Iterative methods for linear systems

Chris H. Rycroft[*]

November 20th, 2007

Introduction

For many elliptic PDE problems, finite-difference and finite-element methods are the techniques of choice. In a finite-difference approach, we search for a solution u_k on a set of discrete gridpoints $1, \ldots, k$. The discretized partial differential equation and boundary conditions give us linear relationships between the different values of u_k. In finite-element method, we express our solution as a linear combination u_k of basis functions λ_k on the domain, and the corresponding finite-element variational problem again gives linear relationships between the different values of u_k.

Regardless of the precise details, all of these approaches ultimately end up with having to find the u_k which satisfy all the linear relationships prescribed by the PDE. This can be written as a matrix equation of the form

$$Au = b$$

where we wish to find a solution u, given that A is a matrix capturing the differentiation operator, and b corresponds to any source or boundary terms. Theoretically, this problem could be solved on a computer by any of the standard methods for dealing with matrices. However, the real challenge for PDEs is that frequently, the dimensionality of the problem can be enormous. For example, for a two dimensional PDE problem, a 100×100 grid would be a perfectly reasonable size to consider. Thus u would be a vector with 10^4 elements, and A would be a matrix with 10^8 elements. Even allocating memory for such a large matrix may be problematic. Direct approaches, such as the explicit construction of A^{-1}, are impractical.

The key to making progress is to note that in general, the matrix A is extremely sparse, since the linear relationships usually only relate nearby gridpoints together. We therefore seek methods which do not require ever explicitly specifying all the elements of A, but exploit its special structure directly. Many of these methods are *iterative* – we start with a guess u_k, and apply a process that yields a closer solution u_{k+1}.

[*]Electronic address: `chr@math.berkeley.edu`

Typically, these iterative methods are based on a *splitting* of A. This is a decomposition $A = M - K$, where M is non-singular. Any splitting creates a possible iterative process. We can write

$$\begin{aligned} Au &= b \\ (M-K)u &= b \\ Mu &= Ku + b \\ u &= M^{-1}Ku + M^{-1}b \end{aligned}$$

and hence a possible iteration is

$$u_{k+1} = M^{-1}Ku_k + M^{-1}b.$$

Of course, there is no guarantee that an arbitrary splitting will result in an iterative method which converges. To study convergence, we must look at the properties of the matrix $R = M^{-1}K$. For convergence analysis, it is helpful to introduce the *spectral radius*

$$\rho(R) = \max_j \{|\lambda_j|\}$$

where the λ_j are the eigenvalues of R. It can be shown [2] that an iterative scheme converges if and only if $\rho(R) < 1$. The size of the spectral radius determines the convergence rate, and ideally we would like to find splittings which result in as small a $\rho(R)$ as possible.

An example: a two dimensional Poisson problem

In the convergence analysis later, we will consider a two dimensional Poisson problem on the square $-1 \leq x \leq 1, -1 \leq y \leq 1$, given by the equation

$$-\nabla^2 u = f,$$

subject to the Dirichlet conditions that $u(x, y)$ vanishes on the boundary. We use a source function of the form

$$f(x,y) = \begin{cases} 1 & \text{if } |x| < 0.5 \text{ and } |y| < 0.5 \\ 0 & \text{otherwise.} \end{cases} \tag{1}$$

This is plotted on a 33×33 grid in figure 1. For convergence properties, the eigenfunctions and eigenvalues of this function are very important, and to determine these, it is helpful to consider an associated one-dimensional Poisson problem on the interval $-1 \leq x \leq 1$,

$$-\frac{d^2u}{dx^2} = f(x),$$

subject to Dirichlet boundary conditions $u(-1) = u(1) = 0$. We consider a discretization into $N+2$ gridpoints such that $x_j = -1 + 2j/(N+1)$ for $j = 0, \ldots, N+1$. When constructing the corresponding matrix problem, u_0 and u_{N+1} need not be considered, since their values are always fixed to zero. By discretizing the second derivative according to

$$\left.\frac{d^2u}{dx^2}\right|_{x=x_j} = \frac{u_{j-1}+u_{j+1}-2u_j}{2h^2}$$

where $h = 2/N$, we can write the corresponding linear system as

$$T_N \begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ u_n \end{pmatrix} = \begin{pmatrix} 2 & -1 & 0 & & \\ -1 & 2 & -1 & \ddots & \\ 0 & -1 & \ddots & -1 & 0 \\ & \ddots & -1 & 2 & -1 \\ & & 0 & -1 & 2 \end{pmatrix} \begin{pmatrix} u_0 \\ u_1 \\ \vdots \\ u_n \end{pmatrix} = 2h^2 \begin{pmatrix} f_0 \\ f_1 \\ \vdots \\ f_n \end{pmatrix}.$$

Motivated by previous lectures on the spectral method, we expect that the eigenvectors of T_N may be based on sine functions. A reasonable guess for the jth eigenfunction is

$$z_j(k) = \sqrt{\frac{2}{N+1}} \sin\frac{\pi kj}{N+1}.$$

To verify this is an eigenfunction, and find its eigenvalue, we apply T_N to obtain

$$\left(T_N z_j\right)(k) = \sqrt{\frac{2}{N+1}} \left(2\sin\frac{\pi kj}{N+1} - \sin\frac{\pi(k+1)j}{N+1} - \sin\frac{\pi(k-1)j}{N+1}\right).$$

Note that this expression will always be valid for the range $k = 1, 2, \ldots, N$, and the boundary values just work out. The last two sine functions can be rewritten using a trigonometric identity to give

$$\begin{aligned} \left(T_N z_j\right)(k) &= \sqrt{\frac{2}{N+1}} \left(2\sin\frac{\pi kj}{N+1} - 2\sin\frac{\pi kj}{N+1}\cos\frac{\pi j}{N+1}\right) \\ &= \sqrt{\frac{2}{N+1}} 2\left(1-\cos\frac{\pi j}{N+1}\right)\sin\frac{\pi kj}{N+1} \\ &= 2\left(1-\cos\frac{\pi j}{N+1}\right) z_j(k) \end{aligned}$$

and thus we see that z_j is an eigenvector with eigenvalue $\lambda_j = 2(1-\cos\pi j/(N+1))$. The smallest eigenvalue is $\lambda_1 = 2(1-\cos\pi/(N+1))$ and the largest is $\lambda_N = 2(1-\cos N\pi/(N+1))$.

Returning to the two dimensional problem, we see that the corresponding derivative matrix $T_{N\times N}$ can be written as the tensor product of two one dimensional problems T_N. Its eigenvectors can be expressed as the tensor product of the one dimensional eigenvectors, namely

$$z_{i,j}(k,l) = z_i(k) z_j(l)$$

and their corresponding eigenvalues are

$$\lambda_{i,j} = \lambda_i + \lambda_j.$$

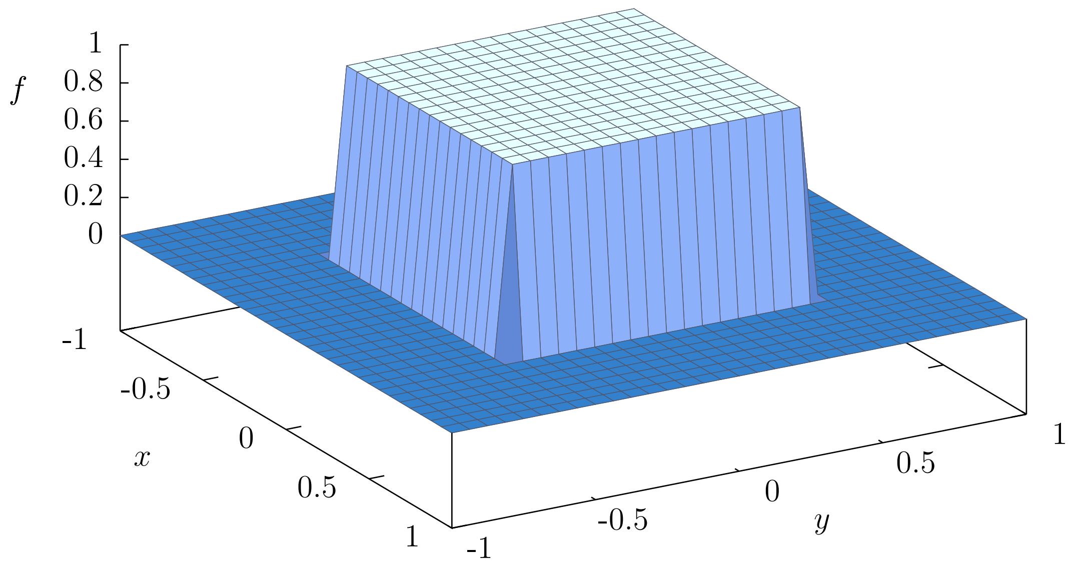

Figure 1: A sample source function $f(x, y)$ on a 33×33 grid.

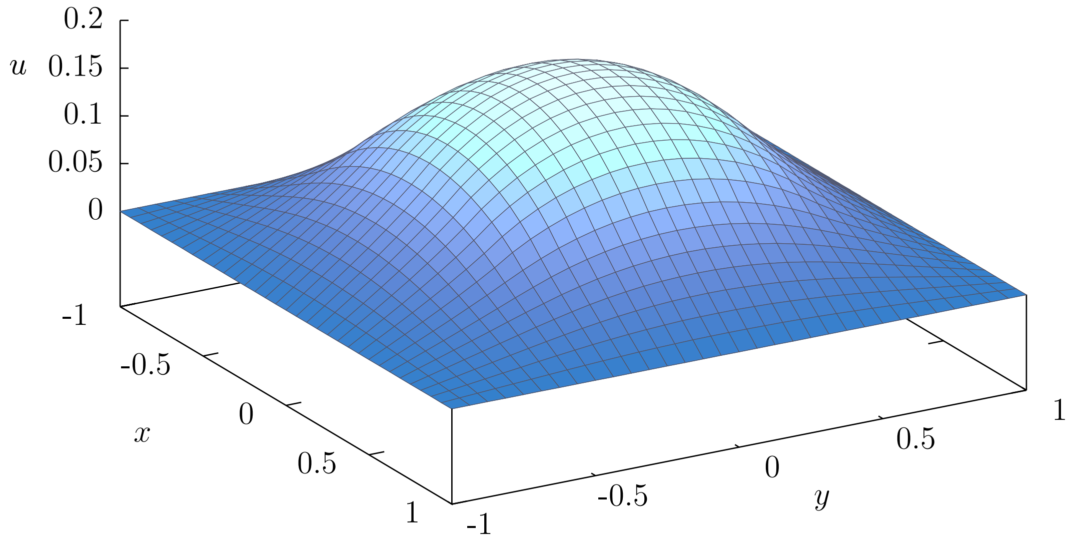

Figure 2: The exact solution to the 2D Poisson problem $-\nabla^2 u = f$, with zero boundary conditions and a source term given in figure 1.

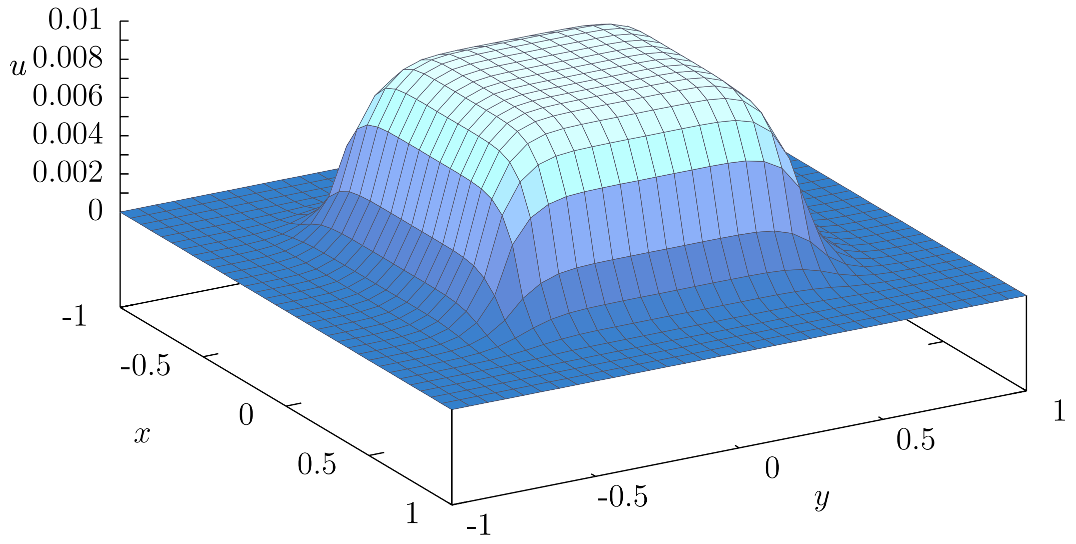

Figure 3: The solution to the example 2D Poisson problem after ten iterations of the Jacobi method.

The Jacobi Method

The Jacobi method is one of the simplest iterations to implement. While its convergence properties make it too slow for use in many problems, it is worthwhile to consider, since it forms the basis of other methods. We start with an initial guess u_0, and then successively improve it according to the iteration

```
**for** $j = 1$ to $N$ **do**
  $u_{m+1,j} = \frac{1}{a_{jj}} \left( b_j - \sum_{k \neq j} a_{jk} u_{m,k} \right)$
**end for**
```

In other words, we set the jth component of u so that it would exactly satisfy equation j of the linear system. For the two dimensional Poisson problem considered above, this corresponds to an iteration of the form

```
**for** $i = 1$ to $N$ **do**
  **for** $j = 1$ to $N$ **do**
    $u_{m+1,i,j} = \left( h^2 f_j + u_{m,i,j+1} + u_{m,i,j-1} + u_{m,i+1,j} + u_{m,i-1,j} \right) / 4$
  **end for**
**end for**
```

To find the corresponding matrix form, write $A = D - L - U$ where D is diagonal, L is lower-triangular, and U is upper-triangular. Then the above iteration can be written as

$$u_{m+1} = D^{-1}(L+U)u_m + D^{-1}b.$$

The convergence properties, discussed later, are then set by the matrix $R_J = D^{-1}(L+U)$.

The Jacobi method has the advantage that for each m, the order in which the components of u_{m+1} are computed has no effect – this may be a favorable property to have in some parallel implementations. However, it can also be seen that u_m must be retained until after u_{m+1} is constructed, meaning we must store u_{m+1} in a different part of the memory. The listing given in appendix A.1 carries out the Jacobi iteration on the Poisson test function. It makes use of two arrays for the storage of u, computing the odd u_k in one and the even u_k in the other. Figure 3 shows a the progress of the Jacobi method after ten iterations.

The Gauss–Seidel Method

The Gauss–Seidel method improves on the Jacobi algorithm, by noting that if we are updating a particular point $u_{m+1,j}$, we might as well reference the already updated values $u_{m+1,1}, \ldots, u_{m+1,j-1}$ in the calculation, rather than using the original values $u_{m,1}, \ldots, u_{m,j-1}$. The iteration can be written as:

```
**for** $j = 1$ to $N$ **do**
    $u_{m+1,j} = \frac{1}{a_{jj}} \left( b_j - \sum_{k=1}^{j-1} a_{jk} u_{m+1,k} - \sum_{k=j+1}^{N} u_{m,k} \right)$
**end for**
```

The Gauss–Seidel algorithm has the advantage that in a computer implementation, we no longer need to allocate two arrays for u_{m+1} and u_m. Instead, we can make just a single array for u_m, and carry out all the updates *in situ*. However, the Gauss–Seidel implementation introduces an additional complication that the order in which the updates are applied will affect the values of u_m. For a two dimensional problem, two particular orderings are worth special attention:

- *Natural ordering* – this is the typical ordering that would result in a **for** loop. We first loop successively through all elements of the first row $(1,1), \ldots, (1,n)$ before moving onto the second row, and so on.

- *Red–Black ordering* – this is the ordering that results by coloring the gridpoints red and black in a checkerboard pattern. Specifically, we color a gridpoint (i,j) red if $i+j$ is even, and black if $i+j$ is odd. During the Gauss–Seidel update, all red points are updated before the black points. For the two dimensional Poisson problem, we see that updating a red grid point only requires information from the black gridpoints, and vice versa. Hence the order in which points in each set are updated does not matter. We can think of the whole Gauss–Seidel update being divided into a red grid point update and black gridpoint update, and this can be helpful in the convergence analysis.

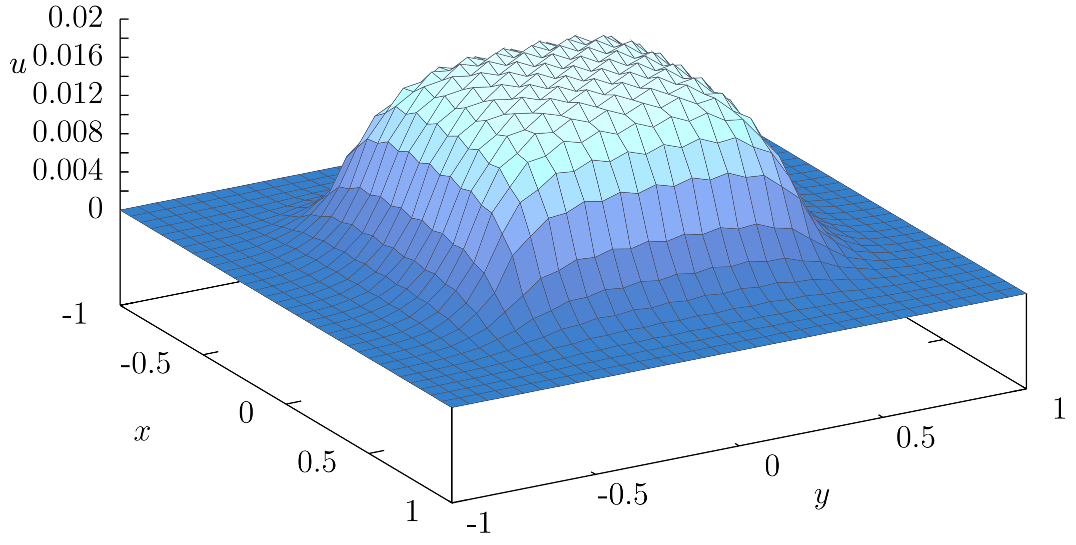

Figure 4: The Gauss–Seidel solution to the example 2D Poisson problem after ten iterations. The crinkles in the solution are due to the Red–Black update procedure.

From the algorithm above, we can write down the corresponding matrix splitting for the Gauss–Seidel method as

$$\begin{aligned}(D-L)u_{m+1} &= Uu_m + b \\ u_{m+1} &= (D-L)^{-1}Uu_m + (D-L)^{-1}b.\end{aligned}$$

Appendix A.2 contains a C++ code to carry out a Gauss–Seidel method on the example problem, and the result after ten iterations is shown in figure 4.

Successive Over-Relaxation

Successive Over-Relaxation (SOR) is a refinement to the Gauss–Seidel algorithm. At each stage in the Gauss–Seidel algorithm, a value $u_{m,j}$ is updated to a new one $u_{m+1,j}$, which we can think of as displacing $u_{m,j}$ by an amount $\Delta u = u_{m+1,j} - u_{m,j}$. The SOR algorithm works by displacing the values by an amount $\omega\Delta u$, where typically $\omega > 1$, in the hope that if Δu is a good direction to move in, we might as well move further in that direction. The iteration can be written as:

```
**for** $j = 1$ to $N$ **do**
  $u_{m+1,j} = (1-\omega)u_{m,j} + \frac{\omega}{a_{jj}}\left(b_j - \sum_{k=1}^{j-1} a_{jk}u_{m+1,k} - \sum_{k=j+1}^{N} u_{m,k}\right)$
**end for**
```

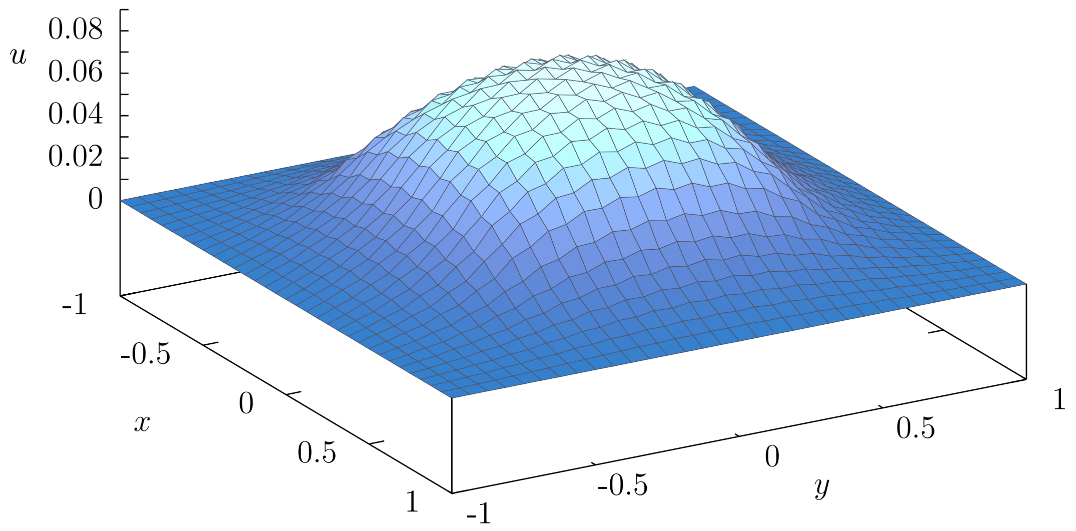

Figure 5: The SOR solution (using the theoretically optimal ω) to the example 2D Poisson problem after ten iterations. The solution is closer to the answer than the Jacobi or Gauss–Seidel methods.

The corresponding matrix form is

$$\begin{aligned}(D+\omega L)u_{m+1} &= [(1-\omega)D - U\omega]\, u_m + \omega b \\ u_{m+1} &= (D+\omega L)^{-1}[(1-\omega)D - U\omega]\, u_m + (D+\omega L)^{-1}\omega b.\end{aligned}$$

Appendix A.3 contains a C++ code to carry out the SOR iteration on the example problem, and the result is shown in figure 5. In the SOR algorithm, we are free to choose the value of ω, and the best choices can be found by considering the eigenfunctions of the associated problem. This is discussed in more detail below.

Convergence analysis and complexity

To examine the convergence properties of the different methods, we need to look at the associated spectral radii. For the Jacobi method, we had $R_J = D^{-1}(L+U)$. For the 2D Poisson problem, $D = 4I$ so we can write $R_J = (4I)^{-1}(4I - T_{N\times N}) = I - T_{N\times N}/4$. The largest eigenvalue of R_J corresponds to the smallest of $T_{N\times N}$, namely

$$\begin{aligned}\lambda_{1,1} &= 4 - 2\cos\left(\frac{\pi}{N+1}\right) - 2\cos\left(\frac{\pi}{N+1}\right) \\ &= 4 - 4\cos\left(\frac{\pi}{N+1}\right).\end{aligned}$$

Hence

$$\rho(R_J) = \cos\left(\frac{\pi}{N+1}\right).$$

A Taylor series expansion shows us that $\rho(R_J) = 1 - 2\pi^2/(N+1)^2$. The time for us to gain an extra digit of accuracy is approximately

$$\frac{1}{\log_{10}\rho(R_J)} \propto N^2.$$

Thus we must run the algorithm for $O(N^2)$ iterations. Since there are $O(N^2)$ gridpoints for the 2D problem, the total running time is $O(N^4)$. For detailed proofs of the convergence properties of the other methods, the reader should refer elsewhere [2]. It can be shown that

$$\rho(R_{GS}) = \cos^2\left(\frac{\pi}{N+1}\right),$$

so that one iteration of the Gauss–Seidel method is equivalent to two Jacobi iterations. Note however the complexity is the same: we still need $O(N^2)$ iterations. For the SOR algorithm, it can be shown that the optimal value of ω is

$$\frac{2}{1+\sqrt{1-\rho(R_J)^2}}.$$

and that for this value,

$$\rho(R_{SOR}) \approx 1 - 2\frac{2\pi}{N+1}.$$

Since there is a factor of N in the denominator as opposed to N^2, the order of computation decreases to $O(N)$ per grid point.

Figure 6 shows a plot of mean square error against the number of iterations for the model problem with the Jacobi, Gauss–Seidel, and optimal SOR method. The lines agree with the above results. The SOR method reaches numerical precision within 1200 iterations, while the other two methods have not fully converged even after 10^4 iterations.

Multigrid

One of the major problems with the three methods considered so far is that they only apply locally. Information about different cell values only propagates by one or two gridpoints per iteration. However, for many elliptic problems, a point source may cause an effect over the entire domain. The above methods have a fundamental limitation that they will need to be applied for at least at many iterations as it takes for information to propagate across the grid. As such, we should not expect to ever do better than $O(N)$ operations per point. This can also be seen by considering the eigenvalues. The maximal eigenvalue of R_J was set by the $\lambda_{1,1}$, corresponding to the lowest order mode. While the methods may effectively damp

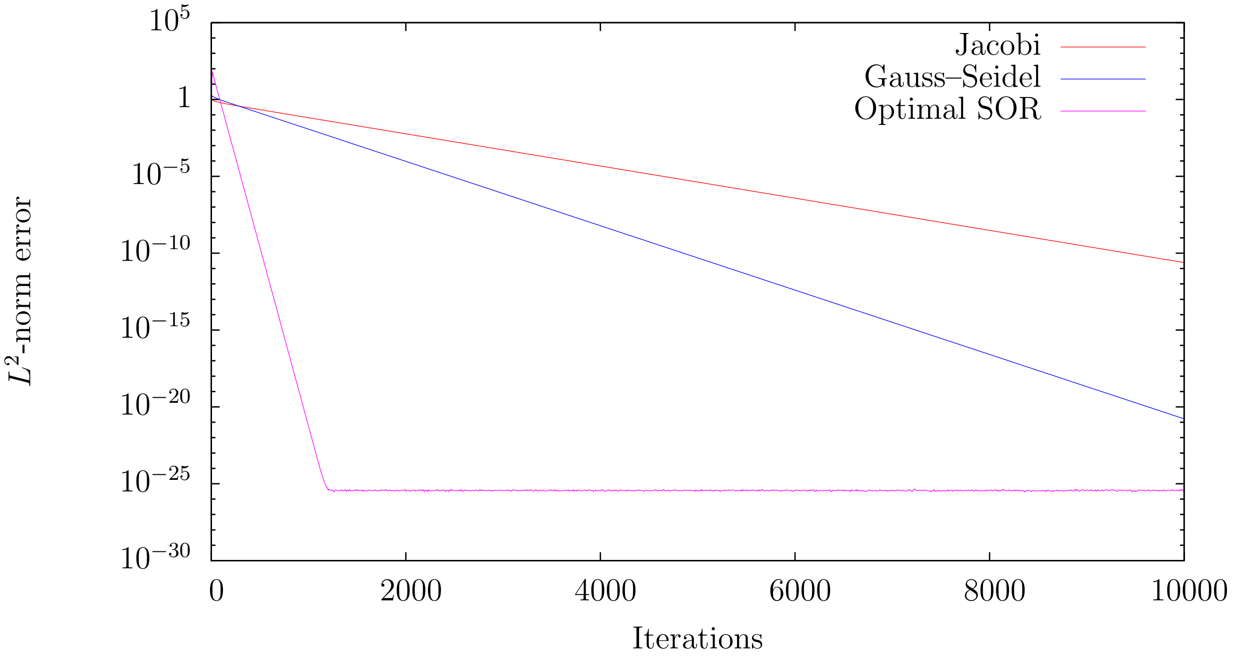

Figure 6: Errors versus the number of effective iterations for the Jacobi, Gauss–Seidel, and SOR methods, applied to the example 2D Poisson problem on a 65×65 grid. The plots are in line with the theoretical results of the text. The Gauss–Seidel method is faster than the Jacobi method, but has still not reached double numerical precision after 10000 iterations. The SOR method is significantly faster, but still requires 1200 iterations to reach double numerical precision.

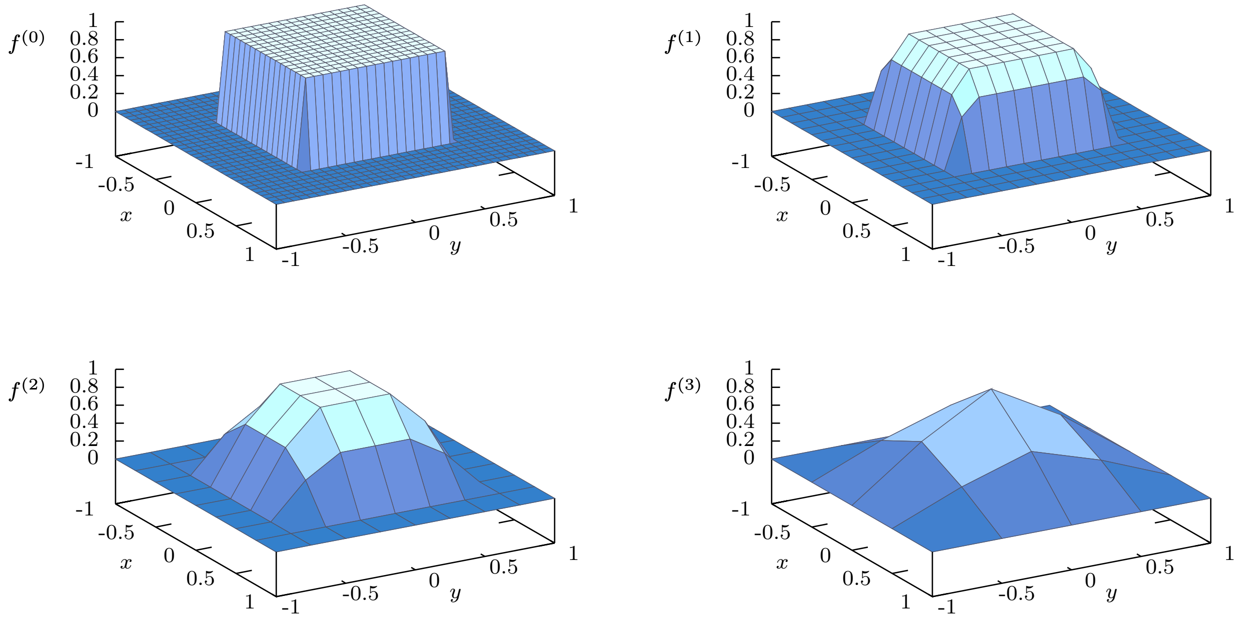

Figure 7: The restriction of the example source term f using the multigrid method with $v_{\text{down}} = 0, v_{\text{up}} = 2$. Top left: the initial grid, with 33×33 gridpoints. Top right: grid 1, with 17×17 gridpoints. Bottom left: grid 2, with 9×9 gridpoints. Bottom right: grid 3, with 5×5 gridpoints.

out high frequency oscillations, it will take a very long time to correctly capture the lowest modes with the largest wavelengths.

The multigrid method circumvents these limitations by introducing a hierarchy of coarser and coarser grids. Typically, at each level, the number of gridpoints is reduced by a factor of two in each direction, with the coarsest grid having ten to twenty points. To find a solution, we restrict the source term to the coarse grids, refine the solution on each, and interpolate up to the original grid. On the coarser grids, the lower frequency modes in the final solution can be dealt with much more effectively. Since the coarser grids have progressively fewer gridpoints, the time spent computing them is minimal. Because of this, the multigrid algorithm requires only $O(1)$ computation per point, which is the best order of complexity that we could hope for.

To be more specific, we let the original problem be on grid 0, and we then introduce a sequence of other successively coarser grids $1, \ldots, g$. We write $u^{(i)}$ and $b^{(i)}$ to represent the solution and source terms on the ith grid. A multigrid algorithm requires the following:

- A solution operator $S(u^{(i)}, b^{(i)})$ which returns a better approximation $u^{(i)}$ to the solution on the ith level
- An interpolation operator $T(u^{(i)})$ which returns an interpolation $u^{(i-1)}$ on the $(i-1)$th level

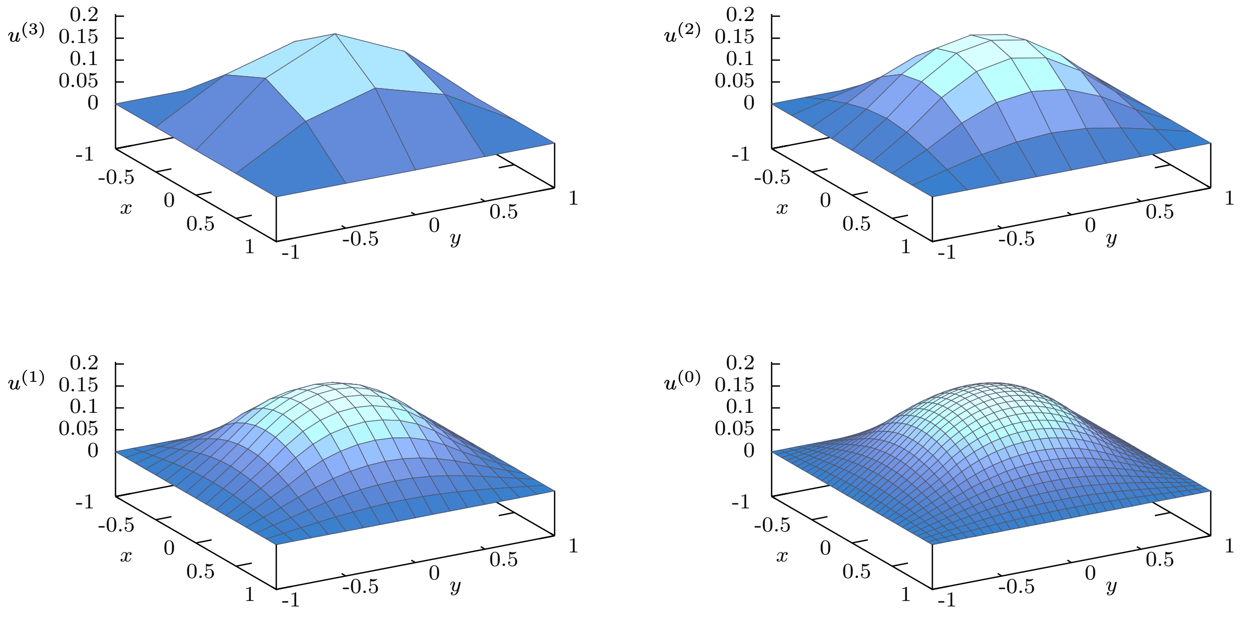

Figure 8: The solution of the v in the first multigrid V-cycle for the example 2D Poisson problem. Top left: the exact solution on grid 3 to the $f^{(3)}$ source term in figure 7. Top right: the interpolation and refinement on grid 2. Bottom left: the interpolation and refinement on grid 1. Bottom right: the interpolation and refinement on grid 0. Even after a single V-cycle, the solution is closer to the exact solution than the plots of 3, 4, and 5.

- A restriction operator $R(b^{(i)})$ which returns a restriction $b^{(i+1)}$ on the $(i+1)$th level

The interpolation and restriction operators can be thought of as rectangular matrices. As an example, consider a problem with 9 equally-spaced gridpoints on the unit interval, at $(0, 1/8, 2/8, \ldots, 1)$. Let grid 1 have 5 points at $(0, 1/4, 2/4, 3/4, 1)$, and let grid 2 have 3 gridpoints at $(0, 1/2, 1)$. Interpolation operators between the grids can be written as

$$T^{(1)} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & \frac{1}{2} & \frac{1}{2} & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{1}{2} & \frac{1}{2} & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & \frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}, \qquad T^{(2)} \begin{pmatrix} 1 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & 0 \\ 0 & 1 & 0 \\ 0 & \frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 1 \end{pmatrix}$$

where we are keeping the values at gridpoints common between the two levels, and introducing extra ones at the midpoints between each. Similarly, the restriction operators can be written as

$$R^{(0)} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & \frac{1}{4} & \frac{1}{2} & \frac{1}{4} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{1}{4} & \frac{1}{2} & \frac{1}{4} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & \frac{1}{4} & \frac{1}{2} & \frac{1}{4} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \qquad R^{(1)} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & \frac{1}{4} & \frac{1}{2} & \frac{1}{4} & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}.$$

It should be noted that without the boundary points, $R^{(i)}$ is the proportional to the transpose of $T^{(i+1)}$, and this property is very useful in proving the convergence of the multigrid method. However, this property is not strictly necessary to create an efficient multigrid algorithm. Other methods of interpolating and restricting are also possible, and it should also be noted that while a grid of size $2^n + 1$ has a particularly convenient multigrid formulation, the multigrid method can be applied to grids of arbitrary size. Given a differential operator matrix $A = A^{(0)}$ on the top level, we can define corresponding matrices on the lower levels according to

$$A^{(i)} = R^{(i-1)} A^{(i-1)} T^{(i)}.$$

With this definition, we can construct a solution operator $S(u^{(i)}, b^{(i)})$ as a single red–black Gauss–Seidel sweep. Given this, we can write a multigrid formulation as

```
**function** Multi($u^{(i)}, b^{(i)}$)
  **if** $i = g$ **then**
    compute exact solution to $A^{(i)}u^{(i)} = b^{(i)}$
    **return** $u^{(i)}$
  **else**
```

```
    for $j = 1$ to $v_{\text{down}}$ do
      $u^{(i)} = S(u^{(i)}, b^{(i)})$
    end for
    $r^{(i)} = b^{(i)} - A^{(i)}u^{(i)}$
    $d^{(i)} = T(\text{Multi}(0^{(i+1)}, R(r^{(i)})))$
    $u^{(i)} = u^{(i)} + d^{(i)}$
    for $j = 1$ to $v_{\text{up}}$ do
      $u^{(i)} = S(u^{(i)}, b^{(i)})$
    end for
  end if
```

The function is applied recursively, and at each stage, the remainder of the problem on the level above is sent to the lower level. The algorithm starts on level 0, descends to level g, and then ascends to level 0 again, following the shape of a V. It is therefore referred to as the *multigrid V-cycle*. Other more elaborate methods of moving between grids are possible, although the V-cycle is extremely efficient in many situations. In the algorithm, we are free to choose the number of times the solution operator is applied on the way down and on the way up, and typical good values to try may be $v_{\text{down}} = v_{\text{up}} = 2$, or even $v_{\text{down}} = 0, v_{\text{up}} = 2$. It may also be worthwhile to carry out more iterations on the coarser grids, since the computation is much cheaper there.

Figure 7 shows the restriction of the source term in the test problem on the coarser grids. Figure 8 shows the solution being successively refined on the grids. Even after a single V-cycle, the solution is close to the exact answer. Figure 9 shows the computation times for two different multigrid algorithms, compared with the previous three methods considered. The multigrid algorithms reach numerical precision extremely quickly, much faster even than SOR. Only twenty Gauss–Seidel iterations are applied at the top level before double numerical precision has been reached.

A Code listings

The following codes were used to generate the Jacobi, Gauss–Seidel, and SOR diagrams in these notes. They are written in C++ and were compiled using the GNU C++ compiler. Each of the first three routines calls a common code listed in appendix A.4 for setting up useful constants and defining common routines. This common code also contains a function for outputting the 2D matrices in a matrix binary format that is readable by the plotting program *Gnuplot* [1]. This output routine could be replaced in order to save to different plotting programs.

A.1 Jacobi method – jacobi.cc

```
// Load common routines and constants
#include "common.cc"
```

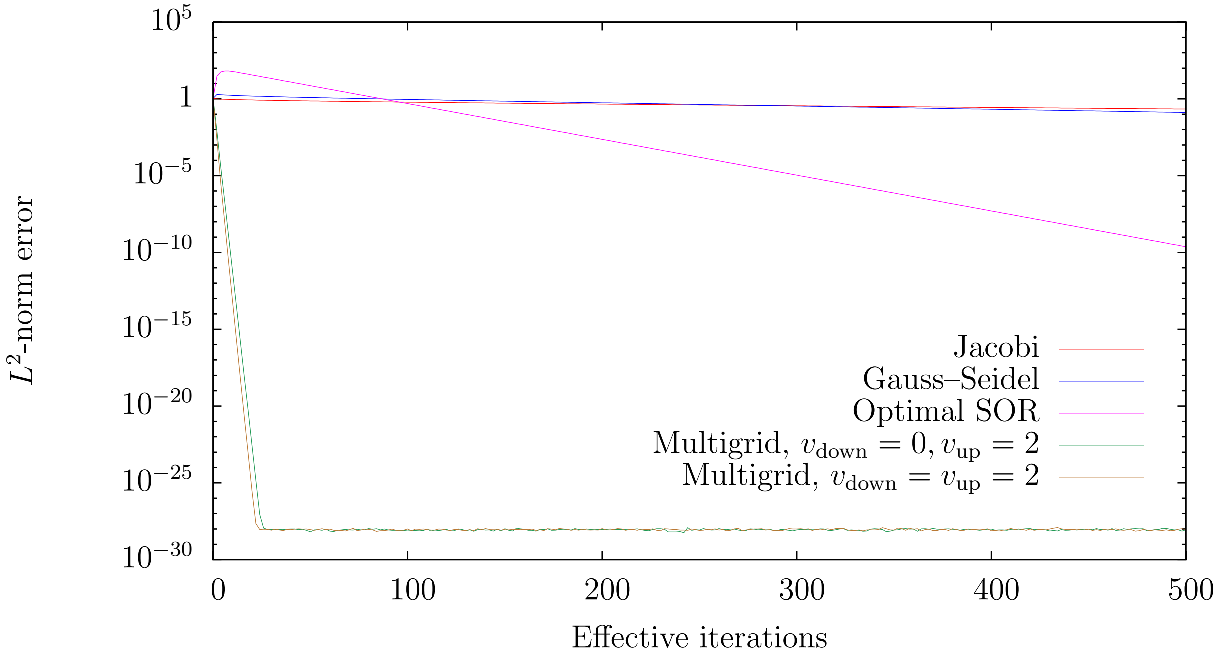

Figure 9: Errors versus the number of effective iterations for the several different iteration techniques. Here, to allow a direct comparison, "effective iterations" for the multigrid methods is defined by the number of Gauss–Seidel iterations that are applied on the top grid level, since the Gauss–Seidel iterations on the coarser grids are small in comparison. For the $v_{\text{down}} = 0, v_{\text{up}} = 2$ method, the effective number of iterations is twice the number of V-cycles. For the $v_{\text{down}} = v_{\text{up}} = 2$ method, the algorithm was slightly modified, so that only two Gauss–Seidel iterations were applied at the top level each time, instead of the expected four. Thus the effective number of iterations is also twice the number of V-cycles. The speed of the multigrid methods is startling when compared to any of the other three iterations.

```
int main() {
  int i,j,ij,k;
  double error,u[m*n],v[m*n],z;
  double *a,*b;

  // Set initial guess to be identically zero
  for(ij=0;ij<m*n;ij++) u[ij]=v[ij]=0;
  output_and_error("jacobi_out",u,0);

  // Carry out Jacobi iterations
  for(k=1;k<=total_iters;k++) {
    // Alternately flip input and output matrices
    if (k%2==0) {a=u;b=v;} else {a=v;b=u;}

    // Compute Jacobi iteration
    for(j=1;j<n-1;j++) {
      for(i=1;i<m-1;i++) {
        ij=i+m*j;
        a[ij]=(f(i,j)+dxxinv*(b[ij-1]+b[ij+1])
          +dyyinv*(b[ij-m]+b[ij+m]))*dcent;
      }
    }

    // Save and compute error if necessary
    output_and_error("jacobi_out",a,k);
  }
}
```

A.2 Gauss–Seidel – gsrb.cc

```
// Load common routines and constants
#include "common.cc"

int main() {
  int i,j,ij,k;
  double error,u[m*n],z;

  // Set initial guess to be identically zero
  for(ij=0;ij<m*n;ij++) u[ij]=0;
  output_and_error("gsrb_out",u,0);

  // Compute Red-Black Gauss-Seidel iteration
  for(k=1;k<=total_iters;k++) {
    for(j=1;j<n-1;j++) {
      for(i=1+(j&1);i<m-1;i+=2) {
        ij=i+m*j;
        u[ij]=(f(i,j)+dxxinv*(u[ij-1]+u[ij+1])
          +dyyinv*(u[ij-m]+u[ij+m]))*dcent;
      }
    }
    for(j=1;j<n-1;j++) {
      for(i=2-(j&1);i<m-1;i+=2) {
        ij=i+m*j;
        u[ij]=(f(i,j)+dxxinv*(u[ij-1]+u[ij+1])
          +dyyinv*(u[ij-m]+u[ij+m]))*dcent;
      }
    }

    // Save the result and compute error if necessary
    output_and_error("gsrb_out",u,k);
  }
}
```

A.3 Successive Over-Relaxation – sor.cc

```
// Load common routines and constants
#include "common.cc"

int main() {
  int i,j,ij,k;
  double error,u[m*n],z;

  // Set initial guess to be identically zero
  for(ij=0;ij<m*n;ij++) u[ij]=0;
  output_and_error("sor_out",u,0);

  // Compute SOR Red-Black iterations
  for(k=1;k<=total_iters;k++) {
    for(j=1;j<n-1;j++) {
      for(i=1+(j&1);i<m-1;i+=2) {
        ij=i+m*j;
        u[ij]=u[ij]*(1-omega)+omega*(f(i,j)
             +dxxinv*(u[ij-1]+u[ij+1])
             +dyyinv*(u[ij-m]+u[ij+m]))*dcent;
      }
    }
    for(j=1;j<n-1;j++) {
      for(i=2-(j&1);i<m-1;i+=2) {
        ij=i+m*j;
        u[ij]=u[ij]*(1-omega)+omega*(f(i,j)
             +dxxinv*(u[ij-1]+u[ij+1])
```

```
            +dyyinv*(u[ij-m]+u[ij+m]))*dcent;
        }
    }

    // Save the result and compute error if necessary
    output_and_error("sor_out",u,k);
  }
}
```

A.4 Common routine for setup and output – common.cc

```
// Load standard libraries
#include <cstdio>
#include <cstdlib>
#include <iostream>
#include <fstream>
#include <cmath>
using namespace std;

// Set grid size and number of iterations
const int save_iters=20;
const int total_iters=200;
const int error_every=2;
const int m=33,n=33;
const double xmin=-1,xmax=1;
const double ymin=-1,ymax=1;

// Compute useful constants
const double pi=3.1415926535897932384626433832795;
const double omega=2/(1+sin(2*pi/n));
const double dx=(xmax-xmin)/(m-1);
const double dy=(ymax-ymin)/(n-1);
const double dxxinv=1/(dx*dx);
const double dyyinv=1/(dy*dy);
const double dcent=1/(2*(dxxinv+dyyinv));

// Input function
inline double f(int i,int j) {
  double x=xmin+i*dx,y=ymin+j*dy;
  return abs(x)>0.5||abs(y)>0.5?0:1;
}

// Common output and error routine
void output_and_error(char* filename,double *a,const int sn) {
  // Computes the error if sn%error_every==0
  if(sn%error_every==0) {
    double z,error=0;int ij;
    for(int j=1;j<n-1;j++) {
      for(int i=1;i<m-1;i++) {
        ij=i+m*j;
        z=f(i,j)-a[ij]*(2*dxxinv+2*dyyinv)
          +dxxinv*(a[ij-1]+a[ij+1])
          +dyyinv*(a[ij-m]+a[ij+m]);
        error+=z*z;
      }
    }
    cout << sn << "␣" << error*dx*dy << endl;
  }

  // Saves the matrix if sn<=save_iters
  if(sn<=save_iters) {
    int i,j,ij=0,ds=sizeof(float);
    float x,y,data_float;const char *pfloat;
    pfloat=(const char*)&data_float;

    ofstream outfile;
    static char fname[256];
    sprintf(fname,"%s.%d",filename,sn);
    outfile.open(fname,fstream::out
                 |fstream::trunc|fstream::binary);

    data_float=m;outfile.write(pfloat,ds);
    for(i=0;i<m;i++) {
      x=xmin+i*dx;
      data_float=x;outfile.write(pfloat,ds);
    }

    for(j=0;j<n;j++) {
      y=ymin+j*dy;
      data_float=y;
      outfile.write(pfloat,ds);
      for(i=0;i<m;i++) {
        data_float=a[ij++];
        outfile.write(pfloat,ds);
      }
    }
    outfile.close();
  }
}
```

References

[1] http://gnuplot.info/.

[2] J. W. Demmel, *Applied numerical linear algebra*, SIAM, 1997.

[3] G. H. Golub and C. H. Van Loan, *Matrix computations*, Johns Hopkins University Publishers, 1996.