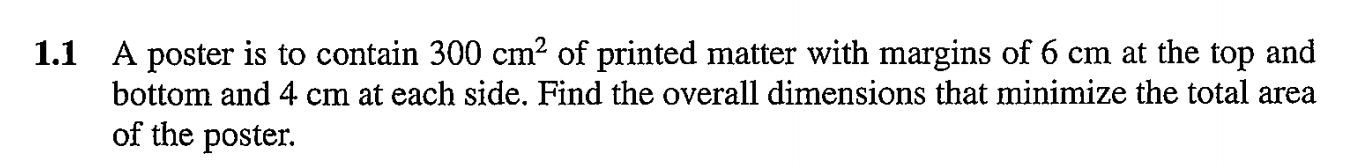
# Graphical user interface, text, application, email Description automatically generated

Text, letter

Description automatically generated

# Problem 1.1



## 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

    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

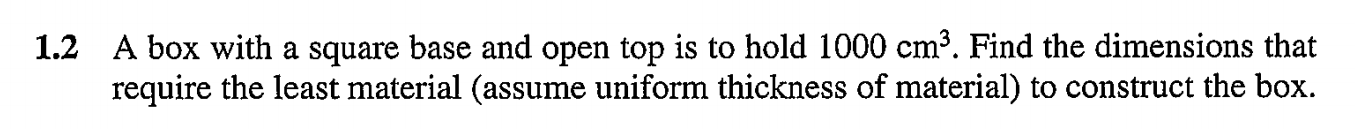
print(f'{sol.x[0]\*sol.x[1] = }')

## Console output

Text

Description automatically generated

# Problem 1.2



## Python solution

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

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

print(sol)

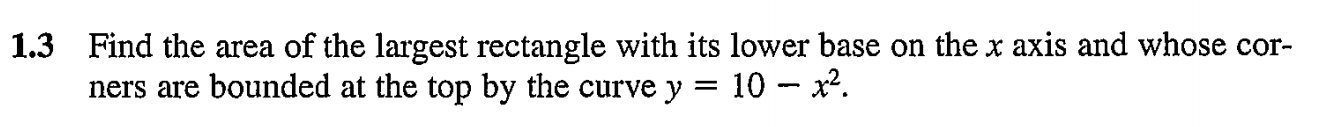
print(f'{sol.x[0]\*sol.x[1]\*sol.x[2] = }')

## Console output

Graphical user interface, text

Description automatically generated

# Problem 1.3



## Python solution

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

Text

Description automatically generated

# Problem 1.4

Text

Description automatically generated with medium confidence

## Python solution

import numpy as np

import scipy.optimize as spo

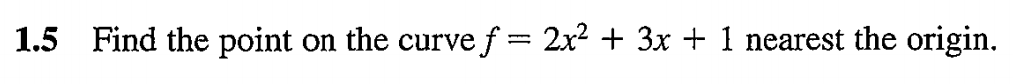
# problem 1.4

# 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



## 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

Graphical user interface, application

Description automatically generated

# Problem 1.8

A picture containing text, indoor, screenshot, document

Description automatically generated

## Python solution

import numpy as np

import scipy.optimize as spo

# Problem 1.8

amount\_of\_truck\_a = 0

amount\_of\_truck\_b = 0

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

Graphical user interface, text, application

Description automatically generated

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

Text

Description automatically generated

1. 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.
2. A and B are dependent variables if they are not chosen as independent.
3. Equality constraints are change of A and change of B. Initial values of A and B are not constraints here.
4. Temperature has an inequality constraint; it must be less or equal to 282℉.
5. 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

Text

Description automatically generated

## 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 = }')

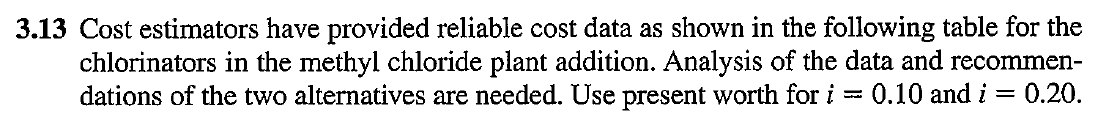
## Console output

Graphical user interface, application

Description automatically generated

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

# Problem 3.13



Table

Description automatically generated

## 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)

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 years

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

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

Graphical user interface, application

Description automatically generated

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.