UNIVERSIDADE ESTADUAL DE CAMPINAS

Faculdade de Engenharia Civil, Arquitetura e Urbanismo

IC639: Métodos Numéricos para Engenharia Civil

List 4

Iterative Methods for Linear Systems of Equations

Student:

Carlos Henrique Chama Puga - 195416

Advisors:

Porf. Dr. Philippe Devloo

Dr. Giovane Avancini

Campinas

2024

Contents

1 Introduction

Iterative methods are widely used for solving linear systems. In comparison with direct methods, the time required to solve the linear system with sufficient accuracy is significantly longer. However, for larger systems, with a high percentage of zeros (sparse matrices), iterative techniques are efficient in terms of computational and storage requirements. Systems of this type usually appear when solving partial differential equations numerically, such as the Finite Element Method.

The main idea behind iterative methods is to solve a $n \times n$ linear system of equations $Ax = b$ starting with an approximation x^0 and generating a sequence of vector x^k that converges to x. In this work, the Jacobi and the Symmetric Successive Over-Relaxation (SSOR) are introduced as preconditioners to the Conjugate Gradient method. As references, the following books were used: (1, 2, 3). The methods are explained for full matrices. However, the implementation is done for sparse matrices.

1.1 The Jacobi's Method

The Jacobi method is obtained by solving the *ith* equation in $Ax = b$ for x_i. For each $k/geq1$, it generates a new approximation x_i^{k+1} of x_i using the previous approximation x^k:

$$x_i^{k+1} = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1, j \neq i}^{n} a_{ij} x_j^k \right). \tag{1.1}$$

The Jacobi method can be written in the matrix form by splitting the matrix A into its diagonal D and off-diagonal $-L - U$ components, in which $-L$ is strictly the negative lower triangle of A and $-U$ is strictly the negative upper

triangle of A. The Jacobi method can be written as:

$$(D - L - U)x = b \Rightarrow Dx = (L + U)x + b,$$

if D is invertible, the matrix form for the Jacobi method reads:

$$x^{k+1} = D^{-1}(L + U)x^k + D^{-1}b, \tag{1.2}$$

in this case, the matrix used as a preconditioner is given by

$$M = D^{-1}. \tag{1.3}$$

1.2 The Gauss-Seidel Method

The main idea behind the Gauss-Seidel method is to use the new approximations x_i^{k+1} as soon as they are computed. The components of x^k are used to compute all the components x_i of x^{k+1}. However, for $i > 1$, the component $x_1^{k+1}, ..., x_{i-1}^{k+1}$ have already been updated and are expected to be better approximations to the solution. The Gauss-Seidel method can be written as:

$$x_i^{k+1} = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{k+1} - \sum_{j=i+1}^{n} a_{ij} x_j^k \right). \tag{1.4}$$

The Gauss-Seidel matrix representation is obtained by multiplying both sides of the equation by the respective diagonal term a_{ii}

$$a_{i,1}x_1^{k+1} + ... + a_{i,i}^{k+1} = b_i - a_{i,i+1}x_{i+1}^k - ... - a_{i,n}x_n^k,$$

using the definitions of D, L, and U previously defined, the Gauss-Seidel method

can be written as:

$$(D - L)x = Ux + b \Rightarrow x = (D - L)^{-1}Ux + (D - L)^{-1}b,$$

assuming that $(D-L)$ is invertible, the matrix form for the Gauss-Seidel method reads:

$$x^{k+1} = (D - L)^{-1}Ux^k + (D - L)^{-1}b, \tag{1.5}$$

The Gauss-Seidel method, however, does not keep the symmetry of the matrix A. Since the Conjugate Gradient method is designed for symmetric positive definite matrices, the Gauss-Seidel method itself cannot be used as a preconditioner.

What is done in practice is to use the Symmetric Successive Over-Relaxation (SSOR) method as a preconditioner for the Conjugate Gradient method. The SSOR method is a combination of the Gauss-Seidel method and the Gauss-Seidel method applied to the transpose of the matrix A. In this case, the matrix used as a preconditioner is given by

$$M = (D - L)^{-1}D^{-1}(D - U). \tag{1.6}$$

1.3 The Conjugate Gradient Method

The Conjugate Gradient method is useful when employed to solve large sparse matrices from symmetric positive definite systems of linear equations. In this section, the non-preconditioned method is explained. First, one defines the inner product of two vectors x and y as

$$(x, y) = x^T y. \tag{1.7}$$

For all the properties used to derive the Conjugate Gradient method, one refers to Section 7.6 of (1). The main idea of the Conjugate Gradient method is to apply concepts of minimizing energy. The vector x^* solves the linear system $Ax = b$ if and only if x^* is the minimizer of the function

$$g(x) = (x, Ax) - 2(x, b). \tag{1.8}$$

It is proved that for any vector $v \neq 0$, $g(x+\alpha v) \leq g(x)$, unless $(v, b-Ax) = 0$. This is fundamental to the Conjugate Gradient method. The search direction v moves from x^k to improve the approximation x^{k+1}, following the path of the steepest descent.

Let $r^k = b - Ax^k$ be the residual at the k-th iteration. If $r \neq 0$ and v and r are orthogonal then $x + \alpha v$ gives a smaller value for g than $g(x)$. The Conjugate Gradient method chooses the search directions $\{v^k\}$ during the iterations such that the residual vectors $\{r^k\}$ are mutually orthogonal.

Given a residual $r^k = b - Ax^k$ and initial search direction v^k, the method follows the procedure of finding the real number α

$$\alpha^k = \frac{(r^k, r^k)}{(v^k, Av^k)}, \tag{1.9}$$

and update the Solution

$$x^{k+1} = x^k + \alpha^k v^k. \tag{1.10}$$

The residual is updated

$$r^{k+1} = r^k - \alpha^k A v^k, \tag{1.11}$$

and a new search direction is computed

$$v^{k+1} = r^{k+1} - \beta^k v^k, \tag{1.12}$$

where

$$\beta^k = \frac{(r^{k+1}, r^{k+1})}{(r^k, r^k)}. \tag{1.13}$$

1.4 The Preconditioned Conjugate Gradient Method

The main reason to precondition the Conjugate Gradient method is to improve the convergence rate. In this scenario, the Conjugate Gradient method is not applied to the original matrix A, but to another positive matrix with a smaller condition number.

The condition number of a matrix A is defined as

$$\kappa(A) = \frac{\lambda_{max}(A)}{\lambda_{min}(A)}, \tag{1.14}$$

where $\lambda_{max}(A)$ and $\lambda_{min}(A)$ are the maximum and minimum eigenvalues of A, respectively. If the condition number is close to 1, the matrix A is said to be well-conditioned. On the other hand, if the condition number is large, the matrix is said to be ill-conditioned.

To maintain the symmetry of the matrix A, the preconditioner matrix M pre and post-multiplies the matrix A in the Conjugate Gradient method

$$\hat{A} = M^{-1} A M^{-T}, \tag{1.15}$$

transforming the original linear system $Ax = b$ into the preconditioned linear system

$$(M^{-1} A M^{-T}) M^{-1} x = M^{-1} b \Rightarrow M^{-1} A x = M^{-1} b.$$

The terms discussed in the previous section are then applied to the preconditioned system. The preconditioned Conjugate Gradient method is given by the following steps: first, the preconditioner matrix M is computed and vector z^k is defined as

$$z^k = M^{-1}r^k, \tag{1.16}$$

then, search direction v^k is initially set to z^k and the method follows the procedure of finding the real number α

$$\alpha^k = \frac{(r^k, z^k)}{(v^k, Av^k)}, \tag{1.17}$$

the solution and residual are updated

$$x^{k+1} = x^k + \alpha^k v^k, \tag{1.18}$$

$$r^{k+1} = r^k - \alpha^k A v^k, \tag{1.19}$$

a new vector z^{k+1} is computed

$$z^{k+1} = M^{-1}r^{k+1}, \tag{1.20}$$

and the real number β is computed

$$\beta^k = \frac{(r^{k+1}, z^{k+1})}{(r^k, z^k)}, \tag{1.21}$$

updating the search direction

$$v^{k+1} = z^{k+1} + \beta^k v^k. \tag{1.22}$$

If M^{-1} is chosen as the inverse of the original matrix A, then the precondi-

tioned Conjugate Gradient method converges in a single iteration. However, it is equivalent to inverting the original matrix A, which is computationally expensive. The choice of the preconditioner matrix M is crucial to the convergence rate of the method and depends on how much the condition number of the matrix A is reduced.

2 Modification in the SparseMatrix Class

Before we properly dive into the implementation of Iterative Methods for solving linear systems, small modifications in the previously implemented SparseMatrix class are required. SparseMatrix class was implemented in List 3 (see Appendix A) and the following modifications are necessary:

- A method to parse a sparse matrix from a file;
- A method to get the diagonal elements;
- A method to perform the inner product between the vector made by the elements of the lower triangle by a vector of the same size;
- The same method as above, but for the upper triangle;

2.1 Parsing a Sparse Matrix from a File

The ParseFromFile method is responsible for reading a file and parsing its content into a SparseMatrix object. Code 1 shows the implementation of this method.

```
def ParseFromFile(self, file:str)->None:
    row = 0
    col = 0
    end_row = False
    self.ptr.append(0)
```

```
6
7     with open(file, "r") as f:
8         lines = f.readlines()
9
10        for line in lines:
11            line = line.strip("= { , \n")
12
13
14            for element in line.split(","):
15                if "}" in element:
16                    element = element.strip("}")
17                    end_row = True
18
19                try: element = float(element)
20                except ValueError: continue
21
22                if element:
23                    self.data.append(element)
24                    self.cols.append(col)
25
26                col += 1
27
28                if end_row:
29                    self.ptr.append(len(self.data))
30                    row += 1
31                    col = 0
32                    end_row = False
33
34    self.size = row
```

Code Listing 1: ParseFromFile method implementation

This method sets the cols, data and ptr vector required to represent a sparse matrix. In line 15, the code checks if the element has a closing bracket, which means that the row has ended. If so, the end_row flag is set to True and the row number is incremented.

Lines 19 and 20 try to convert the element (at this point a string) to a float.

If the conversion is not successful, the code continues the iteration. If the conversion is successful, the element is checked to verify whether it is null. If the element is non-null, it is appended to the data vector and its column to the cols vector.

Lines 28 to 32 update the row and reset the column counter in case the row has ended. The size of the matrix is set to the number of rows found in the file.

2.2 Getting the Diagonal Elements

During the Jacobi method, it is necessary to retrieve the matrix's diagonal elements to perform the update of the residual and the solution. The GetDiagonal method is implemented to perform this task. Code 2 shows the implementation of this method.

```
1 def GetDiagonal(self)->np.array:
2     diag = np.zeros(self.size)
3     for i in range(self.size):
4         diag[i] = self.FindAij(i, i)
5 
6     return diag
```

Code Listing 2: GetDiagonal method implementation

A vector called diag is returned by the method. This vector is filled with the diagonal elements, which are obtained by calling the FindAij method (see List 3 implementation in Appendix A) for each element in the diagonal.

2.3 Performing the Inner Product between the Lower and Upper Triangles and a Vector

Finally, the last modifications are the InnerProductLowerRows and InnerProductUpperRows methods. These methods are responsible for performing the inner product between the lower and upper triangles of the matrix and a

vector, a procedure required in the Gauss-Seidel method. Code 3 shows the implementation of these methods.

```
1  def InnerProductLowerRows(self, vector:np.array, row:int)->np.array:
2      nelem = self.ptr[row+1] - self.ptr[row]
3      ntotal = self.ptr[row]
4
5      sum = 0
6      for j in range(ntotal, ntotal + nelem):
7          if self.cols[j] >= row: continue
8
9          sum += self.data[j] * vector[self.cols[j]]
10
11     return sum
12
13 def InnerProductUpperRows(self, vector:np.array, row:int)->np.array:
14     nelem = self.ptr[row+1] - self.ptr[row]
15     ntotal = self.ptr[row]
16
17     sum = 0
18     for j in range(ntotal, ntotal + nelem):
19         if self.cols[j] <= row: continue
20
21         sum += self.data[j] * vector[self.cols[j]]
22
23     return sum
```

Code Listing 3: InnerProductLowerRows and InnerProductUpperRows methods implementation

The only difference between both methods is that, while the first one searches for elements in the lower triangle (if self.cols[j] >= row - Line 7), the second one searches for elements in the upper triangle (if self.cols[j] <= row - Line 15).

Since the Gauss-Seidel method can be performed backward and forward, both implementations are required. Another method that employs both implementations is the Symmetric Successive Over-Relaxation method, which performs both forward and backward Gauss-Seidel iterations.

With these modifications done, we can now implement the Iterative Methods for solving linear systems.

3 The IndirectSolver Class

This list's main goal is to implement and verify the performance of Iterative Methods for solving linear systems. Herein are implemented and compared the Conjugate Gradient (CG) and the Preconditioned Conjugate Gradient (PCG) methods and as preconditioners, the Jacobi and the SSOR methods. For simplicity, the preconditioned conjugate gradient with Jacobi and with SSOR are here referred to as CG-J and CG-SSOR, respectively.

The IndirectSolver class is responsible for implementing these methods. The class constructor is shown in Code 4.

```
@dataclass
class IndirectSolver:
    A: SparseMatrix
    rhs: np.ndarray
    niter: int
    omega: float = 1.0
    method: callable = None

    resnorm: list[float] = field(init=False, default_factory=list)

    p_k: np.array = field(init=False, default=list)
    res_k: np.array = field(init=False, default=list)

    preconditioner: callable = field(init=False, default=None)
    z: np.array = field(init=False, default=None)
    z_k: np.array = field(init=False, default=None)
```

Code Listing 4: IndirectSolver class constructor

As parameters, the class receives the sparse matrix A, the right-hand side vector rhs and the number of iterations niter. The relaxation parameter omega

and the method to be used are optional parameters, although the method is set after. The resnorm list stores the norm of the residual at each iteration. The p_k, res_k, z, and z_k vectors are used to store the search direction, the residual, and the preconditioned vectors. A preconditioner might be set to the solver.

Hereafter, the methods implemented in the class are presented.

3.1 Solve Method

The solve method is a general function that calls the method set in the constructor to solve the linear system. Code 5 shows the implementation of this method.

```
def Solve(self)->None:
    if not self.method:
        raise ValueError("Method not set")

    sol = ZEROS(len(self.rhs))
    res = self.rhs - self.A.Multiply(sol)

    if self.method == self.ConjugateGradient:
        if not self.preconditioner:
            self.p_k = res.copy()
        else:
            self.z, _ = self.preconditioner(ZEROS(self.A.size), res)
            self.p_k = self.z.copy()
            self.z_k = self.z.copy()

        self.res_k = res.copy()

    self.resnorm = [NORM(res)]
    for i in range(self.niter):
        print(f"Method: {self.method} - Iteration {i}")
        sol, res = self.method(sol, res)
        self.resnorm.append(NORM(res))
```

Code Listing 5: Solve method implementation

Although it is not the purpose of this work, the solve method is implemented in a way that allows the user to choose not only between the CG and PCG. Line 2 checks if a valid method is set. Lines 5 to 17 initialize the required vectors depending on the method set. If the method is the Conjugate Gradient, the search direction p_k is set. If a preconditioner is set, the preconditioned vector z is set.

Line 19 initializes the residual norm vector. At line 20, a loop is performed for the number of iterations set in the constructor and the method performs the solution of the linear system. The residual norm is stored at each iteration.

3.2 Conjugate Gradient Method

The ConjugateGradient method is called by the Solve method and solves the linear system. Code 6 shows the implementation of this method.

```
1  def ConjugateGradient(self, sol:np.array, res:np.array)->tuple[float, list[
       float]]:
2      alpha_k = INNER(res.T, res) / INNER(self.A.Multiply(self.p_k.T), self.
       p_k) if not self.preconditioner else INNER(self.res_k.T, self.z) / INNER
       (self.A.Multiply(self.p_k.T), self.p_k)
3
4      sol += alpha_k * self.p_k
5      res = self.res_k - alpha_k * self.A.Multiply(self.p_k)
6
7      if not self.preconditioner:
8          beta_k = INNER(res.T, res) / INNER(self.res_k.T, self.res_k)
9          self.p_k = res + beta_k * self.p_k
10
11     else:
12         self.z, _ = self.preconditioner(ZEROS(self.A.size), res)
13         beta_k = INNER(res.T, self.z) / INNER(self.res_k.T, self.z_k)
14
15         self.z_k = self.z.copy()
16         self.p_k = self.z + beta_k * self.p_k
17
```

```
18      self.res_k = res.copy()
19      return sol, res
```

Code Listing 6: ConjugateGradient method implementation

Aiming to avoid rewriting the code for the PCG method, the ConjugateGradient method is implemented in a way that allows the user to set a preconditioner. If a preconditioner is set, the method calculates the alpha and beta coefficients using the preconditioned vectors.

Line 2 calculates the alpha coefficient, considering whether a preconditioner is set. The solution is then updated in line 4 and the residual in line 5. If no preconditioner is set, the beta coefficient is calculated in line 8 and the direction is updated in line 10.

Conversely, if a preconditioner is set, the preconditioned vector is updated in line 13, the beta coefficient is calculated in line 14 and the direction is updated in line 18. The residual norm is evaluated in line 20 regardless of the preconditioner.

3.3 The Jacobi Preconditioner

Two preconditioners are implemented in this work: the Jacobi and the SSOR. The Jacobi preconditioner is implemented in Code 7.

```
1 def Jacobi(self, sol:np.array, res:np.array)->None:
2     M = self.A.GetDiagonal()
3     reslocal = res.copy()
4
5     dx = self.omega * np.divide(res, M)
6
7     reslocal -= self.A.Multiply(dx)
8
9     return sol + dx, reslocal
```

Code Listing 7: Jacobi preconditioner implementation

This preconditioner follows the idea behind the Jacobi method, evaluating the matrix M as the diagonal of the matrix A (line 2). The Solution is then updated by the division of the current residual and the matrix M (line 5). The residual for the next step is obtained by subtracting the matrix-vector product of matrix A and the solution update (line 7).

3.4 The SSOR Preconditioner

The second preconditioner implemented is the SSOR. The SSOR's procedure employs the Gauss-Seidel method, performing both forward and backward iterations at each step. Code 8 shows the implementation of this method.

```
1  def GaussSeidelF(self, sol:np.array, res:np.array)->np.array:
2      dx = ZEROS(self.A.size)
3      reslocal = res.copy()
4
5      dx[0] = self.omega * reslocal[0] / self.A.FindAij(0, 0)
6
7      for i in range(1, self.A.size):
8          reslocal[i] -= self.A.InnerProductLowerRows(dx, i)
9          dx[i] += self.omega * reslocal[i] / self.A.FindAij(i, i)
10
11     reslocal = res - self.A.Multiply(dx)
12
13     return sol + dx, reslocal
14
15 def GaussSeidelB(self, sol:np.array, res:np.array)->np.array:
16     dx = ZEROS(self.A.size)
17     reslocal = res.copy()
18
19     dx[self.A.size-1] = self.omega * reslocal[self.A.size-1] / self.A.
    FindAij(self.A.size-1, self.A.size-1)
20
21     for i in range(self.A.size-2, -1, -1):
22         reslocal[i] -= self.A.InnerProductUpperRows(dx, i)
23         dx[i] += self.omega * reslocal[i] / self.A.FindAij(i, i)
```

```
24
25      reslocal = res - self.A.Multiply(dx)
26
27      return sol + dx, reslocal
28
29  def SSOR(self, sol:np.array, res:np.array)->np.array:
30      sol, res = self.GaussSeidelF(sol, res)
31      sol, res = self.GaussSeidelB(sol, res)
32
33      return sol, res
```

Code Listing 8: SSOR preconditioner implementation

As shown in Code 8, the SSOR method calls the GaussSeidelF and GaussSeidelB, performing two iterations at each step (lines 29 to 33). The GaussSeidelF method does a forward iteration, updating and the residual equation-wise (lines 8 and 9). The GaussSeidelB method, on the other hand, performs a backward iteration (lines 18 and 19), updating the residual equation-wise from the last to the first equation.

The updating of the residual is done by the vector inner product with the lower and upper triangles of the matrix A. The solution is updated by the division of the residual and the diagonal of the matrix A. Both methods return the updated solution and residual. However, while the GaussSeidelF uses the initial solution and residual, the GaussSeidelB uses the updated solution and residual from the GaussSeidelF method (lines 30 and 31).

The next section presents the results obtained by the linear system given in class.

4 Comparison of the Conjugate Gradient Method

The main code used to solve the linear system using the CG and PCG methods is shown in Code 9.

```python
from List3.SparseMatrix import SparseMatrix
from IndirectSolver import IndirectSolver

def ParseVector(file: str)->np.array:
    with open(file, "r") as f:
        vector = np.array([float(x) for x in f.read().split(",")])

    return vector

def Main()->None:
    matrix_file = "List4/matrix.dat"
    rhs_file = "List4/rhs.dat"
    results_file = "List4/Results.py"

    niter = 500

    matrix = SparseMatrix()
    matrix.ParseFromFile(matrix_file)

    rhs = ParseVector(rhs_file)

    solver = IndirectSolver(matrix, rhs, niter = niter)

    solver.Set_method("ConjugateGradient")
    solver.Solve()
    resnormCG = solver.resnorm

    solver.Set_preconditioner("Jacobi")
    solver.Solve()
    resnormCGJ = solver.resnorm

    solver.Set_preconditioner("SSOR")
    solver.Solve()
    resnormCGSSOR = solver.resnorm

    with open(results_file, 'w') as f:
        print(f"resnormCG = {str(resnormCG)}", file=f)
        print(f"resnormCGJ = {str(resnormCGJ)}", file=f)
```

```
39          print(f"resnormCGSSOR = {str(resnormCGSSOR)}", file=f)
```

Code Listing 9: Main code to solve the linear system using the CG and PCG methods.

Lines 1 and 2 import the required packages (SparseMatrix, from list 3, and IndirectSolver). Lines 4 to 8 define a function to parse the rhs.dat file and assign these values to a vector.

Line 10 defines the main function. From line 11 to line 22, the sparse matrix object and the solver object are created. Line 24 defines the Conjugate Gradient as the method to solve the linear system. Three different analyses are performed: first without any preconditioner, second with the Jacobi preconditioner and third with the SSOR preconditioner (lines 25 to 34).

Due to the time to iteratively solve the linear system, the results are saved in a file called Results.py, which is later imported in the PlottingResults.py file. Figure 4.1 depicts the results of the convergence, using 500 iterations, for the three different preconditioners.

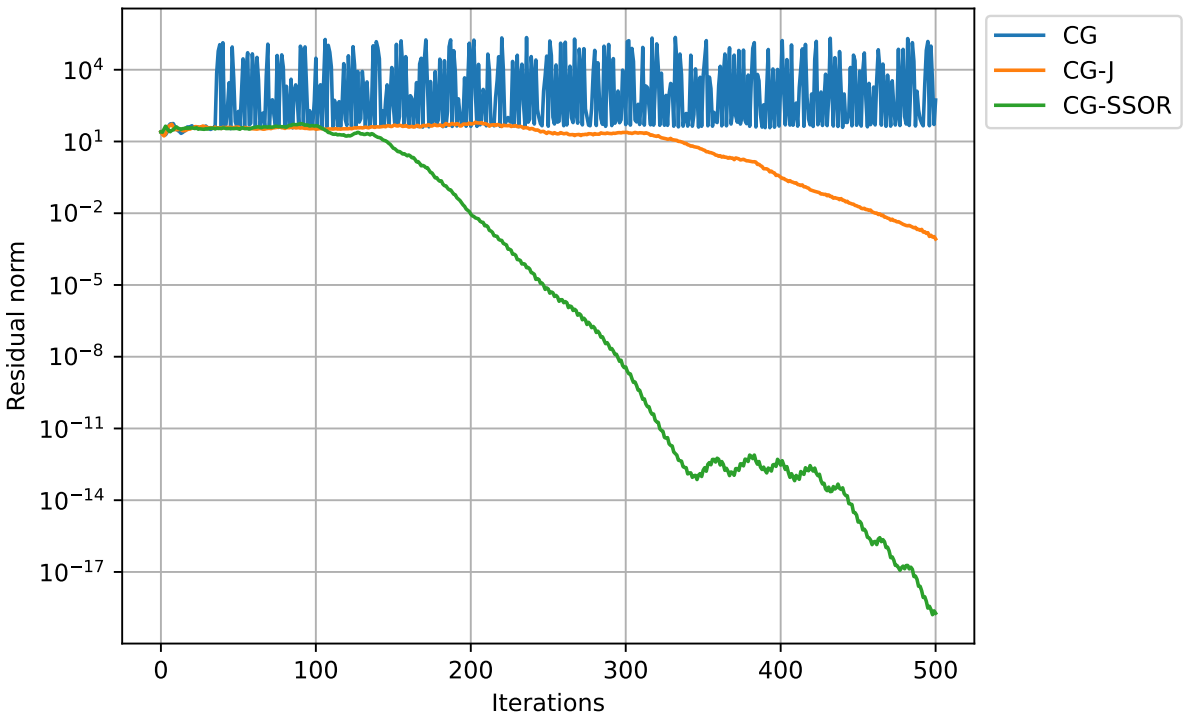

Figure 4.1: Convergence of the Conjugate Gradient method with different preconditioners.

As it's possible to see, the CG without a preconditioner did not converge to the solution. The Jacobi preconditioner improved the convergence and reached an error of 10^{-3} in 500 iterations. The SSOR preconditioner, however, was the most efficient, solving the system in 500 iterations with machine precision.

5 Conclusions

The Conjugate Gradient method is a powerful tool employed to solve large sparse linear systems of equations. However, its convergence rate is directly related to the condition number of the matrix. In this work, the Conjugate Gradient method is applied to solve a 2368×2368 linear system without any preconditioner, and using as preconditioner the Jacobi and SSOR matrices (see eqs. (1.3) and (1.6)).

The results show that the system does not converge to the solution without any preconditioner. For the Jacobi and SSOR preconditioners, the system converges to the solution, but at different rates. For 500 iterations, the Jacobi preconditioner reaches an error of 10^{-3}, while the SSOR preconditioner reaches machine precision. It is important to take into account that the SSOR preconditioner is more computationally expensive since it requires two semi-iterations to solve the linear system.

The use of a given preconditioner can be previously determined by analyzing the matrix A and its condition number. If the matrix is ill-conditioned, then a preconditioned method is recommended to improve convergence. Between the Jacobi and SSOR preconditioners, the choice depends on how much the condition number of the matrix is reduced, which can be determined by analyzing the eigenvalues of the new matrix $\hat{A} = M^{-1}AM^{-T}$ (see Eq. (1.14)). It is known, however, that the SSOR preconditioner usually converges faster than the Jacobi

preconditioner, despite being more computationally expensive.

References

1 BURDEN, R. L.; FAIRES, J. D. *Numerical analysis, brooks*. [S.l.]: Cole publishing company Pacific Grove, CA:, 1997.

2 CUNHA, M. C. de C. *Métodos numéricos*. [S.l.]: Editora da UNICAMP, 2000.

3 GOLUB, G. H.; LOAN, C. F. V. *Matrix computations*. [S.l.]: JHU press, 2013.

A GitHub Repository

The source code for this report and every code inhere mentioned can be found in the following GitHub repository: CarlosPuga14/MetodosNumericos_-2024S1.