Untitled

April 24, 2024

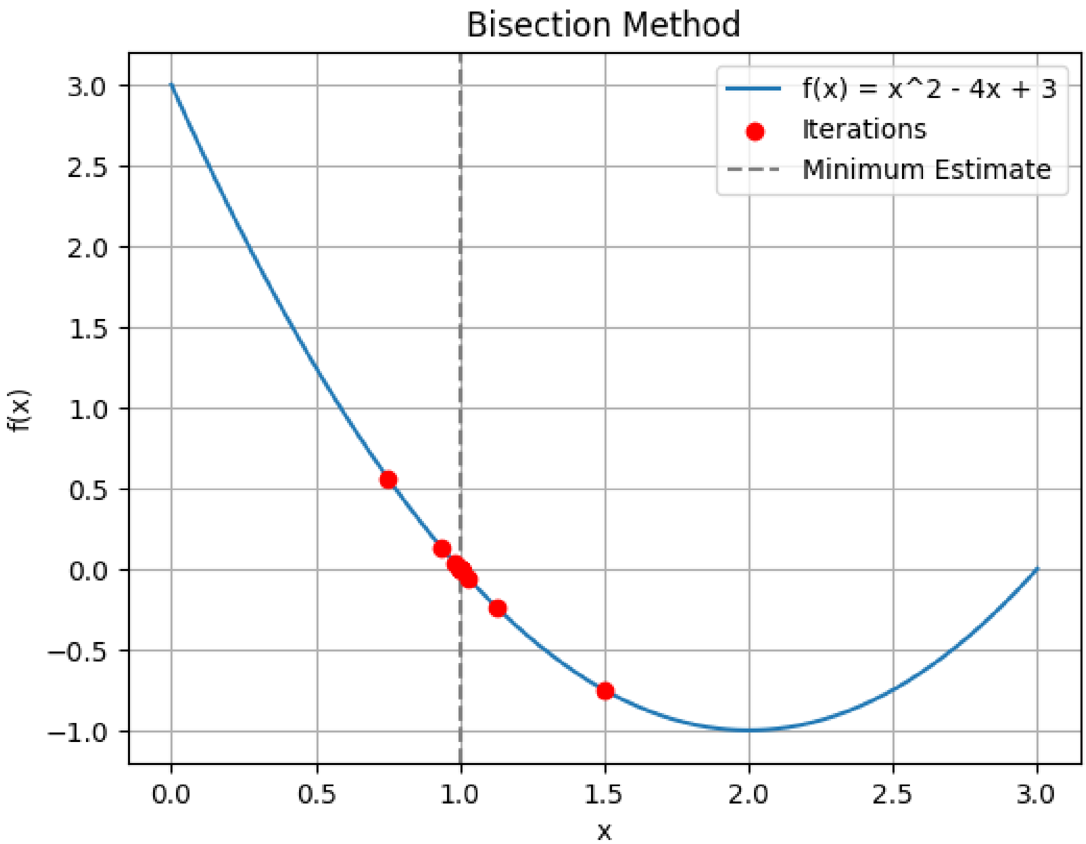
1 PROBLEM 0

Qualitatively reproduce the plots from the section "Minimizing functions of one variable"

[62]:
```
import numpy as np
import matplotlib.pyplot as plt

def bisection_method(f, a, b, tol=1e-6, max_iter=1000):
    """
    Bisection method to find the minimum of a function f(x) within the interval␣
 ↪[a, b].

    Parameters:
        f (function): The objective function.
        a (float): The left endpoint of the interval.
        b (float): The right endpoint of the interval.
        tol (float): Tolerance for the minimum value of f.
        max_iter (int): Maximum number of iterations.

    Returns:
        float: The estimated minimum value of f.
    """
    iter_count = 0
    x_values = []
    y_values = []

    while iter_count < max_iter:
        c = (a + b) / 2  # Compute the midpoint of the interval
        x_values.append(c)
        y_values.append(f(c))

        if abs(b - a) < tol:
            return c, x_values, y_values

        # Check the sign of f(a) * f(c)
        if f(a) * f(c) < 0:
            b = c  # The root lies in the left half
        else:
```

```
            a = c  # The root lies in the right half

        iter_count += 1

    return (a + b) / 2, x_values, y_values  # Return the final estimate

# Define the objective function
def f(x):
    return x**2 - 4*x + 3

# Define the interval [a, b]
a = 0
b = 3

# Call the bisection method
min_value, x_values, y_values = bisection_method(f, a, b)

print("Minimum value:", min_value)

# Plot the function and iterations
x = np.linspace(a, b, 1000)
plt.plot(x, f(x), label='f(x) = x^2 - 4x + 3')
plt.scatter(x_values, y_values, color='red', label='Iterations', zorder=5)
plt.axvline(x=min_value, linestyle='--', color='gray', label='Minimum Estimate')
plt.xlabel('x')
plt.ylabel('f(x)')
plt.title('Bisection Method')
plt.grid(True)
plt.legend()
plt.show()
```

```
Minimum value: 1.0000001192092896
```

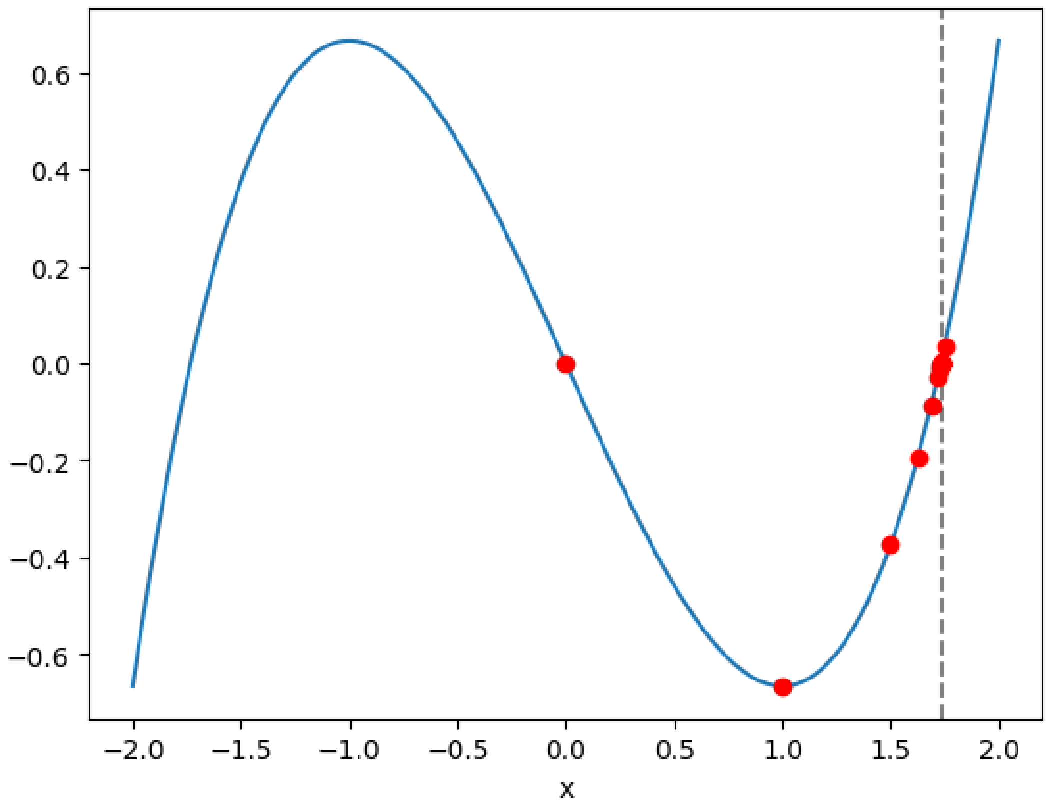


[63]:
```
import numpy as np
import matplotlib.pyplot as plt

def bisection_method(f, a, b, tol=1e-6, max_iter=1000):
    """
    Bisection method to find the minimum of a function f(x) within the interval␣
 ↪[a, b].

    Parameters:
        f (function): The objective function.
        a (float): The left endpoint of the interval.
        b (float): The right endpoint of the interval.
        tol (float): Tolerance for the minimum value of f.
        max_iter (int): Maximum number of iterations.

    Returns:
        float: The estimated minimum value of f.
    """
    iter_count = 0
```

```python
    x_values = []
    y_values = []

    while iter_count < max_iter:
        c = (a + b) / 2  # Compute the midpoint of the interval
        x_values.append(c)
        y_values.append(f(c))

        if abs(b - a) < tol:
            return c, x_values, y_values

        # Check the sign of f(a) * f(c)
        if f(a) * f(c) < 0:
            b = c  # The root lies in the left half
        else:
            a = c  # The root lies in the right half

        iter_count += 1

    return (a + b) / 2, x_values, y_values  # Return the final estimate

# Define the objective function
def f(x):
    return (x**3 / 3) - x

# Define the interval [a, b]
a = -2
b = 2

# Call the bisection method
min_value, x_values, y_values = bisection_method(f, a, b)

print("Minimum value:", min_value)

# Plot the function and iterations
x = np.linspace(a, b, 1000)
plt.plot(x, f(x), label='f(x) = x^3/3 - x')
plt.scatter(x_values, y_values, color='red', label='Iterations', zorder=5)
plt.axvline(x=min_value, linestyle='--', color='gray', label='Minimum Estimate')
plt.xlabel('x')
plt.ylabel
```

```
Minimum value: 1.7320504188537598
```

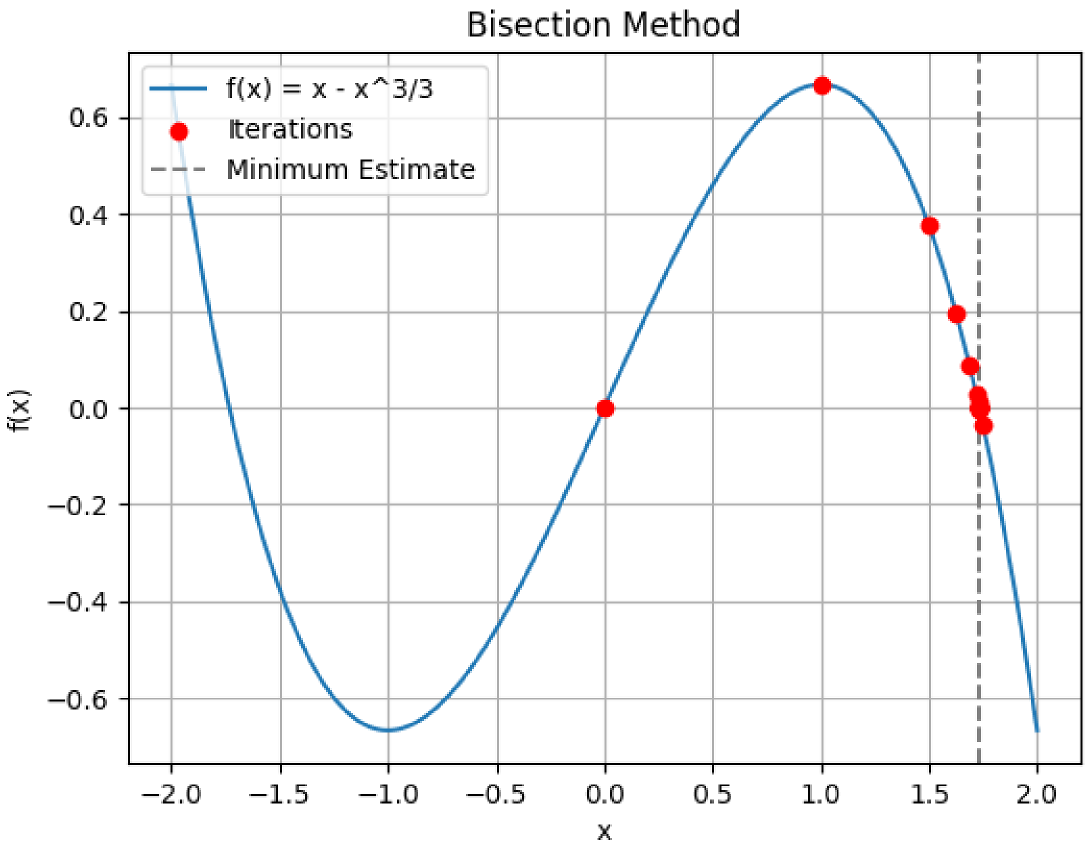
[63]:
```
<function matplotlib.pyplot.ylabel(ylabel: 'str', fontdict: 'dict[str, Any] |
None' = None, labelpad: 'float | None' = None, *, loc: "Literal['bottom',
'center', 'top'] | None" = None, **kwargs) -> 'Text'>
```



[64]:
```
import numpy as np
import matplotlib.pyplot as plt

def bisection_method(f, a, b, tol=1e-6, max_iter=1000):
    """
    Bisection method to find the minimum of a function f(x) within the interval␣
 ↪[a, b].

    Parameters:
        f (function): The objective function.
        a (float): The left endpoint of the interval.
        b (float): The right endpoint of the interval.
        tol (float): Tolerance for the minimum value of f.
        max_iter (int): Maximum number of iterations.

    Returns:
        float: The estimated minimum value of f.
    """
    iter_count = 0
    x_values = []
    y_values = []
```

```python
    while iter_count < max_iter:
        c = (a + b) / 2  # Compute the midpoint of the interval
        x_values.append(c)
        y_values.append(f(c))

        if abs(b - a) < tol:
            return c, x_values, y_values

        # Check the sign of f(a) * f(c)
        if f(a) * f(c) < 0:
            b = c  # The root lies in the left half
        else:
            a = c  # The root lies in the right half

        iter_count += 1

    return (a + b) / 2, x_values, y_values  # Return the final estimate

# Define the objective function
def f(x):
    return x - (x**3 / 3)

# Define the interval [a, b]
a = -2
b = 2

# Call the bisection method
min_value, x_values, y_values = bisection_method(f, a, b)

print("Minimum value:", min_value)

# Plot the function and iterations
x = np.linspace(a, b, 1000)
plt.plot(x, f(x), label='f(x) = x - x^3/3')
plt.scatter(x_values, y_values, color='red', label='Iterations', zorder=5)
plt.axvline(x=min_value, linestyle='--', color='gray', label='Minimum Estimate')
plt.xlabel('x')
plt.ylabel('f(x)')
plt.title('Bisection Method')
plt.grid(True)
plt.legend()
plt.show()
```

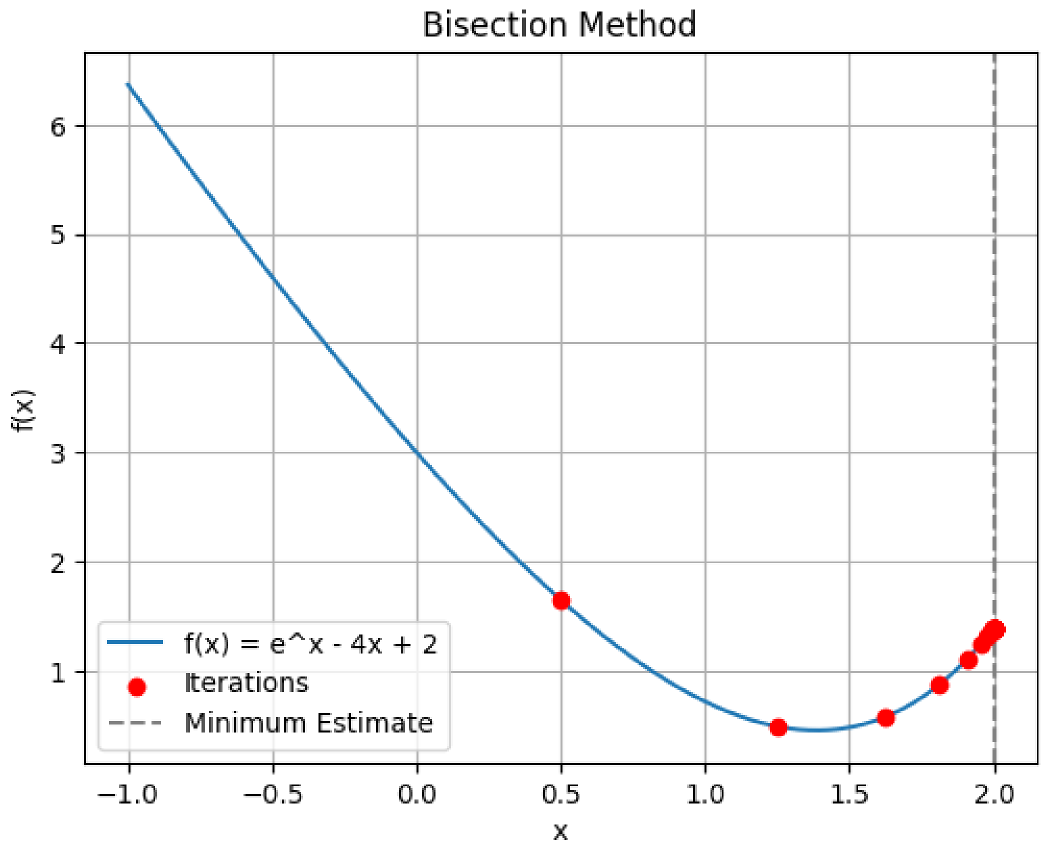
```
Minimum value: 1.7320504188537598
```



[67]:
```
import numpy as np
import matplotlib.pyplot as plt

def bisection_method(f, a, b, tol=1e-6, max_iter=1000):
    """
    Bisection method to find the minimum of a function f(x) within the interval␣
 ↪[a, b].

    Parameters:
        f (function): The objective function.
        a (float): The left endpoint of the interval.
        b (float): The right endpoint of the interval.
        tol (float): Tolerance for the minimum value of f.
        max_iter (int): Maximum number of iterations.

    Returns:
        float: The estimated minimum value of f.
    """
    iter_count = 0
```

```python
    x_values = []
    y_values = []

    while iter_count < max_iter:
        c = (a + b) / 2  # Compute the midpoint of the interval
        x_values.append(c)
        y_values.append(f(c))

        if abs(b - a) < tol:
            return c, x_values, y_values

        # Check the sign of f(a) * f(c)
        if f(a) * f(c) < 0:
            b = c  # The root lies in the left half
        else:
            a = c  # The root lies in the right half

        iter_count += 1

    return (a + b) / 2, x_values, y_values  # Return the final estimate

# Define the objective function
def f(x):
    return np.exp(x) - 4 * x + 2

# Define the interval [a, b]
a = -1
b = 2

# Call the bisection method
min_value, x_values, y_values = bisection_method(f, a, b)

print("Minimum value:", min_value)

# Plot the function and iterations
x = np.linspace(a, b, 1000)
plt.plot(x, f(x), label='f(x) = e^x - 4x + 2')
plt.scatter(x_values, y_values, color='red', label='Iterations', zorder=5)
plt.axvline(x=min_value, linestyle='--', color='gray', label='Minimum Estimate')
plt.xlabel('x')
plt.ylabel('f(x)')
plt.title('Bisection Method')
plt.grid(True)
plt.legend()
plt.show()
```

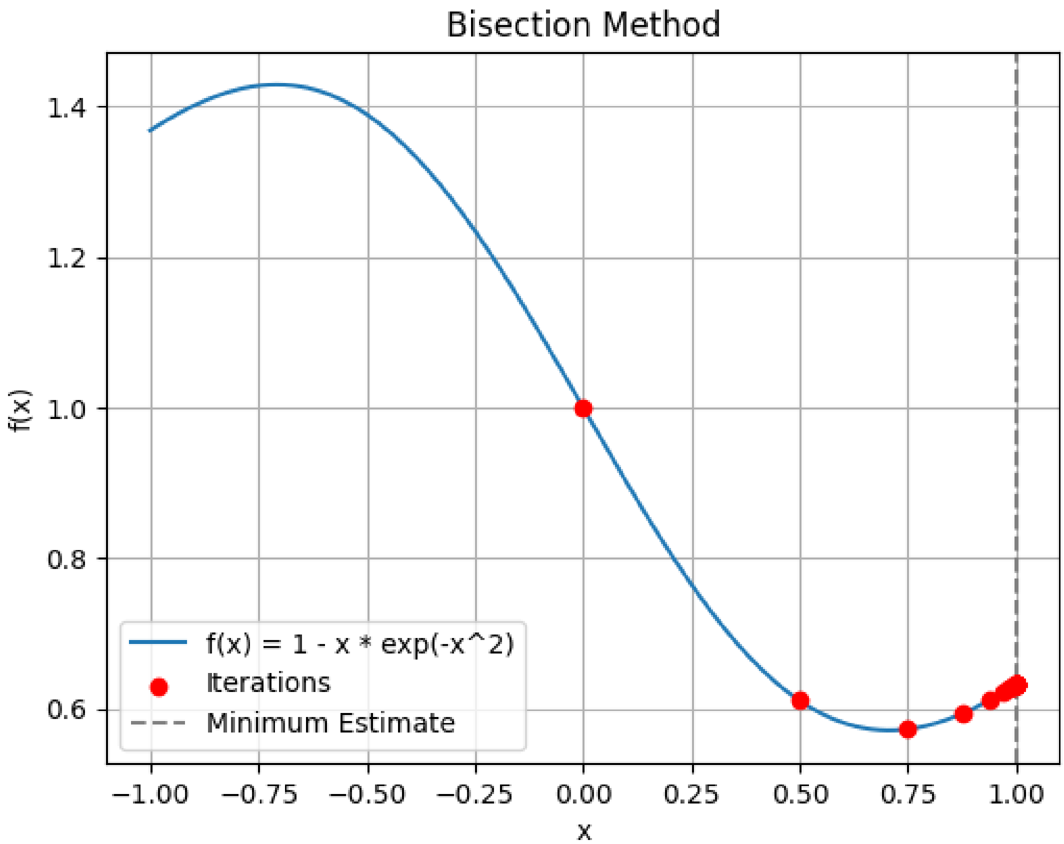
```
Minimum value: 1.9999996423721313
```




```
[68]: import numpy as np
import matplotlib.pyplot as plt

def bisection_method(f, a, b, tol=1e-5, max_iter=1000):
    """
    Bisection method to find the minimum of a function f(x) within the interval␣
↪[a, b].

    Parameters:
        f (function): The objective function.
        a (float): The left endpoint of the interval.
        b (float): The right endpoint of the interval.
        tol (float): Tolerance for the minimum value of f.
        max_iter (int): Maximum number of iterations.

    Returns:
        float: The estimated minimum value of f.
    """
    iter_count = 0
```

```
    x_values = []
    y_values = []

    while iter_count < max_iter:
        c = (a + b) / 2  # Compute the midpoint of the interval
        x_values.append(c)
        y_values.append(f(c))

        if abs(b - a) < tol:
            return c, x_values, y_values

        # Check the sign of f(a) * f(c)
        if f(a) * f(c) < 0:
            b = c  # The root lies in the left half
        else:
            a = c  # The root lies in the right half

        iter_count += 1

    return (a + b) / 2, x_values, y_values  # Return the final estimate

# Define the objective function
def f(x):
    return 1 - x * np.exp(-x**2)

# Define the interval [a, b]
a = -1
b = 1

# Call the bisection method
min_value, x_values, y_values = bisection_method(f, a, b)

print("Minimum value:", min_value)

# Plot the function and iterations
x = np.linspace(a, b, 1000)
plt.plot(x, f(x), label='f(x) = 1 - x * exp(-x^2)')
plt.scatter(x_values, y_values, color='red', label='Iterations', zorder=5)
plt.axvline(x=min_value, linestyle='--', color='gray', label='Minimum Estimate')
plt.xlabel('x')
plt.ylabel('f(x)')
plt.title('Bisection Method')
plt.grid(True)
plt.legend()
plt.show()
```

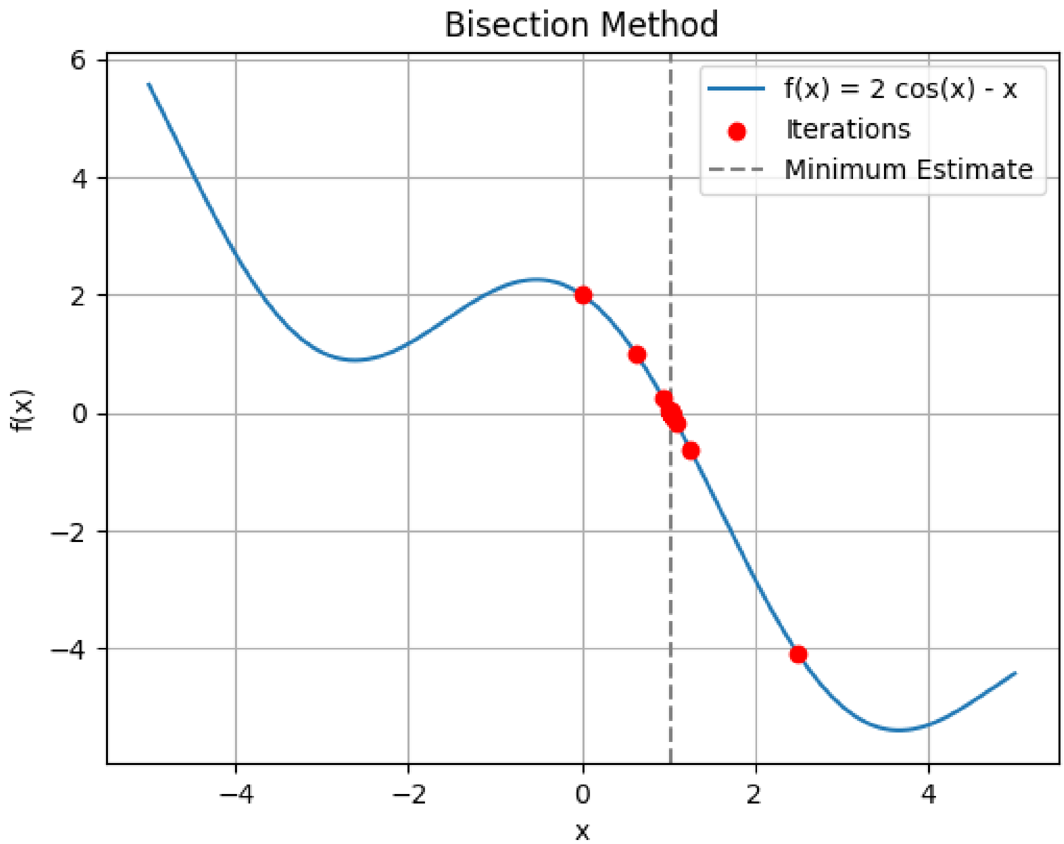
```
Minimum value: 0.9999961853027344
```



[66]:
```
import numpy as np
import matplotlib.pyplot as plt

def bisection_method(f, a, b, tol=1e-6, max_iter=1000):
    """
    Bisection method to find the minimum of a function f(x) within the interval␣
 ↪[a, b].

    Parameters:
        f (function): The objective function.
        a (float): The left endpoint of the interval.
        b (float): The right endpoint of the interval.
        tol (float): Tolerance for the minimum value of f.
        max_iter (int): Maximum number of iterations.

    Returns:
        float: The estimated minimum value of f.
    """
    iter_count = 0
```

```python
    x_values = []
    y_values = []

    while iter_count < max_iter:
        c = (a + b) / 2  # Compute the midpoint of the interval
        x_values.append(c)
        y_values.append(f(c))

        if abs(b - a) < tol:
            return c, x_values, y_values

        # Check the sign of f(a) * f(c)
        if f(a) * f(c) < 0:
            b = c  # The root lies in the left half
        else:
            a = c  # The root lies in the right half

        iter_count += 1

    return (a + b) / 2, x_values, y_values  # Return the final estimate

# Define the objective function
def f(x):
    return 2 * np.cos(x) - x

# Define the interval [a, b]
a = -5
b = 5

# Call the bisection method
min_value, x_values, y_values = bisection_method(f, a, b)

print("Minimum value:", min_value)

# Plot the function and iterations
x = np.linspace(a, b, 1000)
plt.plot(x, f(x), label='f(x) = 2 cos(x) - x')
plt.scatter(x_values, y_values, color='red', label='Iterations', zorder=5)
plt.axvline(x=min_value, linestyle='--', color='gray', label='Minimum Estimate')
plt.xlabel('x')
plt.ylabel('f(x)')
plt.title('Bisection Method')
plt.grid(True)
plt.legend()
plt.show()
```

```
Minimum value: 1.029866635799408
```



2 PROBLEM 3

[11]:
```
import numpy as np

# Load A and b from the text files
A = np.loadtxt("A.txt", delimiter=",")
b = np.loadtxt("b.txt", delimiter=",")

# Convergence criteria
max_iterations = 1000
tolerance = 1e-6

# Initialize x with zeros
x = np.zeros(A.shape[1])  # Shape of x should match the number of columns in A

# Iterate using steepest descent with fixed step size
for i in range(max_iterations):
    gradient = A.T @ (A @ x - b)
    step_size = 1 / np.linalg.norm(gradient)  # Fixed step size
```

```
    x_new = x - step_size * gradient

    # Convergence check
    if np.linalg.norm(x_new - x) < tolerance:
        print("Converged after", i+1, "iterations.")
        break

    x = x_new

# Result
print("Solution (x):", x)
```

```
Solution (x): [ 0.04371307  0.1147413   0.1434078   0.08242965 -0.22570372
-0.11806023
  0.01679239 -0.0998554  -0.04926812  0.2078271   0.08481768  0.25265853
 -0.0436471   0.13455386 -0.07831721 -0.02694556 -0.13841856  0.05137989
  0.13254668  0.15131881]
```

[12]:
```
from scipy.optimize import minimize_scalar

# Define the objective function
def objective_function(alpha, x, A, b):
    return 0.5 * np.linalg.norm(A @ (x - alpha * (A.T @ (A @ x - b))) - b)**2

# Initialize x with zeros
x = np.zeros(A.shape[1])

# Iterate using steepest descent with exact line search
for i in range(1000):  # assuming 1000 iterations
    # Minimize the objective function to find the optimal step size
    result = minimize_scalar(objective_function, args=(x, A, b))
    alpha_k = result.x

    # Update x
    gradient = A.T @ (A @ x - b)
    x = x - alpha_k * gradient

# Result
print("Exact Line Search (b):", x)
```

```
Exact Line Search (b): [-0.01574095  0.12101415  0.04518362  0.04764731
-0.1623104   0.00908704
 -0.0947885  -0.15907534  0.10031669  0.09054754  0.11128148  0.1171073
 -0.01742716  0.16248451  0.06777838  0.05863189 -0.06047807  0.07110943
  0.0825872   0.04114662]
```

[13]:
```
# Armijo's Rule parameters
c = 0.1  # constant between 0 and 1
```

```
alpha_0 = 1  # initial step size

# Initialize x with zeros
x = np.zeros(A.shape[1])
# Iterate using steepest descent with Armijo's rule
for i in range(1000):  # assuming 1000 iterations
    alpha_k = alpha_0
    gradient = A.T @ (A @ x - b)
    while 0.5 * np.linalg.norm(A @ (x - alpha_k * gradient) - b)**2 > 0.5 * np.
↪linalg.norm(A @ x - b)**2 - c * alpha_k * np.linalg.norm(gradient)**2:
        alpha_k *= 0.5  # decrease step size by half
    x = x - alpha_k * gradient

# Result
print("Armijo's Rule (c):", x)
```

```
Armijo's Rule (c): [-0.01574095  0.12101415  0.04518362  0.04764731 -0.1623104
0.00908704
 -0.0947885  -0.15907534  0.10031669  0.09054754  0.11128148  0.1171073
 -0.01742716  0.16248451  0.06777838  0.05863189 -0.06047807  0.07110943
  0.0825872   0.04114662]
```

[ ]:

```

```

[15]:

```
import numpy as np

# Load A and b from the text files
A = np.loadtxt("A.txt", delimiter=",")
b = np.loadtxt("b.txt", delimiter=",")

# Define the objective function and its gradient
def objective_function(x, A, b):
    return 0.5 * np.linalg.norm(A @ x - b)**2

def gradient(x, A, b):
    return A.T @ (A @ x - b)

# Initialize x with zeros
x = np.zeros(A.shape[1])



# Tolerances
epsilon = 1e-4
epsilon_0 = 1e-6
epsilon_1 = 1e-8

# Iteration counter
```

```
k = 0

# Perform steepest descent iterations
while True:
    # Compute gradient
    grad = gradient(x, A, b)

    # Stopping condition based on the norm of the gradient
    if np.linalg.norm(grad) <= epsilon:
        break

    # Stopping condition based on the distance between x* and x^k
    if np.linalg.norm(x - np.linalg.pinv(A) @ b) <= epsilon_0:
        break

    # Stopping condition based on the difference in function values
    if np.abs(objective_function(x, A, b) - objective_function(np.linalg.pinv(A)␣
 ↪@ b, A, b)) <= epsilon_1:
        break

    # Update x using steepest descent
    alpha = np.linalg.norm(grad)**2 / np.linalg.norm(A @ grad)**2
    x = x - alpha * grad

    # Increment iteration counter
    k += 1

# Result
print("Solution (x):", x)
```

```
Solution (x): [-0.01572824  0.12101376  0.04518455  0.04762921 -0.16230696
0.00908313
 -0.09476178 -0.15906747  0.10034552  0.09055638  0.11127076  0.11709538
 -0.01744392  0.1624567   0.06779185  0.05861927 -0.0604696   0.07111115
  0.08258749  0.0411533 ]
```

[ ]:

```
```

[18]:

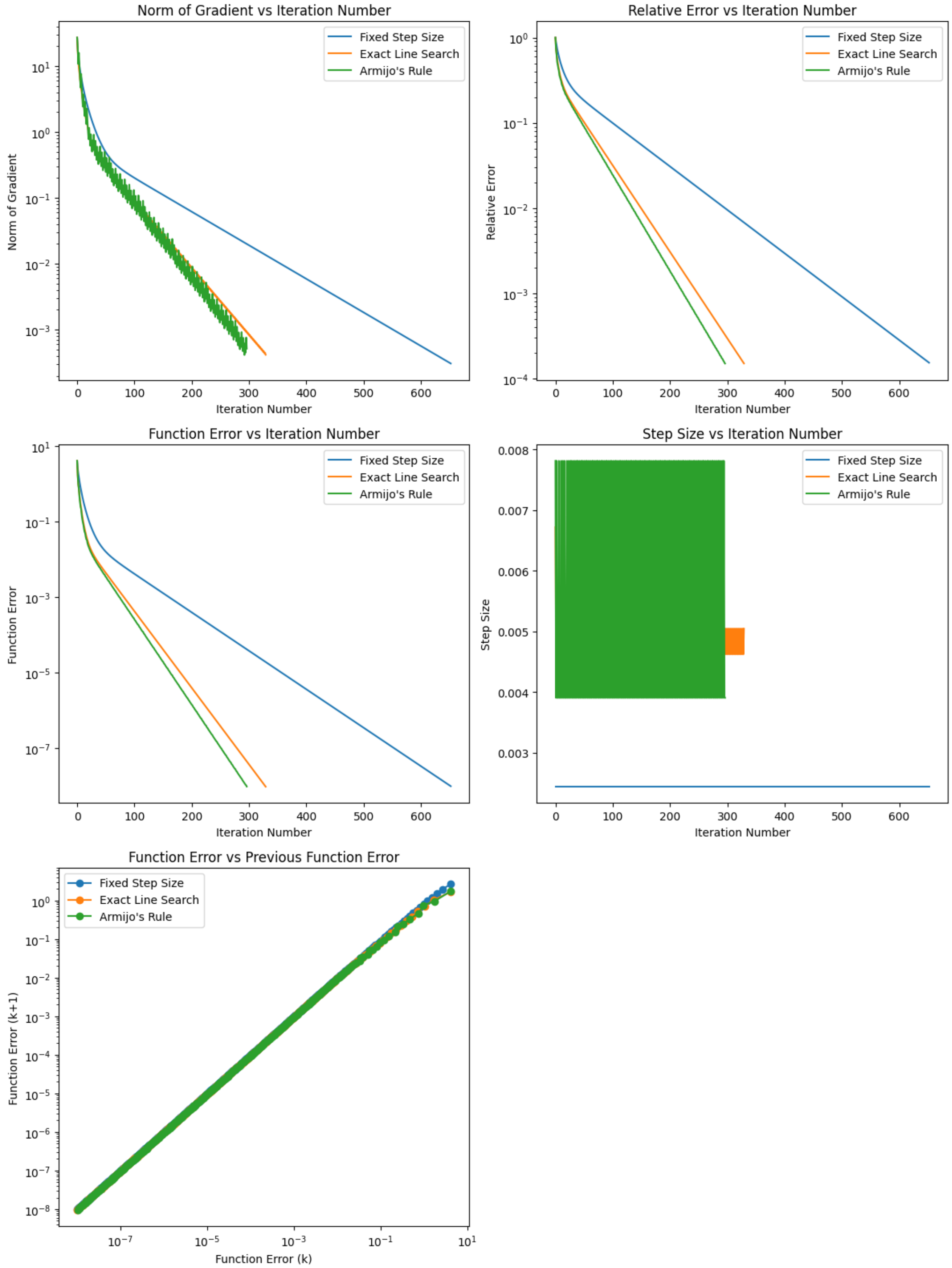
```
import numpy as np
import matplotlib.pyplot as plt

# Load A and b from the text files
A = np.loadtxt("A.txt", delimiter=",")
b = np.loadtxt("b.txt", delimiter=",")

# Define the objective function and its gradient
def objective_function(x, A, b):
```

```
    return 0.5 * np.linalg.norm(A @ x - b)**2

def gradient(x, A, b):
    return A.T @ (A @ x - b)

# Tolerances
epsilon = 1e-4
epsilon_0 = 1e-6
epsilon_1 = 1e-8

# Function to perform steepest descent iterations
def steepest_descent_method(A, b, alpha_type):
    # Initialize x with zeros
    x = np.zeros(A.shape[1])

    # Initialize lists to store data for plotting
    norm_gradient_list = []
    relative_error_list = []
    function_error_list = []
    step_size_list = []
    function_values = []

    # Iteration counter
    k = 0

    # Perform steepest descent iterations
    while True:
        # Compute gradient
        grad = gradient(x, A, b)

        # Compute step size based on alpha_type
        if alpha_type == 'fixed':
            alpha = 1 / np.linalg.eigvalsh(A.T @ A)[-1]  # Fixed step size
        elif alpha_type == 'exact':
            alpha = np.linalg.norm(grad)**2 / np.linalg.norm(A @ grad)**2  #␣
↪Exact line search
        elif alpha_type == 'armijo':
            alpha = 1  # Initial step size for Armijo's rule
            while objective_function(x - alpha * grad, A, b) >␣
↪objective_function(x, A, b) - 0.1 * alpha * np.linalg.norm(grad)**2:
                alpha *= 0.5  # Reduce step size by half until Armijo condition␣
↪is satisfied

        # Compute function value
        function_value = objective_function(x, A, b)

        # Compute relative error
```

```python
        relative_error = np.linalg.norm(x - np.linalg.pinv(A) @ b) / np.linalg.norm(np.linalg.pinv(A) @ b)

        # Compute function error
        function_error = np.abs(function_value - objective_function(np.linalg.pinv(A) @ b, A, b))

        # Append data to lists
        norm_gradient_list.append(np.linalg.norm(grad))
        relative_error_list.append(relative_error)
        function_error_list.append(function_error)
        step_size_list.append(alpha)
        function_values.append(function_value)

        # Stopping condition based on the norm of the gradient
        if np.linalg.norm(grad) <= epsilon:
            break

        # Stopping condition based on the distance between x* and x^k
        if np.linalg.norm(x - np.linalg.pinv(A) @ b) <= epsilon_0:
            break

        # Stopping condition based on the difference in function values
        if np.abs(function_value - objective_function(np.linalg.pinv(A) @ b, A, b)) <= epsilon_1:
            break

        # Update x using steepest descent
        x = x - alpha * grad

        # Increment iteration counter
        k += 1

    # Return computed data
    return norm_gradient_list, relative_error_list, function_error_list, step_size_list, function_values

# Perform steepest descent iterations for each strategy
fixed_norm_gradient, fixed_relative_error, fixed_function_error, fixed_step_size, fixed_function_values = steepest_descent_method(A, b, 'fixed')
exact_norm_gradient, exact_relative_error, exact_function_error, exact_step_size, exact_function_values = steepest_descent_method(A, b, 'exact')
armijo_norm_gradient, armijo_relative_error, armijo_function_error, armijo_step_size, armijo_function_values = steepest_descent_method(A, b, 'armijo')
```

```python
# Plotting
plt.figure(figsize=(12, 16))

# Plot norm of gradient vs iteration number for each strategy
plt.subplot(321)
plt.plot(fixed_norm_gradient, label='Fixed Step Size')
plt.plot(exact_norm_gradient, label='Exact Line Search')
plt.plot(armijo_norm_gradient, label="Armijo's Rule")
plt.yscale('log')
plt.xlabel('Iteration Number')
plt.ylabel('Norm of Gradient')
plt.title('Norm of Gradient vs Iteration Number')
plt.legend()

# Plot relative error vs iteration number for each strategy
plt.subplot(322)
plt.plot(fixed_relative_error, label='Fixed Step Size')
plt.plot(exact_relative_error, label='Exact Line Search')
plt.plot(armijo_relative_error, label="Armijo's Rule")
plt.yscale('log')
plt.xlabel('Iteration Number')
plt.ylabel('Relative Error')
plt.title('Relative Error vs Iteration Number')
plt.legend()

# Plot function error vs iteration number for each strategy
plt.subplot(323)
plt.plot(fixed_function_error, label='Fixed Step Size')
plt.plot(exact_function_error, label='Exact Line Search')
plt.plot(armijo_function_error, label="Armijo's Rule")
plt.yscale('log')
plt.xlabel('Iteration Number')
plt.ylabel('Function Error')
plt.title('Function Error vs Iteration Number')
plt.legend()

# Plot step size vs iteration number for each strategy
plt.subplot(324)
plt.plot(fixed_step_size, label='Fixed Step Size')
plt.plot(exact_step_size, label='Exact Line Search')
plt.plot(armijo_step_size, label="Armijo's Rule")
plt.xlabel('Iteration Number')
plt.ylabel('Step Size')
plt.title('Step Size vs Iteration Number')
plt.legend()

# Plot function error vs previous function error for each strategy
```

```
plt.subplot(325)
plt.plot(fixed_function_error[:-1], fixed_function_error[1:], 'o-', label='Fixed␣
↪Step Size')
plt.plot(exact_function_error[:-1], exact_function_error[1:], 'o-', label='Exact␣
↪Line Search')
plt.plot(armijo_function_error[:-1], armijo_function_error[1:], 'o-',␣
↪label="Armijo's Rule")
plt.xscale('log')
plt.yscale('log')
plt.xlabel('Function Error (k)')
plt.ylabel('Function Error (k+1)')
plt.title('Function Error vs Previous Function Error')
plt.legend()

plt.tight_layout()
plt.show()
```


Norm of Gradient vs Iteration Number
Norm of Gradient
Iteration Number
Fixed Step Size
Exact Line Search
Armijo's Rule
Relative Error vs Iteration Number
Relative Error
Iteration Number
Fixed Step Size
Exact Line Search
Armijo's Rule
Function Error vs Iteration Number
Function Error
Iteration Number
Fixed Step Size
Exact Line Search
Armijo's Rule
Step Size vs Iteration Number
Step Size
Iteration Number
Fixed Step Size
Exact Line Search
Armijo's Rule
Function Error vs Previous Function Error
Function Error (k+1)
Function Error (k)
Fixed Step Size
Exact Line Search
Armijo's Rule

3 PROBLEM 4

[27]:
```
# Define the objective function and its gradient
def objective_function(x):
    return (x[1] - x[0]**2)**2 + 0.01*(1 - x[0])**2

def gradient(x):
    return np.array([-4*x[0]*(x[1] - x[0]**2) - 0.02*(1 - x[0]), 2*(x[1] -␣
↪x[0]**2)])

# Armijo rule for step size
def armijo_step_size(x, grad, alpha_init=1, c=0.5):
    alpha = alpha_init
    while objective_function(x - alpha * grad) > objective_function(x) - c *␣
↪alpha * np.linalg.norm(grad)**2:
        alpha *= 0.5
    return alpha

# Steepest descent method
def steepest_descent(x0, step_size_method):
    x = np.array(x0)
    tol = 1e-5
    max_iter = 10000  # Increase maximum number of iterations
    iter_num = 0
    solution = None
    termination_reason = "Maximum iterations reached"

    while iter_num < max_iter:
        grad = gradient(x)
        if np.linalg.norm(grad) <= tol:
            solution = x
            termination_reason = "Gradient norm below tolerance"
            break

        step_size = step_size_method(x, grad)
        x = x - step_size * grad

        iter_num += 1

    return solution, iter_num, termination_reason

# Initial points
initial_points = [(-0.8, 0.8), (0, 0), (1.5, 1)]

# Run steepest descent method for each initial point
for point in initial_points:
    print(f"Initial Point: {point}")
```

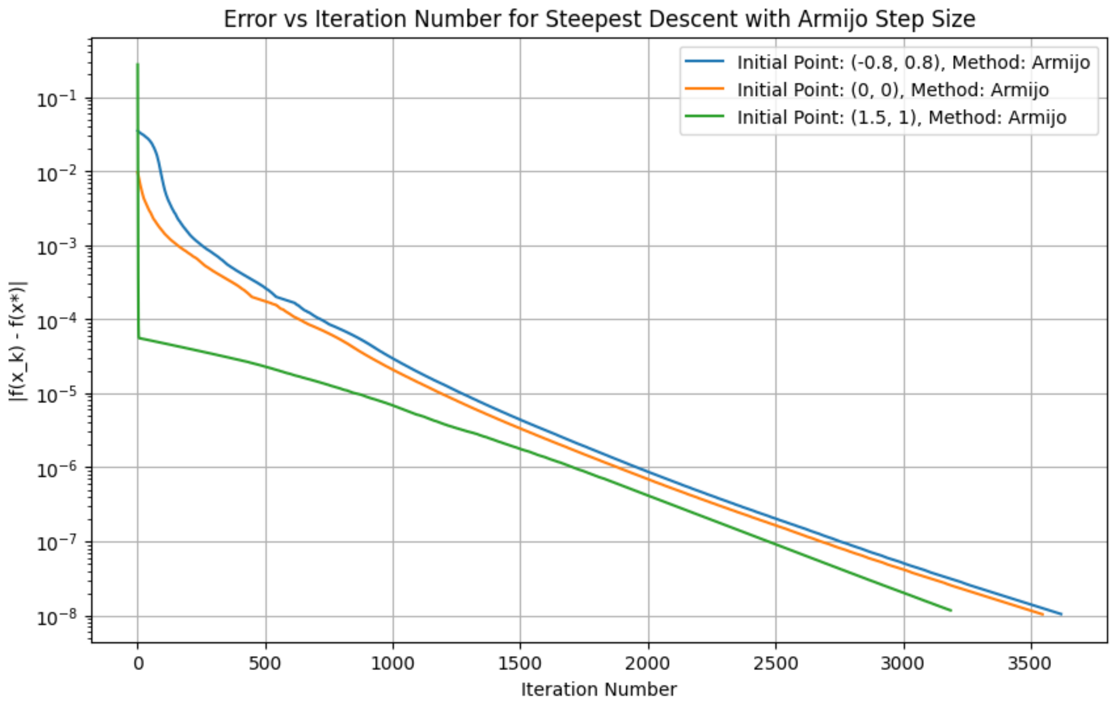
```
    solution, iterations, termination_reason = steepest_descent(point,␣
↪armijo_step_size)
    print(f"Solution: {solution}, Iterations: {iterations}, Termination Reason:␣
↪{termination_reason}")
    print()
```

```
Initial Point: (-0.8, 0.8)
Solution: [0.99897731 0.99795248], Iterations: 3619, Termination Reason:
Gradient norm below tolerance

Initial Point: (0, 0)
Solution: [0.9989817  0.99795945], Iterations: 3546, Termination Reason:
Gradient norm below tolerance

Initial Point: (1.5, 1)
Solution: [1.00108309 1.00217222], Iterations: 3186, Termination Reason:
Gradient norm below tolerance
```

[28]:

```
import numpy as np
import matplotlib.pyplot as plt

# Define the objective function and its gradient
def objective_function(x):
    return (x[1] - x[0]**2)**2 + 0.01*(1 - x[0])**2

def gradient(x):
    return np.array([-4*x[0]*(x[1] - x[0]**2) - 0.02*(1 - x[0]), 2*(x[1] -␣
↪x[0]**2)])

# Armijo rule for step size
def armijo_step_size(x, grad, alpha_init=1, c=0.5):
    alpha = alpha_init
    while objective_function(x - alpha * grad) > objective_function(x) - c *␣
↪alpha * np.linalg.norm(grad)**2:
        alpha *= 0.5
    return alpha

# Steepest descent method
def steepest_descent(x0, step_size_method):
    x = np.array(x0)
    tol = 1e-5
    max_iter = 10000
    iter_num = 0
    function_values = []
    solution = None
    termination_reason = "Maximum iterations reached"
```

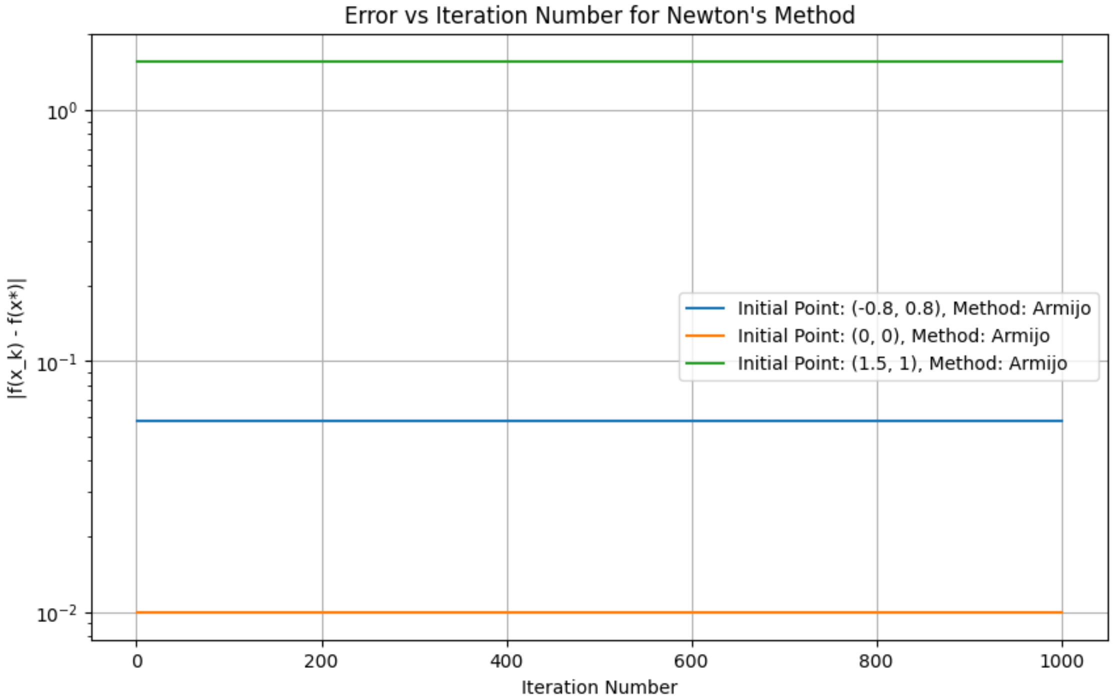
```
    while iter_num < max_iter:
        grad = gradient(x)
        if np.linalg.norm(grad) <= tol:
            solution = x
            termination_reason = "Gradient norm below tolerance"
            break

        step_size = step_size_method(x, grad)
        x = x - step_size * grad

        # Track function value at each iteration
        function_values.append(objective_function(x))

        iter_num += 1

    return function_values, iter_num, termination_reason

# Initial points
initial_points = [(-0.8, 0.8), (0, 0), (1.5, 1)]

# Step size methods
step_size_methods = {'Armijo': armijo_step_size}

# Run steepest descent method for each initial point and step size method
plt.figure(figsize=(10, 6))
for method_name, step_size_method in step_size_methods.items():
    for point in initial_points:
        function_values, iterations, termination_reason =␣
↪steepest_descent(point, step_size_method)
        plt.plot(range(1, iterations + 1), np.abs(np.array(function_values) -␣
↪objective_function([1, 1])), label=f"Initial Point: {point}, Method:␣
↪{method_name}")

plt.xlabel('Iteration Number')
plt.ylabel('|f(x_k) - f(x*)|')
plt.title('Error vs Iteration Number for Steepest Descent with Armijo Step Size')
plt.yscale('log')
plt.legend()
plt.grid(True)
plt.show()
```



[29]:
```
import numpy as np
import matplotlib.pyplot as plt

# Define the objective function, its gradient, and Hessian matrix
def objective_function(x):
    return (x[1] - x[0]**2)**2 + 0.01*(1 - x[0])**2

def gradient(x):
    return np.array([-4*x[0]*(x[1] - x[0]**2) - 0.02*(1 - x[0]), 2*(x[1] -
 ↪x[0]**2)])

def hessian(x):
    return np.array([[12*x[0]**2 - 4*x[1] + 0.02, -4*x[0]], [-4*x[0], 2]])

# Armijo rule for step size
def armijo_step_size(x, direction, alpha_init=1, c=0.5):
    alpha = alpha_init
    while objective_function(x - alpha * direction) > objective_function(x) - c
 ↪* alpha * np.dot(gradient(x), direction):
        alpha *= 0.5
    return alpha

# Newton's method
def newtons_method(x0, step_size_method):
```

```python
    x = np.array(x0)
    tol = 1e-5
    max_iter = 1000
    iter_num = 0
    function_values = []
    solution = None
    termination_reason = "Maximum iterations reached"

    while iter_num < max_iter:
        grad = gradient(x)
        hess = hessian(x)

        if np.linalg.norm(grad) <= tol:
            solution = x
            termination_reason = "Gradient norm below tolerance"
            break

        direction = np.linalg.solve(hess, -grad)
        step_size = step_size_method(x, direction)
        x = x + step_size * direction

        # Track function value at each iteration
        function_values.append(objective_function(x))

        iter_num += 1

    return function_values, iter_num, termination_reason

# Initial points
initial_points = [(-0.8, 0.8), (0, 0), (1.5, 1)]

# Step size methods
step_size_methods = {'Armijo': armijo_step_size}

# Run Newton's method for each initial point and step size method
plt.figure(figsize=(10, 6))
for method_name, step_size_method in step_size_methods.items():
    for point in initial_points:
        function_values, iterations, termination_reason = newtons_method(point, step_size_method)
        plt.plot(range(1, iterations + 1), np.abs(np.array(function_values) - objective_function([1, 1])), label=f"Initial Point: {point}, Method: {method_name}")

plt.xlabel('Iteration Number')
plt.ylabel('|f(x_k) - f(x*)|')
plt.title("Error vs Iteration Number for Newton's Method")
```

```
plt.yscale('log')
plt.legend()
plt.grid(True)
plt.show()
```



4 PROBLEM 5

[30]:

```
import numpy as np

# Define the model function
def model(t, a):
    return a[0] * np.sin(2 * np.pi * a[1] * t) + a[2] * np.sin(2 * np.pi * a[3]␣
 ↪* t)

# Define the objective function
def objective_function(a, t, y):
    f = model(t, a) - y
    return 0.5 * np.sum(f**2)

# Define the gradient of the objective function
def gradient_objective_function(a, t, y):
    f = model(t, a) - y
    df_da1 = np.sin(2 * np.pi * a[1] * t)
```

```
    df_da2 = a[0] * np.cos(2 * np.pi * a[1] * t) * 2 * np.pi * t
    df_da3 = np.sin(2 * np.pi * a[3] * t)
    df_da4 = a[2] * np.cos(2 * np.pi * a[3] * t) * 2 * np.pi * t
    grad = np.array([
        np.sum(f * df_da1),
        np.sum(f * df_da2),
        np.sum(f * df_da3),
        np.sum(f * df_da4)
    ])
    return grad

# Example usage:
# Suppose you have synthetic data t and y
t = np.linspace(0, 4, 10)
y = np.sin(2 * np.pi * 0.5 * t) + 0.5 * np.sin(2 * np.pi * 1.5 * t) + 0.15 * np.
 ↪random.randn(len(t))

# Initial guess for parameters
a_initial = np.array([1.0, 0.5, 0.5, 1.5])

# Compute the objective function and its gradient at the initial guess
print("Initial Objective Function Value:", objective_function(a_initial, t, y))
print("Initial Gradient of Objective Function:",␣
 ↪gradient_objective_function(a_initial, t, y))
```

```
Initial Objective Function Value: 0.1854013358014662
Initial Gradient of Objective Function: [ 0.19469883 -5.20683524 -0.34128542
3.59435594]
```

[31]:
```
import numpy as np

def compute_jacobian(t, a):
    # Compute the elements of the Jacobian matrix
    jacobian = np.zeros((len(t), len(a)))
    jacobian[:, 0] = np.sin(2 * np.pi * a[1] * t)  # Partial derivative with␣
 ↪respect to a1
    jacobian[:, 1] = a[0] * np.cos(2 * np.pi * a[1] * t) * 2 * np.pi * t  #␣
 ↪Partial derivative with respect to a2
    jacobian[:, 2] = np.sin(2 * np.pi * a[3] * t)  # Partial derivative with␣
 ↪respect to a3
    jacobian[:, 3] = a[2] * np.cos(2 * np.pi * a[3] * t) * 2 * np.pi * t  #␣
 ↪Partial derivative with respect to a4
    return jacobian

# Example usage:
# Suppose you have synthetic data t and y
t = np.linspace(0, 4, 10)
```

```
y = np.sin(2 * np.pi * 0.5 * t) + 0.5 * np.sin(2 * np.pi * 1.5 * t) + 0.15 * np.
 ↪random.randn(len(t))

# Initial guess for parameters
a_initial = np.array([1.0, 0.5, 0.5, 1.5])

# Compute the Jacobian matrix at the initial guess
jacobian = compute_jacobian(t, a_initial)
print("Jacobian Matrix:")
print(jacobian)
```

```
Jacobian Matrix:
[[ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]
 [ 9.84807753e-01  4.84917190e-01 -8.66025404e-01 -6.98131701e-01]
 [ 3.42020143e-01 -5.24823366e+00  8.66025404e-01 -1.39626340e+00]
 [-8.66025404e-01 -4.18879020e+00 -4.89858720e-16  4.18879020e+00]
 [-6.42787610e-01  8.55679856e+00 -8.66025404e-01 -2.79252680e+00]
 [ 6.42787610e-01  1.06959982e+01  8.66025404e-01 -3.49065850e+00]
 [ 8.66025404e-01 -8.37758041e+00 -9.79717439e-16  8.37758041e+00]
 [-3.42020143e-01 -1.83688178e+01 -8.66025404e-01 -4.88692191e+00]
 [-9.84807753e-01  3.87933752e+00  8.66025404e-01 -5.58505361e+00]
 [-4.89858720e-16  2.51327412e+01 -1.46957616e-15  1.25663706e+01]]
```

[32]:
```
def compute_hessian(t, a, y):
    # Compute the Jacobian matrix
    jacobian = compute_jacobian(t, a)

    # Compute the residuals
    f = model(t, a) - y

    # Initialize Hessian matrix
    hessian = np.zeros((len(a), len(a)))

    # Compute the Hessian matrix
    for i in range(len(t)):
        for j in range(len(a)):
            for k in range(len(a)):
                # Compute the second-order partial derivative of f_i with␣
 ↪respect to a_j and a_k
                second_order_partial_derivative = np.sin(2 * np.pi * a[1] *␣
 ↪t[i]) * np.sin(2 * np.pi * a[1] * t[i]) * (2 * np.pi * t[i]) ** 2 if j == 1␣
 ↪and k == 1 else 0
                second_order_partial_derivative += np.sin(2 * np.pi * a[3] *␣
 ↪t[i]) * np.sin(2 * np.pi * a[3] * t[i]) * (2 * np.pi * t[i]) ** 2 if j == 3␣
 ↪and k == 3 else 0
```

```
                # Add the contribution of the current data point to the Hessian␣
↪matrix
                hessian[j, k] += jacobian[i, j] * jacobian[i, k] + f[i] *␣
↪second_order_partial_derivative

    return hessian

# Example usage:
# Compute the Hessian matrix at the initial guess
hessian = compute_hessian(t, a_initial, y)
print("Hessian Matrix:")
print(hessian)
```

```
Hessian Matrix:
[[ 4.50000000e+00 -1.10789509e+00 -4.10782519e-15  9.18540242e+00]
 [-1.10789509e+00  1.37939029e+03  1.61550147e+01  2.41956408e+02]
 [-4.10782519e-15  1.61550147e+01  4.50000000e+00 -1.81379936e+00]
 [ 9.18540242e+00  2.41956408e+02 -1.81379936e+00  3.77608793e+02]]
```

[33]:
```
import numpy as np
from scipy.optimize import minimize

# Define the model function with four parameters
def model(t, a):
    return a[0] * np.sin(2 * np.pi * a[1] * t) + a[2] * np.sin(2 * np.pi * a[3]␣
↪* t)

# Generate synthetic data
t = np.linspace(0, 4, 10)
a_true = np.array([1.0, 0.5, 0.5, 1.5])
y_true = model(t, a_true)
y = y_true + 0.15 * np.random.randn(len(t))

# Define the objective function
def objective_function(a):
    return np.sum((model(t, a) - y)**2)

# Define the gradient of the objective function
def gradient_objective_function(a):
    f = model(t, a) - y
    df_da1 = np.sin(2 * np.pi * a[1] * t)
    df_da2 = a[0] * np.cos(2 * np.pi * a[1] * t) * 2 * np.pi * t
    df_da3 = np.sin(2 * np.pi * a[3] * t)
    df_da4 = a[2] * np.cos(2 * np.pi * a[3] * t) * 2 * np.pi * t
    grad = np.array([
        np.sum(f * df_da1),
        np.sum(f * df_da2),
```

```
        np.sum(f * df_da3),
        np.sum(f * df_da4)
    ])
    return grad

# Start optimization close to true parameter values
a_initial = a_true * 1.1

# Use scipy's minimize function with the BFGS method for line search
result = minimize(objective_function, a_initial,␣
 ↪jac=gradient_objective_function, method='BFGS')

# Get the optimized parameters
a_optimized = result.x

print("True Parameters:", a_true)
print("Optimized Parameters:", a_optimized)
```

```
True Parameters: [1.  0.5 0.5 1.5]
Optimized Parameters: [1.34574199 0.50804566 0.40251295 1.74195434]
```

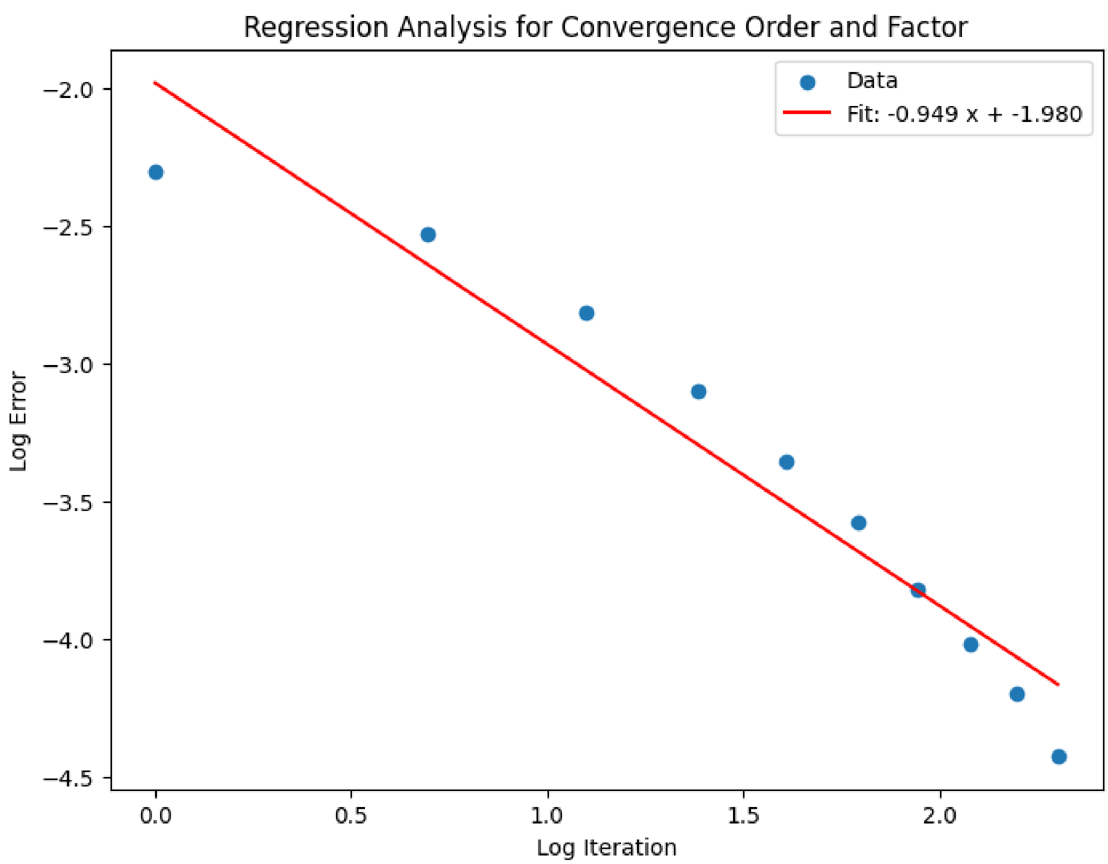
[35]:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit

# Assuming err_data is an array containing the errors at each iteration
# Assuming iter_data is an array containing the iteration numbers

# Define the function for the linear regression
def func(x, a, b):
    return a * x + b

# Example data for illustration purposes (replace with your actual data)
iter_data = np.arange(1, 11)  # Example iteration numbers
err_data = np.array([0.1, 0.08, 0.06, 0.045, 0.035, 0.028, 0.022, 0.018, 0.015,␣
 ↪0.012])  # Example errors

# Fit a linear curve to the logarithm of error vs. logarithm of iteration
popt, pcov = curve_fit(func, np.log(iter_data), np.log(err_data))

# Extract the parameters
convergence_order = -1 / popt[0]
convergence_factor = np.exp(-1 / popt[0])

# Plot the logarithm of error vs. logarithm of iteration
plt.figure(figsize=(8, 6))
plt.scatter(np.log(iter_data), np.log(err_data), label='Data')
```

```
plt.plot(np.log(iter_data), func(np.log(iter_data), *popt), 'r-', label='Fit: %.
 ↪3f x + %.3f' % tuple(popt))
plt.xlabel('Log Iteration')
plt.ylabel('Log Error')
plt.title('Regression Analysis for Convergence Order and Factor')
plt.legend()
plt.grid
```

[35]:

```
<function matplotlib.pyplot.grid(visible: 'bool | None' = None, which:
"Literal['major', 'minor', 'both']" = 'major', axis: "Literal['both', 'x', 'y']"
= 'both', **kwargs) -> 'None'>
```



[36]:

```
import numpy as np
from scipy.optimize import minimize

# Define the model function with four parameters
def model(t, a):
    return a[0] * np.sin(2 * np.pi * a[1] * t) + a[2] * np.sin(2 * np.pi * a[3]␣
 ↪* t)
```

```python
# Define the objective function (sum of squared residuals)
def objective_function(a):
    return np.sum((model(t, a) - y)**2)

# Define the gradient of the objective function
def gradient_objective_function(a):
    f = model(t, a) - y
    df_da1 = np.sin(2 * np.pi * a[1] * t)
    df_da2 = a[0] * np.cos(2 * np.pi * a[1] * t) * 2 * np.pi * t
    df_da3 = np.sin(2 * np.pi * a[3] * t)
    df_da4 = a[2] * np.cos(2 * np.pi * a[3] * t) * 2 * np.pi * t
    grad = np.array([
        np.sum(f * df_da1),
        np.sum(f * df_da2),
        np.sum(f * df_da3),
        np.sum(f * df_da4)
    ])
    return grad

# Generate synthetic data
t = np.linspace(0, 4, 10)
a_true = np.array([1.0, 0.5, 0.5, 1.5])
y_true = model(t, a_true)
y = y_true + 0.15 * np.random.randn(len(t))

# Start optimization process close to true parameter values
a_initial = a_true * 1.1

# Use scipy's minimize function with the Gauss-Newton method
result = minimize(objective_function, a_initial,␣
 ↪jac=gradient_objective_function, method='Newton-CG')

# Get the optimized parameters
a_optimized = result.x

print("True Parameters:", a_true)
print("Optimized Parameters (Gauss-Newton Method):", a_optimized)
```

```
True Parameters: [1.  0.5 0.5 1.5]
Optimized Parameters (Gauss-Newton Method): [1.29657023 0.50049788 0.25299009
1.74950213]
```

5 PROBLEM 6

[37]:
```
import numpy as np

# Define the function f(x)
def f(x, a):
    m, n = len(a), len(x)
    g = -np.sum(np.log(1 - a @ x))  # Sum of log terms involving a
    h = np.sum(np.log(1 + x)) - np.sum(np.log(1 - x))  # Sum of log terms␣
↪involving x
    return g + h

# Define the function to compute the Hessian matrix
def compute_hessian(x, a):
    n = len(x)
    hessian = np.zeros((n, n))
    for i in range(n):
        for j in range(n):
            # Compute the second derivative using the chain rule
            hessian[i, j] = -np.sum(a[:, i] * a[:, j] / (1 - a @ x)**2) if i ==␣
↪j else 0
    return hessian

# Example values for x and A
x = np.array([0.5, -0.3, 0.7])  # Example vector x
A = np.array([[0.1, 0.2, 0.3], [0.4, 0.5, 0.6]])  # Example matrix A

# Compute the Hessian matrix for the given x and A
hessian = compute_hessian(x, A)
print("Hessian matrix:")
print(hessian)
```

```
Hessian matrix:
[[-0.58522272  0.          0.        ]
 [ 0.         -0.95249644  0.        ]
 [ 0.          0.         -1.42221987]]
```

[41]:
```
import numpy as np

# Define the function f(x)
def f(x, a):
    m, n = len(a), len(x)
    g = -np.sum(np.log(1 - a @ x.clip(max=0.999)))  # Clip x to ensure a_j^Tx < 1
    h = np.sum(np.log(1 + x.clip(max=0.999))) - np.sum(np.log(1 - x.clip(max=0.
↪999)))  # Clip x to ensure |x_i| < 1
    return g + h

# Define the gradient of f(x)
```

```python
def gradient_f(x, a):
    m, n = len(a), len(x)
    grad_g = np.sum(a / (1 - a @ x.clip(max=0.999))[:, None], axis=0)  # Clip x␣
↪to ensure a_j^Tx < 1
    grad_h = 1 / (1 + x.clip(max=0.999)) - 1 / (1 - x.clip(max=0.999))  # Clip x␣
↪to ensure |x_i| < 1
    return grad_g + grad_h

# Define the Armijo rule to determine the step size
def armijo_rule(x, a, grad, alpha=0.1, beta=0.5):
    t = 1
    while f(x - t * grad, a) - f(x, a) > -alpha * t * np.sum(grad**2):
        t *= beta
    return t

# Define the steepest descent optimization method
def steepest_descent(x0, a, tol=1e-3, max_iter=1000):
    x = x0.copy()
    iter_count = 0
    while np.linalg.norm(gradient_f(x, a)) > tol and iter_count < max_iter:
        grad = gradient_f(x, a)
        alpha = armijo_rule(x, a, grad)
        x -= alpha * grad
        iter_count += 1
    return x, iter_count

# Set the random seed for reproducibility
np.random.seed(1)

# Generate random matrix A (use the same seed command to ensure reproducibility)
m, n = 20, 10  # Size of A matrix
A = np.random.randn(m, n)

# Initialize the starting point
x0 = np.zeros(n)

# Perform optimization using steepest descent with Armijo rule
optimal_x, num_iterations = steepest_descent(x0, A)
print("Optimal Solution (x):", optimal_x)
print("Number of Iterations:", num_iterations)
```

```
Optimal Solution (x): [ 9.98591466e+05  9.98637376e+05  9.82351353e+05
-9.41254545e+01
 -5.61043763e+01  9.98754143e+05 -6.91634402e+01  9.97636690e+05
 -7.44413725e+01  9.11324681e+05]
Number of Iterations: 1000
```

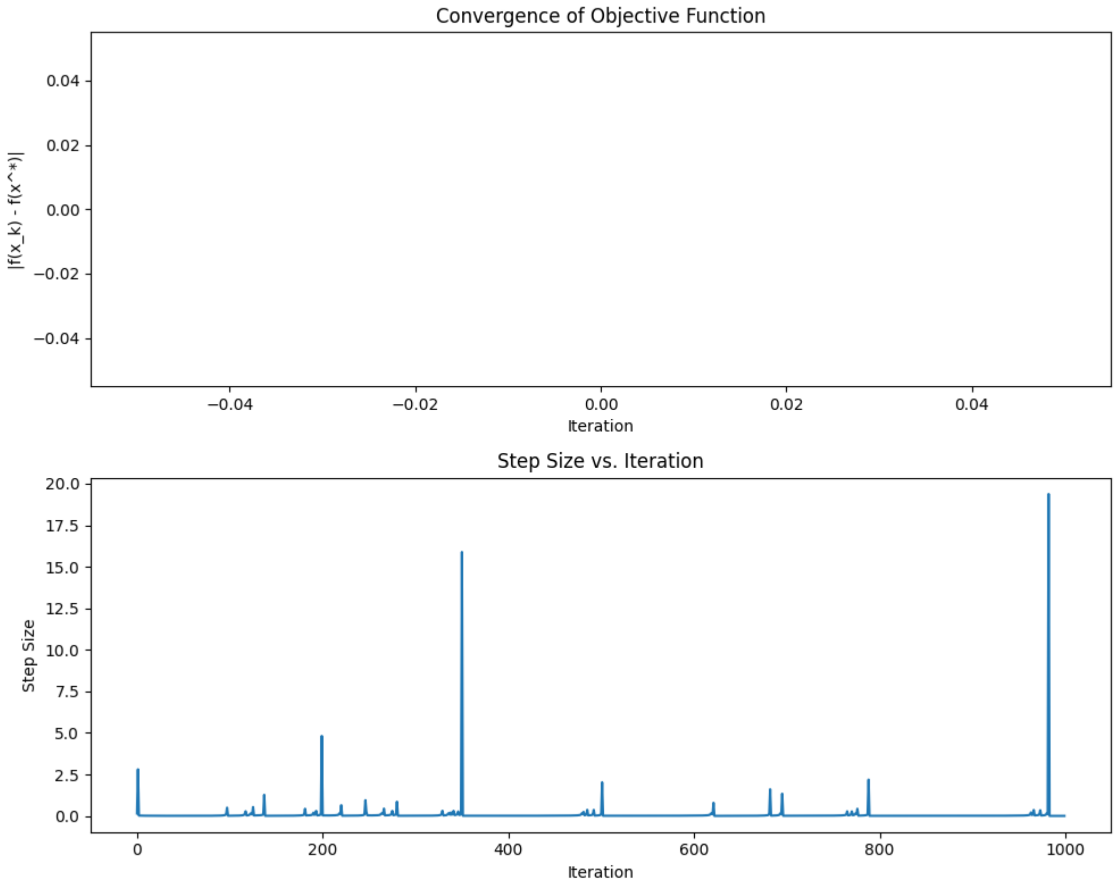
```
C:\Users\Akshay\AppData\Local\Temp\ipykernel_26600\2276783028.py:6:
```

```
RuntimeWarning: invalid value encountered in log
  g = -np.sum(np.log(1 - a @ x.clip(max=0.999)))  # Clip x to ensure a_j^Tx < 1
C:\Users\Akshay\AppData\Local\Temp\ipykernel_26600\2276783028.py:7:
RuntimeWarning: invalid value encountered in log
  h = np.sum(np.log(1 + x.clip(max=0.999))) - np.sum(np.log(1 -
x.clip(max=0.999)))  # Clip x to ensure |x_i| < 1
```

[49]:
```
import numpy as np
import matplotlib.pyplot as plt

# Set random seed
np.random.seed(1)

# Define problem parameters
m = 5  # Number of rows in A
n = 3  # Number of columns in A
max_iter = 1000  # Maximum number of iterations
learning_rate = 0.01  # Learning rate for gradient descent

# Generate random matrix A
A = np.random.randn(m, n)

def f(x):
    return -np.sum(np.log(1 / (1 - np.clip(A.dot(x), -1e15, 1e15) + epsilon))) -
 ↪np.sum(np.log(1 + np.exp(-x))) - np.sum(np.log(1 - np.exp(-x)))




def gradient_f(x):
    return A.T.dot(1 / (1 - A.dot(x))) + 1 / (1 + x) - 1 / (1 - x)

# Initialize x randomly
x = np.random.randn(n)

# Initialize arrays to track convergence
error = np.zeros(max_iter)
step_size = np.zeros(max_iter)

# Perform gradient descent
for iter in range(max_iter):
    # Compute gradient
    grad = gradient_f(x)

    # Update x using gradient descent step
    x_new = x - learning_rate * grad
```

```python
    # Compute error and step size
    error[iter] = np.abs(f(x_new) - f(x))
    step_size[iter] = np.linalg.norm(x_new - x)

    # Update x
    x = x_new

    # Check for convergence
    if error[iter] < 1e-6:
        break

# Plot error and step size vs. iteration number
plt.figure(figsize=(10, 8))

plt.subplot(2, 1, 1)
plt.plot(np.arange(iter + 1), error[:iter + 1])
plt.xlabel('Iteration')
plt.ylabel('|f(x_k) - f(x^*)|')
plt.title('Convergence of Objective Function')

plt.subplot(2, 1, 2)
plt.plot(np.arange(iter + 1), step_size[:iter + 1])
plt.xlabel('Iteration')
plt.ylabel('Step Size')
plt.title('Step Size vs. Iteration')

plt.tight_layout()
plt.show()
```

```
C:\Users\Akshay\AppData\Local\Temp\ipykernel_26600\3952485296.py:17:
RuntimeWarning: invalid value encountered in log
  return -np.sum(np.log(1 / (1 - np.clip(A.dot(x), -1e15, 1e15) + epsilon))) -
np.sum(np.log(1 + np.exp(-x))) - np.sum(np.log(1 - np.exp(-x)))
```



[55]:
```
# Define the Newton's method optimization
def gradient_f(x, a):
    m, n = len(a), len(x)
    x= x.reshape(-1, 1)
    a = a.reshape(1, -1)
    print(a.shape,x.clip(max=0.999).shape)

    grad_g = np.sum(a / (1 - a.T @ x.clip(max=0.999).T)[:, None], axis=0)   #␣
↪Gradient of the log terms involving a
    grad_h = (1 - np.tanh(x)**2) / (1 + np.tanh(x)) - (1 - np.tanh(x)**2) / (2 -␣
↪np.tanh(x))  # Gradient of the log terms involving x
    return grad_g + grad_h
def newtons_method_with_tracking(x0, a, tol=1e-3, max_iter=1000):
    x = x0.copy()
    iter_count = 0
    errors = []
    step_sizes = []
    while np.linalg.norm(gradient_f(x, a)) > tol and iter_count < max_iter:
        grad = gradient_f(x, a)
```

```
        hess = hessian_f(x, a)
        delta_x = np.linalg.solve(hess, -grad)
        alpha = armijo_rule(x, a, delta_x)
        x += alpha * delta_x
        error = np.abs(f(x, a) - f(optimal_x, a))  # Compute error
        errors.append(error)
        step_sizes.append(alpha)
        iter_count += 1
    return x, errors, step_sizes

# Perform optimization using Newton's method with Armijo rule and track error␣
↪and step size
optimal_x_newton, errors_newton, step_sizes_newton =␣
↪newtons_method_with_tracking(x0, A)

# Plot error vs. iteration number for both methods
plt.figure(figsize=(10, 5))
plt.plot(range(1, len(errors) + 1), errors, marker='o', linestyle='-',␣
↪label='Steepest Descent')
plt.plot(range(1, len(errors_newton) + 1), errors_newton, marker='o',␣
↪linestyle='-', label="Newton's Method")
plt.xlabel('Iteration Number')
plt.ylabel('Error |f(x_k) - f(x^*)|')
plt.title('Error vs. Iteration Number')
plt.grid(True)
plt.legend()
plt.show()

# Plot step size vs. iteration number for both methods
plt.figure(figsize=(10, 5))
plt.plot(range(1, len(step_sizes) + 1), step_sizes, marker='o', linestyle='-',␣
↪label='Steepest Descent')
plt.plot(range(1, len(step_sizes_newton) + 1), step_sizes_newton, marker='o',␣
↪linestyle='-', label="Newton's Method")
plt.xlabel('Iteration Number')
plt.ylabel('Step Size')
plt.title('Step Size vs. Iteration Number')
plt.grid(True)
plt.legend()
plt.show()
```

```
(1, 15) (10, 1)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[55], line 29
     26     return x, errors, step_sizes
```

```
     28 # Perform optimization using Newton's method with Armijo rule and track␣
 ↪error and step size
---> 29 optimal_x_newton, errors_newton, step_sizes_newton =␣
 ↪newtons_method_with_tracking(x0, A)
     31 # Plot error vs. iteration number for both methods
     32 plt.figure(figsize=(10, 5))

Cell In[55], line 16, in newtons_method_with_tracking(x0, a, tol, max_iter)
     14 errors = []
     15 step_sizes = []
---> 16 while np.linalg.norm(gradient_f(x, a)) > tol and iter_count < max_iter:
     17     grad = gradient_f(x, a)
     18     hess = hessian_f(x, a)

Cell In[55], line 8, in gradient_f(x, a)
      5 a = a.reshape(1, -1)
      6 print(a.shape,x.clip(max=0.999).shape)
----> 8 grad_g = np.sum(a / (1 - a.T @ x.clip(max=0.999).T)[:, None], axis=0)  #␣
 ↪Gradient of the log terms involving a
      9 grad_h = (1 - np.tanh(x)**2) / (1 + np.tanh(x)) - (1 - np.tanh(x)**2) / (2␣
 ↪- np.tanh(x))  # Gradient of the log terms involving x
     10 return grad_g + grad_h

ValueError: operands could not be broadcast together with shapes (1,15) (15,1,10)
```

```
[ ]:
```