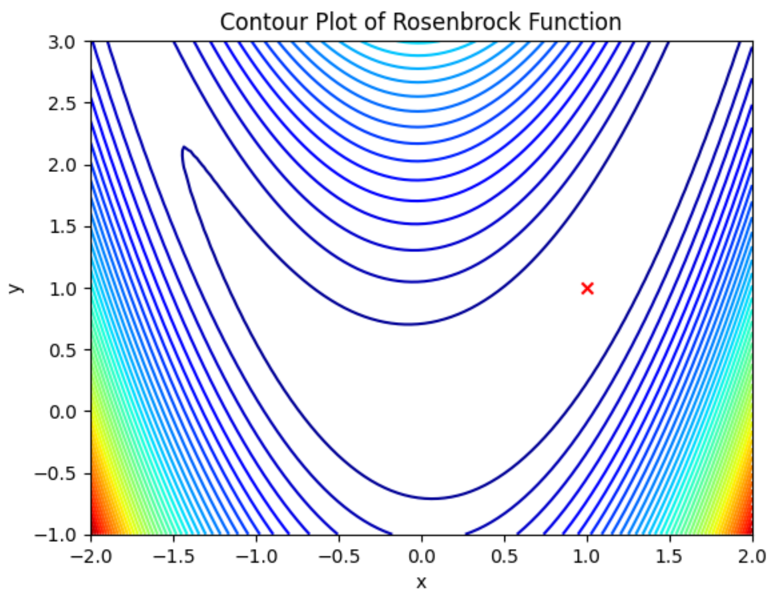
Numerical Optimization - Assignment 1

David Alvear(187594) - Nouf Farhoud(189731)

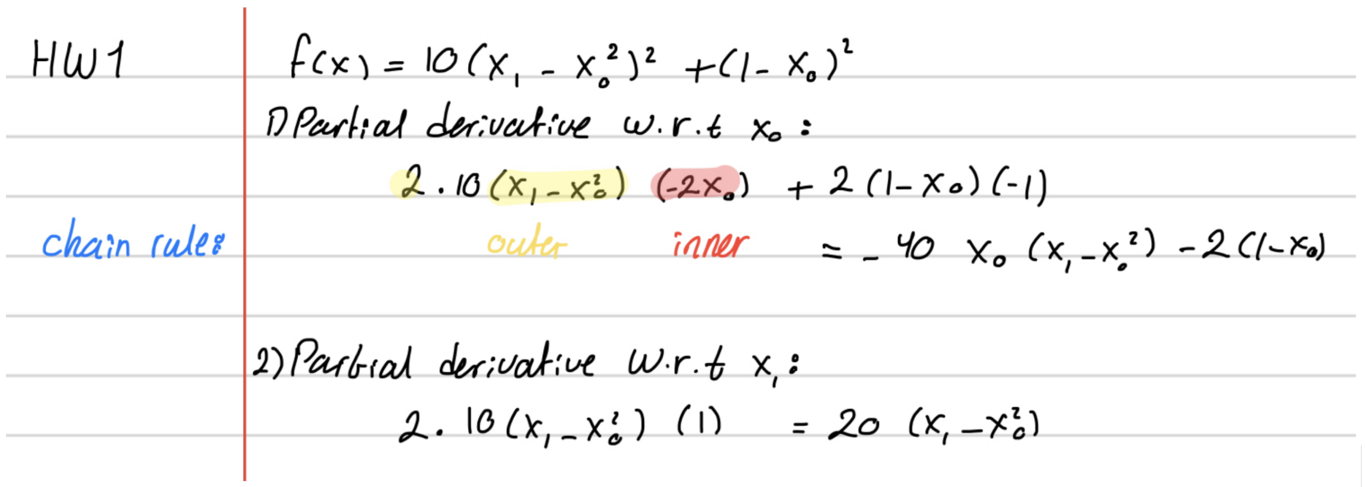
1. Function Plot.

The Rosenbrock function, also referred to as the Valley or Banana function, is a popular test problem for gradient-based optimization algorithms. The function is unimodal, and the global minimum lies in a narrow, parabolic valley. However, even though this valley is easy to find, convergence to the minimum is difficult. The goal of optimization algorithms is to find the input x that minimizes the Rosenbrock function's output f(x).

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

def rosenbrock(x):
    return 10 * (x[1] - x[0]**2)**2 + (1 - x[0])**2

# Generate a grid of points
x = np.linspace(-2, 2, 100)
y = np.linspace(-1, 3, 100)
X, Y = np.meshgrid(x, y)  # The meshgrid function from NumPy is used to
# create a grid of points based on x and y arrays. The resulting X and Y
# arrays are 2D arrays where each element represents a point on the grid.

# Compute the function values at the grid points
Z = rosenbrock([X, Y])  # The computed results are stored in the Z array,
# which will have the same shape as X and Y. Each element Z[i, j]
# corresponds to the value of the Rosenbrock function evaluated at the
# coordinates (X[i, j], Y[i, j])

# Plotting the contour lines
plt.figure()
plt.contour(X, Y, Z, levels=50, cmap='jet')
# levels=50: the number of contour lines to be drawn
# cmap='jet': provides a range of colors from blue to red,
# helps visualize the variation in function values
plt.scatter(1, 1, color='red', marker='x')  # Plotting the minimum point
plt.xlabel('x')
plt.ylabel('y')
plt.title('Contour Plot of Rosenbrock Function')
plt.show()
```



2. Gradient Computation.



HW1

$f(x) = 10(x_1 - x_0^2)^2 + (1-x_0)^2$

1) Partial derivative w.r.t x_0:

chain rule: $$\underbrace{2 \cdot 10(x_1 - x_0^2)}_{\text{outer}}\ \underbrace{(-2x_0)}_{\text{inner}} + 2(1-x_0)(-1) = -40\, x_0 (x_1 - x_0^2) - 2(1-x_0)$$

2) Partial derivative w.r.t x_1:

$$2 \cdot 10(x_1 - x_0^2)\ (1) = 20\,(x_1 - x_0^2)$$

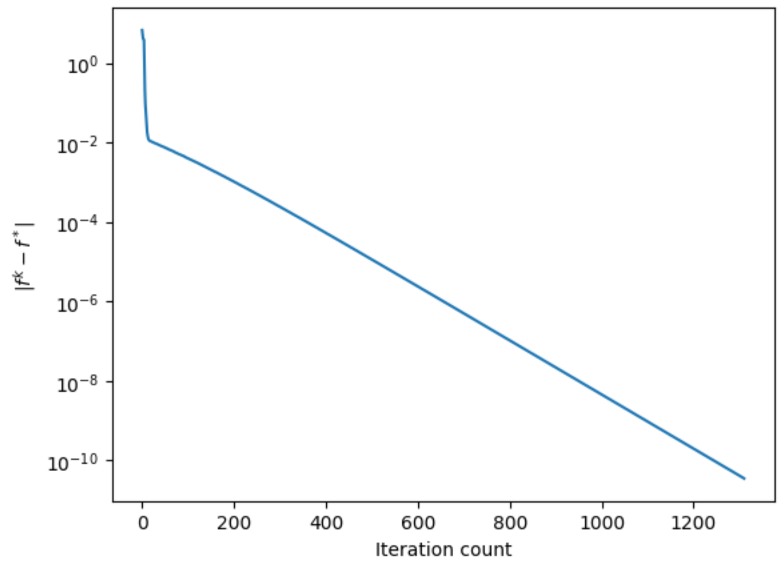
```
def rosen_grad(x):
    # Computing the gradient of the Rosenbrock function at point x
    grad = np.zeros_like(x)  # Initialize the gradient array
    # Partial derivative with respect to x[0]:
    grad[0] = -40 * x[0] * (x[1] - x[0]**2) - 2 * (1 - x[0])
    # Partial derivative with respect to x[1]:
    grad[1] = 20 * (x[1] - x[0]**2)

    return grad
```

3. Backtracking Line Search.

```
import numpy as np

###################################### Functions ######################################

def backtracking_linesearch(f, xk, pk, gk, alpha=0.1, beta=0.8):
  t = 1
  while (f(xk + t*pk) > f(xk) + alpha * t * gk @ pk):
    t *= beta
  return t

def steepest_descent_bt(fun, grad, x0, tol=1e-5):
  x = x0
  history = np.array([x0])
  while (np.linalg.norm(grad(x)) > tol):
    # find the direction of descent
    p = - grad(x)
    # Use line search to estimate the step
    t = backtracking_linesearch(fun, x, p, grad(x))
    # update x
    x += t*p
    history = np.vstack((history, x))
  return x, history

#######################################################################################


# Run

x0 = np.array([-1.2, 1])
x_star, history = steepest_descent_bt(rosenbrock, rosen_grad, x0)
```

4. Convergence Behavior.

```
nsteps = history.shape[0]
fhist = np.zeros(nsteps)
for i in range(nsteps):
  fhist[i] = rosenbrock(history[i,:])

plt.figure('convergence')
plt.semilogy(np.arange(0, nsteps), np.absolute(fhist))
plt.xlabel('Iteration count')
plt.ylabel(r'$|f^k - f^*|$')
plt.show()
```



Comments

We can see that the steepest descent algorithm performed the reduction in the error on each iteration approaching f^k to f^* optimum. We can check that the major reduction occurs in the first 100 iterations achieving an approximate error of 10-2, and then the reduction of the error exhibits a linear behaviour after 100 iterations approximately. The steep decline in the first iteration is due to the error between the initial guess and the objective point, the gradients become smaller as the algorithm approaches to the valley and that's why after 100 iterations the reduction in error per iteration is decreased.