Comparison of Stationary and Non-Stationary Iterative Methods and the Differences in Their Effectiveness

Suryansh Varshney (2022519) **Nikita Bhatia(2022323)**
Suryansh Kumar Singh (2022518)

November 19, 2024

1 Introduction

Consider the linear system $Ax = b$, where A is a non-singular square matrix ($\det(A) \neq 0$) of size $n \times n$, and $x, b \in R^n$ (x is the solution vector). This linear system plays an important role in the fields of engineering and computer science. Over the years, various methods have been developed to solve such equations. Direct methods like Gaussian elimination and LU decomposition pose challenges when solving large and complex systems. Hence, iterative methods, such as Gauss-Seidel and Krylov Subspace methods, were devised. Iterative solutions approximate the solution to the linear system, starting with an initial guess and applying iterations to achieve the desired level of accuracy.
Two classifications of the Iterative methods that we will be studying in this paper are:

1. Stationary methods
2. Non-stationary methods.

2 Stationary methods

Stationar methods are easier to understand and uses less computing resources than non-stationary methods. They have a fixed iteration rule that does not change throughout the process. Stationary methods decompose the matrix A into parts and updates the solution at each step. The convergence rate is usually slower as compared to non-stationary iterative methods and it requires A to be diagonally dominant or positive definite. At times, due to the limited usage, it may cause problems with complex and large matrices. These include the Jacobi Method, Gauss-Seidel Method, and Successive Over-Relaxation (SOR). We implemented the first two methods mentioned in C++, and let's discuss them further.

2.1 Jacobi Method

The matrix A can be decomposed into three matrices such that it can be expressed as the sum of these three components:

$$A = L + U + D \tag{1}$$

Where L is the lower triangular matrix, U is the upper triangular matrix, and D is the diagonal matrix.

Consider the matrix A in the form:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \dots & a_{2n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & a_{n3} & \dots & a_{nn} \end{bmatrix}$$

then,

$$L = \begin{bmatrix} 0 & 0 & 0 & \dots & 0 \\ a_{21} & 0 & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & a_{n3} & \dots & 0 \end{bmatrix} \tag{2}$$

$$D = \begin{bmatrix} a_{11} & 0 & 0 & \dots & 0 \\ 0 & a_{22} & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & a_{nn} \end{bmatrix} \tag{3}$$

$$U = \begin{bmatrix} 0 & a_{12} & a_{13} & \dots & a_{1n} \\ 0 & 0 & a_{23} & \dots & a_{2n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 0 \end{bmatrix} \tag{4}$$

Now we know,

$$Ax = b \tag{5}$$

Substituting (1) in (5)

$$(L + U + D)x = b \tag{6}$$

Upon simplifying,

$$x = D^{-1}(L + U)x + D^{1}b \tag{7}$$

then taking initial guess $x = x_0$

$$x_{i+1} = D^{-1}(L + U)x_i + D^{1}b \qquad i = 0, 1, 2 \dots \tag{8}$$

i.e. x ‘s i-th iteration is found by substituting the i-th value of x in the equation.

We keep on iterating the value of x till we get the desired accuracy and the difference between two consecutive iterations is smaller than the error tolerance, meaning that we are close to the true value.

Algorithm is as follows:

```
VectorXd _vector_X_next=VectorXd::Zero(_dimensions.second);
VectorXd _vector_r;
do {
    iteration++;
    for (int i = 0; i < _dimensions.second; i++) {
        double sum=0;
        for (int j = 0; j < _dimensions.second; j++) {
            if (j != i) {
                sum+=(_matrix_A(i,j)*_vector_X(j));
            }
        }
        _vector_X_next(i) = (_vector_b(i)-sum)/_matrix_A(i,i);
    }
    _vector_r = _vector_b - (_matrix_A*_vector_X_next);
    _vector_X = _vector_X_next;
    cout<<iteration<<" "<<_vector_r.norm()<<endl;
} while(twoNorm(_vector_r)>_epsilon);

cout <<"The solver took "<<iteration <<" iterations to complete."<< endl;
return _vector_X;
```

2.2 Gauss-Seidel Method

Consider the matrix $A = L + D + U$, where:

- L: the lower triangular matrix,
- D: the diagonal elements, and
- U: the upper triangular matrix.

These definitions are consistent with those used in the Jacobi method.
Using the same approach, we have:

$$x = -D^{-1}(L + U)x + D^{-1}b$$

Expanding this equation:

$$x = -D^{-1}Lx - D^{-1}Ux + D^{-1}b$$

Taking the initial guess, $x^{(0)}$, So the iterative formula for the Gauss-Seidel method would be:

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

where,

$$\sum_{j<i} a_{ij} x_j^{(k+1)}$$

stands for L, i.e., the lower triangular matrix;

$$\sum_{j>i} a_{ij} x_j^{(k)}$$

stands for U, i.e., the upper triangular matrix; and,

$$\frac{1}{a_{ii}}$$

is D, the diagonal elements.
In the iteration equation, a_{ij} is the coefficient in the A matrix at row i and column j. And, b_i is the i-th component of the constant matrix on the right side of the equation.
Algorithm:

```
VectorXd _vector_r;
do {
    iteration++;
    for (int i = 0; i < _dimensions.second; i++) {
        double sum=0;
        for (int j = 0; j < _dimensions.second; j++) {
            if (j != i) {
                sum+=(_matrix_A(i,j)*_vector_X(j));
            }
        }
        _vector_X(i) = (_vector_b(i)-sum)/_matrix_A(i,i);
        }
    _vector_r = _vector_b - (_matrix_A*_vector_X);
    cout<<iteration<<" "<<_vector_r.norm()<<endl;
} while(twoNorm(_vector_r)>_epsilon);

cout <<"The solver took "<<iteration <<" iterations to complete."<< endl;
return _vector_X;
```

3 Non-stationary methods

However, when the system size becomes larger, the convergence rate becomes slower, and that's when we prefer to use non-stationary methods.
Non-stationary methods are adaptive in nature. Unlike stationary methods, they do not have a fixed iteration pattern. They adjust their search directions and step sizes during iterations. These adjustments accelerate the convergence of the solution and ensure stability.
Several non-stationary methods are based on Krylov subspaces, which progressively approximate the solution by working within spaces spanned by:

$$\text{span}\{r_0, Ar_0, A^2 r_0, \dots\}$$

where $r_0 = b - Ax_0$ is the initial residual.
There are many types of Krylov subspaces, and we have implemented two of them in C++:

- CG (Conjugate Gradient)
- GMRES (Generalized Minimal Residual Method)

3.1 Conjugate Gradient (CG) Method

The **Conjugate Gradient (CG) method** is an iterative solver designed specifically for solving **symmetric positive-definite (SPD)** linear systems of the form:

$$Ax = b$$

where:

- A is a symmetric positive-definite (SPD) matrix.
- b is the constant vector.
- x is the solution vector to be determined.

Key Idea

The CG method minimizes the **quadratic form** associated with A:

$$f(x) = \frac{1}{2}x^\top Ax - b^\top x$$

Instead of searching the entire space like the steepest descent method, CG chooses **search directions** that are **A-orthogonal (conjugate)** to each other. This avoids revisiting previous directions and speeds up convergence.

Steps of the Conjugate Gradient Method

1. **Initial Guess:** Start with an initial guess x_0 and compute the **initial residual**:

   $$r_0 = b - Ax_0$$

   The residual r_0 also serves as the initial search direction p_0.

2. **Iterative Process:** For each iteration k:

   (a) Compute the step size:

   $$\alpha_k = \frac{r_k^\top r_k}{p_k^\top Ap_k}$$

   (b) Update the solution:

   $$x_{k+1} = x_k + \alpha_k p_k$$

   (c) Update the residual:

   $$r_{k+1} = r_k - \alpha_k Ap_k$$

(d) Compute the conjugate direction adjustment factor:

$$\beta_k = \frac{r_{k+1}^\top r_{k+1}}{r_k^\top r_k}$$

(e) Update the search direction:

$$p_{k+1} = r_{k+1} + \beta_k p_k$$

3. **Termination:** The process is repeated until the residual norm $\|r_k\|$ is small enough.

Algorithm that we implemented is as follows:

```cpp
VectorXd _vector_r_current = _vector_b - (_matrix_A*_vector_X_current);
VectorXd _vector_P_current = _vector_r_current.eval();;
int iteration=0;
do{
    iteration++;
    const double r_current_mod = (_vector_r_current.transpose()*_vector_r_current).value();
    const double P_current_mod_A = (_vector_P_current.transpose()*_matrix_A*_vector_P_current).value();
    double alpha_current = r_current_mod / P_current_mod_A;

    VectorXd _vector_X_next = _vector_X_current + alpha_current * _vector_P_current;

    VectorXd _vector_r_next = _vector_r_current - alpha_current * _matrix_A * _vector_P_current;

    const double r_next_mod = (_vector_r_next.transpose()*_vector_r_next).value();

    double beta_current = r_next_mod / r_current_mod;

    VectorXd _vector_P_next = _vector_r_next + beta_current * _vector_P_current;

    //updating current vectors
    _vector_r_current = _vector_r_next;
    _vector_P_current = _vector_P_next;
    _vector_X_current = _vector_X_next;
    cout<<iteration<<" "<<_vector_r_next.norm()<<endl;
}while (twoNorm(_vector_r_current)>_epsilon);

cout <<"The solver took "<<iteration <<" iterations to complete."<< endl;
return _vector_X_current;
```

3.2 Generalized Minimal Residual (GMRES) Method

GMRES is designed for general non-symmetric and non-positive-definite matrices, and hence it works on a broader class of matrices as well.

Key Idea

We get the solution x_k within the Krylov subspace:

$$K_k(A, r_0) = \text{span}\{r_0, Ar_0, A^2r_0, \ldots, A^{k-1}r_0\},$$

with the initial residual being $r_0 = b - Ax_0$.
The main goal is to minimize the residual norm at each iteration so that GMRES converges reliably.
We start with the initial guess x_0, compute the initial residual, and then normalize it to get the first Krylov subspace vector v_1.
GMRES uses the Arnoldi Process to iteratively construct an orthonormal basis V_k for the Krylov subspace using Gram-Schmidt orthogonalization. This is done to reduce A into a Hessenberg matrix H_k.

Hessenberg Matrix

A Hessenberg matrix is a special type of square matrix that is "almost triangular." Either all entries below the first subdiagonal are zero, or all entries above the first superdiagonal are zero. This particular matrix allows fewer computations compared to working with the full matrix A.
The Arnoldi Process is repeated until the residual $\|b - Ax_k\|_2$ is small enough.
After k iterations, the approximate solution x_k lies in the Krylov subspace $K_k(A, r_0)$.

Method

The GMRES method minimizes the residual norm, expressed as:

$$\arg\min_x \|b - Ax_m\|,$$

which can be reduced to:

$$\arg\min_y \|\beta e_1 - H_m y_m\|, \tag{9}$$

where $\beta = \|r_0\|$ and $H_m$is a Hessenberg matrix of the form $m + 1 \times m$.
We use the Givens Rotation Method to efficiently factorise H_m into Q_m and R_m to solve the problem. We get the next x as follows:

$$x_m = x_0 + V_m y_m$$

where V_m is the matrix we get from Arnoldi Process applied on Krylov subspace $K_m(A, r_0)$.

We perform Arnoldi process to m iteration to generate H_m and V_m and then minimise the residual norm using (9) and update the x and r. If the residual norm $\|r_k\|$ is small enough, the method returns with the found x and if not it repeats the Arnoldi process with updates x and r.

Algorithm that we implemented is as follows:

```cpp
VectorXd _vector_r = _vector_b - _matrix_A*_vector_X;
double r_mod = _vector_r.norm();
VectorXd _vector_V_current = _vector_r / r_mod;
_matrix_V.col(0) = _vector_V_current;

int iter = 0;

do {
    iter++;
    _vector_V_current = _vector_r / r_mod;
    _matrix_V = MatrixXd::Zero(_dimensions.second, _max_iter+1);
    _matrix_V.col(0) = _vector_V_current;
    _matrix_H = MatrixXd::Zero(_max_iter+1,_max_iter);

    for (int j=0; j<_max_iter; j++) {
        VectorXd _vector_AV = _matrix_A*_matrix_V.col(j);
        VectorXd sum_vector=VectorXd::Zero(_dimensions.second);

        for (int i = 0; i < j+1; i++) {
            _matrix_H(i,j) = (_vector_AV.transpose()*_matrix_V.col(i));
            sum_vector+=_matrix_H(i,j)*_matrix_V.col(i);
        }

        VectorXd _vector_V_next = _vector_AV - sum_vector;
        _matrix_H(j+1,j) = twoNorm(_vector_V_next);
        if (_matrix_H(j+1,j) > _epsilon) {
            _vector_V_next = _vector_V_next / _matrix_H(j+1,j);
        }
        _matrix_V.col(j+1)=_vector_V_next;
    }

    VectorXd _vector_e1 = VectorXd::Zero(_max_iter+1);
    _vector_e1(0)=1;

    pair<MatrixXd,MatrixXd> R_Qt = GivensRotationQR(_matrix_H);
    MatrixXd _matrix_R = R_Qt.first;
    MatrixXd _matrix_Qt = R_Qt.second;

    VectorXd _vector_G = r_mod*_matrix_Qt*_vector_e1;
    //back-substitution
    VectorXd _vector_y = _matrix_R.topRows(_max_iter).triangularView<Upper>().solve(_vector_G.head(_max_iter));

    //multiplying V_m*y_m
    _vector_X = _vector_X + _matrix_V.leftCols(_max_iter)*_vector_y;
    _vector_r = _vector_b - _matrix_A*_vector_X;
    r_mod = _vector_r.norm();

    cout<<iter<<" "<<r_mod<<endl;
}while (r_mod > _epsilon);
cout <<"The solver took "<<iter <<" iterations to complete."<< endl;
return _vector_X;
```

4 Comparison

We ran our implemented algorithm on 4 different matrices to benchmark them. We used the following matrices:

- 3x3 matrix with condition number 2

$$\begin{bmatrix} 2 & 1 & 1 \\ 1 & 2 & 2 \\ 1 & 1 & 2 \end{bmatrix}$$

- 4x4 matrix with condition number 5

$$\begin{bmatrix} 5 & 2 & 1 & 2 \\ 2 & 5 & 1 & 2 \\ 1 & 1 & 5 & 2 \\ 2 & 2 & 2 & 5 \end{bmatrix}$$

- 5x5 matrix with condition number 7

$$\begin{bmatrix} 6 & 2 & 1 & 1 & 0 \\ 2 & 5 & 2 & 0 & 1 \\ 1 & 2 & 7 & 1 & 0 \\ 1 & 0 & 1 & 6 & 2 \\ 0 & 1 & 0 & 2 & 5 \end{bmatrix}$$

- 6x6 matrix with condition number 10

$$\begin{bmatrix} 10 & 2 & 1 & 0 & 1 & 0 \\ 2 & 9 & 2 & 1 & 0 & 1 \\ 1 & 2 & 8 & 1 & 0 & 0 \\ 0 & 1 & 1 & 7 & 2 & 1 \\ 1 & 0 & 0 & 2 & 6 & 1 \\ 0 & 1 & 0 & 1 & 1 & 5 \end{bmatrix}$$

Following is the benchmark:

Size	**Jacobi**	**Gauss-Seidel**	**CG**	**GMRES**
3x3	Doesn't Converge	23 iterations, 409 μs	2 iterations, 105 μs	9 iterations, 972 μs
4x4	Doesn't Converge	19 iterations, 360 μs	4 iterations, 180 μs	39 iterations, 5150 μs
5x5	76 iterations, 556 μs	25 iterations, 449 μs	5 iterations, 72 μs	88 iterations, 6153 μs
6x6	51 iterations, 1152 μs	15 iterations, 373 μs	5 iterations, 229 μs	44 iterations, 4312 μs

Table 1: Performance of Iterative Solvers ($\epsilon = 10^8$)

Based on the table comparing the performance of different iterative solvers (Jacobi, Gauss-Seidel, CG, and GMRES), here are some key takeaways:

Convergence Behavior

- **Jacobi**:
  - For the **3x3** and **4x4** matrices, Jacobi doesn't converge. This could mean that the method isn't ideal for smaller systems.
  - For **5x5** and **6x6**, Jacobi eventually converges, but it needs **76 iterations (556 µs)** for the **5x5** and **51 iterations (1152 µs)** for the **6x6**. This is pretty slow, and the method takes a lot of time for larger systems.
- **Gauss-Seidel (GS)**:
  - Gauss-Seidel does much better than Jacobi. For the **3x3** matrix, it takes **23 iterations (409 µs)**.
  - For the **4x4** matrix, it's **19 iterations (360 µs)**, and for the **6x6** matrix, it needs **15 iterations (373 µs)**, which shows it's quite efficient compared to Jacobi.
- **Conjugate Gradient (CG)**:
  - **CG** is really fast, especially for small systems. It takes only **2 iterations (105 µs)** for the **3x3** matrix and **4 iterations (180 µs)** for the **4x4**.
  - For the **5x5** and **6x6**, it still performs well with **5 iterations** (**72 µs** for **5x5** and **229 µs** for **6x6**). It's pretty much the winner when it comes to speed, especially for smaller matrices.
- **GMRES**:
  - **GMRES** takes more iterations than the others, especially for the larger matrices. For the **3x3** matrix, it needs **9 iterations (972 µs)**.
  - For the **4x4**, it requires **39 iterations (5150 µs)**, and for the **5x5**, it takes **88 iterations (6153 µs)**. This is a lot compared to the other methods, but interestingly, it's not as slow in terms of total time for the **6x6** matrix, where it takes **44 iterations (4312 µs)**.

Performance Comparison

- **Iterations**:
  - **CG** is by far the best in terms of iterations, needing only a few for most cases, especially for **3x3** and **4x4**.
  - **GMRES** requires a lot more iterations, which shows that it's not the most efficient when it comes to the number of steps needed to converge.
  - **Gauss-Seidel** also performs decently with fewer iterations than Jacobi for most cases.
- **Time**:
  - **CG** is the fastest, completing most of the problems in a short amount of time. For smaller matrices, it beats all the other methods hands down.
  - **Gauss-Seidel** is also pretty fast, taking moderate time, especially for larger systems.
  - **Jacobi** is definitely the slowest for larger systems with more iterations and higher execution times.

  - **GMRES** takes a lot of time for the smaller systems due to more iterations, but for the **6x6**, it comes close to Gauss-Seidel in terms of execution time. Hence, as system gets larger GMRES turns out to be more and more efficient.

General Observations

- **Jacobi** seems to be inefficient, especially for smaller matrices. It takes a lot of iterations and time, so it's not the best choice here.
- **Gauss-Seidel** is a solid choice for small to medium sized systems because it converges in fewer iterations and doesn't take as long as Jacobi.
- **CG** is the clear winner in terms of speed, particularly for smaller matrices. It's fast and needs fewer iterations to converge.
- **GMRES**, though it needs more iterations, it does better with time for larger matrices. However, it's not as efficient as **CG** or **Gauss-Seidel** for smaller systems.

5 Conclusion

CG should be tried to use for the matrix of all sizes if possible. But **CG** can only be used for **SPD** matrices which poses some challenge.
Gauss-Seidel can be used for small and medium sized matrices as it gives good and efficient performance but not as good as **CG**, though it can be applied to wider systems.
GMRES becomes more efficient when handling larger systems with sparse entries and the fact that the method converges no matter what matrix is used (even unsymmetric non-positive definite matrices). Hence, it becomes a solid choice to be used for real life problems dealing with large sparse matrices.
Jacobi shouldn't be used in any case as it may not even converge and has the longest convergence time out of the four. **Gauss-Seidel** naturally becomes a better choice over **Jacobi**.