EE5121 Convex Optimization - Quiz 2

Anirudh R (EE18B103), Sreekar Sai R (EE18B154), Chandan Bhat(EE16D209)

September 29, 2021

DECLARATION

We, Anirudh R, Sreekar Sai R., and Chandan Bhat declare that the submitted work is entirely our original work.

Contributions:

- Anirudh coded up the basic conjugate gradient, PD matrix generation and generated all the plots for q1.
- Sreekar developed the derivation for preconditioned CG, and modified the code accordingly.
- Chandan coded up nonlinear CG and created all the visualisations.
  All edited and reviewed this report.

In this report, we analyse the conjugate gradient method and its variants. We introduce the problem in section 1, and then describe the conjugate gradient method and update steps in brief. In section 2.2, we describe a modification to CG incorporating preconditioning, and then compare the performance of vanilla and preconditioned conjugate gradient. We also implement a non-linear conjugate gradient method (Fletcher-Reeves and Polak-Ribiere) in section 3 and note some of our interesting observations.

1 Solving a linear system

We have a $n \times n$ symmetric positive definite matrix $A \in \mathbb{R}^{n \times n}$ and $b \in \mathbb{R}^n$. We seek to find the solution to the linear system of equations represented by

$$Ax = b \tag{1}$$

Performing Gaussian elimination is a straight-forward method to obtain the solution but its time complexity is $\mathcal{O}(n^3)$. Finding the inverse will also solve the linear system but the complexity of Gauss-Jordan method of inverse calculation is again $\mathcal{O}(n^3)$. Even other more efficient inverse calculation algorithms have time complexities **higher**[1] than $\mathcal{O}(n^2)$. In addition, when the size of the problem is large, we require large storage for the matrix A, which can become impossible in many practical scenarios.

Here, we pose this linear system of equations problem as an optimization problem by constructing the following loss function.

$$\phi(x) = \frac{x^T A x}{2} - b^T x \tag{2}$$

Here, we seek to find the global minima of this strictly convex loss function(as A(the hessian) is positive definite, $\phi(x)$ is strictly convex). At the minima, we know that the gradient will reach zero.

$$\nabla\phi(x) = Ax - b = 0 \tag{3}$$

Therefore, by finding the minima of $\phi(x)$, we will find x that satisfies $Ax = b$ which is what we require.

2 Conjugate Gradient Method

2.1 Conjugacy

The conjugate gradient method makes use of a conjugate vector of A as the direction of movement in each iteration. The conjugate vectors of A satisfy the following properties. If $\{p_0, p_1, ..., p_{n-1}\}$ are conjugate vectors of A, then,

$$p_i^T A p_j = 0 \quad \forall i \neq j \tag{4}$$

$$p_i^T A p_i \neq 0 \quad \forall i \tag{5}$$

One important property of conjugate vectors is that they are linearly independent. There are multiple ways, but inefficient of generating the conjugate vectors.

- Eigen Decomposition of A: The eigen vectors of A are valid conjugate vectors of A. But, the time complexity of Eigen Value Decomposition is $\mathcal{O}(n^3)$. So, instead of doing this, we could just directly do Gaussian Elimination.
- Modified Gram-Schmidt Ortho-normalization: This is another method to generate conjugate vectors but is also $\mathcal{O}(n^3)$ in time complexity.

We use the following method of generating conjugate vectors that is pivotal to the efficiency of CG method. Residue $r_i = Ax_i - b$, Initial value: $p_0 = -r_0$.

$$p_k = -r_k + \beta_k p_{k-1} \quad \text{where } \beta_{k+1} = \frac{r_{k+1}^T r_{k+1}}{r_k^T r_k} \ \& \ r_{k+1} = r_k + \alpha_k A p_k \tag{6}$$

The overall update equation for x will be,

$$x_{k+1} = x_k + \alpha_k p_k \qquad \text{where } \alpha_k = \frac{r_k^T r_k}{p_k^T A p_k} \tag{7}$$

2.2 Preconditioned CG

We know that the convergence of the conjugate gradient method, follows the following inequality

$$\|x_{k+1} - x^*\| \leq \frac{\lambda_{n-k} - \lambda_1}{(\lambda_{n-k} + \lambda_1)} \|x_o - x^*\| \tag{8}$$

where $\lambda_n \geq ... \geq \lambda_1$ are the eigenvalues of the matrix A. Notice that the lower bound quickly becomes close to zero if the larger eigenvalues are closer to λ_1 (i.e., if it has a lower condition number).

The key to accelerate CG is to modify the eigenvalue distribution of A. Instead of solving (1), we'd instead like to solve

$$PAx = Pb \tag{9}$$

where P is a full rank matrix. We hope to choose a matrix P such that the condition number of PA is lesser than that of A (i.e., it has a better eigenvalue distribution).

$$L^T Ax = L^T b \tag{10}$$
$$(L^T AL)(L^{-1}x) = L^T b, \quad \text{or} \tag{11}$$
$$\hat{A}\hat{x} = \hat{b} \tag{12}$$

Ideally, we'd like to have our system to have all eigenvalues equal to each other, or simply the matrix $L^T AL$ to be equal to, or close to identity. One way of achieving such a form is by decomposing A using the cholesky decomposition:

$$A = CC^T \tag{13}$$

and using $L = (C^{-1})^T$, will lead to $\hat{A} = I$. However, the computational complexity of algorithms to compute the cholesky factorisation is $\mathcal{O}(n^3)$ in general. Instead, a much more cheaper alternative is to compute the incomplete cholesky decomposition, which is a *sparse approximation* of the cholesky factorization.

$$A \approx \tilde{C}\tilde{C}^T \tag{14}$$

Choosing $L = (\tilde{C}^{-1})^T$, we have a 'good enough' matrix L, such that the new system matrix $L^T AL$ is *close* to identity.

The naive approach to solving (12) is to simply solve for $\hat{x}$ using this new system, and then compute $x = L\hat{x}$. Instead we can modify the update equations (section 2) to be in terms of our original variables x.

In our new system, the update directions $\hat{p_k}$ must be conjugate wrt $\hat{A}$. In other words, they must satisfy,

$$\hat{p_i}^T (L^T AL)\hat{p_j} = 0 \quad \forall i \neq j \tag{15}$$

Defining $\hat{p_k} = L^{-1}p_k$ (p_k are conjugate with respect to A), we have

$$\hat{p_i}^T (L^T AL)\hat{p_j} = p_i^T Ap_j = 0 \quad \forall i \neq j \tag{16}$$

therefore ensuring conjugacy of $\hat{p_k}$ with respect to $\hat{A}$.

The new residue $\hat{r_k}$ can be written as,

$$\hat{r_k} = \hat{A}\hat{x} - \hat{b} = L^T(Ax - b) = L^T r_k \tag{17}$$

Using the update equation in section 2 and equation 17, we have

$$\hat{\alpha_k} = \frac{\hat{r_k}^T \hat{r_k}}{\hat{p_k}^T \hat{A}\hat{p_k}} \tag{18}$$
$$= \frac{r_k^T LL^T r_k}{p_k^T Ap_k} \tag{19}$$

For convenience, and to avoid repetitive calculations, we will define

$$y_k = M^{-1}r_k \tag{20}$$

where $M^{-1} = LL^T$, thus simplifying the expression for $\hat{\alpha_k}$. Similarly,

$$\hat{\beta}_{k+1} = \frac{\hat{r_{k+1}}^T \hat{r_{k+1}}}{\hat{r_k}^T \hat{r_k}} = \frac{r_{k+1}^T L L^T r_{k+1}}{r_k^T L L^T r_k} \tag{21}$$

$$= \frac{r_{k+1}^T y_{k+1}}{r_k^T y_k} \tag{22}$$

Now, using these, the update equations for $\hat{x_k}$ and $\hat{p_k}$ can be simplified to their corresponding equations in the original variables :

$$\hat{p_k} = -\hat{r}_k + \hat{\beta}_k \hat{p}_{k-1} \tag{23}$$

$$L^{-1} p_k = -L^T r_k + \hat{\beta}_k L^{-1} p_{k-1}, \quad \text{or} \tag{24}$$

$$p_k = -y_k + \hat{\beta}_k p_{k-1} \tag{25}$$

and

$$\hat{x}_{k+1} = \hat{x}_k + \hat{\alpha}_k \hat{p}_k \tag{26}$$

$$L^{-1} x_{k+1} = L^{-1} x_k + \hat{\alpha}_k L^{-1} p_k, \quad \text{or} \tag{27}$$

$$x_{k+1} = x_k + \hat{\alpha}_k p_k \tag{28}$$

Therefore, the update equations remain the same, except for α_k and β_k being replaced by $\hat{\alpha}_k$ and $\hat{\beta}_k$ respectively, and an additional variable y_k. The final update equations are summarized in Algorithm 1

Algorithm 1: Preconditioned CG

while $\|r\| > \eta$ **do**
- $\hat{\beta}_{k+1} = \frac{r_{k+1}^T y_{k+1}}{r_k^T y_k}$
- $y_k = M^{-1} r_k$
- $p_k = -y_k + \hat{\beta}_k p_{k-1}$, (next conjugate direction)
- $\hat{\alpha_k} = \frac{r_k^T y_k}{p_k^T A p_k}$
- $x_{k+1} = x_k + \hat{\alpha_k} p_k$ (move)
- $k \leftarrow k + 1$

end while

with the initial direction $p_0 = -y_0$. It is easy to see that M^{-1} is precomputed (before the start of iterations) to be the inverse of $M = \tilde{C}\tilde{C}^T$ from the incomplete cholesky decomposition.

The additional overhead we have here is storing an extra variable y_k at each step, and an upfront cost of computing M^{-1}.

2.3 Vanilla CG (vs) Preconditioned CG

In order to verify if our implementations are correct, we verified that the solutions provided are the same as the direct linear system solution provided by MATLAB `mldivide` ($A\backslash b$) for simple 2×2 and 3×3 systems of equations. For the case of $n = 60$, both methods converge to the optimum x^* solution calculated using `mldivide` as shown in Figure 2.

- **Eigen Values & Condition number**: We see (in Figure 1) that the eigenvalue spread of the preconditioned matrix, $\hat{A}$ is much better with lower condition number(8.13 compared to 78.31 of A). The lower condition number and better eigenvalue spread of the preconditioned matrix, $\hat{A}$ has led to convergence of the algorithm in far fewer iterations.(see Figure 2) The reduction in number of iterations is about **3-fold**. Since we deal with a better conditioned problem, the solution will also be more robust to noise and floating point errors in the solution.

- **Time taken**: Even though the number of iterations is lesser for the preconditioned CG, the amount of time taken by preconditioned CG(about 1.69 ms in one of the runs) is about the same or more compared to vanilla CG(about 1.59 ms in one of the runs). It is important to note that the time taken varies quite a bit during each run with the exact same conditions. So, this observation must be taken with a pinch of salt. So, preconditioned CG provides faster convergence in terms of number of iterations but is about the same or worse in terms of time taken.

- **Quality of Solution**: The quality of solution obtained by both methods at convergence is approximately the same. It is also evident from Figure 2 that the norm of error in the solution plateaus when introducing noise in b. (i.e., it converges to a different, 'wrong' minima)

- **Rate of Convergence**: The rate of convergence/rate of decrease in error for both methods **remained constant** throughout the execution of the respective algorithms in the log scale. But the decrease in norm of error per iteration was higher for the preconditioned CG method which led to **convergence of similar quality** in fewer iterations. Also, as the error drops linearly in the log scale, the loss function(in linear scale) drops drastically in the first few iterations and then stabilizes close to the solution.

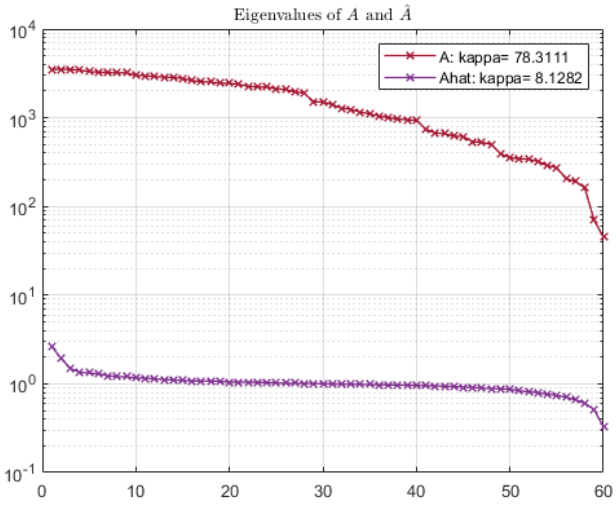


Figure 1: Eigenvalue spread of the original matrix, and the matrix corresponding to the preconditioned system $L^T AL$

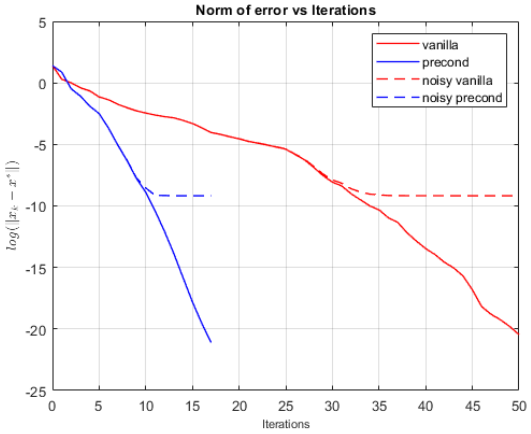


Figure 2: Evolution of the Error norm $\|x_k - x^*\|$

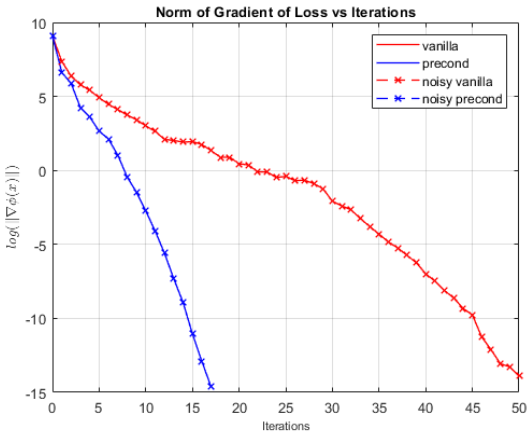


Figure 3: Norm of the gradient (=residue) r_k

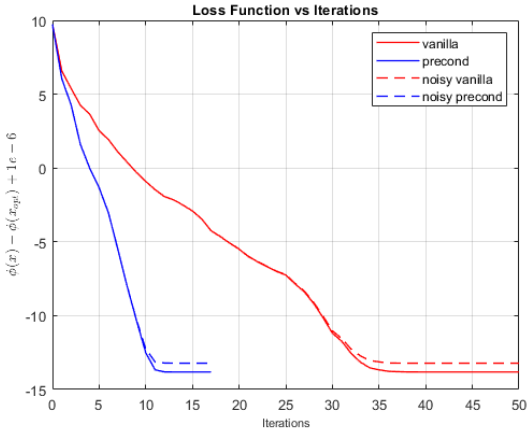


Figure 4: Objective Function values

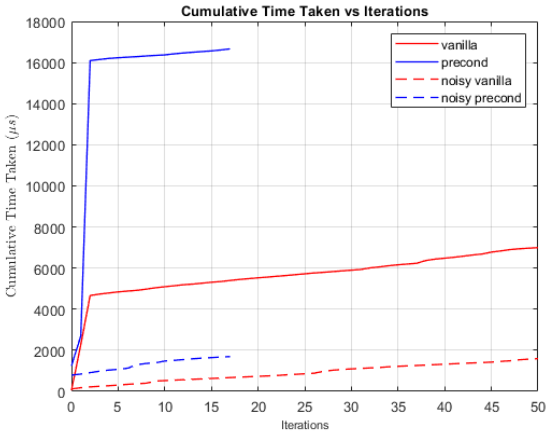


Figure 5: Cumulative time taken vs Iterations

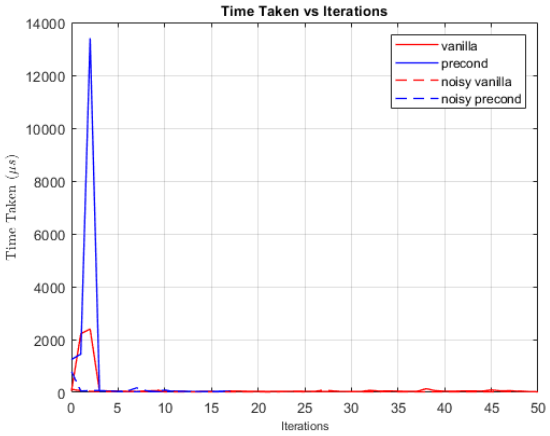


Figure 6: Time taken in each iteration

3 Nonlinear CG

Convex functions give a guarantee of achieving global minima, and preconditioning (section 2.2) efficiently solutions. Similar variants of the CG can be applied for minimizing general non linear functions. However, minimizing non - linear functions comes with the challenges of its own. To start with, lets consider a non-linear function $f(x_1, x_2)$ given by :

$$f(x_1, x_2) = -a \exp\left(-b\sqrt{\frac{1}{2}(x_1^2 + x_2^2)}\right) - \exp\left(\frac{1}{2}(\cos(cx_1) + \cos(cx_2))\right) + a + \exp(1) \quad (29)$$

The function $f(x_1, x_2)$ is known as *Ackley function* whose behaviour is mostly flat outer region with a global minima at the center. Fig. 7 shows the distribution of f in the region $x_1 \in [-2, 2]$ and $x_2 \in [-2, 2]$.

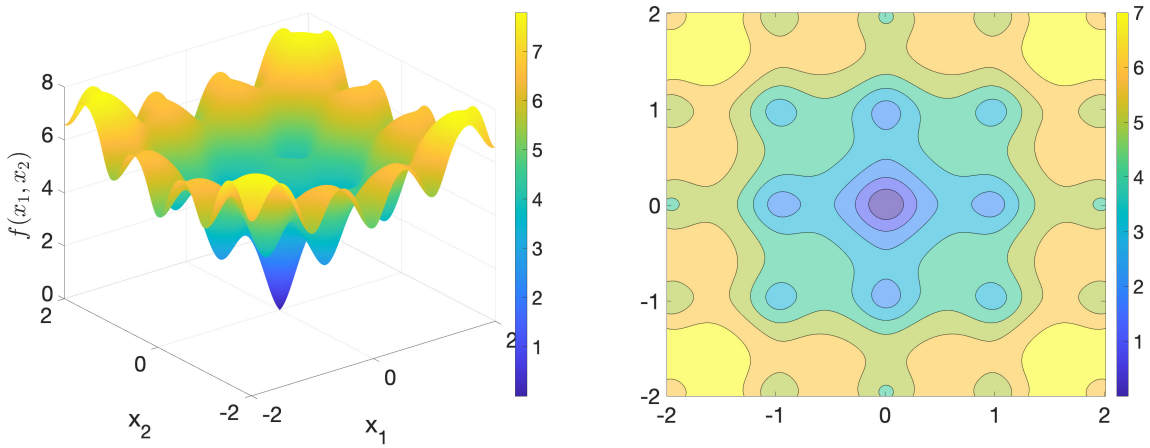


Figure 7: (a) Surface plot of Ackley function (b) Contour plots indicating local minima

Deviation from nice convex functions - Earlier we represented the function to be minimized using matrix relation and enforced the matrix to be positive definite, ensuring strict convexity. The gradients, step length α was calculated using the relation Eq. (3,19) which involved the linear operator A and the residue was just the gradient of ϕ. For non linear optimization we use similar steps but replace the gradient of ϕ with the gradient of non-linear function $f(x_1, x_2)$.

While there are various methods to solve the non-linear optimization, we implement two such algorithms:

1. Fletcher-Reeves method
2. Polak-Ribiere method

The non-linear methods are similar, in the sense that we determine a suitable (legitimate) search direction p_k and *then* the step length (α_k). One major difference is that we cannot clearly define any *conjugacy* wrt the nonlinear system, but we update the search direction with an expression identical to the previous case (which is different for each of these methods). The general flow of both the algorithms is as follows :

1. Set x_0, the initial starting point and evaluate $f(x_0)$ and compute $\nabla f(x_0)$. Set descent direction $p_0 = -\nabla f(x_0)$
2. Compute α_k using line search (using Wolfe conditions)
3. Compute $x_{k+1} = x_k + \alpha_k p_k$
4. Compute β_{k+1} (different for each method)
5. Find descent direction $p_{k+1} = \nabla f_{k+1} + \beta_{k+1} p_k$
6. $k = k + 1$

Repeat step 2-6 until minima is reached. The ideal stopping condition would be $\nabla f_k = 0$, but is impractical. Instead, the stopping condition is set by thresholding the gradient, upper bounded by the number of iterations.

The major implementation difference in Fletcher-Reeves (FR) and Polak-Ribiere (PR) method is in choice of β_{k+1}.

$$\text{FR method :} \quad \beta_{k+1} = \frac{\nabla f_{k+1}^T \nabla f_{k+1}}{\nabla f_k^T \nabla f_k} \tag{30}$$

$$\text{PR method :} \quad \beta_{k+1} = \frac{\nabla f_{k+1}^T (\nabla f_{k+1} - \nabla f_k)}{\|\nabla f_k\|^2} \tag{31}$$

Line Search using strong Wolfe condition - It is important to get α_k correct, such that the *next* iteration has a legitimate descent direction. To illustrate this lets us consider the inner product with the update equation for descent direction at k^{th} step

$$\nabla f_k^T p_k = \|\nabla f_k\|^2 + \beta_k \nabla f_k^T p_{k-1} \tag{32}$$

For the inexact line search there is a possibility to get $\nabla f_k^T p_k > 0$, when the second term in the RHS of Eq. (32) dominates. This implies p_k is not the descent direction. This situation is avoided by considering the step length α_k that satisfies strong Wolfe conditions.[2]

$$f(x_k + \alpha_k p_k) \leq -f(x_k) + c_1 \alpha_k \nabla f_k^T p_k \tag{33}$$

$$|\nabla f(x_k + \alpha_k p_k)^T p_k| \leq -c_2 \nabla f_k^T p_k \tag{34}$$

where $0 < c_1 < c_2 < \frac{1}{2}$.

3.1 Implementation

The gradient of the Ackley function Eq. (29) is given by

$$\nabla f = \begin{bmatrix} c \exp(\cos cx_1/2 + \cos cx_2/2) \sin cx_1/2 + (abx_1 \exp(-b(x_1^2/2 + x_2^2/2)^{1/2}))/(2(x_1^2/2 + x_2^2/2)^{1/2}) \\ c \exp(\cos cx_1/2 + \cos cx_2/2) \sin cx_2/2 + (abx_2 \exp(-b(x_1^2/2 + x_2^2/2)^{1/2}))/(2(x_1^2/2 + x_2^2/2)^{1/2}) \end{bmatrix}$$

For better visualization we consider a region which has the global minima and few local minimas. We consider various different starting points, vary the parameters of sufficient decrease and curvature condition. For Fletcher Reeves to perform correctly the descent direction should always be kept legitimate, hence we use Wolfe conditions for line search. The different starting points (SP) considered are as follows

1. Start near to global minima - $P_1 = [0.03 \;\; 0.31]^T$
2. Start with the farther point to global minima - $P_2 = [0.57 \;\; 0.58]^T$
3. Start near to local minima $P_3 = [1 \;\; 0.2]^T$
4. Start farther to local minima $P_4 = [1.2 \;\; 0.27]^T$

The locations are chosen to generate all possible scenarios. All the figures corresponds to Fletcher-Reeves method with parameters $c_1 = 1e^{-4}$, $c_2 = 0.1$.

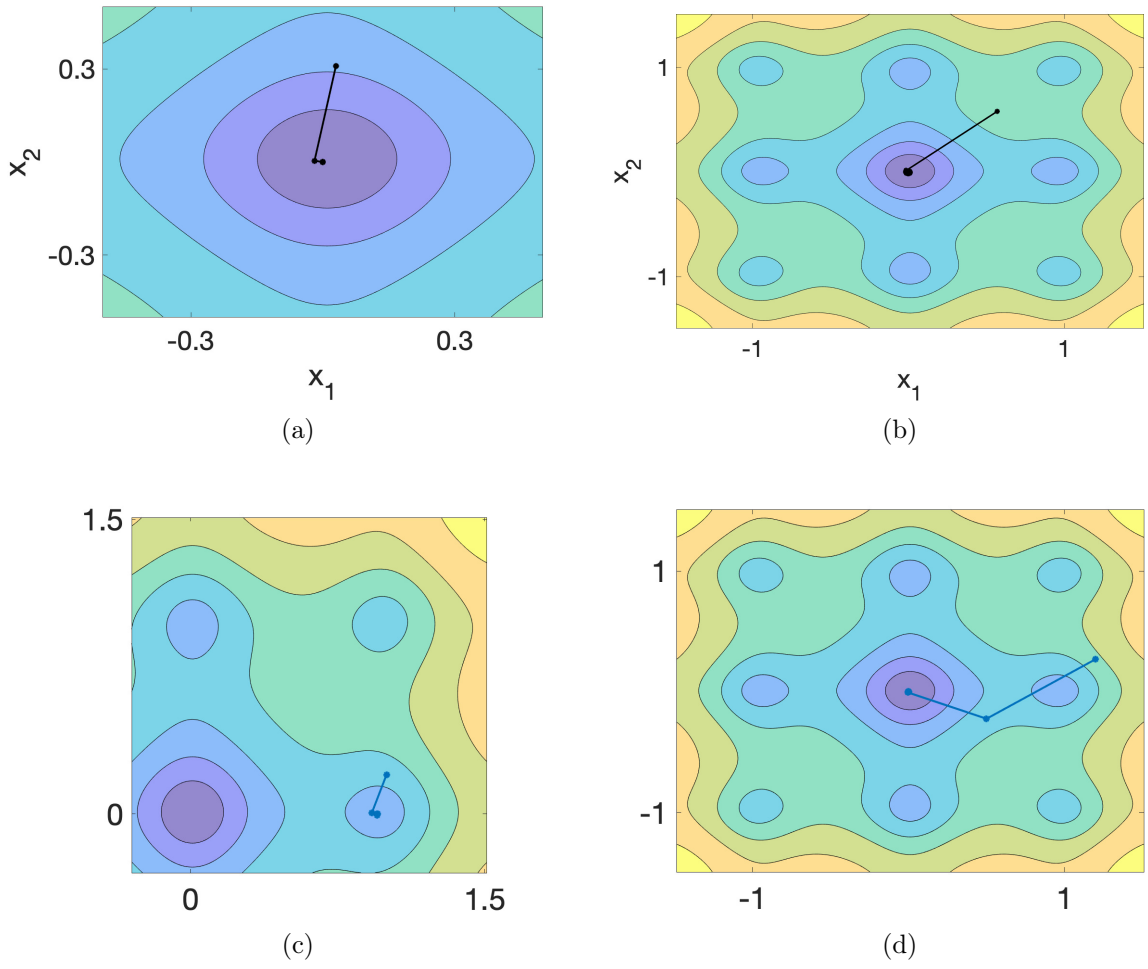


Figure 8: (a) $P_1 - [0.03 \;\; 0.31]^T$, SP is very close to the global minima, reaches the minimum easily (b) $P_2 - [0.57 \;\; 0.58]^T$ SP is far from global minima, but it is equally far from other local minima, the initial direction helps to fall to global minima (c) $P_3 - [1 \;\; 0.2]^T$, SP is very close to local minima and the algorithm stops at the local minima and (d) $P_4 = [1.2 \;\; 0.27]^T$, SP is near the same local minima but still the algorithm manages to reach the global minimum

Fig. 9 visualizes how the local minima is skipped. Similar results as seen in Fig. 8 is observed for PR method.

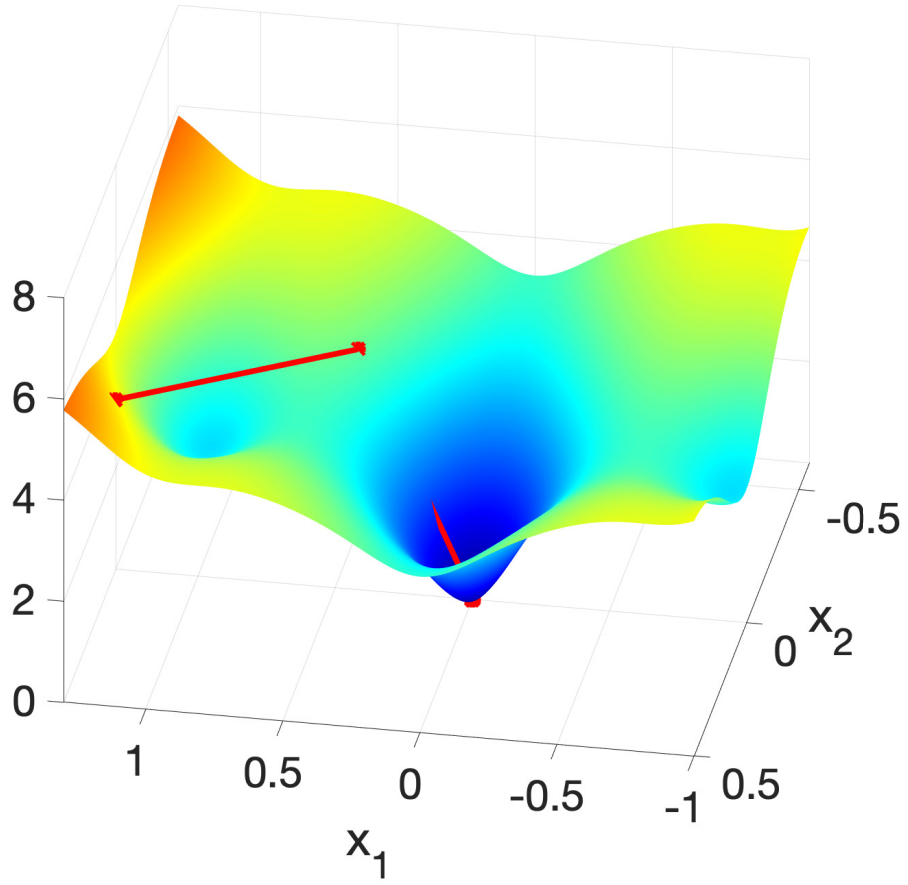


Figure 9: Surface plot showing the progression of x_k

3.2 Analysis

Let us start with observing the update equations x_{k+1} and p_{k+1}. We have

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

$$p_{k+1} = -\nabla f_k + \beta_k p_k$$

The above equation implies

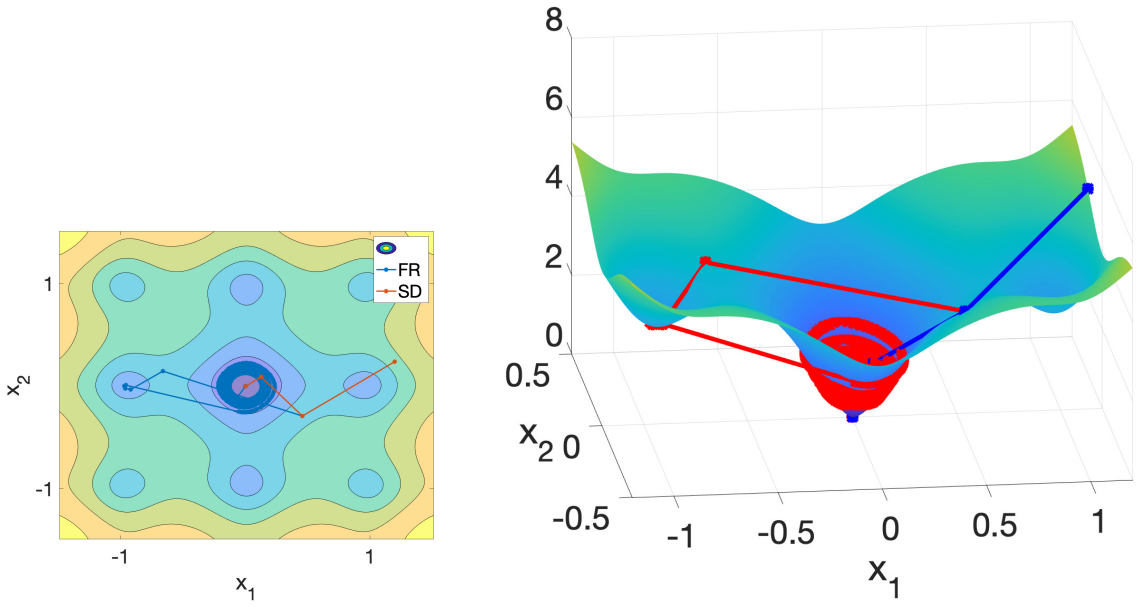
$$p_{k-1} = \frac{x_k - x_{k-1}}{\alpha_{k-1}} \tag{35}$$

Substituting above two equations in x_{k+1} we get

$$x_{k+1} = x_k - \alpha_k \nabla f_k + \frac{\alpha_k \beta_{k-1}}{\alpha_{k-1}}(x_k - x_{k-1}) \tag{36}$$

Eq. 36 is similar to steepest descent update equation but with an additional momentum term. FR and PR methods are momentum based steepest descent methods.

Momentum problem : A general question is does the momentum based methods always perform better compared to SD approach. To understand this let us go back to evaluating α_k which was dependent on c_1 and c_2. Lets change $c_1 - 1e^{-4}$ and $c_2 = 0.4$ and compare SD and PR for starting point $P - [1.2,\ 0.24]$.



(a) (b)

Figure 10: (a) Comparison of steepest descent and Fletcher-Reeves method. The sufficient and curvature parameters are changed. SD acheives minima with just 20 steps and FR takes 780 steps. For this run the threshold for minima is set at $1e^{-4}$, which means when function value reaches below the threshold the algorithm stops. (b) Due to the momentum created the Fletcher-Reeves method skips the global minima and reach a local minima and again jump to global minima from that point. The parameters c_1 and c_2 should be carefully chosen so that the momentum effect should not override the optimization problem.

Resetting the β_k : When the change in the α_k becomes very negligible than FR produces the descent direction which is approximately the same as in previous iteration. This makes the FR algorithm to get stuck and not improve any further. To overcome this problem the β_k is set to zero, which changes the descent direction to $-\nabla f_k$.

A MATLAB Code

A.1 Ordinary CG vs. Preconditioned CG

Listing 1: main script

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% EE5121 - Convex Optimization
% Conjugate Gradient Method
%
% Authors: Anirudh R, Sreekar Sai R, Chandan Bhat
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%Clearing and closing all figures
clear
close all;
%Clear screen
clc;
%Seeding the random number generator for repeatability
rng(5, 'twister');

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%Generating a symmetric positive definite matrix
mat = generatePDMatrix(5, 25);
%Extracting eigen values of B
eig_values = eig(mat);
%Checking if eigen values are all >0
if eig_values > 0
    disp(['Matrix generated is positive definite.']);
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%'size' determines the size of A and b
N = 60;
%Amplitude of noise added to original b
noise = 1e-2;
%Factor multiplied with b_orig
mult = 1;
%Maximum eigen value we are willing to allow
max_eig = N^2;
%Generating matrix A and column vector b for calculations
A = generatePDMatrix(N, max_eig);
b_orig = mult * rand(N,1);
b = b_orig + noise * rand(N,1);

```

```
%Optimal solution to the linear equation Ax = b_orig
x_opt = A \ b_orig;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%Parameters for running the iterative method
%Random initial point
x0 = rand(N, 1);
%Maximum number of iterations
max_iter = N;
%Tolerance for norm of gradient of loss at convergence
tolerance = 1e-6;

%Calling the conjugateGrad and preconditionedCG functions with pure b_orig
[x_hist, gf_hist, time_taken, num_iters] = conjugateGrad(A, b_orig, x0,
    max_iter, tolerance);
[x_hist_pre, gf_hist_pre, time_taken_pre, num_iters_pre, eig_values_pre,
    kappa] = preconditionedCG(A, b_orig, x0, max_iter, tolerance);

%Calling the functions with noisy b
[x_hist_noisy, gf_hist_noisy, time_taken_noisy, num_iters_noisy] =
    conjugateGrad(A, b, x0, max_iter, tolerance);
[x_hist_pre_noisy, gf_hist_pre_noisy, time_taken_pre_noisy,
    num_iters_pre_noisy, eig_values_pre_noisy, kappa_noisy] =
    preconditionedCG(A, b, x0, max_iter, tolerance);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%Plotting log of norm of gradient of loss function
figure;
% subplot(2,3,1);
plot(0:num_iters, log(gf_hist),'r','LineWidth',1,'DisplayName','vanilla');
hold on;
plot(0:num_iters_pre, log(gf_hist_pre),'b','LineWidth',1,'DisplayName','
    precond');
hold on;
plot(0:num_iters_noisy, log(gf_hist_noisy),'r--x','LineWidth',1,'
    DisplayName','noisy vanilla');
hold on;
plot(0:num_iters_pre_noisy, log(gf_hist_pre_noisy),'b--x','LineWidth',1,'
    DisplayName','noisy precond');

grid;
title('Norm of Gradient of Loss vs Iterations', 'FontSize', 10);
ylabel('$log( \| \nabla \phi (x) \| )$','interpreter','latex', 'FontSize',
     10);
xlabel('Iterations', 'FontSize', 8);
legend();
```

```
print('grad_norm','-dpdf');

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%Plotting eigen values of A
figure;
% subplot(2,3,2);
semilogy(sort(eig(A),'descend'),'Marker', 'x','LineStyle', '-', 'Color',
    [0.6350 0.0780 0.1840],'LineWidth',1,'DisplayName', ['A: kappa= '
    num2str(cond(A))]);
hold on;
semilogy(sort(eig_values_pre,'descend'),'Marker', 'x','LineStyle', '-', '
    Color', [0.4940 0.1840 0.5560],'LineWidth',1,'DisplayName', ['Ahat:
    kappa= ' num2str(kappa)]);
grid;
title('Eigenvalues of $A$ and $\hat{A}$', 'interpreter', 'latex', '
    FontSize', 10);
legend();
print('eig','-dpdf');

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%Plotting log of norm of error
figure;
% subplot(2,3,3);
plot(0:num_iters, log( vecnorm( x_hist- x_opt ) ),'r','LineWidth',1,'
    DisplayName', 'vanilla');
hold on;
plot(0:num_iters_pre, log( vecnorm( x_hist_pre- x_opt ) ),'b','LineWidth'
    ,1,'DisplayName', 'precond');
hold on;
plot(0:num_iters_noisy, log( vecnorm( x_hist_noisy- x_opt ) ),'r--','
    LineWidth',1,'DisplayName', 'noisy vanilla');
hold on;
plot(0:num_iters_pre_noisy, log( vecnorm( x_hist_pre_noisy- x_opt ) ),'b--
    ','LineWidth',1,'DisplayName', 'noisy precond');

grid;
title('Norm of error vs Iterations', 'FontSize', 10);
ylabel('$log(\|x_k-x^*\| )$','interpreter','latex', 'FontSize', 10);
xlabel('Iterations', 'FontSize', 8);
legend();
print('error_norm','-dpdf');

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%Plotting time taken
figure;
```

```
% subplot(2,3,4);
plot(0:num_iters, 10^6*time_taken,'r','LineWidth',1,'DisplayName','vanilla
    ');
hold on;
plot(0:num_iters_pre, 10^6*time_taken_pre,'b','LineWidth',1,'DisplayName',
    'precond');
hold on;
plot(0:num_iters_noisy, 10^6*time_taken_noisy,'r--','LineWidth',1,'
    DisplayName','noisy vanilla');
hold on;
plot(0:num_iters_pre_noisy, 10^6*time_taken_pre_noisy,'b--','LineWidth',1,
    'DisplayName','noisy precond');

grid;
title('Time Taken vs Iterations', 'FontSize', 10);
ylabel('Time Taken ($\mu s$)','interpreter','latex', 'FontSize', 10);
xlabel('Iterations', 'FontSize', 8);
legend();
print('time','-dpdf');

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%Plotting cumulative time taken vs iterations
%Calculating cumulative times taken
time_taken_cum = zeros(num_iters+1, 1);
time_taken_cum(1) = time_taken(1);
for i=2:num_iters+1
    time_taken_cum(i) = time_taken_cum(i-1) + time_taken(i);
end

time_taken_cum_pre = zeros(num_iters_pre+1, 1);
time_taken_cum_pre(1) = time_taken_pre(1);
for i=2:num_iters_pre+1
    time_taken_cum_pre(i) = time_taken_cum_pre(i-1) + time_taken_pre(i);
end

time_taken_cum_noisy = zeros(num_iters_noisy+1, 1);
time_taken_cum_noisy(1) = time_taken_noisy(1);
for i=2:num_iters_noisy+1
    time_taken_cum_noisy(i) = time_taken_cum_noisy(i-1) + time_taken_noisy
        (i);
end

time_taken_cum_pre_noisy = zeros(num_iters_pre_noisy+1, 1);
time_taken_cum_pre_noisy(1) = time_taken_pre_noisy(1);
for i=2:num_iters_pre_noisy+1
    time_taken_cum_pre_noisy(i) = time_taken_cum_pre_noisy(i-1) +
        time_taken_pre_noisy(i);
```

```
end

figure;
% subplot(2,3,5);
plot(0:num_iters, 10^6*time_taken_cum,'r','LineWidth',1,'DisplayName','
    vanilla');
hold on;
plot(0:num_iters_pre, 10^6*time_taken_cum_pre,'b','LineWidth',1,'
    DisplayName','precond');
hold on;
plot(0:num_iters_noisy, 10^6*time_taken_cum_noisy,'r--','LineWidth',1,'
    DisplayName','noisy vanilla');
hold on;
plot(0:num_iters_pre_noisy, 10^6*time_taken_cum_pre_noisy,'b--','LineWidth
    ',1,'DisplayName','noisy precond');

grid;
title('Cumulative Time Taken vs Iterations', 'FontSize', 10);
ylabel('Cumulative Time Taken ($\mu s$)','interpreter','latex', 'FontSize'
    , 10);
xlabel('Iterations', 'FontSize', 8);
legend();
print('time_cum','-dpdf');

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%Plotting the values taken by the loss function for each x in x_hist and
    x_hist_pre
%Defining the loss function
f = @(x) 0.5*x'*A*x - b_orig'*x;

%Calculating loss value at columns in x_hist, x_hist_pre, x_hist_noisy and
     x_hist_pre_noisy
func_vals = zeros(num_iters+1, 1);
for i=1:(num_iters+1)
    func_vals(i, 1) = f(x_hist(:, i));
end

func_vals_pre = zeros(num_iters_pre+1, 1);
for i=1:(num_iters_pre+1)
    func_vals_pre(i, 1) = f(x_hist_pre(:, i));
end

func_vals_noisy = zeros(num_iters_noisy+1, 1);
for i=1:(num_iters_noisy+1)
    func_vals_noisy(i, 1) = f(x_hist_noisy(:, i));
end

```

```
func_vals_pre_noisy = zeros(num_iters_pre_noisy+1, 1);
for i=1:(num_iters_pre_noisy+1)
    func_vals_pre_noisy(i, 1) = f(x_hist_pre_noisy(:, i));
end

%Plotting log of loss wrt loss at minima(Loss at x_opt is subtracted from loss at every point)
figure;
% subplot(2,3,6);
plot(0:num_iters, log(func_vals - f(x_opt)+1e-6),'r','LineWidth',1,'DisplayName','vanilla');
hold on;
plot(0:num_iters_pre, log(func_vals_pre - f(x_opt)+1e-6),'b','LineWidth',1,'DisplayName','precond');
hold on;
plot(0:num_iters_noisy, log(func_vals_noisy - f(x_opt)+1e-6),'r--','LineWidth',1,'DisplayName','noisy vanilla');
hold on;
plot(0:num_iters_pre_noisy, log(func_vals_pre_noisy - f(x_opt)+1e-6),'b--','LineWidth',1,'DisplayName','noisy precond');

grid;
title('Loss Function vs Iterations', 'FontSize', 10);
ylabel('$\phi (x) - \phi (x_{opt}) + 1e-6$', 'interpreter', 'latex', 'FontSize', 10);
xlabel('Iterations', 'FontSize', 8);
legend();
print('phi','-dpdf');

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Listing 2: Code to implemented Vanilla CG

```
function [x_hist, gf_hist, time_taken, k] = conjugateGrad(A, b, x0, max_iter, tolerance)
        %%%%%%%%%%%%%%
        % Function to run the Conjugate Gradient Method(CGM).
        % Argument:
        %       A                       Symmetric PD matrix describing the linear system
        %       b                       Target of the linear system
        %       x0                      Initial value of 'x' to start the iterative algorithm
        %       max_iter        Maximum number of iterations allowed
        %       tolerance       Tolerance for norm of gradient of loss function at convergence
        % Return:
        %       x_hist          History of values taken by x through the iterations
```

```
12          %   gf_hist     History of norm of gradient of loss function
                through the iterations
13          %       time_taken      Time taken for execution of each iteration
14          %       k                       Number of iterations that actually
                occurred
15          %%%%%%%%%%%%%%
16
17          %Starting measurement of time for initialization tasks
18          tic
19
20          %The gradient of loss function
21          gradf = @(x) A*x - b;
22
23          %x = x + alpha p, where alpha = r'*r/(p'*A*p)
24          x = x0;
25          r = gradf(x);
26          p = -r;
27          %Storing x and norm of gradient in the history
28          x_hist = [x];
29          gf_hist = [norm(r)];
30          k = 0;
31
32          %Logging the time taken before iterating
33          time_taken = toc;
34
35          %Displaying values before the iterations
36          fprintf('\n');
37          disp('Vanilla Conjugate Gradient Method:')
38          fprintf('\n');
39          disp(['CG: k=0, gf=' num2str(norm(r)) '   , time elapsed: ' num2str
                (time_taken*10^6) ' micro seconds']);
40
41          while norm(r) > tolerance && k < max_iter
42                  %Starting measurement of time for this iteration
43                  tic
44
45                  %Step size calculated according to CGM
46                  step_size = r'*r/(p'*A*p);
47                  %Update of x
48                  x_new = x + step_size * p;
49
50                  %x is updated. Updating parameters for next iteration
51                  r_new = r + step_size * A * p;
52                  beta_ = r_new'*r_new/(r'*r);
53                  p = -r_new + beta_ *p;
54                  r = r_new;
55                  x = x_new;
56
```

```
                %Storing the norm of grad and updated x
                gf_hist = [gf_hist, norm(r_new)];
                x_hist = [x_hist, x_new];
                %Updating iterate for next iteration
                k = k+1;

                %Ending measurement of time and logging
                time_taken = [time_taken, toc];

                disp(['CG: k=' num2str(k) ', gf=' num2str(norm(r_new)) '
                    , time elapsed: ' num2str(time_taken(end)*10^6) '
                    micro seconds']);
        end
        %Displaying relevant details of the CGM execution
        fprintf('\n');
        disp(['Total number of iterations: ' num2str(k) ' , Total time
            taken: ' num2str(sum(time_taken)*10^6) ' micro seconds']);
        fprintf('\n');
end
```

Listing 3: Function to generate random positive definite matrix

```
function [matrix] = generatePDMatrix(n, max_eig)
        %%%%%%%%%%%%%%%
        % Function to generate a nxn positive definite matrix.
        % Argument:
        %       n                       Size of the square matrix
        % Return:
        %       matrix          nxn positive definite matrix
        %%%%%%%%%%%%%%%

        %Generating a random nxn matrix
        temp = rand(n, n);
        %Making a nxn symmetric matrix from random nxn matrix
        temp = 0.5*(temp + temp');
        %Eigen decomposition of nxn symmetric matrix
        [V, D] = eig(temp);
        %Choosing positive eigen values to construct our positive definite
            matrix
        eig_vals = randi(max_eig, n, 1);
        %Constructing the positive definite matrix using eig_vals and V
            corresponding to matrix temp
        matrix = V*diag(eig_vals)*V';
end
```

Listing 4: Function to implemented Preconditioned CG

```
function [x_hist, gf_hist, time_taken, k, eig_values, kappa] =
    preconditionedCG(A, b, x0, max_iter, tolerance)
```

```
        %%%%%%%%%%%%%%
        % Function to run the Preconditioned Conjugate Gradient Method(CGM
            ).
        % Argument:
        %       A                       Symmetric PD matrix describing the
             linear system
        %       b                       Target of the linear system
        %       x0                      Initial value of 'x' to start the
            iterative algorithm
        %       max_iter        Maximum number of iterations allowed
        %       tolerance       Tolerance for norm of gradient of loss
            function at convergence
        % Return:
        %       x_hist          History of values taken by x through the
            iterations
        %   gf_hist     History of norm of gradient of loss function
            through the iterations
        %       time_taken      Time taken for execution of each iteration
        %       k                       Number of iterations that actually
             occurred
        %       eig_values      Eigenvalues of Ahat
        %       kappa           Condition number of Ahat
        %%%%%%%%%%%%%%

        %Starting measurement of time for initialization tasks
        tic

    %     Preconditioning
    op.type = 'ict';
    op.droptol = 1e-2;
    C = ichol(sparse(A),op);
    Minv = inv(C*C');

        %The gradient of loss function
        gradf = @(x) A*x - b;

        %x = x + alpha p, where alpha = r'*r/(p'*A*p)
        x = x0;
        r = gradf(x);
    y = Minv*r;
        p = -y;
        %Storing x and norm of gradient in the history
        x_hist = [x];
        gf_hist = [norm(r)];
        k = 0;

        %Logging the time taken before iterating
        time_taken = toc;
```

```
43
44          %Displaying values before the iterations
45          fprintf('\n');
46          disp('Preconditioned Conjugate Gradient Method:')
47          fprintf('\n');
48          disp(['PCG: k=0, gf=' num2str(norm(r)) '  , time elapsed: '
               num2str(time_taken*10^6) ' micro seconds']);
49
50          while norm(r) > tolerance && k < max_iter
51                  %Starting measurement of time for this iteration
52                  tic
53
54                  %Step size calculated according to CGM
55                  step_size = r'*y/(p'*A*p);
56                  %Update of x
57                  x_new = x + step_size * p;
58
59                  %x is updated. Updating parameters for next iteration
60                  r_new = r + step_size * A * p;
61          y_new = Minv*r_new;
62
63                  beta_ = r_new'*y_new/(r'*y);
64                  p = -y_new + beta_ *p;
65                  r = r_new;
66                  x = x_new;
67          y = y_new;
68
69                  %Storing the norm of grad and updated x
70                  gf_hist = [gf_hist, norm(r_new)];
71                  x_hist = [x_hist, x_new];
72                  %Updating iterate for next iteration
73                  k = k+1;
74
75                  %Ending measurement of time and logging
76                  time_taken = [time_taken, toc];
77
78                  disp(['PCG: k=' num2str(k) ', gf=' num2str(norm(r_new)) '
                         , time elapsed: ' num2str(time_taken(end)*10^6) '
                        micro seconds']);
79          end
80          %Displaying relevant details of the CGM execution
81          fprintf('\n');
82          disp(['Total number of iterations: ' num2str(k) ' , Total time
               taken: ' num2str(sum(time_taken)*10^6) ' micro seconds']);
83          fprintf('\n');
84
85      %Calculating the \hat{A} to study the eigenvalues and condition number
86          Cinv = inv(C);
```

```
        Ahat = Cinv*A*Cinv';
        eig_values = eig(Ahat);
        kappa = cond(Ahat);
end
```

A.2 Nonlinear CG

Listing 5: lineSearch.m

```
function alpha = lineSearch(f,gradf,gradphi_0,c1,c2,x,phi_x,pk,rho)

% This code performs Line search for finding optimal alpha (step length)
% using strong Wolfe conditions. The algorithm for line search method is
% used from  Numerical optimization  by Jorge Nocedal Stephen J. Wright
% Chapter 3, Algorithm 3.5.

% The parameters passed to the search algorithm are
% f       - objective function
% gradf   - gradient of the function
% c1      - sufficient decrease constant
% c2      - curvature condition constant
% x       - solution at current iterate
% pk      - search direction
% phi(alpha) = f(x + alpha p_k)
% phi_x   - function value at current x
% gradphi_0 - gradient of function at x


% Parameters used in the algorithm
% xi        - the updated x (x_prev +  alpha_i*pk)
% alpha_lo  - alpha whoich supplied to zoom function
% alpha_hi  - higher alpha


% Initialize the alpha by specifying the range it can take. alpha > 0
alpha_0  = 0; alpha_max = 200; alpha_1  = 1; % alpha_1 \in (0,alpha_max)
alpha_i = alpha_1; alpha_prev = alpha_0;
phi_prev  = phi_x; phi_0 = phi_x;
x_prev = x; i = 0;

while true && (abs(alpha_prev - alpha_i) > 1e-4)

    xi          = x_prev + alpha_i*pk;
    phi_xi      = f(xi);
    gradphi_xi  = gradf(xi)'*pk;
```

```
40
41      % alpha_1 violates the sufficient condition
42      if  (phi_xi  > phi_0 + c1*alpha_i*gradphi_0) || ((phi_xi >= phi_prev )
            && (i > 0))
43          alpha = zoom(f,gradf,gradphi_0,c1,c2,x,phi_0,phi_prev,pk,
               alpha_prev,alpha_i);
44          break;
45      end
46
47      % strong Wolfe check
48      if(abs(gradphi_xi) <= -c2*gradphi_0)
49          alpha = alpha_i;
50          break;
51      end
52
53      % Check if we moved ahead
54      if (gradphi_xi >= 0)
55          alpha = zoom(f,gradf,gradphi_0,c1,c2,x,phi_0,phi_prev,pk,
               alpha_prev,alpha_i);
56          break;
57      end
58
59      alpha_prev = alpha_i;
60      alpha_i    = min(rho*alpha_i,alpha_max);
61      phi_prev   = phi_xi;
62      i = i + 1;
63
64  end
65
66  end
```

Listing 6: NLCG1.m

```
1
2
3   function [x_k,val,f_hist,x_hist,k,alpha_hist] = NL_CG1(f,gradf,x0,pk,c1,c2
        ,rho,method)
4
5   % This code performs conjugate gradient on Non - linear functions.
6   % The update equation for beta differs by the method of choice
7   % (a) Fletcher Reeves (b) Polak Riebere
8
9   % Initialization
10  tol = 1e-4;
11  gradf_x0     = gradf(x0);
12  gradphi_prev = gradf_x0'*pk;
13  x_prev = x0; phi_xk = f(x0); phi_prev = phi_xk + 10;
14  x_k = x0; gradf_prev = 1;
15  gradf_xk = gradf_x0; k = 0;
```

```
x_hist = x_k; f_hist = f(x_k); alpha_hist = [];

while norm(gradf_xk) > tol && norm(phi_xk - phi_prev) > tol

    alpha_k    = lineSearch(f,gradf,gradphi_prev,c1,c2,x_k,phi_xk,pk,rho);
    x_k        = x_prev + alpha_k*pk;
    gradf_prev = gradf_xk;
    gradf_xk   = gradf(x_k);
    phi_xk     = f(x_k);
    phi_prev   = f(x_prev);
    if strcmp(method,'fletcher')
        beta_k     = (gradf_xk'*gradf_xk)/(gradf_prev'*gradf_prev);
    elseif strcmp(method,'polak')
        beta_k     = (gradf_xk'*(gradf_xk - gradf_prev))/(gradf_prev'*
           gradf_prev);
        if beta_k < 0
           beta_k = 0;
        end
    elseif strcmp(method,'SD')
        beta_k = 0;
    end
    alpha_hist = [alpha_hist,alpha_k];
    if alpha_k < 1e-5
        beta_k = 0;
    end
    pk         = -gradf_xk + beta_k*pk;
    k          =  k + 1;
    x_prev     = x_k;
    gradphi_prev = gradf_xk'*pk;
    x_hist = [x_hist,x_k];
    f_hist = [f_hist,f(x_k)];
end

    val = f(x_k);

end
```

Listing 7: TestNLCG.m

```
clear all; close all; clc

method = 'fletcher';
% Ackley Function, global minimum at (0,0)
syms x
a = 20; b = 0.2; c = 2*pi;
f1     = @(x) -a*exp(-b*sqrt(0.5*(x(1)^2 + x(2)^2))) -exp(0.5*(cos(c*x(1))
   +cos(c*x(2)))) + a + exp(1);
gradf1 = @(x)[(c*exp(cos(c*x(1))/2 + cos(c*x(2))/2)*sin(c*x(1)))/2 + (a*b*
   x(1)*exp(-b*(x(1)^2/2 + x(2)^2/2)^(1/2)))/(2*(x(1)^2/2 + x(2)^2/2)
```

```
    ^(1/2));
9    (c*exp(cos(c*x(1))/2 + cos(c*x(2))/2)*sin(c*x(2)))/2 + (a*b*x(2)*exp(-
        b*(x(1)^2/2 + x(2)^2/2)^(1/2)))/(2*(x(1)^2/2 + x(2)^2/2)^(1/2))];
10
11  % Parameters
12  c1 = 1e-4; c2 = 0.4; rho = 2;
13
14  % starting point for the non-linear CG
15  % Choose any one and comment the rest
16  x0 = [1.2;0.27];
17  %x0 = [-0.55;-0.03];
18  %x0 = [0.57;0.5];
19  %x0 = [0.57;0.58];
20  %x0 = [1;0.2];
21  x0 = [1.2;0.24];
22  %x0 = [0.03; 0.31]
23  % Test for a simple convex function, uncomment following lines
24  % f1 = @(x) 4*x(1)^2+x(2)^2;
25  % gradf1 =@(x) [8*x(1);2*x(2)];
26
27  pk = -gradf1(x0); % inital search direction
28  phi_xk = f1(x0); gradphi_prev = gradf1(x0)'*pk;
29  [~,~,f_hist,x_hist,k,alpha_hist] = NL_CG1(f1,gradf1,x0,pk,c1,c2,rho,method
    );
30
31
32  % Plotting
33  plotting = true;
34
35  if plotting
36      syms x1 y1
37      f2  = @(x1,y1) -a*exp(-b.*sqrt(0.5*(x1.^2 + y1.^2))) -exp(0.5.*(cos(c
        .*x1)+cos(c.*y1))) + a + exp(1);
38      sampling = 0.01;
39      xc  = -1.5 : sampling : 1.5;
40      [Xc,Yc] = meshgrid(xc);
41      f_out = f2(Xc,Yc);
42      figure
43      surf(Xc,Yc,f_out)
44      shading interp
45      hold on
46      plot3(x_hist(1,:),x_hist(2,:),f_hist(:),'*-r','MarkerSize',10,'
        LineWidth',4); hold on
47
48      figure
49      [~, contourObj] = contourf(f_out);
50      % This is the secret that 'keeps' the transparency.
51      eventFcn = @(srcObj, e) updateTransparency(srcObj);
```

```
52      addlistener(contourObj, 'MarkedClean', eventFcn);
53      hold on
54      plot(x_hist(1,:)*(1/sampling)+floor(length(xc)/2),x_hist(2,:)*(1/
            sampling)+floor(length(xc)/2),'*-','MarkerSize',5,'LineWidth',2);
            hold on
55
56  end
57
58  display = true;
59  if display
60
61  fprintf('
        ----------------------------------------------------------------------\n')
62
63  fprintf('Initial Point : [');
64  fprintf('%g, ', x0(1:end-1));
65  fprintf('%g]\n', x0(end));
66
67  fprintf('Minimum Point : [');
68  fprintf('%g, ', x_hist(1:end-1,end));
69  fprintf('%g]\n', x_hist(end,end));
70
71  fprintf('Minimum value : %g \n',f_hist(end));
72  fprintf('Method : %s \n',method);
73
74  fprintf('Steps taken %d \n',k);
75
76  fprintf('
        ----------------------------------------------------------------------\n')
77
78  end
79
80  comparison = true;
81  if comparison
82
83  method = 'SD';
84  [~,~,f_hist_sd,x_hist_sd,k_sd,alpha_hist_sd] = NL_CG1(f1,gradf1,x0,pk,c1,
        c2,rho,method);
85  method = 'fletcher';
86  [~,~,f_hist_fr,x_hist_fr,k_fr,alpha_hist_fr] = NL_CG1(f1,gradf1,x0,pk,c1,
        c2,rho,method);
87  method = 'polak';
88  [~,~,f_hist_pr,x_hist_pr,k_pr,alpha_hist_pr] = NL_CG1(f1,gradf1,x0,pk,c1,
        c2,rho,method);
89
90  fprintf('
        ----------------------------------------------------------------------\n')
91
```

```
fprintf('Minimum value : %g \n',f_hist_sd(end));
fprintf('Method : %s \n','SD');
fprintf('Minimum value : %g \n',f_hist_fr(end));
fprintf('Method : %s \n','fletcher');
fprintf('Minimum value : %g \n',f_hist_pr(end));
fprintf('Method : %s \n','polak');



fprintf('Steps taken SD %d \n',k_sd);
fprintf('Steps taken FR %d \n',k_fr);
fprintf('Steps taken PR %d \n',k_pr);

fprintf('
   ----------------------------------------------------------------------------\n')

end

plotting_cmp = true;
if plotting_cmp
   figure
   plot(f_hist_sd); hold on
   plot(f_hist_fr); hold on
   plot(f_hist_pr); hold on
end
```

Listing 8: zoom.m

```


function alpha = zoom(f,gradf,gradphi_0,c1,c2,x,phi_0,phi_prev,pk,alpha_lo
   ,alpha_hi)

% Zoom function - each iteration generates alpha between alpha_lo and
% alpha_hi, and then replaces one of these endpoints by alpha, in such a
% way that Wolfe condition is satisfied in the interval!

while true

    % Choose an alpha in (alpha_lo, alpha_hi). There are many ways to
    % select the alpha (a) interpolation or (b) just select a midpoint or
       (c)

    alpha  = (alpha_lo+alpha_hi)/2;
    xi     = x + alpha*pk;
    phi_xi = f(xi);
    gradphi_xi = gradf(xi)'*pk;


```

```
    if phi_xi > phi_0 + c1*alpha*gradphi_0 || phi_xi >= phi_prev
        alpha_hi = alpha;
    else
        gradphi_xi = gradf(xi)'*pk;
        % Check strong Wolfe condition
        if abs(gradphi_xi) <= -c2*gradphi_0
            return;
        end

        if gradphi_xi*(alpha_hi-alpha_lo) >= 0
            alpha_hi = alpha_lo;
        end
        alpha_lo = alpha;
        phi_prev = phi_xi;

    end

    if alpha < 1e-5
        return
    end


    end

end
```

References

[1] https://en.wikipedia.org/wiki/Computational_complexity_of_mathematical_operations

[2] Numerical Optimization by Jorge Nocedal and Stephen J. Wright, Springer, 2006.