Report Submission Information (must be completed before submitting report!)

- Student 1 Full Name and Number: Amir Fayaz Heidari (1096900)
- Student 2 Full Name and Number: Tushar Balkrushna Ingle (1117044)
- Student 3 Full Name and Number: Ryan Penrose (998704)
- Workshop day: Friday
- Workshop time: 11 am

Workshop 1 – Optimisation [2 weeks]

Topics Covered

- Optimisation basics, formulation of optimisation problems
- Local/global minima/maxima
- Convex sets and functions
- Unconstrained optimisation and optimality conditions
- Constrained optimisation, equality and inequality constraints, constraint sets
- Lagrange theory and KKT conditions

Topic_mindmap

Topic Notes

Another name for the field of "Optimisation" is "Mathematical Optimisation." As the name indicates, optimisation is an area of applied mathematics. It is possible to study optimisation entirely from a mathematical perspective. However, engineers are interested in solving real-world problems in a principled way. Many engineering problems can be and are formulated as optimisation problems. In those cases, mathematical optimisation provides a solid theoretical foundation for solving them in a principled way.

In this workshop, you will learn how to formulate and solve optimisation problems in practice. This will give you a chance to connect theoretical knowledge and practical usage by doing it yourself. You will familiarise yourself with practical optimisation tools for Python. These are chosen completely for educational reasons (simplicity, accessibility, cost). While the underlying mathematics is timeless, optimisation software evolves with time, and can be diverse. Fortunately, once you learn one or two, it should be rather easy to learn others now and in the future because software designers often try to make it user friendly and take into account what people already know.

> In the future, you should consider learning serious optimisation software for scalability and reliability. They can be complex and/or expensive but they get the job done for serious engineering. Learning such software takes a significant amount of time and is beyond the scope of this subject.

Table of Contents

Workflow and Assessment

This subject follows a problem- and project-oriented approach. In this learning workflow, the focus is on solving practical (engineering) problems, which motivate acquiring theoretical (background) knowledge at the same time.

Objectives:

- Use these problems as a motivation to learn the fundamentals of optimisation covered in lectures.
- Learn how to formulate and solve optimisation problems in practice.
- Familiarise yourself with practical software tools used for optimisation.
- Solve optimisation problems using Python (and/or Matlab).
- Connect theoretical knowledge and practical usage by doing it yourself. #### Common objectives of all workshops Gain hands-on experience and learn by doing! Understand how theoretical knowledge discussed in lectures relates to practice. Develop motivation for gaining further theoretical and practical knowledge beyond the subject material.

> **Self-learning** is one of the most important skills that you should acquire as a student. Today, self-learning is much easier than it used to be thanks to a plethora of online resources.

Assessment Process

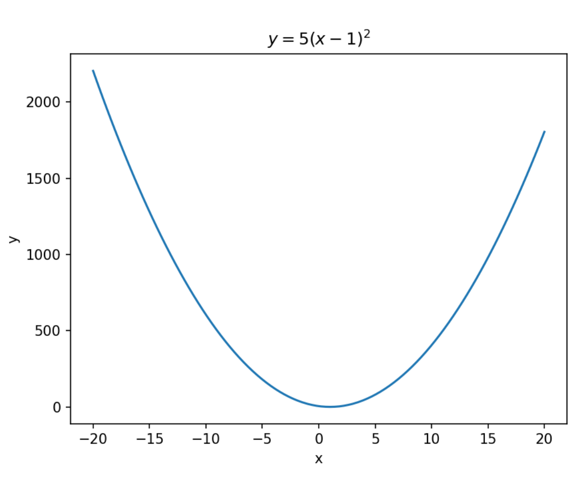
1. Follow the procedures described below, perform the given tasks, and answer the workshop questions **in this Python notebook itself! The resulting notebook will be your Workshop Report!**
2. Submit the workshop report at the announced deadline
3. Demonstrators will conduct a brief (5min) oral quiz on your submitted report in the subsequent weeks.
4. Your workshop marks will be a combination of the report you submitted and oral quiz results.

> **The goal is to learn**, NOT blindly follow the procedures in the fastest possible way! **Do not simply copy-paste answers (from Internet, friends, etc.). You can and should use all available resources but only to develop your own understanding. If you copy-paste, you will pay the price in the oral quiz!**

Section 1: Convex Functions

Remember the definition of convex and concave functions from lecture slides. Functions are mathematical objects but they are used in engineering in very practical ways, for example, to represent the relationship between two quantities. Let's draw a function!

In [91]:
```python
%matplotlib notebook
import numpy as np
import matplotlib.pyplot as plt

# define function f(x)
def f(x):
    return 5*(x-1)**2

# define x and y
x = np.linspace(-20, 20, 100) # 100 equally spaced points on interval [-20,20]
y = f(x) # call function f(x) and set y to the function's return value

# Plot the function y=f(x)
plt.plot(x,y)
plt.xlabel('x')
plt.ylabel('y')
plt.title('$y=5(x-1)^2$')
plt.show()
```

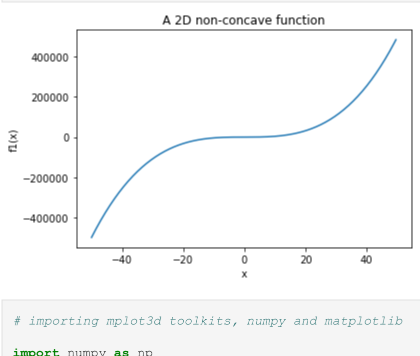


Question 1.1 [5%]

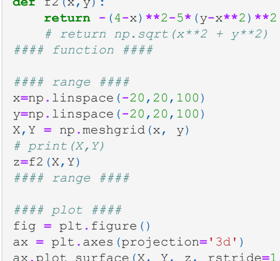
Plot one concave and one non-concave function of your choosing (preferably one in 2 dimensions and the other in 3 dimensions so that it can be visualised, check e.g. this tutorial for hints). Provide their formulas below.

Hint: an interesting and well-known function is [Rosenbrock function].(https://en.wikipedia.org/wiki/Rosenbrock_function) It is already built-in to Scipy optimize as a benchmark.

In [3]:
```python
import numpy as np
import math
import matplotlib.pyplot as plt

#### function ####
def f1(x):
    a = 4*x**3-3*x+7
    return a
#### function ####

#### range ####
x=np.array(np.arange(-50,50,0.5))
y = f1(x)
#### range ####

#### plot ####
plt.figure()
plt.plot(x,y)
plt.xlabel('x')
plt.ylabel('f1(x)')
plt.title('A 2D non-concave function')
plt.show()
#### plot ####
```



In [4]:
```python
# importing mplot3d toolkits, numpy and matplotlib

import numpy as np
import math
import matplotlib.pyplot as plt
from mpl_toolkits import mplot3d

#### function ####
def f2(x,y):
    return -(4-x)**2-5*(y-x**2)**2
    # return np.sqrt(x**2 + y**2)
#### function ####

#### range ####
x=np.linspace(-20,20,100)
y=np.linspace(-20,20,100)
X,Y = np.meshgrid(x, y)
# print(X,Y)
Z=f2(X,Y)
#### range ####

#### plot ####
fig = plt.figure()
ax = plt.axes(projection='3d')
ax.plot_surface(X, Y, Z, rstride=1, cstride=1,cmap='viridis')
ax.set_title('A 3D concave function')
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z');
plt.show()
```



I have considered the function f1(x)=4x**3-3x+7 as a 2 dimentional non-concave function. This function is neither convex nor concave. That is why I chose it.

For solving the second section of this question, I considered the function f(x,y)=-(4-x)**2-5*(y-x**2)**2 , which is the negative of the Rosenbrock function, as a 3 dimentional concave function. Because the Rosenbrock function is convex, its negative is concave.

Question 1.2 [5%]

How would you determine whether a single or multi-variate continuously differentiable function is convex or not?

> Note that the question becomes very tricky if you have a **parametric** multivariate polynomial of degree four or higher!
>
> *[Optional]* An interesting paper (for those who wish to go deeper)
> http://web.mit.edu/~a_a_a/Public/Publications/convexity_nphard.pdf

Answer: At the first step, I find the candidate points which make the gradient of the function zero. Then I calculate the Hessian matrix at these points. If all of the Hessian matrices at the candidate points are positive semidefinite, the function is convex. (A Hessian matrix is positive semidefinite if and only if all of its eigenvalues are non-negative).

If even one of the Hessian matrices is not positive semidefinite, the function will not be convex.

Question 1.3 [5%]

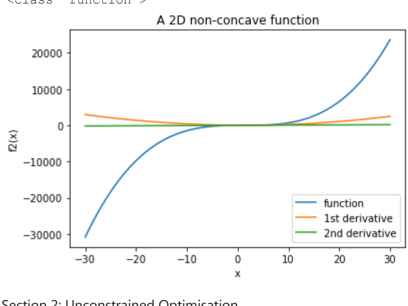
Why are convex optimisation problems considered to be easy to solve? Consider optimality conditions of unconstrained functions in your answer. Plot the first- and second-order derivative functions for one concave and one non-concave function (this time only in 2 dimensions) to further support your argument.

Because all local minimum (maximum) points in a convex (concave) function are global minimum (maximum) points, we can use a local search algorithm to solve the optimization problem. On the other hand, if the function is neither convex nor concave, using a local algorithm search cannot guarantee to find the global minimum (maximum).

In [9]:
```python
## The considered 2 dimentional concave function : y=-x**2+6*x-4

from matplotlib import pyplot as plt
import numpy as np
import sympy as sp
[illegible]
```

[illegible]

```
The second order derivative of f(x) is 6*x - 8
<class 'function'>
```



Section 2: Unconstrained Optimisation

Example 2.1: Aloha communication protocol

Wireless Network

Aloha is a well-known random access or *MAC* (Media/multiple Access Control) communication protocol. It enables multiple nodes to share a broadcast channel without any additional signaling in a distributed manner. Unlike *FDMA* or *TDMA* (frequency or time-division multiple access), the channel is not divided into segments beforehand and collisions of packets due to simultaneous transmissions by nodes are allowed. In slotted Aloha, the nodes can only transmit at the beginning of time slots, which are kept by a global/shared clock. See Aloha for further background information.

Slotted Aloha Efficiency

For an N-node slotted Aloha system, where each node transmits with a probability p, the throughput of the system is given by

$$S(p) = Np(1-p)^{N-1}$$

Question 2.1 [5%]

Formally define the optimisation problem to find the optimal probability p that maximises the throughput. Clearly identify the objective and decision variable(s). Is the objective convex or concave? Show through derivation. Find the optimality conditions for this problem.

Note that there is the constraint $0 \le p \le 1$ on probability p but we will ignore it for now.

$S(P) = NP(1-P)^{N-1}$

$\max_{P \in \mathbb{R}} S(P) \Rightarrow \begin{cases} S(P) & \text{objective function} \\ P & \text{variable} \end{cases}$

$S'(P) = N\{(1-P)^{N-1} - P(N-1)(1-P)^{N-2}\}$

$\Rightarrow \boxed{S'(P) = N(1-P)^{N-2} \times (1-PN)}$ ①

$\Rightarrow S''(P) = N\{-(N-2)(1-P)^{N-3} \times (1-PN)$

$-(1-P)^{N-2} \times N\}$

$\Rightarrow \boxed{S''(P) = N \times (1-P)^{N-3} \times [2-2N+P(N^2-N)]}$ ②

candidat points : $S'(P) = 0 \xrightarrow{①}$

$\begin{cases} (1-P)^{N-2} \times (1-PN) = 0 & N > 2 \\ 1-NP=0 \Rightarrow 1-2P=0 & N=2 \\ \text{infeasible} & N=1 \end{cases}$

$N > 2 \Rightarrow \begin{cases} P=1 & ③ \\ \text{OR} \\ P=\frac{1}{N} & ④ \end{cases}$ $N=2 \Rightarrow P=\frac{1}{2}$ ⑤

N is a positive integer ⑥

③, ⑥ $\rightarrow P=1, N \geq 3$

④, ⑤ $\rightarrow P=\frac{1}{N} \quad N \geq 2$

$S_1 = \{P=1 \mid N \geq 3\}$

$S_2 = \{P=\frac{1}{N} \mid N \geq 2\}$

$S''(P)\Big|_{S_1, N=3} = +6 > 0 \rightarrow$ local minimum

$S''(P)\Big|_{S_1, N>3} = 0 \rightarrow$ saddle point

$S''(P)\Big|_{S_2} = \dfrac{-N \times (N-1)^{(N-2)}}{(N)^{(N-3)}}$

always $\leq 0 \Rightarrow S_2 \sim$ local maximum

So : if $\begin{cases} N=3 \Rightarrow P=1 \rightarrow \text{local minimum} & ⓐ \\ N>3 \Rightarrow P=1 \rightarrow \text{saddle point} & ⓑ \\ N \geq 2 \Rightarrow P=\frac{1}{N} \rightarrow \text{local maximum} & ⓒ \end{cases}$

Thus, the answer depends on the amount of N :

ⓐ ⓑ ⓒ

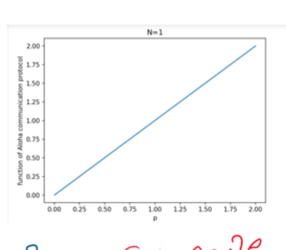
$N=1 \rightarrow$ There is no optimal point which is rational and meaningful.

$N=2 \rightarrow P=\frac{1}{2}$ is the only local maximum point.

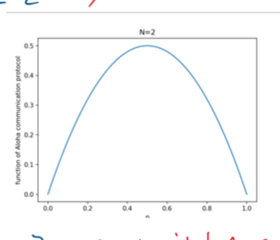
$N=3 \rightarrow P=\frac{1}{3} \rightarrow$ only local maximum point and $P=1 \rightarrow$ only local minimum point.

$N>3 \rightarrow P=\frac{1}{N}$ the only local maximum point and $P=1$ the only saddle point

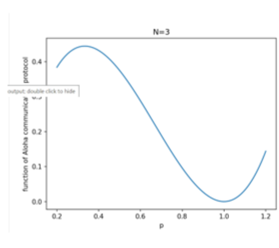
$N=1 \rightarrow$ convex and concave



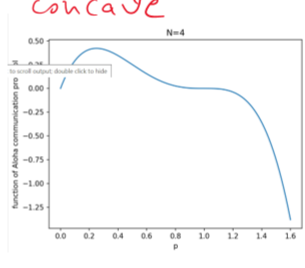
$N=2 \rightarrow$ concave

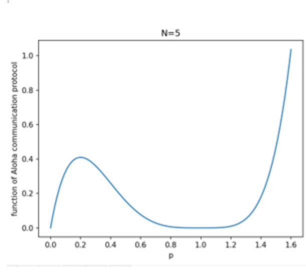


$N=3 \rightarrow$ neither convex nor concave



$N>3 \rightarrow$ neither convex nor concave



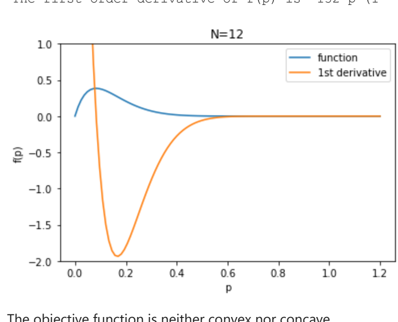


Question 2.2 [5%]

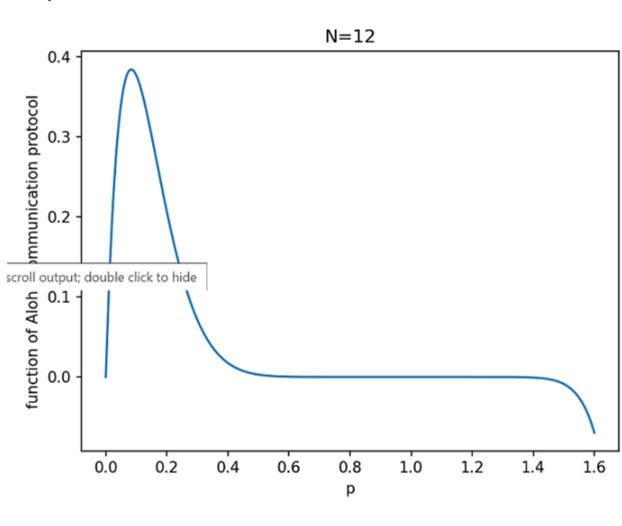
Plot the performance function and its derivative for $N = 12$ nodes. Is the objective function convex or concave? Determine using mathematical methods.

In [27]:
```python
from matplotlib import pyplot as plt
import numpy as np
import sympy as sp

########## Define p as a symbol to make it possible to calculate its derivative ########
p=sp.Symbol("p")
########## Define p as a symbol to make it possible to calculate its derivative ########

########## function ########
def f(p):
    return 12*p*(1-p)**11
########## function ########

########## first derivative ########
first_order_derivative_y=sp.diff(f(p),p)
print('The first order derivative of f(p) is',first_order_derivative_y)
print()
########## first derivative ########

#### make the derivatives possibe to be plotted
f_div1 = sp.lambdify(p,first_order_derivative_y,'numpy')
#### make the derivatives possibe to be plotted

#### plot ####
pp=np.linspace(0,1.2,100)
f=f(pp)
f_1st_prime = f_div1(pp)
plt.plot(pp,f)
plt.plot(pp,f_1st_prime)
plt.xlabel('p')
plt.ylabel('f(p)')
plt.title('N=12')
plt.legend(["function", "1st derivative","2nd derivative"], loc ="upper right")
plt.ylim((-2,1))
plt.show()
#### plot ####
```

```
The first order derivative of f(p) is -132*p*(1 - p)**10 + 12*(1 - p)**11
```



The objective function is neither convex nor concave.



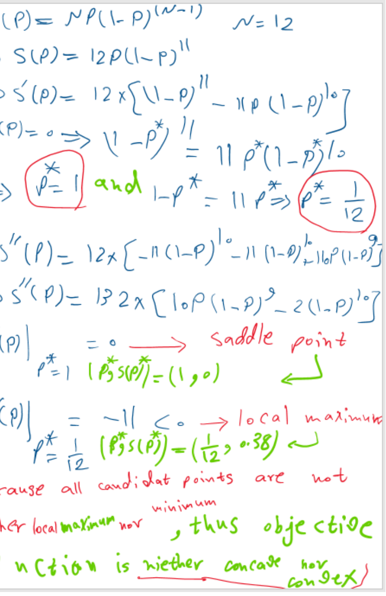
Question 2.3 [5%]

Find the optimal probability p for $N = 12$ nodes. Use an appropriate package from Scipy. Cross-check your answer with a mathematical formula that you should derive by hand.

Hint: see examples and documentation for scalar case.

In [28]:
```python
#### function ####
def f(x):
    return -12*x*(1-x)**11
#### function ####

#### using a package of Scipy ####
from scipy.optimize import minimize_scalar
res = minimize_scalar(f)
#### using a package of Scipy ####

#### Result ####
print('The maximum point is : P = ',res.x)
print('The maximum amount of function is = ',-f(res.x))
#### Result ####
```

```
The maximum point is : P =  0.08333333215749669
The maximum amount of function is =  0.3839952305608767
```



$S(p) = Np(1-p)^{(N-1)}$ $N=12$

$\Rightarrow S(p) = 12p(1-p)^{11}$

$\Rightarrow S'(p) = 12 \times [(1-p)^{11} - 11p(1-p)^{10}]$

$S'(p) = 0 \Rightarrow (1-p^*)^{11} = 11p^*(1-p^*)^{10}$

$\Rightarrow \boxed{p^* = 1}$ and $1 - p^* = 11p^* \Rightarrow \boxed{p^* = \frac{1}{12}}$

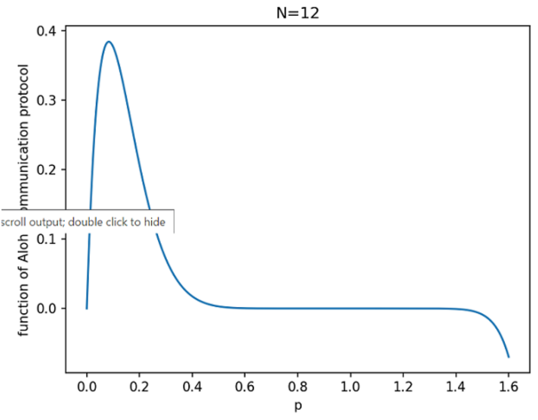
$S''(p) = 12 \times [-11(1-p)^{10} - 11(1-p)^{10} + 110p(1-p)^9]$

$\Rightarrow S''(p) = 132 \times [10p(1-p)^9 - 2(1-p)^{10}]$

$S''(p)\big|_{p^*=1} = 0 \longrightarrow$ saddle point $(p^*, S(p^*)) = (1, 0)$

$S''(p)\big|_{p^*=\frac{1}{12}} = -11 < 0 \rightarrow$ local maximum $(p^*, S(p^*)) = (\frac{1}{12}, 0.38)$

Because all candidat points are not neither local maximum nor minimum, thus objective function is niether concave nor convex



Section 3: Constrained Optimisation

Pyomo is a Python-based tool for modeling and solving optimisation problems. Algebraic modeling languages (AMLs) like Pyomo are high-level languages for specifying and solving mathematical optimisation problems. Widely used commercial AMLs include AIMMS, AMPL, and GAMS.

Pyomo uses the following mathematical concepts that are central to modern modeling activities:

- Variables: These represent unknown or changing parts of a model (e.g., which decisions to take, or the characteristic of a system outcome).
- Parameters: These are symbolic representations for real-world data, which might vary for different problem instances or scenarios.
- Relations: These are equations, inequalities, or other mathematical relationships that define how different parts of a model are related to each other.

Pyomo supports an object-oriented design for the definition of optimisation models. A Pyomo model object contains a collection of modeling components that define the optimisation problem. The Pyomo package includes modeling components that are necessary to formulate an optimisation problem: variables, objectives, and constraints, as well as other modeling components that are commonly supported by modern AMLs, including index sets and parameters.

The pyomo online documentation gives you an excellent starting point. There is also an entire book for those who are interested.

See the instructions to install Pyomo, based on Pyomo's instructions using conda.

> **2021 Installation Problem for Windows:** Ipopt installation is a bit broken under Windows. All you need to do is copy the ipopt.exe we have provided in files (originally from pre-compiled sources, 3.11.1 64 or 32 bit version) and copy that to Anaconda's `\Library\bin` directory. This directory can be found using the command `where conda`. After copying ipopt.exe, close and reopen anaconda (and this notebook) before continuing.

Hint: to process Pyomo results, see working with Pyomo models, e.g. you can access Lagrange multipliers (duals) by passing an argument to solver.

Suggested exercise: verify your answer in 2.3 by formulating that problem as a Pyomo model.

Example 3.1: Economic Dispatch in Power Generation

The problem is formulated as

$$\min_P \sum_{i=1}^{N} c_i P_i$$

$$\text{subject to } P_{i,max} \geq P_i \geq 0,\ \forall i,\ \text{and} \sum_{i=1}^{N} P_i = P_{demand}$$

Here, $P_1, \dots P_N$ are the power generated by Generators $1, \dots, N$, c_i is the per-unit generation cost of the i-*th* generator, and P_{demand} is the instantaneous power demand that needs to be satisfied by aggregate generation. More complex formulations take into account transmission, generator ramp-up and down constraints, and reactive power among other things.

Question 3.1 [20%]

Let us get inspired from generation in Victoria with $N = 12$ biggest generators that have more than 200MW capacity. Choose their maximum generation randomly or from the Victoria generator report if you wish to be more realistic. Generate a random cost vector c varying between $10 - 50$ AUD per MWh (use a group-specific random seed). *(Optionally, you can search and find how much different generation types cost if you are interested and add a bit of random noise to it).* Let the demand be $P_{demand} = 5000MW$.

Solve this simplified economic dispatch problem defined above. The resulting merit order is the generation that would have been if there was no NEM (electricity market).

1. Solve the problem using *Pyomo*.
2. What type of an optimisation problem is this? Briefly explain.

> Further information: you can find more about Australian wholesale electricity market and generation at https://www.aemo.com.au/ See also this NEM overview introductory document (right click to download) and the Victoria generator report as of January 2019.

Note: *if you are in the minority of people who have problem installing pyomo, then you can use scipy or even Matlab.*

The function is convex. Also, the constraint set is convex. Therefore, this optimization problem is convex. Moreover, the problem is linear according to the fact that convex optimizatoin is a generalization of a linear program.

In [4]:

```python
from pyomo.environ import *
import pyomo.environ as pyo
from pyomo.opt import SolverFactory
import numpy as np

# parameters
N=12
P_demand=5000

#Generate N=12 random numbers between 10 and 50
np.random.seed(1)
c = np.random.randint(10, 50, N)
print('The c coefficient vector is = ',c)

P_min = np.zeros(N)
print('The minimum power of each generator is = ',P_min)

#Generate N=12 random numbers between 200 and 1000
P_max = np.random.randint(200, 1000, N)
print('The maximum power of each generator is = ',P_max)

model = ConcreteModel()

# index
model.I = range(N)

# variables
model.P = pyo.Var(model.I)

# objective function
model.obj = Objective(expr=sum(c[i]*model.P[i] for i in model.I),sense = minimize )

# inequality constraints
model.con_lower_band = ConstraintList()
for i in model.I:
    model.con_lower_band.add( model.P[i] - P_min[i] >= 0 )

model.con_upper_band = ConstraintList()
for i in model.I:
    model.con_upper_band.add( P_max[i] - model.P[i] >= 0 )

# equality constraint
model.balance_con = Constraint(expr=sum( model.P[i] for i in model.I ) == P_demand)

# getting dual variables (Lagrange multipliers) in the concrete model
model.dual = Suffix(direction=Suffix.IMPORT)

# define solver
opt = pyo.SolverFactory('glpk')
opt.solve(model)

# show results
model.display()

print()
print('######################################')
print('dual variables (Lagrange multipliers)')
print('######################################')
print()

model.dual.pprint()
```

```
The c coefficient vector is =  [47 22 18 19 21 15 25 10 26 11 22 17]
The minimum power of each generator is =  [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
The maximum power of each generator is =  [949 708 590 481 378 476 454 557 668 452 690 868]
Model unknown

  Variables:
    P : Size=12, Index=P_index
        Key : Lower : Value : Upper : Fixed : Stale : Domain
          0 :  None :   0.0 :  None : False : False :  Reals
          1 :  None : 708.0 :  None : False : False :  Reals
          2 :  None : 590.0 :  None : False : False :  Reals
          3 :  None : 481.0 :  None : False : False :  Reals
          4 :  None : 378.0 :  None : False : False :  Reals
          5 :  None : 476.0 :  None : False : False :  Reals
          6 :  None :   0.0 :  None : False : False :  Reals
          7 :  None : 557.0 :  None : False : False :  Reals
          8 :  None :   0.0 :  None : False : False :  Reals
          9 :  None : 452.0 :  None : False : False :  Reals
         10 :  None : 490.0 :  None : False : False :  Reals
         11 :  None : 868.0 :  None : False : False :  Reals
```

```
  Objectives:
    obj : Size=1, Index=None, Active=True
        Key  : Active : Value
        None :   True : 86491.0

  Constraints:
    con_lower_band : Size=12
        Key : Lower : Body  : Upper
          1 :   0.0 :   0.0 :  None
          2 :   0.0 : 708.0 :  None
          3 :   0.0 : 590.0 :  None
          4 :   0.0 : 481.0 :  None
          5 :   0.0 : 378.0 :  None
          6 :   0.0 : 476.0 :  None
          7 :   0.0 :   0.0 :  None
          8 :   0.0 : 557.0 :  None
          9 :   0.0 :   0.0 :  None
         10 :   0.0 : 452.0 :  None
         11 :   0.0 : 490.0 :  None
         12 :   0.0 : 868.0 :  None
    con_upper_band : Size=12
        Key : Lower : Body  : Upper
          1 :   0.0 : 949.0 :  None
          2 :   0.0 :   0.0 :  None
          3 :   0.0 :   0.0 :  None
          4 :   0.0 :   0.0 :  None
          5 :   0.0 :   0.0 :  None
          6 :   0.0 :   0.0 :  None
          7 :   0.0 : 454.0 :  None
          8 :   0.0 :   0.0 :  None
          9 :   0.0 : 668.0 :  None
         10 :   0.0 :   0.0 :  None
         11 :   0.0 : 200.0 :  None
         12 :   0.0 :   0.0 :  None
    balance_con : Size=1
        Key  : Lower  : Body   : Upper
        None : 5000.0 : 5000.0 : 5000.0

#####################################
dual variables (Lagrange multipliers)
#####################################

dual : Direction=Suffix.IMPORT, Datatype=Suffix.FLOAT
    Key                  : Value
           balance_con :  22.0
    con_lower_band[10] :   0.0
    con_lower_band[11] :   0.0
    con_lower_band[12] :   0.0
     con_lower_band[1] :  25.0
     con_lower_band[2] :   0.0
     con_lower_band[3] :   0.0
     con_lower_band[4] :   0.0
     con_lower_band[5] :   0.0
     con_lower_band[6] :   0.0
     con_lower_band[7] :   3.0
     con_lower_band[8] :   0.0
     con_lower_band[9] :   4.0
    con_upper_band[10] :  11.0
    con_upper_band[11] :   0.0
    con_upper_band[12] :   5.0
     con_upper_band[1] :   0.0
     con_upper_band[2] :  -0.0
     con_upper_band[3] :   4.0
     con_upper_band[4] :   3.0
     con_upper_band[5] :   1.0
     con_upper_band[6] :   7.0
     con_upper_band[7] :   0.0
     con_upper_band[8] :  12.0
     con_upper_band[9] :   0.0
```

Example 3.2: Non-Convex Optimisation_

The Rosenbrock function is a non-convex function, introduced by Howard H. Rosenbrock in 1960, which is used as a performance test problem for global optimisation algorithms. A two-variable, arbitrarily-constrained variant is

$$\min_{x\in\mathcal{A}} f(x) = (1-x_1)^2 + 100(x_2-x_1^2)^2,$$

where the constraint set is defined by

$$\mathcal{A} := \{x\in\mathbb{R}^2 | x_2 \geq x_1 + 1,\ x_1\in[-2,3], x_2\in[-2,2]\}.$$

Let us use the following abstract Pyomo model for this problem:

In [1]:

```python
# A Pyomo model for the Rosenbrock problem
import pyomo.environ as pyo
from pyomo.opt import SolverFactory

model = pyo.AbstractModel()
model.name = 'Rosenbrock'

# note boundaries of variables and initial condition $x_0=[-2,2]$
model.x1 = pyo.Var(bounds=(-2,3), initialize=-2)
model.x2 = pyo.Var(bounds=(-2,2), initialize=2)

def rosenbrock(model):
    f = (1.0-model.x1)**2 + 100.0*(model.x2 - model.x1**2)**2
    return f

def ineqconstr(model):
    return model.x2 >= model.x1+1

model.obj = pyo.Objective(rule=rosenbrock, sense=pyo.minimize)
model.constraint = pyo.Constraint(rule=ineqconstr)
```

This model can be solved in multiple different ways. Since this is a Pyomo AbstractModel, we must create a concrete instance of it before solving.

In [2]:

```python
# create an instance of the problem
arosenbrockproblem = model.create_instance()
# this is to access Lagrange multipliers (dual variables)
arosenbrockproblem.dual = pyo.Suffix(direction=pyo.Suffix.IMPORT)

# define solver
opt = pyo.SolverFactory('ipopt') # we can use other solvers here as well

results = opt.solve(arosenbrockproblem)

# show results
arosenbrockproblem.display()
```

```
Model Rosenbrock

  Variables:
    x1 : Size=1, Index=None
        Key  : Lower : Value                : Upper : Fixed : Stale : Domain
        None :    -2 : -0.6147903137877664 :     3 : False : False :  Reals
    x2 : Size=1, Index=None
        Key  : Lower : Value              : Upper : Fixed : Stale : Domain
        None :    -2 : 0.3852097040554131 :     2 : False : False :  Reals

  Objectives:
    obj : Size=1, Index=None, Active=True
        Key  : Active : Value
        None :   True : 2.612793245502972

  Constraints:
    constraint : Size=1
        Key  : Lower : Body                   : Upper
        None :  None : -1.7843179489496208e-08 :   0.0
```

Now, let's see the Lagrange multipliers.

In [3]:

```python
def display_lagrange(instance):
    # display all duals
    print ("Duals")
    for c in instance.component_objects(pyo.Constraint, active=True):
        print ("   Constraint",c)
        for index in c:
            print ("      ", index, instance.dual[c[index]])

display_lagrange(arosenbrockproblem)
```

```
Duals
   Constraint constraint
      None -1.4485148520538937
```

Display the solution directly

In [4]:

```python
def disp_soln(instance):
    output = []
    for v in instance.component_data_objects(pyo.Var, active=True):
        output.append(pyo.value(v))
        print(v, pyo.value(v))
    print (instance.obj, pyo.value(instance.obj))
    output.append(pyo.value(instance.obj))
    return output


disp_soln(arosenbrockproblem)
```

```
x1 -0.6147903137877664
x2 0.3852097040554131
obj 2.612793245502972
```

Out[4]:

```
[-0.6147903137877664, 0.3852097040554131, 2.612793245502972]
```

Since this is a non-convex optimisation problem, the solver can only find local solutions! What happens if we change the starting point to $x_0 = [1.5, 1.5]$?

In [5]:

```python
arosenbrockproblem.x1 = 1.5
arosenbrockproblem.x2 = 1.5

results = opt.solve(arosenbrockproblem)

arosenbrockproblem.display()
```

```
Model Rosenbrock

  Variables:
    x1 : Size=1, Index=None
        Key  : Lower : Value              : Upper : Fixed : Stale : Domain
        None :    -2 : 1.0000000299152387 :     3 : False : False :  Reals
    x2 : Size=1, Index=None
        Key  : Lower : Value : Upper : Fixed : Stale : Domain
        None :    -2 :   2.0 :     2 : False : False :  Reals

  Objectives:
    obj : Size=1, Index=None, Active=True
        Key  : Active : Value
        None :   True : 99.99998803390469

  Constraints:
    constraint : Size=1
        Key  : Lower : Body                  : Upper
        None :  None : 2.991523873063784e-08 :   0.0
```

In [6]:

```python
display_lagrange(arosenbrockproblem)
```

```
Duals
   Constraint constraint
      None -399.99999594959036
```

In [7]:

```python
disp_soln(arosenbrockproblem)
```

```
x1 1.0000000299152387
x2 2.0
obj 99.99998803390469
```

Out[7]:

```
[1.0000000299152387, 2.0, 99.99998803390469]
```

Example 3.3: Waterfilling in Communications

by Robert Gowers, Roger Hill, Sami Al-Izzi, Timothy Pollington and Keith Briggs. From the book by Boyd and Vandenberghe, Convex Optimization, Example 5.2 page 245.

$$\min_x \sum_{i=1}^N -\log(\alpha_i + x_i)$$

$$\text{subject to } x_i \geq 0,\ \forall i, \text{ and } \sum_{i=1}^N x_i = P$$

This problem arises in information/communication theory, in allocating power to a set of n communication channels. The variable x_i represents the transmitter power allocated to the *i-th* channel, and $\log(\alpha_i + x_i)$ gives the capacity or communication rate of the channel, where $\alpha_i > 0$ represents the floor above the baseline at which power can be added to the channel. The problem is to allocate a total power of one to the channels, in order to maximize the total communication rate.

This can be solved using a classic water filling algorithm.

Waterfilling

Question 3.2 [25%]

1. Is the problem in Example 3.3 convex? Formally explain/argue why or why not. What does this imply regarding the solution?
2. Solve the problem above for $N = 8$ and a randomly chosen α vector (use a group-specific random seed). You *can* use Pyomo for this. Cross-check your answer with another software (package), e.g. Matlab or Scipy.
3. Write the Lagrangian, KKT conditions, and find numerically the Lagrange multipliers associated with the solution (using the software package/function). Which constraints are active? Explain and discuss briefly.

Because the objective function is convex and constraint set is convex, too, so the optimization problem is convex. Therefore, KKT condition is necessary and sufficient to find the global minimum.

In [15]:

```
from pyomo.environ import *
import pyomo.environ as pyo
from pyomo.opt import SolverFactory
import numpy as np

# parameters
N=8
P=1

#Generate N=8 random numbers between 1 and 10
np.random.seed(1)
alpha = np.random.rand(N)
print('The alpha coefficient vector is = ',alpha)

model = ConcreteModel()

# index
model.I = range(N)

# variables
model.x = pyo.Var(model.I,within=NonNegativeReals)

# objective function
model.obj = Objective(expr=sum(-log10(alpha[i]+model.x[i]) for i in model.I),sense = minimize )

# equality constraint
model.balance_con = Constraint(expr=sum( model.x[i] for i in model.I ) == P)

# getting dual variables (Lagrange multipliers) in the concrete model
model.dual = pyo.Suffix(direction=pyo.Suffix.IMPORT)

# define solver
opt = pyo.SolverFactory('ipopt')
opt.solve(model)

# show results
model.display()

print()
print('####################################')
print('dual variables (Lagrange multipliers)')
print('####################################')
print()


#model.dual.pprint()

def display_lagrange(model):
    # display all duals
    print ("Duals")
    for c in model.component_objects(pyo.Constraint, active=True):
        print (" Constraint",c)
        for index in c:
            print (" ", index, model.dual[c[index]])
display_lagrange(model)
```

```
The alpha coefficient vector is =  [4.17022005e-01 7.20324493e-01 1.14374817e-04 3.02332573e-01
 1.46755891e-01 9.23385948e-02 1.86260211e-01 3.45560727e-01]
Model unknown

  Variables:
    x : Size=8, Index=x_index
        Key : Lower : Value                  : Upper : Fixed : Stale : Domain
          0 :     0 :  4.314733333208492e-07 :  None : False : False : NonNegativeReals
          1 :     0 : 1.3548455605249003e-07 :  None : False : False : NonNegativeReals
          2 :     0 :     0.3454000925250562 :  None : False : False : NonNegativeReals
          3 :     0 :    0.04318242468765052 :  None : False : False : NonNegativeReals
          4 :     0 :    0.19875863239478617 :  None : False : False : NonNegativeReals
          5 :     0 :     0.2531759001581889 :  None : False : False : NonNegativeReals
          6 :     0 :    0.15925434447748454 :  None : False : False : NonNegativeReals
          7 :     0 :  0.0002280387989443478 :  None : False : False : NonNegativeReals

  Objectives:
    obj : Size=1, Index=None, Active=True
        Key  : Active : Value
        None :   True : 3.2911693687927355

  Constraints:
    balance_con : Size=1
        Key  : Lower : Body : Upper
        None :   1.0 :  1.0 :   1.0

####################################
dual variables (Lagrange multipliers)
####################################

Duals
 Constraint balance_con
  None -1.2569504716047002
```

$$L(\lambda, \mu, x) = f(x) + \sum_i \lambda_i h_i(x) + \sum_j \mu_j g_j(x)$$

$$f(x) = \sum_{i=1}^{N} -\log(\alpha_i + x_i)$$

$$h(x) = \sum_{i=1}^{N} x_i - P = 0$$

$$g_i(x) = -x_i \leq 0 \qquad i = 1, 2, \ldots, N$$

$$\text{KKT conditions:} \begin{cases} \nabla f(x^*) + \lambda h(x^*) + \sum_{j=1}^{N} \mu_j g_j(x) = 0 \\ h(x^*) = 0 \Rightarrow \sum_{i=1}^{N} x_i^* - P = 0 \\ -x_j \leq 0 \quad \forall j \\ \mu_j \geq 0 \\ \mu_j (-x_j) = 0 \quad \text{complementary slackness condition} \end{cases}$$

Accordint to the results, all of the lagrangian multipliers of the inequalty constraints are zero, and thus, all inequalty constraints are inactive. However, the lagrangian multiplier of the equalty constraint is negative which shows the equalty constraint is active. (Equalty constraints are always active).

Although the above code is correct and works well, I wanted to check whether it is possible to form the Lagrangian function by combining the function and constraints with the lagrangian multipliers or not. (Of course, you said that it is not necessary because pyomo already solves the dual problem) However, I wanted to write the below code to check it with you. That is why I I have written the below code.

In [53]:

```
# from pyomo.environ import *
# import pyomo.environ as pyo
# from pyomo.opt import SolverFactory
# import numpy as np

# # parameters
# N=8
# P=10

# #Generate N=8 random numbers between 1 and 10
# alpha = [6, 9, 6, 1, 1, 2, 8, 7]
# print('The alpha coefficient vector is = ',alpha)

# model = ConcreteModel()

# # index
# model.I = range(N)

# # variables
# model.x = pyo.Var(model.I, within=NonNegativeReals)
# model.Lambda=pyo.Var()

# # objective function
# #model.f = Objective(expr=sum(-log10(alpha[i]+model.x[i]) for i in model.I) + model.Lambda*(sum( model.x[i] i

# def ObjRule(model):
#     return sum(-log10(alpha[i]+model.x[i]) for i in model.I) + model.Lambda*(sum( model.x[i] for i in model.I
# model.obj2 = pyo.Objective(rule=ObjRule, sense=pyo.minimize)

# # define solver
# opt = pyo.SolverFactory('ipopt')
# opt.solve(model)

# # show results
# model.display()
```

```
The alpha coefficient vector is =  [6, 9, 6, 1, 1, 2, 8, 7]
WARNING: Loading a SolverResults object with a warning status into
    model.name="unknown";
      - termination condition: unbounded
      - message from solver: Ipopt 3.11.1\x3a Iterates diverging; problem
        might be unbounded.
Model unknown

  Variables:
    x : Size=8, Index=x_index
        Key : Lower : Value : Upper : Fixed : Stale : Domain
          0 :     0 :  None :  None : False :  True : NonNegativeReals
          1 :     0 :  None :  None : False :  True : NonNegativeReals
          2 :     0 :  None :  None : False :  True : NonNegativeReals
          3 :     0 :  None :  None : False :  True : NonNegativeReals
          4 :     0 :  None :  None : False :  True : NonNegativeReals
          5 :     0 :  None :  None : False :  True : NonNegativeReals
          6 :     0 :  None :  None : False :  True : NonNegativeReals
          7 :     0 :  None :  None : False :  True : NonNegativeReals
    Lambda : Size=1, Index=None
        Key  : Lower : Value : Upper : Fixed : Stale : Domain
        None :  None :  None :  None : False :  True :  Reals

  Objectives:
    obj2 : Size=1, Index=None, Active=True
ERROR: evaluating object as numeric value: x[0]
        (object: <class 'pyomo.core.base.var._GeneralVarData'>)
    No value for uninitialized NumericValue object x[0]
ERROR: evaluating object as numeric value: obj2
        (object: <class 'pyomo.core.base.objective.ScalarObjective'>)
    No value for uninitialized NumericValue object x[0]
        Key : Active : Value
        None :   None :  None

  Constraints:
    None
```

Using random.seed() for pseudo-random number generation

We use Python or numpy random modules to generate random numbers or vectors/matrices. Random numbers and vectors are used a lot in various context, e.g. as asked above or in machine learning problems. They also cause a *reproducibility* problem. In this case, when your demonstrators run your code, they will get a different α vector each time and the results will be different each time. How can we prevent that? Python *random.seed* addresses this problem.

If you call *random.seed(seedvalue)* every time before calling the random number generator, you will get the same "random" number. Change the seed and the number will change as well. This makes it pseudo-random which ensures reproducibility.

See this nice article for basics. To generate random vectors and matrices numpy is very useful.

In [8]:

```
import random
import numpy as np
```

```
print ("Random numbers with a given seed")
random.seed(1215151)
print (random.random())
random.seed(1215151)
print (random.random()) # same seed, same number!
random.seed(1215151)
print (random.random()) # same seed, same number!
random.seed(5298496496)
print (random.random()) # different seed, different number!
random.seed(5298496496)
print (random.random()) # repeat!

print (random.random()) # no seed, purely random number
print (random.random()) # no seed, purely random number

np.random.seed(4198494)
print(np.random.rand(3,2))
np.random.seed(4198494)
print(np.random.rand(3,2))
# now without seed, most probaby it will be different!
print(np.random.rand(3,2))
```

```
Random numbers with a given seed
0.10885225688640254
0.10885225688640254
0.10885225688640254
0.7757440753598988
0.7757440753598988
0.9679503208418999
0.3260031142859906
[[0.56318863 0.21737672]
 [0.01327191 0.49564024]
 [0.70751371 0.34934067]]
[[0.56318863 0.21737672]
 [0.01327191 0.49564024]
 [0.70751371 0.34934067]]
[[0.98241108 0.3293288 ]
 [0.42498814 0.28998687]
 [0.10769687 0.70104546]]
```

Important Note on Random Number/Vector Generation

Each group has to use a different number seed (which is an arbitrary number as illustrated above) and groups cannot share seeds. The pseudo-randomness is used here to create diversity. Otherwise, if groups use the same seed, you would lose points in assessment

Example 3.4: Power Control in Wireless Communication

Adapted from Boyd, Kim, Vandenberghe, and Hassibi, "A Tutorial on Geometric Programming."

The power control problem in wireless communications aims to minimise the total transmitter power available across N trasmitters while concurrently achieving good (or a pre-defined minimum) performance.

The technical setup is as follows. Each transmitter i transmits with a power level P_i bounded below and above by a minimum and maximum level. The power of the signal received from transmitter j at receiver i is $G_{ij}P_j$, where $G_{ij} > 0$ represents the path gain (often loss) from transmitter j to receiver i. The signal power at the intended receiver i is $G_{ii}P_i$, and the interference power at receiver i from other transmitters is given by $\sum_{k\neq i} G_{ik}P_k$. The (background) noise power at receiver i is σ_i. Thus, the *Signal to Interference and Noise Ratio (SINR)* of the ith receiver-transmitter pair is

$$S_i = \frac{G_{ii}P_i}{\sum_{k\neq i} G_{ik}P_k + \sigma_i}.$$

The minimum SINR represents a performance lower bound for this system, S^{min}.

The resulting optimisation problem is formulated as

$$\begin{array}{ll} \min_P & \sum_{i=1}^N P_i \\ \text{subject to} & P^{min} \le P_i \le P^{max},\ \forall i \\ & \dfrac{G_{ii}P_i}{\sigma_i + \sum_{k\neq i} G_{ik}P_k} \ge S^{min},\ \forall i \end{array}$$

Question 3.3 [25%]

Let $N = 10$, $P^{min} = 0.1$, $P^{max} = 5$, $\sigma = 0.2$ (same for all). Create a random path loss matrix G, where off-diagonal elements are between 0.1 and 0.9 and the diagonal elements are equal to 1.

1. Write down the Langrangian and KKT conditions of this problem.
2. Solve the problem first with $S^{min} = 0$ using *Pyomo*. Plot the power levels and SINRs that you obtain.
3. What happens if you choose an S^{min} that is larger? Solve the problem again and document your results. What happens if you choose a very large S^{min}? Observe and comment.

Note: *if you are in the minority of people who have problem installing pyomo, then you can use scipy or even Matlab.*

$f(x) = \sum_{i=1}^{N} P_i$ $\quad g_i(x) = P_i - P^{max} \quad i = 1,\dots,N$

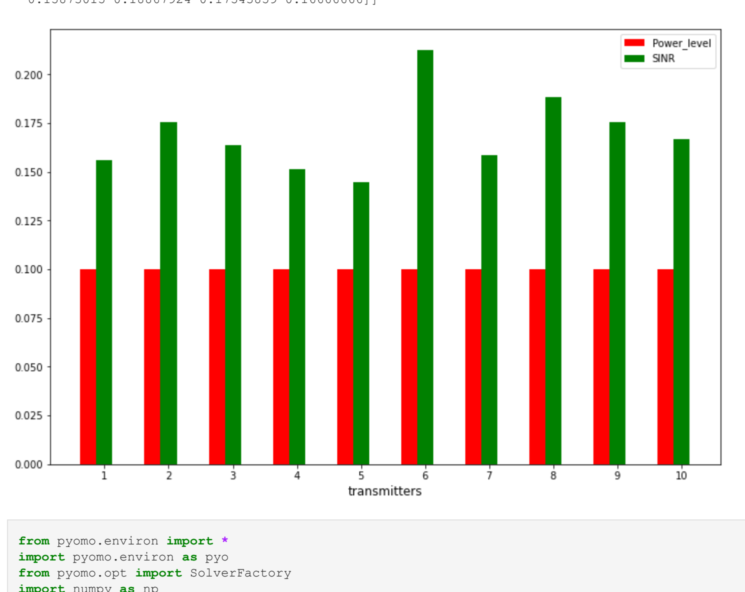
$g_i(x) = P^{min} - P_i \quad i = 1,\dots,N$

$g_i(x) = S^{min} - \dfrac{G_{ii}P_i}{\sigma_i + \sum_{k\neq i} G_{ik}P_k} \quad i = 1,\dots,N$

$\nabla f(x^*) + \sum_{i=1}^{N} g_i(x^*)\mu_i = 0$

```
from pyomo.environ import *
import pyomo.environ as pyo
from pyomo.opt import SolverFactory
import numpy as np
import matplotlib.pyplot as plt

# parameters
N=10

S_min=0
print('The S_min is = ',S_min)
print()

sigma = 0.2*np.ones(N)
print('The minimum power level of each transmitter is = ',sigma)
print()

P_min = 0.1*np.ones(N)
print('The minimum power level of each transmitter is = ',P_min)
print()

P_max = 5*np.ones(N)
print('The maximum power level of each transmitter is = ',P_max)
print()

np.random.seed(1)
G = np.zeros((N,N))
for i in range(N):
    for j in range(N):
        if i==j:
            G[i][j]=1
        else:
            #G[i][j]=np.random.uniform(0.1,0.9)
            G[i][j]=0.1*np.random.randint(1,9)
print('The path loss matrix is = ',G)
print()

model = ConcreteModel()

# indecis
model.I = range(N)
model.K = range(N)

# variables
model.P = pyo.Var(model.I)

# objective function
model.obj = Objective(expr=sum(model.P[i] for i in model.I),sense = minimize )

# inequality constraints
model.con_lower_band = ConstraintList()
for i in model.I:
    model.con_lower_band.add( model.P[i] >= P_min[i] )

model.con_upper_band = ConstraintList()
for i in model.I:
    model.con_upper_band.add( model.P[i] <= P_max[i] )

model.con_SINR = ConstraintList()
for i in model.I:
    model.con_SINR.add( (G[i][i]*model.P[i])/(sigma[i]+sum(G[i][k]*model.P[k] for k in model.K)-G[i][i]*model.P

# getting dual variables (Lagrange multipliers) in the concrete model
model.dual = Suffix(direction=Suffix.IMPORT)

# define solver
opt = pyo.SolverFactory('ipopt')
opt.solve(model)

# show results
model.display()

print()
print('####################################')
print('dual variables (Lagrange multipliers)')
print('####################################')
print()

model.dual.pprint()

# Results

Power_level=[]
SINR=[]

for i in model.I:
    Power_level.append(model.P[i].value)
    SINR.append((G[i][i]*model.P[i].value)/(sigma[i]+sum(G[i][k]*model.P[k].value for k in model.K)-G[i][i]*moc

print('######################################')
print()
print('The power levels are = ',Power_level)
print()
print('The Signal to Interference and Noise Ratios (SINRs) are = ',SINR)

#plot
# fig = plt.figure()
# ax = fig.add_axes([0,0,1,1])
# transmitters = range(N)+1
# ax.bar(transmitters,Power_level)
# plt.show()

data=np.zeros((2,N))

for i in range(N):
    data[0][i]=Power_level[i]
    data[1][i]=SINR[i]

print()
print(data)
print()

barWidth = 0.25
fig = plt.subplots(figsize =(12, 8))

# set height of bar
Power_level = data[0]
SINR        = data[1]

# Set position of bar on X axis
br1 = np.arange(N)
br2 = [x + barWidth for x in br1]

# Make the plot
plt.bar(br1, Power_level, color ='r', width = barWidth, label ='Power_level')
plt.bar(br2, SINR,        color ='g', width = barWidth, label ='SINR')

# Adding Xticks
plt.xlabel('transmitters', fontsize = 12)

plt.xticks([r + barWidth for r in range(N)],['1','2','3','4','5','6','7','8','9','10'])

plt.legend()
plt.show()
```

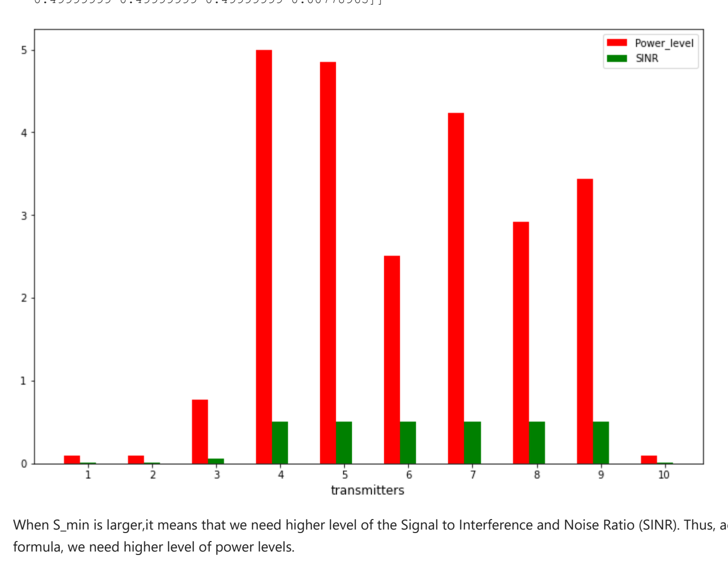
```
The S_min is =  0

The minimum power level of each transmitter is =  [0.2 0.2 0.2 0.2 0.2 0.2 0.2 0.2 0.2 0.2]

The minimum power level of each transmitter is =  [0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1]

The maximum power level of each transmitter is =  [5. 5. 5. 5. 5. 5. 5. 5. 5. 5.]

The path loss matrix is =  [[1.  0.6 0.4 0.5 0.1 0.8 0.2 0.4 0.6 0.8]
 [0.1 1.  0.1 0.2 0.5 0.8 0.6 0.5 0.7 0.2]
 [0.3 0.5 1.  0.7 0.6 0.3 0.5 0.4 0.5 0.3]
 [0.5 0.6 0.7 1.  0.3 0.5 0.8 0.8 0.2 0.2]
 [0.8 0.7 0.1 0.7 1.  0.6 0.2 0.2 0.8 0.8]
 [0.6 0.7 0.2 0.2 0.1 1.  0.5 0.2 0.1 0.1]
 [0.6 0.4 0.8 0.3 0.2 0.1 1.  0.7 0.8 0.4]
 [0.7 0.6 0.2 0.2 0.4 0.5 0.1 1.  0.2 0.4]
 [0.5 0.3 0.5 0.1 0.6 0.4 0.2 0.3 1.  0.8]
 [0.1 0.5 0.2 0.8 0.3 0.8 0.8 0.3 0.2 1. ]]

Model unknown

  Variables:
    P : Size=10, Index=P_index
        Key : Lower : Value                : Upper : Fixed : Stale : Domain
          0 :  None : 0.09999999249322769 :  None : False : False :  Reals
          1 :  None : 0.09999999249506068 :  None : False : False :  Reals
          2 :  None : 0.09999999249027092 :  None : False : False :  Reals
          3 :  None : 0.09999999249174402 :  None : False : False :  Reals
          4 :  None : 0.09999999249022305 :  None : False : False :  Reals
          5 :  None :  0.0999999924976875 :  None : False : False :  Reals
          6 :  None : 0.09999999249311016 :  None : False : False :  Reals
          7 :  None : 0.09999999249352862 :  None : False : False :  Reals
          8 :  None : 0.09999999249404566 :  None : False : False :  Reals
          9 :  None : 0.09999999249363643 :  None : False : False :  Reals

  Objectives:
    obj : Size=1, Index=None, Active=True
        Key  : Active : Value
        None :   True : 0.9999999249325349

  Constraints:
    con_lower_band : Size=10
        Key : Lower : Body                : Upper
          1 :   0.1 : 0.09999999249322769 :  None
          2 :   0.1 : 0.09999999249506068 :  None
          3 :   0.1 : 0.09999999249027092 :  None
          4 :   0.1 : 0.09999999249174402 :  None
          5 :   0.1 : 0.09999999249022305 :  None
          6 :   0.1 :  0.0999999924976875 :  None
          7 :   0.1 : 0.09999999249311016 :  None
          8 :   0.1 : 0.09999999249352862 :  None
          9 :   0.1 : 0.09999999249404566 :  None
         10 :   0.1 : 0.09999999249363643 :  None
    con_upper_band : Size=10
        Key : Lower : Body                : Upper
          1 :  None : 0.09999999249322769 :   5.0
          2 :  None : 0.09999999249506068 :   5.0
          3 :  None : 0.09999999249027092 :   5.0
          4 :  None : 0.09999999249174402 :   5.0
          5 :  None : 0.09999999249022305 :   5.0
          6 :  None :  0.0999999924976875 :   5.0
          7 :  None : 0.09999999249311016 :   5.0
          8 :  None : 0.09999999249352862 :   5.0
          9 :  None : 0.09999999249404566 :   5.0
         10 :  None : 0.09999999249363643 :   5.0
    con_SINR : Size=10
        Key : Lower : Body                : Upper
          1 :   0.0 :   0.156249996333764 :  None
          2 :   0.0 : 0.17543859187279015 :  None
          3 :   0.0 : 0.16393442218984466 :  None
          4 :   0.0 : 0.15151514806609542 :  None
          5 :   0.0 : 0.14492753307332068 :  None
          6 :   0.0 :  0.2127659506596637 :  None
          7 :   0.0 : 0.15873015494757317 :  None
          8 :   0.0 : 0.18867923993823232 :  None
          9 :   0.0 :  0.1754385918719049 :  None
         10 :   0.0 : 0.16666666249637294 :  None

####################################
dual variables (Lagrange multipliers)
####################################

dual : Direction=Suffix.IMPORT, Datatype=Suffix.FLOAT
    Key                : Value
          con_SINR[10] :  1.510341681836488e-08
           con_SINR[1] : 1.6112669034757835e-08
           con_SINR[2] : 1.4365899832144955e-08
           con_SINR[3] : 1.5352431892403227e-08
           con_SINR[4] : 1.6607126177732703e-08
           con_SINR[5] :  1.735421102342391e-08
           con_SINR[6] :  1.184689740188305e-08
           con_SINR[7] :   1.58571671307915e-08
           con_SINR[8] : 1.3348068725517419e-08
           con_SINR[9] : 1.4352687469361522e-08
    con_lower_band[10] :     0.9999999917099875
     con_lower_band[1] :     0.9999999932393383
     con_lower_band[2] :     0.9999999960408248
     con_lower_band[3] :     0.9999999884912787
     con_lower_band[4] :     0.9999999905033621
     con_lower_band[5] :     0.9999999885550357
     con_lower_band[6] :     0.9999999951177605
     con_lower_band[7] :     0.9999999918645842
     con_lower_band[8] :     0.9999999908585341
     con_lower_band[9] :     0.9999999919161895
    con_upper_band[10] : -5.112517286160283e-10
     con_upper_band[1] : -5.112514219234505e-10
     con_upper_band[2] : -5.112511724883412e-10
     con_upper_band[3] : -5.112566684288498e-10
     con_upper_band[4] :  -5.11251324629911e-10
     con_upper_band[5] : -5.112537193654299e-10
     con_upper_band[6] : -5.112508209406572e-10
     con_upper_band[7] : -5.112509741500179e-10
     con_upper_band[8] :  -5.11253831228042e-10
     con_upper_band[9] : -5.112513407105668e-10
######################################

The power levels are =  [0.09999999249322769, 0.09999999249506068, 0.09999999249027092, 0.09999999249174402, 0.09999999249022305, 0.0999999924976875, 0.09999999249311016, 0.09999999249352862, 0.09999999249404566, 0.09999999249363643]

The Signal to Interference and Noise Ratios (SINRs) are =  [0.156249996333764, 0.17543859187279015, 0.16393442218984466, 0.15151514806609542, 0.14492753307332068, 0.2127659506596637, 0.15873015494757317, 0.18867923993823232, 0.1754385918719049, 0.16666666249637294]

[[0.09999999 0.09999999 0.09999999 0.09999999 0.09999999 0.09999999
  0.09999999 0.09999999 0.09999999 0.09999999]
 [0.15625    0.17543859 0.16393442 0.15151515 0.14492753 0.21276595
  0.15873015 0.18867924 0.17543859 0.16666666]]
```



In [69]:

```
from pyomo.environ import *
import pyomo.environ as pyo
from pyomo.opt import SolverFactory
import numpy as np
import matplotlib.pyplot as plt

# parameters
N=10

S_min=0.5
print('The S_min is = ',S_min)
print()

sigma = 0.2*np.ones(N)
print('The minimum power level of each transmitter is = ',sigma)
print()

P_min = 0.1*np.ones(N)
print('The minimum power level of each transmitter is = ',P_min)
print()

P_max = 5*np.ones(N)
print('The maximum power level of each transmitter is = ',P_max)
print()

np.random.seed(1)
G = np.zeros((N,N))
for i in range(N):
    for j in range(N):
        if i==j:
            G[i][j]=1
        else:
            #G[i][j]=np.random.uniform(0.1,0.9)
            G[i][j]=0.1*np.random.randint(1,9)
print('The path loss matrix is = ',G)
print()

model = ConcreteModel()

# indecis
model.I = range(N)
model.K = range(N)

# variables
model.P = pyo.Var(model.I)

# objective function
model.obj = Objective(expr=sum(model.P[i] for i in model.I),sense = minimize )

# inequality constraints
model.con_lower_band = ConstraintList()
for i in model.I:
    model.con_lower_band.add( model.P[i] >= P_min[i] )

model.con_upper_band = ConstraintList()
for i in model.I:
    model.con_upper_band.add( model.P[i] <= P_max[i] )

model.con_SINR = ConstraintList()
for i in model.I:
    model.con_SINR.add( (G[i][i]*model.P[i])/(sigma[i]+sum(G[i][k]*model.P[k] for k in model.K)-G[i][i]*model.P

# getting dual variables (Lagrange multipliers) in the concrete model
model.dual = Suffix(direction=Suffix.IMPORT)

# define solver
opt = pyo.SolverFactory('ipopt')
opt.solve(model)

# show results
model.display()

print()
print('####################################')
print('dual variables (Lagrange multipliers)')
print('####################################')
print()

model.dual.pprint()

# Results

Power_level=[]
SINR=[]

for i in model.I:
    Power_level.append(model.P[i].value)
    SINR.append((G[i][i]*model.P[i].value)/(sigma[i]+sum(G[i][k]*model.P[k].value for k in model.K)-G[i][i]*moc

print('######################################')
print()
print('The power levels are = ',Power_level)
print()
print('The Signal to Interference and Noise Ratios (SINRs) are = ',SINR)

#plot
# fig = plt.figure()
# ax = fig.add_axes([0,0,1,1])
# transmitters = range(N)+1
# ax.bar(transmitters,Power_level)
# plt.show()

data=np.zeros((2,N))

for i in range(N):
    data[0][i]=Power_level[i]
    data[1][i]=SINR[i]

print()
print(data)
print()

barWidth = 0.25
fig = plt.subplots(figsize =(12, 8))

# set height of bar
Power_level = data[0]
SINR        = data[1]

# Set position of bar on X axis
br1 = np.arange(N)
br2 = [x + barWidth for x in br1]

# Make the plot
plt.bar(br1, Power_level, color ='r', width = barWidth, label ='Power_level')
plt.bar(br2, SINR,        color ='g', width = barWidth, label ='SINR')

# Adding Xticks
plt.xlabel('transmitters', fontsize = 12)

plt.xticks([r + barWidth for r in range(N)],['1','2','3','4','5','6','7','8','9','10'])

plt.legend()
plt.show()
```

```
The S_min is =  0.5

The minimum power level of each transmitter is =  [0.2 0.2 0.2 0.2 0.2 0.2 0.2 0.2 0.2 0.2]

The minimum power level of each transmitter is =  [0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1]

The maximum power level of each transmitter is =  [5. 5. 5. 5. 5. 5. 5. 5. 5. 5.]

The path loss matrix is =  [[1.  0.6 0.4 0.5 0.1 0.8 0.2 0.4 0.6 0.8]
 [0.1 1.  0.1 0.2 0.5 0.8 0.6 0.5 0.7 0.2]
 [0.3 0.5 1.  0.7 0.6 0.3 0.5 0.4 0.5 0.3]
 [0.5 0.6 0.7 1.  0.3 0.5 0.8 0.8 0.2 0.2]
 [0.8 0.7 0.1 0.7 1.  0.6 0.2 0.2 0.8 0.8]
 [0.6 0.7 0.2 0.2 0.1 1.  0.5 0.2 0.1 0.1]
 [0.6 0.4 0.8 0.3 0.2 0.1 1.  0.7 0.8 0.4]
 [0.7 0.6 0.2 0.2 0.4 0.5 0.1 1.  0.2 0.4]
 [0.5 0.3 0.5 0.1 0.6 0.4 0.2 0.3 1.  0.8]
 [0.1 0.5 0.2 0.8 0.3 0.8 0.8 0.3 0.2 1. ]]

WARNING: Loading a SolverResults object with a warning status into
    model.name="unknown";
      - termination condition: infeasible
      - message from solver: Ipopt 3.11.1\x3a Converged to a locally
        infeasible point. Problem may be infeasible.
Model unknown

  Variables:
    P : Size=10, Index=P_index
        Key : Lower : Value                : Upper : Fixed : Stale : Domain
          0 :  None : 0.09999999005084921 :  None : False : False :  Reals
          1 :  None : 0.09999999003308209 :  None : False : False :  Reals
          2 :  None :  0.7756516278802369 :  None : False : False :  Reals
          3 :  None :   5.000000049900695 :  None : False : False :  Reals
          4 :  None :    4.85121004276943 :  None : False : False :  Reals
          5 :  None :   2.514014613489206 :  None : False : False :  Reals
          6 :  None :   4.239747822212083 :  None : False : False :  Reals
          7 :  None :   2.917668537110725 :  None : False : False :  Reals
          8 :  None :   3.443703830966128 :  None : False : False :  Reals
          9 :  None : 0.09999999008737301 :  None : False : False :  Reals

  Objectives:
    obj : Size=1, Index=None, Active=True
        Key  : Active : Value
        None :   True : 24.041996494499806

  Constraints:
    con_lower_band : Size=10
        Key : Lower : Body                : Upper
          1 :   0.1 : 0.09999999005084921 :  None
          2 :   0.1 : 0.09999999003308209 :  None
          3 :   0.1 :  0.7756516278802369 :  None
          4 :   0.1 :   5.000000049900695 :  None
          5 :   0.1 :    4.85121004276943 :  None
          6 :   0.1 :   2.514014613489206 :  None
          7 :   0.1 :   4.239747822212083 :  None
          8 :   0.1 :   2.917668537110725 :  None
          9 :   0.1 :   3.443703830966128 :  None
         10 :   0.1 : 0.09999999008737301 :  None
    con_upper_band : Size=10
        Key : Lower : Body                : Upper
          1 :  None : 0.09999999005084921 :   5.0
          2 :  None : 0.09999999003308209 :   5.0
          3 :  None :  0.7756516278802369 :   5.0
          4 :  None :   5.000000049900695 :   5.0
          5 :  None :    4.85121004276943 :   5.0
          6 :  None :   2.514014613489206 :   5.0
          7 :  None :   4.239747822212083 :   5.0
          8 :  None :   2.917668537110725 :   5.0
          9 :  None :   3.443703830966128 :   5.0
         10 :  None : 0.09999999008737301 :   5.0
    con_SINR : Size=10
        Key : Lower : Body                 : Upper
          1 :   0.5 :  0.01027978109838697 :  None
          2 :   0.5 : 0.00822526788759482 :  None
          3 :   0.5 :  0.06213303402617492 :  None
          4 :   0.5 :  0.49999998999977896 :  None
          5 :   0.5 :   0.4999999900002078 :  None
          6 :   0.5 :   0.4999999900018562 :  None
          7 :   0.5 :   0.4999999900004585 :  None
          8 :   0.5 :   0.4999999900012687 :  None
          9 :   0.5 :  0.49999999000087114 :  None
         10 :   0.5 : 0.007789650811544387 :  None

####################################
dual variables (Lagrange multipliers)
####################################

dual : Direction=Suffix.IMPORT, Datatype=Suffix.FLOAT
    Key                : Value
          con_SINR[10] :       999.999999998153
           con_SINR[1] :      999.9999999981436
           con_SINR[2] :      999.9999999981515
           con_SINR[3] :      999.9999999979239
           con_SINR[4] :      603.6998921154137
           con_SINR[5] :      551.4549769546886
           con_SINR[6] :      351.58680429573906
           con_SINR[7] :      519.2163138674754
           con_SINR[8] :      416.2402767274553
           con_SINR[9] :      465.66753125074894
    con_lower_band[10] :     10.402462129306288
     con_lower_band[1] :      17.86675711685788
     con_lower_band[2] :     27.438454640211443
     con_lower_band[3] :  1.3454933220548672e-09
     con_lower_band[4] :  1.8551966378021678e-10
     con_lower_band[5] :  1.9132972572794866e-10
     con_lower_band[6] :  3.7657972518387586e-10
     con_lower_band[7] :  2.1959146109002312e-10
     con_lower_band[8] :   3.226303160655524e-10
     con_lower_band[9] :   2.718723173523049e-10
    con_upper_band[10] :   -1.85519663779765e-10
     con_upper_band[1] : -1.8551966377838212e-10
     con_upper_band[2] :  -1.855196637777094e-10
     con_upper_band[3] : -2.1519354349736048e-10
     con_upper_band[4] :      -9.152992277456027
     con_upper_band[5] :  -6.109883139166232 5e-09
     con_upper_band[6] :   -3.656772464842867e-10
     con_upper_band[7] : -1.1957663055485606e-09
     con_upper_band[8] :  -4.3656448225339187e-10
     con_upper_band[9] :   -5.841283618280628e-10
######################################

The power levels are =  [0.09999999005084921, 0.09999999003308209, 0.7756516278802369, 5.000000049900695, 4.85121004276943, 2.514014613489206, 4.239747822212083, 2.917668537110725, 3.443703830966128, 0.09999999008737301]

The Signal to Interference and Noise Ratios (SINRs) are =  [0.01027978109838697, 0.00822526788759482, 0.06213303402617492, 0.49999998999977896, 0.4999999900002078, 0.4999999900018562, 0.4999999900004585, 0.4999999900012687, 0.49999999000087114, 0.007789650811544387]

[[0.09999999 0.09999999 0.77565163 5.00000005 4.85121004 2.51401461
  4.23974782 2.91766854 3.44370383 0.09999999]
 [0.01027978 0.00822527 0.06213303 0.49999999 0.49999999 0.49999999
  0.49999999 0.49999999 0.49999999 0.00778965]]
```



When S_min is larger,it means that we need higher level of the Signal to Interference and Noise Ratio (SINR). Thus, according to the formula, we need higher level of power levels.

Workshop Assessment Instructions

You should complete the workshop tasks and answer the questions within the allocated session! **Submission deadline is the Friday midnight, the second Week of the workshop.**

It is **mandatory to follow all of the submissions guidelines** given below. *Don't forget the Report submission information on top of this notebook!*

1. The completed Jupyter notebook and its Pdf version (you can simply print-preview and then print as pdf from within your browser) should be uploaded to the right place in Canvas. *It is your responsibility to follow the announcements!* **Late submissions will be penalised (up to 100% of the total mark depending on delay amount)!**
2. Filename should be "ELEN90088 Workshop **W: StudentID1-StudentID2** of session **Day-Time**", where **W** refers to the workshop number, **StudentID1-StudentID2** are your student numbers, **Day-Time** is your session day and time, e.g. *Tue-14*.
3. Answers to questions, simulation results and diagrams should be included in the Jupyter notebook as text, code, plots. *If you don't know latex, you can write formulas/text to a paper by hand, scan it and then include as image within Markdown cells.*
4. Please submit your report individually. Partners can submit the same report.

Workshop Marking

- **Each workshop has 10 points corresponding to 10% of the total subject mark.** You will receive 3 points from the submitted report quality and 7 points from an **indiviudal oral examination.**
- Individual oral quizzes will be scheduled within the next two weeks following the report submission. They will be during workshop hours. Therefore, it is important that you attend the workshops!
- The individual oral examination will assess your answers to workshop questions, what you have done in that workshop, and your knowledge of the subject material in association with the workshop.

Additional guidelines for your programs:

- Write modular code using functions.
- Properly indent your code. But Python forces you do that anyway ;)
- Heavily comment the code to describe your implementation and to show your understanding. No comments, no credit!
- Make the code your own! It is encouraged to find and get inspired by online examples but you should exactly understand, modify as needed, and explain your code via comments. If you resort to blind copy/paste, you will certainly not do well in the individual oral quizzes.

In [ ]: