LAB 4- ITERATIVE SOLVERS

BY

SACHIN SRINIVASA SHETTY

Contents

1 Introduction **2**

2 Part 1:Basic iterative solver, implementation and testing for small cases **2**

2.1 Testing of Richardson Method 3
2.2 Testing of Jacobi Mehod 3
2.3 Testing of Gauss seidel Method 3

3 Part 2:Moving to Larger matrices **4**

4 Part 3:When Conditioning become bad **6**

List of Figures

1 Generic Code for Basic Iterative Solver 3
2 Optimised matrix-vector product code 4
3 Graph of time taken by Simple and Sparse M-v product implementation 5
4 Size of the matrix vs Number of iterations for *squarex.msh* 7

List of Tables

1 Time and Iterations taken by *square* matrices using Gauss seidel 4
2 Time taken (secs) by *cube* matrices using different matrix-vector product implementations 4
3 Time taken(secs) by *cube* matrices using Sparse matrix-vector product and SuperLU implementations 5
4 Time taken and number of iterations by Gauss seidel and Conjugate Gradient methods for *squarex.msh* 6
5 Time taken by various matrices for Conjugate-gradient and Preconditioned Conjugate gradient 7
6 Number of iterations taken by various matrices for Conjugate-gradient and Preconditioned Conjugate gradient 7

1 Introduction

In this lab, We are going to implement the basic iterative methods and the conjugate gradient method. Also, we will implement two different preconditioning methods for the Conjugate Gradient method. An iterative method is a procedure that uses an initial value to generate a sequence of improving approximate solutions, in which the n-th approximation is derived from the previous ones. Various types of basic iterative methods are implemented and the convergence of these methods is studied. The sparsity property of the matrices is utilized to efficiently implement the matrix-vector product. Then, the conjugate gradient method which is a projection method, and the preconditioning are studied on various matrices obtained from finite elements and also on large matrices. The computational time and the number of iterations taken for convergence are studied for these various implementations.

2 Part 1:Basic iterative solver, implementation and testing for small cases

In this section,the basic iterative methods are studied.The Richardson,Jacobi and Gauss seidel methods are implemented on a small matrix.

For the linear system $Ax = b$,the x^k is the iterative solution at the step k. The residual r^k is the residual at step k defined as $r^k = b - Ax^k$.

For the basic iterative methods,the matrix A is splitted as follows,

$$A = M - N$$

The solution x^{k+1} is defined as follows,

$$x^{k+1} = M^{-1}(Nx^k + b)$$

which is also equivalent to,

$$x^{k+1} = x^k + M^{-1}r^k$$

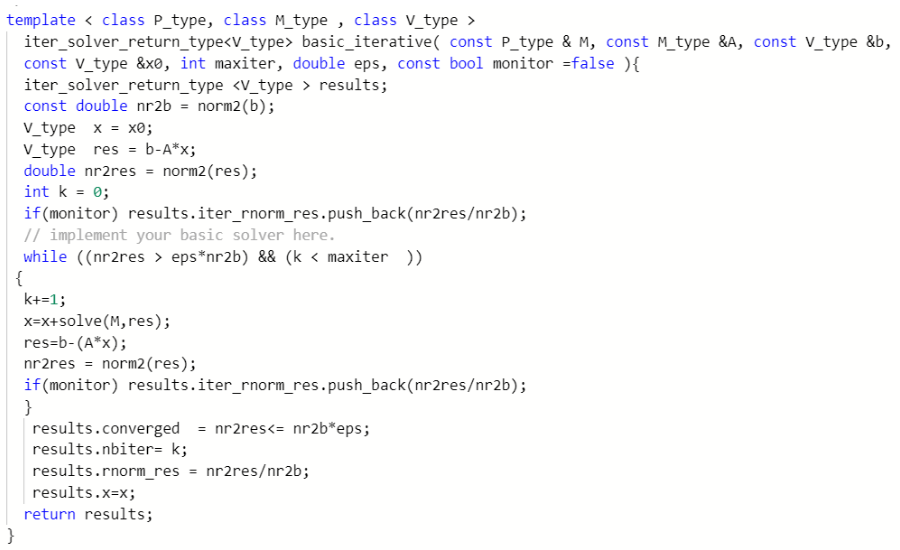
Based on the splitting of A,the different basic iterative methods are defined

- **Richardson Method**
  For Richardson method,
  $$M = I$$
  where I is an Identity matrix and,
  $$x^{k+1} = x^k + r^k$$
- **Jacobi Mathod** For Jacobi method,
  $$M = D$$
  where D is the matrix of diagonal elements of A and,
  $$x^{k+1} = x^k + D^{-1}r^k$$
- **Gauss seidel Method** For Gauss seidel method,
  $$M = D + L$$
  where D is the matrix of diagonal elements of A and L is the lower triangular elements of A and,
  $$x^{k+1} = x^k + (D + L)^{-1}r^k$$

The basic iterative method converges if

$$\rho(M^{-1}N) < 1$$

The following picture shows the generic iterative solver code,it produces the solution by taking the M matrix as the input.



```
template < class P_type, class M_type , class V_type >
  iter_solver_return_type<V_type> basic_iterative( const P_type & M, const M_type &A, const V_type &b,
  const V_type &x0, int maxiter, double eps, const bool monitor =false ){
  iter_solver_return_type <V_type > results;
  const double nr2b = norm2(b);
  V_type  x = x0;
  V_type  res = b-A*x;
  double nr2res = norm2(res);
  int k = 0;
  if(monitor) results.iter_rnorm_res.push_back(nr2res/nr2b);
  // implement your basic solver here.
  while ((nr2res > eps*nr2b) && (k < maxiter  ))
 {
  k+=1;
  x=x+solve(M,res);
  res=b-(A*x);
  nr2res = norm2(res);
  if(monitor) results.iter_rnorm_res.push_back(nr2res/nr2b);
  }
   results.converged  = nr2res<= nr2b*eps;
   results.nbiter= k;
   results.rnorm_res = nr2res/nr2b;
   results.x=x;
  return results;
}
```

Figure 1: Generic Code for Basic Iterative Solver

2.1 Testing of Richardson Method

The Richardson method is tested on the *small_sym_sparse_richardson.mtx*.The epsilon ε is set to 10^{-6}.The method converges in 144 iterations and the time taken for convergence is $6.2192 * 10^{-5}$ sec.

2.2 Testing of Jacobi Mehod

The Jacobi method is test on the *small_sym_sparse.mtx*.The epsilon ε is set to 10^{-6}.The method converges in 20 iterations and the time taken for convergence is $1.6415 * 10^{-5}$ sec.

2.3 Testing of Gauss seidel Method

The Gauss seidel method is test on the *small_sym_sparse.mtx*.The epsilon ε is set to 10^{-6}.The method converges in 9 iterations and the time taken for convergence is $1.4688 * 10^{-5}$ sec.

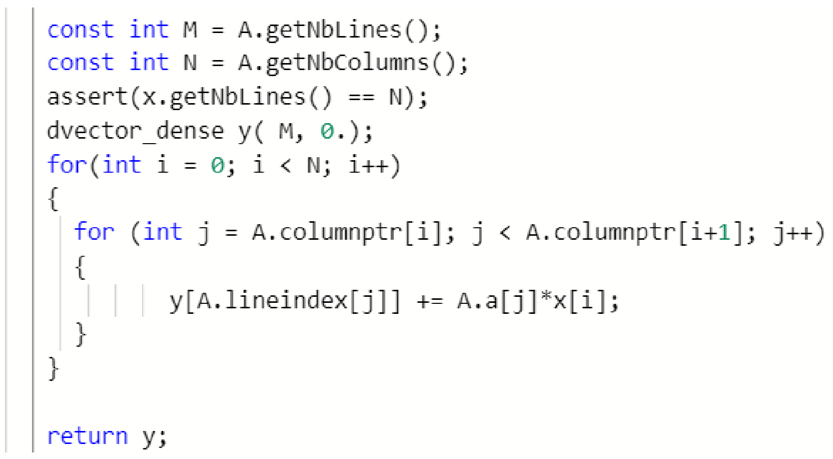
3 Part 2:Moving to Larger matrices

In this section,the basic iterative methods are tested on larger matrices.The Gauss seidel method is tested on the mass matrix from the finite element analysis.The following table shows the time taken and iterations to convergence for different matrices using Gauss seidel.

Mesh	Time Taken	Iterations
square	5.11E-05	10
square2	0.000261597	10
square3	0.0229466	9
square4	0.104743	9
square5	4.63168	9

Table 1: Time and Iterations taken by *square* matrices using Gauss seidel

The iterative methods were implemented using simple matrix-vector product.The sparsity of the matrix A is then utilised to implement the matrix-vector product efficiently.The following code is used to implement the sparse matrix-vector product.



```
const int M = A.getNbLines();
const int N = A.getNbColumns();
assert(x.getNbLines() == N);
dvector_dense y( M, 0.);
for(int i = 0; i < N; i++)
{
    for (int j = A.columnptr[i]; j < A.columnptr[i+1]; j++)
    {
            y[A.lineindex[j]] += A.a[j]*x[i];
    }
}

return y;
```

Figure 2: Optimised matrix-vector product code

The comparison between these are performed on *cube* matrices.The following table and the graph represents the same.

Mesh	Iterations	Gauss (simple M-V product)	Gauss (Sparse M-V product)
cube1	13	4.68E-05	2.91E-05
cube2	12	0.000142648	6.21E-05
cube3	12	0.000451104	0.000121467
cube4	12	0.0518061	0.00343667
cube5	11	7.00832	0.00589752

Table 2: Time taken (secs) by *cube* matrices using different matrix-vector product implementations

It can be seen that the by utilising the Sparsity of the matrix to implement the matrix-vector product,the computational time is significantly reduced. The sparse matrix-vector product implementation is then compared to the implementation using SuperLU.The follow table shows the time-taken and number of iterations by the SuperLU and the Sparse Matrix-Vector product.

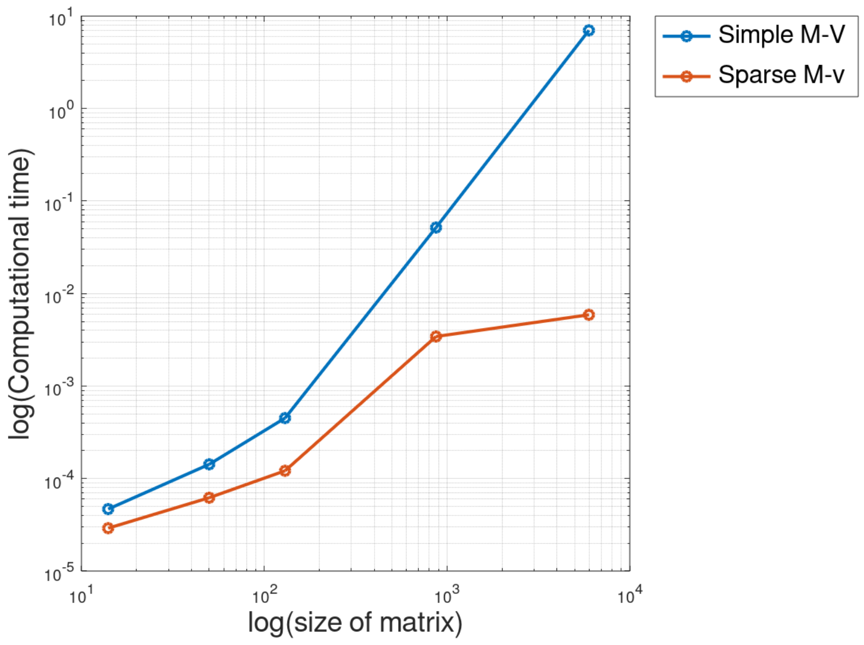


Figure 3: Graph of time taken by Simple and Sparse M-v product implementation

Mesh	Iterations	Gauss (Sparse M-V product)	SuperLU
cube1	13	2.91E-05	0.000218
cube2	12	6.21E-05	0.000459
cube3	12	0.000121467	0.007277
cube4	12	0.00343667	0.020008
cube5	11	0.00589752	1.78344

Table 3: Time taken(secs) by *cube* matrices using Sparse matrix-vector product and SuperLU implementations

It can be observed that the time taken by SuperLU is more than the using the sparsity of the matrix to compute the Matrix-Vector product.

4 Part 3:When Conditioning become bad

The Gauss-seidel method is implement on the Stiffness matrices.It is observed that the stiffness matrix has a higher condition number than the mass matrices.It can be seen that if the gauss-seidel method is implemented on this stiffness matrices,the computation time increases so the method is slow to converge.So,Projection methods are implemented.We are studying one of the projection methods,which is the Conjugate Gradient method.Conjugate Gradient is a projection method used to solve the large symmetric positive definite systems.The following table shows the computational time and iterations taken by the Gauss-seidel method and the Conjugate Gradient method for the *squarex.msh* matrices. The order of the condition number is computed

Mesh/Matrix	Order(k)	GS-time	GS-iterations	CG-time	CG-iterations
square3	10^5	0.459761	23552	0.00146816	74
square4	10^5	8.90696	109627	0.0550791	157
square5	10^6	143.279	500580	0.0864828	333

Table 4: Time taken and number of iterations by Gauss seidel and Conjugate Gradient methods for *squarex.msh*

by taking the reciprocal of the *rcond*.It can be observed that for a higher order of condition number, the gauss seidel method takes much more time to converge and the conjugate gradient method doesn't get significantly influenced by the condition number. So, it converges much faster than the gauss seidel method.

The Conjugate Gradient method can be made more efficient by using a preconditioner. Preconditioning is a technique that is used on the matrix to reduce the condition number of the system. Preconditioning improves the performance of the system and is performed before the Conjugate gradient algorithm. Two Preconditioning methods are studied in this lab, Jacobi Pre-conditioner, and the Incomplete Cholesky preconditioner.

Jacobi Preconditioner
In Jacobi preconditioning,the preconditioner matrix is a diagonal part of the matrix A.

Incomplete Cholesky Preconditioner
In Incomplete Cholesky Preconditioning,the preconditioner matrix is the matrix obtained by the setting the fill-in terms to zero and the cholesky factorisation is performed on this matrix to obtain the incomplete cholesky preconditioner matrix.So,this gives an approximation of factorisation of A.But the incomplete cholesky factorisation can fail even when the full matrix factorisation succeeds.So,if the incomplete cholesky factorisation fails,the diagonal dominance is induced in the matrix as follows,

$$A_m = A + \alpha diag(A)$$

where,α is a parameter known as the diagonal shift coefficient.This diagonal shift helps to increase the diagonal dominance of the matrix,so the Incomplete Cholesky factorisation succeeds.

This preconditioning methods were implemented on various different matrices and the following tables shows the computational time and number of iterations for Conjugate gradient method and the preconditioned conjugate gradient method.It can be observed that with preconditioning the conjugate gradient method becomes more efficient.

Mesh/Matrix	**Order(k)**	**CG**	**CG-J**	**CG-IC**
square3	10^5	0.00146816	0.00152557	0.000878418
square4	10^5	0.0550791	0.00598176	0.0129218
square5	10^6	0.0864828	0.148813	0.0730344
bcsstk07.mtx	10^6	0.127443	0.007275	0.0105772
bcsstk08.mtx	10^7	0.496619	0.0082665	0.00370249
bcsstk14.mtx	10^5	3.50153	0.128213	0.201379
bcsstk15.mtx	10^7	11.7967	0.371018	0.574002

Table 5: Time taken by various matrices for Conjugate-gradient and Preconditioned Conjugate gradient

Mesh/Matrix	**Order(k)**	**CG**	**CG-J**	**CG-IC**
square3	10^5	74	70	34
square4	10^5	157	147	68
square5	10^6	333	304	136
bcsstk07.mtx	10^6	3969	411	227
bcsstk08.mtx	10^7	6879	162	27
bcsstk14.mtx	10^5	14027	409	228
bcsstk15.mtx	10^7	22025	576	280

Table 6: Number of iterations taken by various matrices for Conjugate-gradient and Preconditioned Conjugate gradient

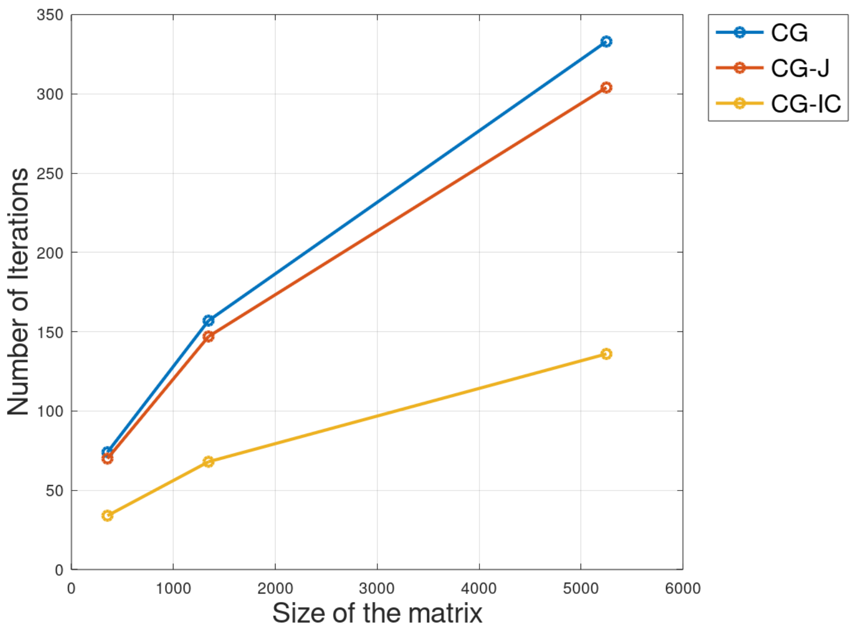


Figure 4: Size of the matrix vs Number of iterations for *squarex.msh*