Numerical Optimization Practical File

Ques list:-

1. WAP for finding optimal solution using Line Search method.
2. WAP to solve a LPP graphically.
3. WAP to compute the gradient and Hessian of the function $f(x) = 100(x2 - x1\ 2\ )\ 2 + (1 - x1)\ 2$
4. WAP to find Global Optimal Solution of a function $f(x) = -10Cos(\pi x - 2.2) + (x + 1.5)x$ algebraically
5. WAP to find Global Optimal Solution of a function $f(x) = -10Cos(\pi x - 2.2) + (x + 1.5)x$ graphically
6. WAP to solve constraint optimization problem.
7. WAP to implement Newton Method
8. WAP to implement gradient descent method
9. WAP to implement steepest descent method

10. WAP to implement Taylor polynomial

Ques 1

```
#Ques 1
import numpy as np
from scipy.optimize import minimize_scalar

def objective_function(x):
    return -(x**2 + 2*x + 1)

print()

min_result = minimize_scalar(objective_function, method='golden', bracket=(0, 1))
max_result = minimize_scalar(objective_function, method='golden', bracket=(0, 1))

optimal_max_solution = max_result.x
optimal_min_solution = min_result.x


print("Maximization Result - Optimal Solution :", optimal_max_solution)
print("Minimization Result - Optimal Solution :", optimal_min_solution)
```

output

```
Maximization Result - Optimal Solution : 1.340780795141782e+154
Minimization Result - Optimal Solution : 1.340780795141782e+154
```

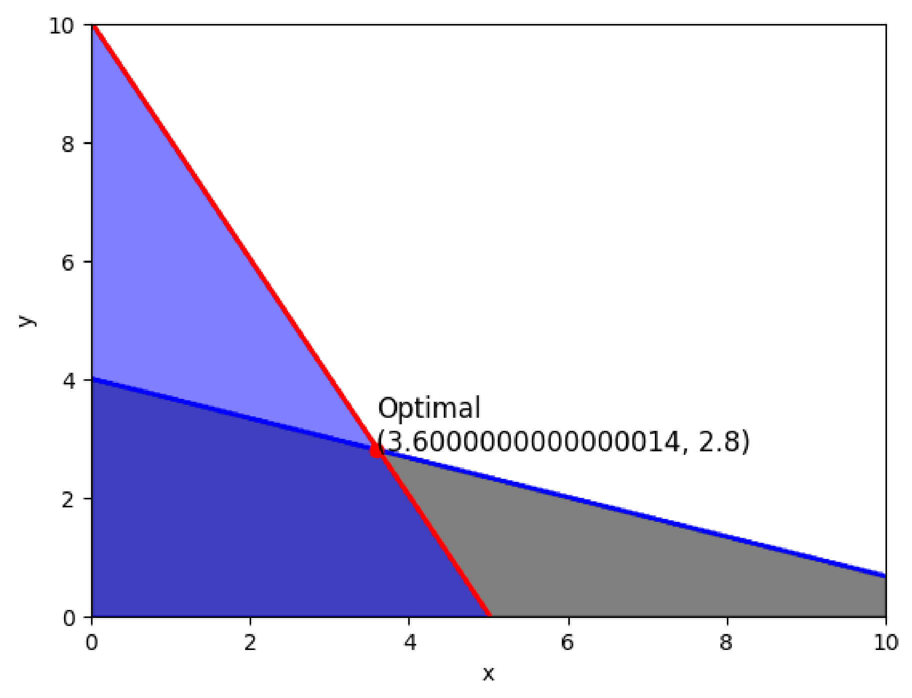
Ques 2

```
#Ques 2
import numpy as np
from scipy.optimize import linprog
import matplotlib.pyplot as plt

# Define the coefficients of the objective function to maximize
c = [-5, -3]  # Coefficients of the function to maximize (e.g., Z = 5x + 3y)

# Define the coefficients of inequality constraints in the form of Ax <= b
A = np.array([[2, 1],  # Coefficients of the first constraint
              [1, 3]])  # Coefficients of the second constraint
b = [10, 12]  # Right-hand side values of the constraints

# Define the bounds for variables (x and y) - e.g., x >= 0, y >= 0
x_bounds = (0, None)  # x >= 0
y_bounds = (0, None)  # y >= 0

# Solve the linear programming problem
result = linprog(c, A_ub=A, b_ub=b, bounds=[x_bounds, y_bounds], method='highs')

if result.success:
    print("Optimal solution found:")
```

```
    print("x =", result.x[0])
    print("y =", result.x[1])
    print("Optimal value (Z) =", -result.fun)
    x = result.x[0]
    y = result.x[1]
else:
    print("No optimal solution found.")

# Create a meshgrid for plotting the feasible region
x_range = np.linspace(0, 10, 400)
y_range = np.linspace(0, 10, 400)
X, Y = np.meshgrid(x_range, y_range)

# Define the constraints in the form Ax <= b
constraint1 = 2 * X + Y
constraint2 = X + 3 * Y

# Plot the feasible region
plt.figure()
plt.contour(X, Y, (constraint1 <= 10) , colors='red', levels=[0], linewidths=2)
plt.contour(X, Y, (constraint2 <= 12), colors='blue', levels=[0], linewidths=2)

plt.fill_between(x_range, 0, (12 - x_range) / 3, where=(12 - x_range) / 3 >= 0,
color='black', alpha=0.5)
plt.fill_between(y_range, 0, (10-2*x_range), where=(10 - 2 * y_range) >= 0,
color='blue', alpha=0.5)

# Plot the optimal solution point
plt.plot(x, y, 'ro')
plt.text(x, y, f'Optimal\n({x}, {y})', fontsize=12, ha='left')

# Set axis labels and limits
plt.xlabel('x')
plt.ylabel('y')
plt.xlim(0, 10)
plt.ylim(0, 10)

# Show the plot
plt.show()
```

output

```
Optimal solution found:
x = 3.6000000000000014
y = 2.8
Optimal value (Z) = 26.400000000000006
```



Ques 3

```
#Ques 3
import sympy as sp

x1, x2 = sp.symbols('x1 x2')

f = 100 * (x2**2 - x1**2)**2 + (1 - x1)**2

gradient = [sp.diff(f, x1), sp.diff(f, x2)]

hessian = sp.hessian(f, (x1, x2))

print("Function:")
print(f)
print("\nGradient:")
print(gradient)
print("\nHessian:")
print(hessian)
```

Output

```
Function:
(1 - x1)**2 + 100*(-x1**2 + x2**2)**2

Gradient:
[-400*x1*(-x1**2 + x2**2) + 2*x1 - 2, 400*x2*(-x1**2 + x2**2)]

Hessian:
Matrix([[1200*x1**2 - 400*x2**2 + 2, -800*x1*x2], [-800*x1*x2, -400*x1**2 + 1200*x2**2]])
```

Ques 4

```
#QUES 4
import numpy as np
from scipy.optimize import minimize

# Define the function to be minimized
def objective_function(x):
    return -10 * np.cos(np.pi * x - 2.2) + (x + 1.5) * x

# Initial guess for the minimum
initial_guess = [0.0]

# Use the minimize function from SciPy to find the minimum
result = minimize(objective_function, initial_guess, method='Nelder-Mead')

# Extract the optimal solution
optimal_solution = result.x[0]
optimal_value = result.fun

print("Optimal Solution:", optimal_solution)
print("Optimal Value:", optimal_value)
```

Output

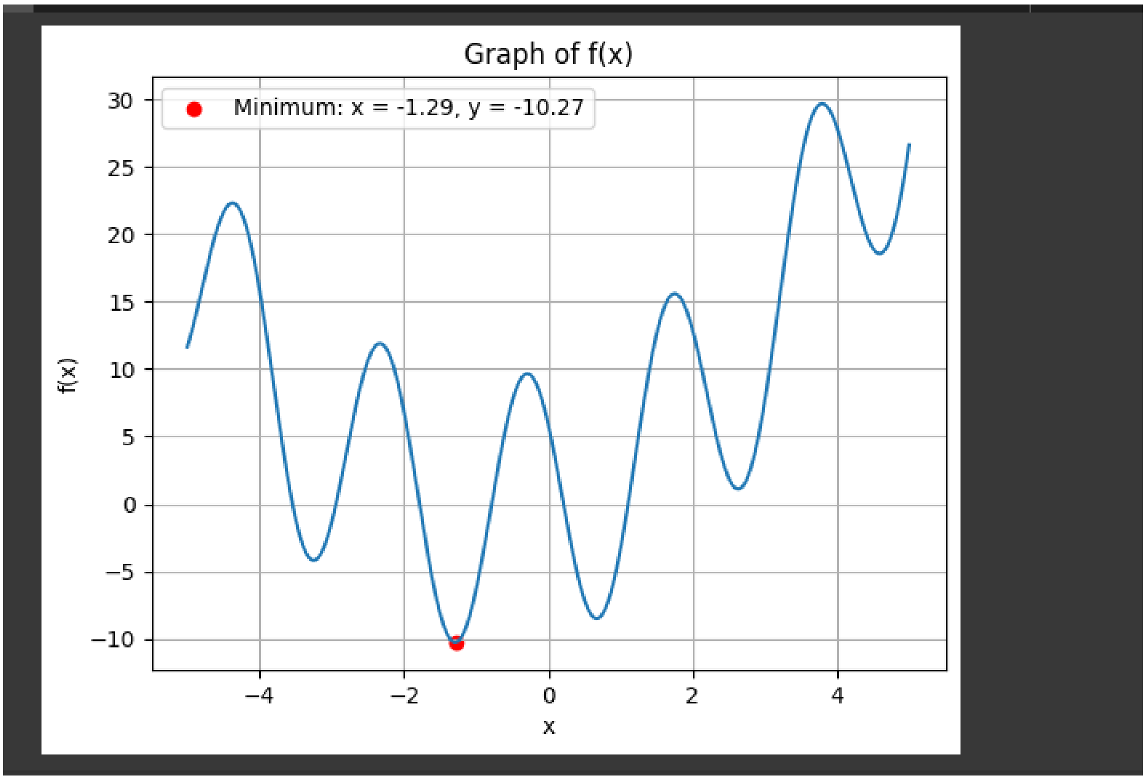
```
Optimal Solution: 0.6714375000000008
Optimal Value: -8.500986423547845
```

Ques 5

```
#Ques 5
import numpy as np
import matplotlib.pyplot as plt

# Define the function
def f(x):
    return -10 * np.cos(np.pi * x - 2.2) + (x + 1.5) * x

# Generate x values
x = np.linspace(-5, 5, 1000)

# Calculate corresponding y values
y = f(x)

# Find the x value where the function is minimized
optimal_x = x[np.argmin(y)]
optimal_y = min(y)

# Plot the function
```

```
plt.plot(x, y)
plt.scatter(optimal_x, optimal_y, color='red', label=f'Minimum: x = 
{optimal_x:.2f}, y = {optimal_y:.2f}')
plt.xlabel('x')
plt.ylabel('f(x)')
plt.legend()
plt.grid(True)
plt.title('Graph of f(x)')
plt.show()
```

Output



Ques 6

```
#Ques 6
from scipy.optimize import minimize

def objective_function(x):
    return x[0]**2 + x[1]**2

def constraint_function(x):
    return x[0] + x[1] - 1

initial_guess = [0.0, 0.0]

constraints = ({'type': 'ineq', 'fun': constraint_function})

result = minimize(objective_function, initial_guess, constraints=constraints)

print("Optimal solution:", result.x)
```

```
print("Optimal value:", result.fun)
```

Output

```
Optimal solution: [0.5 0.5]
Optimal value: 0.5000000000000002
```

Ques 7

```
#Ques 7
def f(x):
    return x**2 - 4*x + 4

def f_prime(x):
    return 2*x - 4

def f_double_prime(x):
    return 2

x0 = 3

tolerance = 1e-6

max_iterations = 100

for i in range(max_iterations):
    x1 = x0 - f_prime(x0) / f_double_prime(x0)

    if abs(x1 - x0) < tolerance:
        break

    x0 = x1

print()
print(f"Minimum Value: {f(x0)}")
print(f"Location: {x0}")
```

Output

```
Minimum Value: 0.0
Location: 2.0
```

Ques 8

```
#Ques 8
import sympy as sp

def gradient_descent(initial_x, learning_rate, num_iterations):
```

```python
    x = sp.symbols('x')

    f = x**2 + 5*x + 6

    df = sp.diff(f, x)

    f_prime = sp.lambdify(x, df, 'numpy')

    for _ in range(num_iterations):
        gradient = f_prime(initial_x)
        initial_x = initial_x - learning_rate * gradient

    return initial_x, f.subs(x, initial_x)

initial_x = 0
learning_rate = 0.1
num_iterations = 1000

min_x, min_value = gradient_descent(initial_x, learning_rate, num_iterations)

print(f"Minimum value is {min_value} at x = {min_x}")
```

Output

```python
#Ques 8
import sympy as sp

def gradient_descent(initial_x, learning_rate, num_iterations):
    x = sp.symbols('x')

    f = x**2 + 5*x + 6

    df = sp.diff(f, x)

    f_prime = sp.lambdify(x, df, 'numpy')

    for _ in range(num_iterations):
        gradient = f_prime(initial_x)
        initial_x = initial_x - learning_rate * gradient

    return initial_x, f.subs(x, initial_x)

initial_x = 0
learning_rate = 0.1
num_iterations = 1000

min_x, min_value = gradient_descent(initial_x, learning_rate, num_iterations)

print(f"Minimum value is {min_value} at x = {min_x}")
```

OUTPUT

```
Minimum value is -0.2500000000000000 at x = -2.499999999999999
```

Ques 9

```
import numpy as np

def objective_function(x):
    return x**2 + 4*x + 4

def gradient(x):
    return 2*x + 4

def steepest_descent(initial_guess, learning_rate, tolerance):
    x = initial_guess

    while True:
        grad_value = gradient(x)

        if np.linalg.norm(grad_value) < tolerance:
            break  # Convergence criteria met

        x = x - learning_rate * grad_value

    return x, objective_function(x)

# Initial guess, learning rate, and tolerance
initial_guess = np.array([0.0])
learning_rate = 0.1
tolerance = 1e-6

result = steepest_descent(initial_guess, learning_rate, tolerance)

print("Optimal solution: x =", result[0])
```

Output

```
Optimal solution: x = [-1.99999959]
```

Ques 10

```
import math

def factorial(n):
    if n == 0:
```

```python
        return 1
    return n * factorial(n - 1)

def taylor_coefficient(func, point, degree, derivative_order):
    return func(point) / factorial(derivative_order)

def taylor_polynomial(func, degree, point, var='x'):
    x = point
    taylor_poly = sum(taylor_coefficient(func, point, i, i) * (x - point)**i for i
in range(degree + 1))
    return taylor_poly

def sin_function(x):
    return math.cos(x)

degree_of_polynomial = 3
center_point = 0

taylor_poly = taylor_polynomial(sin_function, degree_of_polynomial, center_point)

print("Taylor Polynomial:", taylor_poly)
```

Output

```
Taylor Polynomial: 1.0
```