      { Fick's Law proportionality factor }

equations   { The radiation flow equation: }
    erad : div(beta*lambda*grad(erad)) + source = 0

boundaries

    region 1    { the bounding region is tenuous }
      source=0  sigmap=2    sigmaa=1    lambda=10
      start(0,0)
      natural(erad)=0       { along the bottom, a zero-flux symmetry plane }
      line to (1,0)
      natural(erad)=-erad   { right and top, radiation flows out }
      line to (1,1) to (0,1)
      natural(erad)=0       { Symmetry plane on left }
      line to close

    region 2    { this region has a source and large cross-section }
      source=100    sigmap=10       sigmaa=10   lambda=1
      start(0,0)
      line to (0.1,0) to (0.1,0.5) to (0,0.5) to close

    region 3    { this opaque region is driven by radiation }
      source=0  sigmap=10   sigmaa=10   lambda=1
      start(0.7,0)
      line to (0.8,0) to (0.8,0.3) to (0.7,0.3) to close

monitors
    contour(erad)

plots
    contour(erad) as 'Radiation Energy'
    surface(erad) as 'Radiation Energy'
    vector(-beta*lambda*grad(erad)) as 'Radiation Flux'

    { the temperature can be calculated from the assumption of equilibrium: }
    contour(sqrt(sqrt(erad*sigmaa/sigmap))) as 'Temperature'
```

```
    surface(sqrt(sqrt(erad*sigmaa/sigmap))) as 'Temperature'

end
```

6.1.6.8 radiative_boundary

```
{  RADIATIVE_BOUNDARY.PDE

   This example demonstrates the implementation of radiative heat loss
   at the boundary of a heat transfer system.

}

title "Axi-symmetric Anisotropic Heatflow, Radiative Boundary"

select
  errlim=1.0e-4

coordinates
  { Define cylindrical coordinates with
    symmetry axis along "Y" }
  ycylinder("R","Z")

variables
  { Define Temp as the system variable,
    with approximate variation range of 1 }
  Temp(1)

definitions
  kr = 1  { radial conductivity }
  kz = 4  { axial conductivity }

  { define a Gaussian source density: }
  source = exp(-(r^2+(z-0.5)^2))

  { define the heat flux: }
  flux = vector(-kr*dr(Temp),-kz*dz(Temp))

initial values
  Temp = 1

equations   { define the heatflow equation: }
  Temp : div(flux) = Source

boundaries                       { define the problem domain }
  Region 1                       { ... only one region }
    start "RAD" (0,0)            { start at bottom on axis and name the boundary }
    natural(temp)= 0.5*temp^4    { specify a T^4 boundary loss }
    line to (0.5,0)              { walk the boundary }
    arc(center=0.5,0.5) angle 180   { a circular outer edge }
    line to (0,1)
    natural(temp)=0              { define a symmetry boundary at the axis }
    line to close

monitors
  elevation(magnitude(2*pi*r*flux)) on "RAD" as "Heat Flow"
  contour(Temp)                  { show contour plots of solution in progress }

plots                            { write these hardcopy files at completion }
  grid(r,z)                      { show final grid }
  contour(Temp)                  { show solution }
  surface(Temp)
  vector(2*pi*r*flux) as "Heat Flow"
  elevation(magnitude(2*pi*r*flux)) on "RAD" as "Heat Flow" print

end
```

6.1.6.9 slider

```
{  SLIDER.PDE

  This problem represents a cross section of a wood-frame sliding window.
      -- submitted by Elizabeth Finleyson, Lawrence Berkeley Labs
}

title
```

```
  "NFRC Wood Slider"

variables
  Temp

definitions
  K = 0.97      {Thermal Conductivity}
  B1 = 1.34     {Film coefficients interior wood}
  B2 = 1.41     {       ''         interior glass}
  B3 = 5.11     {       ''         exterior glass}
  Tin = 70.0    {Ambient Temperature Inside}
  Tout= 0.0     {       ''           Outside}

equations
  Temp : dx(K*dx(Temp)) + dy(K*dy(Temp)) = 0

boundaries
  region 1      {Defines the maximum extent of the system (wood)}
    start(6.813,1.813)
    natural(Temp) = B1*(Tin - Temp)
    line to (6.813,3.001) to (6.344,3.001) to (6.344,3.323)
    line to (6.183,3.323) to (6.183,4.885) to (5.988,4.885)
    line to (5.988,5.104) to (5.678,5.104)
    natural(Temp) = B2*(Tin - Temp)
    line to (5.678,7.604)
    natural(Temp) = 0.0
    line to (5.153,7.604)
    natural(Temp) = B3*(Tout- Temp)
    line to (5.153,5.104) to (5.012,5.104) to (5.012,4.889)
    line to (4.871,4.889) to (4.871,3.323) to (4.248,3.323)
    line to (4.248,2.845) to (3.233,2.845) to (3.233,3.323) to (2.906,3.323)
    line to (2.906,3.001) to (2.250,3.001) to (2.250,2.501) to (1.156,2.501)
    natural(Temp) = 0.0
    line to (1.156,1.813) to close

  {Rigid PVC}
  region 2      K = 1.18
    start(6.516,2.800)
    line to (6.516,2.845) to (6.344,2.845) to (6.344,3.323)
    line to (5.737,3.323) to (5.737,3.278) to (6.017,3.278)
    line to (6.017,2.845) to (5.002,2.845) to (5.002,3.278)
    line to (5.317,3.278) to (5.317,3.323)
    line to (4.248,3.323) to (4.248,2.845) to (3.233,2.845)
    line to (3.233,3.323) to (2.906,3.323) to (2.906,2.845)
    line to (2.547,2.845) to (2.547,2.800) to close

  {Air cavity overlays}
  region 3     K = 0.59
    start(4.293,2.845)
    line to (4.957,2.845) to (4.957,3.278) to (4.293,3.278) to close

  region 4   k = 0.31
    start(2.951,2.800)
    line to (3.188,2.800) to (3.188,3.278) to (2.951,3.278) to close

  region 5   k = 0.51
    start(2.547,2.501)
    line to (3.188,2.501) to (3.188,2.800) to (2.547,2.800) to close

  region 6    k = 0.81
    start(5.002,2.845)
    line to (6.017,2.845) to (6.017,3.278) to (5.002,3.278) to close

  region 7    k = 0.39
    start(5.317,3.278)
    line to (5.737,3.278) to (5.737,3.551) to (5.317,3.551) to close

  region 8    k = 0.31
    start(6.062,2.800)
    line to (6.299,2.800) to (6.299,3.278) to (6.062,3.278) to close

  region 9   k = 0.41
    start(6.062,2.501)
    line to (6.516,2.501) to (6.516,2.800) to (6.062,2.800) to close

{Silicon sealant}
  region 10  k = 2.5
    start(5.133,4.573)
    line to (5.153,4.573) to (5.153,5.104) to (5.133,5.104) to close
```

```
  region 11  k = 2.5
    start(5.678,4.573)
    line to (5.698,4.573) to (5.698,5.104) to (5.678,5.104) to close

{Glass layers}
  region 12  k = 6.93
    start(5.153,4.573)
    line to (5.678,4.573) to (5.678,7.604) to (5.153,7.604) to close

{Eurythane spacer seal}
  region 13  k = 2.5
    start(5.278,4.573)
    line to (5.553,4.573) to (5.553,4.771) to (5.278,4.771) to close

{Spacer}
  region 14  k = 18.44
    start(5.278,4.771)
    line to (5.553,4.771) to (5.553,5.012) to (5.278,5.012) to close

{Gas gap}
  region 15  k = 0.32
    start(5.278,5.012)
    line to (5.553,5.012) to (5.553,7.604) to (5.278,7.604) to close

{Frame fill}
  region 16  k = 0.21
    start(3.188,2.501)
    line to (6.062,2.501) to (6.062,2.800) to (3.188,2.800) to close

{Spacer air gap}
  region 17  k = 0.28
    start(5.133,4.479)
    line to (5.698,4.479) to (5.698,4.573) to (5.133,4.573) to close


monitors
  contour(Temp)

plots
  grid(x,y)
  contour(Temp)
  contour(Temp) zoom(4.6,4.2,1.8,1.8)
  elevation(Temp) from (5.416,1.813) to (5.416,7.604)
  vector((K*(-dx(Temp))),(K*(-dy(Temp)))) as "HEAT FLUX"

end
```

6.1.7 lasers

6.1.7.1 laser_heatflow

```
{ LASER_HEATFLOW.PDE

    This problem shows a complex heatflow application.

    A rod laser is glued inside a cylinder of copper.

    Manufacturing errors allow the rod to move inside the glue, leaving a
    non-uniform glue layer around the rod.  The glue is an insulator, and
    traps heat in the rod.  The copper cylinder is cooled only on a 60-degree
    portion of its outer surface.

    The laser rod has a temperature-dependent conductivity.

    We wish to find the temperature distribution in the laser rod.

    The heat flow equation is

       div(K*grad(Temp)) + Source = 0.

    We will model a cross-section of the cylinder. While this is a cylindrical
    structure, in cross-section there is no implied rotation out of the
    cartesian plane, so the equations are cartesian.

                                -- Submitted by Luis Zapata, LLNL
}
```

```
title "Nd:YAG Rod - End pumped. 200 W/cm3 volume source. 0.005in uropol"

Variables
    temp    { declare "temp" to be the system variable }

definitions
    k = 3                { declare the conductivity parameter for later use }
    krod=39.8/(300+temp){ Nonlinear conductivity in the rod.(W/cm/K) }
    Rod=0.2              { cm Rod radius }
    Qheat=200            { W/cc, heat source in the rod }

    kuropol=.0019    { Uropol conductivity }
    Qu=0             { Volumetric source in the Uropol }
    Ur=0.005         { Uropol annulus thickness in r dim }

    kcopper=3.0      { Copper conductivity }
    Rcu=0.5          { Copper convection surface radius }

    tcoolant=0.      { Edge coolant temperature }
    ASE=0.           { ASE heat/area to apply to edge, heat bar or mount }
    source=0

initial values
    temp = 50        { estimate solution for quicker convergence }

equations            { define the heatflow equation }
    temp : div(k*grad(temp)) + source = 0;

boundaries
    region 1         { the outer boundary defines the copper region }
        k = kcopper
        start (0,-Rcu)
        natural(temp) = -2 * temp       {convection boundary}
            arc(center=0,0) angle 60
        natural(temp) = 0               {insulated boundary}
            arc(center=0,0) angle 300
            arc(center=0,0) to close

    region 2         { next, overlay the Uropol in a central cylinder }
        k = kuropol
        start (0,-Rod-Ur) arc(center=0,0) angle 360

    region 3         { next, overlay the rod on a shifted center }
        k = krod
        Source = Qheat
        start (0,-Rod-Ur/2) arc(center=0,-Ur/2) angle 360

monitors
    grid(x,y) zoom(-8*Ur, -(Rod+8*Ur),16*Ur,16*Ur)
    contour(temp)

plots
    grid(x,y)
    contour (temp)
    contour(temp) zoom(-(Rod+Ur),-(Rod+Ur),2*(Rod+Ur),2*(Rod+Ur))
    contour(temp) zoom(-(Rod+Ur)/4,-(Rod+Ur),(Rod+Ur)/2,(Rod+Ur)/2)
    vector(-k*dx(temp),-k*dy(temp)) as "heat flow"
    surface(temp)

end
```

6.1.7.2 self_focus

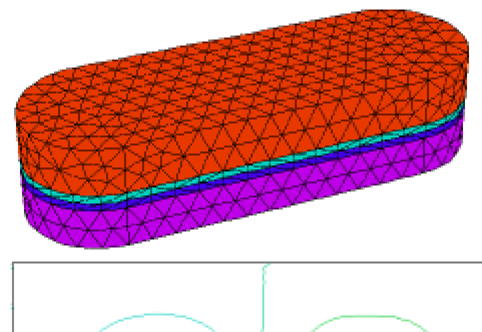
```
{  SELF_FOCUS.PDE

   This problem considers the self-focussing of a laser beam of Gaussian profile.
        -- Submitted by   John Trenholme, LLNL
}

title
  "2D GAUSSIAN BEAM PROFILE"

select
  elevationgrid = 300
  cubic          { use cubic interpolation }

variables
```

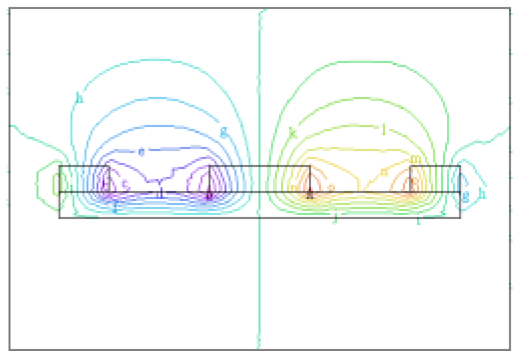
```
  realf (threshold=0.1)       { real (in-phase) part of field envelope }
  imagf (threshold=0.1)       { imaginary (quadrature) part of field envelope }

definitions
  radX = 2                   { X "radius" of beam }
  radY = 2                   { Y "radius" of beam }
  bMax = 2.25                { maximum B integral (= Time)}
  zm = 5                     { zoom-in factor for plots }
  xHi = 7.17 * SQRT( radX * radY)      { size of calculation domain... }
  yHi = 7.17 * SQRT( radX * radY)      { set for field = 0.001 at edge }
  x45 = xHi * 0.7071         { point on boundary at 45 degrees }
  y45 = yHi * 0.7071
  tn =1e-30                  { tiny number to force zero on plot scales }
  power = PI * radX * radY * 2.73 ^ 2 / 8       { analytic integral }
  inten = realf * realf + imagf * imagf   { definition for later use }

time                         { "time" is really B integral }
  0 to bMax by 0.03 * bMax

initial values
  realf = EXP( - ( x / ( radX * 2.73)) ^ 2 - ( y / ( radY * 2.73)) ^ 2)
  imagf = 0

equations          { normalized, low-secular-phase nonlinear propagation }
  realf:    DEL2( imagf) + imagf * ( inten - 1) = -DT( realf)
  imagf:    DEL2( realf) + realf * ( inten - 1) = DT( imagf)

boundaries
  region 1
    start ( 0, 0)            { bump is at center; only do one quadrant }
    natural( realf) = 0       { set slope to zero on boundary }
    natural( imagf) = 0       { if boundary value too big, move boundary out }
    line to ( xHi, 0)
    arc ( center = 0, 0) to ( 0, yHi)
    line to ( 0, 0)
    to close

monitors
  for cycle = 1              { do this every cycle }
    elevation( inten) from ( 0, 0) to ( xHi, 0) as "INTENSITY"
      range( 0, tn)
    contour( inten) as "INTENSITY" zoom ( 0, 0, xHi / zm, yHi / zm)

plots
  for t = starttime          { at the beginning only }
    grid( x, y)
    surface( inten) as "INTENSITY" range( 0, tn) viewpoint( 1000, 200, 40)
    elevation( LOG10( inten)) from ( 0, 0) to ( xHi, 0) as "LOG10 INTENSITY"

  for t = endtime            { at the end only }
    grid( x, y)
    grid( x, y) zoom ( 0, 0, xHi / zm, yHi / zm)
    surface( inten) as "INTENSITY" range( 0, tn) viewpoint( 1000, 200, 40)
      zoom ( 0, 0, xHi / zm, yHi / zm)
    elevation( LOG10( inten)) from ( 0, 0) to ( xHi, 0) as "LOG10 INTENSITY"

  for t = starttime by ( endtime - starttime) / 5 to endtime    { snapshots }
    elevation( ARCTAN( imagf / realf) * 180 / PI) from ( 0, 0)
      to ( xHi / zm, 0) as "PHASE (DEGREES)"
    elevation( inten) from ( 0, 0) to ( xHi / zm, 0) as "INTENSITY"
      range( 0, tn)

histories
  history( inten) at ( 0, 0) ( 0.01 * xHi, 0) ( 0.03 * xHi, 0) ( 0.1 * xHi, 0)
    ( 0.3 * xHi, 0) ( xHi, 0) ( x45, y45) as "INTENSITY" print

  history( realf) at ( 0, 0) ( 0.01 * xHi, 0) ( 0.03 * xHi, 0) ( 0.1 * xHi, 0)
    ( 0.3 * xHi, 0) as "IN-PHASE FIELD"

  history( imagf) at ( 0, 0) ( 0.01 * xHi, 0) ( 0.03 * xHi, 0) ( 0.1 * xHi, 0)
    ( 0.3 * xHi, 0) as "QUADRATURE FIELD"

  history( ARCTAN( imagf / realf) * 180 / PI) at ( 0, 0) ( 0.01 * xHi, 0)
    ( 0.03 * xHi, 0) ( 0.1 * xHi, 0) ( 0.3 * xHi, 0) as "PHASE (DEGREES)"

  history( MIN( ( ABS( inten - 0.33)) ^ ( -0.75), 1)) at ( 0, 0)
    range ( 0.045, 1) as "( INTENSITY - 0.33) ^ -0.75"  { goes linearly to 0}

  history( ABS( INTEGRAL( inten) / power - 1)) as "POWER CHANGE (EXACT = 0)"
```

```
end
```

6.1.8 magnetism

6.1.8.1 3d_magnetron





```
{ 3D_MAGNETRON.PDE

  MODEL OF A GENERIC MAGNETRON IN 3D

  The development of this model is
  described in the Magnetostatics 230 chapter
  of the Electromagnetic Applications 214
  section.

}

TITLE 'Oval Magnet '

COORDINATES
  CARTESIAN3

SELECT
  ngrid=25
  alias(x) = "X(cm)"
  alias(y) = "Y(cm)"
  alias(z) = "Z(cm)"

VARIABLES
  Ax,Ay     { assume Az is zero! }

DEFINITIONS
  MuMag=1.0  ! Permeabilities:
  MuAir=1.0
  MuSST=1000
  MuTarget=1.0
  Mu=MuAir  ! default to Air
  MzMag = 10000    ! permanent magnet strength
  Mz = 0         ! global magnetization variable
  Nx = vector(0,Mz,0)
  Ny = vector(-Mz,0,0)

  B = curl(Ax,Ay,0)  ! magnetic induction vector
  Bxx= -dz(Ay)
  Byy= dz(Ax)  ! unfortunately, "By" is a reserved word.
  Bzz= dx(Ay)-dy(Ax)

EQUATIONS
  Ax:   div(grad(Ax)/mu+Nx) = 0
  Ay:   div(grad(Ay)/mu+Ny) = 0

EXTRUSION
   SURFACE "Boundary Bottom"     Z=-5
   SURFACE "Magnet Plate Bottom"    Z=0
     LAYER "Magnet Plate"
   SURFACE "Magnet Plate Top"    Z=1
     LAYER "Magnet"
   SURFACE "Magnet Top"      Z=2
   SURFACE "Boundary Top"     Z=8

BOUNDARIES
  Surface "boundary bottom" value (Ax)=0  value(Ay)=0
  Surface "boundary top" value (Ax)=0 value(Ay)=0

  REGION 1     {Air bounded by conductive box }
      START (20,-10)
      value(Ax)=0  value(Ay)=0
      arc(center=20,0) ANGLE=180
      Line TO (-20,10)
      arc(center=-20,0) ANGLE=180
        LINE TO CLOSE

  LIMITED REGION 2     { Magnet Plate }
      LAYER "Magnet Plate" Mu=MuSST
      LAYER "Magnet" Mu=MuMag  Mz = MzMag
      START (20,-8)
```

```
        ARC(center=20,0) ANGLE=180
        LINE TO (-20,8)
        ARC(center=-20,0) ANGLE=180
        LINE TO CLOSE

    LIMITED REGION 3      { Inner Gap  }
      LAYER "Magnet"
      START (20,-6)
        ARC(center=20,0) ANGLE=180
        LINE TO (-20,6)
        ARC(center=-20,0) ANGLE=180
        LINE TO CLOSE

    LIMITED REGION 4      {Inner Magnet }
      LAYER "Magnet" Mu=MuMag  Mz = -MzMag
      START (20,-2)
        ARC(center=20,0) ANGLE=180
        LINE TO (-20,2)
        ARC(center=-20,0) ANGLE=180
        LINE TO CLOSE

MONITORS
  grid(y,z) on x=0
  grid(x,z) on  y=0
  grid(x,y) on  z=1.01
  contour(Ax) on x=0
  contour(Ay) on y=0

PLOTS
  grid(y,z) on x=0
  grid(x,z) on y=0
  grid(x,y) on z=1.01
  contour(Ax) on x=0
  contour(Ay) on y=0
  vector(Bxx,Byy) on z=2.01 norm
  vector(Byy,Bzz) on x=0 norm
  vector(Bxx,Bzz) on y=4 norm
  contour(magnitude(Bxx,Byy,Bzz)) on z=2

END
```

6.1.8.2 3d_vector_magnetron

```
{ 3D_VECTOR_MAGNETRON

  MODEL OF A GENERIC MAGNETRON IN 3D USING VECTOR VARIABLES
  This is a modification of 3D_MAGNETRON.PDE 346.

  The development of this model is described in the Magnetostatics 230 chapter of the
  Electromagnetic Applications 214 section.

}

TITLE 'Oval Magnet'

COORDINATES
  CARTESIAN3

SELECT
  ngrid=25
  alias(x) = "X(cm)"
  alias(y) = "Y(cm)"
  alias(z) = "Z(cm)"

VARIABLES
  A = vector (Ax,Ay)        { assume Az is zero! }

DEFINITIONS
  MuMag=1.0 ! Permeabilities:
  MuAir=1.0
  MuSST=1000
  MuTarget=1.0
  Mu=MuAir  ! default to Air

  MzMag = 10000    ! permanent magnet strength
  Mx=0   My=0   Mz=0
  M = vector(Mx,My,Mz)      ! global magnetization variable
```

```
  N = tensor((0,Mz,0),(-Mz,0,0),(0,0,0))

  B = curl(Ax,Ay,0)  ! magnetic induction vector
  Bxx= xcomp(B)
  Byy= ycomp(B) ! unfortunately, "By" is a reserved word.
  Bzz= zcomp(B)

EQUATIONS
  A:  div((grad(A)+N)/mu) = 0

EXTRUSION
  SURFACE "Boundary Bottom"     Z=-5
  SURFACE "Magnet Plate Bottom" Z=0
  LAYER "Magnet Plate"
  SURFACE "Magnet Plate Top"    Z=1
  LAYER "Magnet"
  SURFACE "Magnet Top"       Z=2
  SURFACE "Boundary Top"     Z=8

BOUNDARIES
  Surface "boundary bottom" value (Ax)=0  value(Ay)=0
  Surface "boundary top"    value (Ax)=0  value(Ay)=0
    REGION 1      {Air bounded by conductive box }
      START (20,-10)
      value(A)=vector(0,0,0)
        ARC(center=20,0) angle=180
        LINE TO (-20,10)
        ARC(center=-20,0) angle=180
        LINE TO CLOSE

    LIMITED REGION 2     { Magnet Plate }
      LAYER "Magnet Plate" Mu=MuSST
      LAYER "Magnet" Mu=MuMag  Mz = MzMag
      START (20,-8)
        ARC(center=20,0) angle=180
        LINE TO (-20,8)
        ARC(center=-20,0) angle=180
        LINE TO CLOSE

    LIMITED REGION 3     { Inner Gap  }
      LAYER "Magnet"
      START (20,-6)
        ARC(center=20,0) angle=180
        LINE TO (-20,6)
        ARC(center=-20,0) angle=180
        LINE TO CLOSE

    LIMITED REGION 4     {Inner Magnet }
      LAYER "Magnet" Mu=MuMag  Mz = -MzMag
      START (20,-2)
        ARC(center=20,0) angle=180
        LINE TO (-20,2)
        ARC(center=-20,0) angle=180
        LINE TO CLOSE

MONITORS
  grid(y,z) on x=0
  grid(x,z) on y=0
  grid(x,y) on z=1.01
  contour(Ax) on x=0
  contour(Ay) on y=0

PLOTS
  grid(y,z) on x=0
  grid(x,z) on y=0
  grid(x,y) on z=1.01
  contour(Ax) on x=0
  contour(Ay) on y=0
  vector(Bxx,Byy) on z=2.01  norm
  vector(Byy,Bzz) on x=0  norm
  vector(Bxx,Bzz) on y=4  norm
  contour(magnitude(Bxx,Byy,Bzz)) on z=2

END
```

6.1.8.3 helmholtz_coil

```
{  HELMHOLTZ_COIL.PDE

   This example shows the calculation of magnetic fields in a Helmholtz coil.

   -- submitted by Bill Hallbert, Honeywell
}

TITLE 'Helmholtz Coil'

COORDINATES cartesian3

VARIABLES    Ax, Ay             {Magnetic Vector Potential Components}

DEFINITIONS

    { Defining parameters of the Coil }
    coil_current=200            {Amps in 1 turn}
    Lsep=6.89                   {Layer separation - cm}
    Cthick=2.36                 {Coil thickness - cm}

    { Regional Current Definition }
    CurrentControl=1
    Current=CurrentControl*coil_current

    { Circulating Current Density in Coil }
    J0=Current/Cthick^2         {A/cm^2}
    theta=atan2(y,x)
    Jx=-J0*sin(theta)
    Jy=J0*cos(theta)

    { Magnetic Permeability }
    m0=4*3.1415e-2              {dynes/A^2}

    { Coil Radii }
    Rcoil_inner=Lsep            {cm}
    Rcoil_outer=Lsep+Cthick     {cm}
    Rmax=1.5*Lsep               {cm}

    { Z Surfaces }
    za=2*Lsep                   {cm}
    zb=za+Cthick                {cm}
    zc=zb+Lsep                  {cm}
    zd=zc+Cthick                {cm}
    zmax=zd+2*Lsep              {cm}
    zmiddle=(zd+za)/2           {cm}

    { Magnetic Field }
    H=curl(Ax ,Ay ,0)/m0        {AT}
    Hxx=-dz(Ay)/m0
    Hyy=dz(Ax)/m0
    Hzz=(dx(Ay)-dy(Ax))/m0

    { Magnetic Field Error }
    Hzvec=val(Hzz,0,0,zmiddle)
    H_Error=(magnitude(H)-Hzvec)/Hzvec*100

EQUATIONS
    Ax:    div(grad(Ax))/m0 + Jx = 0
    Ay:    div(grad(Ay))/m0 + Jy = 0

EXTRUSION
    Surface 'Bottom'     z = 0
        Layer 'Bottom_Air'
    Surface 'Coil1B'     z = za
        Layer 'Coil1Cu'
    Surface 'Coil1T'     z = zb
        Layer 'Middle_Air'
    Surface 'Coil2B'     z = zc
        Layer 'Coil2Cu'
    Surface 'Coil2T'     z = zd
        Layer 'Top_Air'
    Surface 'Top'        z = zmax

BOUNDARIES
    Surface "Bottom" value (Ax)=0  value (Ay)=0
    Surface "Top"    value (Ax)=0  value (Ay)=0
```

```
    REGION 1 'Air'
        CurrentControl=0
        start(Rmax,0) arc(center=0,0) angle =360

    LIMITED REGION 2 'Outer Coil'
        CurrentControl=1
        Layer 'Coil1Cu'
        Layer 'Coil2Cu'
        start(Rcoil_outer,0) arc(center=0,0) angle =360

    LIMITED REGION 3 'Inner Coil'
        mesh_spacing = Rcoil_inner/10
        CurrentControl=0
        Layer 'Coil1Cu'
        Layer 'Coil2Cu'
        start(Rcoil_inner,0) arc(center=0,0) angle =360

MONITORS
    grid(y,z) on x=0
    grid(x,y) on surface 'Coil1T'
    contour(Ax) on x=0

PLOTS
    grid(y,z) on x=0
    grid(x,y) on surface 'Coil1T'
    contour(Ax) on x=0
    vector(Hxx,Hyy) on surface 'Coil1T'  norm
    vector(Hyy,Hzz) on x=0  norm
    contour(magnitude(H)) on z=zmiddle
    contour(magnitude(H)) on x=0
    contour(H_Error) on Layer 'Middle_Air' on x=0

END
```

6.1.8.4 magnet_coil

```
{ MAGNET_COIL.PDE

  AXI-SYMMETRIC MAGNETIC FIELDS

  This example considers the problem of determining the magnetic vector
  potential A around a coil.

  According to Maxwell's equations,
        curl H = J
        div B = 0
        B = mu*H

  where B is the manetic flux density
        H is the magnetic field strength
        J is the electric current density
  and  mu is the magnetic permeability of the material.

  The magnetic vector potential A is related to B by
        B = curl A
  therefore
        curl( (1/mu)*curl A ) = J

  This equation is usually supplmented with the Coulomb Gauge condition
        div A = 0.

  In the axisymmetric case, the current is assumed to flow only in the
  azimuthal direction, and only the azimuthal component of the vector
  potential is present.  Henceforth, we will simply refer to this component as A.

  The Coulomb Gauge is identically satisfied, and the PDE to be solved in this
  model takes the form
        curl((1/mu)*curl (A)) = J(x,y)     in the domain
                           A  = g(x,y)     on the boundary.

  The magnetic induction B takes the simple form
        B = (-dz(A), 0, dr(A)+A/r)

  and the magnetic field is given by
        H = (-dz(A)/mu, 0, (dr(A)+A/r)/mu)

  Expanding the equation in cylindrical geometry results in the final equation,
        dz(dz(A)/mu) + dr((dr(A)+A/r)/mu) = -J
```

```
  The interpretation of the natural boundary condition becomes
        Natural(A) = n X H

  where n is the outward surface-normal unit vector.

  Across boundaries between regions of different material properties, the
  continuity of (n X H) assumed by the Galerkin solver implies that the
  tangential component of H is continuous, as required by the physics.


  In this simple test problem, we consider a circular coil whose axis of
  rotation lies along the X-axis. We bound the coil by a distant spherical
  surface at which we specify a boundary condition (n X H) = 0.
  At the axis, we use a Dirichlet boundary condition A=0.

  The source J is zero everywhere except in the coil, where it is defined
  arbitrarily as "10".  The user should verify that the prescribed values
  of J are dimensionally consistent with the units of his own problem.

}

title 'AXI-SYMMETRIC MAGNETIC FIELD'

coordinates
    { Cylindrical coordinates, with cylinder axis along Cartesian X direction }
    xcylinder(Z,R)

variables
    Aphi        { the azimuthal component of the  vector potential }

definitions
    mu = 1              { the permeability }
    rmu = 1/mu
    J = 0               { the source defaults to zero }
    current = 10        { the source value in the coil }
    Bz = dr(r*Aphi)/r

initial values
    Aphi = 2            { unimportant unless mu varies with H }

equations
    { FlexPDE expands CURL in proper coordinates }
    Aphi : curl(rmu*curl(Aphi)) = J

boundaries
    region 1
      start(-10,0)
      value(Aphi) = 0       { specify A=0 along axis }
        line to (10,0)
      natural(Aphi) = 0     { H<dot>n = 0 on distant sphere }
        arc(center=0,0) angle 180 to close

    region 2
      J = current           { override source value in the coil }
      start (-0.25,1)
        line to (0.25,1) to (0.25,1.5) to (-0.25,1.5) to close

monitors
    contour(Bz) zoom(-2,0,4,4) as 'FLUX DENSITY B'
    contour(Aphi) as 'Potential'

plots
    grid(z,r)
    contour(Bz)  as 'FLUX DENSITY B'
    contour(Bz) zoom(-2,0,4,4)  as 'FLUX DENSITY B'
    elevation(Aphi,dr(Aphi),Aphi/r,dr(Aphi)+Aphi/r,Aphi+r*dr(Aphi))
        from (0,0) to (0,1) as 'Bz'
    vector(dr(Aphi)+Aphi/r,-dz(Aphi)) as 'FLUX DENSITY B'
    vector(dr(Aphi)+Aphi/r,-dz(Aphi)) zoom(-2,0,4,4) as 'FLUX DENSITY B'
    contour(Aphi)  as 'MAGNETIC POTENTIAL'
    contour(Aphi) zoom(-2,0,4,4)  as 'MAGNETIC POTENTIAL'
    surface(Aphi)  as 'MAGNETIC POTENTIAL'  viewpoint (-1,1,30)

end
```

6.1.8.5 permanent_magnet

```
{ PERMANENT_MAGNET.PDE

  This example demonstrates the implementation of permanant magnets in magnetic field problems.

  FlexPDE integrates second-order derivative terms by parts, which creates surface integral
  terms at cell boundaries.
  By including magnetization vectors inside the definition of H, these surface terms correctly
  model the effect of magnetization through jump terms at boundaries.
  If the magnetization terms are listed separately from H, they will be seen as piecewise
  constant in space, and their derivatives will be deleted.

  See the Electromagnetic Applications 214 section for further discussion.

}

title 'A PERMANENT-MAGNET PROBLEM'

Variables
    A  { z-component of Vector Magnetic Potential }

Definitions
    mu
    S = 0               { current density }
    Px = 0              { Magnetization components }
    Py = 0
    P = vector(Px,Py)  { Magnetization vector }
    H = (curl(A)-P)/mu { Magnetic field }
    y0 = 8              { Size parameter }

Initial values
     A = 0

Equations
     A : curl(H) + S = 0

Boundaries
    Region 1
      mu = 1
      start(-40,0)
      natural(A) = 0 line to (80,0)
      value(A) = 0   line to (80,80) to (-40,80) to close

    Region 2
      mu = 5000
      start(0,0)
      line to (15,0) to (15,20) to (30,20) to (30,y0) to (40,y0) to (40,40)
                to (0,40) to close

    Region 3    { the permanent magnet }
      mu = 1
      Py = 10
      start (0,0) line to (15,0) to (15,10) to (0,10) to close

Monitors
    contour(A)

Plots
    grid(x,y)
    vector(dy(A),-dx(A)) as 'FLUX DENSITY B'
    vector((dy(A)-Px)/mu, (-dx(A)-Py)/mu) as 'MAGNETIC FIELD H'
    contour(A)  as 'Az MAGNETIC POTENTIAL'
    surface(A)  as 'Az MAGNETIC POTENTIAL'

End
```

6.1.8.6 saturation

```
{ SATURATION.PDE

  A NONLINEAR MAGNETOSTATIC PROBLEM

  This example considers the problem of determining the magnetic vector
  potential A in a cyclotron magnet.

  The problem domain consists of
```

```
    1) a ferromagnetic medium - the magnet core,
    2) the surrounding air medium,
    3) a current-carrying copper coil.

  According to Maxwell's equations,
        curl H = J                      (1)
        div B = 0                       (2)
  with
        B = mu*H
  where B is the magnetic flux density
        H is the magnetic field strength
        J is the electric current density
  and  mu is the magnetic permeability of the material.

  Maxwell's equations can be satisfied if we introduce a magnetic vector
  potential A such that
        B = curl A
  therefore
        curl( (curl A)/mu ) = J

  This equation is usually supplemented with the Coulomb Gauge condition
        div A = 0.

  In most common 2D applications, magnet designers assume either
        1) that the magnet is sufficiently long in the z direction or
        2) that the magnet is axi-symmetric.

  In the first instance the current is assumed to flow parallel to the z axis,
  and in the latter it flows in the azimuthal direction.  Under these conditions,
  only the z or the azimuthal component of A is present. (Henceforth, we will
  simply refer to this component as A).

  In the Cartesian case, the magnetic induction B takes the simple form,
        B = (dy(A), -dx(A), 0)
  and the magnetic field is given by
        H = (dy(A)/mu, -dx(A)/mu, 0).

  We can integrate equation (1) over the problem domain using the curl analog
  of the Divergence Theorem, giving
        Integral(curl(H))dV = Integral(n x H)dS
  where dS is a differential surface area on the bounding surface of any region,
  and n is the outward surface normal unit vector.

  Across interior boundaries between regions of different material properties,
  FlexPDE assumes cancellation of the surface integrals from the two sides of the
  boundary.  This implies continuity of (n X H).

  At exterior boundaries, the same theorem defines the natural boundary condition
  to be the value of (n x H).

  For the present example, let us define the permeability mu by the
  expression
        mu =  1                                                     in the air and the coil
        mu =  mumax/(1+C*grad(A)^2) + mumin     in the core
  where C = 0.05 gives a behaviour similar to transformer steel.

  We assume a symmetry plane along the X-axis, and impose the boundary value
  A = 0 along the remaining sides.

  The core consists of a "C"-shaped region enclosing a rectangular coil region.

  The source J is zero everywhere except in the coil, where it is defined by
               J = - (4*pi/10)*amps/area

  Note:
  This example uses scaled units.  It is important for the user to validate
  the dimensional consistency of his formulation.
}
```

```
Title "A MAGNETOSTATIC PROBLEM"

Select
    { Since the nonlinearity in this problem is driven
      by the GRADIENT of the system variable, we
require
      a more accurate resolution of the solution: }
    errlim = 1e-4

Variables
    A

Definitions
    rmu = 1
    rmu0 = 1
    mu0core = 5000
    mu1core = 200
    mucore = mu0core/(1+0.05*grad(A)^2) + mu1core
    rmucore = 1/mucore
    S = 0
    current = 2
    y0 = 8

Initial Value
    { In nonlinear problems, a good starting value
      is sometimes essential for convergence }
    A = current*(400-(x-20)^2-(y-20)^2)

Equations
    A : curl(rmu*curl(A)) = S

Boundaries
   Region 1             { The IRON core }
      rmu = rmucore                 rmu0 = 1/mu0core
      start(0,0)
      natural(A) = 0          line to (40,0)
      value(A) = 0   line to (40,40) to (0,40) to close

   Region 2             { The AIR gap }
      rmu = 1
      start (15,0) line to (40,0) to (40,y0) to (32,y0)
      arc (center=32,y0+2) to (30,y0+2)
      { short boundary segments force finer gridding: }
      line to (30,19.5) to (30,20) to (29.5,20)
      to (15.5,20) to (15,20) to close

   Region 3             { The COIL }
      S = current
      rmu = 1
      start (15,12) line to (30,12) to (30,20) to (15,20) to close

Monitors
   contour(A)

Plots
   grid(x,y)
   vector(dy(A),-dx(A)) as "FLUX DENSITY B"
   vector(dy(A)*rmu, -dx(A)*rmu) as "MAGNETIC FIELD H"
   contour(A)  as "Az MAGNETIC POTENTIAL"
   surface(A)  as "Az MAGNETIC POTENTIAL"
   contour(rmu/rmu0) painted as "Saturation: mu0/mu"

End
```

6.1.8.7 vector_helmholtz_coil

```
{ VECTOR_HELMHOLTZ_COIL.PDE

  This example is a revision of HELMHOLTZ_COIL.PDE 349 using vector variables.

}

TITLE 'Vector Helmholtz Coil'

COORDINATES cartesian3

VARIABLES    A = vector(Ax, Ay)      {Magnetic Vector Potential Components}
```

```
DEFINITIONS

    { Defining parameters of the Coil }
    coil_current=200                  {Amps in 1 turn}
    Lsep=6.89                         {Layer separation - cm}
    Cthick=2.36                       {Coil thickness - cm}

    { Regional Current Definition }
    CurrentControl=1
    Current=CurrentControl*coil_current

    { Circulating Current Density in Coil }
    J0=Current/Cthick^2          {A/cm^2}
    theta=atan2(y,x)
    Jx=-J0*sin(theta)
    Jy=J0*cos(theta)

    { Magnetic Permeability }
    m0=4*3.1415e-2               {dynes/A^2}

    { Coil Radii }
    Rcoil_inner=Lsep             {cm}
    Rcoil_outer=Lsep+Cthick      {cm}
    Rmax=1.5*Lsep                {cm}

    { Z Surfaces }
    za=2*Lsep                    {cm}
    zb=za+Cthick                 {cm}
    zc=zb+Lsep                   {cm}
    zd=zc+Cthick                 {cm}
    zmax=zd+2*Lsep               {cm}
    zmiddle=(zd+za)/2            {cm}

    { Magnetic Field }
    H=curl(A)/m0               {AT}
    Hxx = Xcomp(H)
    Hyy = Ycomp(H)
    Hzz = Zcomp(H)

    { Magnetic Field Error }
    Hzvec=val(Hzz,0,0,zmiddle)
    H_Error=(magnitude(H)-Hzvec)/Hzvec*100

EQUATIONS

    A:    div(grad(A))/m0 + vector(Jx,Jy,0) = 0

EXTRUSION

    Surface 'Bottom'     z = 0
        Layer 'Bottom_Air'
    Surface 'Coil1B'     z = za
        Layer 'Coil1CU'
    Surface 'Coil1T'     z = zb
        Layer 'Middle_Air'
    Surface 'Coil2B'    z = zc
        Layer 'Coil2CU'
    Surface 'Coil2T'    z = zd
        Layer 'Top_Air'
    Surface 'Top'         z = zmax

BOUNDARIES
    Surface "Bottom" value (Ax)=0  value (Ay)=0
    Surface "Top"       value (Ax)=0  value (Ay)=0

    REGION 1 'Air'
        CurrentControl=0
        start(Rmax,0) arc(center=0,0) angle =360

    LIMITED REGION 2 'Outer Coil'
        CurrentControl=1
        Layer 'Coil1Cu'
        Layer 'Coil2Cu'
        start(Rcoil_outer,0) arc(center=0,0) angle =360

    LIMITED REGION 3 'Inner Coil'
        mesh_spacing = Rcoil_inner/10
        CurrentControl=0
        Layer 'Coil1Cu'
```

```
        Layer 'Coil2Cu'
        start(Rcoil_inner,0) arc(center=0,0) angle =360

MONITORS
    grid(y,z) on x=0
    grid(x,y) on surface 'Coil1T'
    contour(Ax) on x=0

PLOTS
    grid(y,z) on x=0
    grid(x,y) on surface 'Coil1T'
    contour(Ax) on x=0
    vector(Hxx,Hyy) on surface 'Coil1T'  norm
    vector(Hyy,Hzz) on x=0  norm
    contour(magnitude(H)) on z=zmiddle
    contour(magnitude(H)) on x=0
    contour(H_Error) on Layer 'Middle_Air' on x=0

END
```

6.1.8.8 vector_magnet_coil

```
{ VECTOR_MAGNET_COIL.PDE

  AXI-SYMMETRIC MAGNETIC FIELDS

  This example is a modification of MAGNET_COIL.PDE 350 using vector variables.
  See that example for discussion of the problem formulation.

}

Title 'AXI-SYMMETRIC MAGNETIC FIELD - Vector Variables'

Coordinates
    { Cylindrical coordinates, with cylinder axis along Cartesian X direction }
    xcylinder(Z,R)

Variables
    A = vector(0,0,Aphi) { the azimuthal component of the  vector potential }

Definitions
    mu = 1                      { the permeability }
    rmu = 1/mu
    current = 0                 { the source defaults to zero }
    J = vector(0,0,current)
    Bz = dr(r*Aphi)/r

Initial Values
    Aphi = 2                    { unimportant unless mu varies with H }

Equations
    { FlexPDE expands CURL in proper coordinates }
    A : curl(rmu*curl(A)) = J

Boundaries
    Region 1
      start(-10,0)
      value(Aphi) = 0           { specify A=0 along axis }
        line to (10,0)
      natural(Aphi) = 0         { H<dot>n = 0 on distant sphere }
        arc(center=0,0) angle 180 to close

    Region 2
      current = 10              { override source value in the coil }
      start (-0.25,1)
        line to (0.25,1) to (0.25,1.5) to (-0.25,1.5) to close

Monitors
    contour(Bz) zoom(-2,0,4,4) as 'FLUX DENSITY B'
    contour(Aphi) as 'Potential'

Plots
    grid(z,r)
    contour(Bz)  as 'FLUX DENSITY B'
    contour(Bz) zoom(-2,0,4,4)  as 'FLUX DENSITY B'
    elevation(Aphi,dr(Aphi),Aphi/r,dr(Aphi)+Aphi/r,Aphi+r*dr(Aphi))
        from (0,0) to (0,1) as 'Bz'
    vector(dr(Aphi)+Aphi/r,-dz(Aphi)) as 'FLUX DENSITY B'
```

```
    vector(dr(Aphi)+Aphi/r,-dz(Aphi)) zoom(-2,0,4,4) as 'FLUX DENSITY B'
    contour(Aphi)  as 'MAGNETIC POTENTIAL'
    contour(Aphi) zoom(-2,0,4,4)  as 'MAGNETIC POTENTIAL'
    surface(Aphi)  as 'MAGNETIC POTENTIAL'  viewpoint (-1,1,30)

End
```

6.1.9 misc

6.1.9.1 diffusion

```
{ DIFFUSION.PDE

  This problem considers the thermally driven diffusion of a dopant into
  a solid from a constant source.  Parameters have been chosen to be those
  typically encountered in semiconductor diffusion.

    surface concentration = 1.8e20 atoms/cm^2
    diffusion coefficient = 3.0e-15 cm^2/sec

  The natural tendency in this type of problem is to start with zero
  concentration in the material, and a fixed value on the boundary.  This
  implies an infinite curvature at the boundary, and an infinite transport
  velocity of the diffusing particles.  It also generates over-shoot
  in the solution, because the Finite-Element Method tries to fit a step
  function with quadratics.

  A better formulation is to program a large input flux, representative of
  the rate at which dopant can actually cross the boundary, (or approximately
  the molecular velocity times the concentration deficiency at the boundary).

  Here we use a masked source, in order to generate a 2-dimensional pattern.
  This causes the result to lag a bit behind the analytical Plane-diffusion
  result at late times.

}

title
  'Masked Diffusion'

variables
  u(threshold=0.1)

definitions
  concs = 1.8e8             { surface concentration  atom/micron^3}
  D = 1.1e-2                { diffusivity micron^2/hr}
  conc = concs*u
  cexact1d = concs*erfc(x/(2*sqrt(D*t)))
  uexact1d = erfc(x/(2*sqrt(D*t)))
                            { masked surface flux multiplier }
  M = 10*upulse(y-0.3,y-0.7)

initial values
  u = 0

equations
  u : div(D*grad(u)) = dt(u)

boundaries
  region 1
    start(0,0)
    natural(u) = 0      line to (1,0)
    natural(u) = 0      line to (1,1)
    natural(u) = 0      line to (0,1)
    natural(u) = M*(1-u)  line to close

  feature               { a "gridding feature" to help localize the activity }
    start (0.02,0.3) line to (0.02,0.7)

time 0 to 1 by 0.001

plots
  for t=1e-5 1e-4 1e-3 1e-2 0.05 by 0.05 to 0.2 by 0.1 to endtime
    contour(u)
    surface(u)
    elevation(u,uexact1d) from (0,0.5) to (1,0.5)
    elevation(u-uexact1d) from (0,0.5) to (1,0.5)
```

```
histories
  history(u) at (0.05,0.5) (0.1,0.5) (0.15,0.5) (0.2,0.5)

end
```

6.1.9.2 minimal_surface

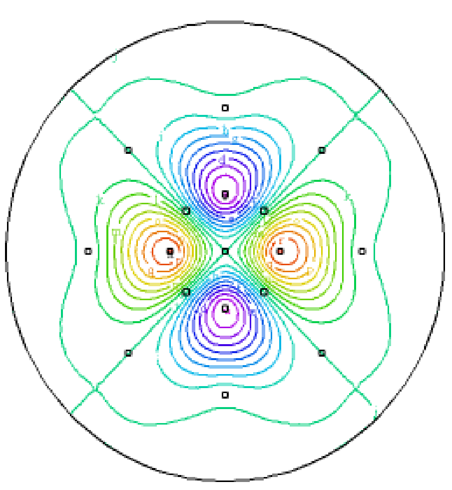
```
{ MINIMUM_SURFACE.PDE

  This example shows the application of FlexPDE to the non-linear problem
  of surface tension or "minimal surface".

  The surface area of an infinitesimal rectangular patch of an arbitrary
  surface
        U = U(x,y)

  is (by the Pythagorean theorem)
        dA = dx*dy*sqrt[1 + (dU/dx)^2 + (dU/dy)^2],
  where dx and dy are the projections of the patch in the X-Y plane.

  The total surface area of a function U(x,y) over a domain is then
        A = integral(dx*dy*sqrt[1 + dx(U)^2 + dy(U)^2])

  For the function U to have minimal surface area, it must satisfy the
  Euler equation
        dx(dF/dUx) + dy(dF/dUy) - dF/dU = 0
  where
        F = sqrt[1 + (dU/dx)^2 + (dU/dy)^2]
        dF/dUx = (dU/dx)/F
        dF/dUy = (dU/dy)/F
        dF/dU  = 0

  The equation for the minimizing surface is therefore (in FlexPDE notation):
        dx((1/F)*dx(U)) + dy((1/F)*dy(U)) = 0

  This is analogous to a heatflow problem
        div(K*grad(T)) = 0
  where the conductivity has the value
        K = 1/F

  This is a highly nonlinear problem, in that the conductivity, K, becomes
  small in regions of high gradient, which tends to increase the gradient
  even more.

  In the present example, we stretch a
  soap-bubble across a square bent wire
  frame, in which the first quadrant of
  the boundary has been bent inward
  and raised up.
}
title "MINIMAL SURFACE"

variables
    u

definitions
    size = 6
    a
    pressure = 0
    r = sqrt(x^2+y^2)

equations
    u : div(a*grad(u)) + pressure = 0

boundaries
    region 1
      a = 1/sqrt(1+grad(u)^2)
      start(-size,-size)
        value(u)=0          line to (size ,-size) to (size,0)
        value(u) = size-r   line to (size/2,0)
        value(u) = size/2   arc(center=size/2,size/2) angle -90
        value(u) = size-r   line to (0,size)
        value(u) = 0        line to (-size,size)
        to close

monitors
    contour(u)
```

```
plots
    grid(x,y)
    contour(u)
    surface(u)

end
```

6.1.9.3 surface_fit

```
{ SURFACE_FIT.PDE

  This problem illustrates the use of FlexPDE in a data fitting
  application.

  THE NUMERICAL SOLUTION OF THE BIHARMONIC EQUATION WITH A DISCONTINUOUS
  LINEAR SOURCE TERM USING FlexPDE.

  STATEMENT OF THE PROBLEM:

  Find the solution U of the fourth order elliptic PDE

        (dxx + dyy)(dxx + dyy) (U) = -beta*(U - C)      in O,       (1)

  where in the usual FlexPDE notation, dxx indicates 2nd partial derivative
  with respect to x, and where O is a given connected domain.  Equation (1)
  arises from the minimization of the strain energy function of a thin plate
  which is constrained to nearly pass thru a given set of discrete set of
  points specified by C and beta.    Namely,  a given  set of n data values
  [C(i)] is assigned at locations [(x(i), y(i))], i=1,..n ,
  and the factor beta has its support only at the locations (x(i), y(i)).

  Along with equation (1), we must prescribe a set of boundary conditions
  involving U and its derivatives which must be satisfied everywhere on the
  domain boundary.

}

title " The Biharmonic Equation in Surface Fitting Designs and Visualization"

select cubic

variables
    U
    V

definitions
    eps = .001
    beta0 = 1.e7
    beta = 0.0
    a = 1/sqrt(2.)
    two = 2.5
    b = two*a

    xbox = array (0, 1, -1, 0,  0, a, -a, a, -a,  two, -two, 0, 0, b, -b, b, -b )
    ybox = array (0, 0,  0, 1, -1, a, -a, -a, a,  0, 0, two, -two, b, -b, -b, b )
```

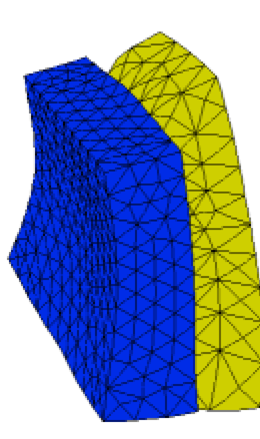
```
    xi = .05   eta = .05
    r0 =  x*x + y*y
    C = exp(-r0/1.)*sin(pi*((x^2-y^2)/64.))

initial values
    U =  0
    V =  .001

equations
    U:  del2(U) = V
    V:  del2(V) = -beta*(U-C)

boundaries
    region 1
      start (-4,0)
      value(U) = C    value(V) = 0.
      arc(center=0.,0.)     angle -360     to close

   region 2  beta = beta0
     repeat i=1 to 17
         start (xbox[i]-xi,ybox[i]-eta)
         line to (xbox[i]+xi,ybox[i]-eta)
         to (xbox[i]+xi,ybox[i]+eta)
         to (xbox[i]-xi,ybox[i]+eta) to close
     endrepeat

monitors
    contour(U)
    contour(C)
    contour(C-U)   as "Error C - U"

plots
    contour (U)  as "Potential"
    surface(U) as "Potential"
    surface(C)   as " Expected Surface"
    contour(beta)
    surface(beta)
    surface(U-C)

end
```



6.1.10 stress

6.1.10.1 3d_bimetal

```
{ 3D_BIMETAL.PDE

  This problem considers a small block of aluminum bonded to a larger block
  of iron.  The assembly is held at a fixed temperature at the bottom, and
  is convectively cooled on the sides and top.  We solve for the 3D temperature
  distribution, and the associated deformation and stress.

  All faces of the assembly are unconstrained, allowing it to grow as the
  temperature distribution demands.  We do not use an integral constraint
  to cancel translation and rotation, as we have done in 2D samples,
  because in 3D this is very expensive.  Instead, we let FlexPDE find a solution,
  and then remove the mean translation and rotation before plotting.

}

title 'Bimetal Part'

coordinates
    cartesian3

select
    painted   { show color-filled contours }

variables
    Tp        { temperature difference from stress-
free state }
    U         { X displacement }
    V         { Y displacement }
    W         { Z displacement }

definitions
    long = 1
```



```
    wide = 0.3
    high = 1
    tabx = 0.2
    taby = 0.4
    K                    { thermal conductivity }
    E                    { Youngs modulus }
    alpha                { expansion coefficient }
    nu                   { Poisson's Ratio  }

    Q = 0                { Thermal source }
    Ta = 0.              { define the ambient thermal sink temperature }

    { define the constitutive relations }
    G = E/((1+nu)*(1-2*nu))
    C11 = G*(1-nu)
    C12 = G*nu
    C13 = G*nu
    C22 = G*(1-nu)
    C23 = G*nu
    C33 = G*(1-nu)
    C44 = G*(1-2*nu)/2
    b = G*alpha*(1+nu)

    { Strains }
    ex = dx(U)
    ey = dy(V)
    ez = dz(W)
    gxy = dy(U) + dx(V)
    gyz = dz(V) + dy(W)
    gzx = dx(W) + dz(U)

    { Stresses }
    Sx  =  C11*ex + C12*ey + C13*ez - b*Tp
    Sy  =  C12*ex + C22*ey + C23*ez - b*Tp
    Sz  =  C13*ex + C23*ey + C33*ez - b*Tp
    Txy =  C44*gxy
    Tyz =  C44*gyz
    Tzx =  C44*gzx

    { find mean translation and rotation }
    Vol = Integral(1)
    Tx = integral(U)/Vol                    { X-motion }
    Ty = integral(V)/Vol                    { Y-motion }
    Tz = integral(W)/Vol                    { Z-motion }
    Rz = 0.5*integral(dx(V) - dy(U))/Vol    { Z-rotation }
    Rx = 0.5*integral(dy(W) - dz(V))/Vol    { X-rotation }
    Ry = 0.5*integral(dz(U) - dx(W))/Vol    { Y-rotation }

    { displacements with translation and rotation removed }
    { This is necessary only if all boundaries are free }
    Up = U - Tx + Rz*y - Ry*z
    Vp = V - Ty + Rx*z - Rz*x
    Wp = W - Tz + Ry*x - Rx*y

    { scaling factors for displacement plots }
    Mx = 0.2*globalmax(magnitude(y,z))/globalmax(magnitude(Vp,Wp))
    My = 0.2*globalmax(magnitude(x,z))/globalmax(magnitude(Up,Wp))
    Mz = 0.2*globalmax(magnitude(x,y))/globalmax(magnitude(Up,Vp))
    Mt = 0.4*globalmax(magnitude(x,y,z))/globalmax(magnitude(Up,Vp,Wp))

initial values
    Tp = 5.
    U = 1.e-5
    V = 1.e-5
    W = 1.e-5

equations
    Tp: div(k*grad(Tp)) + Q = 0.        { the heat equation }
    U:  dx(Sx) + dy(Txy) + dz(Tzx) = 0  { the U-displacement equation }
    V:  dx(Txy) + dy(Sy) + dz(Tyz) = 0  { the V-displacement equation }
    W:  dx(Tzx) + dy(Tyz) + dz(Sz) = 0  { the W-displacement equation }

extrusion z = 0,long

boundaries
    surface 1 value(Tp)=100            { fixed temp bottom }
    surface 2 natural(Tp)=0.01*(Ta-Tp)  { poor convective cooling top }

    Region 1    { Iron }
```

```
        K = 0.11
        E = 20e11
        nu  =0.28
        alpha =  1.7e-6
        start(0,0)
          natural(Tp) = 0.1*(Ta-Tp)      { better convective cooling on vertical sides }
          line to (wide,0)
            to (wide,(high-taby)/2)
            to (wide+tabx,(high-taby)/2)
            to (wide+tabx,(high+taby)/2)
            to (wide,(high+taby)/2)
            to (wide,high)
            to (0,high)
            to close

      Region 2   { Aluminum }
        K = 0.5
        E = 6e11
        nu  =0.25
        alpha =  2*(2.6e-6)              ! Exaggerate expansion
        start(wide,(high-taby)/2)
          line to (wide+tabx,(high-taby)/2)
            to (wide+tabx,(high+taby)/2)
            to (wide,(high+taby)/2)
            to close

monitors
    contour(Tp) on y=high/2  as "Temperature"
    contour(Up) on y=high/2 as "X-displacement"
    contour(Vp) on x=4*wide/5 as "Y-displacement"
    contour(Wp) on y=high/2 as "Z-displacement"
    grid(x+My*Up,z+My*Wp) on y=high/2 as "XZ Shape"
    grid(y+Mx*Vp,z+Mx*Wp) on x=wide/2 as "YZ Shape"
    grid(x+Mz*Up,y+Mz*Vp) on z=long/4 as "XY Shape"
    grid(x+Mt*Up,y+Mt*Vp,z+Mt*Wp)  as "Shape"

plots
    contour(Tp) on y=high/2  as "XZ Temperature"
    contour(Up) on y=high/2 as "X-displacement"
    contour(Vp) on x=4*wide/5 as "Y-displacement"
    contour(Wp) on y=high/2 as "Z-displacement"
    grid(x+My*Up,z+My*Wp) on y=high/2 as "XZ Shape"
    grid(y+Mx*Vp,z+Mx*Wp) on x=4*wide/5 as "YZ Shape"
    grid(x+Mz*Up,y+Mz*Vp) on z=long/4 as "XY Shape"
    grid(x+Mt*Up,y+Mt*Vp,z+Mt*Wp)  as "Shape"
    contour(Sx) on y=high/2 as "X-stress"
    contour(Sy) on y=high/2 as "Y-stress"
    contour(Sz) on y=high/2 as "Z-stress"
    contour(Txy) on y=high/2 as "XY Shear stress"
    contour(Tyz) on y=high/2 as "YZ Shear stress"
    contour(Tzx) on y=high/2 as "ZX Shear stress"

end
```

6.1.10.2 anisotropic_stress

```
{ ANISOTROPIC.PDE

  This example shows the application of FlexPDE to an extremely complex
  problem in anisotropic thermo-elasticity.  The equations of thermal
  diffusion and plane strain are solved simultaneously to give the
  thermally-induced stress and deformation in a laser application.

                -- Submitted by Steve Sutton
                   Lawrence Livermore National Laboratory
}


title "ANISOTROPIC THERMAL STRESS"

select
    errlim = 1e-4       { more accuracy to resolve stresses }

variables
    Tp(5)               { Temperature }
    up(1e-6)            { X-displacement }
    vp(1e-6)            { Y-displacement }
```

```
definitions
    Qs                  { The heat source, to be defined }
    Q0 = 3.16
    ro = 0.2            { Heat source radius }

    W = 2               { slab size constants }
    L = 0.5
    mag = 5000

    kp11 = 0.0135       { anisotropic conductivities }
    kp33 = 0.0135
    kp13 = 0.0016

    C11 = 49.22e5       { anisotropic elastic constants }
    C12 =  3.199e5
    C13 = 23.836e5
    C15 = -3.148e5
    C21 = C12
    C22 = 67.2e5
    C23 =  3.199e5
    C25 =  8.997e5
    C31 = C13
    C32 = C23
    C33 = 49.22e5
    C35 = -3.148e5
    C51 = C15
    C52 = C25
    C53 = C35
    C55 = 24.335e5


    ayy = 34.49e-6      { anisotropic expansion coefficients }
    axx = 34.49e-6
    azz = 25.00e-6
    axy = 9.5e-6

    h = 1.0

    Tb = 0.
    Q = Q0*(exp(-2*(x^2+y^2)/ro^2))  { Gaussian heat distribution }


                { some auxilliary definitions }
    qx = -kp33*dx(Tp) - kp13*dy(Tp)      { heat flux }
    qy = -kp13*dx(Tp) - kp11*dy(Tp)

                                         { expansion stress coefficients }
    apxx = C31*ayy + C32*azz + C33*axx + C35*axy
    apyy = C11*ayy + C12*azz + C13*axx + C15*axy
    apzz = C21*ayy + C22*azz + C23*axx + C25*axy
    apxy = C51*ayy + C52*azz + C53*axx + C55*axy

    exx = dx(up)                         { strain }
    eyy = dy(vp)
    exy = 0.5*(dy(up)+dx(vp))
                                         { stress }
    sxx = C31*eyy + C33*exx + 2*C35*exy - apxx*Tp
    syy = C11*eyy + C13*exx + 2*C15*exy - apyy*Tp
    szz = C21*eyy + C23*exx + 2*C25*exy - apzz*Tp
    sxy = C51*eyy + C53*exx + 2*C55*exy - apxy*Tp

initial values
    Tp = 5.
    up = 0
    vp = 0

equations

    Tp: dx(qx) + dy(qy) = Qs
    Up: dx(sxx) + dy(sxy) = 0.
    Vp: dx(sxy) + dy(syy) = 0.

constraints                              { prevent rigid-body motion: }
    integral(up) = 0                     { cancel X-motion }
    integral(vp) = 0                     { cancel Y-motion }
    integral(dx(vp) - dy(up)) = 0        { cancel rotation }

boundaries
  region 1
```

```
    Qs = Q
    start(-0.5*W,-0.5*L)
        natural(up) = 0.                { zero normal stress on all faces }
        natural(vp) = 0.
        natural(Tp) = h*(Tp-Tb)         { convective cooling on bottom boundary }
        line to (0.5*W,-0.5*L)
        natural(Tp) = 0.                { no heat flux across end }
            line to (0.5*W,0.5*L)
        natural(Tp) = h*(Tp-Tb)         { convective cooling on top boundary }
            line to (-0.5*W,0.5*L)
        natural(Tp) = 0.                { no heat flux across end }
            line to close

monitors
    grid (x+mag*up,y+mag*vp)
    contour(Tp) as "Temperature"

plots
    grid (x+mag*up,y+mag*vp)
    contour(Tp) as "Temperature"
    contour(Tp) as "Temperature" zoom(-.2,-.2,0.4,0.4)
    contour(up) as "x-displacement"
    contour(vp) as "y-displacement"
    vector(up,vp) as "Displacement vector plot"
    contour(sxx) as "x-normal stress"
    contour(syy) as "y-normal stress"
    contour(sxy) as "shear stress"
    elevation(Tp) from (0,-0.5*L) to (0,0.5*L) as "Temperature"
    elevation(sxx) from (0,-0.5*L) to (0,0.5*L) as "x-normal stress"
    elevation(syy) from (0,-0.5*L) to (0,0.5*L) as "y-normal stress"
    surface(Tp) as "Temperature"

end
```

6.1.10.3 axisymmetric_stress

```
{  AXISYMMETRIC_STRESS.PDE

   This example shows the application of FlexPDE to problems in
   axi-symmetric stress.

   The equations of Stress/Strain arise from the balance of forces in a
   material medium, expressed in cylindrical geometry as
        dr(r*Sr)/r - St/r  + dz(Trz) + Fr = 0
        dr(r*Trz)/r + dz(Sz) + Fz = 0

   where Sr, St and Sz are the stresses in the r- theta- and z- directions,
   Trz is the shear stress, and Fr and Fz are the body forces in the
   r- and z- directions.

   The deformation of the material is described by the displacements,
   U and V, from which the strains are defined as
        er = dr(U)
        et = U/r
        ez = dz(V)
        grz = dz(U) + dr(V).

   The quantities U,V,er,et,ez,grz,Sr,St,Sz and Trz are related through the
   constitutive relations of the material,
        Sr  =  C11*er + C12*et + C13*ez - b*Temp
        St  =  C12*er + C22*et + C23*ez - b*Temp
        Sz  =  C13*er + C23*et + C33*ez - b*Temp
        Trz =  C44*grz

   In isotropic solids we can write the constitutive relations as
        C11 = C22 = C33 = G*(1-nu)/(1-2*nu)     = C1
        C12 = C13 = C23 = G*nu/(1-2*nu)         = C2
        b = alpha*G*(1+nu)/(1-2*nu)
        C44 = G/2

   where G = E/(1+nu) is the Modulus of Rigidity
         E is Young's Modulus
         nu is Poisson's Ratio
   and   alpha is the thermal expansion coefficient.

   from which
        Sr  =  C1*er + C2*(et + ez) - b*Temp
        St  =  C1*et + C2*(er + ez) - b*Temp
```

```
        Sz  =  C1*ez + C2*(er + et) - b*Temp
        Trz =  C44*grz

   Combining all these relations, we get the displacement equations:
        dr(r*Sr)/r - St/r + dz(Trz) + Fr = 0
        dr(r*Trz)/r + dz(Sz) + Fz = 0

   These can be written as
        div(P) = St/r - Fr
        div(Q) = -Fz

   where P = [Sr,Trz]
   and   Q = [Trz,Sz]

   The natural (or "load") boundary condition for the U-equation defines the
   outward surface-normal component of P, while the natural boundary condition
   for the V-equation defines the surface-normal component of Q. Thus, the
   natural boundary conditions for the U- and V- equations together define
   the surface load vector.

   On a free boundary, both of these vectors are zero, so a free boundary
   is simply specified by
        load(U) = 0
        load(V) = 0.

   The problem analyzed here is a steel doughnut of rectangular cross-section,
   supported on the inner surface and loaded downward on the outer surface.

}

title "Doughnut in Axial Shear"

coordinates
    ycylinder('R','Z')

variables
    U            { declare U and V to be the system variables }
    V

definitions
    nu = 0.3            { define Poisson's Ratio }
    E  = 20             { Young's Modulus x 10^-11 }
    alpha = 0           { define the thermal expansion coefficient }
    G = E/(1+nu)
    C1 = G*(1-nu)/(1-2*nu)      { define the constitutive relations }
    C2 = G*nu/(1-2*nu)
    b = alpha*G*(1+nu)/(1-2*nu)
    Fr = 0              { define the body forces }
    Fz = 0
    Temp = 0            { define the temperature }

    Sr  =  C1*dr(U) + C2*(U/r + dz(V)) - b*Temp
    St  =  C1*U/r + C2*(dr(U) + dz(V)) - b*Temp
    Sz  =  C1*dz(V) + C2*(dr(U) + U/r) - b*Temp
    Trz =  G*(dz(U) + dr(V))/2

    r1 = 2              { define the inner and outer radii of a doughnut }
    r2 = 5
    q21 = r2/r1
    L = 1.0             { define the height of the doughnut }

initial values
    U = 0
    V = 0

equations               { define the axi-symmetric displacement equations }

    U:  dr(r*Sr)/r - St/r + dz(Trz) + Fr = 0
    V:  dr(r*Trz)/r + dz(Sz) + Fz = 0

boundaries
    region 1
      start(r1,0)
      load(U) =  0          { define a free boundary along bottom }
      load(V) =  0
      line to (r2,0)

      value(U) = 0          { constrain R-displacement on right }
      load(V) = -E/100      { apply a downward shear load }
```

```
    line to (r2,L)

    load(U) =  0          { define a free boundary along top }
    load(V) =  0
    line to (r1,L)

    value(U) = 0          { constrain all displacement on inner wall }
    value(V) = 0
    line to close

monitors
    grid(r+U,z+V)         { show deformed grid as solution progresses }

plots                     { hardcopy at to close: }
    grid(r+U,z+V)         { show final deformed grid }
    contour(U) as "X-Displacement"          { show displacement field }
    contour(V) as "Y-Displacement"          { show displacement field }
    vector(U,V) as "Displacement"           { show displacement field }
    contour(Trz) as "Shear Stress"
    surface(Sr) as "Radial Stress"

end
```

6.1.10.4 bentbar

```
{ BENTBAR.PDE

  This is a test problem from Timoshenko: Theory of Elasticity, pp41-46

  A cantilever is loaded by a distributed shearing force on the free end,
  while a point at the center of the mounted end is fixed.

  The solution is compared to Timoshenko's analytic solution.

  The equations of Stress/Strain arise from the balance of forces in a
  material medium, expressed as
        dx(Sx) + dy(Txy) + Fx = 0
        dx(Txy) + dy(Sy) + Fy = 0
  where Sx and Sy are the stresses in the x- and y- directions,
        Txy is the shear stress, and
        Fx and Fy are the body forces in the x- and y- directions.

  The deformation of the material is described by the displacements,
  U and V, from which the strains are defined as
        ex = dx(U)
        ey = dy(V)
        gxy = dy(U) + dx(V).

  The eight quantities U,V,ex,ey,gxy,Sx,Sy and Txy are related through the
  constitutive relations of the material. In general,
        Sx  =  C11*ex + C12*ey + C13*gxy - b*Temp
        Sy  =  C12*ex + C22*ey + C23*gxy - b*Temp
        Txy =  C13*ex + C23*ey + C33*gxy

  In orthotropic solids, we may take C13 = C23 = 0.
  In this problem we consider the thermal effects to be negligible.

}

title "Timoshenko's Bar with end load"

select
    cubic       { Use Cubic Basis }

variables
    U           { X-displacement }
    V           { Y-displacement }

definitions
    L = 1               { Bar length }
    hL = L/2
    W = 0.1             { Bar thickness }
    hW = W/2
    eps = 0.01*L
    I = 2*hW^3/3        { Moment of inertia }

    nu = 0.3            { Poisson's Ratio }
    E  = 2.0e11         { Young's Modulus for Steel (N/M^2) }
```

```
                        { plane stress coefficients }
    G   = E/(1-nu^2)
    C11 = G
    C12 = G*nu
    C22 = G
    C33 = G*(1-nu)/2

    amplitude=GLOBALMAX(abs(V)) { for grid-plot scaling }
    mag=1/amplitude

    force = -250         { total loading force in Newtons (~10 pound force) }
    dist = 0.5*force*(hW^2-y^2)/I       { Distributed load }

    Sx = (C11*dx(U) + C12*dy(V))        { Stresses }
    Sy = (C12*dx(U) + C22*dy(V))
    Txy = C33*(dy(U) + dx(V))

    { Timoshenko's analytic solution:  }
    Vexact = (force/(6*E*I))*((L-x)^2*(2*L+x) + 3*nu*x*y^2)
    Uexact = (force/(6*E*I))*(3*y*(L^2-x^2) +(2+nu)*y^3 -6*(1+nu)*hW^2*y)
    Sxexact = -force*x*y/I
    Txyexact = -0.5*force*(hW^2-y^2)/I

initial values
    U = 0
    V = 0

equations               { the displacement equations }
    U:  dx(Sx) + dy(Txy) = 0
    V:  dx(Txy) + dy(Sy) = 0

boundaries
    region 1
      start (0,-hW)

      load(U)=0         { free boundary on bottom, no normal stress }
      load(V)=0
        line to (L,-hW)

      value(U) = Uexact { clamp the right end }
      mesh_spacing=hW/10
        line to (L,0) point value(V) = 0
        line to (L,hW)

      load(U)=0         { free boundary on top, no normal stress }
      load(V)=0
      mesh_spacing=10
        line to (0,hW)

      load(U) = 0
      load(V) = dist    { apply distributed load to Y-displacement equation }
        line to close

plots
    grid(x+mag*U,y+mag*V)   as "deformation"   { show final deformed grid }
    elevation(V,Vexact) from(0,0) to (L,0) as "Center Y-Displacement(M)"
    elevation(V,Vexact) from(0,hW) to (L,hW) as "Top Y-Displacement(M)"
    elevation(U,Uexact) from(0,hW) to (L,hW) as "Top X-Displacement(M)"
    elevation(Sx,Sxexact) from(0,hW) to (L,hW) as "Top X-Stress"
    elevation(Txy,Txyexact) from(0,0) to (L,0) as "Center Shear Stress"

end
```

6.1.10.5 elasticity

```
{ ELASTICITY.PDE

  This example shows the application of FlexPDE to a complex problem in
  thermo-elasticity.  The equations of thermal diffusion and
  plane strain are solved simultaneously to give the thermally-induced
  stress and deformation in a laser application.

  A rod amplifier of square cross-section is imbedded in a large copper
  heat-sink. The rod is surrounded by a thin layer of compliant metal.
  Pump light is focussed on the exposed side of the rod.

  We wish to calculate the effect of the thermal load on the laser rod.
```

```
    The equations of Stress/Strain arise from the balance of forces in a
    material medium, expressed as
         dx(Sx) + dy(Txy) + Fx = 0
         dx(Txy) + dy(Sy) + Fy = 0

    where Sx and Sy are the stresses in the x- and y- directions,
    Txy is the shear stress, and Fx and Fy are the body forces in the
    x- and y- directions.

    The deformation of the material is described by the displacements,
    U and V, from which the strains are defined as
         ex = dx(U)
         ey = dy(V)
         gxy = dy(U) + dx(V).

    The eight quantities U,V,ex,ey,gxy,Sx,Sy and Txy are related through the
    constitutive relations of the material. In general,
         Sx  =  C11*ex + C12*ey + C13*gxy - b*Temp
         Sy  =  C12*ex + C22*ey + C23*gxy - b*Temp
         Txy =  C13*ex + C23*ey + C33*gxy

    In orthotropic solids, we may take C13 = C23 = 0.

    Combining all these relations, we get the displacement equations:
         dx[C11*dx(U)+C12*dy(V)] + dy[C33*(dy(U)+dx(V))] + Fx = dx(b*Temp)
         dy[C12*dx(U)+C22*dy(V)] + dx[C33*(dy(U)+dx(V))] + Fy = dy(b*Temp)


    The "Plane-Strain" approximation is appropriate for  the cross-section
    of a cylinder which is long in the Z-direction, and in which there is no
    Z-strain. The cylinder is loaded by surface tractions and body forces
    applied along the length of cylinder, and which are independent of Z.

    In this case, we may write
         C11 = G*(1-nu)  C12 = G*nu                      b = G*alpha*(1+nu)
                         C22 = G*(1-nu)
                                        C33 = G*(1-2*nu)/2

    where G = E/[(1+nu)*(1-2*nu)]
          E is Young's Modulus
          nu is Poisson's Ratio
    and   alpha is the thermal expansion coefficient.

    The displacement form of the stress equations (for uniform temperature
    and no body forces) is then (dividing out G):

         dx[(1-nu)*dx(U)+nu*dy(V)] + 0.5*(1-2*nu)*dy[dy(U)+dx(V)]
                                         = alpha*(1+nu)*dx(Temp)

         dy[nu*dx(U)+(1-nu)*dy(V)] + 0.5*(1-2*nu)*dx[dy(U)+dx(V)]
                                         = alpha*(1+nu)*dy(Temp)

    In order to quantify the "natural" (or "load") boundary condition mechanism,
    consider the stress equations in their original form:
         dx(Sx) + dy(Txy) = 0
         dx(Txy) + dy(Sy) = 0

    These can be written as
         div(P) = 0
         div(Q) = 0

    where P = [Sx,Txy]
    and   Q = [Txy,Sy]

    The natural (or "load") boundary condition for the U-equation defines the
    outward surface-normal component of P, while the natural boundary condition
    for the V-equation defines the surface-normal component of Q. Thus, the
    natural boundary conditions for the U- and V- equations together define
    the surface load vector.

    On a free boundary, both of these vectors are zero, so a free boundary
    is simply specified by
         load(U) = 0
         load(V) = 0.

    -- Submitted by Steve Sutton, Lawrence Livermore National Laboratory

}
```

```
title "Thermo-Elastic Stress"

select errlim = 1.0e-4

variables
    Tp                  { declare the system variables to be Tp, Up and Vp }
    Up
    Vp

definitions
    k                   { declare thermal conductivity - values come later }
    Q                   { declare thermal Source - values come later }
    E                   { declare Young's Modulus - values come later }
    nu                  { declare Poisson's Ratio - values come later }
    alpha               { declare Expansion coefficient - values come later }

                        { The heat deposition function: }
    adep = 1.8          { define the absorption coefficient }
    yo = 0.6            { define the pattern width }
    I0  = 1             { define the input flux }
   Qrodp = adep*I0*(exp(-adep*x))*(exp(-((y/yo)^2)))

    Tb = 0.             { define the distant thermal sink temperature }

                        { define the constitutive relations }
    G = E/((1.+nu)*(1.-2.*nu))
    C11 = G*(1-nu)
    C12 = G*nu
    C22 = G*(1-nu)
    C33 = G*(1-2*nu)/2
    b = G*alpha*(1+nu)

                        { define some utility functions }
    ex = dx(Up)
    ey = dy(Vp)
    gxy = dy(Up) + dx(Vp)
    Sx  =  C11*ex + C12*ey - b*Tp
    Sy  =  C12*ex + C22*ey - b*Tp
    Txy =  C33*gxy


initial values
    Tp = 5.             { give FlexPDE an estimate of variable range }
    Up = 1.e-5
    Vp = 1.e-5

equations
    { the heat equation }
    Tp: dx(k*dx(Tp)) + dy(k*dy(Tp)) + Q = 0.

    { the U-displacement equation }
   Up: dx(C11*dx(Up)+C12*dy(Vp)-b*Tp) + dy(C33*(dy(Up)+dx(Vp))) = 0.

    { the V-displacement equation }
   Vp: dx(C33*(dy(Up)+dx(Vp))) + dy(C12*dx(Up)+C22*dy(Vp)-b*Tp) = 0.

constraints                             { prevent rigid-body motion: }
    integral(up) = 0                    { cancel X-motion }
    integral(vp) = 0                    { cancel Y-motion }
    integral(dx(vp) - dy(up)) = 0       { cancel rotation }

boundaries

  region 1              { region one defines the problem domain as all copper
                            and sets the boundary conditions for the problem }
    k = 0.083
    Q = 0.
    E = 117.0e3
    nu = 0.4
    alpha = 10e-6

    start(0,-5)

    value(Tp) = Tb      { define a distant boundary with fixed temperature }
    natural(Up) = 0.    { zero X-load }
    natural(Vp) = 0.    { and zero Y-load }
    line to (5,-5) to (5,5) to (0,5)
```

```
    natural(Tp) = 0.    { left face has no heat loss }
    natural(Up) = 0.    { left boundary is free }
    natural(Vp) = 0.
    line to close

  region 2              { region two overlays an Indium potting layer }
    k = 0.083
    Q = 0.
    E =  60.0e3
    nu = 0.4
    alpha = 16e-6
    start (0,-0.6)
    line to (0.6,-0.6) to (0.6,0.6) to (0,0.6) to (0,0.5) to (0,-0.5) to close

  region 3              { region three overlays the laser rod }
    k = 0.0098
    Q = Qrodp
    E = 282.0e3
    nu = 0.28
    alpha = 7e-6
    start (0,-0.5)
    line to (0.5,-0.5) to (0.5,0.5) to (0,0.5) to close

monitors
    contour(Tp) as "Temperature"
    contour(Tp) as "Temperature" zoom(0,0,1,1)
    contour(Q) as "Heat deposition" zoom(0,0,1,1)
    contour(Up) as "X-displacement" zoom(0,0,1,1)
    contour(Vp) as "Y-displacement" zoom(0,0,1,1)
    grid(x+10000*Up,y+10000*Vp) as "deformation"

plots
    grid(x,y)
    contour(Tp) as "Temperature"
    contour(Tp) as "Temperature" zoom(0,0,1,1)
    contour(Q) as "Heat deposition" zoom(0,0,1,1)
    contour(Up) as "X-displacement" !zoom(0,0,1,1)
    contour(Vp) as "Y-displacement" !zoom(0,0,1,1)
    contour(Sx) as "X-Stress"   zoom(0,-0.75,1.5,1.5)
    contour(Sy) as "Y-Stress"   zoom(0,-0.75,1.5,1.5)
    contour(Txy) as "Shear Stress"   zoom(0,-0.75,1.5,1.5)
    vector(Up,Vp) as "displacement"
    vector(Up,Vp) as "displacement" zoom(0,0,1,1)
    grid(x+10000*Up,y+10000*Vp) as "deformation"

end
```

6.1.10.6 fixed_plate

```
{ FIXED_PLATE.PDE

  This example considers the bending of a thin rectangular plate under a
  distributed transverse load.

  For small displacements, the deflection U is described by the Biharmonic
  equation of plate flexure
        del2(del2(U)) + Q/D  =  0
  where
        Q is the load distribution,
        D = E*h^3/(12*(1-nu^2))
        E is Young's Modulus
        nu is Poisson's ratio
  and   h is the plate thickness.

  The boundary conditions to be imposed depend on the way in which the
  plate is mounted.  Here we consider the case of a clamped boundary,
  for which
        U = 0
        dU/dn = 0

  FlexPDE cannot directly solve the fourth order equation, but if we
  define V = del2(U), then the deflection equation becomes
        del2(U) = V
        del2(V) + Q = 0
  with the boundary conditions
        dU/dn = 0
        dV/dn = L*U
```

```
  where L is a very large number.

  In this system, dV/dn can only remain bounded if U -> 0, satisfying the
  value condition on U.

  The particular problem addressed here is a plate of 16-gauge steel,
  8 x 11.2 inches, covering a vacuum chamber, with atmospheric pressure
  loading the plate.  The edges are clamped.  Solutions to this problem
  are readily available, for example in Roark's Formulas for Stress & Strain,
  from which the maximum deflection is Umax =  0.219, in exact agreement
  with the FlexPDE result.

  (See FREE_PLATE.PDE 371 for the solution with a simply supported edge.)

  Note: Care must be exercised when extending this formulation to more complex
    problems.  In particular, in the equation del2(U) = V, V acts as a source
    in the boundary-value equation for U.  Imposing a value boundary condition
    on U does not enforce V = del2(U).

}


Title " Plate Bending - clamped boundary "

Select
    errlim = 0.005
    cubic       { Use Cubic Basis }

Variables
     U(0.1)
     V(0.1)

Definitions
    xslab = 11.2
    yslab = 8
    h = 0.0598  {16 ga}
    L = 1.0e4
    E = 29e6
    Q = 14.7
    nu = .3
    D = E*h^3/(12*(1-nu^2))

Initial Values
    U =  0
    V =  0

Equations
    U:  del2(U) = V
    V:  del2(V) = Q/D

Boundaries
    Region 1
      start (0,0)
      natural(U) = 0
      natural(V) = L*U
      line to (xslab,0)
           to (xslab,yslab)
           to (0,yslab)
           to close

Monitors
    contour(U)

Plots
    contour (U)  as "Displacement"
    elevation(U) from (0,yslab/2) to (xslab,yslab/2) as "Displacement"
    surface(U) as "Displacement"

End
```

6.1.10.7 free_plate

```
{ FREE_PLATE.PDE

  This example considers the bending of a thin rectangular plate under a
  distributed transverse load.

  For small displacements, the deflection U is described by the Biharmonic
```

```
  equation of plate flexure
        del2(del2(U)) + Q/D  =  0
  where
        Q is the load distribution,
        D = E*h^3/(12*(1-nu^2))
        E is Young's Modulus
        nu is Poisson's ratio
  and   h is the plate thickness.

  The boundary conditions to be imposed depend on the way in which the
  plate is mounted.  Here we consider the case of a simply supported
  boundary, for which the correct conditions are
        U = 0
        Mn = 0
  where Mn is the tangential component of the bending moment, which in turn
  is related to the curvature of the plate. An approximation to the second
  boundary condition is then
        del2(U) = 0.

  FlexPDE cannot directly solve the fourth order equation, but if we
  define V = del2(U), then the deflection equation becomes
        del2(U) = V
        del2(V) + Q = 0
  with the boundary conditions
        U = 0
        V = 0.

  The particular problem addressed here is a plate of 16-gauge steel,
  8 x 11.2 inches, covering a vacuum chamber, with atmospheric pressure
  loading the plate.  The edges are simply supported.  Solutions to this
  problem are readily available, for example in Roark's Formulas for Stress
  & Strain, from which the maximum deflection is Umax =  0.746, as compared
  with the FlexPDE result of 0.750.

  (See FIXED_PLATE.PDE 370 for the solution with a clamped edge.)

  Note: Care must be exercised when extending this formulation to more complex
    problems.  In particular, in the equation del2(U) = V, V acts as a source
    in the boundary-value equation for U.  Imposing a value boundary condition
    on U does not enforce V = del2(U).

}

Title " Plate Bending - simple support "

Select
    ngrid=10                    { increase initial gridding }
    cubic           { Use Cubic Basis }

Variables
     U(0.1)
     V(0.1)

Definitions
    xslab = 11.2
    yslab = 8
    h = 0.0598  {16 ga}
    L = 1.0e6
    E = 29e6
    Q = 14.7
    nu = .3
    D = E*h^3/(12*(1-nu^2))
```

```
Initial Values
    U =  0
    V =  0

Equations
     U: del2(U) = V
     V: del2(V) = Q/D

Boundaries
    Region 1
      start (0,0)
      value(U) = 0
      value(V) = 0
      line to (xslab,0)
           to (xslab,yslab)
           to (0,yslab)
           to close

Monitors
    contour(U)

Plots
    contour (U)  as "Displacement"
    elevation(U) from (0,yslab/2) to (xslab,yslab/2) as "Displacement"
    surface(U) as "Displacement"

End
```

6.1.10.8 harmonic

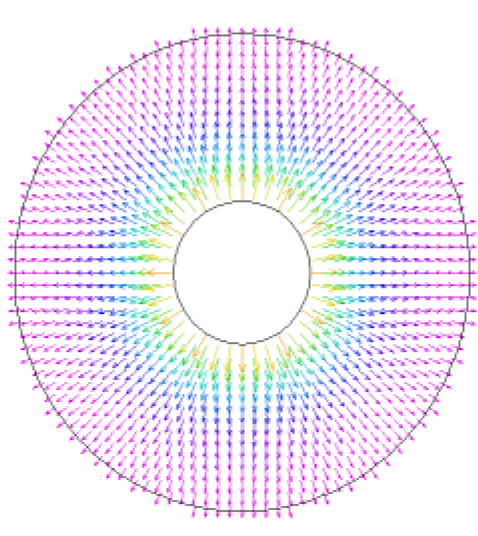
```
{ HARMONIC.PDE

  This example shows the use of FlexPDE in harmonic analysis of
  transient Stress problems.

  The equations of Stress/Strain in a material medium can be given as
        dx(Sx) + dy(Txy) + Fx = 0
        dx(Txy) + dy(Sy) + Fy = 0
  where Sx and Sy are the stresses in the x- and y- directions,
        Txy is the shear stress, and
        Fx and Fy are the body forces in the x- and y- directions.

  In a time-dependent problem, the material acceleration and viscous force
  act as body forces, and are included in a new body force term
        Fx1 = Fx0 - rho*dtt(U) + mu*del2(dt(U))
        Fy1 = Fy0 - rho*dtt(V) + mu*del2(dt(V))
  where rho is the material mass density, mu is the viscosity, and U and V
  are the material displacements in the x and y directions.

  If we assume that the displacement is harmonic in time (all transients
  have died out), then we can assert
        U(t) = U0*exp(-i*omega*t)
        V(t) = V0*exp(-i*omega*t)
  Here U0(x,y) and V0(x,y) are the complex amplitude distributions, and
  omega is the angular velocity of the oscillation.

  Substituting this assumption into the stress equations and dividing out
  the common exponential factors, we get (implying U0 by U and V0 by V)
        dx(Sx) + dy(Txy) + Fx0 + rho*omega^2*U - i*omega*mu*del2(U) = 0
        dx(Txy) + dy(Sy) + Fy0 + rho*omega^2*V - i*omega*mu*del2(V) = 0

  All the terms in this equation are now complex.  Separating into real
  and imaginary parts gives
        U = Ur + i*Ui
        Sx = Srx + i*Six
        Sy = Sry + i*Siy
        etc...

  Expressed in terms of the constitutive relations of the material,
        Srx = [C11*dx(Ur) + C12*dy(Vr)]
        Sry = [C12*dx(Ur) + C22*dy(Vr)]
        Trxy = C33*[dy(Ur) + dx(Vr)]
        etc...

  The final result is a set of four equations in Ur,Vr,Ui and Vi.

  Notice that the stress-balance equation is the Velocity equation, and it
```

```
  is to this equation that boundary loads must be applied.

  In the problem considered here, we have an aluminum bar one meter long and
  5 cm thick suspended on the left, and driven on the right by an oscillatory
  load.  The load frequency is chosen to be near the resonant frequency of
  the bar.

  We run the problem in three stages, first with no viscosity, then with increasing
  viscosities to show the mixing of real and imaginary components.

}

title "Harmonic Stress analysis"

variables  { Recall that the declared variable range, if too large, will affect the
             interpretation of error, and thus the timestep and solution accuracy }
    { Displacements }
    Ur
    Ui
    Vr
    Vi

definitions
    L = 1               { the bar length, in Meters }
    hL = L/2
    W = 0.05            { the bar thickness, in Meters }
    hW = W/2
    eps = 0.01*L

    nu = 0.3            { Poisson's Ratio }
    E  = 6.7e+10        { Young's Modulus for Aluminum (N/M^2) }

                        { plane strain coefficients }
    E1  = E/((1+nu)*(1-2*nu))
    C11 = E1*(1-nu)
    C12 = E1*nu
    C22 = E1*(1-nu)
    C33 = E1*(1-2*nu)/2

    rho = 2700          { Kg/M^3 }
    mu = staged(0,1e3,1e4)    { Estimated viscosity Kg/M/sec }

    cvel = sqrt(E/rho)  { sound velocity, M/sec }
    tau  = L/cvel       { transit time }
    tone = 0.25/tau     { approximate resonant frequency }
    omega = 2*pi*tone   { driving angular velocity  }

    amplitude=1e-8      { a guess for plot scaling }
    mag=1/amplitude

    force = 25          { loading force in Newtons (~1 pound force) }
                        { distribute the force  uniformly over the driven end: }
    fdist = force/W

    Um = sqrt(Ur^2+Ui^2)        { X-displacement amplitude }
    Vm = sqrt(Vr^2+Vi^2)        { X-displacement amplitude }

    Srx = (C11*dx(Ur) + C12*dy(Vr))        { Real Stresses }
    Sry = (C12*dx(Ur) + C22*dy(Vr))
    Trxy = C33*(dy(Ur) + dx(Vr))
    Six = (C11*dx(Ui) + C12*dy(Vi))        { Imaginary Stresses }
    Siy = (C12*dx(Ui) + C22*dy(Vi))
    Tixy = C33*(dy(Ui) + dx(Vi))

    Sxm = sqrt(Srx^2+Six^2)
    Sym = sqrt(Sry^2+Siy^2)
    Txym = sqrt(Trxy^2+Tixy^2)

equations             { define the displacement equations }
    Ur: dx(Srx) + dy(Trxy) + rho*omega^2*Ur + omega*mu*del2(Ui) = 0
    Ui: dx(Six) + dy(Tixy) + rho*omega^2*Ui - omega*mu*del2(Ur) = 0
    Vr: dx(Trxy) + dy(Sry) + rho*omega^2*Vr + omega*mu*del2(Vi) = 0
    Vi: dx(Tixy) + dy(Siy) + rho*omega^2*Vi - omega*mu*del2(Vr) = 0

boundaries
    region 1
      start (0,-hW)

      load(Ur)=0         { free boundary on bottom, no normal stress }
```

```
        load(Ui)=0
        load(Vr)=0
        load(Vi)=0
        line to (L,-hW)

        load(Vr) = force   { Apply oscillatory vertical load on end. }
        line to (L,hW)

        load(Vr)=0         { free boundary on top, no normal stress }
        line to (0,hW)

        value(Ur) = 0      { clamp the left end  }
        value(Ui) = 0
        value(Vr) = 0
        value(Vi) = 0
        line to close

monitors
    elevation(Vr,Vi) from(0,0) to (L,0)
      report(omega) report(mu)

plots
    grid(x+mag*Ur,y+mag*Vr)   as "Real displacement"   { show final deformed grid }
      report(omega) report(mu)
    grid(x+mag*Ui,y+mag*Vi)   as "Imag displacement"
      report(omega) report(mu)
    elevation(Vr,Vi) from(0,0) to (L,0)
      report(omega) report(mu)
    contour(Ur) as "Real X-displacement(M)"
      report(omega) report(mu)
    contour(Vr) as "Real Y-displacement(M)"
      report(omega) report(mu)
    contour(Ui) as "Imag X-displacement(M)"
      report(omega) report(mu)
    contour(Vi) as "Imag Y-displacement(M)"
      report(omega) report(mu)
    contour(Sxm) as "X-Stress amplitude"
      report(omega) report(mu)
    contour(Sym) as "Y-Stress amplitude"
      report(omega) report(mu)
    contour(Txym) as "Shear Stress amplitude"
      report(omega) report(mu)

end
```

6.1.10.9 prestube

```
{ PRESTUBE.PDE

   This example models the stress in a tube with an internal pressure.
         -  from "Fields of Physics on the PC"  by Gunnar Backstrom
}

title
   ' Tube With Internal Pressure'

variables
   u
   v

definitions
   mm = 1e-3
   r1 = 3*mm
   r2 = 10*mm
   q21= r2/r1
   mu = 0.3
   E = 200e9         {Steel}
   c = E/(1-mu^2)
   G = E/2/(1+mu)
   dabs= sqrt(u^2+ v^2)
   ex= dx(u)
   ey= dy(v)
   exy= dx(v)+ dy(u)
   sx= c*(ex+ mu*ey)
   sy= c*(mu*ex+ ey)
   sxy= G*exy

   p1= 1e8           { the internal pressure }
```



```
   { Exact expressions }
   rad= sqrt(x^2+ y^2)
   sr_ex= -p1*((r2/rad)^2 - 1)/(q21^2 - 1)
   st_ex=  p1*((r2/rad)^2 + 1)/(q21^2 - 1)
   dabs_ex= abs( rad/E*(st_ex- mu*sr_ex))

equations                { Constant temperature, no volume forces }
   u:   dx( c*(dx(u) + mu*dy(v)) ) + dy( G*(dx(v)+ dy(u)) )= 0
   v:   dx( G*( dx(v)+ dy(u)) )+  dy( c*(dy(v) + mu*dx(u)) )= 0

constraints              { Since all boundaries are free, it is necessary
                           to apply constraints to eliminate rigid-body motions }
   integral(u) = 0
   integral(v) = 0
   integral(dx(v)-dy(u)) = 0

boundaries
   region 1
   start (r2,0)
   load(u)= 0            { Outer boundary is free }
   load(v)= 0
        arc to (0,r2) to (-r2,0) to (0,-r2) to close
   start (r1,0)                  { Cut-out }
   load(u)= p1*x/r1      { Normal component of x-stress }
   load(v)= p1*y/r1      { Normal component of y-stress }
        arc to (0,-r1) to (-r1,0) to (0,r1) to close

monitors
   contour(dabs)

plots
   grid(x+200*u, y+200*v)
   elevation(sx, sr_ex) from (r1,0) to (r2,0)
   elevation(sy, st_ex) from (r1,0) to (r2,0)
   contour(dabs)         contour((dabs-dabs_ex)/dabs_ex)
   contour(u)            contour(v)
   vector(u,v)           vector(u/dabs, v/dabs)
   contour(sx)           contour(sy)              contour(sxy)
end
```

6.1.10.10 tension

```
{ TENSION.PDE

  This example shows the deformation of a tension bar with a hole.

  The equations of Stress/Strain arise from the balance of forces in a
  material medium, expressed as
        dx(Sx) + dy(Txy) + Fx = 0
        dx(Txy) + dy(Sy) + Fy = 0

  where Sx and Sy are the stresses in the x- and y- directions,
        Txy is the shear stress, and
        Fx and Fy are the body forces in the x- and y- directions.

  The deformation of the material is described by the displacements,
  U and V, from which the strains are defined as
        ex = dx(U)
        ey = dy(V)
        gxy = dy(U) + dx(V).

  The eight quantities U,V,ex,ey,gxy,Sx,Sy and Txy are related through the
  constitutive relations of the material. In general,
        Sx  =  C11*ex + C12*ey + C13*gxy - b*Temp
        Sy  =  C12*ex + C22*ey + C23*gxy - b*Temp
        Txy =  C13*ex + C23*ey + C33*gxy

  In orthotropic solids, we may take C13 = C23 = 0.

  Combining all these relations, we get the displacement equations:
       dx[C11*dx(U)+C12*dy(V)] + dy[C33*(dy(U)+dx(V))] + Fx = dx(b*Temp)
       dy[C12*dx(U)+C22*dy(V)] + dx[C33*(dy(U)+dx(V))] + Fy = dy(b*Temp)

  In the "Plane-Stress" approximation, appropriate for a flat, thin plate
  loaded by surface tractions and body forces IN THE PLANE of the plate,
  we may write
        C11 = G         C12 = G*nu
```

```
                          C22 = G
                                                C33 = G*(1-nu)/2
  where G = E/(1-nu^2)
        E is Young's Modulus
  and   nu is Poisson's Ratio.

  The displacement form of the stress equations (for uniform temperature
  and no body forces) is then (dividing out G):
        dx[dx(U)+nu*dy(V)] + 0.5*(1-nu)*dy[dy(U)+dx(V)] = 0
        dy[nu*dx(U)+dy(V)] + 0.5*(1-nu)*dx[dy(U)+dx(V)] = 0

  In order to quantify the load boundary condition mechanism,
  consider the stress equations in their original form:
        dx(Sx) + dy(Txy) = 0
        dx(Txy) + dy(Sy) = 0

  These can be written as
        div(P) = 0
        div(Q) = 0
  where P = [Sx,Txy]
  and   Q = [Txy,Sy]

  The "load" (or "natural") boundary condition for the U-equation defines
  the outward surface-normal component of P, while the load boundary condition
  for the V-equation defines the surface-normal component of Q. Thus, the
  load boundary conditions for the U- and V- equations together define
  the surface load vector.

  On a free boundary, both of these vectors are zero, so a free boundary
  is simply specified by
        load(U) = 0
        load(V) = 0.

  Here we consider a tension strip with a hole, subject to an X-load.

}

title 'Plane Stress tension strip with a hole'

select
    errlim = 1e-4       { increase accuracy to resolve stresses }
    painted             { paint all contour plots }

variables
    U                   { declare U and V to be the system variables }
    V

definitions
    nu = 0.3            { define Poisson's Ratio }
    E  = 21             { Young's Modulus x 10^-11 }
   G  = E/(1-nu^2)
    C11 = G
    C12 = G*nu
    C22 = G
    C33 = G*(1-nu)/2
    p1 = (1-nu)/2

initial values
    U = 1
    V = 1

equations               { define the Plane-Stress displacement equations }
     U:  dx(dx(U) + nu*dy(V)) + p1*dy(dy(U) + dx(V))  = 0
     V:  dy(dy(V) + nu*dx(U)) + p1*dx(dy(U) + dx(V))  = 0

boundaries
    region 1
      start (0,0)
      load(U)=0         { free boundary, no normal stress }
      load(V)=0
      line to (3,0)     { walk bottom }

      load(U)=0.1       { define an X-stress of 0.1 unit on right edge}
      load(V) = 0
      line to (3,1)

      load(U)=0         { free boundary top }
      load(V)=0
      line to (0,1)
```

```
      value(U)=0        { fixed displacement on left edge }
      value(V)=0
      line to close

                        { Cut out a hole }
      load(U) = 0
      load(V) = 0
      start(1,0.25)
      arc(center=1,0.5) angle=-360

monitors
    grid(x+U,y+V)     { show deformed grid as solution progresses }

plots                 { hardcopy at to close: }
    grid(x+U,y+V)      { show final deformed grid }
    vector(U,V) as "Displacement"        { show displacement field }
    contour(U) as "X-Displacement"
    contour(V) as "Y-Displacement"
    contour((C11*dx(U) + C12*dy(V))) as "X-Stress"
    contour((C12*dx(U) + C22*dy(V))) as "Y-Stress"
    surface((C11*dx(U) + C12*dy(V))) as "X-Stress"
    surface((C12*dx(U) + C22*dy(V))) as "Y-Stress"

end
```

6.1.10.11 vibrate

```
{ VIBRATE.PDE

  This example shows the use of FlexPDE in transient Stress problems.

  The equations of Stress/Strain in a material medium can be given as
        dx(Sx) + dy(Txy) + Fx = 0
        dx(Txy) + dy(Sy) + Fy = 0

  where Sx and Sy are the stresses in the x- and y- directions,
        Txy is the shear stress, and
        Fx and Fy are the body forces in the x- and y- directions.

  In a time-dependent problem, the material acceleration and viscous force
  act as body forces, and are included in a new body force term
        Fx1 = Fx0 - rho*dtt(Ux) + mu*del2(dt(Ux))
        Fy1 = Fy0 - rho*dtt(Uy) + mu*del2(dt(Uy))

  where rho is the material mass density, mu is the viscosity, and Ux and Uy
  are the material displacements.

  The second time derivative in the acceleration term cannot be modelled
  directly in FlexPDE, but the problem can still be solved.
  Define Vx and Vy as the velocities in the x and y directions; then
        Vx = dt(Ux)
    and Vy = dt(Uy)

  The body forces are then
        Fx1 = Fx0 - rho*dt(Vx) + mu*del2(Vx)
        Fy1 = Fy0 - rho*dt(Vy) + mu*del2(Vy)

  This results in a set of four equations in Ux,Uy,Vx and Vy.

  Notice that the stress-balance equation is the Velocity equation, and it
  is to this equation that boundary loads must be applied.

  In the problem considered here, we have an aluminum bar one meter long and
  5 cm thick suspended on the left, and driven on the right by an oscillatory
  load.  The load frequency is chosen to be near the resonant frequency of
  the bar.

}

title "Transient Stress analysis"

select
    deltat=1.0e-7       { Start out at careful timestep, it will grow. }
    ngrid=21            { Grid a little more densely than default }

    regrid = off        { Cell splitting causes instantaneous changes in the
                          effective material properties. These changes act
```

```
                         like small earthquakes in the material, and propagate
                         high-frequency noise. To avoid these effects, we
                         supress grid refinement. }

variables               { Recall that the declared variable range, if too large,
                         will affect the interpretation of error, and thus the
                         timestep and solution accuracy }

    Ux (threshold=1e-7)   { Displacements }
    Uy (threshold=1e-7)
    Vx (threshold=1e-5)   { Velocities }
    Vy (threshold=1e-5)

definitions
    L = 1               { the bar length, in Meters }
    hL = L/2
    W = 0.05            { the bar thickness, in Meters }
    hW = W/2
    eps = 0.01*L

    nu = 0.3            { Poisson's Ratio }
    E  = 6.7e+10        { Young's Modulus for Aluminum (N/M^2) }

                        { plane strain coefficients }
    E1  = E/((1+nu)*(1-2*nu))
    C11 = E1*(1-nu)
    C12 = E1*nu
    C22 = E1*(1-nu)
    C33 = E1*(1-2*nu)/2

    rho = 2700          { Kg/M^3 }
    mu = 1e3            { Estimated viscosity Kg/M/sec }
    smoother = 1        { artificial diffusion to smooth results  (M^2/sec) }

    cvel = sqrt(E/rho)  { sound velocity, M/sec }
    tau  = L/cvel       { transit time }
    tone = 0.25/tau     { approximate resonant frequency }
    freq = 1.1*tone       { driving frequency  }
    period = 1/freq

    amplitude=1e-8      { a guess for plot scaling }
    mag=1/amplitude

    force = 25          { loading force in Newtons (~1 pound force) }
                        { distribute the force  uniformly over the driven end: }
    fdist = force/W
                        { the driving force is sinusoidal in time: }
    jiggle = force*sin(2*pi*freq*t)

    Sx = (C11*dx(Ux) + C12*dy(Uy))        { Stresses }
    Sy = (C12*dx(Ux) + C22*dy(Uy))
    Txy = C33*(dy(Ux) + dx(Uy))

initial values
    Ux = 0              { start at rest }
    Uy = 0
    Vx = 0
    Vy = 0

equations             { define the displacement equations }
    Ux:  Vx + smoother*div(grad(Ux)) = dt(Ux)
    Uy:  Vy + smoother*div(grad(Uy)) = dt(Uy)
    Vx:  dx(Sx) + dy(Txy) + mu*div(grad(Vx)) = rho*dt(Vx)
    Vy:  dx(Txy) + dy(Sy) + mu*div(grad(Vy)) = rho*dt(Vy)

boundaries
    region 1
      start (0,-hW)

      load(Vx)=0         { free boundary on bottom, no normal stress }
      load(Vy)=0
      line to (L,-hW)

      load(Vx) = 0
      load(Vy) = jiggle  { Apply oscillatory vertical load on end.
                           Note that this driving force must be applied to the
                           equation which contains the stress divergence. }
      line to (L,hW)
```

```
    load(Vx)=0         { free boundary on top, no normal stress }
    load(Vy)=0
    line to (0,hW)

    value(Ux) = 0      { freeze left end (both displacement and velocity) }
    value(Uy) = 0
    value(Vx) = 0
    value(Vy) = 0
    line to close

  feature              { a "Gridding Feature" to force grid refinement near
                             the mount }
    start (hW/2,-hW) line to (hW/2,hW)
    start (L-hW/2,-hW) line to (L-hW/2,hW)

time 0 to 4*period

monitors
    for cycle=5
      elevation(Uy) from(0,0) to (L,0) range=(-amplitude,amplitude)

plots
    for t= period/2 by period/2 to endtime
      grid(x+mag*Ux,y+mag*Uy)   as "deformation"     { show final deformed grid }
      vector(Ux,Uy) as "displacement"                { show displacement field }
      vector(Vx,Vy) as "velocity"                    { show velocity field }
      contour(Ux) as "X-displacement(M)"
      contour(Uy) as "Y-displacement(M)"
      contour(Vx) as "X-velocity(M/s)"
      contour(Vy) as "Y-velocity(M/s)"
      contour(Sx) as "X-Stress"
      contour(Sy) as "Y-Stress"
      contour(Txy) as "Shear Stress"

histories
    history(Ux) at (L,0) (0.8*L,0) (hL,0) as "Horizontal Displacement(M)"
    history(Vx) at (L,0) (0.8*L,0) (hL,0) as "Horizontal Velocity(M/s)"
    history(Sx) at (eps,hW-eps) (eps,-hW+eps) (L-eps,hW-eps) (L-eps,-hW+eps)
                as "Horizontal Stress"
    history(Uy) at (L,0) (0.8*L,0) (hL,0) as "Vertical Displacement(M)"
    history(Vy) at (L,0) (0.8*L,0) (hL,0) as "Vertical Velocity(M/s)"
    history(Sy) at (eps,hW-eps) (eps,-hW+eps) (L-eps,hW-eps) (L-eps,-hW+eps)
                as "Vertical Stress"
    history(Txy) at (eps,hW-eps) (eps,-hW+eps) (L-eps,hW-eps) (L-eps,-hW+eps)
                as "Shear Stress"

end
```

6.2 usage

6.2.1 2d_integrals

```
{ 2D_INTEGRALS.PDE
  This problem demonstrates the specification of various integrals in 2D.
}

title '2D Integrals'

coordinates
    ycylinder

variables
    Tp

select errlim=1e-4

definitions
    R0 = 0.1
    R1 = 0.4
    R2 = 0.6
    Long = 1.0
    K                    { thermal conductivity -- values supplied later }
    Q = 10*max(1-((r-R1)^2+z^2),0)              { Thermal source }

    { This definition shows the use of a selector to force integration of Tp only in inner
region }
```

```
    flag2=0
    temp2 = if flag2>0 then Tp else 0

initial values
    Tp = 0.

equations
    Tp:    div(k*grad(Tp)) + Q = 0        { the heat equation }

boundaries

    Region 1        { define full domain boundary }
       K = 1
       start "outside" (R0,-Long/2)
         value(Tp) = 0      { fix all side temps }
         line to (R2,-Long/2)
           to (R2,Long/2)
           to (R0,Long/2)
           to close

    Region 2 "Inner"
       flag2=1
       k=0.2
       start "Inner" (R0,-Long/2)
         line to (R1,-Long/2)
           to (R1,Long/2)
           to (R0,Long/2)
           to close

monitors
    contour(Tp)

plots
    contour(Tp)
    contour(k*dz(Tp))
    contour(q)

    summary
      report("Compare various forms for integrating over region 2")
      report(integral(Tp,2))
      report(integral(Tp,"Inner"))
      report(integral(temp2))   { integrates over full volume, but temp2 is zero in region 1 }
      report '-----'

      report("Compare various forms for integrating over total volume")
      report(integral(Tp,"ALL"))
      report(integral(Tp))
      report '-----'

      report("Compare various forms for integrating over surface of region 2")
      report(sintegral(normal(-k*grad(Tp)),2))
      report(sintegral(normal(-k*grad(Tp)),'Inner'))
      report '-----'

      report("Compare surface flux on region 2 to internal divergence integral")
      report(sintegral(normal(-k*grad(Tp)),"Inner"))
      report(integral(Q,"Inner"))
      report '-----'

      report("Compare surface flux on total volume to internal divergence integral")
      report(sintegral(normal(-k*grad(Tp))))
      report(integral(Q))
      report '-----'

end
```

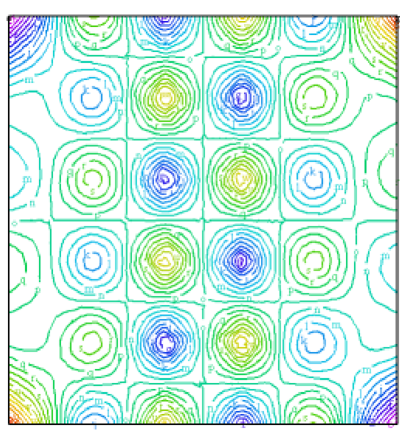
6.2.2 fillet

```
{  FILLET.PDE

   This example demonstrates the use of the FILLET 189 and BEVEL 189 commands
}
```

```
title 'fillet test'

Variables
    u

definitions
    k = 1
    u0 = 1-x^2-y^2
    s = 2*3/4+5*2/4

equations
    U: div(K*grad(u)) +s = 0

boundaries
    Region 1
        start(-1,-1)
        value(u)=u0
        line to (1,-1) FILLET(0.1)
             to (-0.25,-0.25) FILLET(0.1)
             to (-1,1) BEVEL(0.1)
             to close


monitors
     grid(x,y)
     contour(u)
plots
     grid(x,y)
     contour(u)
     contour(u) zoom(0.6,-1, 0.2,0.2)  as "Convex Fillet Closeup"
     contour(u) zoom(-0.3,-0.3, 0.1,0.1) as "Concave Fillet Closeup"

end
```

6.2.3 fit+weight

```
{ FIT+WEIGHT.PDE

  This test shows the use of spatially-varying weights in the FIT 129 function.

  There are no variables or equations here, just a domain and some tabular
  data which is FIT in different ways.

  The weight declared in the FIT statement is effectively the square of the
  spatial distance over which variations are smoothed.

}

title 'Test Variable-Weight FIT'

definitions
    u = table('table.tbl')


boundaries
    region 1
      start(0,10)
      line to (0,0) to (10,0) to (10,10) to close

plots
     grid(x,y)
     contour(u)
     contour(fit(u))   as 'unweighted'
     contour(fit(u,0.2))   as 'constant weight'
     contour(fit(u,0.02*(x-5)^2)) as 'side-weights'
     contour(fit(u,0.05*x)) as 'right-side weight'

end
```



6.2.4 function_definition

```
{  FUNCTION_DEFINITION.PDE

   This example demonstrates the use of functional parameter definitions 163.
```

```
}

title 'Functional Parameter Definition test'

Variables
     u

definitions

  { Declare "Sq" a function of argument "A".
       "A" is a dummy name that represents the actual argument passed
        by an invocation.  }
  Sq(a) = a*a
  { Define two functions for use in domain layout.
        The "n" argument rotates by 90 degree increments.}
  xx(n) = cos(n*pi/2)
  yy(n) = sin(n*pi/2)

equations
  { invoke the "Sq" function as a component of the equation.  This makes
    the system nonlinear }
  U: div(grad(u)) + 80*Sq(u)*dx(u) +4 = 0

boundaries
    region 1
        start(xx(0),yy(0))
        value(u)=0
        line to (xx(1),yy(1))   { definition evaluates corners of a diamond }
             to (xx(2),yy(2))
             to (xx(3),yy(3))
        to close

monitors
    contour(u)

plots
    surface(u)
    contour(u)

end
```

6.2.5 ifthen

```
{ IFTHEN.PDE

  This example demonstrates the use of "IF...THEN" 143 conditionals in arithmetic statements.

  We solve a heat equation in which the conductivity is defined by a conditional
  (IF..THEN) expression.

  Caveat:
    IF..THEN can be dangerous if used improperly.
    Equation coefficients that are discontinuous functions of the system
    variables can cause convergence failure or tiny timesteps and slow
    execution.  See SWAGETEST.PDE 393.

}

title 'Nonlinear heatflow, conditional conductivity'

Variables
    u

definitions
    a =    IF (u<0.5) and (x<100)
           THEN    IF u < 0.2
                   THEN 1.4
                   ELSE 1+2*abs(u)
           ELSE 2

Initial values
    u = 1 - (x-1)^2 - (y-1)^2

equations
    U: div(a*grad(u)) + 4 = 0;

boundaries
```

```
    Region 1
        start(0,0)
            value(u)=0
        line to (2,0) to (2,2) to (0,2) to close

monitors
    contour(u)
plots
    surface(u)
    contour(u)
    contour(a) as "Conditional Conductivity"
    elevation(a,u) from (0,1) to (2,1) as "Conductivity and Solution"

end
```

6.2.6 lump

```
{ LUMP.PDE

  This example illustrates use of the LUMP 130 function.

  LUMP(F) saves an averaged value of F in each mesh cell, and returns the same
  value for any position within the cell.

  Notice that LUMP(F) is NOT the same as the "lumped parameters" frequently referred to
  in finite element literature.

  LUMP(f) 130 is syntactically like SAVE(f) 131, in that it stores a representation of
  its argument for later use.

}
title 'LUMP test'

select
    contourgrid=400  { use a very dense plot grid to show lump structure }

Variables
    u

definitions
    k = 2
    u0 = 1+x^2+y^2
    s = u0 - 4*k
    lumps = lump(s)        { Used in a definition }

Initial values
    u = 1

equations
    U: u - div(K*grad(u))  = s

boundaries
    Region 1
        start(-1,-1)
            value(u)=u0
        line to (1,-1) to (1,1) to (-1,1) to close

monitors
    contour(u)

plots
    grid(x,y)
    contour(u)
    contour(s)
    contour(lump(s))    as "Lumped Source - Direct Reference"
    contour(lumps) as "Lumped Source - Defined Parameter"

end
```

6.2.7 polar_coordinates

```
{  POLAR_COORDINATES.PDE

   This example demonstrates the use of functional parameter definitions
    to pose equations in polar-coordinate form.  The function definitions
    expand polar derivatives in cartesian (XY) geometry.
```

```
}
title 'Polar Coordinates'

Variables
    u

definitions
    k = 1
    u0 = 1-r^2
    s = 4
    dr(f) = (x/r)*dx(f) + (y/r)*dy(f)  { functional definition of polar derivatives... }
    dphi(f) = (-y)*dx(f) + x*dy(f)     {... in cartesian coordinates }

equations  { equation expressed in polar coordinates
                 (Multiplied by r^2 to clear the r=0 singularity) }
    U: r*dr(r*dr(u)) + dphi(dphi(u)) + r*r*s = 0

boundaries
    Region 1
        start(0,0)
        natural(u) = 0 line to (1,0)
        value(u)=u0    arc(center=0,0) angle=90
        natural(u)=0   line to close

monitors
    grid(x,y) as "Computation Mesh"
    contour(u) as "Solution"
    contour(u-u0) as "Error (u-u0)"

plots
    grid(x,y) as "Computation Mesh"
    contour(u) as "Solution"
    contour(u-u0) as "Error (u-u0)"
end
```

6.2.8 repeat

```
{ REPEAT.PDE

  This example illustrates the use of the REPEAT 144 statement to generate
  repetitive structures, and the string facility for creating labels.

}
title 'REPEAT and $string test'

Variables
    u

definitions
    a = 1
    { a list of X-coordinates: }
    xc=array(1/3, 2/3, 3/3, 4/3, 5/3)
    { a list of Y-coordinates: }
    yc=array(1/3, 2/3, 3/3, 4/3, 5/3)

    rad = 0.1  { radius of circular dots }
    s = 0

equations
    U: div(a*grad(u)) + s  = 0;

boundaries
    region 1
        start(0,0)
            value(u)=0

        line to (2,0) to (2,2) to (0,2) to close
    region 2
        a = 0.05
        s = 4*magnitude(x-1,y-1)
        repeat i=1 to 5                    { an indexed loop on X-position }
            repeat j=1 to 5                { an indexed loop on Y-position }
                { an array of circular dots at the tabulated coordinates }
                start "Loop"+$i+$j (xc[i]+rad,yc[j])  {construct loop name using string conversion
}
                arc(center=xc[i],yc[j]) angle=360
            endrepeat
```

```
        endrepeat

monitors
    contour(u)

plots
    contour(u) painted
    surface(u)
    surface(s) as "Source"
        repeat i=1 to 5
            repeat j=1 to 5
                elevation(u) on  'loop'+$i+$(j)
            endrepeat
        endrepeat

end
```

6.2.9 save

```
{ SAVE.PDE

  This example illustrates use of the SAVE 131 function.

  SAVE(F) computes the value of F at each mesh node, and returns interpolated
  values for any position within a cell.
  If  F is very expensive to compute, the use of SAVE can reduce the overall cost
  of a simulation.
  SAVE also hides the complexity of F from differentiation in forming the coupling
  matrix, and may therefore avoid numerical difficulties encountered in computing
  the derivatives of pathological functions.

}
title 'SAVE test'

select
    ngrid=20
    contourgrid=100  { use a very dense plot grid to show data structure }

Variables
    u,v

definitions
    k = 2
    u0 = 1+x^2+y^2
    s = cos(20*x)*cos(20*y)
    save_s = save(s)       { Used in a definition }

Initial values
    u = 1

equations
    U: u - div(K*grad(u))  = s
    V: v - div(K*grad(v))  = save_s

boundaries
    region 1
        start(-1,-1)
            value(u)=u0 value(v)=u0
        line to (1,-1) to (1,1) to (-1,1) to close
    region 2
        k=4
        start(-1,-1) line to (0,-1) to (0,0) to (-1,0) to close

plots
    grid(x,y)
    contour(u)
    contour(v)
    contour(s)
    contour(save_s)
    elevation(s, save_s) from(-1,0) to (1,0)

end
```

6.2.10 spacetime1

```
{ SPACETIME1.PDE

  This example illustrates the use of FlexPDE to solve an initial value problem
  of 1-D transient heatflow as a 2D boundary-value problem.

  Here the spatial coordinate is represented by X, the time coordinate by Y,
  and the temperature by u(x,y).

  With these symbols, the transient heatflow equation is:
        dy(u) = D*dxx(u),
  where D is the diffusivity, given by
        D = K/s*rho,
        K       is the conductivity,
        s       is the specific heat,
  and   rho     is the density.

  The problem domain is taken to be the unit square.

  We specify the initial value of u(x,0) along y=0, as well as the time history
  along the sides x=0 and x=1.

  The value of u is thus assigned everywhere on the boundary except
  along the segment y=1, 0<x<1.  Along that boundary, we use the
  natural boundary condition,
        natural(u) = 0,
  since this corresponds to the application of no boundary sources on this
  boundary segment and hence implies a free segment.  This builds in the
  assumption that y = 1 (and hence t = 1) is sufficiently large for steady
  state to have been reached. [Note that since the only y-derivative term is
  first order, the default procedure of FlexPDE does not integrate this term
  by parts, and the Natural(u) BC does not correspond to a surface flux,
  functioning only as a source or sink.]

  This problem can be solved analytically, so we can plot the deviation
  of the FlexPDE solution from the exact answer.

}

title "1-D Transient Heatflow as a Boundary-Value problem"
select
     alias(x) "distance"
     alias(y) "time"

variables
     u

definitions
     diffusivity = 0.06     { pick a diffusivity that gives a nice graph }
     frequency = 2          { frequency of initial sinusoid }
     fpi = frequency*pi
     ut0 = sin(fpi*x)       { define initial distribution of temperature }
     u0 = exp(-fpi^2 *diffusivity*y)*ut0   { define exact solution }

Initial values
     u = ut0                { initialize all time to t=0 value }

equations
     U: dy(u) = diffusivity*dxx(u) { define the heatflow equation }

boundaries
     Region 1
        start(0,0)
        value(u)=ut0    { set the t=0 temperature }
        line to (1,0)

        value(u) = 0    { always cold at x=1 }
        line to (1,1)

        natural(u) = 0  { no sources at t=1 }
        line to (0,1)

        value(u) = 0    { always cold at x=0 }
        line to close

monitors
     contour(u)
```

```
plots
     contour(u)
     surface(u)
     contour(u-u0) as "error"

end
```

6.2.11 spacetime2

```
{ SPACETIME2.PDE

  This example is a modification of SPACETIME1.PDE 387, showing the solution of
  one-dimensional transient heatflow with differing material properties,
  cast as a boundary-value problem.

  The time variable is represented by Y, and the temperature by u(x,y).

  We specify two regions of differing conductivity, KX.

  The initial Temperature is given as a truncated parabola along y=0.

  We specify reflective boundary conditions in X (natural(u)=0) along
  the sides x=0 and x=1.

  The value of u is thus assigned everywhere on the boundary except
  along the segment y=1, 0<x<1.  Along that boundary, we use the
  natural boundary condition,
               natural(u) = 0,
  since this corresponds to the application of no boundary sources.

}

title "1-D Transient Heatflow as a Boundary-Value Problem"

Variables
     u               { define U as the system variable }

definitions
     kx              { declare KX as a parameter, but leave the value for later }

Initial values
     u = 0           { unimportant, since this problem is masquerading
                                    as a linear boundary-value problem }

equations           { define the heatflow equation }
    U: dy(u)   =  dx(kx*dx(u))

boundaries
    region 1
        kx = 0.1                 { conductivity = 0.1 in region 1 }

        start(0,0)
        value(u)=2.025-10*x^2   { define the temperature at t=0, x<=0.45 }
        line to (0.45,0)

        value(u) = 0            { force zero temperature for t=0, x>0.45 }
        line to (1,0) to (1,1)

        natural(u) = 0          { no flux across x=1 boundary }
        line to (1,1)

        natural(u) =  0         { no sources on t=1 boundary }
        line to (0,1)

        natural(u) = 0          { no flux across x=0 boundary }
        line to close

    region 2
        kx = 0.01                { low conductivity in region 2 }
        start(0.45,0)           { lay region 2 over center strip of region 1 }
        line to (0.55,0)
             to (0.55,1)
             to (0.45,1)
             to close
```

```
monitors
     contour(u)

plots
     contour(u)
     surface(u)

end
```

6.2.12 spline_boundary

```
{ SPLINE_BDRY.PDE

  This example shows the use of the SPLINE 167 statement in constructing boundary curves.

  A circular arc is approximated by five spline segments.
  The end segments are made very short to establish the proper slope at the ends.

  The problem solves a heatflow equation on a quarter circle and compares the solution
  with the analytic value.

}

title 'Spline Boundary'

Variables
    u

definitions
    k = 1
    u0 = 1-r^2
    s = 4

equations
    U: div(k*grad(u)) + s  = 0

boundaries
    Region 1
        start(0,0)
        natural(u) = 0 line to (1,0)
        value(u)=0
        spline to(0.99985,0.01745)  ! short initial interval to establish slope
               to (0.866,0.5)
               to(0.5,0.866)
               to (0.01745,0.99985)    ! short final interval to establish slope
               to (0,1)
        natural(u)=0 line to close

monitors
    grid(x,y)
    contour(u)
    contour(u-u0)
plots
    grid(x,y)
    contour(u)
    contour(u-u0)
end
```

6.2.13 staged_geometry

```
{ STAGED_GEOMETRY.PDE

  This problem shows the use of staging to solve  a problem for a range
  of geometries.

}

title 'Staged Geometry'

select
    stages=3
    autostage=off  { pause after each stage }

definitions
    width = 2*stage
```

```
Variables
    u

equations
    U: div(grad(u)) + 4 = 0;

boundaries
    region 1
        start(0,0)
        value(u)=0
        line to (width,0) to (width,2) to (0,2) to close

monitors
    contour(u)

plots
    grid(x,y)
    surface(u)
    contour(u)

histories
    history(integral(u)) vs width as "Integral vs width"

end
```

6.2.14 stages

```
{ STAGES.PDE

  This example demonstrates the use of staging to solve a problem for a range of parameters.

  We stage both the equation parameters and the solution ERRLIM 148.

  The problem is a nonlinear test, which solves a modified steady-state Burgers equation.

}

title 'Staged Problem'

select
    stages = 3 { run only the first three of the listed stages }
    errlim = staged(0.01, 0.001, 0.0005)

Variables
    u

definitions
    scale = staged(1, 2, 4, 8)     { extra value ignored }
    a = 1/scale

Initial values
    u = 1 - (x-1)^2 - (y-1)^2

equations
    U: div(a*grad(u)) + scale*u*dx(u) +4 = 0;

boundaries
    region 1
        start(0,0)
        value(u)=0
        line to (2,0) to (2,2) to (0,2) to close

monitors
    contour(u)

plots
    surface(u) report scale as "Scale"
    contour(u) report scale as "Scale"

histories
    history(integral(u)) vs scale  as "Ingegral vs Scale"
end
```

6.2.15 stage_vs

```
{ STAGE_VS.PDE

  This problem is a modification of STAGES.PDE 390 in which the VERSUS 211 qualifier
  has been used to change the abcissa of the history plot.
}

title 'Staged Problem - Versus parameter'

select
    stages=3
    errlim = staged(0.01, 0.001, 0.0005)

Variables
    u

definitions
    scale = staged(1, 2, 4, 8)     { extra value ignored }
    a = 1/scale

Initial values
    u = 1 - (x-1)^2 - (y-1)^2

equations
    U: div(a*grad(u)) + scale*u*dx(u) +4 = 0;

boundaries
    region 1
        start(0,0)
        value(u)=0
        line to (2,0) to (2,2) to (0,2) to close

monitors
    contour(u)

plots
    surface(u) report scale as "Scale"
    contour(u) report scale as "Scale"

histories
    history(integral(u)) versus scale
end
```

6.2.16 standard_functions

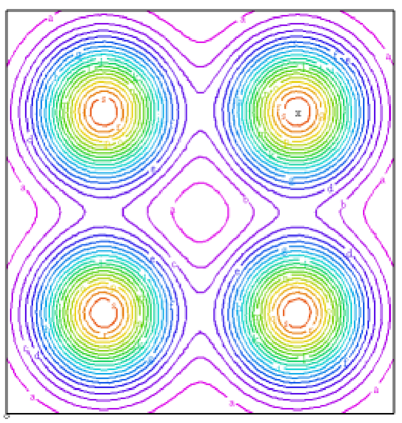
```
{ STANDARD_FUNCTIONS.PDE

   This example illustrates available mathematical functions 126 in FlexPDE.
   It also shows the use of FlexPDE as a plot utility.

}
title "Test Standard Functions"

coordinates cartesian1

{ -- No variables, no equations -- }

{ -- Definitions can be included, if desired -- }

{ -- We need a plot domain: -- }
boundaries
    region 1
      start(-1) line to (1)

plots
    elevation(sqrt(x)) from (0) to (1)
    elevation(dx(sqrt(x)),0.5/sqrt(x)) from (0.01) to (1)

    elevation(sin(pi*x)) from (-1) to (1)
    elevation(dx(sin(pi*x)),pi*cos(pi*x)) from (-1) to (1)

    elevation(cos(pi*x)) from (-1) to (1)
    elevation(dx(cos(pi*x)),-pi*sin(pi*x)) from (-1) to (1)

    elevation(tan(pi*x)) from (-0.499) to (0.499)
    elevation(dx(tan(pi*x)),pi/cos(pi*x)^2) from (-0.499) to (0.499)
```

```
    elevation(exp(x)) from (-1) to (1)
    elevation(dx(exp(x)),exp(x)) from (-1) to (1)

    elevation(ln(x)) from (0.01) to (1)
    elevation(dx(ln(x)),1/x) from (0.01) to (1)

    elevation(log10(x)) from (0.01) to (1)
    elevation(dx(log10(x)),1/(x*ln(10))) from (0.01) to (1)

    elevation(arcsin(x)) from (-1) to (1)
    elevation(dx(arcsin(x)),1/sqrt(1-x^2)) from (-0.999) to (0.999)

    elevation(arccos(x)) from (-1) to (1)
    elevation(dx(arccos(x)),-1/sqrt(1-x^2)) from (-0.999) to (0.999)

    elevation(arctan(x)) from (-1) to (1)
    elevation(dx(arctan(x)),1/(1+x^2)) from (-1) to (1)

    elevation(abs(x)) from (-1) to (1)
    elevation(dx(abs(x))) from (-1) to (1)

    elevation(sinh(x)) from (-1) to (1)
    elevation(dx(sinh(x)),cosh(x)) from (-1) to (1)

    elevation(cosh(x)) from (-1) to (1)
    elevation(dx(cosh(x)),sinh(x)) from (-1) to (1)

    elevation(tanh(x)) from (-1) to (1)
    elevation(dx(tanh(x)),1/cosh(x)^2) from (-1) to (1)

    elevation(erf(x)) from (-1) to (1)
    elevation(dx(erf(x)),2*exp(-x^2)/sqrt(pi)) from (-1) to (1)

    elevation(erfc(x)) from (-1) to (1)
    elevation(dx(erfc(x)),-2*exp(-x^2)/sqrt(pi)) from (-1) to (1)

    elevation(sign(x)) from (-1) to (1)
    elevation(dx(sign(x))) from (-1) to (1)

    elevation(x^(-4)) from (0.01) to (0.1)
    elevation(dx(x^(-4)),-4*x^(-5)) from (0.01) to (0.1)

    elevation(x^(2*x)) from (0.001) to (1)
    elevation(dx(x^(2*x)),2*x^(2*x)*(1+ln(x))) from (0.001) to (1)

    elevation(bessj(0,20*x),bessj(1,20*x),bessj(2,20*x)) from (0) to (1)
    elevation(bessy(0,20*x),bessy(1,20*x),bessy(2,20*x)) from (0.05) to (1)
    elevation(dx(bessj(0,20*x)),-20*bessj(1,20*x)) from (0) to (1)
    elevation(dx(bessj(1,20*x)),20*(bessj(1,20*x)/(20*x)-bessj(2,20*x))) from (0.001) to (1)

    elevation(expint(1,2*x),expint(2*x)) from (0.001) to (1)
    elevation(1/gammaf(1,2*x),1/gammaf(2*x)) from (0.001) to (1)

end
```

6.2.17 sum

```
{ SUM.PDE

  This example demonstrates the use of the SUM 131 function.
  It poses a heatflow problem with a heat source made up of four
  gaussians.  The source is composed by a SUM over gaussians
  referenced to arrays of center coordinates.

}
title 'Sum test'

Variables
    u

definitions
    k = 1
    u0 = 1-x^2-y^2      { boundary forced to parabolic values }
    xc = array(-0.5,0.5,0.5,-0.5)  { arrays of source spot coordinates }
    yc = array(-0.5,-0.5,0.5,0.5)
    s = sum( i, 1, 4, exp(-10*((x-xc[i])^2+(y-yc[i])^2)) )        { summed Gaussian source }
```

```
equations
    U: div(K*grad(u)) +s = 0

boundaries
    region 1
        start(-1,-1)
        value(u)=u0
        line to (1,-1)
               to (1,1)
               to (-1,1)
               to close

monitors
    grid(x,y)
    contour(u)
    contour(s)

plots
    grid(x,y)
    contour(u)
    contour(s)

end
```



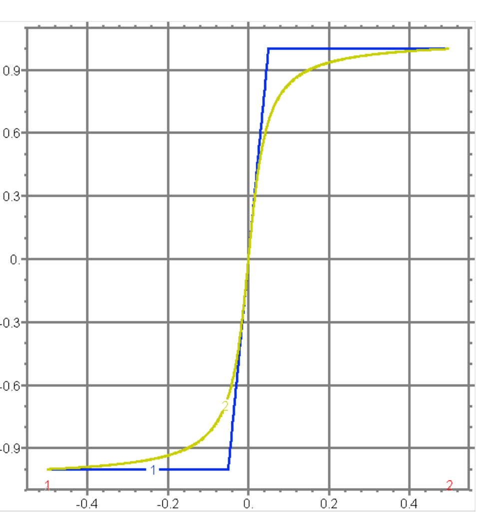
6.2.18 swage_pulse

```
{ SWAGE_PULSE.PDE

  A pulse can be made by two ifs:
  r1 = IF x<x1 THEN 0 ELSE 1
  r2 = IF x<x2 THEN 1 ELSE 0
  pulse = r1*r2

  This can be directly translated in to SWAGE 132 or RAMP 130 statements with width W:
  spulse = SWAGE(x-x1,0,1,w) * SWAGE(x-x2,1,0,w)
  rpulse = RAMP(x-x1,0,1,w) * RAMP(x-x2,1,0,w)

}

title "SWAGE and RAMP Pulses"

select
     elevationgrid=2000

{ -- No variables, no equations -- }

definitions
  x1 = -0.5
  x2 = 0.5
  w = 0.05
  swage_pulse = SWAGE(x-x1,0,1,w) * SWAGE(x-x2,1,0,w)
  ramp_pulse = RAMP(x-x1,0,1,w) * RAMP(x-x2,1,0,w)

boundaries
    region 1
      start(-1,-0.1) line to (1,-0.1) to (1,0.1) to (-1,0.1) to close

plots
    elevation(swage_pulse) from (-1,0) to (1,0)
    elevation(ramp_pulse) from (-1,0) to (1,0)

end
```

6.2.19 swage_test

```
{ SWAGE_TEST.PDE

  This example illustrates the use of the SWAGE 132 and RAMP 130 functions to generate
  smoother alternatives to the IF..THEN 143 construct.

  IF..THEN is frequently used to turn sources on and off, to define discontinuous
  initial conditions and the like.
```

```
  But in an adaptive system like FlexPDE, discontinuities can be very troublesome.
  They create very high frequency transients which can cause intense regridding
  and tiny timesteps.  When they occur in equation coefficients, they can cause
  convergence failure in Newton's method iterations.

  The SWAGE and RAMP functions are an attempt to give users an alternative to the
  IF..THEN for defining transitions.  These functions, particularly SWAGE, allow
  FlexPDE to sense the presence of  a transition and follow it in the iterative solver.

  In the plots created by this problem, we show both the values generated by the functions,
  and their derivatives.  By contrast, an IF..THEN has an infinite (ie, undefined) derivative
  which is impossible to accurately represent numerically.
}

title "SWAGE and RAMP Functions"

select
    elevationgrid=2000

{ -- No variables, no equations -- }

{ -- Definitions can be included, if desired -- }

{ -- We need a plot domain: -- }
boundaries
    region 1
      start(-1,-0.1) line to (1,-0.1)
        to (1,0.1) to (-1,0.1) to close

plots
    elevation(ramp(x,-1,1,0.1), swage(x,-1,1,0.1))
        from (-0.5,0) to (0.5,0)

    elevation(dx(ramp(x,-1,1,0.1)), dx(swage(x,-
1,1,0.1)))
        from (-0.5,0) to (0.5,0)

end
```



6.2.20 tabulate

```
{ TABULATE.PDE

  This problem tabulates an arithmetic expression into a data table.

  The structure of the inline tabulate command is:
    TABULATE <range_controls> : <expression>

  The <range_control> clause is
    VERSUS  <name> ( <list_specification> )
  or
    FOR  <name> ( <list_specification> )

  A <list_specification> may be the name of an array or a list of values,
  possibly including "BY <step> TO <last>" clauses.

  A TABULATE 169 command can be preceded by SPLINE 167 to request
  spline interpolation rather than the default linear interpolation.

  TABLES 165 may be constructed with one, two or three coordinates.

  The constructed tables are exported in various forms, to show the use
  of TABULATE 169 to create tables for other FlexPDE applications to use.

}
title 'Tabulation Test'

select
  regrid=off

variables
  u

definitions
  alpha =tabulate versus x(0 by 0.1 to 10)
                  versus y(0 by 0.1 to 10)
```

```
                  : sin(x)*sin(y)+1.1
  xar =array (0 by 0.1 to 10)
  beta =spline tabulate for x(xar)
                        for y(0 by 0.1 to 10)
                        : sin(x)*sin(y)+1.1
  p = x
  q = p+y
  s = y^2*(p+q)

equations
  U:   div(beta*grad(u)) + alpha = 0

boundaries
  region 1
    start(0,10)
    value(u) = 0
    line to (0,0) to (10,0) to (10,10) to close

monitors
  contour(u)

plots
  grid(x,y) as "computation mesh"
  contour(u) as "solution"
  surface(u) as "solution"
  contour(alpha)  as "tabulated data" export file='alpha.tbl'
  contour(beta) as "spline-tabulated data"
  contour(alpha-beta)  as "linear-spline difference"
  vector(grad(alpha))  as "table gradient"
  vector(grad(beta))  as "spline gradient"
  surface(alpha)  as "tabulated data"
  surface(beta) as "spline data"
  table(alpha)
  table(s)
  vtk(beta)

contour(error)

end
```

6.2.21 tintegral

```
{ TINTEGRAL.PDE

  This example illustrates use of the TINTEGRAL 136 function in time-dependent problems.
}

title
  "Float Zone"

coordinates
  xcylinder('Z','R')

variables
  temp (threshold=100)

definitions
  k = 0.85                          {thermal conductivity}
  cp = 1                            { heat capacity }
  long = 18
  H = 0.4                           {free convection boundary coupling}
  Ta = 25                           {ambient temperature}
  A = 4500                          {amplitude}

  source = A*exp(-((z-1*t)/.5)^2)*(200/(t+199))

  tsource = time_integral(vol_integral(source))
  t1 = time_integral(1.0)

initial value
  temp = Ta

equations
  temp:  div(k*grad(temp)) + source = cp*dt(temp)

boundaries
  region 1
```

```
    start(0,0)
    natural(temp) = 0 line to (long,0)
    value(temp) = Ta line to (long,1)
    natural(temp) = -H*(temp - Ta) line to (0,1)
    value(temp) = Ta line to close
  feature
    start(0.01*long,0) line to (0.01*long,1)

time -0.5 to 19

monitors
  for t = -0.5 by 0.5 to (long + 1)
  elevation(temp) from (0,1) to (long,1) range=(0,1800) as "Surface Temp"
  contour(temp)
  contour(dt(temp))

plots
  for t = -0.5 by 0.5 to (long + 1)
  elevation(temp) from (0,0) to (long,0) range=(0,1800) as "Axis Temp"

histories
  history(temp,dt(temp)) at (0,0) (1,0) (2,0) (3,0) (4,0) (5,0) (6,0) (7,0) (8,0)
                 (9,0) (10,0) (11,0) (12,0) (13,0) (14,0) (15,0) (16,0)
                 (17,0) (18,0)
  history(t1) as "Tintegral(1)"
  history(tsource)  as "Tintegral(Source)"

end
```

6.2.22 two_histories

```
{ TWO_HISTORIES.PDE

  This example illustrates use of multiple arguments in a HISTORY 211 plot.
  It also shows the use of the WINDOW plot qualifier on a HISTORY 211 plot.
  The problem is the same as FLOAT_ZONE.PDE 337.
}

title
  "Multiple HISTORY functions"

coordinates
  xcylinder('Z','R')

select
  cubic                         { Use Cubic Basis }

variables
  temp (threshold=100)

definitions
  k = 0.85                            {thermal conductivity}
  cp = 1                              { heat capacity }
  long = 18
  H = 0.4                             {free convection boundary coupling}
  Ta = 25                             {ambient temperature}
  A = 4500                            {amplitude}

  source = A*exp(-((z-1*t)/.5)^2)*(200/(t+199))

initial value
  temp = Ta

equations
  temp:  div(k*grad(temp)) + source = cp*dt(temp)

boundaries
  region 1
    start(0,0)
    natural(temp) = 0 line to (long,0)
    value(temp) = Ta line to (long,1)
    natural(temp) = -H*(temp - Ta) line to (0,1)
    value(temp) = Ta line to close
  feature
    start(0.01*long,0) line to (0.01*long,1)

time -0.5 to 19 by 0.01
```

```
monitors
  for t = -0.5 by 0.5 to (long + 1)
  elevation(temp) from (0,1) to (long,1) range=(0,1800) as "Surface Temp"
  contour(temp)

plots
  for t = -0.5 by 0.5 to (long + 1)
  elevation(temp) from (0,0) to (long,0) range=(0,1800) as "Axis Temp"

histories

  history(temp,dt(temp)) at (5,0) (10,0) (15,0)
  history(temp,dt(temp)) at (5,0) (10,0) (15,0) window = 5  ! moving window
  history(temp,dt(temp)) at (5,0) (10,0) (15,0) window(3,8) ! fixed window
  history(integral(temp),integral(dt(temp)))

end
```

6.2.23 unit_functions

```
{ UNIT_FUNCTIONS.PDE

 This example illustrates the unit step, unit pulse,
 and unit ramp functions ustep(arg1), upulse(arg1,arg2),
 and uramp(arg1,arg2) See Unit Functions 128.

}

title
  "unit functions"

select
  elevationgrid=500

{no variables}

definitions
  x1 = 0.2
  x2 = 0.4

{no equations}

{plot domain -- required}
boundaries
  region 1
    start (-1,0)
    line to (1,0) to (1,1) to (-1,1) to close

plots
  elevation(ustep(x-x1)) from (0,0) to (1,0)
  elevation(dx(ustep(x-x1))) from (0,0) to (1,0)
  elevation(upulse(x-x1,x-x2)) from (0,0) to (1,0)
  elevation(dx(upulse(x-x1,x-x2))) from (0,0) to (1,0)
  elevation(uramp(x-x1,x-x2)) from (0,0) to (1,0)
  elevation(dx(uramp(x-x1,x-x2))) from (0,0) to (1,0)
  ! generating a square wave by clipping a cosine
  elevation(ustep(cos(4*pi*x))) from (-1,0) to (1,0)
  ! the duty cycle can be controlled by offsetting the cosine from zero
  elevation(ustep(cos(4*pi*x)-0.3)) from (-1,0) to (1,0)

end
```

6.2.24 vector_functions

```
{ VECTOR_FUNCTIONS.PDE

  This example illustrates the vector functions 140

  VECTOR
  MAGNITUDE
  DOT
  CROSS
  NORMAL
  TANGENTIAL
```

```
}

title
  "vector functions"

select
  elevationgrid=500

{no variables}

definitions

  u= exp(-x^2+ y)               { A scalar potential, perhaps }
  f= grad(u)                    { F = grad(u)  is a vector }
  df= div(f)                    { Divergence of F is a scalar}
  cf= curl(f)                   { Curl of F is a new vector }
  vx= -sin(y)   vy= 2*sin(x)    { vector components }
  v= vector(vx,vy)              { Another vector }
  mv= magnitude(v)              { Magnitude of v }
  cv= curl(v)
  ccv= curl(curl(v))

{no equations}

{plot domain -- required}
boundaries
  region 1
    start "Outer" (-1,0)
    line to (1,0) to (1,1) to (-1,1) to close

  feature
   start "inner" (-1/2,1/2) line to (1/2,1/2)

plots
  vector(f)
  elevation(normal(f)) on "Outer"
  elevation(tangential(f)) on "inner"
  contour(df)   as "Div F"
  contour(mv)   as "Magnitude V"
  contour(dot(v,vector(x,0)))
  contour(zcomp(cross(f,v)))
  contour(zcomp(cv))   as "Curl V"
  vector(ccv)   as "Curl Curl V"

end
```

6.2.25 1D

6.2.25.1 1d_cylinder

```
{ 1D_CYLINDER.PDE

  This problem tests the implementation of 1D cylindrical coordinates in FlexPDE.
  A distributed source is applied to a heatflow equation.  The source is chosen as
  the analytic derivative of an assumed Gaussian solution.  The numerical solution
  is then compared to the analytical solution.

}
title '1D Cylinder Test -- Gaussian'

coordinates
    cylinder1   { default coordinate name is 'R' }

variables
    u

definitions
    k = 1
    w=0.1
    { assume a gaussian solution }
    u0 = exp(-r^2/w^2)
    { apply the correct analytic source for cylindrical geometry (we could use
      div(k*grad(u0)) here, but that would not test the 1D Cylinder expansions) }
    s = -(4/w^2)*(r^2/w^2-1)*u0

    left=point(0)
    right=point(1/10)
```

```
equations
    U: div(K*grad(u)) +s = 0

boundaries
    region 1
        start   left   point value(u)=u0
        line to right  point load(u)=(-2*k*r*u0/w^2)

monitors
    elevation(u) from left to right

plots
    elevation(u,u0) from left to right
    elevation(u-u0) from left to right as "Error"
    elevation(-div(grad(u)),s) from (0.01) to right
    elevation(-grad(u),-grad(u0)) from (0.01) to right

end
```

6.2.25.2 1d_cylinder_transient

```
{ 1D_CYLINDER_TRANSIENT.PDE

  This problem analyzes the diffusive loss of a solute from a solvent due to leakage
  across an outer boundary using 1D cylindrical coordinates.

}

title '1D time dependent diffusion in a Cylinder'

coordinates
    cylinder1("R")

variables
    C

definitions
    D = 1
    source = 0
    b = 1
    a = 2
    C0 = 10
    diss = 0.01     ! dissolution coefficient
    Cext = 0        ! external sink concentration
    Flux = -D*dr(C)

initial values
    C = C0

equations
    C: div(D*grad(C)) + source = dt(C)

boundaries
    region 1
        start (b)   point load(C)=0
        line to (a) point load(C)=diss*(Cext-C)  !outer leakage rate

time 0 to 10

monitors
    for cycle=1
        elevation(C) from (b) to (a)

plots
    for cycle=10
        elevation(C) from (b) to (a)
        elevation(Flux) from (b) to (a)  range=(0,0.01) {minimum plot range}
    history(C) at (b) ((b+a)/2) (a)
    history(Flux) at (b) ((b+a)/2) (a)

end
```

6.2.25.3 1d_float_zone

```
{ 1D_FLOAT_ZONE.PDE

  This is a version of the standard example "Float_Zone.pde" 337 in 1D cartesian geometry.

}

title
  "Float Zone in 1D Cartesian geometry"

coordinates
  cartesian1

variables
  temp(threshold=100)

definitions
  k =10                 { thermal conductivity }
  cp = 1                { heat capacity }
  long = 18
  H = 0.4               { free convection boundary coupling }
  Ta = 25               { ambient temperature }
  A = 4500              { amplitude }

  source = A*exp(-((x-1*t)/.5)^2)*(200/(t+199))

initial value
  temp = Ta

equations
  Temp: div(k*grad(temp)) + source -H*(temp - Ta) = cp*dt(temp)

boundaries
  region 1
    start(0) point value(temp) = Ta
    line to (long) point value(temp) = Ta

time -0.5 to 19 by 0.01

monitors
  for t = -0.5 by 0.5 to (long + 1)
  elevation(temp) from (0) to (long) range=(0,1800) as "Surface Temp"

plots
  for t = -0.5 by 0.5 to (long + 1)
  elevation(temp) from (0) to (long) range=(0,1800) as "Axis Temp"
  elevation(source) from(0) to (long)
  elevation(-k*grad(temp)) from(0) to (long)

histories
  history(temp) at (0) (1) (2) (3) (4) (5) (6) (7) (8)
                   (9) (10) (11) (12) (13) (14) (15) (16)
                   (17) (18)

end
```

6.2.25.4 1d_slab

```
{ 1D_SLAB.PDE

    this problem analyzes heat flow in a slab using 1D cartesian coordinates.
}

TITLE 'Heat flow through an Insulating layer in 1D'
COORDINATES
  Cartesian1  { default coordinate is 'X' }
VARIABLES
  Phi       { the temperature }
DEFINITIONS
  K = 1     { default conductivity }
  R = 0.5       { insulator thickness }
EQUATIONS
  Phi: Div(-k*grad(phi)) = 0

BOUNDARIES
  REGION 1      { the total domain }
```

```
      START(-1)    POINT VALUE(Phi)=0
      LINE TO (1) POINT VALUE(Phi)=1
      { note: no 'close'! }
    REGION 2  'blob'  { the embedded layer }
      k = 0.001
      START (-R) LINE TO (R)
  PLOTS
    ELEVATION(Phi) FROM (-1) to (1)
  END
```

6.2.25.5 1d_sphere

```
{ 1D_SPHERE.PDE

  This problem demonstrates the use of 1D spherical coordinates.

}

title '1D Sphere Test -- Gaussian'

coordinates
    sphere1 { default coordinate name is "R" }

variables
    u

definitions
    k = 1
    w=0.1
    { assume a gaussian solution }
    u0 = exp(-r^2/w^2)
    { apply the correct analytic source for spherical geometry
       (we could use div(k*grad(u0)) here, but that would not test the 1D Sphere expansions) }
    s = -(2/w^2)*(2*r^2/w^2-3)*u0

    left=point(0)
    right=point(1/10)

equations
    U:  div(K*grad(u)) +s = 0

boundaries
    region 1
        start left      point value(u)=u0
        line to right   point load(u)=(-2*k*r*u0/w^2)

monitors
    elevation(u) from left to right

plots
    elevation(u,u0) from left to right
    elevation(u-u0) from left to right as "Error"
    elevation(-div(grad(u)),s) from (0.01) to right

end
```

6.2.26 3D_domains

6.2.26.1 2d_sphere_in_cylinder

```
{ 2D_SPHERE_IN_CYLINDER.PDE

  2D cylindrical (axi-symmetric) model of an empty sphere in a cylindrical box.

}

title '2D sphere in a can'

coordinates
    ycylinder("R","Z")  { vertical coordinate is cylinder axis }

variables
    u

definitions
    k = 1
```

```
    R0 = 1
    box = 2*R0

equations
    U: div(k*grad(u)) = 0

boundaries
    Region 1
        start(0,-box)
        value(u)=0 line to (box,-box)
        natural(u)=0 line to (box,box)
        value(u)=1 line to (0,box)
        natural(u)=0 line to (0,R0)     { cylindrical axis }
            arc(center=0,0) angle=-180  { spherical cutout }
            line to close               { cylindrical axis }

monitors
    grid(r,z)
    contour(u)
plots
    grid(r,z)
    contour(u)

end
```

6.2.26.2 3d_box_in_sphere

```
{ 3D_BOX_IN_SPHERE.PDE

  This problem demonstrates the construction of a box inside a sphere.

  We use two conical frustums to define an extrusion layer to contain the box.
  The flat surfaces define top and bottom of the box and the cones fall
  to meet at the diameter of the sphere.

  The box is then defined as a square section of the layer between the
  flat surfaces of the frustums.

  Click "Controls->Domain Review" 7 to watch the domain construction process.

  We solve a heat equation for demonstration purposes.

}
title '3D Box in a Sphere'

coordinates
    cartesian3

Select regrid=off       { for quicker completion }

variables
    u

definitions
    R0 = 1          { sphere radius }
    hbox = R0/4     { box half-size }
    { Make the box-bounding circle slightly bigger than box, or corner
      intersections will confuse the mesh generator. }
    Rbox = 1.1*sqrt(2)*hbox

    rho = sqrt(x^2+y^2)             { 2d radius - don't use 'R', it's 3D radius! }

    zsphere = SPHERE ((0,0,0),R0)   {  hemisphere shape }
    zbottom = -zsphere              { bottom of sphere }
    ztop = zsphere                  { top of sphere }

    zboxbottom = -hbox  { default box-bounding surfaces - patched later in outer sphere }
    zboxtop = hbox
    zcone = hbox*(R0-rho)/(R0-Rbox) { cone shape for bringing box top to sphere diameter }

    K = 1               { Define all parameter defaults for non-box volume}
    source = 0

equations
    U: div(K*grad(u)) + source   = 0

extrusion
    surface z = zbottom         { the bottom hemisphere and plane }
```

```
    surface z = zboxbottom
    surface z = zboxtop
    surface z = ztop            { the top hemisphere and plane }

boundaries
     surface 1 value(u)=0       { for demonstration purposes }
     surface 4 value(u)=0

     Region 1       { The sphere }
        zboxbottom = -zcone
        zboxtop = zcone
        start  (R0,0)
            arc(center=0,0) angle=360 to close

     limited region 2       { smaller circle overlays sphere }
        layer 2             { ... and exists only in layer 2 }
        start(Rbox,0)
            arc(center=0,0) angle=360 to close

     limited region 3       { the box outline }
        layer 2             { box exists only in layer 2 }
        source = 1
        K = 0.1
        start(-hbox,-hbox) line to (hbox,-hbox) to (hbox,hbox) to (-hbox,hbox) to close

plots
    grid(x,y,z) as "outer sphere"
    grid(x,z) on y=0  nolines as "cross-section showing box"
    grid(x,z) on y=0 paintregions nolines  as "region and layer structure"
    grid(x,y) on z=0 paintregions nolines  as "region and layer structure"
    contour(u) on y=0  as "temperature"

end
```

6.2.26.3 3d_cocktail

```
{ 3D_COCKTAIL.PDE

  This problem constructs a cocktail glass.
  It is the geometric construction only, there are no variables or equations.
  LIMITED 186 regions are used to remove parts of the extruded shape.

  Click "Controls->Domain Review" 7 to watch the mesh construction process.
}

TITLE 'Cocktail Glass'
COORDINATES cartesian3

DEFINITIONS
  rad=sqrt( x^2+ y^2)
  router = 0.3      { outer radius of glass }
  zglass = 0.5      { glass height }
  rbase = 0.2       { radius of the base }
  zbase = 0.02      { thickness of the base and cone}
  rstem = 0.02      { radius of the stem }
  zstem = 0.3       { height of the stem }
  zslope = (zglass-zstem)/(router-rstem){ slope of conic surface }
  glassangle = arctan(zslope)          { slope of conic surface }
  zcone = max(0,(rad-rstem)*zslope)    { conic surface of the glass }

EXTRUSION
  surface 'bottom' z=0
    layer 'base layer'
  surface 'stem1'  z=zbase
    layer 'stem layer'
  surface 'lower'  z = zstem + zcone
    layer 'cone layer'
  surface 'upper'  z = zbase*cos(glassangle) + min(zglass, zstem + zcone)

BOUNDARIES

limited region 'outer'
    layer 'cone layer'      { outer region exists only in cone  }
    start (router,0) arc( center=0,0) angle=360

limited region 'base'
    layer 'base layer'      { base region exists only in base }
```

```
    start(rbase,0) arc(center=0,0) angle=360

limited region 'stem'
    layer 'stem layer'  { stem region exists in the stem and the bottom of the cone }
    layer 'cone layer'
    start(rstem,0) arc(center=0,0) angle=360

PLOTS
    grid(x,y,z) paintregions    as "final mesh"
    grid(y,z) on x=0 nolines paintregions as "Region Map"

END
```

6.2.26.4 3d_cylspec

```
{  3D_CYLSPEC.PDE
    This problem considers the construction of a cylindrical domain in 3D.
}

title '3D Cylinder Generator'

coordinates
    cartesian3

variables
    u

definitions
    K = 0.1                     { thermal conductivity }
    R0 = 1                      { radius of the cylinder }
    Heat = 1                    { total heat generation }
    theta = 45                  { axis direction in degrees }
    c = cos(theta degrees)      { direction cosines of the axis direction }
    s = sin(theta degrees)
    axis = vector(c,s)          { the axis direction vector }
    len = 3                     { cylinder length }
    x0 = -(len/2)*c             { beginning point of the cylinder axis }
    y0 = -(len/2)*s
    zoff = 10                   { a z-direction offset for the entire figure }

    { the cylinder function constructs the top surface of a cylinder with azis
      along z=0.5. The positive and negative values of this surface will be
      separated by a distance of one unit at the diameter. }
    zs = cylinder((x0,y0,0.5), (x0+len*c,y0+len*s, 0.5), R0)

    flux = -k*grad(u)       { heat flux vector }

equations
    U: div(K*grad(u)) + heat   = 0

extrusion
    surface z = zoff-zs         { the bottom half-surface }
    surface z = zoff+zs         { the top half-surface }

boundaries
     surface 1 value(u) = 0     { fixed value on cylinder surfaces }
     surface 2 value(u) = 0
     Region 1
        start  (x0,y0)
        value(u)=0              { fixed value on sides and end planes }
        line to (x0+R0*c,y0-R0*s)
             to (x0+len*c+R0*c,y0+len*s-R0*s)
             to (x0+len*c-R0*c,y0+len*s+R0*s)
             to (x0-R0*c,y0+R0*s)
             to close

 plots
     grid(x,y,z)  png(3072,1)
     grid(x,z) on y=0
     contour(u) on x=0 as "U on X=0"
     contour(u) on x-y=0 as "U on vertical plane through cylinder axis"
     contour(u) on x+y=0 as "U on plane normal to axis"
     vector(flux-DOT(flux,axis)*flux) on x=0  as "Flux in X=0 plane"
     contour(DOT(flux,axis)) on x=0 as "Flux normal to X=0 plane"
     contour(magnitude(flux)) on x=0 as "Total flux in X=0 plane"
     contour(magnitude(flux)) on y=0 as "Total flux in Y=0 plane"

 end
```

6.2.26.5 3d_ellipsoid

```
{ 3D_ELLIPSOID.PDE

  This problem constructs an ellipsoid.
  It is the geometric construction only, there are no variables or equations.

}

title '3D Ellipsoid'

coordinates cartesian3

definitions
  a=3  b=2  c=1   { x,y,z radii }
  xc=1 yc=1 zc=1  { coordinates of ellipsoid center }

  { top half of ellipsoid surface :
    the MAX function is used to ensure the surface is defined throughout all
    x,y space - essentially placing an x=0 'skirt' on the ellipsoid surface }

  ellipsoid = c*sqrt( max(0,1-(x-xc)^2/a^2-(y-yc)^2/b^2) )

extrusion
    surface 'bottom' z = zc - ellipsoid
    surface 'top'    z = zc + ellipsoid

boundaries
    region 'ellipse'
        start(xc+a,yc)
        arc(center=xc,yc) to (xc,yc+b) to (xc-a,yc) to (xc,yc-b) to close

plots
    grid(x,y,z)
    grid(x,y) on z=zc
    grid(y,z) on x=xc
    grid(x,z) on y=yc

end
```

6.2.26.6 3d_ellipsoid_shell

```
{ 3D_ELLIPSOID_SHELL.PDE

  This problem constructs an elliptical shell.
  It is the geometric construction only, there are no variables or equations.

}


title '3D Ellipsoid Shell'

coordinates cartesian3

definitions
  ao=3.2  bo=2.2  co=1.2   { x,y,z radii - outer ellipse }
  ai=3.0  bi=2.0  ci=1.0   { x,y,z radii - inner ellipse }
  xc=1 yc=1 zc=1  { coordinates of ellipsoid center }

  { top half of ellipsoid surface :
    the MAX function is used to ensure the surface is defined throughout all
    x,y space - essentially placing a 'skirt' on the top ellipsoid surface }

  outer_ellipsoid = co*sqrt( max(0,1-(x-xc)^2/ao^2-(y-yc)^2/bo^2) )
  inner_ellipsoid = ci*sqrt( max(0,1-(x-xc)^2/ai^2-(y-yc)^2/bi^2) )

extrusion
    surface 'outer bottom' z = zc - outer_ellipsoid
    surface 'inner bottom' z = zc - inner_ellipsoid
    surface 'inner top'    z = zc + inner_ellipsoid
    surface 'outer top'    z = zc + outer_ellipsoid

boundaries
    region 'outer ellipse'
        start(xc+ao,yc)
```

```
        arc(center=xc,yc) to (xc,yc+bo) to (xc-ao,yc) to (xc,yc-bo) to close

    limited region 'inner ellipse'
      layer 2 void
        start(xc+ai,yc)
        arc(center=xc,yc) to (xc,yc+bi) to (xc-ai,yc) to (xc,yc-bi) to close

plots
    grid(x,y,z)
    grid(x,y) on z=zc paintregions
    grid(y,z) on x=xc paintregions
    grid(x,z) on y=yc paintregions

end
```

6.2.26.7 3d_extrusion_spec

```
{ 3D_EXTRUSION_SPEC.PDE

  This descriptor is a demonstration of the grammar of 3D extrusions.
  It is a completion of the 3D specification example shown in
  "Help | Technical Notes | Extrusions in 3D" 265.
  It describes a strip capacitor fabricated as a sandwich of
  air | metal | glass | metal | air.

  Click "Controls->Domain Review" 7 to watch the domain construction process.

  See the sample problem "3D_Capacitor" 295 for a somewhat more complicated
  and interesting version.
}


TITLE '3D Extrusion Spec'

SELECT regrid=off { for quicker solution }

COORDINATES
  CARTESIAN3

DEFINITIONS
  Kdiel= 6
  Kmetal=1e6
  Kair=1
  K = Kair      { default to Kair }
  V0 = 0
  V1 = 1

VARIABLES
  V

EQUATIONS
  V:  DIV(K*GRAD(V)) = 0

EXTRUSION
  SURFACE       "Bottom"                        Z=0
    LAYER       "Bottom Air"
  SURFACE       "Bottom Air - Metal"            Z=0.9
    LAYER       "Bottom Metal"
  SURFACE       "Bottom Metal - Dielectric"     Z=1
    LAYER       "Dielectric"
  SURFACE       "Top Metal - Dielectric"        Z=2
    LAYER       "Top Metal"
  SURFACE       "Top Metal - Air"               Z=2.1
    LAYER       "Top Air"
  SURFACE       "Top"                           Z=3

BOUNDARIES
  SURFACE "Bottom" VALUE(V)=0
  SURFACE "Top" VALUE(V)=1
  REGION 1                          { this is the outer boundary of the system }
      LAYER "Dielectric" K = Kdiel  { all other layers default to Kair }
      START(0,0)
      LINE TO (5,0)  TO (5,5)  TO(0,5)  to close
  LIMITED REGION 2      { this region exists only in the "bottom metal" layer,
                          and describes the larger plate }
      LAYER "Bottom Metal" K = Kmetal
      START(1,0)
```

```
        LAYER "Bottom Metal" VALUE(V)=V0
        LINE TO (4,0)
        LAYER "Bottom Metal" NATURAL(V)=0
        Line TO (4,5) TO (1,5) to close
    LIMITED REGION 3     { this region exists only in layer "Top Metal",
                          and describes the smaller plate }
        LAYER "Top Metal" K = Kmetal
        START(2,0)
        LINE TO (3,0) TO (3,5)
        LAYER "Top Metal" VALUE(V)=V1
        LINE TO (2,5)
        LAYER "Top Metal" NATURAL(V)=0
        LINE to close

  SELECT painted
  PLOTS
    CONTOUR(V) ON X=2.5 as "V on X-cut"
    CONTOUR(V) ON Y=2.5 as "V on Y-cut"
    CONTOUR(V) ON Z=1.5 as "V on Z-cut"
    GRID(x,z) ON Y=2.5 paintregions nolines as "Region Map"
    GRID(x,z) ON Y=2.5 paintmaterials nolines as "Material Map"
    GRID(x,y,z) ON LAYER 2 ON REGION 2 as "Bottom Plate"
    GRID(x,y,z) ON "Top Metal" ON REGION 3 as "Top Plate"

  END
```

6.2.26.8 3d_fillet

```
{ 3D_FILLET.PDE

  This problem demonstrates the use of the FILLET 189 and BEVEL 189 commands.
  Both controls act in the 2D layout, and are extruded into the z dimension.
}

title 'fillet test'

coordinates
    cartesian3

variables
    u

definitions
    k = 1
    u0 = 1-x^2-y^2
    s = 2*3/4+5*2/4

equations
    U: div(K*grad(u)) +s = 0

extrusion z=0,1

boundaries
    Region 1
        start(-1,-1)
        value(u)=u0 line to (1,-1)       FILLET(0.1)
                    to (-0.25,-0.25)     FILLET(0.1)
                    to (-1,1)            BEVEL(0.1)
                    to close

monitors
    grid(x,y,z)
    contour(u) on z=0.5

plots
    grid(x,y) on z=0.005
    grid(x,y) on z=0.5
    contour(u) on z=0.5
    contour(u)  on z=0.5 zoom(0.6,-1, 0.2,0.2)
    contour(u)  on z=0.5 zoom(-0.3,-0.3, 0.1,0.1)
end
```

6.2.26.9 3d_helix_layered

```
{ 3D_HELIX_LAYERED.PDE

  This problem demonstrates the construction of a helix by layered half-turns.

  Each half-turn of the helix is represented by two layers: a layer for the coil and
  a separating layer for the gap.

  The top and bottom surfaces of the helix are formed as spiral ribbons : z=twist*angle+offset.
  The turns of the helix are divided into half-turn layers by spiral ribbons of opposite twist :
  z=offset-cuttwist*angle.

  The top surface of the lower half turn meets the bottom surface of the upper half turn
  in the region where the cut ribbon crosses the helix.  Since these two surfaces must
  be separated by a "layer", there must be an empty layer between each pair of half-turns
  of the helix.  This layer exists only in the region of contact between the two half turns,
  and in this region, the layer has zero thickness.

  In this sample problem, we solve a heat conduction problem in the helix simply
  for demonstration purposes.

  See "3d_helix_wrapped.pde" 410 for a different approach to constructing a helix.

}

title '3D layered helix'

coordinates
    cartesian3

variables
    Tp

definitions
    xwide = 1   { width of coil band }
    zhigh = 1   { height of coil band }
    zhaf = zhigh/2
    pitch = 2*zhigh { z rise per turn }
    x0 = 3      { center radius }
    xin = x0-xwide/2        { inner radius }
    xout = x0+xwide/2   { outer radius }

    { cut layers with reverse-helix. choose a steep cutpitch to avoid overlapping cut regions: }
    cutpitch = 4*pitch          { z fall per turn of layer-cutting ribbon }
    { Compute the half-angle of the baseplane projection of the intersection between the
       helix ribbon and the cut ribbon.  This determines the size of the Regions necessary
      to describe the intersections. }
    thetai = 2*pi*zhaf/(pitch+cutpitch)
    ci = cos(thetai)
    si = sin(thetai)

    twist = pitch/(2*pi)    { z-offset per radian }
    cuttwist = cutpitch/(2*pi)    {  "  }
    { measure angles from positive x-axis for right arcs and from negative x-axis
      for left arcs, to avoid jumps in the atan2 function.}
    alphar = atan2(-y,-x)-pi    ! angular position (-pi,pi) relative to positive x-axis
    alphal = atan2(y,x)-pi  ! angular position (-pi,pi) relative to negative x-axis
    { calculate layer-cut surfaces for left and right arcs (relative to the arc center) }
    rlo = -(1/4)*pitch - cuttwist*(alphar+pi/2)     ! floor value for right arc
    rhi = (1/4)*pitch - cuttwist*(alphar-pi/2)      ! ceiling value for right arc
    llo = -(1/4)*pitch - cuttwist*(alphal+pi/2)     ! floor value for left arc
    lhi = (1/4)*pitch - cuttwist*(alphal-pi/2)      ! ceiling value for left arc

    {Define functions to generate the  z position of turn n offset by h*zhaf : }
    zr(n,h) = max(rlo, min(rhi, twist*alphar + h*zhaf)) + n*pitch
    zl(n,h) = max(llo, min(lhi, twist*alphal + h*zhaf)) + n*pitch

    { Thermal source }
   Q =   10*exp(-x^2-(y-x0)^2-(z-pitch/4)^2)
    { Thermal conductivity }
    K = 1

initial values
    Tp = 0.

equations
    Tp:   div(k*grad(Tp)) + Q = 0
```

```
extrusion
    surface z=zr(-2,-1)      { right arc bottom, turn -2 }
    surface z=zr(-2,1)       { right arc top, turn -2 }
    surface  z=zl(-3/2,-1)   { left arc bottom, turn -2 }
    surface  z=zl(-3/2,1)    {left arc top, turn -2 }
    surface z=zr(-1,-1)      { right arc bottom, turn -1 }
    surface z=zr(-1,1)       { right arc top, turn -1 }
    surface  z=zl(-1/2,-1)   { left arc bottom, turn -1 }
    surface  z=zl(-1/2,1)    { left arc top, turn -1 }
    surface z=zr(0,-1)       { right arc bottom, turn 0 }
    surface z=zr(0,1)        { right arc top, turn 0 }
    surface  z=zl(1/2,-1)    { left arc bottom, turn 0 }
    surface  z=zl(1/2,1)     { left arc top, turn 0 }
    surface  z=zr(1,-1)      { right arc bottom, turn 1 }
    surface z=zr(1,1)        { right arc top, turn 1 }
    surface  z=zl(3/2,-1)    { left arc bottom, turn 1 }
    surface  z=zl(3/2,1)     { left arc top, turn 1 }
    surface  z=zr(2,-1)      { right arc bottom, turn 2 }
    surface z=zr(2,1)        { right arc top, turn 2 }
    surface  z=zl(5/2,-1)    { left arc bottom, turn 2 }
    surface  z=zl(5/2,1)     { left arc top, turn 2 }

boundaries
    surface 1 value(Tp)=0
    surface 20  value(Tp)=0

    Limited Region 1    "lower cut "
      layer 1                    {skip layer 2}
      layer 3  layer 4 layer 5   {skip layer 6}
      layer 7 layer 8 layer 9    {skip layer 10}
      layer 11 layer 12 layer 13 {skip layer 14}
      layer 15 layer 16 layer 17 {skip layer 18}
      layer 19
      start(-xout*si,-xout*ci)
        arc(center=0,0) to(xout*si,-xout*ci)
        line to (xin*si,-xin*ci)
        arc(center=0,0) to(-xin*si,-xin*ci)
        line to close

    Limited Region 2     " right arc "
      layer 1         {skip layers 2,3,4}
      layer 5         {skip layers 6,7,8}
      layer 9         {skip layers 10,11,12}
      layer 13        {skip layers 14,15,16}
      layer 17
      start(xout*si,-xout*ci)
        arc(center=0,0) to(xout*si,xout*ci)
        line to (xin*si,xin*ci)
        arc(center=0,0) to(xin*si,-xin*ci)
        line to close

    Limited Region 3     "upper cut "
      layer 1 layer 2 layer 3    {skip layer 4}
      layer 5  layer 6  layer 7  {skip layer 8}
      layer 9  layer 10  layer 11 {skip layer 12}
      layer 13  layer 14  layer 15{skip layer 16}
      layer 17  layer 18  layer 19
      start(xout*si,xout*ci)
        arc(center=0,0) to(-xout*si,xout*ci)
        line to (-xin*si,xin*ci)
        arc(center=0,0) to(xin*si,xin*ci)
        line to close

    Limited Region 4     "left arc "
      layer 3         {skip layers 4,5,6}
      layer 7         {skip layers 8,9,10}
      layer 11        {skip layers 12,13,14}
      layer 15        {skip layers 16,17,18}
      layer 19
      start(-xout*si,xout*ci)
        arc(center=0,0) to(-xout*si,-xout*ci)
        line to (-xin*si,-xin*ci)
        arc(center=0,0) to(-xin*si,xin*ci)
        line to close

monitors
      grid(x,y,z)

plots
```

```
        grid(x,y,z) paintregions
        grid(x,y,z) on regions 1,2,3 on layer 1   paintregions   as "first right arc"
        grid(x,y,z) on regions 3,4,1 on layer 3   paintregions   as "first left arc"
        grid(x,y,z) on regions 1,2,3,4 on layers 1,3 paintregions as "first full arc"

        grid(x,z) on y=0
        contour(Tp) on x=0  as "ZY Temp" painted
        contour(Tp) on z=pitch/4  as "XY Temp" painted

end
```

6.2.26.10 3d_helix_wrapped

```
{ 3D_HELIX_WRAPPED

  This problem shows the use of the function definition facility of FlexPDE to
  create a helix of square cross-section in 3D.

  The mesh generation facility of FlexPDE extrudes a 2D figure along a straight
  path in Z, so that it is not possible to directly define a helical shape.

  However, by defining a coordinate transformation, we can build a straight rod
  in 3D and interpret the coordinates in a rotating frame.

  Define the twisting coordinates by the transformation
    xt = x*cos(y/R);
    yt = x*sin(y/R);
    zt = z

  In this transformation, x and y are the coordinates FlexPDE believes it is working
  with, and they are the coordinates that move with the twisting.
  xt and yt are the "lab coordinates" of the twisted figure.

  The chain rule gives
    dF/d(xt) = (dx/dxt)*(dF/dx)  + (dy/dxt)*(dF/dy) + (dz/dxt)*(dF/dz)
  with similar rules for yt and zt.
  Some tedious algebra gives
    dx/dxt = cos(y/R)   dy/dxt = -(R/x)*sin(y/R)    dz/dxt = 0
    dx/dyt = sin(y/R)   dy/dyt =  (R/x)*cos(y/R)    dz/dyt = 0
    dx/dzt = dy/dzt = 0    dz/dzt = 1

  These relations are defined in the definitions section, and used in the equations
  section, perhaps nested as in the heat equation shown here.

  Notice that this formulation produces the upward motion by tilting the bar in
  the un-twisted space and wrapping the resulting figure around a cylinder.

  We have added a cylindrical mounting pad at each end of the helix.

}

title '3D Helix - transformation with no shear'

coordinates
    cartesian3

select
  ngrid=160    { generate enough mesh cells to resolve the twist }

variables
    Tp

definitions
    zlong = 60
    turns =   4
    pitch = zlong/turns        { z rise per turn }

    xwide = 4.5
    zhigh = 4.5
    Rc = 22 - xwide/2                 { center radius }
    alpha = y/Rc
    zstub = 5*zhigh     { rod pieces at each end }
    sturn = Rc*2*pi     { arc length per turn }
    yolap = pi*Rc*zhigh/pitch

    slong = turns*sturn { arc length of spring }
    stot = slong + 2*sturn  { add one turn at each end for rod }
```

```
    xin = Rc-xwide/2
    xout = Rc+xwide/2
    xbore = Rc/2

    { transformations }
    rise = pitch/(2*pi)     { z-rise per radian }
    c = cos(alpha)
    s = sin(alpha)
    xt = x*c
    yt = x*s
    zt = z-zlong/2

    { functional definition of derivatives }
    dxt(f) = c*dx(f) - s*(Rc/x)*dy(f)
    dyt(f) = s*dx(f) + c*(Rc/x)*dy(f)
    dzt(f) = dz(f)

    { Thermal source }
    Q = 10*exp(-(xt-Rc)^2-yt^2-zt^2)

    z1 = -zstub
    z2 = max( 0, min(zlong, pitch*y/sturn - zhigh/2))
    z3 = max(0, min(zlong, pitch*y/sturn + zhigh/2))
    z4 = zlong + zstub

initial values
    Tp = 0.

equations
    { the heat equation using transformed derivative operators }
    Tp:    dxt(dxt(Tp)) + dyt(dyt(Tp)) + dzt(dzt(Tp)) + Q = 0

extrusion z = z1, z2, z3, z4

boundaries

    Limited Region 1        { the spring }
       layer 2
       start(xin,yolap)
        line to (xout,yolap)
        line to (xout, slong-yolap)
        line to (xin,slong-yolap)
        line to close

    Limited Region 2                { top rod overlap with coil }
        surface 4      value(Tp)=0   {cold at the end of the rod }
        layer 2  layer 3
        start(xbore,slong-yolap)
        line to (xout,slong-yolap) to (xout,slong+yolap) to (xbore,slong+yolap) to close

    Limited Region 3                { top rod free of coil }
        surface 4      value(Tp)=0   {cold at the end of the rod }
        layer 2  layer 3
        start(xbore,slong+yolap)
        line to (xout,slong+yolap) to (xout,slong+sturn-yolap) to (xbore,slong+sturn-yolap)
             to close

    Limited Region 4                { bottom rod overlap with coil }
        surface 1      value(Tp)=0   {cold at the end of the rod }
        layer 1  layer 2
        start(xbore,-yolap)
        line to (xout,-yolap) to (xout,yolap) to (xbore,yolap) to close

    Limited Region 5                { bottom rod free of coil }
        surface 1      value(Tp)=0   {cold at the end of the rod }
        layer 1  layer 2
        start(xbore,-sturn+yolap)
        line to (xout,-sturn+yolap) to (xout,-yolap) to (xbore,-yolap) to close

monitors
    grid(xt,yt,zt) paintregions     { the twisted shape }

plots
    grid(xt,yt,zt) paintregions     { the twisted shape again }

    { In the following, recall that x is really radius, and y is really azimuthal distance.
       It is not possible at present to construct a cut in the "lab" coordinates. }
    grid(x,z) on y=0
    contour(Tp) on y=0  as "ZX Temp"
```

```
    contour(Tp) on z=0  as "XY Temp"
    elevation(Tp) from(Rc,0,0) to (Rc,slong,zlong)  { centerline of coil }

end
```

6.2.26.11 3d_integrals

```
{ 3D_INTEGRALS.PDE
  This problem demonstrates the specification of various integrals in 3D.
  ( This is a modification of problem  3D_BRICKS.PDE 335 )
}

title '3D Integrals'

coordinates
    cartesian3

variables
    Tp

definitions
    long = 1
    wide = 1
    K                    { thermal conductivity -- values supplied later }
    Q = 10*max(1-x^2-y^2-z^2,0)              { Thermal source }

    { These definitions create a selector that supresses evaluation
        of Tp except in region 2 of layer 2 }
    flag22=0
    check22 = if flag22>0 then Tp else 0

    { These definitions create a selector that supresses evaluation
        of Tp except in region 2 of all layers }
    flag20=0
    check20 = if flag20>0 then Tp else 0

initial values
    Tp = 0.

equations
    Tp:    div(k*grad(Tp)) + Q = 0        { the heat equation }

extrusion
    surface "bottom" z = -long
      layer 'bottom'
    surface "middle" z=0
      layer 'top'
    surface 'top' z= long   { divide Z into two layers }

boundaries
    surface 1 value(Tp)=0   { fix bottom surface temp }
    surface 3 value(Tp)=0   { fix top surface temp }

    Region 1        { define full domain boundary in base plane }
       layer 1 k=1      { bottom right brick }
       layer 2 k=0.1        { top right brick }
       start "outside" (-wide,-wide)
         value(Tp) = 0      { fix all side temps }
         line to (wide,-wide)   { walk outer boundary in base plane }
           to (wide,wide)
           to (-wide,wide)
           to close

    Region 2 "Left"     { overlay a second region in left half }
       flag20=1
       layer 1 k=0.2        { bottom left brick }
       layer 2 k=0.4  flag22=1  { top left brick }
       start(-wide,-wide)
         line to (0,-wide)      { walk left half boundary in base plane }
           to (0,wide)
           to (-wide,wide)
           to close

monitors
    contour(Tp) on surface z=0  as "XY Temp"
    contour(Tp) on surface x=0  as "YZ Temp"
    contour(Tp) on surface y=0  as "ZX Temp"
    elevation(Tp) from (-wide,0,0) to (wide,0,0)  as "X-Axis Temp"
```

```
    elevation(Tp) from (0,-wide,0) to (0,wide,0)  as "Y-Axis Temp"
    elevation(Tp) from (0,0,-long) to (0,0,long)  as "Z-Axis Temp"

plots
    contour(Tp) on z=0  as "XY Temp"
    contour(Tp) on x=0  as "YZ Temp"
    contour(Tp) on y=0  as "ZX Temp"
    contour(k*dz(Tp)) on z=-0.001  as "Low Middle Z-Flux"
    contour(k*dz(Tp)) on z=0.001  as "High Middle Z-Flux"

    summary
      report("Compare various forms for integrating over region 2 of layer 2")
      report(integral(Tp,2,2))
      report(integral(Tp,"Left","Top"))
      report(integral(check22))
      report '-----'

      report("Compare various forms for integrating over region 2 in all layers")
      report(integral(Tp,2,0))
      report(integral(check20))
      report '-----'

      report("Compare various forms for integrating over total volume")
      report(integral(Tp,"ALL","ALL"))
      report(integral(Tp))
      report '-----'

      report("Compare various forms for integrating over surface 'middle'")
      report(sintegral(normal(-k*grad(Tp)),2))
      report(sintegral(normal(-k*grad(Tp)),'Middle'))
      report(sintegral(-k*dz(Tp),2))
      report '-----'

      report("Compare various forms for integrating over surfaces")
      report(sintegral(normal(-k*grad(Tp)),1))  as "Bottom Flux"
      report(sintegral(normal(-k*grad(Tp)),3))  as "Top Flux"
      report(sintegral(normal(-k*grad(Tp)),"outside")) as "Side Flux"
      report(sintegral(normal(-k*grad(Tp)),1)+sintegral(normal(-k*grad(Tp)),3)
            +sintegral(normal(-k*grad(Tp)),"outside")) as "Bottom+Top+Side Flux"
      report(sintegral(normal(-k*grad(Tp))))    { surface integral on total outer surface }
      report(integral(Q) )  as "Source Integral"
      report(sintegral(normal(-k*grad(Tp)),"outside")
           +sintegral(normal(-k*grad(Tp)),1)+sintegral(normal(-k*grad(Tp)),3)
            -integral(Q) )  as "Energy Error"
      report(integral(div(-k*grad(Tp))) )   as "Divergence Integral"
      report '-----'

      report("Compare surface flux on region 2 of layer 2 to internal divergence integral")
      { surface integral over outer surface of region 2, layer 2 }
      report(sintegral(normal(-k*grad(Tp)),"Left","Top"))
      report(integral(Q,"Left","Top"))
      report '-----'

end
```

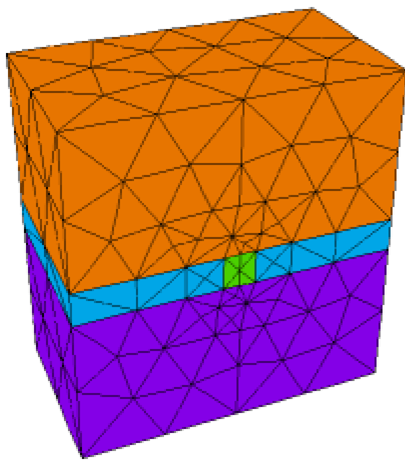
6.2.26.12 3d_lenses

```
{ 3D_LENSES.PDE

  This problem considers the flow of heat in a lens-shaped body
  of square outline.  It demonstrates the use of FlexPDE in problems
  with non-planar extrusion surfaces.

  Layer 1 consists of a flat bottom with a paraboloidal top.
  Layer 2 is a paraboloidal sheet of uniform thickness.

  Plots on various cut planes show the ability of FlexPDE to
  detect intersection surfaces.
```

```
}
  title '3D Test - Lenses'

  coordinates
     cartesian3

  Variables
     u

  definitions
     k = 0.1
     heat = 4

  equations
    U: div(K*grad(u)) + heat   = 0

  extrusion
    surface z = 0
    surface z = 0.8-0.3*(x^2+y^2)
    surface z = 1.0-0.3*(x^2+y^2)


 boundaries
     { implicit natural(u) = 0 on top and bottom faces }
     Region 1
        layer 2 k = 1    { layer specializations must follow regional defaults }
        start(-1,-1)
        value(u) = 0     { Fixed value on sides }
        line to (1,-1) to (1,1) to (-1,1) to close

 select painted
 plots
     contour(u) on x=0.51       as "YZ plane"
     contour(u) on y=0.51       as "XZ plane"
     contour(u) on z=0.51       as "XY plane cuts both layers and part of outline"
     contour(u) on z=0.75       as "XY plane cuts both layers, but not the outline"
     contour(u) on z=0.8        as "XY plane cuts only layer 2"
     contour(u) on z=0.95       as "XY plane cuts small patch of layer 2"
     contour(u) on z=0.95 zoom  as "small cut patch, zoomed to fill frame"
     contour(u) on surface 1    as "on bottom surface"
     contour(u) on surface 2    as "on paraboloidal layer interface"
     contour(u) on x=y          as "oblique plot plane"
     contour(u) on x+y=0        as "another oblique plot plane"

 end
```

6.2.26.13 3d_limited_region

```
{ 3D_LIMITED_REGION.PDE

  This example shows the use of LIMITED REGIONS 186 in 3D applications.

  The LIMITED qualifier applied to a REGION section tells FlexPDE to construct
  the region only in those layers or surfaces specifically referenced in the region
  definition.

  In this problem, we have a heat equation with a small cubical heated box in the middle layer.

}
```



```
title '3D LIMITED REGION TEST'

coordinates
    cartesian3

select
    errlim = 0.005
    ngrid=1  { exaggerate cell size disparity }

variables
    u

definitions
    k = 0.1
    h=0
    Lx=3    Ly=3  Lz=3
    w = 0.15        { box size }
    x0=Lx/2-w  y0=Ly/2-w  z0=Lz/2-w { box coords }
    x1=Lx/2+w  y1=Ly/2+w  z1=Lz/2+w

equations
    U:  div(K*grad(u)) + h = 0

extrusion z=0, z0, z1, Lz

boundaries
    Region 1
      start(0,y0)
      value(u)=0
      line to (Lx,y0) to (Lx,Ly) to (0,Ly) to close

    limited region 2        { insert exists only on layer 2 }
      layer 2 K=9 h=1
      start(x0,y0)
      line to (x1,y0) to (x1,y1) to (x0,y1) to close

monitors
    grid(x,z) on y=Ly/2
    contour(u) on z=Lz/2

plots
    grid(x,z) on y=Ly/2
    contour(u) on z=Lz/2  painted
    contour(u) on y=Ly/2  painted

end
```

6.2.26.14 3d_pinchout

```
{ 3D_PINCHOUT.PDE

  This problem demonstrates the merging
  of extrusion surfaces and the
  'Pinch-Out' of a layer.
}

title '3D Layer Pinch-out Test'

coordinates
    cartesian3

variables
    Tp

definitions
    long = 1
    wide = 1
    K = 1     { thermal conductivity default -- other values supplied later: }
    Q = 10*exp(-x^2-y^2-z^2) { Thermal source }
    z1 = 0
    z2        { surface will be defined later in each region: }
    z3 = 1

initial values
    Tp = 0.

equations
```

```
    Tp:  div(k*grad(Tp)) + Q = 0        { the heat equation }

extrusion z = z1,z2,z3      { divide Z into two layers }

boundaries
    surface 1 value(Tp)=0       { fix bottom surface temp }
    surface 3 value(Tp)=0       { fix top surface temp }

    Region 1            { define full domain boundary in base plane }
       z2 = 1       { surface 2 merges with surface 3 in this region }
       start(-wide,-wide)
         value(Tp) = 0          { fix all side temps }
         line to (wide,-wide)   { walk outer boundary in base plane }
           to (wide,wide)
           to (-wide,wide)
           to close

    Region 2            { Overlay a second region in left half.
            This region delimits the area in which surfaces 2 and 3 differ.
            Surfaces meet at the region boundary.  }
       z2 = 1 + x/2
       layer 2 k=0.1    { redefine conductivity in layer 2 of region 2 }
       start(-wide,-wide)
         line to (0,-wide)              { walk left half boundary in base plane }
           to (0,wide)
           to (-wide,wide)
           to close

monitors
    grid(x,z) on y=0

plots
    grid(x,z) on y=0
    contour(Tp) on y=0  as "ZX Temp"

end
```

6.2.26.15 3d_planespec

```
{ 3D_PLANESPEC.PDE

  This problem demonstrates the use of the PLANE 179 generating function in
  3D domain specifications.

  We construct a hexahedron using two PLANE 179 statements.

}

title 'PLANE surface generation'

coordinates
    cartesian3

variables
    Tp

definitions
    long = 1
    wide = 1
    K   = 1
    Q = 10*exp(-x^2-y^2-z^2)

    { define three points in the plane surface }
    bll = point(-1,-1,0)
    blr = point(1,-1,0.2)
    bul = point(-1,1,0.3)

initial values
    Tp = 0.

equations
    Tp:  div(k*grad(Tp)) + Q = 0

extrusion
    { bottom surface using named points }
    surface 'bottom'  z = PLANE(bll,blr,bul)
    { top surface using explicit points }
```

```
    surface 'top'     z = PLANE((-1,-1,1), (1,-1,1.2), (1,1,2))

boundaries
    surface 1 value(Tp)=0
    surface 2 value(Tp)=0

    Region 1
       start(-wide,-wide)
         value(Tp) = 0
         line to (wide,-wide)
           to (wide,wide)
           to (-wide,wide)
           to close

monitors
    grid(x,z) on y=0

plots
    grid(x,y,z)   viewpoint(-7,-9,10)
    grid(x,z) on y=0
    contour(Tp) on y=0  as "ZX Temp"
    contour(Tp) on x=0  as "YZ Temp"

end
```

6.2.26.16 3d_pyramid

```
{ 3D_PYRAMID.PDE

  This problem considers the flow of heat in a pyramid-shaped body.
  It demonstrates the use of FlexPDE in 3D problems with non-planar
  extrusion surfaces.

  Note that FEATURE 187 paths are used to delineate discontinuities in the
  extrusion surfaces.

  The outer edge is used as a heat source, so it is clipped to form an edge wall.


}
title '3D Test - Pyramid'

coordinates
    cartesian3

select
    regrid=off

variables
    u

definitions
    k = 0.1
    heat = 4

equations
    U: div(K*grad(u)) + heat   = 0

extrusion
    surface z = 0
    surface z = min(1.1- abs(x),1.1-abs(y))


boundaries
     { implicit natural(u) = 0 on top and bottom faces }
    Region 1
        start(-1,-1)
        value(u) = 0            { Fixed value on short vertical sides }
            line to (1,-1) to (1,1) to (-1,1) to close

    { Features delineate hidden discontinuities in surface slope.
      This forces gridding nodes along break lines. }
    feature start(-1,-1) line to (1,1)
    feature start(-1,1) line to (1,-1)

plots
    contour(u) on x=0          as "YZ plane intersects peak"
```

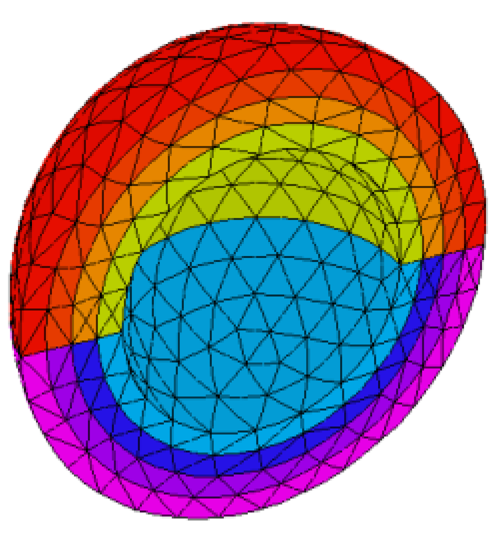
```
    contour(u) on y=0           as "XZ plane intersects peak"
    contour(u) on z=0.1         as "XY plane intersects full outline"
    contour(u) on x=0.51        as "YZ plane near midpoint of side slope"
    contour(u) on x+y=0.51      as "Oblique plane cuts corner"
    contour(u) on z=0.8         as "XY plane near tip"
    contour(u) on z=0.8  zoom   as "XY plane near tip - zoomed"

 end
```

6.2.26.17 3d_shell

```
{ 3D_SHELL.PDE

  This problem considers heatflow in a
  spherical shell.

  We solve a heatflow equation with
  fixed temperatures on inner and outer
  shell surfaces.
}

title '3D Test - Shell'

coordinates
    cartesian3

variables
    u

select
    aspect=8    { allow long thin cells in joint between hemispheres }
    autostage=off

definitions
    k = 10                      { conductivity }
    heat =6*k                   { internal heat source }
    rad=sqrt(x^2+y^2)
    R1 = 1
    thick = staged(0.1,0.03,0.01)
    R2 = R1-thick

equations
    U: div(K*grad(u)) + heat   = 0

extrusion
    surface z =  -SPHERE ((0,0,0),R1)      { the bottom hemisphere }
    surface z =  -SPHERE ((0,0,0),R2)
    surface z =  SPHERE ((0,0,0),R2)
    surface z =  SPHERE ((0,0,0),R1)       { the top hemisphere }

boundaries

    surface 1 value(u) = 0      { fixed values on outer sphere surfaces }
    surface 4 value(u) = 0

    Region 1   { The outer boundary in the base projection }
        layer 1 k=0.1  mesh_spacing=10*thick { force resolution of shell curve }
        layer 2 k=0.1
        layer 3 k=0.1  mesh_spacing=10*thick
        start(R1,0)
        value(u) = 0            { Fixed value on outer vertical sides }
            arc(center=0,0) angle=180
        natural(u)=0 line to close

     Limited Region 2   { The inner cylinder shell boundary in the base projection }
       surface 2 value(u) = 1   { fixed values on inner sphere surfaces }
       surface 3 value(u) = 1
        layer 2 void            { empty center }
        start(R2,0)
        arc(center=0,0) angle=180
        nobc(u) line to close

monitors
     grid(x,y,z)
     grid(x,z) on y=0
     grid(rad,z) on x=y
     contour(u) on x=0          { YZ plane through diameter }
     contour(u) on y=0          { XZ plane through diameter }
```

```
      contour(u) on z=0          { XY plane through diameter }
      contour(u) on x=0.5        { YZ plane off center }
      contour(u) on y=0.5        { XZ plane off center }

definitions
      yp = 0.5
      rp = sqrt(R2^2-yp^2)
      xp = rp/sqrt(2+thick)
plots
      grid(x,y,z)
      grid(x,z) on y=0
      contour(u) on x=0          as "Temp on YZ plane through diameter"
      contour(u) on y=0          as "Temp on XZ plane through diameter"
      contour(u) on z=0          as "Temp on XY plane through diameter"
      contour(u) on z=0.001         as "Temp on XY plane through diameter"
      contour(u) on x=0.5        as "Temp on YZ plane off center"
      contour(u) on y=0.5        as "Temp on XZ plane off center"
      contour(magnitude(grad(u))) on y=yp
                                 zoom(xp,xp, thick*sqrt(2+thick),thick*sqrt(2+thick))
                                 as "Flux on XZ plane off center"

 end
```

6.2.26.18 3d_shells



```
{ 3D_SHELLS.PDE

  This problem demonstrates the construction
  of multiple nested spherical shells.

  We solve a heatflow equation with fixed
  temperatures on inner and outer
  shell surfaces.
}

title 'Nested 3D Shells'

coordinates
    cartesian3

variables
    u

definitions
    k = 10
    heat =6*k
    rad=sqrt(x^2+y^2)
    R1 = 1
    thick = 0.1
    R2 = R1-thick
    R3 = R2-thick
    R4 = R3-thick
    R5 = R4-thick

equations
    U: div(K*grad(u)) + heat   = 0


extrusion
    surface 'SB1' z =  -SPHERE ((0,0,0),R1)      { the bottom hemisphere }
  layer 'LB1'
    surface 'SB2' z =  -SPHERE ((0,0,0),R2)
  layer 'LB2'
    surface 'SB3' z =  -SPHERE ((0,0,0),R3)
  layer 'LB3'
    surface 'SB4' z =  -SPHERE ((0,0,0),R4)
  layer 'LB4'
    surface 'SB5' z =  -SPHERE ((0,0,0),R5)
  layer 'LB5'
    surface 'ST5' z =  SPHERE ((0,0,0),R5)
  layer 'LT4'
    surface 'ST4' z =  SPHERE ((0,0,0),R4)
  layer 'LT3'
    surface 'ST3' z =  SPHERE ((0,0,0),R3)
  layer 'LT2'
    surface 'ST2' z =  SPHERE ((0,0,0),R2)
  layer 'LT1'
    surface 'ST1' z =  SPHERE ((0,0,0),R1)       { the top hemisphere }
```

```
boundaries

    surface 'SB1' value(u) = 0      { fixed values on outer sphere surfaces }
    surface 'ST1' value(u) = 0

    Region 1
        layer 'LB1' k=1
        layer 'LT1' k=1
         start(R1,0)
        value(u) = 0
            arc(center=0,0) angle=180
        natural(u)=0 line to close

     Limited Region 2
        layer 'LB2' k=2
        layer 'LT2'  k=2
         ! include the region in all layers that must merge out:
         layer 'LB3'  layer 'LB4' layer 'LB5' layer 'LT4' layer 'LT3'
        start(R2,0)
        arc(center=0,0) angle=180
        nobc(u) line to close

     Limited Region 3
        layer 'LB3'  k=3
        layer 'LT3'  k=3
         ! include the region in all layers that must merge out:
         layer 'LB4' layer 'LB5' layer 'LT4'
        start(R3,0)
        arc(center=0,0) angle=180
        nobc(u) line to close

     Limited Region 4
        layer 'LB4'  k=4
        layer 'LT4'  k=4
         ! include the region in all layers that must merge out:
         layer 'LB5'
        start(R4,0)
        arc(center=0,0) angle=180
        nobc(u) line to close

     Limited Region 5
        surface 'SB5' value(u) = 1   { fixed values on inner sphere surfaces }
        surface 'ST5' value(u) = 1
        layer 'LB5' void            { empty center }
        start(R5,0)
        arc(center=0,0) angle=180
        nobc(u) line to close

monitors
     grid(x,y,z)
     grid(x,z) on y=0
     grid(rad,z) on x=y
     contour(u) on x=0          { YZ plane through diameter }
     contour(u) on y=0          { XZ plane through diameter }
     contour(u) on z=0          { XY plane through diameter }
     contour(u) on x=0.5        { YZ plane off center }
     contour(u) on y=0.5        { XZ plane off center }

definitions
     yp = 0.5
     rp = sqrt(R2^2-yp^2)
     xp = rp/sqrt(2+thick)
plots
     grid(x,y,z)
     grid(x,z) on y=0
     contour(u) on x=0          as "Temp on YZ plane through diameter"
     contour(u) on y=0          as "Temp on XZ plane through diameter"
     contour(u) on z=0          as "Temp on XY plane through diameter"
     contour(u) on z=0.001         as "Temp on XY plane through diameter"
     contour(u) on x=0.5        as "Temp on YZ plane off center"
     contour(u) on y=0.5        as "Temp on XZ plane off center"
     contour(magnitude(grad(u))) on y=yp
                                zoom(xp,xp, thick*sqrt(2+thick),thick*sqrt(2+thick))
                                as "Flux on XZ plane off center"

 end
```

6.2.26.19 3d_sphere

```
{ 3D_SPHERE.PDE

  This problem considers the construction of a spherical domain in 3D.

  The heat equation is  Div(-K*grad(U)) = h, wth U the temperature and
  h the volume heat source.

  A sphere with uniform heat source h will generate a total amount of heat
    H = (4/3)*Pi*R^3*h, from which  h = 3*H/(4*Pi*R^3).

  The normal flux at the surface will be Fnormal = -K*grad(U) <dot> Normal,
  where Normal is the surface-normal unit vector.  On the sphere, the unit
  normal is [x/R,y/R,z/R].
  At the surface, the flux will be uniform, so the surface integral of flux is
    TOTAL = 4*pi*R^2*normal(-K*grad(U)) = H
  or  normal(-K*grad(u)) = H/(4*pi*R^2)  =  R*h/3.

  In the following, we set R=1 and H = 1, from which
    h = 3/(4*pi)
    normal(-k*grad(u)) = 1/(4*pi)
}

title '3D Sphere'

coordinates
    cartesian3

variables
    u

definitions
    K = 0.1   { conductivity }
    R0 = 1    { radius }
    H0 = 1    { total heat }
    { volume heat source }
    heat =3*H0/(4*pi*R0^3)

equations
    U: div(K*grad(u)) + heat   = 0

extrusion
    surface z = -SPHERE ((0,0,0),R0)       { the bottom hemisphere }
    surface z = SPHERE ((0,0,0),R0)        { the top hemisphere }

boundaries
    surface 1 value(u) = 0     { fixed value on sphere surfaces }
    surface 2 value(u) = 0
    Region 1
        start(R0,0)
        arc(center=0,0) angle=360

plots
    grid(x,y,z)
    grid(x,z) on y=0
    contour(u) on x=0
    contour(4*pi*magnitude(k*grad(u))) on x=0
    contour(4*pi*magnitude(k*grad(u))) on y=0
    contour(-4*pi*k*(x*dx(u)+y*dy(u)+z*dz(u))/sqrt(x^2+y^2+z^2)) on x=0  as "normal flux"
    contour(-4*pi*k*(x*dx(u)+y*dy(u)+z*dz(u))/sqrt(x^2+y^2+z^2)) on y=0  as "normal flux"
    vector(-grad(u)) on x=0
    vector(-grad(u)) on y=0

    contour(4*pi*normal(-k*grad(u))) on surface 1  as "4*pi*Normal Flux=1"  { bottom surface }
    contour(4*pi*normal(-k*grad(u))) on surface 2  as "4*pi*Normal Flux=1"  { top surface }
    surface(4*pi*normal(-k*grad(u))) on surface 1  as "4*pi*Normal Flux=1"  { bottom surface }
    surface(4*pi*normal(-k*grad(u))) on surface 2  as "4*pi*Normal Flux=1"  { top surface }

    summary
      report(sintegral(normal(-k*grad(u)),1)) as "Bottom current :: 0.5 "
      report(sintegral(normal(-k*grad(u)),2)) as "Top current :: 0.5 "
      report(vintegral(heat)) as "Total heat :: 1"
      report(sintegral(normal(-k*grad(u)))) as "Total Flux :: 1"

 end
```

6.2.26.20 3d_spherebox

```
{ 3D_SPHEREBOX.PDE

  An empty 3D sphere inside a box.
}

title 'Empty 3D Sphere in a box'

coordinates
    cartesian3

variables
    u

definitions
    K = 0.1                     { conductivity }
    R0 = 1                      { radius }
    box = 2*R0

    zsphere = SPHERE ((0,0,0),R0)    {  hemisphere shape }

equations
    U: div(K*grad(u))     = 0

extrusion
    surface z=-box
    surface z = -zsphere        { the bottom hemisphere and plane }
    surface z = zsphere         { the top hemisphere and plane }
    surface z=box

boundaries
    surface 1 value(u) = 0      { fixed value on box surfaces }
    surface 4 value(u) = 1

    Region 1        { the bounding box - defaults to insulating sidewalls }
        start(-box,-box)
        line to (box,-box) to (box,box) to (-box,box) to close

    Limited Region 2            { sphere exists only in region 2 }
        layer 2 void            { ... and layer 2 }
        start  (R0,0)
            arc(center=0,0) angle=360

plots
    grid(x,y,z)
    grid(x,z) on y=0
    contour(u) on x=0

end
```

6.2.26.21 3d_spherespec

```
{ 3D_SPHERESPEC

  This problem demonstrates the use of the SPHERE[179] function for construction
  of a spherical domain in 3D.  It is a modification of the example problem 3D_SPHERE.PDE[421].

}

title '3D Sphere'

coordinates
    cartesian3

variables
    u

definitions
    K = 0.1                 { conductivity }
    R0 = 1                  { radius }
    H0 = 1                  { total heat input }

    heat =3*H0/(4*pi*R0^3)  { volume heat source }
    zs = sphere((0,0,0),R0) { the top hemisphere }

equations
```

```
    U: div(K*grad(u)) + heat   = 0

extrusion
    surface z = -zs          { the bottom hemisphere }
    surface z = zs           { the top hemisphere }

boundaries
    surface 1 value(u) = 0  { fixed value on sphere surfaces }
    surface 2 value(u) = 0
    Region 1
        start  (R0,0)
        arc(center=0,0) angle=360

plots
    grid(x,y,z)
    grid(x,z) on y=0
    contour(u) on x=0
    contour(4*pi*magnitude(k*grad(u))) on x=0
    contour(4*pi*magnitude(k*grad(u))) on y=0
    contour(-4*pi*k*(x*dx(u)+y*dy(u)+z*dz(u))/sqrt(x^2+y^2+z^2)) on x=0  as "normal flux"
    contour(-4*pi*k*(x*dx(u)+y*dy(u)+z*dz(u))/sqrt(x^2+y^2+z^2)) on y=0  as "normal flux"
    vector(-grad(u)) on x=0
    vector(-grad(u)) on y=0

    contour(4*pi*normal(-k*grad(u))) on surface 1 as "4*pi*Normal Flux=1" { bottom surface }
    contour(4*pi*normal(-k*grad(u))) on surface 2 as "4*pi*Normal Flux=1" { top surface }
    surface(4*pi*normal(-k*grad(u))) on surface 1 as "4*pi*Normal Flux=1" { bottom surface }
    surface(4*pi*normal(-k*grad(u))) on surface 2 as "4*pi*Normal Flux=1" { top surface }

    summary
      report(sintegral(normal(-k*grad(u)),1)) as "Bottom current :: 0.5 "
      report(sintegral(normal(-k*grad(u)),2)) as "Top current :: 0.5 "
      report(vintegral(heat)) as "Total heat :: 1"
      report(sintegral(normal(-k*grad(u)))) as "Total Flux :: 1"

 end
```

6.2.26.22 3d_spool

```
{ 3D_SPOOL.PDE

  This example shows the use of LIMITED REGIONS 186 to construct a spool in a box in 3D.
  The core of the spool has a section of low conductivity at the center.
  The LAYER 71 structure is as follows:
    Layers 1 and 7 are the sections of the box above and below the spool
    Layers 2 and 6 are the flanges of the spool and the box area surrounding the flanges.
    Layers 3 and 5 are the high-conductivity portions of the core and the surrounding box area.
    Layer 4 is the low-conductivity portion of the core and the surrounding box area.

  Click "Controls|Domain Review" 7 or the "Domain Review" 10 tool to watch the mesh
construction.

}
title '3D LIMITED REGION EXAMPLE'

coordinates
    cartesian3

Variables
    U

definitions
    K
    K1 = 1
    K2 = 10
    K3 = 0.01
    Lx = 1   Ly = 1 Lz = 1
    {extrusion values}
    t  = 0.25
    m = 0.05
    h = 0.25
    z0 = t/2
    z1 = t/2 + m
    z2 = t/2 + m + h
    z3 = t/2 + m + h + 2*m
    z4 = t/2 + m + h + 2*m + h
    z5 = t/2 + m + h + 2*m + h + m
```

```
    {radii}
    rad = 0.5 - h/2
    rad1 = 0.5 - h/1

    {boundary values}
    U0 = 0
    U1 = 1

equations
    U:  DIV(K*GRAD(U)) = 0

extrusion
    surface "bottom of box" z=0
        layer "bottom gap"
    surface "spool bottom" z=z0
        layer "bottom flange"
    surface "top of bottom flange" z=z1
        layer "bottom core section"
    surface "bottom of core insert" z=z2
        layer "core insert"
    surface "top of core insert" z=z3
        layer "top core section"
    surface "bottom of top flange" z=z4
        layer "top flange"
    surface "top of spool" z=z5
        layer "top gap"
    surface "top of box" z=1

boundaries
    Surface 1 Value(U)=U0
    Surface 8 Value(U)=U1

    Region 1 "Box"
        K = K1
        start(0,0)
        line to (1,0) to (1,1) to (0,1) to close

    limited region 2  "Flanges"
        layer 2          K =K2
        layer 6          K =K2

        START (1/2,rad1)
        ARC(CENTER=1/2,1/2) ANGLE=360
        TO CLOSE

    limited Region 3    "Core"
        layer 3          K =K2
        layer 4          K =K3
        layer 5          K =K2

        START (1/2,rad)
        ARC(CENTER=1/2,1/2) ANGLE=360
        TO CLOSE

MONITORS
    plots
    grid(x,z) on y=0.5 paintregions
    contour(U) on y=0.5
    contour(U) on z=0.5
    contour(K) on x=0.5 painted

end
```

6.2.26.23 3d_thermocouple

```
{  3D_THERMOCOUPLE.PDE

   This problem constructs a thermocouple inside a box.
   It is the geometric construction only, there are no variables or equations.

   Thermocouple rods are inserted exactly half way into the sphere. Rod tops are rounded.
   Partial insertion is more difficult to generate the appropriate surfaces.

}

Title 'Thermocouple'
```

```
Coordinates Cartesian3

Definitions

  len = 10    ! length of rods
  rr = 1      ! radius of rods
  rs = 3      ! radius of sphere
  b = 1       ! box offset
  d = 0.5     ! distance between rods

  h = sqrt(rr^2 - (2*rs)^2)  ! additional height from top of rod to center of sphere
  c = len + h                ! z value for center of sphere
  xr = rr+d/2                ! x center for rods

  zsphere = sphere((0,0,0),rs)       ! top sphere surface at origin (untranslated)
  rsphere1 = sphere((-xr,0,0),rr)    ! rod1 sphere surface at z=0 (untranslated)
  rsphere2 = sphere((xr,0,0),rr)     ! rod2 sphere surface at z=0 (untranslated)

  zrods = c   ! regionally defined surface with default value of c
  k = 1       ! regionally defined material property with default value of 1

Extrusion

  Surface 'box bottom' z = -b
  Surface 'rod bottom' z = 0
  Surface 'sphere bottom' z = c - zsphere
  Surface 'rod top' z = zrods
  Surface 'sphere top' z = c + zsphere
  Surface 'box top' z = c + rs + b

Boundaries

  Region 'box'
    start(b+rs,b+rs)
    line to (-b-rs,b+rs) to (-b-rs,-b-rs) to (b+rs,-b-rs) to close

  Limited Region 'sphere'
    layer 3    k = 2
    layer 4    k = 2
    start(rs,0)
    arc(center=0,0) angle = 360

  Limited Region 'rod1'
    zrods = c + rsphere1
    layer 2    k = 3
    layer 3    k = 3
    start(-xr,rr)
    arc(center=-xr,0) angle = 360

  Limited Region 'rod2'
    zrods = c + rsphere2
    layer 2    k = 4
    layer 3    k = 4
    start(xr,rr)
    arc(center=xr,0) angle = 360

Plots
  grid(x,y,z) on region 'rod1' on region 'rod2'
  grid(x,y,z) on region 'sphere' on region 'rod1' on region 'rod2'
  grid(x,y,z)
  grid(x,z) on y=0

End
```

6.2.26.24 3d_toggle

```
{ 3D_TOGGLE.PDE

  This problem shows the use of curved extrusion surfaces and VOID 186 layers
  to construct a transverse cylindrical hole in an upright cylinder.
```

```
  The domain consists of three layers:
    1) the cylinder below the hole
    2) the hole
    3) the cylinder above the hole.
  Layer 2 has zero thickness outside the
  hole region, and is VOID[186] (excluded
  from the mesh) inside the hole.

  Click "Controls->Domain Review"[7] to watch
  the domain construction process.

}

title '3D CYLINDRICAL VOID LAYER TEST'

coordinates
    cartesian3

select
    errlim = 0.005

variables
    u

definitions
    k = 0.1
    h = 1
    L = 1
    Ro = 1          { the cylinder radius }
    Ri = Ro/2       { the hole radius }
    { the base-plane Y-coordinate of the intersection of the hole projection with the
    cylinder projection: }
    Yc = sqrt(Ro^2-Ri^2)
    Z4 = L          { Z-height of the cylinder top }
    { the Z-shape function for the hole top (zero beyond +-Ri): }
    Z3 = CYLINDER ((0,1,0), (0,-1,0), Ri)
    { the Z-shape function for the hole bottom (zero beyond +-Ri): }
    Z2 = -Z3
    Z1 = -L         { Z-height of the cylinder bottom }

equations
    U: div(K*grad(u)) + h = 0       { a heat equation for demonstration purposes }

extrusion z=Z1,Z2,Z3,Z4     { short-form specification of the extrusion surfaces }

boundaries

    Region 1        { this region is the projection of the outer cylinder shape }
      start(Ro,0)
      value(u)=0                { Force U=0 on perimeter }
      arc(center=0,0) angle=360  to close

    limited region 2        { this region is the projection of the transverse hole }
      layer 2 void          { the region exists only in layer 2.  Its bounding surfaces
                                merge beyond the edges of the hole  }
      start(Ri,Yc) arc(center=0,0) to (-Ri,Yc)
      line to (-Ri,-Yc)
      arc(center=0,0) to (Ri,-Yc)
      line to close

monitors
    grid(x,y,z)
    elevation(u) from (-Ro,0,0) to (Ro,0,0)
    contour(u) on z=0
    contour(u) on y=0

plots
    grid(x,y,z)
    elevation(u) from (-Ro,0,0) to (Ro,0,0)
    contour(u) on z=0
    contour(u) on y=0

end
```

6.2.26.25 3d_torus

```
{ 3D_TORUS.PDE

   This problem constructs a torus.
   The top surface and bottom surface meet along the diameter of the torus.
}

title '3D Torus'

coordinates
    cartesian3

select
    errlim = 0.005
    ngrid = 20    { get better mesh resolution of curved surfaces }
    painted

variables
    u

definitions
    Raxis = 4           { the radius of the toroid axis }
    Rtube = 1           { the radius of the toroid tube }
    Rad = sqrt(x^2+y^2) { cylindrical radius of point (x,y) }
    { the torus surface is the locus of points where (Rad-Raxis)^2+Z^2 = Rtube^2 }
    ZTorus = sqrt(Rtube^2-(Rad-Raxis)^2)

equations
    U:  del2(u) + 1 = 0

extrusion
    Surface "Bottom" z = -ZTorus
    Surface "Top" z = ZTorus

boundaries
    surface 1 value(u)=0
    surface 2 value(u) = 0

region 1
    start(Raxis+Rtube, 0)
      value(u) = 0
      arc(center=0,0) angle=360    { the outer boundary }
    start(Raxis-Rtube, 0)
      value(u) = 0
      arc(center=0,0) angle=360    { the inner boundary }

monitors
    grid(x,y,z)
    contour(u) on surface z=0
    contour(u) on surface y=0

plots
    grid(x,y,z)
    contour(u) on surface z=0
    contour(u) on surface y=0

end
```

6.2.26.26 3d_torus_tube

```
{ 3D_TORUS_TUBE.PDE

  This problem constructs a "U" of pipe by connecting two cylindrical stubs to the
  ends of a 180-degree arc of a torus.

  There are three layers:
        1) the bottom half of the outer pipe
        2) the inner fluid
        3) the top half of the outer pipe.
  Layers 1 and 3 wrap around layer 3 and meet on the center plane.

  There are six regions, the inside and outside parts of the torus and the two stubs.

}

title '3D Torus Tube'
```

```
coordinates
    cartesian3

select
    errlim = 0.005
    painted

variables
    u

definitions
    Ra = 4              { the radius of the toroid axis }
    Rt = 1              { the radius of the outer toroid }
    Ri = 0.6                  { the radius of the inner toroid }
    Len = 3                   { the length of the side tubes }

    { Surface Definitions - Toroids and Tubes}
    Rad = sqrt(x^2+y^2)
    ZTorus1 = sqrt(Rt^2-(Rad-Ra)^2) ! outside toroid
    ZTorus2 = sqrt(Ri^2-(Rad-Ra)^2) ! inside toroid

    ZTube1a = CYLINDER ((Ra,0,0), (Ra,1,0), Rt)         ! outside tube A
    ZTube1b = CYLINDER ((-Ra,0,0), (-Ra,1,0), Rt)         ! outside tube B

    ZTube2a = CYLINDER ((Ra,0,0), (Ra,1,0), Ri)         ! inside tube A
    ZTube2b = CYLINDER ((-Ra,0,0), (-Ra,1,0), Ri)         ! inside tube B

    { Surface Definitions - default values for region 1 }
    z1 = -ZTorus1
    z2 = 0
    z3 = 0
    z4 = ZTorus1

    { heat source and conductivity }
    s = 1
    k = 1

equations
    u: div(k*grad(u)) + s = 0

extrusion
    Surface "Bottom1" z = z1
    Surface "Bottom2" z = z2
    Surface "Top2" z = z3
    Surface "Top1" z = z4

boundaries
    surface "Bottom1" value(u)=0
    surface "Top1" value(u) = 0

    region 1 "Outside Toroid"
      mesh_spacing = Rt/2
      layer 1 s = 1 k = 10
      layer 3 s = 1 k = 10
      start(Ra+Rt, 0)
        value(u) = 0
        arc(center=0,0) angle=180            { the outer boundary }
        natural(u) = 0
        line to (-Ra+Rt, 0)
        value(u) = 0
        arc(center=0,0) angle=-180    { the inner boundary }
        natural(u) = 0
        line to close

    limited region 2  "Inside Toroid"
      z2 = -ZTorus2
      z3 = ZTorus2
      mesh_spacing = Ri/2
      layer 2 s = 100 k = 1
      start(Ra+Ri, 0)
        arc(center=0,0) angle=180            { the outer boundary }
        line to (-Ra+Ri, 0)
        arc(center=0,0) angle=-180    { the inner boundary }
        line to close

    region 3 "Outside TubeA"
      z1 = -ZTube1a
      z4 = ZTube1a
```

```
        mesh_spacing = Rt/2
        layer 1 s = 1 k = 10
        layer 3 s = 1 k = 10
        start (Ra+Rt,0)
          line to (Ra+Rt,-Len)
          line to (Ra-Rt,-Len)
          line to (Ra-Rt,0)
          line to close

      limited region 4 "Inside TubeA"
        z1 = -ZTube1a
        z2 = -ZTube2a
        z3 = ZTube2a
        z4 = ZTube1a
        mesh_spacing = Ri/2
        layer 2 s = 100 k = 1
        start (Ra+Ri,0)
          line to (Ra+Ri,-Len)
          line to (Ra-Ri,-Len)
          line to (Ra-Ri,0)
          line to close

      region 5 "Outside TubeB"
        z1 = -ZTube1b
        z4 = ZTube1b
        mesh_spacing = Rt/2
        layer 1 s = 1 k = 10
        layer 3 s = 1 k = 10
        start (-Ra-Rt,0)
          line to (-Ra-Rt,-Len)
          line to (-Ra+Rt,-Len)
          line to (-Ra+Rt,0)
          line to close

      limited region 6 "Inside TubeB"
        z2 = -ZTube2b
        z3 = ZTube2b
        mesh_spacing = Ri/2
        layer 2 s = 100 k = 1
        start (-Ra-Ri,0)
          line to (-Ra-Ri,-Len)
          line to (-Ra+Ri,-Len)
          line to (-Ra+Ri,0)
          line to close

monitors
    grid(x,y,z)
    contour(u) on surface z=0
    contour(u) on surface y=0

plots
    grid(x,y,z)
    contour(u) on surface z=0
    contour(u) on surface y=0

end
```

6.2.26.27 3d_twist

```
{ 3D_TWIST.PDE

  This problem shows the use of the function definition facility of FlexPDE to create a
  twisted shaft in 3D.

  The mesh generation facility of FlexPDE extrudes a 2D figure along a straight path in Z,
  so that it is not possible to directly define a screw-thread shape.

  However, by defining a coordinate transformation, we can build a straight rod in 3D and
  interpret the coordinates in a rotating frame.

  Define the twisting coordinates by the transformation
    xt = x*cos(a) - y*sin(a);     x = xt*cos(a) + yt*sin(a)
    yt = x*sin(a) + y*cos(a);     y = yt*cos(a) - xt*sin(a)
    zt = z
      with
    a = 2*pi*z/Length = twist*z    (for a total twist of 2*pi radians over the length )

  In this transformation, x and y are the coordinates FlexPDE believes it is working with,
```

```
  and they are the coordinates that move with the twisting, so that the cross section is
  constant in x,y.  xt and yt are the "lab coordinates" of the twisted figure.

  The chain rule then gives
    dF/d(xt) = (dx/dxt)*(dF/dx)  + (dy/dxt)*(dF/dy) + (dz/dxt)*(dF/dz)
    (with similar rules for yt and zt).
  and dx/dzt = twist*[-xt*sin(a) + yt*cos(a)] = y*twist, etc.

  In FlexPDE notation, this becomes
    dxt(F) = cos(a)*dx(F) - sin(a)*dy(F)
    dyt(F) = sin(a)*dx(F) + cos(a)*dy(F)
    dzt(F) = twist*[y*dx(F) - x*dy(F)] + dz(F)

  These relations are defined in the definitions section, and used in the equations
  section, perhaps nested as in the heat equation shown here.

}

title '3D Twisted Rod'

coordinates
    cartesian3

select
    ngrid=25    { use enough mesh cells to resolve the twist }

variables
    Tp

definitions
    long = 20
    wide = 1
    z1 = -long/2
    z2 = long/2

    { transformations }
    twist = 2*pi/long    { radians per unit length }
    c = cos(twist*z)
    s = sin(twist*z)
    xt = c*x-s*y
    yt = s*x+c*y

    { functional definition of derivatives }
    dxt(f) = c*dx(f) - s*dy(f)
    dyt(f) = s*dx(f) + c*dy(f)
    dzt(f) = twist*(y*dx(f) - x*dy(f)) + dz(f)

    { Thermal source }
    Q = 10*exp(-(xt+wide)^2-(yt+wide)^2-z^2)

initial values
    Tp = 0.

equations
    { the heat equation using transformed derivative operators }
    Tp:  dxt(dxt(Tp)) + dyt(dyt(Tp)) + dzt(dzt(Tp)) + Q = 0

extrusion z = z1,z2

boundaries
    surface 1 value(Tp)=0           { fix bottom surface temp }
    surface 2 value(Tp)=0           { fix top surface temp }

    Region 1
         start(-wide,-wide)         { default to insulating sides }
         line to (wide,-wide)
           to (wide,wide)
           to (-wide,wide)
           to close
monitors
    grid(xt,yt,z)                   { the twisted shape }

plots
    grid(xt,yt,z)                   { the twisted shape again }

    { In the following, recall that x and y are the coordinates which
        follow the twist.  It is not possible at present to construct a
        cut in the "lab" coordinates. }
    grid(x,z) on y=0
```

```
    contour(Tp) on y=0  as "ZX Temp"
    contour(Tp) on z=0  as "XY Temp"

end
```

6.2.26.28 3d_void

```
{ 3D_VOID.PDE

  This example shows the use of empty layers in 3D applications.

  The VOID 186 statement appears inside a REGION 183 section, in the position of a
  layer parameter definition.

  The syntax is:
        LAYER number VOID

  This statement causes the stated layer to be excluded from the problem domain
  in the current REGION. (Remember that a REGION refers to a partition of the
  2D projection plane.)

  Boundary conditions on the surface of the void are specified by the standard
  boundary condition facilities.

  In this problem, we have a heat equation with an off-center void in an irregular
  figure.  The Y faces held at zero, the Z-faces are insulated, and the sides
  of the void are held at 1.

}

title '3D VOID LAYER TEST'

coordinates
    cartesian3

select
    errlim = 0.005

variables
    u

definitions
    k = 0.1
    h=0
    x0=0.2  y0=-0.3
    x1=1  y1 = 0.3

equations
    U: div(K*grad(u)) + h = 0

extrusion z=0, 0.3, 0.7, 1


boundaries
    region 1
      start(-1,-1)
      value(u)=0            { Force U=0 on perimeter }
      line to (1,-1)
      arc(center=-1,0) to (1,1)
      line to (-1,1)
      arc(center= -3,0)  to close

    limited region 2        { void exists only on layer 2 }
      layer 2 VOID
      start(x0,y0)
      layer 2  value(u)=1
      line to (x1,y0) to (x1,y1) to (x0,y1) to close

monitors
    elevation(u) from (-0.8,0,0.5) to (1.25,0,0.5)
    elevation(u) from (-0.8,0,0.8) to (1.25,0,0.8)
    contour(u) on z=0
    contour(u) on z=0.5
    contour(u) on z=1
    contour(u) on y=0

plots
    elevation(u) from (-0.8,0,0.5) to (1.25,0,0.5)
```

```
    elevation(u) from (-0.8,0,0.8) to (1.25,0,0.8)
    contour(u) on z=0       painted
    contour(u) on z=0.5     painted
    contour(u) on z=0.499   painted
    contour(u) on z=1       painted
    contour(u) on y=0       painted

end
```

6.2.26.29 regional_surfaces

```
{ REGIONAL_SURFACES.PDE

  This problem demonstrates the use of regional definition of   3D extrusion surfaces.

  There are three "REGIONS" defined, the cubical body of the domain, and two circular
  patches. The circular patches each exist only on a single surface, and in no volumes.
  The patch regions are used to define alternate extrusion surface shapes, and
  insert two parabolic depressions in the top and bottom faces of the cube.

  Click "Domain Review" to watch the gridding process.

}

title 'Regional surface definition'

coordinates
    cartesian3

variables
    Tp

definitions
    long = 1          { domain size }
    wide = 1
    z1 = -1           { bottom surface default shape }
    z2 = 1            { top surface default shape }
    xc = wide/3       {some locating coordinates }
    yc = wide/3
    rc = wide/2
    h = 0.8

    K  = 1            { heat equation parameters }
    Q = exp(-(x^2+y^2+z^2))

initial values
    Tp = 0.

equations
    Tp:  div(k*grad(Tp)) + Q = 0

extrusion z = z1,z2

boundaries
    surface 1 value(Tp)=0
    surface 2 value(Tp)=0

    { define full domain boundary in base plane }
    Region 1
       start(-wide,-wide)
         value(Tp) = 0
         line to (wide,-wide)
           to (wide,wide)
           to (-wide,wide)
           to close

    Limited region 2
      { redefine bottom surface shape in region 2 }
      { note that this shape must meet the default shape at the edge of the region }
      z1 = -1+h*(1-((x+xc)^2+(y+yc)^2)/rc^2)  { a parabolic dent }
      surface 1    { region exists only on surface 1 }
      start(-xc,-yc-rc) arc(center=-xc,-yc) angle=360

    Limited region 3
      { redefine top surface shape in region 3 }
      { note that this shape must meet the default shape at the edge of the region }
      z2 = 1-h*(1-((x-xc)^2+(y-yc)^2)/rc^2)
```

```
        surface 2   { region exists only on surface 2 }
        start(xc,yc-rc) arc(center=xc,yc) angle=360

plots
    grid(x,y,z)
    contour(Tp) on x=y

end
```

6.2.26.30 tabular_surfaces

```
{ TABULAR_SURFACES.PDE

  This problem demonstrates the use of tabular input and regional definition
  for 3D extrusion surfaces.

  The bottom surface of a brick is read from a table.
  Note:  Tables by default use bilinear interpolation.  Mesh cell boundaries do NOT
    automatically follow table boundaries, and sharp slope breaks in table data
    can result in ragged surfaces.  You should always make surface tables dense
    enough to avoid sharp breaks, or put domain boundaries or features along
    breaks in the table slope.  You should also specify mesh density controls
    sufficiently dense to resolve table features.

  The top surface is defined by different functions in two regions.
  Note: the regional surface definitions must coincide at the region boundaries
    where they meet.  Surfaces must be continuous and contain no jumps.

}

title 'tabular surface definition'

coordinates
    cartesian3

variables
    Tp

definitions
    long = 1
    wide = 1
    K  = 1
    Q = 10*exp(-x^2-y^2-z^2)

    { read the table file for surface 1 definition: }
    z1 = table('surf.tbl')
    { use regional parameters for surface 2 definition: }
    z2

initial values
    Tp = 0.

equations
    Tp:  div(k*grad(Tp)) + Q = 0


extrusion z = z1,z2     { define two surfaces from previously declared parameters }

boundaries
    surface 1 value(Tp)=0
    surface 2 value(Tp)=0

    Region 1
       z2 = 1            { default surface 2 over total domain }
       start(-wide,-wide)
         value(Tp) = 0
         line to (wide,-wide)
           to (wide,wide)
           to (-wide,wide)
           to close

    Region 2
       z2 = 1 + x/2     { override surface 2 definition in region 2 }
       start(-wide,-wide)
         line to (0,-wide)
           to (0,wide)
           to (-wide,wide)
           to close
```

```
monitors
    grid(x,z) on y=0

plots
    grid(x,z) on y=0
    contour(Tp) on y=0  as "ZX Temp"
    contour(Tp) on x=0  as "YZ Temp"

end
```

6.2.26.31 two_spheres

6.2.26.31.1 twoz_direct

```
{ TWOZ_DIRECT.PDE

  This problem constructs two non-coplanar spheres inside a box by constructing
  a single dividing surface  to delimit both spheres.

  The domain consists of three layers.
    layer 1 is the space below the spheres
    layer 2 contains the sphere bodies, and is of zero thickness outside the spheres
    layer 3 is the space above the spheres
  The sphere interiors are Void, and are thus excluded from analysis. You could
  just as well fill them with material if you wanted to model the insides.

  The bounding surfaces of layer 2 are specified as a slope perpendicular to the
  centerline of the spheres and over-ridden by regional expressions within
  the (X,Y) extent of each sphere.

  Click "Controls->Domain Review" 7 to watch the mesh construction process.

  See TWOZ_PLANAR.PDE 438, TWOZ_EXPORT.PDE 435 and TWOZ_IMPORT.PDE 436 for other methods of
  treating spheres with centers on differing Z coordinates.

  }

  title 'Two Spheres in 3D - direct surface matching'

  coordinates
     cartesian3

  Variables
     u

  definitions
     K = 1                 { dielectric constant of box filler (vacuum?) }
     box = 1  { bounding box size }

     { read sphere specs from file, to guarantee that they are the same as those in surfgen }
     #include "sphere_spec.inc"

     { sphere shape functions }
     sphere1_shape = SPHERE ((x1,y1,0),R1)
     sphere2_shape = SPHERE ((x2,y2,0),R2)

     { construct an extrusion surface running through both sphere diameters
           by building an embankment between the spheres }
     Rc = sqrt((x2-x1)^2+(y2-y1)^2)-R1-R2
     Rx = Rc*(x2-x1)/Rc/4
     Ry = Rc*(y2-y1)/Rc/4
     xm = (x1+x2)/2
     ym = (y1+y2)/2
     xa = xm - Rx
     ya = ym - Ry
     xb = xm + Rx
     yb = ym + Ry
     xc = xm + Ry
     yc = ym - Rx
     slope = PLANE((xa,ya,z1), (xb,yb,z2), (xc,yc,0))
     zbottom = min(z2,max(z1,slope))
     ztop = zbottom


  equations
    U: div(K*grad(u))  = 0
```

```
 extrusion
   surface "box_bottom"  z=-box
   surface "sphere_bottoms"  z = zbottom
   surface "sphere_tops"  z = ztop
   surface "box_top" z=box

boundaries
    surface "box_bottom" natural(u) = 0  {insulating boundaries top and bottom }
    surface "box_top" natural(u) = 0

    Region 1  { The bounding box }
 start(-box,-box) line to (box,-box) to (box,box) to (-box,box) to close

   limited region 2  { sphere 1 }
       mesh_spacing = R1/5    { force a dense mesh on the sphere }
       zbottom = Z1-sphere1_shape  { shape of surface 2 in sphere 1}
       ztop = Z1+sphere1_shape    { shape of surface 3 in sphere 1}
       layer 2 void
       surface 2 value(u)=V1    { specify sphere1 voltage on top and bottom }
       surface 3 value(u)=V1
       start  (x1+R1,y1)
   arc(center=x1,y1) angle=360

   limited region 3  { sphere 2 }
       mesh_spacing = R2/5    { force a dense mesh on the sphere }
       zbottom = Z2-sphere2_shape  { shape of surface 2 in sphere 2}
       ztop = Z2+sphere2_shape    { shape of surface 3 in sphere 2}
       layer 2 void
       surface 2 value(u)=V2    { specify sphere2 voltage on top and bottom }
       surface 3 value(u)=V2
       start  (x2+R2,y2)
   arc(center=x2,y2) angle=360

plots
    grid(x,y,z)
    grid(x,z) on y=y1  paintregions as "Y-cut through lower sphere"
    contour(u) on y=y1  as "Solution on Y-cut through lower sphere"
    grid(x,z) on y=y2  paintregions as "Y-cut through upper sphere"
    contour(u) on y=y2  as "Solution on Y-cut through upper sphere"
    grid(x*sqrt(2),z) on x-y=0  paintregions  as "Diagonal cut through both spheres"
    contour(u) on x-y=0  as "Solution on Diagonal cut through both spheres"

end
```

6.2.26.31.2 twoz_export

```
{ TWOZ_EXPORT.PDE

  This script uses plate-bending equations to generate a surface that
  passes through the waist of two spheres of differing Z-coordinates.
  The surface is exported with TRANSFER 169 and read into 3D problem
  TWOZ_IMPORT.PDE 436 as the layer-dividing surface.
  (See "Samples | Applications | Stress | Fixed_Plate.pde" 370 for notes on
  plate-bending equations.)
```

```
}
title 'Generating extrusion surfaces'

variables
    U,V

definitions
    box = 1  { bounding box size }

    { read sphere specs from file, to guarantee
      the same values as later including script }
    #include "sphere_spec.inc"

    ! penalty factor to force boundary compliance
    big = 1e6
    ztable = U

equations
    U:  del2(U) = V
    V:  del2(V) = 0

boundaries
   Region 1  { The bounding box }
     start(-box,-box)
     line to (box,-box) to (box,box) to (-box,box) to close

   Region 2    { sphere 1 }
     ztable = Z1            { force a clean table value inside sphere }
     start  (x1+1.01*R1,y1)
     mesh_spacing = R1/5    { force a dense mesh on the sphere }
     load(U) = 0  load(V) = big*(U-Z1)
     arc(center=x1,y1) angle=360

   Region 3     { sphere 2 }
     ztable = Z2
     start  (x2+1.01*R2,y2)
     mesh_spacing = R2/5    { force a dense mesh on the sphere }
     load(U) = 0  load(V)= big*(U-Z2)
     arc(center=x2,y2) angle=360

plots
    elevation(U) from(-box,-box) to (box,box)
    elevation(ztable) from(-box,-box) to (box,box)
    contour(U)
    surface(U)
    contour(ztable) zoom(x1-1.3*R1, y1-1.3*R1, 2.6*R1,2.6*R1)
    contour(ztable) zoom(x2-1.3*R2, y2-1.3*R2, 2.6*R2,2.6*R2)

    transfer(ztable) file = "two_sphere.xfr"

end
```

6.2.26.31.3 twoz_import

```
{ TWOZ_IMPORT.PDE

  This problem constructs two non-coplanar spheres inside a box using an extrusion
  surface generated by TWOZ_EXPORT.PDE 435, which must be run before this problem.

  The domain consists of three layers.
    layer 1 is the space below the spheres
    layer 2 contains the sphere bodies, and is of zero thickness outside the spheres
    layer 3 is the space above the spheres
  The sphere interiors are Void, and are thus excluded from analysis. You could just as
  well fill them with material if you wanted to model the insides.
  The bounding surfaces of layer 2 are specified as a default surface read from a
  TRANSFER 169, over-ridden by regional expressions within  the (X,Y) extent of each sphere.

  Click "Controls->Domain Review" 7 to watch the mesh construction process.

  See  TWOZ_DIRECT.PDE 434 and TWOZ_PLANAR.PDE 438 for other methods of treating spheres
  with centers on differing Z coordinates.
```

```
}

title 'Two Spheres in 3D'

coordinates
    cartesian3

variables
    u

definitions
    { dielectric constant of box filler (vacuum?) }
    K = 1
    box = 1 { bounding box size }

    { read sphere specs from file, to guarantee
      that they are the same as those in surfgen }
    #include "sphere_spec.inc"

    { sphere shape functions }
    sphere1_shape = SPHERE ((x1,y1,0),R1)
    sphere2_shape = SPHERE ((x2,y2,0),R2)

    { read dividing surface generated by surfgen script }

    TRANSFER("two_sphere.xfr",zbottom)
    ztop = zbottom

equations
    U: div(K*grad(u))  = 0

extrusion
    surface "box_bottom"  z=-box
    surface "sphere_bottoms"  z = zbottom
    surface "sphere_tops"  z = ztop
    surface "box_top" z=box

boundaries
    {insulating boundaries top and bottom }
    surface "box_bottom" natural(u) = 0
    surface "box_top" natural(u) = 0

    Region 1  { The bounding box }
        start(-box,-box) line to (box,-box) to (box,box) to (-box,box) to close

    limited region 2  { sphere 1 }
        mesh_spacing = R1/5          { force a dense mesh on the sphere }
        zbottom = Z1-sphere1_shape   { shape of surface 2 in sphere 1}
        ztop = Z1+sphere1_shape      { shape of surface 3 in sphere 1}
        layer 2 void
        surface 2 value(u)=V1        { specify sphere1 voltage on top and bottom }
        surface 3 value(u)=V1
        start  (x1+R1,y1)
            arc(center=x1,y1) angle=360

    limited region 3  { sphere 2 }
        mesh_spacing = R2/5          { force a dense mesh on the sphere }
        zbottom = Z2-sphere2_shape   { shape of surface 2 in sphere 2}
        ztop = Z2+sphere2_shape      { shape of surface 3 in sphere 2}
        layer 2 void
        surface 2 value(u)=V2        { specify sphere2 voltage on top and bottom }
        surface 3 value(u)=V2
        start  (x2+R2,y2)
            arc(center=x2,y2) angle=360

plots
    grid(x,y,z)
    grid(x,z) on y=y1  paintregions as "Y-cut through lower sphere"
    contour(u) on y=y1  as "Solution on Y-cut through lower sphere"
    grid(x,z) on y=y2  paintregions as "Y-cut through upper sphere"
    contour(u) on y=y2  as "Solution on Y-cut through upper sphere"
    grid(x*sqrt(2),z) on x-y=0  paintregions  as "Diagonal cut through both spheres"
    contour(u) on x-y=0  as "Solution on Diagonal cut through both spheres"

end
```

6.2.26.31.4 twoz_planar

```
{ TWOZ_PLANAR.PDE

  This problem constructs two spheres inside a box by constructing multiple planar
  extrusion layers.
  It presents an alternate method for comparison to that of TWOZ_EXPORT.PDE 435 and TWOZ_IMPORT.
PDE 436 .

  The domain consists of five layers.
    layer 1 is the space below the lower sphere
    layer 2 contains the lower sphere body, and is of zero thickness outside the sphere
    layer 3 is the space between the spheres
    layer 4 contains the upper sphere body, and is of zero thickness outside the sphere
    layer 5 is the space above the upper sphere
  The sphere interiors are Void, and are thus excluded from analysis. You could just as
  well fill them with material if you wanted to model the insides.

  The bounding surfaces of layers 2 and 4 are specified as planes at the level of the sphere
  center, over-ridden by regional expressions within  the (X,Y) extent of each sphere.

  Click "Controls->Domain Review" 7  to watch the mesh construction process.

}

title 'Two Spheres in 3D - planar formulation'

coordinates
    cartesian3

variables
    u

definitions
    K = 1       { dielectric constant of box filler (vacuum?) }
    box = 1     { bounding box size }

    { read sphere specs from file, to guarantee that they are the same as those in surfgen }
    #include "sphere_spec.inc"

    { sphere shape functions }
    sphere1_shape = SPHERE ((x1,y1,0),R1)
    sphere2_shape = SPHERE ((x2,y2,0),R2)

    zbottom1 = z1
    ztop1 = z1
    zbottom2 = z2
    ztop2 = z2

equations
    U: div(K*grad(u))  = 0

extrusion
    surface "box_bottom"        z = -box
    surface "sphere1_bottom"    z = zbottom1
    surface "sphere1_top"       z = ztop1
    surface "sphere2_bottom"    z = zbottom2
    surface "sphere2_top"       z = ztop2
    surface "box_top"           z = box

boundaries
    surface "box_bottom" natural(u) = 0  {insulating boundaries top and bottom }
    surface "box_top" natural(u) = 0

    Region 1  { The bounding box }
        start(-box,-box) line to (box,-box) to (box,box) to (-box,box) to close

    limited region 2  { sphere 1 }
        mesh_spacing = R1/5         { force a dense mesh on the sphere }
        zbottom1 = Z1-sphere1_shape { shape of surface 2 in sphere 1}
        ztop1 = Z1+sphere1_shape    { shape of surface 3 in sphere 1}
        layer 2 void
        surface 2 value(u)=V1       { specify sphere1 voltage on top and bottom }
        surface 3 value(u)=V1
        start  (x1+R1,y1)
            arc(center=x1,y1) angle=360

    limited region 3  { sphere 2 }
```

```
        mesh_spacing = R2/5         { force a dense mesh on the sphere }
        zbottom2 = Z2-sphere2_shape { shape of surface 2 in sphere 2}
        ztop2 = Z2+sphere2_shape    { shape of surface 3 in sphere 2}
        layer 4 void
        surface 4 value(u)=V2       { specify sphere2 voltage on top and bottom }
        surface 5 value(u)=V2
        start  (x2+R2,y2)
            arc(center=x2,y2) angle=360

plots
    grid(x,y,z)
    grid(x,z) on y=y1  paintregions as "Y-cut through lower sphere"
    contour(u) on y=y1  as "Solution on Y-cut through lower sphere"
    grid(x,z) on y=y2  paintregions as "Y-cut through upper sphere"
    contour(u) on y=y2  as "Solution on Y-cut through upper sphere"
    grid(x*sqrt(2),z) on x-y=0  paintregions  as "Diagonal cut through both spheres"
    contour(u) on x-y=0  as "Solution on Diagonal cut through both spheres"

end
```

6.2.26.31.5 two_spheres

```
{ TWO_SPHERES.PDE

  This problem constructs two spheres inside a box.  The centers of the spheres lie
  on a single z-plane, which simplifies the domain construction.

  The domain consists of three layers.
    layer 1 is the space below the spheres
    layer 2 contains the sphere bodies, and is of zero thickness outside the spheres
    layer 3 is the space above the spheres
  The sphere interiors are void, and are thus excluded from analysis. They could just as
  well be filled with material if one wanted to model the insides.
  The bounding surfaces of layer 2 are specified as a default, over-ridden by regional
  expressions within the (X,Y) extent of each sphere.

  See TWOZ_PLANAR.PDE 438, TWOZ_DIRECT.PDE 434, TWOZ_EXPORT.PDE 435 and TWOZ_IMPORT.PDE 436
  for methods of treating spheres with centers on differing Z coordinates.

}

title 'Two Spheres in 3D'

coordinates
    cartesian3

variables
    u

definitions
    K = 1           { dielectric constant of box filler (vacuum?) }
    box = 1         { bounding box size }

    R1 = 0.25       { sphere 1 radius, center and voltage}
    x1 = -0.5
    y1 = -0.5
    V1 = -1

    R2 = 0.4        { sphere 2 radius, center and voltage}
    x2 = 0.2
    y2 = 0.2
    V2 = 2

    { sphere shape functions }
    sphere1_shape = SPHERE ((x1,y1,0),R1)
    sphere2_shape = SPHERE ((x2,y2,0),R2)

    { default position of layer 2 surfaces }
    zbottom = 0
    ztop = 0

equations
    U: div(K*grad(u))  = 0

extrusion
    surface "box_bottom"  z=-box
    surface "sphere_bottoms"  z = zbottom
    surface "sphere_tops"  z = ztop
```

```
    surface "box_top" z=box

boundaries
     surface "box_bottom" natural(u) = 0    {insulating boundaries top and bottom }
     surface "box_top" natural(u) = 0

     Region 1    { The bounding box }
    start(-box,-box) line to (box,-box) to (box,box) to (-box,box) to close

    limited region 2    { sphere 1 }
        mesh_spacing = R1/5          { force a dense mesh on the sphere }
        zbottom = -sphere1_shape     { shape of surface 2 in sphere 1}
        ztop = sphere1_shape         { shape of surface 3 in sphere 1}
        layer 2 void
        surface 2 value(u)=V1        { specify sphere1 voltage on top and bottom }
        surface 3 value(u)=V1
        start  (x1+R1,y1)
            arc(center=x1,y1) angle=360

    limited region 3    { sphere 2 }
        mesh_spacing = R2/5          { force a dense mesh on the sphere }
        zbottom = -sphere2_shape     { shape of surface 2 in sphere 2}
        ztop = sphere2_shape         { shape of surface 3 in sphere 2}
        layer 2 void
        surface 2 value(u)=V2        { specify sphere2 voltage on top and bottom }
        surface 3 value(u)=V2
        start  (x2+R2,y2)
            arc(center=x2,y2) angle=360

plots
    grid(x,y,z)
    grid(x,z) on y=y1   as "Grid on Y-cut at sphere 1 center"
    contour(u) on y=y1  as "Solution on Y-cut at sphere 1 center"
    grid(x,z) on y=y2   as "Grid on Y-cut at sphere 2 center"
    contour(u) on y=y2  as "Solution on Y-cut at sphere 2 center"
    { sqrt(2) is needed to plot true distance along diagonal: }
    grid(x*sqrt(2),z) on x-y=0  as "Grid on x=y diagonal"
    contour(u) on x-y=0  as "Solution on x=y diagonal"

end
```

6.2.27 accuracy

6.2.27.1 forever

```
{  FOREVER.PDE

   This problem displays the behaviour of FlexPDE in time dependent problems.
   We posit a field with paraboloidal shape and with amplitude sinusoidal
   in time.  We then derive the source function necessary to achieve this
   solution, and follow the integration for ten cycles, comparing the solution
   to the known analytic solution.

}

title 'A forever test'

variables
    Temp (threshold=0.1)

definitions
    K = 1
    eps = 0
    shape = (1-x^2-y^2)
    Texact = shape*sin(t)
    source = shape*cos(t) - div(K*grad(shape))*sin(t)

initial values
    Temp = Texact

equations
    Temp : div(K*grad(Temp)) + source = dt(Temp)

boundaries
    Region 1
        start(-1,-1)
        value(Temp)=Texact
```

```
        line to (1,-1) to (1,1) to (-1,1) to close

time 0 to 20*pi by 0.01

monitors
    for cycle=5
        contour(Temp)          { show the Temperature during solution }

plots                          { write these plots to the .PGX file }
    for t = pi/2 by pi to endtime
        contour(Temp)
        surface(Temp)
        contour(Temp-Texact) as "Error"
        vector(-dx(Temp),-dy(Temp)) as "Heat Flow"

histories
    history(Temp) at (0,0) (0.5,0.5) integrate
    history(Temp-Texact) at (0,0) (0.5,0.5)

end
```

6.2.27.2 gaus1d

```
{ GAUS1D.PDE

  This test solves a 1D heat equation with a Gaussian solution and compares
  actual deviations from the exact solution with the error estimates made by
  FlexPDE.
  The problem runs a set of ERRLIM 148 levels and plots the history of the comparison.

}
title '1D Accuracy Test - Gaussian'

select
    stages = 5
    ngrid=10
    errlim = staged(1e-2, 1e-3, 1e-4, 1e-5, 1e-6)
    autostage=off

coordinates
    cartesian1

Variables
    u

definitions
    k = 1
    w=0.25
    u0 = exp(-x^2/w^2)
    s = -dxx(u0)

    left=point(-1)
    right=point(1)

equations
    U: div(K*grad(u)) +s = 0

boundaries
    Region 1
        start left  point value(u)=u0
        line to right  point value(u)=u0

monitors
    elevation(u) from left to right

plots
    elevation(u,u0) from left to right report(errlim)
    elevation(u-u0) from left to right as "absolute error"  report(errlim)
    elevation(-div(grad(u)),s) from left to right report(errlim)

histories
    history(sqrt(integral((u-u0)^2))/sqrt(integral(u0^2)), errlim) log

end
```

6.2.27.3 gaus2d

```
{  GAUS2D.PDE

  This test solves a 2D heat equation with a Gaussian solution and compares
  actual deviations from the exact solution with the error estimates made by
  FlexPDE.
  The problem runs a set of ERRLIM 148 levels and plots the history of the comparison.
}

title '2D Accuracy Test - Gaussian'

variables
    u

select
    stages = 4
    errlim = staged(1e-2, 1e-3, 1e-4, 1e-5)

definitions
    k = 1
    h = 0.1
    w=0.2    ! gaussian width
    u0 = exp(-(x^2+y^2)/w^2)
    source = -(dxx(u0)+dyy(u0))
    uxx_exact = dxx(u0)

equations
    U: div(K*grad(u)) + source = 0

boundaries
    Region 1
        start(-1,-1)  natural(u)=0 line to (1,-1)
        value(u)=u0 line to (1,1)
        natural(u)=0  line to (-1,1)
        value(u) = u0 line to close

monitors
    grid(x,y)
    contour(u)

plots
    grid(x,y)
    contour(u)
    elevation(u,u0) from(-1,0) to (1,0)
    elevation(u-u0) from(-1,0) to (1,0)
    elevation(dxx(u),uxx_exact) from(-1,0) to (1,0)
    elevation(dxx(u)+dyy(u),-source) from(-1,0) to (1,0)
    contour(dxx(u)) contour(dxy(u)) contour(dyy(u))

histories
    history(sqrt(integral((u-u0)^2))/sqrt(integral(u0^2)), errlim) LOG

end
```

6.2.27.4 gaus3d

```
{ GAUS3D.PDE

  This test solves a 3D heat equation with a known Gaussian solution and compares
  actual deviations from the exact solution with the error estimates made by
  FlexPDE.
  The problem runs a set of ERRLIM 148 levels and plots the history of the comparison.

  The equation is solved in two forms, letting FlexPDE compute the correct source,
  and imposing analytic derivatives for the source.
}

title '3D Accuracy Test - Gaussian'

coordinates
    cartesian3

select
    ngrid = 5
    stages = 3
    errlim = staged(1e-2, 1e-3, 1e-4)
```

```
variables
    u,v

definitions
    long = 1
    wide = 1
    z1 = -1
    z2 = 1
    w = 0.25  ! gaussian width
    uexact = exp(-(x^2+y^2+z^2)/w^2)
    sfpde = -(dxx(uexact)+dyy(uexact)+dzz(uexact))  ! let FlexPDE do the differentials
    sexact =  -(4/w^4*(x^2+y^2+z^2) - 6/w^2)*uexact ! manual differentiation

initial values
    u = 0.

equations
    U:    div(grad(u)) + sfpde = 0
    V:    div(grad(v)) + sexact = 0

extrusion z = z1,z2

boundaries
    surface 1 value(u)=uexact   value(v)=uexact       { fix bottom surface temp }
    surface 2 value(u)=uexact   value(v)=uexact       { fix top surface temp }

    Region 1                    { define full domain boundary in base plane }
       start(-wide,-wide)
         value(u) = uexact   value(v)=uexact        { fix all side temps }
         line to (wide,-wide)   { walk outer boundary in base plane }
           to (wide,wide)
           to (-wide,wide)
           to close

monitors
    grid(x,z) on y=0
    contour(uexact) on y=0
    contour(u) on y=0
    contour(v) on y=0
    contour(u-uexact) on y=0
    contour(v-uexact) on y=0

plots
    grid(x,z) on y=0
    contour(uexact) on y=0
    contour(u) on y=0
    contour(v) on y=0
    contour(u-uexact) on y=0
    contour(v-uexact) on y=0
    elevation(uexact,u,v) from(-wide,0,0) to (wide,0,0)
    elevation(sfpde,sexact) from(-wide,0,0) to (wide,0,0)

summary
   report(errlim)
   report(sqrt(integral((u-uexact)^2)/sqrt(integral(uexact^2))))
   report(sqrt(integral((v-uexact)^2)/sqrt(integral(uexact^2))))

histories
   history(sqrt(integral((u-uexact)^2)/sqrt(integral(uexact^2))), errlim) log

end
```

6.2.27.5 sine1d

```
{ SINE1D.PDE

  This problem compares the solution accuracy for four different levels of ERRLIM 148.

}

title '1D Accuracy test - Sine'

select
    ngrid=10
    stages = 4
    errlim = staged(1e-2, 1e-3, 1e-4, 1e-5)
```

```
coordinates
    cartesian1

variables
    u

definitions
    k = 1
    h = 0.1
    w=0.1
    rs = abs(x)/w
    u0 = sin(rs)/max(rs,1e-18)
    s = -dxx(u0)

equations
    U: div(K*grad(u)) +s = 0

boundaries
    Region 1
        start(-1)  point value(u)=u0
        line to (1)  point value(u)=u0

monitors
    elevation(u) from (-1) to (1)

plots
    elevation(u,u0) from (-1) to (1)
    elevation(u-u0) from (-1) to (1)
    elevation(-div(grad(u)),s) from (-1) to (1)

histories
    history(sqrt(integral((u-u0)^2)/sqrt(integral(u0^2))),errlim) LOG

end
```

6.2.27.6 sine2d

```
{ SINE2D.PDE

  This problem compares the solution accuracy for four different levels of ERRLIM[148].

}

title '2D Accuracy Test - Sine'

select
    stages = 4
    errlim = staged(1e-2, 1e-3, 1e-4, 1e-5)

variables
    u

definitions
    k = 1
    h = 0.1
    w=0.1
    rs = r/w
    u0 = sin(rs)/rs
    s = -dxx(u0)-dyy(u0)

equations
    U: div(K*grad(u)) +s = 0

boundaries
    Region 1
        start(-1,-1)  value(u)=u0
        line to (1,-1) to (1,1) to (-1,1) to close

monitors
    grid(x,y)
    contour(u)

plots
    grid(x,y)
    contour(u)
    elevation(u,u0) from(-1,0) to (1,0)
    elevation(u-u0) from(-1,0) to (1,0)
    elevation(dxx(u),dxx(u0)) from(-1,0) to (1,0)
```

```
    elevation(dxx(u)+dyy(u),-s) from(-1,0) to (1,0)
    contour(dxx(u)) contour(dxy(u)) contour(dyy(u))

histories
    history(sqrt(integral((u-u0)^2)/sqrt(integral(u0^2))), errlim) LOG

end
```

6.2.27.7 sine3d

```
{ SINE3D.PDE

  This problem compares the solution accuracy for three different levels of ERRLIM 148.

}

title '3D Accuracy Test - Sine'

coordinates
    cartesian3

select
    ngrid = 5
    stages = 3
    errlim = staged(1e-2, 5e-3, 1e-3)

variables
    u

definitions
    long = 1
    wide = 1
    z1 = -1
    z2 = 1
    w=0.1
    rs = r/w
    uex = sin(rs)/rs
    s = -(dxx(uex)+dyy(uex)+dzz(uex))

equations
    U:    div(grad(u)) + s = 0

extrusion z = z1,z2

boundaries
    surface 1 value(u)=uex      { fix bottom surface temp }
    surface 2 value(u)=uex      { fix top surface temp }

    Region 1                { define full domain boundary in base plane }
       start(-wide,-wide)
         value(u) = uex         { fix all side temps }
         line to (wide,-wide)   { walk outer boundary in base plane }
           to (wide,wide)
           to (-wide,wide)
           to close

monitors
    grid(x,z) on y=0
    contour(uex) on y=0
    contour(u) on y=0
    contour(u-uex) on y=0

plots
    grid(x,z) on y=0
    contour(uex) on y=0
    contour(u) on y=0
    contour(u-uex) on y=0
    summary
        report(errlim)
        report(sqrt(integral((u-uex)^2)/sqrt(integral(uex^2))))

histories
    history(sqrt(integral((u-uex)^2)/sqrt(integral(uex^2))), errlim) LOG

end
```

6.2.28 arrays+matrices

6.2.28.1 arrays

```
{ ARRAYS.PDE

  This example demonstrates a few uses of data ARRAYS 159 .

}

title 'ARRAY test'

Variables
     u

definitions
     a = 1

     ! literal data specification
     v = array(0,1,2,3,4,5,6,7,8,9,10)
     ! literal data specification with incrementation
     w = array(0 by 0.1 to 10)
     ! functional definition
     alpha =array for x(0 by 0.1 to 10)  : sin(x)+1.1
     ! construction of a new array by arithmetic operations
     beta = sin(w)+1.1  { this results in the same data as alpha }
     gamma = sin(v)+0.1 { this array is sparsely defined }

     rad = 0.1
     s = 0

equations
     u:  div(a*grad(u)) + s  = 0           { a heat equation }

boundaries
     region 1
        start(0,0)
            value(u)=0
        line to (2,0) to (2,2) to (0,2) to close

plots
    elevation(alpha)
    elevation(alpha,beta) vs w
    elevation(gamma) vs v
summary
    report(sizeof(w))

end
```

6.2.28.2 array_boundary

```
{ ARRAY_BOUNDARY.PDE

  This problem demonstrates the use of data ARRAYS 159 in boundary definition.
  Coordinate arrays are constructed by functional array definition
  and joined in a spline fit to form the system boundary.
}

title 'ARRAY_BOUNDARY test'

variables
     u

definitions
     a = 1
     rad = 1
     ! construct x and y coordinates on a semicircle
     xb =array for ang(-pi/2 by pi/10 to pi/2)  : rad*cos(ang)
     yb =array for ang(-pi/2 by pi/10 to pi/2)  : rad*sin(ang)
     ! multiplying an array by a constant
     xba = 10*xb
     yba = 10*yb
     ! adding a constant to an array
     xbb = xba+11

     s = 1
```

```
equations
     u:  div(a*grad(u)) + s  = 0;           { a heatflow equation }

boundaries
    region 1        { a half-circle built of line segments }
        start(0,-10*rad)
        value(u)=0
          line list (xba, yba)
        natural(u)=0
          line to close
    region 2        { a half-circle built of spline segments }
        start(11,-10*rad)
        value(u)=0
          spline list (xbb, yba)
        natural(u)=0
          line to close

plots
    grid(x,y)
    contour(u) painted
    surface(u)

end
```

6.2.28.3 matrices

```
{ MATRICES.PDE

  This example demonstrates a few uses of data MATRICES 161
}

title 'MATRIX test'

definitions

    { -- literal matrix definition -- }
    m1 = matrix((1,2,3),(4,5,6),(7,8,9))

    { -- functional matrix definition -- }
    { a 79x79 diagonal matrix of amplitude 10: }
    m2 = matrix for x(0.1 by 0.1 to 5*pi/2)
                for y(0.1 by 0.1 to 5*pi/2)
                :   if(x=y) then 10 else 0
    { a 79x79 matrix of sin products: }
    m3 = matrix for x(0.1 by 0.1 to 5*pi/2)
                for y(0.1 by 0.1 to 5*pi/2)
                :    sin(x)*sin(y) +1

    { -- literal array definition -- }
    { a 101-element array of constants: }
    v = array [79] (0.1 by 0.1 to 5*pi/2)

    ! multiply V by matrix M3
    p = m3**v

    ! multiply V by matrix M3, scale by 1e5 and take the sine of each entry
    q = sin((m3**v)/100000)

    rad = 0.1
    s = 0

    ! solve m3*B = P
    b = p // m3

{ no variables }
{ no equations }

boundaries
    region 1
        start(0,0)
          line to (2,0) to (2,2) to (0,2) to close

{ no monitors }

plots
```

```
    elevation(q) vs v as "array vs array"
    elevation(q)  as "array vs index"
    contour(m3) vs v vs v  as "matrix vs two arrays"
    contour(m3) vs v as "matrix vs array and index"

    contour(m2) as "matrix vs indexes"
    surface(m3*m2) as "element product"
    surface(m3+m2) as "element sum"
    surface(m3-m2) as "element difference"
    surface(m3**m2) as "matrix product"
    elevation(b,v) as "matrix inverse times array"
    elevation(m3**b,p)  as "matrix times array and array"

    summary ("selected values")
    report m3[1,1]
    report m3[3,4]
    report v[1]
    report q[1]

end
```

6.2.28.4 matrix_boundary

```
{ MATRIX_BOUNDARY.PDE

  This example demonstrates the use of a data MATRIX 161 in boundary definition.
  Coordinates are constructed by functional matrix definition,
  rotated by multiplication by a rotation matrix
  and joined in a spline fit to form the system boundary.
}

title 'MATRIX_BOUNDARY test'

Variables
    u

definitions
    a = 1
    rad = 1
    ! build a 2 x 21 matrix of x and y coordinates
    mb =matrix for i(1,2)
               for ang(-pi/2 by pi/20 to pi/2)
               : if(i=1) then rad*cos(ang) else rad*sin(ang)
    ! build a 2 x 2 rotation matrix
    rota=45
    rot = matrix[2,2] ((cos(rota degrees), -sin(rota degrees)),
                         (sin(rota degrees), cos(rota degrees)))
    ! rotate the coordinate list
    mbr = rot**mb

    s = 1

equations
    u:  div(a*grad(u)) + s  = 0;           { the heatflow equation }

boundaries
    region 1
      ! start curve at first point of rotated coordinates
      start(mbr[1,1], mbr[2,1])
          value(u)=0
          ! spline fit the 21-point table
          spline list (mbr)
          natural(u)=0
          line to close

plots
    contour(u) painted
    surface(u)

end
```

6.2.29 complex_variables

6.2.29.1 complex+time

```
{ COMPLEX+TIME.PDE

  This example shows the use of complex 89 variables in time-dependent systems.

  The equation that is solved is not intended to represent any real application.

}

title 'Complex transient equations'

Variables
    U(0.01) = complex (Ur,Ui)          { creates variables Ur and Ui  }

definitions
    u0 = 1-x^2-y^2
    s = complex(4,x)

equations
    { create two scalar equations, one for Ur and one for Ui }
    U:  del2(U) +s = dt(U)

boundaries
    Region 1
        start(-1,-1)
        natural(Ur)=u0-Ur
          line to (1,-1) to (1,1) to (-1,1) to close

time 0 to 1

plots
  for cycle=10
       contour(Ur,Ui)
       contour(Real(u),Imag(U))
       contour(U)
       vector(U)
       elevation(u,s) from(-1,0) to (1,0)
       history(u,s) at (0,0)

end
```

6.2.29.2 complex_emw21

```
{ COMPLEX_EMW21.PDE

  This problem is an image of "Backstrom_Books|Waves|Electrodynamics|emw21.pde"
  rewritten in terms of complex 89 variables.

}
TITLE                               { emw21.pde }
   'Plane Wave in a Conductor'
SELECT
   errlim= 1e-3                     { Limit of relative error }
debug(formulas)

VARIABLES
   Ez = complex(Ezr,Ezi)            { Real and imaginary parts }

DEFINITIONS                         { SI units throughout }
   Lx= 1.0             Ly= 0.2      { Domain size }
   eps0= 8.85e-12      eps          { Permittivity }
   mu0= 4*pi*1e-7      mu           { Permeability }
   sigma                            { Electric conductivity }
   omega= 5e9                       { Angular frequency }
   Ez_in= 1.0                       { Input field Ez }
   Ep= magnitude(Ez)                { Modulus of Ez }
   phase= carg(Ez)/pi*180           { Angle }

EQUATIONS
   Ez:  del2( Ez)+ mu*omega*complex(eps*omega, -sigma)*Ez= 0

BOUNDARIES
region 'conductor'      eps= eps0       mu= mu0      sigma= 1e-1
```

```
   start 'outer' (0,0)
   natural(Ez)= complex(0,0)    line to (Lx,0)
   value(Ez)= complex(0,0)      line to (Lx,Ly)     { Conducting }
   natural(Ez)= complex(0,0)    line to (0,Ly)
   value(Ez)= complex(Ez_in,0)  line to close       { Input field }

PLOTS
   elevation( Ez, Ep) from (0,Ly/2) to (Lx,Ly/2)
   elevation( phase) from (0,Ly/2) to (Lx,Ly/2)
   elevation( Ez, Ep) on 'outer'
   contour( Ezr)     contour( Ezi)
END
```

6.2.29.3 complex_variables

```
{ COMPLEX_VARIABLES.PDE

  This example demonstrates the use of complex variables in FlexPDE.

  Declaring a variable COMPLEX 89 causes the definition of two subsidiary variables,
  either named by default or by use choice.  These variables represent the
  real and imaginary parts of the complex variable.

}

title 'Complex variables test'

variables
    U = complex (Ur,Ui)   { creates variables (Ur,Ui) }

definitions
    u0 = 1-x^2-y^2
    s = complex(4,x)

equations
    { create two coupled scalar equations, one for Ur and one for Ui }
    U: del2(U) + conj(U) + s = 0

boundaries
    Region 1
      start(-1,-1)
        value(Ur)=u0        { apply BC to Ur.  Ui defaults to natural(Ui)=0 }
        line to (1,-1) to (1,1) to (-1,1) to close

plots
    contour(Ur,Ui)          { plot both Ur and Ui overlaid }
    contour(Real(U),Imag(U))        { an equivalent representation }
    contour(U)              { another equivalent representation }
    vector(U)               { plot vectors with Ur as X component and Ui as Y component }
    elevation(U,s) from(-1,0) to (1,0)   { plot three traces: Ur, Ui and S }
    vtk(U,s)                { test various export forms }
    cdf(U,s)
    transfer(U,s)

end
```

6.2.29.4 sinusoidal_heat

```
{ SINUSOIDAL_HEAT.PDE

  This example demonstrates the use of COMPLEX 89 variables and ARRAY 159 definitions
  to compute the time-sinusoidal behavior of a rod in a box.

  The heat equation is
    div(k*grad(temp)) = cp*dt(temp)

  If we assume that the sources and solutions are in steady oscillation at a frequency
  omega, then we can write
    temp(x,y,t) = phi(x,y)*exp(i*omega*t) = phi(x,y)*(cos(omega*t) + i*sin(omega*t))

  Substituting this into the heat equation and dividing the exp(i*omega*t) out of the
  result leaves
    div(k*grad(phi)) - i*omega*cp*phi = 0

  The temperature temp(x,y,t) can be reconstructed at any time by expanding the above
  definition.
```

```
  In this example, we construct an array of sample times and the associated arrays
  of sine and cosine factors.  These arrays are then used to display a time history of
  temperature at various points in the domain.
}

TITLE 'Time Sinusoidal Heat flow around an Insulating blob '

VARIABLES
    ! define the complex amplitude function phi and its real and imaginary components
    phi=complex(phir,phii)

DEFINITIONS
    k=1
    ts = array (0 by pi/20 to 2*pi) ! an array of sample times
    fr = cos(ts)                    ! sine and cosine arrays
    fi = sin(ts)
    ! define a function for evaluating a time array of temp(px,py,t) at a point
    temp(px, py) = eval(phir,px,py)*fr + eval(phii,px,py)*fi

EQUATIONS
    phi:    Div(k*grad(phi)) - complex(0,1)*phi= 0

BOUNDARIES

  REGION 1  'box'
      START(-1,-1)
      VALUE(Phi)=complex(0,0)    LINE TO(1,-1)   { Phi=0 in base line }
      NATURAL(Phi)=complex(0,0) LINE TO (1,1)    { normal derivative =0 on right side }
      VALUE(Phi)=complex(1,0)    LINE TO (-1,1)  { Phi = 1 on top }
      NATURAL(Phi)=complex(0,0) LINE TO CLOSE    { normal derivative =0 on left side }

  REGION 2  'rod'   { the embedded circular rod }
      k=0.01
      START 'ring' (1/2,0)
      ARC(CENTER=0,0) ANGLE=360  TO FINISH

PLOTS
    CONTOUR(Phir)       ! plot the real part of phi
        REPORT(k)  REPORT(INTEGRAL(Phir, 'rod'))
    CONTOUR(Phii)       ! plot the imaginary part of phi
        REPORT(k)  REPORT(INTEGRAL(Phii, 'rod'))

    ! reconstruct the temperature distribution at a few selected times
    REPEAT tx=0 by pi/2 to 2*pi
        SURFACE(phir*cos(tx)+phii*sin(tx)) as "Phi at t="+$[4]tx
    ENDREPEAT

    ! plot the time history at a few selected positions
    ELEVATION(temp(0,0), temp(0,0.2), temp(0,0.4), temp(0,0.5)) vs ts  as "Histories"

    VECTOR(-k*grad(Phir))

    ! plot a lineout of phir and phii through the domain
    ELEVATION(Phi) FROM (0,-1) to (0,1)
    ! plot the real component of flux on the surface of the rod
    ELEVATION(Normal(-k*grad(Phir))) ON 'ring'

END
```

6.2.30 constraints

6.2.30.1 3d_constraint

```
{ 3D_CONSTRAINT.PDE

 This problem demonstrates the specification of region-specific CONSTRAINTS 178 in 3D.
 This is a modification of problem  3D_BRICKS.PDE 335.
 We apply a constraint on the integral of temperature in a single region/layer compartment.
 For validation, we define a check function that has nonzero value only in the selected
    compartment and compare its integral to the region-selection form of the integral
    statement.
 Value boundary conditions are applied, so the solution is unique, so the constraint
    acts as a source or sink to maintain the constrained value, we report
    the energy lost to the constraining mechanism.

}
```

```
title '3D constraint'

coordinates
    cartesian3

variables
    Tp

definitions
    long = 1
    wide = 1
    K
    Q = 10*exp(-x^2-y^2-z^2)              { Thermal source }

    flag22=0    { build a test function for region 2, layer 2 }
    check22 = if flag22>0 then Tp else 0

initial values
    Tp = 0.

equations
    Tp:  div(k*grad(Tp)) + Q = 0

constraints
    { constrain temperature integral in region 2 of layer 2 }
    integral(Tp,2,2) = 1

extrusion
    surface "bottom" z = -long
      layer 'bottom'
    surface "middle" z=0
      layer 'top'
    surface 'top' z= long

boundaries
    surface 1 value(Tp)=0   { fix bottom surface temp }
    surface 3 value(Tp)=0   { fix top surface temp }

    Region 1                { define full domain boundary in base plane }
       layer 1 k=1          { bottom right brick }
       layer 2 k=0.1        { top right brick }
       start(-wide,-wide)
         value(Tp) = 0          { fix all side temps }
         line to (wide,-wide)   { walk outer boundary in base plane }
           to (wide,wide)
           to (-wide,wide)
           to close

    Region 2 "Left"         { overlay a second region in left half }
       layer 1 k=0.2        { bottom left brick }
       layer 2 k=0.4  flag22=1  { top left brick }
       start(-wide,-wide)
         line to (0,-wide)      { walk left half boundary in base plane }
           to (0,wide)
           to (-wide,wide)
           to close

monitors
    contour(Tp) on surface z=0  as "XY Temp"
    contour(Tp) on surface x=0  as "YZ Temp"
    contour(Tp) on surface y=0  as "ZX Temp"
    elevation(Tp) from (-wide,0,0) to (wide,0,0)  as "X-Axis Temp"
    elevation(Tp) from (0,-wide,0) to (0,wide,0)  as "Y-Axis Temp"
    elevation(Tp) from (0,0,-long) to (0,0,long)  as "Z-Axis Temp"

plots
    contour(Tp) on z=0  as "XY Temp"
    contour(Tp) on x=0  as "YZ Temp"
    contour(Tp) on y=0  as "ZX Temp"

    summary
      report("Compare integral forms in region 2 of layer 2:")
      report(integral(Tp,2,2))
      report(integral(Tp,"Left","Top"))
      report(integral(check22))
      report("-----")
      report "Constraint acts as an energy sink:"
```

```
        report(integral(Q))        as "Source Integral "
        report(sintegral(normal(-k*grad(Tp)))) as "Surface integral on total outer surface "
        report(integral(Q)-sintegral(normal(-k*grad(Tp)))) as "Energy lost to constraint "

   end
```

6.2.30.2 3d_surf_constraint

```
{ 3D_SURF_CONSTRAINT.PDE

  This problem demonstrates the use of CONSTRAINTS 178 on surface integrals in 3D.
  This is a modification of problem  3D_BRICKS.PDE 335.
  We apply the constraint that the total flux leaving the figure must be 1.0.
  The constraint acts as an auxilliary energy sink, so we report the amount
    of energy lost to the constraint.

  See the problems in the  APPLICATIONS | CONTROL folder for methods
    that control the input power to achieve the same kind of goal.

}

title '3D Surface Constraint'

select
    regrid=off  { use fixed grid to speed up demonstration }

coordinates
    cartesian3

variables
    Tp

definitions
    long = 1
    wide = 1
    K                  { thermal conductivity -- values supplied later }
    Q = 10*exp(-x^2-y^2-z^2)        { Thermal source }

initial values
    Tp = 0.

equations
    Tp:  div(k*grad(Tp)) + Q = 0    { the heat equation }

constraints
    sintegral(normal(k*grad(Tp))) = 1   { force  total surface integral to 1 }

extrusion
    surface "bottom" z = -long
      layer 'bottom'
    surface "middle" z=0
      layer 'top'
    surface 'top' z= long   { divide z into two layers }

boundaries
    surface 1 value(Tp)=0   { fix bottom surface temp }
    surface 3 value(Tp)=0   { fix top surface temp }

    Region 1                { define full domain boundary in base plane }
       layer 1 k=1          { bottom right brick }
       layer 2 k=0.1        { top right brick }
       start(-wide,-wide)
         value(Tp) = 0      { fix all side temps }
         line to (wide,-wide)    { walk outer boundary in base plane }
           to (wide,wide)
           to (-wide,wide)
           to close

    Region 2 "Left"         { overlay a second region in left half }
       layer 1 k=0.2        { bottom left brick }
       layer 2 k=0.4        { top left brick }
       start(-wide,-wide)
         line to (0,-wide)  { walk left half boundary in base plane }
           to (0,wide)
           to (-wide,wide)
           to close
```

```
monitors
    contour(Tp) on surface z=0  as "XY Temp"
    contour(Tp) on surface x=0  as "YZ Temp"
    contour(Tp) on surface y=0  as "ZX Temp"
    elevation(Tp) from (-wide,0,0) to (wide,0,0)  as "X-Axis Temp"
    elevation(Tp) from (0,-wide,0) to (0,wide,0)  as "Y-Axis Temp"
    elevation(Tp) from (0,0,-long) to (0,0,long)  as "Z-Axis Temp"

plots
    contour(Tp) on surface z=0  as "XY Temp"
    contour(Tp) on surface x=0  as "YZ Temp"
    contour(Tp) on surface y=0  as "ZX Temp"

    summary
      report("Constraint Validation:")
      report(sintegral(normal(k*grad(Tp)))) as "Constrained surface integral on total outer
surface"
      report(integral(Q))   as "Total interior source"
      report(integral(Q) - sintegral(normal(k*grad(Tp))))  as "Energy lost to constraint"

end
```

6.2.30.3 boundary_constraint

```
{ BOUNDARY_CONSTRAINT.PDE

  This problem demonstrates the use of boundary-integral CONSTRAINTS 178 .

  A heat equation is solved subject to the constraint that the average temperature
  on the outer boundary must be 1.0.

  Only natural (derivative) boundary conditions are applied, so the solution is
  underdetermined subject to an arbitrary additive constant.

  The constraint provides the additional information necessary to make the
  solution unique.

}

title 'Boundary Constraint Test'

variables
     u

equations
     U: div(grad(u)) +x = 0;

constraints
     { force the average boundary value to 1 }
     bintegral(u,"outer") = bintegral(1,"outer")

boundaries
     Region 1
        start "outer" (-1,-1)
        natural(u) = 0       line to (1,-1) to (1,1) to (-1,1) to close

monitors
    contour(u)  report(bintegral(u,"outer"))
plots
    contour(u)   surface(u)
    elevation(u) on "outer" report(bintegral(u,"outer")/bintegral(1,"outer")) as "Average"
    summary
     report("Constraint Validation:")
     report(bintegral(u,"outer")/bintegral(1,"outer")) as "Average boundary value"

end
```

6.2.30.4 constraint

```
{  CONSTRAINT.PDE

    This problem shows the use of CONSTRAINTS 178 to resolve an ill-posed problem.
    There are no value boundary conditions in any of the three equations, so
    there are infinitely many solutions that satisfy the PDE's.  The constraints
    select from the family of solutions those which have a mean value of 1.
```

```
}

title 'Constraint Test'

variables
    u1 u2 u3

equations
    u1: div(grad(u1)) +x = 0
    u2: div(grad(u2)) +x+y = 0
    u3: div(grad(u3)) +y = 0

constraints
    integral(u1) = integral(1)
    integral(u2) = integral(1)
    integral(u3) = integral(1)

boundaries
    Region 1
      start(-1,-1) line to (1,-1) to (1,1) to (-1,1) to close

monitors
    contour(u1)
    contour(u2)
    contour(u3)
plots
    contour(u1)  report(integral(u1)/integral(1)) as "Average"
    contour(u2)  report(integral(u2)/integral(1)) as "Average"
    contour(u3)  report(integral(u3)/integral(1)) as "Average"
    surface(u1)  report(integral(u1)/integral(1)) as "Average"
    surface(u2)  report(integral(u2)/integral(1)) as "Average"
    surface(u3)  report(integral(u3)/integral(1)) as "Average"

end
```

6.2.31 coordinate_scaling

6.2.31.1 scaled_z

```
{ SCALED_Z.PDE

  This example applies a 10:1 expansion to the z coordinate in a single imbedded layer.
  Compare solution to UNSCALED_Z.PDE 456, which does not scale the z-coordinate.

  See "Help->Technical Notes->Coordinate Scaling 282" for a discussion of the techniques
  used in this example.

}

title 'Scaled Z-coordinate'

coordinates
    cartesian3

variables
    Tp

definitions
    long = 1/2      { thickness of the upper and lower layers }
    wide = 1
    w=0.01          { half-thickness of the imbedded slab }
    zscale=1        { The global Z-Scaling factor, defaulted to 1 for top and bottom layers }
    zscale2=20      { The desired Z-Scaling factor for the center layer }
    ws = w*zscale2  { the scaled half-thickness of the slab }

    K  =0.1         { thermal conductivity -- modified later by layer }
    Q = 0           { Thermal source - modified later by layer }
    T0 = 0

initial values
    Tp = 0.

equations
    { equations are written using the global scaling factor name.
      Layer-specific values will be assigned during evaluation }
    Tp:  dx(k*dx(Tp))/zscale + dy(k*dy(Tp))/zscale + dz(k*zscale*dz(Tp)) + Q/zscale = 0
```

```
extrusion
    surface 'bottom' z = -long-ws
      layer 'under'
    surface 'slab_bottom' z = -ws
      layer 'slab'
    surface 'slab_top' z= ws
      layer 'over'
    surface 'top' z= long+ws

boundaries
    surface 'bottom'  load(Tp)=0.1*(T0-Tp)
    surface 'top'  load(Tp)=0.1*(T0-Tp)

    Region 1
      layer 2
        Q = 100*exp(-x^2-y^2)   { a heat source in the slab layer only }
        zscale = zscale2        { redefine the Z-scaling factor in layer 2 }
        k = 1                   { redefine conductivity in layer 2 }
      start 'sidewall' (-wide,-wide)
        load(Tp) = 0
        layer 2 load(Tp)=0.1*(T0-Tp)/zscale2
        line to (wide,-wide)
             to (wide,wide)
             to (-wide,wide)
             to close

monitors
      contour(Tp) on z=0  as "XY Temp"
      contour(Tp) on x=0  as "YZ Temp"
      contour(Tp) on y=0  as "ZX Temp"

plots
      contour(Tp) on z=0  as "XY Temp"
      contour(Tp) on x=0  as "YZ Temp"
      contour(Tp) on y=0  as "ZX Temp"
      elevation(Tp) from (-wide,0,0) to (wide,0,0)  as "X-Axis Temp"
      elevation(Tp) from (0,-wide,0) to (0,wide,0)  as "Y-Axis Temp"
      elevation(Tp) from (0,0,-long-ws) to (0,0,long+ws)  as "Z-Axis Temp"
      vector(-k*dx(Tp),-k*dz(Tp)) on y=0 as "Flux on Y=0"
      vector(-k*dx(Tp),-k*dy(Tp)) on z=0 as "Flux on Z=0"
      { since "k" refers to energy passing a through unit surface area in the unscaled
        system, its value is unmodified: }
      elevation(k*dx(Tp)) from (-wide,0,0) to (wide,0,0)  as "Center X-Flux"
      { since differentiation with respect to z involves a scaling, the flux must be
        multiplied by the scale factor: }
      elevation(k*dz(Tp)*zscale) from (0,0,-(long+ws)) to (0,0,(long+ws))  as "Center Z-Flux"

      SUMMARY
        { form some integrals for comparison with Unscaled_Z: }
        { the Z flux derivative must be multiplied by the scale factor, but the area
          of integration is in true coordinates }
        { flux leaving the slab, evaluated in the slab: }
        report(sintegral(-k*zscale2*dz(Tp),'slab_top','slab'))
        { flux leaving the slab, evaluated in the upper layer: }
        report(sintegral(-k*1*dz(Tp),'slab_top','over'))
        report("--")
        { The transverse fluxes are in the correct units, but the area integration must be
          corrected by dividing by the scale factor (notice that "zscale" will evaluate to
          "zscale2" in the slab)}
        report(sintegral(-normal(k*grad(Tp))/zscale,'sidewall','slab'))

end
```

6.2.31.2 unscaled_z

```
{ UNSCALED_Z.PDE

  This is a reference problem for SCALED_Z.PDE 455.
  It solves for heatflow in a sandwich.

}

title 'Unscaled Z coordinate'

coordinates
    cartesian3

variables
```

```
    Tp

definitions
    long = 1/2  { thickness of the upper and lower layers }
    wide = 1
    w=0.01      { half-thickness of the imbedded slab }

    K  =0.1     { thermal conductivity -- modified later by layer }
    Q = 0       { Thermal source - modified later by layer }
    T0 = 0

initial values
    Tp = 0.

equations       { the heat equation }
  Tp:  dx(k*dx(Tp)) + dy(k*dy(Tp)) + dz(k*dz(Tp)) + Q = 0

extrusion
  surface 'bottom' z = -long-w
    layer 'under'
  surface 'slab_bottom' z = -w
    layer 'slab'
  surface 'slab_top' z= w
    layer 'over'
  surface 'top' z= long+w

boundaries
  surface 'bottom'  load(Tp)=0.1*(T0-Tp)
  surface 'top'  load(Tp)=0.1*(T0-Tp)

  Region 1
    layer 2
      Q = 100*exp(-x^2-y^2)   { a heat source in the slab layer only }
      k = 1          { redefine conductivity in layer 2 }
    start 'sidewall' (-wide,-wide)
      load(Tp) = 0
      layer 2 load(Tp) = 0.1*(T0-Tp)
      line to (wide,-wide)
           to (wide,wide)
           to (-wide,wide)
           to close

monitors
  contour(Tp) on z=0  as "XY Temp"
  contour(Tp) on x=0  as "YZ Temp"
  contour(Tp) on y=0  as "ZX Temp"

plots
  contour(Tp) on z=0  as "XY Temp"
  contour(Tp) on x=0  as "YZ Temp"
  contour(Tp) on y=0  as "ZX Temp"
  elevation(Tp) from (-wide,0,0) to (wide,0,0)  as "X-Axis Temp"
  elevation(Tp) from (0,-wide,0) to (0,wide,0)  as "Y-Axis Temp"
  elevation(Tp) from (0,0,-long-w) to (0,0,long+w)  as "Z-Axis Temp"
  vector(-k*dx(Tp),-k*dz(Tp)) on y=0 as "Flux on Y=0"
  vector(-k*dx(Tp),-k*dy(Tp)) on z=0 as "Flux on Z=0"
  elevation(k*dx(Tp)) from (-wide,0,0) to (wide,0,0)  as "Center X-Flux"
  elevation(k*dz(Tp)) from (0,0,-(long+w)) to (0,0,(long+w))  as "Center Z-Flux"
  SUMMARY
    { form some integrals for comparison with Scaled_Z: }
    report(sintegral(-k*dz(Tp),'slab_top','slab'))
    report(sintegral(-k*dz(Tp),'slab_top','over'))
    report("--")
    report(sintegral(-normal(k*grad(Tp)),'sidewall','slab'))

end
```

6.2.32 discontinuous_variables

6.2.32.1 3d_contact

```
{ 3D_CONTACT.PDE

  This problem shows the use of a contact resistance boundary between layers in 3D.
  The resistance model is applied to the entire boundary surface.

  See 3D_CONTACT_REGION.PDE 459 for restriction of the resistance model to a single region.
```

```
  (This is a modification of problem 3D_BRICKS.PDE 335).
}

title 'steady-state 3D heat conduction with Contact Resistance'

select
    regrid=off  { use fixed grid }

coordinates
    cartesian3

variables
    Tp

definitions
    long = 1
    wide = 1
    K                           { thermal conductivity -- values supplied later }
    Q = 10*exp(-x^2-y^2-z^2)    { Thermal source }

initial values
    Tp = 0.

equations
    Tp : div(k*grad(Tp)) + Q = 0 { the heat equation }

extrusion z = -long,0,long      { divide Z into two layers }

boundaries
    surface 1 value(Tp)=0       { fix bottom surface temp }
    surface 2 contact(tp)=jump(tp)/10   { THE CONTACT RESISTANCE }
    surface 3 value(Tp)=0       { fix top surface temp }

    Region 1            { define full domain boundary in base plane }
       layer 1 k=1              { bottom right brick }
       layer 2 k=0.1            { top right brick }
       start(-wide,-wide)
         value(Tp) = 0          { fix all side temps }
         line to (wide,-wide)   { walk outer boundary in base plane }
           to (wide,wide)
           to (-wide,wide)
           to close

    Region 2            { overlay a second region in left half }
       layer 1 k=0.2            { bottom left brick }
       layer 2 k=0.4            { top left brick }
       start(-wide,-wide)
         line to (0,-wide)              { walk left half boundary in base plane }
           to (0,wide)
           to (-wide,wide)
           to close

monitors
    contour(Tp) on z=0.01  as "XY Temp - Upper"
    contour(Tp) on z=-0.01  as "XY Temp - Lower"
    contour(Tp) on x=0  as "YZ Temp"
    contour(Tp) on y=0  as "ZX Temp"
    elevation(Tp) from (-wide,0,0) to (wide,0,0)  as "X-Axis Temp"
    elevation(Tp) from (0,-wide,0) to (0,wide,0)  as "Y-Axis Temp"
    elevation(Tp) from (0,0,-long) to (0,0,long)  as "Z-Axis Temp"

plots
    contour(Tp) on z=0.01  as "XY Temp - Upper"
    contour(Tp) on z=-0.01  as "XY Temp - Lower"
    contour(Tp) on x=0  as "YZ Temp"
    contour(Tp) on y=0  as "ZX Temp"
    surface(Tp) on y=0  as "ZX Temp"
    elevation(Tp) from (-wide,0,0) to (wide,0,0)  as "X-Axis Temp"
    elevation(Tp) from (0,-wide,0) to (0,wide,0)  as "Y-Axis Temp"
    elevation(Tp) from (0,0,-long) to (0,0,long)  as "Z-Axis Temp"

end
```

6.2.32.2 3d_contact_region

```
{ 3D_CONTACT_REGION.PDE

  This problem shows the use of a contact resistance boundary between layers.
  The resistance model is applied only to one region of the boundary surface.
  (This is a modification of problem 3D_CONTACT.PDE 457 ).

}

title 'steady-state 3D heat conduction with Contact Resistance'

select
    regrid=off  { use fixed grid }

coordinates
    cartesian3

variables
    Tp

definitions
    long = 1
    wide = 1
    K                           { thermal conductivity -- values supplied later }
    Q = 10*exp(-x^2-y^2-z^2)    { Thermal source }

initial values
    Tp = 0.

equations
    Tp : div(k*grad(Tp)) + Q = 0 { the heat equation }

extrusion z = -long,0,long      { divide Z into two layers }

boundaries
    surface 1 value(Tp)=0       { fix bottom surface temp }
    surface 3 value(Tp)=0       { fix top surface temp }

    Region 1            { define full domain boundary in base plane }
       layer 1 k=1              { bottom right brick }
       layer 2 k=0.1            { top right brick }
       start(-wide,-wide)
         value(Tp) = 0          { fix all side temps }
         line to (wide,-wide)   { walk outer boundary in base plane }
           to (wide,wide)
           to (-wide,wide)
           to close

    Region 2            { overlay a second region in left half }
       { CONTACT RESISTANCE IN REGION 2 ONLY:  }
       surface 2 contact(tp)=jump(tp)/10
       layer 1 k=0.2            { bottom left brick }
       layer 2 k=0.4            { top left brick }
       start(-wide,-wide)
         line to (0,-wide)              { walk left half boundary in base plane }
           to (0,wide)
           to (-wide,wide)
           to close

monitors
    contour(Tp) on z=0.01  as "XY Temp - Upper"
    contour(Tp) on z=-0.01  as "XY Temp - Lower"
    contour(Tp) on x=0  as "YZ Temp"
    contour(Tp) on y=0  as "ZX Temp"
    elevation(Tp) from (-wide/2,0,-long) to (wide/2,0,long)  as "Left Side Temp"

plots
    contour(Tp) on z=0.01  as "XY Temp - Upper"
    contour(Tp) on z=-0.01  as "XY Temp - Lower"
    contour(Tp) on x=0  as "YZ Temp"
    contour(Tp) on y=0  as "ZX Temp"
    elevation(Tp) from (-wide/2,0,-long) to (-wide/2,0,long)  as "Left Side Temp"
    surface(Tp) on y=0  as "ZX Temp" Viewpoint(-3.5,8.2,31)

end
```

6.2.32.3 contact_resistance_heating

```
{  CONTACT_RESISTANCE_HEATING.PDE

   Contact resistance is modeled using the keywords JUMP 192 and CONTACT 215.

   JUMP 192 represents the "jump" in the value of a variable across an interface
     (outer value minus inner value, as seen from each cell),
     and is meaningful only in boundary condition statements.

   CONTACT 215 is a special form of NATURAL 189, which requests that the boundary
   should support a discontinuous value of the variable.

   The model is one of "contact resistance", where the outward current across an
   interface is given by
     R*I = -Jump(V) [=(Vinner-Vouter)],
   and R is the contact resistance.

   Since CONTACT, like NATURAL, represents the outward normal component
   of the argument of the divergence operator,  the contact resistance condition
   for this problem is represented as
     CONTACT(V) = JUMP(Temp)/R

   In this problem, we have two variables, voltage and temperature.
   There is an electrical contact resistance of 2 units at the interface between
   two halves, causing a jump in the voltage across the interface.

   The current through the contact is a source of heat in the temperature equation,
   of value P = R*I^2 = Jump(V)^2/R


}

title "contact resistance heating"

variables
    V
    Temp

definitions
    Kt       { thermal conductivity }
    Heat  =0
    Rc = 2       { Electrical contact resistance }
    rho = 1 { bulk resistivity }
    sigma = 1/rho    { bulk conductivity, I=sigma*grad(V) }

    temp0=0

Initial values
     Temp = temp0

equations
    V:       div(sigma*grad(V))   = 0
    Temp:    div(Kt*grad(Temp)) + Heat =0

boundaries
  Region 1
    Kt=5
    start (0,0)
    natural(V)=0     natural(temp)=0 line to (3,0)
    value(V)=1       value(temp)=0   line to (3,3)
    natural(V)=0     natural(temp)=0 line to (0,3)
    value(V)=0       value(temp)=0   line to close

  Region 2
    Kt=1
    start (0,0)
    line to (1.5,0)
    contact(V) = (1/rc)*JUMP(V) { resistance jump }
    natural(temp) = JUMP(V)^2/Rc     { heat generation }
        line to(1.5,3)
    natural(V)=0     natural(Temp)=0 line to (0,3) to close

monitors
    contour(Temp)

plots
    grid(x,y)
```

```
    contour(V)    painted
    contour(Temp)    painted
    surface(Temp)
    contour(kt*dx(temp))    painted
    contour(kt*dx(temp))    painted
    elevation(V) from(0,1.5) to (3,1.5)
    elevation(temp) from(0,1.5) to (3,1.5)
    elevation(dx(v)) from(0,1.5) to (3,1.5)
    elevation(kt*dx(temp))  from(0,1.5) to (3,1.5)

end
```

6.2.32.4 thermal_contact_resistance

```
{ THERMAL_CONTACT_RESISTANCE.PDE

  This sample demonstrates the application of FlexPDE to heatflow
  problems with contact resistance between materials.

  We define a square region of material with a conductivity of 5.
  Imbedded in this square is a diamond-shaped region of material with a
  uniform heat source of 1, and a conductivity of 1.

  There is a contact resistance of 1/2 unit between the materials.

  Contact resistance is modeled using the keywords JUMP 192 and CONTACT 215.

  JUMP 192 represents the "jump" in the value of a variable across an interface
  (outer value minus inner value, as seen from each cell),
  and is meaningful only in boundary condition statements.

  CONTACT 215 is a special form of NATURAL 189, which requests that the boundary
  should support a discontinuous value of the variable.

  The model is one of "contact resistance", where the flux across an interface
  is given by flux(Temp) = -Jump(Temp)/R,
  and R is the contact resistance.

  Since CONTACT, like NATURAL, represents the outward normal component
  of the argument of the divergence operator,  the contact resistance condition is
  represented as
    CONTACT(Temp) = -JUMP(Temp)/R

 }

title "Thermal Contact Resistance"

variables
    Temp

definitions
    { thermal conductivity - values given in regions: }
    K
    Heat              { Heat source }
    Flux = -K*grad(Temp)
    Rc = 1/2          { contact resistance }

initial values
    Temp = 0

equations
    Temp: div(Flux) = Heat

boundaries
    Region 1            { the outer boundary }
        K=5
        Heat=0
        start "Outer" (0,0)
        value(Temp)=0            { cold boundary }
        line to (3,0)  to (3,3) to (0,3) to close

    Region 2           { an imbedded diamond }
        K=1
        Heat=1         { heat source in the inner diamond }
        start "Inner" (1.5,0.5)

        contact(Temp) = -JUMP(Temp)/Rc  { the contact flux }
```

```
        line to (2.5,1.5)  to (1.5,2.5) to (0.5,1.5) to close

monitors
    contour(Temp)

plots
    grid(x,y)
    contour(Temp) as "Temperature"
    contour(magnitude(grad(temp))) points=5 as "Flux"

    contour(Temp) zoom(2,1,1,1)  as "Temperature Zoom"
    elevation(Temp) from (0,0) to (3,3)

    surface(Temp)
    surface(Temp) zoom(2,1,1,1)
    vector(-dx(Temp),-dy(Temp)) as "Heat Flow"

    elevation(normal(flux)) on "Outer"
    elevation(normal(flux)) on "Inner"

end
```

6.2.32.5 transient_contact_resistance_heating

```
{ TRANSIENT_CONTACT_RESISTANCE_HEATING.PDE

  This is a time-dependent version of the example CONTACT_RESISTANCE_HEATING.PDE
```
460
```
  An electrical current passes through a material with an electrical contact resistance
  on the center plane.  The resistance heating at the contact drives a time-dependent
  heat equation.

}

title "transient contact resistance heating"

variables
    V
    Temp(0.001)

definitions
    Kt              { thermal conductivity }
    Heat  =0
    Rc = 2          { Electrical contact resistance }
    rho = 1         { bulk resistivity }
    sigma = 1/rho   { bulk conductivity, I=sigma*grad(V) }


Initial values
    V = x/3     { a reasonable guess }
    Temp = 0

equations
    V:       div(sigma*grad(V))   = 0
    Temp:    div(Kt*grad(Temp)) + Heat = dt(Temp)

boundaries
     Region 1
        Kt=15
        start (0,0)
        natural(V)=0    natural(temp)=0     line to (3,0)
        value(V)=1      value(temp)=0       line to (3,3)
        natural(V)=0    natural(temp)=0     line to (0,3)
        value(V)=0      value(temp)=0       line to close

     Region 2
        Kt=5
        start (0,0)     line to (1.5,0)
        contact(V) = (1/rc)*JUMP(V)      { resistance jump }
        natural(temp) = JUMP(V)^2/Rc     { heat generation }
              line to(1.5,3)
        natural(V)=0  natural(Temp)=0
              line to (0,3) to close

time 0 to 5 by 1e-6

monitors
```

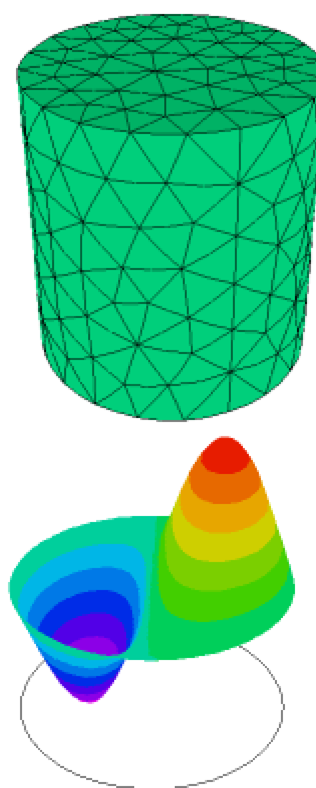
```
    for cycle=5
      contour(Temp)

plots
    for cycle=20
      grid(x,y)
      contour(V)    painted
      contour(Temp)    painted
      surface(Temp)
      contour(kt*dx(temp))    painted
      contour(kt*dx(temp))    painted
      elevation(V) from(0,1.5) to (3,1.5)
      elevation(temp) from(0,1.5) to (3,1.5)
      elevation(dx(v)) from(0,1.5) to (3,1.5)
      elevation(kt*dx(temp))  from(0,1.5) to (3,1.5)
histories
      history(Temp) at (0.5,1.5) (1.0,1.5) (1.5,1.5) (2.0,1.5) (2.5,1.5)

end
```

6.2.33 eigenvalues

6.2.33.1 3d_oildrum

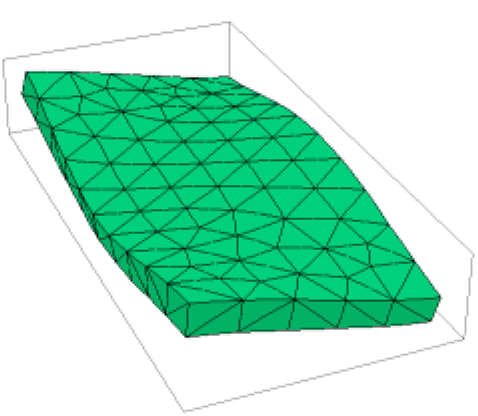
```
{ 3D_OILDRUM.PDE

   ********************************************************************
    This example illustrates the use of FlexPDE in Eigenvalue problems, or
    Modal Analysis.
   ********************************************************************

    In this problem, we determine the four lowest-energy vibrational modes of
    a circular cylinder, or "oil drum", clamped on the periphery.

    What we see as results are the pressure distributions of the air inside the
    drum.

    The three-dimensional initial-boundary value problem associated with the
    scalar wave equation for sound speed "c" can be written as
        c^2*del2(u) - dtt(u) = 0,
    with accompanying initial values and boundary conditions:
        u = f(s,t)              on some part S1 of the boundary
        dn(u) + a*u = g(s,t)    on the remainder S2 of the boundary.

    If we assume that solutions have the form
        u(x,y,z,t) = exp(i*w*t)*v(x,y,z)
    (where "w" is a frequency) then the equation becomes
        del2(v) + lambda*v = 0
    with lambda = (w/c)^2, and with boundary conditions
        v = 0                   on S1
        dn(v) + a*v = 0         on S2.

    The values of lambda for which this system has a non-trivial solution
    are known as the eigenvalues of the system, and the corresponding solutions
    are known as the eigenfunctions or vibration modes of the system.

}

title "Vibrational modes of an Oil Drum"

coordinates cartesian3

select
     modes=4    { Define the number of vibrational modes desired.
                  The appearance of this selector tells FlexPDE
                  to perform an eigenvalue calculation, and to
                  define the name LAMBDA to represent the eigenvalues }
     ngrid=6            { reduced mesh density for demo }
     nodelimit = 3000   { keep problem small for demo }
```

```
Variables
     u

equations       {  the eigenvalue equation }
    U: div(grad(u)) + lambda*u   = 0

{ define the bounding z-surfaces }
extrusion  z = -1,1

boundaries
     { clamp the bottom and top faces }
     surface 1 value(u) = 0
     surface 2 value(u) = 0
     { define circular sidewall }
     Region 1
        start(0,-1)
        value(u) = 0    { clamp the sides }
        arc(center=0,0) angle 360

monitors                {  repeated for all modes }
     contour(u) on x=0
     contour(u) on y=0
     contour(u) on z=1/2

plots                   {  repeated for all modes }
     contour(u) on x=0   surface(u) on x=0
     contour(u) on y=0   surface(u) on y=0
     contour(u) on z=1/2    surface(u) on z=1/2

end
```



6.2.33.2 3d_plate

```
{ 3D_PLATE.PDE

  This problem considers the oscillation modes of a glass plate in space
  ( no mountings to constrain motion ).
            -- Submitted by John Trenholme, Lawrence Livermore Nat'l Lab.

}

TITLE 'Oscillation of a Glass Plate'

COORDINATES
  cartesian3

SELECT
    modes = 5
    ngrid=10
    errlim = 0.01 { 1 percent is good enough }

VARIABLES
    U           { X displacement }
    V           { Y displacement }
    W           { Z displacement }

DEFINITIONS
    cm = 0.01       { converts centimeters to meters }

    long = 20*cm    { length of plate along Y axis }
    wide = 10*cm    { width of plate along X axis }
    thick = 1.2*cm  { thickness of plate along Z axis }


    E = 50e9        { Youngs modulus in Pascals }
    nu = 0.256      { Poisson's ratio  }
    rho = 2500      { density in kg/m^3 = 1000*[g/cc] }


    { constitutive relations - isotropic material }
```



```
    G = E/((1+nu)*(1-2*nu))
    C11 = G*(1-nu)    C12 = G*nu    C13 = G*nu
    C22 = G*(1-nu)    C23 = G*nu    C33 = G*(1-nu)
    C44 = G*(1-2*nu)/2

    { Strains }
    ex = dx(U)    ey = dy(V)    ez = dz(W)
    gxy = dy(U) + dx(V)    gyz = dz(V) + dy(W)    gzx = dx(W) + dz(U)

    { Stresses }
    Sx  =  C11*ex + C12*ey + C13*ez
    Sy  =  C12*ex + C22*ey + C23*ez
    Sz  =  C13*ex + C23*ey + C33*ez
    Txy =  C44*gxy    Tyz =  C44*gyz    Tzx =  C44*gzx

    { find mean Y and Z translation and X rotation }
    Vol = Integral(1)

    { scaling factor for displacement plots }
   Mt =0.1*globalmax(magnitude(x,y,z))/globalmax(magnitude(U,V,W))

INITIAL VALUES
    U = 1.0e-5    V = 1.0e-5    W = 1.0e-5

EQUATIONS
    { we assume sinusoidal oscillation at angular frequency omega =sqrt(lambda) }
    U:  dx(Sx) + dy(Txy) + dz(Tzx) + lambda*rho*U = 0   { X-displacement equation }
    V:  dx(Txy) + dy(Sy) + dz(Tyz) + lambda*rho*V = 0   { Y-displacement equation }
    W:  dx(Tzx) + dy(Tyz) + dz(Sz) + lambda*rho*W = 0   { Z-displacement equation }

CONSTRAINTS
    integral(U)=0               { eliminate translations }
    integral(V)=0
    integral(W)=0
    integral(dx(V)-dy(U)) = 0   { eliminate rotations }
    integral(dy(W) - dz(V)) = 0
    integral(dz(U) - dx(W))  = 0

EXTRUSION
    surface "bottom" z = -thick / 2
    layer "plate"
    surface "top" z = thick / 2

BOUNDARIES
    region 1  { all sides, and top and bottom, are free }
        start( -wide/2, -long/2 )
        line to ( wide/2, -long/2 )
        line to ( wide/2, long/2 )
        line to ( -wide/2, long/2 )
        line to close

MONITORS
    grid(x+Mt*U,y+Mt*V,z+Mt*W)  as "Shape"
        report sqrt(lambda)/(2*pi) as "Frequency in Hz"

PLOTS
    contour( W ) on z = 0 as "Mid-plane Displacement"
        report sqrt(lambda)/(2*pi) as "Frequency in Hz"
    grid(x+Mt*U,y+Mt*V,z+Mt*W)  as "Shape"
        report sqrt(lambda)/(2*pi) as "Frequency in Hz"

    summary
        report lambda
        report sqrt(lambda)/(2*pi) as "Frequency in Hz"

END
```

6.2.33.3 drumhead

```
{ DRUMHEAD.PDE

    *******************************************************************
    This example illustrates the use of FlexPDE in Eigenvalue problems, or
    Modal Analysis.
    *******************************************************************

    The two-dimensional initial-boundary value problem associated with the
    scalar wave equation can be written as
```

```
        c^2*del2(u) - dtt(u) = 0
    with accompanying initial values and boundary conditions
        u = f(s,t)              on S1
        dn(u) + a*u = g(s,t)    on S2.

    If we assume that solutions have the form
        u(x,y,t) = exp(i*w*t)*v(x,y)
    then the equation becomes
        del2(v) + lambda*v = 0
    with lambda = (w/c)^2, and with boundary conditions
        v = 0                   on S1
        dn(v) + a*v = 0         on S2.

    The values of lambda for which this system has a non-trivial solution
    are known as the eigenvalues of the system, and the corresponding solutions
    are known as the eigenfunctions or vibration modes of the system.

    In this problem, we determine the eight lowest-energy vibrational modes of
    a circular drumhead, clamped on the periphery.

    This problem can be solved analytically.  The solutions are of the form
        v = Jn(r*jnm)*exp(i*n*theta),
    where Jn is the Bessel function of order n,
          jnm is the m-th root of Jn.

    The eigenvalues are then just the sequence of jnm^2. In increasing order, or
        5.7832, 14.682, 14.682, 26.375, 26.375, 30.471, 40.706, 40.706
    With errlim=0.001, FlexPDE in the current test gives
        5.7832, 14.682, 14.682, 26.377, 26.377, 30.476, 40.718, 40.720

}

title "Vibrational modes of a drumhead"

select
     { Define the number of vibrational modes desired.
       The appearance of this selector tells FlexPDE
       to perform an eigenvalue calculation, and to
       define the name LAMBDA to represent the eigenvalue: }
     modes=8

Variables
     u

equations           { the eigenvalue equation }
    U: div(grad(u)) + lambda*u   = 0

boundaries
    Region 1
        start(0,-1)
        value(u) = 0
        arc(center=0,0) angle 360

monitors            { repeated for all modes }
    contour(u)

plots               { repeated for all modes }
    contour(u)
    surface(u)

end
```

6.2.33.4 drumhole

```
{ DRUMHOLE.PDE

  ********************************************************************
    This example is a modification of DRUMHEAD.PDE 465, in which
    the vibrating membrane has a circular hole.
    ********************************************************************
}
```

```
title "Vibrational modes of a drumhead with a hole"

select
    modes=8             { Define the number of vibrational modes desired.
                              The appearance of this selector tells FlexPDE
                              to perform an eigenvalue calculation, and to
                              define the name LAMBDA to represent the eigenvalue }

variables
     u

equations              { the eigenvalue equation }
    U: div(grad(u)) + lambda*u   = 0

boundaries
    Region 1
        start(0,-1)
        value(u) = 0
            arc(center=0,0) angle 360
        start(0,-0.4)
        natural(u)=0
            arc(center=0,-0.2) angle=360

monitors               { repeated for all modes }
    contour(u)

plots                  { repeated for all modes }
    contour(u)
    surface(u)

end
```

6.2.33.5 filledguide

```
{ FILLEDGUIDE.PDE

   This problem models an inhomogeneously filled waveguide.
   See discussion in Help section "Electromagnetic Applications | Waveguides" 245.
}

title "Filled Waveguide"

select
  modes = 8                { This is the number of Eigenvalues desired. }
  ngrid=30  regrid=off

variables
  hx,hy

definitions
  cm = 0.01    ! conversion from cm to meters
  b = 1*cm     ! box height
  L = 2*b      ! box width
  epsr
  epsr1=1
  epsr2=1.5
  ejump = 1/epsr2-1/epsr1   ! the boundary jump parameter
  eps0 = 8.85e-12
  mu0 = 4e-7*pi
  c =  1/sqrt(mu0*eps0)     ! light speed
  k0b = 4
  k0 = k0b/b
  k02 = k0^2                ! k0^2=omega^2*mu0*eps0

  curlh = dx(Hy)-dy(Hx)     ! terms used in equations and BC's
  divh = dx(Hx)+dy(Hy)

equations
  Hx:       dx(divh)/epsr - dy(curlh/epsr) + k02*Hx - lambda*Hx/epsr = 0
  Hy:       dx(curlh/epsr) + dy(divh)/epsr + k02*Hy - lambda*Hy/epsr = 0

boundaries
  region 1  epsr=epsr1
    start(0,0)
    natural(Hx) = 0  value(Hy)=0
    line to (L,0)
    value(Hx) = 0  value(Hy)=0  natural(Hy)=0
```

```
    line to (L,b)
    natural(Hx) = 0 value(Hy)=0
    line to (0,b)
    value(Hx) = 0  natural(Hy)=0
    line to close

  region 2  epsr=epsr2
    start(b,b)
    line to (0,b) to (0,0) to (b,0)
    natural(Hx) = normal(-ejump*divh,ejump*curlh)
    natural(Hy) = normal(-ejump*curlh,-ejump*divh)
    line to close

monitors
   contour(Hx) range=(-2,2)
   contour(Hy) range=(-2,2)

plots
   contour(Hx) range=(-2,2) report(k0b) report(sqrt(abs(lambda))/k0)
   surface(Hx) range=(-2,2) report(k0b) report(sqrt(abs(lambda))/k0)
   contour(Hy) range=(-2,2) report(k0b) report(sqrt(abs(lambda))/k0)
   surface(Hy) range=(-2,2) report(k0b) report(sqrt(abs(lambda))/k0)

summary  export
   report(k0b)
   report lambda
   report(sqrt(abs(lambda))/k0)

end
```

6.2.33.6 shiftguide

```
{  SHIFTGUIDE.PDE

   This problem demonstrates the technique of eigenvalue shifting to select
   an eigenvalue band for analysis.  Compare these results to the problem
   Waveguide20, and you will see that the negative modes here correspond to
   the modes below the shift value, while the positive modes here correspond
   to the modes above the shift value.   The result modes  in the shifted calculation
   comprise a complete range of the unshifted modes. (The correspondence is
   1:9, 2:8, 3:10, 4:11, 5:12, 6:13, 7:7, 8:6).

   The solution algorithm used in FlexPDE finds the eigenvalues of lowest
   magnitude, so you will always see a band of positive and negative values
   centered on the shift value.
}

title "TE Waveguide - eigenvalue shifting"


select
  modes = 8
  ngrid=20

variables
  hz

definitions
  L = 2
  h = 0.5         ! half box height
  g = 0.01        ! half-guage of wall
  s = 0.3*L       ! septum depth
  tang = 0.1      ! half-width of tang
  Hx = -dx(Hz)
  Hy = -dy(Hz)
  Ex = Hy
  Ey = -Hx

  shift = 40      ! PERFORM AN EIGENVALUE SHIFT

equations
  Hz:  del2(Hz) + lambda*Hz + shift*Hz = 0

constraints
  integral(Hz) = 0  { since Hz has only natural boundary conditions,
                      we need an additional constraint to make
                      the solution unique }
```

```
boundaries
  region 1
    start(0,0)
    natural(Hz) = 0     line to (L,0) to (L,1) to (0,1)  to (0,h+g)
    natural(Hz) = 0
        line to (s-g,h+g) to (s-g,h+g+tang) to (s+g,h+g+tang)
             to (s+g,h-g-tang) to (s-g,h-g-tang) to (s-g,h-g) to (0,h-g)
    line to close

monitors
    contour(Hz)

plots
    contour(Hz) painted  report (lambda+shift) as "Shifted Lambda"

summary
    report lambda
    report (lambda+shift) as "Shifted Lambda"

end
```

6.2.33.7 vibar

```
{ VIBAR.PDE

  This problem analyzes the standing-wave vibrational modes of an elastic bar.

  The equations of Stress/Strain in a material medium can be given as
        dx(Sx) + dy(Txy) + Fx = 0
        dx(Txy) + dy(Sy) + Fy = 0

  where Sx and Sy are the stresses in the x- and y- directions,
  Txy is the shear stress, and Fx and Fy are the body forces in the
  x- and y- directions.

  In a time-dependent problem, the material acceleration and viscous force
  act as body forces, and are included in a new body force term
        Fx1 = Fx0 - rho*dtt(U) + mu*del2(dt(U))
        Fy1 = Fy0 - rho*dtt(V) + mu*del2(dt(V))

  where rho is the material mass density, mu is the viscosity, and U and V
  are the material displacements in the x and y directions.

  If we assume that the displacement is harmonic in time (all transients
  have died out), then we can assert
        U(t) = U0*exp(-i*omega*t)
        V(t) = V0*exp(-i*omega*t)

  Here U0(x,y) and V0(x,y) are the complex amplitude distributions, and
  omega is the angular velocity of the oscillation.

  Substituting this assumption into the stress equations and dividing out
  the common exponential factors, we get (implying U0 by U and V0 by V)
        dx(Sx) + dy(Txy) + Fx0 + rho*omega^2*U - i*omega*mu*del2(U) = 0
        dx(Txy) + dy(Sy) + Fy0 + rho*omega^2*V - i*omega*mu*del2(V) = 0

  All the terms in this equation are now complex.  Separating into real
  and imaginary parts gives
        U = Ur + i*Ui
        Sx = Srx + i*Six
        Sy = Sry + i*Siy
        etc...

  Expressed in terms of the (assumed real) constitutive relations of the material,
        Srx = [C11*dx(Ur) + C12*dy(Vr)]
        Sry = [C12*dx(Ur) + C22*dy(Vr)]
        Trxy = C33*[dy(Ur) + dx(Vr)]
        etc...

  The final result is a set of four equations in Ur,Vr,Ui and Vi.
    Ur: dx(Srx) + dy(Trxy) + rho*omega^2*Ur + omega*mu*del2(Ui) = 0
    Ui: dx(Six) + dy(Tixy) + rho*omega^2*Ui - omega*mu*del2(Ur) = 0
    Vr: dx(Trxy) + dy(Sry) + rho*omega^2*Vr + omega*mu*del2(Vi) = 0
    Vi: dx(Tixy) + dy(Siy) + rho*omega^2*Vi - omega*mu*del2(Vr) = 0

  In the absence of viscous effects, these equations separate, with no imaginary
  terms appearing in the real equations, and vice versa.
```

```
  We can therefore solve only for the real components Ur and Vr, which we
  will continue to refer to as U and V.

  Solving the eigenvalue system
    U:        dx(Sx)  + dy(Txy) + lambda*rho*U = 0
    V:        dx(Txy)   + dy(Sy) + lambda*rho*V = 0
  we find the resonant frequencies lambda = omega^2 together with the
  corresponding spatial amplitude distributions Uand V.

  In order to quantify the "natural" (or "load") boundary condition mechanism,
  we can write the equations as
    U:     div(P) + lambda*rho*U = 0
    V:     div(Q) + lambda*rho*V = 0
  where P = [Sx,Txy]
  and   Q = [Txy,Sy]

  The natural (or "load") boundary condition for the U-equation defines the
  outward surface-normal component of P, while the natural boundary condition
  for the V-equation defines the surface-normal component of Q. Thus, the
  natural boundary conditions for the U- and V- equations together define
  the surface load vector.

  On a free boundary, both of these vectors are zero, so a free boundary
  is simply specified by
    load(U) = 0
    load(V) = 0.

 }

title "Vibrating Bar - Modal Analysis"

select
    modes=8
    cubic   { Use Cubic Basis }
    errlim = 0.005

variables
    U    { X-displacement }
    V    { Y-displacement }

definitions
    L = 1                 { Bar length }
    hL = L/2
    W = 0.1               { Bar thickness }
    hW = W/2
    nu = 0.3              { Poisson's Ratio }
    E  = 20               { Young's Modulus for Steel x10^11(dynes/cm^2) }
    G  = 0.5*E/(1+nu)
    rho = 7.8             { Density (g/cm^3) }

    { plane strain coefficients }
    E1 = E/((1+nu)*(1-2*nu))
    C11 = E1*(1-nu)
    C12 = E1*nu
    C22 = E1*(1-nu)
    C33 = E1*(1-2*nu)/2

    { Stresses }
    Sx = (C11*dx(U) + C12*dy(V))
    Sy = (C12*dx(U) + C22*dy(V))
    Txy = C33*(dy(U) + dx(V))

    mag=0.05

initial values
    U = 0
    V = 0

equations                 { define the displacement equations }
    U:  dx(Sx)  + dy(Txy) + lambda*rho*U = 0
    V:  dx(Txy)  + dy(Sy) + lambda*rho*V = 0

boundaries
    region 1
      start (0,-hW)

      { free boundary on bottom, no normal stress }
      load(U)=0     load(V)=0       line to (L,-hW)
```

```
        { clamp the right end }
        value(U) = 0  line to (L,0)   point value(V) = 0
        line to (L,hw)

        { free boundary on top, no normal stress }
        load(U)=0     load(V)=0       line to (0,hw)

        load(U) = 0   load(V) = 0     line to close

  monitors
      grid(x+mag*U,y+mag*V)   as "deformation"   { show final deformed grid }
  plots
      grid(x+mag*U,y+mag*V)   as "deformation"   { show final deformed grid }
      contour(U) as "X-Displacement(M)"
      contour(V) as "Y-Displacement(M)"

  end
```

6.2.33.8 waveguide

```
{ WAVEGUIDE.PDE

  This problem solves for the Transverse-Electric modes of a T-septate
  rectangular waveguide.

  Assuming that Z is the propagation direction, we can write
    E(x,y,z) = E(x,y)*exp(i*(omega*t-Kz*z))
    H(x,y,z) = H(x,y)*exp(i*(omega*t-Kz*z))
  where omega is the angular frequency and kz denotes the propagation constant.

  In a Transverse-Electric waveguide, the electric field component in the propagation
  direction is zero, or Ez = 0.

  Substituting these equations into the source-free Maxwell's equations and rearranging,
  we can write
    Ey =  -(omega*mu/kz)*Hx
    Ex = (omega*mu/kz)*Hy
    Hx = -i*dx(Hz)*kz/kt
    Hy = i*dy(Hz)*kz/kt
    with kt = [omega^2*eps*mu - kz^2]

  It can also be shown that in this case Hz satisfies the homogeneous Helmholtz equation
    dxx(Hz) + dyy(Hz) + Kt^2*Hz = 0
  together with the homogeneous Neumann boundary condition on the conducting wall
    dn(Hz) = 0

  In order to avoid clutter in this example script, we will supress the proportionality factors.
  (The leading "i" in the definition of Hx and Hy is merely a phase shift.)
         -----  From J. Jin,  "The Finite Element Method in Electromagnetics", p. 197
}

title "TE Waveguide"

select
  modes = 4     { This is the number of Eigenvalues desired. }

variables
  hz

definitions
  L = 2
  h = 0.5        ! half box height
  g = 0.01       ! half-guage of wall
  s = 0.3*L      ! septum depth
  tang = 0.1     ! half-width of tang
  Hx = -dx(Hz)
  Hy = dy(Hz)
  Ex = Hy
  Ey = -Hx

equations
  Hz:  del2(Hz) + lambda*Hz = 0        { lambda = Kt^2 }

constraints
  integral(Hz) = 0  { since Hz has only natural boundary conditions,
                      we need an additional constraint to make
                      the solution unique }
```

```
boundaries
  region 1
    start(0,0)
    natural(Hz) = 0     ! this condition applies to all subsequent segments
      ! walk the box body
      line to (L,0) to (L,1) to (0,1)  to (0,h+g)
      ! walk the T-septum
               to (s-g,h+g) to (s-g,h+g+tang) to (s+g,h+g+tang)
               to (s+g,h-g-tang) to (s-g,h-g-tang) to (s-g,h-g) to (0,h-g)
      line to close

monitors
  contour(Hz)

plots
  contour(Hz) painted
  vector(Hx,Hy)  as "Transverse H" norm
  vector(Ex,Ey)  as "Transverse E" norm

end
```

6.2.33.9 waveguide20

```
{  WAVEGUIDE20.PDE

   This problem solves for the Transverse-Electric modes of a T-septate
    rectangular waveguide.  It is a copy of WAVEGUIDE.PDE 471 with more modes.
}

title "TE Waveguide"

select
  modes = 20        { This is the number of Eigenvalues desired. }
  ngrid=20          { we need enough density to resolve higer modes }

variables
  hz

definitions
  L = 2
  h = 0.5                 ! half box height
  g = 0.01                ! half-guage of wall
  s = 0.3*L               ! septum depth
  tang = 0.1              ! half-width of tang
  Hx = -dx(Hz)
  Hy = -dy(Hz)
  Ex = Hy
  Ey = -Hx

equations
   Hz:  del2(Hz) + lambda*Hz = 0

constraints
  integral(Hz) = 0  { since Hz has only natural boundary conditions,
            we need to constrain the answer }

boundaries
  region 1
    start(0,0)
    natural(Hz) = 0     line to (L,0) to (L,1) to (0,1)  to (0,h+g)
    natural(Hz) = 0
        line to (s-g,h+g) to (s-g,h+g+tang) to (s+g,h+g+tang)
             to (s+g,h-g-tang) to (s-g,h-g-tang) to (s-g,h-g) to (0,h-g)
    line to close

monitors
  contour(Hz)

plots
  contour(Hz) painted

end
```

6.2.34 import-export

6.2.34.1 3d_mesh_export

```
{ 3D_MESH_EXPORT.PDE

  This example shows the use of the TRANSFER 169 command to export problem data
  and mesh structure in 3D problems.

  The accompanying test 3D_MESH_IMPORT.PDE 473 reads the transfer file produced here.

  (The framework of the problem is a version of 3D_ANTIPERIODIC.PDE 504.)

}

title '3D MESH TRANSFER TEST'

coordinates cartesian3

variables
    u

definitions
    k = 1
    an = pi/4          { this is the angular size of the repeated segment }
    crot = cos(an)     { the sine and cosine needed in the transformation }
    srot = sin(an)
    H = 0
    xc = 1.5
    yc = 0.2
    rc = 0.1

equations
    U: div(K*grad(u)) + H = 0

extrusion z=0,0.4,0.6,1

boundaries
    Region 1

       start(1,0) line to (2,0)

       value(u) = 0  arc(center=0,0) to (2*crot,2*srot)

       antiperiodic(x*crot+y*srot, -x*srot+y*crot)
       line to (crot,srot)

       value(u)=0
       arc(center= 0,0)  to close

    Limited Region 2
        layer 2 H=1
        start(xc-rc,0) line to (xc+rc,0) to (xc+rc,rc) to (xc-rc,rc) to close

    Limited Region 3
        layer 2 H=-1
        start((xc-rc)*crot,(xc-rc)*srot)
        line to ((xc+rc)*crot,(xc+rc)*srot)
             to ((xc+rc)*crot+rc*srot,(xc+rc)*srot-rc*crot)
             to ((xc-rc)*crot+rc*srot,(xc-rc)*srot-rc*crot) to close

  plots
     contour(u) on z=0.5 paint
     grid(x,y,z)

     transfer(u)  file="mesh3u.xfr" ! Export mesh and data
     transfer()   file="mesh3.xfr"  ! Export mesh only

  end
```

6.2.34.2 3d_mesh_import

```
{ 3D_MESH_IMPORT.PDE

    This example shows the use of the TRANSFERMESH 169 command to import a
    3D Mesh.  The mesh file is created by running 3D_MESH_EXPORT.PDE 473.
```

```
    Note that the domain structure must exactly match that of the exporting problem.
    Periodicity condtions must also be the same, except that periodic and antiperiodic
    may be exchanged.

    (The framework of this problem is 3D_ANTIPERIODIC.PDE 504.)

}

title '3D MESH IMPORT TEST'

coordinates cartesian3

variables
    u

definitions
    k = 1
    { angular size of the repeated segment: }
    an = pi/4
    { sine and cosine needed in transformation }
    crot = cos(an)
    srot = sin(an)
    H = 0
    xc = 1.5
    yc = 0.2
    rc = 0.1

    transfermesh("mesh3.xfr")       ! << read the mesh file

equations
    U: div(K*grad(u)) + H = 0

extrusion z=0,0.4,0.6,1

boundaries
    Region 1
       start(1,0) line to (2,0)
       value(u) = 0  arc(center=0,0) to (2*crot,2*srot)

       antiperiodic(x*crot+y*srot, -x*srot+y*crot)
       line to (crot,srot)

       natural(u)=x-2.4*y       ! BC changed from exporting problem
       arc(center= 0,0)  to close

    Limited Region 2
        layer 2 H = 1
        start(xc-rc,0) line to (xc+rc,0) to (xc+rc,rc) to (xc-rc,rc) to close

    Limited Region 3
        layer 2 H = -1
        start((xc-rc)*crot,(xc-rc)*srot)
        line to ((xc+rc)*crot,(xc+rc)*srot)
                to ((xc+rc)*crot+rc*srot,(xc+rc)*srot-rc*crot)
                to ((xc-rc)*crot+rc*srot,(xc-rc)*srot-rc*crot) to close

plots
     contour(u) on z=0.5 paint
     grid(x,y,z)

end
```

6.2.34.3 3d_post_processing

```
{ 3D_POST_PROCESSING.PDE

  This example demonstrates the use of the TRANSFERMESH 169 facility to import
  both data and mesh structure from 3D_MESH_EXPORT.PDE 473 and perform
  post-processing without gridding or solving any equations.

  This is easily accomplished in a step-wise process:
  1) make a copy of the script that generated the exported data
  2) remove the VARIABLES and EQUATIONS sections
  3) remove any boundary conditions stated in the BOUNDARIES section
  4) add the TRANSFERMESH statement in the DEFINITIONS section
  5) add any new plots that you desire
```

```
  Note that the domain structure must exactly match that of the exporting problem.

  3D_MESH_EXPORT.PDE 473 must be run before running this problem.

}

title 'Using TRANSFERMESH for post-processing'

coordinates cartesian3

definitions
    k = 1
    an = pi/4           { this is the angular size of the repeated segment }
    crot = cos(an)      { the sine and cosine needed in the transformation }
    srot = sin(an)
    H = 0
    xc = 1.5
    yc = 0.2
    rc = 0.1
    transfermesh("mesh3u.xfr",U)

extrusion z=0,0.4,0.6,1

boundaries
    Region 1

       start(1,0) line to (2,0)
       arc(center=0,0) to (2*crot,2*srot)
       line to (crot,srot)
       arc(center= 0,0)  to close

    Limited Region 2
        layer 2 H=1
        start(xc-rc,0) line to (xc+rc,0) to (xc+rc,rc) to (xc-rc,rc) to close

    Limited Region 3
        layer 2 H=-1
        start((xc-rc)*crot,(xc-rc)*srot)
        line to ((xc+rc)*crot,(xc+rc)*srot)
             to ((xc+rc)*crot+rc*srot,(xc+rc)*srot-rc*crot)
             to ((xc-rc)*crot+rc*srot,(xc-rc)*srot-rc*crot) to close

plots
    grid(x,y,z)
    grid(x,y) on z=0.5
    contour(u) on z=0.5 zoom(1.3,0,0.4,0.4)
    contour(u) on z=0.5 zoom(1.4,0,0.2,0.2) paint

end
```

6.2.34.4 3d_surf_export

```
{ 3D_SURF_EXPORT.PDE

  This problem shows data export on an extrusion surface in 3D.

  Values are exported on a cut plane in default text format,
  and on a cut plane and an extrusion surface in user-specified columnar format.
  (See "Format 'string' 202" in the Help Index for formatting rules.)

  The output files will be given the default names
  "3d_surf_export.p02", "...p03" and "...p04", corresponding to the second,
  third and fourth plot specifications.

  The problem is a modification of 3D_SPHERE.PDE 421.
}

title '3D Export Test - Sphere'

coordinates
     cartesian3

variables
     u

definitions
```

```
     k = 0.1                    { conductivity }
     heat =6*k                  { internal heat source }
    u0 = exp(-x^2-y^2)

equations
    U: div(K*grad(u)) + heat   = 0

extrusion
    surface z = -sqrt(1-(x^2+y^2))      { the bottom hemisphere }
    surface z = sqrt(1-(x^2+y^2))       { the top hemisphere }

boundaries
    surface 1 value(u) = u0     { fixed value on sphere surfaces }
    surface 2 value(u) = u0
    region 1
        start(1,0) arc(center=0,0) angle=360

plots
    grid(x,y,z)
    contour(u) on x=0          { YZ plane through diameter }
    export
    contour(u) on z=0.5        { XY plane above center }
            export format "#x#b#y#b#z#b#1"
    contour(u) on surface 2            { top surface }
            export format "#x#b#y#b#z#b#1"

end
```

6.2.34.5 blocktable

```
{ BLOCKTABLE.PDE

  This example shows the use of the BLOCK 167 modifier in reading TABLE 165 data.

  The BLOCK 167 modifier allows table data to be interpreted in Histogram profile.
  The default interpretation imposes a 10% rise width on the histogram blocks,
  to avoid dramatic timestep cuts when data are used as driving profiles in
  time-dependent problems.

  The BLOCK(rise) 167 qualifier allows the specification of a rise width as a fraction
  of block width.

}

title '1D BLOCK table'

select
    regrid=off

{ No Variables are necessary }

definitions
    { single value format with default 10% rise width: }
    u = block table("table1.tbl")
    { assignment list format with 50% rise width: }
    block(0.5) tabledef("table1.tbl",v)
    { single value format with un-blocked interpretation: }
    w = table("table1.tbl")

boundaries
    Region 1
      start(0,0)
      line to (10,0) to (10,1) to (0,1) to close

plots
    contour(u)  as "10% rise"
    contour(v)  as  "50% rise"
    contour(w)  as "Unblocked"
    elevation(u) as "10% rise" from(0,0.5) to (10,0.5)
    elevation(v) as "50% rise" from(0,0.5) to (10,0.5)
    elevation(w) as "Unblocked" from(0,0.5) to (10,0.5)
    elevation(u, v, w) from(0,0.5) to (10,0.5)

end
```

6.2.34.6 export

```
{ EXPORT.PDE

  This sample demonstrates the use of several forms of data export selectors.
  All exports use the default file naming conventions, which append modifiers
  to the problem name.

  A heat flow problem is solved on a square for example purposes.

}

title "Demonstrate forms of export"

select
    contourgrid=50

variables
    Temp

definitions
    K = 1
    source = 4
    Texact = 1-x^2-y^2
    flux=magnitude(K*grad(Temp))

Initial values
    Temp = 0

equations
    Temp: div(K*grad(Temp)) + source = 0

boundaries
    Region 1
        start "BDRY" (-1,-1)
        value(Temp)=Texact
        line to (1,-1)
             to (1,1)
             to (-1,1)
             to close

monitors
    contour(Temp)

plots
    { this contour plot exports graphic images in four formats:
        ( notice that the BMP file is 46 times larger than the other formats!)  }
    contour(Temp) PNG EPS EMF BMP
    { export temperature and flux in NetCDF format }
    cdf(temp,flux)
    { export FlexPDE TABLE format }
    table(temp)
    { export temperature and flux in TecPlot format }
    tecplot(temp,flux)
    { export temperature and flux in linearized VTK format }
    vtklin(temp,flux)


end
```

6.2.34.7 export_format

```
{ EXPORT_FORMAT.PDE

  This problem demonstrates a few variations on the use of the
  FORMAT 202 modifier in data export.

}

Title 'Test FORMATTED export'

Variables
  u(1.0)

Equations
  U:  dxx(u) + dyy(u) = -4
```

```
Boundaries
  region 1
    start(0.5,1)
    value(u)=0                    { the cold outer boundary }
    line to (2.5,1) to (2.5,2) to (0.5,2) to close

    start(1,1.2)
    natural(u) = 0
    line to (1,1.8) to (2,1.8)
    line to (1.52,1.52) to (1,1.52) to (1,1.48) to (1.52,1.48)
         to (2,1.2) to close

Monitors
  contour(u)

Plots

  { An ELEVATION plot prints a tag-delimited data list to the file "PTABLE.TXT":}
  elevation(u) from (1.5,1) to (1.5,2) export format "#y#b#1" file="ptable.txt"

  { A CONTOUR plot prints a tab-delimited table of values in the default
        file "export_format.p02": }
  contour(u^2)  export format "#x#b#y#b#1"

  { A VECTOR plot prints a table of vectors delimited by commas and parentheses
       in the file "VECTOR.TXT": }
  vector(-dx(u),-dy(u))  zoom(1.9,1.7,0.2,0.2)  export format "(#x,#y)=(#1,#2)"
       file "vectors.txt"

  { A TABLE output without graphics writes a 10x10 table of FIXED POINT gridding
       statements suitable for inclusion in another PDE descriptor
       (in the default file "export_format_01.tbl"): }
  table(u) format "fixed point (#x,#y) point load(u)=(#1-u)" points=10

  { A TABLE output without graphics writes a 12x10 table of gaussian source
       statements suitable for inclusion in another PDE descriptor
       (in the default file "export_format_02.tbl"): }
  table(u) format "+a*exp(-((x-#x)/c)^2-((y-#y)/c)^2)*(#1-u)" points=(12,10)

End
```

6.2.34.8 export_history

```
{ EXPORT_HISTORY.PDE

  This example illustrates use of the FORMAT 202 modifier in the export of a
  HISTORY 211 plot.

  The repeat (#R 202) construct is used to create a comma-delimited data list.

  The problem is the same as FLOAT_ZONE.PDE 337.

}

title
  "FORMATTED HISTORY EXPORT"

coordinates
  xcylinder('Z','R')

select
  cubic          { Use Cubic Basis }

variables
  temp (threshold=100)

definitions
  k = 0.85       { thermal conductivity}
  cp = 1         { heat capacity }
  long = 18
  H = 0.4        { free convection boundary coupling }
  Ta = 25        { ambient temperature }
  A = 4500       { amplitude }

  source = A*exp(-((z-1*t)/.5)^2)*(200/(t+199))

initial value
```

```
  temp = Ta

equations
  Temp:  div(k*grad(temp)) + source = cp*dt(temp)

boundaries
  region 1
    start(0,0)
    natural(temp) = 0 line to (long,0)
    value(temp) = Ta line to (long,1)
    natural(temp) = -H*(temp - Ta) line to (0,1)
    value(temp) = Ta line to close
  feature
    start(0.01*long,0) line to (0.01*long,1)

time -0.5 to 19 by 0.01

monitors
  for t = -0.5 by 0.5 to (long + 1)
  elevation(temp) from (0,1) to (long,1) range=(0,1800) as "Surface Temp"
  contour(temp)

plots
  for t = -0.5 by 0.5 to (long + 1)
  elevation(temp) from (0,0) to (long,0) range=(0,1800) as "Axis Temp"

histories
  { EXPORT a formatted HISTORY file: }
   history(temp) at (0,0) (1,0) (2,0) (3,0) (4,0) (5,0) (6,0) (7,0) (8,0)
                  (9,0) (10,0) (11,0) (12,0) (13,0) (14,0) (15,0) (16,0)
                  (17,0) (18,0)  export format "#t#r,#i"

end
```

6.2.34.9 mesh_export

```
{ MESH_EXPORT.PDE

  This example uses a modification of the sample problem HEAT_BOUNDARY.PDE 338
  to illustrate the use of the TRANSFER 197 output function.

  Both the temperatures calculated here and the final mesh structure are transferred
  as input to the stress calculation MESH_IMPORT.PDE 480

}

title "Test TRANSFER output"

variables
     Temp

definitions
     K = 1
     source = 4
     Tzero = 0
     flux = -K*grad(Temp)

equations
    Temp: div(K*grad(Temp)) + source = 0

boundaries
     Region 1
        start "OUTER" (0,0)
        natural(Temp)=0     line to(1,0)

        natural(Temp)=0     arc (center=0,0) to (0,1)

        natural(Temp)=0     line to close

        start "INNER" (0.4,0.2)
        natural(Temp)=Tzero-Temp
          arc (center=0.4,0.4)
             to (0.6,0.4)
             to (0.4,0.6)
             to (0.2,0.4)
             to close
```

```
monitors
     contour(Temp)

plots
     grid(x,y)
     contour(Temp)
     surface(Temp)
     vector(-K*dx(Temp),-K*dy(Temp)) as "Heat Flow"
     contour(source)
     elevation(normal(flux)) on "outer" range(-0.08,0.08)
         report(bintegral(normal(flux),"outer")) as "bintegral"
     elevation(normal(flux)) on "inner" range(1.95,2.3)
         report(bintegral(normal(flux),"inner")) as "bintegral"

     { HERE IS THE TRANSFER OUTPUT COMMAND: }
     transfer(Temp,source) file="transferm.dat"

end
```

6.2.34.10 mesh_import

```
{ MESH_IMPORT.PDE

  This problem demonstrates the use of the TRANSFERMESH 197 facility to import
  both data and mesh structure from MESH_EXPORT.PDE 479.

  MESH_EXPORT.PDE 479 must be run before running this problem.

}

title 'Testing the TRANSFERMESH statement'

select
    painted             { paint all contour plots }

variables
    U
    V

definitions
    nu = 0.3            { define Poisson's Ratio }
    E  = 21             { Young's Modulus x 10^-11 }
    G  = E/(1-nu^2)
    C11 = G
    C12 = G*nu
    C22 = G
    C33 = G*(1-nu)/2

    alpha = 1e-3
    b = G*alpha*(1+nu)

   { HERE IS THE TRANSFERMESH INPUT FUNCTION: }
    transfermesh('transferm.dat',Temp)

    Sxx = C11*dx(U) + C12*dy(V) - b*Temp
    Syy = C12*dx(U) + C22*dy(V) - b*temp
    Sxy = C33*(dy(U) + dx(V))

initial values
    U = 0
    V = 0

equations
    U:  dx(Sxx) + dy(Sxy)  = 0
    V:  dy(Syy) + dx(Sxy) = 0

boundaries
    Region 1
        start "OUTER" (0,0)
        natural(U)=0 value(V)=0        { no y-motion on x-axis }
          line to(1,0)
        natural(U)=0 natural(V)=0      { free outer boundary }
          arc (center=0,0) to (0,1)
        value(U)=0 natural(V)=0        { no x-motion on y-axis }
          line to close

        natural(U)=0 natural(V)=0      { free inner boundary }
```

```
        start "INNER" (0.4,0.2)
          arc (center=0.4,0.4)
             to (0.6,0.4)
             to (0.4,0.6)
             to (0.2,0.4)
             to close

monitors
      grid(x+100*U,y+100*V)

plots
     contour(Temp)
     grid(x+100*U,y+100*V)
     vector(U,V) as "Displacement"
     contour(U) as "X-Displacement"
     contour(V) as "Y-Displacement"
     contour(Sxx) as "X-Stress"
     contour(Syy) as "Y-Stress"
     surface(Sxx) as "X-Stress"
     surface(Syy) as "Y-Stress"

end
```

6.2.34.11 post_processing

```
{ POST_PROCESSING.PDE

  This example demonstrates the use of the TRANSFERMESH 169 facility to import
  both data and mesh structure from MESH_EXPORT.PDE 479 and perform
  post-processing without gridding or solving any equations.

  This is easily accomplished in a step-wise process:
  1) make a copy of the script that generated the exported data
  2) remove the VARIABLES and EQUATIONS sections
  3) remove any boundary conditions stated in the BOUNDARIES section
  4) add the TRANSFERMESH statement in the DEFINITIONS section
  5) add any new plots that you desire

  Note that the domain structure must exactly match that of the exporting problem.

  MESH_EXPORT.PDE 479 must be run before running this problem.

}

title "Using TRANSFERMESH for post-processing"

definitions
    K = 1
    transfermesh('transferm.dat',Temp)

boundaries
     Region 1
        start "OUTER" (0,0)
        line to(1,0)
        arc (center=0,0) to (0,1)
        line to close

        start "INNER" (0.4,0.2)
          arc (center=0.4,0.4)
             to (0.6,0.4)
             to (0.4,0.6)
             to (0.2,0.4)
             to close

plots
     grid(x,y)
     contour(Temp)
     contour(Temp) zoom(0.2,0.2,0.1,0.1)
     surface(Temp)
     vector(-K*dx(Temp),-K*dy(Temp)) as "Heat Flow"

end
```

6.2.34.12 splinetable

```
{ SPLINETABLE.PDE

   This example solves the same system as TABLE.PDE 482, using a Spline interpretation of the
   data in the table file 'TABLE.TBL'.
   The file format is the same for TABLE 165 or SPLINE TABLE 167 input.

   The SPLINE TABLE operator can be used to build spline tables of one or two dimensions.
   The resulting interpolation is third order in the coordinates, with continuous values
   and derivatives. First or second derivatives of the interpolated function may be computed.

   Here the table is used as source and diffusivity in a fictitious heat equation, merely to
   show the use of the table variable.

   The SAVE function is used to construct a Finite Element interpolation of the data from the
   spline table, for comparison of derivatives.  Cubic FEM basis is used so that the second
   derivative is meaningful.
}
title 'Spline Table Input Test'

select
  regrid=off

variables
  u

definitions
  alpha = spline table('table.tbl')  ! construct spline fit of table:
  beta = 1/alpha
  femalpha = save(alpha)             ! save a FEM interpolation of table:

equations
  U:  div(alpha*grad(u)) + beta = 0

boundaries
  region 1
    start(0,10)
    value(u) = 0
    line to (0,0) to (10,0) to (10,10) to close
  monitors
  contour(u)

plots
  grid(x,y)
  contour(alpha) as 'table'
  contour(dx(alpha)) as 'dx(table)'
  contour(dy(alpha)) as 'dy(table)'
  vector(grad(alpha)) as 'grad(table)'
  surface(alpha) as 'table'
  contour(dxx(alpha)) as 'dxx(table)'
  contour(dxy(alpha)) as 'dxy(table)'
  contour(dyy(alpha)) as 'dyy(table)'
  contour(dxx(alpha)+dyy(alpha))  as "Table Curvature"
  contour(div(grad(femalpha)))  as "FEM Curvature"
  surface(beta) as "table reciprocal"
  contour(u) as "temperature solution"
  surface(u) as "temperature solution"

end
```

6.2.34.13 table

```
{  TABLE.PDE

   This problem demonstrates the use of tabular data.
   It reads the file "TABLE.TBL", uses the data in a heat equation,
   and displays the table data.

}
title 'Table Input Test'

select
  errlim = 0.0005

variables
```

```
  u

definitions
  alpha = table('table.tbl')
  beta = 1/alpha

equations
  U:  div(alpha*grad(u)) + beta = 0

boundaries
  region 1
    start(0,10)
    value(u) = 0
    line to (0,0) to (10,0) to (10,10) to close

monitors
  contour(u)

plots
  grid(x,y)
  contour(alpha)  as "Conductivity (Table data)"
  surface(alpha) as 'Conductivity (Table data)'
  vector(grad(alpha)) as 'grad(table)'
  surface(beta)  as "Source (Table Reciprocal)"
  contour(u)  as "Temperature solution"
  surface(u)  as "Temperature solution"

end
```

6.2.34.14 tabledef

```
{  TABLEDEF.PDE

    This problem illustrates the use of the TABLEDEF 167 function to define several
    parameters from an imported table named TABLEDEF.TBL

    Note that the TABLEDEF 167 function has the same syntax as the TRANSFER 169 function.
    The difference is that TABLEDEF 167 uses a rectangular grid of data values,
    while TRANSFER 169 uses an unstructured triangular finite element mesh created
    by a prior FlexPDE run.

}
title 'Table Input Test'

select
  errlim = 0.0005

variables
  u

definitions
  tabledef('tabledef.tbl',alpha,beta)

equations
  U:  div(alpha *grad(u)) + beta = 0

boundaries
  region 1
    start(0,10)
    value(u) = 0
    line to (0,0) to (10,0) to (10,10) to close

monitors
  contour(u)

plots
  grid(x,y)
  contour(u)
  surface(u)
  contour(alpha)
  contour(beta)
  vector(grad(alpha))

end
```

6.2.34.15 table_export

```
{  TABLE_EXPORT.PDE

  This example shows the use of FlexPDE as a generator of data tables
  in proper format to be read in by other FlexPDE problems.

  We define a domain which is the domain of the table coordinates, and
  compute and export the table.

  No variables or equations are declared.

  This example exports both a 1D and a 2D table of a Gaussian in the table
   files "GAUS1.TBL" and "GAUS2.TBL".

  The output is in default format, suitable for TABLE 165 input to other FlexPDE runs.
  See "FORMAT 'string'" in the Help Index for formatting controls.

  See TABLE_IMPORT.PDE 484 for an example of reading the TABLE 165 created here.

}

title 'TABLE generation'

select
    regrid=off

definitions
    u = exp(-16*(x^2+y^2))

boundaries
    Region 1
        start(-1,-1)
        line to (1,-1) to (1,1) to (-1,1) to close
plots
    contour(u)
    surface(u)
    ! 2D table
    table(u) points=51 file='gauss2.tbl'
    ! 1D table
    elevation(u) from(-1,0) to (1,0) export file='gauss1.tbl'

end
```

6.2.34.16 table_import

```
{  TABLE_IMPORT.PDE

  This example reads a 1D table created by TABLE_EXPORT.PDE 484 and fits
  the data with a cubic spline.  It then compares derivatives with
  analytic values.

}

title '1D Spline table import'

select
    regrid=off

definitions
     u = spline table("gauss1.tbl")
     gu = exp(-16*x^2)
boundaries
     Region 1
        start(-1,-1)
        line to (1,-1) to (1,1) to (-1,1) to close
plots
     contour(u) as "imported data"
     contour(dx(u)) as "X-derivative of imported data"
     contour(dxx(u)) as "XX-derivative of imported data"
     elevation(u, gu) from(-1,0) to (1,0) as "Imported data and exact function"
     elevation(dx(u), dx(gu)) from(-1,0) to (1,0) as "Imported X-derivative and exact function"
     elevation(dxx(u), dxx(gu)) from(-1,0) to (1,0) as "Imported XX-derivative and exact function"


end
```

6.2.34.17 transfer_export

```
{  TRANSFER_EXPORT.PDE

  This example uses a modification of the sample problem
  HEAT_BOUNDARY.PDE 338 to illustrate the use of the TRANSFER 169 output
  function.  Temperatures calculated here are transferred as
  input to the stress calculation TRANSFER_IMPORT.PDE 485

}


title "TRANSFER export test"

variables
    Temp (threshold=0.1)

definitions
    K = 1
    source = 4
    Tzero = 0
    flux = -K*grad(Temp)

equations
    Temp:  div(K*grad(Temp)) + source = 0

boundaries
    Region 1
        start "OUTER" (0,0)
        natural(Temp)=0        line to(1,0)

        natural(Temp)=0        arc (center=0,0) to (0,1)

        natural(Temp)=0        line to close

        start "INNER" (0.4,0.2)
        natural(Temp)=Tzero-Temp
        arc (center=0.4,0.4)
           to (0.6,0.4)
           to (0.4,0.6)
           to (0.2,0.4)
           to close

monitors
    contour(Temp)

plots
    grid(x,y)
    contour(Temp)
    surface(Temp)
    vector(-K*dx(Temp),-K*dy(Temp)) as "Heat Flow"
    contour(source)
    elevation(normal(flux)) on "outer" range(-0.08,0.08)
       report(bintegral(normal(flux),"outer")) as "bintegral"
    elevation(normal(flux)) on "inner" range(1.95,2.3)
       report(bintegral(normal(flux),"inner")) as "bintegral"

    { HERE IS THE TRANSFER OUTPUT COMMAND: }
    transfer(Temp,K) file="transfer.dat"

end
```

6.2.34.18 transfer_import

```
{ TRANSFER_IMPORT.PDE

  This problem demonstrates the use of the TRANSFER 169 facility to import
  temperatures from TRANSFER_EXPORT.PDE 485 as the source of thermal expansion
  driving a stress calculation.

  TRANSFER_EXPORT.PDE 485 must be run before running this problem.
}

title 'Testing the TRANSFER input function'

select
    painted              { paint all contour plots }
```

```
variables
    U
    V

definitions
    nu = 0.3              { define Poisson's Ratio }
    E  = 21               { Young's Modulus x 10^-11 }
    G  = E/(1-nu^2)
    C11 = G
    C12 = G*nu
    C22 = G
    C33 = G*(1-nu)/2

    alpha = 1e-3
    b = G*alpha*(1+nu)

    { HERE IS THE TRANSFER INPUT FUNCTION: }
    transfer('transfer.dat',Temp,Kxfer)

    Sxx = C11*dx(U) + C12*dy(V) - b*Temp
    Syy = C12*dx(U) + C22*dy(V) - b*temp
    Sxy = C33*(dy(U) + dx(V))

initial values
    U = 0
    V = 0

equations
    U:  dx(Sxx) + dy(Sxy)  = 0
    V:  dy(Syy) + dx(Sxy) = 0

constraints
     integral(u) = 0
     integral(v) = 0
     integral(dx(v)-dy(u)) = 0

boundaries
     Region 1
        start "OUTER" (0,0)

        natural(U)=0 value(V)=0     line to(1,0)
        natural(U)=0 natural(V)=0
            arc (center=0,0) to (0,1) { free outer boundary }
        value(U)=0 natural(V)=0     line to close

        { free inner boundary }
        start "INNER" (0.4,0.2)
        natural(U)=0 natural(V)=0
        arc (center=0.4,0.4)
           to (0.6,0.4)
           to (0.4,0.6)
           to (0.2,0.4)
           to close

monitors
      grid(x+100*U,y+100*V)

plots
     contour(Temp)  report(Kxfer)
     grid(x+100*U,y+100*V)
     vector(U,V) as "Displacement"
     contour(U) as "X-Displacement"
     contour(V) as "Y-Displacement"
     contour(Sxx) as "X-Stress"
     contour(Syy) as "Y-Stress"
     surface(Sxx) as "X-Stress"
     surface(Syy) as "Y-Stress"

end
```

6.2.35 mesh_control

6.2.35.1 3d_curvature

```
{ 3D_CURVATURE.PDE

  This problem demonstrates automatic mesh densification due to curvature and
  proximity to small features.

  The example consists of a three-layer heatflow problem.  The bottom layer contains
  a hidden rise, or "dimple", that rises close to the base of the adjoining layer.

  FlexPDE detects this dimple and automatically refines the computation mesh to
  resolve the curvature of the tip.

  It also detects the proximity of the dimple peak to the adjoining layer and refines the
  mesh in that layer as well.

}

title '3D Layer curvature resolution Test'

coordinates
    cartesian3

select
    paintregions

variables
    Tp

definitions
    long = 1
    wide = 1
    K  = 1
    Q = 0
    narrow = 0.2
    z1 = 0
    z2 = 0.1+0.3*exp(-(x^2+y^2)/narrow^2)
    z3 = 0.5
    z4= 1

initial values
    Tp = 0.

equations
    Tp:  div(k*grad(Tp)) + Q = 0

extrusion z = z1,z2,z3,z4


boundaries
    surface 1 value (Tp)=0
    surface 4 value (Tp)=1

    Region 1
       layer 1 k=10
       layer 3 k=5
       start (-wide,-wide)
         line to (wide,-wide)  to (wide,wide) to (-wide,wide) to close

monitors
      grid (x,z) on y=0
      contour (Tp) on z=0.38  painted

plots
      grid(x,y,z) on layer 1
      grid (x,z) on y=0
      grid(x,y) on surface 2
      contour (Tp) on y=0  as "ZX Temp"
      contour (Tp) on z=0.38  painted

end
```

6.2.35.2 boundary_density

```
{  BOUNDARY_DENSITY.PDE

   This problem demonstrates the use of the MESH_DENSITY 173 parameter to
   control mesh density along a boundary.

   The boundary of the inner region is forced to a grid spacing of 0.02

}

title 'Cell Size Control'

variables
     u

definitions
     k = 1
     u0 = 1-x^2-y^2
     s = 2*3/4+5*2/4
     b = 0.1
     c = 0.02

equations
    U: div(K*grad(u)) +s = 0

boundaries
     Region 1
        start(-1,-1)
        value(u)=u0
        line to (1,-1) to (1,1) to (-1,1) to close
     Region 2
        start(-b,-b)
        mesh_density = 1/c    { command inside the boundary path }
        line to (b,-b) to (b,b) to (-b,b) to close

plots
     grid(x,y)
     contour(u)  on region 2

end
```

6.2.35.3 boundary_spacing

```
{  BOUNDARY_SPACING.PDE

   This problem demonstrates the use of the MESH_SPACING 173 parameter to
   control mesh density along a boundary.

   The boundary of the inner region is forced to a grid spacing of 0.02
}


title 'Cell Size Control'

variables
     u

definitions
     k = 1
     u0 = 1-x^2-y^2
     s = 2*3/4+5*2/4
     b = 0.1
     c = 0.02

equations
     U: div(K*grad(u)) +s = 0

boundaries
     Region 1
        start(-1,-1)
        value(u)=u0
        line to (1,-1) to (1,1) to (-1,1) to close
     Region 2
        start(-b,-b)
        mesh_spacing=c  { command placed inside the boundary path }
        line to (b,-b) to (b,b) to (-b,b) to close
```

```
plots
     grid(x,y)
     contour(u)  on region 2

end
```

6.2.35.4 front

```
{ FRONT.PDE

  This example demonstrates the use of the FRONT 194 statement
  to create a dense mesh at a moving front.

  The FRONT 194 command is used to force mesh refinement wherever the
  concentration variable passes through a value of 0.5.

  The problem is the same as CHEMBURN.PDE 288.

}

title
  'FRONT statement in Chemical Reactor'

select
   painted      { make color-filled contour plots }

variables
  Temp (threshold=1)
  C (threshold=1)

definitions
  Lz = 1
  r1=1
  heat=0
  gamma = 16
  beta = 0.2
  betap = 0.3
  BI = 1
  T0 = 1
  TW = 0.92
  { the very nasty reaction rate: }
  RC = (1-C)*exp(gamma-gamma/Temp)
  xev=0.96    { some plot points }
  yev=0.25

initial value
  Temp=T0
  C=0

equations
  Temp:  div(grad(Temp)) + heat + betap*RC = dt(Temp)
  C:  div(grad(C)) + beta*RC = dt(C)

boundaries
  region 1
    start (0,0)

   { a mirror plane on X-axis }
    natural(Temp) = 0
    natural(C) = 0
    line to (r1,0)

    { "Strip Heater" at fixed temperature }
    { ramp the boundary temp in time, because  discontinuity is costly to diffuse }
    value(Temp)=T0 + 0.2*uramp(t,t-0.05)
    natural(C)=0                { no mass flow on strip heater }
    arc(center=0,0) angle 5

    { convective cooling and no mass flow on outer arc }
    natural(Temp)=BI*(TW-Temp)
    natural(C)=0
    arc(center=0,0) angle 85

    { a mirror plane on Y-axis }
    natural(Temp) = 0
    natural(C) = 0
```

```
    line to (0,0) to close

time 0 to 1

{ FORCE CELLS TO SPAN NO MORE THAN 0.1 ACROSS C=0.5 }
front(C-0.5, 0.1)

plots
  for cycle=10                    { watch the fast events by cycle }
    grid(x,y)
    contour(Temp)
    contour(C) as "Completion"

  for t= 0.2 by 0.05 to 0.3        { show some surfaces during burn }
    surface(Temp)
    surface(C) as "Completion"

histories
  history(Temp,C) at (0,0) (xev/2,yev/2) (xev,yev) (yev/2,xev/2) (yev,xev)

end
```

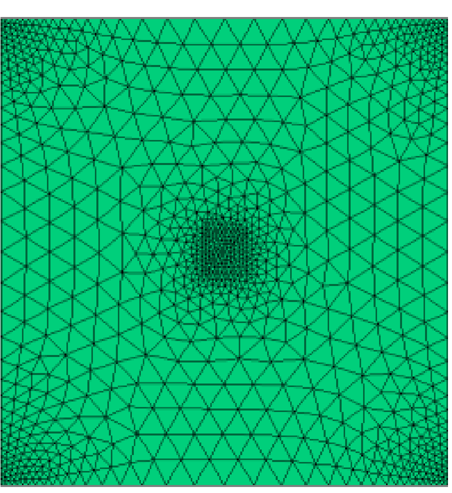
6.2.35.5 mesh_density

```
{ MESH_DENSITY.PDE

   This example demonstrates the use of the MESH_DENSITY 173 parameter to
   control mesh density.

   A global density function is defined as a Gaussian distribution returning
   1 cell-per-unit density at the center, rising to 54.6 cell-per-unit density
   at the corners.

   This global distribution is overridden by a regional definition of 50 cell-per-unit
   density in a central region.

}

title 'Cell Size Control'

variables
    u

definitions
    k = 1
    u0 = 1-x^2-y^2
    s = 2*3/4+5*2/4
    mesh_density = exp(2*(x^2+y^2))
    box = 0.1

equations
    u : div(K*grad(u)) +s = 0

boundaries
    Region 1
        start(-1,-1)
            value(u)=u0
        line to (1,-1) to (1,1) to (-1,1) to close
    Region 2
        mesh_density = 50
        start(-box,-box)
        line to (box,-box) to (box,box) to (-box,box) to close

plots
    grid(x,y)
    contour(u)

end
```

6.2.35.6 mesh_spacing

```
{ MESH_SPACING.PDE

  This example demonstrates the use of the MESH_SPACING 173 parameter to
  control mesh density.
```

```
  A global density function is defined as a Gaussian distribution returning
  1 unit mesh spacing at the center, falling to 0.018 at the corners.

  This global distribution is overridden by a regional definition of
  0.02 mesh spacing in a central region.

}

title 'Cell Size Control'

variables
     u

definitions
     k = 1
     u0 = 1-x^2-y^2
     s = 2*3/4+5*2/4
     mesh_spacing = exp(-2*(x^2+y^2))
     b = 0.1
     c = 0.02

equations
     u : div(K*grad(u)) +s = 0

boundaries
     Region 1
        start(-1,-1)
        value(u)=u0
        line to (1,-1) to (1,1) to (-1,1) to close
     Region 2
        mesh_spacing = c
        start(-b,-b)
        line to (b,-b) to (b,b) to (-b,b) to close

plots
     grid(x,y)
     contour(u)

end
```



6.2.35.7 resolve

```
{ RESOLVE.PDE

  This is a test problem from Timoshenko: Theory of Elasticity, p41

  The RESOLVE 195 statement has been added to force regridder to resolve the
  shear stress.

 }

title "RESOLVE shear stress in bent bar"

select
    elevationgrid=500
    cubic

variables
    U   { X-displacement }
    V    { Y-displacement }

definitions
    L = 1                { Bar length }
    hL = L/2
    W = 0.1              { Bar thickness }
    hW = W/2
    eps = 0.01*L
    I = 2*hW^3/3         { Moment of inertia }

    nu = 0.3             { Poisson's Ratio }
    E  = 2.0e11          { Young's Modulus for Steel (N/M^2) }
                         { plane stress coefficients }
    G  = E/(1-nu^2)
    C11 = G
    C12 = G*nu
    C22 = G
    C33 = G*(1-nu)/2
```

```
    amplitude=1e-6      { a guess for grid-plot scaling }
    mag=0.1/amplitude

    force = 250         { total loading force in Newtons (~10 pound force) }
    dist = 0.5*force*(hW^2-y^2)/I      { Distributed load }

    Sx = (C11*dx(U) + C12*dy(V))        { Stresses }
    Sy = (C12*dx(U) + C22*dy(V))
    Txy = C33*(dy(U) + dx(V))

    Vexact = (force/(6*E*I))*((L-x)^2*(2*L+x) + 3*nu*x*y^2)
    Uexact = (force/(6*E*I))*(3*y*(L^2-x^2) +(2+nu)*y^3 -6*(1+nu)*hW^2*y)
    Sxexact = -force*x*y/I
    Txyexact = -0.5*force*(hW^2-y^2)/I

    small = 1e-5

initial values
    U = 0
    V = 0

equations  { define the displacement equations }
    U:  dx(C11*dx(U) + C12*dy(V)) + dy(C33*(dy(U) + dx(V))) = 0
    V:  dx(C33*(dy(U) + dx(V)))   + dy(C12*dx(U) + C22*dy(V)) = 0

{  force regridder to resolve the shear stress.
     Avoid the ends, where the stress is extreme. }
resolve (Txy, 100*(x/L)*(1-x/L))

boundaries
    region 1
      start (0,-hW)

      { free boundary on bottom, no normal stress }
      load(U)=0  load(V)=0  line to (L,-hW)

      { clamp the right end }
      value(U) = Uexact     line to (L,0) point value(V) = 0
      line to (L,hW)

      { free boundary on top, no normal stress }
      load(U)=0  load(V)=0  line to (0,hW)

      { apply distributed load to Y-displacement equation }
      load(U)=0  load(V)=dist   line to close

plots
    grid(x+mag*U,y+mag*V)   as "deformation"   { show final deformed grid }
    elevation(V,Vexact) from(0,0) to (L,0) as "Center Y-Displacement(M)"
    elevation(V,Vexact) from(0,hW) to (L,hW) as "Top Y-Displacement(M)"
    elevation(U,Uexact) from(0,hW) to (L,hW) as "Top X-Displacement(M)"
    elevation(Sx,Sxexact) from(0,hW) to (L,hW) as "Top X-Stress"
    elevation(Sx,Sxexact) from(0,0) to (L,0) as "Center X-Stress"
    elevation(Txy,Txyexact) from(0,hW) to (L,hW) as "Top Shear Stress"
    elevation(Txy,Txyexact) from(0,0) to (L,0) as "Center Shear Stress"
    elevation(Txy,Txyexact) from(hL,-hW) to (hL,hW) as "Center Shear Stress"

end
```

6.2.36 moving_mesh

6.2.36.1 1d_stretchx

```
{ 1D_STRETCHX

  This example demonstrates moving meshes in 1D.
  A Gaussian distribution is defined on a 1D mesh.
  The mesh is  then stretched to twice its initial size,
  while the Gaussian remains fixed in space.

  Mesh motion is imposed by explicit positions of the endpoints.

}
TITLE "stretching line"

COORDINATES
```

```
  cartesian1

VARIABLES
  u
  vx
  xm = move(x)

DEFINITIONS
  Hl = 1/2
  gwid = 0.15
  u0= exp(-x^2/gwid^2)
  lmove = Hl + t

INITIAL VALUES
  u= u0
  vx=x/Hl

EULERIAN EQUATIONS
  U:   dt(u)=0
  Vx:  div(grad(vx))=0
  Xm:  dt(xm) = vx

BOUNDARIES
  REGION 1
    { In 1D, "point" boundary conditions must FOLLOW the point at which
        they are to be applied: }
    START(-Hl)   point value(u)=0 point value(vx)=-1 point value(xm)= -lmove
    Line to (Hl) point value(u)=0 point value(vx)=1  point value(xm)= lmove

TIME 0 TO 0.5 by 0.01

MONITORS
  for cycle=1
    elevation(u,u0) from(-10*Hl) to (10*Hl) range (0,1)
    elevation(dt(xm)) from(-10*Hl) to (10*Hl) range (0,1)

PLOTS
  for time=0.1 by 0.1 to endtime
    elevation(u,u0)   from(-10*Hl) to (10*Hl) range (0,1)
    elevation(vx)     from(-10*Hl) to (10*Hl) range (0,1)
    elevation(dt(xm)) from(-10*Hl) to (10*Hl) range (0,1)

END
```

6.2.36.2 2d_Lagrangian_shock

```
{  2D_LAGRANGIAN_SHOCK.PDE

    This example demonstrates moving meshes in 2D by solving Sod's shock tube problem
    (a 1D problem) on a 2D moving mesh.

    Mesh nodes are given the local fluid velocity, so the model is fully Lagrangian.

    Ref: G.A. Sod, "A Survey of Several Finite Difference Methods for Systems of Nonlinear
    Hyperbolic Conservation Laws", J. Comp. Phys. 27, 1-31 (1978)

    See also Kershaw, Prasad and Shaw, "3D Unstructured ALE Hydrodynamics with the
    Upwind Discontinuous Finite Element Method", UCRL-JC-122104, Sept 1995.

    Other versions of this problem can be found in the "Applications | Fluids" folder.

}

TITLE "Sod's Shock Tube Problem - 2D Lagrangian"
SELECT
  ngrid= 100
  regrid=off
  errlim=1e-4

VARIABLES
  rho(1)
  u(1)
  P(1)
  xm=move(x)

DEFINITIONS
  len = 1
  wid = 0.01
```

```
  gamma = 1.4
  { define a damping term to kill unwanted oscillations }
  eps =  0.001

  v = 0
  rho0  = 1.0 - 0.875*uramp(x-0.49, x-0.51)
  p0 = 1.0 - 0.9*uramp(x-0.49, x-0.51)

INITIAL VALUES
  rho = rho0
  u = 0
  P = p0

EULERIAN EQUATIONS
  { equations are stated as appropriate to the Eulerian (lab) frame.
    FlexPDE will add motion terms to convert to Lagrangian form for moving mesh }
  { since the equation is really in x only, we add dyy(.) terms with natural(.)=0
    on the sidewalls to impose uniformity across the fictitious y coordinate }
  rho:   dt(rho) + u*dx(rho) + rho*dx(u)  = dyy(rho) + eps*dxx(rho)
  u:     dt(u) + u*dx(u) + dx(P)/rho      = dyy(u) + eps*dxx(u)
  P:     dt(P) + u*dx(P) + gamma*P*dx(u)  = dyy(P) + eps*dxx(P)
  xm:    dt(xm) = u

BOUNDARIES
   REGION 1
    { we must impose the same equivalence dt(xm)=u on the side boundaries
    as in the body equations: }
    START(0,0)
    natural(u)=0    dt(xm)=u        line to (len,0)
    value(xm)=len   value(u)=0      line to (len,wid)
    dt(xm)=u        natural(u)=0    line to (0,wid)
    value(xm)=0     value(u)=0      line to close

TIME 0 TO 0.375

MONITORS
  for cycle=5
    grid(x,10*y)
    elevation(rho) from(0,wid/2) to (len,wid/2) range (0,1)
    elevation(u)   from(0,wid/2) to (len,wid/2) range (0,1)
    elevation(P)   from(0,wid/2) to (len,wid/2) range (0,1)

PLOTS
  for t=0 by 0.02 to 0.143, 0.16 by 0.02 to 0.375
    grid(x,10*y)
    elevation(rho) from(0,wid/2) to (len,wid/2) range (0,1)
    elevation(u)   from(0,wid/2) to (len,wid/2) range (0,1)
    elevation(P)   from(0,wid/2) to (len,wid/2) range (0,1)

END
```

6.2.36.3 2D_movepoint

```
{  2D_MOVEPOINT.PDE

  This example is a variation  of 2D_STRETCH_XY.PDE 497 demonstrating the use of
  moving and non-moving point declarations.

  A point defined by name and used in the construction of the domain will move
  when the mesh moves.  The "NODE POINT" declaration will also create a movable
  point.

  Points declared explicitly or not used in the mesh construction will remain fixed.

}
 TITLE "stretching brick"

SELECT
  regrid=off

coordinates
  cartesian2('x','y')

VARIABLES
  u
  vx
  xm = move(x)
  vy
```

```
  ym = move(y)

DEFINITIONS
  Hl = 1/2
  gwid = 0.15
  u0= exp(-(x^2+y^2)/gwid^2)
  lmove = Hl + t
  ms = gwid^2/u0
  P = point(Hl,Hl)    ! a point that IS used in domain construction
  Q = point(0.1,0)    ! a point that is used in "node point"
  R = point(-0.2,-0.2)    ! a point that is NOT used in domain construction

INITIAL VALUES
  u= u0
  vx = x/Hl
  vy = y/Hl

EQUATIONS
  U:   dt(u)=0
  vx:  div(grad(vx))=0
  Xm:  dt(xm) = vx
  vy:  div(grad(vy))=0
  Ym:  dt(ym) = vy

BOUNDARIES
  REGION 1
    mesh_spacing = ms
    START(-Hl,-Hl)
    value(u) = 0  nobc(vx) nobc(xm) value(vy)=-1 value(ym)=-lmove
    Line to (Hl,-Hl)
    value(u)=0  value(vx) = 1 value(xm)=lmove nobc(vy) nobc(ym)
    line to P
    value(u)=0 nobc(vx) nobc(xm) value(vy)=1 value(ym) = lmove
    line to (-Hl,Hl)
    value(u)=0 value(vx)=-1 value(xm)=-lmove nobc(vy) nobc(ym)
    line to close

  NODE POINT q

TIME 0 TO 0.5 by 0.01! 10

MONITORS
  for cycle=1
    grid(x,y) zoom(-Hl-1/2,-Hl-1/2, 2*Hl+1,2*Hl+1)
grid(x,y) zoom(-0.6,0.4, 0.2,0.2)
contour(vx) zoom(-0.6,0.4, 0.2,0.2)
contour(vy) zoom(-0.6,0.4, 0.2,0.2)
    contour(u)
    elevation(u,u0) from(-10*Hl,0) to (10*Hl,0) range (0,1)
    elevation(u,u0) from(0,-10*Hl) to (0,10*Hl) range (0,1)

PLOTS
  for time=0.1 by 0.1 to endtime
    grid(x,y) zoom(-Hl-1/2,-Hl-1/2, 2*Hl+1,2*Hl+1)
    contour(u)
    contour(u-u0)  as "True Total Error"
    contour(error) as "Estimated Step Error" painted
    elevation(u,u0)   from(-10*Hl,0) to (10*Hl,0) range (0,1)
    elevation(dt(xm)) from(-10*Hl,0) to (10*Hl,0) range (0,1)
    elevation(u,u0)   from(0,-10*Hl) to (0,10*Hl) range (0,1)
    elevation(dt(ym)) from(0,-10*Hl) to (0,10*Hl) range (0,1)

  History(u) at P,Q, (0.2,0)  as "Points a and b move, c does not"
  History(u) at R,(-0.2,-0.2) as "neither point moves"

END
```

6.2.36.4 2d_position_blob

```
{ 2D_POSITION_BLOB.PDE

  This problem illustrates moving meshes in 2D.
  A circular boundary shrinks and grows sinusoidally in time.
  The mesh coordinates are solved directly, without a mesh velocity variable.

  See 2D_VELOCITY_BLOB.PDE 498 for a version that uses mesh velocity variables.

}
```

```
TITLE 'Pulsating circle in 2D - position specification'

COORDINATES
  cartesian2

VARIABLES
  Phi            { the temperature }
  Xm = MOVE(x)   { surrogate X }
  Ym = MOVE(y)   { surrogate Y }

DEFINITIONS
  K = 1          { default conductivity }
  R0 = 0.75      { initial blob radius }
  Um = dt(Xm)
  Vm = dt(Ym)

INITIAL VALUES
  Phi = (y+1)/2

EULERIAN EQUATIONS
  Phi:  Div(-k*grad(phi)) = 0
  Xm:  div(grad(Xm)) = 0
  Ym:  div(grad(Ym)) = 0

BOUNDARIES
  REGION 1 'box'
    START(-1,-1)
      VALUE(Phi)=0
      VELOCITY(Xm)=0 VELOCITY(Ym)=0
    LINE TO (1,-1)
      NATURAL(Phi)=0
    LINE TO (1,1)
      VALUE(Phi)=1
    LINE TO (-1,1)
      NATURAL(Phi)=0
    LINE TO CLOSE
  REGION 2  'blob'   { the embedded blob }
    k = 0.001
    START 'ring' (R0,0)
      VELOCITY(Xm) = -0.25*sin(t)*x/r
      VELOCITY(Ym) = -0.25*sin(t)*y/r
    ARC(CENTER=0,0) ANGLE=360 TO CLOSE

TIME 0 TO 2*pi

MONITORS
  for cycle=1
    grid(x,y)
    contour(phi)
PLOTS
  FOR T = 0 BY pi/20 TO 2*pi
    GRID(x,y)
    CONTOUR(Phi)  notags nominmax
    VECTOR(-k*grad(Phi))
    CONTOUR(magnitude(Um,Vm))
    VECTOR(Um,Vm) fixed range(0,0.25)
    ELEVATION(Phi) FROM (0,-1) to (0,1)
    ELEVATION(Normal(-k*grad(Phi))) ON 'ring'
END
```

6.2.36.5 2d_stretch_x

```
{  2D_STRETCH_X.PDE

  This example demonstrates moving meshes in 2D.

  A 1D Gaussian distribution is defined on a 2D mesh.
  The mesh is then stretched to twice its initial X size,
  while the Gaussian remains fixed in space.

  Elevation displays show that the Gaussian retains its correct
  shape as it moves through the mesh.

  Mesh motion is imposed by explicit positions of the endpoints.

}
TITLE "2D brick stretching in x"
```

```
VARIABLES
  u
  vx
  xm = move(x)

DEFINITIONS
  Hl = 1/2
  wid = 0.01
  gwid = 0.15
  u0= exp(-x^2/gwid^2)
  lmove = Hl + t

INITIAL VALUES
  u= u0
  vx = x/Hl

EULERIAN EQUATIONS
  U:   dt(u)=0
  Vx:  div(grad(vx)) = 0
  Xm:  dt(xm) = vx

BOUNDARIES
  REGION 1
    START(-Hl,0)                                  line to (Hl,0)
    value(u)=0 value(vx)=1 value(xm)=lmove        line to (Hl,wid)
    natural(u)=0 nobc(vx) nobc(xm)                line to (-Hl,wid)
    value(u)=0 value(vx)=-1 value(xm)=-lmove      line to close

TIME 0 TO 0.5 by 0.01

MONITORS
  for time=0
    grid(x,10*y) as "Initial mesh"
    contour(vx)

  for cycle=1
    grid(x,10*y)
    contour(u)      zoom(-2*Hl,0, 4*Hl,2*wid)
    contour(vx)     zoom(-2*Hl,0, 4*Hl,2*wid)
    contour(dt(xm)) zoom(-2*Hl,0, 4*Hl,2*wid)
    elevation(u,u0) from(-10*Hl,wid/2)  to (10*Hl,wid/2) range (0,1)
    elevation(vx)   from(-10*Hl,wid/2)  to (10*Hl,wid/2) range (0,1)
    elevation(dt(xm)) from(-10*Hl,wid/2) to (10*Hl,wid/2) range (0,1)

PLOTS
  for time=0.1 by 0.1 to endtime
    grid(x,10*y)
    contour(u) zoom(-2*Hl,0, 4*Hl,2*wid)
    contour(vx) zoom(-2*Hl,0, 4*Hl,2*wid)
    contour(dt(xm)) zoom(-2*Hl,0, 4*Hl,2*wid)
    elevation(u,u0) from(-10*Hl,wid/2) to (10*Hl,wid/2) range (0,1)
END
```

6.2.36.6 2d_stretch_xy

```
{  2D_STRETCH_XY.PDE

  This example demonstrates moving meshes in 2D.
  A Gaussian distribution is defined on a 2D mesh.
  The mesh is  then stretched to twice its initial size,
  while the Gaussian remains fixed in space.

  Output plots show that the Gaussian has retained its shape as
  it moves through the mesh.

  Mesh motion is imposed by explicit positions of the endpoints.

}

TITLE "stretching brick"

SELECT
  regrid=off

coordinates
  cartesian2('x','y')

VARIABLES
```

```
  u
  vx
  xm = move(x)
  vy
  ym = move(y)

DEFINITIONS
  Hl = 1/2
  gwid = 0.15
  u0= exp(-(x^2+y^2)/gwid^2)
  lmove = Hl + t
  ms = gwid^2/u0

INITIAL VALUES
  u= u0
  vx = x/Hl
  vy = y/Hl

EULERIAN EQUATIONS
  U:  dt(u)=0
  vx:  div(grad(vx))=0
  xm:  dt(xm) = vx
  vy:  div(grad(vy))=0
  ym:  dt(ym) = vy

BOUNDARIES
  REGION 1
    mesh_spacing = ms
    START(-Hl,-Hl)
    value(u) = 0  nobc(vx) nobc(xm) value(vy)=-1 value(ym)=-lmove
      line to (Hl,-Hl)
    value(u)=0  value(vx)=1 value(xm)=lmove nobc(vy) nobc(ym)
      line to (Hl,Hl)
    value(u)=0 nobc(vx) nobc(xm) value(vy)=1 value(ym)=lmove
      line to (-Hl,Hl)
    value(u)=0 value(vx) = -1 value(xm)=-lmove nobc(vy) nobc(ym)
      line to close

TIME 0 TO 0.5 by 0.01! 10

MONITORS
  for cycle=1
    grid(x,y) zoom(-Hl-1/2,-Hl-1/2, 2*Hl+1,2*Hl+1)
    contour(u)
    elevation(u,u0) from(-10*Hl,0) to (10*Hl,0) range (0,1)
    elevation(u,u0) from(0,-10*Hl) to (0,10*Hl) range (0,1)

PLOTS
  for time=0.1 by 0.1 to endtime
    grid(x,y) zoom(-Hl-1/2,-Hl-1/2, 2*Hl+1,2*Hl+1)
    contour(u)
    contour(u-u0) as "True Total Error"
    contour(error) as "Estimated Step Error"
    elevation(u,u0)    from(-10*Hl,0) to (10*Hl,0) range (0,1)
    elevation(dt(xm)) from(-10*Hl,0) to (10*Hl,0) range (0,1)
    elevation(u,u0)    from(0,-10*Hl) to (0,10*Hl) range (0,1)
    elevation(dt(ym)) from(0,-10*Hl) to (0,10*Hl) range (0,1)

END
```

6.2.36.7 2d_velocity_blob

```
{ 2D_VELOCITY_BLOB.PDE

  This problem illustrates moving meshes in 2D.
  A circular boundary shrinks and grows sinusoidally in time.
  The mesh coordinates are solved by reference to a mesh velocity variable.

  See 2D_POSITION_BLOB.PDE 495 for a version that uses no mesh velocity variables.

}
TITLE 'Pulsating circle in 2D - velocity specification'

COORDINATES
  cartesian2

VARIABLES
  Phi           { the temperature }
```

```
  Xm = MOVE(x)   { surrogate X }
  Ym = MOVE(y)   { surrogate Y }
  Um(0.1)        { mesh x-velocity }
  Vm(0.1)        { mesh y-velocity }

DEFINITIONS
  K = 1    { default conductivity }
  R0 = 0.75    { initial blob radius }

INITIAL VALUES
  Phi = (y+1)/2

EULERIAN EQUATIONS
  Phi:  Div(-k*grad(phi)) = 0
  Xm:  dt(Xm) = Um
  Ym:  dt(Ym) = Vm
  Um:  div(grad(Um)) = 0
  Vm:  div(grad(Vm)) = 0

BOUNDARIES
  REGION 1 'box'
    START(-1,-1)
      VALUE(Phi)=0
      VELOCITY(Xm)=0 VELOCITY(Ym)=0
      VALUE(Um)=0 VALUE(Vm)=0
    LINE TO (1,-1)
      NATURAL(Phi)=0
    LINE TO (1,1)
      VALUE(Phi)=1
    LINE TO (-1,1)
      NATURAL(Phi)=0
    LINE TO CLOSE
  REGION 2  'blob'  { the embedded blob }
    k = 0.001
    START 'ring' (R0,0)
      VELOCITY(Xm) = Um
      VELOCITY(Ym) = Vm
      VALUE(Um) = -0.25*sin(t)*x/r
      VALUE(Vm) = -0.25*sin(t)*y/r
    ARC(CENTER=0,0) ANGLE=360 TO CLOSE

TIME 0 TO 2*pi

MONITORS
  for cycle=1
    grid(x,y)
    contour(phi)

PLOTS
  FOR T = 0 BY pi/20 TO 2*pi
    GRID(x,y)
    CONTOUR(Phi)  notags nominmax
    VECTOR(-k*grad(Phi))
    CONTOUR(magnitude(Um,Vm))
    VECTOR(Um,Vm) fixed range(0,0.25)
    ELEVATION(Phi) FROM (0,-1) to (0,1)
    ELEVATION(Normal(-k*grad(Phi))) ON 'ring'

END
```

6.2.36.8 3d_position_blob

```
{ 3D_POSITION_BLOB.PDE

  This problem illustrates moving meshes in 3D.
  A spherical boundary shrinks and grows sinusoidally in time.
  The mesh coordinates are solved directly, without a mesh velocity variable.

  See 3D_VELOCITY_BLOB.PDE 501 for a version that uses mesh velocity variables.

}
TITLE 'Pulsating circle in 3D - position specification'

COORDINATES
  cartesian3

VARIABLES
  Phi    { the temperature }
```

```
  Xm = MOVE(x)  { surrogate X }
  Ym = MOVE(y)  { surrogate Y }
  Zm = MOVE(z)  { surrogate Z }

DEFINITIONS
  K = 1    { default conductivity }
  R0 = 0.75    { initial blob radius }

  zsphere = SPHERE ((0,0,0),R0)
  z1, z2
  Um = dt(Xm)
  Vm = dt(Ym)
  Wm = dt(Zm)

INITIAL VALUES
  Phi = (z+1)/2

EULERIAN EQUATIONS
  Phi:  Div(-k*grad(phi)) = 0
  Xm:  div(grad(Xm)) = 0
  Ym:  div(grad(Ym)) = 0
  Zm:  div(grad(Zm)) = 0

EXTRUSION
  SURFACE 'Bottom'           z = -1
  SURFACE 'Sphere Bottom'    z=z1
  SURFACE 'Sphere Top'       z=z2
  SURFACE 'Top'              z=1

BOUNDARIES
 SURFACE 1
    VALUE(Phi)=0 VELOCITY(Xm)=0 VELOCITY(Ym)=0 VELOCITY(Zm)=0
 SURFACE 4
    VALUE(Phi)=1 VELOCITY(Xm)=0 VELOCITY(Ym)=0 VELOCITY(Zm)=0

REGION 1 'box'
  z1=0   z2=0
  START(-1,-1)
    NATURAL(Phi)=0 VELOCITY(Xm)=0 VELOCITY(Ym)=0 VELOCITY(Zm)=0
  LINE TO (1,-1) TO (1,1) TO (-1,1) TO CLOSE

LIMITED REGION 2  'blob'  { the embedded blob }
  z1 = -zsphere
  z2 = zsphere
  layer 2 k = 0.001
  SURFACE 2
    VELOCITY(Xm) = -0.25*sin(t)*x/r
    VELOCITY(Ym) = -0.25*sin(t)*y/r
    VELOCITY(Zm) = -0.25*sin(t)*z/r
  SURFACE 3
    VELOCITY(Xm) = -0.25*sin(t)*x/r
    VELOCITY(Ym) = -0.25*sin(t)*y/r
    VELOCITY(Zm) = -0.25*sin(t)*z/r
  START 'ring' (R0,0)
  ARC(CENTER=0,0) ANGLE=360 TO CLOSE

TIME 0 TO 2*pi

MONITORS
  FOR cycle=1
    GRID(x,y,z) ON 'blob' ON LAYER 2
    CONTOUR(phi) ON y=0
PLOTS
  FOR T = 0 BY pi/20 TO 2*pi
    GRID(x,y,z) ON 'blob' ON LAYER 2 FRAME(-R0,-R0,-R0, 2*R0,2*R0,2*R0)
    CONTOUR(Phi)  notags nominmax ON y=0
    VECTOR(-k*grad(Phi)) ON y=0
    CONTOUR(magnitude(Um,Vm,Wm)) ON y=0
    VECTOR(Um,Wm) ON y=0 FIXED RANGE(0,0.25)
    ELEVATION(Phi) FROM (0,0,-1) TO (0,0,1)
    ELEVATION(magnitude(Um,Vm,Wm)) FROM (0,0,-1) TO (0,0,1)

END
```

6.2.36.9 3d_velocity_blob

```
{ 3D_VELOCITY_BLOB.PDE

  This problem illustrates moving meshes in 3D.
  A spherical boundary shrinks and grows sinusoidally in time.
  The mesh coordinates are solved by reference to a mesh velocity variable.

  See 3D_POSITION_BLOB.PDE 499 for a version that uses no mesh velocity variables.

}
TITLE 'Pulsating circle in 3D - velocity specification'

COORDINATES
  cartesian3

VARIABLES
  Phi     { the temperature }
  Xm = MOVE(x)   { surrogate X }
  Ym = MOVE(y)   { surrogate Y }
  Zm = MOVE(z)   { surrogate Z }
  Um(0.1)    { mesh x-velocity }
  Vm(0.1)     { mesh y-velocity }
  Wm(0.1)     { mesh z-velocity }

DEFINITIONS
  K = 1     { default conductivity }
  R0 = 0.75     { initial blob radius }

  zsphere = SPHERE ((0,0,0),R0)
  z1, z2

INITIAL VALUES
  Phi = (z+1)/2

EULERIAN EQUATIONS
  Phi:  Div(-k*grad(phi)) = 0
  Xm:  dt(Xm) = Um
  Ym:  dt(Ym) = Vm
  Zm:  dt(Zm) = Wm
  Um:  div(grad(Um)) = 0
  Vm:  div(grad(Vm)) = 0
  Wm:  div(grad(Wm)) = 0

EXTRUSION
  SURFACE 'Bottom'             z = -1
  SURFACE 'Sphere Bottom'      z=z1
  SURFACE 'Sphere Top'         z=z2
  SURFACE 'Top'                z=1

BOUNDARIES
 SURFACE 1
    VALUE(Phi)=0 VELOCITY(Xm)=0 VELOCITY(Ym)=0 VELOCITY(Zm)=0
    VALUE(Um)=0 VALUE(Vm)=0 VALUE(Wm)=0
 SURFACE 4
    VALUE(Phi)=1 VELOCITY(Xm)=0 VELOCITY(Ym)=0 VELOCITY(Zm)=0
    VALUE(Um)=0 VALUE(Vm)=0 VALUE(Wm)=0

REGION 1 'box'
  z1=0   z2=0
  START(-1,-1)
    NATURAL(Phi)=0
    VELOCITY(Xm)=0 VELOCITY(Ym)=0 VELOCITY(Zm)=0
    VALUE(Um)=0 VALUE(Vm)=0 VALUE(Wm)=0
  LINE TO (1,-1) TO (1,1) TO (-1,1) TO CLOSE

LIMITED REGION 2  'blob'  { the embedded blob }
  z1 = -zsphere
  z2 = zsphere
  layer 2 k = 0.001
  SURFACE 2
    VELOCITY(Xm) = Um  VELOCITY(Ym) = Vm  VELOCITY(Zm) = Wm
    VALUE(Um) = -0.25*sin(t)*x/r
    VALUE(Vm) = -0.25*sin(t)*y/r
    VALUE(Wm) = -0.25*sin(t)*z/r
  SURFACE 3
    VELOCITY(Xm) = Um  VELOCITY(Ym) = Vm  VELOCITY(Zm) = Wm
    VALUE(Um) = -0.25*sin(t)*x/r
    VALUE(Vm) = -0.25*sin(t)*y/r
```

```
    VALUE(Wm) = -0.25*sin(t)*z/r
  START 'ring' (R0,0)
  ARC(CENTER=0,0) ANGLE=360 TO CLOSE

TIME 0 TO 2*pi

MONITORS
  FOR cycle=1
    GRID(x,y,z) ON 'blob' ON LAYER 2
    CONTOUR(phi) ON y=0
PLOTS
  FOR T = 0 BY pi/20 TO 2*pi
    GRID(x,y,z) ON 'blob' ON LAYER 2
    CONTOUR(Phi)  notags nominmax ON y=0
    VECTOR(-k*grad(Phi)) ON y=0
    CONTOUR(magnitude(Um,Vm,Wm)) ON y=0
    VECTOR(Um,Wm) ON y=0 FIXED RANGE(0,0.25)
    ELEVATION(Phi) FROM (0,0,-1) TO (0,0,1)
    ELEVATION(magnitude(Um,Vm,Wm)) FROM (0,0,-1) TO (0,0,1)

END
```

6.2.37 ode

6.2.37.1 linearode

```
{ LINEARODE.PDE

  This example shows the application of FlexPDE to the solution of a linear
  first-order differential equation.

  We select the simple example
     dH/dt = a - b*H

  This equation has the exact solution
     H(t) = H(0)*exp(-b*t) + (a/b)*(1-exp(-b*t))

  The existence of an exact solution allows us to analyze the errors
  in the FlexPDE solution.

  Since FlexPDE requires a spatial domain, we solve the system over
  a simple box with minimum mesh size.

}

title
  "FIRST ORDER ORDINARY DIFFERENTIAL EQUATION"

select
  { Since no spatial information is required, use the minimum grid }
  ngrid=1
  errlim = 1e-4

variables
  { declare Height to be the system variable }
  Height(threshold=1)

definitions
  { define the equation parameters }
  a = 2
  b = 0.1
  H0 = 100
  { define the exact solution: }
  Hexact = H0*exp(-b*t) + (a/b)*(1-exp(-b*t))

initial values
  Height = H0

equations
  Height : dt(Height)  = a - b*Height    { The ODE }

boundaries
  region 1
  start (0,0)
  line to (1,0) to (1,1) to (0,1) to close

time 0 to 100
```

```
plots
  for time = 0,1,10 by 10 to 100
  { Plot the solution: }
  history(Height) at (0.5,0.5)
  { Plot the error check: }
  history((Height-Hexact)/Hexact) at (0.5,0.5) as "Relative Error"

end
```

6.2.37.2 nonlinode

```
{ NONLINODE.PDE

  This example shows the application of FlexPDE to the solution of a
  non-linear first-order differential equation.

  A liquid flows into the top of a reactor vessel through an unrestricted
  pipe and exits from the bottom through a choke value.  This problem is
  discussed in detail in Silebi and Schiesser.

  This is a problem in viscous flow:
     dH/dt = a - b*sqrt(H)

  The analytic solution satisfies the relation
     sqrt(H0) + (a/b)ln[a-b*sqrt(H0)]
      - sqrt(H) - (a/b)ln[a-b*sqrt(H)] = (b/2)*t

  which can be used as an accuracy check.

  Since FlexPDE requires a spatial domain, we solve the equation on
  a simple box with minimum mesh size.

}

title
  "NONLINEAR FIRST ORDER ORDINARY DIFFERENTIAL EQUATION"

select
  { Since there is no spatial information required, use the minimum grid size }
  ngrid=1

variables
  { declare Height to be the system variable }
  Height(threshold=1)

definitions
  { define the equation parameters }
  a = 2
  b = 0.1
  H0 = 100
  { define the accuracy check }
  T0 = sqrt(H0) + (a/b)*ln(a-b*sqrt(H0))
  Tcheck = sqrt(Height) + (a/b)*ln(a-b*sqrt(Height))

initial values
  Height = H0

equations   { The ODE }
  Height : dt(Height)  = a - b*sqrt(Height)

boundaries
  { define a fictitious spatial domain }
  region 1
    start (0,0)
    line to (1,0) to (1,1) to (0,1) to close

{ define the time range }
time 0 to 1000

plots
  for t=0, 1, 10 by 10 to endtime
    { Plot the solution: }
    history(Height) at (0.5,0.5)
    { Plot the accuracy check: }
    history((T0 - Tcheck - (b/2)*t)/((b/2)*t)) at (0.5,0.5)
            as "Relative Error"
```

```
end
```

6.2.37.3 second_order_time

```
{  SECOND_ORDER_TIME.PDE

  This example shows the integration of Bessel's Equation as a test of the
  time integration capabilities of FlexPDE.

  Bessel's Equation for order zero can be written as
    t^2*dtt(w) + t*dt(w) + t^2*w = 0

  Dividing by t^2 and avoiding the pole at t=0, we can write
    dtt(w) + dt(w)/t + w = 0

  FlexPDE cannot directly integrate second order equations, so we define an
  auxiliary variable v=dt(w) and write a coupled pair of equations
    dt(v) + v/t + w = 0
    dt(w) = v

  We use a dummy spatial grid of two cells and solve the equation at each node.

  You can try varying the value given for ERRLIM 148 to see how it behaves.

 }

title "Integration of Bessel's Equation"

select
    ngrid=1
    errlim=1e-5  { increase accuracy to prevent accumulation of errors }

Variables
    v (threshold=0.1)
    w (threshold=0.1)

definitions
    L = sqrt(2)
    t0 = 0.001    { Start integration at t=0.001 }

Initial values    { Initialize to known values at t=t0 }
    w = 1-2.25*(t0/3)^2
    v = -0.5*t0 + 0.5625*t0*(t0/3)^2

equations
    v:  dt(v) +v/t + w = 0
    w:  dt(w) =  v

boundaries
    region 1
        start(-L,-L) line to (L,-L) to (L,L) to (-L,L) to close

time 0.001 to 4*pi    { Exclude t=0 }

plots
    for t=0.01 by 0.01 to 0.1 by 0.1 to 1 by 1 to endtime
      history(w,bessj(0,t)) at (0,0)  as "W(t) and BESSJ0(t)"
      history(w-bessj(0,t)) at (0,0)  as "Absolute Error"
      history(v,-bessj(1,t)) at (0,0)  as "V(t) and dt(BESSJ0(t))"
      history(v+bessj(1,t)) at (0,0)  as "Slope Error"
      history(deltat)

end
```

6.2.38 periodicity

6.2.38.1 3d_antiperiodic

```
{  3D_ANTIPERIODIC.PDE

  This example shows the use of FlexPDE in a 3D problem with azimuthal
  anti-periodicity.
  (See the example ANTIPERIODIC.PDE 507 for notes on antiperiodic boundaries.)

  In this problem we create a repeated 45-degree segment of a ring.
```

```
}

title '3D AZIMUTHAL ANTIPERIODIC TEST'

coordinates cartesian3

Variables
     u

definitions
     k = 1
     { angular size of the repeated segment: }
     an = pi/4
     { the sine and cosine for transformation }
     crot = cos(an)
     srot = sin(an)
     H = 0
     xc = 1.5
     yc = 0.2
     rc = 0.1

equations
     u : div(K*grad(u)) + H = 0

extrusion z=0,0.4,0.6,1

boundaries
     region 1
       { this line forms the remote boundary for the later periodic statement }
       start(1,0) line to (2,0)

       value(u) = 0  arc(center=0,0) to (2*crot,2*srot)

       { The following line segment is periodic under an angular rotation.
            The mapping expressions take each point on the line into a corresponding
            point in the base line.  Note that although all the mapped y-coordinates
            will be zero, we give the general expression so that the transformation
            will be invertible. }
       antiperiodic(x*crot+y*srot, -x*srot+y*crot)
       line to (crot,srot)

       value(u)=0
       arc(center= 0,0)  to close

     limited region 2
        layer 2 H = 1
        start(xc-rc,0) line to (xc+rc,0) to (xc+rc,rc) to (xc-rc,rc) to close

     limited region 3
        layer 2 H = -1
        start((xc-rc)*crot,(xc-rc)*srot)
        line to ((xc+rc)*crot,(xc+rc)*srot)
                to ((xc+rc)*crot+rc*srot,(xc+rc)*srot-rc*crot)
                to ((xc-rc)*crot+rc*srot,(xc-rc)*srot-rc*crot) to close

monitors
     grid(x,y,z)
     contour(u) on z=0.1
     contour(u) on z=0.5
     contour(u) on z=0.9

plots
     grid(x,y,z)
     contour(u) on z=0.1        painted
     contour(u) on z=0.5        painted
     contour(u) on z=0.9        painted
end
```

6.2.38.2 3d_xperiodic

```
{  3D_XPERIODIC.PDE

  This example shows the use of FlexPDE in 3D applications with periodic boundaries.

  The PERIODIC 193 statement appears in the position of a boundary condition, but
  the syntax is slightly different, and the requirements and implications are
  more extensive.
```

```
  The syntax is:
        PERIODIC(X_mapping,Y_mapping)
  The mapping expressions specify the arithmetic required to convert a point
  (X,Y) in the immediate boundary to a point (X',Y') on a remote boundary.
  The mapping expressions must result in each point on the immediate boundary
  mapping to a point on the remote boundary.  Segment endpoints must map to
  segment endpoints.  The transformation must be invertible; do not specify
  constants as mapped coordinates, as this will create a singular transformation.

  The periodic boundary statement terminates any boundary conditions in effect,
  and instead imposes equality of all variables on the two boundaries.  It is
  still possible to state a boundary condition on the remote boundary,
  but in most cases this would be inappropriate.

  The periodic statement affects only the next following LINE[181] or ARC[181] path.
  These paths may contain more than one segment, but the next appearing
  LINE or ARC statement terminates the periodic condition unless the periodic
  statement is repeated.

  In this problem, we have a heat equation with an off=center source in an irregular
  figure.  The figure is periodic in X, with Y faces held at zero, and Z-faces insulated.

}

title '3D X-PERIODIC BOUNDARY TEST'

coordinates
    cartesian3

Variables
    u

definitions
    k = 0.1
    h=0
    x0=0.5   y0=-0.2
    x1=1.1   y1 = 0.2

equations
    u : div(K*grad(u)) + h = 0

extrusion z=0,0.4,0.6,1

boundaries
    region 1
      start(-1,-1)
      value(u)=0   line to (1,-1)   { Force U=0 on Y=-1 }

      { The following arc is required to be a periodic image of an arc
        two units to its left. (This image boundary has not yet been defined.) }
      periodic(x-2,y) arc(center=-1,0) to (1,1)

      value(u)=0   line to (-1,1)      { Force U=0 on Y=1 }

      { The following arc provides the required image boundary for the previous
                periodic statement }
      nobc(u)    { turn off the value BC }
      arc(center= -3,0)  to close

    { an off-center heat source in layer 2 provides the asymmetric conditions to
        demonstrate the periodicity of the solution }
    limited region 2
      layer 2 h=10 k=10
      start(x0,y0) line to (x1,y0) to (x1,y1) to (x0,y1) to close

monitors
    contour(u) on z=0
    contour(u) on z=0.5
    contour(u) on z=1
    contour(u) on y=0

plots
    contour(u) on z=0   painted
    contour(u) on z=0.5 painted
    contour(u) on z=1   painted
    contour(u) on y=0   painted

end
```

6.2.38.3 3d_zperiodic

```
{  3D_ZPERIODIC.PDE

  This example shows the use of FlexPDE in 3D applications with periodic
  boundaries in the Z-direction.

  For Z-periodicity, we merely precede  the EXTRUSION 179 statement by the
  qualifier PERIODIC 193.  The top and bottom surfaces are assumed to match,
  and values are made equal on the two surfaces.

  In this problem we have a heat equation in an irregular figure.
  An off-center source heats the body, while all the vertical surfaces are
  held at U=0.

}

title '3D Z-PERIODIC BOUNDARY TEST'

coordinates
    cartesian3

Variables
     u

definitions
    k = 0.1
    h=0
    x0=0.3  y0=-0.2
    x1=0.7  y1 = 0.2

equations
    u : div(K*grad(u)) + h = 0

periodic extrusion z=0, 0.8, 1

boundaries
    Region 1
      start(-1,-1)
        value(u)=0
      line to (1,-1)
      arc(center=-1,0) to (1,1)
      line to (-1,1)
      arc(center=-3,0) to close


    { an off-center heat source in layer 2 provides the asymmetric
        conditions to demonstrate the periodicity of the solution }
    limited region 2
      layer 2 h=10 k=10
      surface 1 { include insert patch in surface 1 so surfaces match }
      start(x0,y0) line to (x1,y0) to (x1,y1) to (x0,y1) to close

monitors
    contour(u) on y=0

plots
    grid(x,z)  on y=0
    contour(u) on y=0 painted
end
```

6.2.38.4 antiperiodic

```
{  ANTIPERIODIC.PDE

  This example shows the use of FlexPDE in applications with antiperiodic
   boundaries.

  The ANTIPERIODIC 193 statement appears in the position of a boundary condition, but
  the syntax is slightly different, and the requirements and implications are
  more extensive.

  The syntax is:
        ANTIPERIODIC(X_mapping,Y_mapping)
  The mapping expressions specify the arithmetic required to convert a point
```

```
  (X,Y) in the immediate boundary to a point (X',Y') on a remote boundary.
  The mapping expressions must result in each point on the immediate boundary
  mapping to a point on the remote boundary.  Segment endpoints must map to
  segment endpoints.  The transformation must be invertible; do not specify
  constants as mapped coordinates, as this will create a singular transformation.

  The antiperiodic boundary statement terminates any boundary conditions in effect,
  and instead imposes equality of all variables on the two boundaries.  It is
  still possible to state a boundary condition on the remote boundary,
  but in most cases this would be inappropriate.

  The antiperiodic statement affects only the next following LINE 181 or ARC 181 path.
  These paths may contain more than one segment, but the next appearing
  LINE or ARC statement terminates the periodic condition unless the periodic
  statement is repeated.
}

title 'ANTI-PERIODIC BOUNDARY TEST'

Variables
     u

definitions
    k = 0.1
    h=0

equations
    u : div(K*grad(u)) + h = 0

boundaries
    Region 1
      start(-1,-1)
        value(u)=0    line to (1,-1)


      { The following arc is required to be an antiperiodic image of an arc
            two units to its left. (This image boundary has not yet been defined.) }
      antiperiodic(x-2,y) arc(center=-1,0) to (1.2,-0.2)
      antiperiodic(x-2,y) line to (1.2,0.2)
      antiperiodic(x-2,y) arc(center=-1,0) to (1,1)

      value(u)=0 line to (-1,1)

      { The following arc provides the required image boundary for the previous
            antiperiodic statement }
      nobc(u)       { turn off the value BC }
      arc(center= -3,0) to (-0.8,0.2) line to (-0.8,-0.2) arc(center=-3,0) to close

    { an off-center heat source provides the asymmetric conditions to
        demonstrate the antiperiodicity of the solution }
    region 2  h=10 k=10
      start(1.2,-0.2) line to (1.2,0.2) to (1,0.2) to (1,-0.2) to close

    region 3  h=-10 k=10
      start(-0.6,-0.2) line to (-0.6,0.2) to (-0.8,0.2) to (-0.8,-0.2) to close

monitors
     grid(x,y)
     contour(u)

plots
     grid(x,y)
     contour(u)
end
```

6.2.38.5 azimuthal_periodic

```
{  AZIMUTHAL_PERIODIC.PDE

  This example shows the use of FlexPDE in a problem with azimuthal periodicity.
  (See the example PERIODIC.PDE 510 for notes on periodic boundaries.)

  In this problem we create a repeated 45-degree segment of a ring.
}
```

```
title 'AZIMUTHAL PERIODIC TEST'

Variables
    u

definitions
    k = 1
    { angular size of the repeated segment: }
    an = pi/4
    { the sine and cosine for transformation }
    crot = cos(an)
    srot = sin(an)
    H = 0
    xc = 1.5
    yc = 0.2
    rc = 0.1

equations
    u : div(K*grad(u)) + H = 0

boundaries
    region 1
       { this line forms the remote boundary for the later periodic statement }
       start(1,0) line to (2,0)

       value(u)=0  arc(center=0,0) to (2*crot,2*srot)

       { The following line segment is periodic under an angular rotation.
         The mapping expressions take each point on the line into a corresponding
         point in the base line.  Note that although all the mapped y-coordinates
         will be zero, we give the general expression so that the transformation
         will be invertible. }
       periodic(x*crot+y*srot, -x*srot+y*crot)
       line to (crot,srot)

       value(u)=0
       arc(center= 0,0)  to close

    region 2
        H = 1
        start(xc-rc,yc) arc(center=xc,yc) angle=360

monitors
     grid(x,y)
     contour(u)

plots
     grid(x,y) contour(u)

end
```

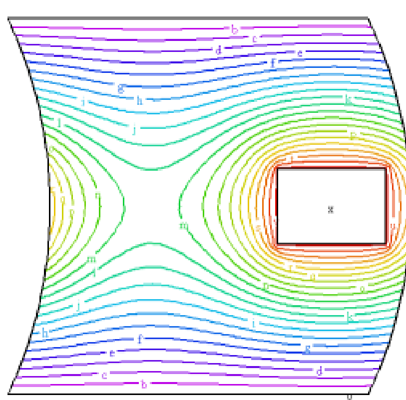
6.2.38.6 periodic+time

```
{ PERIODIC+TIME.PDE

  This example is a time-dependent version of PERIODIC.PDE 510

}

title 'Time-dependent Periodic Boundary Test'

variables
     u(0.01)

definitions
    k = 0.1
    h=0
    x0=0.5  y0=-0.2
    x1=1.1  y1=0.2

equations
    u : div(K*grad(u)) + h = dt(u)

boundaries
    region 1
      start(-1,-1)
      value(u)=0  line to (0.9,-1) to (1,-1)
```

```
    { The following arc is required to be a periodic image of an arc
      two units to its left. (This image boundary has not yet been defined.) }
    periodic(x-2,y) arc(center=-1,0) to (1,1)

    value(u)=0   line to (-1,1)

    { The following arc provides the required image boundary for the previous
      periodic statement }
    nobc(u)   { turn off the value BC }
    arc(center= -3,0)  to close

  { an off-center heat source provides the asymmetric conditions to
    demonstrate the periodicity of the solution }
  region 2  h=10 k=10
    start(x0,y0) line to (x1,y0) to (x1,y1) to (x0,y1) to close

time 0 to 10

monitors
  for cycle=1
    grid(x,y)
    contour(u)

plots
  for cycle=10
    grid(x,y)
    contour(u)

end
```

6.2.38.7 periodic

```
{ PERIODIC.PDE

  This example shows the use of FlexPDE in applications with periodic boundaries.

  The PERIODIC 193 statement appears in the position of a boundary condition, but
  the syntax is slightly different, and the requirements and implications are
  more extensive.

  The syntax is:
        PERIODIC(X_mapping,Y_mapping)
  The mapping expressions specify the arithmetic required to convert a point
  (X,Y) in the immediate boundary to a point (X',Y') on a remote boundary.
  The mapping expressions must result in each point on the immediate boundary
  mapping to a point on the remote boundary.  Segment endpoints must map to
  segment endpoints.  The transformation must be invertible; do not specify
  constants as mapped coordinates, as this will create a singular transformation.

  The periodic boundary statement terminates any boundary conditions in effect,
  and instead imposes equality of all variables on the two boundaries.  It is
  still possible to state a boundary condition on the remote boundary,
  but in most cases this would be inappropriate.

  The periodic statement affects only the next following LINE 181 or ARC 181 path.
  These paths may contain more than one segment, but the next appearing
  LINE or ARC statement terminates the periodic condition unless the periodic
  statement is repeated.

}

title 'PERIODIC BOUNDARY TEST'

variables
     u

definitions
    k = 0.1
    h=0
    x0=0.5  y0=-0.2
    x1=1.1  y1=0.2

equations
    u : div(K*grad(u)) + h = 0

boundaries
    region 1
```



```
        start(-1,-1)
        value(u)=0    line to (0.9,-1) to (1,-1)

        { The following arc is required to be a periodic image of an arc
          two units to its left. (This image boundary has not yet been defined.) }
        periodic(x-2,y) arc(center=-1,0) to (1,1)

        value(u)=0   line to (-1,1)

        { The following arc provides the required image boundary for the previous
          periodic statement }
        nobc(u)   { turn off the value BC }
        arc(center= -3,0)  to close

      { an off-center heat source provides the asymmetric conditions to
        demonstrate the periodicity of the solution }
      region 2   h=10 k=10
        start(x0,y0) line to (x1,y0) to (x1,y1) to (x0,y1) to close

monitors
     grid(x,y)
     contour(u)

plots
     grid(x,y)
     contour(u)

end
```

6.2.38.8 two-way_periodic

```
{ TWO-WAY_PERIODIC.PDE

  This example shows the use of FlexPDE in applications with two-way periodic boundaries.

   FlexPDE cannot support multiple periodic images of a single point, so straightforward
     request for periodicity in both X nad Y will not work.
   If a small boundary segment is introduced at the corner point, however, then no point
     is imaged twice, and the specification will be accepted.
   The default boundary condition on the small non-periodic segment will be natural()=0, which
     is not strictly correct, but if the segment is short and located in a region of relative
     inactivity, the distortion should not be significant.
   Alternatively, the "tautological" boundary condition may be used.  This condition merely
     supplies the surface terms required by the definition of the natural BC.  In the diffusion
     equation used in this example it is simply Natural()=normal(K*grad(u)).
}

title 'TWO-WAY PERIODIC BOUNDARY TEST'

variables
     u

definitions
    k = 1
    h=0
    x0=0.4   x1=0.9  { right box x-coordinates }
    x2=-0.5  x3=0.0  { left box x-coordinates }
    y0=-0.7 y1 = -0.3  {y-coordinates for both boxes }

equations
    u : div(K*grad(u)) + h = 0

boundaries
    region 1

      { Periodic bottom boundary }
      start(-1,-1)
      periodic(x,y+2)  line to(0.95,-1)
      { New "line" spec breaks periocity }

      {   optional:   natural(u) = normal(K*grad(u)) }
      line to (1,-1)

      { Periodic right boundary }
      periodic(x-2,y) arc(center=-1,0) to (1,1)

      { Images of non-periodic stub and periodic bottom boundry }
      line to (0.95,1) to (-1,1)
```

```
        { Image of periodic right boundary }
        arc(center= -3,0)  to close

        { off-center hot box }
        start(x0,y0)
        value(u)=1    line to (x1,y0) to (x1,y1) to (x0,y1) to close

        { off-center cold box }
        start(x2,y0)
        value(u)=-1   line to (x3,y0) to (x3,y1) to (x2,y1) to close

monitors
     grid(x,y)
     contour(u)

plots
     grid(x,y)
     contour(u)
end
```

6.2.39 plotting

6.2.39.1 3d_ploton

```
{ 3D_PLOTON.PDE

  This problem shows some of the possible 'ON' 206 qualifiers for 3D plots.

}

title '3D Test -- Plot Qualifiers'

coordinates
    cartesian3

Variables
    u

definitions
    k = 0.1
    heat = 4

equations
    U:     div(K*grad(u)) + heat   = 0

extrusion
    surface z = 0
    surface z = 0.8-0.3*(x^2+y^2)
    surface z = 1.0-0.3*(x^2+y^2)


boundaries
    region 1 'outer'
        layer 2 k = 1
        start(-1,-1)
            value(u) = 0
        line to (1,-1) to (1,1) to (-1,1) to close
    region 2 'plug'
        layer 2 k = 1
        start 'dot' (0.5,0.5) arc(center=0,0) angle=360

plots
     grid(x,y,z) on region 1 as "Only Region 1, both layers"
     grid(x,y,z) on region 'plug' on layer 2  as "Region 2 Layer 2"
     grid(x,y,z) on region 'plug' on layers 1,2 paintregions as "Region 2, both layers"
     grid(y,z)  on x=0 on 'plug'    as "Cut plane on region 2"
     contour(u) on x=0.51    on layer 2  as "Solution on X-cut in layer 2"
     contour(u) on z=0.51   on  region 2  as "Solution on Z-cut in region 2"
     contour(u) on surface 2  on region 2          as "Solution on paraboloidal layer interface"
     vector(grad(u)) on surface 2 on 'outer'  as "Flux on layer interface in region 1"

end
```

6.2.39.2 plot_on_grid

```
{ PLOT_ON_GRID.PDE

  This is a variation of BENTBAR.PDE 366 that makes use
  of the version 6.03 capability to plot contours
  on a deformed grid.

  The syntax of the plot command is
        CONTOUR(data) ON GRID(Xposition,Yposition) 208

}

title "Contour plots on a deformed grid"

select
    cubic       { Use Cubic Basis }

variables
    U           { X-displacement }
    V           { Y-displacement }


definitions
    L = 1               { Bar length }
    hL = L/2
    W = 0.1             { Bar thickness }
    hW = W/2
    eps = 0.01*L
    I = 2*hW^3/3        { Moment of inertia }

    nu = 0.3            { Poisson's Ratio }
    E  = 2.0e11         { Young's Modulus for Steel (N/M^2) }
                        { plane stress coefficients }
    G  = E/(1-nu^2)
    C11 = G
    C12 = G*nu
    C22 = G
    C33 = G*(1-nu)/2

    amplitude=GLOBALMAX(abs(v)) { for grid-plot scaling }
    mag=1/amplitude

    force = -250         { total loading force in Newtons (~10 pound force) }
    dist = 0.5*force*(hW^2-y^2)/I       { Distributed load }

    Sx = (C11*dx(U) + C12*dy(V))        { Stresses }
    Sy = (C12*dx(U) + C22*dy(V))
    Txy = C33*(dy(U) + dx(V))

    { Timoshenko's analytic solution:  }
    Vexact = (force/(6*E*I))*((L-x)^2*(2*L+x) + 3*nu*x*y^2)
    Uexact = (force/(6*E*I))*(3*y*(L^2-x^2) +(2+nu)*y^3 -6*(1+nu)*hW^2*y)
    Sxexact = -force*x*y/I
    Txyexact = -0.5*force*(hW^2-y^2)/I

initial values
    U = 0
    V = 0

equations               { the displacement equations }
    U:  dx(Sx) + dy(Txy) = 0
    V:  dx(Txy) + dy(Sy) = 0

boundaries
    region 1
      start (0,-hW)

      load(U)=0         { free boundary on bottom, no normal stress }
      load(V)=0
        line to (L,-hW)

      value(U) = Uexact { clamp the right end }
      mesh_spacing=hW/10
        line to (L,0) point value(V) = 0
        line to (L,hW)

      load(U)=0         { free boundary on top, no normal stress }
```

```
        load(V)=0
        mesh_spacing=10
          line to (0,hW)

        load(U) = 0
        load(V) = dist    { apply distributed load to Y-displacement equation }
          line to close

plots
    grid(x+mag*U,y+mag*V)   as "deformation"   { show final deformed grid }

 ! STANDARD PLOTS:
 contour(U)
    surface(U)

    ! THE DEFORMED PLOTS:
 contour(U) on grid(x+mag*U,y+mag*V)
    surface(U) on grid(x+mag*U,y+mag*V)

end
```

6.2.39.3 plot_test

```
{ PLOT_TEST.PDE

  This example shows the use of various options in plotted output.

  The problem is the same as PLATE_CAPACITOR.PDE 301 .

}

title 'Plate capacitor'

select
     contourgrid=50     { default=40 }
     surfacegrid=60     { default=40 }
     elevationgrid=200  { default=120 }

Variables
     v

definitions
     Lx=2       Ly=2
     delx=0.25*Ly
     d=0.1*Ly         ddy=0.1*d
     Ex=-dx(v)   Ey=-dy(v)
     Eabs=sqrt(Ex^2+Ey^2)
     eps0=8.854e-12
     eps
     DEx=eps*Ex         DEy=eps*Ey
     Dabs=sqrt(DEx^2+DEy^2)
     zero=1.e-15

equations
     V:     div(-eps*grad(v)) = 0

boundaries
  region 1
     eps=eps0

     start(-Lx,-Ly) Load(v)=0
     line to (Lx,-Ly) to (Lx,Ly) to (-LX,Ly) to close

     start "Plate1" (-delx/2,-d/2)        value(v)=0
     line to (delx/2,-d/2) to (delx/2,-d/2-ddy) to(-delx/2,-d/2-ddy)
        to close

     start "Plate2" (-delx/2,d/2+ddy)        value(v)=1
     line to (delx/2,d/2+ddy) to (delx/2,d/2) to(-delx/2,d/2)
        to close

  region 2 { Dielectric }
     eps = 7.0*eps0
     start(-delx/2,-d/2)
     line to (delx/2,-d/2) to (delx/2,d/2) to(-delx/2,d/2)
        to close

  MONITORS
```

```
  contour(v)

 PLOTS
  ! Contour plots
  contour(v) as "Potential"
  contour(v) contours=50 as "More Contours"
  contour(v) contours=10 fixed range=(0.4,0.6)  as "Fixed Range"
  contour(v) levels=0, 0.1, 0.3, 0.5, 0.7, 0.9 as "Selected Levels"
  contour(v) zoom(-Lx/2,-Ly/2,Lx,Ly) as "Zoomed Contour"
  contour(v) on region 2 as "Region 2 Contour"
  contour(magnitude(grad(v))) log as "Field (Log divisions)"
       integrate
       report integral(magnitude(grad(v))) as "Integral Report"
  contour(magnitude(grad(v))) as "Field (NO Log divisions)"

  ! Surface Plots
  surface(magnitude(grad(v))) log as "Field (Log divisions)"
       integrate
       report integral(magnitude(grad(v))) as "Integral Report"
  surface(v) as "Surface(V)"
  surface(v) gray as "Surface(V) Gray"
  surface(v) gray mesh points=20 as "Surface(V) Gray Mesh"

  ! Vector plots
  vector(dx(v),dy(v)) zoom(-Lx/2,-Ly/2,Lx,Ly) as " Zoomed Field Vectors"

  ! Elevations
  elevation(v, dy(v)*d) from (0,-Ly) to (0,Ly) points=1000 as "1000 Point Elevation" integrate
  elevation(normal(grad(v))) on "Plate1" as "Elevation Plot on Boundary "    integrate
  elevation(magnitude(grad(v))) from (0,-0.9*Ly) to (0,0.9*Ly) log as "LOG Field"

  ! Grid plots
  grid(x,y) paintmaterials as "Mesh Plot"
  grid(x,y) paintmaterials nolines as "Materials Plot"

 end
```

6.2.39.4 print_test

```
{ PRINT_TEST.PDE

    This sample demonstrates the use of PRINT 205 selectors in PLOT 197 output.
    It creates eight output files in the same folder as the script.

    Note:  "PRINT" is synonymous with "EXPORT", and the two terms can
    be used interchangeably.

}


title "Simple Heatflow"

select
    contourgrid=50

Variables
    Temp                        { Identify "Temp" as the system variable }

definitions
    K = 1                       { declare and define the conductivity }
    source = 4                  { declare and define the source }
    Texact = 1-x^2-y^2          { for convenience, define the exact solution }

initial values
    Temp = 0                    { unimportant in linear steady-state problems }

equations
    Temp:   div(K*grad(Temp)) + source = 0   { define the heatflow equation }

boundaries                      { define the problem domain }
    Region 1                    { ... only one region }
        start "BDRY" (-1,-1)    { specify the starting point }
        value(Temp)=Texact      { specify Dirichlet boundary at exact solution }
        line to (1,-1)          { walk the boundary }
             to (1,1)
             to (-1,1)
             to close           { bring boundary back to starting point }
```

```
monitors
    contour(Temp)               { show the Temperature during solution }

plots                           { write these hardcopy files at completion }
    contour(Temp)      print   as "Contour Print"
    contour(Temp)      print(20)   as "Contour Print(20)"
    contour(Temp)      printonly  as "contour Printonly"

    contour(Temp)      export traces   as "Trace Print"

    vector(-dx(Temp),-dy(Temp)) as "Heat Flow" print

    elevation(temp) from (-1,-1) to (1,1) print as "E-print"
    elevation(temp) from (-1,-1) to (1,1) print(300) as "E-print(300)"
    elevation(temp) on "BDRY" print as "Bdry-print"

end
```

6.2.40 regional_variables

6.2.40.1 regional_variables

```
{ REGIONAL_VARIABLES.PDE

    This example demonstrates the use of variables absent in selected regions.

    The problem is a modification of LOWVISC.PDE 324, in which the bottom half of the channel
    has been filled with a solid.

    The fluid equations are declared INACTIVE 97 in the solid region, but a temperature
    equation has been added that is active everywhere.

    The bottom of the solid is held at temperature = 0, while the fluid has an incoming
    temperature of 1.

    We solve the equations in sequence: first the fluid equations, then the temperature.

}

title 'Variables inactive in regions'

variables
    u(0.1)
    v(0.01)
    p(0.1)
    temp(0.1)

definitions
   Lx = 5        Ly = 1.5
   Gx = 0        Gy = 0
   u0 = 0                        { default initial u-velocity }
   p0 = 0                        { default initial pressure }
   pin=2                         { inlet pressure }
   speed2 = u^2+v^2
   speed = sqrt(speed2)
   dens = 1
   visc = 0.04
   vxx = (p0/(2*visc*(2*Lx)))*y^2*(Ly-y)^2      { open-channel x-velocity }
   k = 0.1                         { default thermal conductivity }

   rball=0.5
   cut = 0.1        { bevel the corners of the obstruction }

   penalty = 100*visc/rball^2
   Re = globalmax(speed)*(Ly/2)/(visc/dens)

initial values
   u = u0   v = 0  p = p0

equations
   u:  visc*div(grad(u)) - dx(p) = dens*(u*dx(u) + v*dy(u))
   v:  visc*div(grad(v)) - dy(p) = dens*(u*dx(v) + v*dy(v))
   p:  div(grad(p)) = penalty*(dx(u)+dy(v))
then
   temp:  div(k*grad(temp)) - u*dx(temp) - v*dy(temp) = 0
```

```
Boundaries
   { bound the entire region, placing temperature boundary conditions }
   region 1
     INACTIVE (u,v,p)       { Inactivate the fluid in this region }
     start(-Lx,-Ly)
        value(temp)=0       line to (Lx,-Ly)
        natural(temp)=0     line to (Lx,0)
        value(temp)=1       line to (Lx,Ly)   { inlet fluid temp = 1 }
        natural(temp)=0     line to (-Lx,Ly)
        natural(temp)=-k*dx(temp)   line to close { outlet diffusive temperature flux }

   { overlay the fluid region onto the total domain, including obstruction,
        and place fluid boundary conditions }
   region 2
      u0 = 0.5*vxx  P0=pin*x/(2*Lx)     { initial values in fluid region }
      K = 0.01                          { fluid thermal conductivity }
      start(-Lx,0)
        value(u)=0  value(v) = 0
      line to (Lx/2-rball,0)
           to (Lx/2-rball,rball) bevel(cut)
           to (Lx/2+rball,rball) bevel(cut)
           to (Lx/2+rball,0)
           to (Lx,0)
        load(u) = 0  value(v) = 0  value(p) = pin
      line to (Lx,Ly)
        value(u) = 0  value(v) = 0  load(p) = 0
      line to (-Lx,Ly)
        load(u) = 0 value(v) = 0  value(p) = 0
      line to close

monitors
   contour(speed)
   contour(u)  report(Re)
   contour(v)  report(Re)
   contour(p)  as "Pressure" painted
   contour(temp)

plots
   contour(u)  report(Re)
   contour(v)  report(Re)
   contour(p)  as "Pressure" painted
   contour(temp)
   contour(speed) painted  report(Re)
   vector(u,v) as "flow"   report(Re)
   contour(dx(u)+dy(v)) as "Continuity Error"

end
```

6.2.41 sequenced_equations

6.2.41.1 theneq+time

```
{ THENEQ+TIME.PDE

    This example demonstrates the use of sequenced equations 175 in time-dependent problems.

    The variable U is given a source consistent with the desired solution of
        U=A-(x^2+y^2)
    The variable V has a source equal to -U.  The analytic solution to this equation is
        V = A*(x^2+y^2)/4 - (x^4+y^4)/12
    The variable V therefore depends strongly on U, but U is unaffected by V.

    In this case, we can separate the equations and solve for V in a THEN clause.

}
title 'Sequenced equations in time-dependent systems'

select ngrid=40

variables
    u(0.01),v(0.01)

definitions
    k = 1
    a=2
    ! analytic solutions
    u0 = (a-x^2-y^2)
```

```
    v0 = (a*(x^2+y^2)/4-(x^4+y^4)/12)

equations
    u: div(K*grad(u)) + 4 = dt(u)
then
    v: div(K*grad(v)) - u = dt(v)

boundaries
    Region 1
    start(-1,-1)
      ! ramp the boundary values, so that the initial BV's are consistent with the initial
interior values.
      value(u)=u0*Uramp(t, t-10)
      value(v)=v0*Uramp(t, t-10)
    line to (1,-1) to (1,1) to (-1,1) to close

time 0 to 100

plots
    for cycle=10
      contour(u)  paint
      surface(u)
      contour(v)  paint
      surface(v)
      elevation(u,div(K*grad(v))) from(-1,0) to (1,0)
      history(u,v) at (0,0)

end
```

6.2.41.2 theneq

```
{ THENEQ.PDE

    This example demonstrates the use of sequenced equations 175 in a steady-state problem.
    The equations are not coupled, and are solved individually.

}
title 'Sequenced Equations'

select
   errlim=1e-5
   ngrid=50

Variables
   u,v,w

definitions
   k1 = 1
   k2 = 2
   k3 = 3
   u0 = 1-x^2-y^2
   v0 = 2-x^2-y^2
   w0 = 3-x^2-y^2
   su = 4*k1
   sv = 4*k2
   sw = 4*k3

equations
   u:     div(K1*grad(u)) +su = 0
then
   v:     div(K2*grad(v)) +sv = 0
then
   w:     div(K3*grad(w)) +sw = 0

boundaries
   Region 1
      start(-1,-1)
      value(u)=u0         value(v)=v0         value(w)=w0
      line to (1,-1) to (1,1) to  (-1,1) to close

   plots
      surface(u)  paint
      surface(v)  paint
      surface(w)  paint
      elevation(u,v,w,su,sv,sw) from (-1,0) to (1,0)
end
```

6.2.42 stop+restart

6.2.42.1 restart_export

```
{  RESTART_EXPORT.PDE

   This example demonstrates the restart facilities of FlexPDE.
   The problem is a copy of 2D_LAGRANGIAN_SHOCK.PDE 493,
   with TRANSFER 169 file output every 0.02 units of problem time.

   The associated script RESTART_IMPORT.PDE 520 reads one of these
   TRANSFER files to resume the computation from the time of the
   file output.

   Alternatively, the Finish Timestep item on the Stop menu could be
   used with the "Dump for Restart" 8 checkbox set to write the automatic
   TRANSFER file "restart_export.rst".  This file could also be used in
   RESTART_IMPORT.PDE 520 to resume the computation from the point
   of the interrupt.

}

TITLE "Stop and Restart Test - Export"
SELECT
  ngrid= 100
  regrid=off
  errlim=1e-4

VARIABLES
  rho(1)
  u(1)
  P(1)
  xm = move(x)

DEFINITIONS
  len = 1
  wid = 0.01
  gamma = 1.4
  { define a damping term to kill unwanted oscillations }
  eps =  0.001

  v = 0
  rho0  = 1.0 - 0.875*uramp(x-0.49, x-0.51)
  p0 = 1.0 - 0.9*uramp(x-0.49, x-0.51)

INITIAL VALUES
  rho = rho0
  u = 0
  P = p0

EULERIAN EQUATIONS
  { Equations are stated as appropriate to the Eulerian (lab) frame.
    FlexPDE will convert to Lagrangian form for moving mesh.
    Since the equation is really in x only, we add dyy(.) terms with natural(.)=0
    on the sidewalls to impose uniformity across the fictitious y coordinate }
  rho: dt(rho) + u*dx(rho) + rho*dx(u)  = dyy(rho) + eps*dxx(rho)
  u:   dt(u) + u*dx(u) + dx(P)/rho  = dyy(u) + eps*dxx(u)
  P:   dt(P) + u*dx(P) + gamma*P*dx(u)  = dyy(P) + eps*dxx(P)
  xm:  dt(xm) = u

BOUNDARIES
   REGION 1
    { we must impose the same equivalence dt(xm)=u on the side boundaries
    as in the body equations: }
    START(0,0)
        natural(u)=0
        dt(xm)=u
    line to (len,0)
        value(xm)=len
        value(u)=0
    line to (len,wid)
        dt(xm)=u
        natural(u)=0
    line to (0,wid)
        value(xm)=0
        value(u)=0
    line to close
```

```
TIME 0 TO 0.1

MONITORS
  for cycle=5
    grid(x,10*y)
    elevation(rho) from(0,wid/2) to (len,wid/2) range (0,1)
    elevation(u) from(0,wid/2) to (len,wid/2) range (0,1)
    elevation(P) from(0,wid/2) to (len,wid/2) range (0,1)

PLOTS
  for t=0 by 0.02 to endtime
    grid(x,10*y)
    elevation(rho) from(0,wid/2) to (len,wid/2) range (0,1)
    elevation(u) from(0,wid/2) to (len,wid/2) range (0,1)
    elevation(P) from(0,wid/2) to (len,wid/2) range (0,1)

    !>>>>> HERE IS THE RESTART DUMP COMMAND:
    transfer(rho,u,p)

END
```

6.2.42.2 restart_import

```
{  RESTART_IMPORT.PDE

   This example reads the TRANSFER 169 file created by RESTART_EXPORT.PDE 519
   and resumes execution at the exported time.

}

TITLE "Sod's Shock Tube Problem - restart"
SELECT
  ngrid= 100
  regrid=off
  errlim=1e-4

VARIABLES
  rho(1)
  u(1)
  P(1)
  xm = move(x)

DEFINITIONS
  len = 1
  wid = 0.01
  gamma = 1.4
  { define a damping term to kill unwanted oscillations }
  eps =  0.001  ! 3e-4

  { Read in the file exported by restart_export.pde.
    Use the imported mesh and problem time. }
  transfermeshtime( 'restart_export_01_6.dat', rho0,u0,p0)

INITIAL VALUES
  rho = rho0
  u = u0
  P = p0

EULERIAN EQUATIONS
  { equations are stated as appropriate to the Eulerian (lab) frame.
    FlexPDE will convert to Lagrangian form for moving mesh
    Since the equation is really in x only, we add dyy(.) terms with natural(.)=0
    on the sidewalls to impose uniformity across the fictitious y coordinate }
  rho:   dt(rho) + u*dx(rho) + rho*dx(u)  = dyy(rho) + eps*dxx(rho)
  u:   dt(u) + u*dx(u) + dx(P)/rho  = dyy(u) + eps*dxx(u)
  P:   dt(P) + u*dx(P) + gamma*P*dx(u)  = dyy(P) + eps*dxx(P)
  xm:  dt(xm) = u

BOUNDARIES
  REGION 1
    { we must impose the same equivalence dt(xm)=u on the side boundaries
    as in the body equations: }
    START(0,0)    natural(u)=0  dt(xm)=u line to (len,0)
    value(xm)=len     value(u)=0   line to (len,wid)
    dt(xm)=u   natural(u)=0        line to (0,wid)
    value(xm)=0      value(u)=0   line to close
```

```
TIME 0 TO 0.375

MONITORS
  for cycle=5
    grid(x,10*y)
    elevation(rho) from(0,wid/2) to (len,wid/2) range (0,1)
    elevation(u) from(0,wid/2) to (len,wid/2) range (0,1)
    elevation(P) from(0,wid/2) to (len,wid/2) range (0,1)

PLOTS
  for t=0 by 0.02 to 0.143, 0.16 by 0.02 to 0.375
    grid(x,10*y)
    elevation(rho) from(0,wid/2) to (len,wid/2) range (0,1)
    elevation(u) from(0,wid/2) to (len,wid/2) range (0,1)
    elevation(P) from(0,wid/2) to (len,wid/2) range (0,1)

END
```

6.2.43 variable_arrays

6.2.43.1 array_variables

```
{ ARRAY_VARIABLES.PDE

    This example demonstrates the use of ARRAY VARIABLES 156 .
    A set of heat equations is solved as a demonstration.

}

title 'ARRAY Variable test'

variables
    U=array[10]                 { an array of field variables }

global variables
    g(threshold=0.1) = array[10] { and an array of global variables }

definitions
    u0 = 1-x^2-y^2
    s = array(1,2,3,4,5,6,7,8,9,10) { each equation has a different source }

equations
    repeat i=1 to 10
        U[i]:   del2(u[i]) +s[i] = 0
        g[i]:   dt(g[i]) = i-g[i]
    endrepeat

boundaries
    Region 1
        start(-1,-1)
            repeat i=1 to 10
                value(u[i])=u0
            endrepeat
        line to (1,-1) to (1,1) to (-1,1) finish

time 0 to 10

plots
    for cycle=10
     contour(u_1)
     repeat i=1 to 10
        contour(u[i])   as "U_"+$i
     endrepeat
     history(g)
     history(u)  at  (0,0) (1/4,1/4)(1/2,1/2)(3/4,3/4)
     vtk(u,g)
     cdf(u,g)
     table(u,g)
     transfer(u,g)

end
```

6.2.44 vector_variables

6.2.44.1 vector+time

```
{ VECTOR+TIME.PDE

  This example demonstrates the use of Vector variables 157 in time-dependent problems.

  A vector variable is controlled by a heat equation.  The X and Y components are given
  source terms consistent with an arbitrarily chosen final result.

  This problem is not intended to represent any real application, but is constructed
  merely to demonstrate the use of some features of vector variable support
  in FlexPDE.

}

title 'Vector transient heatflow'

Variables
    { declare a vector variable with components Ux and Uy.
      Each component is expected to have a variation large compared to 0.01 }
    U(0.01) = vector(Ux,Uy)
    { declare a scalar field variable to validate the y-component }
    V(0.01)

definitions
     { Define the expected solutions for the components. }
     u0 = (1-x^2-y^2)
     u1 = (1+y+x^3)
     { Define source terms that will result in the programmed solutions }
     s = vector(4,-6*x)

equations
     U: del2(U) +s = dt(U)
     V: del2(V) +ycomp(s) = dt(V)

boundaries
     Region 1
        start 'outer' (-1,-1)
            { Apply a time ramp to the value boundary conditions, so that the
                initial boundary values agree with the initial field values. }
            value(U)=vector(u0,u1)*uramp(t, t-1)
            value(V)=u1*uramp(t, t-1)
        line to (1,-1) to (1,1) to (-1,1) to close

time 0 to 5

plots
     for cycle=10
       { various uses of vector variables in plot statements: }
       contour(Ux, u0)
       contour(Uy, u1)
       contour(V, u1)
       contour(Ux, Uy)
       contour(U)
       vector(U)
       elevation(U, V) from(-1,0) to (1,0)
       history(U, V) at(0,0)

       elevation(u1, Uy, V) on 'outer'
       elevation(u0, Ux) on 'outer'
       elevation(normal(grad(Ux)), normal(grad(u0))) on 'outer'
       elevation(normal(grad(V)), normal(grad(Uy)), normal(grad(u1))) on 'outer'

end
```

6.2.44.2 vector_lowvisc

```
{  VECTOR_LOWVISC.PDE

  This example is an implementation of LOWVISC.PDE 324 using vector variables 157.

}

title 'Viscous flow in 2D channel, Re > 40'
```

```
select errlim = 0.005

variables
   vel(0.01) = vector(u,v)
   p(1)

definitions
   Lx = 5        Ly = 1.5
   Gx = 0        Gy = 0
   p0 = 2
   speed2 = u^2+v^2
   speed = sqrt(speed2)
   dens = 1
   visc = 0.04
   vxx = (p0/(2*visc*(2*Lx)))*(Ly-y)^2      { open-channel x-velocity }

   rball=0.25
   cut = 0.05        { bevel the corners of the obstruction }

   penalty = 100*visc/rball^2
   Re = globalmax(speed)*(Ly/2)/(visc/dens)

initial values
   vel = vector(0.5*vxx ,0)
   p = p0*x/(2*Lx)

equations
   vel:  visc*div(grad(vel)) - grad(p) = dens*dot(vel,grad(vel))
   p:  div(grad(p)) = penalty*div(vel)

Boundaries
   region 1
      start(-Lx,0)
      load(u) = 0   value(v) = 0   load(p) = 0
        line to (Lx/2-rball,0)

      value(vel)=vector(0,0)  load(p)= 0
        line to (Lx/2-rball,rball) bevel(cut)
             to (Lx/2+rball,rball) bevel(cut)
             to (Lx/2+rball,0)

      load(u) = 0  value(v) = 0  load(p) = 0
        line to (Lx,0)

      load(u) = 0  value(v) = 0  value(p) = p0
        line to (Lx,Ly)

      value(vel)=vector(0,0)  load(p) = 0
        line to (-Lx,Ly)

      load(u) = 0  value(v) = 0  value(p) = 0
        line to close

monitors
   contour(speed)

plots
   contour(u)  report(Re)
   contour(v)  report(Re)
   contour(speed) painted  report(Re)
   vector(u,v) as "flow"  report(Re)
   contour(p) as "Pressure" painted
   contour(dx(u)+dy(v)) as "Continuity Error"
   elevation(u) from (-Lx,0) to (-Lx,Ly)
   elevation(u) from (0,0) to (0,Ly)
   elevation(u) from (Lx/2,0) to (Lx/2,Ly)
   elevation(u) from (Lx,0) to (Lx,Ly)

end
```

6.2.44.3 vector_variables

```
{ VECTOR_VARIABLES.PDE

    This example demonstrates the use of vector-valued variables 157.
    The equations are not intended to represent any real application,
    but merely to show some vector constructs.
```

```
}

title 'Vector Variables'

variables
    U = vector(Ux,Uy)   { declares component variables Ux and Uy }
    V                   { a scalar variable to validate Uy }

definitions
    u0 = 1-x^2-y^2
    u1 = 1+y+x^3
    s = vector(4,-6*x)

equations
    U: div(grad(U)) +s = 0
    V: del2(v) +ycomp(s) = 0

boundaries
    Region 1
      start(-1,-1)
        value(U)=vector(u0,u1)
        value(v)=u1
      line to (1,-1) to (1,1) to (-1,1) finish

plots
    contour(Ux)
    contour(Uy,u1)
    contour(v,u1)
    contour(Ux,Uy)
    vector(U)
    elevation(u) from(-1,0) to (1,0)
    vtk(u,s)
    cdf(u,s)
    transfer(u,s)
    table(u,s)

end
```

Index

- " -

"Include" Files 119

- # -

 # 106, 119

- $ -

$ 128

- . -

.DBG file 3
.EIG file 3
.LOG file 3
.PDE file 3, 117
.PG6 file 3, 210

- 1 -

1D
 boundary conditions 191
 coordinates 153

- 2 -

2D
 coordinates 153

- 3 -

3D
 coordinates 153
 PLOTS 197
 Problems 69, 265

- A -

ABS function 126
Accuracy
 Controlling 45
 Threshold 155
Adaptive Mesh Refinement 217
ALE 100, 177
ALIAS 151
aliasing coordinates 153
Analytic Functions 126
Animation 284
ANTIPERIODIC Boundary Conditions 193
ARC 37, 181
ARCCOS function 126
ARCSIN function 126
ARCTAN function 126
AREA_INTEGRAL 46, 138
AREA_INTEGRATE 200
Arithmetic Operators 133
ARRAY
 Definition 159
 Size 159
 Variables 156
ARRAYS
 Using 109
AS 'string' 200
ASPECT 146
AT 211
ATAN2 function 126
AutoCAD 265
AUTOHIST 151, 211
AUTOSTAGE 147

- B -

Batch runs 212
Bessel Function 126
BESSJ function 126
BESSY function 126
Bevels 189
BINTEGRAL 46, 136
Bitmaps 210
BLACK 151, 200
BLOCK 167
BMP 200, 210
BOUNDARIES 33, 37, 180, 265
 Paths and Path Names 181
 Search 133
Boundary Conditions 215
 1D 191
 3D 78, 191
 ANTIPERIODIC 193
 CONTACT 189, 192
 Default 189
 Dirichlet 189
 Discontinuous 263
 Flux 63, 189
 JUMP 192
 LOAD 189
 NATURAL 34, 41, 61, 189, 260
 NOBC 189
 PERIODIC 193
 Point Load 190
 Point Value 190
 Remain in effect 189
 Syntax 190
 Terminating 189
 VALUE 34, 41, 189
 VELOCITY 189
Boundary Paths and Path Names 181

Boundary Search 133

- C -

Canister Example 76
CARG 134
cartesian 153
CARTESIAN1 153
CARTESIAN2 153
CARTESIAN3 153
case sensitivity 36, 118
CDF output 106, 197
CDFGRID 151
CENTER 181
CEXP 134
CHANGELIM 60, 147
Colors
- Spectral 152
- Thermal 152

Command-line
- Running without graphics 114
- Switches 113

Commas 124
Comments 121
COMPLEX
- Operators 134
- Variables 89, 156

Components
- Tensor 141
- Vector 140

conditional expressions 143
CONJ 134
CONST
- Array 159
- definitions 158
- Matrix 161

Constants 125
CONSTRAINTS 178
CONTACT 189, 192
Contact Resistance 65
CONTOUR 41
CONTOUR plot 198
CONTOURGRID 151
CONTOURS 151, 201
Controls Menu 7
coordinates
- 1D 153
- 2D 153
- 3D 153
- aliasing 153
- Cartesian 153
- CARTESIAN1 153
- CARTESIAN2 153
- CARTESIAN3 153
- CYLINDER1 153
- Cylindrical 153
- renaming 153
- scaling 282
- SPHERE1 153
- Transformation 153
- XCYLINDER 153
- YCYLINDER 153

COS function 126
COSH function 126
CRITICAL 196
CROSS product 140
CUBIC 147, 148
CURL Operator 135
Curl Theorem 62
CURVEGRID 146
Cut Planes 207
CYCLE plot interval 209
CYLINDER 83, 179
CYLINDER1 153
cylindrical 153

- D -

Decimal Numbers 125
Decoupling Variables 66
DEFINITIONS 33, 40, 158
- Array 159
- Function 163
- Matrix 161
- Point 165
- Stage 164

DEL2 - Laplacian Operator 135
DELETE 3
Dependent Variables 154
Derivative operators 135
Derivatives
- high order 174

Descriptor formatting 118
Differential Operators 135, 174
Differentiation
- Notation 36
- Suppressing 172

Dirichlet Boundary Conditions 189
Discontinuities 263
Discontinuous Variables 64
Display
- Modifiers 200
- Specifications 197

DIV - Divergence Operator 135
Divergence Theorem 62
Domain
- Description 37
- Menu 4
- Review 12

DOTproduct
- Tensor 141
- Vector 140

DROPOUT 201

DXF 265

- E -

Edit Menu 4
Edit while Run 17
Editing Problem Descriptors 10
Eigenvalue 142, 176
 Shifting 263
 Summary 58
Eigenvalues 55
Electromagnetics 270
Electrostatics 218, 270
ELEVATION 41
 Plot 198, 207
ELEVATIONGRID 151
Elliptical segments 181
EMF 201
Empty Layers in 3D 186
END 212
ENDREPEAT 144
Engineering Notation 125
EPS 201
EQUATIONS 33, 36
 and Variables 175
 Section 174
 Sequencing 175
ERF function 126
ERFC function 126
ERRLIM 45, 147, 148, 155
Error 209
 Estimates 280
 Function 126
 Tolerance 45
Eulerian 100, 177
EVAL function 132
Examples 20
 Constraints 136
 Graphics 210
 Integrals 136
 Simple 119
 Tutorial 42
Excludes 187
EXP function 126
Exponential Integral - EXPINT 126
EXPORT 106, 201
 Graphics 210
 Movie 19
Export Plot 16
expressions 143
EXTRUSION 69, 265
 Notation 70
 Section 179

- F -

FEATURE_INDUCTION 146
FEATUREPLOT 151
Features 187
File
 Extension 3, 117
 Menu 4, 6
 Name 3, 117
FILE 'string' 202
Fillets 189
FINALLY
 Equation Sequencing 175
Find 10
FINISH 181
Finite Element Mesh 39
Finite Element Methods 214
FIRSTPARTS 147, 148
FIT Function 129
FIXDT 147, 148
FIXED RANGE 202
FlexPDE
 Application 32
 Facilities 31
 Overview 30
 Script 33
Flux Boundary Conditions 63, 189
FONT 10, 151
FORMAT 106
FORMAT 'string' 202
Formatting 118
FRAME 202
FRONT 194
Function definition 163
Functions
 Analytic 126
 Non-analytic 126
 String 128
fuzzy IF 132

- G -

GAMMA function 126
Global Graphics Controls 150
GLOBAL VARIABLES 157
GLOBALMAX function 126
GLOBALMAX_X function 126
GLOBALMAX_Y function 126
GLOBALMAX_Z function 126
GLOBALMIN function 126
GLOBALMIN_X function 126
GLOBALMIN_Y function 126
GLOBALMIN_Z function 126
GRAD - Gradient Operator 135

Graphic Display modifiers 200
Graphical Output 41
Graphics
  Export 210
  Global Controls 150
  Running without 114
Graphics Examples 210
GRAY 151, 202
Grid Control Features 187
GRID plot 198
GRIDARC 146
GRIDLIMIT 146
GUI
  Running without 114
Guidelines for Problem Setup 34, 35

- H -

HALT 196
Hardcopy 210
HARDMONITOR 151
Heat Equation 62
Help Menu 4
HISTORIES
  Section 211
  Staged Problems 211
  Windowing 211
HYSTERESIS 147, 148

- I -

ICCG 147, 148
IF 143
IMAG 134
import 265
Importing Data from Other Applications 108
INACTIVE 97
include files 119
Inconsistent Initial Conditions 54
INITGRIDLIMIT 146
Initial Conditions 54, 263
INITIAL VALUES 33, 60, 174
input 117
Instantaneous Switching 54
INTEGRAL 46, 138, 139
Integral Rules 216
Integrals 46, 50
  3D 84
  Area 138
  Constraints 178
  Line 136
  Operators 136
  Surface 137, 138
  Time 136
  Volume 138, 139
INTEGRATE 202
Integration by Parts 62, 216
ITERATE 147, 148

- J -

JUMP 64, 67, 192

- L -

Lagrange 100, 177
LAMBDA 142
LAYER 70, 265
  Extrusion 71, 179
  Shaped Interfaces 81
LEVELS 202
License 21, 22, 23, 24, 25, 26
LIMITED REGIONS 74, 186
LINE 37, 181
LINE_INTEGRAL 46, 136
LINE_INTEGRATE 203
LINUPDATE 147, 148
Literal Strings 125
LN function 126
LOAD 189
LOG 203
LOG10 function 126
logarithm 126
LOGLIMIT 151
LUMP function 130

- M -

MacOS 2
Magnetic Field 62
Magnetic Vector Potential 270
Magnetostatics 270
MAGNITUDE of vector 140
Material
  Parameters 40, 72, 184
  Properties 40, 72, 184
MATRICES
  Using 109
MATRIX
  Definition 161
  Size 161
MAX function 126
Maximize 15
Maxwell's Equations 270
Menu 4
MERGE 146, 203
MERGEDIST 146
MESH 203
Mesh Generation
  Controls 146, 187
  MESH_DENSITY 173

Mesh Generation
MESH_SPACING 39, 173
Mesh Refinement
Controls 103
FRONT 194
RESOLVE 195
MESH_DENSITY 173
MESH_SPACING 173
Metafile 200
MIN function 126
MOD function 126
Modal Analysis 55, 176, 263
MODE_SUMMARY plot 198
MODES 147, 148
Modify Menu 4
MONITORS 41, 197
Steady State 209
Time Selection 209
MOVE 156
Movie 19
Making 284
Moving Meshes 100, 156, 177
Balancing 101
Example 102
Multiple processors 112, 150

- N -

Named Paths 181
NATURAL 189
Boundary Condition 34, 41, 61, 62, 216, 260
Network License Manager 25
NEWTON 147, 148, 261
Newton-Raphson Iteration 60
Next 19
NGRID 146
NOBC 189
NODE POINT (Node Placement) 187
NODELIMIT 146
NOHEADER 203
NOLINES 203
NOMERGE 203
NOMINMAX 151, 203
Non-Analytic Functions 126
NONLINEAR 147, 148
Coefficients and Equations 58
Problems 60, 110, 261
NONSYMMETRIC 147, 148
NORM 203
NORMAL component 140
NOTAGS 151, 204
Notation 36
NOTIFY_DONE 147, 149
NOTIPS 151, 204
NRMATRIX 149
NRMINSTEP 147, 149
NRSLOPE 147, 149
Numbering and Naming Regions 188
Numeric
Constants 125
Range 125
Reports 208

- O -

ON
Equation 197, 204, 206
GRID 204
LAYER 87, 204, 206
REGION 87, 204, 206
Selectors 206
SURFACE 87, 197, 204, 206
One-Dimensional Problems 68
Operators
Arithmetic 133
COMPLEX 134
Differential 135
Integral 136
Relational 139
Smoothing 277
String 140
Tensor 141
Vector 140
ORDER 147, 149
Ordering Regions 188
oscillation 263
OVERSHOOT 147, 149

- P -

PAINTED 152, 204
PAINTGRID 152
PAINTMATERIALS 152, 204
PAINTREGIONS 152, 204
Parameter Studies 48
Parameterized Definitions 163
Parameters
Material 40, 184
redefining 158
Regional 184, 185
PASSIVE Modifier 172
Paths and Path Names 181
PENWIDTH 152, 204
PERIODIC Boundary Conditions 193
PI 142
PLANE 83, 179
Plot
Cutplanes 207
Domain 206
Elevation 207
Export 16, 200, 201, 204, 205
Fixed Range 202

Plot
Integrate 202
Labels 16
Maximize 15
Menu 4
Modifiers 200
On Grid 208
POINTS 204
Print 15
Range 200, 205
Restore 15
Rotate 16
Scaling 202
Time Selection 209
View Saved Files 19
Windows 15
Zoom 206
PLOTINTEGRATE 152
PLOTS 33, 41, 197
Cutplane 75, 87
Display 197
Print 205
Printonly 205
Time Selection 209
Viewpoint 205
PNG 204, 210
POINT 181
Definitions 165
LOAD Boundary Conditions 190
Movable 165
VALUE Boundary Conditions 190
POINTS 204
Post-processing 105
PostScript 200
Potential 270
PPM 205, 210
PRECONDITION 147, 149
PREFER_SPEED 147, 149, 261
PREFER_STABILITY 147, 149, 261
Preparing a Descriptor File 117
Previous 19
PRINT 205
Print Plot 15
Print Script 10
PRINTMERGE 152
PRINTONLY 205
Problem Descriptor Structure 117
Problem Setup Guidelines 34, 35
Problem Solving Environment 30
Projection 265
Properties
Material 40, 184
Regional 184
Proxy Server Settings 4
pulse function 128

- Q -

QUADRATIC 147, 149
questions 114
quoted strings 125

- R -

R 142
RADIUS 142, 181
RAMP function 128, 130
RANDOM function 126
RANDOM_SEED 149
RANGE 205
REAL 134
REGION 37, 265
Regional Parameter Values 184
Regions
1D 184
3D 185
Excluded 187
Numbering and naming 188
Ordering 188
Overlaying 183
Registering 21, 22, 23, 24, 25, 26
REGRID 147
REINITIALIZE 147, 149
Relational Operators 139
REMATRIX 149, 261
renaming coordinates 153
REPEAT 144
Repeated Text 144
REPORT 47, 208
Reporting numbers 208
Reserved Words and Symbols 121
RESOLVE 195
Restart 19
Restore Plot 15
ROTATE 181
Rotate Plot 16
Run Menu 4

- S -

SAVE function 131
SCALAR VARIABLES 157
Script 117
Interpretation 45
script editing module 31
Scripting Language 30
section names 117
SELECT 33, 145
Semicolons 124
SENSITIVITY 147

Separators 124
Shaped Layer Interfaces 81
SIGN function 126
SIN function 126
SINH function 126
SINTEGRAL 137, 138
SIZEOF 159
SMOOTH 167
SMOOTHINIT 147
Spaces 124
SPECTRAL_COLORS 152
SPHERE 83, 179
SPHERE1 153
SPLINE 167, 181
SPLINETABLE 166
SQRT function 126
STAGE 142
STAGED
Definitions 164
Geometry 164
Parameters 48
STAGEGRID 147
STAGES 48, 147, 149, 164
START 37, 181
Status Panel 14
step function 128
Stop Menu 4, 8
String
Functions 128
Literal 125
Operators 140
SUBSPACE 147, 150
SUM function 131
SUMMARY 47, 58
SUMMARY plot 198
SURF_INTEGRAL 137, 138
SURFACE 41, 70, 265
Extrusion 179
Surface Generators 179
Surface Integrals 84
SURFACE plot 199
Surface-Generating Functions 83
SURFACEGRID 152
SWAGE function 132
Switches
Command-line 113
symbolic equation analyzer 31

- T -

TABLE
File format 168
Input function 166
Modifiers 167
Output 106, 199
usage 108
Tabledef Input 167
TABULATE definitions 169
TAN function 126
TANGENTIAL component 140
TANH function 126
TE and TM Modes 247
TECPLOT output 106, 199
Tensor Operators 141
TERRLIM 147, 150
Text Strings 125
TEXTSIZE 152
THEN
Equation Sequencing 175
THERMAL_COLORS 152
THETA 142
Threads 112, 150
Three-Dimensional Problems 69, 265
THRESHOLD 155
Time
Critical 196
Halt 196
Integration 217
Range 196
Time Dependence 52
Things to avoid 54
TIME_INTEGRAL 136
TIMEMAX function 126
TIMEMAX_T function 126
TIMEMIN function 126
TIMEMIN_T function 126
TINTEGRAL 136
Title 33, 145
TNORM 147, 150
Tool Bar 10
TRANSFER
Exporting Data 106
File format 170
Output 199
Post-processing 105
Statement 169
TRANSFERMESH
Output 199
Post-processing 105
Statement 169
TRANSFERMESHTIME 169
Transferring Data 106
TRANSPOSE
matrix 161
tensor 141
trigonometric functions 126

- U -

Unit Functions 128
UNIX/Linux 2

UNORMAL 140
Updates
  Bypass Auto Check 4
  Check for 4
UPFACTOR 147, 150
UPULSE function 128
UPWIND 147, 150
URAMP function 128
User Guide 30
USTEP function 128

- V -

VAL function 132
VALUE 189
  Boundary Condition 34, 41
  Initial 174
VANDENBERG 147, 150
VARIABLES 33, 36
  and Equations 175
  Array 156
  Complex 89, 156
  Dependent 154
  Global 157
  Regional 97
  Threshold 155
  Vector 94, 157
VECTOR 41
  composition 140
  Curvilinear Derivatives 95
  Operators 140
  Plot 199
  Potential 270
  Variables 94, 157
VECTORGRID 152
VELOCITY 189
  Boundary Condition 101
Version 4
  Converting to version 5 284
Version 5
  Converting from version 4 284
  Converting to version 6 285
Version 6
  Converting from version 5 285
VERSUS 211
View
  Saved Graphics Files 19
View Menu 4
VIEWPOINT 152, 205
VisIt 106
Visualization 106
VOID
  Compartments 74
  Layers 186
VOL_INTEGRAL 46, 138, 139
VOL_INTEGRATE 205
Volume Integrals 84
VTK output 106, 200
VTKLIN 200

- W -

Waveguides
  Homogeneous 246
  Non-Homogeneous 251
While the Problem Runs 14
white space 124
WINDOW 211
Window Tiling 209
Windows 95/98/ME/NT/2000/XP/Vista 2

- X -

XBOUNDARY 133
XCOMP
  Point operator 165
  vector operator 140
XCYLINDER 153
XERRLIM 147, 150
XMERGEDIST 147
XPM 205, 210
XServer
  Running without 114
XXCOMP 141
X-Y 265
XYCOMP 141
X-Z 265
XZCOMP 141

- Y -

YBOUNDARY 133
YCOMP
  Point operator 165
  vector operator 140
YCYLINDER 153
YMERGEDIST 147
YXCOMP 141
YYCOMP 141
YZCOMP 141

- Z -

ZBOUNDARY 133
ZCOMP
  Point operator 165
  vector operator 140
Z-dimension 265
ZMERGEDIST 147
ZOOM 206
ZXCOMP 141
ZYCOMP 141
ZZCOMP 141