An Overview on Linear Solvers

Shiyi Chen
Tutor: Leonardo T. Rolla

January 6, 2021

Abstract

The Conjugate Gradient Method (CG) is a prevalent method for solving linear systems. In this report, we include a self-contained description of CG and Steepest Descent (SD) without introducing Krylov space. We also describe the preconditioning and its application on CG (PCG). We then compare SD, CG, PCG by considering their iterations. We find that due to roundoff error, the iteration number can go beyond the theoretical upper bound for SD. Finally, we compare those gradient methods with PLU factorization. The resulting linear solver is given in the conclusion part.

Contents

1 Gradient Methods **2**
1.1 Steepest Descent Method 3
1.2 Conjugate Gradient Method 5
1.3 Condition number 10
1.4 Conditioning 11

2 Convergence **12**
2.1 Relative error measurement 13
2.2 Flops at each iteration 14
2.3 Number of iteration comparison - SD and CG 14
2.4 Number of iteration comparison - CG and PCG 17
2.5 The density for which PCG and CG takes the same amounts of iteration 19

3 Comparison between PLU and Gradient Methods **19**

1 Gradient Methods

Gradient Method is an approach to iteratively minimize or maximize the object function. In this report, we use these methods to solve linear equation $Ax = b$, while A is a symmetric positive definite matrix. Besides iterative methods, PLU factorization can also be applied in solving the linear systems. In this section, we describe several iterative methods, we will compare the PLU factorization for the linear equation solving.

The simplest Gradient Method is Steepest Descent Method which could be traced back to Cauchy in 1847 [5]. In the 20th century, the Conjugate Gradient Method was gradually developed and built on Steepest Descent [4]. Preconditioned Conjugate Gradient Method is a variation of the Conjugate Gradient Method by applying preconditioner in each iteration. In this section, we give a basic yet self-contained introduction on these methods by its application on solving $Ax = b$.

Define the quadratic form as following:

$$f(x) = \frac{1}{2}x^T Ax - b^T x + c, \tag{1}$$

where A is a symmetric positive definite matrix and x is a vector. After differentiating $f(x)$,

$$\nabla f(x) = \begin{pmatrix} \frac{\partial f(x)}{\partial x_1} \\ \frac{\partial f(x)}{\partial x_2} \\ \vdots \\ \frac{\partial f(x)}{\partial x_n} \end{pmatrix} = \frac{1}{2}A^T x + \frac{1}{2}Ax - b = Ax - b. \tag{2}$$

Since A is symmetric positive definite, the global minimum of the quadratic form is achieved when $\nabla f(x) = 0$. Thus, x satisfying $Ax = b$ also minimizes $f(x)$.

Since solving $Ax = b$ is equivalent to minimizing Equation (1), we can seek methods which can help us minimize the quadratic form. SD and CG can help us do that.

1.1 Steepest Descent Method

The method starts by choosing vector x_0. In each iteration, we define d_k as searching direction, α_k as the step size. Thus, we have

$$x_{k+1} = x_k + \alpha_k d_k. \tag{3}$$

We define $\hat{x}$ such that

$$A\hat{x} = b. \tag{4}$$

The principles about Steepest Descent(SD) are:

1) It chooses the searching direction d_k such that $f(x_k)$ decreases the fastest.

2) It chooses the step size α_k such that $f(x_{k+1})$ is minimized in the direction of d_k.

Our purpose for the next step is to find α_k and d_k.

For the searching direction d_k, we know that the direction $\nabla f(x_k)$ gives us fastest ascent on point x_k. Letting

$$d_k = -\nabla f(x_k) \tag{5}$$

gives us the desired direction, we define residual at k-th step - r_k as

$$r_k = b - Ax_k. \tag{6}$$

As a result of Equation (2), Equation (5) and Equation (6), we have

$$r_k = -\nabla f(x_k) = b - Ax_k \quad \text{and} \quad d_k = r_k. \tag{7}$$

The next step is finding step size α_k in each step. We want the partial derivative on the searching line d_k for x_{k+1} to be 0, thus $f(x_{k+1})$ can be minimized, i.e.,

$$\frac{\partial f(x_k)}{\partial \alpha_k} = d_k \cdot \nabla f(x_{k+1}) = 0 \tag{8}$$

By Equation (7), Equation (8) can be written as

$$-d_k \cdot r_{k+1} = 0, \tag{9}$$

$$r_k \cdot r_{k+1} = 0. \tag{10}$$

By Equation (6) and Equation (9),

$$-d_k \cdot (b - Ax_{k+1}) = 0.$$

By Equation (3) and the definition of r_k as shown in Equation (6),

$$-d_k \cdot (b - A(x_k + \alpha_k d_k)) = 0,$$

$$-d_k \cdot (r_k - \alpha_k d_k) = 0,$$

$$\alpha_k = \frac{d_k \cdot r_k}{d_k \cdot Ad_k}. \tag{11}$$

Using Equation (7) again,

$$\alpha_k = \frac{r_k \cdot r_k}{r_k \cdot Ar_k}. \tag{12}$$

We have both α_k and d_k in each iteration now. Moreover, Equation (10) shows that the two adjacent searching directions are orthogonal to each other. Additionally, we have

$$r_{k+1} = b - Ax_{k+1} = b - A(x_k + \alpha_k d_k) = r_k - \alpha_k Ad_k. \tag{13}$$

We will discuss later how we calculate residual r_k in each step, i.e., whether to choose Equation (13) or Equation (6). To sum up, in each iteration of SD we need:

1)calculate the direction r_k by Equation (6),
2)calculate α_k by Equation (12),
3)let $x_{k+1} = x_k + \alpha_k r_k$.

Script 1 presents how we do SD programming in Matlab/Octave. The function takes a symmetric positive definite matrix A, target vector b, initial vector x_0, allowed error e, and allowed maximum iterations m as input. Iteration number i and the result vector x as output. We calculate the error by Equation (14):

$$error = \frac{||b - Ax_k||}{||b||} = \frac{||r_k||}{||b||}. \tag{14}$$

In the initial step, we let `r = norm(b)+e` because we want the statement `norm(r)/norm(b)>e` to be true in the first iteration.

Figure 1 is a visualization of the Gradient Descent Method, the contour map represents the value of the function

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

Script 1 A function for SD

```
function [x,i] = SD(A,b,x0,e,m)
i = 0;   x = x0; r = norm(b)+e;
while i < m && norm(r)/norm(b)>e
    r = b - A * x;
    alpha = r' * r / (r' * A * r);
    x = x + alpha * r;
    i = i + 1;
end
end
```

while $x = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$. We choose $x_0 = \begin{pmatrix} -12.3 \\ 1.25 \end{pmatrix}, A = \begin{pmatrix} 3 & 2 \\ 2 & 6 \end{pmatrix}, b = \begin{pmatrix} 2 \\ -8 \end{pmatrix}, m = 100, e = 0.01$. After 13 iterations, SD finds the minimum value of $f(x)$, and the error is less than 0.01. Figure 1 shows that adjacent searching directions are orthogonal, which corresponds to Equation (10).

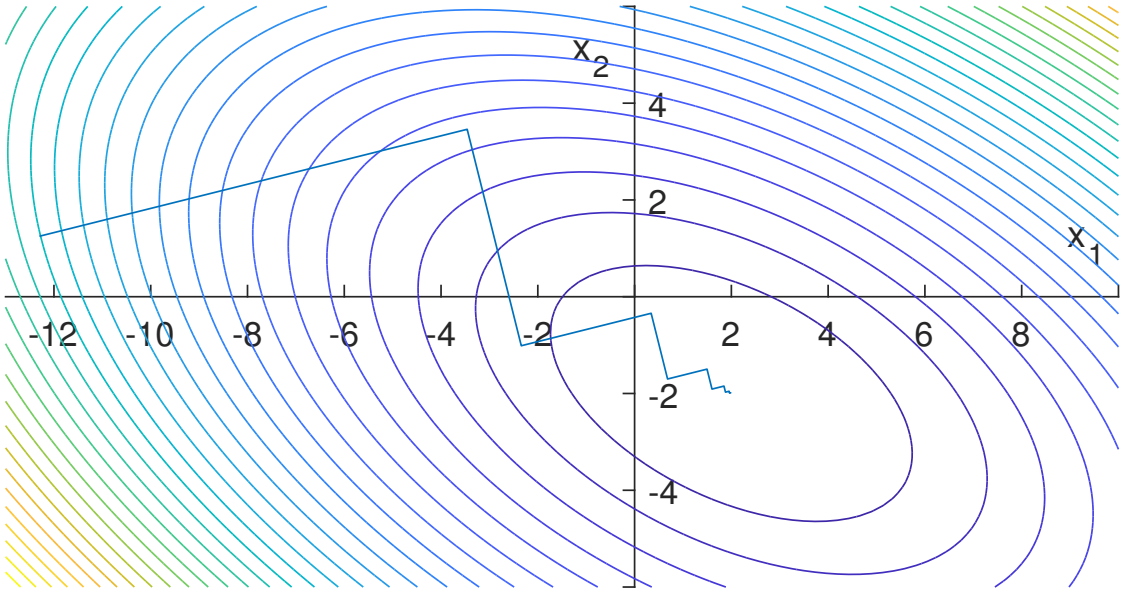


Figure 1: SD on 2D plane.

1.2 Conjugate Gradient Method

The Steepest Descent Method is the easiest gradient method but the least efficient one based on iterations. Figure 1 shows that SD goes in two directions on the 2d plane many times which takes quite a lot of iterations in each direction; we may consider a method going through one direction

in each step and never step back the same direction again, thus it may only take 2 iterations on the 2d plane. Conjugate Gradient Method, in theory - without considering the roundoff error, can manage that.

In the rest of the report, we will consider linear operators from $\mathbb{R}^n$ to $\mathbb{R}^n$. Suppose the searching directions $d_0, d_1, \cdots, d_{n-1}$ make up a basis of $\mathbb{R}^n$ and satisfy

$$d_j \cdot Ad_i = 0 \quad \text{if } i \neq j \text{ and } i, j < n. \tag{15}$$

If two vectors d_j and d_i satisfy Equation (15), we call them *A-orthogonal* to each other or *conjugate*. In the rest of report, we denote the error in k-th step as

$$e_k = \hat{x} - x_k,$$

while $\hat{x}$ is defined in Equation (4). Thus, the initial error e_0 can be uniquely expressed as:

$$e_0 = \hat{x} - x_0 = \sum_{k=0}^{n-1} \alpha_k d_k. \tag{16}$$

Suppose we have already made j steps, we can write e_j as

$$\begin{aligned} e_j &= \hat{x} - x_j \\ &= \hat{x} - x_0 - (x_j - x_0) \\ &= e_0 - \sum_{r=0}^{j-1} \alpha_r d_r \\ &= \sum_{r=0}^{n-1} \alpha_r d_r - \sum_{r=0}^{j-1} \alpha_r d_r. \end{aligned} \tag{17}$$

After n iterations,

$$e_n = \sum_{r=0}^{n-1} \alpha_r d_r - \sum_{r=0}^{n-1} \alpha_r d_r = 0. \tag{18}$$

Equation (18) tells us the method stops exactly at n-th step. In practice, if n is large, the number of iterations is more than n due to the roundoff error, we will discuss the roundoff error problem in Section 2.
For $j < n$, we have,

$$e_j = \sum_{r=0}^{n-1} \alpha_r d_r - \sum_{r=0}^{j-1} \alpha_r d_r = \sum_{k=j}^{n-1} \alpha_k d_k, \tag{19}$$

where d_k should be conjugate to any other searching directions. Since e_{k+1} is a linear combination of d_j's. For $j > k$, by Equation (19),

$$d_k \cdot Ae_{k+1} = 0,$$

By Equation (17),

$$\begin{aligned} d_k \cdot A(\hat{x} - x_{k+1}) &= 0, \\ d_k \cdot A(\hat{x} - x_k - \alpha_k d_k) &= 0 \end{aligned}$$

$$d_k \cdot A(e_k - \alpha_k d_k) = 0 \tag{20}$$

Moreover, by Equation (4) and Equation (17),

$$\begin{aligned} r_i &= b - Ax_i \\ &= A(\hat{x} - x_i) \\ &= Ae_j. \end{aligned} \tag{21}$$

Plug Equation (21) into Equation (20),

$$\alpha_k = \frac{d_k \cdot r_k}{d_k \cdot Ad_k}. \tag{22}$$

Equation (22) tells that α_k in CG is the same as that in SD (11). Just like SD, $f(x_j)$ is minimized in CG in the step j since Equation (8) holds, and the searching direction is orthogonal to the gradient of the next point, since Equation (10) holds.

Equation (19) can be viewed as, in k-th iteration, e_k is eliminated in direction d_{k-1}. This process continues until e_n finally equals to the zero vector.

We already know the expression for α_k in each step, to complete the algorithm, our next goal is to produce a set of $d_1, d_2, \ldots$ which are Conjugate to each other. We use *Conjugate Gram Schmidt Process* [4] to produce a set of conjugate vectors. Different from the normal Gram Schmidt, Conjugate Gram Schmidt produces a set of conjugate vectors instead of orthogonal vectors. The processes for these two versions of Gram Schmidt resemble, they both generate n Conjugate/Orthogonal vectors from n linearly independent vectors. And the j-th vector comes from $v_1, v_2, \ldots, v_{j-1}$. As a result, $\text{span}(u_1, u_2, \ldots, u_j) = \text{span}(d_1, d_2, \ldots, d_j)$.

CG takes $r_1, r_2, \ldots$ as input vectors. In the i-th step,

$$d_i = r_i + \sum_{k=0}^{i-1} \beta_{ik} d_k. \tag{23}$$

Multiply $d_j^T A$ on both sides, we have

$$d_i \cdot Ad_j = d_j \cdot Ar_i + \sum_{k=0}^{i-1} \beta_{ik} d_j \cdot Ad_k \tag{24}$$

Since $d_j \cdot Ad_k = 0$ for all k other than $k = j$, we have,

$$d_i \cdot Ad_j = d_j \cdot Ar_i + \beta_{ij} d_j \cdot Ad_j \tag{25}$$

$$\beta_{ij} = -\frac{r_i \cdot Ad_j}{d_j \cdot Ad_j}. \tag{26}$$

For the next step, we are going to simplify the expression for β_{ij}.

The nice property for us to choose $r_1, r_2, \cdots$ as our input vector is that we will find $\beta_{ij} = 0$ for all $j < i - 1$, as we will show this nice property subsequently.

Multiply $d_i^T A$ $(j < i)$ on both side of Equation (19), by Equation (21),

$$d_i \cdot r_j = d_i \cdot Ae_j = \sum_{k=j}^{n-1} \alpha_k d_i \cdot d_k = 0 \text{ for all } j < i. \tag{27}$$

Thus, r_j is orthogonal to span$(d_1, d_2, \cdots, d_{j-1})$. Since $d_1, d_2, \cdots, d_{j-1}$ is generated by $r_1, r_2, \cdots, r_{j-1}$, by Conjugate Gram Schmidt Process, we have span$(d_1, d_2, \cdots, d_{j-1})$ = span$(r_1, r_2, \cdots, r_{j-1})$. Thus, r_j is also orthogonal to span$(r_1, r_2, \cdots, r_{j-1})$, we have,

$$r_j \cdot r_i = 0 \text{ for all } j < i. \tag{28}$$

Taking dot product r_i with Equation (13), we have,

$$r_i \cdot r_{j+1} = r_i \cdot r_j - \alpha_j r_i \cdot Ad_j.$$

For $j < i - 1$,

$$r_i \cdot Ad_j = \frac{r_i \cdot r_j - r_i \cdot r_{j+1}}{\alpha_j} = 0,$$

$$\beta_{ij} = -\frac{r_i \cdot Ad_j}{d_j \cdot Ad_j} = 0. \tag{29}$$

Script 2 A function for CG.

```
function [x,i] = CG(A,b,x0,e,m,recal)
r = b - A*x0;
d = r; x = x0; i = 0;
while i<m && norm(r)/norm(b)>e
    q = r'*r;  Ad = A*d; d_A_product= d'*Ad;
    alpha = q/d_A_product;
    x = x + alpha*d;
    if mod(i,recal) == 0; r = b-A*x;
    else;                 r = r - alpha*Ad;
    end
    d = r + (r'*r/q)*d;
    i = i + 1;
end
end
```

For $j = i - 1$,

$$\beta_{i,i-1} = -\frac{r_i \cdot Ad_{i-1}}{d_j \cdot Ad_j} = \frac{r_i \cdot r_i}{d_{i-1} \cdot Ad_{i-1}} \frac{1}{\alpha_{i-1}} = \frac{r_i \cdot r_i}{d_{i-1} \cdot r_{i-1}}. \tag{30}$$

Taking r_i dot product with Equation (23),

$$d_i \cdot r_i = r_i \cdot r_i + \sum_{k=0}^{i-1} \beta_{ik} d_k = r_i \cdot r_i. \tag{31}$$

Since all $\beta_{ij} = 0$ except $\beta_{i,i-1}$ by Equation (29), we denote $\beta_{i,i-1}$ by β_i in rest of the report. By (31), Equation (30) can be rewrite as

$$\beta_i = \beta_{i,i-1} = \frac{r_i \cdot r_i}{d_{i-1} \cdot r_{i-1}} = \frac{r_i \cdot r_i}{r_{i-1} \cdot r_{i-1}}. \tag{32}$$

By Equation (30) and Equation (32), we can write the Conjugate Gram Schmidt Process in Equation (23) as

$$d_i = r_i + \beta_i d_i. \tag{33}$$

As we have already shown how we construct searching directions d_k and the step size α_k, we can give a complete algorithm for CG as shown in Script 2. We calculate r_k by Equation (21) every `recal` iterations. We will explain the reason why we calculate r_k with two different formulas in Section 2.3. Using exactly the same A, b, x_0, m, e in the Section 1.1 and set `recal=50`, we get Figure 2.

Compared to Figure 1, CG takes exactly two steps to find the optimal point on the 2d plane, and the two directions are Conjugate to each other.

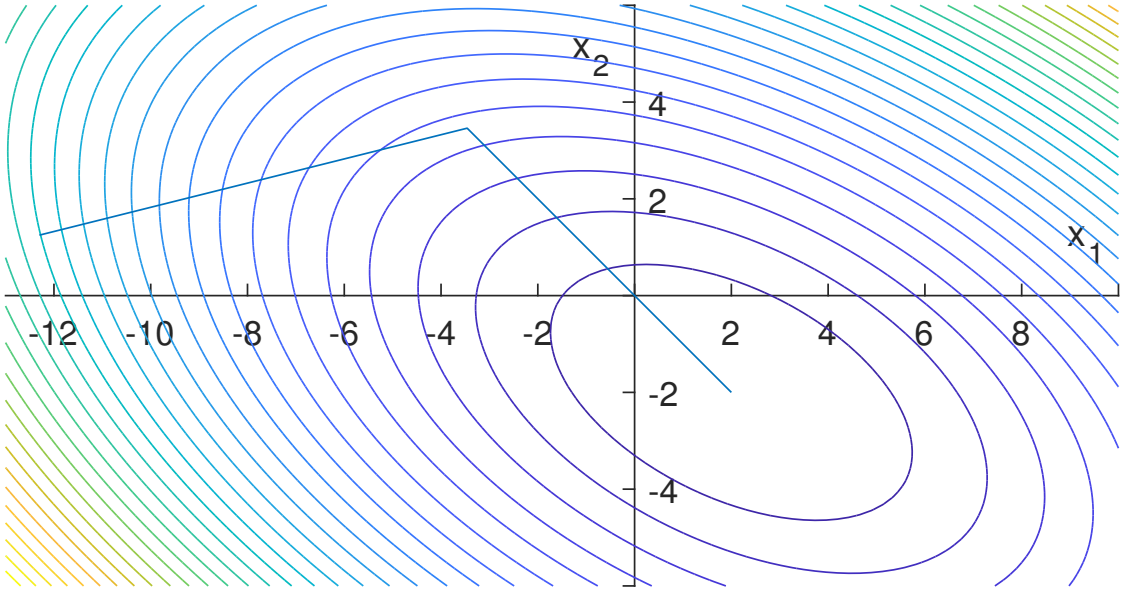


Figure 2: CG on 2d plane.

1.3 Condition number

The condition number is an index to quantify the degree of conditioning. An ill-conditioned matrix has a big condition number.

Mathematically, condition number κ describes the worst case of input that can change with a certain amount of output change. Suppose the linear system is $Ax = b$ and when we add perturbation p [3] on the right hand side, the system becomes

$$A\bar{x} = b + p. \tag{34}$$

Defining the relative changes on the right hand side, the left hand side can be written as

$$e_b = ||p||/||b|| \text{ and } e_x = ||x - \bar{x}||/||x||. \tag{35}$$

Thus, the change can be expressed as

$$e_x/e_b = \frac{||\bar{x} - x||}{||x||} \frac{||b||}{||p||}. \tag{36}$$

We define the norm of a matrix as

$$||A|| = \max(||Ae||) \quad \text{while } ||e|| = 1. \tag{37}$$

The following inequality follows from the definition

$$||b|| = ||Ax|| = ||x|| \cdot ||Ae|| \leq ||A|| \cdot ||x||.$$

Thus,

$$\begin{aligned} e_x/e_b &= \frac{||\bar{x} - x||}{||x||} \frac{||b||}{||p||} = \frac{||A^{-1}p||}{||x||} \frac{||b||}{||p||} \\ &\leq \frac{||A^{-1}|| \cdot ||p||}{||x||} \frac{||b||}{||p||} \leq ||A^{-1}|| \cdot ||b|| \cdot \frac{||A||}{||b||} \\ &= ||A^{-1}|| \cdot ||A|| \\ &= \max(\lambda_i(A)) \cdot \max(\lambda_i(A^{-1})) \quad \text{(for symmetric random matrices)} \\ &= \frac{\max(\lambda_i(A))}{\min(\lambda_i(A))}. \end{aligned}$$

Since x_x/x_b is bounded by $\frac{\max(\lambda_i(A))}{\min(\lambda_i(A))}$, we let the upper bound be the condition number, i.e.,

$$\kappa(A) = \frac{\max(\lambda_i(A))}{\min(\lambda_i(A))}. \tag{38}$$

In Matlab/Octave, in-built `cond` function will give us the condition number of the matrix.

1.4 Conditioning

CG is fast and easy to implement and it should converge at exactly n steps by Equation (18). However, in practice, different matrices with the same size may take a different number of iterations. Some matrices may be hard for CG to deal with, we call this kind of matrices *ill-conditioned*.

We will encounter the situation when A is ill-conditioned sometimes, i.e., $\kappa(A)$ is large. Under this situation, CG converges slowly. Preconditioning is a strategy to make the condition number of the linear system smaller. We do preconditioning by pre-multiplying M^{-1} on both sides of the equation $Ax = b$, the symmetric positive definite matrix M is called preconditioner, and we can write the matrix M^{-1} as LL^T.

Thus, solving $Ax = b$ is equivalent to solve $(L^TAL)(L^{-1}x) = (L^Tb)$. Let $\hat{A} = L^TAL$, $\hat{x} = L^{-1}x$, $\hat{b} = L^Tb$, $Ax = b$ becomes

$$\hat{A}\hat{x} = \hat{b}. \tag{39}$$

We do CG on Equation (39), we have following equivalence:

$$\hat{r}_n = L^T r_n \quad \hat{d}_n = L^{-1} d_n. \tag{40}$$

It will be nice if we can get rid of L^T, L^{-1} in our iterations.
By Equation (13), we have

$$\begin{aligned}\hat{r}_n = L^T r_n &= \hat{r}_{n-1} - \hat{\alpha}_n \hat{A}\hat{d}_{n-1} \\ &= L^T r_{n-1} - \hat{\alpha}_n (L^T A L) L^{-1} d_{n-1} \\ &= L^T (r_{n-1} - \hat{\alpha}_n A d_{n-1}).\end{aligned}$$

Hence,

$$r_n = r_{n-1} - \hat{\alpha}_n A d_{n-1}. \tag{41}$$

Similarly, Let $s_n = M^{-1} r_n$, we have

$$\begin{aligned}\hat{\alpha}_n &= \frac{r_{n-1}^T s_{n-1}}{d_{n-1}^T A d_{n-1}}, \\ d_n &= s_n + \hat{\beta}_n d_{n-1}, \\ \hat{\beta}_n &= \frac{r_n^T s_n}{r_{n-1}^T s_{n-1}}.\end{aligned}$$

Compared to CG, in PCG, we need to calculate $M^{-1} r_n$ in each iteration. It is important to choose our preconditioner M. If we choose $M = I$, the PCG process will be identical to CG. If we choose $M = A$, we need to solve $A r_{n+1} = r_n$ in each iteration which does not simplify the process at all.

There are many ways to choose M, here, we choose M to be the diagonal of the A, this conditioner is called *Jacobi preconditioner* [4]. It is important to notice that the diagonal entries of A cannot be zero for the Jacobi preconditioner. In the rest of the report, whenever we mention PCG method, we mean PCG by Jacobi preconditioner.

2 Convergence

The convergence is determined by the condition number of A, the choice of x_0 and b. Theoretically, the number of iteration equals the size of the matrix A. However, due to the roundoff error, when the condition number is big, it usually takes more than n. Shewchuk show us the error term in

the i-th iteration in SD can be bounded by $(\frac{\kappa-1}{\kappa+1})^i||e_0||_A$ in Chapter 6.2[4]. Thus,

$$e_i \cdot r_i = e_i \cdot Ae_i = ||e_i||_A \leq (\frac{\kappa-1}{\kappa+1})^i||e_0||_A. \tag{42}$$

Similar conclusion can be drawn from CG as he show us in his Chapter 9.2,

$$e_i \cdot r_i = e_i \cdot Ae_i = ||e_i||_A \leq (\frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1})^i||e_0||_A. \tag{43}$$

We find that by substitute e_i to r_i in the left hand side of Inequality (42) and Inequality (43), his conclusion still holds, i.e.

$$\begin{aligned} r_i \cdot r_i = ||r_i|| \leq (\frac{\kappa-1}{\kappa+1})^i||r_0||, \\ r_i \cdot r_i = ||r_i|| \leq (\frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1})^i||r_0||. \end{aligned} \tag{44}$$

We will numerically see these two inequalities immediately.

2.1 Relative error measurement

We use two ways to measure errors in each iteration. We used to take the same relative error measurement as Shewchuk, i.e.,

$$\tilde{e}_i = \frac{||r_i||}{||r_0||} \tag{45}$$

This measurement has no problem for small linear systems. However, it gives inaccuracies in our runtest on big matrices. Here is how we generate our matrices: we generate a symmetric semi-positive definite matrix by firstly generate a random matrix using `a=randi(n,n)` command in Matlab/Octave, then we let `A = a'*a` ($A = a^T a$). When we increase the size n, the magnitude of the entries in A increases accordingly. We generate x_0 by letting `x0 = rand(n,1)`, also generate `b` by letting `b = rand(n,1)`. Since the magnitude of entries in A is much larger than that in x_0 and b, the norm of Ax_0 will be large. Thus, in Equation (45), the denominator $||b - Ax_0|| = ||r_0||$ is large. For a given $\tilde{e}$, allowed $||r_i||$ will also be large, this gives us inaccurate result.

The solution is changing our error measure to be:

$$\tilde{e}_i = \frac{||r_i||}{||b||}. \tag{46}$$

Method	Flops of each iteration
SD	$4n^2 + 7n + 2$
CG	$4n^2 + 11n + 3$
PCG	$4n^2 + 12n + 3$

Table 1: Flops at each iteration.

The allowed $||r_i||$ will be much smaller than that in Equation (45). We will use Equation (46) to measure the relative error in the rest of the report.

2.2 Flops at each iteration

We compare the flops for SD, PCG and CG for each iteration in this section. Suppose the size of the matrix is n, consider SD is done in Script 1, CG done in Script 2 and PCG is done with Jacobi preconditioner. We will use Equation (6) to calculate r_k in each iteration for SD, CG, PCG. Table 1 is the result.

From Table 1, SD, CG and PCG all have the same leading term at each iteration. When counting the total flops, the number of iteration matters more than the flops of each iteration.

2.3 Number of iteration comparison - SD and CG

We compare the number of iteration for CG and SD working on matrices generated by method described in Section 2.1 in script `runtest.m`. In this script, we generate matrices with sizes from `2^strt` to `2^stop`, then do CG and SD to them separately. For each of the matrices, we calculate the condition number of the matrices using the in-built `cond` function. If one of the iterations exceeds the maximum number of iteration `m`, CG or SD fails and we drop that data from our sample. We do this process repeatedly. Users can change all the variables in the workspace including the repeat rounds `repeat` - maximum iteration rounds allowed as described in a variable called `m`. For the output, the program generates a plot about condition numbers vs iterations, a plot about size of the matrix vs iterations (see Figure 5 for example), a TXT report containing the description about the test and all the data in the test. In the meanwhile, there are messages about the execution process appearing on the terminal.

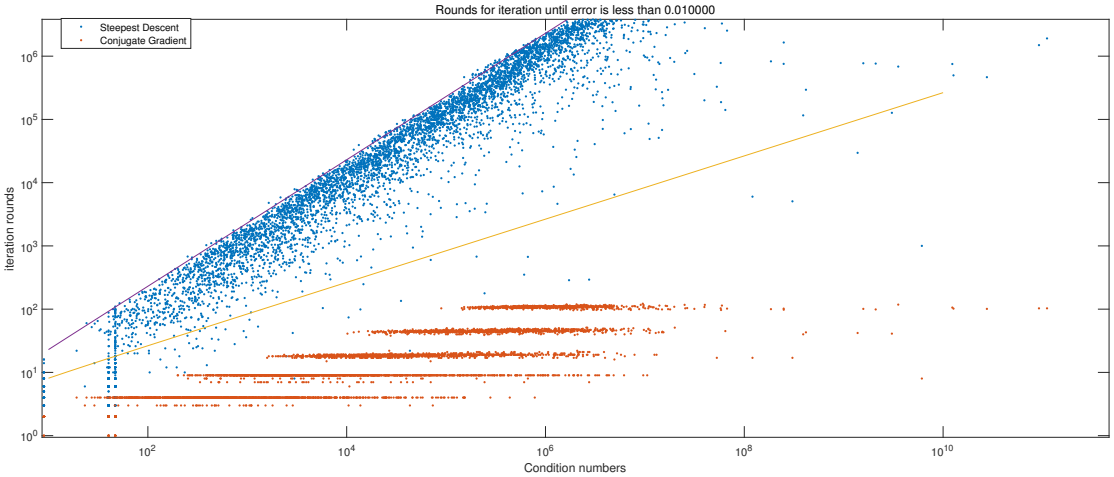


Figure 3: Result by `runtest.m`.

By setting `m = 4000000, strt = 1, stop = 7, e = 0.01, repeat = 1200`, running few hours on Matlab, we get Figure 3. The purple and orange lines are theoretical boundaries deducing from Equation (44) for CG and SD respectively. For the CG in red dots, each cluster belongs to one matrix size. We use Equation (13) to calculate the residual for both SD and CG to minimize the roundoff error in each iteration. We have most of the points below the theoretical boundary with few of them above the boundary of SD, two hypotheses are:

1)The measure for relative error is different i.e. when we are doing SD, we measure relative error using Equation (46) while we get the theoretical boundary measuring error by Equation (45).

2)Those points out of SD boundary are big matrices. Since the magnitude of entries in big matrices are with larger magnitude, roundoff errors will be more significant than small matrices with smaller entries.

As for hypothesis 1, we cannot generate matrices along the boundary for CG using the Equation (45), thus we have not verified this hypothesis yet.

Hypothesis 2 can be easily verified, we only need to keep track of the size of those matrices which go beyond the SD theoretical boundary. We change the variables to be `m = 4000000, strt = 3, stop = 6, e = 0.01, repeat = 120`. Ideally, there should be $120 \cdot 4 = 480$ points, however, 71 of them reach the maximum iterations `m`, we do not count them into

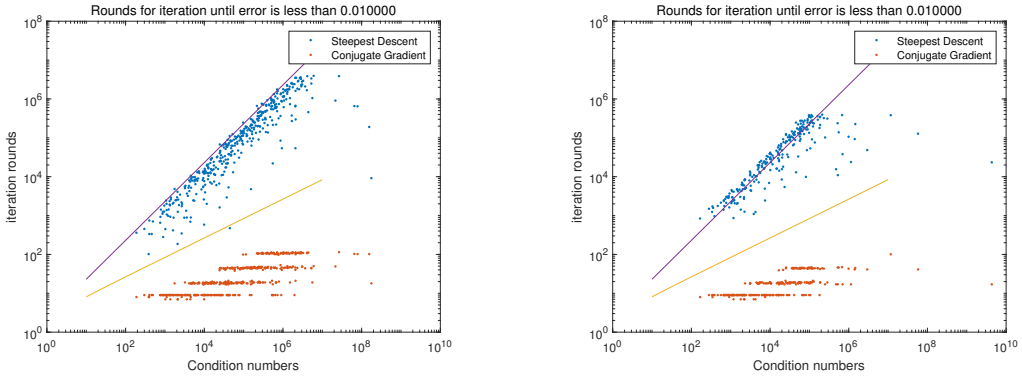
size of the matrix	number of iteration	median iteration
64	708,530 398,180 300,630 288,817 236,148 217,764 97,124	1,042,116
32	52,428 44,013 24,735	192,995
16	4,540 1,760	35,482

Table 2: Information for 12 points that go beyond the boundary.

the sample points. In the 409 sample points, 12 of them go beyond the boundary (2.93%). Table 2 gives detailed information about those 12 points. From the table, matrices with larger sizes are more likely to go beyond the boundary. Also, those matrices above boundary have smaller numbers of iteration compared to the median number of iteration with the same size. This aligns our second hypothesis - with fixed condition number, bigger matrices (with big entries) are more likely to go beyond the boundary.

Another thing we find is that when we change the residual formula to Equation (13), the number of points that go beyond the boundary increases significantly. Equation (13) takes fewer flops than Equation (6) in each iteration but Equation (13) is less accurate since no feedback is given from b, the roundoff error accumulates in each iteration. Thus, it requires more iterations to converge. When we use Equation (6) to calculate r_k in every iterations, using the same parameters discussed above, 59.35% of points go beyond the boundary as Figure 4(b) shown.

As a conclusion for this section, we know that hypothesis 2 is true. As for hypothesis 1, we suggest future research to investigate in by changing the way of generating the matrices, i.e. push the points around the boundary while keep varieties of condition numbers. For next stage of research, we would choose CG as our linear equation solver because it takes less iterations than SD. Also, we need to try to strike the balance between the flops and iteration rounds by choosing residual formula al-



(a) Residual by Equation (6) (b) Residual by Equation (13)

Figure 4: Iterations using different residual formulas.

ternatively as we had already implemented in our algorithm of CG in Script 2.

2.4 Number of iteration comparison - CG and PCG

In this section, we compare the numbers of iteration for CG and PCG by considering two main situations - the matrices are dense and the matrices are sparse. We generate a random symmetric dense matrix first by letting `A = a'*a, a = rand(n,n)`. For a sparse matrix, we take the diagonal of the dense matrix we just generated, then adding elements off-diagonal randomly until the density of the matrix is fulfilled. We set $density = \frac{2}{n}$ at first.

Similar to `runtest.m`, we generate a series of matrices with sizes are multiples of two (either sparse or dense), run CG and PCG for them separately for the rounds executed until the methods converge.

As shown by Figure 5, for the dense matrices, iterations of CG and PCG are almost the same, with CG taking fewer flops. However, PCG takes fewer flops for the sparse matrices.

For the flops in each iteration, PCG takes more flops as shown in Table 1. Since M^{-1} is a diagonal matrix, it only takes $2n$ flops for that step, which is not the leading term as shown in Section 3.

To sum up, the result suggests us to use CG for dense matrix while PCG for sparse matrices, but we do not know which density shall we

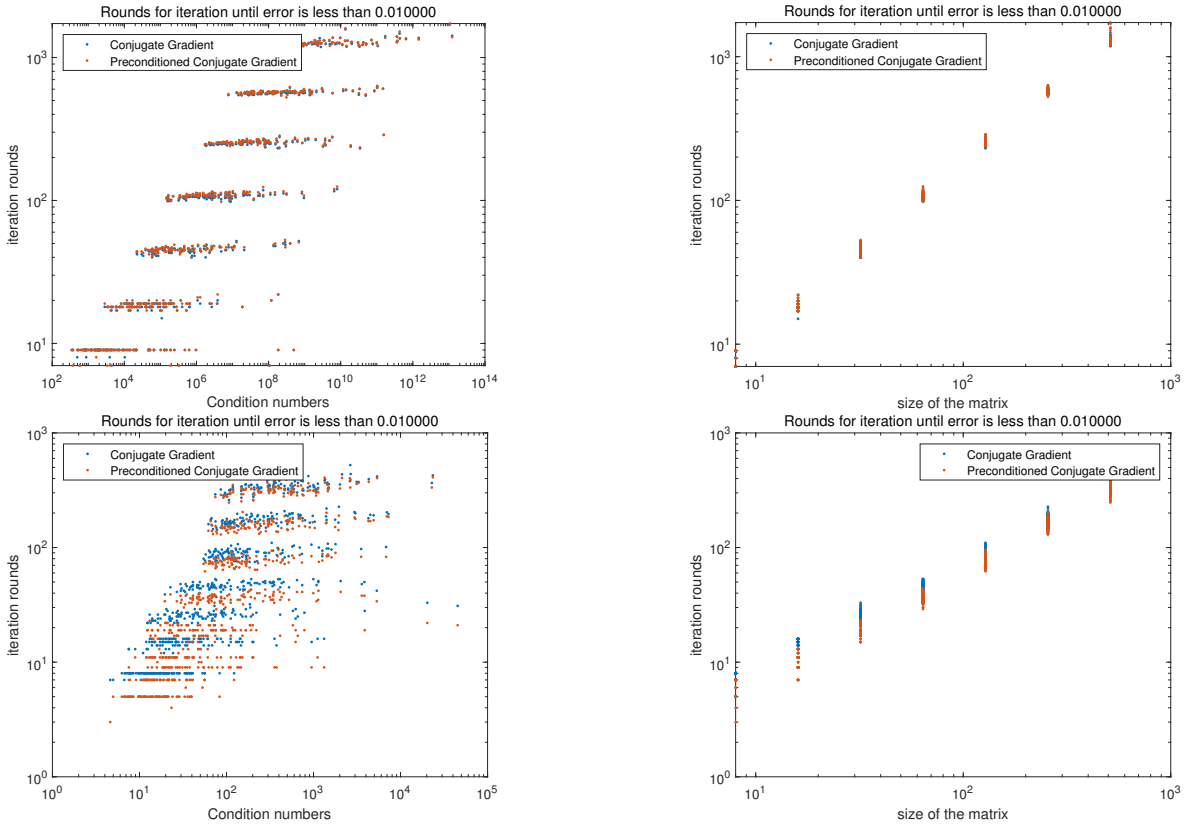


Figure 5: Graphs in the first row are about dense matrices, the second row describes the sparse matrices, the first column is about condition number vs iteration rounds, the second column is about the size of the matrices vs iteration rounds.

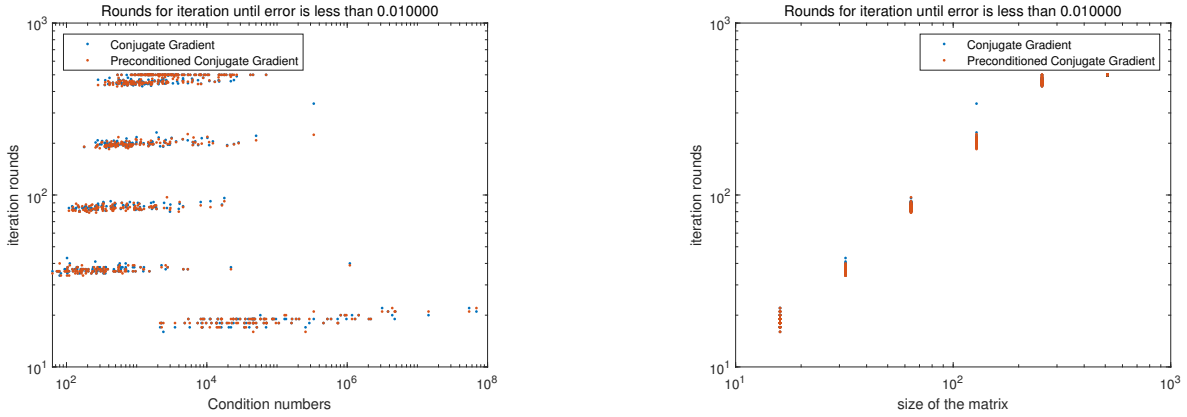


Figure 6: When $density = \frac{16}{i}$, the iterations

switch between those two so far.

2.5 The density for which PCG and CG takes the same amounts of iteration

Following from the discussion in Section 2.4, PCG (Jacobi) and CG have different iteration rounds for the sparse and the dense matrices. For sparse matrices, PCG takes fewer iterations. For dense matrices, CG takes fewer iterations. We are curious about for which density the CG and PCG will have similar amount of iterations. We choose density to be $32/i, 16/i, 8/i, 4/i$ while i denotes the size of the matrix, we find that when the density equals to $16/i$, the two methods take almost the same amounts of iterations as shown in Figure 6.

3 Comparison between PLU and Gradient Methods

We have introduced some Gradient Methods for solving the linear systems. In this section, we will compare the complexity of the algorithm with traditional linear solver - PLU. The PLU factorization takes a square matrix as input and an upper triangular matrix and a lower triangular matrix as output, this process takes about $\frac{2}{3}n^3$ flops. Adding the solving process about $2n^2$ flops, the total flops is about $\frac{2}{3}n^3 + 2n^2$.

We use the flop tool (https://github.com/Eleven7825/flopTool) to obtain the total flops for CG on matrices generated with method described in Section 2.3 with different size n. After obtain their flops, we do the linear regression in log-log scale on those (n,flops) pairs. For the dense matrices, the relation is about `4.5 * n^3.1` while the sparse situation described in Section 2.4 is around `4.8 * n^2.9` (setting `repeat = 12, strt = 3, stop = 8`, using Equation (6) to calculate r_k). Thus, CG is roughly a $\mathcal{O}(n^3)$ algorithm.

Data motion is another factor for the performance of an algorithm [2]. The advantage for CG is that it never forms or stores the matrix A [1], the data motion for CG is lower than that of PLU. This can explain why CG is faster than PLU. Thus, we are going to use CG and PCG as our linear solver in the next step for the matrices with relatively larger sizes.

4 Conclusion

In this report, we describe the Steepest Descent, Conjugate Gradient Method, and Preconditioned Conjugate Gradient Method. We summarize Shewchuck's introduction on gradient method, and made our description short yet self-contained. After that, we describe the preconditioning and its application on CG.

To compare those Gradient Methods, we compare the flops at each iteration at first. We find that the leading term for CG, SD, and PCG are the same, so we decide to compare those three methods based on numbers of iteration. For SD, we find that the iterations may exceed the theoretical boundary because of the roundoff error. We conclude that CG is a better linear solver than SD based on the number of iteration. For CG and PCG, we conclude that Jacobi preconditioned version of PCG is better than CG for sparse matrices while CG is better for dense matrices. We also explore the density for which PCG and CG have similar amount of iterations.

For the comparison between Gradient Methods and PLU, although PLU takes fewer flops than CG, CG is more data motion friendly than PLU, PLU can help us solve the smaller linear systems.

Based on the discussion above, we write a linear solver in Matlab code

and Figure 7 is the flowchart of the linear solver program.

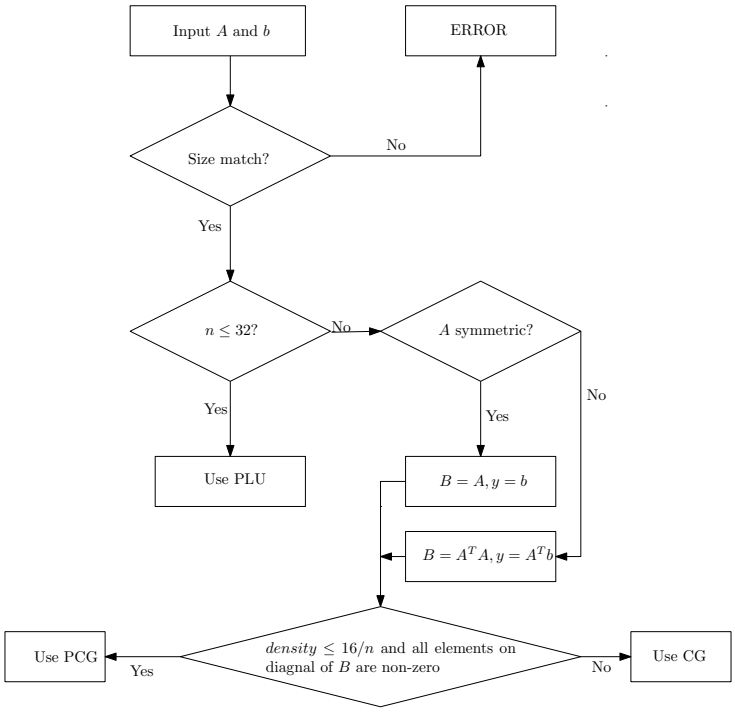


Figure 7: The flowchart of the linear solver program

References

[1] Stephen Boyd and J Duchi. Ee364b: Convex optimization ii. *Course Notes, http://www. stanford. edu/class/ee364b*, 2012.

[2] Gene H. Golub. *Matrix Computations*. 1983.

[3] Wen Shen. *Introduction To Numerical Computation, An*. World Scientific, 2019.

[4] Jonathan Richard Shewchuk et al. An introduction to the conjugate gradient method without the agonizing pain, 1994.

[5] Ya-xiang Yuan. A new stepsize for the steepest descent method. *Journal of Computational Mathematics*, pages 149–156, 2006.