gridIt 1.0 User Manual

Bruce P. Ayati

Abstract

gridIt is a two-dimensional linear interpolation program. It takes randomly distributed data of the form x, y, z and interpolates it onto a regular rectangular grid.

Key words and phrases: Linear Interpolation, Two Dimensions.

AMS Subject Classifications (1991): 65D05.

1 Introduction

This document is both a user manual and mathematical description of the two dimensional interpolation program **gridIt**. **gridIt** was designed as an alternative to methods using Delaunay triangulations and Voronoi cells. These methods can do a poor job with data that come from a finite element triangulation. **gridIt** allows gridding of such data without delving into the structure of the mesh. The algorithm used in **gridIt** can be generalized to higher dimensions. Since gridIt penalizes jumps in slope, it will not do a good job interpolating functions that have a crease or other significant non-smooth parts. If the location of the slope discontinuities is known in advance, gridIt can be modified to not penalize those areas. gridIt solves a linear system using a conjugate gradient method. Attempts to interpolate a low number of data points may cause an increase in the number of iterations due to rounding error.

1.1 Disclaimer

gridIt is a research tool, not a commercial product. It is made available to other researchers subject to the following restrictions and disclaimers.

2 How to Use gridIt

The executable for **gridIt** has the command line synopsis:

gridIt {data} [{grid}]

The file **data** contains a list of numbers in the form:

$$\begin{array}{ccc} \tilde{x}_0 & \tilde{y}_0 & \tilde{z}_0 \\ \tilde{x}_1 & \tilde{y}_1 & \tilde{z}_1 \\ \vdots & \vdots & \vdots \\ \tilde{x}_{d-1} & \tilde{y}_{d-1} & \tilde{z}_{d-1} \end{array}$$

Here $\tilde{z}_i = g(\tilde{x}_i, \tilde{y}_i)$, where g is the function to be approximated. The optional file **grid** contains the description of the grid onto which the data is to be interpolated. It contains seven numbers in the following order: $x_{\min}$, $x_{\max}$, N_x, $y_{\min}$, $y_{\max}$, N_y, and δ. N_x and N_y are the number of nodes in the x-direction and y-direction, respectively. The numbers $x_{\min}$, $x_{\max}$, $y_{\min}$, $y_{\max}$ define the x and y ranges. δ is the accuracy of the data. If there is assumed to be no error in the original data, choose $\delta = 0$. If the file **grid** is not provided, **gridIt** prompts the user for the grid description. For reasons that will be discussed later, **gridIt** can do a better job if N_x and N_y are both odd.

The solution is placed in the file **data.out** using a list format similar to that of **data**. The data triplet, (x_i, y_i, z_i), is associated with a node, v_i, as in figure 1. The ordering begins in the lower left corner, then works its way up by going along the x-direction.

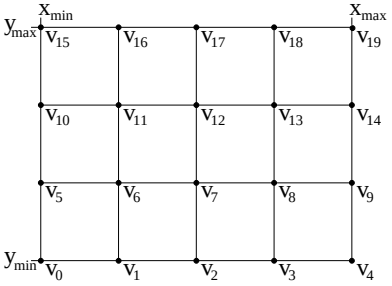

Figure 1: Ordering of nodes on a sample grid; $N_x = 5$, $N_y = 4$.

The hidden file, **.data.clean**, contains the data triplets in **data** whose x and y coordinates satisfy $\tilde{x} \in [x_{\min}, x_{\max}]$, $\tilde{y} \in [y_{\min}, y_{\max}]$.

3 How gridIt Works

gridIt solves the problem

$$\min_u \left\{ \sum_{\substack{0<i<N_x-1 \\ 0<j<N_y-1}} (\varepsilon_x(\partial_x^2 u)_{ij}^2 + \varepsilon_y(\partial_y^2 u)_{ij}^2)\Delta x \Delta y + \frac{|\Omega|}{d} \sum_{i=0}^{d-1} |u(\tilde{x}_i, \tilde{y}_i) - \tilde{z}_i|^2 \right\}, \qquad (1)$$

where $f_{ij} = f(x_{\min} + i\Delta x\,, y_{\min} + j\Delta y)$, $\Omega = [x_{\min}\,, x_{\max}] \times [y_{\min}\,, y_{\max}]$, and d is the number of data points to be interpolated. The difference operators are defined as

$$(\partial_x^2 u)_{ij} = \frac{1}{2(\Delta x)^2}\left[u_{i+1\,,j} - 2u_{i\,,j} + u_{i-1\,,j}\right], \tag{2}$$

$$(\partial_y^2 u)_{ij} = \frac{1}{2(\Delta y)^2}\left[u_{i\,,j+1} - 2u_{i\,,j} + u_{i\,,j-1}\right]. \tag{3}$$

Based on the following rationale, gridIt chooses ε_x and ε_y. From approximation theory [1], if $g(x\,, y)$ is the true solution and u is its linear interpolant,

$$\|g - u\|^2_{L^2(\Omega)} \approx \frac{1}{120}\left[(\Delta x)^4 \int_\Omega (g_{xx})^2 + (\Delta y)^4 \int_\Omega (g_{yy})^2\right] + \delta^2|\Omega|, \tag{4}$$

where δ is the average error in the original data, $\tilde{z}_i$. We want to weigh the penalty on the second derivatives appropriately, so that

$$\|g - u\|^2_{L^2(\Omega)} \approx \varepsilon \int_\Omega \left((g_{xx})^2 + (g_{yy})^2\right) + \varepsilon_1 \int_\Omega (g_{xx})^2 + \varepsilon_2 \int_\Omega (g_{yy})^2. \tag{5}$$

So we choose

$$\varepsilon = \frac{\delta^2|\Omega|}{\int_\Omega \left((g_{xx})^2 + (g_{yy})^2\right)}, \qquad \varepsilon_1 = \frac{(\Delta x)^4}{120}, \qquad \varepsilon_2 = \frac{(\Delta y)^4}{120}. \tag{6}$$

Thus, we want to solve the problem

$$\min_g \left\{ \left[(\varepsilon + \varepsilon_1) \int_\Omega (g_{xx})^2 + (\varepsilon + \varepsilon_2) \int_\Omega (g_{yy})^2\right] + \frac{|\Omega|}{d} \sum_{i=0}^{d-1} |g - \tilde{z}_i|^2 \right\}. \tag{7}$$

The discretization of this, with $\varepsilon_x = \varepsilon + \varepsilon_1$ and $\varepsilon_y = \varepsilon + \varepsilon_2$, is (1).

Problem (1) can be restated in matrix form as the least squares problem

$$\min_U \|BU - R\|_2^2, \tag{8}$$

where $\|\cdot\|_2$ is the Euclidian norm. The matrix B and vector R are given by

$$B = \begin{pmatrix} \sqrt{e_x} A_1 \\ \sqrt{e_y} A_2 \\ M \end{pmatrix}, \qquad R = \begin{pmatrix} 0 \\ 0 \\ r \end{pmatrix}, \tag{9}$$

where $e_x = d\varepsilon_x/|\Omega|$ and $e_y = d\varepsilon_y/|\Omega|$.

The $MU - r$ term in the minimization reflects the sum in (7). M has size $d \times N$, where d is the number of data points that lie inside the grid boundary and $N = N_x \cdot N_y$ is the number of nodes in the grid. Every data point is inside one of the "boxes" in the

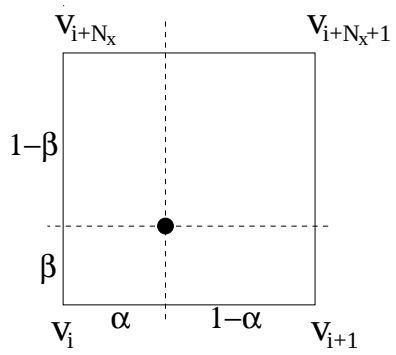

Figure 2: A data point is associated with a particular node.

grid, and is associated with the lower left node of that box. For each of the d data points, there is a corresponding row in the matrix M of the form:

$$[\ 0 \quad \cdots \quad 0 \quad p_1 \quad p_2 \quad 0 \quad \cdots \quad 0 \quad p_3 \quad p_4 \quad 0 \quad \cdots \quad 0\ ]. \tag{10}$$

If the data point is associated with the ith node, then p_1 is in column i, p_2 is in column $i+1$, p_3 is in column $i+N_x$ and p_4 is in column $i+N_x+1$. The values of the $p_i, i = 1 \ldots 4$, are determined by the ratio of their distance to their associated node in the x-direction and the y-direction. If α is the offset in the x-direction and β is the offset in the y-direction, then we have the following values for the p_i (fig. 2):

$$p_1 = (1-\alpha)(1-\beta), \tag{11}$$
$$p_2 = \alpha(1-\beta), \tag{12}$$
$$p_3 = \beta(1-\alpha), \tag{13}$$
$$p_4 = \alpha\beta. \tag{14}$$

r is the vector of the z-values of the data points:

$$r = \begin{pmatrix} z_0 \\ z_1 \\ \vdots \\ z_{d-1} \end{pmatrix} \tag{15}$$

The $N \times N$ matrices A_1 and A_2 are used to compute the second differences, equations (2),(3), respectively. As a patch, they also compute third differences on the boundaries. Since the components of R corresponding to A_1 and A_2 are zero, we are placing a penalty on the jumps in slope. A_1 has the following structure, where

$h = 3\Delta x$:

$$A_1 = \frac{1}{2(\Delta x)^2} \begin{pmatrix} \frac{-1}{h} & \frac{3}{h} & \frac{-3}{h} & \frac{1}{h} & & & & & & & \\ 1 & -2 & 1 & & & & & & & & \\ & \ddots & \ddots & \ddots & & & & & & & \\ & & 1 & -2 & 1 & & & & & & \\ & \frac{-1}{h} & \frac{3}{h} & \frac{-3}{h} & \frac{1}{h} & & & & & & \\ & & & & & \frac{-1}{h} & \frac{3}{h} & \frac{-3}{h} & \frac{1}{h} & & \\ & & & & & 1 & -2 & 1 & & & \\ & & & & & & \ddots & \ddots & \ddots & & \end{pmatrix}, \tag{16}$$

A_2 has a similar structure, but with x replaced by y and with a different ordering of rows and a different spacing of non-zero elements within the rows.

For A_1, if the row number, i, corresponds to a node with $x_i \in (x_{\min}, x_{\max})$, row i represents the second difference:

$$\frac{1}{2(\Delta x)^2} \left(U_{i-1} - 2U_i + U_{i+1}\right). \tag{17}$$

If the row number corresponds to a node with $x_i = x_{\min}$ or $x_{\max}$ (i.e. $i \bmod N_x = 0$ or $N_x - 1$), row i represents the third difference:

$$\frac{1}{6(\Delta x)^3} \left(-U_i + 3U_{i+1} - 3U_{i+2} + U_{i+3}\right), \qquad \text{if } i \bmod N_x = 0, \tag{18}$$

$$\frac{1}{6(\Delta x)^3} \left(-U_{i-3} + 3U_{i-2} - 3U_{i-1} + U_i\right), \qquad \text{if } i \bmod N_x = N_x - 1. \tag{19}$$

Similarly, for A_2, row i represents:

$$\frac{1}{6(\Delta y)^3} \left(-U_i + 3U_{i+N_x} - 3U_{i+2N_x} + U_{i+3N_x}\right), \qquad \text{if } i \text{ div } N_x = 0, \tag{20}$$

$$\frac{1}{6(\Delta y)^3} \left(-U_{i-3N_x} + 3U_{i-2N_x} - 3U_{i-N_x} + U_i\right), \qquad \text{if } i \text{ div } N_x = N_y - 1 \tag{21}$$

$$\frac{1}{2(\Delta y)^2} \left(U_{i-N_x} - 2U_i + U_{i+N_x}\right), \qquad \text{otherwise.} \tag{22}$$

The least squares problem, (8), is solved on a coarse grid using conjugate gradient with initial guess $g(x, y) = 0$ and e_x and e_y chosen with $\varepsilon = 0$. We approximate the denominator for ε in (6), using the solution from the coarse grid, as

$$\int_\Omega \left((u_{xx})^2 + (u_{yy})^2\right) \approx \|A_1 U\|_2^2 + \|A_2 U\|_2^2. \tag{23}$$

The solution on the coarse grid is interpolated onto the fine grid and is used as an initial guess for conjugate gradient. The coarse grid is chosen to be one subdivision of the mesh up from the fine grid. Thus, if either N_x or N_y is even, a coarse mesh can not be made, e_x and e_y are chosen with $\varepsilon = 0$, and the initial guess is $g(x, y) = 0$.

4 How to Acquire gridIt

gridIt is currently available via the World Wide Web at URL:

http://www.cs.uchicago.edu/~bruce/software/

5 Source Code

5.1 main.c

```
/* gridIt Version 1.0                                  */
/* January 9, 1996                                     */
/* Copyright 1996 by Bruce Pirooz Ayati.               */
/* See copyright file or user manual for disclaimer. */

#include<stdlib.h>
#include<iostream.h>
#include<fstream.h>
#include "vector.h"

extern
double setEpsilon(vector& v, int N_xC, int N_yC, double dxC, double dyC,
                double dx, double dy, double delta);

extern
void convertData(int N_x, int N_y, int corner[], vector& alpha, vector& beta,
                 vector& rhs,double x_min,double dx,double y_min, double dy);

extern
void conjGrad(int N_x, int N_y, double dx, double dy, int corner[],
              vector& alpha, vector& beta, vector& rhs, vector& v, double eps);


main(int argc, char* argv[])
{
int i,j,d=0;
int N_x,N_y,N_xC,N_yC;
double x_max,y_max,x_min,y_min,dx,dy,dxC,dyC,delta;
double x,y,z;
double eps = 0.0;

ifstream inputStream (argv[1], ios::in);
ofstream inGridStream (".data.clean", ios::out);
if(!inputStream) {cerr << "No such data file." << endl; exit(0);}
if (argc>2){ // read in grid data
```

```
   ifstream gridStream (argv[2], ios::in);
   if(!gridStream) {cerr << "No such grid file." << endl; exit(0);}
   gridStream >> x_min;
   gridStream >> x_max;
   gridStream >> N_x;
   gridStream >> y_min;
   gridStream >> y_max;
   gridStream >> N_y;
   gridStream >> delta;
   gridStream.close();
}
else{ //read in grid information from keyboard
   cout << "Input the grid." << endl;
   cout << "What is x_min?" << endl;
   cin >> x_min;
   cout << "What is x_max?" << endl;
   cin >> x_max;
   cout << "How many points do you want in the x direction?" << endl;
   cin >> N_x;
   cout << "What is y_min?" << endl;
   cin >> y_min;
   cout << "What is y_max?" << endl;
   cin >> y_max;
   cout << "How many points do you want in the y direction?" << endl;
   cin >> N_y;
   cout << "What is the assumed average error in the original data?" << endl;
   cin >> delta;
}

dx=(x_max-x_min)/(N_x-1);
dy=(y_max-y_min)/(N_y-1);

if (dx<=0 || dy<=0) {cerr << "Bad grid, try again." << endl; exit(0);}
else {
  cout << "x:[" << x_min << "," << x_max <<"], dx=" << dx << ", ";
  cout << "y:[" << y_min << "," << y_max <<"], dy=" << dy << endl;
}

//read in raw data in x-y-z format
for(i=0; inputStream >> x >> y >> z; i++){
   if( x<x_max && x>=x_min && y<y_max && y>=y_min )
     {
       inGridStream << x << " " << y << " " << z << endl;
       d++;
     }
}
```

```
inputStream.close();
inGridStream.close();

cout << d << " data points out of " << i << " are in the grid." << endl;

cout << "Interpolating " << d << " data points onto " <<  N_x*N_y
     << " grid points." << endl;

int *corner = new int[d];
vector alpha(d);
vector beta(d);
vector v(N_x*N_y);
vector rhs(N_x*N_y);
rhs.zero();

if( (N_x-1)%2==0 && (N_y-1)%2==0 ){
   cout << "Assembling the Coarse Linear System." << endl;
   N_xC = (N_x-1)/2 + 1;
   N_yC = (N_y-1)/2 + 1;
   dxC=(x_max-x_min)/(N_xC-1);
   dyC=(y_max-y_min)/(N_yC-1);
   cout << "x:[" << x_min << "," << x_max <<"], dx=" << dxC << ", ";
   cout << "y:[" << y_min << "," << y_max <<"], dy=" << dyC << endl;
   vector vC(N_xC*N_yC);
   vector rhsC(N_xC*N_yC);
   vC.zero();
   rhsC.zero();
   convertData(N_xC,N_yC,corner,alpha,beta,rhsC,x_min,dxC,
               y_min,dyC);
   cout << "Solving the Coarse Linear System." << endl;
   conjGrad(N_xC,N_yC,dxC,dyC,corner,alpha,beta,rhsC,vC,eps);
   eps = setEpsilon(vC,N_xC,N_yC,dxC,dyC,dx,dy,delta);
   cout << "Epsilon is " << eps << endl;
   // Interpolate vC onto v for initial guess for conjugate Gradient
   v.zero();
   for(i=0; i<N_x*N_y; i++){
      j = ((i/N_x)/2)*N_xC + (i%N_x)/2;
      if( (i/N_x)%2 == 0 ){
        if( i%2 == 0 ){
          v(i) = vC(j);
          if( i+N_x < N_x*N_y ){
            v(i+N_x) = 0.5*( vC(j) + vC(j+N_xC) );
          }
        }
        else{
          v(i) = 0.5*( vC(j) + vC(j+1));
```

```
          if( i+N_x < N_x*N_y ){
            v(i+N_x) = 0.25*( vC(j) + vC(j+1) + vC(j+N_xC)+ vC(j+N_xC+1) );
          }//if
        }//else
      }//if
   }//for
}//big if
else{
   cout << "Not Able to Coarsen the Grid." << endl;
   v.zero();
}

cout << "Assembling the Linear System." << endl;
convertData(N_x,N_y,corner,alpha,beta,rhs,x_min,dx,y_min,dy);


//Interpolate data points onto the grid using a conjugate gradient method
cout << "Solving the Linear System." << endl;
conjGrad(N_x,N_y,dx,dy,corner,alpha,beta,rhs,v,eps);


//send the gridded data to the output file
ofstream answerStream ("data.out", ios::out);
  for(i=0;i<N_x*N_y;i++){
      answerStream << (i%N_x)*dx+x_min << " " << (i/N_x)*dy+y_min
                   << " " << v(i) << endl;
  }
answerStream.close();


delete[] corner;

cout << "Done." << endl;

}// End Main
```

5.2 conjGrad.c

```
/* gridIt Version 1.0                                  */
/* January 9, 1996                                     */
/* Copyright 1996 by Bruce Pirooz Ayati.               */
/* See copyright file or user manual for disclaimer. */

#include "vector.h"

extern
```

```
void matrixMult(vector& y, vector& v, int N_x, int N_y,
                double dx, double dy, int corner[],
                vector& alpha, vector& beta, double eps);

void conjGrad(int N_x, int N_y, double dx, double dy, int corner[],
              vector& alpha, vector& beta, vector& rhs, vector& v, double eps)
{
int i,j, k=N_x*N_y;
vector s(k), r(k), temp(k), temp2(k), z(k),olds(k);
double t, c, f, coeff;
const double triv = (1.0e-8)*sqrt((rhs,rhs));
const int itnum = k + 10;

   // define the residual: r = rhs - BtBv
   matrixMult(z,v,N_x,N_y,dx,dy,corner,alpha,beta,eps);
   coeff = -1.0;
   vectorLinComb(r,rhs,coeff,z);

   // define the search direction
   s = r;
   c = (r,r);

   // Search until the norm of the residual is less than triv or itnum
   // iterations have been performed, whichever comes first.
   // The functions vectorIncrement and vectorLinComb are used instead
   // of overloaded operators to maintain control over copying of data.
   for(i=0; i < itnum; i++) {
      if(sqrt(c) < triv) break;
        matrixMult(z,s,N_x,N_y,dx,dy,corner,alpha,beta,eps);
        t = c/(s,z);
        vectorIncrement(v,t,s);          // v = v + (t*s);
        coeff = -t;
        vectorIncrement(r,coeff,z);      // r = r - (t*z);
        f = (r,r);
        coeff = f/c;
        olds = s;
        vectorLinComb(s,r,coeff,s);      // s = r + (f/c)*s;
        c = f;
   } // end for

   cout << "Conjugate gradient took " << i << " iterations." <<  endl;
   if (i >= k){ cout << "Maximum number of iterations reached." << endl;}

} // end conjGrad
```

5.3 convertData.c

```
/* gridIt Version 1.0                                  */
/* January 9, 1996                                     */
/* Copyright 1996 by Bruce Pirooz Ayati.               */
/* See copyright file or user manual for disclaimer.  */

#include "vector.h"

void convertData(int N_x, int N_y, int corner[], vector& alpha, vector& beta,
                 vector& rhs, double x_min,double dx, double y_min, double dy)
{//convert xyz data to corner/alpha/beta data and rhs
int i,k,corner_x,corner_y;
double x, y, z;

ifstream dataStream (".data.clean", ios::in);

for(i=0; dataStream >> x >> y >> z; i++){
    for(k=1; k<N_x; k++){
       if( x_min + (k-1)*dx <= x && x < x_min + k*dx ){
           corner_x = k-1;
           alpha(i) = (x - (x_min + corner_x*dx))/dx;
           break;
       }
    }
    for(k=1; k<N_y; k++){
       if( y_min + (k-1)*dy <= y && y < y_min + k*dy ){
           corner_y = k-1;
           beta(i) =  (y - (y_min + corner_y*dy))/dy;
           break;
       }
    }
    corner[i] = N_x*corner_y + corner_x;
    rhs(corner[i]) += z*(1-beta(i))*(1-alpha(i));
    rhs(corner[i]+1) += z*(1-beta(i))*alpha(i);
    rhs(corner[i]+N_x) += z*beta(i)*(1-alpha(i));
    rhs(corner[i]+N_x+1) += z*beta(i)*alpha(i);
}

dataStream.close();

} //End convertData
```

5.4 matrixMult.c

```
/* gridIt Version 1.0                                  */
```

```
/* January 9, 1996                                    */
/* Copyright 1996 by Bruce Pirooz Ayati.              */
/* See copyright file or user manual for disclaimer. */

#include "vector.h"

void matrixMult(vector& y, vector& v, int N_x, int N_y,
                double dx, double dy, int corner[],
                vector& alpha, vector& beta, double eps)
{ //return the vector y=BtBv where B = ( eA1 eA2 M )
   int i, k=N_x*N_y;
   double coeff,temp;
   vector temp2(k);
   y.zero();

   //compute MtMv
   int bsize = beta.size();
   for(i=0;i<bsize;i++){
      //compute temp = Mv
      temp = (1-beta(i))*(1-alpha(i))*v(corner[i])
              + (1-beta(i))*alpha(i)*v(corner[i]+1)
              + beta(i)*(1-alpha(i))*v(corner[i]+N_x)
              + beta(i)*alpha(i)*v(corner[i]+N_x+1);
      //compute y = MtMv
      y(corner[i]) += (1-beta(i))*(1-alpha(i))*temp;
      y(corner[i]+1) += (1-beta(i))*alpha(i)*temp;
      y(corner[i]+N_x) += beta(i)*(1-alpha(i))*temp;
      y(corner[i]+N_x+1) += beta(i)*alpha(i)*temp;
   }

   temp2.zero();

   //compute A1tA1v
   for(i=0;i<k;i++){
      //compute temp1 = A1v then temp2 = A1t*temp1
      if( (i%N_x)!=0 && (i%N_x)!=(N_x-1) ){
        temp = v(i-1) - 2*v(i) + v(i+1);
        temp2(i-1) += temp;
        temp2(i) -= 2*temp;
        temp2(i+1) += temp;
      }
      else if ( (i%N_x)==0 ){
        temp = ( -v(i) + 3*v(i+1) - 3*v(i+2) + v(i+3) )/(3*dx);
        temp2(i) -= temp/(3*dx);
        temp2(i+1) += 3*temp/(3*dx);
        temp2(i+2) -= 3*temp/(3*dx);
```

```
        temp2(i+3) += temp/(3*dx);
      }
      else if ( (i%N_x)==(N_x-1) ){
        temp = ( -v(i-3) + 3*v(i-2) - 3*v(i-1) + v(i) )/(3*dx);
        temp2(i-3) -= temp/(3*dx);
        temp2(i-2) += 3*temp/(3*dx);
        temp2(i-1) -= 3*temp/(3*dx);
        temp2(i) += temp/(3*dx);
      }
   }

   coeff = bsize*(eps/(4*dx*dx*dx*dx) + 1.0/480.0)/( dx*(N_x-1)*dy*(N_y-1) );
   vectorIncrement(y,coeff,temp2);  // y = y + coeff*temp2

   temp2.zero();

   //compute A2tA2v by computing temp1=A2v and then temp2=A2t*temp1
   for(i=0;i<N_x;i++){
      temp = ( -v(i) + 3*v(i+N_x) - 3*v(i+2*N_x) + v(i+3*N_x) )/(3*dy);
      temp2(i) -= temp/(3*dy);
      temp2(i+N_x) += 3*temp/(3*dy);
      temp2(i+2*N_x) -= 3*temp/(3*dy);
      temp2(i+3*N_x) += temp/(3*dy);
   }
   for(i=N_x;i<N_x*(N_y-1);i++){
      temp = v(i-N_x) - 2*v(i) + v(i+N_x);
      temp2(i-N_x) += temp;
      temp2(i) -= 2*temp;
      temp2(i+N_x) += temp;
   }
   for(i=N_x*(N_y-1);i<k;i++){
      temp = ( -v(i-3*N_x) + 3*v(i-2*N_x) - 3*v(i-N_x) + v(i) )/(3*dy);
      temp2(i-3*N_x) -= temp/(3*dy);
      temp2(i-2*N_x) += 3*temp/(3*dy);
      temp2(i-N_x) -= 3*temp/(3*dy);
      temp2(i) += temp/(3*dy);
   }

   coeff = bsize*(eps/(4*dy*dy*dy*dy) + 1.0/480.0)/(dx*(N_x-1)*dy*(N_y-1) );
   vectorIncrement(y,coeff,temp2);    // y = y + coeff*temp2

}// End matrixMult
```

5.5 setEpsilon.c

```
/* gridIt Version 1.0                                  */
```

```
/* January 9, 1996                                    */
/* Copyright 1996 by Bruce Pirooz Ayati.              */
/* See copyright file or user manual for disclaimer. */

#include "vector.h"
#include <math.h>


double setEpsilon(vector& v, int N_xC, int N_yC, double dxC, double dyC,
                double dx, double dy, double delta)
{
   int i;
   double C1, C2, temp;
   vector y(N_xC*N_yC);
   y.zero();

   //compute A1tA1v
   for(i=0;i<N_xC*N_yC;i++){
      //compute temp = A1v then y = A1t*temp
      if( (i%N_xC)!=0 && (i%N_xC)!=(N_xC-1) ){
        temp = v(i-1) - 2*v(i) + v(i+1);
        y(i-1) += temp;
        y(i) -= 2*temp;
        y(i+1) += temp;
      }
      else if ( (i%N_xC)==0 ){
        temp = ( -v(i) + 3*v(i+1) - 3*v(i+2) + v(i+3) )/(3*dx);
        y(i) -= temp/(3*dx);
        y(i+1) += 3*temp/(3*dx);
        y(i+2) -= 3*temp/(3*dx);
        y(i+3) += temp/(3*dx);
      }
      else if ( (i%N_xC)==(N_xC-1) ){
        temp = ( -v(i-3) + 3*v(i-2) - 3*v(i-1) + v(i) )/(3*dx);
        y(i-3) -= temp/(3*dx);
        y(i-2) += 3*temp/(3*dx);
        y(i-1) -= 3*temp/(3*dx);
        y(i) += temp/(3*dx);
      }
   }

   C1 = (1.0/(dxC*dxC*dxC*dxC))*(y,y)/(N_xC*N_yC);

   y.zero();

   //compute A2tA2v by computing temp=A2v and then y=A2t*temp
```

```
   for(i=0;i<N_xC;i++){
      temp = ( -v(i) + 3*v(i+N_xC) - 3*v(i+2*N_xC) + v(i+3*N_xC) )/(3*dy);
      y(i) -= temp/(3*dy);
      y(i+N_xC) += 3*temp/(3*dy);
      y(i+2*N_xC) -= 3*temp/(3*dy);
      y(i+3*N_xC) += temp/(3*dy);
   }
   for(i=N_xC;i<N_xC*(N_yC-1);i++){
      temp = v(i-N_xC) - 2*v(i) + v(i+N_xC);
      y(i-N_xC) += temp;
      y(i) -= 2*temp;
      y(i+N_xC) += temp;
   }
   for(i=N_xC*(N_yC-1);i<N_xC*N_yC;i++){
      temp = ( -v(i-3*N_xC) + 3*v(i-2*N_xC) - 3*v(i-N_xC) + v(i) )/(3*dy);
      y(i-3*N_xC) -= temp/(3*dy);
      y(i-2*N_xC) += 3*temp/(3*dy);
      y(i-N_xC) -= 3*temp/(3*dy);
      y(i) += temp/(3*dy);
   }

   C2 = (1.0/(dyC*dyC*dyC*dyC))*(y,y)/(N_xC*N_yC);

   return (delta*delta)*(dxC*(N_xC-1)*dyC*(N_yC-1))/(C1 + C2);

} //End setEpsilon
```

5.6 vector.h

```
/* gridIt Version 1.0                                  */
/* January 9, 1996                                     */
/* Copyright 1996 by Bruce Pirooz Ayati.               */
/* See copyright file or user manual for disclaimer. */

#include<iostream.h>
#include<stdlib.h>
#include<math.h>
#include<fstream.h>

#ifndef VECTOR_h
#define VECTOR_h

class vector {
    int len;
    double *v;
```

```
    public:
        vector(int n){ // constructor
            len=n; v=new double[n];
        };
        ~vector(){delete[] v;}; // define the vector destructor
        double& operator() (int i){ // subscripting
            if(i<0||i>=len){
              cerr << "index, " << i
                   << " exceeds vector dimension,"
                   << len <<endl;
              exit(0);
            }
            return v[i];
        };
        void operator=(const vector& x) { // *this=x
            int i, n=x.len;
            if(len != n){cerr<<"(=)Incorrect vector size"<<endl; exit(0);}
            for(i=0; i<n; i++) v[i]=x.v[i];
        };
        void operator+=(const vector& x) {//form x+y
            int i, n=x.len;
            if(len != n){cerr<<"(+=)Incorrect vector size"<<endl; exit(0);}
            for(i=0;i<n;i++) v[i]+=x.v[i];
        }
        void operator-=(const vector& x) {//form x+y
            int i, n=x.len;
            if(len != n){cerr<<"(-=)Incorrect vector size"<<endl; exit(0);}
            for(i=0;i<n;i++) v[i]-=x.v[i];
        }
        void zero(){for(int i=0; i<len; i++) v[i]=0.0;}
        int size(){return len;}

    friend double operator,(const vector&, const vector&);
    friend void vectorIncrement( vector& x, double c, vector& y);
    friend void vectorLinComb( vector& v, vector& x, double c, vector& y);
}; // end vector class definition

#endif
```

5.7 vector.c

```
/* gridIt Version 1.0                                */
/* January 9, 1996                                   */
/* Copyright 1996 by Bruce Pirooz Ayati.             */
/* See copyright file or user manual for disclaimer. */
```

```
#include "vector.h"

double operator,(const vector& x, const vector& y) {//form inner product (x,y)
   int i, m=x.len, n=y.len;
   double c=0.;
   if(m!=n || m<=0) {cerr<<"(,)Incorrect vector size"<<endl; exit(0);}
   for(i=0;i<n;i++) c += x.v[i]*y.v[i];
   return c;
}

void vectorIncrement( vector& x, double c, vector& y){ // x += c*y
   int i, m=x.len, n=y.len;
   if(m!=n || m<=0)
     {cerr<<"(vectorIncrement)Incorrect vector size"<<endl; exit(0);}
   for(i=0;i<n;i++) x.v[i] += c*y.v[i];
}

void vectorLinComb( vector& v, vector& x, double c, vector& y){ // v = x + c*y
   int i, m=x.len, n=y.len, l=v.len;
   if(m!=n || m!=l || m<=0)
     {cerr<<"(vectorLinComb)Incorrect vector size"<<endl; exit(0);}
   for(i=0;i<n;i++) v.v[i] = x.v[i] + c*y.v[i];
}
```

References

[1] Brenner, S.C, Scott, L.R. 1994. *The Mathematical Theory of Finite Element Methods*, Springer-Verlag, New York.

[2] Stoustrup, Bjarne 1991. *The C++ Programming Language*, Addison-Wesley, Reading, Massachusetts.