Imanol Rojas Pérez

TOML

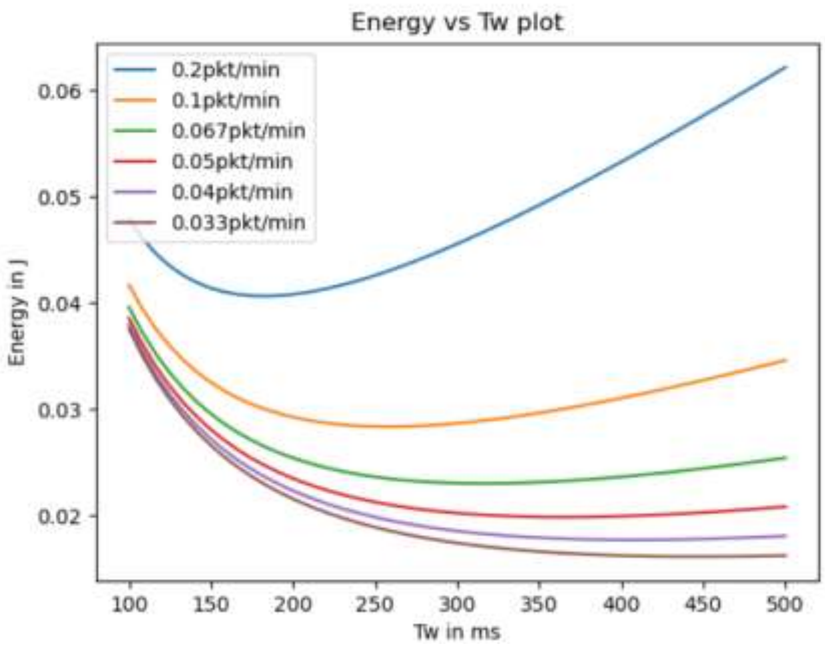
Project 2: Optimization of energy consumption and end to end delay in a wireless sensor network using duty-cycle MAC protocols.

Introduction

In this report sensor network deployment with wireless technology is considered. The objective of the project is to model the network to be able to optimize certain parameters such as the time awake (time that a node is awake and thus, consuming energy), the delay of the communication between nodes and the energy consumption per node. The objective is to minimize the time awake to have a lower energy consumption but at the same time the time awake cannot be too small or the delay in the communication between peers would be too large. A good balance must be found in this trade-off situation. The MAC protocol that is going to be used is already given in the project guide. The MAC used is X-MAC and its equations (also given in the project guide) must be implemented in the python code.

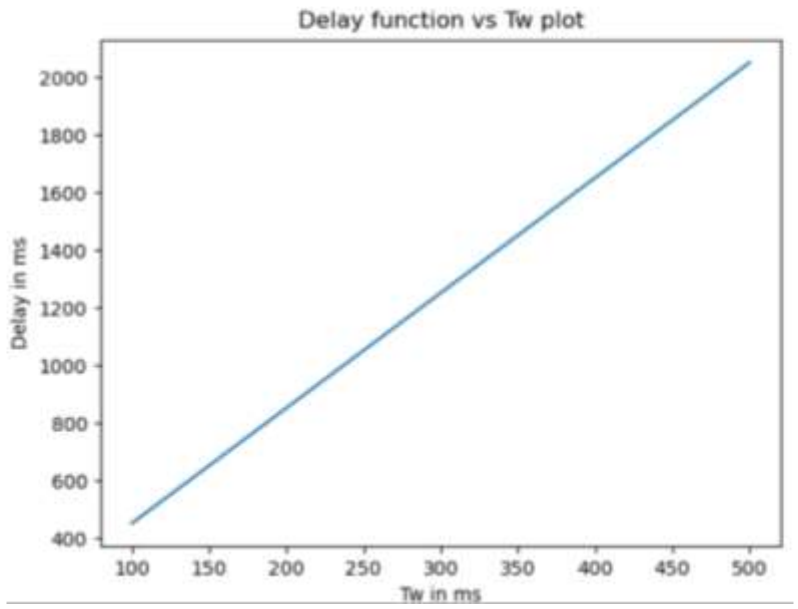
First part: energy and delay vs Tw and energy vs delay plots

In this first part, the delay, energy, and time awake parameters are represented in several graphs to analyze the relationship between them and to find or gist where the optimal solution might be. The next graph represents the Energy consumption of a node with respect to the time that the node is awake for different values of Fs (sampling frequency). The time awake interval used for the plot is 100 ms to 500 ms.



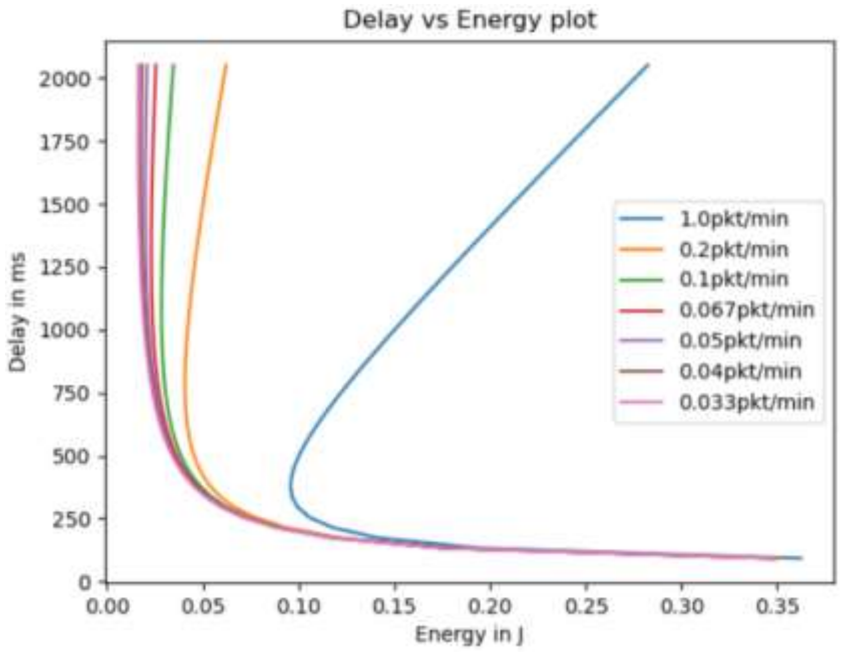
A short period on the sampling frequency will increase the energy consumption for idle listening, in the other hand, with higher periods, the energy consumption is also increased in the wake-up preamble transceiving. Therefore, the optimal sampling time depends on the Tw parameter. When Tw time is too small the energy consumption is high, this also happens when it is too long, so the best point is the region between the 100 ms and the 200 ms which contains the minimums of the energy function.

Next graph represents the Delay function versus the time awake of the node. The interval of Tw used is the same as the last case (100 – 500 ms).



The delay in the communication is proportional to the amount of time that the node is not transmitting or in sleep state (Tw). The longer the nose stays without communicating (without being awake), the longer the delay of the communication between two nodes.

The next plot is the Delay vs Energy plot. The interval of Tw used in this plot is large since otherwise the graph does not show all the shape of the curve.



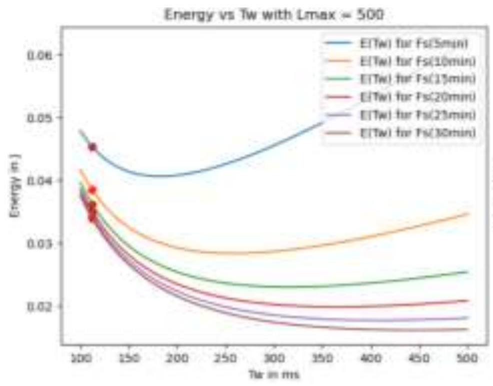
The more delay that the communication has, less energy is consumed (depending on Fs as explained in the first graph). On the other hand, the less delay that the system has, the more energy consumption the system suffers.

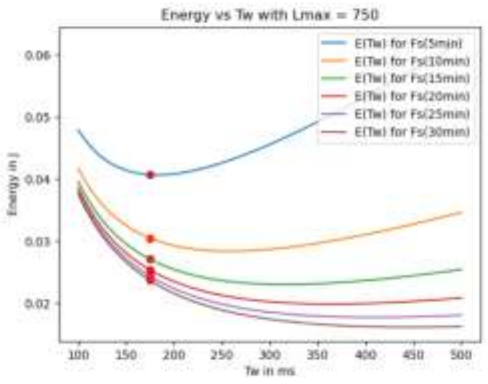
Second part: optimization of the energy consumption and the delay

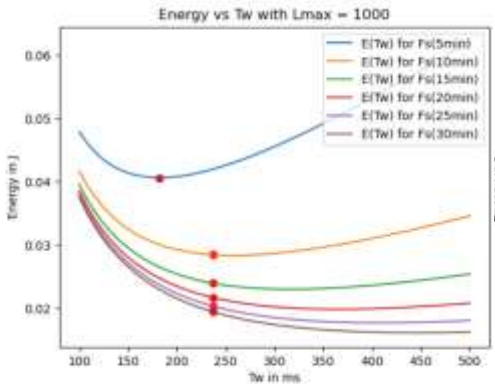
In this second part the optimizations of the delay and the energy functions are carried out. The gpkit library for python has been used to optimize both functions because is a geometric optimization problem. The model of the first problem is the following:

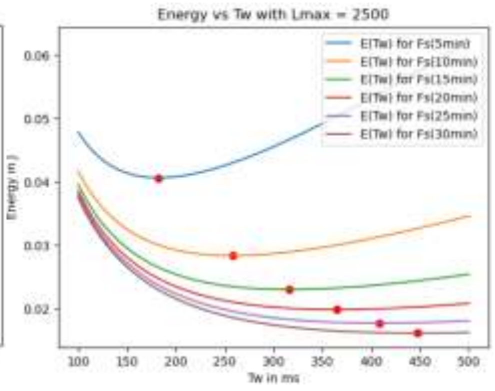
$$\begin{array}{lll} (P1) & Minimize & E^{XMAC}(T_w) \\ & s.t. & L^{XMAC}(T_w) \leqslant L_{max} \\ & & T_w \geqslant T_w^{min} \\ & & |I^0|\, E_{tx}^1 \leqslant 1/4 \\ & Var. & T_w \end{array}$$

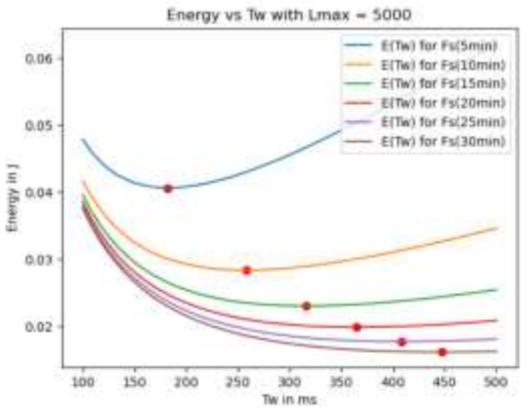
Next graphs correspond to the delay vs time awake curves (for each Fs) that happen given a certain Lmax in the constraints of the model.









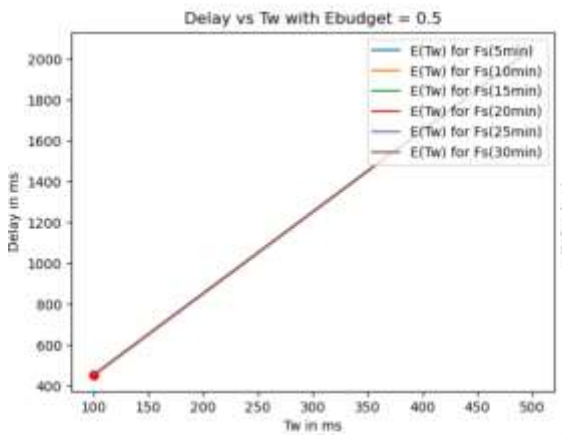


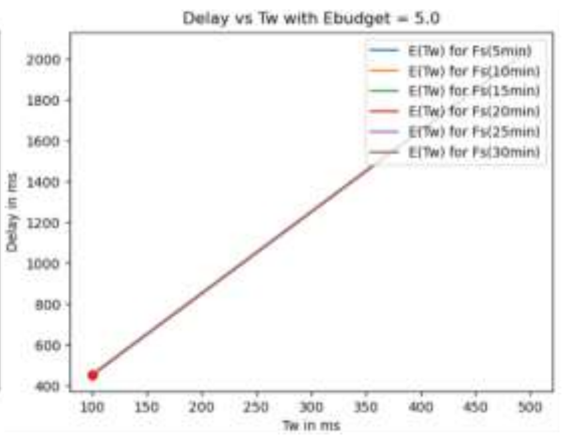
All the solution points are in the lowest point of the curves. This happens because the optimal point is the minimum of the function. From 2000 Lmax all the energy curves are the same so the minimum point will be selected from those plots (all the solutions for the energy will be the same). The lowest point for those graphs is obtained with Fs = 0.033 pkts/min. The optimal energy will be E = 0.016227707747754578.

In the second model, the changes are made in the energy budget. The Ebudgets used are 0.5, 1, 2, 3, 4 and 5. The model implemented in python is the following one.

$$
\begin{array}{lll}
(P2) & Minimize & L^{XMAC}(T_w) \\
 & s.t. & E^{XMAC}(T_w) \leqslant E_{budget} \\
 & & T_w \geqslant T_w^{min} \\
 & & |I^0|\, E_{tx}^1 \leqslant 1/4 \\
 & Var. & T_w
\end{array}
$$

Due that all the Ebudgets have the same graph, only the 0.5 J and the 5 J graphs are shown.





As explained before, the delay is proportional to the Tw that the node has so in this case the graph follows the same path. As seen in the delay graph before, it does not depend on the sampling frequency of the system, all lines are in the same location. In this case though, the solution of the model is plotted as a red dot. The solution for the delay that will be used for the next part (Nash bargaining) is 452.04768314235565 ms. The Tw is around the 100 ms mark.

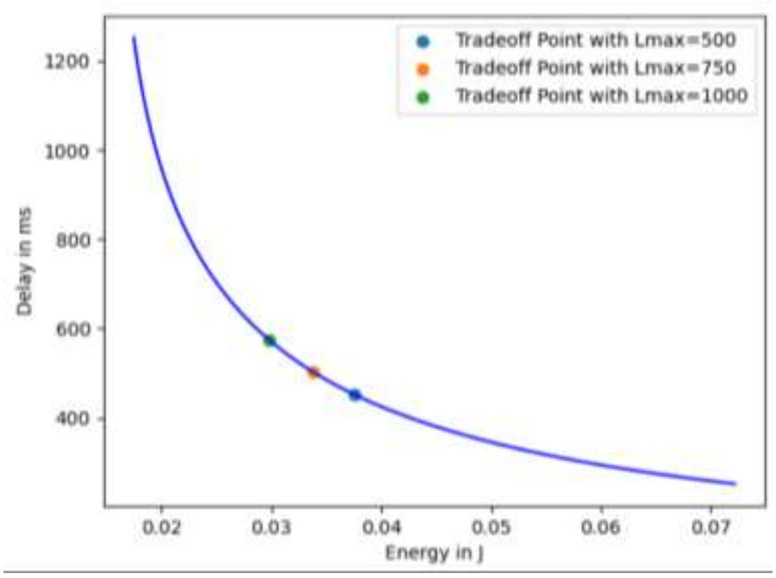
There is only one solution because all the graphs are the same line, and the minimum point is the same for all of them.

Third part: Gaming theory, Nash Bargaining scheme

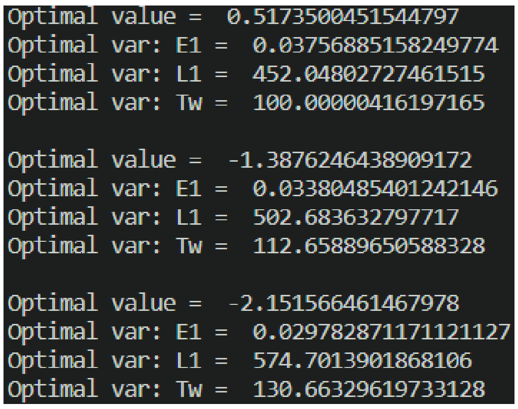
In gaming theory, games with a bargaining scheme are those ones where the players can collaborate to find a better result for both rather than playing selfish and end up getting a worst result. To be a Nash Bargaining scheme, the problem is expressed as follows:

$$\begin{array}{lll} \text{(NBS)} & \max & (E_{\text{worst}} - E(T_w))(L_{\text{worst}} - L(T_w)) \\ & \text{s.t.} & (E_{\text{budget}}, L_{\max}) \geq E(T_w), L(T_w) \\ & & (E_{\text{worst}}, L_{\text{worst}}) \geq E(T_w), L(T_w) \\ & & E(T_w), L(T_w) \in S \\ & \text{var.} & T_w \end{array}$$

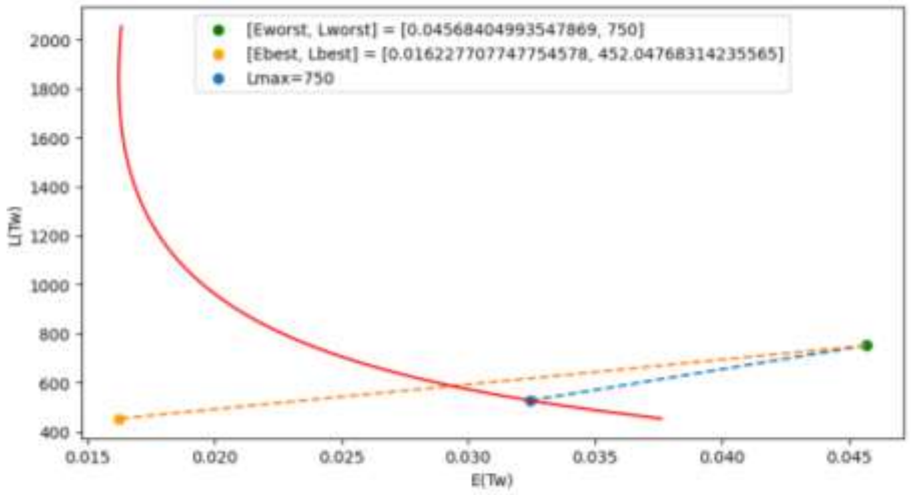
After this model is implemented in python and the solutions vector is calculated for various Lw (500, 750, 1000, 250 and 5000 milisecons) and the same Energy (0.5 J), the solutions are plotted on the Delay – Energy curve and the results between [0, 0.5] J of energy and [0, 800] ms of delay are shown. This graph is the result:



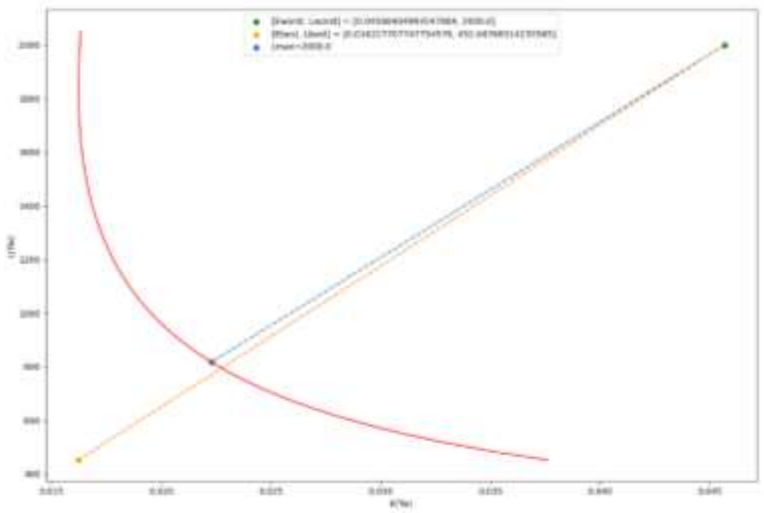
The optimal results in the defined interval fall (more or less) into the middle zone between the delay and the energy. The points have these values:


```
Optimal value =  0.5173500451544797
Optimal var: E1 =  0.03756885158249774
Optimal var: L1 =  452.04802727461515
Optimal var: Tw =  100.00000416197165

Optimal value =  -1.3876246438909172
Optimal var: E1 =  0.03380485401242146
Optimal var: L1 =  502.683632797717
Optimal var: Tw =  112.65889650588328

Optimal value =  -2.151566461467978
Optimal var: E1 =  0.029782871171121127
Optimal var: L1 =  574.7013901868106
Optimal var: Tw =  130.66329619733128
```

The best value would be the point that stays the most centered in the function. This point is defined by the line (straight) line of the worst point to the best point. To check if the points plotted in the last graph are correct, lets see if the line of the result is close to the one described before (the line between the best and the worst results). The next graph is the representation of the line between the worst and the best result and the line between the worst result and the optimal found with a Lmax of 750.



The 750 case has too much energy consumption related to the delay so is not that balanced as it seemed. The next graph is for a Lmax of 2000 as seen previously when the optimal value of the energy was taken.



This case is clearly more centered and has better results. This means that the results that this Lmax gives are better and more optimal than the first ones. The 2000 ms Lmax would be the optimal parameter but better approximations could be done.

ANNEX: Code of the project

```
from variables import *
from gpkit import *
import matplotlib.pyplot as plt

### Sampling frequency
Fs = 0
def fs(time):
    return 1/(time*60*1000)   # e.g. Min traffic rate 1 pkt/half_hour =
1/(60*30*1000) pk/ms

# Sleep period: Parameter Bounds
Tw_max  = 500.       # Maximum Duration of Tw in ms
Tw_min  = 100.       # Minimum Duration of Tw in ms
Tw0 = np.linspace(Tw_min, Tw_max)


# Definition of Id
def Id(d):
    if d == D:
        return 0
    elif d == 0:
        return C
    else:
        return (2*d + 1)/(2*d -1)

# Definition of Fout
def Fout(d):
    if d == D:
        return Fs
    else:
        return Fs*((D**2 - d**2 + 2*d -1)/(2*d - 1))


# Definition of FI
def FI(d):
    if d == 0:
        return Fs*(D**2)*C
    else:
        return Fs*((D**2 - d**2)/2*d - 1)

# Definition of FB
def FB(d):
    return (C - np.abs(Id(d)))*Fout(d)

# Definition of the alphas
def alphas(d):
    alpha1 = Tcs + Tal + (3/2)*Tps*((Tps + Tal)/2 + Tack + Tdata)*FB(d)
    alpha2 = Fout(d)/2
    alpha3 = ((Tps + Tal)/2 + Tcs + Tal + Tack + Tdata)*Fout(d) +
((3/2)*Tps + Tack + Tdata)*FI(d) + (3/4)*Tps*FB(d)
    return alpha1, alpha2, alpha3

# Definition of the betas
def betas(d):
    beta1 = sum([1/2]*D)
    beta2 = sum([Tcw/2 + Tdata]*d)
    return beta1, beta2


########## 1. a) Plot the energy as a function of Tw ##########
def energy(Tw):
    d = 1
    alpha1, alpha2, alpha3 = alphas(d)
    return alpha1/Tw + alpha2*Tw + alpha3

time = [5, 10, 15, 20, 25, 30]

for minutes in time:
    Fs = fs(minutes)
    label = str(round(1/minutes, 3)) + "pkt/min"
    plt.plot(Tw0, energy(Tw0), label = label)
plt.title("Energy vs Tw plot")
plt.xlabel("Tw in ms")
plt.ylabel("Energy in J")
plt.legend()
plt.show()

########## 1. b) Plot delay as a function of Tw ##########
def delay(Tw):
    d = D
    beta1, beta2 = betas(d)
    return beta1*Tw + beta2

plt.plot(Tw0, delay(Tw0))
plt.xlabel("Tw in ms")
plt.ylabel("Delay in ms")
plt.title("Delay function vs Tw plot")
plt.show()

########## 1. c) Plot the curve E-L ##########
Tw0 = np.linspace(0, Tw_max)
for minutes in time:
    Fs = fs(minutes)
    label = str(round(1/minutes, 3)) + "pkt/min"
    plt.plot(energy(Tw0), delay(Tw0), label = label)
plt.title("Delay vs Energy plot")
plt.xlabel("Energy in J")
plt.ylabel("Delay in ms")
plt.legend()
plt.show()

########## 2. OPTIMIZATION AND GRAPH ##########
L = [500, 750, 1000, 2500, 5000]
E = np.linspace(Emax,Emin)
Tw0 = np.linspace(Tw_min, Tw_max)


SolutionsP1 = []
SolutionsP2 = []

time = [5, 10, 15, 20, 25, 30]

for l in L:

    fig = plt.figure()
    ax = fig.add_subplot(1, 1, 1)

    for minutes in time:
        Fs = 1.0 / (minutes * 60 * 1000)

        x = Variable('x')


        alpha1, alpha2, alpha3 = alphas(1)
        beta1, beta2 = betas(D)

        f1 = beta1*x + beta2 # Define all the constraints
        f2 = x
        f3 = abs(Id(0))*(Tcs + Tal + ((x/(Tps + Tal))*((Tps + Tal)/2) +
Tack + Tdata))*Fout(1)

        f01 = alpha1/x + alpha2*x + alpha3 # Objective function P1
        constraintsP1 = [f1 <= l, f2 >= Tw_min, f3 <= 1/4]
        P1 = Model(f01, constraintsP1)
        sol1 = P1.solve()
        SolutionsP1.append(sol1["cost"])
        plt.plot(Tw0, energy(Tw0), label='E(Tw) for
Fs('+str(minutes)+'min)')
        ax.scatter(sol1['variables'][x], sol1['cost'], color="red")

    plt.xlabel('Tw in ms')
    plt.ylabel("Energy in J")
    plt.legend(loc='upper right')
    plt.title("Energy vs Tw with Lmax = "+str(l))
    plt.show()
```

```
E = [0.5, 1.0, 2.0, 3.0, 4.0, 5.0]

for e in E:

    fig = plt.figure()
    ax = fig.add_subplot(1, 1, 1)

    for minutes in time:
        Fs = 1.0 / (minutes * 60 * 1000)

        x = Variable('x')


        alpha1, alpha2, alpha3 = alphas(1)
        beta1, beta2 = betas(D)

        f1 = alpha1/x + alpha2*x + alpha3 # Define all the constraints
        f2 = x
        f3 = abs(Id(0))*(Tcs + Tal + ((x/(Tps + Tal))*((Tps + Tal)/2) +
Tack + Tdata))*Fout(1)

        f02 =  beta1*x + beta2 # Objective function P2
        constraintsP2 = [f1 <= e, f2 >= Tw_min, f3 <= 1/4]
        P2 = Model(f02, constraintsP2)
        sol2 = P2.solve()
        SolutionsP2.append(sol2["cost"])
        plt.plot(Tw0, delay(Tw0), label='E(Tw) for
Fs('+str(minutes)+'min)')
        ax.scatter(sol2['variables'][x], sol2['cost'], color="red")

    plt.xlabel('Tw in ms')
    plt.ylabel("Delay in ms")
    plt.legend(loc='upper right')
    plt.title("Delay vs Tw with Ebudget = "+str(e))
    plt.show()


######### 3. Nash Bargaining Scheme ##########
import cvxpy as cvx

Tw0 = np.linspace(50, 300, 100)

x = cvx.Variable(3, name='x')

fig = plt.figure()
ax = fig.add_subplot(1, 1, 1)
```

```
for item in L:

    Ew = 0.05
    Lw = item

    f0 = - cvx.log(Ew - x[0]) - cvx.log(Lw - x[1])

    f1 = Ew
    f2 = x[0]
    f3 = Lw
    f4 = x[1]
    f5 = x[2]
    f6 = abs(Id(0))*(Tcs + Tal + (x[2]/(Tps + Tal)*(Tps + Tal)/2 + Tack +
Tdata))*Fout(1)

    constraints = [f1 >= (alpha1 * cvx.power(x[2], -1) + alpha2 * x[2] +
alpha3),
                    f2 >=  (alpha1 * cvx.power(x[2], -1) + alpha2 * x[2]
+ alpha3),
                    f3 >= (beta1 * x[2] + beta2),
                    f4 >= (beta1 * x[2] + beta2),
                    f5 >= Tw_min,
                    f6 <= (1 / 4)]

    P3 = cvx.Problem(cvx.Minimize(f0), constraints)


    if x[0].value and x[1].value != None:
        if x[0].value <= 0.06 and x[1].value <= 800:
            ax.scatter(x[0].value, x[1].value, label = 'Tradeoff Point
with Lmax=' + str(item))
            print("Optimal value = ", P3.solve())
            print("Optimal var: E1 = ", x[0].value)
            print("Optimal var: L1 = ", x[1].value)
            print("Optimal var: Tw = ", x[2].value)
            print()

plt.plot(energy(Tw0), delay(Tw0), color='b')
plt.xlabel("Energy in J")
plt.ylabel("Delay in ms")
plt.legend(loc="upper right")
plt.show()
```