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
prompt1_path = '/content/drive/MyDrive/Thesis/Prompts/Prompt1_MathematicalModel.txt'
prompt2_path = '/content/drive/MyDrive/Thesis/Prompts/Prompt2_PyomoCode.txt'
problem_desc_path = '/content/drive/MyDrive/Thesis/ProblemDescriptions/MIP/MIP2.txt'

prompt1_file = open(prompt1_path, "r")
prompt2_file = open(prompt2_path, "r")
problem_desc_file = open(problem_desc_path, "r")

prompt1 = prompt1_file.read()
print("Prompt 1:\n", prompt1)

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

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

```
Prompt 1:
 Please write a mathematical optimization model for this problem. Include parameters, decision variables, the objective
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 the production planner in a food and beverage production fascility. You are tasked with planning production fo
```

```
!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: Create Mathematical Model

```
message = client.messages.create(
    model="claude-3-opus-20240229",
    system=prompt1,
    max_tokens=4096,
    messages=[
        {"role": "user", "content": problem_desc}
    ]
)

response_p1 = message.content[0].text


# Print response
print(response_p1)
```

```
To formulate the mathematical optimization model for the food and beverage production planning problem, we need to defin

Parameters:
- T: Set of time periods (1, 2, ..., 12)
- P: Set of products (food1, food2, beverage1, beverage2, beverage3)
- fixed_cost[p, t]: Fixed cost of producing product p in period t
- variable_cost_production[p, t]: Variable cost of producing product p in period t
- variable_cost_storage[p, t]: Variable cost of storing product p in period t
- demand[p, t]: Maximum demand for product p in period t
- revenue[p, t]: Revenue per unit of product p sold in period t
- storage_capacity[p]: Storage capacity for product p
- production_capacity[p]: Production capacity for product p

Decision Variables:
- production[p, t]: Quantity of product p produced in period t (non-negative, continuous for beverages, integer for fooc
- storage[p, t]: Quantity of product p stored at the end of period t (non-negative, continuous for beverages, integer fo
- sales[p, t]: Quantity of product p sold in period t (non-negative, continuous for beverages, integer for food)

Objective Function:
Maximize:
  sum(revenue[p, t] * sales[p, t] - fixed_cost[p, t] * (production[p, t] > 0) - variable_cost_production[p, t] * product

Constraints:
1. Inventory balance constraints:
   storage[p, t] = storage[p, t-1] + production[p, t] - sales[p, t] for all p in P, t in T (t > 1)
   storage[p, 1] = production[p, 1] - sales[p, 1] for all p in P

2. Demand constraints:
   sales[p, t] <= demand[p, t] for all p in P, t in T

3. Storage capacity constraints:
   storage[p, t] <= storage_capacity[p] for all p in P, t in T

4. Production capacity constraints:
   production[p, t] <= production_capacity[p] for all p in P, t in T

5. Non-negativity constraints:
   production[p, t], storage[p, t], sales[p, t] >= 0 for all p in P, t in T

6. Integrality constraints:
   production[p, t], storage[p, t], sales[p, t] are integers for p in {food1, food2}, t in T

The objective function aims to maximize the total profit, which is calculated as the sum of revenue from sales minus the

The constraints ensure that:
1. Inventory balance is maintained between consecutive time periods.
2. Sales do not exceed the maximum demand for each product and time period.
3. Storage quantities do not exceed the storage capacity for each product.
4. Production quantities do not exceed the production capacity for each product.
5. Decision variables are non-negative.
6. Production, storage, and sales quantities for food products are integers.

This mathematical optimization model can be solved using mixed-integer programming (MIP) techniques to determine the opt
```

2. Prompt 2: Write Code based on Mathematical Model

```
message2 = client.messages.create(
    model="claude-3-opus-20240229",
    system=prompt2,
    max_tokens=4069,
    messages=[
        {"role": "user", "content": response_p1}
    ]
)

response_p2 = message2.content[0].text
```

```
# Print response
print(response_p2)
```

```
        'food2': 250,
        'beverage1': 500,
        'beverage2': 600,
        'beverage3': 400
    })

    # Define the decision variables
    model.production = Var(model.P, model.T, domain=NonNegativeReals)
    model.storage = Var(model.P, model.T, domain=NonNegativeReals)
    model.sales = Var(model.P, model.T, domain=NonNegativeReals)

    # Define the objective function
    def objective_rule(model):
        return sum(model.revenue[p, t] * model.sales[p, t] -
                   model.fixed_cost[p, t] * (model.production[p, t] > 0) -
                   model.variable_cost_production[p, t] * model.production[p, t] -
                   model.variable_cost_storage[p, t] * model.storage[p, t]
                   for p in model.P for t in model.T)
    model.objective = Objective(rule=objective_rule, sense=maximize)

    # Define the constraints
    def inventory_balance_rule(model, p, t):
        if t == 1:
            return model.storage[p, t] == model.production[p, t] - model.sales[p, t]
        else:
            return model.storage[p, t] == model.storage[p, t-1] + model.production[p, t] - model.sales[p, t]
    model.inventory_balance = Constraint(model.P, model.T, rule=inventory_balance_rule)

    def demand_rule(model, p, t):
        return model.sales[p, t] <= model.demand[p, t]
    model.demand_constraint = Constraint(model.P, model.T, rule=demand_rule)

    def storage_capacity_rule(model, p, t):
        return model.storage[p, t] <= model.storage_capacity[p]
    model.storage_capacity_constraint = Constraint(model.P, model.T, rule=storage_capacity_rule)

    def production_capacity_rule(model, p, t):
        return model.production[p, t] <= model.production_capacity[p]
    model.production_capacity_constraint = Constraint(model.P, model.T, rule=production_capacity_rule)

    # Define integrality constraints for food products
    def integrality_rule(model, p, t):
        if p in ['food1', 'food2']:
            return model.production[p, t].is_integer() and model.storage[p, t].is_integer() and model.sales[p, t].is_integ
        else:
            return Constraint.Skip
    model.integrality_constraint = Constraint(model.P, model.T, rule=integrality_rule)

    # Solve the model
    solver = SolverFactory('glpk')
    results = solver.solve(model)

    # Print the results
    print("Objective value:", value(model.objective))
    for p in model.P:
        for t in model.T:
            print(f"Product: {p}, Period: {t}")
            print(f"  Production: {value(model.production[p, t])}")
```

4. Input Problem Data and Test Model Code

```
# Installing pyomo nd solver
!pip install -q pyomo
!apt-get install -y -qq glpk-utils
!pip install glpk
!pip install pandas
```

Show hidden output

```
from pyomo.environ import *
import pandas as pd

# Read data from CSV files ADJUSTED THE DATA LOADS TO WORK
fixed_cost_production = pd.read_csv("/content/drive/MyDrive/Thesis/ProblemData/MIP/MIP2/fixed_cost_production.csv")
fixed_cost_production.index += 1
fixed_cost_production = fixed_cost_production.drop("Unnamed: 0", axis = 1)
fixed_cost_production.columns = fixed_cost_production.columns.astype(int)

variable_cost_production = pd.read_csv("/content/drive/MyDrive/Thesis/ProblemData/MIP/MIP2/variable_cost_production.csv")
variable_cost_production.index += 1
variable_cost_production = variable_cost_production.drop("Unnamed: 0", axis = 1)
variable_cost_production.columns = variable_cost_production.columns.astype(int)

variable_cost_storage = pd.read_csv("/content/drive/MyDrive/Thesis/ProblemData/MIP/MIP2/variable_cost_storage.csv")
variable_cost_storage.index += 1
variable_cost_storage = variable_cost_storage.drop("Unnamed: 0", axis = 1)
variable_cost_storage.columns = variable_cost_storage.columns.astype(int)

demand = pd.read_csv("/content/drive/MyDrive/Thesis/ProblemData/MIP/MIP2/demand.csv")
demand.index += 1
demand = demand.drop("Unnamed: 0", axis = 1)
demand.columns = demand.columns.astype(int)

revenue = pd.read_csv("/content/drive/MyDrive/Thesis/ProblemData/MIP/MIP2/revenue.csv")
revenue.index += 1
revenue = revenue.drop("Unnamed: 0", axis = 1)
revenue.columns = revenue.columns.astype(int)

# Define the model
model = ConcreteModel()

# Define the sets
model.T = RangeSet(1, 12)  # Time periods
model.P = Set(initialize=[1, 2, 3, 4, 5])  # Products

# Define parameters
model.fixed_cost = Param(model.P, model.T, initialize=fixed_cost_production.stack().to_dict())
model.variable_cost_production = Param(model.P, model.T, initialize=variable_cost_production.stack().to_dict())
model.variable_cost_storage = Param(model.P, model.T, initialize=variable_cost_storage.stack().to_dict())
model.demand = Param(model.P, model.T, initialize=demand.stack().to_dict())
model.revenue = Param(model.P, model.T, initialize=revenue.stack().to_dict())
model.storage_capacity = Param(model.P, initialize={1: 580,
    2: 687,
    3: 599,
    4: 788,
    5: 294
})
model.production_capacity = Param(model.P, initialize={1: 1080,
    2: 908,
    3: 408,
    4: 1000,
    5: 403
})

# Define the decision variables
model.production = Var(model.P, model.T, domain=NonNegativeReals)
model.storage = Var(model.P, model.T, domain=NonNegativeReals)
model.sales = Var(model.P, model.T, domain=NonNegativeReals)

# Define the objective function
def objective_rule(model):
    return sum(model.revenue[p, t] * model.sales[p, t] -
               model.fixed_cost[p, t] * (model.production[p, t] > 0) -
               model.variable_cost_production[p, t] * model.production[p, t] -
               model.variable_cost_storage[p, t] * model.storage[p, t]
               for p in model.P for t in model.T)
model.objective = Objective(rule=objective_rule, sense=maximize)

# Define the constraints
def inventory_balance_rule(model, p, t):
    if t == 1:
        return model.storage[p, t] == model.production[p, t] - model.sales[p, t]
    else:
        return model.storage[p, t] == model.storage[p, t-1] + model.production[p, t] - model.sales[p, t]
model.inventory_balance = Constraint(model.P, model.T, rule=inventory_balance_rule)

def demand_rule(model, p, t):
    return model.sales[p, t] <= model.demand[p, t]
model.demand_constraint = Constraint(model.P, model.T, rule=demand_rule)

def storage_capacity_rule(model, p, t):
    return model.storage[p, t] <= model.storage_capacity[p]
model.storage_capacity_constraint = Constraint(model.P, model.T, rule=storage_capacity_rule)

def production_capacity_rule(model, p, t):
    return model.production[p, t] <= model.production_capacity[p]
model.production_capacity_constraint = Constraint(model.P, model.T, rule=production_capacity_rule)

# Define integrality constraints for food products
def integrality_rule(model, p, t):
    if p in [1, 2]:
        return model.production[p, t].is_integer() and model.storage[p, t].is_integer() and model.sales[p, t].is_integer()
    else:
        return Constraint.Skip
model.integrality_constraint = Constraint(model.P, model.T, rule=integrality_rule)

# Solve the model
solver = SolverFactory('glpk')
results = solver.solve(model)

# Print the results
print("Objective value:", value(model.objective))
for p in model.P:
    for t in model.T:
        print(f"Product: {p}, Period: {t}")
        print(f"  Production: {value(model.production[p, t])}")
        print(f"  Storage: {value(model.storage[p, t])}")
        print(f"  Sales: {value(model.sales[p, t])}")
```

```
ERROR:pyomo.core:Rule failed when generating expression for Objective objecti
TypeError: unsupported operand type(s) for *: 'float' and 'InequalityExpressi
ERROR:pyomo.core:Constructing component 'objective' from data=None failed:
    TypeError: unsupported operand type(s) for *: 'float' and 'InequalityExpr
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-31-6b3028166f9e> in <cell line: 68>()
     66                  model.variable_cost_storage[p, t] * model.storage[p,
t]
     67                  for p in model.P for t in model.T)
---> 68 model.objective = Objective(rule=objective_rule, sense=maximize)
     69
     70 # Define the constraints

                                5 frames
<ipython-input-31-6b3028166f9e> in <genexpr>(.0)
     62 def objective_rule(model):
     63     return sum(model.revenue[p, t] * model.sales[p, t] -
---> 64                  model.fixed_cost[p, t] * (model.production[p, t] > 0)
-
     65                  model.variable_cost_production[p, t] *
model.production[p, t] -
     66                  model.variable_cost_storage[p, t] * model.storage[p,
```

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

```
# Download Gurobi
!wget https://packages.gurobi.com/9.5/gurobi9.5.2_linux64.tar.gz

# Extract the tarball
!tar -xvzf gurobi9.5.2_linux64.tar.gz

# Set up environment variables for Gurobi
import os
os.environ['GUROBI_HOME'] = "/content/gurobi952/linux64"
os.environ['PATH'] += ":/content/gurobi952/linux64/bin"
os.environ['LD_LIBRARY_PATH'] = "/content/gurobi952/linux64/lib"
```

Show hidden output

```
import shutil
shutil.move('/content/drive/MyDrive/gurobi.lic', '/root/gurobi.lic')
```

```
'/root/gurobi.lic'
```

```
from pyomo.environ import *
import pandas as pd

# Read data from CSV files ADJUSTED THE DATA LOADS TO WORK
fixed_cost_production = pd.read_csv("/content/drive/MyDrive/Thesis/ProblemData/MIP/MIP2/fixed_cost_production.csv")
fixed_cost_production.index += 1
```

```
    fixed_cost_production = fixed_cost_production.drop("Unnamed: 0", axis = 1)
    fixed_cost_production.columns = fixed_cost_production.columns.astype(int)

    variable_cost_production = pd.read_csv("/content/drive/MyDrive/Thesis/ProblemData/MIP/MIP2/variable_cost_production.csv")
    variable_cost_production.index += 1
    variable_cost_production = variable_cost_production.drop("Unnamed: 0", axis = 1)
    variable_cost_production.columns = variable_cost_production.columns.astype(int)

    variable_cost_storage = pd.read_csv("/content/drive/MyDrive/Thesis/ProblemData/MIP/MIP2/variable_cost_storage.csv")
    variable_cost_storage.index += 1
    variable_cost_storage = variable_cost_storage.drop("Unnamed: 0", axis = 1)
    variable_cost_storage.columns = variable_cost_storage.columns.astype(int)

    demand = pd.read_csv("/content/drive/MyDrive/Thesis/ProblemData/MIP/MIP2/demand.csv")
    demand.index += 1
    demand = demand.drop("Unnamed: 0", axis = 1)
    demand.columns = demand.columns.astype(int)

    revenue = pd.read_csv("/content/drive/MyDrive/Thesis/ProblemData/MIP/MIP2/revenue.csv")
    revenue.index += 1
    revenue = revenue.drop("Unnamed: 0", axis = 1)
    revenue.columns = revenue.columns.astype(int)

    # Define the model
    model = ConcreteModel()

    # Define the sets
    model.T = RangeSet(1, 12)  # Time periods
    model.P = Set(initialize=[1, 2, 3, 4, 5])  # Products

    # Define parameters
    model.fixed_cost = Param(model.P, model.T, initialize=fixed_cost_production.stack().to_dict())
    model.variable_cost_production = Param(model.P, model.T, initialize=variable_cost_production.stack().to_dict())
    model.variable_cost_storage = Param(model.P, model.T, initialize=variable_cost_storage.stack().to_dict())
    model.demand = Param(model.P, model.T, initialize=demand.stack().to_dict())
    model.revenue = Param(model.P, model.T, initialize=revenue.stack().to_dict())
    model.storage_capacity = Param(model.P, initialize={1: 580,
        2: 687,
        3: 599,
        4: 788,
        5: 294
    })
    model.production_capacity = Param(model.P, initialize={1: 1080,
        2: 908,
        3: 408,
        4: 1000,
        5: 403
    })

    # Define the decision variables
    model.production = Var(model.P, model.T, domain=NonNegativeReals)
    model.storage = Var(model.P, model.T, domain=NonNegativeReals)
    model.sales = Var(model.P, model.T, domain=NonNegativeReals)

    # Define binary variables to indicate production activity MODIFIED ADDED BINARY VARIABLE FOR LOGIC IN OBJECTIVE FUNCTION
    model.Y = Var(model.P, model.T, domain=Binary)

    #set domain to integer for food MODIFIED ADDED INTEGRALITY HERE INSTEAD OF CONSTRAINT
    for t in model.T:
        model.production[2, t].domain = NonNegativeIntegers
        model.production[1, t].domain = NonNegativeIntegers
        model.storage[2, t].domain = NonNegativeIntegers
        model.storage[1, t].domain = NonNegativeIntegers
        model.sales[2, t].domain = NonNegativeIntegers
        model.sales[1, t].domain = NonNegativeIntegers

    # Define the objective function
    def objective_rule(model):
        return sum(model.revenue[p, t] * model.sales[p, t] -
                   model.fixed_cost[p, t] * model.Y[p, t] - # MODIFIED: REMOVE INEQUALITY
                   model.variable_cost_production[p, t] * model.production[p, t] -
                   model.variable_cost_storage[p, t] * model.storage[p, t]
                   for p in model.P for t in model.T)
    model.objective = Objective(rule=objective_rule, sense=maximize)

    # Define the constraints
    def inventory_balance_rule(model, p, t):
        if t == 1:
            return model.storage[p, t] == model.production[p, t] - model.sales[p, t]
        else:
            return model.storage[p, t] == model.storage[p, t-1] + model.production[p, t] - model.sales[p, t]
    model.inventory_balance = Constraint(model.P, model.T, rule=inventory_balance_rule)
```

```
def demand_rule(model, p, t):
    return model.sales[p, t] <= model.demand[p, t]
model.demand_constraint = Constraint(model.P, model.T, rule=demand_rule)

def storage_capacity_rule(model, p, t):
    return model.storage[p, t] <= model.storage_capacity[p]
model.storage_capacity_constraint = Constraint(model.P, model.T, rule=storage_capacity_rule)

# Ensure Y is 1 if P is positive MODIFIED: ADDED BINARY VARIABLE TO CONSTRAINT
def ensure_binary_rule(model, p, t):
    return model.production[p, t] <= model.Y[p, t] * model.production_capacity[p]

model.EnsureBinary = Constraint(model.P, model.T, rule=ensure_binary_rule)

# Solve the model
solver = SolverFactory('gurobi')
results = solver.solve(model)

# Print results
print("Objective function value:", model.objective())
print()
for t in model.T:
    for i in model.P:
        print(f"X[{i}, {t}]: {model.production[i, t].value}, Y[{i}, {t}]: {model.Y[i, t].value}, S[{i}, {t}]: {model.storage
```

```
X[2, 1]: -0.0, Y[2, 1]: -0.0, S[2, 1]: -0.0
X[3, 1]: 13.879999999999999, Y[3, 1]: 1.0, S[3, 1]: 8.2
X[4, 1]: 0.0, Y[4, 1]: -0.0, S[4, 1]: 0.0
X[5, 1]: 0.0, Y[5, 1]: -0.0, S[5, 1]: 0.0
X[1, 2]: -0.0, Y[1, 2]: -0.0, S[1, 2]: -0.0
X[2, 2]: -0.0, Y[2, 2]: -0.0, S[2, 2]: -0.0
X[3, 2]: 0.0, Y[3, 2]: 0.0, S[3, 2]: 0.0
X[4, 2]: 0.0, Y[4, 2]: -0.0, S[4, 2]: 0.0
X[5, 2]: 0.0, Y[5, 2]: 0.0, S[5, 2]: 0.0
X[1, 3]: -0.0, Y[1, 3]: -0.0, S[1, 3]: -0.0
X[2, 3]: -0.0, Y[2, 3]: -0.0, S[2, 3]: -0.0
X[3, 3]: 13.68, Y[3, 3]: 1.0, S[3, 3]: 5.83
X[4, 3]: 0.0, Y[4, 3]: -0.0, S[4, 3]: 0.0
X[5, 3]: 0.0, Y[5, 3]: -0.0, S[5, 3]: 0.0
X[1, 4]: -0.0, Y[1, 4]: -0.0, S[1, 4]: -0.0
X[2, 4]: -0.0, Y[2, 4]: -0.0, S[2, 4]: -0.0
X[3, 4]: 0.0, Y[3, 4]: 0.0, S[3, 4]: 0.0
X[4, 4]: 0.0, Y[4, 4]: -0.0, S[4, 4]: 0.0
X[5, 4]: 0.0, Y[5, 4]: 0.0, S[5, 4]: 0.0
X[1, 5]: -0.0, Y[1, 5]: -0.0, S[1, 5]: -0.0
X[2, 5]: -0.0, Y[2, 5]: -0.0, S[2, 5]: -0.0
X[3, 5]: 14.18, Y[3, 5]: 1.0, S[3, 5]: 6.13
X[4, 5]: 0.0, Y[4, 5]: -0.0, S[4, 5]: 0.0
X[5, 5]: 0.0, Y[5, 5]: 0.0, S[5, 5]: 0.0
X[1, 6]: -0.0, Y[1, 6]: -0.0, S[1, 6]: -0.0
X[2, 6]: -0.0, Y[2, 6]: -0.0, S[2, 6]: -0.0
X[3, 6]: 0.0, Y[3, 6]: 0.0, S[3, 6]: 0.0
X[4, 6]: 0.0, Y[4, 6]: -0.0, S[4, 6]: 0.0
X[5, 6]: 7.79, Y[5, 6]: 1.0, S[5, 6]: 3.94
X[1, 7]: -0.0, Y[1, 7]: -0.0, S[1, 7]: -0.0
X[2, 7]: -0.0, Y[2, 7]: -0.0, S[2, 7]: -0.0
X[3, 7]: 20.43, Y[3, 7]: 1.0, S[3, 7]: 14.05
X[4, 7]: 0.0, Y[4, 7]: -0.0, S[4, 7]: 0.0
X[5, 7]: 0.0, Y[5, 7]: 0.0, S[5, 7]: 0.0
X[1, 8]: -0.0, Y[1, 8]: -0.0, S[1, 8]: -0.0
X[2, 8]: -0.0, Y[2, 8]: -0.0, S[2, 8]: -0.0
X[3, 8]: 0.0, Y[3, 8]: 0.0, S[3, 8]: 6.75
X[4, 8]: 0.0, Y[4, 8]: -0.0, S[4, 8]: 0.0
X[5, 8]: 7.779999999999999, Y[5, 8]: 1.0, S[5, 8]: 3.8
X[1, 9]: -0.0, Y[1, 9]: -0.0, S[1, 9]: -0.0
X[2, 9]: -0.0, Y[2, 9]: -0.0, S[2, 9]: -0.0
X[3, 9]: 0.0, Y[3, 9]: 0.0, S[3, 9]: 0.0
X[4, 9]: 0.0, Y[4, 9]: -0.0, S[4, 9]: 0.0
X[5, 9]: 0.0, Y[5, 9]: 0.0, S[5, 9]: 0.0
X[1, 10]: -0.0, Y[1, 10]: -0.0, S[1, 10]: -0.0
X[2, 10]: -0.0, Y[2, 10]: -0.0, S[2, 10]: -0.0
X[3, 10]: 11.690000000000001, Y[3, 10]: 1.0, S[3, 10]: 5.86
X[4, 10]: 0.0, Y[4, 10]: -0.0, S[4, 10]: 0.0
X[5, 10]: 0.0, Y[5, 10]: 0.0, S[5, 10]: 0.0
X[1, 11]: -0.0, Y[1, 11]: -0.0, S[1, 11]: -0.0
X[2, 11]: -0.0, Y[2, 11]: -0.0, S[2, 11]: -0.0
X[3, 11]: 0.0, Y[3, 11]: 0.0, S[3, 11]: 0.0
X[4, 11]: 0.0, Y[4, 11]: -0.0, S[4, 11]: 0.0
X[5, 11]: 0.0, Y[5, 11]: 0.0, S[5, 11]: 0.0
X[1, 12]: -0.0, Y[1, 12]: -0.0, S[1, 12]: -0.0
X[2, 12]: -0.0, Y[2, 12]: -0.0, S[2, 12]: -0.0
X[3, 12]: 5.85, Y[3, 12]: 1.0, S[3, 12]: 0.0
X[4, 12]: 0.0, Y[4, 12]: -0.0, S[4, 12]: 0.0
```