

FlexPDE User Guide

Version 5.0
5/28/05

© 2005 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 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 and/or other countries.

This version of this and the companion manuals are current as of the initial release of version 5. **Electronic versions of this manual** together with subsequent release notices and the companion manuals in the FlexPDE documentation series 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

1. Foreword	1
2. Overview	2
2.1. What Is FlexPDE?	2
2.2. What Can FlexPDE Do?	4
2.3. How Does It Do It?	5
2.4. Who Can Use FlexPDE?	7
2.5. What Does A Script Look Like?	8
2.6. What About Boundary Conditions?	10
3. Basic Usage	11
3.1. How Do I Set Up My Problem?	11
3.2. Problem Setup Guidelines	13
3.3. Notation	15
3.4. Variables and Equations	16
3.5. Mapping the Domain	17
3.6. An Example Problem	19
3.7. Generating A Mesh	20
3.8. Defining Material Parameters	22
3.9. Setting the Boundary Conditions	24
3.10. Requesting Graphical Output	26
3.11. Putting It All Together	28
4. Some Common Variations	31
4.1. Controlling Accuracy	31
4.2. Computing Integrals	33
4.3. Reporting Numerical Results	35
4.4. Summarizing Numerical Results	36
4.5. Parameter Studies Using STAGES	37
4.6. Cylindrical Geometry	40
4.6.1. Integrals In Cylindrical Geometry	40
4.6.2. A Cylindrical Example	41
4.7. Time Dependence	44
4.7.1. Bad Things To Do In Time Dependent Problems	47
4.8. Eigenvalues and Modal Analysis	48
4.8.1. The Eigenvalue Summary	51
5. Addressing More Difficult Problems	53
5.1. Nonlinear Coefficients and Equations	54
5.1.1. Complications Associated with Nonlinear Problems	56
5.2. Natural Boundary Conditions	58
5.2.1. Some Typical Cases	59
5.2.2. An Example of a Flux Boundary Condition	61
5.3. Discontinuous Variables	63
5.3.1. Contact Resistance	63
5.3.2. Decoupling	66
5.3.3. Using JUMP in problems with many variables	67
6. Using FlexPDE in One-Dimensional Problems	69

7. Using FlexPDE in Three-Dimensional Problems	71
7.1. The Concept of Extrusion	72
7.2. Extrusion Notation in FlexPDE	74
7.3. Layering	76
7.4. Setting Material Properties by Region and Layer	78
7.5. Void Compartments	80
7.6. Limited Regions	81
7.7. Specifying Plots on Cut Planes	83
7.8. The Complete 3D Canister	84
7.9. Setting Boundary Conditions in 3D	87
7.10. Shaped Layer Interfaces	91
7.11. Integrals in Three Dimensions	95
7.12. More Advanced Plot Controls	99
8. Moving Meshes	101
8.1. Mesh Balancing	102
8.2. The Pulsating Blob	103
9. Controlling Mesh Density	105
10. Exporting Data to Other Applications	108
11. Solving Nonlinear Problems	112
12. Getting Help	115
Index	116

1. Foreword

This document attempts to introduce 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 companion volume, the FlexPDE Problem Descriptor Reference.

2. Overview

2.1. 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

$$\text{Div}(\text{grad}(u)) = 0.$$

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 effect 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.

2.2. What Can FlexPDE Do?

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

2.3. How Does It Do It?

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

- **A script editing module** with syntax highlighting provides a full text editing facility and a graphical domain preview.
- **A symbolic equation analyzer** expands defined parameters and relations, 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 module** 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 module** 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 either 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 module** accepts arbitrary algebraic functions of the solution and plots contour, surface, vector or elevation plots.

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

2.4. 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.

2.5. 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 over 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 "Sections".]

Comments can be placed anywhere in a script.

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

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.

2.6. 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")

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.

options is detailed in the FlexPDE Reference. Many will also be addressed in subsequent topics.]

3. Basic Usage

3.1. 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

3.2. 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.

3.3. 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 $dx(x(u))$ 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 for a full description of FlexPDE notation.]

3.4. 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 labelled 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.

3.5. 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; you cannot build a tilted ellipse.

- 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" menu button to preview a drawing of what you have created so far.

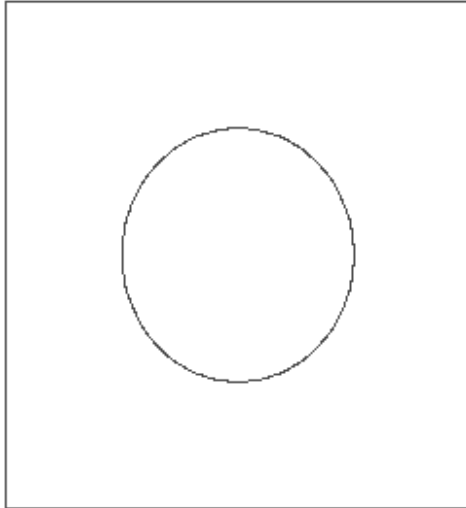
The "Save" and "Save_As" menu buttons allow you to frequently save your work, just in case.

3.6. 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
```

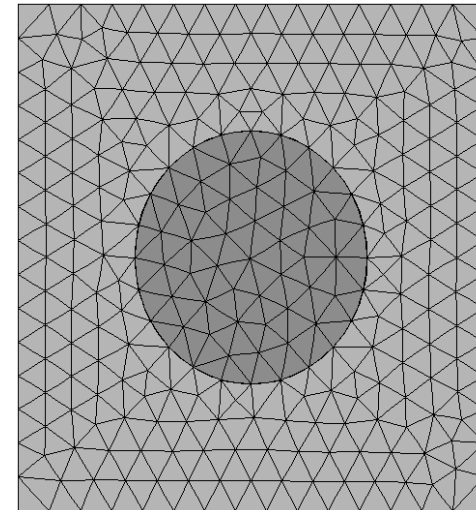
The resulting figure displayed by the "domain" button is this:



3.7. 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" below.

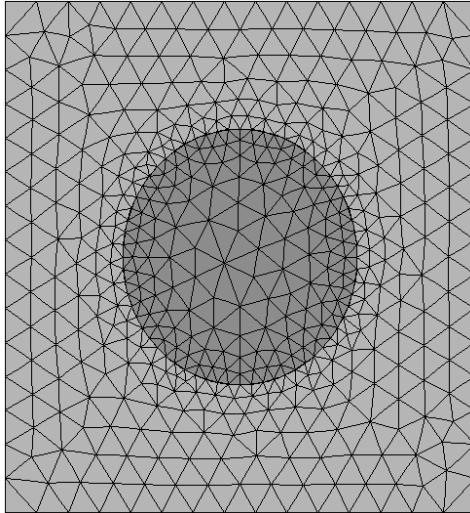
As an example, we can cause the circular boundary of region 2 to be gridded more densely by using the modifier **MESH_SPACING**:

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

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

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.

3.8. 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 two 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 each 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.

3.9. 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."]

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 determined to define 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".]

3.10. 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, $k \cdot \text{grad}(\Phi)$, 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 .PG5 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 .PG5 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".]

3.11. 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:

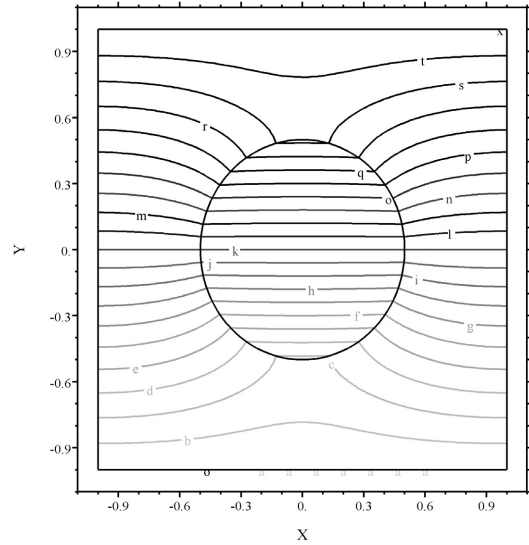
```
TITLE 'Heat flow around an Insulating blob'
VARIABLES
  Phi    { the temperature }
DEFINITIONS
  K = 1    { default conductivity }
  R = 0.5  { blob radius }
EQUATIONS
  Div(-k*grad(phi)) = 0

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:

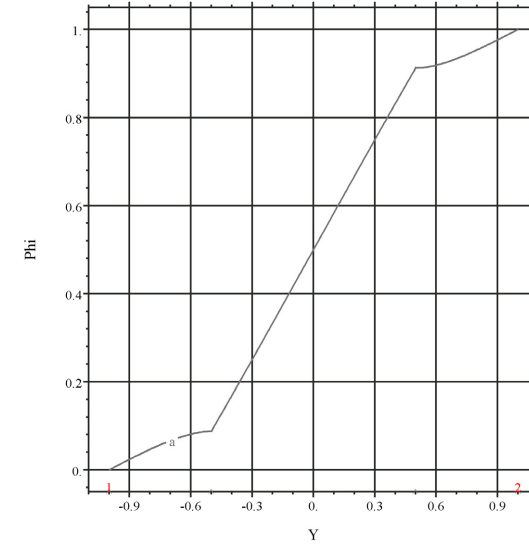
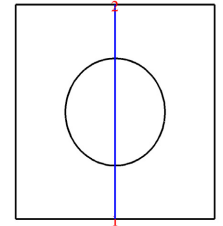
Heat flow around an Insulating blob

21:12:24 5/23/05
FlexPDE 5.0.0

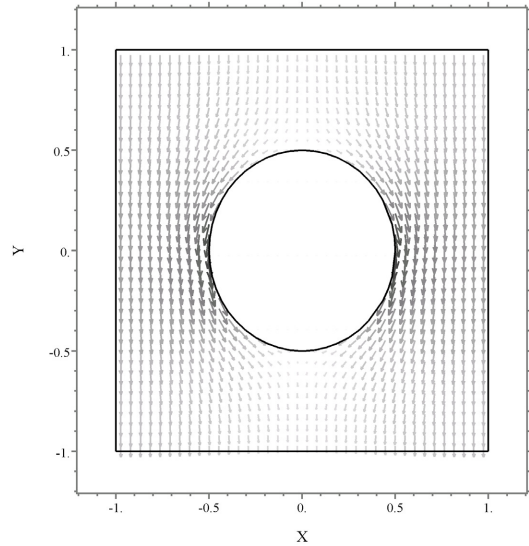
Phi	
max	1.00
u :	1.00
t :	0.95
s :	0.90
r :	0.85
q :	0.80
p :	0.75
o :	0.70
n :	0.65
m :	0.60
l :	0.55
k :	0.50
j :	0.45
i :	0.40
h :	0.35
g :	0.30
f :	0.25
e :	0.20
d :	0.15
c :	0.10
b :	0.05
a :	0.00
min	0.00

ex1: Grid#1 p2 Nodes=1201 Cells=570 RMS Err= 1.3e-4
Integral= 1.999997

Heat flow around an Insulating blob

21:12:24 5/23/05
FlexPDE 5.0.0Phi
FROM (0,-1)
to (0,1)
a: Phiex1: Grid#1 p2 Nodes=1201 Cells=570 RMS Err= 1.3e-4
Integral= 0.999998

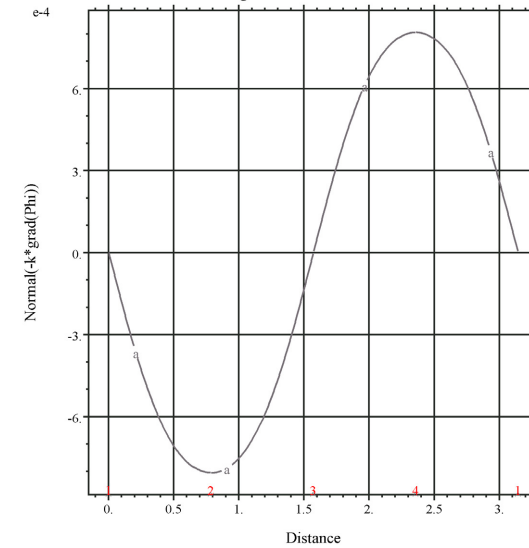
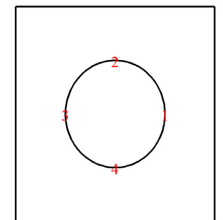
Heat flow around an Insulating blob

21:12:24 5/23/05
FlexPDE 5.0.0

-k*grad(Phi)	
0.90	
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	

ex1: Grid#1 p2 Nodes=1201 Cells=570 RMS Err= 1.3e-4

Heat flow around an Insulating blob

21:12:24 5/23/05
FlexPDE 5.0.0Normal(-k*grad(Phi))
ON ring
a: Normal(-k*grad(Phi))ex1: Grid#1 p2 Nodes=1201 Cells=570 RMS Err= 1.3e-4
Integral= 1.266992e-7

4. Some Common Variations

4.1. 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 0.001, which means that FlexPDE will refine the mesh until the estimated error in any variable (relative to the variable range) is less than 0.1% over every cell of the mesh.

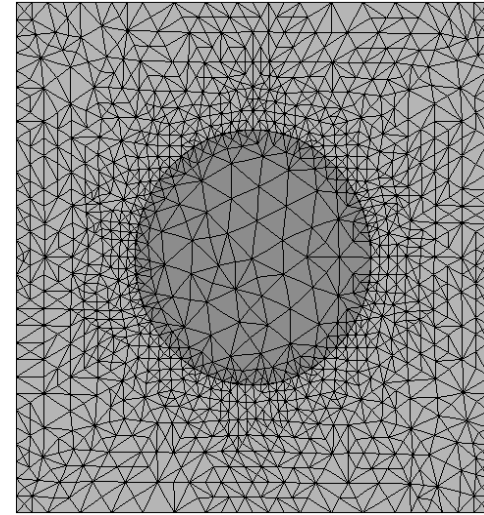
[Note: This does not mean that FlexPDE can guarantee that the solutions is accurate to 0.1% over the domain. Individual cell errors may cancel or accumulate in ways that are hard to predict.]

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.]

4.2. 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 Reference section "Elements | Operators | Integral Operators".]

4.3. 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 **REPORT**s 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
```

4.4. 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 **REPORT**s.

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
```

4.5. 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 run 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

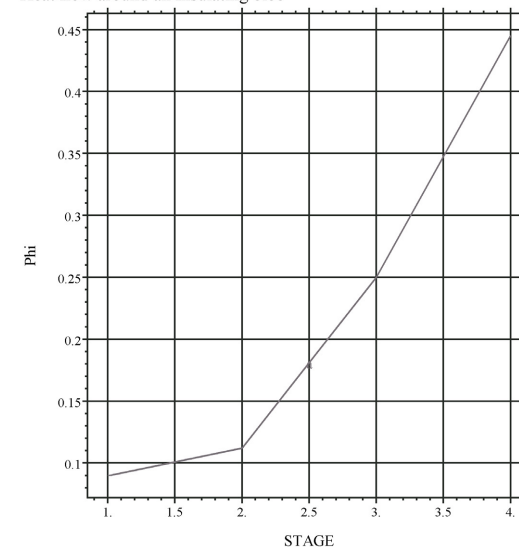
Kins = STAGED(0.01, 0.1, 1, 10)

{ 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:

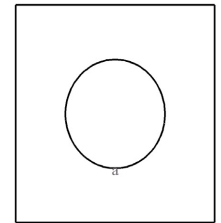
Heat flow around an Insulating blob



ex1s: Grid#1 p2 Nodes=1201 Cells=570 RMS Err= 1.6e-4
Stage 4

22:21:58 5/23/05
FlexPDE 5.0.0

HISTORY
1: Phi



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 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.

4.6. 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.
- **YCYLINDER** places the rotation axis of the cylinder, the Z coordinate, along the ordinate (or "y" axis) of the plot, with axial extension along the abscissa.

Either form may optionally be followed by a parenthesized renaming of the coordinates. In either case, the names are (abscissa, ordinate). The defaults are

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

4.6.1. 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 $2\pi R$.

4.6.2. 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:

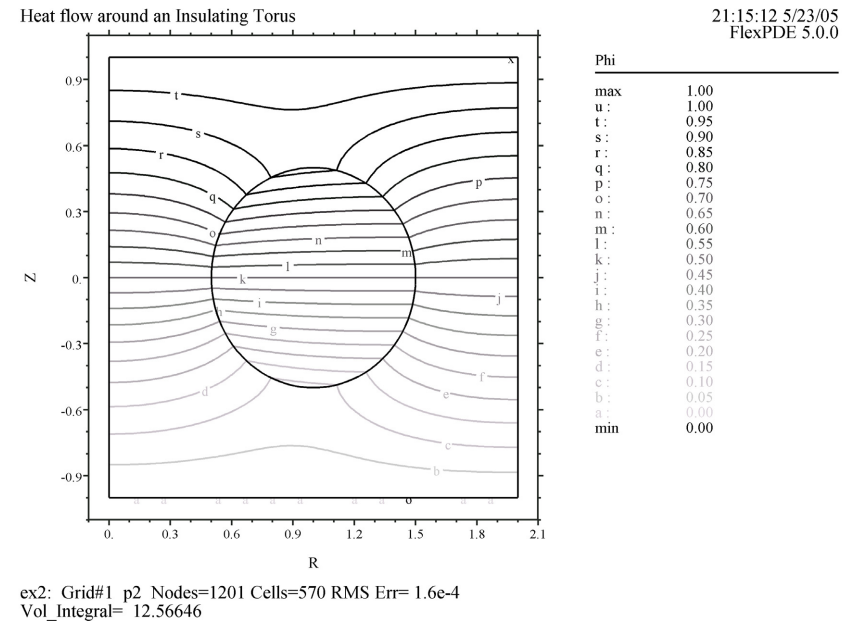
- We must offset the coordinates, so the left boundary becomes $R=0$.
- Since we want the rotation axis in the Y-direction, we must use YCYLINDER coordinates.
- Since 'R' is now a coordinate name, we must rename the 'R' used for the blob radius.

The full script, converted to cylindrical coordinates is then:

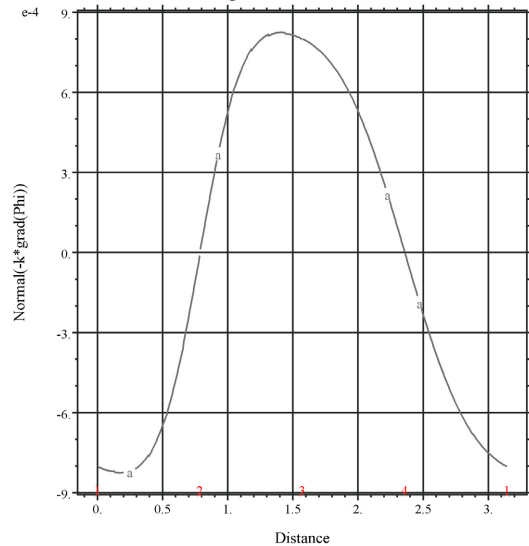
```
TITLE 'Heat flow around an Insulating Torus'
COORDINATES
  YCYLINDER
VARIABLES
  Phi { the temperature }
DEFINITIONS
  K = 1 { default conductivity }
  Rad = 0.5 { blob radius (renamed) }
EQUATIONS
  Div(-k*grad(phi)) = 0
BOUNDARIES
  REGION 1 'box'
    START(0,-1)
    VALUE(Phi)=0 LINE TO (2,-1)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



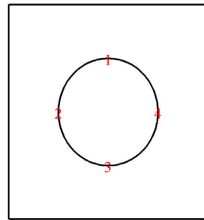
Heat flow around an Insulating Torus



ex2: Grid#1 p2 Nodes=1201 Cells=570 RMS Err= 1.6e-4
Surf_Integral= 1.813059e-5

21:15:12 5/23/05
FlexPDE 5.0.0

Normal(-k*grad(Phi))
ON ring
a: Normal(-k*grad(Phi))



4.7. 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. Second-order equations 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:

$$\text{Div}(k*\text{grad}(\Phi)) = c*dt(\Phi)$$

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:

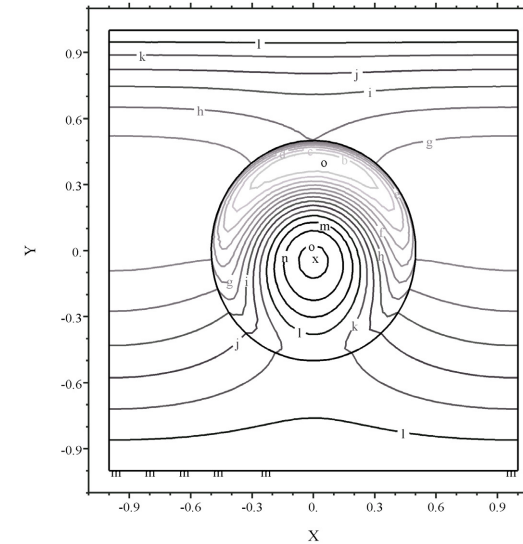
```

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:

Transient Heat flow around an Insulating blob

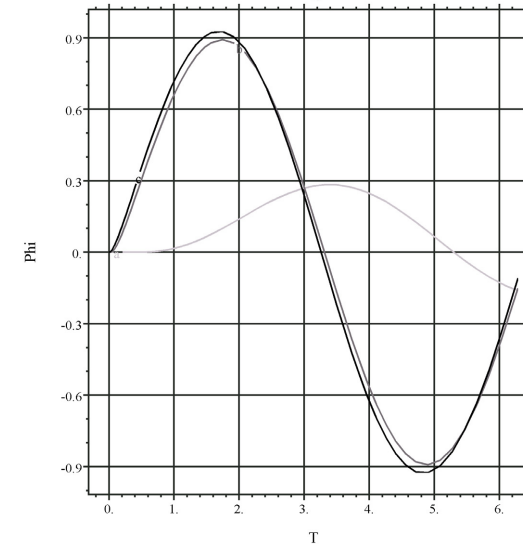


21:15:58 5/23/05
FlexPDE 5.0.0

Phi	
max	0.07
o :	0.06
n :	0.03
m :	0.00
l :	-0.03
k :	-0.06
j :	-0.09
i :	-0.12
h :	-0.15
g :	-0.18
f :	-0.21
e :	-0.24
d :	-0.27
c :	-0.30
b :	-0.33
a :	-0.36
min	-0.36

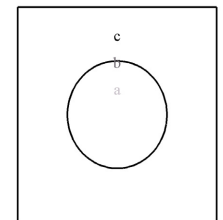
ex3: Cycle=92 Time= 6.2832 dt= 0.0869 p2 Nodes=1825 Cells=882 RMS Err= 3.6e-4
Integral= -0.453430

Transient Heat flow around an Insulating blob



22:25:18 5/23/05
FlexPDE 5.0.0

HISTORY
1: Phi



ex3: Cycle=92 Time= 6.2832 dt= 0.0869 p2 Nodes=1825 Cells=882 RMS Err= 3.6e-4

4.7.1. 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 than 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.

4.8. 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($k \cdot \text{grad}(\text{Phi})$) + **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(- $k \cdot \text{grad}(\text{Phi})$)

ELEVATION(Phi) FROM (0,-1) to (0,1)

ELEVATION(Normal(- $k \cdot \text{grad}(\text{Phi})$)) ON 'ring'

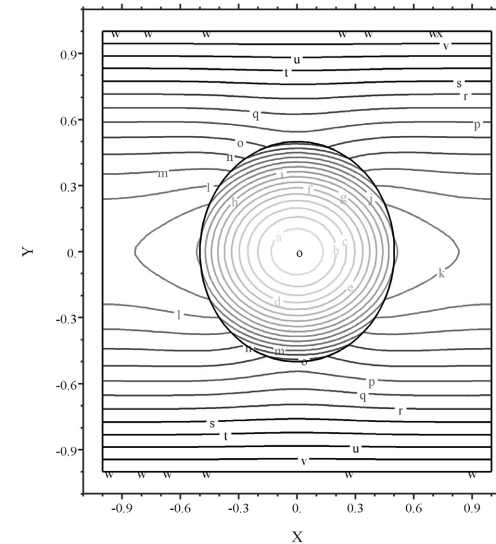
END

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:

Modal Heat Flow Analysis



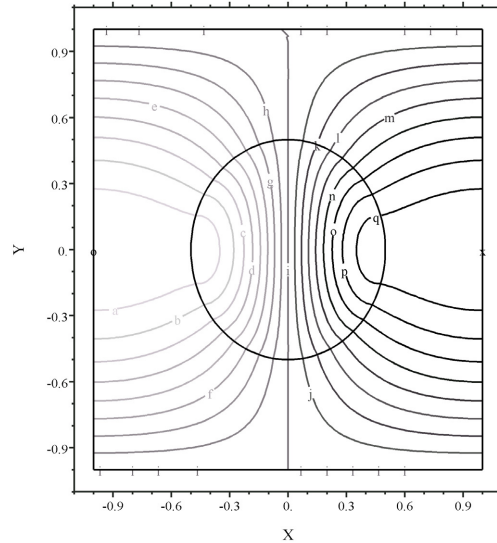
ex4: Grid#1 p2 Nodes=1201 Cells=570 RMS Err= 1.8e-4
Mode 1 Lambda= 2.0761 Integral= -3.406523

21:17:34 5/23/05
FlexPDE 5.0.0

Phi	
max	0.00
w :	0.00
v :	-0.10
u :	-0.20
t :	-0.30
s :	-0.40
r :	-0.50
q :	-0.60
p :	-0.70
o :	-0.80
n :	-0.90
m :	-1.00
l :	-1.10
k :	-1.20
j :	-1.30
i :	-1.40
h :	-1.50
g :	-1.60
f :	-1.70
e :	-1.80
d :	-1.90
c :	-2.00
b :	-2.10
a :	-2.20
min	-2.28

Modal Heat Flow Analysis

21:17:34 5/23/05
FlexPDE 5.0.0



Phi	
max	1.79
q :	1.60
p :	1.40
o :	1.20
n :	1.00
m :	0.80
l :	0.60
k :	0.40
j :	0.20
i :	0.00
h :	-0.20
g :	-0.40
f :	-0.60
e :	-0.80
d :	-1.00
c :	-1.20
b :	-1.40
a :	-1.60
min	-1.79

ex4: Grid#1 p2 Nodes=1201 Cells=570 RMS Err= 1.8e-4
Mode 2 Lambda= 3.4320 Integral= -3.331903e-5

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

4.8.1. 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))
```

5. 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.

5.1. 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.

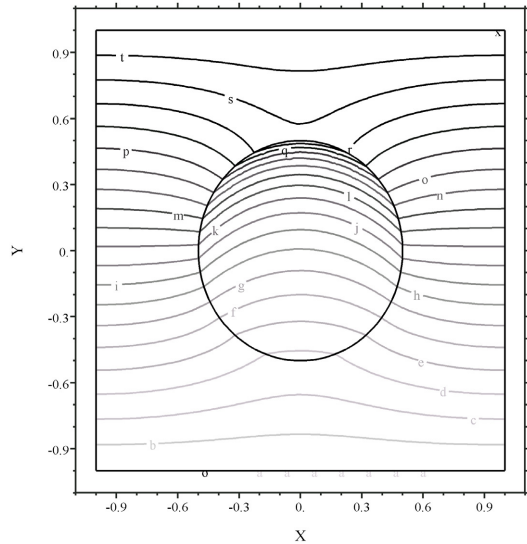
Suppose, for example, that the conductivity in the 'blob' of our example problem were in fact a strong function of the temperature. Say, for example, that $K = \exp(-5 \cdot \phi)$. The solution couldn't be simpler. Just define it the way you want it and click "run":

```
...  
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:

Heat flow around an Insulating blob

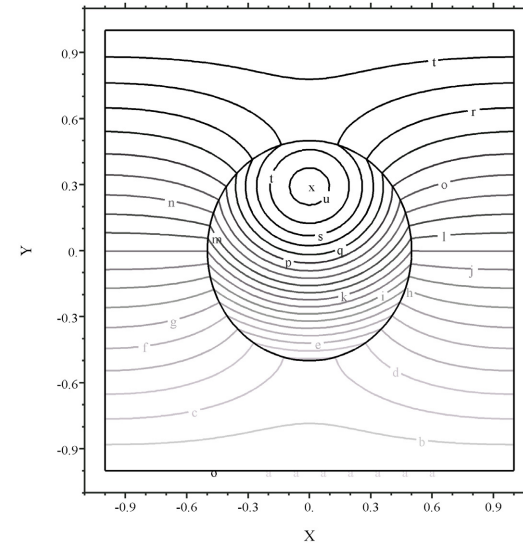
21:19:02 5/23/05
FlexPDE 5.0.0

Phi

max	1.00
u :	1.00
t :	0.95
s :	0.90
r :	0.85
q :	0.80
p :	0.75
o :	0.70
n :	0.65
m :	0.60
l :	0.55
k :	0.50
j :	0.45
i :	0.40
h :	0.35
g :	0.30
f :	0.25
e :	0.20
d :	0.15
c :	0.10
b :	0.05
a :	0.00
min	0.00

ex5: Grid#2 p2 Nodes=1365 Cells=652 RMS Err= 3.9e-4
Integral= 1.915986

Heat flow around an Insulating blob

21:19:39 5/23/05
FlexPDE 5.0.0

Phi

max	1.02
u :	1.00
t :	0.95
s :	0.90
r :	0.85
q :	0.80
p :	0.75
o :	0.70
n :	0.65
m :	0.60
l :	0.55
k :	0.50
j :	0.45
i :	0.40
h :	0.35
g :	0.30
f :	0.25
e :	0.20
d :	0.15
c :	0.10
b :	0.05
a :	0.00
min	0.00

ex6: Grid#1 p2 Nodes=1201 Cells=570 RMS Err= 1.6e-4
Integral= 2.123970

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

EQUATIONS

$$\text{Div}(k*\text{grad}(\text{phi})) + 0.01*\text{phi}*\sin(\text{phi}) = 0$$

Click "run", and the solution appears:

5.1.1. 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 to the question of intransigent nonlinear problems.

5.2. 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,

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

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

The surface value of $n \cdot \nabla u$ 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,

$$\int_a^b u dv = uv \Big|_a^b - \int_a^b v du$$

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 $\mathbf{n} \cdot \nabla u$.

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 PDE by parts.

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.

5.2.1. 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

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

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

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

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

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

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

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

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

The Heat Equation

$\text{Div}(-k*\text{grad}(\text{Temp})) + \text{Source} = 0$
 Natural(Temp) = outward surface-normal flux = normal(-k*grad(Temp))
 [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

$\text{dx}(-k*\text{dx}(\text{Temp})) + \text{Source} = 0$
 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

$\text{curl}(\text{curl}(A)/\mu) = J$
 Natural(A) = tangential component of H = tangential(curl(A)/mu)

Convection Equation

$\text{dx}(u)-\text{dy}(u)=0$
 Natural(u) is undefined, because there are no second-order terms.
 See the section "Hyperbolic systems" for further discussion.

5.2.2. 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

$$\text{Flux} = -K*\text{grad}(\text{Phi}) = \text{Phi} - \text{Phi}_0$$

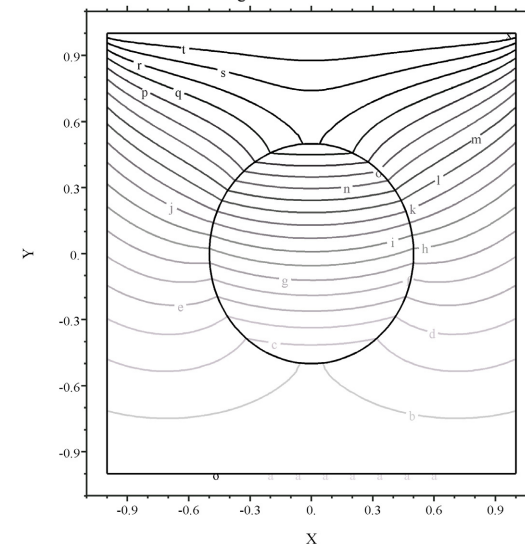
where Phi0 is a reference cooling temperature. With convectively cooled sides, our boundary specification looks like this (assuming Phi0=0):

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:

Heat flow around an Insulating blob



21:20:09 5/23/05
 FlexPDE 5.0.0

Phi	
max	1.00
u :	1.00
t :	0.95
s :	0.90
r :	0.85
q :	0.80
p :	0.75
o :	0.70
n :	0.65
m :	0.60
l :	0.55
k :	0.50
j :	0.45
i :	0.40
h :	0.35
g :	0.30
f :	0.25
e :	0.20
d :	0.15
c :	0.10
b :	0.05
a :	0.00
min	0.00

ex7: Grid#2 p2 Nodes=1268 Cells=601 RMS Err= 2.e-4
 Integral= 1.590400

5.3. 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 $(V_2 - V_1)$ in material '1', and $(V_1 - V_2)$ 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 | Misc | 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.

5.3.1. 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

$$F_1 = -K_1 \cdot dn(T) = -(T_2 - T_1)/R$$

where F_1 is the value of the outward heat flux, K_1 is the heat conductivity, $dn(T)$ is the outward normal derivative of T , R is the resistance of the interface film, and T_1 and T_2 are the two values of the temperature at the interface.

As seen from material '2',

$$F_2 = -K_2 \cdot dn(T) = -(T_1 - T_2)/R = -F_1$$

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

$$\text{div}(-K \cdot \text{grad}(T)) = H$$

is given by the divergence theorem as

$$\text{Natural}(T) = -K \cdot dn(T),$$

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 $(T_2 - T_1)$ in material '1' and $(T_1 - T_2)$ in material '2'.

The representation of the contact resistance boundary condition is therefore

$$\text{CONTACT}(T) = -\text{JUMP}(T)/R$$

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
```

```

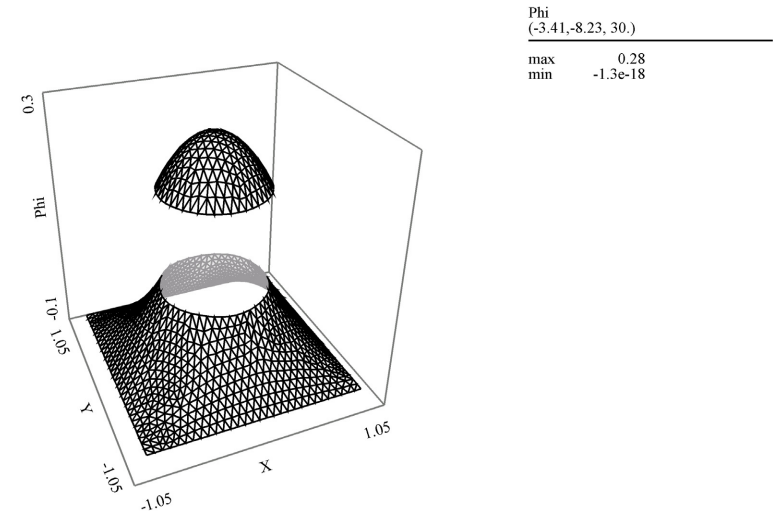
Phi    { the temperature }
DEFINITIONS
K = 1  { default conductivity }
R = 0.5 { blob radius }
H = 0  { internal heat source }
Res = 0.5 { contact resistance }
EQUATIONS
Div(-k*grad(phi)) = H
BOUNDARIES
REGION 1 'box'
START(-1,-1)
VALUE(Phi)=0 { cold outer walls }
LINE TO (1,-1) TO (1,1) TO (-1,1) TO 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) mesh
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:

Contact Resistance on a heated blob

21:24:25 5/23/05
FlexPDE 5.0.0



ex8: Grid#1 p2 Nodes=1249 Cells=570 RMS Err= 1.3e-4
Integral= 0.304612

5.3.2. 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

$$\text{CONTACT}(\phi) = 0 * \text{JUMP}(\phi)$$

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.]

5.3.3. 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

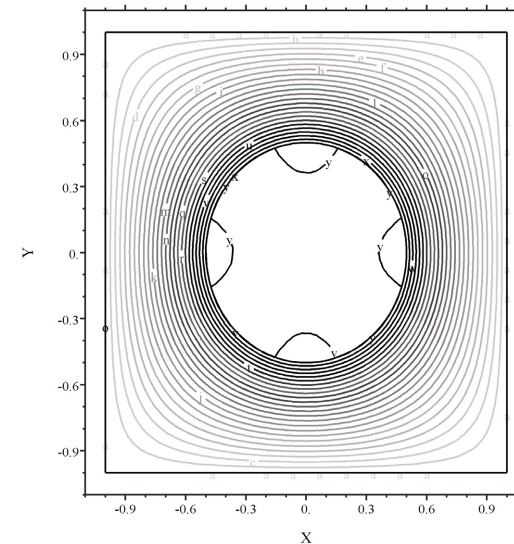
CONTOUR(Phi) SURFACE(Phi)

CONTOUR(temp) SURFACE(temp)

END

The temperature shows the effect of the surface source:

Contact Resistance as a heat source



21:28:39 5/23/05
FlexPDE 5.0.0

temp	
max	2.41
y :	2.40
x :	2.30
w :	2.20
v :	2.10
u :	2.00
t :	1.90
s :	1.80
r :	1.70
q :	1.60
p :	1.50
o :	1.40
n :	1.30
m :	1.20
l :	1.10
k :	1.00
j :	0.90
i :	0.80
h :	0.70
g :	0.60
f :	0.50
e :	0.40
d :	0.30
c :	0.20
b :	0.10
a :	0.00
min	0.00

Scale = E-2

ex9: Grid#1 p2. Nodes=1249 Cells=570 RMS Err= 2.8e-4
Integral= 0.045590

6. 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 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 endpoints 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
```

7. 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.

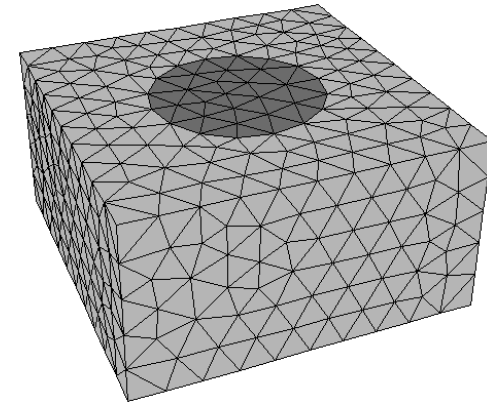
The finite element model constructed by FlexPDE in three-dimensional domains is fully general. The domain definition process is not.

7.1. 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.

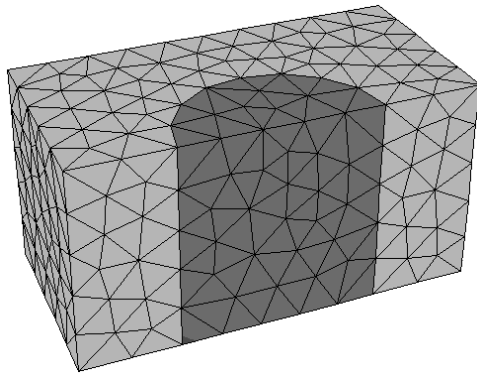
[Note: It is important to consider carefully the characteristics of any given problem, to determine the orientation most amenable to extrusion.]

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:



7.2. 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.

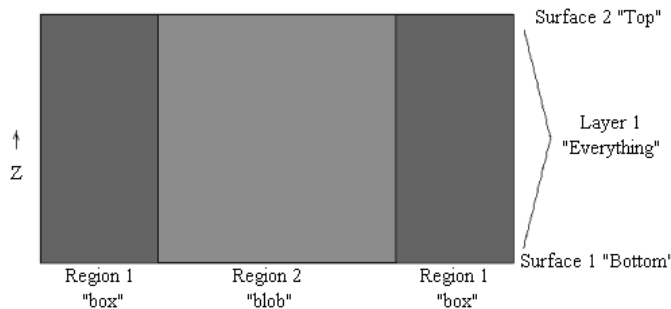
The layer between these two surfaces comprises everything in the domain, so we can name it 'Everything'.

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.

7.3. 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):

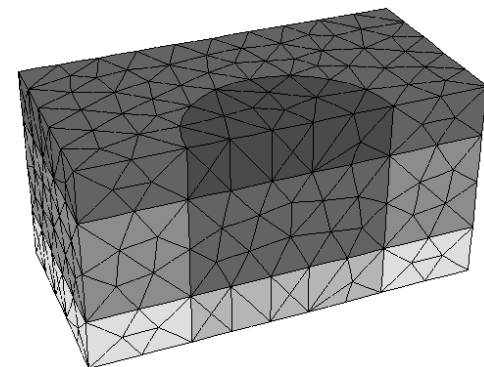
```

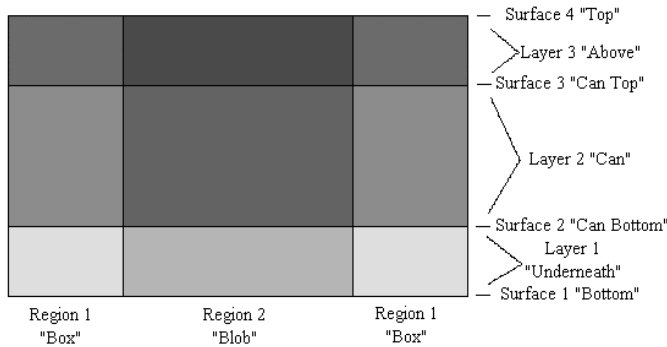
EXTRUSION
  SURFACE "Bottom" z=-1/2
  LAYER "Underneath"
  SURFACE "Can Bottom" z=-1/4
  LAYER "Can"
  SURFACE "Can Top" z=1/4
  LAYER "Above"
  SURFACE "Top" z=1/2
  
```

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.

7.4. Setting Material Properties by Region and Layer

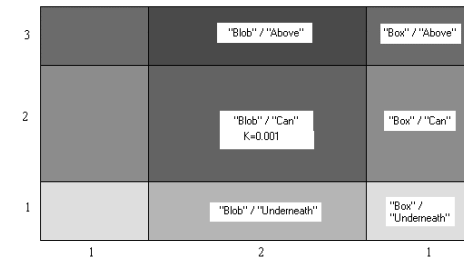
In our 2D problem, we specified the conductivity of the blob inside the **REGION** definition for the blob, and that continues to be the technique in 3D.

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 compartment 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 }

```

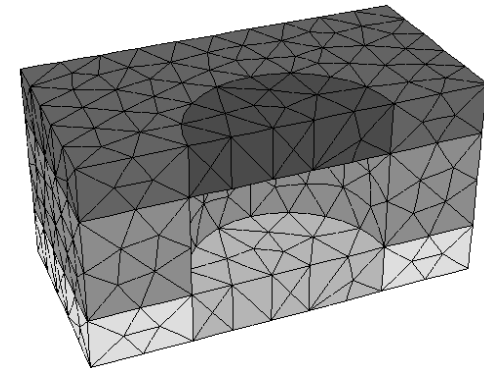
7.5. Void Compartments

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

```



	"Blob" / "Above"	"Box" / "Above"
		"Box" / "Can"
	"Blob" / "Undereath"	"Box" / "Undereath"

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

7.6. Limited Regions

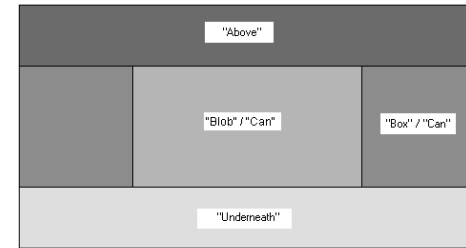
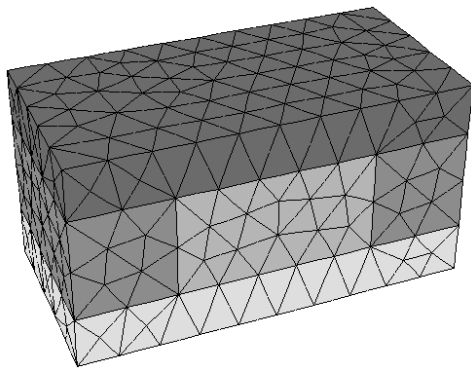
In what we have discussed so far, the region structure specified in the 2D base plane has been propagated unchanged throughout the extrusion dimension. FlexPDE uses the specifier **LIMITED REGION** to restrict the defined region to a specified set of layers and/or surfaces.

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 cannister is not propagated through the "Above" and "Underneath" layers:



7.7. 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.]

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.

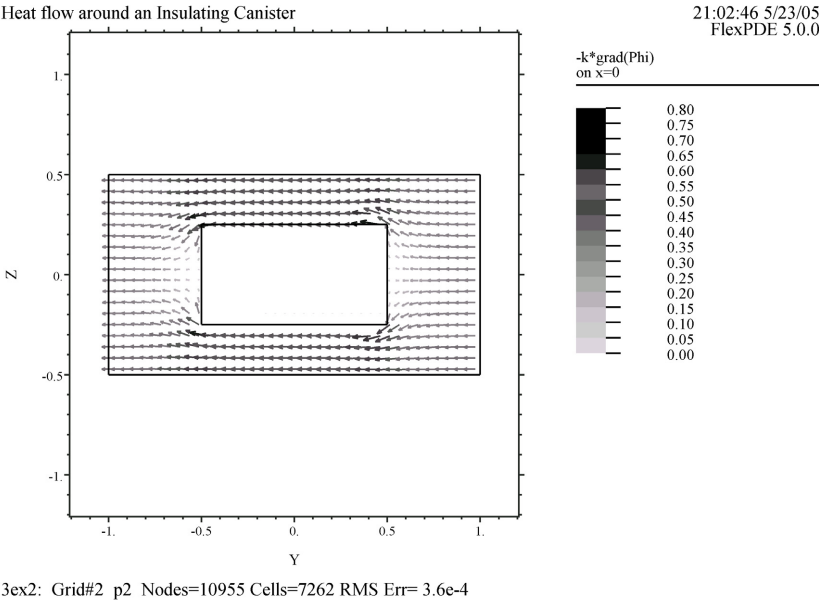
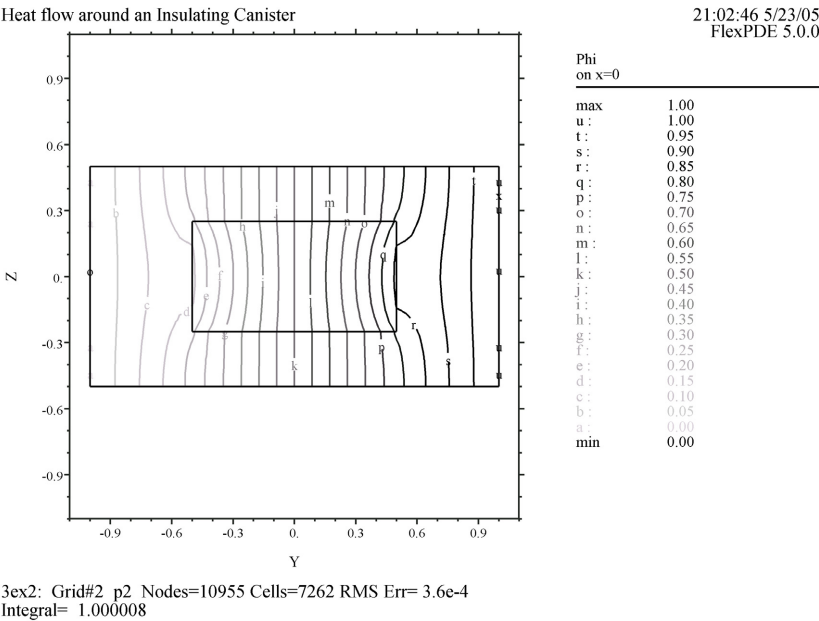
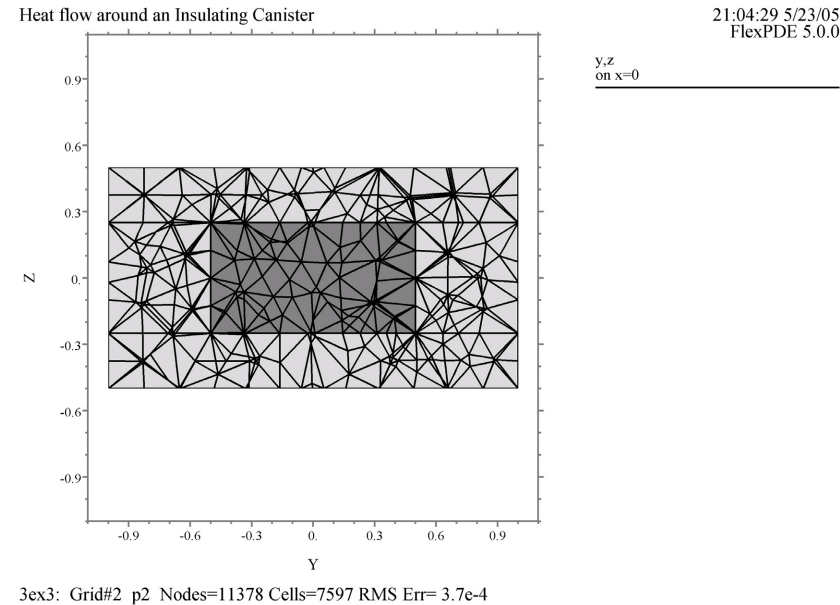
7.8. The Complete 3D Canister

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)=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
  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:

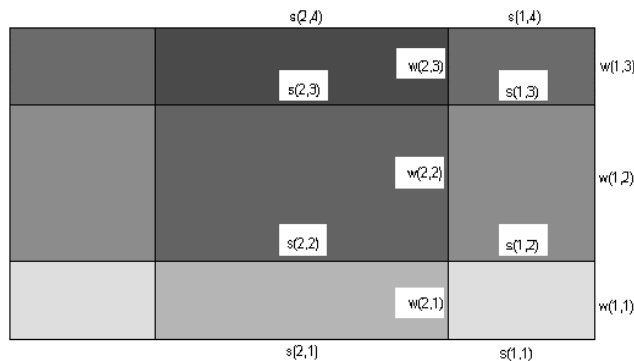


7.9. Setting Boundary Conditions in 3D

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) { boundary conditions on surface 1 over full domain }
SURFACE 2
  s(all, 2) { boundary conditions on surface 2 over full domain }
  {...other surfaces }
REGION 1
  SURFACE 1
    s(1,1) { boundary conditions on surface 1, restricted to region 1 }
  SURFACE 2
    s(1,2) { boundary conditions on surface 2, restricted to region 1 }
  ...
  START(,) { -- begin the perimeter of region m }
    w(1,...) { boundary conditions on following segments of sidewall
of region 1 on all layers }
    LAYER 1
      w(1,1) { boundary conditions on following segments of
sidewall of region 1, restricted to layer 1 }
    LAYER 2
      w(1,2) { boundary conditions 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 boundary conditions 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) { boundary conditions on surface 1, restricted to region 2 }
  SURFACE 2
    s(2,2) { boundary conditions on surface 2, restricted to region 2 }
  ...
  START(,) { -- begin the perimeter of region m }
    w(2,...) { boundary conditions on following segments of sidewall of
region 2 on all layers }
    LAYER 1
      w(2,1) { boundary conditions on following segments of sidewall
of region 2, restricted to layer 1 }

```

```

LAYER 2
  w(2,2) { boundary conditions 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 boundary conditions 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)=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 }
  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

```

7.10. 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

$$Z = Z_{\text{center}} + \sqrt{R^2 - x^2 - y^2}$$

Or,

$$Z = Z_{\text{top}} - R + \sqrt{R^2 - x^2 - y^2}$$

- 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

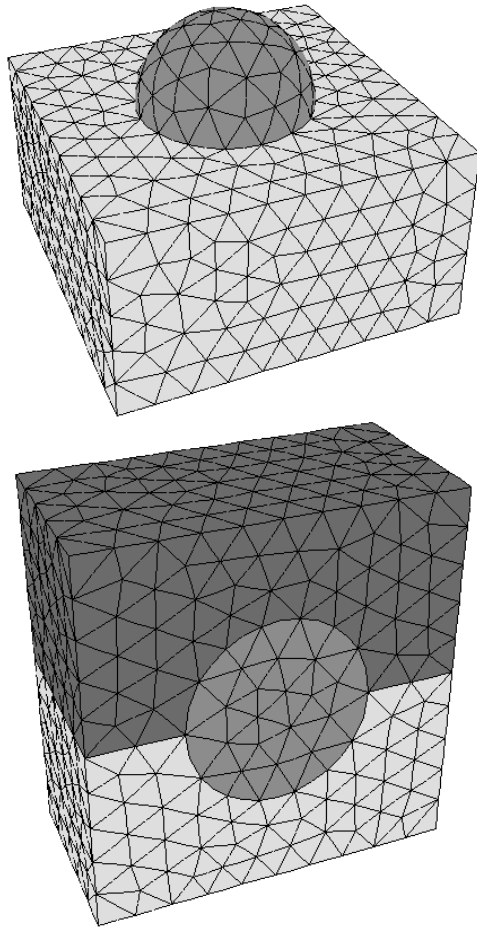
```
REGION 1 'box'
START(-1,-1)
VALUE(Phi)=0 LINE TO (1,-1)
NATURAL(Phi)=0 LINE TO (1,1)
VALUE(Phi)=1 LINE TO (-1,1)
NATURAL(Phi)=0 LINE TO CLOSE
LIMITED REGION 2 'blob' { the embedded blob }
LAYER 2 K = 0.001
START 'ring' (RSphere,0) ARC(CENTER=0,0) ANGLE=360
TO CLOSE
```

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)
```

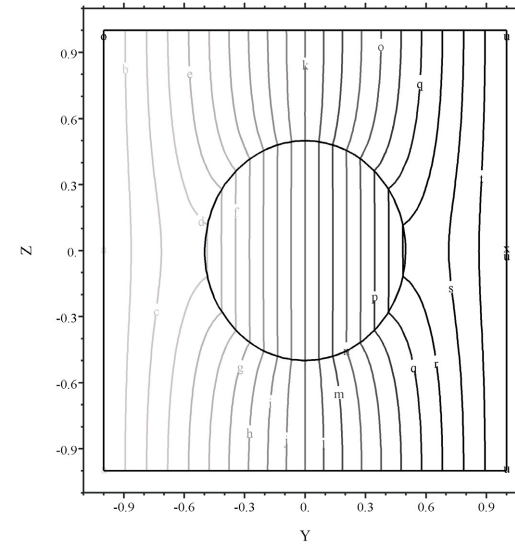
END

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:

Heat flow around an Insulating Sphere



3ex4: Grid#1 p2 Nodes=12525 Cells=8188 RMS Err= 3.3e-4

21:06:18 5/23/05
FlexPDE 5.0.0

Phi
on x=0.01

max	1.00
u :	1.00
t :	0.95
s :	0.90
r :	0.85
q :	0.80
p :	0.75
o :	0.70
n :	0.65
m :	0.60
l :	0.55
k :	0.50
j :	0.45
i :	0.40
h :	0.35
g :	0.30
f :	0.25
e :	0.20
d :	0.15
c :	0.10
b :	0.05
a :	0.00
min	0.00

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".

7.11. 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 | Misc | 3D_Integrals.pde" 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)=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' { 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'

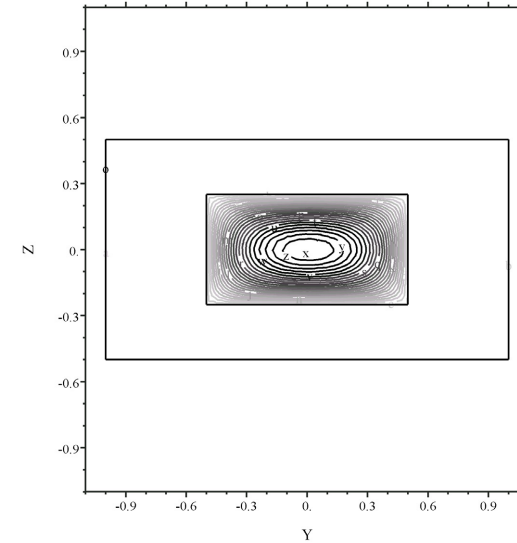
REPORT(Vol_Integral(S,'blob','can')-Surf_Integral(NORMAL(-k*grad(Phi))))

AS 'Energy Error'

END

The contour plot is as follows:

Heat flow from an Insulating Canister



3ex5: Grid#3 p2 Nodes=20322 Cells=14201 RMS Err= 5.5e-4
Integral= 7.383990

21:11:06 5/23/05
FlexPDE 5.0.0

Phi
on x=0

max	25.9
z :	25.0
y :	24.0
x :	23.0
w :	22.0
v :	21.0
u :	20.0
t :	19.0
s :	18.0
r :	17.0
q :	16.0
p :	15.0
o :	14.0
n :	13.0
m :	12.0
l :	11.0
k :	10.0
j :	9.00
i :	8.00
h :	7.00
g :	6.00
f :	5.00
e :	4.00
d :	3.00
c :	2.00
b :	1.00
a :	0.00
min	0.00

The summary page shows the integral reports:

SUMMARY

Source Integral= 0.392690

Can Heat Loss= 0.392680

Box Heat Loss= 0.392680

Energy Error= 1.048038e-5

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

7.12. 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.

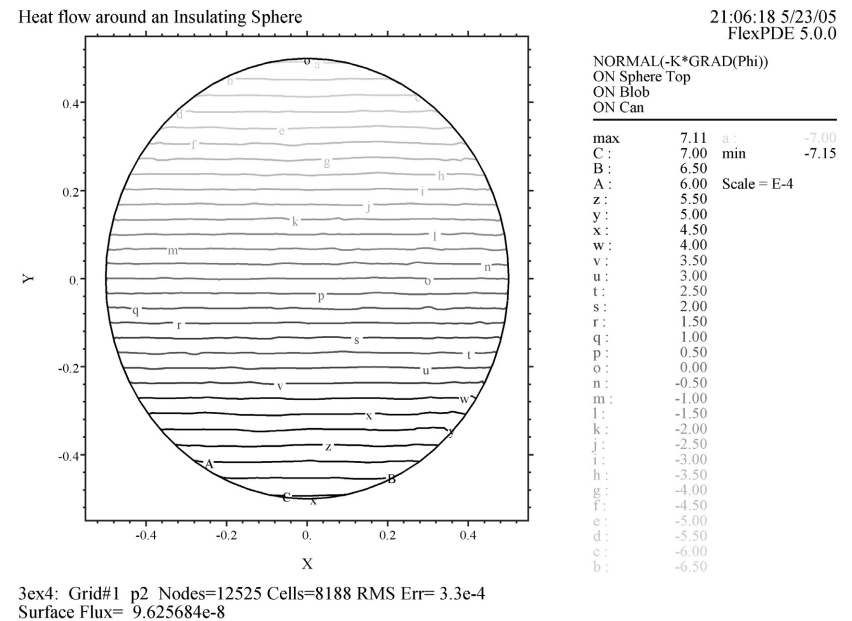
- In general, the qualifier **ON <name>** will request a localization of the plot, depending on the type of object names by <name>.
-
- The qualifier **ON REGION <number>** selects a region by number, rather than by name.
-
- The qualifier **ON SURFACE <number>** selects a layer interface surface by 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 value $9.6e-8$ is well within the default error target of $ERR_{LIM}=0.001$. The plot grid integral, "Surf_Integral", shows greater error at $8.96e-6$, due to poorer resolution of integrating the area-weighted function in the plot plane.

8. 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 always 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.

8.1. Mesh Balancing

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

For example, suppose we change our basic example problem to model a sphere of oscillating size $R_m = 0.5 + 0.25 \sin(t)$.

We will define surrogate coordinates for X and Y and mesh velocity variables:

```
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 simply the geometric rules

```
VALUE(Um) = 0.25*cos(t)*x/r
VALUE(Vm) = 0.25*cos(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
```

8.2. The Pulsating Blob

The modified script for our example problem is now:

```
TITLE 'Heat flow around an Insulating blob'
VARIABLES
  Phi    { the temperature }
  Xm = MOVE(x) { surrogate X }
  Ym = MOVE(y) { surrogate Y }
  Um     { mesh x-velocity }
  Vm     { mesh y-velocity }

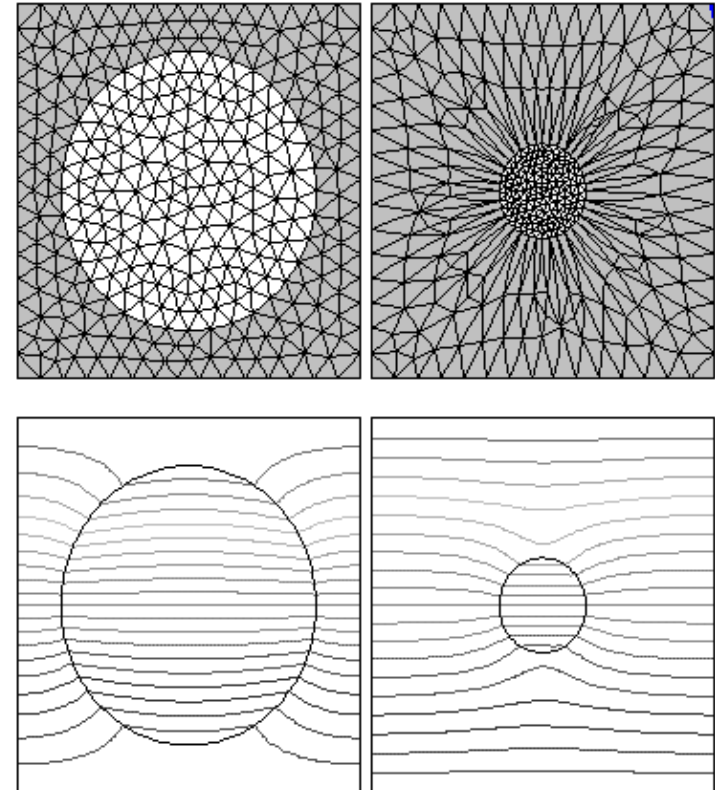
DEFINITIONS
  K = 1      { default conductivity }
  R0 = 0.5   { initial blob radius }

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' (R,0)
    VELOCITY(Xm) = Um
    VELOCITY(Ym) = Vm
```

```
VALUE(Um) = 0.25*cos(t)*x/r
VALUE(Vm) = 0.25*cos(t)*y/r
ARC(CENTER=0,0) ANGLE=360 TO CLOSE
PLOTS
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)
    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.



9. 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** and **MESH_DENSITY**. **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.

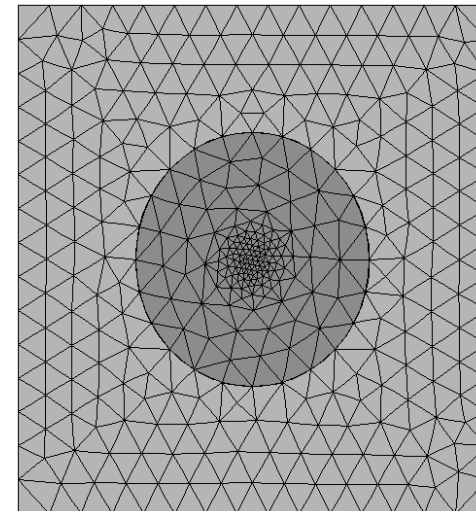
Used as defined parameters, these controls may appear in the **DEFINITIONS** section or may be redefined in subsequent regional redefinition sections. In this use, the controls specify the volume or area mesh density over a region or over the entire domain.

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", 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 | Misc | Mesh_Control | Mesh_Spacing.pde" and "Samples | Misc | Mesh_Control | Mesh_Density.pde".

Adaptive 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 wherever something interesting is known to take place in the problem.]

See also the FRONT and RESOLVE statements for additional controls.

10. 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.
- See the example problems "export_format" and "export_history".

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 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"

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.visuallogic.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 gridsizes 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	Misc	Import-Export	Export.pde
Samples	Misc	Import-Export	Export_Format.pde
Samples	Misc	Import-Export	Export_History.pde
Samples	Misc	Import-Export	Transfer_Out.pde
Samples	Misc	Import-Export	Transfer_In.pde
Samples	Misc	Import-Export	Table.pde

Note:

Reference to products from other suppliers does not constitute an endorsement by PDE Solutions Inc.

11. 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.

Several selectors are provided to enable more robust (but more expensive) treatment in difficult cases. The primary selector **PREFER_STABILITY** allows up to three Newton iterations in each timestep, with derivatives recomputed at each iteration. It also modifies the error weighting scheme to place more emphasis on very localized activity. **PREFER_STABILITY** resets the values of the **NRUPDATE** and **TNORM**.

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 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 characteristics 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 $\text{Div}(\text{grad}(u))$ is a low-pass filter that can be used to smooth oscillations in the solution. See the Technical Note "Smoothing Operators in PDE's" for technical discussion of this operator. In brief, you can use a term $\text{eps} * \text{Div}(\text{Grad}(u))$ in a PDE to smooth oscillations of spatial extent D by setting $\text{eps} = D^2 / \pi^2$ in steady state or $\text{eps} = 2 * D * c / \pi$ 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 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 $\text{DIV}(K * \text{GRAD}(U)) = \text{DT}(U)$. 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.

12. 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.

Index

Accuracy	31	INTEGRAL	33
Application	7	Integrals	33, 40
ARC	17	Integrals in Three Dimensions	95
AREA_INTEGRAL	33	integration by parts	59
BINTEGRAL	33	JUMP	63, 67
BOUNDARIES	8, 17	LAYER	74
boundary conditions	10	Layer Interfaces	91
Boundary conditions	24	Layering	76
Boundary Conditions in 3D	87	LINE	17
case sensitivity	15	LINE_INTEGRAL	33
CDF	108	Magnetic Field	59
CHANGELIM	56	material parameters	22
Contact Resistance	63	Material Properties	78
CONTOUR	26	mesh	20
Curl Theorem	59	Mesh Density	105
Decoupling Variables	66	MONITORS	26
DEFINITIONS	8, 22	NATURAL	10
Differentiation	15	Natural boundary condition	59
Discontinuous Variables	63	NATURAL boundary condition	24
Divergence Theorem	59	Newton-Raphson iteration	56
Domain Description	17	nonlinear problems	56
Eigenvalue Summary	51	Nonlinear Problems	112
ELEVATION	26	Notation	15
EQUATIONS	8, 16	ON	99
ERRLIM	31	ON LAYER	99
error tolerance	31	ON REGION	99
EXPORT	108	ON SURFACE	99
Extrusion	72	Parameter Studies	37
EXTRUSION	74	parameters	22
Extrusion Notation	74	PLOTS	8, 26
finite element mesh	20	plots on cut planes	99
Flux Boundary Condition	61	problem setup	11
FORMAT	108	Problem Setup Guidelines	13
guidelines	13	problem solving environment	2
Heat Equation	59	questions	115
Inconsistent Initial Conditions	47	REGION	17
Initial Conditions	47	REPORT	35
initial value	56	script editing module	5
INITIAL VALUES	56	scripting language	2
INITIALVALUES	8	SELECT	8
Instantaneous Switching	47		

Shaped Layer Interfaces	91	Transferring Data	108
STAGED	37	VALUE	10
STAGES	37	VALUE (or Dirichlet) boundary	
START	17	condition	24
SUMMARY	36, 51	VARIABLES	8, 16
SURFACE	26, 74	VECTOR	26
surface integrals	95	VisIt	108
symbolic equation analyzer	5	Void Compartments	80
TABLE Output	108	VOL_INTEGRAL	33
TECPLOT	108	volume integrals	95
TITLE	8	VTK	108