Optimization Algorithms: Theory and Software Implementation

Prof. Thirumulanathan D

Department of Mathematics

Institute of IIT Kanpur

Lecture: 50

Hello everyone, this is lecture 5 of week 10. Recall that in the past few lectures, we were learning about the affine scaling method. That is an interior point method where you start with an interior point of a linear programming problem and converge to the solution. We ended with the steps of the algorithm. Let me go through that once more before starting with an example.

The initial phase of the affine scaling algorithm requires converting the linear programming problem into its standard form, characterized by the matrices A, b, and the vector c. The objective is to

minimize $c^T x$

under the constraints

$A x = b$ and $x \geq 0$.

Here, A is an m×n matrix assumed to have full row rank, meaning its rank is m.

A fundamental requirement of this interior point method is the selection of a starting point, x_0. This point must be an interior point of the feasible polytope, signifying that it strictly satisfies all constraints: $A x_0 = b$ and $x_0 > 0$ (all components are positive).

A natural question arises: what occurs if a boundary point, such as a vertex (commonly used as an initial point in the simplex method), is chosen instead? This choice is problematic for the following reason:

The algorithm relies heavily on the scaling matrix, denoted X_k.

This matrix is a diagonal matrix constructed from the elements of the vector x_k at each iteration k (i.e., $X_k = \mathrm{diag}(x_k)$).

This matrix is used in operations involving its inverse, as seen in the expression $y_k = X_k^{-1} x_k$.

If any component of x_k is zero, the corresponding diagonal entry in X_k becomes zero. Consequently, the matrix X_k becomes singular, and its inverse, X_k^{-1}, is undefined. This renders calculations like $y_k = X_k^{-1} x_k$ impossible and halts the algorithm.

While it is theoretically possible for the larger matrix operation $A X^2 A^T$ to remain invertible even if X has zeros, the core mechanics of the algorithm depend on the invertibility of X_k itself. Therefore, to avoid numerical instability and ensure the algorithm functions correctly, the initial point x_0 must be chosen such that all its entries are strictly positive.

The affine scaling method differs fundamentally from the simplex method in its choice of initial point. The simplex method typically starts at a vertex, often the origin (0, 0, 0, 0), and moves along the boundary of the feasible region. In contrast, the affine scaling method must start from a strictly interior point where $x_0 > 0$.

Choosing a point with zero components is infeasible because the algorithm relies on the diagonal scaling matrix $X_k = \mathrm{diag}(x_k)$, which must be invertible.

If any element of x_k is zero, X_k becomes singular, and operations like $X_k^{-1} x_k$ are undefined. Although the larger matrix operation $(A X^2 A^T)$ might remain invertible with some zeros, the inherent risk of numerical failure necessitates a strictly positive initial point.

Therefore, we always choose an x_0 with all positive components.

The initialization for iteration k = 0 is crucial and involves two key constructs:

1. The Scaling Matrix: Form the diagonal matrix X_0 where each diagonal entry is the corresponding element from vector x_0.

2. The Dual Variable: Calculate μ_0 using the formula $\mu_0 = (A X_0^2 A^T)^{-1} A X_0^2 c$.

The dual variable μ is computed initially to evaluate the duality gap, $c^T x_k - b^T \mu_k$. This value serves as a stopping criterion; the algorithm iterates until this gap is within a specified tolerance.

After initialization, the algorithm enters its main loop, which consists of the following steps for each iteration k:

1. Compute the Direction: Calculate the direction vector d_k using the formula:

   $d_k = X_k (A^T \mu_k - c)$

2. Determine the Step Size: Calculate the maximum step size α_k that ensures the next iterate remains positive. This is done by considering only the negative components of d_k. The condition $1 + \alpha_k d_{kj} > 0$ must hold for all j. For any $d_{kj} < 0$, this implies $\alpha_k < -1 / d_{kj}$. To ensure strict positivity, we choose a step size slightly smaller than the most restrictive bound:

   $\alpha_k = \gamma \; \min(-1 / d_{kj})$ for all j where $d_{kj} < 0$.

   A common choice for the scaling factor γ is 0.9.

3. Update the Primal Variable: The new iterate x_{k+1} is obtained by moving in the calculated direction and then scaling back from the "centered" coordinates:

   $x_{k+1} = X_k (1 + \alpha_k d_k)$

   This update can be implemented efficiently via element-wise multiplication as

   $x_{k+1} = x_k + \alpha_k (x_k \; d_k)$.

The auxiliary variables $y_k = X_k^{-1} x_k$ are conceptual tools for deriving the algorithm but are not needed in the practical implementation and can be omitted. After updating x, the matrices X and μ are recalculated for the next iteration. This process repeats until the duality gap is sufficiently small.

#Practical Example

We will illustrate this algorithm using a familiar linear programming problem:

Objective: Minimize $-x_2$ (which is equivalent to maximizing x_2).

Constraints:

$-x_1 + x_2 \leq 0$ => $x_1 \geq x_2$

$x_1 + x_2 \leq 1$

$x_2 \geq 0$

Standard Form Conversion:

To convert this into standard form ($A x = b$, $x \geq 0$), we introduce slack variables and split the free variable x_1 into its positive and negative parts (x_1^+, x_1^-).

Variables: $x = [x_1^+, x_1^-, x_2, x_3, x_4]^T$

Constraint Matrix A:

A = [[-1, 1, 1, 1, 0], (from $-x_1^+ + x_1^- + x_2 + x_3 = 0$)

[1, -1, 1, 0, 1]] (from $x_1^+ - x_1^- + x_2 + x_4 = 1$)

Right-hand side b: $b = [0, 1]^T$

Cost Vector c: $c = [0, 0, -1, 0, 0]^T$ (since we minimize $-x_2$)

Choosing an Initial Point:

We must choose a strictly interior point $x_0 > 0$ that satisfies $A x_0 = b$. Let's select a point in the original space: $(x_1, x_2) = (0.1, 0.05)$.

We map this to the standard form variables:

$x_1^+ = 0.10$, $x_1^- = 0.00$, $x_2 = 0.05$

To satisfy the first constraint: $-0.10 + 0.00 + 0.05 + x_3 = 0$ => $x_3 = 0.05$

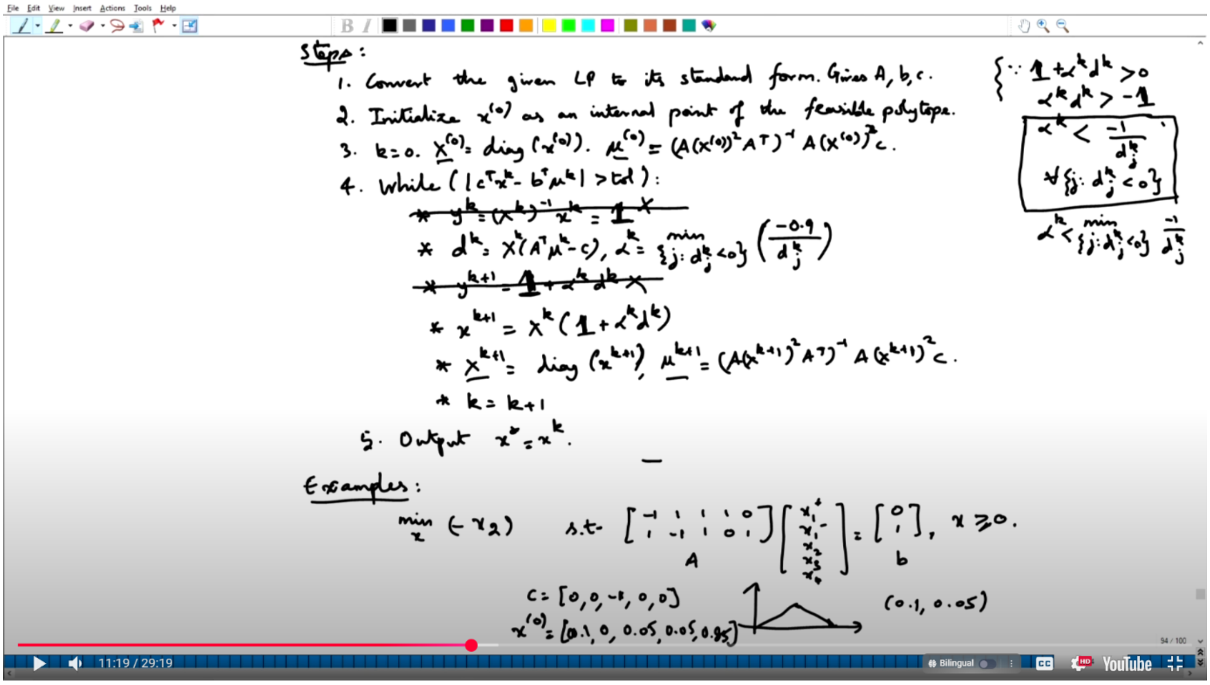
To satisfy the second constraint: $0.10 - 0.00 + 0.05 + x_4 = 1$ => $x_4 = 0.85$

Thus, our initial interior point is:

$x_0 = [0.10, 0.00, 0.05, 0.05, 0.85]^T$

Note: This point has a zero component ($x_1^- = 0$), which violates the requirement for a strictly interior point. A correct initial point must have $x_1^- > 0$ as well. A better choice would be a point like $(x_1, x_2) = (0.25, 0.2)$, which would lead to all positive values for x_1^+, x_1^-, x_2, and the slack variables. The provided example contains an oversight; the algorithm must start from a fully positive vector.

(Refer Slide Time 11:19)



Steps:

```
1. Convert the given LP to its standard form. Given A, b, c.
2. Initialize $x^{(0)}$ as an internal point of the feasible polytope.
3. k=0. $\underline{X}^{(0)} = \mathrm{diag}(x^{(0)})$. $\underline{\mu}^{(0)} = (A(X^{(0)})^2 A^T)^{-1} A (X^{(0)})^2 c$.
4. While ( $|c^T x^k - b^T \mu^k| > tol$ ):
      ~~* $y^k = (X^k)^{-1} x^k = \mathbb{1}$~~
      * $d^k = X^k(A^T \mu^k - c)$, $\alpha^k = \min_{\{j : d_j^k < 0\}} \left(\frac{-0.9}{d_j^k}\right)$
      ~~* $y^{k+1} = \mathbb{1} + \alpha^k d^k$~~
      * $x^{k+1} = X^k(\mathbb{1} + \alpha^k d^k)$
      * $\underline{X}^{k+1} = \mathrm{diag}(x^{k+1})$, $\underline{\mu}^{k+1} = (A(X^{k+1})^2 A^T)^{-1} A (X^{k+1})^2 c$.
      * $k = k+1$
5. Output $x^* = x^k$.
```

$\because \mathbb{1} + \alpha^k d^k > 0$

$\alpha^k d^k > -\mathbb{1}$

$$\boxed{\begin{array}{c} \alpha^k < \frac{-1}{d_j^k} \\ \forall \{j : d_j^k < 0\} \end{array}}$$

$$\alpha^k < \min_{\{j : d_j^k < 0\}} \frac{-1}{d_j^k}$$

Examples:

$$\min_x (-x_2) \quad \text{s.t.} \quad \underbrace{\begin{bmatrix} -1 & 1 & 1 & 1 & 0 \\ 1 & -1 & 1 & 0 & 1 \end{bmatrix}}_{A} \begin{bmatrix} x_1^+ \\ x_1^- \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \underbrace{\begin{bmatrix} 0 \\ 1 \end{bmatrix}}_{b}, \quad x \geq 0.$$

$c = [0, 0, -1, 0, 0]$

$x^{(0)} = [0.1, 0, 0.05, 0.05, 0.85]$

$(0.1, 0.05)$

With this, I will write down the algorithm in Python for the affine scaling method. I will start by importing the NumPy library as usual.

We define our problem matrices and vectors:

A = np.array([[-1, 1, 1, 1, 0], [1, -1, 1, 0, 1]])

b = np.array([0, 1])

c = np.array([0, 0, -1, 0, 0])

x = np.array([0.10, 0.05, 0.05, 0.85])

The first two steps are now complete: the LP has been converted to its standard form, and an initial interior point x_0 has been chosen.

We set the iteration counter k = 0.

The next steps involve initializing the matrices for the first iteration.

X = np.diag(x) creates the diagonal matrix X_k from the vector x_k.

The dual variable μ is calculated using the formula

μ = np.linalg.inv(A @ X @ X @ A.T) @ A @ X @ X @ c.

It is crucial to use the @ operator for matrix multiplication, as the asterisk would perform element-wise multiplication, which is incorrect in this context.

Given these initial values, we will first print them. We are interested in the iteration count k, the primal variable x, and the absolute value of the duality gap $|c^T x - b^T \mu|$.

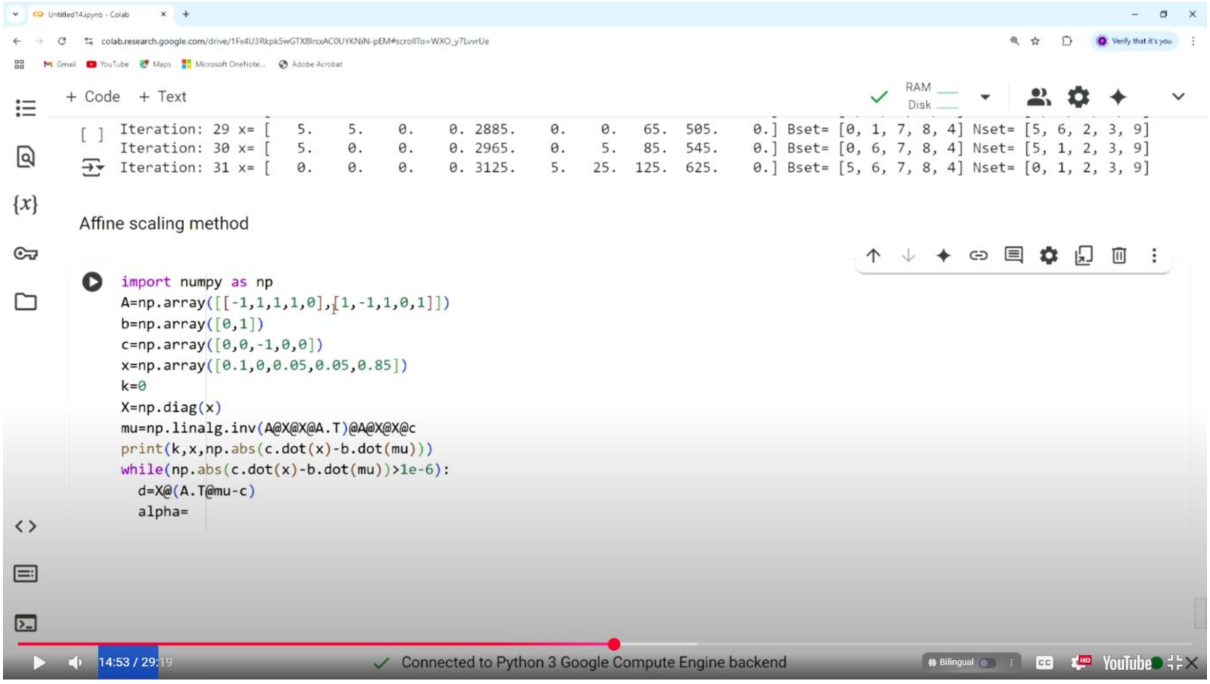
We then enter the main loop. The loop continues while the absolute duality gap is greater than a specified tolerance (e.g., 1e-6).

Within the loop, the first step is to compute the direction vector d:

d = X @ (A.T @ μ - c)

Next, we calculate the step size α.

(Refer Slide Time 14:53)



```
Iteration: 29 x= [   5.    5.    0.    0. 2885.    0.    0.   65.  505.    0.] Bset= [0, 1, 7, 8, 4] Nset= [5, 6, 2, 3, 9]
Iteration: 30 x= [   5.    0.    0.    0. 2965.    0.    5.   85.  545.    0.] Bset= [0, 6, 7, 8, 4] Nset= [5, 1, 2, 3, 9]
Iteration: 31 x= [   0.    0.    0.    0. 3125.    5.   25.  125.  625.    0.] Bset= [5, 6, 7, 8, 4] Nset= [0, 1, 2, 3, 9]
```

Affine scaling method

```
import numpy as np
A=np.array([[-1,1,1,1,0],[1,-1,1,0,1]])
b=np.array([0,1])
c=np.array([0,0,-1,0,0])
x=np.array([0.1,0,0.05,0.05,0.85])
k=0
X=np.diag(x)
mu=np.linalg.inv(A@X@X@A.T)@A@X@X@c
print(k,x,np.abs(c.dot(x)-b.dot(mu)))
while(np.abs(c.dot(x)-b.dot(mu))>1e-6):
  d=X@(A.T@mu-c)
  alpha=
```

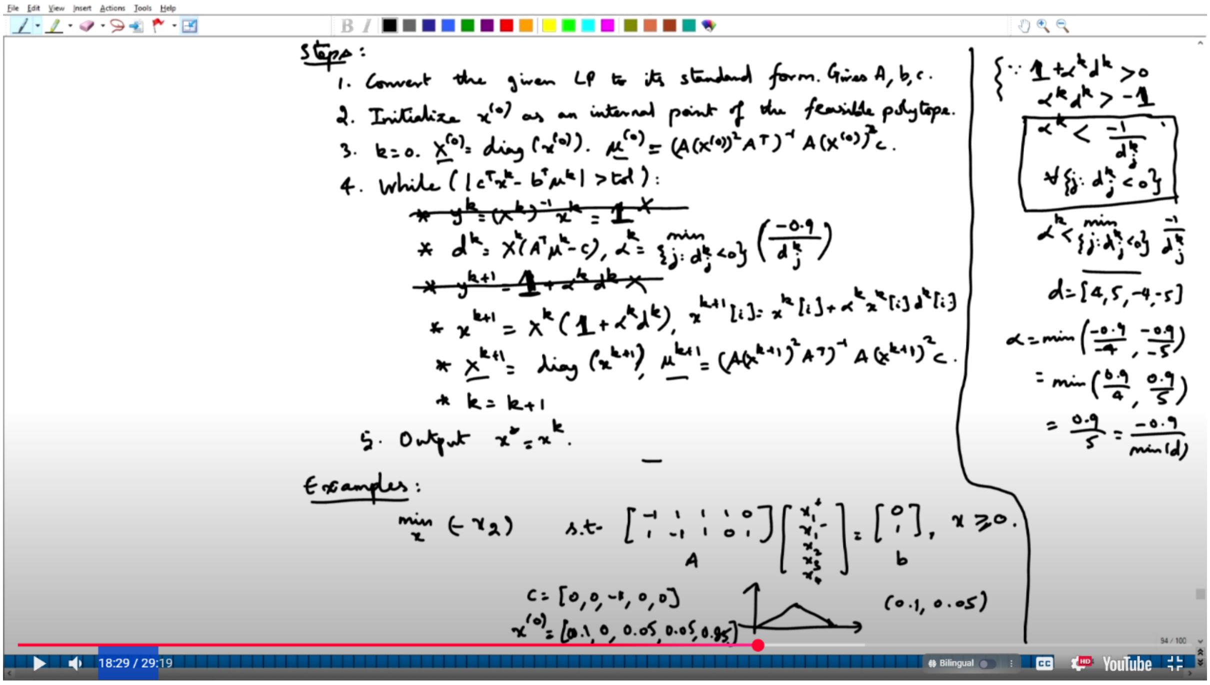
Untitled14.ipynb - Colab
colab.research.google.com/drive/1Fx4U3RkpkSwGTXBlrsxAC0UYKNiN-pEM#scrollTo=WXO_y7LvvrUe
+ Code
+ Text
RAM
Disk
14:53 / 29:19
Connected to Python 3 Google Compute Engine backend

There is an important detail here. The step size must be chosen to ensure the next iterate remains strictly positive. We only need to consider the negative components of d. For each component d_j where $d_j < 0$, we require $\alpha < -1 / d_j$.

To ensure this holds for all such components, we set α to be slightly less than the minimum of $-1 / d_j$ over all negative d_j. A common choice is $\alpha = -0.9 / \min(d)$, where min(d) finds the smallest (most negative) value in the vector d.

This is equivalent to calculating $\min(-0.9 / d_j)$ for all j where $d_j < 0$.

(Refer Slide Time 18:29)



Steps:

1. Convert the given LP to its standard form. Gives A, b, c.
2. Initialize $x^{(0)}$ as an internal point of the feasible polytope.
3. $k=0$. $\underline{X}^{(0)} = \text{diag}(x^{(0)})$. $\underline{\mu}^{(0)} = (A(X^{(0)})^2 A^T)^{-1} A(X^{(0)})^2 c$.
4. While $(|c^T x^k - b^T \mu^k| > tol)$:
   - ~~$* \; y^k = (X^k)^{-1} x^k = \mathbb{1}$~~ ✗
   - $* \; d^k = X^k(A^T\mu^k - c)$, $\alpha^k = \min_{\{j : d_j^k < 0\}} \left(\frac{-0.9}{d_j^k}\right)$
   - ~~$* \; y^{k+1} = \mathbb{1} + \alpha^k d^k$~~ ✗
   - $* \; x^{k+1} = X^k(\mathbb{1} + \alpha^k d^k)$, $x^{k+1}[i] = x^k[i] + \alpha^k x^k[i] d^k[i]$
   - $* \; \underline{X}^{k+1} = \text{diag}(x^{k+1})$, $\underline{\mu}^{k+1} = (A(X^{k+1})^2 A^T)^{-1} A(X^{k+1})^2 c$.
   - $* \; k = k+1$
5. Output $x^* = x^k$.

$\because \; \mathbb{1} + \alpha^k d^k > 0$

$\alpha^k d^k > -\mathbb{1}$

$$\boxed{\alpha^k < \frac{-1}{d_j^k} \quad \forall \{j : d_j^k < 0\}}$$

$\alpha^k < \min_{\{j : d_j^k < 0\}} \frac{-1}{d_j^k}$

$d = [4, 5, -4, -5]$

$\alpha = \min\left(\frac{-0.9}{-4}, \frac{-0.9}{-5}\right)$

$= \min\left(\frac{0.9}{4}, \frac{0.9}{5}\right)$

$= \frac{0.9}{5} = \frac{-0.9}{\min(d)}$

Examples:

$$\min_x (-x_2) \quad \text{s.t.} \underbrace{\begin{bmatrix} -1 & 1 & 1 & 1 & 0 \\ 1 & -1 & 1 & 0 & 1 \end{bmatrix}}_{A} \begin{bmatrix} x_1^+ \\ x_1^- \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \underbrace{\begin{bmatrix} 0 \\ 1 \end{bmatrix}}_{b}, \; x \geq 0.$$

$c = [0, 0, -1, 0, 0]$

$x^{(0)} = [0.1, 0, 0.05, 0.05, 0.95]$

$(0.1, 0.05)$

With d and α computed, we update the primal variable:

x = X @ (1 + α d)

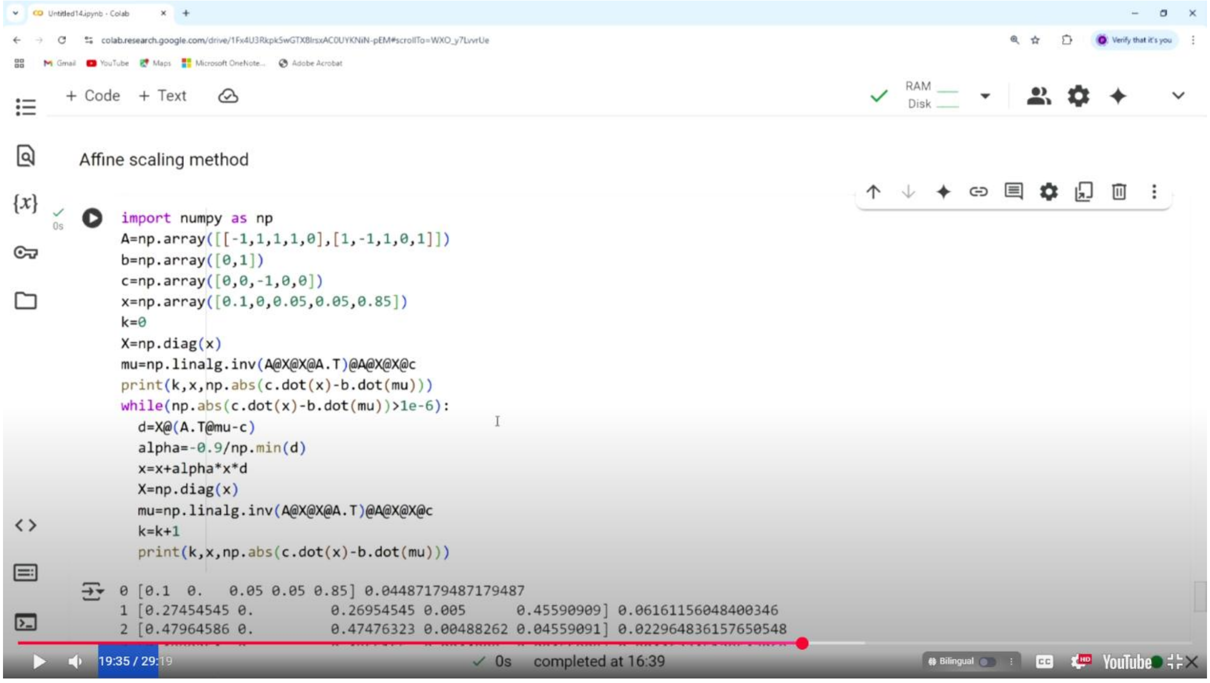
Note that X is a diagonal matrix. Multiplying it by the vector of ones simply returns the original vector x. The term α d is a vector. Therefore, this update is equivalent to the element-wise operation:

x = x + α (x d)

where x d denotes element-wise multiplication.

After updating x, we must recalculate the diagonal matrix X and the dual variable μ for the next iteration. Finally, we increment the iteration counter k and print the new values.

(Refer Slide Time 19:35)



Affine scaling method

```
import numpy as np
A=np.array([[-1,1,1,1,0],[1,-1,1,0,1]])
b=np.array([0,1])
c=np.array([0,0,-1,0,0])
x=np.array([0.1,0,0.05,0.05,0.85])
k=0
X=np.diag(x)
mu=np.linalg.inv(A@X@X@A.T)@A@X@X@c
print(k,x,np.abs(c.dot(x)-b.dot(mu)))
while(np.abs(c.dot(x)-b.dot(mu))>1e-6):
  d=X@(A.T@mu-c)
  alpha=-0.9/np.min(d)
  x=x+alpha*x*d
  X=np.diag(x)
  mu=np.linalg.inv(A@X@X@A.T)@A@X@X@c
  k=k+1
  print(k,x,np.abs(c.dot(x)-b.dot(mu)))
```

```
0 [0.1  0.   0.05 0.05 0.85] 0.04487179487179487
1 [0.27454545 0.         0.26954545 0.005      0.45590909] 0.06161156048400346
2 [0.47964586 0.         0.47476323 0.00488262 0.04559091] 0.022964836157650548
```

I expect this implementation to work correctly.

Let us check the result. The expected solution is [0.5, 0, 0.5, 0, 0].

This corresponds to $x_1 = 0.5$ and $x_2 = 0.5$, which is the point where x_2 is maximized.

This outcome is straightforward, given that we have implemented the algorithm step by step.

I would like to do one more thing: plot the trajectory of the points generated by the algorithm. I want to demonstrate that the trajectory remains within the interior of the feasible set. We started at the interior point (0.1, 0.05).

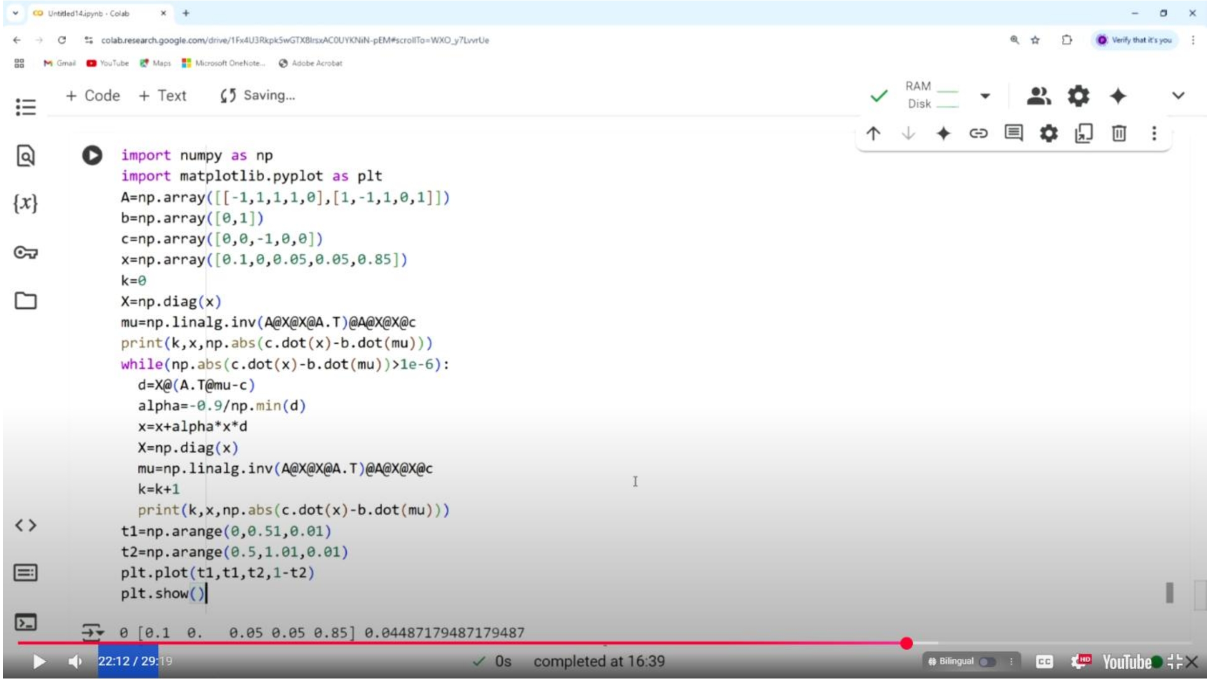
I want to show how the points move before reaching the solution (0.5, 0.5).

I will use the matplotlib.pyplot library for plotting. The first task is to plot the feasible set, which is defined by the lines $x_1 = x_2$ and $x_1 + x_2 = 1$.

Let t_1 = np.arange(0, 0.51, 0.01). For the line $x_1 = x_2$, we plot t_1 against t_1.

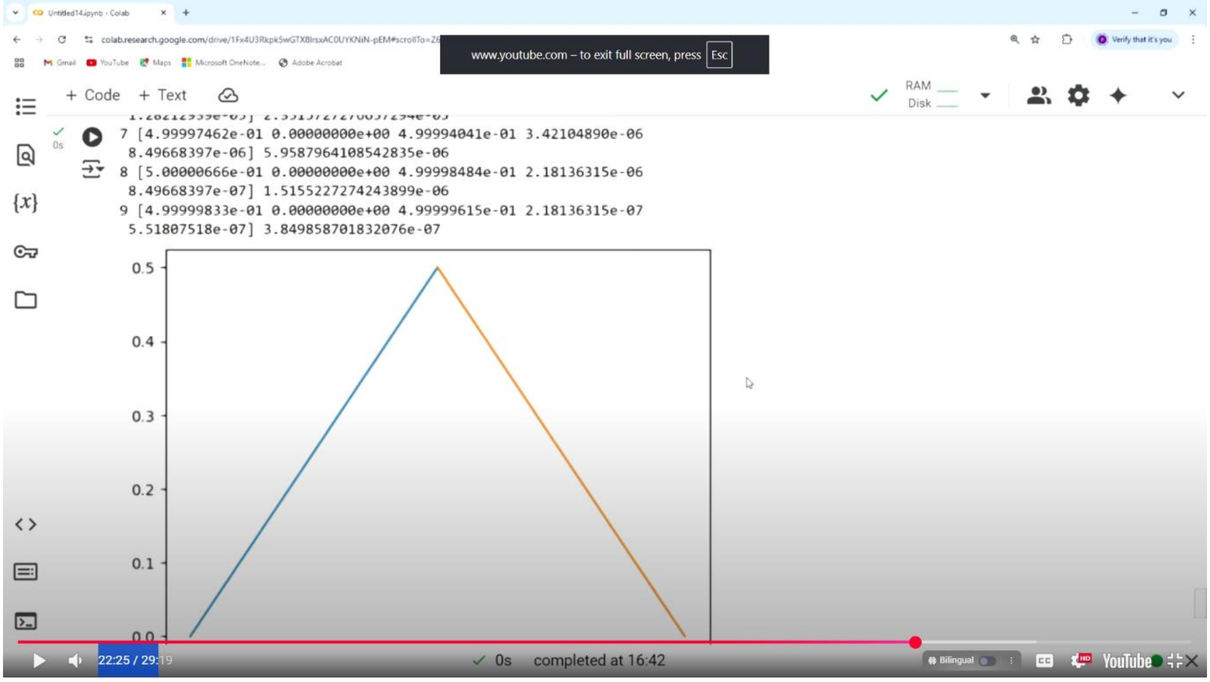
Let t_2 = np.arange(0.5, 1.01, 0.01). For the line $x_1 + x_2 = 1$, we plot t_2 against $1 - t_2$.

(Refer Slide Time 22:00-23:00)



```
import numpy as np
import matplotlib.pyplot as plt
A=np.array([[-1,1,1,1,0],[1,-1,1,0,1]])
b=np.array([0,1])
c=np.array([0,0,-1,0,0])
x=np.array([0.1,0,0.05,0.05,0.85])
k=0
X=np.diag(x)
mu=np.linalg.inv(A@X@X@A.T)@A@X@X@c
print(k,x,np.abs(c.dot(x)-b.dot(mu)))
while(np.abs(c.dot(x)-b.dot(mu))>1e-6):
  d=X@(A.T@mu-c)
  alpha=-0.9/np.min(d)
  x=x+alpha*x*d
  X=np.diag(x)
  mu=np.linalg.inv(A@X@X@A.T)@A@X@X@c
  k=k+1
  print(k,x,np.abs(c.dot(x)-b.dot(mu)))
t1=np.arange(0,0.51,0.01)
t2=np.arange(0.5,1.01,0.01)
plt.plot(t1,t1,t2,1-t2)
plt.show()
```

```
0 [0.1  0.   0.05 0.05 0.85] 0.04487179487179487
```



```
7 [4.99997462e-01 0.00000000e+00 4.99994041e-01 3.42104890e-06
 8.49668397e-06] 5.958796410854283e-06
8 [5.00000666e-01 0.00000000e+00 4.99998484e-01 2.18136315e-06
 8.49668397e-07] 1.5155227274243899e-06
9 [4.99999833e-01 0.00000000e+00 4.99999615e-01 2.18136315e-07
 5.51807518e-07] 3.849858701832076e-07
```

This will visualize the triangular feasible region.

Now, we need to capture the trajectory.

The variable x is a five-dimensional vector. To recover the original variables (x_1, x_2), we calculate x_1 = x[0] - x[1] and x_2 = x[2]. At each iteration of the algorithm, we will append the calculated (x_1, x_2) point to a list for plotting.

After running the algorithm, we can plot this list of points. The blue and orange lines will represent the boundaries of the feasible set, and the green line will show the algorithm's trajectory. The trajectory starts at (0.1, 0.05).

It moves very close to the line $x_1 = x_2$ and then continues roughly parallel to this line until it converges to (0.5, 0.5). This plot confirms that the entire path lies within the interior of the feasible set.

To visualize this more clearly, we can try a different initial point, such as (0.45, 0.05).

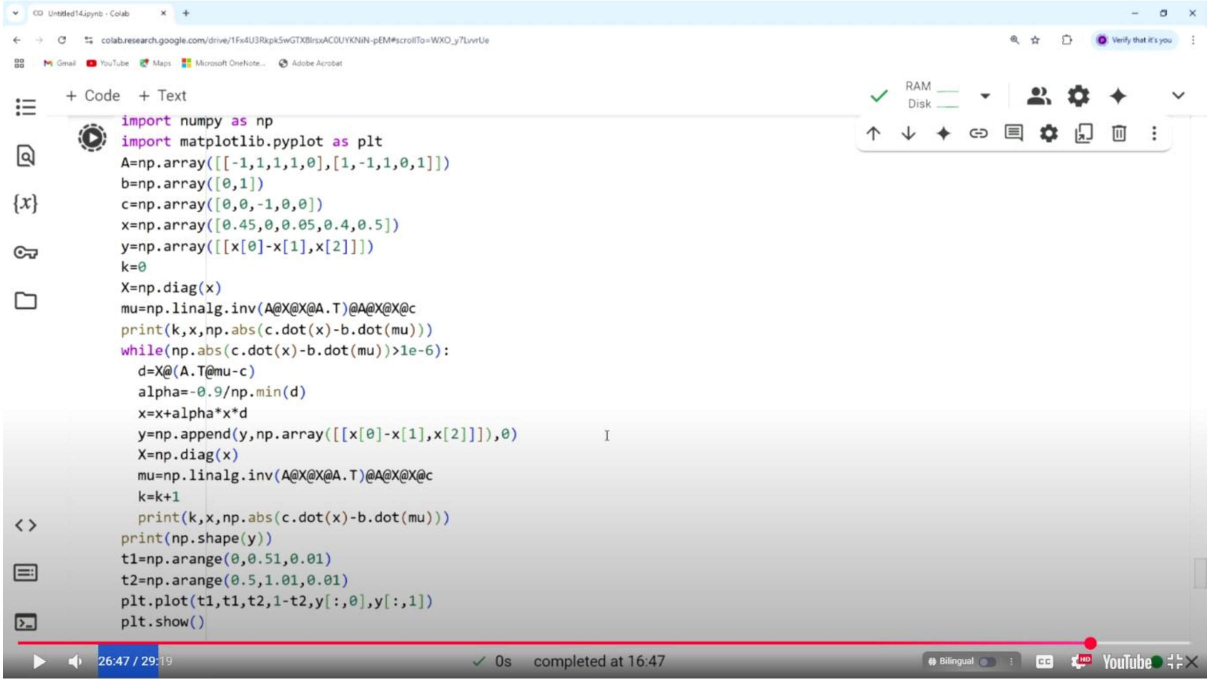
The slack variables must be adjusted accordingly to ensure A x = b is satisfied. The resulting trajectory will also remain entirely within the interior.

We can also experiment with different objective functions. For example, to maximize x_1 instead, we change the cost vector to c = [-1, 0, 0, 0, 0].

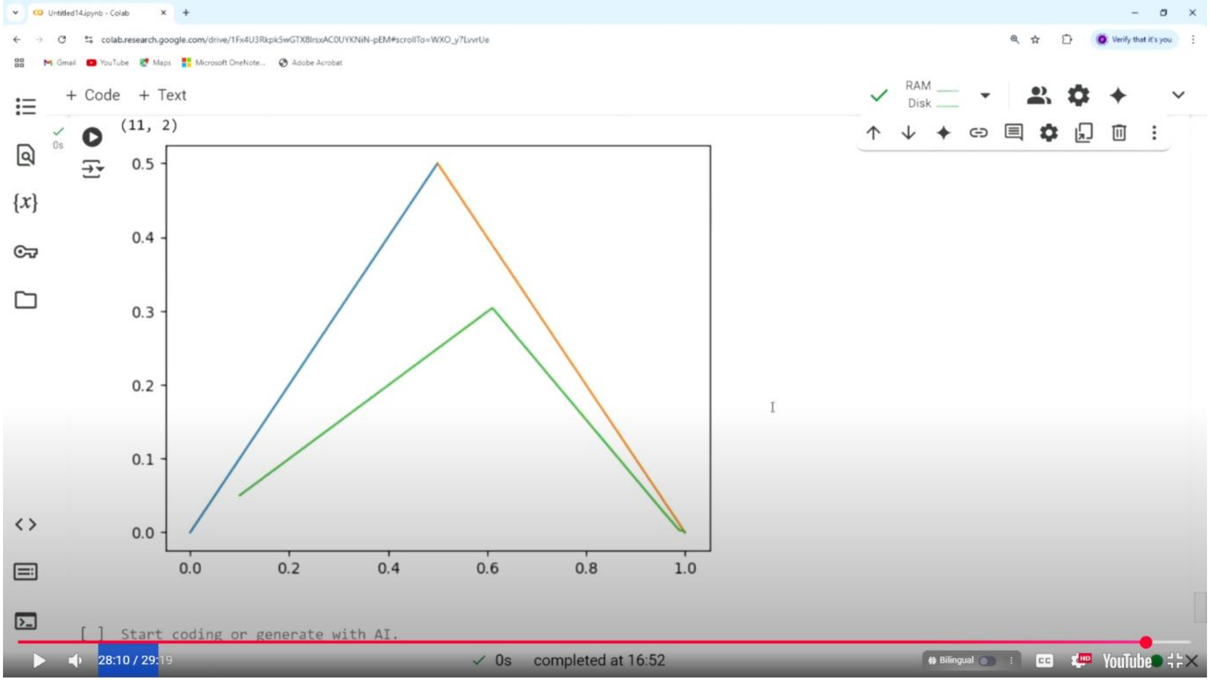
The expected solution is (1, 0).

The trajectory for this problem will move parallel to the boundary $x_2 = 0$ until it converges to the solution. The trajectory for the initial point [0.1, 0.05, 0.05, 0.85] with this new objective will also remain in the interior.

(Refer Slide Time 26:00-29:00)



```
import numpy as np
import matplotlib.pyplot as plt
A=np.array([[-1,1,1,1,0],[1,-1,1,0,1]])
b=np.array([0,1])
c=np.array([0,0,-1,0,0])
x=np.array([0.45,0,0.05,0.4,0.5])
y=np.array([[x[0]-x[1],x[2]]])
k=0
X=np.diag(x)
mu=np.linalg.inv(A@X@X@A.T)@A@X@X@c
print(k,x,np.abs(c.dot(x)-b.dot(mu)))
while(np.abs(c.dot(x)-b.dot(mu))>1e-6):
  d=X@(A.T@mu-c)
  alpha=-0.9/np.min(d)
  x=x+alpha*x*d
  y=np.append(y,np.array([[x[0]-x[1],x[2]]]),0)
  X=np.diag(x)
  mu=np.linalg.inv(A@X@X@A.T)@A@X@X@c
  k=k+1
  print(k,x,np.abs(c.dot(x)-b.dot(mu)))
print(np.shape(y))
t1=np.arange(0,0.51,0.01)
t2=np.arange(0.5,1.01,0.01)
plt.plot(t1,t1,t2,1-t2,y[:,0],y[:,1])
plt.show()
```



```
(11, 2)
```

The reason for demonstrating these trajectories is to show that the algorithm's path is always within the interior of the feasible polytope. It only touches the boundary upon convergence to the optimal solution. This property is the defining characteristic of interior point methods.

In the next lecture, we will examine a few more examples before concluding our discussion on the affine scaling method. Thank you.