Optimization Algorithms: Theory and Software Implementation

Prof. Thirumulanathan D

Department of Mathematics

Indian Institute of Technology Kanpur

Lecture: 17

Hello everyone. This is the second lecture in the fourth week. In the previous lecture, we derived the relation between the error function at the iterate x^{k+1} and the error function at x^k. The relation is that the error at x^{k+1} is less than or equal to $[(\lambda_n - \lambda_1) / (\lambda_n + \lambda_1)]^2$ multiplied by the error at x^k, where λ_1 is the smallest eigenvalue and λ_n is the largest eigenvalue of the Hessian matrix H.

When the ratio λ_1/λ_n is very low, the number of iterations required for the algorithm to converge to the solution becomes very large. We also examined an illustration of this phenomenon. Let us revisit that illustration.

Consider the function

$f(x_1, x_2) = x_1^2 + x_2^2$. For this function, the Hessian

H is [[2, 0], [0, 2]].

The eigenvalues are $\lambda_1=2$ and $\lambda_2=2$, so the ratio λ_1/λ_2 is 1.

The convergence occurred in just one step.

When the function was

$f(x_1, x_2) = 2x_1^2 + x_2^2$, the Hessian

H = [[4, 0], [0, 2]].

The eigenvalues are 4 and 2, so the ratio λ_1/λ_2 is 1/2. The convergence took 5 steps in this case.

When the function was

$f(x_1, x_2) = 5x_1^2 + x_2^2$, the Hessian

H = [[10, 0], [0, 2]].

The eigenvalues are 10 and 2, so the ratio λ_1/λ_2 is 1/5. The convergence took 8 steps.

When the function was

$f(x_1, x_2) = 50x_1^2 + x_2^2$,

the Hessian

H is [[100, 0], [0, 2]].

The eigenvalues are 100 and 2, so the ratio λ_1/λ_2 is 1/50. This took 316 steps to converge.

Finally, when the function was

$f(x_1, x_2) = 100x_1^2 + x_2^2$,

the Hessian

H = [[200, 0], [0, 2]].

The eigenvalues are 200 and 2, so the ratio λ_1/λ_2 is 1/100. This took 973 steps to converge.

We observed this behavior through a Python code in the last class. The trajectory for the last case started at (1, 100) and oscillated for a long time over 973 steps before finally converging to (0, 0).

The reason for this slow convergence is the quantity we derived, which shows that the error reduction depends on λ_n and λ_1. If λ_1/λ_n is very low, the number of steps required is enormous.

This analysis provides a clear picture of how the gradient descent algorithm performs. It is important to note that this analysis was done for a quadratic function. What happens if the function is not quadratic? The algorithm can still be used.

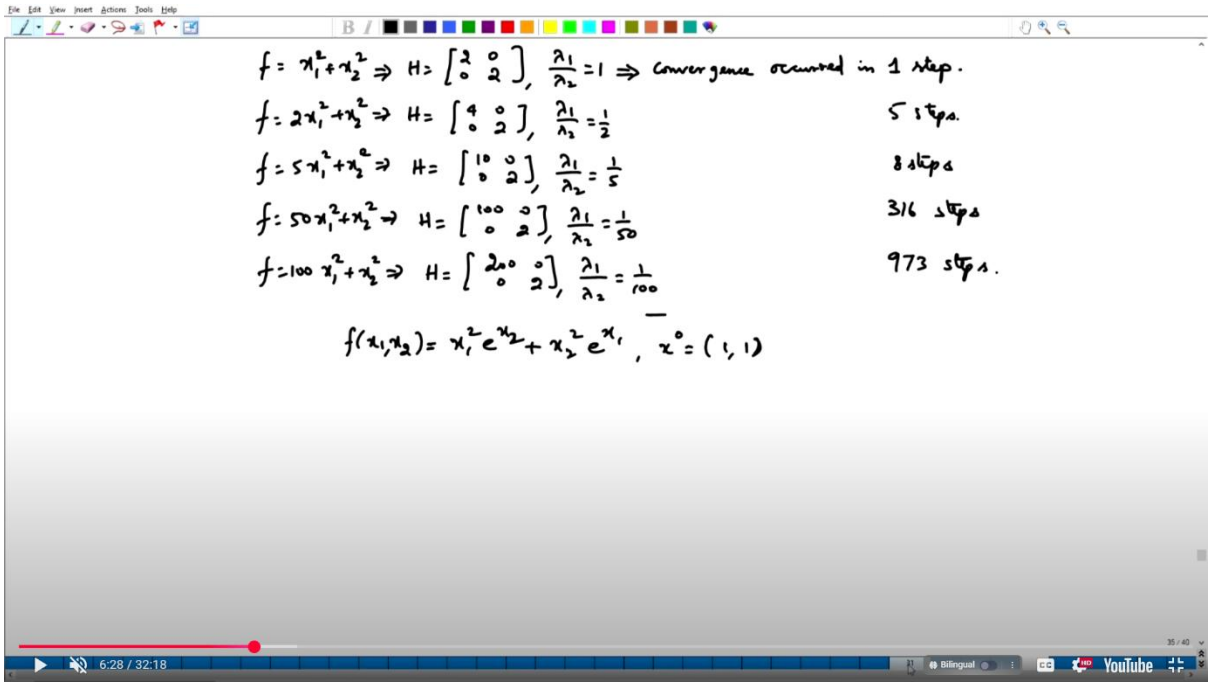
However, for a non-quadratic function, the Hessian matrix changes at every iteration. Furthermore, we must use a backtracking line search instead of the exact line search algorithm that we used for quadratic functions.

Let us consider an example that is hard to solve analytically.

Let $f(x_1, x_2) = x_1^2 e^{x_2} + x_2^2 e^{x_1}$.

If you look closely, you will notice that the minimum occurs at (0, 0) because all terms are non-negative and become zero only when $x_1 = x_2 = 0$. The minimizer is clearly (0, 0).

(Refer Slide Time 6:28)



Let us see how we can find this using the algorithm.

We initialize x^0 to be (1, 1). We set a tolerance, as usual, to 10^{-6}.

We choose the descent direction $d^k = -g^k$, which is the negative gradient.

We choose the step size α^k using the backtracking line search algorithm because an exact line search is analytically intractable for this problem.

We update $x^{k+1} = x^k + \alpha^k d^k$, which is $x^k - \alpha^k g^k$, and then increment k by 1.

We will implement this algorithm. The structure is similar to the previous algorithm for quadratic functions, but the method for choosing α changes.

We define the function f(x) = x[0]**2 * np.exp(x[1]) + x[1]**2 * np.exp(x[0]).

The gradient is computed as follows:

the partial derivative with respect to x_0 is $2*x_0*e^{x_1} + x_1^2*e^{x_0}$, and the partial derivative with respect to x_1 is $2*x_1*e^{x_0} + x_0^2*e^{x_1}$.

We start with x^0 = [1.0, 1.0].

We initialize parameters for backtracking line search: ρ = 0.8, c_1 = 0.75, and an initial α = 1.0. We then run a while loop until the norm of the gradient is below the tolerance.

Inside the loop, we set d = -g. We then enter an inner while loop for backtracking: while f(x + α*d) > f(x) + c_1 * α * np.dot(g, d), we reduce α by multiplying by ρ (α = α * ρ).

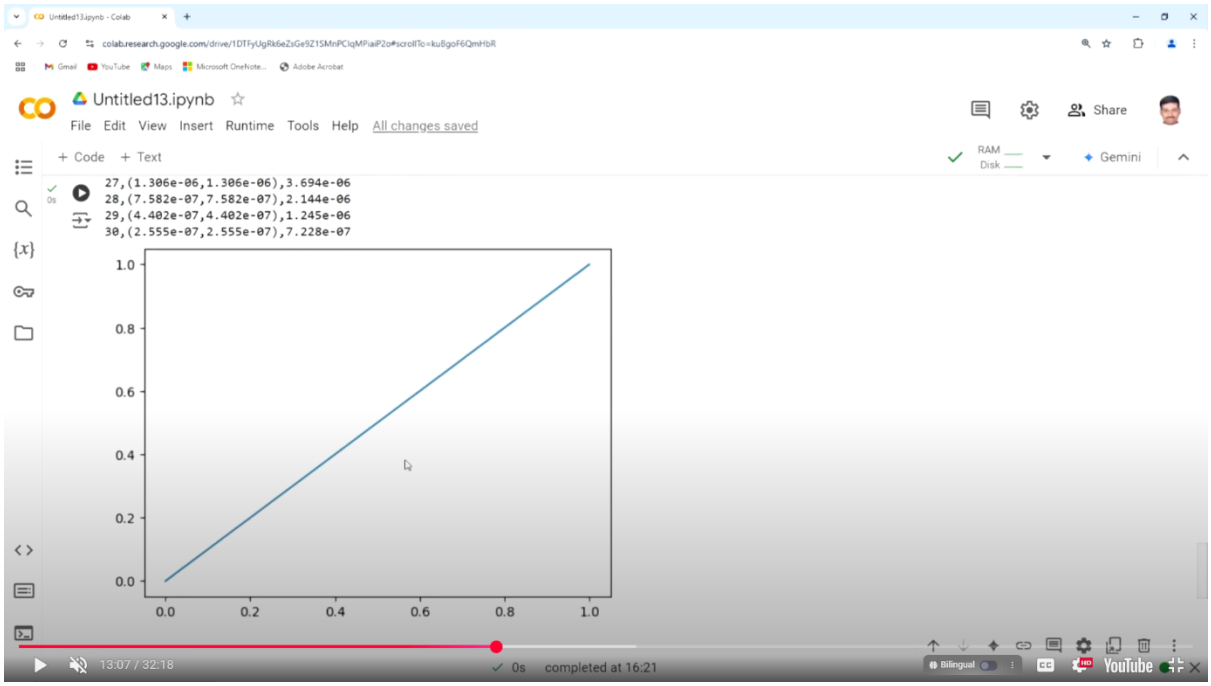
After finding a suitable α, we update x = x + α*d, increment the iteration count, and record the trajectory.

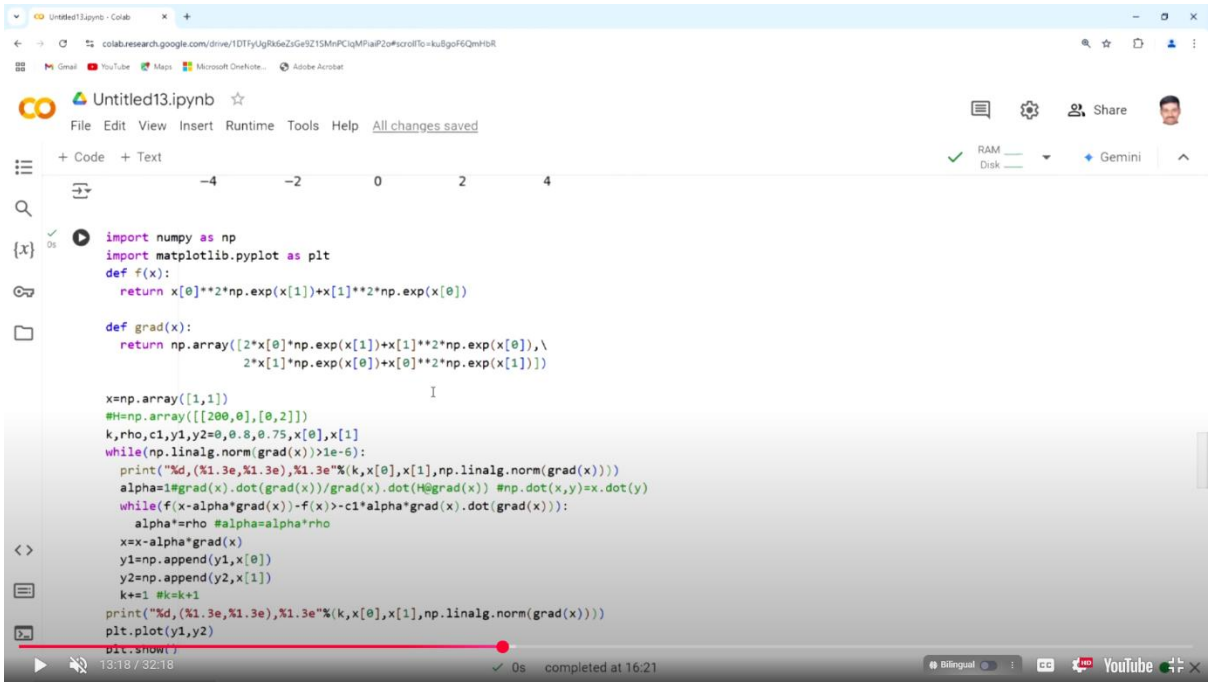
This algorithm converged in 30 steps for this function, following a nearly straight path without zigzags.

This same algorithm can be used for any function by changing the definitions of f(x) and its gradient ∇f(x).

The initial point and the parameters for backtracking (ρ, c_1, initial α) might need adjustment for different problems.

(Refer Slide Time 13:00-13:30)





As an exercise, try the following functions:

1. $f(x_1, x_2) = (1 - x_1)^2 + 100*(x_2 - x_1^2)^2$.
2. This is called the Rosenbrock function. Convergence for any algorithm on this function is typically very slow. The unique minimizer is at (1, 1).
3. Try initial points like (0, 0) or (0.5, 0.5).

2. $f(x_1, x_2) = x_1e^{x_2} + x_2e^{x_1}$. The answer is not immediately obvious, so you can find it by applying the gradient descent algorithm using the Python code.

We will now conclude the discussion on gradient descent. We have identified a significant issue: for a quadratic function where the condition number λ_1/λ_n is much less than 1, gradient descent takes a very long time to converge.

This happens even for simple functions like $100x_1^2 + x_2^2$. How can we improve this?

We will now discuss a different method. Instead of using $d^k = -g^k$, the negative gradient, we will use coordinate directions.

Consider a simple quadratic function $f(x_1, x_2, ..., x_n) = \frac{1}{2} x^THx$, where H is a diagonal matrix $diag(\lambda_1, \lambda_2, ..., \lambda_n)$ with all $\lambda_i > 0$.

The function expands to $(\lambda_1/2)x_1^2 + (\lambda_2/2)x_2^2 + ... + (\lambda_n/2)x_n^2$.

The ratio λ_1/λ_n can be anything; it does not need to be close to 1.

The change from gradient descent is in the choice of descent direction. Instead of $d^k = -g^k$, we choose:

$d^0 = (-1, 0, ..., 0)$ // move in the negative x_1 direction

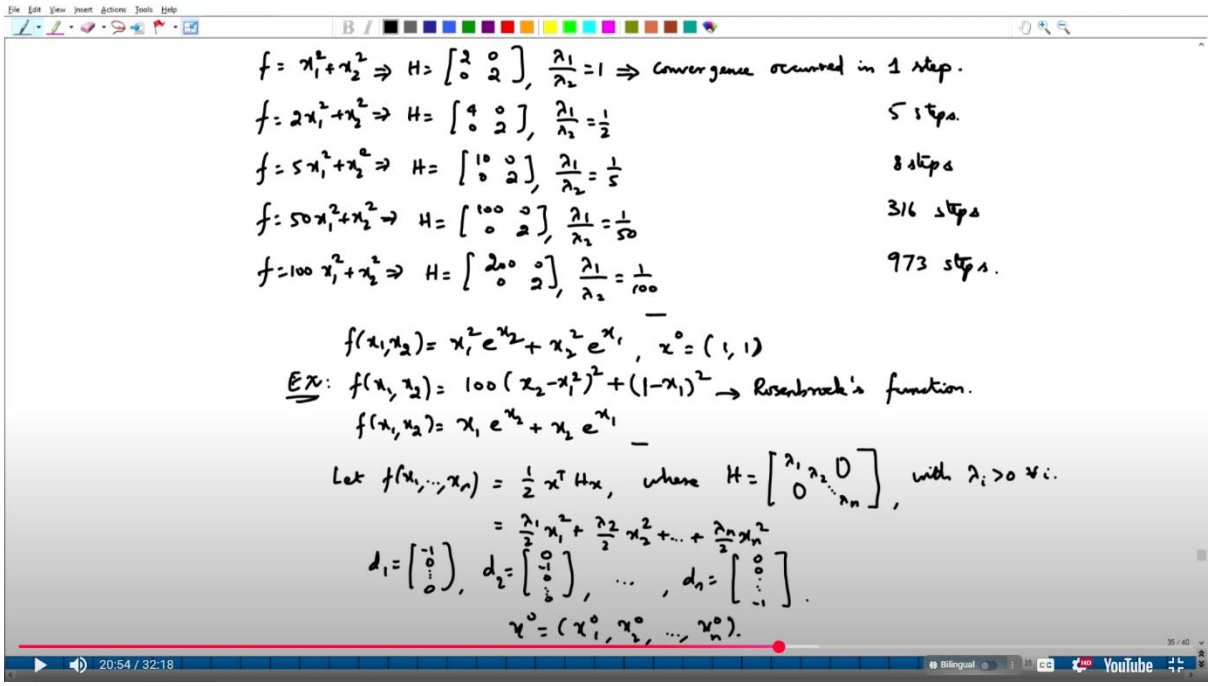
$d^1 = (0, -1, 0, ..., 0)$ // move in the negative x_2 direction

...

$d^{n-1} = (0, 0, ..., -1)$ // move in the negative x_n direction

We cycle through these directions. Let x^0 be any initial point $(x_1^0, x_2^0, ..., x_n^0)$. The minimizer is at (0, 0, ..., 0).

(Refer Slide Time 20:54)



Let us apply the algorithm.

For the first step, $x^1 = x^0 + \alpha^0 d^0$.

The gradient at x^0 is $g^0 = (\lambda_1 x_1^0, \lambda_2 x_2^0, \ldots, \lambda_n x_n^0)$.

We compute α^0 using the exact line search formula for quadratic functions:

$\alpha^0 = - (g^{0T} d^0) / (d^{0T} H d^0)$.

We have $g^{0T} d^0 = (\lambda_1 x_1^0, \lambda_2 x_2^0, \ldots) \cdot (-1, 0, \ldots) = -\lambda_1 x_1^0$.

We have $d^{0T} H d^0 = (-1, 0, \ldots) H (-1, 0, \ldots)^T = \lambda_1$.

Therefore, $\alpha^0 = -(-\lambda_1 x_1^0) / \lambda_1 = x_1^0$.

Thus, $x^1 = x^0 + \alpha^0 d^0 = (x_1^0, x_2^0, \ldots) + (x_1^0)*(-1, 0, \ldots) = (0, x_2^0, x_3^0, \ldots, x_n^0)$.

The first coordinate is set to zero.

For the second step, $x^2 = x^1 + \alpha^1 d^1$. Now $x^1 = (0, x_2^0, \ldots, x_n^0)$.

The gradient at x^1 is $g^1 = (0, \lambda_2 x_2^0, \ldots, \lambda_n x_n^0)$.

We have $d^1 = (0, -1, 0, \ldots, 0)$.

We compute $\alpha^1 = - (g^{1T} d^1) / (d^{1T} H d^1)$.

We have $g^{1T} d^1 = (0, \lambda_2 x_2^0, \ldots) \cdot (0, -1, \ldots) = -\lambda_2 x_2^0$.

We have $d^{1T} H d^1 = (0, -1, \ldots) H (0, -1, \ldots)^T = \lambda_2$. Therefore, $\alpha^1 = -(-\lambda_2 x_2^0) / \lambda_2 = x_2^0$.

Thus, $x^2 = x^1 + \alpha^1 d^1 = (0, x_2^0, \ldots, x_n^0) + (x_2^0)*(0, -1, \ldots) = (0, 0, x_3^0, \ldots, x_n^0)$.

The second coordinate is set to zero.

Continuing this process, after n steps, we will have $x^n = (0, 0, ..., 0)$.

This algorithm is called the coordinate descent algorithm. For example, if we applied the coordinate descent algorithm to minimize $f(x_1, x_2) = 100x_1^2 + x_2^2$, it would converge in exactly two steps because n=2.

This is a significant improvement over the steepest descent algorithm, which took 973 steps for the same function.

However, we must check if this works for other quadratic functions where the Hessian H is not diagonal. Consider the function

$f(x_1, x_2) = x_1^2 + x_1x_2 + x_2^2$.

This can be written as $\frac{1}{2} x^T H x$ where H = [[2, 1], [1, 2]].

This matrix is positive definite but not diagonal. Let us apply the coordinate descent algorithm to see if it converges in two steps.

Let $x^0 = (1, 1)$. For the first step, we use $d^0 = (-1, 0)$.

The gradient at x^0 is $g^0 = Hx^0 = [[2,1],[1,2]] * (1,1) = (3, 3)$.

We compute $\alpha^0 = - (g^{0T} d^0) / (d^{0T} H d^0)$. We have $g^{0T} d^0 = (3, 3) \cdot (-1, 0) = -3$.

We have $d^{0T} H d^0 = (-1, 0) * [[2,1],[1,2]] * (-1, 0)^T$.

First, $H d^0 = [[2,1],[1,2]] * (-1, 0)^T = (-2, -1)$. Then $d^{0T} (H d^0) = (-1, 0) \cdot (-2, -1) = 2$.

Thus, $\alpha^0 = -(-3) / 2 = 3/2$.

Then $x^1 = x^0 + \alpha^0 d^0 = (1, 1) + (3/2)*(-1, 0) = (1 - 3/2, 1) = (-1/2, 1)$.

For the second step, we use $d^1 = (0, -1)$.

The gradient at x^1 is $g^1 = Hx^1 = [[2,1],[1,2]] * (-1/2, 1) = (2*(-1/2)+1*1, 1*(-1/2)+2*1) = (-1+1, -0.5+2) = (0, 1.5) = (0, 3/2)$.

We compute $\alpha^1 = - (g^{1T} d^1) / (d^{1T} H d^1)$. We have $g^{1T} d^1 = (0, 3/2) \cdot (0, -1) = -3/2$.

We have $d^{1T} H d^1 = (0, -1) * [[2,1],[1,2]] * (0, -1)^T$.

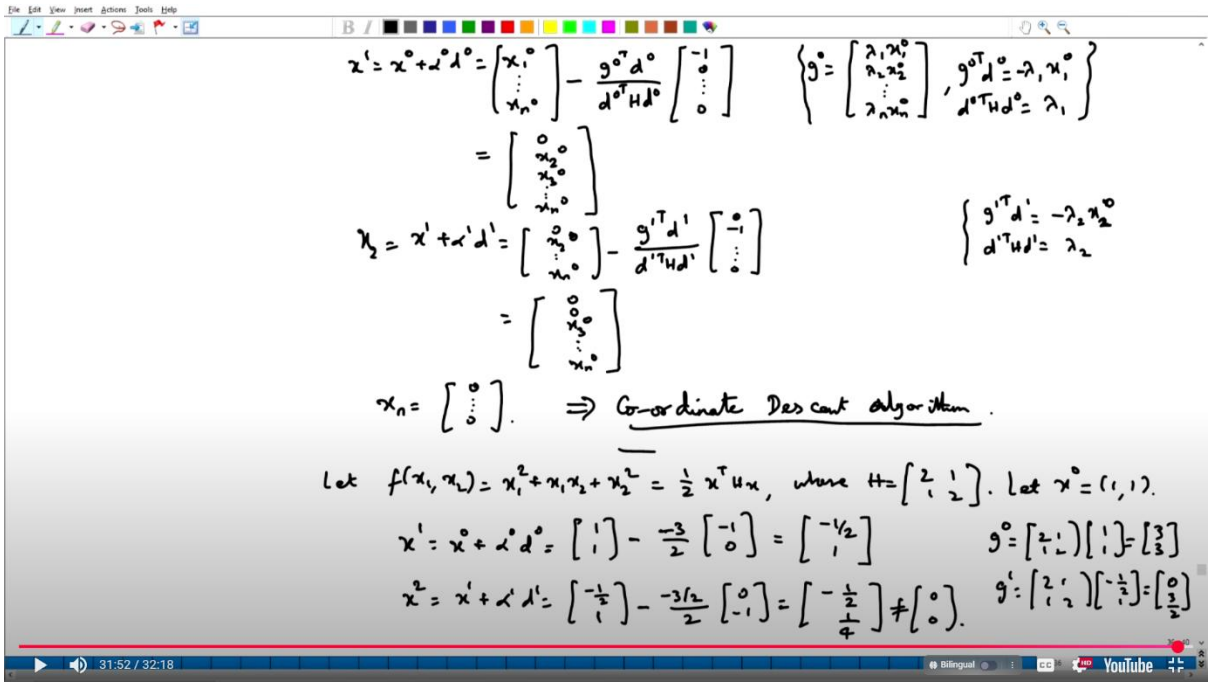
First, $H d^1 = [[2,1],[1,2]] * (0, -1)^T = (-1, -2)$.

Then $d^{1T} (H d^1) = (0, -1) \cdot (-1, -2) = 2$. Thus, $\alpha^1 = -(-3/2) / 2 = (3/2)/2 = 3/4$.

Then $x^2 = x^1 + \alpha^1 d^1 = (-1/2, 1) + (3/4)*(0, -1) = (-1/2, 1 - 3/4) = (-1/2, 1/4)$.

The result (-1/2, 1/4) is not the minimizer (0, 0).

(Refer Slide Time 31:55)



This demonstrates that the coordinate descent algorithm, which converged in n steps for a diagonal Hessian, does not converge to the solution in n steps for a non-diagonal Hessian.

We will explore how to address this issue with non-diagonal matrices in the next lecture. Thank-you