

Appendix A—Solving PDEs with PDE2D

A.1 History

The author began development of a general purpose partial differential equation solver in 1974 and has been working continuously on this project since that date [earliest reference: Sewell 1976]. Originally dubbed 2DEPEP, it was marketed by IMSL (now VNI) from 1980 to 1984 under the name TWODEPEP [Sewell 1982b], then from 1984 to 1991 as PDE/PROTRAN [Sewell 1985]. User input to all three programs was in the form of a high level language; a sample input program for PDE/PROTRAN, which shared an input language with three other IMSL “PROTRAN” products, is shown in Figure A.1.1. These programs solved progressively more general PDEs in general two-dimensional regions, including nonlinear steady-state and time-dependent systems, and linear eigenvalue systems. TWODEPEP and PDE/PROTRAN had the ability to display their solutions graphically, through surface, contour, and vector field plots. These programs were written entirely in FORTRAN and used a Galerkin finite element method with triangular elements to solve 2D problems.

In 1991 the author developed a new version, with an interactive user interface, which has been sold under a new name, PDE2D, since that date. Despite its name, PDE2D 9.3 can now solve problems in one- and three- as well as two-dimensional regions, and many other new features have been added, such as adaptive grid refinement for 2D problems, adaptive time step control for time-dependent problems, and sparse direct linear system solvers.

The PDE2D web page (<http://www.pde2d.com>) contains further information, including a list of over 200 journal articles in which TWODEPEP, PDE/PROTRAN, or PDE2D were used to generate some or all of the numerical results.

```

$ PDE2D
C*** PDE/PROTRAN SOLVES DA/DX + DB/DY + F = 0
    F = 1.0
    A = UX
    B = UY
    SYMMETRIC
C*** INITIAL TRIANGULATION SPECIFICATION
    VERTICES = (0,0) (0,-1) (1,-1) (1,0) (1,1) (0,1) (-1,1)
    * (-1,0) (-.5,.5) (.5,.5) (.5,-.5)
    TRIANGLES = (1,2,11,-1) (2,3,11,-1) (3,4,11,-1) (4,1,11,0)
    * (4,5,10,-1) (5,6,10,-1) (6,1,10,0) (1,4,10,0)
    * (6,7,9,-1) (7,8,9,-1) (8,1,9,-1) (1,6,9,0)
C*** NUMBER OF TRIANGLES AND GRADING DESIRED FOR FINAL TRIANGULATION
    NTRIANGLES = 100
    TRIDENSITY = 1./(X*X+Y*Y)
C*** USE CUBIC ELEMENTS
    DEGREE = 3
C*** OUTPUT PARAMETERS
    SAVEFILE = PLOT
    GRIDPOINTS = 17,17
$ PLOTSURFACE
    TITLE = 'SURFACE PLOT OF SOLUTION'
$ END

```

Figure A.1.1
Sample PDE/PROTRAN Program

A.2 The PDE2D Interactive and Graphical User Interfaces

When introduced in 1991, PDE2D's primary new feature was an interactive user interface. Now all documentation, including 15 prepared examples, is available on-line through the interactive driver—in fact, there is no user's manual! The user only has to answer a series of interactive questions about the region, partial differential equations, and boundary conditions and select solution method and output options. The questions are worded to reflect the number and names of the unknowns, the type of system, and other information supplied in answers to previous questions. If a single linear steady-state problem is solved, for example, the user will not be bothered with information or options relevant only to time-dependent or nonlinear problems, or systems of several PDEs. Extensive error checking is done during the interactive session, and also at execution time.

```

-----
.
.
.
What type of PDE problem do you want to solve?

    1. a steady-state (time-independent) problem
    2. a time-dependent problem
    3. a linear, homogeneous eigenvalue problem

Enter 1,2 or 3 to select a problem type.
|---- Enter an integer value in the range 1 to 3
->1
    Is this a linear problem? ("linear" means all differential equations
    and all boundary conditions are linear)
|---- Enter yes or no
->no
    Give an upper limit on the number of Newton's method iterations
    (NSTEPS) to be allowed for this nonlinear problem. NSTEPS defaults
    to 15.

+++++ THE "FINE PRINT" (CAN USUALLY BE IGNORED) +++++
+ The iteration will stop if convergence occurs before the upper +
+ limit has been reached. In the FORTRAN program created by the +
+ preprocessor, NCON8Z will be .TRUE. if Newton's method has converged.+
+
+ For highly non-linear problems you may want to construct a one- +
+ parameter family of problems using the variable T, such that for +
+ T=1 the problem is easy (e.g. linear) and for T > N (N is less +
+ NSTEPS), the problem reduces to the original highly nonlinear +
+ problem. For example, the nonlinear term(s) may be multiplied by +
+ MIN(1.0,(T-1.0)/N).
+
+++++ END OF "FINE PRINT" +++++
NSTEPS = (Press [RETURN] to default)
|----Enter constant or FORTRAN expression-----|
->
    How many differential equations (NEQN) are there in your problem?
    NEQN =
|---- Enter an integer value in the range 1 to 99
->2
    You may now choose names for the component(s) of the (possibly vector)
    solution U. Each must be an alphanumeric string of one to three
    characters, beginning with a letter in the range A-H or O-Z. The
    variable names X,Y,T,S,A and B must not be used. The name should
    start in column 1.
    U1 =
->U
    U2 =
->V

```

```

.
.
.
Now enter FORTRAN expressions to define the PDE coefficients, which
may be functions of

                X,Y,U,Ux,Uy,V,Vx,Vy

They may also be functions of the initial triangle number KTRI
and, in some cases, of the parameter T.

Recall that the PDEs have the form

                d/dX*A1 + d/dY*B1 = F1
                d/dX*A2 + d/dY*B2 = F2

Do you want to write a FORTRAN block to define some parameters to be
used in the definition of these coefficients?
|---- Enter yes or no
->yes
Remember to begin FORTRAN statements in column 7
|-----7-----Input FORTRAN now (type blank line to terminate)-----|
->      Visc = 0.002
->      Rho = 1.0
->      P = -alpha*(Ux+Vy)

F1 =          (Press [RETURN] to default to 0)
|----Enter constant or FORTRAN expression-----|
-> Rho*(U*Ux+V*Uy)
A1 =          (Press [RETURN] to default to 0)
|----Enter constant or FORTRAN expression-----|
-> -P+2*Visc*Ux
B1 =          (Press [RETURN] to default to 0)
|----Enter constant or FORTRAN expression-----|
-> Visc*(Uy+Vx)
F2 =          (Press [RETURN] to default to 0)
|----Enter constant or FORTRAN expression-----|
-> Rho*(U*Vx+V*Vy)
A2 =          (Press [RETURN] to default to 0)
|----Enter constant or FORTRAN expression-----|
-> Visc*(Uy+Vx)
B2 =          (Press [RETURN] to default to 0)
|----Enter constant or FORTRAN expression-----|
-> -P+2*Visc*Vy
.
.
.
-----|

```

Figure A.2.1
Portion of PDE2D Interactive Session

A portion of a sample interactive session is shown in Figure A.2.1. Based on the user's answers, PDE2D creates a FORTRAN program that contains

calls to precompiled PDE2D library subroutines designed to solve the PDEs. The FORTRAN program thus created is highly readable—most of the interactive questions are repeated in the comments—so the user can make corrections and modifications directly to the FORTRAN program and does not have to work through a new interactive session each time minor modifications to the problem or solution method are desired. Although the PDE2D user does NOT need to be a FORTRAN programmer (for most problems, he/she only needs to know how to write basic FORTRAN expressions such as $X*Y+Z$), the PDE2D user has all the flexibility of FORTRAN at his/her disposal. For example, any PDE or boundary condition coefficient can be defined by a user-written function subprogram, or a DO loop can be put around the main program so that the program is run multiple times, with different parameters, in a single run. It is also very easy to add calls to user-supplied subroutines to plot or otherwise postprocess the PDE2D solution. In fact, PDE2D now automatically outputs the solution to a *MATLAB*[®] m-file, from which you have easy access to all of MATLAB's graphical abilities.

[Continue](#)

Begin Describing your PDE Problem

Dimension	<input type="text" value="3d"/>	
Problem type	<input type="text" value="steady-state"/>	
Number of PDEs (NEQN)	<input type="text" value="3"/>	
Precision	<input type="text" value="double"/>	(Double strongly recommended)
Linear?	<input checked="" type="radio"/>	(If unsure, assume nonlinear)

Figure A.2.2
Page of PDE2D GUI Session

Edition 9.0 introduced a new Graphical User Interface (GUI), which can be used as an alternative to the interactive user interface to create PDE2D

programs for 0D, 1D, 2D and 3D problems, using the collocation method. Users who have to solve problems in 2D regions with complex geometry, which requires the Galerkin method, still have to use the interactive user interface, not the GUI. However, for 0D and 1D problems, and for 2D and 3D problems in a wide range of simple regions, the GUI is an extraordinarily easy, user-friendly way to construct PDE2D programs. Figure A.2.2 shows a page of a PDE2D GUI session.

A.3 One-Dimensional Steady-State Problems

For one-dimensional problems, the PDE2D user has a choice between a collocation method and a Galerkin method. The Galerkin method is very similar to the 2D Galerkin method described in the next section (1D Lagrange polynomials of up to fourth degree are used) and thus will not be further discussed here.

The collocation algorithm solves one-dimensional, nonlinear, steady-state systems of the form:

$$\begin{aligned} F_1(x, U_1, \dots, U_M, U_{1x}, \dots, U_{Mx}, U_{1xx}, \dots, U_{Mxx}) &= 0, \\ &= \\ &= \\ F_M(x, U_1, \dots, U_M, U_{1x}, \dots, U_{Mx}, U_{1xx}, \dots, U_{Mxx}) &= 0. \end{aligned}$$

Boundary conditions for 1D problems have the form:

$$\begin{aligned} G_1(x, U_1, \dots, U_M, U_{1x}, \dots, U_{Mx}) &= 0, \\ &= \\ &= \\ G_M(x, U_1, \dots, U_M, U_{1x}, \dots, U_{Mx}) &= 0. \end{aligned}$$

at $x = xa, x = xb$; or periodic boundary conditions:

$$\begin{aligned} U_i(xa) &= U_i(xb) \quad (\text{for } i = 1, \dots, M). \\ U_{ix}(xa) &= U_{ix}(xb) \end{aligned}$$

The PDE2D collocation finite element method uses cubic Hermite basis functions, as described in Section 5.4. That is, each unknown function is assumed to be a linear combination of the 2 NXGRID basis functions:

$$H_i(x), S_i(x) \quad (\text{for } i = 1, \dots, NXGRID),$$

where the functions H_i and S_i are defined in 5.3.5. The approximate solution is required to satisfy the PDEs exactly at two collocation points $x_i + \beta_{1,2}(x_{i+1} - x_i)$ ($\beta_1 = 0.5 - 0.5/\sqrt{3}, \beta_2 = 0.5 + 0.5/\sqrt{3}$) in each of the NXGRID-1 subintervals (x_i, x_{i+1}) , and to satisfy the boundary conditions at xa and xb . The number of boundary collocation points (2) plus the number of interior collocation points ($2(\text{NXGRID}-1)$) is equal to the number of basis functions (2 NXGRID). Since there are M PDEs to satisfy at each collocation point, and M unknowns per basis function (each U_i is expanded as a linear combination of the basis functions), this ensures that the number of algebraic equations equals the number of unknowns, $N = 2 \text{ NXGRID } M$.

Newton's method (Section 4.2), with finite difference calculated (or user-supplied) Jacobian matrix elements, is used to solve the nonlinear algebraic equations produced when the collocation method is applied to a nonlinear steady-state PDE system, and the linear system that must be solved each Newton iteration is solved using the Harwell Library sparse direct solver MA37 (used by permission), which employs a minimal degree algorithm (see Section 0.5). Normally, this linear system will involve a nonsymmetric band matrix of half-bandwidth $L=3 M-1$, which could be solved efficiently by a band solver such as LBAND (Figure 0.4.4), but when periodic boundary conditions are requested, the matrix has nonzero elements in the upper right and lower left corners, and the half-bandwidth becomes large ($L = N - 1$). Though it is fairly easy (see Problem 3 of Chapter 4) to modify a band solver to handle such systems efficiently, a sparse direct solver such as Harwell's MA37 [Duff and Reid 1984] can also handle this sparse structure efficiently. If the PDE system is linear, Newton's method is still used, but then only one iteration is necessary.

The use of cubic Hermite basis functions produces solutions with $O(h^4)$ accuracy, where $h = \max(x_{i+1} - x_i)$, which compares favorably with the $O(h^2)$ accuracy obtained when centered finite difference methods or finite element methods with linear elements are used. To illustrate the high accuracy obtained using cubic elements, we solve the problem:

$$\begin{aligned} U_{rr} + U_r/r &= r^2, \\ U(1) &= 0 \end{aligned}$$

(no boundary condition at $r = 0$).

This is the equation $U_{xx} + U_{yy} = x^2 + y^2$, with $U = 0$ on the boundary of the unit circle, after it is reduced to polar coordinates. The exact solution is $U_{\text{true}} = (r^4 - 1)/16$. The relative errors using a uniform grid of NXGRID points are shown in Table A.3.1, where "relative error" is defined as the integral of $|U - U_{\text{true}}|$ over the interval $(0, 1)$ divided by the integral of $|U_{\text{true}}|$. (Integrals and boundary integrals of any function of the solution and its derivatives are computed accurately and automatically by PDE2D for 1D, 2D, and 3D problems.)

Table A.3.1
Errors for 1D Problem

NXGRID	N	Relative Error	Total CPU Time on Cray J90
26	52	2.66E-7	0.076 sec.
51	102	1.67E-8	0.107
101	202	1.11E-9	0.167

The experimental rate of convergence obtained by comparing the last two errors is $\log(1.67 * 10^{-8}/1.11 * 10^{-9})/\log(100/50) = 3.9$, which agrees well with the expected $O(h^4)$ accuracy ($h = 1/(\text{NXGRID}-1)$).

Notice there is no boundary condition at $r = 0$; thus rather than requiring that the approximate solution satisfy a boundary condition there, the PDE is enforced at a point very near the boundary (not quite on the boundary, to avoid the singularity there). In other words, the boundary collocation point is treated as an additional interior collocation point.

A.4 Two-Dimensional Steady-State Problems

For two-dimensional problems, the PDE2D user has a choice between a collocation method and a Galerkin method. The collocation method can be used to solve problems in a wide range of simple 2D regions, and is virtually identical to the 3D collocation method described in the next two sections (bicubic Hermite basis functions are used instead of tricubic Hermites) and thus will not be further discussed here.

The Galerkin algorithm solves nonlinear, steady-state systems in the “divergence” form:

$$\begin{aligned} \frac{\partial A_1}{\partial x} + \frac{\partial B_1}{\partial y} &= F_1, \\ &= \\ &= \\ \frac{\partial A_M}{\partial x} + \frac{\partial B_M}{\partial y} &= F_M, \end{aligned}$$

where the A_i , B_i , and F_i are functions of x , y , $U_1, \dots, U_M, U_{1x}, \dots, U_{Mx}, U_{1y}, \dots, U_{My}$, in general two-dimensional regions. The F_i may contain Dirac delta functions.

Boundary conditions have the form:

$$\begin{aligned}
U_1 &= FB_1(x, y), \\
&= \\
&= \\
UM &= FB_M(x, y),
\end{aligned}$$

or

$$\begin{aligned}
A_1 N_x + B_1 N_y &= GB_1(x, y, U_1, \dots, UM, U_{1x}, \dots, UM_x, U_{1y}, \dots, UM_y), \\
&= \\
&= \\
A_M N_x + B_M N_y &= GB_M(x, y, U_1, \dots, UM, U_{1x}, \dots, UM_x, U_{1y}, \dots, UM_y),
\end{aligned}$$

where (N_x, N_y) is the unit outward normal to the boundary.

Note that for a diffusion equation such as 5.1.1, $A = DU_x$, $B = DU_y$, and so the second type of boundary condition reduces to $DU_x N_x + DU_y N_y = GB$, or $D\partial U/\partial n = GB$. Thus the second, or “natural,” boundary condition solved by PDE2D is just a generalization of the second boundary condition in 5.1.1.

The PDE2D Galerkin finite element method uses piecewise polynomial basis functions of degree 1, 2, 3, or 4 on triangular elements, as outlined in Section 5.5. A detailed description of the elements used is given in Sewell [1985, Section 2.2]. Lagrangian finite elements are used, which means that every basis function is a piecewise polynomial that is one at “its” node and zero at every other node. Elements of degree 1, 2, 3, and 4 have 3, 6, 10, and 15 nodes, respectively, in each triangle. Figure A.4.1 shows the locations of the nodes for each element, and of the integration points used (when IDEG > 0) in the computation of the integrals required by the Galerkin method. (See Table 2.3.1 of Sewell [1985] for a list of the integration points and weights.) As mentioned in Section 5.3, the integration rules must be exact for polynomials of degree $2n - 2$ in order to preserve the order of accuracy normally expected for elements of degree n ; these rules are exact for polynomials of degree 2, 5, 6, and 8, when $n = 1, 2, 3,$ and 4 , and thus are more than accurate enough. The integration rules used to compute the boundary integrals use the boundary nodes as integration points and are exact for polynomials of degrees 1, 3, 5, and 7, respectively.

The “isoparametric” method is used in triangles adjacent to a curved boundary; this means that the nodes on one edge of the triangle are moved out to interpolate the boundary, and a change of coordinates is used to map the nodes of the original (straight) triangle onto the nodes of the curved triangle.

The change of coordinates used involves polynomials of the same degree as the basis functions.

Again, Newton's method is used to solve the nonlinear algebraic equations resulting from the Galerkin method formulation, and there are several options available to solve the linear system, which is assembled "element-by-element" (see Section 5.6), each Newton iteration:

ISOLVE=1: A band solver, with a reverse Cuthill-McKee ordering of the unknowns (described in Section 5.5, and in more detail in Section 3.2 of Sewell [1985]).

ISOLVE=2: An out-of-core band solver, called a frontal method (described in Section 3.3 of Sewell [1985]). This is essentially the same as the band solver, but it requires much less memory because the rows of the band matrix are written to disk after processing during the forward elimination and read back in when needed during the back substitution. Hence only a small portion of the band matrix is held in memory at any given time.

ISOLVE=3: A preconditioned biconjugate-gradient (Lanczos) iterative method (described in Section 3.4 of Sewell [1985]). This is a generalization of the conjugate-gradient method described in Section 4.8, designed to handle nonsymmetric linear systems.

ISOLVE=4: Harwell minimal degree (see Section 0.5) sparse direct solvers MA27 (for symmetric problems [Duff and Reid 1983]) and MA37 (nonsymmetric problems).

When the coefficient matrix is symmetric, all four solvers take advantage of the symmetry to reduce the memory and CPU time requirements (see Problem 1 of Chapter 0).

The linear system solvers account for most of the computer time and memory requirements when a finite element program is used to solve a large PDE problem, so the overall efficiency of the program is highly dependent on the efficiency of its linear system solvers. To compare the time and memory requirements of the PDE2D linear system solvers, the 2D Navier-Stokes equations were solved to compute the fluid flow around a bend in a pipe (see Example 4, Section A.10). The resulting linear system, which must be solved each Newton iteration, is highly nonsymmetric. The tests were run on a Cray J90 (one processor). Note that the time reported in Table A.4.1 is the total PDE2D CPU time (for one Newton iteration), not just the time used by the linear system solvers.

PDE2D has an interface (ISOLVE=5) that makes it very easy for users to "plug in" and test other linear system solvers. Using this interface, the sparse direct nonsymmetric linear system solvers in the well-known math software libraries, IMSL and NAG, were also tested, to compare them with the existing PDE2D solvers.

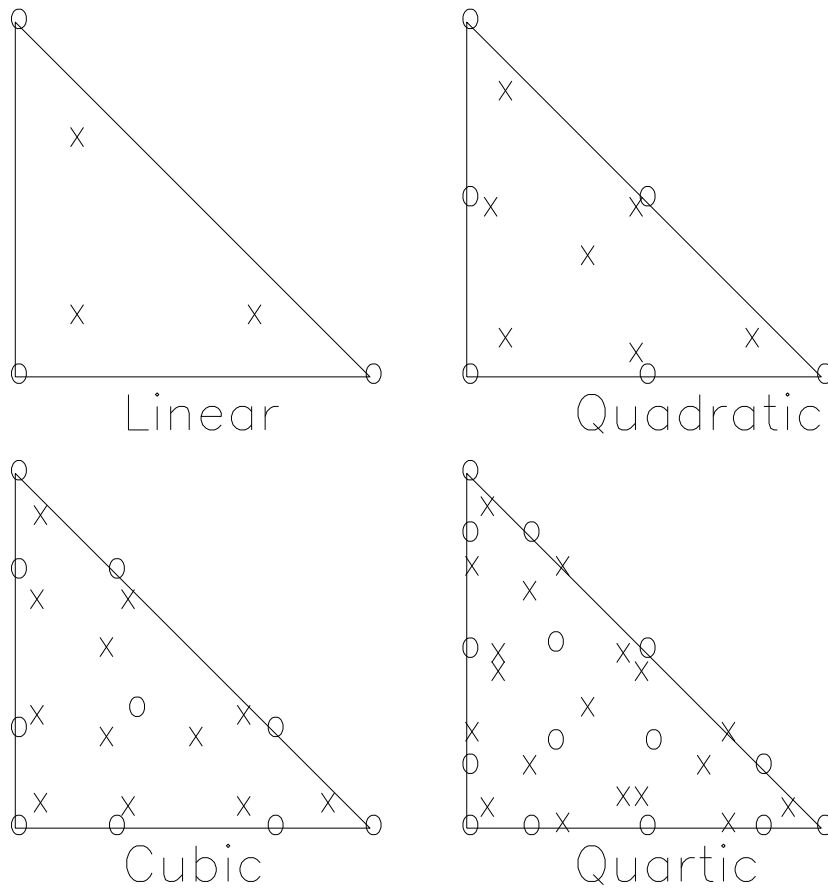


Figure A.4.1
PDE2D's Triangular Elements
 O = nodes, X = integration points

As can be seen from these results, the Harwell routine MA37 ran many times faster than the IMSL and NAG sparse direct solvers, and even the band and frontal solvers did much better than LSLXG and F04AXE. In other tests [Sewell 1993], two other widely used sparse direct nonsymmetric solvers performed equally poorly compared to MA37. In fairness, it should be mentioned that MA37 is primarily designed for the type of sparse systems that are generated by Galerkin finite element methods, while the other sparse solvers are not. Iterative methods generally perform very poorly on highly nonsymmetric and ill-conditioned problems such as this one; in fact, the PDE2D Lanczos iterative solver failed to converge in these tests.

Note that the computation rate (in millions of floating point operations per second) for the band solvers (ISOLVE=1 and 2) is much higher than for the sparse direct solvers. This is because the band solvers spend most of their time adding a multiple of one long row to another, in a DO loop like:

```

DO 10 I=1,L
10 A(I) = A(I) + AMUL*B(I)

```

where L is large, for large problems. A pipeline computer like the Cray J90 can automatically and efficiently “vectorize” this loop; that is, it does the calculations in “assembly line” fashion [Leiss 1995], beginning the calculation of $A(I)$ before finishing the calculation of earlier elements of A . The pipeline (assembly line) consists of a number of units (workers), each of which performs one task on the stream of numbers $A(1), A(2), \dots$ and $B(1), B(2), \dots$ as it passes down the pipeline. For example, one “worker” multiplies the mantissas of $AMUL$ and $B(I)$, another adds their exponents, another renormalizes this product, and so on. If the calculation $A(I) = A(I) + AMUL*B(I)$ is broken down into m tasks, and if each unit performs its task once per clock cycle, then after a few start-up cycles, the pipeline processor updates one element of the vector A every cycle, while a serial processor, which waits until $A(I-1)$ is done before starting on $A(I)$, calculates one answer every m cycles. Thus a vector processor can speed up vectorizable loops like this by a factor of nearly m , when L is large. If the execution of one pass through a loop requires knowledge of results from previous passes (which may not yet be finished), the loop is not “vectorizable,” and the Cray compiler will refuse to vectorize it—the calculations for each loop pass will be finished before the next pass is started down the pipeline.

Table A.4.1
Comparison of Linear System Solvers on 2D Problem

500 fourth degree elements (7840 unknowns, half-bandwidth = 964):					
ISOLVE	Library	Solver	Total CPU time	Megaflops/second	Total Memory
1	PDE2D	Band solver	30 sec	54.1	15.2 Mwords
2	PDE2D	Frontal method	45	36.7	1.1
3	PDE2D	Lanczos iteration	(did not converge)	69.7	0.7
4	PDE2D	MA37	18	9.9	2.3
5	IMSL	LSLXG	634	0.6	9
5	NAG	F04AXE	3198	12.5	8
1500 fourth degree elements (23696 unknowns, half-bandwidth = 1514):					
ISOLVE	Library	Solver	Total CPU time	Megaflops/second	Total Memory
1	PDE2D	Band solver	183 sec	63.1	72 Mwords
2	PDE2D	Frontal method	246	47.1	2.7
3	PDE2D	Lanczos iteration	(did not converge)	72.1	2.3
4	PDE2D	MA37	61	11.6	7.3
3000 fourth degree elements (47584 unknowns, half-bandwidth = 2284):					
ISOLVE	Library	Solver	Total CPU time	Megaflops/second	Total Memory
1	PDE2D	Band solver	(insufficient memory)		218 Mwords
2	PDE2D	Frontal method	783 sec	44.8	6.1
3	PDE2D	Lanczos iteration	(did not converge)	–	4.0
4	PDE2D	MA37	136	13.3	15.8

To measure the speed-up due to vectorization, we can direct the compiler to force all loops to be executed serially and compare the serial and vector codes. When we do this for the 500-triangle problem of Table A.4.1, using the band solver, we see that the serial code runs at only about 5 megaflops/second, so vectorization speeds the program up by a factor of about 11. Thus, although the calculations are not quite done in parallel, for this problem the result of vectorization is much the same as if we were running the code efficiently in parallel on 11 different processors. Clearly it is important, when running on vector computers, to make sure the bulk of the CPU time is spent in long vectorizable loops.

The Harwell sparse direct solvers MA27 and MA37 outperform the other PDE2D options by an even wider margin when tests are run on serial computers, where the fact that these codes do not vectorize as well is not relevant.

To demonstrate the high accuracy available using the high order PDE2D triangular elements, we solve the problem

$$U_{xx} + U_{yy} = 4(1 + x^2 + y^2)U \quad \text{in the unit circle,}$$

$$U = 1 \quad \text{on the boundary.}$$

The relative errors (measured by an integral again) using different degree elements are shown in Table A.4.2, with uniform triangulations of 100, 1000, and 10000 elements. The interpolation error bound 5.5.1, derived for elements of degree $n = 1$, can be generalized (see 2.2.5 in Sewell [1985]) to

$$\|W - u\|_{\infty} \leq C \max_k h_k^{n+1} D_k^{n+1} u, \quad (\text{A.4.1})$$

which suggests the finite element error should be of order $O(h^{n+1})$, where n is the element degree and h is the maximum triangle diameter. In fact, this error estimate holds even in regions with curved boundaries (or curved interior interfaces), when the isoparametric method is used, as with PDE2D. The exact solution is $U = e^{x^2+y^2-1}$.

Table A.4.2
Errors for 2D Problem

Element Degree	Triangles	N	Relative Error	Total CPU Time on Cray J90 (ISOLVE=4)
1	100	41	3.23E-2	0.37 sec
2	100	180	1.49E-3	0.55
3	100	418	1.86E-4	0.91
4	100	755	5.71E-5	1.48
1	1000	469	3.70E-3	2.37
2	1000	1937	6.11E-5	4.51
3	1000	4405	1.73E-6	8.23
4	1000	7873	3.12E-7	14.71
1	10000	4873	2.81E-4	37.55
2	10000	19745	1.71E-6	62.93
3	10000	44617	1.50E-8	104.53
4	10000	79489	2.50E-9	151.06

Although it is misleading to compare the accuracy for elements of different degrees on the same triangulation, note that 1000 quartic triangles gave an accuracy one thousand times better than 10000 linear elements, while taking much less computer time.

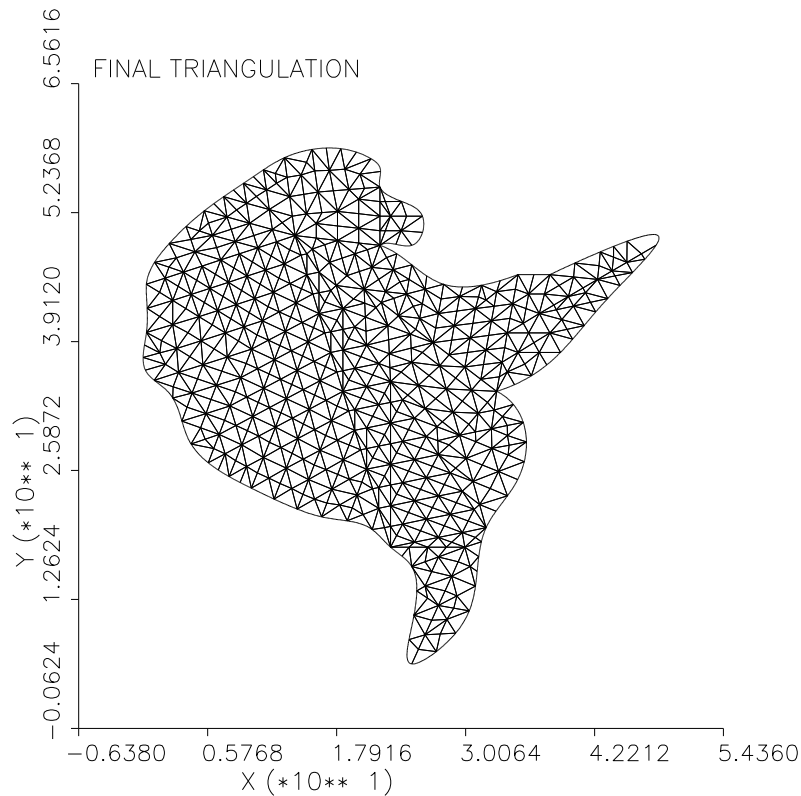


Figure A.4.2
Triangulation of Corpus Christi Bay

The PDE2D user specifies an initial triangulation “by hand,” with just enough triangles to define the region (usually 3 to 10 triangles, for simple regions). Curved boundaries are specified either by their parametric equations or by a set of boundary points, through which PDE2D draws a cubic spline. When the region is a rectangle or a disk, or any other region that can be specified in the form $X = X(P, Q), Y = Y(P, Q)$, with constant limits on P and Q , the user can specify a set of P and Q grid lines, and a (regular) initial triangulation will be generated automatically, with 3 or 4 triangles in each (curved) grid rectangle. The initial triangulation shown in Figure A.10.2 was generated in this way. Then this initial triangulation can be refined automatically until the number of triangles reaches a number specified by the user. By default, the triangles will be approximately of equal size, but a graded triangulation can be generated in one of two ways. The user can

specify a positive function of x and y , which is largest where he/she wants the triangulation to be most dense, or the triangulation grading can be done adaptively. If adaptive grading of the triangulation is requested, the problem is solved once with a (usually uniform) triangulation, and information about the gradient of the solution is saved in a file. The problem is then re-solved; this time the gradient information is read from the file and the triangulation is refined most where the gradients are largest. The first version, 2DEPEP, of this program had one of the first adaptive triangulation schemes [Sewell 1976], which was designed to make the triangulation most dense where the $(n+1)$ st derivatives of the solution were largest ($n =$ element degree), as suggested by A.4.1. However, this algorithm was abandoned in a few years, because of the unreliability of the estimates of the $(n+1)$ st derivatives (after all, the $(n+1)$ st derivatives of a piecewise n th degree polynomial are zero everywhere except at the triangle boundaries, where they are infinite!). The new algorithm used by PDE2D, based on estimates of the first derivatives (regardless of the element degree), is difficult to justify theoretically, but in practice it works much better than the old scheme, because estimates of the first derivatives are much more reliable, and in most problems, the size of the gradients is a good indicator of where the triangulation needs to be most dense. In any case, if the user does not like the adaptively generated triangulation, he/she can control it manually. Typical initial and graded final triangulations are shown in Figures A.10.4 and A.10.5. Figure A.4.2 shows the final triangulation for a complicated region, whose boundary was defined by a set of user-supplied points through which PDE2D has drawn a cubic spline.

A.5 Three-Dimensional Steady-State Problems

PDE2D solves three-dimensional, nonlinear, steady-state systems of the form:

$$\begin{aligned}
 F_1(x, y, z, U_1, \dots, U_{1x}, \dots, U_{1y}, \dots, U_{1z}, \dots, U_{1xx}, \dots, U_{1yy}, \dots, U_{1zz}, \dots, U_{1xy}, \dots, U_{1xz}, \dots, U_{1yz}) &= 0, \\
 &= \\
 &= \\
 F_M(x, y, z, U_1, \dots, U_{1x}, \dots, U_{1y}, \dots, U_{1z}, \dots, U_{1xx}, \dots, U_{1yy}, \dots, U_{1zz}, \dots, U_{1xy}, \dots, U_{1xz}, \dots, U_{1yz}) &= 0,
 \end{aligned}$$

in $x_a \leq x \leq x_b$, $y_a \leq y \leq y_b$, $z_a \leq z \leq z_b$.

Boundary conditions for 3D problems have the form:

$$\begin{aligned}
G_1(x, y, z, U_1, \dots, U_M, U_{1_x}, \dots, U_{M_x}, U_{1_y}, \dots, U_{M_y}, U_{1_z}, \dots, U_{M_z}) &= 0, \\
&= \\
&= \\
G_M(x, y, z, U_1, \dots, U_M, U_{1_x}, \dots, U_{M_x}, U_{1_y}, \dots, U_{M_y}, U_{1_z}, \dots, U_{M_z}) &= 0.
\end{aligned}$$

Periodic boundary conditions are also permitted.

For three-dimensional problems, PDE2D uses a collocation finite element method, with “tricubic Hermite” basis functions. That is, each unknown is assumed to be a linear combination of the 8 NXGRID NYGRID NZGRID basis functions:

$$\begin{aligned}
H_i(x)H_j(y)H_k(z), & \quad H_i(x)H_j(y)S_k(z), \\
H_i(x)S_j(y)H_k(z), & \quad H_i(x)S_j(y)S_k(z), \\
S_i(x)H_j(y)H_k(z), & \quad S_i(x)H_j(y)S_k(z), \\
S_i(x)S_j(y)H_k(z), & \quad S_i(x)S_j(y)S_k(z)
\end{aligned}$$

$$(i = 1, \dots, \text{NXGRID}, j = 1, \dots, \text{NYGRID}, k = 1, \dots, \text{NZGRID}),$$

where the cubic Hermite basis functions H_i and S_i are defined in 5.3.5. This choice of basis function ensures that the first derivatives of the approximate solution are all continuous, as required by the collocation method. The approximate solution is required to satisfy the PDEs exactly at 8 collocation points $(x_i + \beta_{1,2}(x_{i+1} - x_i), y_j + \beta_{1,2}(y_{j+1} - y_j), z_k + \beta_{1,2}(z_{k+1} - z_k))$, where $\beta_1 = 0.5 - 0.5/\sqrt{3}$, $\beta_2 = 0.5 + 0.5/\sqrt{3}$, in each of the $(\text{NXGRID}-1)(\text{NYGRID}-1)(\text{NZGRID}-1)$ subrectangles, and to satisfy the boundary conditions at certain boundary points. The number of boundary collocation points plus the number of interior collocation points is equal to the number of basis functions (8 NXGRID NYGRID NZGRID), so that the number of equations equals the number of unknowns ($N = 8 \text{ NXGRID NYGRID NZGRID M}$).

Although it appears from the above discussion that PDE2D can only solve 3D problems in rectangular “boxes,” we will see in the next section how it can easily solve problems in a wide range of simple 3D regions, such as spheres, cylinders, tori, ellipsoids, parallelepipeds, and many others.

Again, Newton’s method is used to solve the nonlinear algebraic equations resulting from the collocation method formulation, and again there are several options available to solve the linear system each Newton iteration. The linear systems generated by Galerkin finite element methods have symmetric nonzero structures, even when the matrices themselves are nonsymmetric. The systems generated by collocation finite element methods, on the other hand, do not even have symmetric nonzero structures, and all iterative and sparse direct solvers tested on such systems perform

“extremely” poorly; even MA37 does not fare too well (see Table A.5.1). The half-bandwidth of these linear systems is (normally) approximately $L = N/\max(NXGRID, NYGRID, NZGRID)$.

Table A.5.1
Comparison of Linear System Solvers on 3D Problem

NXGRID=NYGRID=NZGRID=8 (4096 unknowns, half-bandwidth = 546):					
ISOLVE	Library	Solver	Total CPU Time	Megaflops/second	Total Memory
(nonsymmetric) original equations					
2	PDE2D	Frontal method	68 sec	37.6	0.7 Mwords
5	Harwell	MA37	327	57.4	20
5	IMSL	LSLXG	11575	0.5	30
5	NAG	F04AXE	33980	17.6	20
(positive-definite) normal equations					
1	PDE2D	MA27	97	12.6	4.5
3	PDE2D	CG iteration	245	4.6	0.8
4	IMSL	LSLXD	215	5.9	20
NXGRID=NYGRID=NZGRID=13 (17576 unknowns, half-bandwidth = 1406):					
ISOLVE	Library	Solver	Total CPU Time	Megaflops/second	Total Memory
(nonsymmetric) original equations					
2	PDE2D	Frontal method	1089 sec	67.3	4.2 Mwords
(positive-definite) normal equations					
1	PDE2D	MA27	979	37.2	32
3	PDE2D	CG iteration	1919	4.4	3.5
4	IMSL	LSLXD	1728	25.6	100

Only the PDE2D frontal method (ISOLVE=2) solves these collocation equations with reasonable efficiency; however, since it is an out-of-core band solver, it becomes slow on very large problems. Thus for 3D problems, PDE2D multiplies both sides of the linear system to be solved, $A\mathbf{x} = \mathbf{b}$, by A^T , yielding the “normal” equations, $A^T A\mathbf{x} = A^T \mathbf{b}$, and solves them using either the Harwell routine MA27 (ISOLVE=1) or a conjugate-gradient iterative method (ISOLVE=3). Since $A^T A$ is always symmetric and positive-definite, the normal equations can be solved efficiently by many sparse direct or iterative methods (cf Sections 4.6 and 4.8). In fact, the normal equations are essentially the equations that would result if a “least squares” finite element method were used (see Problem 10 of Chapter 5), and the nonzero structure of these equations is the same as for a Galerkin method. However, the normal equations are significantly more ill-conditioned than the original equations, and roundoff error can be a concern, if low precision is used.

Thus the options for solving the linear systems are:

ISOLVE=1: Harwell sparse direct solver MA27, applied to the symmetric, positive-definite normal equations, $A^T A\mathbf{x} = A^T \mathbf{b}$.

ISOLVE=2: A frontal method (out-of-core band solver), applied to $A\mathbf{x} = \mathbf{b}$.

ISOLVE=3: A “Jacobi” conjugate-gradient method applied to the normal equations, $A^T \mathbf{Ax} = A^T \mathbf{b}$. This means that the conjugate gradient method (Section 4.8) is applied to the preconditioned equations $D^{-1} A^T \mathbf{Ax} = D^{-1} A^T \mathbf{b}$, where D is the diagonal part of the positive-definite matrix $A^T A$ (see Problem 11 of Chapter 4).

These linear solution options were compared by solving a nonlinear equation, $U_{xx} + U_{yy} + U_{zz} = e^U$, on a Cray J90, with results as reported in Table A.5.1 (times are for one Newton iteration). Some other solvers were also “plugged in” and used to solve the original equations ($\mathbf{Ax} = \mathbf{b}$, ISOLVE=5) or the normal equations ($A^T \mathbf{Ax} = A^T \mathbf{b}$, ISOLVE=4).

The frontal method is still competitive with MA27 on these problems, run on a vector computer, but for still larger problems, and on serial computers, the sparse direct solver is far superior—at least with regard to computer time; MA27 requires a lot of memory compared to the frontal method and conjugate-gradient iterative methods, however.

To measure the accuracy of the tricubic Hermite elements, we solved the problem $U_{xx} + U_{yy} + U_{zz} = 3U$ in the unit sphere with $U = e^{x+y+z}$ on the surface. The exact solution is $U = e^{x+y+z}$, and the errors with different gridsizes are reported in Table A.5.2. The problem is actually solved in a spherical coordinate system (see the next section) and NXGRID, NYGRID, NZGRID are the number of divisions in the ρ , θ , and ϕ directions.

Table A.5.2
Errors for 3D Problem

NXGRID=NYGRID =NZGRID	N	Relative Error	Total CPU Time on Cray T3E (32 processors)
6	1728	1.95E-3	4 sec.
11	10648	1.05E-4	37
16	32768	1.76E-5	343

The experimental rate of convergence obtained by comparing the last two errors is $\log(1.05 \cdot 10^{-4} / 1.76 \cdot 10^{-5}) / \log((16-1)/(11-1)) = 4.4$, which agrees reasonably well with the $O(h^4)$ accuracy expected using cubic polynomials.

A.6 Nonrectangular 3D Regions

In 1994, PDE2D was generalized to handle three-dimensional PDE systems. The logical extension of the 2D algorithm would have been to use a Galerkin method with (possibly isoparametric) tetrahedral elements and require the user to supply an initial “tetrahedralization” of the region, thereby facilitating the solution of problems in general 3D regions. But developing a user interface for defining general regions and boundary conditions is a *much* more difficult problem in three dimensions than in two, for reasons that are obvious and well-known. The decision was made *initially*, therefore, to avoid the difficulties in handling general three-dimensional regions, and to develop software that could

solve 3D PDE systems as general as those solved by the 2D algorithm, with comparable ease-of-use, but only in 3D boxes.

For 3D problems, then, a collocation finite element method was selected, with tricubic Hermite basis functions, because the fact that the Galerkin method is easier to apply to general regions was now not an issue, and the collocation method has some important advantages over the Galerkin brand with regard to ease-of-use. In particular, the user does not have to put his/her equations in the “divergence” form required by the Galerkin method, and so not only the PDEs but the boundary conditions are usually more convenient to formulate. For many scientific applications, the divergence form and “natural” boundary conditions required by the Galerkin method are indeed quite natural; but general PDEs with general boundary conditions (e.g., many equations of mathematical finance, see Topper [2005]) are often difficult, sometimes impossible, to express in the required format.

However, after the abilities to handle periodic and “no” boundary conditions [see Section A.3] were added, it became possible to solve, with high accuracy, $O(h^4)$, problems in many simple nonrectangular domains, such as spheres, cylinders, tori, pyramids, ellipsoids, and cones, by writing the PDEs in terms of an appropriate system of variables with constant limits. However, rewriting the partial differential equations in the new coordinate system was often extremely unpleasant. For example, suppose we want to solve $\nabla^2 U = 1$, with $U = 0$ on the boundary, in a torus of major radius R_0 and minor radius R_1 . A “toroidal” coordinate system can be used, where

$$\begin{aligned} X &= (R_0 + P3 * \cos(P2))\cos(P1), \\ Y &= (R_0 + P3 * \cos(P2))\sin(P1), \\ Z &= P3 * \sin(P2). \end{aligned} \tag{A.6.1}$$

Here X, Y , and Z are Cartesian coordinates, $P1$ is the major (toroidal) angle, $P2$ is the minor (polodial) angle, and $P3$ is radial distance from the torus centerline. In the new coordinate system the region is rectangular, because the limits on $P1, P2$, and $P3$ are constants, and PDE2D can be used to solve this problem, with periodic boundary conditions at $P1 = 0, 2\pi$ and also at $P2 = 0, 2\pi$, “no” boundary conditions at $P3 = 0$ (because there is no boundary!), and $U = 0$ at $P3 = R_1$. To convert the Laplacian, $U_{xx} + U_{yy} + U_{zz}$, to the new coordinate system, one has to use the chain rule; for example, U_{xx} , the second derivative of U with respect to X , is ($U_i = \frac{\partial U}{\partial P_i}$, etc.):

$$\begin{aligned}
U_{xx} &= \left(U_{11} \frac{\partial P1}{\partial X} + U_{12} \frac{\partial P2}{\partial X} + U_{13} \frac{\partial P3}{\partial X} \right) \frac{\partial P1}{\partial X} \\
&+ \left(U_{21} \frac{\partial P1}{\partial X} + U_{22} \frac{\partial P2}{\partial X} + U_{23} \frac{\partial P3}{\partial X} \right) \frac{\partial P2}{\partial X} \\
&+ \left(U_{31} \frac{\partial P1}{\partial X} + U_{32} \frac{\partial P2}{\partial X} + U_{33} \frac{\partial P3}{\partial X} \right) \frac{\partial P3}{\partial X} \\
&+ U_1 \frac{\partial^2 P1}{\partial X^2} + U_2 \frac{\partial^2 P2}{\partial X^2} + U_3 \frac{\partial^2 P3}{\partial X^2}.
\end{aligned}$$

The bad news is below, where $\frac{\partial^2 P3}{\partial X^2}$ is displayed for the toroidal coordinate transformation, as computed by *Mathematica*[®]. Transforming PDEs to a new coordinate system by hand can be quite a task!

$$\begin{aligned}
\frac{\partial^2 P3}{\partial X^2} &= - \frac{X^2(-R_0 + \sqrt{X^2 + Y^2})^2}{(X^2 + Y^2)((-R_0 + \sqrt{X^2 + Y^2})^2 + Z^2)^{3/2}} \\
&+ \frac{X^2}{(X^2 + Y^2)\sqrt{(-R_0 + \sqrt{X^2 + Y^2})^2 + Z^2}} \\
&- \frac{X^2(-R_0 + \sqrt{X^2 + Y^2})}{(X^2 + Y^2)^{3/2}\sqrt{(-R_0 + \sqrt{X^2 + Y^2})^2 + Z^2}} \\
&+ \frac{-R_0 + \sqrt{X^2 + Y^2}}{\sqrt{X^2 + Y^2}\sqrt{(-R_0 + \sqrt{X^2 + Y^2})^2 + Z^2}}.
\end{aligned}$$

Now most of the pain has been removed from this process: now the user only has to supply the global coordinate transformation equations (e.g., A.6.1) and can then write his/her PDEs in their usual Cartesian form. For example, the PDE above would be written simply as $U_{xx} + U_{yy} + U_{zz} = 1$; PDE2D will automatically compute $U_{xx} + U_{yy} + U_{zz}$ in terms of U_{11}, U_{12}, \dots using the chain rule, and solve the problem internally in the $P1, P2, P3$ coordinate system.

If a cylindrical or spherical coordinate system is used, PDE2D automatically supplies the first and second derivatives of $P1, P2, P3$ with respect to X, Y, Z , required to apply the chain rule; the user only has to indicate that $P1, P2, P3$ represent cylindrical or spherical coordinates. If another user-specified system is used, such as toroidal coordinates, PDE2D will use finite differences to compute the first and second derivatives of X, Y, Z with respect to $P1, P2, P3$; alternatively, the user can supply these analytically. Then PDE2D computes the required derivatives of $P1, P2, P3$ with respect to X, Y, Z from these. For the first derivatives, the conversion uses the fact

that the Jacobian matrices for the forward and inverse transforms are inverses:

$$J \equiv \begin{bmatrix} \frac{\partial P1}{\partial X} & \frac{\partial P1}{\partial Y} & \frac{\partial P1}{\partial Z} \\ \frac{\partial P2}{\partial X} & \frac{\partial P2}{\partial Y} & \frac{\partial P2}{\partial Z} \\ \frac{\partial P3}{\partial X} & \frac{\partial P3}{\partial Y} & \frac{\partial P3}{\partial Z} \end{bmatrix} = \begin{bmatrix} \frac{\partial X}{\partial P1} & \frac{\partial X}{\partial P2} & \frac{\partial X}{\partial P3} \\ \frac{\partial Y}{\partial P1} & \frac{\partial Y}{\partial P2} & \frac{\partial Y}{\partial P3} \\ \frac{\partial Z}{\partial P1} & \frac{\partial Z}{\partial P2} & \frac{\partial Z}{\partial P3} \end{bmatrix}^{-1}$$

The second derivatives are calculated using ($P_i = P1, P2, P3$):

$$P_{iH} = -J^T \left[\frac{\partial P_i}{\partial X} X_H + \frac{\partial P_i}{\partial Y} Y_H + \frac{\partial P_i}{\partial Z} Z_H \right] J$$

where the subscript H denotes the Hessian matrix. (This formula can be derived directly from the chain rule.)

Implementing the coordinate transformation did not involve any internal modifications to the PDE2D library routines, only to the function subprograms where the PDE coefficients and boundary condition coefficients are defined by the user. These functions are called by PDE2D with various values of $P1, P2, P3, U, U_1, U_2, U_3, U_{11}, \dots$; all that had to be done was to insert code to compute $X, Y, Z, U_x, U_y, U_z, U_{xx}, \dots$ for given $P1, P2, P3, U_1, U_2, U_3, U_{11}, \dots$, using the chain rule. Then the user can simply define his/her PDE and boundary condition coefficients in terms of $X, Y, Z, U, U_x, U_y, U_z, U_{xx}, \dots$, though he/she can still use the non-Cartesian variables and derivatives as well, if desired.

As an example, an elasticity problem was solved in a region that is half a torus, of major radius $R_0 = 5$ and minor radius $R_1 = 4$, with a smaller torus, of minor radius $0.5R_1$, removed (Figure A.6.1).

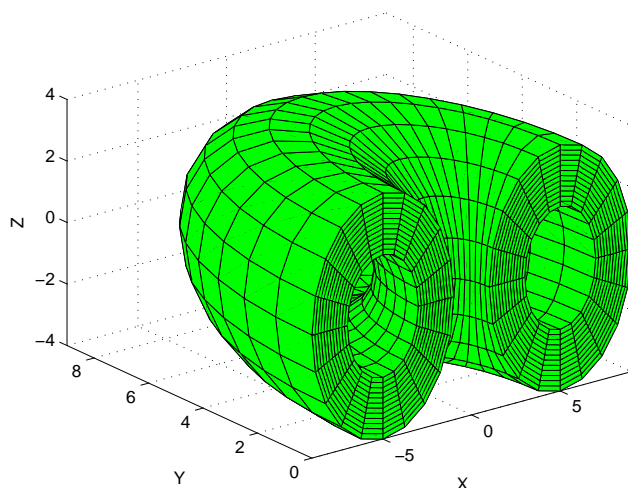


Figure A.6.1
Part of a Torus

The partial differential equations for the elastic body are

$$\begin{aligned} AU_{xx} + BV_{yx} + BW_{zx} + C(U_{yy} + V_{xy}) + C(U_{zz} + W_{xz}) &= 0, \\ C(U_{yx} + V_{xx}) + AV_{yy} + BU_{xy} + BW_{zy} + C(V_{zz} + W_{yz}) &= 0, \\ C(U_{zx} + W_{xx}) + C(V_{zy} + W_{yy}) + AW_{zz} + BU_{xz} + BV_{yz} &= 0, \end{aligned}$$

where (U, V, W) is the displacement vector, $A = 2.963$, $B = 1.460$, and $C = 0.752$ are constants (involving the elastic modulus and Poisson ratio).

There are periodic boundary conditions on $P2$, and at $P3 = 0.5R_1$ the displacements are set to zero. On the outer surface of the torus, $P3 = R_1$, there is a unit inward boundary force; that is, the boundary force vector is $-(Nx, Ny, Nz)$, where (Nx, Ny, Nz) is the unit outward normal to the boundary, in Cartesian coordinates. This means the boundary conditions are:

$$\begin{aligned} (AU_x + BV_y + BW_z)Nx + C(U_y + V_x)Ny + C(U_z + W_x)Nz + Nx &= 0 \\ C(U_y + V_x)Nx + (AV_y + BU_x + BW_z)Ny + C(V_z + W_y)Nz + Ny &= 0 \\ C(U_z + W_x)Nx + C(V_z + W_y)Ny + (AW_z + BU_x + BV_y)Nz + Nz &= 0 \end{aligned}$$

Continue

Fortran expressions, up to 65 characters

F1	<input type="text" value="A*Uxx + B*Vyx + B*Wzx + C*(Uyy+Vxy) + C*(Uzz+Wxz)"/>
F2	<input type="text" value="C*(Uyx+Vxx) + A*Vyy + B*Uxy + B*Wzy + C*(Vzz+Wyz)"/>
F3	<input type="text" value="C*(Uzx+Wxx) + C*(Vzy+Wyy) + A*Wzz + B*Uxz + B*Vyz"/>
F4	<input type="text"/>
F5	<input type="text"/>
F6	<input type="text"/>
F7	<input type="text"/>
F8	<input type="text"/>

Figure A.6.2
Page of PDE2D GUI Session (Torus Problem)

On one flat end ($P1 = 0$), there are zero displacements, and on the other ($P1 = \pi$), there are zero boundary forces.

Once the coordinate transformation is defined (A.6.1), the user does not need to convert his/her PDEs or boundary conditions into the new coordinate system—saving tremendous human effort. The PDEs and boundary conditions are input exactly as shown above (see Figure A.6.2), in a GUI session. Volume integrals and boundary integrals can also be written using Cartesian coordinates. The unit outward normal vector in Cartesian coordinates is available to the boundary condition functions and boundary integrals. In short, while PDE2D internally solves the problem in the new $P1, P2, P3$ coordinate system, the user can supply *everything* in Cartesian coordinate form.

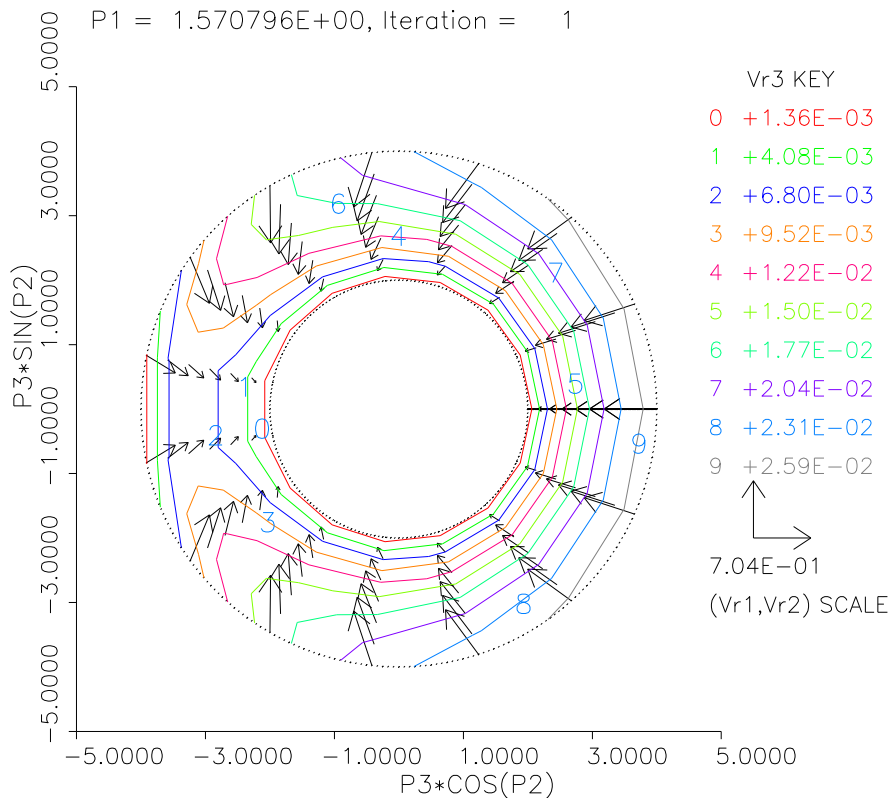


Figure A.6.3
Displacement Field at Cross-section Midway Between Ends

Cross-sectional contour plots of scalar variables, and cross-sectional vector field plots, can be made that reflect the true geometry of the cross section. For example, Figure A.6.3 shows the displacement field (U, V, W) at a cross section midway between the two flat ends, $P1 = \frac{\pi}{2}$, which is plotted using axes $P3 * \cos(P2)$ vs $P3 * \sin(P2)$, rather than $P2$ vs $P3$, so that the cross section looks like an annulus, as it should, rather than a rectangle. If the MATLAB m-file generated automatically by PDE2D is run, it produces 3D cross-sectional plots, with values coded by color, which give an even better picture of the 3D

solution. Figure A.6.4 shows a plot of the vertical displacement W at a $P3 =$ constant cross-section.

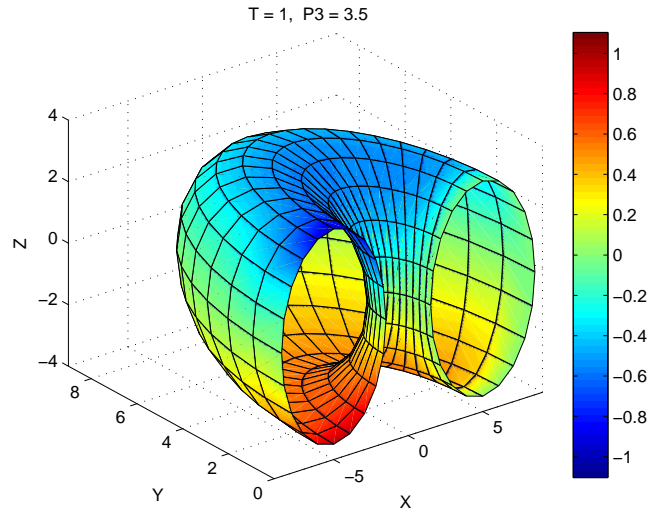


Figure A.6.4
Vertical Displacement (W), at $P3=$ constant Cross Section

When $NP1GRID = NP2GRID = 12$, $NP3GRID = 8$, the computed integral of $Ux + Vy + Wz$ (total volume change) is -229.821 . The computed value of the boundary integral of $U Nx + V Ny + W Nz$ is -229.785 ; by the divergence theorem this should equal the volume change, and in fact they differ by only 0.02%.

As a second example, the first eigenvalue of the Laplacian was computed in a cylinder with a hemispherical cap, with $U = 0$ on the surface. Here the transformation equations are

$$\begin{aligned} X &= R(P3) * P2 * \cos(P1), \\ Y &= R(P3) * P2 * \sin(P1), \\ Z &= P3, \end{aligned}$$

where $R(P3) = 1$ when $P3 \leq 0$ and $R(P3) = \sqrt{1 - P3^2}$ when $P3 \geq 0$.

Variable	Lower Limit	BC	Upper Limit	BC
P1	0	Periodic	2π	Periodic
P2	0	None	1	$U=0$
P3	-1	$U=0$	1	None

Figure A.6.5 shows the first eigenfunction at the cross section $P1 = 0$. With $NP1GRID = 5$, $NP2GRID = NP3GRID = 25$, a first eigenvalue of

-8.9294195 was calculated, which is correct to six significant figures (exact value = -8.9294183). Note that high accuracy is achieved despite the discontinuity in some of the second derivatives, for example, $\frac{\partial^2 X}{\partial P_3^2}$. Continuity of the *first* derivatives is essential, however; for example, if the hemispherical cap is replaced by a cone, the solution will not converge.

PDE2D still cannot solve problems in complicated 3D regions, with many boundary pieces. For *simple* 3D regions, however, the coordinate transformation described here offers significant advantages over that used by other FEM software designed to handle more general 3D regions, particularly with regard to ease of use. Once the user has supplied the transformation equations, the rest of the problem description is as simple as if the region were rectangular. Furthermore, this approach produces $O(h^4)$ accuracy even in regions with curved boundaries; without a global transformation, comparable accuracy can only be obtained if third-order *isoparametric* elements are used, and the use of high-order isoparametric elements involves much more development effort than the global transformation approach.

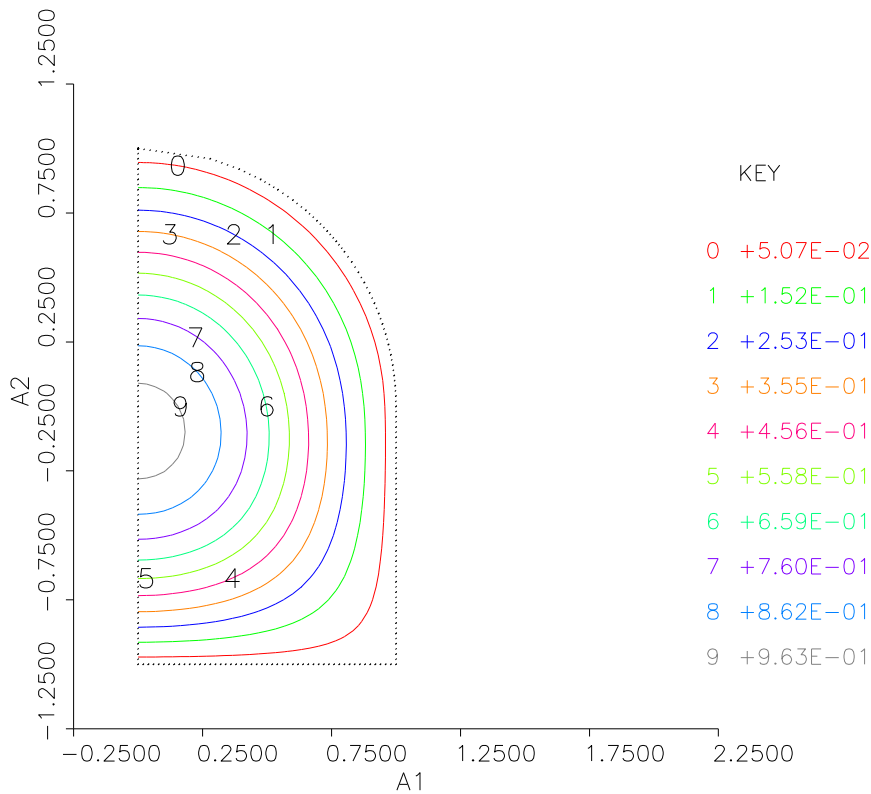


Figure A.6.5
Cylinder with Hemispherical Cap

A.7 Time-Dependent Problems

The time-dependent systems solved by PDE2D have the forms:

(0D–ODEs)

$$\begin{aligned} C_{11} \frac{\partial U_1}{\partial t} + \dots + C_{1M} \frac{\partial UM}{\partial t} &= F_1(t, U_1, \dots, UM), \\ &= \\ &= \\ C_{M1} \frac{\partial U_1}{\partial t} + \dots + C_{MM} \frac{\partial UM}{\partial t} &= F_M(t, U_1, \dots, UM). \end{aligned}$$

The C_{ij} are also functions of t, U_1, \dots, UM .

(1D–collocation option)

$$\begin{aligned} C_{11} \frac{\partial U_1}{\partial t} + \dots + C_{1M} \frac{\partial UM}{\partial t} &= F_1, \\ &= \\ &= \\ C_{M1} \frac{\partial U_1}{\partial t} + \dots + C_{MM} \frac{\partial UM}{\partial t} &= F_M. \end{aligned}$$

All coefficients are functions of $x, t, U_1, \dots, UM, U_{1x}, \dots, U_{Mx}, U_{1xx}, \dots, U_{Mxx}$.

(2D–Galerkin option)

$$\begin{aligned} C_{11} \frac{\partial U_1}{\partial t} + \dots + C_{1M} \frac{\partial UM}{\partial t} &= \frac{\partial A_1}{\partial x} + \frac{\partial B_1}{\partial y} - F_1, \\ &= \\ &= \\ C_{M1} \frac{\partial U_1}{\partial t} + \dots + C_{MM} \frac{\partial UM}{\partial t} &= \frac{\partial A_M}{\partial x} + \frac{\partial B_M}{\partial y} - F_M. \end{aligned}$$

All coefficients are functions of $x, y, t, U_1, \dots, UM, U_{1x}, \dots, U_{Mx}, U_{1y}, \dots, U_{My}$.

(3D)

$$\begin{aligned} C_{11} \frac{\partial U_1}{\partial t} + \dots + C_{1M} \frac{\partial UM}{\partial t} &= F_1, \\ &= \\ &= \\ C_{M1} \frac{\partial U_1}{\partial t} + \dots + C_{MM} \frac{\partial UM}{\partial t} &= F_M. \end{aligned}$$

All coefficients are functions of $x, y, z, t, U_1, \dots, U_{1_x}, \dots, U_{1_y}, \dots, U_{1_z}, \dots, U_{1_{xx}}, \dots, U_{1_{yy}}, \dots, U_{1_{zz}}, \dots, U_{1_{xy}}, \dots, U_{1_{xz}}, \dots, U_{1_{yz}}, \dots$

The boundary conditions are the same as for the steady-state problems, except that coefficients may be functions of t also. The time-dependent system also has initial conditions $U_i(x, \dots, t_0) = U_{i_0}(x, \dots)$.

Time-dependent problems are solved in much the same way, regardless of the spatial dimension. In all cases, a solution of the form 5.7.3 is assumed; that is, the coefficients of the basis functions are now taken to be functions of time; only the basis functions change when the dimension of the problem changes. When the Galerkin method is used, the expansion 5.7.3 is plugged into each PDE, multiplied by a basis function Φ_k , and integrated. When the collocation method is used, the expansion 5.7.3 is plugged into each PDE and required to satisfy these PDEs exactly at the collocation points. In either case, an ordinary differential equation system results (of the form 5.7.6 in the linear case), for the coefficients $\mathbf{a}_i(t)$ in the expansion 5.7.3. The initial values $\mathbf{a}_i(t_0)$ are found by interpolating the initial conditions. For the collocation method, least squares approximation to the initial conditions is also available, since interpolating the derivatives of the cubic Hermites can result in large spikes when the initial conditions are not smooth.

This stiff ODE system is solved using either the first-order backward difference, or backward Euler, method (Table 1.5.1, $m = 1$) or the second-order Adams-Moulton method (Table 1.4.1, $m = 1$), also known in this context as the Crank-Nicolson method. Both methods are implicit, so the solution of a large (generally nonlinear) system is required each time step. Newton's method is used to solve this nonlinear system, but only one iteration is done, because a very good initial guess is available—the solution on the previous step. In Problem 4.3 of Sewell [1985] it is shown that one iteration is sufficient to preserve the order of convergence. The linear system that must be solved each step has the same nonzero structure as in the steady-state case, and the same linear system solvers are available.

The user can either specify a constant, user-chosen, stepsize dt , or request adaptive stepsize control. If adaptive stepsize control is chosen, then each time step, two steps of size $dt/2$ are taken, and that solution is compared with the result when one step of size dt is taken. If the difference between the two answers is less than a user-supplied tolerance (for each variable), the time step dt is accepted (and the next step dt is doubled, if the agreement is “too” good); otherwise dt is halved and the process is repeated. When the step is accepted, an extrapolation is done using the answer obtained using one step of size $dt(U_1(t_{n+1}))$ and two steps of size $\frac{dt}{2}(U_2(t_{n+1}))$. When the backward Euler method is used to compute both answers, the extrapolated value is $U(t_{n+1}) = 2U_2(t_{n+1}) - U_1(t_{n+1})$, which increases the order of the truncation error of the backward Euler method from $O(dt)$ to $O(dt^2)$ while preserving its stability on stiff systems. When the Crank-Nicolson method is used, the extrapolated value is $U(t_{n+1}) = \frac{1}{2}U_2(t_{n+1}) + \frac{1}{2}U_1(t_{n+1})$, which

preserves the $O(dt^2)$ truncation error of the Crank-Nicolson method, while greatly improving its stability on stiff systems (see Problem 8). The extrapolation $U(t_{n+1}) = \frac{4}{3}U_2(t_{n+1}) - \frac{1}{3}U_1(t_{n+1})$ would increase the truncation error order but is not used because it produces an unstable method (except for the “0D” problem, where it is used).

The PDE2D adaptive algorithm is similar to that used in the RKF program of Figure 1.6.1, except that with the RKF procedure, two different ODE solvers are used and compared each step; PDE2D uses the same method with two different step sizes (dt and $\frac{dt}{2}$).

If a constant stepsize is chosen, and if the problem is linear and all PDE and boundary condition coefficients are independent of time (except possibly nonhomogeneous terms), then the linear system that must be solved has exactly the same coefficient matrix every time step. In this case, the LU decomposition (Section 0.3) computed on the first time step can be used to solve each subsequent linear system much more rapidly, and this is taken advantage of by each of the PDE2D direct linear system solvers. In the time-dependent problem 5.7.1, for example, if $c, D, a,$ and p are not functions of time (even if $f, r,$ and q are), then this problem can be solved very efficiently, with a constant time step.

PDE2D has a dump/restart option (also useful for nonlinear steady-state problems) that makes it easy to stop at some value of t , adjust the grid (or even the boundary) or “constant” time step appropriately, and restart. For 2D problems, the grid can be made to automatically and adaptively move with the solution, by putting a DO loop around the main program to vary the initial and final time values each pass, and requesting a restart with an adaptively determined grid. Communication between PDE2D programs is also automated; for example, it is easy to take the solution of a steady-state problem and use it as initial values for a time-dependent problem.

A.8 Eigenvalue Problems

The eigenvalue problems solved by PDE2D have the forms:

(1D—Collocation option)

$$\begin{aligned} F_1 &= \lambda[P_{11}(x)U_1 + \dots + P_{1M}(x)UM], \\ &= \\ &= \\ F_M &= \lambda[P_{M1}(x)U_1 + \dots + P_{MM}(x)UM]. \end{aligned}$$

The F_i are functions of $x, U_1, \dots, UM, U_{1x}, \dots, U_{Mx}, U_{1xx}, \dots, U_{Mxx}$.

(2D—Galerkin option)

$$\begin{aligned} \frac{\partial A_1}{\partial x} + \frac{\partial B_1}{\partial y} &= F_1 + \lambda[P_{11}(x, y)U1 + \dots + P_{1M}(x, y)UM], \\ &= \\ &= \\ \frac{\partial A_M}{\partial x} + \frac{\partial B_M}{\partial y} &= F_M + \lambda[P_{M1}(x, y)U1 + \dots + P_{MM}(x, y)UM]. \end{aligned}$$

The A_i, B_i, F_i are functions of $x, y, U1, \dots, UM, U1_x, \dots, UM_x, U1_y, \dots, UM_y$.

(3D)

$$\begin{aligned} F_1 &= \lambda[P_{11}(x, y, z)U1 + \dots + P_{1M}(x, y, z)UM], \\ &= \\ &= \\ F_M &= \lambda[P_{M1}(x, y, z)U1 + \dots + P_{MM}(x, y, z)UM]. \end{aligned}$$

The F_i are functions of $x, y, z, U1, \dots, U1_x, \dots, U1_y, \dots, U1_z, \dots, U1_{xx}, \dots, U1_{yy}, \dots, U1_{zz}, \dots, U1_{xy}, \dots, U1_{xz}, \dots, U1_{yz}, \dots$

The eigenvalue problem must be linear and homogeneous, however.

The boundary conditions are the same as for the steady-state problems, except that they must also be linear and homogeneous. Eigenvalue problems are also handled in much the same way regardless of the dimension of the problem; only the basis functions are different. In all cases, the eigenfunction is assumed to be a linear combination 5.10.3 of the basis functions. When the Galerkin method is used, the expansion 5.10.3 is plugged into each eigenvalue PDE, multiplied by a basis function Φ_k , and integrated. When the collocation method is used, the expansion 5.10.3 is plugged into each eigenvalue PDE and is required to satisfy it exactly at the collocation points. In either case, a generalized matrix eigenvalue problem of the form 5.10.4 results, for the coefficients a_i in the expansion 5.10.3. In every case, the generalized matrix eigenvalue problem is solved using the shifted inverse power method 4.11.6, which finds the eigenvalue closest to a user-chosen number, α , and the corresponding eigenfunction. Each iteration of the inverse power method requires the solution of the linear system 4.11.6, which has the same nonzero structure as in the steady-state and time-dependent cases, and can be solved using the same options. The eigenvector is renormalized each iteration to prevent overflow or underflow. The coefficient matrix of the linear system 4.11.6 does not change each iteration, and all the direct linear system solvers used by PDE2D take advantage of this fact, forming the LU decomposition the first iteration and using it on subsequent iterations. Though the user can specify the initial values for the shifted inverse power iteration, by default they are chosen using

a random number generator, which virtually eliminates the slim possibility of an “unlucky” initial vector choice (i.e., one that makes $P^{-1}FP\mathbf{u}_0 = \mathbf{0}$ in 4.11.5).

For 0D,1D,2D and 3D eigenvalue problems, PDE2D also offers the option to compute all eigenvalues, including complex eigenvalues. The generalized discrete eigenvalue problem $A\mathbf{a} = \lambda B\mathbf{a}$ is converted to a standard eigenvalue problem $(A - pB)^{-1}B\mathbf{a} = \frac{1}{\lambda - p}\mathbf{a}$, where p is not an eigenvalue, so $(A - pB)^{-1}$ exists, and the eigenvalues of $(A - pB)^{-1}B$ are found using the EISPACK [Smith et al., 1974] implementation of the shifted QR method. $(A - pB)^{-1}B$ is unfortunately a full matrix, but the computations are done efficiently in parallel, on multiple processor machines.

If A is symmetric and $B = LL^T$ is positive definite, $A\mathbf{a} = \lambda B\mathbf{a}$ can be written as $L^{-1}AL^{-T}\mathbf{b} = \lambda\mathbf{b}$, where (see 4.10.5) $\mathbf{b} = L^T\mathbf{a}$, and the matrix $H = L^{-1}AL^{-T}$ will be symmetric. H will generally be full even when A and B are banded, but if B is positive and *diagonal*, $L = L^T = B^{\frac{1}{2}}$ and $H = B^{-\frac{1}{2}}AB^{-\frac{1}{2}}$ will be symmetric and banded. Now the EISPACK routine BANDR can be used, which takes advantage of the band structure of this matrix, resulting in a dramatic savings in computer time and memory. For the collocation method, unfortunately, A is never symmetric, so this savings is only possible for 1D and 2D symmetric problems using the Galerkin method. Further, the requirement that B must be diagonal means that if $M > 1$, the P matrix must be diagonal, and that the integration points be the same as the nodes, which is the case for all 1D Galerkin problems, but for 2D Galerkin problems only if $IDEG = -1$ or -3 (for triangular elements of degrees 2 and 4, if the nodes are used as integration points, some weights will be negative or zero). For such problems, however, the decrease in computer time is spectacular. For example, in one 2D symmetric problem with 3493 unknowns, the time was cut by a factor of 15, while in 1D problems the time and memory are often decreased by much greater factors, since then the matrices have very small bandwidths.

A.9 The PDE2D Parallel Linear System Solvers

Even the sparse direct and iterative linear system solvers employed by PDE2D require prohibitively large amounts of computer time and memory when very large two- and (especially) three-dimensional problems are solved, so PDE2D provides an additional option to solve linear systems efficiently on multiprocessor machines, such as the Cray T3D and T3E series. In theory, such machines can achieve extremely high computational rates, but taking advantage of the parallel processing abilities of such computers generally requires substantial programming effort. A multiprocessor system can be thought of as consisting of several autonomous computers, each with its own memory (distributed memory system), or at least its own section of memory (shared memory systems), with the ability to pass data back and forth between com-

puters. Although the communication speed between these “computers” (processors) is high, it is very slow compared to the speed with which the data are processed internally within a processor, so it is absolutely critical to minimize the amount of “message passing” between processors. A computation that is ideally suited for such a multiprocessor machine would be one where each processor does hours of calculations and computes one number each, then, at the end, the processors simply add their results together. Unfortunately, most algorithms require more communication between processors!

There are a number of libraries that provide routines that can be called by users’ programs to pass messages back and forth between processors. PDE2D uses MPI library routines [Pacheco 1996; Bisseling 2004]; since this is the most widely used set of message passing routines, this ensures that PDE2D can be run on many different distributed and shared-memory multiprocessor systems.

PDE2D provides an MPI-based parallel band solver (ISOLVE=6), which can be used to solve the linear systems generated by 2D or 3D problems. This band solver is similar to others (cf. Figure 0.4.4); the primary difference is that the columns of the band matrix are distributed cyclically over the available processors. That is, column 1 is stored only in the first processor’s memory, column 2 is stored in the second processor’s memory, and so on until we run out of processors, then the next column is stored again in the first processor’s memory. Of course, as with any band solver, only the portion of a column that is nonzero or may fill-in during elimination is actually stored in memory; the elements outside this band are understood to be zero and never referenced.

Figure A.9.1 illustrates how the forward elimination proceeds when the matrix is distributed over the processors in this way, for a linear system with $N = 24$ unknowns, with a half-bandwidth of $L = 7$, when we have $NPES = 3$ processors. After the first 4 columns have been zeroed, the “active” column is column 5, held in processor #2’s memory. Now we need to switch row 5 with the row corresponding to the largest element in the active column, and then take a multiple $(-A_{65}/A_{55})$ of the fifth row and add it to the sixth row, another multiple $(-A_{75}/A_{55})$ of the fifth row and add it to the seventh row, and so on. Processor #3 can do its share of these row operations, to “its” columns 6, 9, 12, 15, and 18, but it has to see the active column before it can know which row to switch with row 5, and what multiples of row 5 to add to rows 6 through 12. So processor #2 has to “broadcast” the active column to the other processors.

when the bandwidth is large compared to the number of processing elements (NPES), each processor will have approximately the same number of columns to work on, and the total work should be decreased by approximately a factor of NPES (in theory, at least!). Notice that the amount of memory required per processor is also decreased by a factor of about NPES, because the matrix is distributed nearly evenly over all the processors' memories. After the forward elimination, the solution is found by back substitution; this step is also parallelized.

The 2D and 3D iterative solvers (ISOLVE=3) have also been "MPI-enhanced", that is, they also distribute the nonzeros of the matrix over the available processors. The most time-consuming calculation for these conjugate gradient-type iterative methods involves repeatedly multiplying a (sparse) matrix times a vector. This computation is relatively easy to distribute over the processors, each multiplying the vector by its part of the matrix. Then the different portions of this matrix-vector product, computed on the different processors, are summed together at the end (i.e., $A\mathbf{p} = A_1\mathbf{p} + \dots + A_{\text{NPES}}\mathbf{p}$, where A_i is the portion of matrix A that is stored on processor i). These solvers also distribute the other conjugate-gradient vector operations 4.8.1 over the available processors.

To measure the performance of the MPI-based parallel band solver and the MPI-enhanced iterative methods, a 2D nonsymmetric plate bending problem ($U_{xx} + U_{yy} = V, V_{xx} + V_{yy} = \delta(x, y)$ in the unit disk) and the 3D problem $U_{xx} + U_{yy} + U_{zz} = 3U$ in a cube were solved on a Cray-Dell Xeon cluster at the Texas Advanced Computing Center in Austin, and also on a Cray T3E, a distributed-memory multiprocessor system at the Pittsburgh Supercomputing Center. The results for the 2D problem (Galerkin option) are given in Table A.9.1, and Table A.9.2 displays the results for the 3D problem.

From these results, it can be seen that, for the 2D problems, the parallel performance of the parallel solvers (ISOLVE=3 and 6) is not too impressive, and the sparse direct method (ISOLVE=4) is still faster than the parallel solvers, until it runs out of memory. For 3D problems, the MPI band solver and the MPI-enhanced Jacobi conjugate-gradient solver far outperform the other methods; however, and it is 3D problems where speed is most critical. When it converges, the MPI-enhanced iterative solver is often faster than the MPI band solver, but for ill-conditioned problems the iterative solver will

Table A.9.1
Performance of Parallel Solvers on 2D Problem

1500 quartic elements (23746 unknowns, half-bandwidth = 1842):				
Solver (ISOLVE)	Number of Processors	Elapsed sec. Cray-Dell	Elapsed sec. Cray T3E	Per-Processor Memory (MW)
Band solver (1)	1	38	memory	87.0
Frontal method (2)	1	62	675	3.9
Lanczos iteration (3)	1	9	142	4.7
	4	9	87	2.7
	16		65	1.2
	64		59	0.8
MA37 (4)	1	3	16	6.9
MUMPS (5)	1	NA	memory	
	4	NA	11	
	16	NA	7	
MPI band solver (6)	1	60	memory	135.0
	4	19	memory	35.0
	16	20	72	9.1
	64		59	2.6
3000 quartic elements (47490 unknowns, half-bandwidth = 2366):				
Solver (ISOLVE)	Number of Processors	Elapsed sec. Cray-Dell	Elapsed sec. Cray T3E	Per-Processor Memory (MW)
Band solver (1)	1	133	memory	225.0
Frontal method (2)	1	192	2105	6.6
Lanczos iteration (3)	1	22	371	9.4
	4	23	223	5.4
	16		155	2.4
	64		150	1.6
MA37 (4)	1	6	memory	17.4
MUMPS (5)	4	NA	memory	
	8	NA	15	
	16	NA	14	
MPI band solver (6)	1	178	memory	346.0
	4	73	memory	88.0
	16	45	memory	23.0
	32	44	165	11.8
	64		156	6.3

sometimes fail to converge, despite the fact that convergence is theoretically guaranteed on positive-definite problems (see Section 4.8).

The computation rate for PDE2D, on the largest 3D problem with 128 Cray-Dell Xeon processors using the MPI band solver, is about 21 billion “useful” floating point calculations per second (21 gigaflops). (The number of “useful” calculations is the number that would be done if only one processor were used, so redundant calculations are not counted.) As these results illustrate, for any given problem there is a limit to the number of processors that can be utilized beneficially; beyond this limit, the time required to execute the nonparallel portions of the code, and the overhead due to communication between processors, prevents any further reduction in computing time. However, for bigger problems, more processors can be employed productively.

Table A.9.2
Performance of Parallel Solvers on 3D Problem

NXGRID=NYGRID=NZGRID=13 (17576 unknowns, half-bandwidth = 1406):				
Solver (ISOLVE)	Number of Processors	Elapsed sec. Cray-Dell	Elapsed sec. Cray T3E	Per-Processor Memory (MW)
----- (nonsymmetric) original equations -----				
Frontal method (2)	1	573	6163	4.2
MUMPS (5)	128	NA	memory	
MPI band solver (6)	1	137	memory	78.0
	4	63	memory	20.0
	8	35	269	10.2
	16	28	151	5.2
	64		63	1.4
	256		49	0.4
----- (positive-definite) normal equations -----				
MA27 (1)	1	63	memory	32.0
CG iteration (3)	1	6	67	3.6
	4	4	27	1.9
	16		18	0.6
	32		16	0.4
MUMPS (4)	16	NA	memory	
	32	NA	35	
	64	NA	35	
NXGRID=NYGRID=NZGRID=20 (64000 unknowns, half-bandwidth = 3282):				
Solver (ISOLVE)	Number of Processors	Elapsed sec. Cray-Dell	Elapsed sec. Cray T3E	Per-Processor Memory (MW)
----- (nonsymmetric) original equations -----				
Frontal method (2)	1	18269	memory	22.0
MUMPS (5)	128	NA	memory	
MPI band solver (6)	1	memory	memory	656.0
	2	7415	memory	328.0
	4	1196	memory	164.0
	16	348	memory	41.3
	64	163	872	10.6
	128	133		5.5
	256		448	2.9
----- (positive-definite) normal equations -----				
MA27 (1)	1	1495	memory	253.0
CG iteration (3)	1	66	491	12.9
	4	25	174	6.8
	16	24	97	2.2
	64		73	1.0
MUMPS (4)	128	NA	memory	

For 2D and 3D problems, PDE2D also allows the installer to plug in his/her own local linear system solver(s). If ISOLVE=5 is chosen for a 2D problem (Galerkin option), or ISOLVE=4 or 5 for a 3D problem, the matrix and right-hand side vectors are passed to a subroutine where the installer can insert code to call a local solver. The nonzeros of the matrix are supplied in sparse format, and if multiple processors are employed, these nonzeros will already be distributed over the processors, so the installer can plug in a parallel linear system solver. In fact, an interface to the very fast parallel sparse linear system solver MUMPS (mumps.enseeiht.fr) is provided and can be activated by simply removing the indicated comments. Results using MUMPS are in-

cluded in Tables A.9.1 and A.9.2, which suggest that MUMPS is faster than the other options for 2D problems.

A.10 Examples

To illustrate the range of problems solved by PDE2D, we consider five example problems.

Example 1 is the 1D time-dependent wave equation (cf. 3.0.2):

$$U_{tt} = U_{xx} \quad \text{in } 0 \leq x \leq 10,$$

with reflecting boundary conditions

$$U_x(0, t) = U_x(10, t) = 0$$

and initial conditions

$$\begin{aligned} U(x, 0) &= \max(0, 1 - |x - 5|), \\ U_t(x, 0) &= 0. \end{aligned}$$

$U(x, t)$ represents the height of the point x on a vibrating string, at time t . This equation is second order in t , so it must be broken into two first-order equations before it can be solved by PDE2D:

$$\begin{aligned} U_t &= V, \\ V_t &= U_{xx}, \end{aligned}$$

with

$$\begin{aligned} U_x(0, t) &= U_x(10, t) = 0, \\ V_x(0, t) &= V_x(10, t) = 0, \\ U(x, 0) &= \max(0, 1 - |x - 5|), \\ V(x, 0) &= 0. \end{aligned}$$

Figure A.10.1 plots U as a function of x and t , up to $t = 10$. The initial “bump” separates into two smaller bumps, which move to the edges of the string, reflect back, and recombine at $t = 10$.

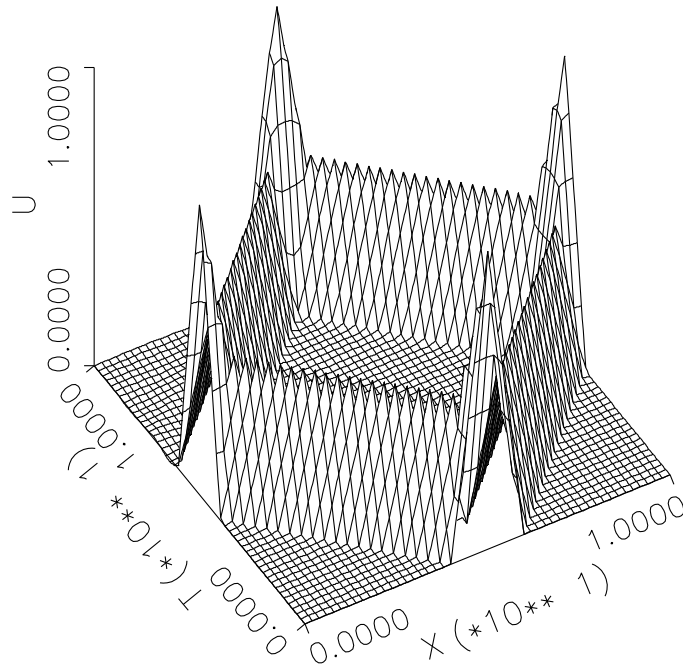


Figure A.10.1
Vibrating String Problem

Example 2 is the highly nonlinear, 2D, minimal surface problem:

$$\frac{\partial}{\partial x} \left[\frac{U_x}{\sqrt{1 + U_x^2 + U_y^2}} \right] + \frac{\partial}{\partial y} \left[\frac{U_y}{\sqrt{1 + U_x^2 + U_y^2}} \right] = 0,$$

with $U = (r - 1)(3 - r)$, where $r = \sqrt{x^2 + y^2}$, on the boundary of the region shown in Figure A.10.2. $U(x, y)$ is the height of an elastic membrane stretched over a frame whose position is given by the boundary conditions. Figure A.10.3 gives a surface plot of the membrane height, generated by PDE2D.

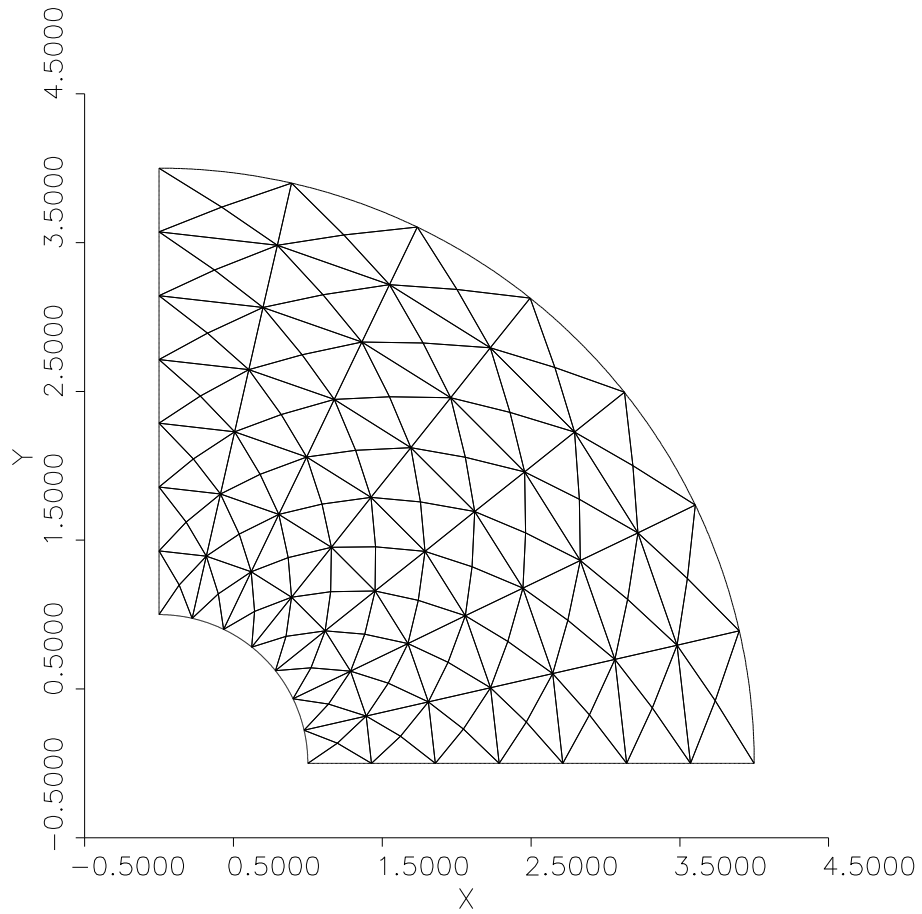


Figure A.10.2

Automatically Generated Initial Triangulation (Minimal Surface Problem)

This nonlinear problem was parameterized; that is, Newton's method was applied to solve

$$\frac{\partial}{\partial x} \left[\frac{U_x}{\sqrt{1 + \beta(U_x^2 + U_y^2)}} \right] + \frac{\partial}{\partial y} \left[\frac{U_y}{\sqrt{1 + \beta(U_x^2 + U_y^2)}} \right] = 0,$$

where β is set to 0 the first iteration and gradually increased to 1 as the iteration proceeds. Thus, it is not necessary to supply good initial values, because after one iteration we have the solution to the linear problem $U_{xx} + U_{yy} = 0$, which serves as a good initial guess for the next iteration, and so on.

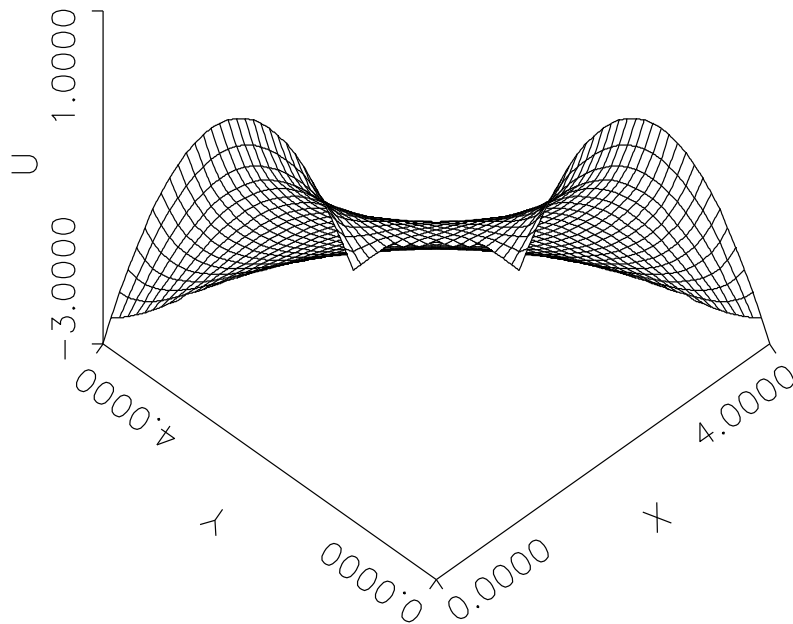


Figure A.10.3
Minimal Surface

Example 3 is the steady-state, 2D, diffusion problem (cf. 4.0.1):

$$\frac{\partial}{\partial x}[D(x, y)U_x] + \frac{\partial}{\partial y}[D(x, y)U_y] = 0,$$

in a rectangle with a “knob” on top, shown in Figure A.10.4. The diffusion coefficient $D(x, y)$ is equal to 5 in the “knob” (initial triangles 7–10) and 1 in the lower region. The Galerkin method is used for this 2D problem, so the fact that the diffusivity is discontinuous at an interface is not a problem, even though the interface is curved. As discussed in Section 5.11, $D(x, y)$ can simply be defined as a discontinuous function, and no interface conditions are needed between the two rectangles. PDE2D cannot handle discontinuous material properties in 1D or 3D problems, because only the collocation method is available in these cases.

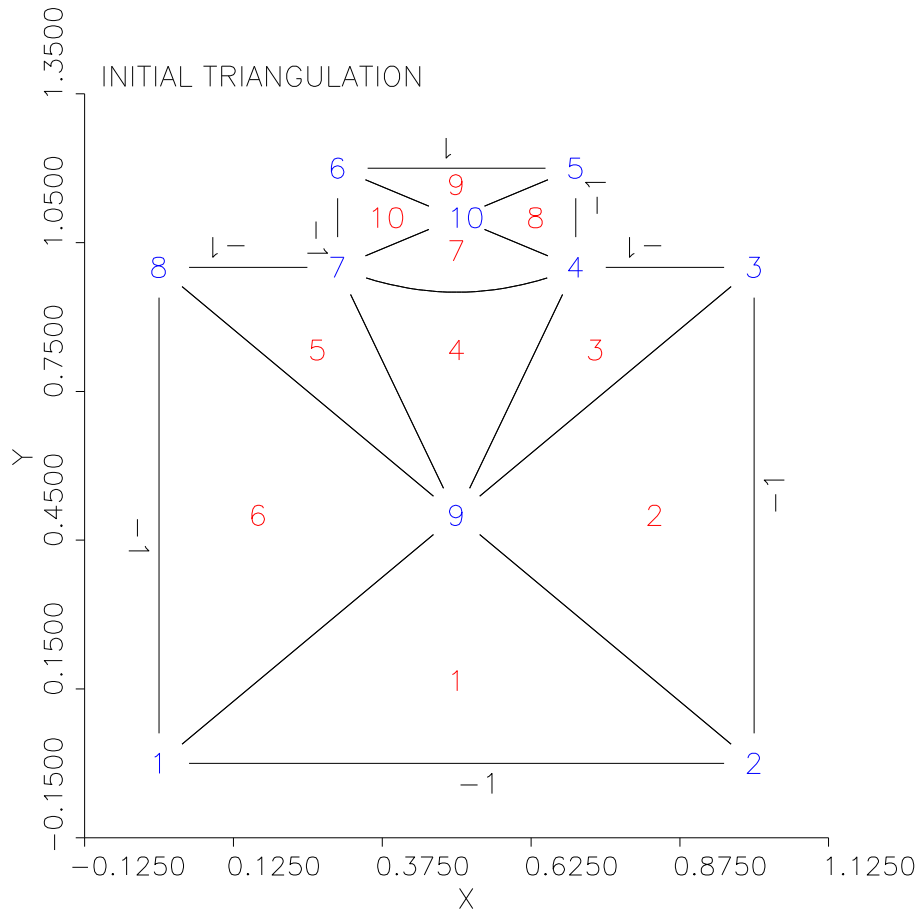


Figure A.10.4

Manually Generated Initial Triangulation (Diffusion Problem)

The boundary conditions are

$$D \frac{\partial U}{\partial n} = 1 \quad \text{at the top of the knob,}$$

$$U = 0 \quad \text{on the rest of the boundary.}$$

Thus, there is a constant flux of U into the region along the top boundary, while along the rest of the boundary the density U is held at 0.

A uniform triangulation of 500 quadratic triangles is first used, then the problem is re-solved using adaptive grid generation, resulting in the graded grid shown in Figure A.10.5. The adaptively generated grid is most dense near the top, where the solute is entering. A contour plot, generated by PDE2D, of the density is shown in Figure A.10.6.

Example 4 is the 2D steady-state Navier-Stokes incompressible fluid flow

problem

$$\begin{aligned}\frac{\partial}{\partial x}[-Re p + 2U_x] + \frac{\partial}{\partial y}[U_y + V_x] &= Re[UU_x + VU_y], \\ \frac{\partial}{\partial x}[U_y + V_x] + \frac{\partial}{\partial y}[-Re p + 2V_y] &= Re[UV_x + VV_y], \\ U_x + V_y &= 0 \text{ (continuity equation),}\end{aligned}$$

solved in a pipe with an elbow. $(U(x, y), V(x, y))$ is the fluid velocity vector, and $p(x, y)$ is the pressure. A penalty function approach is used, where the pressure, p , is replaced by $-\alpha(U_x + V_y)$, where α is a large number. The continuity equation is not enforced exactly, but since $U_x + V_y = -p/\alpha$, for large α it is almost satisfied. Physically, the penalty formulation means it requires a very large increase in pressure to make a small change in volume, so the fluid is “almost” incompressible.

At the pipe inlet, $U = 0, V = -x(1 - x)$, and at the outlet the tractions are zero ($GB_1 = GB_2 = 0$); on the rest of the boundary “no-slip” conditions hold, $U = V = 0$. The Reynold’s number (Re) is set to 500. The velocity field as drawn by PDE2D is shown in Figure A.10.7.

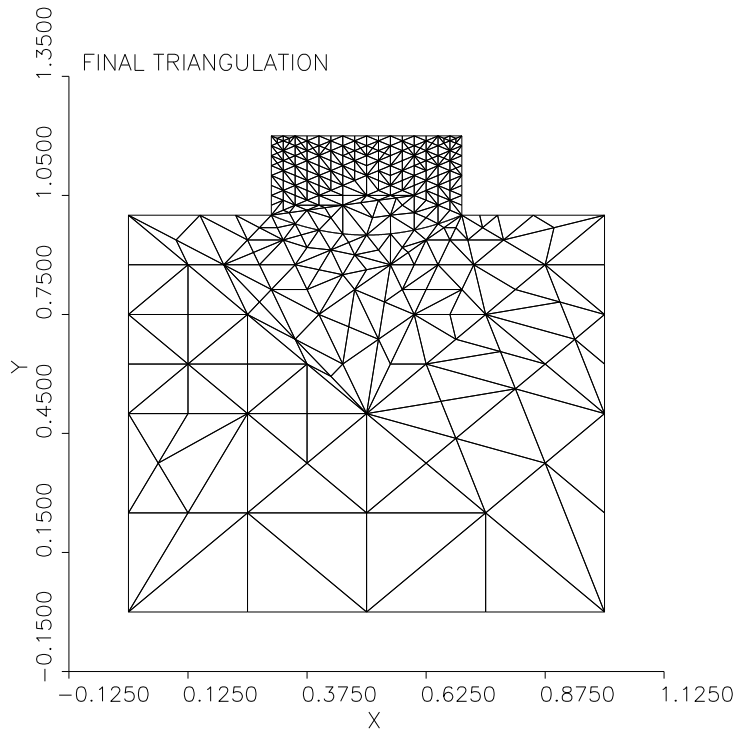


Figure A.10.5
Adaptively Generated Final Triangulation (Diffusion Problem)

Example 5 is a 3D Schrödinger eigenvalue problem [Fitzgerald and Sewell 2000]:

$$-\frac{1}{2}(U_{xx} + U_{yy} + U_{zz}) + (1/R_0 - 1/R_a - 1/R_b)U = \lambda U,$$

where

$$R_0 = 2,$$

$$R_a = \sqrt{(x^2 + y^2 + (z + R_0/2)^2)},$$

$$R_b = \sqrt{(x^2 + y^2 + (z - R_0/2)^2)}.$$

In this problem, $U(x, y, z)$ represents the wave function for an electron under the influence of two protons, located at $(0, 0, -R_0/2)$ and $(0, 0, R_0/2)$. The boundary condition is $U = 0$ at $r = \infty$. This boundary condition is approximately imposed by setting $U = 0$ on the boundary of a large cube, $(-10, 10) \times (-10, 10) \times (-10, 10)$. Then a nonuniform grid is created that is most dense near the protons, as shown in Figure A.10.8.

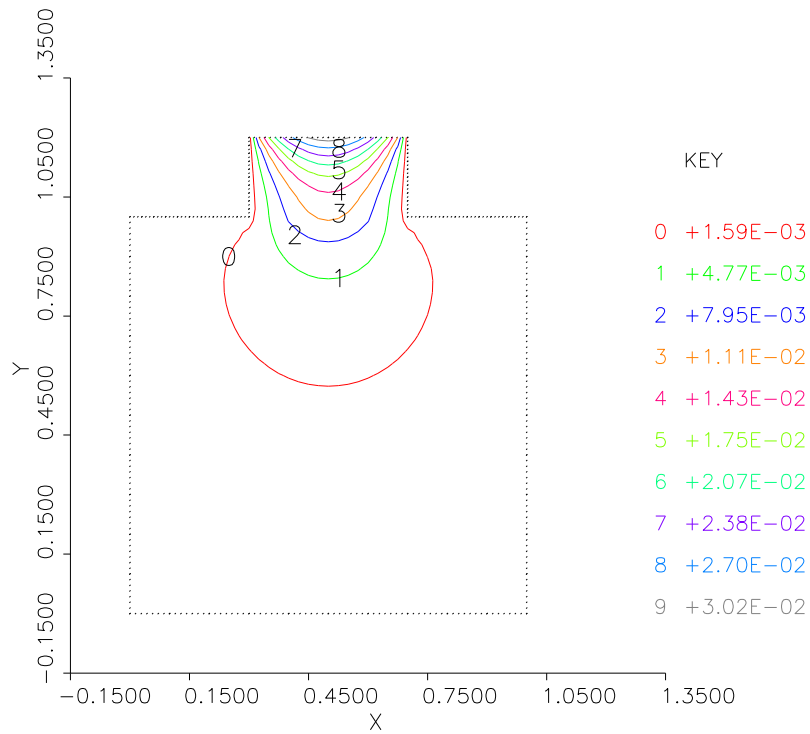


Figure A.10.6
Diffusion Problem

The eigenvalue λ represents the energy of the electron, and U^2 is the electron's probability distribution function. The equation is in dimensionless

form, and the first three eigenvalues are found (in separate computations) to be approximately $\lambda = -0.598, -0.163,$ and 0.078 . The eigenfunction corresponding to the first eigenvalue is shown in Figure A.10.9, using the contour “surface” plotting algorithm described in Sewell [1988]. Normally the different surfaces are coded by color. Figure A.10.10 shows a cross-sectional contour plot, at $x = 0$, of the second eigenfunction.

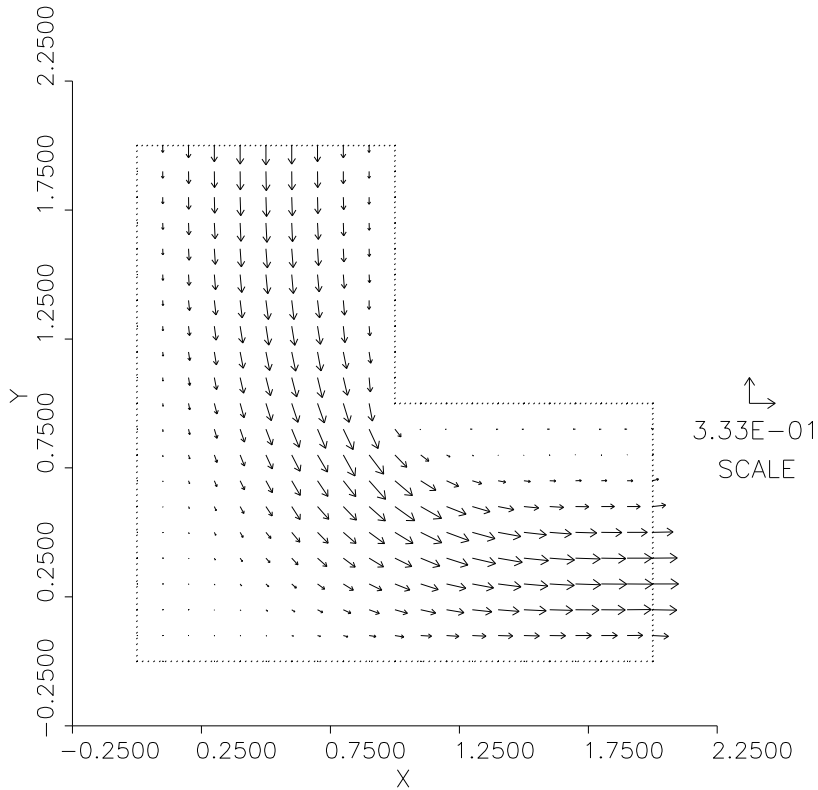


Figure A.10.7
Fluid Flow Problem

A.11 Problems

Use PDE2D to solve the following problems:

1. Solve 1.6.3, with adaptive stepsize control ($NSTEPS = 8$, so $dt_{\max} = 0.25$) and compare with the steps taken by the adaptive RKF routine (Table 1.6.3). Note that, in addition to 1D, 2D, and 3D problems, PDE2D also solves “0D” problems, that is, algebraic systems, ordinary differential equation systems, and algebraic eigenvalue problems.

2. Solve the complex equation

$$\begin{aligned} U_{xx} + U_{yy} &= U^4 && \text{in the unit circle,} \\ \text{with } U &= i && \text{on the boundary.} \end{aligned}$$

Write $U = UR + i UI$ and break this complex equation into a system of two real equations, for the variables UR and UI . Although the term $(UR+i UI)^4$ can be broken into its real and imaginary parts analytically, it is easier to do this using FORTRAN complex arithmetic. Plot the complex solution as a vector (UR, UI) .

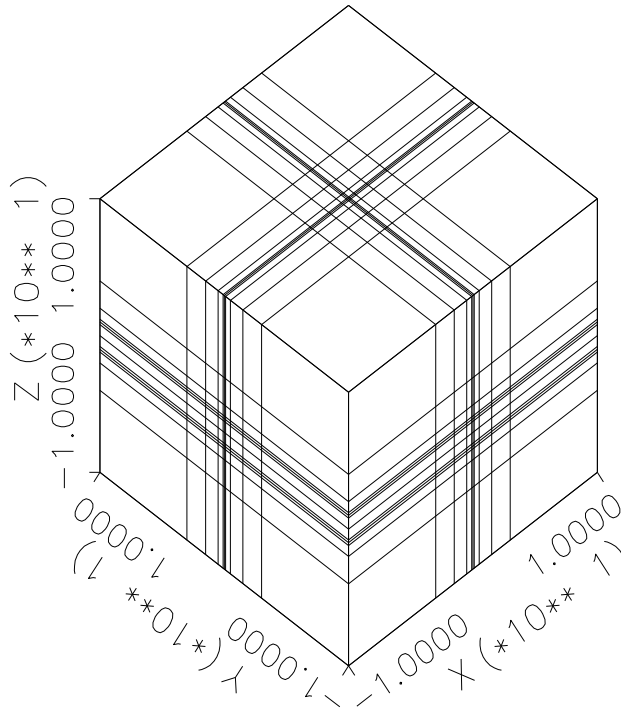


Figure A.10.8
Gridlines for Schrödinger Equation

3.
 - a. Solve the time-dependent diffusion-reaction problem of Section 2.5, generating two plots (U and V) similar to Figure 2.5.2.
 - b. Solve the damped wave equation of Section 3.4, generating a plot similar to Figure 3.4.3.
 - c. Solve the 3D steady-state problem 4.7.6, and generate a plot similar to Figure 4.7.3. Also plot the gradient (U_x, U_y, U_z) at the cross section $y = 0.5$, using PDE2D's 3D vector field plotter, which superimposes a contour plot of the out-of-plane component (U_y) on a vector plot of the in-plane components (U_x, U_z).

- d. Solve the 1D eigenvalue problem 4.11.7, and compare your first two eigenvalues with the values in Table 4.11.1.
- e. Solve the 2D eigenvalue problem 5.11.2, and generate a contour plot like Figure 5.11.4. The smallest eigenvalue should be -6.9423 .

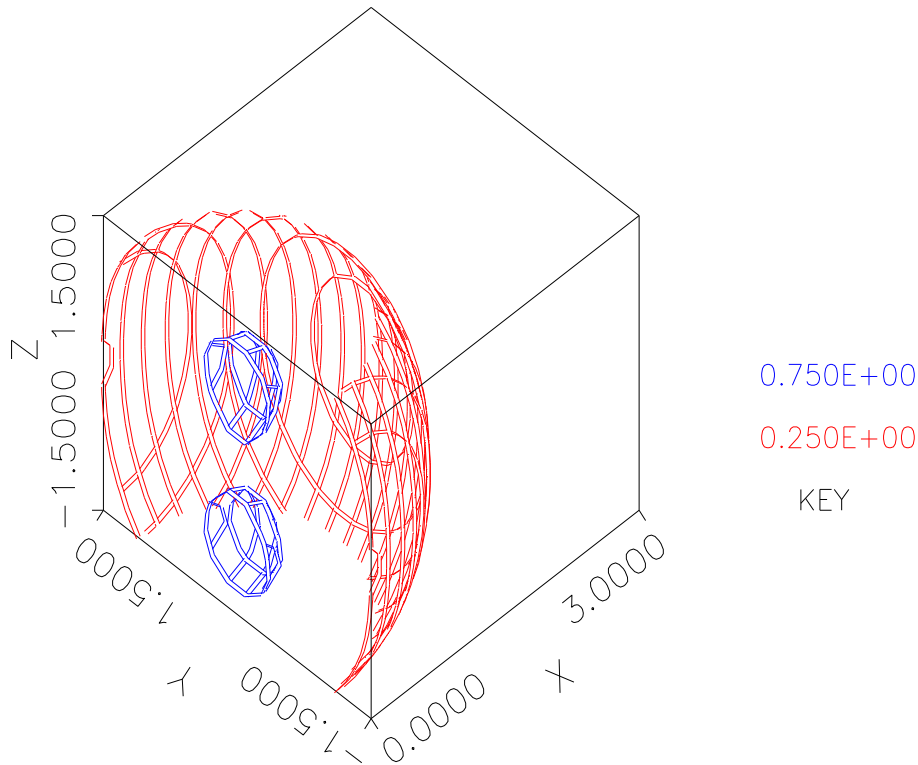


Figure A.10.9
First Eigenfunction for Schrödinger Equation

4. In a 2D linear elastic body under steady-state and plain strain conditions, the 3D elasticity equations 3.0.7 reduce to

$$\frac{\partial \sigma_{xx}}{\partial x} + \frac{\partial \sigma_{xy}}{\partial y} + f_1 = 0,$$

$$\frac{\partial \sigma_{xy}}{\partial x} + \frac{\partial \sigma_{yy}}{\partial y} + f_2 = 0,$$

where

$$\begin{aligned}\sigma_{xx} &= \frac{E[(1-\mu)U_x + \mu V_y]}{(1+\mu)(1-2\mu)}, \\ \sigma_{xy} &= \frac{E[U_y + V_x]}{2(1+\mu)}, \\ \sigma_{yy} &= \frac{E[(1-\mu)V_y + \mu U_x]}{(1+\mu)(1-2\mu)}.\end{aligned}$$

Suppose a (2D) square box, with unit edges, is pushed with a unit horizontal force along a slippery floor, against a sticky wall. We might model the boundary conditions (see Section 3.0) as

$$\begin{aligned}U = V = 0 & \quad \text{at } x = 1 \text{ (zero displacements at wall),} \\ \sigma_{xx}N_x + \sigma_{xy}N_y = 0 & \quad \text{at } y = 1 \text{ (zero forces on top),} \\ \sigma_{xy}N_x + \sigma_{yy}N_y = 0 & \\ \sigma_{xx}N_x + \sigma_{xy}N_y = 1 & \quad \text{at } x = 0 \text{ (unit horizontal force on left side),} \\ \sigma_{xy}N_x + \sigma_{yy}N_y = 0 & \\ \sigma_{xy} = V = 0 & \quad \text{at } y = 0 \text{ (rolling friction condition on floor).}\end{aligned}$$

If $E = 10^6$, $\mu = 0.2$, and there are no internal forces ($f_1 = f_2 = 0$), solve this elasticity problem, and plot the displacement (U, V) vector field and the stress field. The rolling friction (or symmetry) boundary condition at $y = 0$ is of mixed type. Explain why this can be handled by setting $GB_1 = 0$ and $GB_2 = \beta V$, where β is a large number (say, 10^{20}).

5. Solve the clamped elastic plate problem:

$$\begin{aligned}U_{xxxx} + 2U_{xxyy} + U_{yyyy} &= 1 & \text{in the unit circle,} \\ \text{with } U = \frac{\partial U}{\partial n} &= 0 & \text{on the boundary.}\end{aligned}$$

This fourth-order PDE can be reduced to a system of two second-order equations with the introduction of the variable $V = U_{xx} + U_{yy}$. The mixed boundary condition can be handled in a manner similar to the way the rolling friction boundary condition was treated in the previous problem. Compute the integral of the absolute value of the error, given that the exact solution is $U = (1 - x^2 - y^2)^2/64$.

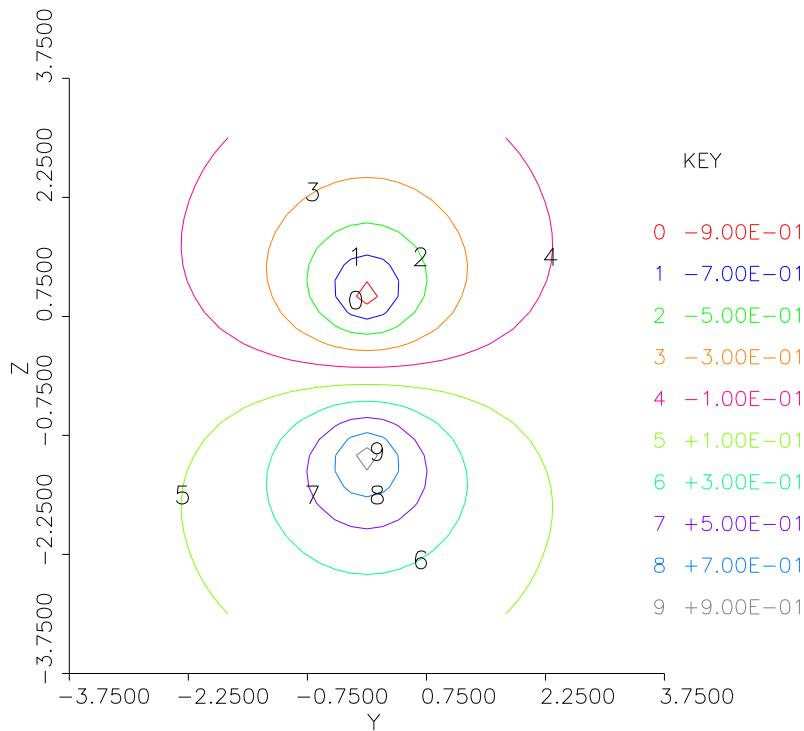


Figure A.10.10
X = 0 Cross Section for Second Eigenfunction

6. Consider the first-order transport problem (cf. 3.0.1):

$$\begin{aligned}
 &U_t = -U_x \quad \text{in } 0 \leq x \leq 10, \\
 \text{with } &U(x, 0) = \max(1 - x, 0), \\
 &U(0, t) = 1.
 \end{aligned}$$

Notice that there is a boundary condition only at the “upwind” boundary, $x = 0$, for this first-order equation.

- a. Solve this as a 1D collocation problem and plot U as a function of x for $t = 0, 1, 2, \dots, 12$. Note that you need to set the boundary condition at $x = 10$ to “NONE”.
 - b. Solve this as a 1D Galerkin problem and plot U as a function of x for $t = 0, 1, 2, \dots, 12$. If you set $F = U_x$, $A = 0$ then to impose “no” boundary condition on a boundary, simply set $GB = 0$, because then the boundary condition reduces to $A N_x = GB$, or $0 = 0$.
7. a. Find the smallest eigenvalue of

$$\begin{aligned}
 U_{xx} + U_{yy} + U_{zz} &= \lambda U && \text{in a cylinder of radius 1 and height 1,} \\
 U &= 0 && \text{on the boundary.}
 \end{aligned}$$

Use cylindrical coordinates (ITRANS=1). U must be 0 when $r = 1$, $z = 0$ and $z = 1$. There will also be periodic boundary conditions on θ , but no boundary condition at $r = 0$.

- b. Since the eigenfunction corresponding to the smallest eigenvalue is a function of r and z only, this eigenvalue (but not all eigenvalues) can also be found using the PDE2D 2D Galerkin algorithm, by solving the “axisymmetric” equation:

$$U_{rr} + U_r/r + U_{zz} = \lambda U \quad 0 \leq r \leq 1, \quad 0 \leq z \leq 1$$

Solve this 2D equation, after converting it to the “divergence” form required by the Galerkin method:

$$\frac{\partial}{\partial r}[rU_r] + \frac{\partial}{\partial z}[rU_z] = \lambda rU.$$

Note that since $A = rU_r$ and $B = rU_z$ are 0 at $r = 0$, setting $GB = 0$ there is equivalent to “no” boundary condition.

- c. Solve the axisymmetric equation (Problem 7b) using the 2D collocation option.
- d. Find the smallest eigenvalue of the Laplacian in a parallelepiped with edges given by the vectors $\mathbf{A}=(2,1,1)$, $\mathbf{B}=(1,2,1)$ and $\mathbf{C}=(1,1,2)$, with $U = 0$ on the boundary. You can define this region using the parametric equations $(x, y, z) = P1 * \mathbf{A} + P2 * \mathbf{B} + P3 * \mathbf{C}$, where the parameters $P1, P2$ and $P3$ vary from 0 to 1. Make a contour plot of the eigenfunction in the plane $P3 = \frac{1}{2}$ with axes x and y .
8. Solve the time-dependent version (cf. 2.0.4) of Example 3, Section A.10:

$$\frac{\partial U}{\partial t} = \frac{\partial}{\partial x}[D(x, y)U_x] + \frac{\partial}{\partial y}[D(x, y)U_y],$$

with initial condition $U(x, y, 0) = 0$. Integrate until a steady state is reached (about $t = 0.1$), which should look like Figure A.10.6. Compute the integral of U over the entire region (total solute mass), and the integral of $D \frac{\partial U}{\partial n}$ over the entire boundary (total solute flux), and plot each as a function of time. (Recall that $\frac{\partial U}{\partial n} = U_x N_x + U_y N_y$.) Look at prepared interactive driver example #7 to see how to make time plots of integrals. As a steady state is reached, the total solute mass should approach a constant, and the total solute flux should approach zero. The interface is defined by the three points $(0.3, 1.0)$, $(0.5, 0.95)$, and $(0.7, 1.0)$, through which PDE2D will fit a cubic spline. The top of the knob is $y = 1.2$.

Solve this problem using the backward Euler and Crank-Nicolson methods, with and without adaptive time step control. Note that there are

large oscillations in the solute flux versus time plot when the Crank-Nicolson method is used with a constant time step, but not in the other three cases. Explain why the Crank-Nicolson error is prone to oscillation, by looking at its characteristic polynomial (Table B.1).

9. Nick Trefethen of Oxford University published a “100-dollar, 100-digit Challenge” set of problems in the *SIAM News* [Trefethen 2002], which consisted of ten difficult numerical analysis problems. The answer to each was a single real number; the challenge was to compute it to 10 significant digits. Two of the problems involved solving partial differential equations, so I naturally used PDE2D in my attempts. Here are the problems, see how many digits you can get:
 - a. A square plate $[-1, 1] \times [-1, 1]$ is at a temperature $u = 0$. At time $t = 0$ the temperature is increased to $u = 5$ along one of the four sides while being held at $u = 0$ along the other three sides, and heat then flows into the plate according to $u_t = u_{xx} + u_{yy}$. When does the temperature reach $u = 1$ at the center of the plate? (Hint: It is much easier to obtain high accuracy if this is solved as a 3D steady-state problem, with z replacing t .)
 - b. A particle at the center of a 10×1 rectangle undergoes Brownian motion (i.e., 2D random walk with infinitesimal step lengths) until it hits the boundary. What is the probability that it hits at one of the ends rather than at one of the sides? (Hint: Though certainly not obvious, it can be shown that this problem is equivalent to solving the PDE $u_{xx} + u_{yy} = \delta(x, y)$ in this rectangle, with $u = 0$ on the boundary; then the probability that the particle hits one of the ends is just the integral of $\frac{\partial u}{\partial n}$ over the ends. Note that the 2D Galerkin solver allows the inhomogeneous PDE coefficients to include Dirac delta functions.)