Week 2
Single Variable Optimization

ENGR. ROMMEL N. GALVAN

Contents

1. Introduction

A single variable problem is such that

Min f(x), $x \in [a,b]$

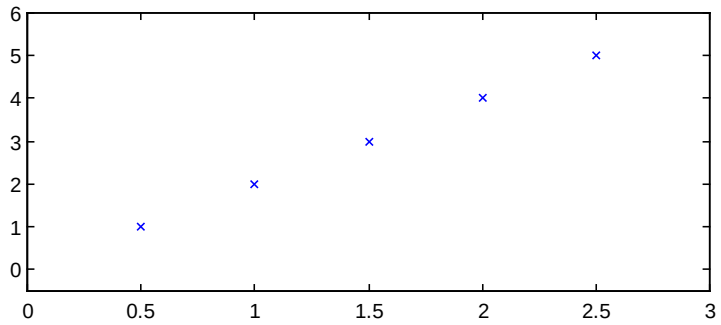
Given f continuous, the optimality criteria is such that:

df/dx=0 at x=x*

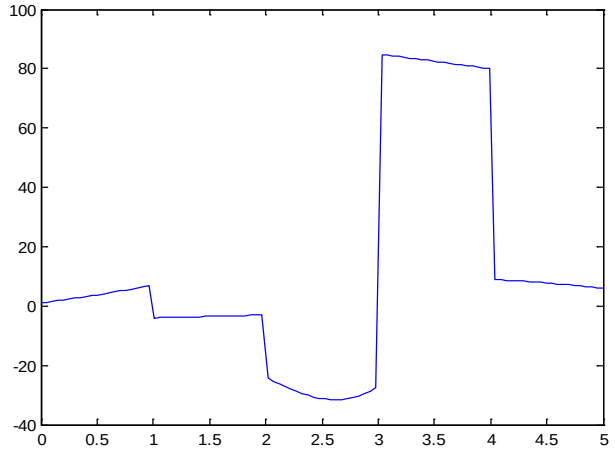
where x* is either a local minimum or a local maximum or a saddle point (a stationary point)

Property of a single variable function

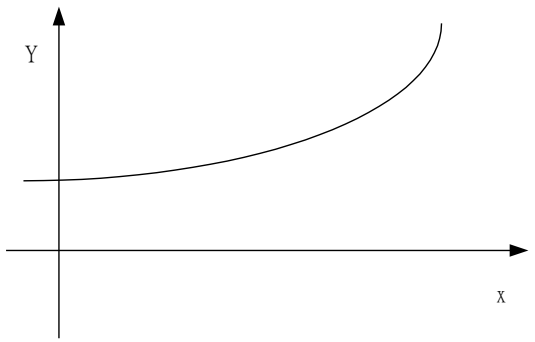
(i) a discrete function



(ii) a discontinuous function



(iii) a continuous function



2. Example

In a chemical plant, the cost of pipes, their fittings, and pumping are important investment cost. Consider a design of a pipeline L feet long that should carry fluid at the rate of Q gpm. The selection of economic pipe diameter D(in.) is based on minimizing the annual cost of pipe, pump, and pumping. Suppose the annual cost of a pipeline with a standard carbon steel pipe and a motor-driven centrifugal pump can be expressed as:

$$f = 0.45L + 0.245LD^{1.5} + 3.25(hp)^{1/2} + 61.6(hp)^{0.925} + 102$$

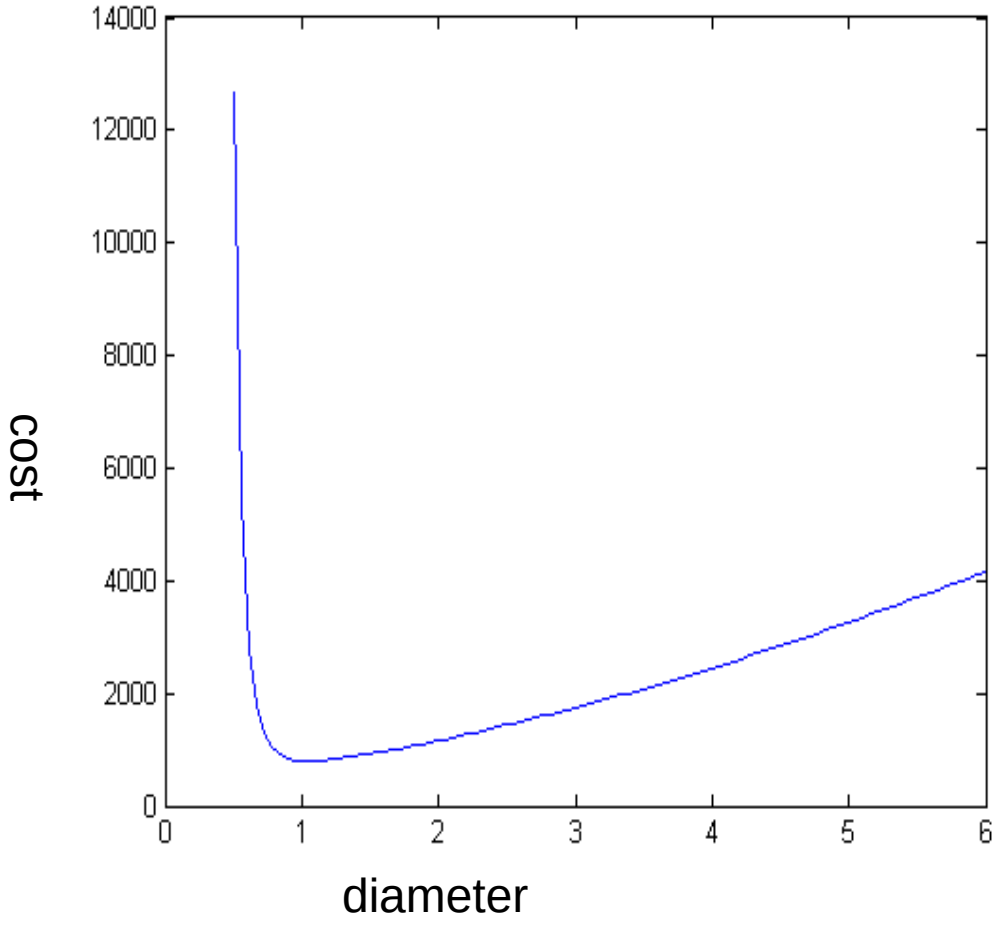
where

$$hp = 4.4\times10^{-8}\frac{LQ^3}{D^5} + 1.92\times10^{-9}\frac{LQ^{2.68}}{D^{4.68}}$$

Formulate the appropriate single-variable optimization problem for designing a pipe of length 1000ft with a fluid rate of 20gpm. The diameter of the pipe should be between 0.25 to 6 in.

Process Design Example

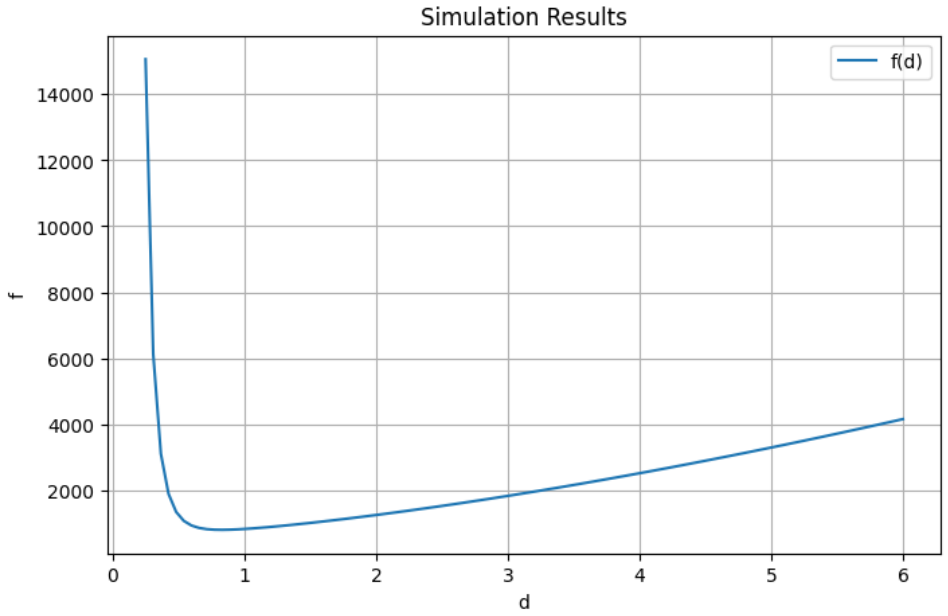
```python
import numpy as np
import matplotlib.pyplot as plt

# 1. Setup the variables
d = np.linspace(0.25, 6, 100)

l = 1000
q = 20

# 2. Perform Calculations
hp = 4.4e-8 * (l * q**3) / d**5 + 1.92e-9 * (l * q**2.68) / d**4.68


f = 0.45 * l + 0.245 * l * d**1.5 + 3.25 * hp**0.5 + 61.6 * hp**0.925 + 102

# 3. Output the results
# Printing the first few values to verify
print("First 5 values of f:")
print(f[:5])

# Optional: Plotting the data
plt.figure(figsize=(8, 5))
plt.plot(d, f, label='f(d)')
plt.xlabel('d')
plt.ylabel('f')
plt.title('Simulation Results')
plt.grid(True)
plt.legend()
plt.show()
```



2. Example- Continued

Carry out a single-variable search of $f(x)=3x^2+12/x-5$

$df/dx=6x-12/x^2=0$



Solutions Methods (1) Analytical Approach

Problem:

$$\min_{x} f(x)$$
$$s.t. \quad a \le x \le b$$

Algorithm

Step 1: Set $df/dx=0$ and solve all stationary points.

Step 2: Select all stationary point $x_1, x_2,\ldots,x_N$ in $[a,b]$.

Step 3: Compare function values $f(x_1), f(x_2),\ldots,f(x_N)$ and find global minimum.

Example: Polynomial Problem

Example:

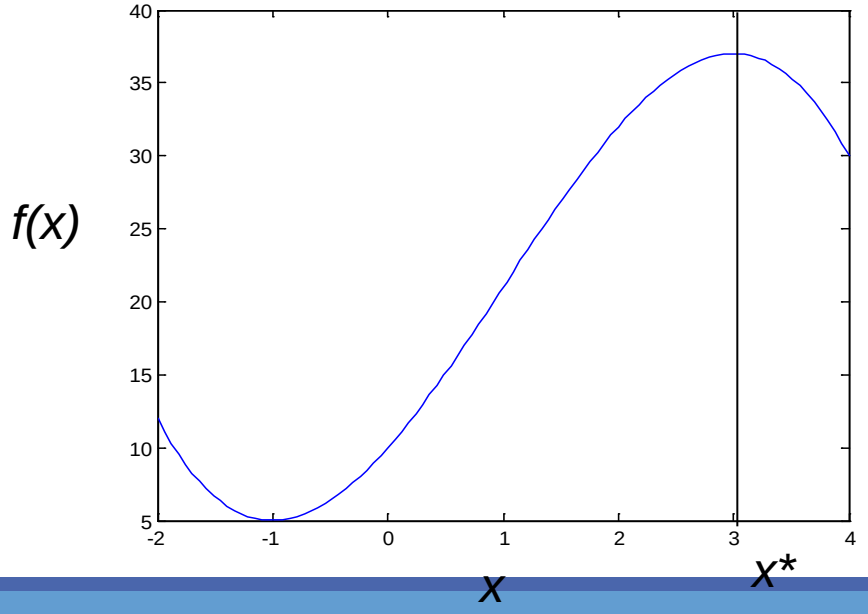
$$\max f(x) = -x^3 + 3x^2 + 9x + 10$$

In the interval of [-2,4].

$$\left.\frac{df}{dx}\right|_{x=x^*} = -3x^2 + 6x + 9 = 0 \Rightarrow x^* = 3, -1$$

and $f(3)=37$, $f(-1)=5$

$\therefore 3$ is the optimum point.



Example- Inventory Problem

Example : Inventory Control (Economic Order Quantity)

Inventory quantity each time =Q units

Set up cost or ordering cost =$K

Acquisition cost =$C/unit

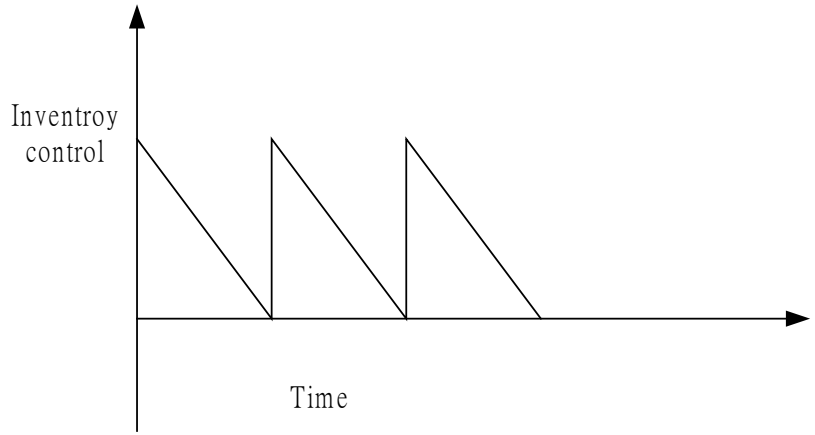
Storing cost per unit =$h/year

Demand (constant)=λ units/time unit, this implies ordering period =T=Q/ λ

Question: What is optimum ordering amount Q?

Solution

Cost per ordering cycle $= K + CQ + QhT/2$



Objective function =Total cost per year

f(*Q*)=Cost per cycle* $\frac{1\ year}{T\ year/cycle} = \frac{K\lambda}{Q} + c\lambda + \frac{Qh}{2}$

$$f'(Q) = -\frac{K\lambda}{Q^2} + \frac{h}{2} \Rightarrow Q^* = \sqrt{\frac{2\lambda K}{h}} > 0 \text{ or } T^* = \frac{Q^*}{\lambda} = \sqrt{\frac{2K}{\lambda h}}$$

Problems

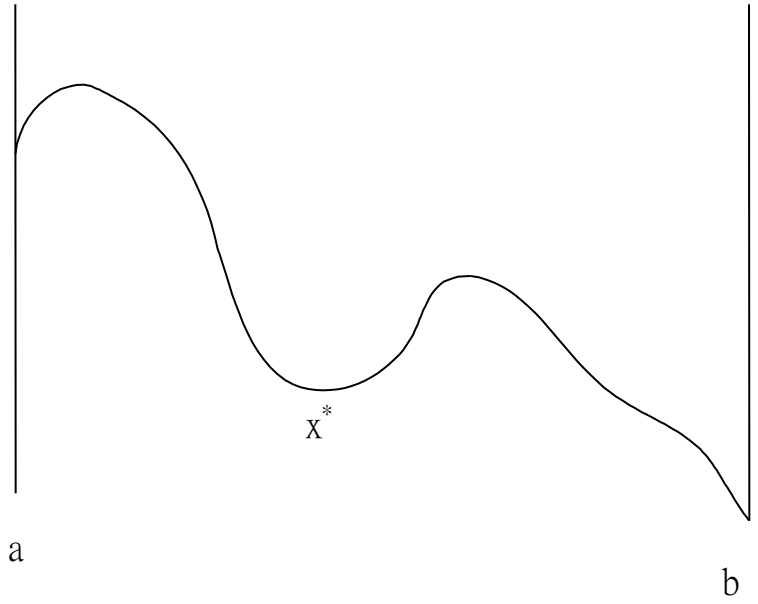
Cost function must be explicitly expressed

The derivative of the cost function must also be explicitly written down

The derivative equation must be explicitly solved

In many cases, the derivative equation is solved numerically, such as Newton's method, it is more convenient to solve the optimization problem numerically.

Problems - Continued



Location of global minimum

The Importance of One-dimensional Problem - The Iterative Optimization Procedure

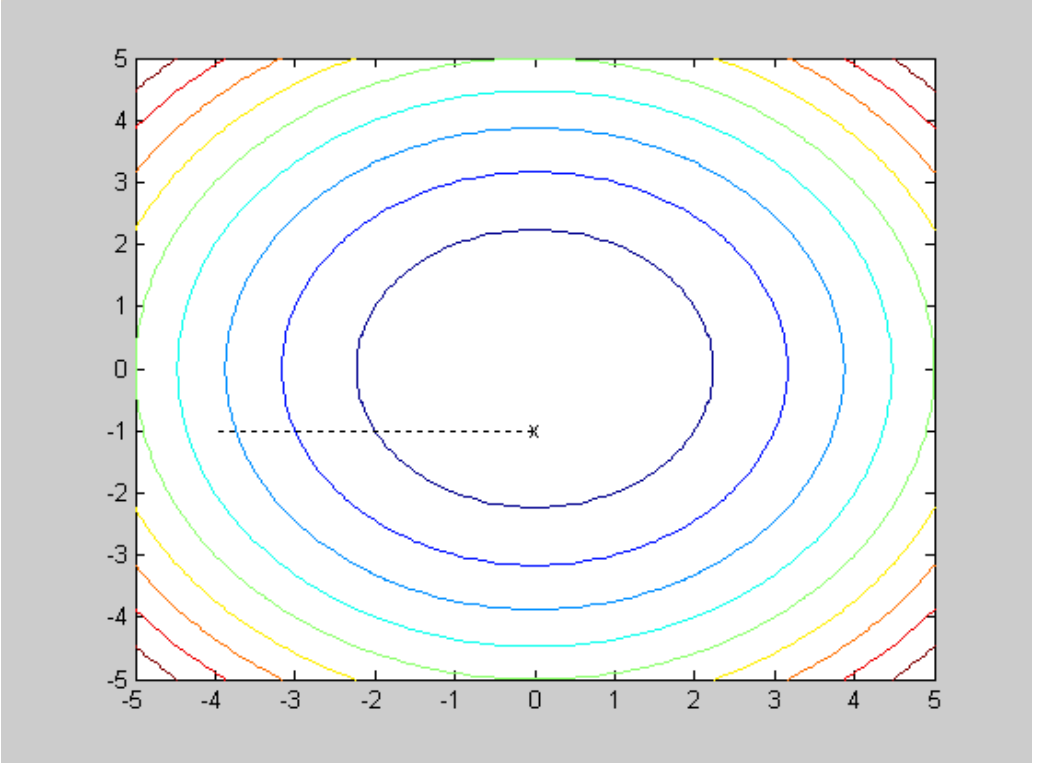
Optimization is basically performed in a fashion of iterative optimization. We give an initial point **x**0, and a direction **s**, and then perform the following line search:

$$\mathbf{x}_1 = \mathbf{x}_0 + \alpha^* \mathbf{s}$$

where α^* is the optimal point for the objective function and satisfies all the constraints. Then we start from **x**1 and find the other direction **s**', and perform a new line search until the optimum is reached.

The Importance of One-dimensional Problem - The Iterative Optimization Procedure - Continued

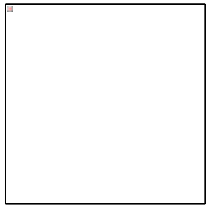
Consider a objective function Min f(X)= $x_1^2+x_2^2$, with an initial point X_0=(-4,-1) and a direction (1,0), what is the optimum at this direction, i.e. $X_1=X_0+\alpha*(1,0)$. This is a one-dimensional search for α.



The Importance of One-dimensional Problem - The Iterative Optimization Procedure - Continued

The problem can be converted into:

$$\min_{\alpha} f(\alpha) = (-4+\alpha)^2 + (-1)^2 = 17 - 8\alpha + \alpha^2$$

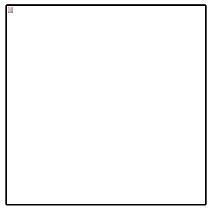


$$\frac{df(\alpha)}{d\alpha} = -8 + 2\alpha = 0$$

$$\alpha = 4 \quad \text{QED}$$

The new point is at

(0,-1) as shown



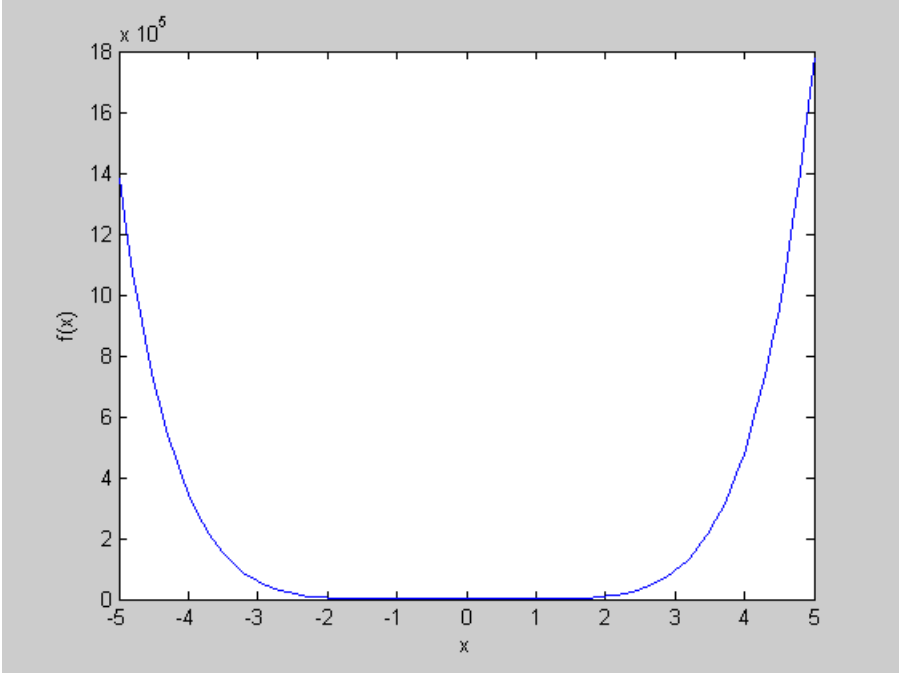
Solution Methods (2) – Numerical Approaches: (i) Numerical Solution to Optimality Condition

Example: Determine the minimum of

$f(x)=(10x^3+3x^2+x+5)^2$

The optimality criteria leads:

$2(10x^3+3x^2+x+5)(30x^2+6x+1)=0$



Problem: What is the root of the above equation?

Newton's Method

Consider a equation to be solved:

f(x)=0;

Step 1: give an initial point x0

Step 2: $x_{n+1}=x_n-f(x_n)/f'(x_n)$

Step 3: is $f(x_n)$ small enough? If not go back to step 2

Step 4: stop

Newton's Method (example)

https://colab.research.google.com/drive/1wZY_PvcXL1Or-uRIQEHxLN4Hb8ajmGY_?usp=sharing

```
x0 = 10
fx = 100
iteration = 0  # Renamed from 'iter' because iter is a built-in Python function
ff = []
xx = []

while abs(fx) > 1.0e-5:
    # Calculate fx
    # Note: x^3 in MATLAB becomes x**3 in Python
    fx = 2 * (10*x0**3 + 3*x0**2 + x0 + 5) * (30*x0**2 + 6*x0 + 1)

    # Append to list (equivalent to MATLAB's ff = [ff; fx])
    ff.append(fx)

    # Calculate fxp
    # Note: I have kept your formula exactly as written, including the
    # term (10*x0**3 + 3*x0**2 + 5), which is missing the '+ x0' seen in the first line.
    fxp = 2 * ((30*x0**2 + 6*x0 + 1) * (30*x0**2 + 6*x0 + 1) +
               (10*x0**3 + 3*x0**2 + 5) * (60*x0 + 6))

    # Newton-Raphson update
    x0 = x0 - fx / fxp

    # Append to list
    xx.append(x0)

    iteration += 1

# Print results to match the provided output
print(f"fx   = {fx:.4e}")
print(f"iter = {iteration}")
print(f"x0   = {x0:.4f}")
```

A Numerical Differentiation Approach

Problem: Find df/dx at a point x_k

Approach:

- Define Δx_k
- Find $f(x_k)$
- Find $f(x_k + \Delta x_k)$
- Approximate $df/dx = [(f(x_k + \Delta x_k) - f(x_k))/ \Delta x_k]$

A Numerical Differentiation Approach

https://colab.research.google.com/drive/19bP_0dAUmPY-1cZ8JMzz8cvo8FhFX88p?usp=sharing

```
x0 = 10
fx = 100
iteration = 0
ff = []
xx = []
dx = 0.001

while abs(fx) > 1.0e-5:
    # Calculate fx
    fx = 2 * (10*x0**3 + 3*x0**2 + x0 + 5) * (30*x0**2 + 6*x0 + 1)

    # Append fx to list
    ff.append(fx)

    # Calculate perturbed x
    xp = x0 + dx

    # Calculate f(x + dx)
    ffp = 2 * (10*xp**3 + 3*xp**2 + x0 + 5) * (30*xp**2 + 6*xp + 1)

    # Numerical derivative (Finite Difference)
    fxp = (ffp - fx) / dx

    # Newton-Raphson update
    x0 = x0 - fx / fxp

    # Append x0 to list
    xx.append(x0)

    iteration += 1

# Print results
print(f"fx   = {fx:.4e}")
print(f"iter = {iteration}")
print(f"x0   = {x0:.4f}")
```

Remarks (Numerical Solution to Optimality Condition)

Difficult to formulate the optimality condition

Difficult to solve (multi-solutions, complex number solutions)

Derivative may be very difficult to solve numerically

Function calls are not saved in most cases

Solution Methods (2) – Numerical Approaches (ii) Reginal Elimination Methods

Theorem: Suppose f is uni-model on the interval $a \leq x \leq b$, with a minimum at x^* (not necessary a stationary point), let $x_1, x_2 \in [a,b]$ such that $a < x_1 < x_2 < b$, then:

- If $f(x_1) > f(x_2) \Rightarrow x^* \in [x_1, b]$
- If $f(x_1) < f(x_2) \Rightarrow x^* \in [a, x_2]$

Two Phase Approach

Phase I. Bounding Phase: An initial course search that will bound or bracket the optimum

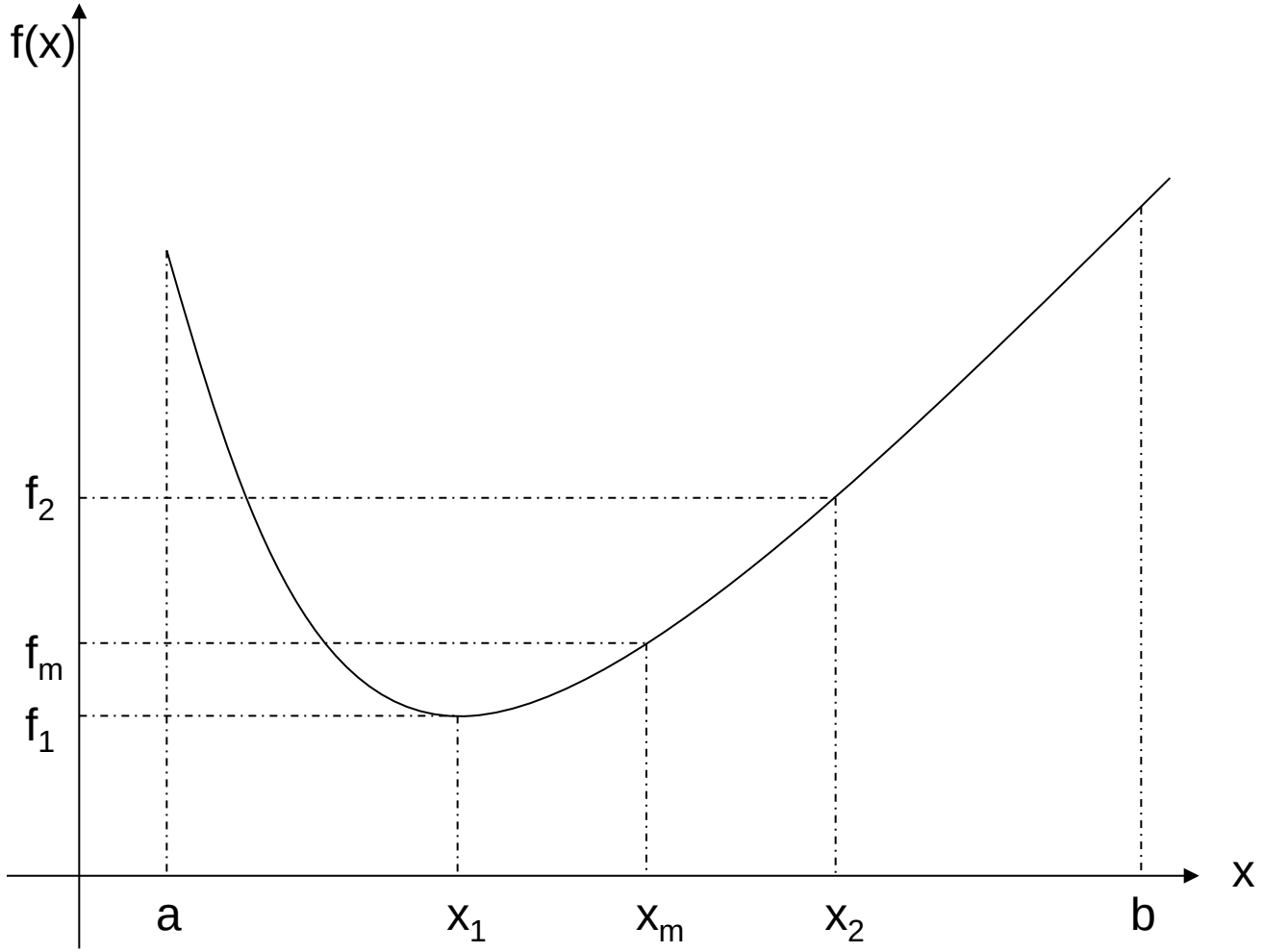
Phase II. Interval Refinement Phase: A finite sequence of interval reductions or refinements to reduce the initial search interval to desired accuracy.

Phase II- Interval Refinement Phase - Interval Halving

Algorithm

- Step 1: Let , $L= b-a$, $x_m = \frac{a+b}{2}$ find $f(x_m)$
- Step 2: Set $x_1=a+L/4$, $x_2=b-L/4$.
- Step 3: Find $f(x_1)$, if $f(x1)<f(x_m)$, then $b \leftarrow x_m$, go to step 1.
- if $f(x_1)>f(x_m)$, continue
- Step 4: Find $f(x_2)$
  - If $f(x_2)<f(x_m)$, then $a \leftarrow x_m$, go to step 1.
  - If $f(x_2)>f(x_m)$, then $a \leftarrow x_1$, $b \leftarrow x_2$, go to step 1.

Interval Halving



Example: y=(10*x^3+3*x^2+x+5)^2;

https://colab.research.google.com/drive/1H6aMHSZt5mxs5qeu4l462xSx4T1bewhu?usp=sharing

```
# Initialize global counter
fun_call = 0

def inter_hal_obj(x):
    """
    Objective function: (10x^3 + 3x^2 + x + 5)^2
    Updates the global call counter.
    """
    global fun_call
    fun_call += 1
    return (10*x**3 + 3*x**2 + x + 5)**2

# Algorithm Initialization
a = -3
b = 3
l = b - a
xm = (a + b) / 2
x1 = (a + xm) / 2
x2 = (xm + b) / 2

# Initial function evaluation
fm = inter_hal_obj(xm)
iteration = 1
fun_call = 1  # Matches the single call above (if fun_call wasn't reset)
```

```
while l > 1.0e-8:
    f1 = inter_hal_obj(x1)

    if f1 <= fm:
        b = xm
        fm = f1
    else:
        f2 = inter_hal_obj(x2)
        if f2 <= fm:
            a = xm
            fm = f2
        else:
            a = x1
            b = x2

    # Recalculate midpoints
    xm = (a + b) / 2
    x1 = (a + xm) / 2
    x2 = (xm + b) / 2
    l = b - a
    iteration += 1

# Print results
print(f"b         = {b:.4f}")
print(f"iter      = {iteration}")
print(f"fun_call = {fun_call}")
```

Remarks

At each stage of the algorithm, exactly half of the search is deleted.

At most two function evaluations are necessary at each iteration.

After n iterations, the initial search interval will be reduced to

According to Krefer, thethree point search is most efficient among all equal-interval searches.

$$\left(\frac{1}{2}\right)^n$$

Phase II- Interval Refinement Phase – The Golden Search (Non-equal Interval Search)

Assume that the total length of the region of search =1, two experiments are done at τ and 1-τ. One can compare the above two experiments and hence needs to delete either section between the end point and two trial points. Then one trial point is the new end point, the other is a new comparing point.

Problem: We want the original trial point to be a new trial point, i.e.,. It is possible to solve $\tau^2 = 1 - \tau$, τ=0.61803…

The Algorithm: **Golden-Search Method**

Step 1: Set $L = b-a$, $\tau = 0.61803\ldots$, $x2 = a+\tau L$, $x1 = a+(1-\tau)L$

Step 2: Find $f(x1)$, $f(x2)$, compare

If $f(x1) > f(x2) \Rightarrow a = x1$, $x1 \leftarrow x2$, go to step 3.

If $f(x1) < f(x2) \Rightarrow b = x2$, $x2 \leftarrow x1$, go to step 3.

Step 3: Set $L = b-a$, $x2 \leftarrow a+\tau L$, if (i) true $x1 \leftarrow a+(1-\tau)L$, if (ii) true. Go to step 2.

The Golden Search

https://colab.research.google.com/drive/1ELt-4O9XAx85oqPmDfpMG8kznu-aJgA7?usp=sharing

---

```python
import numpy as np

def goldsec(op2_func, tol, x0, d):
    """
    Golden Section Search for Line Search Optimization.

    Parameters:
    op2_func : callable
        The objective function to minimize.
    tol : float
        Tolerance for stopping criteria.
    x0 : array-like or float
        Current position.
    d : array-like or float
        Search direction.

    Returns:
    al_opt : float
        Optimal step size (alpha).
    """
    b = 1.0
    a = 0.0
    l = b - a
    tau = 0.61803

    # Calculate initial internal points
    x2 = a + tau * l
    x1 = a + (1 - tau) * l

    while l > tol:
        # Calculate actual points in the domain
        # Note: Using numpy allows direct vector addition: vector + scalar * vector
        xx1 = x0 + x1 * d
        xx2 = x0 + x2 * d

        # Evaluate function (equivalent to feval)
        y1 = op2_func(xx1)
        y2 = op2_func(xx2)

        # Update bounds based on function values
        if y1 >= y2:
            a = x1
            x1 = x2
            l = b - a
            x2 = a + tau * l
        else:
            b = x2
            x2 = x1
            l = b - a
            x1 = a + (1 - tau) * l

    al_opt = b
    return al_opt
```

```python
if __name__ == "__main__":
    # Define a simple objective function: f(x) = x^2 + 2x
    # We want to find the step size that minimizes this along a direction.
    def my_obj(x):
        return x**2 + 2*x

    # Setup parameters
    # If x0 and d are scalars:
    x_start = 0
    direction = 1
    tolerance = 1e-5

    # If x0 and d are vectors (requires numpy arrays for the math to work):
    # x_start = np.array([0, 0])
    # direction = np.array([1, 1])

    optimal_step = goldsec(my_obj, tolerance, x_start, direction)

    print(f"Optimal alpha: {optimal_step:.5f}")
```

Example : The Piping Problem

https://colab.research.google.com/drive/1IWRQAKQLgCSE9NfeP6tVWsGpEixElTRG?usp=sharing

---

```python
def obj_piping(D):
    """
    Objective function for piping optimization.
    Inputs:
        D: Diameter (in)
    """
    global fun_call
    fun_call += 1

    # Constants
    L = 1000.0  # ft
    Q = 20.0    # gpm

    # Calculate head loss (hp)
    # 4.4e-8*(L*Q^3)/(D^5) + 1.92e-9*(L*Q^2.68)/(D^4.68)
    term1 = 4.4e-8 * (L * Q**3) / (D**5)
    term2 = 1.92e-9 * (L * Q**2.68) / (D**4.68)
    hp = term1 + term2

    # Calculate Cost (y)
    # 0.45*L + 0.245*L*D^1.5 + 3.25*(hp)^0.5 + 61.6*(hp)^0.925 + 102
    y = (0.45 * L +
         0.245 * L * D**1.5 +
         3.25 * (hp)**0.5 +
         61.6 * (hp)**0.925 +
         102)

    return y
```

Remarks

At each stage, only one function evaluation (or one experiment) is needed.

The length of searching is narrowed by a factor of τ at each iteration, i.e.

Variable transformation technique may be useful for this algorithm, i.e. set the initial length equal to 1.

$$L_{N+1} / L_N = 0.618...$$

Remarks - Continued

Define $F_R = \frac{L_N}{L_1}$, *N* = number of experiments, or function evaluations.

$$F_R = \begin{cases} 0.5^{N/2} & \text{for interval halving} \\ 0.618^{N-1} & \text{for golden search} \end{cases}$$

Let *E = FR(N)*

$$N = \frac{2 \ln E}{\ln 0.5} \quad \text{for interval halving}$$

$$N = \frac{\ln E}{\ln 0.618} + 1 \quad \text{for gold search}$$

Method	E=0.1	E=0.05	E=0.01	E=0.001
I.H.	7	9	14	20
G.S.	6	8	11	16

Solution Methods (2) – Numerical Approaches
(iii) Polynomial Approximation Methods –Powell's Method

Powell's method is to approximate an objective function by a quadratic function such as $f(x)=ax^2+bx+c$, then it can be shown the optimum is located at $x^*=-b/2a$.

Given the above equation we need to do three experiments (function calls) to fit a quadratic function, let the three experiments (function calls) located at: $f(x_1)$, $f(x_2)$, $f(x_3)$ and let's rewrite the quadratic equation based on the new notation:

$$q(x) = a_0 + a_1(x - x_1) + a_2(x - x_1)(x - x_2)$$

Powell's Method- Continued

The parameters in the previous slide can be found using three experiments:

$$f(x_1) = f_1 = a_0$$

$$f(x_2) = f_2 = a_0 + a_1(x_2 - x_1) \Rightarrow a_1 = \frac{f_2 - f_1}{x_2 - x_1}$$

$$f(x_3) = f_3 = a_0 + a_1(x_3 - x_1) + a_2(x_3 - x_1)(x_3 - x_2) \Rightarrow a_2 = \frac{1_1}{x_3 - x_2}\left(\frac{f_3 - f_1}{x_3 - x_1} - \frac{f_2 - f_1}{x_2 - x_1}\right)$$

At the optimum point, it can be derived based on the above three experiments, such that

$$\frac{dq}{dx} = 0 = a_1 + a_2(x^* - x_2) + a_2(x^* - x_1) \qquad \text{or} \qquad x^* = \frac{x_2 + x_1}{2} - \frac{a_1}{2a_2}$$

Algorithm (Powell's Method)

Step 1: Given $x0$, Δx, $x1=x0+\Delta x$, ε, δ

Step 2: Evaluate $f(x0)$, $f(x1)$

If $f(x1)> f(x0)$, then $x2=x0-\Delta x$

If $f(x1)< f(x0)$, then $x2=x0+2\Delta x$

Step 3: find $f(x2)$

Step 4: Find $Fmin$=min ($f(x0)$, $f(x1)$, $f(x2)$)

$Xmin= x0, x1, x2,$ such that $f(x_{min})= F_{min}$, i=0,1,2.

Step 5: Get $a1$, $a2$

Step 6: Get x^*, find $f(x^*)$.

Step 7: Check if (i) $\left|F_{\min} - f(x^*)\right| \le \varepsilon$ or (ii) $\left|x^* - x_{\min}\right| \le \delta$ Yes, stop.

No, continue

Step 8: set $x2 \leftarrow x^*$, $x1 \leftarrow x\min$, $x0 \leftarrow$ one of $x0, x1, x2$ not $x\min$

Go to step 4.

Powell's Method

https://colab.research.google.com/drive/1dUfpO_TMuasJ9JS_jdgYLCDGCl0m29eX?usp=sharing

```python
import numpy as np

def one_dim_pw(xx, s, op2_func):
    """
    Powell's Quadratic Interpolation Method (One-Dimensional Search)

    Parameters:
    xx : numpy array
        Current starting point (vector).
    s : numpy array
        Search direction (vector).
    op2_func : function
        Objective function to minimize.
    """

    dela = 0.005
    alp0 = 0.01

    # Initialize arrays for 3 points
    alpha = [0.0] * 3
    y = [0.0] * 3

    # --- Point 1 ---
    alpha[0] = alp0
    x1 = xx + alpha[0] * s
    y[0] = op2_func(x1)

    # --- Point 2 ---
    alpha[1] = alpha[0] + dela
    x2 = xx + alpha[1] * s
    y[1] = op2_func(x2)

    # --- Bracketing Logic ---
    # Determine the location of the 3rd point based on the slope
    if y[1] >= y[0]:
        # Function is increasing, go backward
        alpha[2] = alpha[0] - dela
    else:
        # Function is decreasing, go forward further
        alpha[2] = alpha[0] + 2 * dela

    eps = 100.0
    delta = 100.0

    # --- Optimization Loop ---
    while eps > 0.001 or delta > 0.001:

        # Evaluate Point 3
        x3 = xx + s * alpha[2]
        y[2] = op2_func(x3)

        fmin = min(y)

        a0, a1, a2 = alpha[0], alpha[1], alpha[2]
        f0, f1, f2 = y[0], y[1], y[2]

        # Denominator for quadratic interpolation formula
        denom = 2 * ((a1 - a2)*f0 + (a2 - a0)*f1 + (a0 - a1)*f2)

        if denom == 0:
            # Avoid division by zero if points are collinear
            break

        # Numerator
        num = (a1**2 - a2**2)*f0 + (a2**2 - a0**2)*f1 + (a0**2 - a1**2)*f2

        # Optimal alpha (vertex of the parabola)
        alpha_opt = num / denom

        # --- Update Stopping Criteria ---
        # eps: change in the variable (alpha)
        # delta: change in the function value

        # Find index of the best point so far
        best_idx = np.argmin(y)
        best_alpha = alpha[best_idx]
        best_y = y[best_idx]

        eps = abs(alpha_opt - best_alpha)

        # Evaluate function at the new optimal prediction
        x_opt = xx + s * alpha_opt
        y_opt = op2_func(x_opt)
        delta = abs(best_y - y_opt)

        # --- Prepare for Next Iteration ---
        # Replace the point with the highest function value (worst point)
        # with the new optimal point to refine the parabola.
        worst_idx = np.argmax(y)
        alpha[worst_idx] = alpha_opt
        y[worst_idx] = y_opt

    return best_alpha
```

Powell's Method

https://colab.research.google.com/drive/1dUfpO_TMuasJ9JS_jdgYLCDGCl0m29eX?usp=sharing

---

```
# --- Usage Example ---
if __name__ == "__main__":
    # Define a test function: f(x) = x1^2 + x2^2 - 4x1 - 4x2
    # Minimum is at [2, 2]
    def my_obj(x):
        return x[0]**2 + x[1]**2 - 4*x[0] - 4*x[1]

    # Starting point and direction
    x_start = np.array([0.0, 0.0])
    search_dir = np.array([1.0, 1.0]) # Searching along the diagonal

    al_opt = one_dim_pw(x_start, search_dir, my_obj)

    print(f"Optimal Step Size (alpha): {al_opt:.4f}")
    print(f"New Point: {x_start + al_opt * search_dir}")
```

Example: Piping Design

https://colab.research.google.com/drive/1nVGqN1fWdCWnz4ljvv8c0LpTydFCjneV?usp=sharing

```
def obj_piping(D):
    """
    Objective function for piping optimization.
    Updates the global call counter.
    """
    global fun_call
    fun_call += 1

    # Constants
    L = 1000.0  # ft
    Q = 20.0    # gpm

    # Calculate head loss (hp)
    # 4.4e-8*(L*Q^3)/(D^5) + 1.92e-9*(L*Q^2.68)/(D^4.68)
    term1 = 4.4e-8 * (L * Q**3) / (D**5)
    term2 = 1.92e-9 * (L * Q**2.68) / (D**4.68)
    hp = term1 + term2

    # Calculate Cost (y)
    # 0.45*L + 0.245*L*D^1.5 + 3.25*(hp)^0.5 + 61.6*(hp)^0.925 + 102
    y = (0.45 * L +
         0.245 * L * D**1.5 +
         3.25 * (hp)**0.5 +
         61.6 * (hp)**0.925 +
         102)

    return y
```

Comparison – Interval_halving (tol=1.e-6)

l = 6.8545e-007

b = 0.8250

a = 0.8250

iter = 24

fun_call = 63

Any Questions?