Optimization Algorithms: Theory and Software Implementation

Prof. Thirumulanathan D

Department of Mathematics

Institute of IIT Kanpur

Lecture: 39

Hello everyone, this is Lecture Four in Week Eight. In the previous three lectures, we learned about solving constrained optimization problems with equality constraints or inequality constraints using Augmented Lagrangian Methods.

We will now start with a practical example that we worked on long back in the very first week of this course. That is the consumer utility maximization problem.

If you recall the problem, it is in two dimensions. The more general problem is the one we have here. We maximize some utility of the consumer, which is a function of x_1, x_2, ..., x_n. To recall what these x_1, x_2, ..., x_n are: the consumer has n items to buy, and these are the quantities that he buys.

He buys a quantity x_1 of item 1, x_2 of item 2, ..., x_n of item n. The price of each of these items is given by p_1, p_2, ..., p_n.

This is the budget constraint. w is his monthly income, or wealth. So, he cannot spend more than his wealth: $\sum_{i=1}^{n} p_i x_i \leq w$.

And of course, x_i must be non-negative; he cannot buy negative quantities of any items.This is the more general problem of consumer utility maximization.

We actually considered the utility to be the product of the quantities that he buys. x_1 is the quantity of item 1, x_2 is the quantity of item 2, and so on.

We solved a problem analytically where he buys only two items, with quantities x_1 and x_2.

We will now consider an n-item setting, and instead of the product, we will consider a slightly different utility. The utility we consider is the following:

We minimize $-\log x_1 - \log x_2 - ... - \log x_n$.

We consider the general n-item problem, but the utility that he is maximizing is the logarithm of the quantity of the first item plus the logarithm of the quantity of the second item plus ... plus the logarithm of the quantity of the n^{th} item that he is going to buy.

Since we are considering a minimization problem, we consider the negative of that.

So, that is: minimize $-\log x_1 - \log x_2 - ... - \log x_n$.

Of course, the constraints are the same:

You have $\sum_{i=1}^{n} p_i x_i \leq w$, and $x_i \geq 0$.

I will write that as $-x_i \leq 0$ for all i = 1 to n. So, there are n + 1 constraints, all inequality constraints. This is the quantity that he is going to minimize.

Note that the problem I consider here and the problem considered earlier are quite the same because when you are maximizing $x_1 x_2$, it is equivalent to maximizing a monotonic function of $x_1 x_2$.

Log is actually a monotonic function in x. So, I have just considered the log of the product of x_i's, which is the sum of the log of x_i's.

Both problems are the same; I am considering it this way because this is a convex optimization problem, for no other reason.

Let us actually work on solving this particular problem. Maybe before solving it using Python, we will solve it by hand. It is not very difficult.

To solve it by hand, I would like you to observe something. Note that this constraint, $\sum_{i=1}^{n} p_i x_i \leq w$, is an increasing function of x.

And you can also see that $\log x_1 + \log x_2 + ... + \log x_n$ is also an increasing function of x.

The solution is never going to lie at a point where $\sum_{i=1}^{n} p_i x_i$ is strictly less than w.

That is because if you increase the values of x_i, you will still satisfy the constraint $\sum_{i=1}^{n} p_i x_i \leq w$, and you are actually increasing the utility $\log x_1 + \log x_2 + ... + \log x_n$.

I am doing this to simplify the problem; analytically, it will be a little simpler if I do this.

So, the claim is that this constraint can be simplified to $\sum_{i=1}^{n} p_i x_i = w$.

That is because if the optimal solution of this particular problem occurred when $\sum_{i=1}^{n} p_i x_i < w$, then by increasing x_i a little bit, I can increase the utility function $\log x_1 + \log x_2 + ... + \log x_n$.

So, that is certainly not the solution. The solution is going to occur on the plane $\sum_{i=1}^{n} p_i x_i = w$.

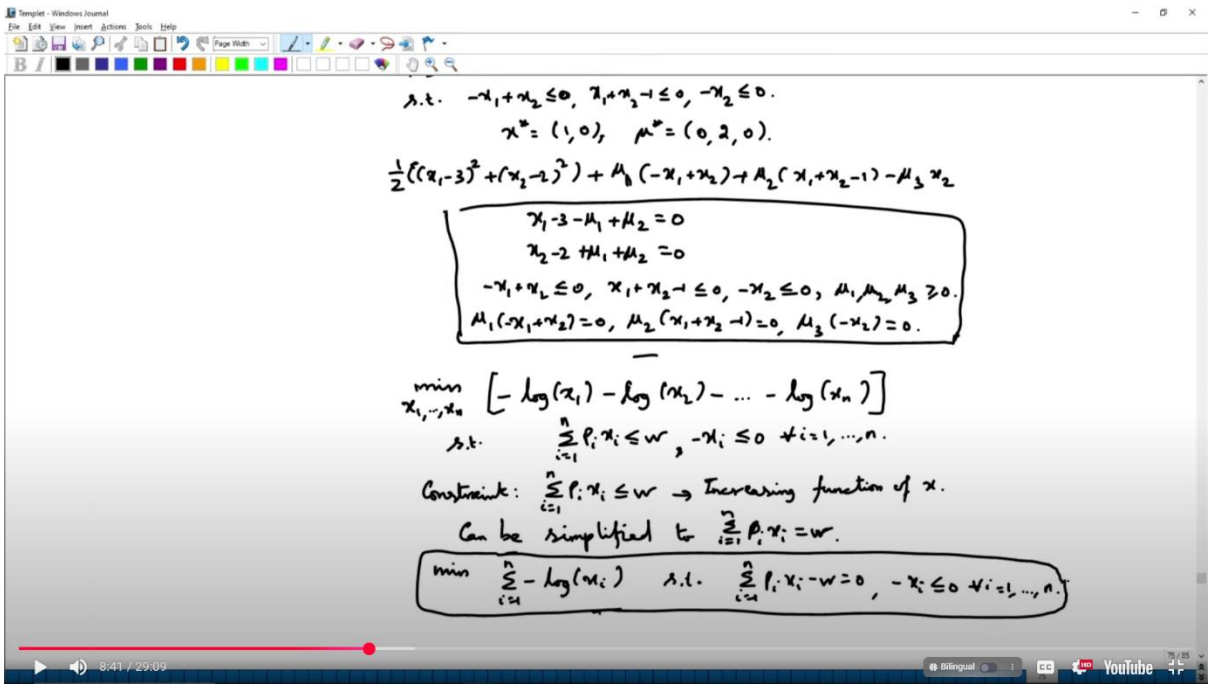
So, we can now rewrite this problem as:

Minimize $\sum_{i=1}^{n} -\log x_i$

Subject to $\sum_{i=1}^{n} p_i x_i - w = 0$

And $-x_i \leq 0$, for i= 1, 2, …n. This is a more simplified problem.

(Refer Slide Time 8:41)



Now, we can write down the Lagrangian since we are solving it analytically. The Lagrangian will be:

$L=-\sum_{i=1}^{n} \log x_i + \lambda(\sum_{i=1}^{n} p_i x_i - w) - \sum_{i=1}^{n} \mu_i x_i$

If we differentiate (take the gradient of this with respect to x), you will have:

For each i: $-1/x_i + \lambda p_i - \mu_i = 0$

If we assume all μ_i's to be 0 for the time being, then we will have for all i = 1 to n:

$-1/x_i + \lambda p_i = 0 \Rightarrow 1/x_i = \lambda p_i \Rightarrow \lambda = 1/(p_i x_i)$ for all i.

What this tells us is that $p_i x_i$ is actually a constant over i. Since $\sum_{i=1}^{n} p_i x_i = w$, this implies that $p_i x_i = w/n$ for each i. That implies that $x_i^* = w/(n\, p_i)$.

We will also get $\lambda^* = 1/(p_i x_i) = 1/(w/n) = n/w$. And of course, we have already assumed that μ^* is 0. We want the complementary slackness conditions to need $\mu_i x_i = 0$.

x_i is not 0, so μ_i's must be 0. So, this is the answer:

$x_i^* = w/(n\, p_i)$

$\lambda^* = n/w$

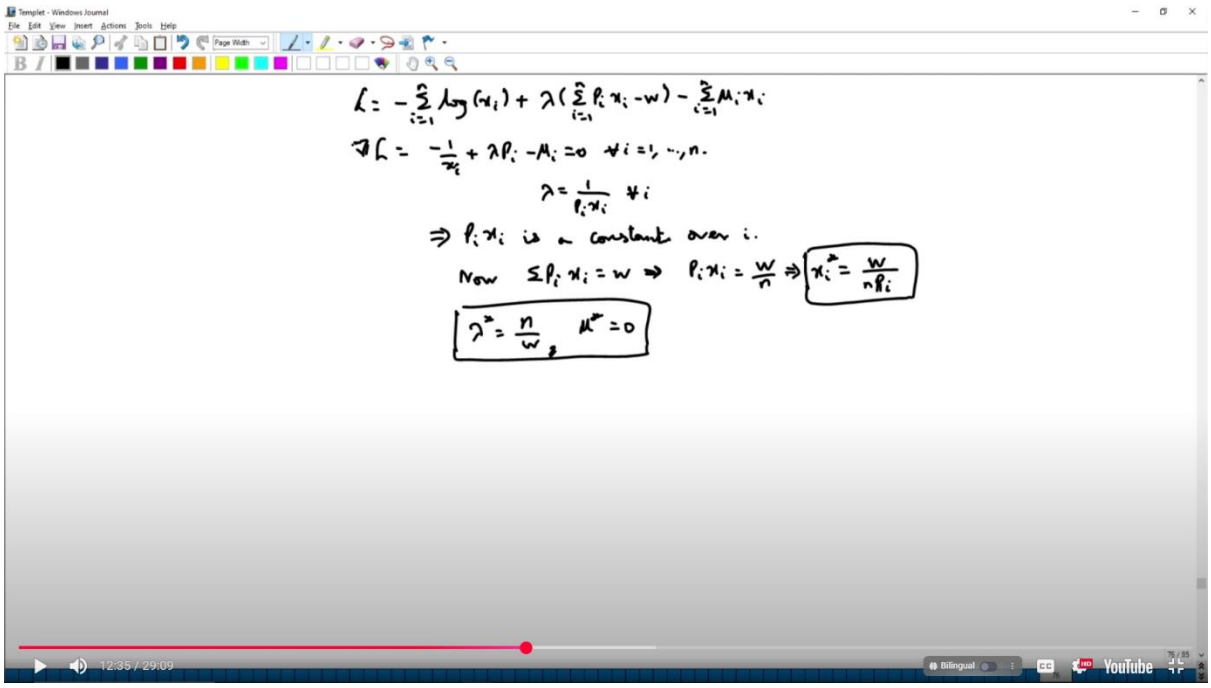
$\mu^* = 0$

Just to check if we got the same answer here: in the two-item problem, you had $x_1^* = w/(2\, p_1)$ and $x_2^* = w/(2\, p_2)$. Here we have $w/(n\, p_i)$, so when n = 2, you have $w/(2\, p_1)$ and $w/(2\, p_2)$.

Of course, λ^* is different because the problem we considered is slightly different.

For this logarithm problem, we have $\lambda^* = n/w$.

(Refer Slide Time 12:35)



We will now try to write a code using the Augmented Lagrangian Method.

The difference between the earlier examples and this one is that we are solving for any general n.

You can just modify certain parameters: modify the value of n, modify the value of p_i's, make some very small modifications, and the code you are going to write will work for every n and every set of p_i's that you give.

Since the base structure is going to be the same, we will just copy down the previous code and then make modifications.

We will write down the title as "Augmented Lagrangian method for utility maximization".

Now, f(x) is the negative of the sum of log of x_i's. We will write it as `-np.sum(np.log(x))`.

x will be a vector. `np.log(x)` will be a vector, and we sum those elements and put a negative sign in front. So, that is -log x_1 - log x_2 - ... - log x_n.

g(x) is slightly tricky since we have n + 1 constraints.

The last n constraints are just $-x_1, -x_2, \ldots, -x_n$.

The first constraint is $\sum_{i=1}^{n} p_i x_i - w$.

All are less than or equal to 0.

I am going to write it this way: the size of x is `np.shape(x)[0]`.

We need to insert the vector $[p_1, p_2, ..., p_n]$ for the first constraint.

We will use the `insert` command to put the budget constraint at the beginning of the vector - x.

The gradient: for g(x), we have the first constraint derivative as p, and the next n constraints derivatives are -I (negative identity).

So, we will construct the gradient matrix accordingly. For the gradient of f(x), it is -1/x.

We will write that. For the Hessian of f(x), it is a diagonal matrix with elements $1/x_i^2$.

We will write `np.diag(1/x**2)`. We will define p and w.

Let us consider a 4-dimensional problem: a consumer buying 4 items. The price of item 1 is 4, item 2 is 3, item 3 is 2, item 4 is 1.

So, p = [4, 3, 2, 1].

Let the wealth w be 120.

Since it is a 4-dimensional problem, x has to have 4 values. It is not good to start with 0 because log(0) is not defined, so we will start with 1. The number of constraints is 4 + 1 = 5, so μ will be a vector of 5 zeros. γ is 1, and k starts at 0.

For this specific case, the analytical solution is:

$x_1^* = 120/(4*4) = 120/16 = 7.5$

$x_2^* = 120/(4*3) = 120/12 = 10$

$x_3^* = 120/(4*2) = 120/8 = 15$

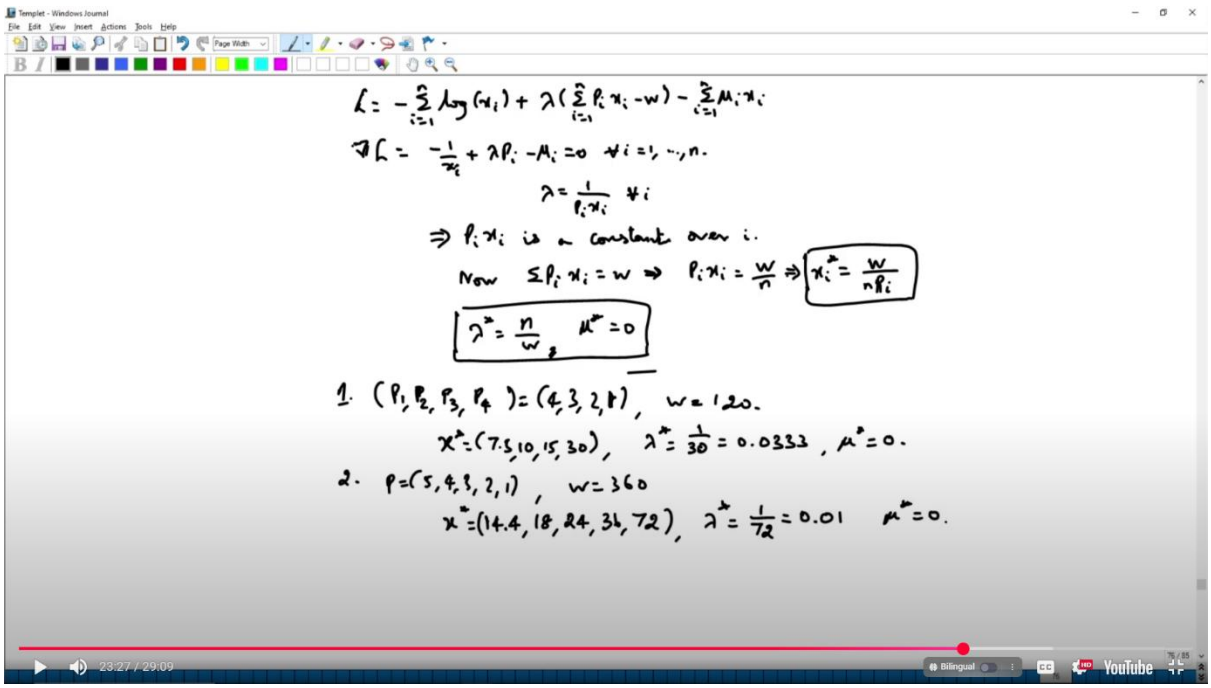
$x_4^* = 120/(4*1) = 120/4 = 30$

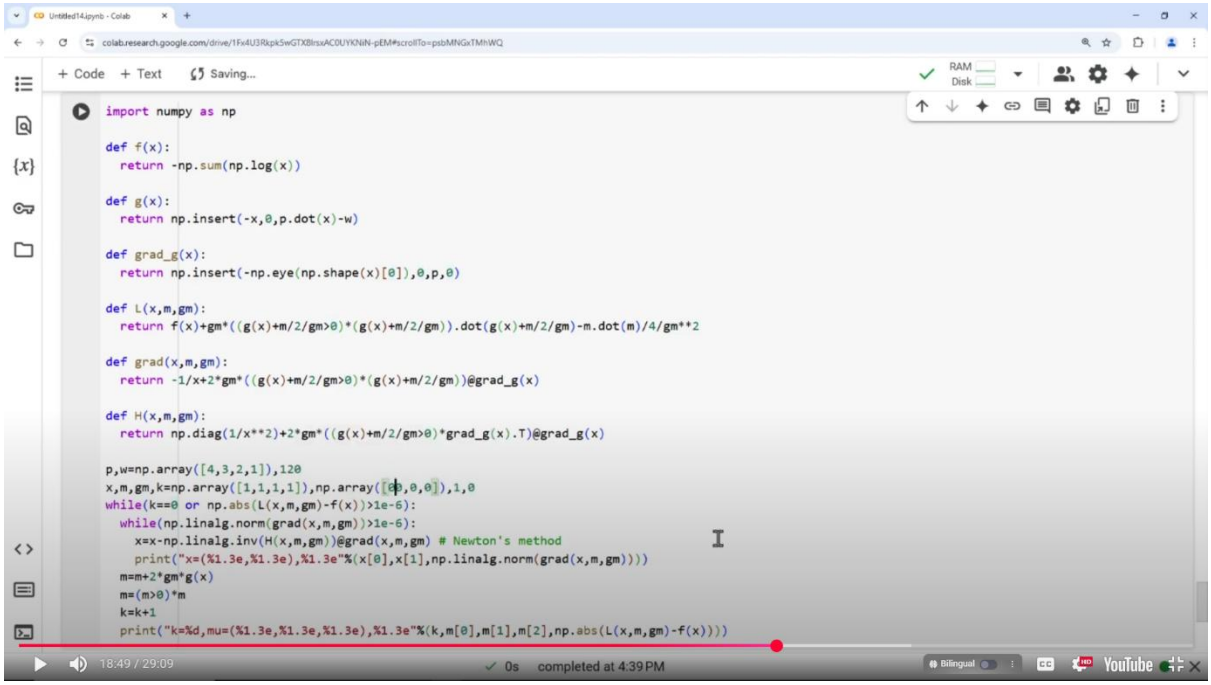
$\lambda^* = 4/120 = 1/30 \approx 0.0333$

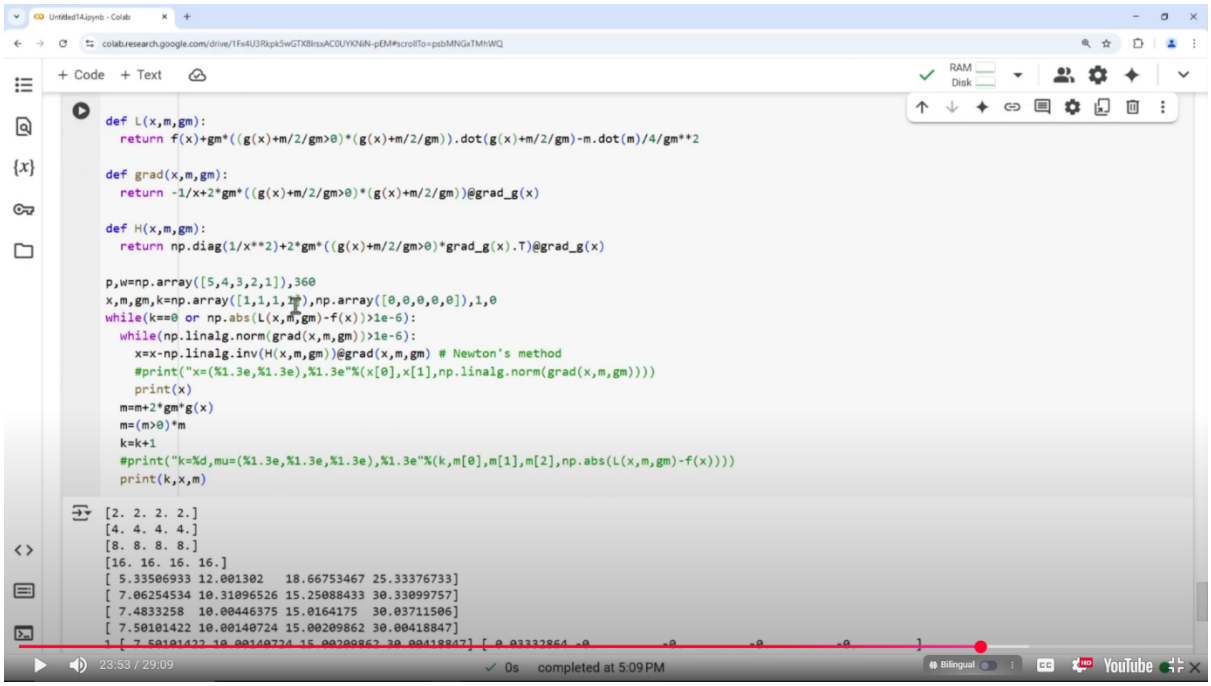
$\mu^* = 0$

We will run the code to verify if we get this solution. In a few steps, we should get the solution: $x \approx [7.5, 10, 15, 30]$, and μ should be very small (effectively 0).

(Refer Slides Time 18:55-24:30)


Templet - Windows Journal
$L = -\sum_{i=1}^{n} \log(x_i) + \lambda\left(\sum_{i=1}^{n} p_i x_i - w\right) - \sum_{i=1}^{n} \mu_i x_i$
$\nabla L = -\frac{1}{x_i} + \lambda p_i - \mu_i = 0 \quad \forall i = 1, \dots, n.$
$\lambda = \frac{1}{p_i x_i} \quad \forall i$
$\Rightarrow p_i x_i$ is a constant over i.
Now $\sum p_i x_i = w \Rightarrow p_i x_i = \frac{w}{n} \Rightarrow x_i^* = \frac{w}{n p_i}$
$\lambda^* = \frac{n}{w}, \quad \mu^* = 0$
1. $(p_1, p_2, p_3, p_4) = (4, 3, 2, 1), \quad w = 120.$
$x^* = (7.5, 10, 15, 30), \quad \lambda^* = \frac{1}{30} = 0.0333, \quad \mu^* = 0.$
2. $p = (5, 4, 3, 2, 1), \quad w = 360$
$x^* = (14.4, 18, 24, 36, 72), \quad \lambda^* = \frac{1}{72} = 0.01 \quad \mu^* = 0.$
23:27 / 29:09
Bilingual
YouTube

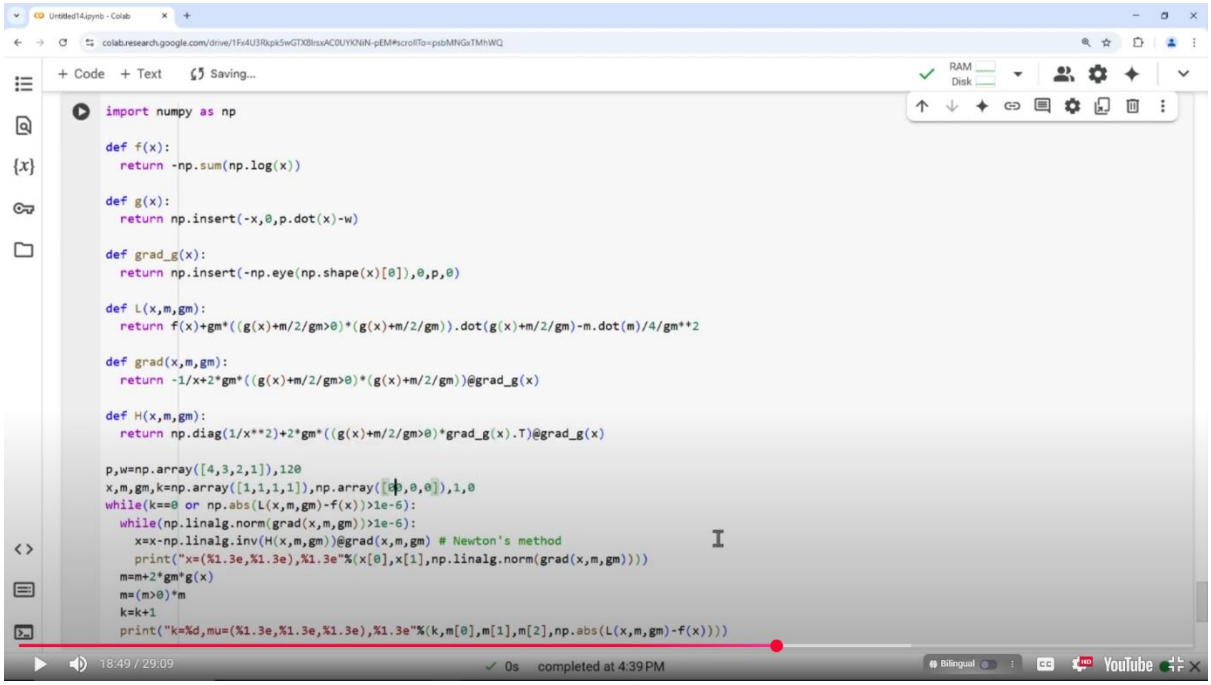

Untitled14.ipynb - Colab
colab.research.google.com/drive/1Fx4U3Rkpk5wGTX8lrsxAC0UYKNiN-pEM#scrollTo=psbMNGxTMhWQ
+ Code + Text Saving...
RAM
Disk
import numpy as np
def f(x):
return -np.sum(np.log(x))
def g(x):
return np.insert(-x,0,p.dot(x)-w)
def grad_g(x):
return np.insert(-np.eye(np.shape(x)[0]),0,p,0)
def L(x,m,gm):
return f(x)+gm*((g(x)+m/2/gm>0)*(g(x)+m/2/gm)).dot(g(x)+m/2/gm)-m.dot(m)/4/gm**2
def grad(x,m,gm):
return -1/x+2*gm*((g(x)+m/2/gm>0)*(g(x)+m/2/gm))@grad_g(x)
def H(x,m,gm):
return np.diag(1/x**2)+2*gm*((g(x)+m/2/gm>0)*grad_g(x).T)@grad_g(x)
p,w=np.array([4,3,2,1]),120
x,m,gm,k=np.array([1,1,1,1]),np.array([0,0,0,0]),1,0
while(k==0 or np.abs(L(x,m,gm)-f(x))>1e-6):
while(np.linalg.norm(grad(x,m,gm))>1e-6):
x=x-np.linalg.inv(H(x,m,gm))@grad(x,m,gm) # Newton's method
print("x=(%1.3e,%1.3e),%1.3e"%(x[0],x[1],np.linalg.norm(grad(x,m,gm))))
m=m+2*gm*g(x)
m=(m>0)*m
k=k+1
print("k=%d,mu=(%1.3e,%1.3e,%1.3e),%1.3e"%(k,m[0],m[1],m[2],np.abs(L(x,m,gm)-f(x))))
18:49 / 29:09
0s completed at 4:39 PM
Bilingual
YouTube



```
def L(x,m,gm):
  return f(x)+gm*((g(x)+m/2/gm>0)*(g(x)+m/2/gm)).dot(g(x)+m/2/gm)-m.dot(m)/4/gm**2

def grad(x,m,gm):
  return -1/x+2*gm*((g(x)+m/2/gm>0)*(g(x)+m/2/gm))@grad_g(x)

def H(x,m,gm):
  return np.diag(1/x**2)+2*gm*((g(x)+m/2/gm>0)*grad_g(x).T)@grad_g(x)

p,w=np.array([5,4,3,2,1]),360
x,m,gm,k=np.array([1,1,1,1,1]),np.array([0,0,0,0,0]),1,0
while(k==0 or np.abs(L(x,m,gm)-f(x))>1e-6):
  while(np.linalg.norm(grad(x,m,gm))>1e-6):
    x=x-np.linalg.inv(H(x,m,gm))@grad(x,m,gm) # Newton's method
    #print("x=(%1.3e,%1.3e),%1.3e"%(x[0],x[1],np.linalg.norm(grad(x,m,gm))))
    print(x)
  m=m+2*gm*g(x)
  m=(m>0)*m
  k=k+1
  #print("k=%d,mu=(%1.3e,%1.3e,%1.3e),%1.3e"%(k,m[0],m[1],m[2],np.abs(L(x,m,gm)-f(x))))
  print(k,x,m)
```

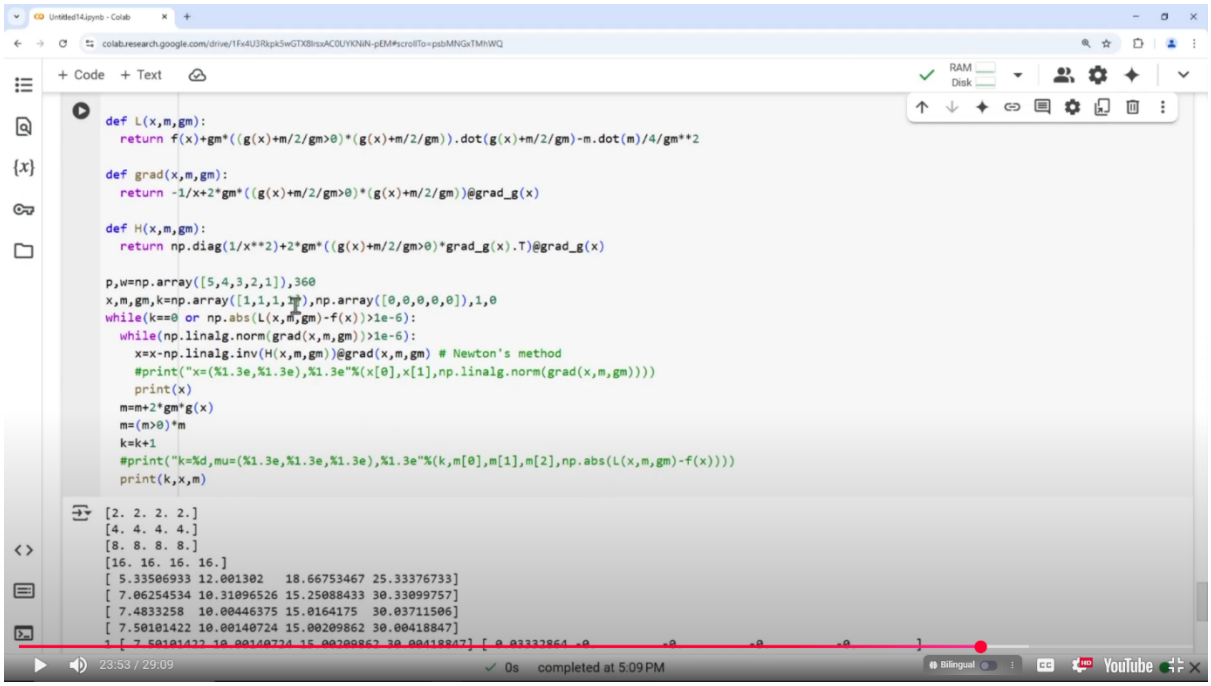
```
[2. 2. 2. 2.]
[4. 4. 4. 4.]
[8. 8. 8. 8.]
[16. 16. 16. 16.]
[ 5.33506933 12.001302   18.66753467 25.33376733]
[ 7.06254534 10.31096526 15.25088433 30.33099757]
[ 7.4833258  10.00446375 15.0164175  30.03711506]
[ 7.50101422 10.00140724 15.00209862 30.00418847]
1 [ 7.50101422 10.00140724 15.00209862 30.00418847] [ 0.03332864 -0.         -0.         -0.         -0.        ]
```

23:53 / 29:09 ✓ 0s completed at 5:09 PM



```
import numpy as np

def f(x):
  return -np.sum(np.log(x))

def g(x):
  return np.insert(-x,0,p.dot(x)-w)

def grad_g(x):
  return np.insert(-np.eye(np.shape(x)[0]),0,p,0)

def L(x,m,gm):
  return f(x)+gm*((g(x)+m/2/gm>0)*(g(x)+m/2/gm)).dot(g(x)+m/2/gm)-m.dot(m)/4/gm**2

def grad(x,m,gm):
  return -1/x+2*gm*((g(x)+m/2/gm>0)*(g(x)+m/2/gm))@grad_g(x)

def H(x,m,gm):
  return np.diag(1/x**2)+2*gm*((g(x)+m/2/gm>0)*grad_g(x).T)@grad_g(x)

p,w=np.array([4,3,2,1]),120
x,m,gm,k=np.array([1,1,1,1]),np.array([0,0,0,0]),1,0
while(k==0 or np.abs(L(x,m,gm)-f(x))>1e-6):
  while(np.linalg.norm(grad(x,m,gm))>1e-6):
    x=x-np.linalg.inv(H(x,m,gm))@grad(x,m,gm) # Newton's method
    print("x=(%1.3e,%1.3e),%1.3e"%(x[0],x[1],np.linalg.norm(grad(x,m,gm))))
  m=m+2*gm*g(x)
  m=(m>0)*m
  k=k+1
  print("k=%d,mu=(%1.3e,%1.3e,%1.3e),%1.3e"%(k,m[0],m[1],m[2],np.abs(L(x,m,gm)-f(x))))
```

18:49 / 29:09 ✓ 0s completed at 4:39 PM



The beauty of this code is that it is written for any general n. The previous examples were for specific problems where n and the number of constraints were fixed. This code is written in general for any n, any p, and any w. You can give different values of p and w, and even change the dimension n. You just have to modify the initial values of x and μ accordingly. You can make the code more general by initializing x as `np.ones(len(p))` and μ as `np.zeros(len(p)+1)`.

You can try with various values of p and w, and the dimension n can also vary. This is the example I wanted to work out.

In the next lecture, I would like to let you know some things about the linear programming problem. Initially, before starting with the quadratic penalty method in Week 7, we discussed the differences between linear optimization problems and non-linear optimization problems.

We said that for non-linear optimization problems, we would discuss the quadratic penalty method and the augmented Lagrangian method.

What happens if we use these methods for linear programming? It appears general enough: in linear programming, the objective function is linear ($f(x) = c^Tx$), and the constraints are linear ($g_i(x) = a_i^Tx - b_i \leq 0$, $h_j(x) = \bar{a}_j^Tx - \bar{b}_j = 0$).

You could substitute these into the methods. However, in most cases, when you use the quadratic penalty method and the augmented Lagrangian methods, you will not get the answer for a linear programming problem.

You need a separate set of algorithms for linear programming problems. I will explain that in the next lecture.

Thank you.