Marco Carega - UPC

Project 2 – Optimization of a network of sensors

Introduction

The purpose of this project is to optimize a network of sensors in terms of energy and latency. Precisely we do it in three ways:

- Firstly, we optimize only the energy consume;
- Secondly, we optimize only the latency;
- In the end, we will try to find a balance between energy consumption and latency.

We did these exercises in Python, precisely we used the library gpkit for the first and the second points, instead we used cvxpy to do the last one.

We used gpkit because the first and the second problems are geometric, in fact this library is particularly adapted because it allows us to solve geometric problems in a simple way, so we needn't convert the problem manually into a convex one.

For all the exercises, we uses four frequencies, all of these has its graphs that we analyze in the following sections:

- 1/(60*30*1000)
- 2/(60*30*1000)
- 5/(60*30*1000)
- 10/(60*30*1000)

Classes

To simplify the work, we developed some classes allowing us to compute some values useful for the optimization, precisely we implemented:

- An “abstract” class Optimizer, containing common parts;
- A class EnergyOptimizer, useful to solve the first point;
- A class LatencyOptimizer in order to get results from the second point.

Class Optimizer

The class Optimizer contains the following methods:

- A constructor that keeps all the parameters it needs, from the payload size to the times;
- numNodes, returning the number of nodes in base of the number of neighbors and rings;
- outFrequency that calculates the Fout at the ring d;
- inFrequency that computes the Fin at the ring d;
- numBias node compute the number of bias node at the ring d;
- numInput computes the input nodes at the ring d;
- calcAlpha calculate the alpha values useful for the energy function;
- calcBeta compute the beta values used for the latency function;
- getCoefficients computes alphas and betas for the worst case;
- bottleneck is the bottleneck constraint;
- calcEnergy computes the energy used in the transmission at ring d.
- the solve method allows us to create the optimization problem and solve it returning the results.

Moreover, this class contains two abstract method that will be implemented in the subclasses (we’ll talk about these later):

- makeFzero returns the function we have to optimize;
- makeConstraints returns a list of inequalities that will be useful in the problem resolution.

Computing of Fout and Fin

The computing of the frequencies Fout and Fin of a sensor depends on the ring where it is ubicated. For the output frequency, the compute depends on the number of nodes in input and the sampling frequency, so we can distinct two cases:

- in the last node (the ring D), there are no input nodes, so the Fout is equals to Fs;
- In the general case, we need the number of inputs and the Fs to find Fout.

From these assumptions, we can conclude that Fout is higher when the node is closer to the gateway, this happens because the closer nodes have to route the packets they receive from the outer rings to the gateway, so the first ones produce more packets.

Regarding the input frequency, what can we say about it? It depends on the Fout of the closer node, so we can expect a similar behavior

```
def outFrequency(self,Fs,d):
    D=self.numRings
    if(d==D):
        return Fs
    else:
        return ((D**2-d**2+2*d-1)/(2*d-1))*Fs

def inFrequency(self,Fs,d):
    D=self.numRings
    C=self.numNeighbors
    if(d==0):
        return (C*D**2)*Fs
    else:
        return ((D**2-d**2)/(2*d-1))*Fs
```

Coefficients alphas and betas

The coefficients alphas and betas indicate constant values in function of output, input and bias frequencies, we used them to compute respectively the energy and the latency. However these frequencies are different for each level, so what can we do for a better estimation? We consider the worst case:

- for the energy, the largest amount of consumption is in the closest node to the gateway because they must routes all the packets of the previous levels directly to it, so we calculate the alphas at ring 1;
- for the latency instead, the reasoning is the opposite, in fact the further nodes from the gateway are slower because the packets must go in each level. So, we can conclude that the worst case is in the latest ring and, for this reason, we compute the betas on it.

```
def calcAlpha(self,d):
    Tcs=self.time.cs
    Tal=self.time.al
    Tps=self.time.ps
    Tack=self.time.ack
    Tdata=self.time.data
    B=self.numBiasNodes(d)
    Fout=self.outFrequency(self.samplingFreq,d)
    Fin=self.inFrequency(self.samplingFreq,d)
    Fb=B*Fout
    a1=Tcs+Tal+(3/2)*Tps*((Tps+Tal)/2+Tack+Tdata)*Fb
    a3=((Tps+Tal)/2+Tcs+Tal+Tack+Tdata)*Fout+((3/2)*Tps+Tack+Tdata)*Fin+(3/4)*Tps*Fb
    a2=Fout/2
    return [a1,a2,a3]

def calcBeta(self,d):
    Tcw=self.time.cw
    Tdata=self.time.data
    b1=d/2
    b2=(Tcw/2+Tdata)*d
    return [b1,b2]

def getCoefficients(self):
    a=self.calcAlpha(1)
    b=self.calcBeta(self.numRings)
    return a,b
```

The bottleneck constraint

The bottleneck constraint is a particular inequality where, in order to do a successful communication, the energy used in transmission by the input nodes of the gateway (that is at ring 0) must be less or equal 1/4.

```
def bottleneck(self):
    I=self.numInputs(0)
    return GeoIneq(lambda Tw: I*self.calcEnergy(Tw,1),lambda x:1/4)

def calcEnergy(self,Tw,d):
    Tack=self.time.ack
    Tdata=self.time.data
    Tcs=self.time.cs
    Tal=self.time.al
    numInput=self.numInputs(d-1)
    Fout=self.outFrequency(self.samplingFreq,d)
    Ttx=Tack+Tdata+Tw/2
    return (Tcs+Tal+Ttx)*Fout
```

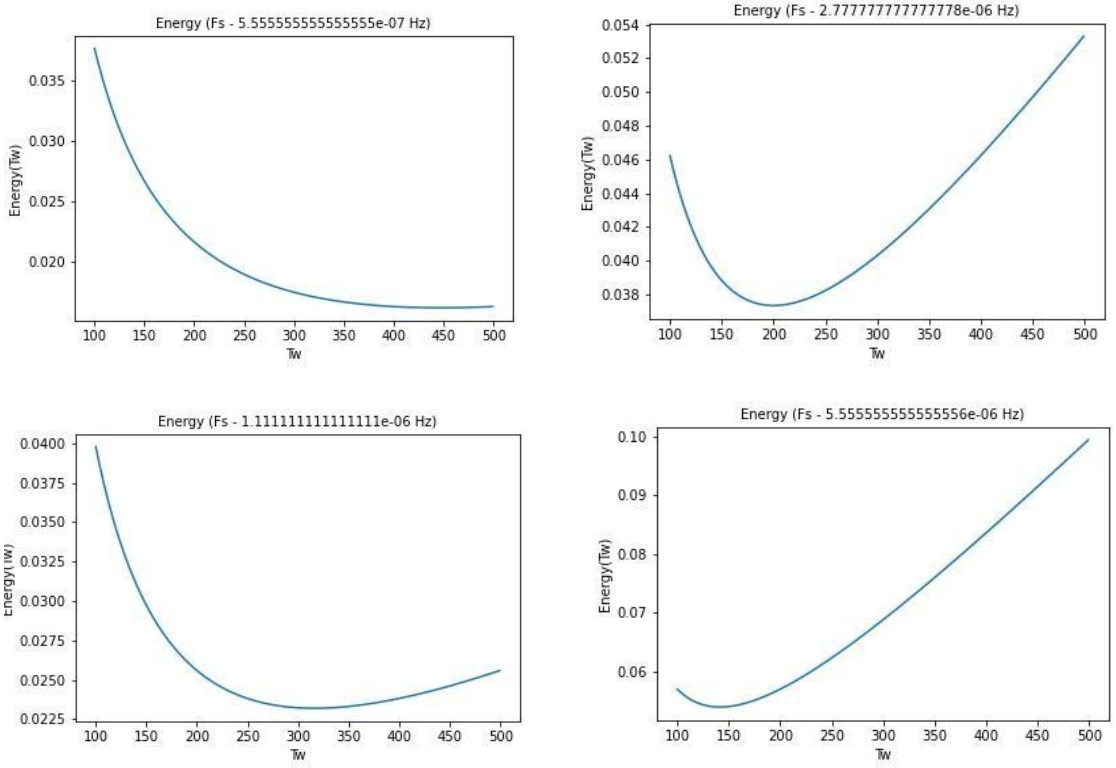
Optimizing energy

For energy optimization, we use the class EnergyOptimizer that extends the class Optimizer. Precisely we implement the methods “makeConstraints” and “makeFzero”:

- In the first one we put some geometric inequalities, one for each constraint in the problem. So, Tw must be between the range (Twmin, Twmax) and the latency at time Tw must be less than its maximum value. Last but not least, there is the bottleneck constraint we explained previously.
- The second method returns the function we have to optimize. To do it we use the alpha coefficients and, as we can see in the code of “makeFzero”, the x is the denominator in the first addend, so we can conclude it is hyperbolic.

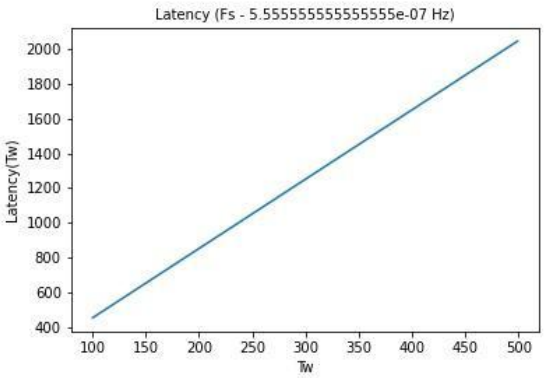
```
def makeFzero(self):
    a,b=self.getCoefficients()
    return lambda Tw:(a[0]/Tw)+a[1]*Tw+a[2]

def makeConstraints(self):
    bot=self.bottleneck()
    a,b=self.getCoefficients()
    ineq1=GeoIneq(lambda Tw:self.time.wmin,lambda Tw:Tw)
    ineq0=GeoIneq(lambda Tw:Tw,lambda Tw: self.time.wmax)
    ineq2=GeoIneq(lambda Tw:b[0]*Tw+b[1],lambda Tw:self.latency.max)
    return [ineq1,ineq2,ineq0,bot]
```

The graphs



These are graphs for each sampling frequency, in some of them it is clearly visible the hyperbolic evolution of the function. From these graphs we can make these conclusions:

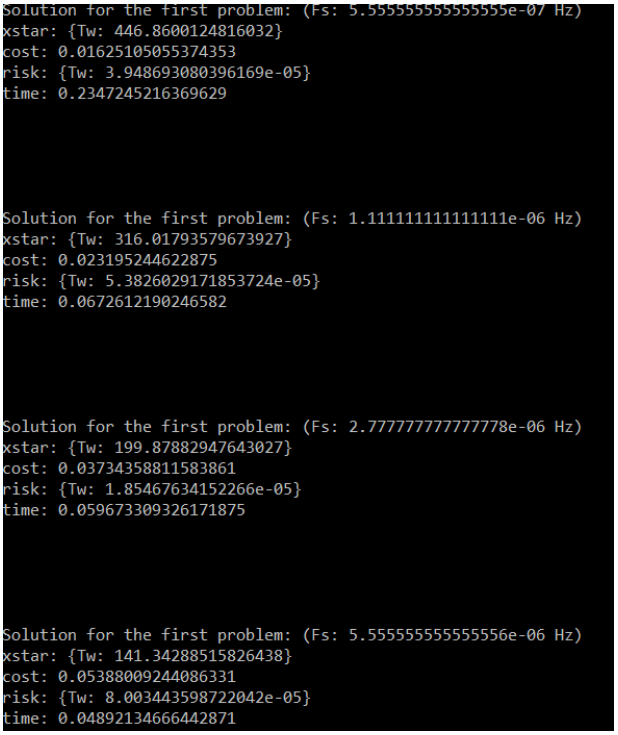
- In the first graph, the best value is between 450 and 500 ms, in this interval the function returns similar values and so the solver may give a solution in these range;
- Considering the second graph, it is clearly visible that the optimal point is near 200 ms;
- In the third graph, the best value is between 300 and 350 ms, probably near the first value;
- At the end the optimal value is clearly close to 150 in the last graph.



What can we say about the latency?

- First of all, we can notice that all the graphs are equals , so we decided to make only one plot;
- Using the energy optimal point, it is clearly visible that where we have the best energy, the latency is bad. The only points where energy and latency are more or less the best at the same time are in the second and in the last graph, respectively at 200 and 150 ms.

Results



As we can see in the photo, if we compare them with the respective graph, we see that the optimal value is more or less where we said. We can also see there are riskiest values of Tw, everyone is close to 0, so we can conclude that in these values the energy function doesn't compute the value for that specific Tw. In fact it is a hyperbolic function that goes to infinity where the independent variable belongs to zero.

Optimizing latency

To optimize the latency we used the class LatencyOptimizer, which as EnergyOptimizer does, extends Optimizer and implements the methods "makeFzero" and "makeConstraints".

Considering the method "makeFzero", the method computes the beta coefficients and uses it to return a function. As we can see in the code, the returned function is linear.

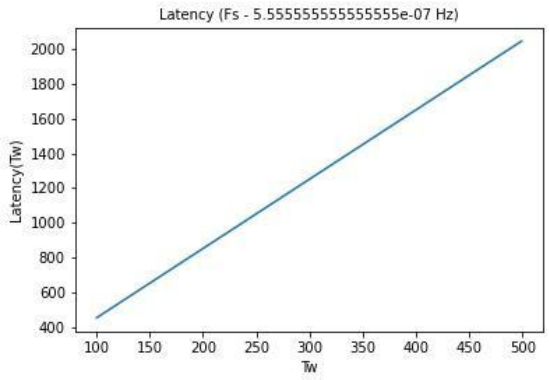
```
def makeFzero(self):
    a,b=self.getCoefficients()
    return lambda Tw:b[0]*Tw+b[1]

def makeConstraints(self):
    a,b=self.getCoefficients()
    bot=self.bottleneck()
    ineq1=GeoIneq(lambda Tw:self.time.wmin,lambda Tw:Tw)
    ineq2=GeoIneq(lambda Tw:a[0]/Tw+a[1]*Tw+a[2],lambda Tw:self.maxEnergy)
    ineq3=GeoIneq(lambda Tw:Tw,lambda Tw:self.time.wmax)
    return [ineq1,ineq2,ineq3,bot]
```

Regarding "makeConstraints", what can we say about it? The constraints are practically the same as the previous exercise, with one difference: instead of using a constraint on the latency, we use a constraint on

the energy. This constraint implements the energy function and limits it to the budget energy, that is the maximum value we can get.

The graph

As we said previously, the latency function is linear and its is clearly visible in the graph below. From this graph, we can conclude the best latency value is at 100 ms. This conclusion is right for all the sampling frequencies we used because the graphs are equals.



Results

Comparing the results we got solving the problem, we can see that all the values of Tw are close to our conclusion and the differences are in terms of decimal values.

```
Solution for the second problem: (Fs: 5.555555555555555e-07 Hz)
xstar: {Tw: 99.99999675692185}
cost: 452.0479937785625
risk: {Tw: 1.769755579658893}
time: 0.0392301082611084



Solution for the second problem: (Fs: 1.111111111111111e-06 Hz)
xstar: {Tw: 99.99999952194183}
cost: 452.04799928857716
risk: {Tw: 1.769734492717332}
time: 0.047423601150512695



Solution for the second problem: (Fs: 2.777777777777778e-06 Hz)
xstar: {Tw: 99.99999947598002}
cost: 452.0479987003181
risk: {Tw: 1.769734842883828}
time: 0.042342185974121094



Solution for the second problem: (Fs: 5.555555555555556e-06 Hz)
xstar: {Tw: 99.99999956373418}
cost: 452.0479991135299
risk: {Tw: 1.7697360479360653}
time: 0.04546713829040527
```

Balancing between energy and latency

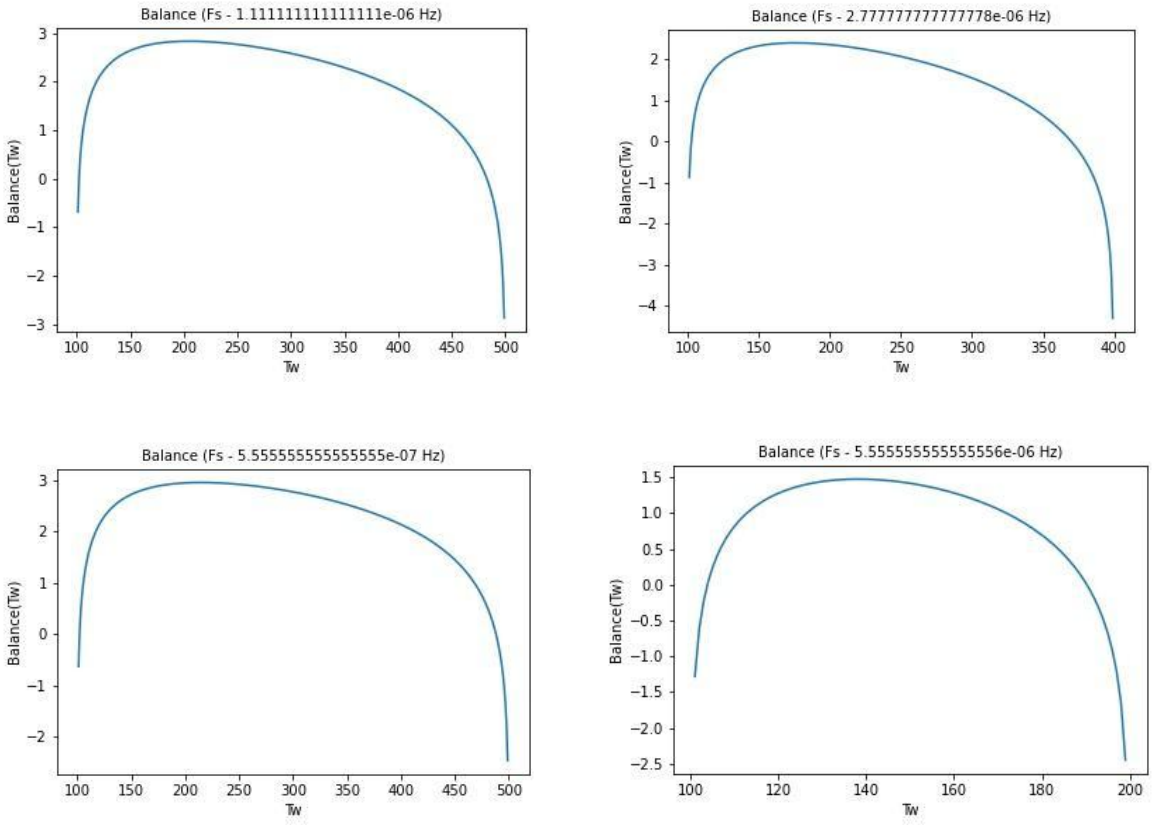
After computing the best energy and the best latency, is it possible to find a balance in these terms? Yes and it is the third exercise. What can we do to find a balance? We use the game theory, in fact considering

energy and latency as agents, each one competes trying its best to maximize its value and minimize the opponent one. Based on this assumption, we model the problem.

$$
\begin{aligned}
\text{(NBS-XMAC*)} \quad \max \quad & log(E^{\text{XMAC}}_{\text{worst}} - E_1) + log(L^{\text{XMAC}}_{\text{worst}} - L_1) \\
\text{s.t.} \quad & E^{\text{XMAC}}_{\text{worst}} \geqslant E^{\text{XMAC}}(T_w) \\
& E_1 \geqslant E^{\text{XMAC}}(T_w) \\
& L^{\text{XMAC}}_{\text{worst}} \geqslant L^{\text{XMAC}}(T_w) \\
& L_1 \geqslant L^{\text{XMAC}}(T_w) \\
& T_w \geqslant T_w^{\min} \\
& |I^0| \, E^1_{tx} \leqslant 1/4 \\
\text{var.} \quad & E_1, L_1, T_w
\end{aligned}
$$

In the problem Eworst and Lworst are the energy and the latency in the worst case, that are respectively the minimum and the maximum Tw. E1 and L1 are auxiliary variables that are useful to make the problem convex.

The graphs



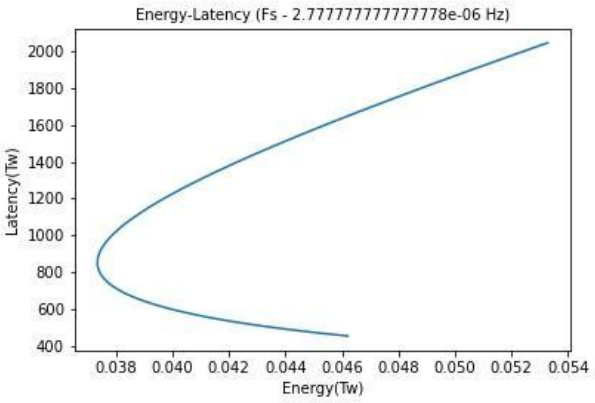
In the graph we can see the function is clearly convex and we have to maximize, even though we are considering only the variable Tw. In the first three graphs, the optimal point is between 200 ans 250, instead in the last one it is near 140.

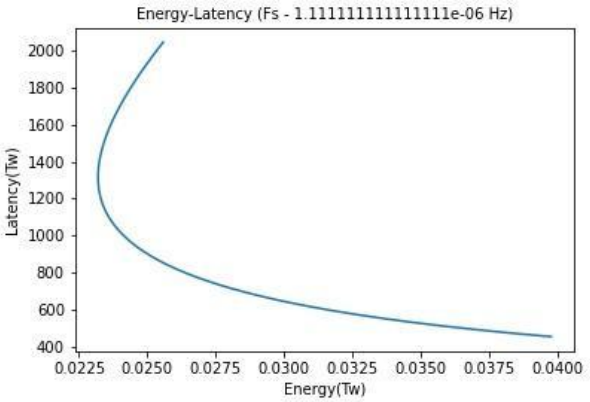
Results

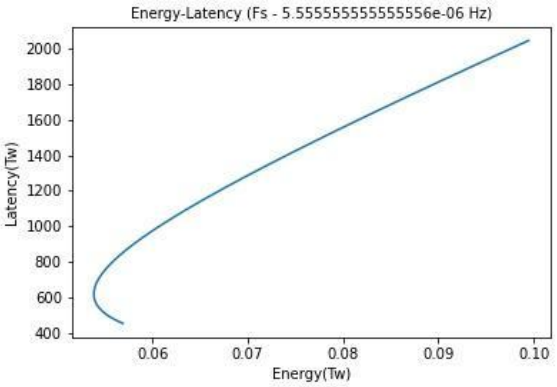
```
Solution of the third problem (Fs: 5.555555555555555e-07 Hz)
xstar: [array(214.62846366), array(0.02071892), array(910.56185465)]
pstar: 2.961012719586826
lambdas: [0.0, 7.125089720875672e-07, 59.09401096661376, -7.886452688602565e-12, 0.0008759406483821151, -1.755746945706149e-12, -4.427366132502662e-11]
numIt: None



Solution of the third problem (Fs: 1.111111111111111e-06 Hz)
xstar: [array(204.94831297), array(0.02533476), array(871.84125187)]
pstar: 2.836338225313332
lambdas: [0.0, 3.4836235339144737e-07, 69.20783598098402, -1.6755028042191356e-12, 0.0008473001312562795, -1.2336996290791479e-13, -1.1090464806697826e-11]
numIt: None



Solution of the third problem (Fs: 2.777777777777778e-06 Hz)
xstar: [array(175.50688405), array(0.03764433), array(754.07553619)]
pstar: 2.4085956808866884
lambdas: [0.0, 3.2604434991858336e-06, 116.7434635506139, -8.978459398659627e-12, 0.0007704203451895504, 1.0138426969413689e-10, -6.319491081155089e-11]
numIt: None



Solution of the third problem (Fs: 5.555555555555556e-06 Hz)
xstar: [array(138.14494033), array(0.05389335), array(604.62776133)]
pstar: 1.4774078348347448
lambdas: [-2.22761563458226e-09, 3.58823790024872 4e-06, 330.34242694485965, -5.396624830225967e-12, 0.0006908959449733037, 1.1600572965608278e-10, 0.0]
numIt: None
```

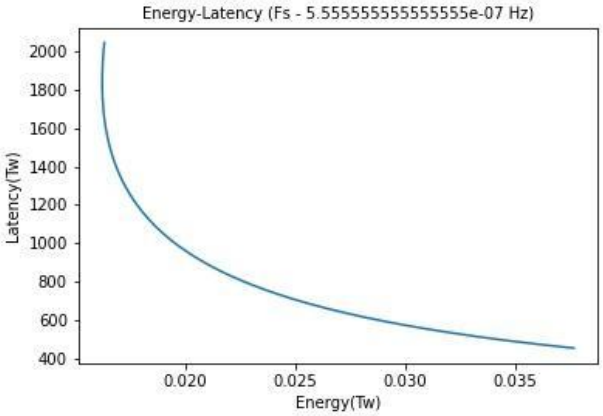
Considering the results and the graphs in the previous section, we can see that the results confirms our conclusions, the only one that is a little different is the third, maybe it is the gradient satisfy the exit condition before 200, in fact in the graph the values between 150 and 200 are very similar.

Behavior of the curves E-L



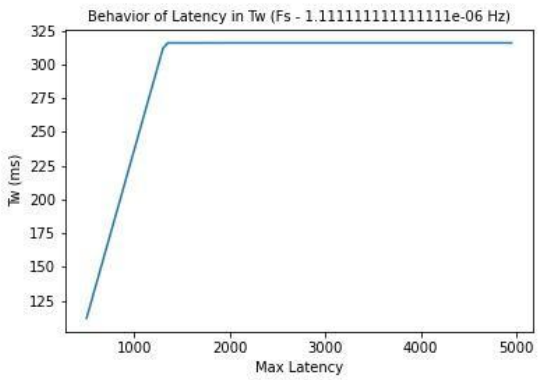


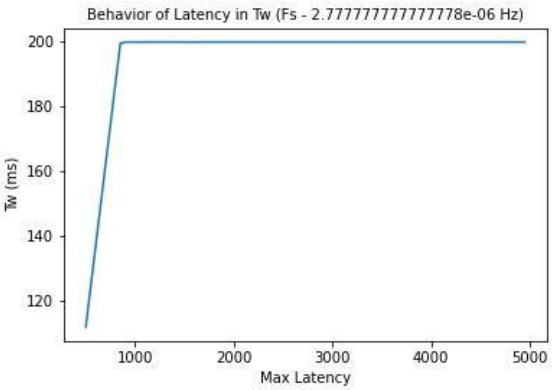


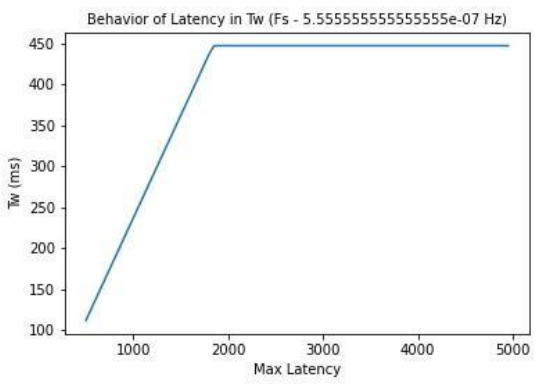


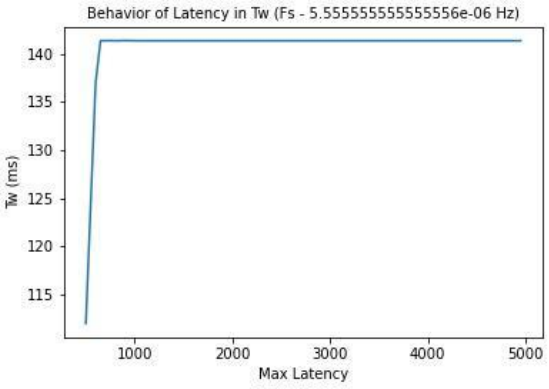
Considering the plots above, we can see the correlation between latency and energy, what can we say about them? We can see that the optimal points of the third problem are clearly visible, they are in the points of the line where the function converges. In the first plot for example, it is more or less at (0.03, 850).

Behavior of maximum latency and budget energy





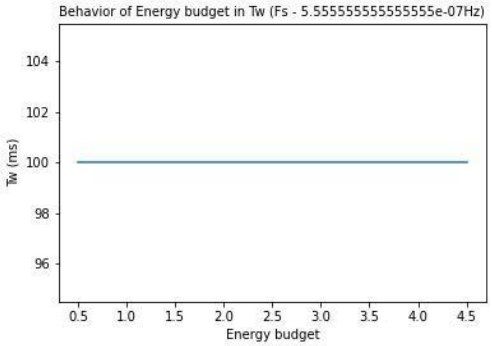




Now we analyze the behavior of the budget energy and the maximum latency, to do this we also consider the optimal Tws we found with certain values of latency/energy.

Considering the latency plots, we can see that the function begins linearly and becomes constant after a certain point, this trend becomes fast when we increase the sampling frequency. To do these experiments, we used latency values that are between 500 and 5000 because lower values than the first one results unfeasible for the problem.

Now we consider the energy plot below. We can see that the function is constant, so we can conclude that the amount of energy does not influence the optimal point as the latency does.



Notes

The code we used for this project is in this repository:

https://github.com/MarcoCarry97/Topic-On-Optimization-And-Machine-Learning.git