Differential Equations and Dynamical Systems, Volume 8, Number 1, January 2000, pp. 29 - 49.

Solution of the Linear Diffusion Equation on a Nonuniform Grid in Three Dimensions

A. Komashko, D. Laney, M. Prasad and R. Vemuri

(Submitted by: V. Sree Hari Rao)

Abstract

The point source problem for the linear diffusion equation was solved numerically on a nonuniform mesh in 3D using Galerkin's finite element method. The hexahedral computational mesh was constructed with random recursive subdivision. It was shown that despite irregularity introduced by the grid, the numerical solution still possesses important symmetry properties and remains good approximation to the analytical solution.

1 Introduction

We describe here the 3D finite element solution of an exactly solvable diffusion equation on highly distorted meshes and compare it to the corresponding analytical solution. Such distorted meshes are typical of Lagrangian hydrocode simulations such as DYNA3D [4] and it is particularly important that the diffusion solver be robust and accurate on such

[0]Received on January 12, 2000.

[0]AMS (MOS) 1991 Subject classifications: 65N30.

[0]

meshes.

We show here that the finite element method indeed offers a robust and accurate solution of the diffusion equation on 3D distorted meshes. In particular, we use linear finite elements for the spatial discretization and an unconditionally stable fully implicit backward Euler time differencing. On distorted meshes the condition number of the finite element stiffness matrix can become quite large. For this reason we used the preconditioned incomplete Cholesky conjugate gradient (ICCG) iterative solution for the implicit equations ([7], [5]). It is a well known empirical fact that ICCG is a particularly robust method for solving ill-conditioned systems.

The paper is organized as follows. In Section 2, the problem to be solved is stated along with the initial and boundry conditions and it's exact solution is presented. In Section 3, we derive the linear system using the Galerkin method. We then present the basis functions in Section 4 and the mesh generation method in Section 5. Section 6 describes the conditions under which the numerical experiments were conducted and the results are summarized. The appendices provide details on the method of ICCG, and the *Mathematica* script we used to visualize three dimensional data.

2 Problem Statement

The linear diffusion equation is typically used for description of heat or particle transfer. It is given by:

$$\alpha(\vec{r},t)\frac{\partial u(\vec{r},t)}{\partial t} = \nabla.\Big(D(\vec{r},t)\nabla u(\vec{r},t)\Big) + f(\vec{r},t), \quad \vec{r} = (x,y,z) \in \Omega, \quad (1)$$

where u is an unknown function (temperature or density), Ω is the problem domain and D, f, α are functions specified by the problem domain. The initial condition is $u(\vec{r}, t = 0) = u_0(\vec{r})$. The equation has to be provided with a boundary condition. Typically this is either the Dirichlet condition ,

$$u(\vec{r},t) = g(\vec{r},t), \quad \vec{r} \in \Gamma, \quad (2)$$

where Γ is boundary of Ω and g is some given function defined on Γ; the Neumann condition ,

$$D(\vec{r},t)(\vec{n}\cdot\nabla u(\vec{r},t)) = 0, \tag{3}$$

where $\vec{n}$ is unit vector orthogonal to the Γ; or the mixed condition ,

$$D(\vec{r},t)(\vec{n}\cdot\nabla u(\vec{r},t)) + cu(\vec{r},t) = g(\vec{r},t), \tag{4}$$

where $c \geq 0$. In general, equation (1) has to be solved numerically, but for some cases it is possible to obtain an analytical solution. For example, when the problem domain Ω is three dimensional space, the conductivity D and capacity α are equal to one, the source f is zero and the initial condition is $u_0(\vec{r}) = Q\cdot\delta(\vec{r})$, where δ the Dirac delta function, the solution is well known [2] and is given by

$$u(\vec{r},t) = \frac{Q}{8(\pi Dt)^{\frac{3}{2}}}\exp\left(-\frac{r^2}{4Dt}\right). \tag{5}$$

This is the so called instantaneous point source problem. It can be used for a description of heat transfer after some amount of energy equal to Q is deposited instantly at the origin.

It was already mentioned that the goal of this paper is to solve (1) on a nonuniform mesh using the Galerkin method. As a benchmark the point source problem was used. Since, it is impossible to represent an infinite domain on the computer, the problem was solved inside a cube with a source in the corner and the Neumann boundary condition everywhere on the cube faces. If the elapsed time in the problem is not big, so that temperature spread is less than cube size, equation (5) should be good approximation of exact solution.

3 Galerkin finite element scheme with explicit time stepping

In this section the Galerkin method [8] and time discretization are briefly described. Let us first discuss time discretization.

We applied the following scheme to solve the diffusion equation. Label

the time steps by n. Define t^{n-1} and t^n to be the time respectively at time steps $n-1$ and n. Let u^{n-1} and u^n be the corresponding solutions at these time steps. Let f^{n-1} and f^n be the values of function f at these time steps. For simplicity, D and α do not depend on time. Then the following equations will linearly interpolate these variables between time steps $n-1$ and n:

$$f_\theta = f^{n-1}(1-\theta) + f^n\theta, \quad u_\theta = u^{n-1}(1-\theta) + u^n\theta \tag{6}$$

where the interpolating factor is given by

$$\theta \equiv \frac{t - t^{n-1}}{\triangle t}, \quad \triangle t = t^n - t^{n-1}.$$

The diffusion equation, namely equation (1), in terms of the new variables is,

$$\alpha\frac{u^n - u^{n-1}}{\triangle t} = \nabla.\Big(D\nabla u_\theta\Big) + f_\theta. \tag{7}$$

When $\theta = 1$ this is the implicit scheme. The explicit scheme is given by $\theta = 0$. When $\theta = 1/2$ it is called the Crank-Nicolson scheme. In our calculations the implicit scheme was used. Although it has first order of accuracy in time, it is simple and unconditionally stable. Rearrangement of equation (7) gives the following result:

$$\alpha\frac{u^n}{\triangle t} - \nabla.\Big(D\nabla u_\theta\Big) = \tilde{f}^n, \tag{8}$$

where $\tilde{f}^n \equiv f^n + \alpha\frac{u^{n-1}}{\triangle t}$.

Now Galerkin's method can be applied. This method can be described as follows. First, we choose some set of linearly independent functions $\phi_j(\vec{r})$ and expand our unknown function $u(\vec{r})$ over this set,

$$u_{fe}(\vec{r}) = \sum u_j\phi_j(\vec{r}), \tag{9}$$

where u_j are expansion coefficients and subscript fe stands for finite element. Obviously $u_{fe}(\vec{r})$ is not equal to $u(\vec{r})$, but if we have reasonable choice of ϕ, which are often called basis functions, u_{fe} will be a good approximation of u. This method allows to reduce the original partial differential equation with unknown function $u(\vec{r})$ to a set of algebraic

equation with a vector of unknown u_j. The question is how this reduction should be done.

After substitution of expression (9) into equation (7) one may define the residual as,

$$R[u_{fe}] \equiv \alpha \frac{u^n_{fe}}{\triangle t} - \nabla.\Big(D\nabla u^n_{fe}\Big) - \tilde{f}^n. \tag{10}$$

Now, u_{fe} is said to be finite element solution of diffusion equation if the residual vanishes with respect to the basis functions, that is:

$$\int_\Omega \phi_i R[u_{fe}] d\vec{r} = 0. \tag{11}$$

Substitution of equation (10) into (11) gives the following system of equations,

$$\int_\Omega \alpha \frac{u^n_{fe}}{\triangle t}\phi_i d\vec{r} - \int_\Omega \phi_i \nabla.\Big(D\nabla u^n_{fe}\Big) d\vec{r} = \int_\Omega \phi_i \tilde{f}^n d\vec{r}. \tag{12}$$

For convenience the index fe will be dropped from now onwards. Applying the formula $\nabla.(uf\nabla v) = u\nabla.(f\nabla v) + f\nabla u.\nabla v$ and Gauss' theorem, equation (12) is reduced to,

$$\int_\Omega \alpha \frac{u^n}{\triangle t}\phi_i d\vec{r} + \int_\Omega D(\nabla\phi_i \cdot \nabla u^n) d\vec{r} - \int_\Gamma \phi_i D\nabla u^n.\vec{n} d\vec{S} = \int_\Omega \phi_i \tilde{f}^n d\vec{r}. \tag{13}$$

Using equation (9), equation (13) is transformed into set of ordinary algebraic equations,

$$M\frac{\vec{u}^n}{\triangle t} + A\vec{u}^n = \vec{F}^n, \tag{14}$$

where $\vec{u}^n$ is a vector of unknown coefficients u_j. M is a mass matrix given by

$$M_{ij} = \int_\Omega \alpha\phi_i\phi_j d\vec{r}. \tag{15}$$

Matrix A is called stiffness matrix and equals to

$$A_{ij} = \int_\Omega D(\nabla\phi_i \cdot \nabla\phi_j) d\vec{r} + \int_\Gamma \psi_{ij}(\vec{r}) d\vec{S}, \tag{16}$$

where $\psi_{ij} = 0$ for Neumann and Dirichlet boundary conditions, $\psi_{ij} = c\phi_i\phi_j$ for the mixed conditions. Finally, $\vec{F}$, which is called load vector, is defined as:

$$F_i = \int_\Omega f\phi_i d\vec{r} + \int_\Gamma \xi_i dS + \int_\Omega \alpha\phi_i \frac{u^{n-1}}{\triangle t} d\vec{r}, \tag{17}$$

where $\xi_i = 0$ for Dirichlet and Neumann boundary conditions and $\xi_i = \phi_i g$ for the mixed boundary condition. It is easy to see that the final matrix $A + M/\triangle t$ is symmetric and it can be proved to be positive definite. For any general set of basis functions this matrix will be dense. A good decomposition of the problem domain coupled with compactly supported basis functions can reduce the CPU time needed to solve the resulting linear system. For the point source problem, a hexahedral mesh was chosen. Each corner (node) of each hexahedron is assigned a global node number. There will be one continuous basis function associated with every node and it will have the same index as the node. These functions are equal to zero everywhere except inside the hexahedra that have this node as one of their corners. The total number of basis functions is equal to number of nodes and the basis functions are linearly independent. Each hexahedron in the mesh has 26 nearest neighbors. The resulting matrix will be sparse and have 27 diagonals (each node is coupled to itself). This will allow us to solve equation (14) using iterative techniques [1]. The method used in our calculations is described in Appendix A.

In some cases, the mass matrix M is replaced with a diagonal lumped mass matrix,

$$M_{lumped} = \delta_{ij} \int_\Omega \alpha \phi_i d\vec{r}, \tag{18}$$

where δ_{ij} is Kronecker delta. There is empirical evidence that calculations using a lumped mass matrix tend to be more accurate. We used a diagonal lumped mass matrix in this work.

After evaluation of the mass and stiffness matrices and the load vector advancing in time begins.

Once $\vec{F}$, M, and A have been evaluated we use equation (6) to advance the time variable. There is no requirement on $\triangle t$ from stability condition. Thus, $\triangle t$ may be modified every step depending on how fast the solution is changing.

4 Basis Functions

In this section we will give a short description of our choice of basis functions, how it affects the calculation of matrix elements and how it was related to mesh construction.

We have already specified general properties of basis functions. In practice the diffusion equation is often solved with the so called hat functions. These are defined on the unit cube as follows:

$$\begin{array}{cccc} j & (\ \xi, & \psi, & \theta\ ) & \phi_j \\ 1 & (\ -1, & -1, & -1\ ) & (1-\xi)(1-\psi)(1-\theta)/8 \\ 2 & (\ 1, & -1, & -1\ ) & (1+\xi)(1-\psi)(1-\theta)/8 \\ 3 & (\ 1, & 1, & -1\ ) & (1+\xi)(1+\psi)(1-\theta)/8 \\ 4 & (\ -1, & 1, & -1\ ) & (1-\xi)(1+\psi)(1-\theta)/8 \\ 5 & (\ -1, & -1, & 1\ ) & (1-\xi)(1-\psi)(1+\theta)/8 \\ 6 & (\ 1, & -1, & 1\ ) & (1+\xi)(1-\psi)(1+\theta)/8 \\ 7 & (\ 1, & 1, & 1\ ) & (1+\xi)(1+\psi)(1+\theta)/8 \\ 8 & (\ -1, & 1, & 1\ ) & (1-\xi)(1+\psi)(1+\theta)/8, \end{array} \tag{19}$$

where j is a local node number, ξ, ψ, θ are coordinates of the location of cube nodes, ϕ_i in the third column are basis functions. It is easy to see that the basis functions are equal to unity at the corresponding nodes and zero on the opposite faces of the cube. This choice of functions will give us second order accuracy.

The mapping from the problem space x, y, z to the parametric space ξ, ψ, θ is accomplished by the following equations:

$$\begin{aligned} \vec{r}(x,y,z) &= \sum_{j=1}^{8} \vec{r}_j \phi_j(\xi,\psi,\theta) \\ u(x,y,z) &= \sum_{j=1}^{8} u_j \phi_j(\xi,\psi,\theta), \end{aligned} \tag{20}$$

where summation is done over the nodes of the hexahedron we are mapping, $\vec{r}_j$ contains the x, y, z coordinates of node j and u_j is value of the function u at node j. This transformation is called isoparametric mapping and it simplifies the calculation of matrix and vector elements of the equation (14). Each integral in equations (15,16,17) is evaluated in ξ, ψ, θ space. This procedure is described in Appendix B.

5 Mesh Construction

The construction of the mesh is dictated by the specifics of the problem. The point source problem has a symmetric solution. This allows us to use a simple cubic domain with one corner situated at the origin of the coordinate system. The mesh in Figure (1) was generated by a randomized subdivision process. Regular connectivity was retained between nodes, giving the following expression for the global node number of each node:

$$i = k + l(N+1) + m(N+1)(N+1), \tag{21}$$

where i is global node number, k, l, m are indices along axes. k, l, m correspond to the x, y, z axes in Figure 1. All of them are changing from 0 to N, giving $(N+1)^3$ global nodes.

The mesh was constructed recursively by subdividing each hexahedron into eight random hexahedrons. This is accomplished by adding a new node at a random location along each edge of the hexahedron. A $13th$ node is added at a random location inside the hexahedron. Finally, these new nodes are connected to form eight new hexahedra. Since calculations in x, y, z space could be difficult, the parametric map of each node was used. Nodes are generated according to the formula:

$$\begin{aligned} t_{12} &= s t_1 + (1-s) t_2, \\ s &= f + (1-2f) \quad random(), \end{aligned} \tag{22}$$

where t is ξ, ψ or θ, t_1 and t_2 are nodes connected by and edge, t_{12} is the new node on that edge, $random()$ is a subroutine generating random numbers uniformly distributed from 0 to 1, and f controls the amount of randomness in the resulting mesh. It is easy to see that if $f = 0.5$ the grid is structured and the more it deviates from 0.5 the more random is our mesh. In the Figure (1) we present pictures describing mesh that was generated with this method.

f can not deviate too much from 0.5. The parametric mapping (20) must establish a one-to-one relation between points in different spaces. This requirement can be violated if the grid is distorted. Two dimensional triangular and quadrilateral elements can be checked for unstable numerical behavior by insuring that the Jacobian is positive at the nodes. However, [6] shows that there is no such simple test for the 3D case and even

more elaborated tests do not give absolute assurance. The good news is that requirement of positiveness of the Jacobian at the nodes works approximately in 99% of the cases. We used this test in our calculations and it showed that the parameter f could not deviate from 0.5 more then approximately by 0.1. See Appendix B for a description of how the Jacobian was calculated. The mesh shown in Figure (1) is produced with a limiting value $f = 0.39$.

6 Simulations

We investigated how the Galerkin method reproduces analytical solutions, how numerical error changes with discretization, and what is most important for problems in physics - conservation of general properties like symmetry. The point source problem has been chosen to investigate the above properties. The analytical solution (5) has spherical symmetry. Thus, the source was placed at the origin, in the corner of a unit cube with side length equal one. The boundary condition was of Neumann type everywhere. In this case the cube represents one eighth of the original problem. The numerical solution should be well behaved if the diffusion is calculated over a short time period. This ensures that the spreading is less than the cube size. Under this condition, we can compare the numerical solution with the analytical solution given in equation (5).

The initial condition was described by the function $u_0(\vec{r})$. This function was non-zero at the origin and zero everywhere else. The value at the origin set to some number q. When the numerical and analytical solutions were compared, parameter Q of the analytical solution (8) was calculated as $8 \int\limits_{cube} u_0(\vec{r})d\vec{r}$. The results were compared using l^2 norm :

$$\| u \|_{l^2} = \sqrt{\sum_j u_j^2}$$

The point source problem is a difficult test since initially the solution has derivatives with large magnitude, so it was necessary to let the temperature (or density) spread a little bit before any comparisons.

Three dimensional calculations are compute intensive. The largest meshes

used in this work had 35937 nodes which corresponds to $N = 32$. This has prevented us from tracking how numerical error approaches zero with growth of number of nodes, but we did see that increasing N from 16 to 32 changed the error approximately two times. This is consistent with the accuracy expected of the Galerkin method: second order local and first order global.

After the testing we concluded that Galerkin method performed very well indeed. It allowed us to reproduce the analytical solution with decent accuracy and conserve it's important properties even on a nonuniform mesh.

To illustrate this, pictures describing the results of simulation on the grid with parameter $f = 0.39$ are presented. The mesh itself is presented in Figure (1). Part (a) of the figure shows grid lines on the faces of the cube. part (b) shows grid surfaces when some of the indices were fixed. Since the process of mesh construction is random there are an infinite number of meshes with the same magnitude of distortion characterized by parameter f. This one is presented because the next pictures show calculations on this particular mesh. The initial condition was q equal to 30, which gave "energy" $Q = 6.41 \cdot 10^{-4}$ after integration.

Figure 2 is a graph of the numerical solution at time $t = 0.021$ along coordinate axes x, y, z. We see that these curves lay on top of each other and it is an indication of solution symmetry.

Figure 3 is a plot of the numerical and analytical solutions along axis z at time $t = 0.021$. The error is 6.31%. This error is only .30% more than the same simulation conditions on a uniform mesh. The greatest difference between the solutions is at the origin, attesting to the difficulty of handling point sources in numerical calculations.

7 Visualization

Figures 3-6 are two and three dimensional contour plots. This type of data visualization was not built into the tools available to us. A *Mathematica* program [9] was developed to cope with 3D data. The technique we used is described in appendix C. To illustrate mesh distortion, contour plots are presented in klm space also.

Figure (4) is similar to Figure (2), but these are overlayed two dimensional contour plots on some of the cube faces. Figure (5) is of the same type as previous one but it shows also how contour lines look in space of indices.

Figure (6a,b) is three dimensional contour plots in k, l, m and x, y, z spaces. One can clearly see that the solution possesses spherical symmetry. The small deviations for the outer surface is due to the influence of the boundary condition.

These results are very important since one of the basic principles of computer simulations is that numerical solution has to have the same general properties as the analytical solution even if the numerical error is large. Only in this case may one hope to get the correct picture of physical processes.

8 Conclusion

We presented simulations of point source problem on nonuniform hexahedron meshes in three dimensions with up to 36937 nodes in a unit size cube using Galerkin's finite element method. Grids were generated with random recursive subdivision of the cube. The linear algebraic system obtained after applying Galerkin's method was solved using the incomplete Cholesky conjugate gradient iterative technique. Visualization of the data was created in *Mathematica* by creating a small program for that purpose.

We have shown that the diffusion solver is robust and accurate on highly distorted meshes. The ICCG algorithm provides a stable solver for the ill-conditioned linear system produced by distorted meshes. Indeed, numerical error on the distorted meshes was about the same as for the uniform meshes. The numerical solution exhibited expected spherical symmetry despite mesh irregularity.

Appendices

A. Numerical Solution of Equation $A\vec{x} = \vec{b}$

Many numerical methods applied to equation (1) finally require solution of linear system :

$$A\vec{x} = \vec{b}. \tag{23}$$

Matrix A in this case is quite special and has following properties :

- It is sparse. By sparsity we mean that the ratio of the number of zero elements to total number of matrix elements approaches unity as size of the matrix grows.
- It is symmetric ($A = \{a_{ij}\}$, $a_{ij} = a_{ji}$).
- It is positive definite ($(\vec{x}, A\vec{x}) \geq 0$. This product is equal to zero only if $\vec{x} = 0$.).
- It is ill-conditioned (Matrix is ill-conditioned when its condition number $\| A \| \cdot \| A^{-1} \|$ is much greater then one).

It is also desirable that matrix A have the M property. Definition for an M-matrix is

$$a_{ij} \leq 0 \quad \text{for} \quad i \neq j, det(A) \neq 0 \quad \text{and} \quad A^{-1} \geq 0.$$

Since solution of equation (1) is essentially positive (it is temperature or concentration), one should try to have numerical solution with the same property. This is possible if A is an M-matrix.

Typically this type of equation is solved by iterative methods [1], very often it is the conjugate gradient method. This technique can be a perfectly good choice for problems on regular grids, but it turns out that introduction of irregularity may significantly increase the condition number which defines the convergence rate [3]. This issue is often addressed with a use of preconditioning. The idea is simple : the matrix is transformed to some new coordinate system where the condition number will be smaller. However it has to be an inexpensive procedure since our final goal is solving equation (23) as fast as possible.

In our calculations incomplete Cholesky preconditioning [7] was used. It

is based on a factorization of the matrix into upper and lower triangular parts. This algorithm was modified so that :

$$C = LDL^T \approx A, \tag{24}$$

where D is a diagonal matrix with $D_{jj} = 1/A_{jj}$; L is a lower triangular matrix with $L_{ij} = 0$ if $A_{ij} = 0$ and the rest of the elements are evaluated according to the original Cholesky method; L^t is the transpose of L. Usage of D is not required, but it allows us to avoid the evaluation of square roots.

If we had used a complete factorization, this would have solved the problem, but it also would have been very expensive. Therefore, the elements of triangular matrices are evaluated only if the elements of the original matrix are non-zero. This also preserves sparsity of the matrix so important for iterative techniques.

In this case algorithm for conjugate gradient as follows :

$$\begin{aligned}
\vec{x}^{k+1} &= \vec{x}^k + \alpha^k \vec{d}^k, \\
\vec{R}^{k+1} &= \vec{R}^k - \alpha^k A\vec{d}^k, \ \vec{R}^k = \vec{b} - A\vec{x}^k, \\
\vec{d}^{k+1} &= C^{-1}\vec{R}^{k+1} + \beta^k \vec{d}^k, \ \vec{d}^0 = C^{-1}\vec{R}^k, \\
\alpha^k &= (\vec{R}^k, C^{-1}\vec{R}^k)/(\vec{d}^k, A\vec{d}^k), \\
\beta^k &= (\vec{R}^{k+1}, C^{-1}\vec{R}^{k+1})/(\vec{R}^k, C^{-1}\vec{R}^k).
\end{aligned} \tag{25}$$

Vector $\vec{R}$ is called the residual and $\vec{d}$ is the search direction. The iterations are repeated until the relative change $\| \vec{x}^{k+1} - \vec{x}^k \| \ / \ \| \vec{x}^k \|$ becomes smaller than some number, which is usually about 10^{-13}.

B. Numerical Integration

All integrals were calculated in ξ, ψ, θ space. For this purpose we evaluate of Jacobian matrix :

$$J = \begin{vmatrix} \partial x/\partial\xi & \partial y/\partial\xi & \partial z/\partial\xi \\ \partial x/\partial\psi & \partial y/\partial\psi & \partial z/\partial\psi \\ \partial x/\partial\theta & \partial y/\partial\theta & \partial z/\partial\theta \end{vmatrix},$$

where x, y and z are calculated according to (20). The Jacobian is used for the transformation of the gradient via $\nabla\phi(x,y,z) = J\nabla\phi(\xi,\psi,\theta)$ and for defining the differential space element via $dxdydz = det(J)d\xi d\psi d\theta$.

Theoretically all integrals can be evaluated analytically. Analytic evaluation using Mathematica gave very long expressions. Therefore they were calculated numerically. One can do this if numerical error introduced by integration is of the same order as Galerkin method. Hence Gaussian quadrature of second order will suffice. All integrals are substituted by the sum :

$$\int_{cube} f(\xi,\psi,\theta)d\xi d\psi d\theta = \sum_j f(\xi_j,\psi_j,\theta_j),$$

where evaluation points are :

$$(\xi_j,\psi_j,\theta_j) = (\pm 1/\sqrt{3}, \pm 1/\sqrt{3}, \pm 1/\sqrt{3}).$$

Since matrix A is symmetric, only half of its elements have to be evaluated.

C. Visualization

Since three-dimensional calculations are not quite typical because of their computational requirements, there are not many packages allowing interpretation and visualization of 3D data, especially on an irregular grid. We decided to use *Mathematica* - a computer algebra system from Wolfram Research [9]. Although it does not have direct ability to work with data on a nonuniform mesh, its unique handling of graphics and powerful language made our goal feasible.

Mathematica's graphical capabilities are organized in a highly modular way. Any data, whether it is analytical or numerical, is initially transformed to a set of display independent graphical primitives like point, line, or polygon. These primitives are translated into an appropriate graphical form, for instance Postscript. For the ordinary user these steps are usually hidden, but if necessary it is possible to save the intermediate results. This feature is essential for our solution to the visualization problem.

Now let us state clearly what we have and what we want to do. As a result of the numerical calculations we have an array of data representing the numerical solution of the diffusion equation at the grid nodes.

This grid is a structured cubic mesh in the space of indices k, l, m, but it doesn't have this property in actual space x, y, z. Our purpose is the construction of contour plots in x, y, z space.

Mathematica has built-in or loadable functions that can make contour plots on a regular grid. Using them it is possible to generate necessary graphs in k, l, m space and, if intermediate results are saved, this will give us their representation in terms of graphical primitives. These primitives use plot coordinates, but not display coordinates. For example, in a 3D plot there is a line from the origin (0,0,0) to the point (1,1,1). In this case it will be described in a following way : `Line[{{0,0,0},{1,1,1}}]`. Therefore, if we could substitute indices k, l, m by their respective x, y, z coordinates in the graphics primitives, this would give us the desired plot. It turns out that using the *Mathematica* language it can be done in a rather elegant way.

Here we present the program used for producing 3D contour surface plots in Figure (6). The same method was applied for making 2D contour plots in the Figures 4,5. Here is the program. Verbatim environment

```
 1 DataContours = { 10^-3, 10^-5, 10^-8 };
 2 DataColors = { {GrayLevel[0.25]},{GrayLevel[0.55]},
                  {GrayLevel[0.85]} };
 3
 4 ud = ReadList["u.dat",Number];
 5 grid = ReadList["mesh.dat",{Number,Number,Number}];
 6
 7 NumberOfNodes = Length[ud];
 8 imx = NumberOfNodes^(1/3);
 9
10 M = Compile[ {j}, Floor[j/(imx*imx)] ];
11 L = Compile[ {j}, Floor[(j - M[j]*imx*imx)/imx] ];
12 K = Compile[ {j}, (j - L[j]*imx - M[j]*imx*imx) ];
13
14 X=Interpolation[Table[{K[i],L[i],M[i],grid[[i+1,1]]},
                        {i,0,NumberOfNodes-1}]];
15 Y=Interpolation[Table[{K[i],L[i],M[i],grid[[i+1,2]]},
                        {i,0,NumberOfNodes-1}]];
16 Z=Interpolation[Table[{K[i],L[i],M[i],grid[[i+1,3]]},
                        {i,0,NumberOfNodes-1}]];
17 U=Interpolation[Table[{K[i],L[i],M[i],ud[[i+1]]},
                        {i,0,NumberOfNodes-1}]];
```

```
18
19 TransformPoint[ r_List ] := { Apply[X,r], Apply[Y,r],
                                 Apply[Z,r] };
20
21 <<Graphics`ContourPlot3D`
22 KLMplot = ContourPlot3D[ U[k,l,m],{k,0,imx-1},{l,0,imx-1},
                                          {m,0,imx-1},
23       MaxRecursion->2,
24       Lighting->False,
25       Contours->DataContours,
26       ContourStyle->DataColors,
27       DisplayFunction -> Identity,
28       Axes->True ];
29
30 XYZplot = KLMplot/.{p:_Polygon:>Map[TransformPoint,p,{2}]};
31
32 Save["output_data.m",KLMplot,XYZplot];
```

Numbers on the left are line numbers and are not part of the program.

Let us go through the text and explain its meaning. On the first two lines lists of contour values and their respective intensities of gray are created. The higher the value of the function the darker the surface. Next, two `ReadList` commands are used for reading data from files. Values of the function u are stored in the file "u.dat" as one column. Each number represents function value at some global node. First number describes node 0, next is node 1 and so on. The same is true for the file "mesh.dat", where the coordinates of the nodes are stored, except that there are three columns : x, y, z. After the data is read, the variable `ud` is a list of function u values and `grid` is a list of node coordinates (x, y, z). In order to minimize input to the program the number of nodes is calculated from the length of data array (line 7). The variable `imx` represents the maximum value for the local index k, l or m. On lines 10-12 we define functions for calculating local indices from a global one. Directive `Compile` means our functions will operate only on numerical input. This should speed up their execution. Next, function `Interpolate` is used to construct functions that will take indices k, l, m and give us point coordinates or the value of the numerical solution. On line 19 we define the transformation function `TransformPoint` that maps a point from k, l, m space to x, y, z. After that, *Mathematica* package `ContourPlot3D` is loaded and one of its

functions is used to produce a plot of contour surfaces in k, l, m space and save it as `KLMplot`. On line 30 this plot is mapped to x, y, z space and after that we save both variables in the file "output_data.m". Later they can be loaded in again to *Mathematica* and redisplayed.

Since all the "magic" happens on line 30, it deserves a more thorough description. Variable `KLMplot` is the graphics object containing a set of graphics primitives plus options specifying how the plot should be displayed. Displaying this variable would give us the plot in Figure 6a. *Mathematica* uses primitive `Polygon` for the presentation of surfaces. Our purpose is to change the coordinates of these polygons by applying the mapping described by the function `Transform Point`.

Before we explain how it happens let us mention the powerful and essential idea of an *expression*. There are many different objects in *Mathematica*, but they are all viewed as expressions. One may consider them as some general objects that have parameters, like `x,y` in `f[x,y]` and tags - `f`. These tags are called the heads of the *expressions*. Depending on what head an *expression* has, it's parameters may be viewed as arguments or elements. In the first case, the whole *expression* is a function. In the latter case, the *expression* can be a list and so on. Of course *expressions* can be nested, so that function parameters are *expressions* themselves. In this case we may think about levels in *expressions*.

An understanding of *expressions* is needed to make the command on line 30. The notation `XYZplot=KLMplot/.{ }` means that *Mathematica* has to apply to the object `KLMplot` some rule described inside of the brackets `{}`. The result of this is assigned to the variable `XYZplot`. The rule should be applied to any *expression* with the head `Polygon` which is referred further with a name `p` (`p:_Polygon`). Once this object is found, it should be transformed according to the function on the right side of the command `:>` . This command means that the function should be evaluated not before the search, but when the required *expression* is found. This is necessary because transformation function has the input parameter : `p`. The action it performs can be easily understood from its name - `Map`. Since point coordinates k, l, m are on the second level of nesting, there is a parameter `{2}` saying that `TransformPoint` has to be applied on the second level of the `Polygon` object.

References

[1] O. Axelsson, " Iterative Solution Methods", Cambridge University Press, 1996.

[2] E. Butkov, " Mathematical Physics", Addison-Wesley, 1968.

[3] I. Fried, Condition of finite element matrices generated from nonuniform meshes, AIAA J. 10(1972), p. 219, 1972.

[4] J.O. Hallquist and D.W. Stallman, VEC/DYNA3D users manual (nonlinear dynamic analysis of structures in three dimensions), Livermore Software Technology Corperation Report 1018, 1990.

[5] D.S. Kershaw, Incomplete Cholesky conjugate gradient method for the iterative solution of linear equations, J. Computational Physics, 26(1978).

[6] P.M. Knupp, On the invertibility of the isoparametric map, Comp. Meth. in Appl. Mech. and Engg., 78(1990), 313-329.

[7] J.A. Meijerink and H.A. van der Vorst, An iterative solution method for linear systems of which the coefficients is a symmetric M-matrix, Mathematics of Computation, 31(137), 1997.

[8] Vidar Thomeé, " Galerkin Finite Methods for Parabolic Problems", Springer-Verlag, New York, 1997.

[9] S. Wolfram, " Mathematica: A System For Doing Mathematics By Computer", Addison-Wesley, 1988.

A. Komashko, D. Laney, M. Prasad and R. Vemuri: Department of Applied Science, University of California at Davis
P.O box 808 L-794 Livermore, CA 94550, USA.
Email: [komashko@wente, dlaney@, prasad1@, vemuri1@].llnl.gov