Python Practical

DISCIPLINE SPECIFIC CORE COURSE – 09:

Numerical

Optimization

Semester - III

Course: B.Sc. (H) Computer Science

Introduction to Linear Programming

- Linear Programming (LP), also known as linear optimization is a mathematical programming technique to obtain the best result or outcome, like maximum profit or least cost, in a mathematical model whose requirements are represented by linear relationships.

- Linear programming is a special case of mathematical programming, also known as mathematical optimization.

Continued…

- Generally, an organization or a company has mainly two objectives, the first one is minimization and the other is maximization.
- Minimization means to minimize the total cost of production while maximization means to maximize their profit.
- So with the help of linear programming graphical method, we can find the optimum solution.

Basic terminologies of LP

- **Objective Function:** The main aim of the problem, either to maximize of to minimize, is the objective function of linear programming. In the problem shown below, **Z (to minimize)** is the **objective function**.

- **Decision Variables:** The variables used to decide the output as decision variables. They are the unknowns of the mathematical programming model. In the next slide, we are to determine the value of **x** and **y** in order to minimize **Z**. Here, **x** and **y** are the decision variables.

Continued…

- **Constraints:** These are the restrictions on the decision variables. The limitations on the decision variables given under subject to the constraints in the below problem are the constraints of the Linear programming.
- **Non – negativity restrictions:** In linear programming, the values for decision variables are always greater than or equal to 0.
- **Note:** For a problem to be a linear programming problem, the objective function, constraints, and the non – negativity restrictions must be linear.

Example 1: Consider the following problem:

Minimize : Z = 3x + 5y

Subject to the constraints:

2x + 3y >= 12

-x + y <= 3

x >= 4

y <= 3

x, y >= 0

Solving the previous example of LPP in Python Using PuLP Library

- PuLP is one of many libraries in Python ecosystem for solving optimization problems.
- We can install PuLp as follows:
  - Code for installing using Jupyter notebook:

    ```
    import sys !{sys.executable} -m pip install pulp
    ```

  - Code for installing using Command prompt:
    C:\Users\mypr\AppData\Local\Programs\Python\Python310\Scripts>pip install pulp

PYTHON CODE: To solve the aforementioned LPP

```
# import the library pulp as p
import pulp as p

# Create a LP Minimization problem
Lp_prob = p.LpProblem('Problem', p.LpMinimize)

# Create problem Variables
x = p.LpVariable("x", lowBound = 0)    # Create a variable x >= 0
y = p.LpVariable("y", lowBound = 0)    # Create a variable y >= 0

# Objective Function
Lp_prob += 3 * x + 5 * y

# Constraints:
Lp_prob += 2 * x + 3 * y >= 12
Lp_prob += -x + y <= 3
Lp_prob += x >= 4
Lp_prob += y <= 3

# Display the problem
print(Lp_prob)

status = Lp_prob.solve()    # Solver
print(p.LpStatus[status])    # The solution status

# Printing the final solution
print(p.value(x), p.value(y), p.value(Lp_prob.objective))
```

EXPLANATION

Now, let's understand the code step by step:

- **Line 1-2:** First import the library pulp as p.
- **Line 4-5:** Define the problem by giving a suitable name to your problem, here I have given the name 'Problem'. Also, specify your aim for the objective function of whether to Maximize or Minimize.
- **Line 7-9:** Define LpVariable to hold the variables of the objective functions. The next argument specifies the lower bound of the defined variable, i.e. 0, and the upper bound is none by default. You can specify the upper bound too.
- **Line 11-12:** Denotes the objective function in terms of defined variables.
- **Line 14-18:** These are the constraints on the variables.
- **Line 21:** This will show you the problem in the output screen.
- **Line 23:** This is the problem solver.
- **Line 24:** Will display the status of the problem.
- **Line 27:** Will print the value for x and y and the minimum value for the objective function.

OUTPUT

```
====================== RESTART: C:/Users/mypr/Desktop/LPP.py =
Problem:
MINIMIZE
3*x + 5*y + 0
SUBJECT TO
_C1: 2 x + 3 y >= 12

_C2: - x + y <= 3

_C3: x >= 4

_C4: y <= 3

VARIABLES
x Continuous
y Continuous

Optimal
6.0 0.0 18.0
```

```
# Display the problem
print(Lp_prob)
```

```
status = Lp_prob.solve()   # Solver
print(p.LpStatus[status])   # The solution status
```

```
# Printing the final solution
print(p.value(x), p.value(y), p.value(Lp_prob.objective))
```

Resource Allocation Problem

- Now we will concrete on a practical optimization problem related to resource allocation in manufacturing industry.

- A factory produces four different products, and that the daily produced amount of the first product is X1, the amount produced of the second product is X2, and so on. The objective is to determine the profit-maximizing daily production amount for each product, bearing in mind the following conditions:

❖ The profit per unit of product is $20, $12, $40, and $25 for the first, second, third, and fourth product, respectively.

❖ Due to manpower constraints, the total number of units produced per day can not exceed fifty.

Continued…

- For each unit of the first product, three units of the raw material A are consumed. Each unit of the second product requires two units of the raw material A and one unit of the raw material B. Each unit of the third product needs one unit of A and two units of B. Finally, each unit of the fourth product requires three units of B.
- Due to the transportation and storage constraints, the factory can consume up to one hundred units of the raw material A and ninety units of B per day.

Solution

The mathematical model can be defined like this:

$$
\begin{aligned}
\max\ & 20x_1+12x_2+40x_3+25x_4 && \text{(profit)}\\
\text{s.t.:}\ & x_1+x_2+x_3+x_4 \le 50 && \text{(manpower)}\\
& 3x_1+2x_2+x_3 \le 100 && \text{(material A)}\\
& x_2+2x_3+3x_4 \le 90 && \text{(material B)}\\
& x_1, x_2, x_3, x_4 \ge 0
\end{aligned}
$$

- The objective function (profit) is defined in condition 1. The manpower constraint follows from condition 2. The constraints on the raw materials A and B can be derived from conditions 3 and 4 by summing the raw material requirements for each product.

- Finally, the product amounts can not be negative, so all decision variables must be greater than or equal to zero.

PYTHON CODE: To solve the aforementioned LPP

```
# import the library pulp as p
import pulp as p

# Create a LP Minimization problem
Lp_prob = p.LpProblem('Problem', p.LpMaximize)

# Create problem Variables
x1 = p.LpVariable("x1", lowBound = 0) # Create a variable x >= 0
x2 = p.LpVariable("x2", lowBound = 0) # Create a variable y >= 0
x3 = p.LpVariable("x3", lowBound = 0) # Create a variable x >= 0
x4 = p.LpVariable("x4", lowBound = 0) # Create a variable y >= 0


# Objective Function
Lp_prob += 20 * x1 + 12 * x2 + 40 * x3 + 25 * x4

# Constraints:
Lp_prob += x1 + x2 + x3 + x4 <= 50
Lp_prob += 3 * x1 + 2 * x2 + x3 <= 100
Lp_prob += x2 + 2 * x3 + 3 * x4 <= 90


# Display the problem
print(Lp_prob)

status = Lp_prob.solve() # Solver
print(p.LpStatus[status]) # The solution status

# Printing the final solution
print(p.value(x1), p.value(x2), p.value(x3), p.value(x4), p.value(Lp_prob.objective))
```

PYTHON OUTPUT

```
Problem:
MAXIMIZE
20*x1 + 12*x2 + 40*x3 + 25*x4 + 0
SUBJECT TO
_C1: x1 + x2 + x3 + x4 <= 50

_C2: 3 x1 + 2 x2 + x3 <= 100

_C3: x2 + 2 x3 + 3 x4 <= 90

VARIABLES
x1 Continuous
x2 Continuous
x3 Continuous
x4 Continuous

Optimal
5.0 0.0 45.0 0.0 1900.0
```

```
# Display the problem
print(Lp_prob)
```

```
status = Lp_prob.solve() # Solver
print(p.LpStatus[status]) # The solution status
```

```
# Printing the final solution
print(p.value(x1), p.value(x2), p.value(x3), p.value(x4), p.value(Lp_prob.objective))
```

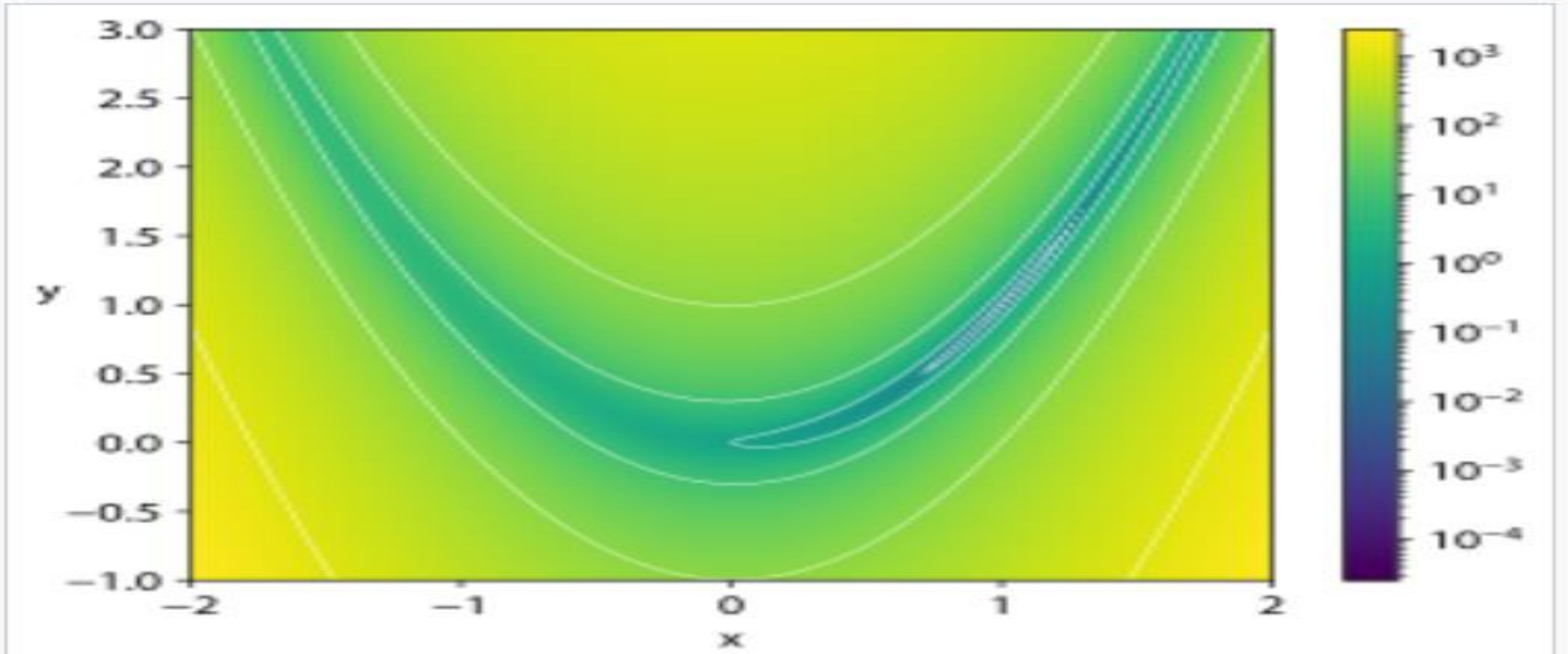
Unconstrained Optimization in SciPy

We use the **minimize()** function for the performing minimization on the scalar function. As an example function, we use the **Rosenbrock scalar** function.

The function is defined by

$$f(x, y) = (a - x)^2 + b(y - x^2)^2$$

It has a global minimum at $(x, y) = (a, a^2)$, where $f(x, y) = 0$. Usually, these parameters are set such that $a = 1$ and $b = 100$. Only in the trivial case where $a = 0$ the function is symmetric and the minimum is at the origin.



Plot of the Rosenbrock function of two variables. Here $a = 1, b = 100$, and the minimum value of zero is at $(1, 1)$.

Therefore, we minimize the following unconstraint problem

$$\text{Min } f(x,x) = (1-x)^2 + 100 * \left(x - x^2\right)^2$$

```
##SciPy Least- Square Minimization
##We can solve the least squares with bound variables.
##We take two inputs - a residual cost function
##and a loss scalar function and
##calculate the minimum cost function.
import numpy as np
from scipy.optimize import minimize
from scipy.optimize import least_squares
def rosen(x):
  return sum(100.0*(x[1:]-x[:-1]**2.0)**2.0 + (1-x[:-1])**2.0)

input = np.array([1, 3])
res = least_squares(rosen, input)
print(res.x)
```

Global Optimization in Python

- In data analysis, finding the global minimum of a function is a common task. However, it can be challenging to find the optimal solution due to the presence of multiple local minima.
- The "scipy" library provides a convenient method for global optimization called "scipy.optimize.basinhopping()".
- This function implements the basin-hopping algorithm, which uses a combination of local optimization methods and stochastic jumps to find the global minimum of a given function.
- Now, we will take an example of using the "scipy.optimize.basinhopping()" function to find the global minimum of a one-dimensional multimodal function.

Implimentation in Python

- **Step 1: Import Library**

- **Step 2: Define objective function and initial guess:- We converted the following function into Python, and gave an initial guess of x = -2 for the location of the global minimum.**

$$f(x) = -10\cos(\pi x - 2.2) + (x + 1.5)x$$

- **Step 3: Setup and call `basinhopping()` function:-** In this case, we call on the **BFGS (Broyden, Fletcher, Goldfarb, and Shanno)** method for unconstrained minimization. **"scipy"** implements a number of other methods as well. We have also specified how many basin-hopping iterations to run via the argument **"niter"**.

Continued…

- **Step 4: Print results**

From the results, stored in the `message`, `x`, and `fun` attributes, we can see that the algorithm detected the global minimum at:

$$x = -1.299$$
$$f(x) = -10.266$$

- **Step 5: Plot function, and check results**

Python Code

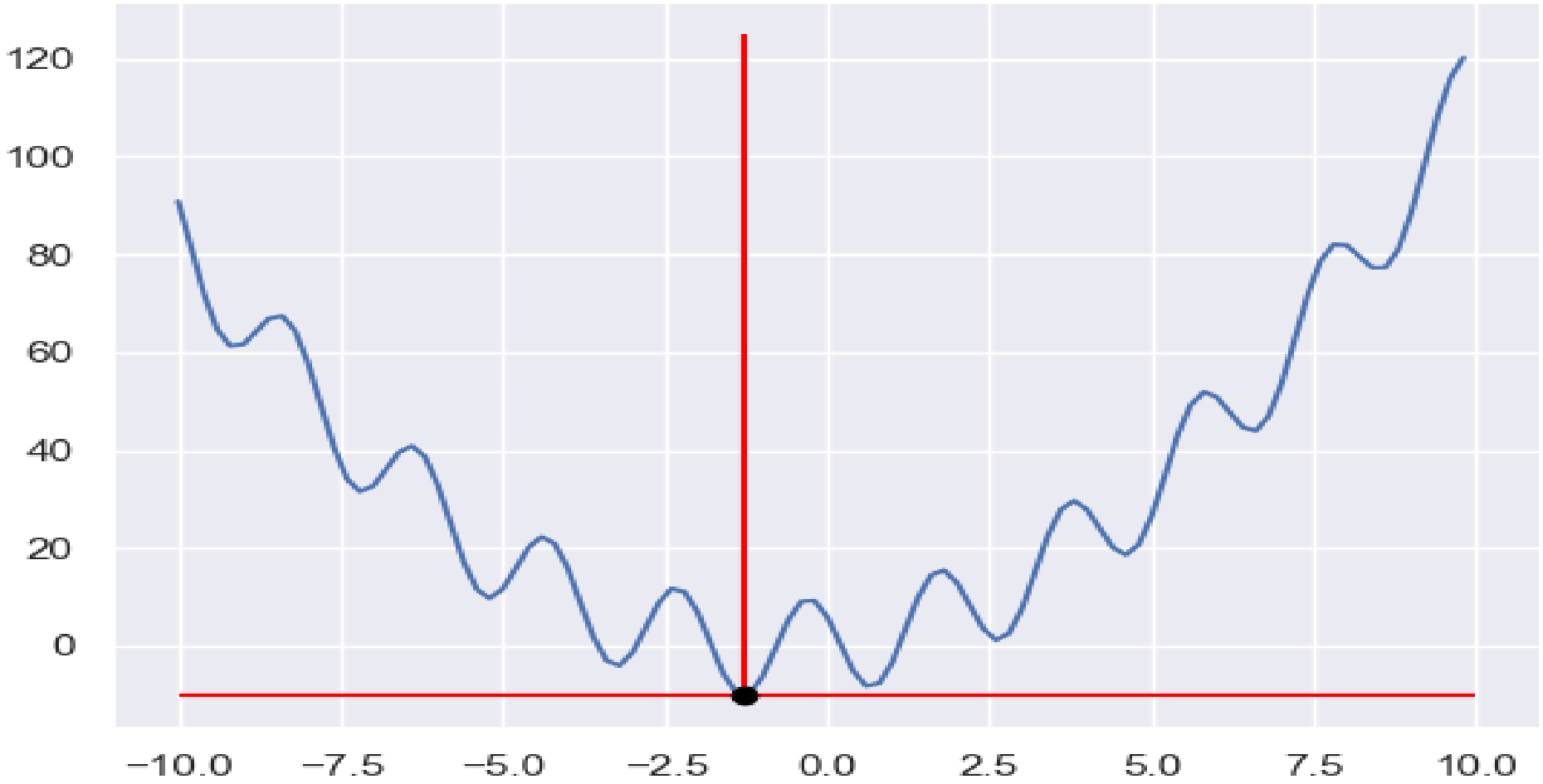
```
# Import the necessary libraries
import numpy as np
import scipy.optimize as opt
# Define objective function
def f(x):
    return -10*np.cos(np.pi*x - 2.2) + (x + 1.5)*x

# Set initial guess
x0 = [-2]
# Set up args for basinhopping and call function
minimizer_kwargs = {"method": "BFGS"}
optimization_algorithm = opt.basinhopping(f, x0, minimizer_kwargs = minimizer_kwargs, niter = 200)
print("1-D function")
print(optimization_algorithm.message[0])

# Save results
optimized_x = optimization_algorithm.x
optimized_fun = optimization_algorithm.fun

# Print results
print("Optimized x: ", optimized_x)
print("Optimized function value: ", optimized_fun)
```

Python Code for Plot function, and checking results

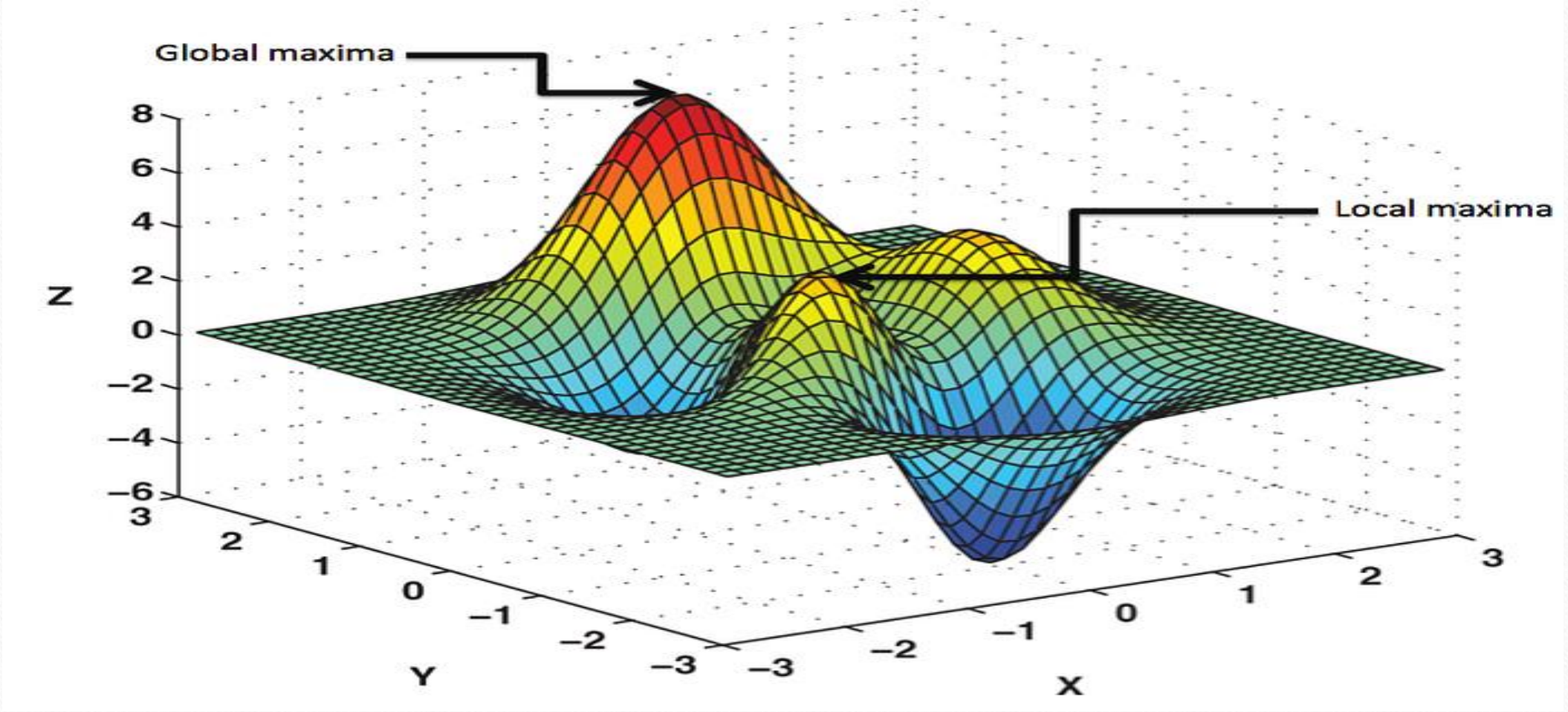
```
#Plot Code
import seaborn as sns
import matplotlib.pyplot as plt
sns.set_theme()

# Generate data for objective function graph
X = np.arange(-10, 10, 0.2)
y = [f(x) for x in X]

# Plot global minimum
plt.vlines(x = optimized_x, ymin = -10, ymax = 125, colors = 'red')
plt.hlines(y = optimized_fun, xmin = -10, xmax = 10, colors = 'red')
plt.plot(X, y)
plt.plot(optimized_x, optimized_fun,'o', color = "black")
plt.show()
```

OUTPUT

```
= RESTART: H:\Ramanujan College\Numerical Optimization\Practical\Prac_3_Global_a
nd_Local_Optimization.py
1-D function
requested number of basinhopping iterations completed successfully
Optimized x:  [-1.28879778]
Optimized function value:  -10.26631244852453
```


120
100
80
60
40
20
0
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0

Line Search and Trust Region Methods: Two Optimization Strategies



Line Search and Trust Region Methods: Two Optimization Strategies

- The objective of the optimization problem is to find the optimal point. The problem aims to construct the sequences of point, X[2].
- Each next point in the sequence can be determined by moving some distance in the direction, d.
- The update rule of such methods can be formulated as below.

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

- There are two optimization strategies: line search and trust region.

LINE SEARCH

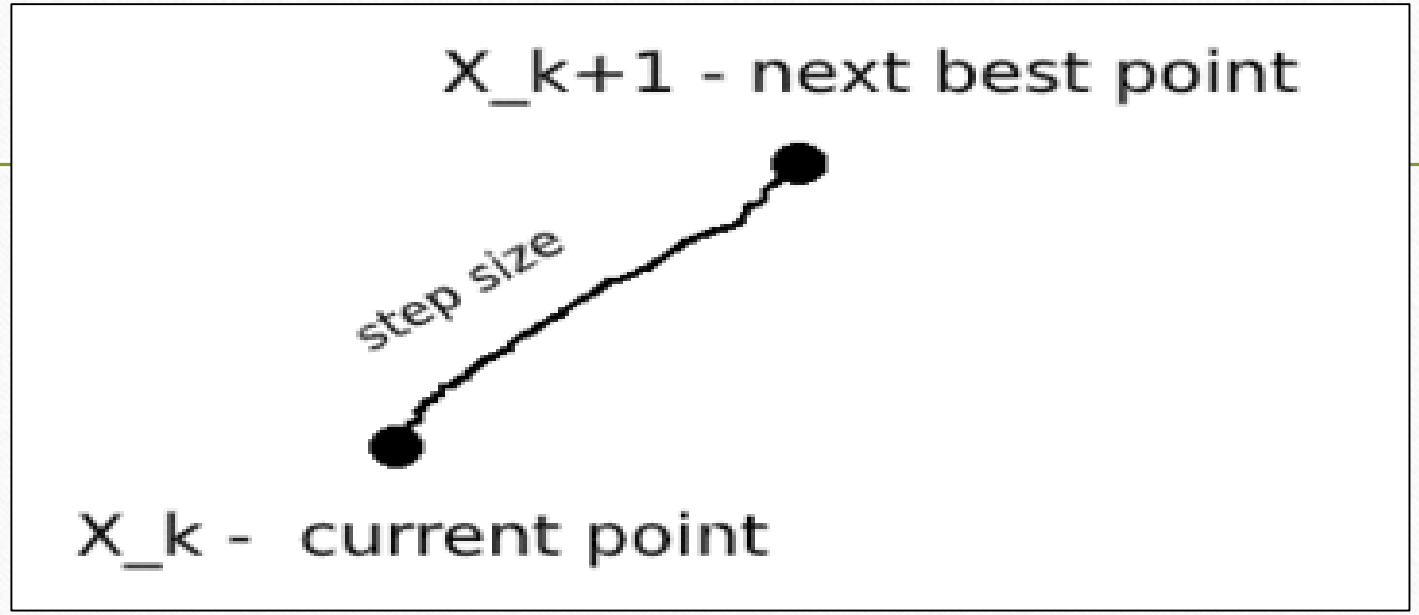
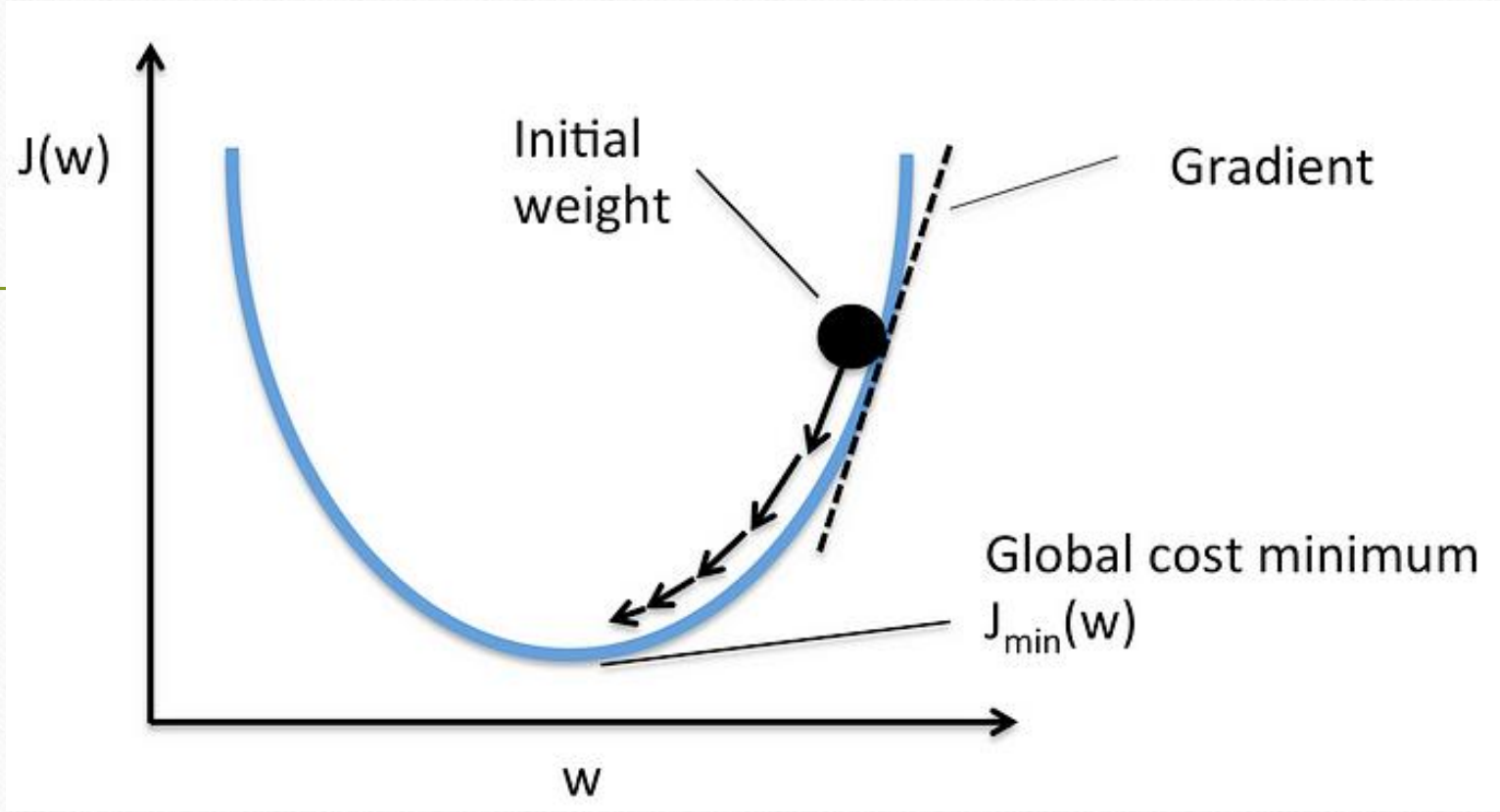


Fig. 1: Illustration of line search

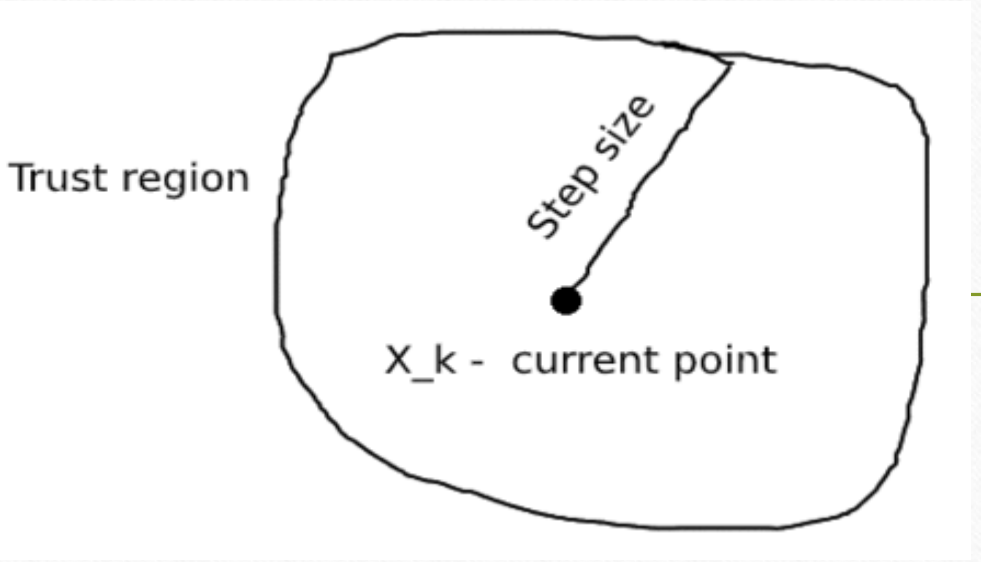
- In line search, we first find the best direction d and decide the step size, α to determine how long we need to go in that direction.

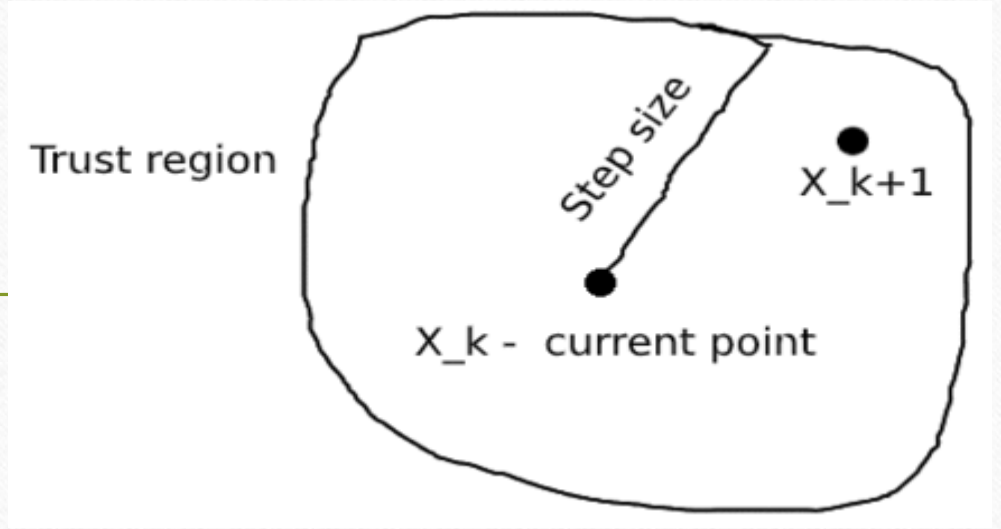
LINE SEARCH

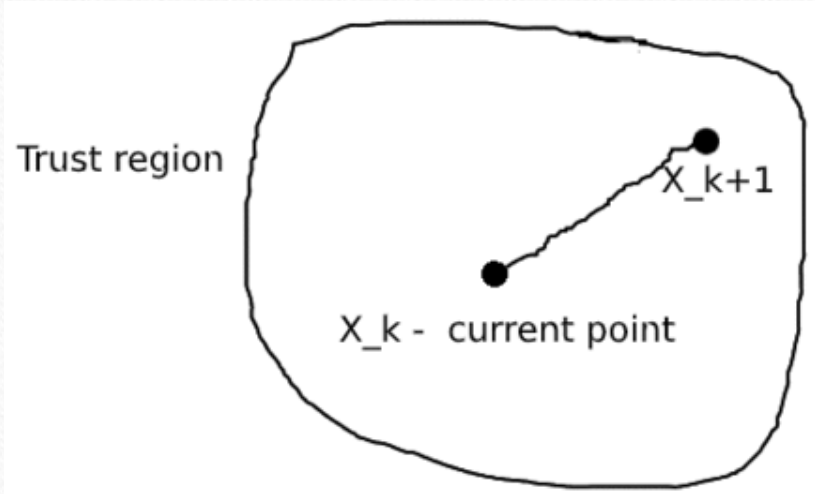


- As shown above, Gradient accent and decent approaches are the examples of the line search.

Trust Region







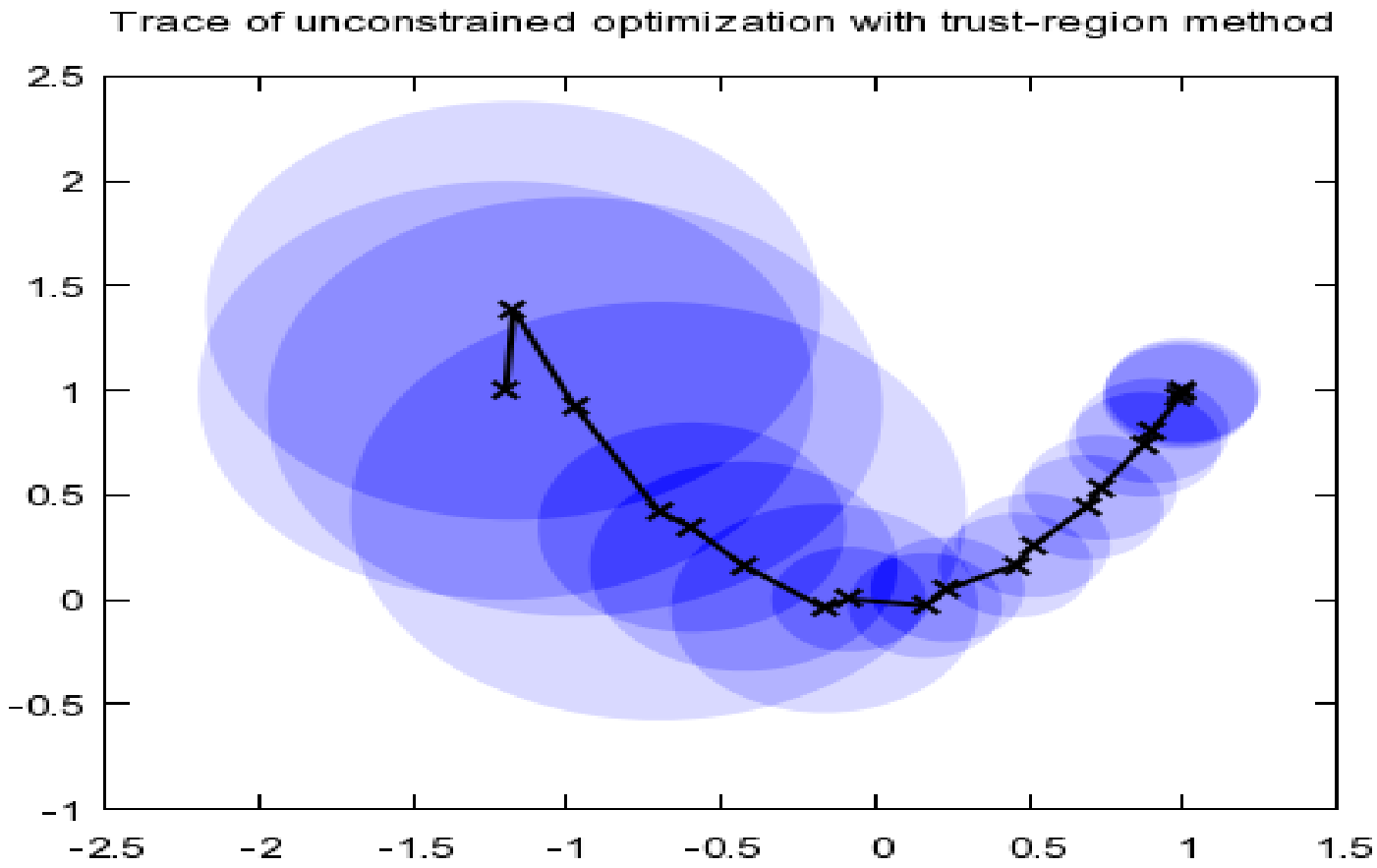
Trust Region

- In trust region, we first decide the step size, α. We can construct a region by considering the α as the radius of the circle. We can call this region a trust region.

- The search for the best point (local minimum or local maximum in that region) for the improvement is bounded in that region.

- Once we have the best point it determines the direction. This process repeats until the optimal point is reached.

Trust Region



- **The above picture is an example of how trust-region optimization works. Let's deep dive into this method in the next slide.**

Trust Region

- The idea here is to approximate objective F(x) with a smaller f(x).
- It means that we can restrict the search to a trusted region to approximate F well.
- f is often chosen to be a quadratic approximation of F.
- The quadratic form of the trust-region optimization, as shown below:

$$f(c)+\nabla f(c)^T(x-c)+\frac{1}{2!}(x-c)^T H(c)(x-c),\ where\ \|x-c|\ \leq \delta$$

- Minimize this quadratic form of trust-region to find the best point for the improvement in that region.
- Where ∇f is the gradient (first-order derivative), H is the Hessian (second-order derivative), and c is the certain point in the trust region.

Algorithm for the Trust-Region Optimization Method

- **Step 1:** Initialize x, δ
- **Step 2:** Find the best point for improvement(local minimum or local maximum)in the trust region by using the quadratic approximation as shown below.

$$f(c)+\nabla f(c)^T(x-c)+\frac{1}{2!}(x-c)^T H(c)(x-c),\ where\ \|x-c|\ \leq \delta$$

- Minimize this quadratic form of trust-region to find the best point for the improvement in that region.
- **Step 3:** Once we have the best point for improvement, check whether the approximation is good or not. If the approximation is good, then increase the δ. Otherwise, decrease the δ.
- **Step 4:** Construct the trust-region based on the new point and δ.
- **Step 5:** Repeat 2 to 4 until we reach the optimal point.

Line Search in Python

- **Step 1: Define the objective function:-** In this lecture, we will use a one-dimensional objective function, specifically x^2 shifted by a small amount away from zero. This is a convex function and was chosen because it is easy to understand and to calculate the first derivative.

$$Objective\ Function\ (x) = (x - 5)^2$$

- Note that the line search is not limited to one-dimensional functions or convex functions.
- **Step 2: Calculate the gradient (first-order derivative):-** The first derivative for this function can be calculated analytically; as follows:

$$gradient\ (x) = 2 * (x - 5)$$

- The gradient for each input value just indicates the slope towards the optima at each point.
- **Step 3: Define an input range:-** In this example, we define input range for x from -10 to 20 and calculate the objective value for each input.

Line Search in Python

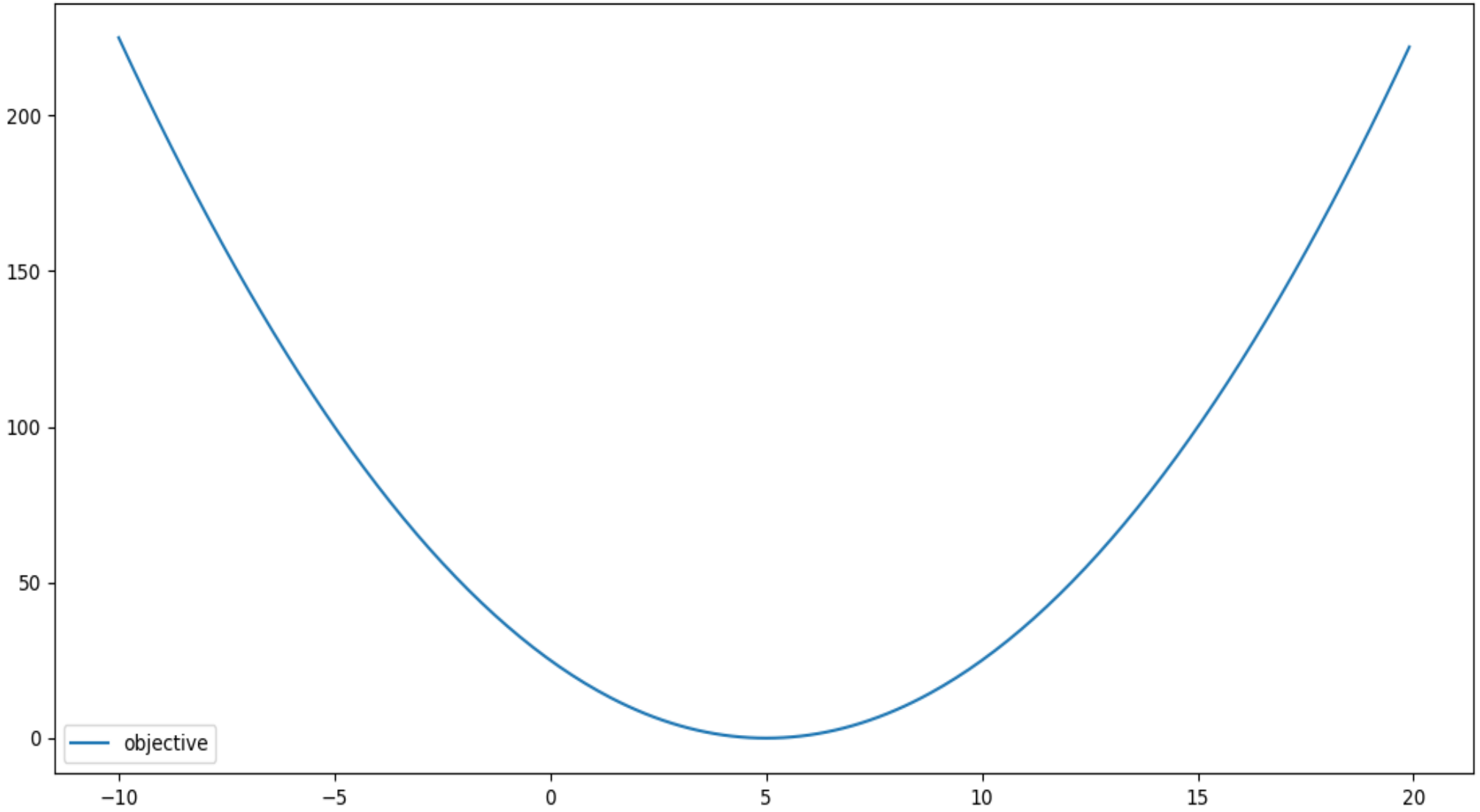
- **Step 4: Plot the input values versus the objective values:** To get an idea of the shape of the function.
- **Step 5: Define the starting point for the search and the direction to search:-**
  - The direction is like the step size and the search will scale the step size to find the optima.
  - The search will then seek out the optima and return the alpha or distance to modify the direction.
  - We can retrieve the alpha from the result, as well as the number of function evaluations that were performed.
  - We can use the alpha, along with our starting point and the step size to calculate the location of the optima and calculate the objective function at that point (which we would expect would equal 0.0).

Line Search in Python

- In the next slide, we will see initial Code for evaluating input values (x) in the range from -10 to 20 and creates a plot showing the familiar parabola U-shape.

- The optima for the function appears to be at x=5.0 with an objective value of 0.0.

```
#initial Code for evaluating input values (x) in the range from -10 to 20
#and creates a plot showing the familiar parabola U-shape.
#The optima for the function appears to be at x=5.0
#with an objective value of 0.0.
from numpy import arange
from matplotlib import pyplot
# objective function
def objective(x):
 return (-5.0 + x)**2.0
# gradient for the objective function
def gradient(x):
 return 2.0 * (-5.0 + x)

# define range
r_min, r_max = -10.0, 20.0
# prepare inputs
inputs = arange(r_min, r_max, 0.1)
# compute targets
targets = [objective(x) for x in inputs]

# plot inputs vs objective
pyplot.plot(inputs, targets, '-', label='objective')
pyplot.legend()
pyplot.show()
```


200
150
100
50
0
objective
−10
−5
0
5
10
15
20

Line Search Code in Python

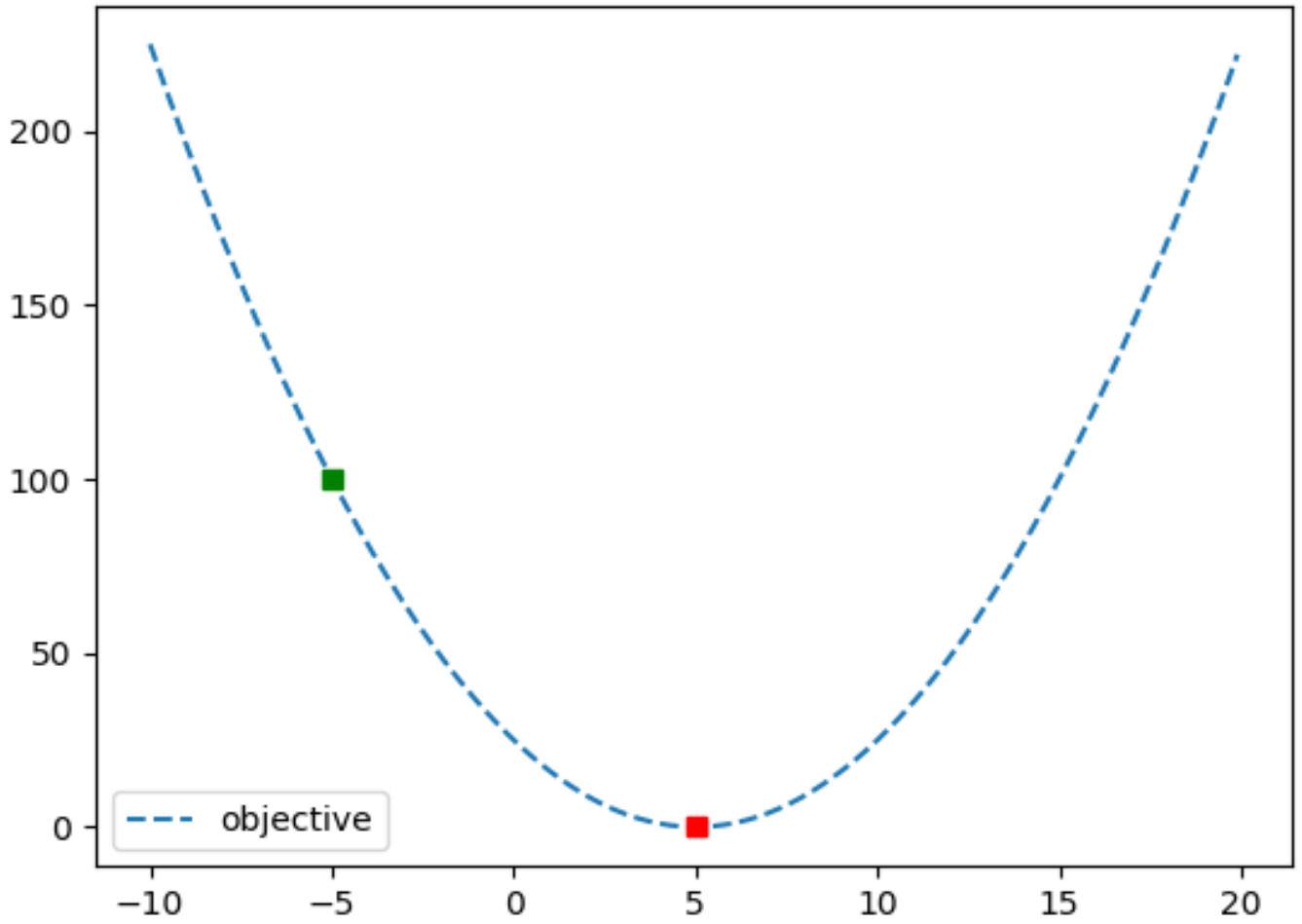
```
# perform a line search on a convex objective function
from numpy import arange
from scipy.optimize import line_search
from matplotlib import pyplot

# objective function
def objective(x):
 return (-5.0 + x)**2.0

# gradient for the objective function
def gradient(x):
 return 2.0 * (-5.0 + x)

# define the starting point
point = -5.0
# define the direction to move
direction = 100.0
# print the initial conditions
print('start=%.1f, direction=%.1f' % (point, direction))
```

Continued...

```
# perform the line search
result = line_search(objective, gradient, point, direction)
# summarize the result
alpha = result[0]
print('Alpha: %.3f' % alpha)
print('Function evaluations: %d' % result[1])
# define objective function minima
end = point + alpha * direction
# evaluate objective function minima
print('f(end) = f(%.3f) = %.3f' % (end, objective(end)))
# define range
r_min, r_max = -10.0, 20.0
# prepare inputs
inputs = arange(r_min, r_max, 0.1)
# compute targets
targets = [objective(x) for x in inputs]
# plot inputs vs objective
pyplot.plot(inputs, targets, '--', label='objective')
# plot start and end of the search
pyplot.plot([point], [objective(point)], 's', color='g')
pyplot.plot([end], [objective(end)], 's', color='r')
pyplot.legend()
pyplot.show()
```

OUTPUT

```
= RESTART: H:\Ramanujan College\Numerical Optimization\Practical\Prac_4_Line Search Optimiz
ation_Code.py
start=-5.0, direction=100.0
Alpha: 0.100
Function evaluations: 3
f(end) = f(5.000) = 0.000
```



Explain the OUTPUT of Line Search

- The search is performed and an alpha is located that modifies the direction to locate the optima, in this case, 0.1, which was found after three function evaluations.

- The point for the optima is located at 5.0, which evaluates to 0.0, as expected.

- Finally, a plot of the function is created showing both the starting point (green) and the objective (red).

How to Handle the Failure Cases of Line Search

- The search is not guaranteed to find the optima of the function.
- This can happen if a direction is specified that is not large enough to encompass the optima.
- For example, if we use a direction of three, then the search will fail to find the optima. We can demonstrate this with a complete code, listed in the next slide.

Failure Cases of Line Search

```
# perform a line search on a convex objective function with a direction that is too small
from numpy import arange
from scipy.optimize import line_search
from matplotlib import pyplot

# objective function
def objective(x):
 return (-5.0 + x)**2.0

# gradient for the objective function
def gradient(x):
 return 2.0 * (-5.0 + x)
# define the starting point
point = -5.0
# define the direction to move
direction = 3.0
# print the initial conditions
print('start=%.1f, direction=%.1f' % (point, direction))
# perform the line search
result = line_search(objective, gradient, point, direction)
# summarize the result
alpha = result[0]
print('Alpha: %.3f' % alpha)
```

Failure Cases of Line Search

```
# define objective function minima
end = point + alpha * direction
# evaluate objective function minima
print('f(end) = f(%.3f) = %.3f' % (end, objective(end)))
```

OUTPUT

```
= RESTART: H:\Ramanujan College\Numerical Optimization\Practical\Prac_5_Failure
case of Line Search Optimization_Code.py
start=-5.0, direction=3.0
Alpha: 1.000
f(end) = f(-2.000) = 49.000
```

- In the above output, the search reaches a limit of an alpha of 1.0 which gives an end point of -2 evaluating to 49. A long way from the optima at f(5) = 0.0.

Failure Cases of Line Search

- Additionally, we can choose the wrong direction that only results in worse evaluations than the starting point.

- In this case, the wrong direction would be negative away from the optima, e.g. all uphill from the starting point.
- The expectation is that the search would not converge as it is unable to locate any points better than the starting point.
- The complete example of the search that fails to converge is provided in the next slide.

Code

```
# perform a line search on a convex objective function that does not converge
from numpy import arange
from scipy.optimize import line_search
from matplotlib import pyplot

# objective function
def objective(x):
 return (-5.0 + x)**2.0

# gradient for the objective function
def gradient(x):
 return 2.0 * (-5.0 + x)

# define the starting point
point = -5.0
# define the direction to move
direction = -3.0
# print the initial conditions
print('start=%.1f, direction=%.1f' % (point, direction))
# perform the line search
result = line_search(objective, gradient, point, direction)
# summarize the result
print('Alpha: %s' % result[0])
```

OUTPUT

```
Warning (from warnings module):
  File "C:\Users\mypr\AppData\Local\Programs\Python\Python310\lib\site-packages\
scipy\optimize\linesearch.py", line 327
    warn('The line search algorithm did not converge', LineSearchWarning)
LineSearchWarning: The line search algorithm did not converge
Alpha: None
```

Explanation:

- Running the example results in a LineSearchWarning indicating that the search could not converge, as expected.

- The alpha value returned from the search is None.


Thank You