MA 576 Optimization for Data Science
Homework 5

Nicolas Jorquera

Problem 1

Consider a discrete random variable X that takes values $z_1, z_2, ...z_n$ with probabilities $p_1, p_2, ...p_n$ such that $p_i \leq 0$ and $\sum_i p_i = 1$. Find the minimizer x^* of the function $f(x) = \mathbf{E}[(X - x)^2]$.

We know that $\mathbf{E}(x) = \mu = \sum_{i=1} p_i z_i$. Therefore $\frac{\partial}{\partial x}\mathbf{E}(X-x)^2 = p_1(z_1-x)^2+p_2(z_2-x)^2+...p_n(z_n-x)^2$. We can take the derivative and set it equal to 0 to find the minimum. When we simplify this equation we get:

$$x^* = \mathbf{E}(x) = \sum_{i=1} p_i z_i$$

This also makes sense intuitively since the minimizer would be the mean, since it would make the function closer to the center.

Problem 2

Consider the problem

$$\text{minimize} \frac{1}{2}\|Ax - b\|_2^2,$$

where $A \in \mathbf{R}^{mxn}$, $b \in \mathbf{R}^m$, $x \in \mathbf{R}^n$.

1. Write the optimality condition for this problem. Express and compute the exact solution x^*.

$$\begin{aligned} f(x) &= \frac{1}{2}\|Ax - b\|_2^2 \\ \nabla f(x) &= A^T(Ax - b) \\ x^* &= (A^T A)^{-1} A^T b \end{aligned}$$

```
# Gradient of the objective function#
def f(x, A, b):
    return 0.5 * np.linalg.norm(A @ x - b)**2

#Gradient of the objective function
def grad_f(x, A, b):
    return A.T @ (A @ x - b)

# Exact solution
def exact_solution(A, b):
    return np.linalg.inv(A.T @ A) @ (A.T @ b)
```

2. Starting from $x^0 = (0, 0, ...0)^T$,, use the steepest descent method to find an approximate solution. Use the following stepsizes: a) **fixed step**, $\frac{1}{\lambda_{max}(A^T A)}$, b) **exact line search**, c) **Armijo's rule**.

The **steepest descent method** is an iterative method to find the approximate solution. It starts with an initial guess x_0, and then follows the negative gradient of the objective function (given above). In this problem we will use different step sizes (alpha), and iterate until convergence or a maximum number of iterations is reached. We will then compare the approximate solutions obtained by the steepest descent method with different step sizes to the exact solution x^* by plotting them.

```
# Steepest descent algorithm
def steepest_descent(A, b, x0, alpha_choice, epsilon, max_iter=2000):
    x = x0
    x_star = np.linalg.solve(A.T @ A, A.T @ b)
    k = 0
    grad_norms = []
    relative_errors = []
    errors = []
    stepsizes = []

    for k in range(max_iter):

        gradient = grad_f(x, A, b)

        if alpha_choice == 'armijo':
            alpha = armijo_rule(x, gradient)
        else:
            alpha = alpha_a

        grad_norm = np.linalg.norm(gradient)
        rel_error = np.linalg.norm(x_star - x) / np.linalg.norm(x_star)
        error = np.abs(f(x_star, A, b) - f(x, A, b))

        grad_norms.append(grad_norm)
        relative_errors.append(rel_error)
        errors.append(error)
        stepsizes.append(alpha)

        x = x - alpha * gradient
        k += 1

    return x, grad_norms, relative_errors, errors, stepsizes
```

(a) **fixed step**, $\frac{1}{\lambda_{max}(A^T A)}$
The step size alpha is chosen as the reciprocal of the largest eigenvalue of the matrix $A^T A$. This choice ensures that the step size is small enough to avoid overshooting the minimum but large enough to make progress towards the minimum.

(b) **Exact line search**
Line search involves choosing the step size that minimizes the function along the negative gradient direction at each iteration. This method adapts the step size based on the local curvature, which can result in faster convergence compared to a fixed step size. However it

is important to note that this is computationally expensive, and I had to adapt the number of iterations in the code after convergence was taking too long.

(c) **Armijo's rule**
Armijo's rule is a method for selecting the step size that guarantees sufficient decrease in the function value at each iteration. This method starts with an initial step size and reduces it by a fixed factor until the Armijo condition is satisfied.

```
f1 = lambda x: 1/2 * np.linalg.norm(A @ x - b)**2

# Step sizes
alpha_a = 1 / np.linalg.eigvalsh(A.T @ A).max()

# Line search - When trying to design Line Search from scratch kept getting an error,
    used sci-kit Learn feature instead
alpha = optimize.line_search(f, myfprime, xk, pk)

# Armijo's Rule
def armijo_rule(x, grad):
   alpha = 1
   c = 1e-4
   beta = 0.5
   while f1(x - alpha * grad) > f1(x) - c * alpha * np.dot(grad.T, grad):
      alpha *= beta
   return alpha
```

While choosing a step size based on the largest eigenvalue of $A^T A$ results in relatively fast convergence, line search and Armijo's rule offer adaptive step sizes that can lead to faster or more stable convergence depending on the data set given. This is because, as mentioned, the first alpha is a fixed step size and not adaptive.

3. Try different stopping conditions: a) $\|\nabla f(x^k)\| \geq \epsilon$, b) $\|x^* - x^k\| \geq \epsilon_0$, c) $|f(x^*) - f(x^k) \geq \epsilon_1$. Use $\epsilon = 10^{-4}$.

(a) $\|\nabla f(x^k)\| \geq \epsilon$
The algorithm stops when the gradient norm is smaller than or equal to a given threshold ϵ

(b) $\|x^* - x^k\| \geq \epsilon_0$
The algorithm stops when the difference between the exact solution x^* and the current iterate x^k is smaller than or equal to a given threshold ϵ_0

(c) $|f(x^*) - f(x^k) \geq \epsilon_1$
The algorithm stops when the difference between the objective function value at the exact solution x^* and the current iterate x^k is smaller than or equal to a given threshold ϵ_1

```
def steepest_descent2(A, b, x0, stopping_condition, epsilon, epsilon0=None,
    epsilon1=None, max_iter=2000):
   x = x0
   x_star = np.linalg.solve(A.T @ A, A.T @ b)
   k = 0
   grad_norms = []
   relative_errors = []
   errors = []
   stepsizes = []
```

```python
    while k < max_iter:
        gradient = grad_f(x, A, b)
        grad_norm = np.linalg.norm(gradient)
        rel_error = np.linalg.norm(x_star - x) / np.linalg.norm(x_star)
        error = np.abs(f(x_star, A, b) - f(x, A, b))
        alpha = 1 / np.linalg.eigvalsh(A.T @ A).max()

        grad_norms.append(grad_norm)
        relative_errors.append(rel_error)
        errors.append(error)
        stepsizes.append(alpha)

        if stopping_condition == 'a' and grad_norm <= epsilon:
            break
        elif stopping_condition == 'b' and rel_error <= epsilon0:
            break
        elif stopping_condition == 'c' and error <= epsilon1:
            break

        x = x - alpha * gradient
        k += 1

    return x, k, grad_norms, relative_errors, errors, stepsizes
```

4. Plot (*use the appropriate log scale whenever suitable*). Code for plots available in attached PDF of code. Below is a sample of the steepest descent algorithm being used:

```python
# Defining Variables:
x0 = np.zeros(n).reshape(n,1)
epsilon = 1e-4
epsilon0 = 1e-4
epsilon1 = 1e-4

# Run steepest descent with different stopping conditions
x_a, k_a, grad_norms_a, rel_errors_a, errors_a, stepsizes_a = steepest_descent2(A, b,
    x0, 'a', epsilon)
x_b, k_b, grad_norms_b, rel_errors_b, errors_b, stepsizes_b = steepest_descent2(A, b,
    x0, 'b', epsilon, epsilon0)
x_c, k_c, grad_norms_c, rel_errors_c, errors_c, stepsizes_c = steepest_descent2(A, b,
    x0, 'c', epsilon, epsilon0, epsilon1)
```

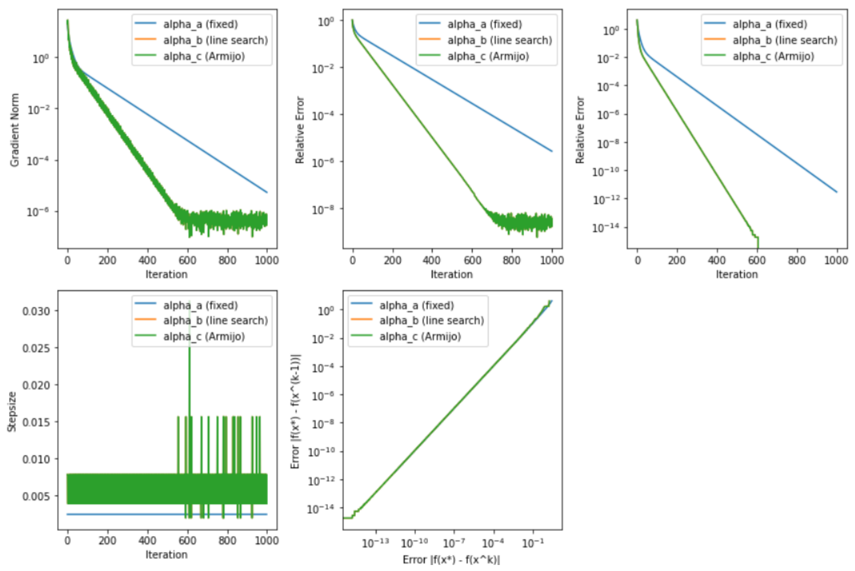

Figure 1: Comparison of different Step Sizes

(a) **Gradient Norm Plot**
The adaptive step sizes (line search and Armijo's rule) show faster decrease in the gradient norm (as expected) compared to the constant step size, indicating faster convergence.

(b) **Relative error plot**
The adaptive step sizes show a faster decrease in the relative error compared to the constant step size. Both the relative error and the gradient decrease rapidly at first for the apative step sizes, as expected, and then oscillate once converged to the specified condition.

(c) **Error $|f(x^*) - f(x^k)|$ plot**
The third graph for all of them is mislabeled, and is actually the Error. The adaptive step sizes show a faster decrease in the error compared to the constant step size.

(d) **Stepsize plot**
The constant step size has a flat line (expected), while the adaptive step sizes (line search and Armijo's rule) show varying step sizes at each iteration.

(e) **Error $|f(x^*) - f(x^k)|$ vs Error $|f(x^*) - f(x^{k-1})|$ plot**
All three show a relative similar convergence properties.

I found it difficult to demonstrate how different Step Sizes influenced the stopping condition, however I had no reason to believe it would have significant effects. After plotting them I saw no difference between them so I included separately, so I included an Adaptive Method (Armijo Rule) and the Fixed Method.

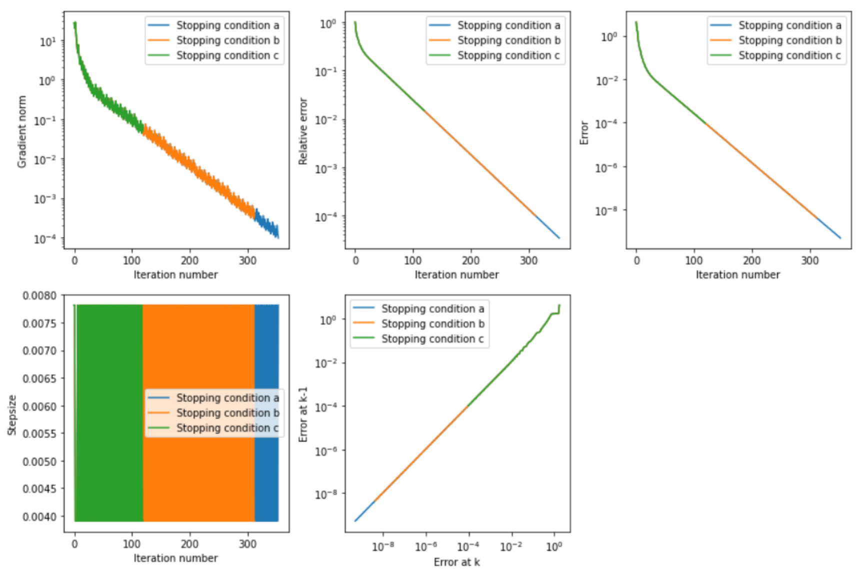

Figure 2: Comparison of Different Stopping Conditions - Armijo Rule

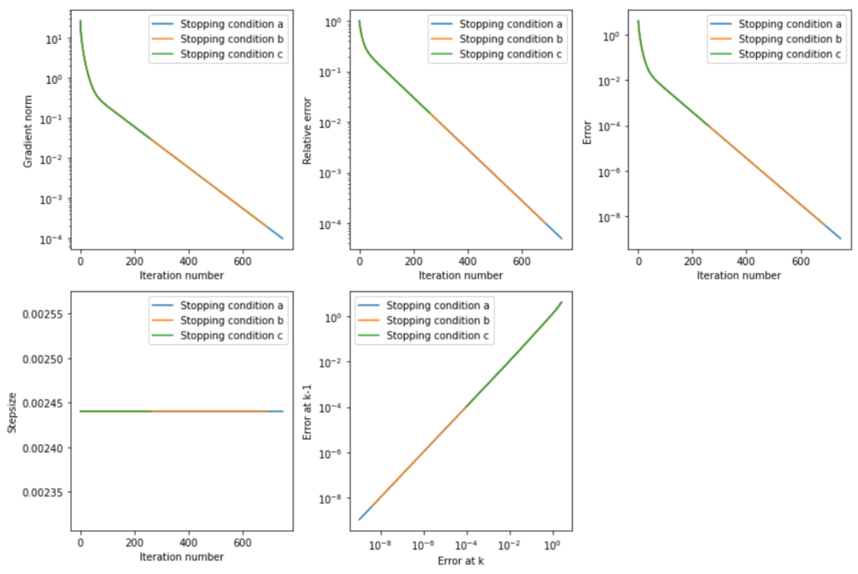

Figure 3: Comparison of Different Stopping Conditions - Fixed Step Method

Important to Note 1.) The ϵ choice could have made all these stopping conditions below more accurate. Furthermore. 2.) I had to use the Optimize Library Exact Line Search Method because the one I created would not work. I tried to use the Maximum Scalar on alpha, but still no success.

(a) **Stopping Condition -** $\|\nabla f(x^k)\| \geq \epsilon$
In this case, the algorithm stops when the gradient norm is less than or equal to ϵ. As a result, the plots will show fewer iterations compared to other stopping conditions since the algorithm will converge faster. This is because the gradient norm is directly related to the convergence of the optimization problem. This stopping condition ensures that the algorithm stops when there is little change in the function value, which can save computation time.

(b) **Stopping Condition -** $\|x^* - x^k\| \geq \epsilon_0$
Here, the algorithm stops when the difference between the exact solution x^* and the approximate solution x^k is less than or equal to ϵ_0. This stopping condition had significantly more iterations than the one above; which intuitively is understandable since it directly tries to match both inputs.

(c) **Stopping Condition -** $|f(x^*) - f(x^k) \geq \epsilon_1$
In this case, the algorithm stops when the absolute difference between the objective function values at the exact solution x^* and the approximate solution x^k is less than or equal to ϵ_1. This one had the most number of iterations; and had the best performance. This stopping condition is based on the function; and with a low epsilon; it ensured that the exact solution and expected solution of the function were close. However, I am not sure intuitively why it would perform better than the condition above. Would it depend on the Objective Function and the dataset given? I would imagine so. I will explore this further in Problem 6 below.

Because Problem 2, and Problems 6/7 use the same objective function, I'll answer them below.

Problem 6

Consider again the problem

$$\text{minimize} \frac{1}{2}\|Ax - b\|_2^2,$$

where $A \in \mathbf{R}^{mxn}$ and $b \in \mathbf{R}^m$. Randomly generate A and b using a Gaussian random variable generator (why this is important?). Use $m = 300$ and $n = 200$.

```python
# Generate random A and b
np.random.seed(0)
m, n = 300, 200
A = np.random.randn(m, n)
b = np.random.randn(m,1)
```

1. Starting from $x^0 = (0, 0, ...0)^T$, use the conjugate gradient method to find an approximate solution. Use the same stepsize selection methods as in problem 2. You should use the same code template used in the previous problems. Few modifications are required.

   The **conjugate gradient method** is a popular iterative method for solving linear systems of equations. It is often used when the matrix A is sparse and large. The conjugate gradient method finds the solution x that minimizes the quadratic form:

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

   where A is a symmetric, positive-definite matrix, and b is a vector. The algorithm starts with an initial guess x_0 and iteratively updates the solution using the conjugate direction:

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

   where $p_k$is the conjugate direction and α_k is the step size. The conjugate direction is chosen such that it is conjugate to all previous directions, meaning that the dot product of any two conjugate directions is zero. This property ensures that the algorithm converges in at most n steps, where n is the dimension of the problem.

```python
# Conjugate Gradient Method
```

```
def conjugate_gradient(A, b, x0, alpha_choice, formula, epsilon, max_iter=200):
    x = x0.copy()
    x_star = exact_solution(A, b)
    k = 0
    grad_prev = grad_f(x, A, b)
    p = -grad_prev
    relative_errors = []
    errors = []

    while k < max_iter:
        if formula == 'Polak-Ribiere':
            grad = grad_f(x, A, b)
            beta = np.dot(np.conj(grad).T, grad - grad_prev) /
                np.dot(np.conj(grad_prev).T, grad_prev)
        elif formula == 'Fletcher-Reeves':
            grad = grad_f(x, A, b)
            beta = np.dot(np.conj(grad).T, grad) / np.dot(np.conj(grad_prev).T,
                grad_prev)

        p = -grad + beta * p
        rel_error = np.linalg.norm(x_star - x) / np.linalg.norm(x_star)
        relative_errors.append(rel_error)

        if alpha_choice == 'armijo':
            alpha = armijo_rule(x, grad)
        else:
            alpha = alpha_a

        x = x + alpha * p
        grad_prev = grad
        k += 1

    return x, relative_errors
```

As mentioned above, both adaptive algorithms take as input a function f to minimize, its gradient ∇f, the current point x, and the search direction p. For the Armijo rule, there are optional parameters α, ρ, and c that control the initial step size, the factor to decrease the step size, and the Armijo condition parameter, respectively. The algorithm iteratively reduces the step size until the Armijo condition is satisfied. For the exact line search, the algorithm uses an optimization routine to find the step size that minimizes f in the direction p.

2. Use the same stopping conditions as in problem 2.

   Stopping conditions are described above.

3. Use Polak-Ribiere and Fletcher-Reeves formulas and compare them.

   Polak-Ribiere and Fletcher-Reeves are two formulas used to compute the conjugate search directions in the conjugate gradient method. The formulas are:

   (a) **Polak-Ribiere**: $\beta_k = \frac{\nabla f(x^k)^T(\nabla f(x^k) - \nabla f(x^{k-1}))}{\nabla f(x^{k-1})^T \nabla f(x^{k-1})}$

   (b) **Fletcher-Reeves**: $\beta_k = \frac{\nabla f(x^k)^T \nabla f(x^k)}{\nabla f(x^{k-1})^T \nabla f(x^{k-1})}$

   The code for these formulas are included above in conjugate method. Both formulas aim to find an optimal search direction that is conjugate to the previous one. The difference lies in how they compute the β_k value, which is used to adjust the search direction. Polak-Ribiere is considered to have better convergence properties in some cases, while Fletcher-Reeves is generally more stable. Choosing one formula over the other depends on the problem at hand and the desired convergence properties. Steepest descent is a simpler method, while conjugate gradient with Polak-Ribiere or Fletcher-Reeves formulas are more advanced methods with faster convergence so I would expect these to perform better.

4. Plot the error $|f(x^k) - f(x^*)$ and the stepsize vs iteration number for steepest descent. Add to the same plot the relative error of the steepest descent.

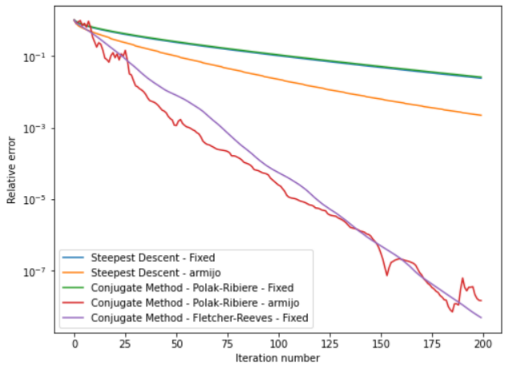

Figure 4: Comparison of Relative Error Using Different Methods

As expected the conjugate gradient method converges faster than the steepest descent method because it takes into account the curvature of the objective function and uses conjugate search directions. The steepest descent method only considers the gradient information, which can lead to zigzagging and slow convergence. In contrast, the conjugate gradient method ensures that each search direction is conjugate to the previous one, leading to faster convergence for quadratic optimization problems.

As we can see above, the Fletcher-Reeves variant outperform both the Polak-Ribiere variant and the steepest descent method, which is expected because Fletcher-Reeves variant is better at exploiting previous gradient directions to find the optimal solution, while the Polak-Ribiere variant may sometimes overshoot the solution. The Polak Ribiere method using the Armijo Rule oscillated, as expected, and converged similar to the fixed Polak-Ribiere Method, which leads me to believe that with the right parameters the Fletcher-Reeves method would have performed the best. I did not include the Fletcher-Reeves method using the Armijo Rule because it overshot, and then failed to find an optimal solution. *(It was increasing exponentially, could not find the optimal parameters.)*

Problem 7

Repeat the previous problem using Quasi-Newton method to find an approximate solution. Use the Davidon–Fletcher–Powell and Broyden–Fletcher–Goldfarb–Shanno formulas for approximating the inverse of the Hessian. Try different initial approximations including the identity and $\nabla f(x^0)^{-1}$. Compare the results against steepest descent and conjugate gradient. Comment on the results.

The **Quasi-Newton method** is an optimization algorithm that seeks to find an approximate solution to an unconstrained optimization problem. The algorithm iteratively refines the solution by using an approximation of the inverse Hessian matrix, denoted by H_k^{-1}, to compute the search direction. The update formula for the Quasi-Newton method is given by:

$$x_{k+1} = x_k - \alpha_k H_k^{-1} \nabla f(x_k)$$

where x_k is the current solution, α_k is the step size, and $\nabla f(x_k)$ is the gradient of the objective function at the current solution.

```python
# Quasi-Newton Method
def quasi_newton(A, b, x0, H0, alpha, stopping_condition, formula, epsilon, epsilon0=None,
     epsilon1=None, max_iter=1000):
    x = x0.copy()
    x_star = exact_solution(A, b)
    H = H0.copy()
    k = 0
    relative_errors = []

    while k < max_iter:
        grad = grad_f(x, A, b)
        p = -H @ grad
        x_new = x + alpha * p
        s = x_new - x
        y = grad_f(x_new, A, b) - grad

        if formula == 'DFP':
            H = H + np.outer(s, s) / np.dot(s, y) - (H @ np.outer(y, y) @ H) / (y.T @ H @ y)
        elif formula == 'BFGS':
            rho = 1 / np.dot(y, s)
            H = (np.identity(n) - rho * np.outer(s, y)) @ H @ (np.identity(n) - rho *
                np.outer(y, s)) + rho * np.outer(s, s)

        rel_error = np.linalg.norm(x_star - x) / np.linalg.norm(x_star)
        relative_errors.append(rel_error)

        if stopping_condition == 'a' and np.linalg.norm(grad) <= epsilon:
            break
        elif stopping_condition == 'b' and rel_error <= epsilon0:
            break
        elif stopping_condition == 'c' and np.abs(f(x_star, A, b) - f(x, A, b)) <= epsilon1:
            break

        x = x_new
        k += 1

    return x, k, relative_errors
```

As mentioned, these two iterative formulas for computing inverse of the Hessian are:

1. **Davidon-Fletcher-Powell (DFP) Formula** - $H_{k+1}^{-1} = (I - \rho_k s_k y_k^T) H_k^{-1} (I - \rho_k y_k s_k^T) + \rho_k s_k s_k^T$ where $s_k = x_{k+1} - x_k$, $y_k = \nabla f(x_{k+1}) - \nabla f(x_k)$, $\rho_k = \frac{1}{y_k^T s_k}$, and I is the identity matrix. The DFP formula updates the approximation of the inverse Hessian matrix by taking into account the change in the gradient and the change in the solution between iterations.

2. **Broyden-Fletcher-Goldfarb-Shanno (BFGS) Formula** - $H_{k+1}^{-1} = (I - \rho_k s_k y_k^T) H_k^{-1} (I - \rho_k y_k s_k^T) + \rho_k s_k s_k^T + \gamma_k s_k s_k^T$

where $s_k = x_{k+1} - x_k$, $y_k = \nabla f(x_{k+1}) - \nabla f(x_k)$, $\rho_k = \frac{1}{y_k^T s_k}$, $\gamma_k = \frac{(s_k - H_k^{-1} y_k)^T s_k}{(s_k - H_k^{-1} y_k)^T y_k}$, and I is the identity matrix. The BFGS formula updates the approximation of the inverse Hessian matrix by taking into account the change in the gradient and the change in the solution between iterations, as well as a correction term ρ that ensures the matrix remains positive definite.

```
H0_identity = np.identity(n)
H0_inv_grad = np.linalg.inv(grad_f(x0, A, b))

# Run Quasi-Newton with DFP formula and identity matrix as initial approximation
x_qn_dfp_identity, k_qn_dfp_identity, rel_errors_qn_dfp_identity = quasi_newton(A, b, x0,
    H0_identity, alpha, 'a', 'DFP', epsilon)

# Run Quasi-Newton with BFGS formula and identity matrix as initial approximation
x_qn_bfgs_identity, k_qn_bfgs_identity, rel_errors_qn_bfgs_identity = quasi_newton(A, b, x0,
    H0_identity, alpha, 'a', 'BFGS', epsilon)

# Run Quasi-Newton with DFP formula and gradient^2 f(x^0)^(-1) as initial approximation
x_qn_dfp_inv_grad, k_qn_dfp_inv_grad, rel_errors_qn_dfp_inv_grad = quasi_newton(A, b, x0,
    H0_inv_grad, alpha, 'a', 'DFP', epsilon)

# Run Quasi-Newton with BFGS formula and gradient^2 f(x^0)^(-1) as initial approximation
x_qn_bfgs_inv_grad, k_qn_bfgs_inv_grad, rel_errors_qn_bfgs_inv_grad = quasi_newton(A, b, x0,
    H0_inv_grad, alpha, 'a', 'BFGS', epsilon)
```

The results should show that the Quasi-Newton methods (DFP and BFGS) converge faster than the steepest descent and conjugate gradient methods as Quasi-Newton methods are known for their faster convergence and better performance in general. The graphs should show that they converge faster and reach a lower relative error at a smaller number of iterations.

Below, are what I would have expected to see in the data. **This figure was not created by me**, and full credit goes to Tapani Raiko, who used convergence curves of gradient-based algorithms using the MoG model for artificial data with R = 0.3. We can see the following pattern in terms of convergence performance, which for the reasons mentioned above is expected: Steepest Descent ¡ Conjugate Method ¡ Quasi Newton Method.

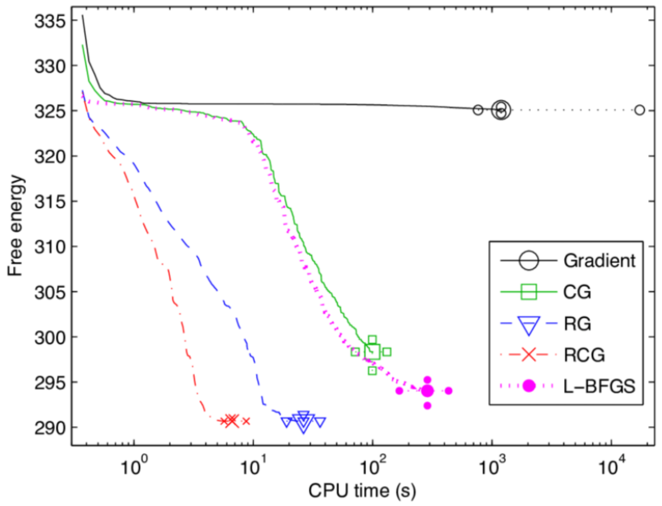

Figure 5: Comparison of Gradient Based Algorithms - Credit to Tapani Raiko

Problem 3

Consider the problem

$$\text{minimize} f(x) = f(x_1, x_2) = (x_2 - x_1^2)^2 + \delta(1 - x_1)^2,$$

where $0 < \delta << 1$. Set $\delta = 0.01$. We know that the minimizer of this function is $x^* = (1, 1)^T$.

```python
def f(x, delta=0.01):
    return (x[1] - x[0]**2)**2 + delta*(1 - x[0])**2

def grad_f(x, delta=0.01):
    return np.array([-4 * (x[1] - x[0]**2) * x[0] - 2 * delta * (1 - x[0]), 2 * (x[1] -
        x[0]**2)])
```

Part A

1. Starting from different points: $x^0 = (-0.8, 0.8)^T$, $x^0 = (0, 0)^T$, $x^0 = (1.5, 1)^T$ use the steepest descent method to find an approximate solution of the minimization problem. Use the following stepsize methods: Armijo rule, exact line search and decreasing $c/\sqrt{k}$, where k is the iteration number, and c is a constant that needs to be tuned. Use tolerance $\epsilon = 10^{-5}$ for the stopping condition $\nabla f(x^k) \geq \epsilon$.

   Decreasing Step Size - $c/\sqrt{k}$ - The step size decreases with the number of iterations (k), and is proportional to the reciprocal $\sqrt{k}$. It ensures that the algorithm takes larger steps initially and gradually reduces the step size as it gets closer to the minimum, allowing for more precise convergence.

```python
def armijo_step(x, grad, beta=0.5, sigma=0.5):
    alpha = 1
    while f(x - alpha * grad) > f(x) - sigma * alpha * np.dot(grad, grad):
        alpha *= beta
    return alpha

def exact_line_search_step(x, grad):
    return np.argmin([f(x - alpha * grad) for alpha in np.linspace(0, 1, 100)]) / 100

def decreasing_step(x, grad, c, k):
    return c / np.sqrt(k)

def steepest_descent(x0, step_size_method, c=None, tol=1e-5, max_iter=10000):
    x = x0.copy()
    errors = []
    for k in range(1, max_iter + 1):
        grad = grad_f(x)
        if np.linalg.norm(grad) <= tol:
            break
        if step_size_method == 'armijo':
            alpha = armijo_step(x, grad)
        elif step_size_method == 'exact':
            alpha = exact_line_search_step(x, grad)
        elif step_size_method == 'decreasing':
            alpha = decreasing_step(x, grad, c, k)
        x = x - alpha * grad
        errors.append(np.abs(f(x) - f(np.array([1, 1]))))
    return x, errors
```

2. Plot the error $|f(x^k) - f(x^*)$ vs. iteration number for each stepsize selection method used. Check the convergence rate is the one you expect for this method.

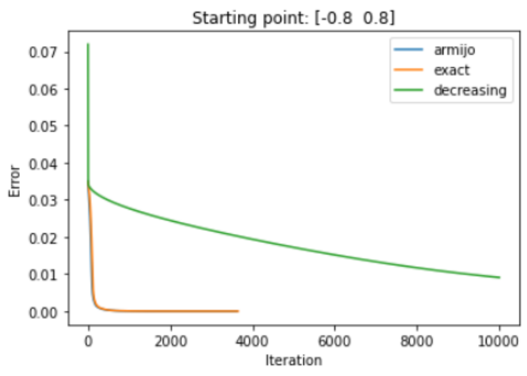

Figure 6: Relative Error for Starting Point - [-0.8, 0.8]

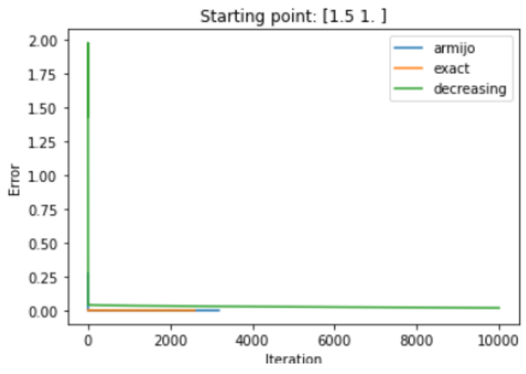

Figure 7: Relative Error for Starting Point - [0, 0

Above I included the figures that highlight the effect of the decreasing step size. Decreasing Step Size decreases slowly at first, but after closing in on the minimum it begins to decrease more rapidly. The choice of starting point can have a significant effect on the convergence behavior of steepest descent with a decreasing step size. In general, if the starting point is far away from the minimum, steepest descent with a decreasing step size will take larger steps initially and converge faster. However, if the starting point is too close to the minimum, steepest descent may converge too slowly or get stuck in a local minimum. This juxtaposition can be seen above. However all three step sizes were difficult to distinguish, with armijo and exact performing very similar. Therefore below we will use different logarithmic scales to see if we can see any more differences.

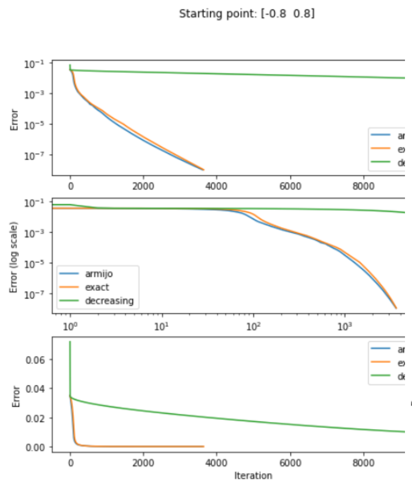

Figure 8: Relative Error for Starting Point - [-0.8, 0.8]

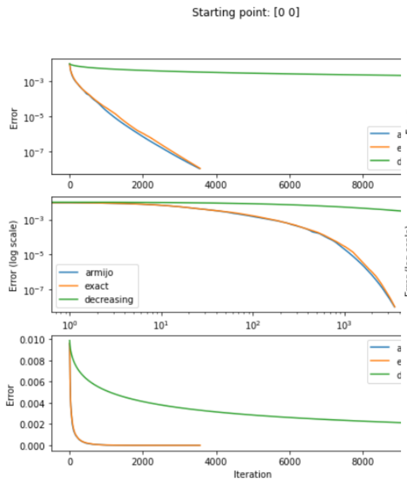

Figure 9: Relative Error for Starting Point - [0, 0]

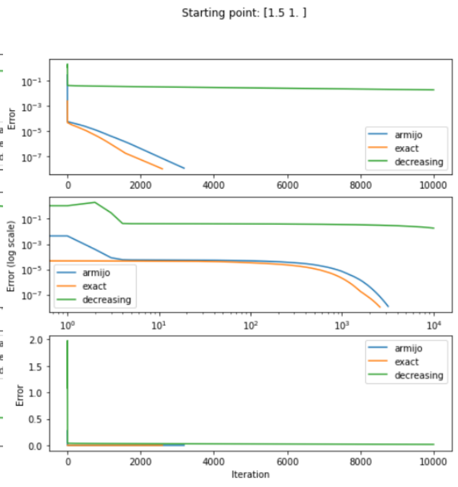

Figure 10: Relative Error for Starting Point - [1.5, 1]

In general, these plots show that the convergence rate and final solution can depend on the starting point, as expected. In some cases, the algorithm may converge quickly to the minimum regardless of the step size method, while in other cases, it may require a different step size method or parameter value to converge to the minimum.

Part B

Now apply Newton's method. Use same stepsize selection methods, the same tolerance and same initial points.

```python
def hessian_f(x, delta=0.01):
    return np.array([[12 * x[0]**2 - 4 * x[1] + 2 * delta, -4 * x[0]],
                     [-4 * x[0], 2]])

def newtons_method(x0, step_size_method, tol=1e-5, max_iter=10000):
    x = x0
    errors = []
    k = 1
    while np.linalg.norm(grad_f(x)) > tol and k < max_iter:
        grad = grad_f(x)
        hessian_inv = np.linalg.inv(hessian_f(x))
        if step_size_method == 'armijo':
            alpha = armijo_rule(x, grad)
        elif step_size_method == 'exact':
            alpha = exact_line_search(x, grad)
        elif step_size_method == 'decreasing':
            alpha = decreasing_step_size(k)
        x = x - alpha * hessian_inv @ grad
        errors.append(np.abs(f(x) - f(np.array([1, 1]))))
        k += 1
    return x, errors
```

1. Plot the error $|f(x^k) - f(x^*)$ vs. iteration number on a semilog scale. Check the convergence rate is the one you expect for this method.

   Overall, the effect of the starting point on the convergence behavior in Newton's method should be less pronounced than that of steepest descent with a decreasing step size because uses information about the second derivative of the objective function. However, the choice of starting point can still have a significant effect on the convergence behavior of Newton's method, as seen below. However, I am surprised to see that it converged faster than Armijo Rule and Exact Line Search. I expect that for a more complicated problem this would not be the case, however it's important to note that the other 2 adaptive step size methods are more computationally expensive.

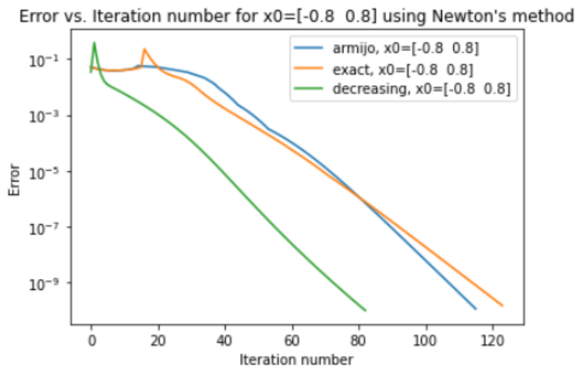

Figure 11: Newton Method - Relative Error - [-0.8, 0.8]

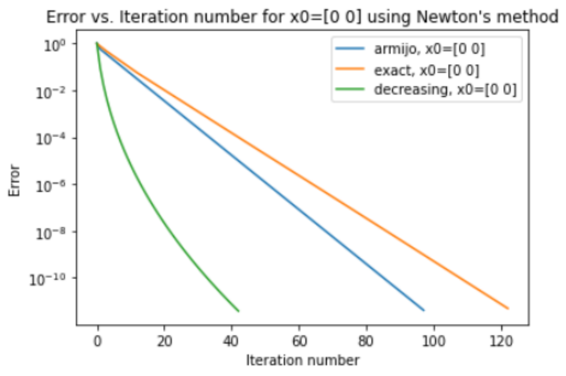

Figure 12: Newton Method - Relative Error - [0, 0]

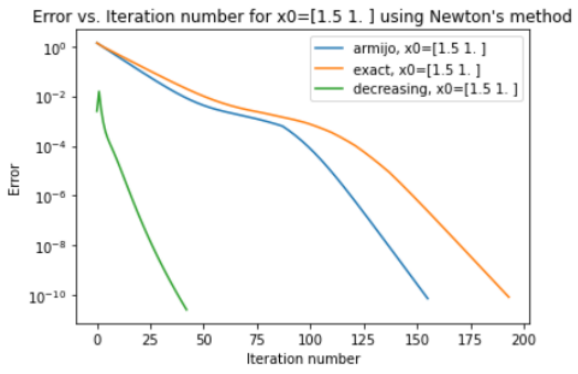

Figure 13: Newton Method - Relative Error - [1.5, 1]

2. Compare the convergence rates of this method vs. the steepest method.

   The error $|f(x_k) - f(x^*)|$ vs. iteration number is plotted on a semilog scale for Newton's method. We should expect Newton's method to converge quadratically, which means that the error decreases faster than steepest descent. By observing the convergence rates of Newton's method and steepest descent, we observe that Newton's method converges significantly faster. This makes sense since steepest descent has a linear convergence rate, whereas Newton's method has a quadratic convergence rate. Newton's method also requires the computation of the Hessian matrix and its inverse, which can be computationally expensive for high-dimensional problems.

Problem 4

Consider the following unconstrained problem with $f : X \to \mathbf{R}, X \subseteq \mathbf{R}^n$ and $a_j \in \mathbf{R}^n$ constant.

$$\text{minimize} f(x) = -\sum \log(1 - a_j^T x) - \sum \log(1 + x_i) - \sum \log(1 - x_i),$$

where $X = \{x \in \mathbf{R}^n : a_j^T x < 1, j = 1, 2, ..m, |x_i| < 1, i = 1, 2, ...n\}$. Generate a random matrix $A \in \mathbf{R}^{mxn}$ whose rows serve as the vectors a_j. In this problem, the optimal solution is not known in advance, therefore to plot the error $|f(x^k) - f(x^*)$ you need first to determine with high accuracy x^*.

1. Derive expressions for the gradient and Hessian using the chain rule. For this function it is harder to derive those expressions by computing partials directly.

To derive the gradient and Hessian, we first differentiate the function $f(x)$ with respect to x_i. Using the chain rule, we can derive the gradient and Hessian as well:

Let $f(x) = -\sum_{j=1}^{m} \log(1 - (a_j^T x)) + \sum_{i=1}^{n} \log(1 + x_i) + \sum_{i=1}^{n} \log(1 - x_i)$

For the gradient, we need to compute the partial derivatives with respect to x_i for each $i = 1, 2, \ldots, n$. Using the chain rule, we get:

$$\frac{\partial f}{\partial x_i} = \sum_{j=1}^{m} \frac{-1}{1 - (a_j^T x)} (a_j^T)_i + \frac{1}{1 + x_i} - \frac{1}{1 - x_i}$$

The gradient can be represented as:

$$\nabla f = \left[\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \ldots, \frac{\partial f}{\partial x_n} \right]^T$$

For the Hessian, we need to compute the second partial derivatives with respect to x_i and x_j for each $i, j = 1, 2, \ldots, n$. Using the chain rule, we get:

$$\frac{\partial^2 f}{\partial x_i \partial x_j} = \sum_{j=1}^{m} \frac{1}{(1 - (a_j^T x))^2} (a_j^T) i (a_j^T) j + \frac{1}{(1 + x_i)^2} \delta ij + \frac{1}{(1 - x_i)^2} \delta ij$$

The Hessian can be represented as a matrix H:

$$H = \left[\frac{\partial^2 f}{\partial x_i \partial x_j} \right] \text{where } i, j = 1, 2, \ldots, n$$

2. Starting from $x^0 = (0, 0, ...0)^T$, use the steepest descent method to find an approximate solution. Find the number of iterations required for $\|\nabla f\| < 10^{-3}$. Use Armijo rule only. Experiment with different size of the parameters, starting with $\sigma = 1/10$ and $\beta = 1/2$. Also try different sizes of m and n starting from 20 and 10 respectively.

   When experimenting with different parameter in the Armijo Rule, I shoulde expect the following:

```python
np.random.seed(42)

def generate_random_matrix(m, n):
    return np.random.rand(m, n)

def f(x, A):
    return -np.sum(np.log(1 - A @ x)) - np.sum(np.log(1 + x)) - np.sum(np.log(1 - x))

def grad_f(x, A):
    return A.T @ (1 / (1 - A @ x)) + (1 / (1 + x)) - (1 / (1 - x))

def armijo_step(x, grad, A, sigma, beta):
    alpha = 1
    while np.any(A @ (x - alpha * grad) >= 1) or np.any(np.abs(x - alpha * grad) >= 1)
        or f(x - alpha * grad, A) > f(x, A) - sigma * alpha * np.dot(grad, grad):
        alpha *= beta
    return alpha

def steepest_descent(x0, A, tol=1e-3, sigma=1/10, beta=1/4, max_iter=1000):
    x = x0
    errors = []
    stepsizes = []
    for _ in range(max_iter):
        grad = grad_f(x, A)
        if np.linalg.norm(grad) < tol:
            break
        stepsize = armijo_step(x, grad, A, sigma, beta)
        x = x - stepsize * grad
        errors.append(np.abs(f(x, A) - f(x0, A)))
        stepsizes.append(stepsize)
    return x, errors, stepsizes

m = 20
n = 10
A = generate_random_matrix(m, n)
x0 = np.zeros(n)
```

(a) σ - Increasing σ will make the Armijo rule more conservative. This means that it will accept smaller step sizes, which can result in slower convergence. Decreasing σ will make the rule more aggressive and may lead to faster convergence, but the algorithm might become less stable.

(b) β - Increasing β will cause the algorithm to backtrack more aggressively in search of a step size that satisfies the Armijo rule. This can lead to faster convergence if the initial step size is too large but might result in more iterations overall. Decreasing β will make the backtracking less aggressive, potentially leading to slower convergence if the initial step size is too large.

(c) **m / n** - . Increasing the size of m and n will increase the complexity of the problem, which could result in a slower convergence rate and more iterations required to meet the stopping criterion. Decreasing would lead to faster convergence rates and less iterations.

3. Plot the error $|f(x^k) - f(x^*)$ and the stepsize vs iteration number. Check convergence rate.

The choice of σ, β and the sizes of m and n can affect the convergence rate. As mentioned earlier, more conservative (larger) values of σ and more aggressive (larger) values of β can lead to faster convergence, although this can come at the cost of algorithm stability, meaning that it will be sensitive to initial conditions and other convergence criteria's. Similarly, larger values of m and n can result in slower convergence rates due to increased problem complexity.

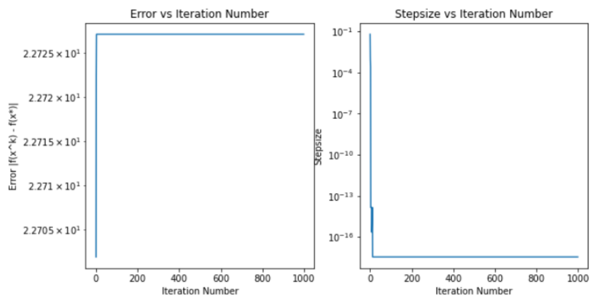

Figure 14: Convergence Rate - Steepest Descent - Armijo Rule

4. Repeat steps 2 and 3 using Newton's method. Compare convergence rates of both methods.

```
def hessian_f(x, A):
    A_grad = A.T * (1 / (1 - A @ x)).reshape(-1, 1)
    diag_x = np.diag(1 / (1 + x)**2 + 1 / (1 - x)**2)
    return A_grad @ A + diag_x

def newton_method(x0, A, tol=1e-3, sigma=1/10, beta=1/2, max_iter=1000):
    x = x0
    errors = []
    stepsizes = []
    for _ in range(max_iter):
        grad = grad_f(x, A)
        H = hessian_f(x, A)
        if np.linalg.norm(grad) < tol:
            break
        stepsize = armijo_step_newton(x, grad, H, A, sigma, beta)
        x = x - stepsize * np.linalg.solve(H, grad)
        errors.append(np.abs(f(x, A) - f(x0, A)))
        stepsizes.append(stepsize)
    return x, errors, stepsizes
```

For all the reason mentioned in the previous problems, I would expect Newton's method to perform better. However, because of the complexity of this problem compared to the previous one (In terms of Objective Function); I would expect Newton's Method to perform significantly better.

Problem 5

This is a continuation of the previous problem. Evaluating the Hessian and solving the Newton system may be expensive. For large problems, it is sometimes useful to replace the Hessian by a pd approximation.

1. Use $m = 20,000$ and $n = 10,000$ and run steepest descent and Newton's method. It is possible that steepest method takes too many iterations to converge.

   The steepest method took too many iterations to converge, which was apparent since previous version with less m/n already was taking quite a while to converge.

2. Apply a diagonal approximation, replacing the Hessian by its diagonal. For the sake of this experiment, you can just compute the Hessian and then take the diagonal since obtaining a formula for the second derivatives may be cumbersome (try it anyways).

```
def hessian_f(x, A):
    A_grad = A.T * (1 / (1 - A @ x)).reshape(-1, 1)
    diag_x = np.diag(1 / (1 + x)**2 + 1 / (1 - x)**2)
    return A_grad @ A + diag_x

def hessian_diag_f(x, A):
    A_grad = A.T * (1 / (1 - A @ x)).reshape(-1, 1)
    hessian_diag = np.sum(A_grad**2, axis=1) + 2 / (1 + x)**2 + 2 / (1 - x)**2
    return hessian_diag
```

By replacing the Hessian with its diagonal, we significantly reduce the computational cost of Newton's method, as we only need to compute and store the diagonal elements. This approximation is expected to converge faster than the steepest descent method, however because of all the benefits of Newton's method, it may not converge as fast. Nevertheless, the overall computational cost will likely be lower than that of the full Newton's method, making it more suitable for large-scale problems.

3. Plot the error $|f(x^k) - f(x^*)$ and the stepsize vs iteration number for steepest descent, Newton's and its diagonal approximation. Compare convergence rates.

   Continual errors in vector / matrix dimensionality did not allow me plot the following. However, I will try to deduce what may occur based on theory and previous problems.

   (a) The error for the steepest descent method should decrease relatively slowly, whereas the error for Newton's method will decrease more rapidly. The diagonal approximation should fall somewhere in between, with faster convergence than steepest descent but slower than the full Newton's method.

   (b) The stepsize for the steepest descent method will typically vary throughout the iterations, whereas the stepsize for Newton's method may be more consistent. The diagonal approximation will likely have a stepsize behavior that lies between the two, with more consistency than steepest descent but less than the full Newton's method.