C1 - Assignment 1 Report

Student Number: u1858921

Submission Deadline: 6pm November 7, 2018

Contents

1 Introduction

In this assignment we have been tasked with approximating solutions to problems of the form:

> For $A \in \mathbb{R}^{N\times N}$ and $x, b \in \mathbb{R}^N$, find a solution x to the equation $Ax = b$ where we are given A and b.

using the Gauss-Siedel numerical scheme. The pseudocode/algorithm for the Gauss-Seidel numerical scheme is as follows:

```
Choose $x^{(0)}$
Set $k = 0$
repeat
    for $i = 1$ to $n$ do
        $x_i^{(k+1)} = \left(b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij}x_j^{(k)}\right)/a_{ii}$
    end for
    $r^{(k+1)} = b - Ax^{(k)}$
    $k = k + 1$
until $\|r^{(k)} \leqslant$ TOL
```

One can see that the elements of the $k + 1$ th iteration of the solution, $x^{(k+1)}$, are updated sequentially, which removes the need to store both $x^{(k)}$ and $x^{(k+1)}$ separately, as a result improving efficincy.

The Gauss-Seidel method is a special case of a general class of methods called splitting methods, which are a family of iterative methods which has a form

$$Px^{(k+1)} = Nx^{(k)} + b, \quad \text{or} \quad x^{(k+1)} = x^{(k)} + P^{-1}r^{(k)},$$

where $A = P - N$ is the matrix *splitting*, P is the preconditioner matrix (chosen *easy* to invert), and $r^{(k)}$ is the *residual*, $r^{(k)} = b - Ax^{(k)}$. A standard and obvious splitting is of the form $A = D - (E + F)$, where D is the diagonal of A and E and F are the lower/upper triangular parts of A respectively. The *Gauss-Seidel* method involves taking $P = D - E$ and $N = F$, where P and N are as in the general splitting method described above.

A theoretical result regarding Gauss-Seidel that we shall utilise in this report is the following:

Theorem 1.1
If $A \in \mathbb{R}^{N \times N}$ is symmetric and positive definite, then the Gauss-Seidel iteration converges for any $x^{(0)}$.

Using Theorem 1.1, within the tests, we can guarantee that the algorithm will converge for any guessed initial input $x^{(0)}$. This is particularly useful since this provides a check on the accuracy of the implementation.

In Section 2, we describe the creation of the class `SparseMatrix` into which we then implement, as a member function, the Gauss-Seidel algorithm. In Section 3, we test the implementation of the Gauss-Seidel algorithm with sparse matrices which are both symmetric and positive definite of various dimensions and parameter values. In Section 4, we give a conclusion on the results, describing the outcomes for varying inputs and also provide a possible way to improve the speed of the implementation.

2 Solving Linear Systems Using Gauss-Seidel

Within this section, we describe the implementation of the Gauss-Seidel algorithm, which will be used to invert a given matrix A against the vector b to obtain an approxate solution x of linear system of the form $Ax = b$. We first begin by constructing a C++ class which will create and store sparse matrices (these are the matrices which we shall be working with within this assignment). This method is also applicaple to non sparse matrices, but note that we expect the time taken to obtain an approximate solution will be much longer than when using sparse matrices.

2.1 The `SparseMatrix` Class

We begin by creating the header file SparseMatrix.hh, which in conjuction with the file SparseMatrix.cc will allow for the construction of sparse matrices.

```
class SparseMatrix
{
public:
   SparseMatrix(); // Default Constructor
   SparseMatrix(unsigned int rowSize, unsigned int columnSize); // Default Constructor
   SparseMatrix(const SparseMatrix& matrix); // Copy Constructor
   ~SparseMatrix(); // Destructor

   // Operators
   bool operator==(const SparseMatrix& matrix);
   std::vector<double> operator*(const std::vector<double>& vector) const;

   unsigned int getRowSize() const;
   unsigned int getColumnSize() const;
   double getEntry(unsigned int rowNum, unsigned int colNum) const;
   void addEntry(unsigned int rowNum, unsigned int colNum, double input);
   void printMatrix();

private:
   int unsigned rowSize_, columnSize_;
   std::vector<std::vector<double>* >* rowList_;
   std::vector<std::vector<unsigned int>* >* colIndex_;
};
```

`rowSize_`	Stores the number of rows of the sparse matrix
`columnSize_`	Stores the number of columns of the sparse matrix
`rowList_`	Stores the rows of the sparse matrix in a vector format
`colIndex_`	Stores the indicies of the elements of each row in a vector format

Table 1: Private variables of the `SparseMatrix` class.

One can notice that both `rowList_` and `colIndex_` are vectors of pointers of vectors of pointers. This has been chosen in particular to save on memory since we do not have to store empty vectors when the default constructor is called, but instead just a vector of pointers. The way some of the constructors have been implemented is as follows (done in the SparseMatrix.cc file):

```
// Default Constructor
SparseMatrix::SparseMatrix( unsigned int rowSize, unsigned int columnSize )
{
   rowSize_ = rowSize;
   columnSize_ = columnSize;
   if ( rowSize < 1 || columnSize < 1 )
   {
   throw std::invalid_argument("Cannot create matrix of size 0 or negative size");
   }
   rowList_ = new std::vector<std::vector<double>* > (rowSize);
   colIndex_= new std::vector<std::vector<unsigned int>* > (rowSize);
}

// Destructor
SparseMatrix::~SparseMatrix()
{
   if (rowList_)
   {
      for (std::vector<double>* v : *rowList_)
      {
         delete v;
      }
      for (std::vector<unsigned int>* v : *colIndex_)
      {
         delete v;
      }
   }
   delete(rowList_);
   delete(colIndex_);
}
```

2.2 Implementation of Gauss-Seidel

To implement the Gauss-Seidel numerical scheme, we add to the `SparseMatrix` class the following memeber function:

```
void GaussSeidel(std::vector<double>& x_0, const double TOL, const int maxIter, const
    std::vector<double>& b, std::string myName);
```

which takes as inputs an initial guess, $x^{(0)}$, the vector to be inverted against, b, the tolerance, `TOL` (how close is enough), the maximum number of iterations we would like the algorithm to run for, `maxIter`, and a string, `myName`, into which at each iteration Gauss-Seidel prints the current iteration and the L^∞ norm of the residual. The member function `GaussSeidel` is implemented according to the pseudocode provided above:

```
void SparseMatrix::GaussSeidel(std::vector<double>& x_0, const double TOL, const int
    maxIter, const std::vector<double>& b, std::string myName)
{
   std::ofstream myFile;
   myFile.open(myName, std::ios::out);
   if (!myFile.good())
   {
   throw std::invalid_argument("Failed to open file");
   }
   myFile << "This is the Gauss Seidel data file for: N = " << N << " delta = " <<
       delta << " lambda = " << lambda << std::endl;
   myFile.width(20);
   myFile << std::left << "Iteration k" << "norm(residual_k)" << std::endl;
   int k = 0;
   double sum;
   std::vector<double> residual_k = v_minus_w(b, (*this)*x_0);
   while(inftyNorm(residual_k) > TOL && k <= maxIter)
   {
      for (unsigned int i = 0; i < rowSize_; ++i)
      {
         sum = 0;
         for (unsigned int j : *colIndex_->at(i))
            {
               if (j!=i)
                  {
                     sum += getEntry(i,j)*x_0[j];
                  }
            }
         x_0[i] = (b[i] - sum)/(getEntry(i,i));
      }
      residual_k = v_minus_w(b, (*this)*x_0);
      myFile.width(20);
      myFile << k << inftyNorm(residual_k) << std::endl;
      k++;
   }
std::cout << "Number of iterations: " << k << std::endl;
myFile.close();
}
```

3 Results

The implementation of the Gauss-Seidel method was tested by solving the linear system $Ax = b$ where A was chosen to be a positive definite, symmetric, tridiagonal matrix. This is in particular quite useful for testing, as by Theorem 1.1, the Gauss-Seidel algorithm is guaranteed to converge for any guess $x^{(0)}$. As a starting point for the iteration, we give it an initial guess of $x_0 = \mathbf{0} \in \mathbb{R}^N$ and a tolerance of $\varepsilon = 10^{-6}$. We define the vector $w \in \mathbb{R}^N$ by $w_i = \frac{i+1}{N+1}$ and define matrix A as follows

$$a_{ij} = \begin{cases} -D_{i-1} & j = i-1, \\ D_i + D_{i-1} & j = i, \\ -D_i & j = i+1, \\ 0 & \text{otherwise,} \end{cases}$$

for $i, j \in \{0, \ldots, N-1\}$, with $D_i = a(w_i - \frac{1}{2})^2 + \delta$, $D_{-1} = D_0$ and constants $a = 4(1-\delta)$, $\delta > 0$. We test the `GaussSeidel` algorithm for $N = 100, 1000$ and 10000. We know that if $\delta = 1$, then the solution to the matrix equation will just be w defined above. This gives an excellent opportunity to test the accuracy of the numerical scheme for when $\delta = 1$, and the results for this will be outlined in Table 2. We also further test for the case when the matrix A is replaced with $B = A + \lambda I$, $\lambda \in \mathbb{R}_+$ for various values of λ and present the results in a plot for $N = 100$. In the following, we note that $k \in \mathbb{N}$ denotes the number of iterations.

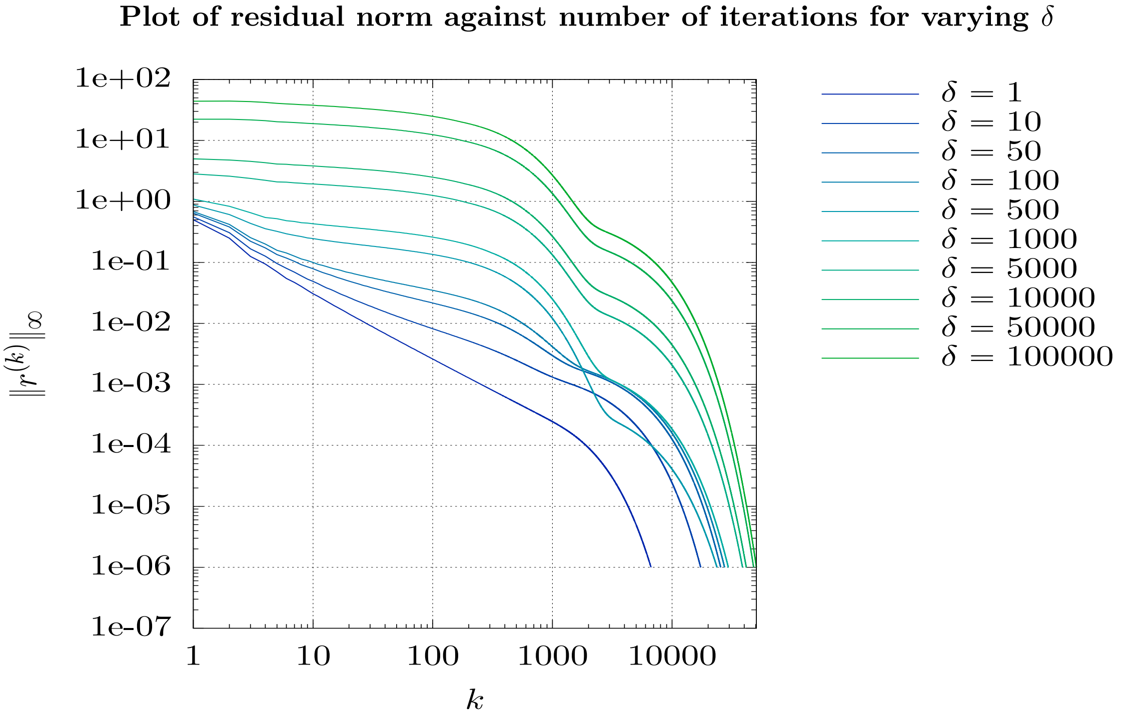

Figure 1: The figure shows a plot of the infinity norm of the kth residual, $\|r^{(k)}\|$, against the number of iterations, k. It can be seen that generally, as δ gets larger, the number of iterations required to get the residual norm below the tolerance `TOL` increases logarithmically.

It can be seen in Figure 1 that as δ gets increasingly larger, the number of iterations required for the residual norm to get below the required tolerance, $\texttt{TOL} = 10^{-6}$, gets expontentially larger. This can be explained since as we increase δ, the spectral raidus of the matrix $B := P^{-1}N$ (chosen as in the introduction) gets increasingly larger, and since the asymptotic rate of convergence, R_∞, of the scheme is given by $R_\infty := -\log \rho(B)$, where $\rho(B)$ denotes the spectral radius of B, the rate of convergence decreases and hence the number of iterations required to reach the tolerance increases. Furthermore, in Figure 2, we observe that as λ gets larger, the number of iterations required for the residual norm to be below the required tolerance rapidly decreases. A possible explanation of this is that as λ gets very large, the order of the matrix λI exceeds that of A, and hence the resulting matrix $B = A + \lambda I$ looks more and more like λI, decreasing the spectral radius of $P^{-1}N$, and hence by a similar argument to that of δ will result in less iterations being required to reach the tolerance.

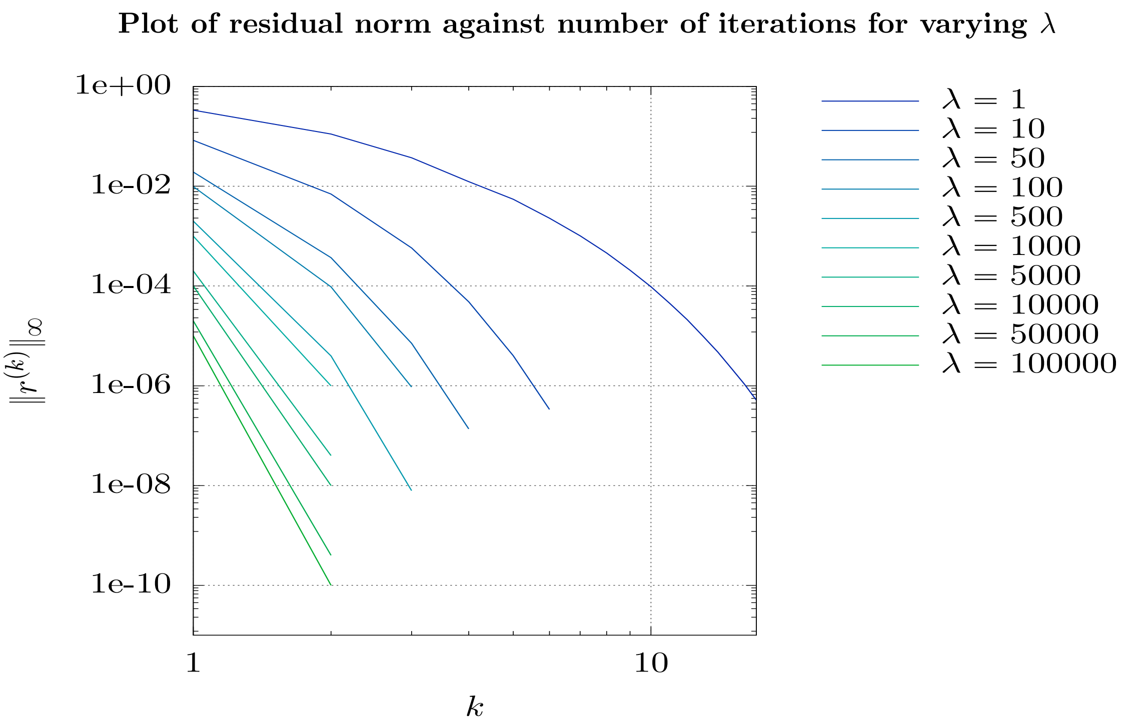

Figure 2: The figure shows a plot of the infinity norm of the kth residual, $\|r^k\|$, against the number of iterations, k. We notice that as λ gets larger, the number of iterations required to get the residual norm below the tolerance `TOL` decreases.

Size of matrix N	Number of iterations k	Error
100	6664	0.00103237
1000	186647	0.101512
10000	242319	0.819303

Table 2: Comparing the number of iterations required to reach the required tolerance, 10^{-6}, along with the error of the aquired solution to the real solution for three different values of N.

4 Conclusion

The `GaussSeidel` numerical scheme works successfully and quite rapidly, even when N is very large, but one can see from Table 2 that the error between the exact solution and the result of the iterative scheme in the maximum norm increases with N. Therefore it would be wise, when dealing with extemely large matrices, one adopts an improved numerical scheme as to obtain more accurate solutions. One could obtain more accurate solutions with the above `GaussSeidel` implementation by chooing an even smaller tolerance for the residual norm, since the norm of the error is related to the residual norm, but this would be done with sacrifice to computation time, which is not a problem if one is patient enough. The implementation of Gauss-Seidel could also be altered to improve the speed of algorithm by removing the use of `getEntry` in the `GaussSeidel` function. This could possibly increase the speed of the execution since it would not require the algorithm to call `getEntry` at every look within an iteration. We could further increase the performance by storing the indices of entries in order when calling `addEntry`, and then implementing a smarter "`getEntry`" type loop within `GaussSeidel`.