## 0. Imports and Setting up Anthropic API Client

```
from google.colab import drive

drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

```
!pip install python-dotenv

import os
import dotenv

dotenv.load_dotenv('/content/drive/MyDrive/.env')
```

```
Collecting python-dotenv
  Downloading python_dotenv-1.0.1-py3-none-any.whl (19 kB)
Installing collected packages: python-dotenv
Successfully installed python-dotenv-1.0.1
True
```

```
# Load Prompts and Problem Description
# Variables Prompt
prompt11_path = '/content/drive/MyDrive/Thesis/Prompts/Prompt11_MathematicalModel.txt'

# Objective Prompt
prompt12_path = '/content/drive/MyDrive/Thesis/Prompts/Prompt12_MathematicalModel.txt'

# Constraint Prompt
prompt13_path = '/content/drive/MyDrive/Thesis/Prompts/Prompt13_MathematicalModel.txt'

# Code Prompt
prompt2_path = '/content/drive/MyDrive/Thesis/Prompts/Prompt2_PyomoCode.txt'
problem_desc_path = '/content/drive/MyDrive/Thesis/ProblemDescriptions/NL/NL2.txt'

prompt11_file = open(prompt11_path, "r")
prompt12_file = open(prompt12_path, "r")
prompt13_file = open(prompt13_path, "r")
prompt2_file = open(prompt2_path, "r")
problem_desc_file = open(problem_desc_path, "r")

prompt11 = prompt11_file.read()
print("Prompt 1.1 (Variables):\n", prompt11)

prompt12 = prompt12_file.read()
print("Prompt 1.2 (Objctive):\n", prompt12)

prompt13 = prompt13_file.read()
print("Prompt 1.3 (Constraints):\n", prompt13)

prompt2 = prompt2_file.read()
print("Prompt 2:\n", prompt2)

problem_desc = problem_desc_file.read()
print("Problem Description:\n", problem_desc)
```

```
Prompt 1.1 (Variables):
 Please formulate only the variables for this mathematical optimization problem.
Prompt 1.2 (Objctive):
 Please formulate only the objective function for this mathematical optimization problem.
Prompt 1.3 (Constraints):
 Please formulate only the constraints for this mathematical optimization problem.
Prompt 2:
 Please write a python pyomo code for this optimization problem.
Use sample data where needed.
Indicate where you use sample data.
Problem Description:
 You are a chemist in charge of a process that requires two liquids to reach the end product. Your goal is to maximize t
```

```
!pip install anthropic
```

Show hidden output

```
# Importing Anthropic & Setting Headers
import anthropic

client = anthropic.Anthropic(
    # defaults to os.environ.get("ANTHROPIC_API_KEY")
    api_key=os.environ.get("ANTHROPIC_API_KEY"),
)
```

## 1. Prompt 1.1: Create Variables for Mathematical Model

```
message11 = client.messages.create(
    model="claude-3-opus-20240229",
    system=prompt11,
    max_tokens=4096,
    messages=[
        {"role": "user", "content": problem_desc}
    ]
)

response_p11 = message11.content[0].text


# Print response
print(response_p11)
```

```
To formulate the mathematical optimization model, let's define the variables:

Let x₁ be the number of litres used of the first liquid component.
Let x₂ be the number of litres used of the second liquid component.

Now, we can formulate the optimization problem based on the given information:

Objective function:
Maximize f(x₁, x₂) = ln(x₁ · x₂) - ³√(x₁ + 2x₂)

Constraints:
1. Stability constraint: (x₁ - (x₁ + x₂)/2)² + (x₂ - (x₁ + x₂)/2)² ≤ 10
2. Mixture tank capacity constraint: x₁ + x₂ ≤ 80
3. Non-negativity constraints: x₁ ≥ 0, x₂ ≥ 0

Therefore, the complete mathematical optimization model is:

Maximize f(x₁, x₂) = ln(x₁ · x₂) - ³√(x₁ + 2x₂)
Subject to:
1. (x₁ - (x₁ + x₂)/2)² + (x₂ - (x₁ + x₂)/2)² ≤ 10
2. x₁ + x₂ ≤ 80
3. x₁ ≥ 0
4. x₂ ≥ 0
```

## 1. Prompt 1.2: Create Objective for Mathematical Model

```
message12 = client.messages.create(
    model="claude-3-opus-20240229",
    system=prompt12,
    max_tokens=4096,
    messages=[
        {"role": "user", "content": (problem_desc + response_p11)}
    ]
)

response_p12 = message12.content[0].text


# Print response
print(response_p12)
```

```
The objective function for this mathematical optimization problem is:

Maximize f(x₁, x₂) = ln(x₁ · x₂) - ³√(x₁ + 2x₂)

where:
x₁ is the number of litres used of the first liquid component
x₂ is the number of litres used of the second liquid component
ln() represents the Napierian logarithm (natural logarithm)
³√() represents the cube root
```

## 1. Prompt 1.3: Create Constraints for Mathematical Model

```
message13 = client.messages.create(
    model="claude-3-opus-20240229",
    system=prompt13,
    max_tokens=4096,
    messages=[
        {"role": "user", "content": (problem_desc + response_p11 + response_p12)}
    ]
)

response_p13 = message13.content[0].text


# Print response
print(response_p13)
```

```
The constraints for this mathematical optimization problem are:

1. Stability constraint:
   (x₁ - (x₁ + x₂)/2)² + (x₂ - (x₁ + x₂)/2)² ≤ 10
   This constraint ensures that the squared difference between the litres used of each component and the mean litres use

2. Mixture tank capacity constraint:
   x₁ + x₂ ≤ 80
   This constraint ensures that the total litres of both components used does not exceed the mixture tank capacity of 80

3. Non-negativity constraints:
   x₁ ≥ 0
   x₂ ≥ 0
   These constraints ensure that the number of litres used for each component is non-negative.
```

## 2. Prompt 2: Write Code based on Mathematical Model

In case more than the component of each step was generated, use "last instance"/ "newest version" of the components.

Example: Prompt 1.1 (Variables): Model generates only variables Prompt 1.2 (Objectvie): Model generates objective and constraints Prompt 1.3 (Constraints): Model generates only constraints

Use Variables from step 1.1, use objective from 1.2 and use constraints from 1.3 (since constraints version 1.3 is more recent than constraints version 1.2).

This set up allows for potential self-correction of the model throughout the three prompts.

Correctness of each component is still generated in the respective step, so self-correction is independent of this.

```
response_p11_adjusted = """
To formulate the mathematical optimization model, let's define the variables:

Let x₁ be the number of litres used of the first liquid component.
Let x₂ be the number of litres used of the second liquid component.
"""


message2 = client.messages.create(
    model="claude-3-opus-20240229",
    system=prompt2,
    max_tokens=4069,
    messages=[
        {"role": "user", "content": (response_p11_adjusted + response_p12 + response_p13)}
    ]
)

response_p2 = message2.content[0].text


# Print response
print(response_p2)
```

~~~
Here's the Python Pyomo code for the optimization problem:

```python
from pyomo.environ import *
from math import log, pow

# Create a Pyomo model
model = ConcreteModel()

# Define the decision variables
model.x1 = Var(domain=NonNegativeReals)
model.x2 = Var(domain=NonNegativeReals)

# Define the objective function
~~~

```
    def objective_rule(model):
        return log(model.x1 * model.x2) - pow(model.x1 + 2 * model.x2, 1/3)
    model.objective = Objective(rule=objective_rule, sense=maximize)

    # Define the stability constraint
    def stability_constraint_rule(model):
        return (model.x1 - (model.x1 + model.x2) / 2)**2 + (model.x2 - (model.x1 + model.x2) / 2)**2 <= 10
    model.stability_constraint = Constraint(rule=stability_constraint_rule)

    # Define the mixture tank capacity constraint
    def capacity_constraint_rule(model):
        return model.x1 + model.x2 <= 80
    model.capacity_constraint = Constraint(rule=capacity_constraint_rule)

    # Solve the optimization problem
    solver = SolverFactory('ipopt')  # Using the 'ipopt' solver
    solver.solve(model)

    # Print the optimal solution
    print("Optimal solution:")
    print("x1 =", model.x1.value)
    print("x2 =", model.x2.value)
    print("Objective value =", model.objective())
    ```

    In this code, we define the decision variables `x1` and `x2` as non-negative real variables using `Var(domain=NonNegativ

    The objective function is defined using the `objective_rule` function, which implements the equation `ln(x₁ · x₂) - ³√(x

    The stability constraint is defined using the `stability_constraint_rule` function, which implements the constraint `(x₁

    The mixture tank capacity constraint is defined using the `capacity_constraint_rule` function, which implements the cons

    The non-negativity constraints are automatically enforced by specifying the domain of the variables as `NonNegativeReals

    The 'ipopt' solver is used to solve the optimization problem. You may need to install the 'ipopt' solver separately if i

    Note: This code does not use any sample data as the problem statement does not provide specific values for the parameter
```

## 4. Input Problem Data and Test Model Code

```
# Installing pyomo nd solver
!pip install -q pyomo
!wget -N -q "https://matematica.unipv.it/gualandi/solvers/ipopt-linux64.zip"
!unzip -o -q ipopt-linux64
```

```
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 12.8/12.8 MB 14.1 MB/s eta 0:00:00
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 49.6/49.6 kB 2.3 MB/s eta 0:00:00
```

```
from pyomo.environ import *
from math import log, pow

# Create a Pyomo model
model = ConcreteModel()

# Define the decision variables
model.x1 = Var(domain=NonNegativeReals)
model.x2 = Var(domain=NonNegativeReals)

# Define the objective function
def objective_rule(model):
    return log(model.x1 * model.x2) - pow(model.x1 + 2 * model.x2, 1/3)
model.objective = Objective(rule=objective_rule, sense=maximize)

# Define the stability constraint
def stability_constraint_rule(model):
    return (model.x1 - (model.x1 + model.x2) / 2)**2 + (model.x2 - (model.x1 + model.x2) / 2)**2 <= 10
model.stability_constraint = Constraint(rule=stability_constraint_rule)

# Define the mixture tank capacity constraint
def capacity_constraint_rule(model):
    return model.x1 + model.x2 <= 80
model.capacity_constraint = Constraint(rule=capacity_constraint_rule)

# Solve the optimization problem
solver = SolverFactory('ipopt')  # Using the 'ipopt' solver
solver.solve(model)

# Print the optimal solution
print("Optimal solution:")
print("x1 =", model.x1.value)
print("x2 =", model.x2.value)
```
```

```
print("Objective value =", model.objective())
```

```
ERROR:pyomo.core:Rule failed when generating expression for Objective objective with index None:
TypeError: Implicit conversion of Pyomo numeric value (x1*x2) to float is disabled.
This error is often the result of using Pyomo components as arguments to
one of the Python built-in math module functions when defining
expressions. Avoid this error by using Pyomo-provided math functions or
explicitly resolving the numeric value using the Pyomo value() function.
ERROR:pyomo.core:Constructing component 'objective' from data=None failed:
    TypeError: Implicit conversion of Pyomo numeric value (x1*x2) to float is disabled.
This error is often the result of using Pyomo components as arguments to
one of the Python built-in math module functions when defining
expressions. Avoid this error by using Pyomo-provided math functions or
explicitly resolving the numeric value using the Pyomo value() function.
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-16-531391957513> in <cell line: 14>()
     12 def objective_rule(model):
     13     return log(model.x1 * model.x2) - pow(model.x1 + 2 * model.x2, 1/3)
---> 14 model.objective = Objective(rule=objective_rule, sense=maximize)
     15
     16 # Define the stability constraint

                                 4 frames
<ipython-input-16-531391957513> in objective_rule(model)
     11 # Define the objective function
     12 def objective_rule(model):
---> 13     return log(model.x1 * model.x2) - pow(model.x1 + 2 * model.x2, 1/3)
     14 model.objective = Objective(rule=objective_rule, sense=maximize)
     15

pyomo/core/expr/numeric_expr.pyx in pyomo.core.expr.numeric_expr.NumericValue.__float__()

TypeError: Implicit conversion of Pyomo numeric value (x1*x2) to float is disabled.
This error is often the result of using Pyomo components as arguments to
one of the Python built-in math module functions when defining
expressions. Avoid this error by using Pyomo-provided math functions or
explicitly resolving the numeric value using the Pyomo value() function.
```

## 5. Correct The Model Code to Test Mathematical Model (if applicable)

```
from pyomo.environ import *

# Create a Pyomo model
model = ConcreteModel()

# Define the decision variables
model.x1 = Var(domain=NonNegativeReals)
model.x2 = Var(domain=NonNegativeReals)

# Define the objective function
def objective_rule(model):
    return log(model.x1 * model.x2) - (model.x1 + 2 * model.x2)**(1/3)
model.objective = Objective(rule=objective_rule, sense=maximize)

# Define the stability constraint
def stability_constraint_rule(model):
    return (model.x1 - (model.x1 + model.x2) / 2)**2 + (model.x2 - (model.x1 + model.x2) / 2)**2 <= 10
model.stability_constraint = Constraint(rule=stability_constraint_rule)

# Define the mixture tank capacity constraint
def capacity_constraint_rule(model):
    return model.x1 + model.x2 <= 80
model.capacity_constraint = Constraint(rule=capacity_constraint_rule)

# Solve the optimization problem
solver = SolverFactory('ipopt')  # Using the 'ipopt' solver
solver.solve(model)

# Print the optimal solution
print("Optimal solution:")
print("x1 =", model.x1.value)
print("x2 =", model.x2.value)
print("Objective value =", model.objective())
```

```
Optimal solution:
x1 = 42.236067853543105
x2 = 37.76393234768411
Objective value = 2.473033919646447
```