Illinois Institute of Technology

ILLINOIS TECH

Armour College of Engineering

Project 1

Student :
Carlos Martin

Course :
ECE 505 Applied Optimization for Engineers

November 5, 2024

Contents

1 Part 1: Algorithmic Implementation

1.1 Steepest Descent Algorithm

The steepest descent algorithm starts with an initial estimate for the minimum design, then computes the direction of steepest descent at that point.

```
def steepest_descent(f, x0, epsilon=1e-10, max_iter=10000):
    x = np.array(x0, dtype=float)
    for i in range(max_iter):
        grad = gradient(f, x)
        grad_norm = np.linalg.norm(grad)
        if grad_norm < epsilon:
            break
        p = -grad / grad_norm  # Normalized descent direction
        alpha = armijo_line_search(f, x, p, grad)
        if alpha < epsilon:
            break
        x = x + alpha * p
        print("Iteration: ", i)
        print("Grad: ", grad_norm)
        print("Alpha: ", alpha)
        print("X = ", x)
        print("f(x): ", f(x))
    return x
```

The `steepest descent` function minimizes an objective function f starting from an initial point x_0. The function iteratively updates x until it converges to a minimum or reaches the maximum number of iterations. The main steps are as follows:

1. **Initialization:** Set x to the initial point x_0.
2. **Gradient Calculation:** In each iteration, compute the gradient $\nabla f(x)$ at the current point x using the `gradient` function. Calculate the norm of the gradient, $\|\nabla f(x)\|$, to determine if convergence has been reached.
3. **Convergence Check:** If the gradient norm is below a small threshold ϵ, the algorithm considers the minimum reached and exits the loop.
4. **Descent Direction:** Calculate the normalized descent direction $p = -\frac{\nabla f(x)}{\|\nabla f(x)\|}$, pointing in the opposite direction of the gradient.
5. **Step Size Selection:** Use the `armijo_line_search` function to determine the optimal step size α along the direction p. If α is below the threshold ϵ, the algorithm stops to avoid making negligible updates.
6. **Update x:** Update the current point x by moving along the descent direction, using $x = x + \alpha \cdot p$.

1.1.1 Results

Function 1 The function we are minimizing is:

$$f_1(x) = x_0^2 + x_1^2 + x_2^2$$

with the initial point:

$$x_0 = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$$

After several iterations of the steepest descent method, the final solution found is:

$$x^* = \begin{bmatrix} -2.8039 \times 10^{-6} \\ -2.8039 \times 10^{-6} \\ -2.8039 \times 10^{-6} \end{bmatrix} \approx \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

The norm of the gradient at the last iteration is:

$$\|\nabla f(x^*)\| \approx 2.5175 \times 10^{-4}$$

The value of the function at the final solution x^* is:

$$f(x^*) \approx 2.3586 \times 10^{-11}$$

Values that are very close to zero, such as -2.8039×10^{-6} and 2.3586×10^{-11}, can be considered as zero for practical purposes.

The steepest descent method required 10 iterations to reach this solution. The gradual decrease in the step size, along with the decreasing gradient norm, demonstrates good convergence behavior. The algorithm successfully minimized the function to a very small value, close to zero, as expected for this quadratic function centered at the origin. The number of iterations and small final gradient norm indicate that the method converged efficiently for this problem.

Steepest Descent Optimization Results for required functions The table summarizes the steepest descent optimization results for five different functions. All functions minimized effectively to values near their theoretical minima. For functions f_1 and f_4, the optimization converged to values close to zero, indicating successful minimization. Values with magnitudes on the order of 10^{-6} or smaller, such as some gradient norms and final x values, can be considered zero in practice.

Across all functions, final gradient norms approached zero, indicating the method converged to the minimum. Step sizes decreased with each iteration, especially for more complex functions like f_3 and f_5, allowing the algorithm to make finer adjustments as it approached the minimum.

Function	Initial x_0	Iterations	Final Grad. Norm	Final α	Final x	Final $f(x)$
f_1	$\begin{bmatrix}1.0\\1.0\\1.0\end{bmatrix}$	9	2.52×10^{-4}	1.22×10^{-4}	$\begin{bmatrix}-2.80 \times 10^{-6}\\-2.80 \times 10^{-6}\\-2.80 \times 10^{-6}\end{bmatrix}$	2.36×10^{-11}
f_2	$\begin{bmatrix}0.0\\0.0\end{bmatrix}$	40	1.15×10^{-5}	1.86×10^{-9}	$\begin{bmatrix}0.999979\\0.999987\end{bmatrix}$	-0.99999999976
f_3	$\begin{bmatrix}-1.2\\1.0\end{bmatrix}$	2024	3.81×10^{-3}	7.63×10^{-6}	$\begin{bmatrix}0.9985\\0.9970\end{bmatrix}$	2.25×10^{-6}
f_4	$\begin{bmatrix}2.0\\-2.0\end{bmatrix}$	320	3.53×10^{-5}	1.53×10^{-5}	$\begin{bmatrix}-0.0107\\-1.82 \times 10^{-7}\end{bmatrix}$	1.30×10^{-8}
f_5, $c = 1$	$\begin{bmatrix}1.0\\-1.0\end{bmatrix}$	14	1.06×10^{-4}	7.63×10^{-6}	$\begin{bmatrix}0.5641\\0.5641\end{bmatrix}$	0.5293
f_5, $c = 10$	$\begin{bmatrix}1.0\\-1.0\end{bmatrix}$	24	1.99×10^{-4}	3.81×10^{-6}	$\begin{bmatrix}0.4026\\0.4026\end{bmatrix}$	0.7688
f_5, $c = 100$	$\begin{bmatrix}1.0\\-1.0\end{bmatrix}$	123	1.13×10^{-3}	1.91×10^{-6}	$\begin{bmatrix}0.3598\\0.3598\end{bmatrix}$	0.8277

Table 1: Optimization results for various functions using the Steepest Descent method.

1.2 Newton Algorithm

```
def newton_method(f, x0, epsilon=1e-5, max_iter=100):
    x = np.array(x0, dtype=float)
    for _ in range(max_iter):
        grad = gradient(f, x)
        grad_norm = np.linalg.norm(grad)
        if grad_norm < epsilon:
            break
        hess = hessian(f, x)
        try:
            hess_inv = np.linalg.inv(hess)
        except np.linalg.LinAlgError:
            hess_inv = np.linalg.pinv(hess)
        p = -hess_inv @ grad
        x = x + p
    return x
```

- **Initialization**: The algorithm starts at an initial point x_0 and iteratively updates x to approach a minimum of the function f.
- **Gradient Calculation**: In each iteration, compute the gradient $\nabla f(x)$, which provides the direction of steepest ascent.
- **Hessian Calculation**: Compute the Hessian matrix $\nabla^2 f(x)$.If the Hessian is a singular matrix, it calculates the pseudo inverse matrix.
- **Newton Step**: The Newton step $p = \nabla^2 f(x)^{-1} * \nabla f(x)$ is calculated, giving the direction and magnitude for the next update of x.
- **Update** x: Move to a new point $x = x + p$
- **Convergence Check**: The process repeats until the gradient norm $\|\nabla f(x)\|$ falls below a small threshold ϵ

1.2.1 Results

The results obtained from optimizing the functions given with the Newton Method algorithm

Function	Initial x_0	Iterations	Final Grad. Norm	Final x	Final $f(x)$
f_1	$\begin{bmatrix}1.0\\1.0\\1.0\end{bmatrix}$	1	3.46×10^{-6}	$\begin{bmatrix}-4.92 \times 10^{-6}\\-4.92 \times 10^{-6}\\-4.92 \times 10^{-6}\end{bmatrix}$	7.25×10^{-11}
f_2	$\begin{bmatrix}0.0\\0.0\end{bmatrix}$	1	1.99998	$\begin{bmatrix}0.99998\\0.999985\end{bmatrix}$	-0.99999999975
f_3	$\begin{bmatrix}-1.2\\1.0\end{bmatrix}$	5	7.59×10^{-5}	$\begin{bmatrix}0.9970\\0.9940\end{bmatrix}$	8.94×10^{-6}
f_4	$\begin{bmatrix}2.0\\-2.0\end{bmatrix}$	13	2.07×10^{-5}	$\begin{bmatrix}1.03 \times 10^{-2}\\-5.0 \times 10^{-6}\end{bmatrix}$	1.12×10^{-8}
$f_5, c = 1$	$\begin{bmatrix}1.0\\-1.0\end{bmatrix}$	5	3.08×10^{-4}	$\begin{bmatrix}0.5641\\0.5641\end{bmatrix}$	0.5293
$f_5, c = 10$	$\begin{bmatrix}1.0\\-1.0\end{bmatrix}$	8	2.77×10^{-5}	$\begin{bmatrix}0.4026\\0.4026\end{bmatrix}$	0.7688
$f_5, c = 100$	$\begin{bmatrix}1.0\\-1.0\end{bmatrix}$	12	9.29×10^{-5}	$\begin{bmatrix}0.3598\\0.3598\end{bmatrix}$	0.8277

Table 2: Optimization results for various functions using Newton's Method.

1.3 BFGS Quasi-Newton Algorithm

```
def BFGS(f, x0, epsilon=1e-10, max_iter=100):
    xk = np.array(x0, dtype=float)
    B = np.eye(len(xk))
    for i in range(max_iter):
        grad_xk = gradient(f, xk)
        try:
            B_inv = np.linalg.inv(B)
        except np.linalg.LinAlgError:
            B_inv = np.linalg.pinv(B)
        p = - B_inv @ grad_xk
        alpha = armijo_line_search(f , xk , p , grad_xk)
        xk1 = xk + alpha * p
        grad_xk1 =  gradient(f, xk1)
        s = xk1 - xk
        y =  grad_xk1 - grad_xk
        if np.dot(s.T , y) > epsilon:
            B = B + np.outer(y , y) / np.dot(y , s) - np.dot(B @ np
               .outer(s , s) , B) / np.dot(s , B @ s)
        xk = xk1
        if np.linalg.norm(grad_xk) / (1 + abs(f(xk))) <  epsilon:
            return xk
    return xk
```

- **Initialization**: Start with an initial point x_0 and set an initial approximation B of the Hessian matrix as the identity matrix I.
- **Gradient Calculation**: At each iteration, calculate the gradient $\nabla f(x_k)$ at the current point x_k using the `gradient` function.
- **Direction Calculation**: Compute the inverse of B, denoted B^{-1}, or use a pseudo-inverse if B is nearly singular. Calculate the descent direction $p = -B^{-1}\nabla f(x_k)$.
- **Step Size Selection**: Use the `armijo_line_search` function to determine an appropriate step size α that satisfies the Armijo condition.
- **Update** x: Compute the new point $x_{k+1} = x_k + \alpha p$ and calculate the gradient $\nabla f(x_{k+1})$.
- **BFGS Matrix Update**: Compute the update vectors: $s = x_{k+1} - x_k$ and $y = \nabla f(x_{k+1}) - \nabla f(x_k)$

  If $s^T y > \epsilon$ (to ensure numerical stability), update B. This update ensures B remains a positive-definite approximation of the Hessian matrix.
- **Repeat Until Convergence**: Continue the iteration until the gradient norm is below a specified tolerance ϵ or the maximum number of iterations is reached.

1.3.1 Results

The results obtained from optimizing the functions given with the Newton Method algorithm

Function	**Initial** x_0	**Iterations**	**Final Grad. Norm**	**Final** x	**Final** $f(x)$
f_1	$\begin{bmatrix}1.0\\1.0\\1.0\end{bmatrix}$	1	1.39×10^{-11}	$\begin{bmatrix}-5.0 \times 10^{-6}\\-5.0 \times 10^{-6}\\-5.0 \times 10^{-6}\end{bmatrix}$	7.5×10^{-11}
f_2	$\begin{bmatrix}0.0\\0.0\end{bmatrix}$	2	5.00×10^{-6}	$\begin{bmatrix}0.99998\\0.999985\end{bmatrix}$	-0.99999999975
f_3	$\begin{bmatrix}-1.2\\1.0\end{bmatrix}$	999	3.18×10^{-2}	$\begin{bmatrix}0.9981\\0.9962\end{bmatrix}$	4.08×10^{-6}
f_4	$\begin{bmatrix}2.0\\-2.0\end{bmatrix}$	289	1.86×10^{-9}	$\begin{bmatrix}-7.73 \times 10^{-4}\\-5.00 \times 10^{-6}\end{bmatrix}$	2.53×10^{-11}
$f_5, c = 1$	$\begin{bmatrix}1.0\\-1.0\end{bmatrix}$	999	5.76×10^{-6}	$\begin{bmatrix}0.5641\\0.5641\end{bmatrix}$	0.5293
$f_5, c = 10$	$\begin{bmatrix}1.0\\-1.0\end{bmatrix}$	999	2.16×10^{-5}	$\begin{bmatrix}0.4026\\0.4026\end{bmatrix}$	0.7688
$f_5, c = 100$	$\begin{bmatrix}1.0\\-1.0\end{bmatrix}$	999	4.91×10^{-5}	$\begin{bmatrix}0.3598\\0.3598\end{bmatrix}$	0.8277

Table 3: Optimization results for various functions using the BFGS Method.

1.4 Conjugate Gradient Algorithm

```
def conjugated_gradient(f, x0, epsilon=1e-10,max_iter = 1000):
    x = np.array(x0, dtype=float)
    for i in range(max_iter):
        grad_xk = gradient(f, x)
        dk = - grad_xk
        grad_norm_xk = np.linalg.norm(grad_xk)
        alpha = armijo_line_search(f , x , dk, grad_xk)
        x =  x + alpha * dk
        grad_xk1 = gradient(f,x)
        grad_norm_xk1 = np.linalg.norm(grad_xk1)
        beta = ((grad_xk1)**2)/((grad_norm_xk)**2)
        dk = grad_xk1 + beta * dk
        if grad_norm_xk1 / (1 + abs(f(x))) <  epsilon:
            return x
    return x
```

- **Initialization**: Start at an initial point x_0 and set the initial search direction $d_k = -\nabla f(x_0)$, which points in the steepest descent direction. Compute the initial gradient norm to monitor convergence.
- **Iterative Update**: For each iteration, use the `armijo_line_search` function to find an optimal step size α in the current direction d_k, then update x to the new point $x_{k+1} = x_k + \alpha d_k$. Calculate the new gradient $\nabla f(x_{k+1})$.
- **Direction Adjustment (Conjugate Update)**: Calculate the conjugate gradient update coefficient β and update the search direction as $d_{k+1} = -\nabla f(x_{k+1}) + \beta d_k$. This adjustment makes d_k conjugate to previous directions, improving efficiency. The process repeats until the gradient norm falls below ϵ or the maximum number of iterations is reached.

1.4.1 Results

In the results obtained from the Conjugate Gradient method, we observe that for several functions, the algorithm reaches the maximum number of iterations (`max_iter`). This is due to the algorithm oscillating near the solution, struggling to converge precisely to the minimum.

For quadratic functions, however, the Conjugate Gradient method theoretically converges in n steps, where n is the dimension of the problem, as it generates conjugate directions that span the entire space of the solution. This rapid convergence in n steps makes the method highly efficient for purely quadratic problems, but the oscillatory behavior indicates challenges with non-quadratic functions or those with complex landscapes.

Function	Initial x_0	Iterations	Final Grad. Norm	Final x	Final $f(x)$
f_1	$\begin{bmatrix}1.0\\1.0\\1.0\end{bmatrix}$	1	1.39×10^{-11}	$\begin{bmatrix}-5.00 \times 10^{-6}\\-5.00 \times 10^{-6}\\-5.00 \times 10^{-6}\end{bmatrix}$	7.50×10^{-11}
f_2	$\begin{bmatrix}0.0\\0.0\end{bmatrix}$	999	7.63×10^{-6}	$\begin{bmatrix}0.99998\\0.999986\end{bmatrix}$	-0.99999999974
f_3	$\begin{bmatrix}-1.2\\1.0\end{bmatrix}$	999	5.02×10^{-2}	$\begin{bmatrix}0.9714\\0.9434\end{bmatrix}$	8.24×10^{-4}
f_4	$\begin{bmatrix}2.0\\-2.0\end{bmatrix}$	999	5.60×10^{-6}	$\begin{bmatrix}1.12 \times 10^{-2}\\-7.85 \times 10^{-6}\end{bmatrix}$	1.57×10^{-8}
$f_5, c = 1$	$\begin{bmatrix}1.0\\-1.0\end{bmatrix}$	999	4.30×10^{-5}	$\begin{bmatrix}0.5641\\0.5641\end{bmatrix}$	0.5293
$f_5, c = 10$	$\begin{bmatrix}1.0\\-1.0\end{bmatrix}$	999	7.84×10^{-5}	$\begin{bmatrix}0.4026\\0.4026\end{bmatrix}$	0.7688
$f_5, c = 100$	$\begin{bmatrix}1.0\\-1.0\end{bmatrix}$	999	6.27×10^{-4}	$\begin{bmatrix}0.3598\\0.3597\end{bmatrix}$	0.8277

Table 4: Optimization results for various functions using the Conjugate Gradient Method.

1.5 Other functions

```python
def gradient(f, x, eps=1e-5):
    grad = np.zeros_like(x, dtype=float)
    for i in range(len(x)):
        x_eps = np.array(x, dtype=float)
        x_eps[i] += eps
        grad[i] = (f(x_eps) - f(x)) / eps
    return grad
```

The `gradient` function approximates the gradient of a function f at a given point x using finite differences. For each component i of x, it changes x by a small value ϵ in the i-th direction, calculates the change in f, and then divides by ϵ to approximate the partial derivative with respect to x_i.

```python
def hessian(f, x, eps=1e-5):
    n = len(x)
    hess = np.zeros((n, n), dtype=float)
    for i in range(n):
        x_eps = np.array(x, dtype=float)
        x_eps[i] += eps
        grad_plus = gradient(f, x_eps)  # Gradient at x + eps in
            the i-th direction
        x_eps[i] -= 2 * eps
        grad_minus = gradient(f, x_eps)  # Gradient at x - eps in
            the i-th direction
        hess[i, :] = (grad_plus - grad_minus) / (2 * eps)
    return hess
```

The `hessian` function calculates an approximate Hessian matrix of a function f at a point x using finite differences. For each variable x_i, the function perturbs x slightly in the positive and negative i-th directions, then calculates the gradient of f at these perturbed

points. The finite difference between these two gradients provides an approximation of the second-order partial derivatives with respect to x_i.

```
def armijo_line_search(f, x, p, grad, alpha_init=1.0, c=1e-4,
   max_iter=100):
    alpha = alpha_init
    for _ in range(max_iter):
        if f(x + alpha * p) <= f(x) + c * alpha * np.dot(grad, p):
            return alpha
        alpha /= 2  # Reduce alpha by half each time
    print("Warning: Armijo line search did not converge.")
    return alpha
```

The `armijo_line_search` function finds a suitable step size α for moving along a descent direction p to minimize a function f at point x. Starting with an initial step size α, it iteratively halves α until the **Armijo condition** is satisfied: $f(x + \alpha p) \leq f(x) + \mu\alpha \nabla f(x) \cdot p$, where μ (default 10^{-4}) controls the tolerance for sufficient decrease.

1.6 Conclusion

In this project, we compared four optimization methods—Steepest Descent, Newton's Method, BFGS, and Conjugate Gradient—on various test functions, observing distinct strengths and limitations for each.

Steepest Descent was straightforward but slow, often requiring many iterations for complex functions due to its reliance on only gradient information.

Newton's Method converged quickly near the minimum by leveraging the exact Hessian, but its computational cost and instability issues with nearly singular Hessians limited its use on larger problems.

BFGS balanced efficiency and stability, approximating the Hessian to achieve faster convergence than Steepest Descent without the high cost of Newton's Method. It performed well but occasionally oscillated near the solution.

Conjugate Gradient excelled for quadratic functions, theoretically converging in n steps, but for non-quadratic functions, it sometimes oscillated near the minimum, hitting the iteration limit.

Overall, Newton's Method and BFGS are ideal for smaller, complex functions, while Conjugate Gradient is efficient for large-scale quadratic problems. Steepest Descent, though slower, provides a simple alternative when computational resources are limited.

2 Part 2

In this part, we aim to estimate the proportion of cancerous pixels in a mammogram image by modeling the pixel intensity values using a Gaussian mixture model. Specifically, we assume that pixels in a cancerous region follow a Gaussian distribution with parameters (μ_1, σ_1^2), while pixels in the non-cancerous background follow a separate Gaussian distribution with parameters (μ_2, σ_2^2). The overall proportion of cancerous pixels is denoted by P_1. Given a set of pixel intensity values sampled from the image, we estimate the parameters $(\mu_1, \sigma_1^2, \mu_2, \sigma_2^2, P_1)$ using maximum likelihood estimation (MLE).

We need to maximize the likelihood function, which represents the probability of observing the given pixel intensities under the Gaussian mixture model. As we have implemented different optimization methods for minimizing the output of a function, we need to put look for the minimum of $-f(x)$ However, since the likelihood is defined as a product of probabilities, calculating it can lead to numerical underflow. To solve this issue, we instead maximize the sum of the log-likelihoods, which is mathematically equivalent and numerically more stable. This approach simplifies the optimization process by converting the product of probabilities into a sum of log-probabilities.

To set up the initial values, we set them based on assumptions about the pixel intensities in the mammogram image. Since cancerous regions are typically associated with brighter pixels, we initialize the mean intensity of cancerous pixels, μ_1, at a relatively high value of 200. Conversely, for non-cancerous regions, which are expected to have lower intensities, we set the mean μ_2 to 100. Both variances σ_1 and σ_2 are initialized at 250 to allow flexibility in capturing the spread of pixel intensities in each region. Finally, we initialize the proportion of cancerous pixels P_1 at 0, reflecting an initial assumption of no cancerous pixels in the image. These starting values provide a reasonable baseline while allowing the optimization process to adjust the parameters to fit the data accurately.

Running the optimization of the steepest descent method, the results are the following:

Iteration	Grad	Alpha	X Values	f(x)
0	65.290	0.125	[199.999, 250.000, 100.016, 250.001, 0.376]	1076.943
1	34.269	0.063	[199.998, 250.001, 100.032, 250.001, 0.437]	1076.313
100	25.051	0.063	[199.952, 250.015, 100.705, 250.021, 0.386]	1075.197
500	25.161	0.063	[199.630, 250.196, 108.065, 250.180, 0.374]	1073.593
960	6.277	0.016	[199.730, 250.360, 113.029, 250.231, 0.389]	1072.438

Table 5: Reduction of $f(x)$ across selected iterations.

The array x is represented as:

$$x = \left(\mu_1, \sigma_1^2, \mu_2, \sigma_2^2, P_1\right)$$

- μ_1: Mean of the first distribution
- σ_1^2: Variance of the first distribution
- μ_2: Mean of the second distribution
- σ_2^2: Variance of the second distribution
- P_1: Probability of cancer

Although we have not reached the minimum value after 1000 iterations applying the steepest descent method explain in Part 1., we can see that results should be near this values of x.

Discussing the Newton Method function, we can mentioned that at iteration 68, the gradient norm reached a very small value of 1.39×10^{-5}, indicating that the optimization

has nearly converged.

The parameters at convergence are as follows:

Parameter	Estimated Value
μ_1	197.687
σ_1^2	552.646
μ_2	118.284
σ_2^2	196.124
P_1 (Proportion of cancerous pixels)	0.418

Table 6: Estimated parameters at convergence with Newton method.

The final log-likelihood value of 980.615 reflects the best fit of the model to the data. The estimates of μ_1, σ_1^2, μ_2, σ_2^2, and P_1 correspond to the parameters for two distributions, representing different pixel classes (e.g., cancerous vs. non-cancerous), where P_1 gives the proportion of cancerous pixels. The results suggest that the model successfully distinguished between the two pixel classes, as seen in the estimated parameter values.

Parameter	Estimated Value
μ_1	197.684
σ_1^2	552.791
μ_2	118.283
σ_2^2	196.110
P_1 (Proportion of cancerous pixels)	0.418

Table 7: Estimated parameters for the BFGS method at convergence (30 iterations).

Solving this problem with the BFGS method converges in 30 iterations and the results obtained are showed in the upper table.

Parameter	Estimated Value
μ_1	22.636
σ_1^2	16000.000
μ_2	0.00001
σ_2^2	16000.000
P_1 (Proportion of cancerous pixels)	1.000

Table 8: Estimated parameters for the Conjugate Gradient method after 1000 iterations (non-convergence).

The Conjugate Gradient method did not converge within the 1000 iterations. Initially, it appeared to perform well, showing promising convergence behavior similar to that of the other methods. However, as the iterations progressed, the convergence rate began to slow significantly, and eventually, the method stopped making meaningful progress toward the optimal solution. This led to unrealistic parameter estimates, particularly for σ_1^2, σ_2^2, and P_1. The observed behavior suggests that while the Conjugate Gradient method can be effective in some stages of the optimization, it may struggle with complex, non-quadratic surfaces or require further tuning to achieve stable convergence in this specific problem.