# Math/CS 714: Iterative methods for linear systems

## 1 Introduction

For many elliptic PDE problems, finite-difference and finite-element methods are the techniques of choice. In a finite-difference approach, a solution $u_j$ on a set of discrete gridpoints $1, \ldots, N$ is searched for. The discretized partial differential equation and boundary conditions create linear relationships between the different values of $u_j$. In the finite-element method, the solution is expressed as a linear combination $u_j$ of basis functions $\lambda_j$ on the domain, and the corresponding finite-element variational problem again gives linear relationships between the different values of $u_j$.

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

$$Au = b \tag{1.1}$$

where one aims 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 $100 \times 100$ 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. It is therefore natural to seek methods that do not require ever explicitly specifying all the elements of $A$, but exploit its special structure directly. Many of these methods are ***iterative***—starting with a guess $u_k \in \mathbb{R}^N$, a process is applied that yields a closer solution $u_{k+1}$. 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. Equation (1.1) can be rewritten as

$$\begin{aligned} (M - K)u &= b, \\ Mu &= Ku + b, \\ u &= M^{-1}Ku + M^{-1}b, \end{aligned} \tag{1.2}$$

and hence a possible iteration is

$$u_{k+1} = M^{-1}Ku_k + M^{-1}b. \tag{1.3}$$

The true solution $u_*$ is a fixed point of this iterative process, and therefore the iteration may converge to it. However, there is no guarantee that an arbitrary splitting will result in an iterative method that converges. To study convergence, one must look at the properties of the matrix $R = M^{-1}K$. Suppose that $R$ is diagonalizable and has eigenvectors $v_j$ with corresponding eigenvalues $\lambda_j$. Let the initial guess be $u_0$, and write it in the basis of eigenvectors as

$$u_0 = u_* + \sum_{j=1}^{N} \alpha_j v_j \tag{1.4}$$

for some parameters $\alpha_j$. Then

$$
\begin{aligned}
u_1 &= R\left(u_* + \sum_{j=1}^{N} \alpha_j v_j\right) + M^{-1}b \\
&= \sum_{j=1}^{N} \alpha_j \lambda_j v_j + M^{-1}Ku_* + M^{-1}b \\
&= u_* + \sum_{j=1}^{N} \alpha_j \lambda_j v_j, \qquad (1.5)
\end{aligned}
$$

where on the final line, the relationship $u_* = M^{-1}Ku_* + M^{-1}b$ from Eq. (1.2) has been used. Each $v_j$ term is therefore multiplied by its corresponding eigenvalue. Generalizing, it follows that

$$
u_k = u_* + \sum_{j=1}^{N} \alpha_j \lambda_j^k v_j \qquad (1.6)
$$

since each successive iteration will multiply each eigenvector by its corresponding eigenvalue. Thus if all of the eigenvalues have magnitude less than one, the iteration will converge. This connects to the *spectral radius* introduced earlier, defined as

$$
\rho(R) = \max_j \{|\lambda_j|\}, \qquad (1.7)
$$

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

## 2 An example: a two-dimensional Poisson problem

We will now introduce an example Poisson problem on a square that will be used for the convergence analysis later. This example is given in chapter 7 of Demmel's book [2], and is near-identical to the elliptic problem considered by LeVeque [4] that was considered in the lectures, except that it uses different coordinate ranges and flips the sign of the discretization. LeVeque's discretization leads to a symmetric negative definite matrix. We will use Demmel's discretization, because it leads to a symmetric positive definite matrix, which eliminates some minus signs from the convergence analysis. However, all of the conclusions about convergence, eigenvalues, and eigenvectors are essentially equivalent.

We solve the Poisson problem on the square $\Omega = [-1, 1]^2$, given by the equation

$$
-\nabla^2 u = f, \qquad (2.1)
$$

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

$$
f(x, y) = \begin{cases} 1 & \text{if } |x| < 0.5 \text{ and } |y| < 0.5 \\ 0 & \text{otherwise.} \end{cases} \qquad (2.2)
$$

This is plotted on a $33 \times 33$ grid in Fig. 2.1. 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 $-1 \leq x \leq 1$,

$$
-\frac{d^2 u}{dx^2} = f(x), \qquad (2.3)
$$

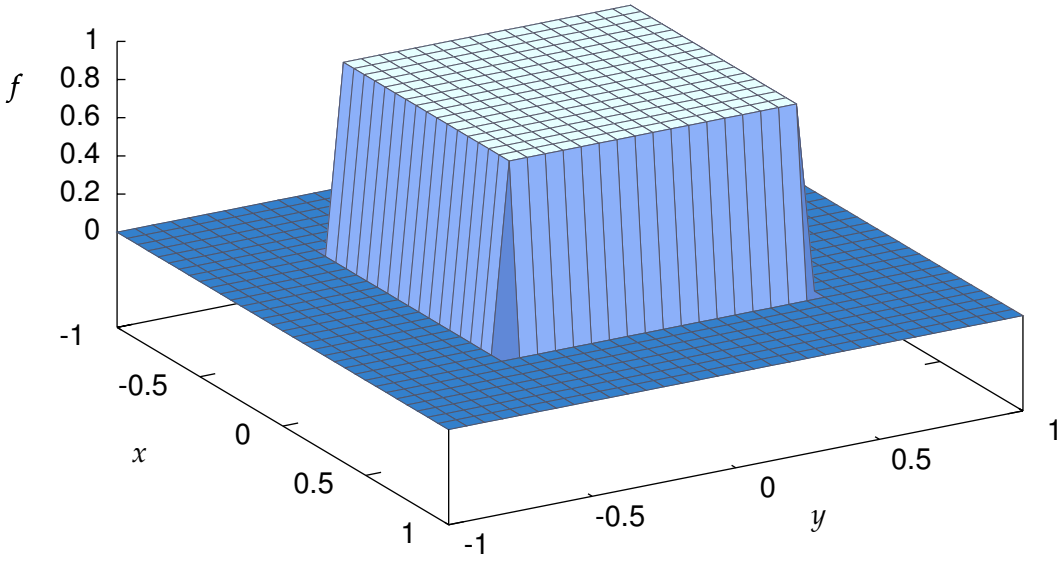


**Figure 2.1:** A sample source function $f(x, y)$ on a $33 \times 33$ grid.

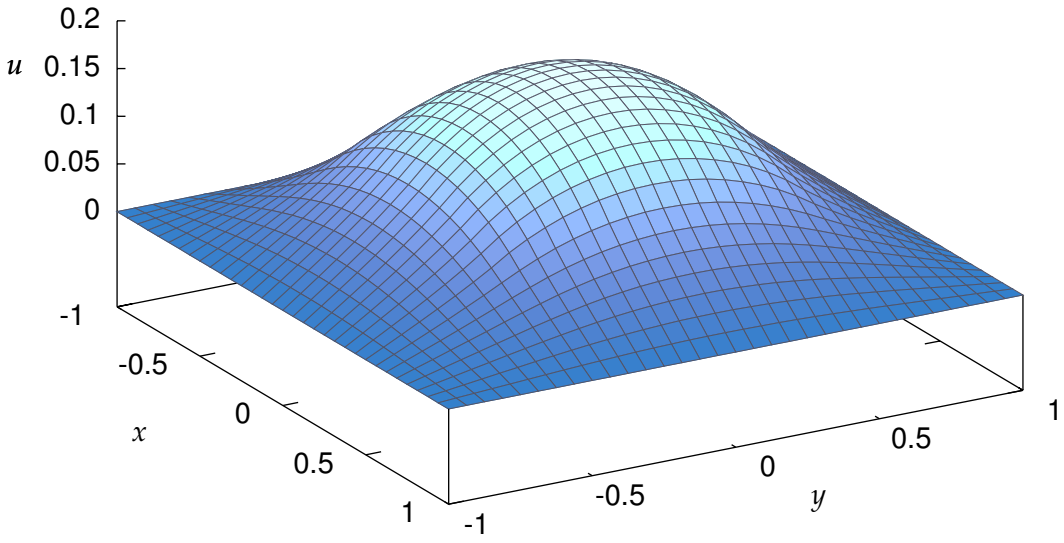


**Figure 2.2:** The exact solution to the 2D Poisson problem $-\nabla^2 u = f$, with zero boundary conditions and a source term given in Fig. 2.1.

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

$$\left.\frac{d^2u}{dx^2}\right|_{x=x_j} = \frac{u_{j-1} + u_{j+1} - 2u_j}{h^2}, \tag{2.4}$$

the corresponding linear system is

$$T_m \begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ u_m \end{pmatrix} = \begin{pmatrix} 2 & -1 & 0 & & \\ -1 & 2 & -1 & \ddots & \\ 0 & -1 & \ddots & -1 & 0 \\ & \ddots & -1 & 2 & -1 \\ & & 0 & -1 & 2 \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ u_m \end{pmatrix} = h^2 \begin{pmatrix} f_1 \\ f_2 \\ \vdots \\ f_m \end{pmatrix}. \tag{2.5}$$

We expect that the eigenvectors of $T_m$ may be based on sine functions. A reasonable guess for the $p$th eigenfunction is

$$z_p(i) = \sqrt{\frac{2}{m+1}} \sin \frac{\pi p i}{m+1}. \tag{2.6}$$

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

$$\left(T_m z_p\right)(i) = \sqrt{\frac{2}{m+1}} \left( 2\sin \frac{\pi i p}{m+1} - \sin \frac{\pi (i+1) p}{m+1} - \sin \frac{\pi (i-1) p}{m+1} \right). \tag{2.7}$$

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

$$\begin{aligned} \left(T_m z_p\right)(i) &= \sqrt{\frac{2}{m+1}} \left( 2\sin \frac{\pi i p}{m+1} - 2\sin \frac{\pi i p}{m+1} \cos \frac{\pi p}{m+1} \right) \\ &= \sqrt{\frac{2}{m+1}} 2 \left( 1 - \cos \frac{\pi p}{m+1} \right) \sin \frac{\pi i p}{m+1} \\ &= 2 \left( 1 - \cos \frac{\pi p}{m+1} \right) z_p(i) \end{aligned} \tag{2.8}$$

and hence $z_p$ is an eigenvector with eigenvalue

$$\lambda_p = 2 \left( 1 - \cos \frac{\pi p}{m+1} \right). \tag{2.9}$$

The smallest eigenvalue is $\lambda_1 = 2(1 - \cos \pi/(m+1))$ and the largest is $\lambda_m = 2(1 - \cos m\pi/(m+1))$.

Returning to the two-dimensional problem, the corresponding derivative matrix $T_{m \times m}$ can be written as the tensor product of two one-dimensional problems $T_m$. Its eigenvectors are the tensor products of the one-dimensional eigenvectors, namely

$$z_{p,q}(i,j) = z_p(i) z_q(j), \tag{2.10}$$

and the corresponding eigenvalues are

$$\lambda_{p,q} = \lambda_p + \lambda_q. \tag{2.11}$$

## 3 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 worth considering since it forms the basis of other methods. Starting with an initial guess $u_0 \in \mathbb{R}^N$, one successively applies an iteration of the form

**Jacobi method**

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

In other words, the $j$th component of $u$ is set so that it exactly satisfies equation $j$ of the linear system. For the two-dimensional Poisson problem considered above, this corresponds to an iteration of the form

**Jacobi method**

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

$$u_{k+1} = D^{-1}(L + U)u_k + D^{-1}b. \tag{3.1}$$

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 $k$, the order in which the components of $u_{k+1}$ are computed has no effect—this may be a favorable property to have in some parallel implementations. However, it can also be seen that $u_k$ must be retained until after $u_{k+1}$ is constructed, meaning we must store $u_{k+1}$ 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 3.1 shows a the progress of the Jacobi method after ten iterations.

## 4 The Gauss–Seidel method

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

**Gauss–Seidel method**

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

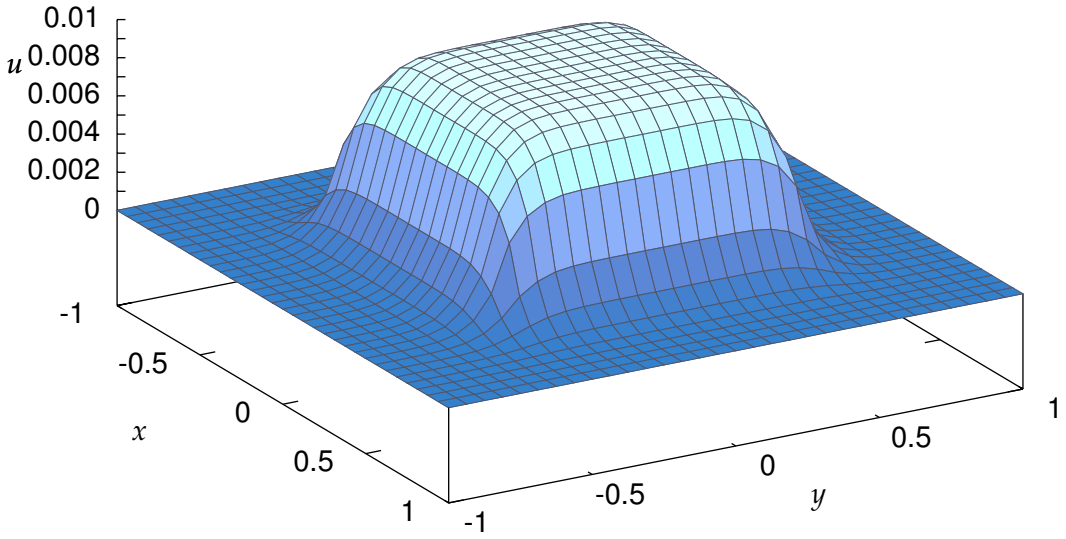


**Figure 3.1:** The solution to the example 2D Poisson problem after ten iterations of the Jacobi method.

The Gauss–Seidel algorithm has the advantage that in a computer implementation, it is not necessary to allocate two arrays for $u_{k+1}$ and $u_k$. Instead, a single array for $u_k$ can be used and all updates can be carried out *in situ*. However, the Gauss–Seidel implementation introduces an additional complication that the order in which the updates are applied affects the values of $u_k$. 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. Since this frequently corresponds to the order that the grid is stored within the computer memory, this order can lead to performance improvements due to good cache efficiency.
- *Red–Black ordering* – this is the ordering that results by coloring the gridpoints red and black in a checkerboard pattern. Specifically, a gridpoint $(i,j)$ is colored red if $i+j$ is even and colored 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, 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. The whole Gauss–Seidel update is divided into a red gridpoint update and black gridpoint update, and this can be helpful in the convergence analysis.

From the algorithm above, the corresponding matrix splitting for the Gauss–Seidel method is

$$\begin{aligned}(D-L)u_{k+1} &= Uu_k + b,\\ u_{k+1} &= (D-L)^{-1}Uu_k + (D-L)^{-1}b.\end{aligned} \tag{4.1}$$

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 Fig. 4.1.

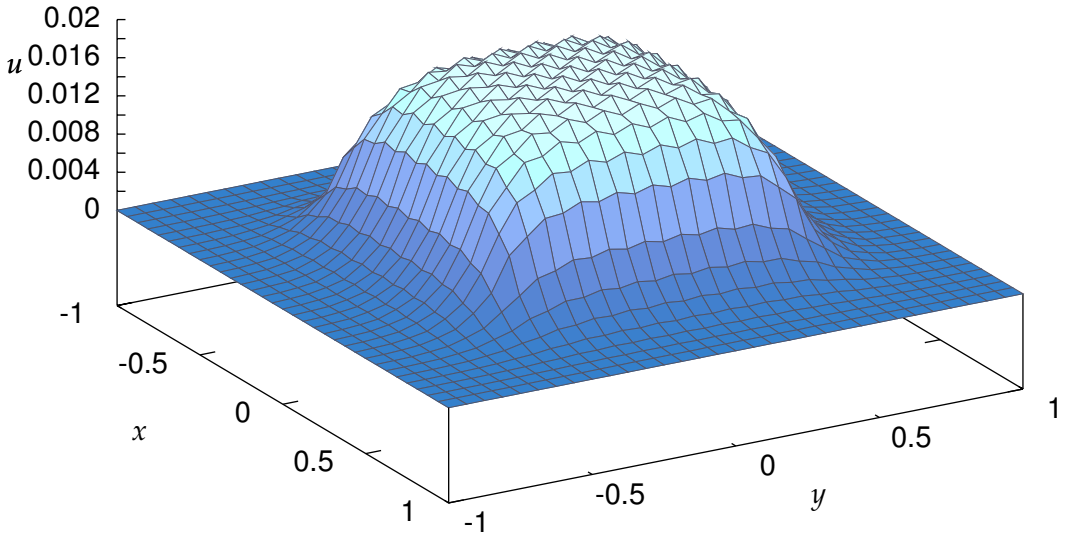


**Figure 4.1:** The Gauss–Seidel solution to the example 2D Poisson problem after ten iterations. The crinkles in the solution are due to the Red–Black update procedure.

# 5 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 $u_{k,j}$ is updated to a new one $u_{k+1,j}$, which is equivalent to displacing $u_{k,j}$ by an amount $\Delta u = u_{k+1,j} - u_{k,j}$. The SOR algorithm works by displacing the values by an amount $\omega\Delta u$, where typically $\omega > 1$, in the hope that if $\Delta u$ is a good direction to move in, one might as well move further in that direction. The iteration is

**SOR method**

**for** $j = 1$ to $N$ **do**

$u_{k+1,j} = (1-\omega)u_{k,j} + \frac{\omega}{a_{jj}}\left(b_j - \sum_{k=1}^{j-1} a_{jk}u_{k+1,k} - \sum_{k=j+1}^{N} u_{k,k}\right)$

**end for**

The corresponding matrix form is

$$(D - \omega L)u_{k+1} = [(1-\omega)D + U\omega]\, u_k + \omega b,$$
$$u_{k+1} = (D - \omega L)^{-1}\,[(1-\omega)D + U\omega]\, u_k + (D - \omega L)^{-1}\omega b. \tag{5.1}$$

Appendix A.3 contains a C++ code to carry out the SOR iteration on the example problem, and the result is shown in Fig. 5.1. In the SOR algorithm, the value of $\omega$ can be chosen freely, and the best choices can be determined by considering the eigenfunctions of the associated problem. This is discussed in more detail below.

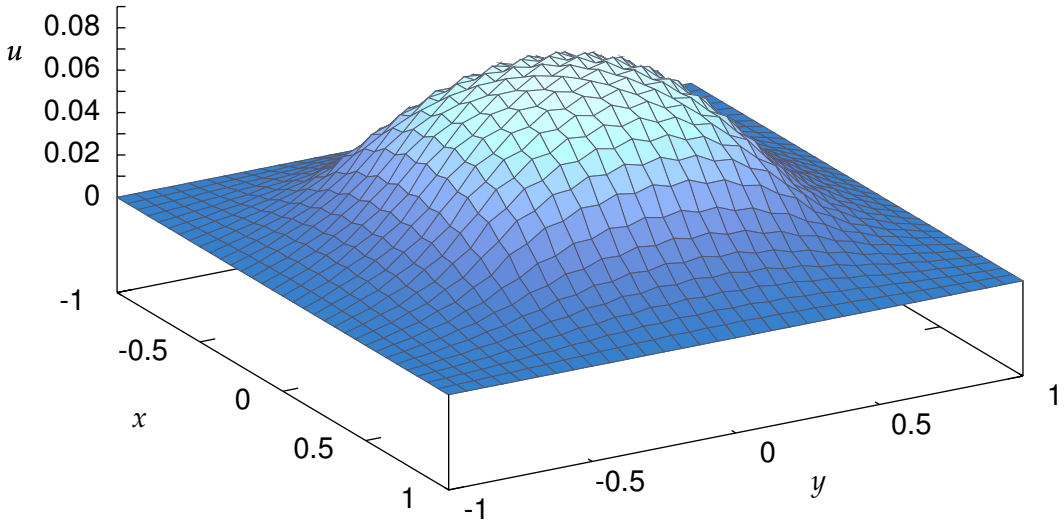


**Figure 5.1:** The SOR solution (using the theoretically optimal $\omega$) to the example 2D Poisson problem after ten iterations. The solution is closer to the answer than the Jacobi or Gauss–Seidel methods.

# 6 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, the matrix of interest is $R_J = D^{-1}(L+U)$. For the 2D Poisson problem, $D = 4I$ and hence

$$R_J = (4I)^{-1}(4I - T_{m\times m}) = I - \frac{T_{m\times m}}{4}. \tag{6.1}$$

The largest eigenvalue of $R_J$ corresponds to the smallest of $T_{m\times m}$, namely

$$\begin{aligned}\lambda_{1,1} &= 4 - 2\cos\left(\frac{\pi}{m+1}\right) - 2\cos\left(\frac{\pi}{m+1}\right) \\ &= 4 - 4\cos\left(\frac{\pi}{m+1}\right).\end{aligned} \tag{6.2}$$

Hence

$$\rho(R_J) = \cos\left(\frac{\pi}{m+1}\right). \tag{6.3}$$

Expanding the cosine as a Taylor series shows that $\rho(R_J) \approx 1 - \frac{\pi^2}{2(m+1)^2}$. The time to gain an extra digit of accuracy is approximately

$$\frac{1}{\log_{10}\rho(R_J)} \propto m^2, \tag{6.4}$$

so the algorithm must be run for $O(m^2)$ iterations to attain a specific level of accuracy. Since there are $O(m^2)$ gridpoints for the 2D problem, the total running time is $O(m^4)$. For detailed proofs of the

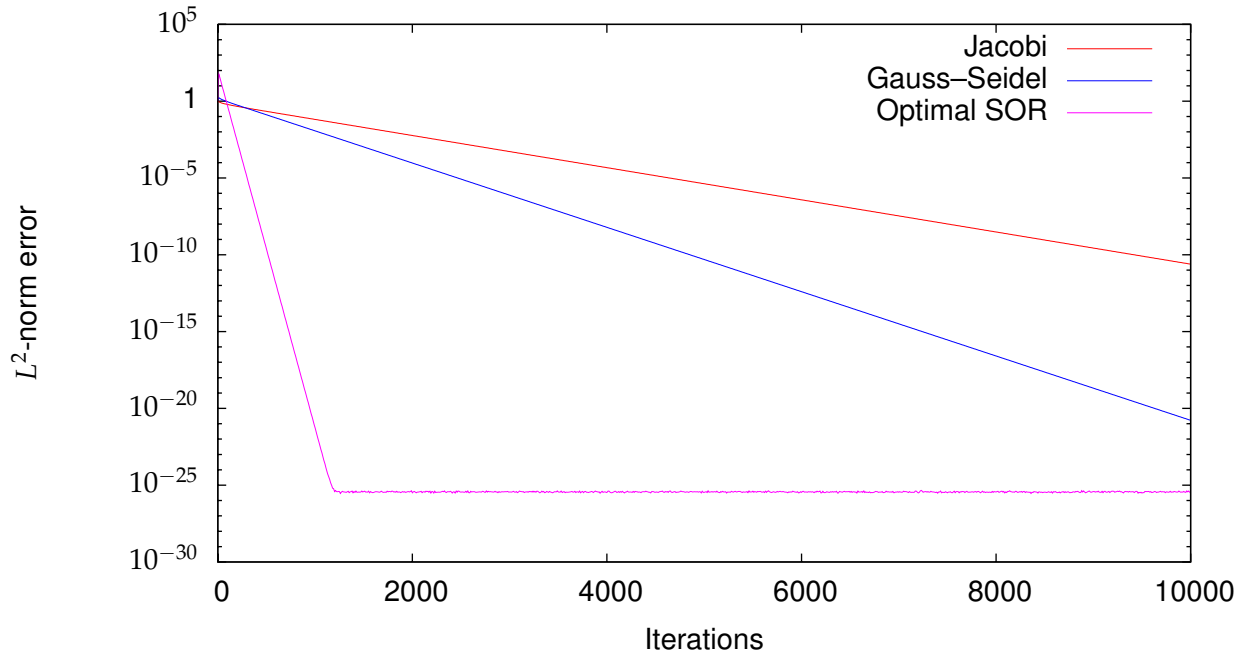


**Figure 6.1:** Errors versus the number of effective iterations for the Jacobi, Gauss–Seidel, and SOR methods, applied to the example 2D Poisson problem on a $65 \times 65$ grid. The plots are in line with the theoretical results of the text. The Gauss–Seidel method is faster than the Jacobi method, 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.

convergence properties of the other methods, the reader should consult Demmel's book [2]. It can be shown that

$$\rho(R_{GS}) = \cos^2\left(\frac{\pi}{m+1}\right), \tag{6.5}$$

so that one iteration of the Gauss–Seidel method is equivalent to two Jacobi iterations. Note however the complexity is the same: $O(m^2)$ iterations are still required to reach a desired level of accuracy. For the SOR algorithm, it can be shown that the optimal value of $\omega$ is

$$\frac{2}{1+\sqrt{1-\rho(R_J)^2}}. \tag{6.6}$$

and that for this value

$$\rho(R_{SOR}) \approx 1 - 2\frac{2\pi}{m+1}. \tag{6.7}$$

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

Figure 6.1 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.

# 7 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 as many iterations as it takes for information to propagate across the grid. As such, we should not expect to ever do better than $O(m)$ 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 out high frequency oscillations, it takes 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 roughly 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 can be achieved. The multigrid method will be covered in an optional activity.

# 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"

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=3.14159265358979323846264338327795;
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] https://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.

[4] Randall J. LeVeque, *Finite difference methods for ordinary and partial differential equations*, SIAM, 2007.