Importing the necessary libraries

In [8]:
```
import itertools
import sympy as sym
```

Generalized n variable Template Function for any given mathematical problem with given constraints

Generalized Optimizer - Lagrangian Solver - All feasible solutions are given along with local minima/maxima

Input:

1. Minimize/Maximize Objective Function

$$f(x_1, x_2, \ldots, x_n)$$

2. Equality Constraints -

$$h_i(x_1, x_2, \ldots, x_n) = 0 \quad i = 1, 2, \ldots, p$$

3. Inequality Constraints

$$g_j(x_1, x_2, \ldots, x_n) \leq 0 \quad j = 1, 2, \ldots, m$$

Solver Variables:

1. Lagrange Function - $L(f, x, \lambda, h, \mu, g, s)$

$$L(f, x, \lambda, h, \mu, g, s) = f(x) + \sum_{i=1}^{p} \lambda_i h_i(x) + \sum_{j=1}^{m} \mu_i (g_j(x) + s_j^2)$$

2. Lagrange Multipliers - λ_i, μ_j, s_j

$$\mu_j \geq 0, s_j \geq 0 \quad j = 1, 2, \ldots, m$$

Necessary Conditions (Stationarity):

1. Lagrange Equation

$$\frac{\partial L}{\partial x_k} = 0 \quad k = 1, 2, \ldots, n$$

2. KKT Conditions

$$\frac{\partial L}{\partial \lambda_i} = 0 \quad i = 1, 2, \ldots, p$$

$$\frac{\partial L}{\partial \mu_j} = 0 \quad j = 1, 2, \ldots, m$$

$$\frac{\partial L}{\partial s_j} = 0 \quad j = 1, 2, \ldots, m$$

Feasibility Conditions:

$$s_j^2 \geq 0 \quad j = 1, 2, \ldots, m$$

$$g_j(x) \leq 0 \quad j = 1, 2, \ldots, m$$

Switching/Orthogonality Conditions:

$$\mu_j \times g_j(x) = 0 \quad j = 1, 2, \ldots, m$$

Sufficient Conditions:

$$\exists \lambda^* = [\lambda_1^*, \lambda_2^*, \ldots, \lambda_p^*] \quad \exists \mu^* = [\mu_1^*, \mu_2^*, \ldots, \mu_m^*]$$

$$\nabla_x^2 f(x^*) + \sum_{i=1}^{p} \lambda_i^* \nabla_x^2 h_i(x^*) + \sum_{j=1}^{m} \mu_j^* \nabla_x^2 g_j(x^*) \geq 0$$

In [59]:
```
def solve_lagrangian(f, h, g, side_constraints):

    '''
    This function solves the Constrained Optimization Problem using the Lagr

    f : sympy expression

    h : list of sympy expressions (Equality Constraints)

    g : list of sympy expressions (Inequality Constraints)

    side_constraints : list of sympy expressions (Side Constraints)

    '''

    # Type check if f, h, g are sympy expressions
    if not isinstance(f, sym.Expr) or not isinstance(h, list) or not isinsta
        raise Exception("f, h, g must be sympy expressions")

    # Get Free Symbols from f, h, g
    symbols = list(set(f.free_symbols).union(*[set(h_i.free_symbols) for h_i

    # Find number of equality constraints
    p = len(h)

    # Creating Lambda matrix
    l = sym.Matrix([sym.symbols(f"l_{i}") for i in range(p)])
```

```python
    # Find number of inequality constraints
    m = len(g)

    # Creating Mu matrix
    u = sym.Matrix([sym.symbols(f"u_{j}") for j in range(m)])

    # Creating S matrix
    s = sym.Matrix([sym.symbols(f"s_{j}") for j in range(m)])

    # Writing Lagrangian
    L = f + sum([l[i]*h[i] for i in range(p)]) + sum([u[j]*(g[j] + s[j]**2)f

    # Finding the partial derivatives of the Lagrangian
    L_x = sym.Matrix([sym.diff(L, x) for x in symbols])

    # Finding Hessians of f, every h_i, and every g_j
    H_f = sym.hessian(f, symbols)
    H_h = [sym.hessian(h_i, symbols) for h_i in h]
    H_g = [sym.hessian(g_j, symbols) for g_j in g]

    # Now we need Switching Conditions
    # We will have 2^m cases

    # First we need to find all possible combinations of m binary variables
    # We will use 0 to represent m_i = 0 and 1 to represent m_i > 0
    # We will use 0 to represent g_i = 0 and 1 to represent g_i < 0

    # All possible combinations of m binary variables
    all_combinations = list(itertools.product([0, 1], repeat=m))

    # Defining cases and their checks
    cases = []
    case_checks = []

    for combination in all_combinations:

        # 0 represents m_i = 0 i.e. g_i < 0 and 1 represents m_i > 0 i.e. g_
        curr_case = []
        curr_checks = []

        for i in range(m):
            if combination[i] == 0:
                curr_case.append(sym.Eq(u[i], 0))
                curr_checks.append(g[i] < 0)
            else:
                curr_case.append(sym.Eq(g[i], 0))
                curr_checks.append(u[i] > 0)

        # Adding side constraints
        curr_checks += side_constraints

        # Append Lagrandian partial derivative equations
        curr_case += [sym.Eq(L_x[i], 0) for i in range(len(symbols))]

        # Add h_i = 0 equations
        curr_case += [sym.Eq(h_i, 0) for h_i in h]
```

```
        # Adding the case and its checks
        cases.append(curr_case)
        case_checks.append(curr_checks)

    # Let's solve all the cases and check if they are feasible
    solutions = []

    symbols_to_solve = symbols.copy()
    symbols_to_solve.extend(l)
    symbols_to_solve.extend(u)

    for i in range(len(cases)):

        # Solving the current case
        curr_solution = sym.solve(cases[i], symbols_to_solve)

        # Check if curr_solution is a list of tuples
        if isinstance(curr_solution, list):
            # There can be multiple solutions
            # We will try all the solutions and check if they are feasible
            for solution in curr_solution:

                # Flag to check if solution is a real number
                flag = True

                # Converting the solution to a dictionary
                sol_dict = {}
                for j in range(len(symbols_to_solve)):

                    # Checking if the solution[j] is a Complex Number
                    if sym.im(solution[j]) != 0:
                        flag = False
                        break

                    sol_dict[symbols_to_solve[j]] = solution[j]

                if not flag:
                    solutions.append(None)
                    continue

                # Checking if the current solution is feasible
                curr_checks_res = [check.subs(sol_dict) for check in case_ch

                solutions.append(solution if all(curr_checks_res) else None)

        else:

            # Single solution

            # Flag to check if solution is a real number
            flag = True

            # Curr_soln is a dictionary
            for sol in curr_solution.values():
                if sym.im(sol) != 0:
```

```
                    flag = False
                    break

            if not flag:
                solutions.append(None)
                continue

            curr_checks_res = [check.subs(curr_solution) for check in case_c

            solutions.append(curr_solution if all(curr_checks_res) else None

    # Now we need to check if the solutions are local minima or local maxima
    # We will use the following conditions
    # H_f + sum(l_i*H_h_i) + sum(u_j*H_g_j) is positive semidefinite
    for i in range(len(solutions)):

        if solutions[i] is not None:

            # Substituting the solution in the Hessian and adding all the He
            R = H_f.copy()
            for i in range(p):
                R += l[i]*H_h[i]
            for j in range(m):
                R += u[j]*H_g[j]

            R = R.subs(solutions[i])

            # Checking if the Hessian is positive semidefinite
            solutions[i]["type"] = "max" if not R.is_positive_semidefinite e

            # Add F-value to the solution
            solutions[i]["F"] = f.subs(solutions[i])

    # Return Non-None Solutions
    return [solution for solution in solutions if solution is not None]
```

Example 1:

Minimize - $f(x_1, x_2) = x_1^2 + x_2^2 - 4x_1 - 6x_2$ subject to $x_1 + x_2 - 2 \leq 0$ and $2x_1 + 3x_2 - 12 \leq 0$ and $x_1 \geq 0$ and $x_2 \geq 0$

In [60]:
```
x1, x2 = sym.symbols("x1 x2")
f = x1**2 + x2**2 - 4*x1 - 6*x2
g = [x1 + x2 - 2, 2*x1 + 3*x2 - 12]
h = []
side_constraints = [x1 >= 0, x2 >= 0]

solve_lagrangian(f, h, g, side_constraints)
```

Out[60]:
```
[{x2: 3/2, x1: 1/2, u_0: 3, u_1: 0, 'type': 'min', 'F': -17/2}]
```

Example 2:

Minimize - $f(x_1, x_2) = (x_1 - 2.5)^2 + (x_2 - 2.5)^2$ subject to $2x_1 + 2x_2 - 3 \leq 0$ and $x_1 \geq 0$ and $x_2 \geq 0$

In [61]:
```
x1, x2 = sym.symbols("x1 x2")
f = (x1 - 2.5)**2 + (x2 - 2.5)**2
h = []
g = [2*x1 + 2*x2 - 3]
side_constraints = [x1 >= 0, x2 >= 0]

solve_lagrangian(f, h, g, side_constraints)
```

Out[61]:
```
[{x2: 0.750000000000000,
  x1: 0.750000000000000,
  u_0: 1.75000000000000,
  'type': 'min',
  'F': 6.12500000000000}]
```

Example 3:

Minimize - $f(x_1, x_2) = (x_1 - 2.5)^2 + (x_2 - 2.5)^2$ subject to $2x_1 + 2x_2 = 3$ and $x_1 \geq 0$ and $x_2 \geq 0$

In [62]:
```
x1, x2 = sym.symbols("x1 x2")
f = (x1 - 2.5)**2 + (x2 - 2.5)**2
g = []
h = [2*x1 + 2*x2 - 3]
side_constraints = [x1 >= 0, x2 >= 0]

solve_lagrangian(f, h, g, side_constraints)
```

Out[62]:
```
[{x2: 0.750000000000000,
  x1: 0.750000000000000,
  l_0: 1.75000000000000,
  'type': 'min',
  'F': 6.12500000000000}]
```

Example 4:

Minimize - $f(x_1, x_2, x_3) = x_1^2 + 2x_1x_2 + 3x_2^2 + 4x_2x_3 + x_3^2 - 6x_3$

In [65]:
```
x1, x2, x3 = sym.symbols("x1 x2 x3")
f = x1**2 + 2*x1*x2 + 3*x2**2 + 4*x2*x3 + x3**2 - 6*x3
g = []
h = []
side_constraints = []

solve_lagrangian(f, h, g, side_constraints)
```

Out[65]:
```
[{x3: -3, x2: 3, x1: -3, 'type': 'max', 'F': 9}]
```

Example 5:

Minimize - $f(x_1, x_2) = x_1^2 + 2x_2^2 - 3x_1 - 6x_2$ subject to $x_1 + x_2 \leq 3$ and $x_1 + 3x_2 \leq 10$ and $x_2 \geq 0$

In [66]:
```
x1, x2 = sym.symbols("x1 x2")
f = x1**2 + 2*x2**2 - 3*x1 - 6*x2
g = [x1 + x2 - 3, x1 + 3*x2 - 10]
h = []
side_constraints = [x2 >= 0]

solve_lagrangian(f, h, g, side_constraints)
```

Out[66]:
```
[]
```

If you wish to test more examples, you can input you functions f, g, h and run it on this code.

Solving Example 1 Step-by-Step to show the execution of the code

Problem Statement:

Write a code to minimise $f(x) = x_1^2 + x_2^2 - 4x_1 - 6x_2$ and subject to the constraints $x_1 + x_2 \leq 2$ and $2x_1 + 3x_2 \leq 12$ and $x_1 \geq 0$ and $x_2 \geq 0$.

Solution:

We will use the **Lagrange Multiplier Method** to solve this problem.

Step 1: We will first define the function $f(x)$ and the constraints $g(x)$ and $h(x)$.

Step 2: We will then define the Lagrangian function $L(x, \lambda, \mu)$.

Step 3: We will then find the partial derivatives of the Lagrangian function with respect to x_1, x_2 and put them equal to 0 to get some conditions.

Step-4: We will create switching conditions using the equation: $\mu_j g_j(x) = 0$.

Step-5: Solve all switch cases using Constraints, and another inequality which tells $\mu_j \geq 0$.

Step-6: We will then compare solutions of all switch cases and find the optimal solution.

Code

Step-1: Define the function $f(x)$ and the constraints $g(x)$ and $h(x)$.

In [12]:
```
x1, x2, l1, l2, m1, m2 = sym.symbols('x1 x2 l1 l2 m1 m2')

f = x1**2 + x2**2 - 4*x1 - 6*x2

g_1 = x1 + x2 - 2

g_2 = 2*x1 + 3*x2 - 12
```

Step-2: Define the Lagrangian function $L(x, \lambda, \mu)$.

$$L(x, \lambda, \mu) = f(x) + \sum_{i=1}^{p} \lambda_i h_i(x) + \sum_{j=1}^{m} \mu_i (g_j(x) + s_j^2)$$

$$L(x, \lambda, \mu) = x_1^2 + x_2^2 - 4x_1 - 6x_2 + \mu_1(x_1 + x_2 - 2 + s_1^2) + \mu_2(2x_1 + 3x_2 - 12 + s_2^2)$$

(We do not need to consider λ as we do not have equality constraints.)

(Also we do not need to code this function as we will be using the partial derivatives of this function.)

Step-3: Find the partial derivatives of the Lagrangian function with respect to x_1, x_2 and put them equal to 0 to get some conditions.

$$\frac{\partial L}{\partial x_1} = 2x_1 - 4 + \mu_1 + 2\mu_2 = 0$$

$$\frac{\partial L}{\partial x_2} = 2x_2 - 6 + \mu_1 + 3\mu_2 = 0$$

In [13]:
```
# Lagrangian Derivatives
L = f + m1*g_1 + m2*g_2

L_x1 = sym.diff(L, x1)
L_x2 = sym.diff(L, x2)

L_x1, L_x2
```

Out[13]:
```
(m1 + 2*m2 + 2*x1 - 4, m1 + 3*m2 + 2*x2 - 6)
```

Step-4: Create switching conditions using the equation: $\mu_j g_j(x) = 0$.

$\mu_1 g_1(x) = 0$ and $\mu_2 g_2(x) = 0$

CASE 1: $\mu_1 = 0$ ($g_1(x) < 0$) and $\mu_2 = 0$ ($g_2(x) < 0$)

CASE 2: $\mu_1 = 0$ ($g_1(x) < 0$) and $g_2(x) = 0$ ($\mu_2 > 0$)

CASE 3: $g_1(x) = 0$ ($\mu_1 > 0$) and $\mu_2 = 0$ ($g_2(x) < 0$)

CASE 4: $g_1(x) = 0$ ($\mu_1 > 0$) and $g_2(x) = 0$ ($\mu_2 > 0$)

In [14]:
```
# Switching Conditions
case_1 = [sym.Eq(m1, 0), sym.Eq(m2, 0)]
case_1_checks = [g_1 < 0, g_2 < 0, x1 >= 0, x2 >= 0]

case_2 = [sym.Eq(m1, 0), sym.Eq(g_2, 0)]
case_2_checks = [g_1 < 0, m2 > 0, x1 >= 0, x2 >= 0]

case_3 = [sym.Eq(g_1, 0), sym.Eq(m2, 0)]
case_3_checks = [m1 > 0, g_2 < 0, x1 >= 0, x2 >= 0]

case_4 = [sym.Eq(g_1, 0), sym.Eq(g_2, 0)]
case_4_checks = [m1 > 0, m2 > 0, x1 >= 0, x2 >= 0]

case_1, case_1_checks, case_2, case_2_checks, case_3, case_3_checks, case_4,
```

Out[14]:
```
([Eq(m1, 0), Eq(m2, 0)],
 [x1 + x2 - 2 < 0, 2*x1 + 3*x2 - 12 < 0, x1 >= 0, x2 >= 0],
 [Eq(m1, 0), Eq(2*x1 + 3*x2 - 12, 0)],
 [x1 + x2 - 2 < 0, m2 > 0, x1 >= 0, x2 >= 0],
 [Eq(x1 + x2 - 2, 0), Eq(m2, 0)],
 [m1 > 0, 2*x1 + 3*x2 - 12 < 0, x1 >= 0, x2 >= 0],
 [Eq(x1 + x2 - 2, 0), Eq(2*x1 + 3*x2 - 12, 0)],
 [m1 > 0, m2 > 0, x1 >= 0, x2 >= 0])
```

Step-5: Solve all switch cases using Constraints, and another inequality which tells $\mu_j \geq 0$.

CASE 1:

$\mu_1 = 0$ ($g_1(x) < 0$) and $\mu_2 = 0$ ($g_2(x) < 0$)

$$2x_1 + 3x_2 - 12 = 0$$

$$x_1 + x_2 - 2 = 0$$

$$x_1 \geq 0$$

$$x_2 \geq 0$$

$$\mu_1 \geq 0$$

$$\mu_2 \geq 0$$

In [15]:
```
# Case 1
case_1_constraints = [sym.Eq(L_x1, 0), sym.Eq(L_x2, 0)] + case_1
case_1_solution = sym.solve(case_1_constraints, [x1, x2, m1, m2])

case_1_solution
```

Out[15]:
```
{x1: 2, x2: 3, m1: 0, m2: 0}
```

In [16]:
```
# Perform checks
case_1_checks_res = [check.subs(case_1_solution) for check in case_1_checks]

if all(case_1_checks_res):
    print('Case 1 is valid')
```

```python
else:
    print('Case 1 is invalid')
```

```
Case 1 is invalid
```

Case 1 is not feasible

CASE 2:

$\mu_1 = 0$ ($g_1(x) < 0$) and $g_2(x) = 0$ ($\mu_2 > 0$)

$$2x_1 + 3x_2 - 12 = 0$$

$$x_1 + x_2 - 2 = 0$$

$$x_1 \geq 0$$

$$x_2 \geq 0$$

$$\mu_1 \geq 0$$

$$\mu_2 \geq 0$$

In [17]:
```python
# Case 2
case_2_constraints = [sym.Eq(L_x1, 0), sym.Eq(L_x2, 0)] + case_2

# Solve Case 2 and find x1, x2, m1, m2
case_2_solution = sym.solve(case_2_constraints, [x1, x2, m1, m2])

case_2_solution
```

Out[17]:
```
{x1: 24/13, x2: 36/13, m1: 0, m2: 2/13}
```

In [18]:
```python
# Perform checks for Case 2
case_2_checks_res = [check.subs(case_2_solution) for check in case_2_checks]

if all(case_2_checks_res):
    print('Case 2 is valid')
else:
    print('Case 2 is invalid')
```

```
Case 2 is invalid
```

Case 2 is not feasible

CASE 3:

$g_1(x) = 0$ ($\mu_1 > 0$) and $\mu_2 = 0$ ($g_2(x) < 0$)

$$2x_1 + 3x_2 - 12 = 0$$

$$x_1 + x_2 - 2 = 0$$

$$x_1 \geq 0$$

$$x_2 \geq 0$$

$$\mu_1 \geq 0$$

$$\mu_2 \geq 0$$

In [19]:
```
# Case 3
case_3_constraints = [sym.Eq(L_x1, 0), sym.Eq(L_x2, 0)] + case_3

# Solve Case 3 and find x1, x2, m1, m2
case_3_solution = sym.solve(case_3_constraints, [x1, x2, m1, m2])

case_3_solution
```

Out[19]: {x1: 1/2, x2: 3/2, m1: 3, m2: 0}

In [20]:
```
# Perform checks for Case 3
case_3_checks_res = [check.subs(case_3_solution) for check in case_3_checks]

if all(case_3_checks_res):
    print('Case 3 is valid')
else:
    print('Case 3 is invalid')
```

```
Case 3 is valid
```

Since Case 3 is feasible, we will solve it.

Finding the function value at $x_1, x_2 = (\frac{1}{2}, \frac{3}{2})$

In [21]:
```
f_val = f.subs(case_3_solution)

f_val
```

Out[21]: $-\frac{17}{2}$

The function value is -8.5

Checking for local minima/maxima.

To check this, we will use the following condition:

$$\nabla^2 f(x) + \mu_1 \nabla^2 g_1(x) + \mu_2 \nabla^2 g_2(x) \geq 0$$

In [22]:
```
# Checking if the solution is a local minimum or a local maximum
F = sym.hessian(f, [x1, x2])
G1 = sym.hessian(g_1, [x1, x2])
G2 = sym.hessian(g_2, [x1, x2])
R = F + m1*G1 + m2*G2

R.subs(case_3_solution)

R
```

Out[22]: $\begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix}$

In [23]:
```
F, G1, G2, R
```

Out[23]:
```
(Matrix([
 [2, 0],
 [0, 2]]),
 Matrix([
 [0, 0],
 [0, 0]]),
 Matrix([
 [0, 0],
 [0, 0]]),
 Matrix([
 [2, 0],
 [0, 2]]))
```

In [24]:
```
if sym.Matrix(R.subs(case_3_solution)).is_positive_semidefinite:
    print('Case 3 is a local minimum')
else:
    print('Case 3 is a local maximum')
```

```
Case 3 is a local minimum
```

The solution is a local minimum.

We cannot say that this is the true minimum value of this function using this method because we have not checked for all the possible solutions till now.

CASE 4:

$g_1(x) = 0$ ($\mu_1 > 0$) and $g_2(x) = 0$ ($\mu_2 > 0$)

$$2x_1 + 3x_2 - 12 = 0$$

$$x_1 + x_2 - 2 = 0$$

$$x_1 \geq 0$$

$$x_2 \geq 0$$

$$\mu_1 \geq 0$$

$$\mu_2 \geq 0$$

In [25]:
```
# Case 4
case_4_constraints = [sym.Eq(L_x1, 0), sym.Eq(L_x2, 0)] + case_4

# Solve Case 4 and find x1, x2, m1, m2
case_4_solution = sym.solve(case_4_constraints, [x1, x2, m1, m2])

case_4_solution
```

Out[25]:
```
{x1: -6, x2: 8, m1: 68, m2: -26}
```

In [26]:
```
# Perform checks for Case 4
case_4_checks_res = [check.subs(case_4_solution) for check in case_4_checks]

if False in case_4_checks_res:
    print('Case 4 is not feasible')
else:
    print('Case 4 is feasible')
```

```
Case 4 is not feasible
```

Case 4 is not feasible

Step-6: Find the true minimum value returned by this function by checking all the possible solutions.

Since, we have gotten only one feasible situation, and that **solution is a local minimum, we can say that this is the true minimum value of this function.**

Hence, the **true minimum value of this function is -8.5 at** $x_1, x_2 = (\frac{1}{2}, \frac{3}{2})$