PREPARED FOR THE U.S. DEPARTMENT OF ENERGY,
UNDER CONTRACT DE-AC02-76CH03073

PPPL-3552
UC-70

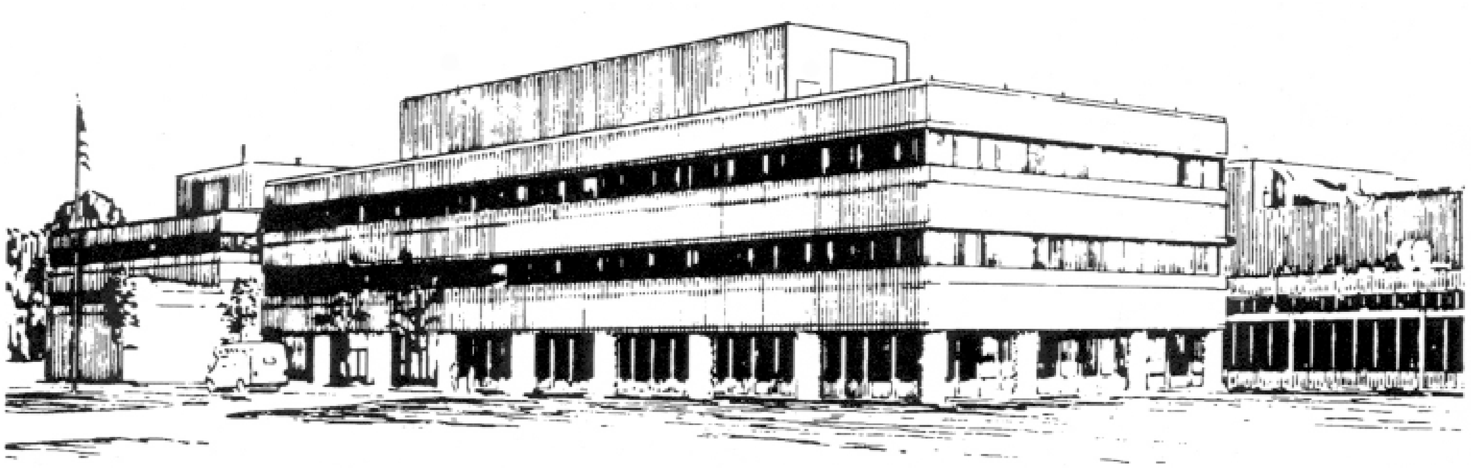
PPPL-3552

ELLIPT2D: A Flexible Finite Element Code Written in Python

by

Alex Pletzer and John C. Mollis

March 2001



ELLIPT2D: a flexible finite element code written in Python

Alex Pletzer and John C Mollis
Princeton Plasma Physics Laboratory
pletzer@pppl.gov

Abstract

The use of the Python scripting language for scientific applications and in particular to solve partial differential equations is explored. It is shown that Python's rich data structure and object oriented features can be exploited to write programs that are not only significantly more concise than their counterparts written in Fortran, C or C++, but are also numerically efficient. To illustrate this, a two-dimensional finite element code (ELLIPT2D) has been written. ELLIPT2D provides a flexible and easy-to-use framework for solving a large class of second order elliptic problems. The program allows for structured or unstructured meshes. All functions defining the elliptic operator are user supplied and so are the boundary conditions, which can be of Dirichlet, Neumann or Robbins type. ELLIPT2D makes extensive use of dictionaries (hash tables) as a way to represent sparse matrices. Other key features of the Python language that have been widely used include: operator overloading, error handling, array slicing and the Tkinter module for building graphical user interfaces. As an example of the utility of ELLIPT2D, a nonlinear solution of the Grad-Shafranov equation is computed using a Newton iterative scheme. A second application focuses on a solution of the toroidal Laplace equation coupled to a magnetohydrodynamic stability code, a problem arising in the context of magnetic fusion research.

Keywords: Elliptic, partial differential equation, two-dimensional, triangulation, conjugate gradient.

1. Motivation

It is no secret that problems that were 10 years ago considered intractable except on dedicated supercomputers, now routinely run on modest commodity personal computers. However, it is not quite as well recognized that these advances have also been accompanied by important changes in programming languages, all of which aim at greatly reducing the effort of writing codes. Scientific applications are no exceptions; many legacy programs written in Fortran, C or even C++ could probably be rewritten in a more concise way using a variety of scripting languages, such as Matlab, Scilab, IDL, Mathematica and others. Scripting languages have several advantages over ‘conventional’ languages, these include: (1) scripting languages produce portable codes, (2) require little or nor memory management responsibility by the programmer and (3) allow data types to be dynamically set at run time. All of these factors contribute to making codes less bug prone and so ultimately lead to increased productivity and shorter code development cycles.

These advantages, unfortunately, have a price; scripting languages tend to be inferior in raw numerical performance when benchmarked against compiled codes. Our experience, however, has been that scripting languages, and in particular Python, often perform above expectation, and are thus well positioned to find the optimum between satisfactory execution time and acceptable program development cost.

It is convenient here to distinguish between matrix oriented scripting languages (Matlab, Scilab, IDL,...) geared towards engineering and scientific applications from symbolic languages (Mathematica, Maple,...) that are strong in manipulating mathematical expressions, and general purpose scripting languages (Python, Perl,...). Although this distinction has become increasingly blurred, Matlab now allows for some degree of abstract programming while Mathematica improved its raw numerical performance, it is generally true that languages in group one are superior in number crunching at the expense, perhaps, of limited programming flexibility. Languages in group three on the other hand are traditionally used for web programming, as substitutes to shell scripts, for text processing and the production of graphical user interfaces. Python is one such language although one could also claim that, thanks to the NumPy [Ascher et al.] module extension, Python has some overlap with languages in group one, too.

Although extremely useful for rapid prototyping, we have found that matrix oriented languages are not always suitable for scientific applications. Many objects such as trees, graphs and sparse matrices do not neatly fit into a matrix cast. In dealing with unstructured meshes for instance, there can be an arbitrary number of connections between a node and its neighbors, which may only be known at run time. These are typical cases where Python nested structures are best suited. We have also found the dictionary data type in Python, where values are accessed by keys, extremely useful. Dictionaries (or hash tables) can grow dynamically during a calculation. Keys can be (immutable) aggregate objects, including strings, integers, doubles, a list, or a mixture of these. Finally, Python is unique in fully supporting (yet not enforcing) object oriented programming. The usual arithmetic operators can be overloaded as in C++. However, in addition to C++ the slicing operators such as a(n:m) (available in Fortran 90 and Matlab) can also be overloaded.

In summary, Python contains features individually but not globally available in Fortran 90, Java and C++. It is for the above reasons that we have selected Python as the programming language for ELLIPT2D.

2. ELLIPT2D's scope

ELLIPT2D solves equations of the form

$$-\nabla \cdot F \cdot \nabla v + g\ v = s \qquad (1)$$

on arbitrary two-dimensional domains Ω, subject to boundary conditions of Robbins type,

$$n \cdot F \cdot \nabla v + \alpha v = \beta \qquad (\text{on } \Gamma) \qquad (2)$$

Here, $\nabla = [\partial/\partial x \ \partial/\partial y]$ denotes the Cartesian "Del" operator, which applies to everything on its right, and "n" is the vector normal to the boundary contour Γ ($n \cdot \nabla$ would be the normal derivative). The solution to be determined is v(x,y), with F a positive definite, in general 2 by 2 tensor, and g(x,y) and s(x,y) arbitrary user-defined functions of the independent variables (x, y).

Equation (1) occurs in a wide range of engineering and scientific problems. For instance, the electric potential due to a charge distribution s(x,y) can be simulated in Cartesian coordinates by choosing F(x,y)=[[1,0],[0,1]] and g(x,y)=0, while the same Poisson equation in cylindrical geometry can be solved by setting F(x,y)=[[x,0],[0,x]] (x is the radius). The function g(x, y) often enters in stationary wave propagation (Helmholtz) problems in acoustics, electromagnetism and quantum mechanics. In all these applications, x and y typically represent space dimensions and F is positive definite and remains so under transformations to different coordinate systems.

In addition to the functions F(x,y), g(x, y) and s(x, y), the user must also provide the functions α and β which may vary piecewise on the boundary Γ. Letting α in (2) go to zero, one obtains Neumann boundary conditions

$$n \cdot F \cdot \nabla v = \beta \qquad (\text{on } \Gamma)$$

whereas Dirichlet boundary conditions

$$v = \beta / \alpha \qquad (\text{on } \Gamma),$$

are obtained by taking the limit of α and β large. These two types of boundary conditions have a simple physical interpretation. For the Poisson problem mentioned above, Neumann boundary conditions correspond to an electric field being applied on the boundary whereas Dirichlet conditions simulate the application of electrodes, which force the potential to take imposed values. ELLIPT2D allows either Neumann, Dirichlet or Robbins boundary conditions to be applied on various boundary regions including internal boundaries. Thus, Eq. (1) with (2) can be regarded as the most general form of linear, self-adjoint, second order elliptic equation in two dimensions.

3. Solution procedure

ELLIPT2D solves (1) and (2) using the finite element (FE) method. To allow for maximum flexibility in the shape of the domain, linear triangular elements were chosen to approximate the solution v(x, y).

A detailed description of the FE method is beyond the scope of this paper and thus the FE method will only be sketched here. There are numerous books on the subject (e.g. [Braess]) which the interested reader can consult. The implementation of the FE method in ELLIPT2D is standard. First the weak form of Eq.(1) is obtained by multiplying (1) by a test function (a finite element) and integrating over the domain. Second an integration by parts is performed to decrease the order of the "Del" operators applying on v(x, y), and third v(x, y) is expanded in the basis of finite element functions. This reduces (1) to a linear matrix system

$$A \cdot V = b \quad (3)$$

where ‘A’ is the ‘stiffness’ matrix, ‘V’ the unknown vector whose elements are the node values of v(x,y), and ‘b’ a vector containing the source contributions. The matrix elements of ‘A’ are integrals coupling two finite elements. Similarly, the ‘b’ vector comprises integrals of the test functions times the source term s(x,y). In the case of linear finite elements, these integrals are straightforward to evaluate, depending only on the node values of the F(x, y), g(x, y) and s(x, y) functions. Due to the integration by parts mentioned above, there is in addition a contribution to ‘A’ and ‘b’ from the boundary when Robbins or Neumann type boundary conditions are applied. In the case of Dirichlet boundary conditions, the A-matrix and b-vector must be modified so as to force ‘v’ to take the prescribed values.

Because the finite elements have finite support (i.e. have a zero value away from neighboring cells), coupling is non zero only for a limited number of ‘A’ elements, and so ‘A’ is sparse. The elements of ‘A’ are stored efficiently using a dictionary representation {(i0, j0): v0, (i1, j1): v1, ...} where (i0, j0) are the row/column key indices and v0 the corresponding value of the (nonzero) matrix element (i0, j0). Together with a vector class, the usual matrix operations addition, multiplication, etc. are overloaded in a sparse matrix class.

One difficulty arising with unstructured meshes is the extra bookkeeping required to track the connectivity between nodes. A dictionary data type ‘node’ is used for this purpose: {i0 : [(x0, y0), [im, ...], a0, ...} where i0 is a node index (a key), (x0, y0) its coordinates, [im, ...] are the connections of i0 to its neighbors, and a0 some additional node attribute (used for instance to indicate if a node is on a boundary). Note that the length of the connectivity list [im, ...] varies from node to node for unstructured meshes. Such a data structure is straightforward to implement in Python but often difficult or cumbersome in other languages.

Efficient methods to solve (3) exist when ‘A’ is sparse. One such method, which is particularly easy to implement for matrices that have arbitrary sparsity, is the conjugate gradient (CG) method. With the matrix operations overloaded in the sparse matrix and vector class, the CG method could be implemented in fewer than 30 Python lines.

The process of solving the elliptic equation can thus be summarized as follows:

1. Triangulate the domain

2. Build the stiffness matrix ‘A’ and the right hand side vector ‘b’

3. Update ‘A’ and ‘b’ by applying boundary conditions

4. Solve the linear system of equations

4. A simple example

Let us go through each of these steps in detail for a simple example. Consider the problem of solving the Laplace equation on a rectangular mesh

```
xmin, xmax = 0., 1.
ymin, ymax = 0., 1.4

N = 101
nx1 = int(sqrt(float(N)*(xmax-xmin)/(ymax-ymin)))
ny1 = int(sqrt(float(N)*(ymax-ymin)/(xmax-xmin)))
```

with ‘nx1’ nodes in the ‘x’ direction and ‘ny1’ nodes in the ‘y’ direction. A grid object (nodes + connections) is constructed by calling a ‘canned’ method

```
grid = reg2tri.rect2criss((xmin, ymin, xmax, ymax), nx1, ny1)
```

which breaks a rectangular domain into a regular set of triangles. For more general structured meshes, ‘grid’ can be constructed from a set of interleaving (x, y) coordinates. For totally arbitrary domains, a Delaunay triangulation can be performed using the `Ireg2tri’ module. In the latter case, a list of boundary segments, possibly holes and regions need in addition be provided.

Next we apply boundary conditions, which we choose to be a sine wave on the west edge, and zero Dirichlet conditions on the south and north edges:

```
db = DirichletBound()
for i in range(nx1):
    db[i]             = 0.0  # south
    db[nx1*(ny1-1)+i] = 0.0   # north

for i in range(ny1):
    y = grid.y(i*nx1)
    db[i*nx1] = sin(pi*(y-ymin)/(ymax-ymin))   # west
```

On the east edge, we impose zero-flux Neumann boundary conditions. Since zero-flux conditions are the default boundary conditions of the FE method, there is no need to create a ‘Neumann’ object in this case.

The ‘ellipt2d’ object can now be built

```
F='1.'
```

```
g, s = '0.', '0.'
equ = ellipt2d.ellipt2d(grid, F, g, s)
```

and the stiffness matrix computed

```
[amat, b] = equ.stiffnessMat()
```

Notice that the functions ‘F’, ‘g’ and ‘s’, which are passed as arguments to the ellipt2d constructor, are strings. For more general equations, these can be any expressions of ‘x’ and ‘y’ such that ‘F’ is positive definite and the functions are finite at the nodes. Thus, we could just as well have chosen `F="2 + cos(pi*x**2 *y)"` for instance. For problems where these functions cannot be expressed in closed form, ELLIPT2D offers in addition the option to supply ‘F’, ‘g’ or ‘s’ as vector instances, instead of strings; `vector.zeros(N)` is equivalent to supplying `"0."`, for instance. Also note that a scalar form of ‘F’ was fed into ellipt2d: that is "1." is formally equivalent to the identity matrix `[["1.", "0."],["0.","1."]]`.

Also of interest, the constructor ellipt2d does not take the boundary object ‘db’ as input, the role of the boundary condition object being to update the stiffness matrix and right hand side vector. When the boundary conditions are of the zero-flux type, a natural boundary condition of the problem, there is no boundary contribution from the weak form and this step can be avoided altogether, as in the present case for the east edge.

```
equ.dirichletB(db,amat,b)
```

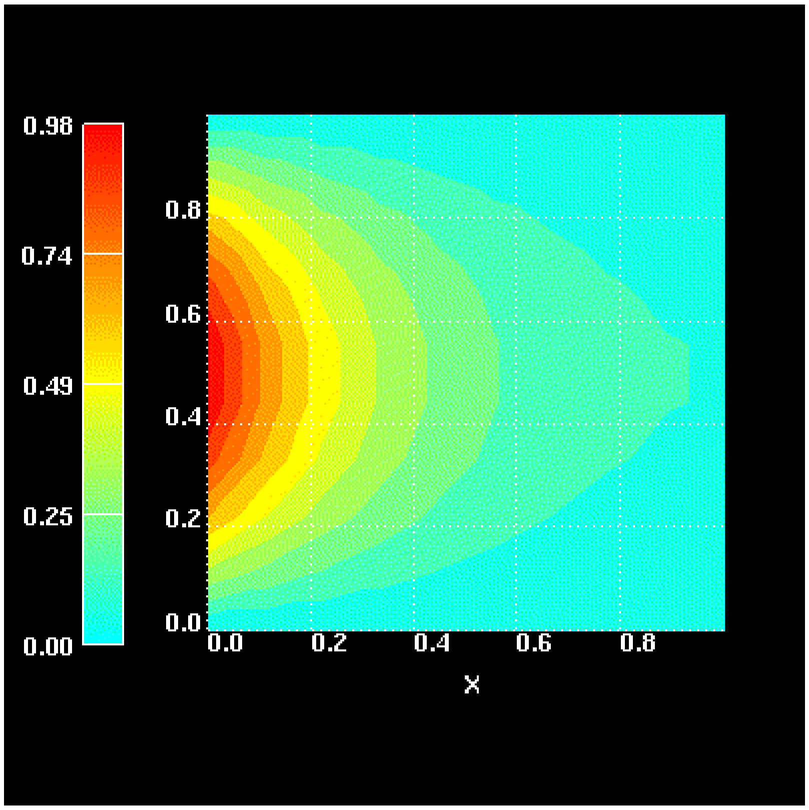
In the final step, we solve Eq.(3) by invoking the conjugate gradient method

```
v0 = vector.zeros(len(b))
v = amat.CGsolve(b, v0, 1.e-6, 2*len(b))
```

The CG method is iterative in nature and requires an initial guess ‘v0'. The next arguments specify the maximum tolerance and the maximum number of iterations. Methods for saving the results in various formats (toUCD for AVSExpress, toDX for openDX) are available

```
equ.toUCD(v, 'ex.inp')
```

for graphical postprocessing.



Laplace equation with imposed sine function on the west, v=0 on the north and south boundaries and zero flux on the east.

5. Application I

Next consider the more challenging problem of solving the Grad-Shafranov equation

$$-\nabla \cdot (1/x) \nabla v = F(v)\, x + G(v)/x \qquad (v=0 \text{ on } \Gamma)$$

where the functions F(v) and G(v) depend nonlinearly on the solution ‘v’. This equation models various physical phenomena such as the magnetohydrodynamic equilibrium of a fusion or space plasma as well as fire whirls, all in cylindrical geometry where ‘y’ denotes the vertical direction of axial symmetry and ‘x’ the radius.

In order to be solved, this equation must first be linearized by expanding the functions around some approximate solution ‘u’, $F(v) \sim F(u) + (v - u)\, F'(u)$ and likewise for $G(v) \sim G(u) + (v - u)\, G'(u)$, where prime denotes derivative with respect to the variable ‘v’. The linearized equation then becomes

$$-\nabla \cdot (1/x) \nabla v - [F'(u)\, x + G'(u)/x]\, v = [F(u) - u\, F'(u)]\, x + [G(u) - u\, G'(u)]/x.$$

This is Newton's method to solve the equation by iterations:

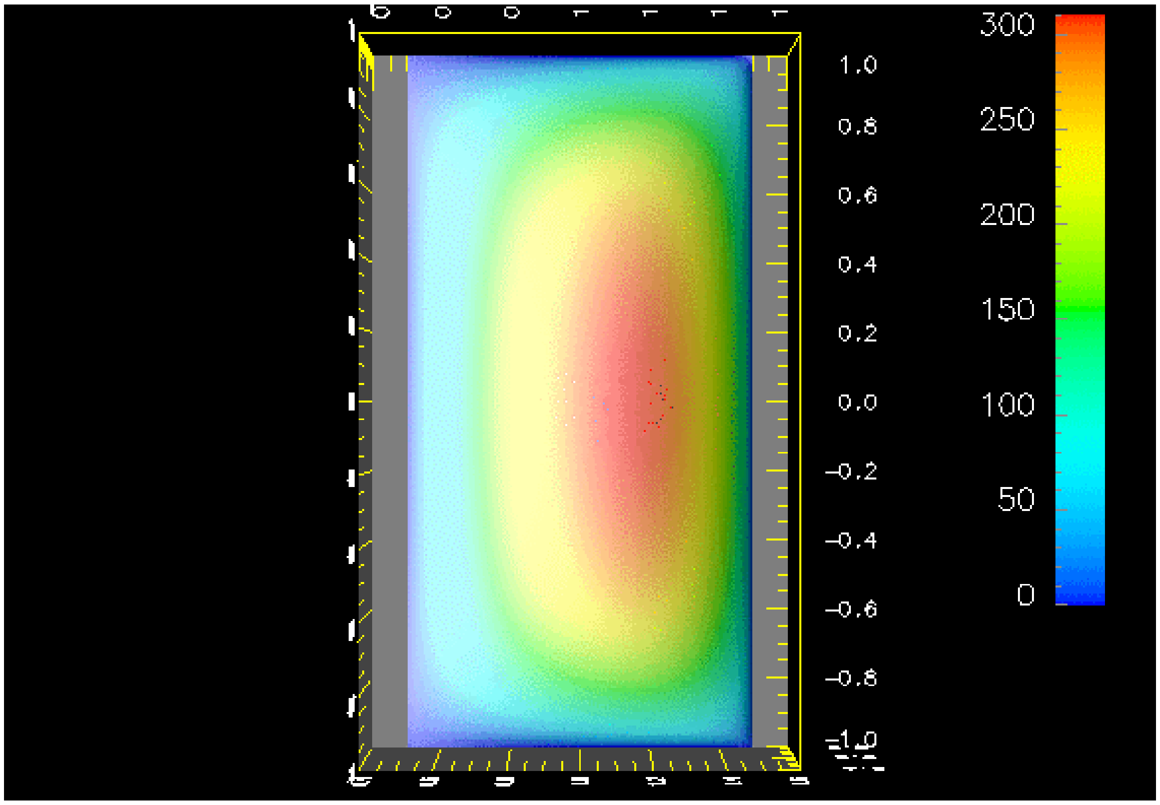
1. Choose an initial guess u(x, y)

2. Solve the equation for v(x, y)

3. v(x, y) becomes u(x,y)

4. Repeat step 2.

until the average change v(x, y) – u(x, y) is smaller than a prescribed tolerance. The figure below shows a converged solution for a right hand side function $(F_0\, x + G_0 / x)*\exp(-\kappa / (v + \delta))$, $F_0 = 1600$, $G_0 = 100$, $\kappa = 0.1$, and $\delta = 0.1$. The graphics was obtained using openDX.



Converged solution after 6 iterations for central values of F=1600 and G=100

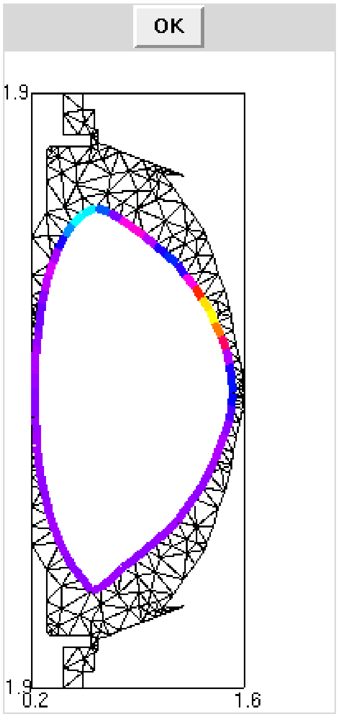
6. Application II

Finally consider the problem of determining the magnetic field in a cylindrical vacuum region of complex geometry with a central hole. This problem is relevant to fusion plasma physics where, due to some plasma instability developing in the core, a small magnetic field perturbation is detected in the vacuum from which the mode structure can be inferred. The natural representation of the magnetic field in vacuum is in term of a gradient, since no current can exist in vacuum. For an instability with dependence exp(-i nφ) in the cylindrical angle φ, this yields a cylindrical Laplace equation with F = [[x, 0],[0,x]], $g = n^2$ x and s = 0. At the vacuum to plasma interface, inhomogeneous Neumann conditions are applied whereas a perfectly conducting wall on the outer boundary gives zero-flux conditions there.

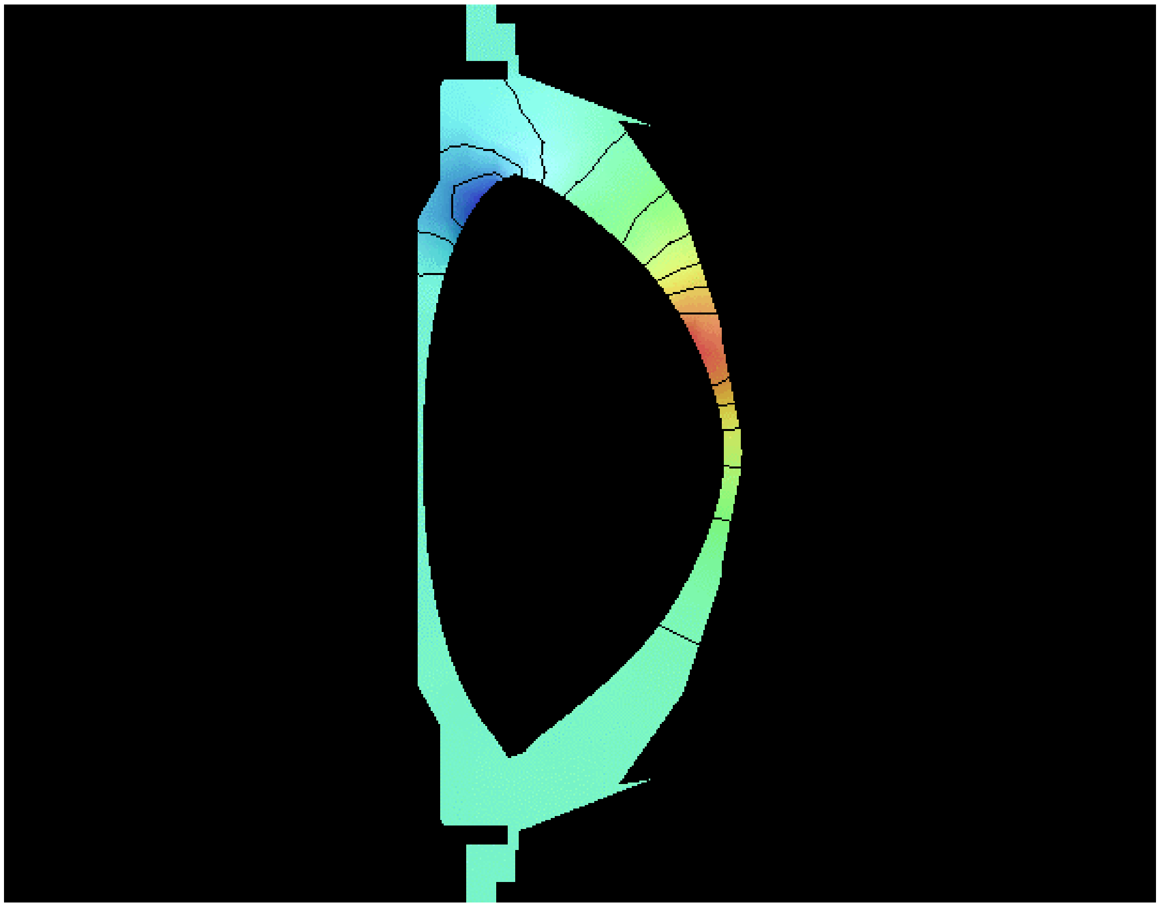
The Neumann boundary conditions at the plasma surface interface depend on details of the mode structure, as provided by an independent code (PEST3) in the form of a displacement field stored in netCDF file format. We used Hinsen's Scientific module [Hinsen] to read this file and the Numeric module [Ascher et al.] to translate these data into a magnetic field.

The unstructured mesh generation was performed using TRIANGLE [Shewchuk], a robust and very fast triangulation routine written in C for which we have developed a Python interface. The linear sparse system of rank 300 was solved using a Python module provided by Travis Oliphant, which interfaces to the Fortran library UMFPACK [Davis]. Both packages are part of the ELLIPT2D distribution.

The vacuum geometry with color coded Neumann boundary conditions is shown below:



Triangulation of the vacuum region. The magnitude of the applied Neumann boundary conditions at the plasma surface is shown color coded.



Solution of the toroidal Laplace equation.

7. Summary

By writing ELLIPT2D, we have found that Python is ideal for the rapid development and deployment of numerically intensive calculations. A Laplace equation with 100 nodes is typically solved in 3-10 CPU secs on a Pentium III 550 MHz PC in pure Python mode. The execution time largely depends on the type of boundary conditions, on the quality of the initial guess and on the tolerance fed to the conjugate gradient solver. Interestingly, we found that the most recent version 2.0 of Python is 10 to 25 % slower than Python 1.6, which is itself slower than Python 1.5.2 by a few percents.

Since most of ELLIPT2D does not rely on Numeric or any other C package, a subset of the program can run under Jython (Python implemented in Java and available at http://jython.sourceforge.net) with little modification. This allows applets based on ELLIPT2D to be built and run through a web browser. Our tests revealed that recent browsers such as Opera 5 or Netscape 6 run ELLIPT2D programs at about 60% the speed of their Python 2.0 counterparts.

If we break up the CPU time between modules, we find the conjugate gradient solver to be the numerically most expensive by far, with the matrix assembly part requiring < 10 %

of the CPU time. Hence we strongly recommend the use of Oliphant's interface to the Fortran direct sparse matrix solver of the UMFPACK library, which is in excess of 20 times faster than our 30-line Python conjugate gradient (CG) implementation.

When the size of the problem N increases, the story changes. A direct solver such as UMFPACK has a less favorable scaling $\sim N^{2.7}$ than CG ($\sim N^{1.7}$) so that for $N > 5000$ there would be a clear need to use another solver. So far, the matrix assembly in Python has given full satisfaction in terms of performance with a scaling in $\sim N^{1.6}$. The UMFPACK CPU time dominating over the matrix assembly time for $N > 200$.

ELLIPT2D [Pletzer, Mollis] is free and can be downloaded from http://w3.pppl.gov/~pletzer/programs/fortran-c-c++/ellipt2d/HTMLdoc/home.html.

8. References

- [Ascher et al.] David, Dubois Paul F, Hinsen Konrad, Hugunin Jim and Oliphant Travis (1996), "A tutorial for the Python Numeric module" http://starship.python.net/~da/numtut/ (accessed 11/4/00)
- [Braess] Dietriech (1997), "Finite elements". Cambridge University Press
- [Davis] Timothy A and Duff Iain S (1997), "UMFPACK Version 2.2: Unsymmetric-pattern Multifrontal Package" http://www.netlib.no/netlib/linalg (accessed 7/24/00)
- [Hinsen] Konrad (2000), "Scientific Python", http://starship.python.net/crew/hinsen/scientific.html (accessed 11/4/00)
- [Oliphant] Travis (2000), provided Python interface to UMFPACK.
- [Pletzer] Alexander and Mollis John C (2000), "ELLIPT2D: a general 2-D elliptic equation solver", http://w3.pppl.gov/~pletzer/programs/fortran-c-c++/ellipt2d/HTMLdoc/home.html (accessed 11/4/00)
- [Shewchuk] Jonathan R (1996), "Triangle", http://www.cs.cmu.edu/~quake/triangle.html (accessed 11/4/00)

External Distribution

Plasma Research Laboratory, Australian National University, Australia
Professor I.R. Jones, Flinders University, Australia
Professor João Canalle, Instituto de Fisica DEQ/IF - UERJ, Brazil
Mr. Gerson O. Ludwig, Instituto Nacional de Pesquisas, Brazil
Dr. P.H. Sakanaka, Instituto Fisica, Brazil
The Librarian, Culham Laboratory, England
Library, R61, Rutherford Appleton Laboratory, England
Mrs. S.A. Hutchinson, JET Library, England
Professor M.N. Bussac, Ecole Polytechnique, France
Librarian, Max-Planck-Institut für Plasmaphysik, Germany
Jolan Moldvai, Reports Library, MTA KFKI-ATKI, Hungary
Dr. P. Kaw, Institute for Plasma Research, India
Ms. P.J. Pathak, Librarian, Insitute for Plasma Research, India
Ms. Clelia De Palo, Associazione EURATOM-ENEA, Italy
Dr. G. Grosso, Instituto di Fisica del Plasma, Italy
Librarian, Naka Fusion Research Establishment, JAERI, Japan
Library, Plasma Physics Laboratory, Kyoto University, Japan
Research Information Center, National Institute for Fusion Science, Japan
Dr. O. Mitarai, Kyushu Tokai University, Japan
Library, Academia Sinica, Institute of Plasma Physics, People's Republic of China
Shih-Tung Tsai, Institute of Physics, Chinese Academy of Sciences, People's Republic of China
Dr. S. Mirnov, Triniti, Troitsk, Russian Federation, Russia
Dr. V.S. Strelkov, Kurchatov Institute, Russian Federation, Russia
Professor Peter Lukac, Katedra Fyziky Plazmy MFF UK, Mlynska dolina F-2, Komenskeho Univerzita, SK-842 15 Bratislava, Slovakia
Dr. G.S. Lee, Korea Basic Science Institute, South Korea
Mr. Dennis Bruggink, Fusion Library, University of Wisconsin, USA
Institute for Plasma Research, University of Maryland, USA
Librarian, Fusion Energy Division, Oak Ridge National Laboratory, USA
Librarian, Institute of Fusion Studies, University of Texas, USA
Librarian, Magnetic Fusion Program, Lawrence Livermore National Laboratory, USA
Library, General Atomics, USA
Plasma Physics Group, Fusion Energy Research Program, University of California at San Diego, USA
Plasma Physics Library, Columbia University, USA
Alkesh Punjabi, Center for Fusion Research and Training, Hampton University, USA
Dr. W.M. Stacey, Fusion Research Center, Georgia Institute of Technology, USA
Dr. John Willis, U.S. Department of Energy, Office of Fusion Energy Sciences, USA
Mr. Paul H. Wright, Indianapolis, Indiana, USA