TASKS:

Work and discuss the following exercises during class, if possible. It is recommended to work on teams of 2 or 3 people. If you prefer to work off-line, you can work individually on your own preferred time.

From the textbook:

- Set up and solve the following problems:

Problems 1.1, 1.2,1.3,1.4,1.5, 1.8 (solve with the Excel solver when numerical values are given).

- Discuss problem 1.13 (interpret the optimization problem set up and discuss the question, but do not solve it).

- Solve problem 3.12 of the textbook. Notice that you need to consider a lifetime of 30 years for each project, so it will be necessary to build 3 flumes during the life of the second project. Ignore inflation effects.

- Solve problem 3.13 of the textbook. Report on your conclusions.

## PROBLEMS

*For each of the following six problems, formulate the objective function, the equality constraints (if any), and the inequality constraints (if any). Specify and list the independent variables, the number of degrees of freedom, and the coefficients in the optimization problem. Solve the problem using calculus as needed, and state the complete optimal solution values.*

## Problem 1.1

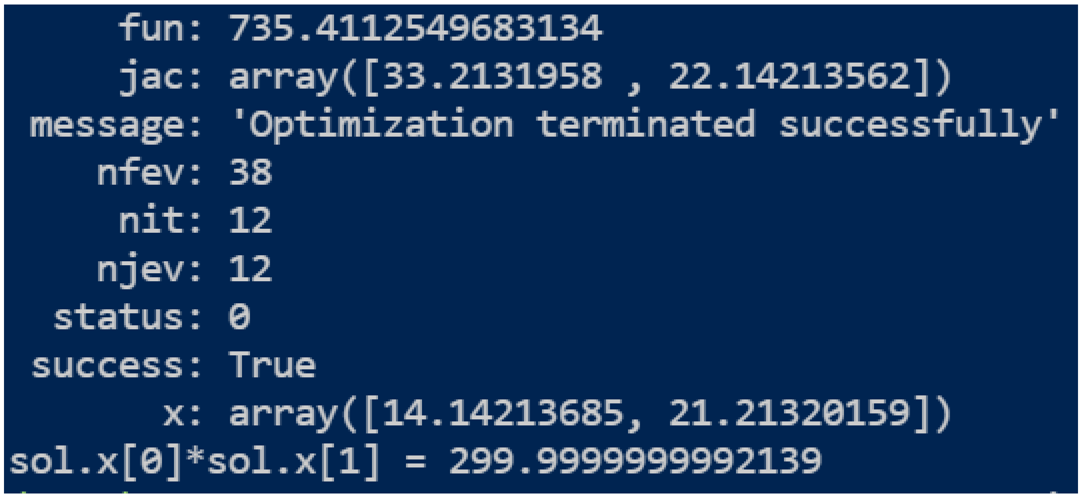
**1.1** A poster is to contain 300 cm$^2$ of printed matter with margins of 6 cm at the top and bottom and 4 cm at each side. Find the overall dimensions that minimize the total area of the poster.

### Python solution

```
import numpy as np
import scipy.optimize as spo
# problem 1.1

def objective(x: list[float]) -> float:
    """function to minimize

    Args:
        x (list[float]): vector of decision variables

    Returns:
        float: result of the function, minimize this value
    """
    return (x[0]+8)*(x[1]+12)


def constraint1(x: list[float]) -> float:
    """equality constraint: area should equal 300
```

```python
    Args:
        x (list[float]): vector of decision variables

    Returns:
        float: value that should be zero or very very close
    """
    return x[0]*x[1]-300


# initial guess
x0 = [6, 6]

# constraints
con1 = {"type": "eq", "fun": constraint1}
cons = [con1]  # list of constraints

# solution
sol = spo.minimize(objective, x0, constraints=cons)

# print of the solution
print(sol)

# checking if the solution obeys the constraints
print(f'{sol.x[0]*sol.x[1] = }')
```

Console output



```
     fun: 735.4112549683134
     jac: array([33.2131958 , 22.14213562])
 message: 'Optimization terminated successfully'
    nfev: 38
     nit: 12
    njev: 12
  status: 0
 success: True
       x: array([14.14213685, 21.21320159])
sol.x[0]*sol.x[1] = 299.9999999992139
```

## Problem 1.2

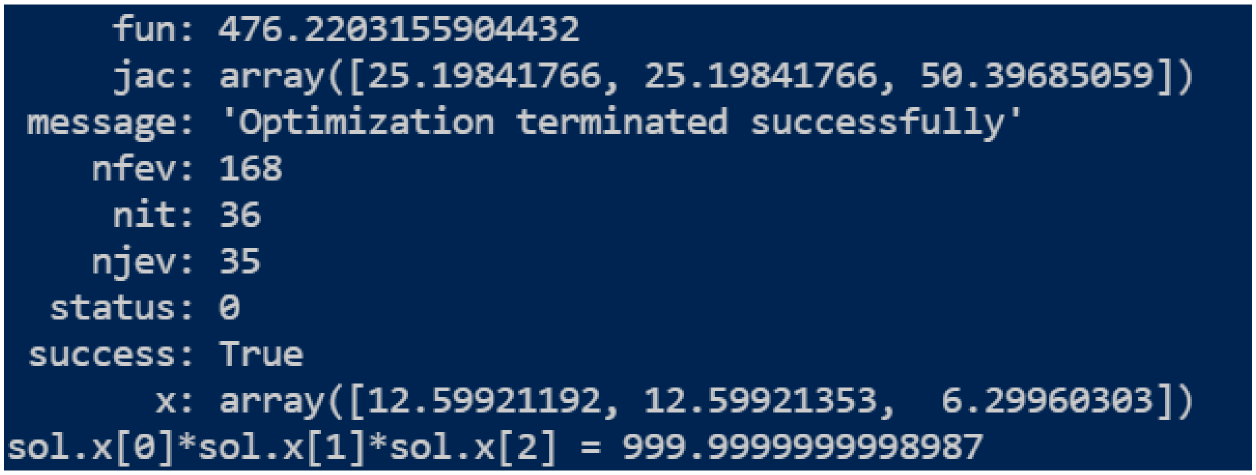
**1.2** A box with a square base and open top is to hold 1000 cm$^3$. Find the dimensions that require the least material (assume uniform thickness of material) to construct the box.

### Python solution

```python
import numpy as np
import scipy.optimize as spo
# problem 1.2

# 1 degree of freedom, because changing one variable all others have to obey the constraints
# Width/length independant variables but they are the same variable in this case
box_width = 22
box_length = box_width
box_height = 3

# vector of initial guesses
x0 = [box_width, box_length, box_height]


def objective(x: list[float]) -> float:
    """Objective function to minimize

    Args:
        x (list[float]): vector of decision variables

    Returns:
        float: sum of area
    """
    base_area = x[0]*x[1]
    side_area_1 = x[0]*x[2]
    side_area_2 = x[1]*x[2]
    return base_area+side_area_1*2+side_area_2*2


def constraint1(x: list[float]) -> float:
    return x[0]*x[1]*x[2]-1000


# constraints
con1 = {"type": "eq", "fun": constraint1}
cons = [con1]  # list of constraints

# solution
```

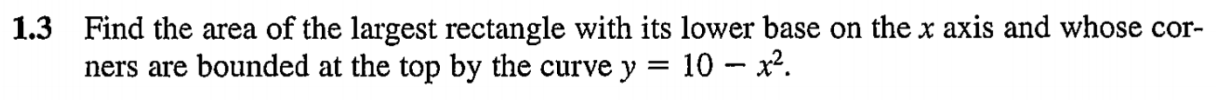
```python
sol = spo.minimize(objective, x0, constraints=cons)

print(sol)
print(f'{sol.x[0]*sol.x[1]*sol.x[2] = }')
```

Console output



```
     fun: 476.2203155904432
     jac: array([25.19841766, 25.19841766, 50.39685059])
 message: 'Optimization terminated successfully'
    nfev: 168
     nit: 36
    njev: 35
  status: 0
 success: True
       x: array([12.59921192, 12.59921353,  6.29960303])
sol.x[0]*sol.x[1]*sol.x[2] = 999.9999999998987
```

## Problem 1.3

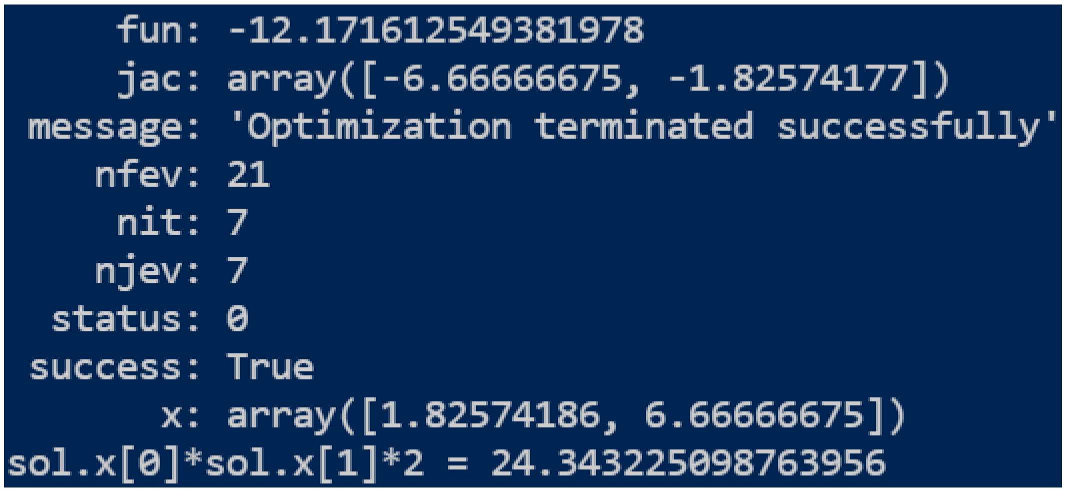


**1.3** Find the area of the largest rectangle with its lower base on the $x$ axis and whose corners are bounded at the top by the curve $y = 10 - x^2$.

Python solution

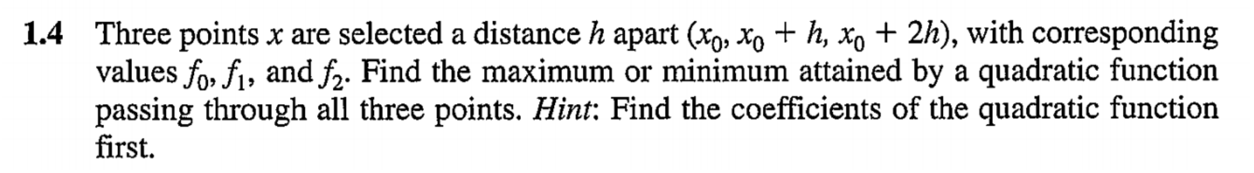
```python
import numpy as np
import scipy.optimize as spo
# problem 1.3

# 1 degree of freedom
# x_length only independant variable
x_lenght = 3
y_height = 10-x_lenght**2

# vector of initial guesses
x0 = [x_lenght, y_height]

def objective(x: list[float]) -> float:
    """Objective function to maximize

    Args:
        x (list[float]): vector of decision variables
    Returns:
        float: area
    """
    return -
(x[0]*x[1]) #minimizing the negative area is the same as maximizing the positive


def constraint1(x: list[float]) -> float:
    return x[1]+(x[0]**2)-10

# constraints
con1 = {"type": "eq", "fun": constraint1}
cons = [con1]  # list of constraints

# Bound by x axis
bnds = ((None, None), (0, None))

# solution
sol = spo.minimize(objective, x0, bounds=bnds, constraints=cons)

print(sol)
print(f'{sol.x[0]*sol.x[1]*2 = }')
```

Console output



```
     fun: -12.171612549381978
     jac: array([-6.66666675, -1.82574177])
 message: 'Optimization terminated successfully'
    nfev: 21
     nit: 7
    njev: 7
  status: 0
 success: True
       x: array([1.82574186, 6.66666675])
sol.x[0]*sol.x[1]*2 = 24.343225098763956
```

## Problem 1.4

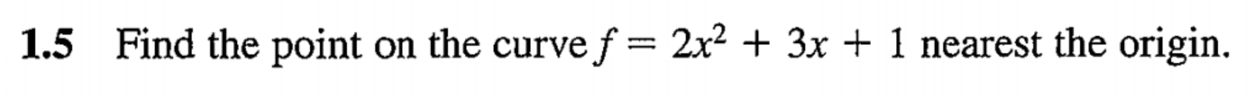


**1.4** Three points $x$ are selected a distance $h$ apart ($x_0$, $x_0 + h$, $x_0 + 2h$), with corresponding values $f_0$, $f_1$, and $f_2$. Find the maximum or minimum attained by a quadratic function passing through all three points. *Hint*: Find the coefficients of the quadratic function first.

### Python solution

```
import numpy as np
import scipy.optimize as spo
# problem 1.4
# quadratic function (a*x**2)+b*x+c
# find function that gives the max or min of all three points

# ??? isn't the solution just infinite or negative infinite
```

## Problem 1.5

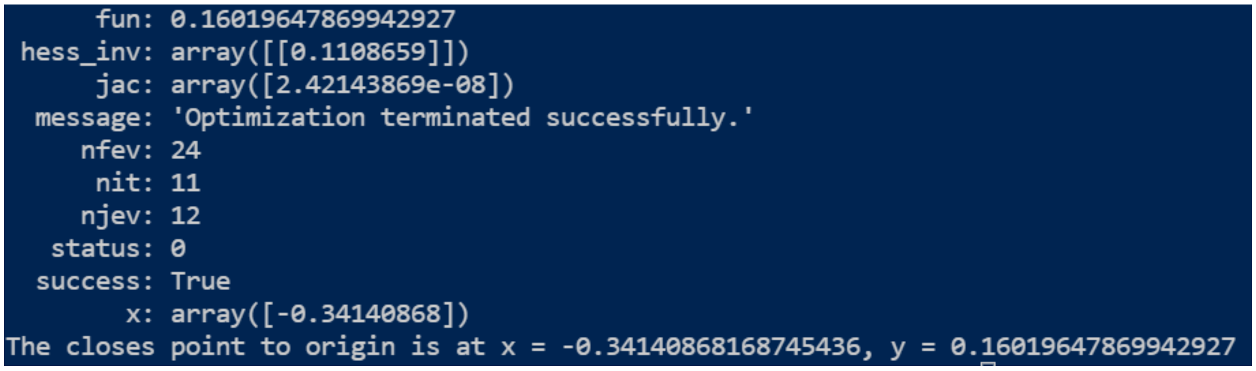


**1.5** Find the point on the curve $f = 2x^2 + 3x + 1$ nearest the origin.

### Python solution

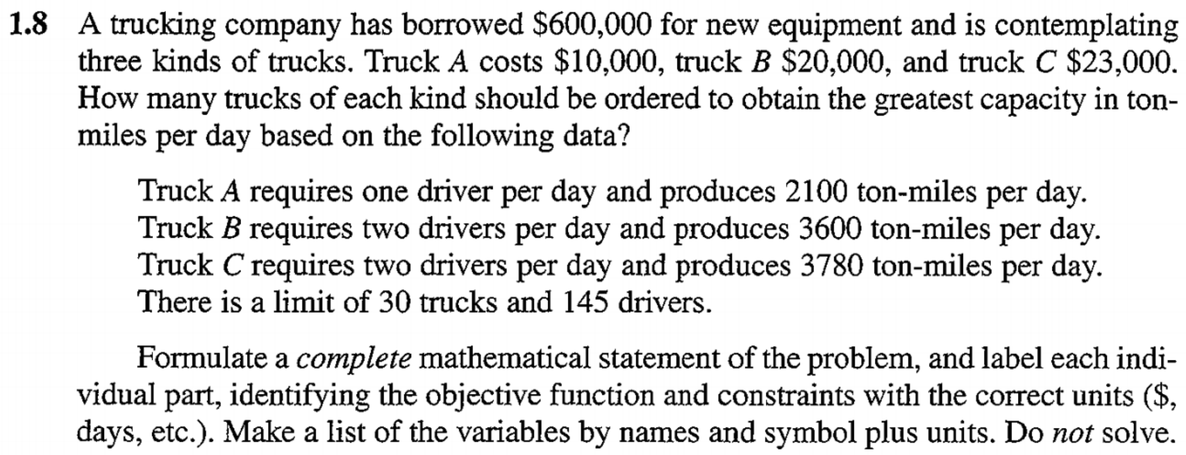
```
from math import sqrt
import numpy as np
import scipy.optimize as spo
# problem 1.5

x_value = 2

# vector of initial guesses
x0 = [x_value]

def objective(x: list[float]) -> float:
    """Objective function to maximize

    Args:
        x (list[float]): vector of decision variables

    Returns:
        float: area
    """
    y = (2*x[0]**2)+3*x+1
    # pythagoras, but really no need to find the square root
    distance_from_origin = (x[0]**2)+(y**2)
    return distance_from_origin

# solution
sol = spo.minimize(objective, x0, bounds=None, constraints=None)

print(sol)

print(f'The closes point to origin is at x = {sol.x[0]}, y = {sol.fun}')
```

### Console output



```
      fun: 0.16019647869942927
 hess_inv: array([[0.1108659]])
      jac: array([2.42143869e-08])
  message: 'Optimization terminated successfully.'
     nfev: 24
      nit: 11
     njev: 12
   status: 0
  success: True
        x: array([-0.34140868])
The closes point to origin is at x = -0.34140868168745436, y = 0.16019647869942927
```

## Problem 1.8



**1.8** A trucking company has borrowed $600,000 for new equipment and is contemplating three kinds of trucks. Truck *A* costs $10,000, truck *B* $20,000, and truck *C* $23,000. How many trucks of each kind should be ordered to obtain the greatest capacity in ton-miles per day based on the following data?

Truck *A* requires one driver per day and produces 2100 ton-miles per day.
Truck *B* requires two drivers per day and produces 3600 ton-miles per day.
Truck *C* requires two drivers per day and produces 3780 ton-miles per day.
There is a limit of 30 trucks and 145 drivers.

Formulate a *complete* mathematical statement of the problem, and label each individual part, identifying the objective function and constraints with the correct units ($, days, etc.). Make a list of the variables by names and symbol plus units. Do *not* solve.

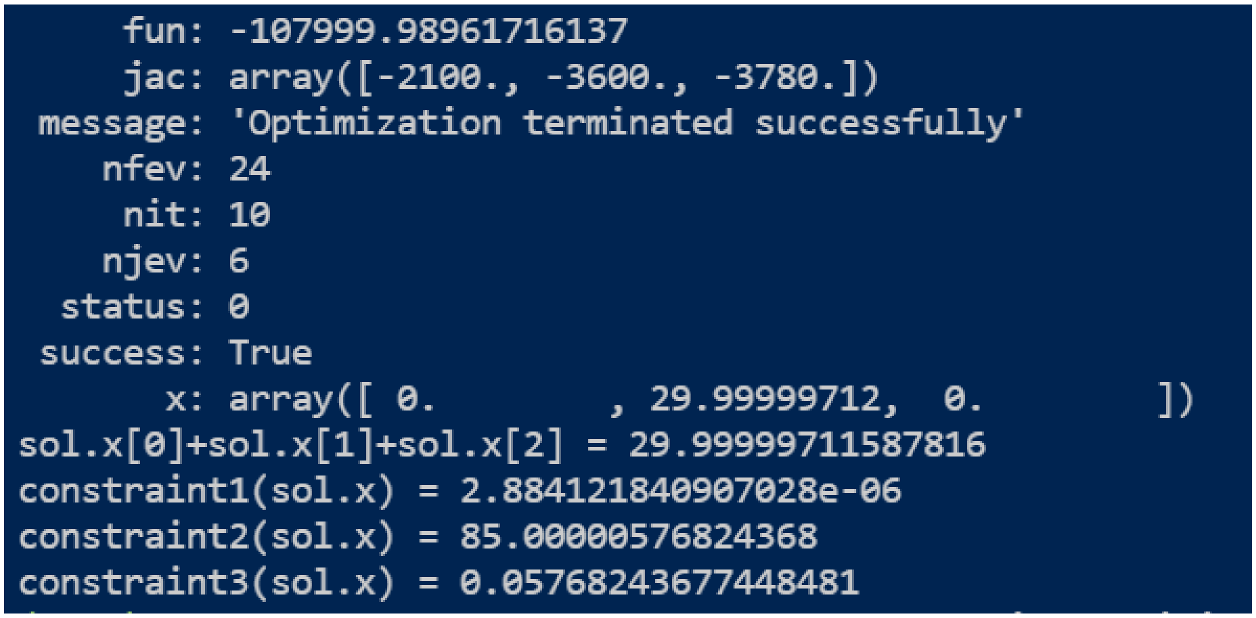
### Python solution

```
import numpy as np
import scipy.optimize as spo
# Problem 1.8

amount_of_truck_a = 0
amount_of_truck_b = 0
```

```python
amount_of_truck_c = 30

# vector of initial guesses
x0 = [amount_of_truck_a, amount_of_truck_b, amount_of_truck_c]

price_of_truck_a = 10_000
price_of_truck_b = 20_000
price_of_truck_c = 23_000

prices = [price_of_truck_a, price_of_truck_b, price_of_truck_c]

capacity_of_truck_a = 2100
capacity_of_truck_b = 3600
capacity_of_truck_c = 3780

capacities = [capacity_of_truck_a, capacity_of_truck_b, capacity_of_truck_c]

required_drivers_truck_a = 1
required_drivers_truck_b = 2
required_drivers_truck_c = 2

drivers = [required_drivers_truck_a,
           required_drivers_truck_b, required_drivers_truck_c]

maximum_trucks = 30
maximum_drivers = 145
maximum_budget = 600_000


def objective(x: list[float]) -> float:
    """Objective function to minimize

    Args:
        x (list[float]): vector of decision variables

    Returns:
        float: total capacity
    """
    return -
(x[0]*capacities[0]+x[1]*capacities[1]+x[2]*capacities[2])  # minimum of negative
 = maximum

def constraint1(x: list[float]) -> float:
    """ineq: maximum amount of trucks
    Args:
        x (list[float]): vector of decision variables

    Returns:
        float: number that is >=0 if constraint is obeyed
    """
    return -(x[0]+x[1]+x[2]-maximum_trucks)

def constraint2(x: list[float]) -> float:
    """ineq: maximum amount of drivers
    Args:
        x (list[float]): vector of decision variables

    Returns:
        float: number that is >=0 if constraint is obeyed
    """
    return -(x[0]*drivers[0]+x[1]*drivers[1]+x[2]*drivers[2]-maximum_drivers)

def constraint3(x: list[float]) -> float:
    """ineq: maximum budget

    Args:
        x (list[float]): vector of decision variables

    Returns:
        float: number that is >=0 if constraint is obeyed
    """
    return -(x[0]*prices[0]+x[1]*prices[1]+x[2]*prices[2]-maximum_budget)


def integer_constraint1(x: list[float]) -> float:
    return max([x[0]-int(x[0])])


def integer_constraint2(x: list[float]) -> float:
    return max([x[1]-int(x[1])])


def integer_constraint3(x: list[float]) -> float:
    return max([x[2]-int(x[2])])


bnds = ((0, 30), (0, 30), (0, 30))

# constraints
con1 = {"type": "ineq", "fun": constraint1}
```

```
con2 = {"type": "ineq", "fun": constraint2}
con3 = {"type": "ineq", "fun": constraint3}
con4 = {"type": "eq", "fun": integer_constraint1}
con5 = {"type": "eq", "fun": integer_constraint2}
con6 = {"type": "eq", "fun": integer_constraint3}
con7 = {'type':'eq','fun': lambda x : max([x[i]-
int(x[i]) for i in range(len(x))])}
cons = [con1, con2, con3]  # list of constraints

# solution
sol = spo.minimize(objective, x0, bounds=bnds, constraints=cons)

print(sol)
print(f'{sol.x[0]+sol.x[1]+sol.x[2] = }')
print(f'{constraint1(sol.x) = }')
print(f'{constraint2(sol.x) = }')
print(f'{constraint3(sol.x) = }')
```

## Console output



```
     fun: -107999.98961716137
     jac: array([-2100., -3600., -3780.])
 message: 'Optimization terminated successfully'
    nfev: 24
     nit: 10
    njev: 6
  status: 0
 success: True
       x: array([ 0.        , 29.99999712,  0.        ])
sol.x[0]+sol.x[1]+sol.x[2] = 29.99999711587816
constraint1(sol.x) = 2.884121840907028e-06
constraint2(sol.x) = 85.00000576824368
constraint3(sol.x) = 0.05768243677448481
```

The used library doesn't support integer constraints. And a hacky work around I found on stack overflow didn't work so I didn't apply it to the solution-attempt. But it looks like this solver only wants to buy truck b, which is a bit strange because it leaves us with 85 less than the maximum number of drivers. But, the maximum trucks constraint of 30 makes the real driver constraint 60 so that explains it. The budget is also used up, so everything seems right after all.

## Problem 1.13

**1.13** The following problem is formulated as an optimization problem. A batch reactor operating over a 1-h period produces two products according to the parallel reaction mechanism: $A \rightarrow B$, $A \rightarrow C$. Both reactions are irreversible and first order in $A$ and have rate constants given by

$$k_i = k_{io} \exp\{E_i/RT\} \quad i = 1,2$$

where
$$k_{10} = 10^6/\text{s}$$
$$k_{20} = 5.10^{11}/\text{s}$$
$$E_1 = 10{,}000 \text{ cal/gmol}$$
$$E_2 = 20{,}000 \text{ cal/gmol}$$

The objective is to find the temperature–time profile that maximizes the yield of $B$ for operating temperatures below 282°F. The optimal control problem is therefore

$$\text{Maximize:} \quad B(1.0)$$

$$\text{Subject to:} \quad \frac{dA}{dt} = -(k_1 + k_2)A$$

$$\frac{dB}{dt} = k_1 A$$

$$A(0) = A_0$$

$$B(0) = 0$$

$$T \leq 282°\text{F}$$

(a) What are the independent variables in the problem?
(b) What are the dependent variables in the problem?
(c) What are the equality constraints?
(d) What are the inequality constraints?
(e) What procedure would you recommend to solve the problem?

a) Temperature (T) is the independent variable because every other value changes when the temperature is changed. Optionally A or B can be chosen as the independent variable.
b) A and B are dependent variables if they are not chosen as independent.
c) Equality constraints are change of A and change of B. Initial values of A and B are not constraints here.
d) Temperature has an inequality constraint; it must be less or equal to 282°F.
e) To solve this, you need to use numeric integration to find the objective function. Then use an optimization tool to find the maximum.

## Problem 3.12

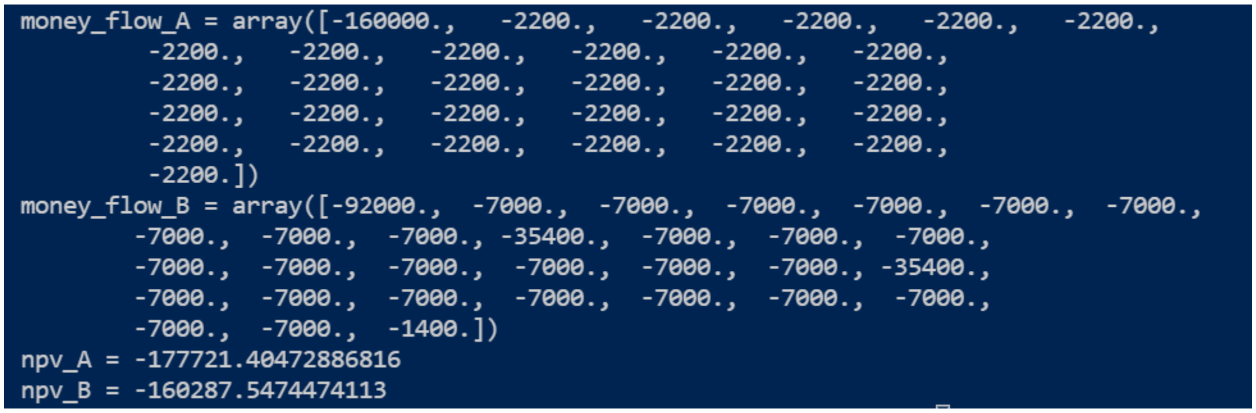
**3.12** Consideration is being given to two plans for supplying water to a plant. Plan *A* requires a pipeline costing \$160,000 with annual operation and unkeep costs of \$2200, and an estimated life of 30 years with no salvage. Plan *B* requires a flume costing \$34,000 with a life of 10 years, a salvage value of \$5600, and annual operation and upkeep of \$4500 plus a ditch costing \$58,000, with a life of 30 years and annual costs for upkeep of \$2500. Using an interest rate of 12 percent, compare the net present values of the two alternatives.

### Python solution

```
import numpy as np
import numpy_financial as npf
import scipy.optimize as spo
# Problem 3.12

#Common:
interest_rate = 0.12

#Plan A:
pipeline_A = 160_000 #$
annual_operation_and_upkeep_A = 2200 #$
lifetime_A = 31 #years

#* Money flow A
money_flow_A = np.zeros(lifetime_A)
money_flow_A[0] += -pipeline_A
for i in range(1,lifetime_A):
    money_flow_A[i] += -annual_operation_and_upkeep_A
print(f'{money_flow_A = }')

#Plan B
flume_B = 34_000 #$
flume_lifetime_B = 10 #years
flume_B_salvage_value = 5600 #$
annual_operation_flume_B = 4500 #$
ditch_B = 58_000 #$
annual_ditch_B_upkeep = 2500 #$
lifetime_B = 31 #years

#* Money flow B
money_flow_B = np.zeros(lifetime_B)
# year 0 investments
money_flow_B[0] += -flume_B
```

```
money_flow_B[0] += -ditch_B
#from year 1 and even year 30
for i in range(1,lifetime_B):
    money_flow_B[i] += -annual_operation_flume_B
    money_flow_B[i] += -annual_ditch_B_upkeep
    #last year we don't buy a new flume
    if(i%30 == 0):
        money_flow_B[i] += flume_B_salvage_value
    #every 10th year we salvage and buy new flume
    elif(i%10 == 0):
        money_flow_B[i] += flume_B_salvage_value
        money_flow_B[i] += -flume_B

print(f'{money_flow_B = }')


npv_A = npf.npv(interest_rate,money_flow_A)
npv_B = npf.npv(interest_rate,money_flow_B)

print(f'{npv_A = }')
print(f'{npv_B = }')
```

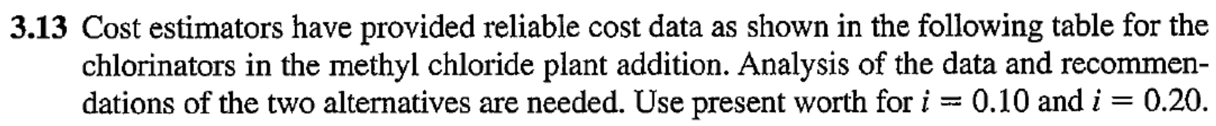
### Console output



```
money_flow_A = array([-160000.,   -2200.,   -2200.,   -2200.,   -2200.,   -2200.,
         -2200.,   -2200.,   -2200.,   -2200.,   -2200.,   -2200.,
         -2200.,   -2200.,   -2200.,   -2200.,   -2200.,   -2200.,
         -2200.,   -2200.,   -2200.,   -2200.,   -2200.,   -2200.,
         -2200.,   -2200.,   -2200.,   -2200.,   -2200.,   -2200.,
         -2200.])
money_flow_B = array([-92000.,  -7000.,  -7000.,  -7000.,  -7000.,  -7000.,  -7000.,
        -7000.,  -7000.,  -7000., -35400.,  -7000.,  -7000.,  -7000.,
        -7000.,  -7000.,  -7000.,  -7000.,  -7000.,  -7000., -35400.,
        -7000.,  -7000.,  -7000.,  -7000.,  -7000.,  -7000.,  -7000.,
        -7000.,  -7000.,  -1400.])
npv_A = -177721.40472886816
npv_B = -160287.5474474113
```

Looks like option B is the least negative, so it should be cheaper.

## Problem 3.13



**3.13** Cost estimators have provided reliable cost data as shown in the following table for the chlorinators in the methyl chloride plant addition. Analysis of the data and recommendations of the two alternatives are needed. Use present worth for $i = 0.10$ and $i = 0.20$.

| | Chlorinators | |
|---|---|---|
| | Glass-lined | Cast iron |
| Installed cost | $24,000 | $7200 |
| Estimated useful life | 10 years | 4 years |
| Salvage value | $4000 | $800 |
| Miscellaneous annual costs as percent of original cost | 10 | 20 |

**Maintenance costs**

*Glass-lined.* $230 at the end of the second year, $560 at the end of the fifth year, and $900 at the end of each year thereafter.

*Cast iron.* $730 each year.

The product from the glass-lined chlorinator is essentially iron-free and is estimated to yield a product quality premium of $1700 per year. Compare the two alternatives for a 10-year period. Assume the salvage value of $800 is valid at 10 years.

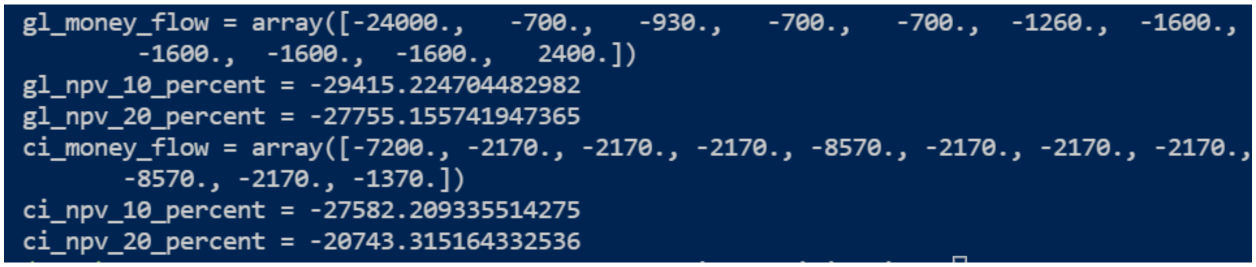
## Python solution

```
import numpy as np
import numpy_financial as npf
import scipy.optimize as spo

#Comparison of chlorinators

#interest rates
interest_rates = [0.1,0.2]

#period in years: year zero to year 10
period = 11

#Glass-linted chlorinator:
gl_installed_cost = 24_000
gl_life_estimate = 10
gl_salvage_value = 4000
gl_annual_cost = gl_installed_cost*0.1
gl_yearly_premium = 1700

gl_maintenance_cost = np.zeros(period)
```

```python
gl_maintenance_cost[2] = 230 #maintenance cost at end of 2nd year
gl_maintenance_cost[5] = 560 #maintenance cost at end of 5th year

for i in range(6,len(gl_maintenance_cost)):
    gl_maintenance_cost[i] = 900 #maintenece cost at the end of the rest of the y
ears

gl_money_flow = np.zeros(period)
gl_money_flow[0] -= gl_installed_cost #initial cost

for i in range(1, len(gl_money_flow)):
    gl_money_flow[i] -= gl_annual_cost
    gl_money_flow[i] -= gl_maintenance_cost[i]
    gl_money_flow[i] += gl_yearly_premium
    if(i%10 == 0):
        gl_money_flow[i] += gl_salvage_value

gl_npv_10_percent = npf.npv(interest_rates[0], gl_money_flow)
gl_npv_20_percent = npf.npv(interest_rates[1], gl_money_flow)

print(f'{gl_money_flow = }')
print(f'{gl_npv_10_percent = }')
print(f'{gl_npv_20_percent = }')

#Cast iron chlorinator:
ci_installed_cost = 7200
ci_life_estimate = 4
ci_salvage_value = 800
ci_annual_cost = ci_installed_cost*0.2
ci_yearly_premium = 0

ci_maintenence_cost = np.full(11,730)
ci_maintenence_cost[0] = 0

ci_money_flow = np.zeros(period)
ci_money_flow[0] -= ci_installed_cost

for i in range(1, len(ci_money_flow)):
    ci_money_flow[i] -= ci_annual_cost
    ci_money_flow[i] -= ci_maintenence_cost[i]
    if(i%4 == 0):
        ci_money_flow[i] += ci_salvage_value
        ci_money_flow[i] -= ci_installed_cost
#salvaging the chlorinator in year 10 even though it has more life
ci_money_flow[-1] += ci_salvage_value
```

```python
ci_npv_10_percent = npf.npv(interest_rates[0], ci_money_flow)
ci_npv_20_percent = npf.npv(interest_rates[1], ci_money_flow)

print(f'{ci_money_flow = }')
print(f'{ci_npv_10_percent = }')
print(f'{ci_npv_20_percent = }')
```

## Console output



```
gl_money_flow = array([-24000.,   -700.,   -930.,   -700.,   -700.,  -1260.,  -1600.,
        -1600.,  -1600.,  -1600.,   2400.])
gl_npv_10_percent = -29415.224704482982
gl_npv_20_percent = -27755.155741947365
ci_money_flow = array([-7200., -2170., -2170., -2170., -8570., -2170., -2170., -2170.,
       -8570., -2170., -1370.])
ci_npv_10_percent = -27582.209335514275
ci_npv_20_percent = -20743.315164332536
```

The cast iron option was clearly the winner with both interest rates. Especially interesting is that the cast iron option is much cheaper at 20% interest compared to the glass lined because of the big difference in initial and early costs. With a high interest rate you would much rather invest your money to access easy profits, while you delay payments for your project.