Optimization Algorithms: Theory and Software Implementation

Prof. Thirumulanathan D

Department of Mathematics

Indian Institute of Technology Kanpur

Lecture: 18

Hello everyone. This is the third lecture of week four. In the previous session, we were learning about the gradient descent algorithm and introduced a new method called the coordinate descent algorithm. The coordinate descent algorithm follows the same procedural steps as gradient descent, but with a key difference in the choice of descent direction. Instead of selecting d^k as the negative gradient $-g^k$, we choose the negatives of the coordinate axes directions.

We observed that this method works effectively when the function f(x) is of the form $\frac{1}{2} x^T H x$, where H is a diagonal positive definite matrix. However, we also encountered a counterexample where the algorithm failed to converge to the solution when the Hessian matrix was not diagonal. Specifically, for the function

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

with Hessian

H = [[2, 1], [1, 2]], the algorithm did not reach the minimizer (0, 0) after two steps.

This leads to an important question: can we intelligently choose the descent directions d^0 and d^1 such that convergence to the solution is achieved in n steps for a non-diagonal Hessian? Let us explore this possibility with a more challenging initial point.

Consider the same function

$f(x_1, x_2) = \frac{1}{2} x^T H x$ with H = [[2, 1], [1, 2]] and choose the initial point $x^0 = (4, -5)$.

Instead of the coordinate directions, let us select $d^0 = (-1, -1)$ and $d^1 = (1, -1)$.

We will now compute the iterations.

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

The gradient at x^0 is $g^0 = H x^0 = [[2, 1], [1, 2]] * (4, -5) = (3, -6)$.

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

$g^{0T} d^0 = (3, -6) \cdot (-1, -1) = 3$.

To find $d^{0T} H d^0$, first compute $H d^0 = [[2,1],[1,2]] * (-1, -1)^T = (-3, -3)$.

Then $d^{0T} (H d^0) = (-1, -1) \cdot (-3, -3) = 6$.

Thus, $\alpha^0 = -3 / 6 = -0.5$.

Therefore, $x^1 = (4, -5) + (-0.5)*(-1, -1) = (4.5, -4.5)$.

For the second step, $x^2 = x^1 + \alpha^1 d^1$, with $d^1 = (1, -1)$.

The gradient at x^1 is $g^1 = H\, x^1 = [[2, 1], [1, 2]] * (4.5, -4.5) = (4.5, -4.5)$.

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

$g^{1T} d^1 = (4.5, -4.5) \cdot (1, -1) = 9$.

To find $d^{1T} H\, d^1$, first compute $H\, d^1 = [[2,1],[1,2]] * (1, -1)^T = (1, -1)$.

Then $d^{1T} (H\, d^1) = (1, -1) \cdot (1, -1) = 2$.

Thus, $\alpha^1 = -9 / 2 = -4.5$.

Therefore, $x^2 = (4.5, -4.5) + (-4.5)*(1, -1) = (0, 0)$.

This demonstrates that with the carefully chosen directions $d^0 = (-1, -1)$ and $d^1 = (1, -1)$, the algorithm converges to the solution in exactly two steps.

The critical insight is that these directions are the eigenvectors of the Hessian matrix H. For a symmetric positive definite matrix H, the eigenvectors form an orthonormal basis. Choosing the descent directions as these eigenvectors ensures convergence in n steps for an n-dimensional quadratic function.

The eigenvectors of $H = [[2, 1], [1, 2]]$ are indeed (1, 1) and (1, -1), corresponding to eigenvalues 3 and 1, respectively.

This principle generalizes: for any quadratic function $\frac{1}{2} x^T H\, x$, if we set the descent directions $d^0, d^1, ..., d^{n-1}$ to be the eigenvectors of H, the algorithm will converge to the minimizer in n steps.

A formal proof of this statement relies on the properties of symmetric matrices.

We are analyzing a quadratic function:

$f(x) = \frac{1}{2} x^T H\, x$

where H is an n×n symmetric positive definite matrix.

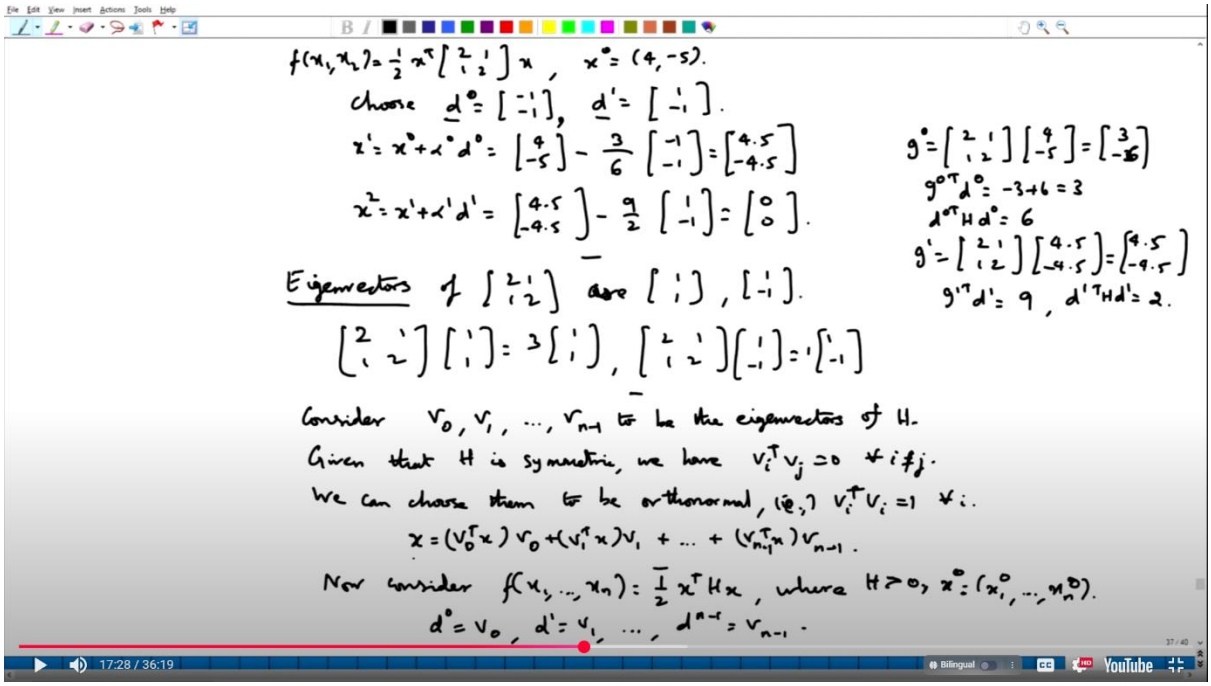
Let $v_0, v_1, ..., v_{n-1}$ be the normalized eigenvectors of H (so $v_i^T v_i = 1$), with corresponding eigenvalues $\lambda_0, \lambda_1, ..., \lambda_{n-1}$.

We choose our search directions to be these eigenvectors:

$d^0 = v_0, d^1 = v_1, ..., d^{n-1} = v_{n-1}$

We start from an arbitrary initial point x^0.

(Refer Slide Time 17:20)



Reformulating the Function

First, we can express any point x in the eigenvector basis:

$x = (v_0^T x)\, v_0 + (v_1^T x)\, v_1 + \cdots + (v_{n-1}^T x)\, v_{n-1}$

The function f(x) can be rewritten using this expansion. Substituting into the original function:

$f(x) = \frac{1}{2}\, x^T H x = \frac{1}{2} [\, (v_0^T x)\, v_0 + (v_1^T x)\, v_1 + \cdots]^T H [\, (v_0^T x)\, v_0 + (v_1^T x)\, v_1 + \cdots]$

Because H is linear and the eigenvectors are orthogonal, this simplifies. Crucially, since each v_i is an eigenvector ($H v_i = \lambda_i v_i$), the expression becomes:

$f(x) = \frac{1}{2} [\, \lambda_0 (v_0^T x)^2 + \lambda_1 (v_1^T x)^2 + \cdots + \lambda_{n-1} (v_{n-1}^T x)^2]$

This reformulation shows that the function f(x) is a weighted sum of squares along the directions of the eigenvectors. The "axes" of this new coordinate system are no longer the standard ones ($x_1, x_2, \ldots, x_n$) but are instead defined by the projections onto the eigenvectors ($v_0^T x, v_1^T x, \ldots, v_{n-1}^T x$).

Example: If H = [[2, 1], [1, 2]], its eigenvectors are proportional to (1, 1) and (1, -1).

The new "axes" for the function are indeed $x_1 + x_2$ and $x_1 - x_2$.

Step 1: Update along $d^0 = v_0$

The next iterate is found by: $x^1 = x^0 + \alpha^0 d^0$. The optimal step size α^0 for this descent direction is given by:

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

where g^0 is the gradient of f at x^0. Since $f(x) = \frac{1}{2} x^T H x$, the gradient is $g^0 = H x^0$.

Substituting $d^0 = v_0$ and $g^0 = H x^0$ into the formula for α^0:

$\alpha^0 = - ((H x^0)^T v_0) / (v_0^T H v_0)$

$= - (x^{0T} H^T v_0) / (v_0^T H v_0)$

Because H is symmetric ($H = H^T$), this simplifies to:

$\alpha^0 = - (x^{0T} H v_0) / (v_0^T H v_0)$

Now, we use the fact that v_0 is an eigenvector of H, so $H v_0 = \lambda_0 v_0$. Substituting this: $\alpha^0 = - (x^{0T} (\lambda_0 v_0)) / (v_0^T (\lambda_0 v_0))$

$= - (\lambda_0 (x^{0T} v_0)) / (\lambda_0 (v_0^T v_0))$

Since the eigenvectors are normalized ($v_0^T v_0 = 1$), the eigenvalues λ_0 cancel out: $\alpha^0 = - (x^{0T} v_0) / 1$

$\alpha^0 = - (x^{0T} v_0)$

Therefore, the update becomes:

$x^1 = x^0 + \alpha^0 d^0 = x^0 + [- (x^{0T} v_0)] v_0 = x^0 - (x^{0T} v_0) v_0$

The term $(x^{0T} v_0) v_0$ is the projection of the vector x^0 onto the axis defined by the eigenvector v_0.

The update $x^1 = x^0$ - (projection of x^0 onto v_0) means we have completely removed (or "zeroed out") the component of x^0 that lies along the v_0 direction.

This process is analogous to how standard coordinate descent removes the component along a canonical axis (like the x_1-axis). Here, we are performing the same operation but along the rotated axes defined by the eigenvectors of H.

This specific choice of search directions leads to convergence in exactly n steps for this quadratic function.

The axes we need to choose are the eigenvectors of the matrix H. This is very similar to performing coordinate descent, but on a rotated set of axes instead of the standard ones. The process follows the same logic as the coordinate descent algorithm. We start with an arbitrary point x^0.

The update is $x^1 = x^0 + \alpha^0 d^0$, where the step is designed to make the first component along the new axis vanish. Here, we set d^0 to be the first eigenvector v_0.

The formula for the step size is $\alpha^0 = (g^{0T} v_0) / (v_0^T H v_0)$.

Since the gradient g^0 is $H x^0$, we substitute to get $\alpha^0 = (x^{0T} H v_0) / (v_0^T H v_0)$.

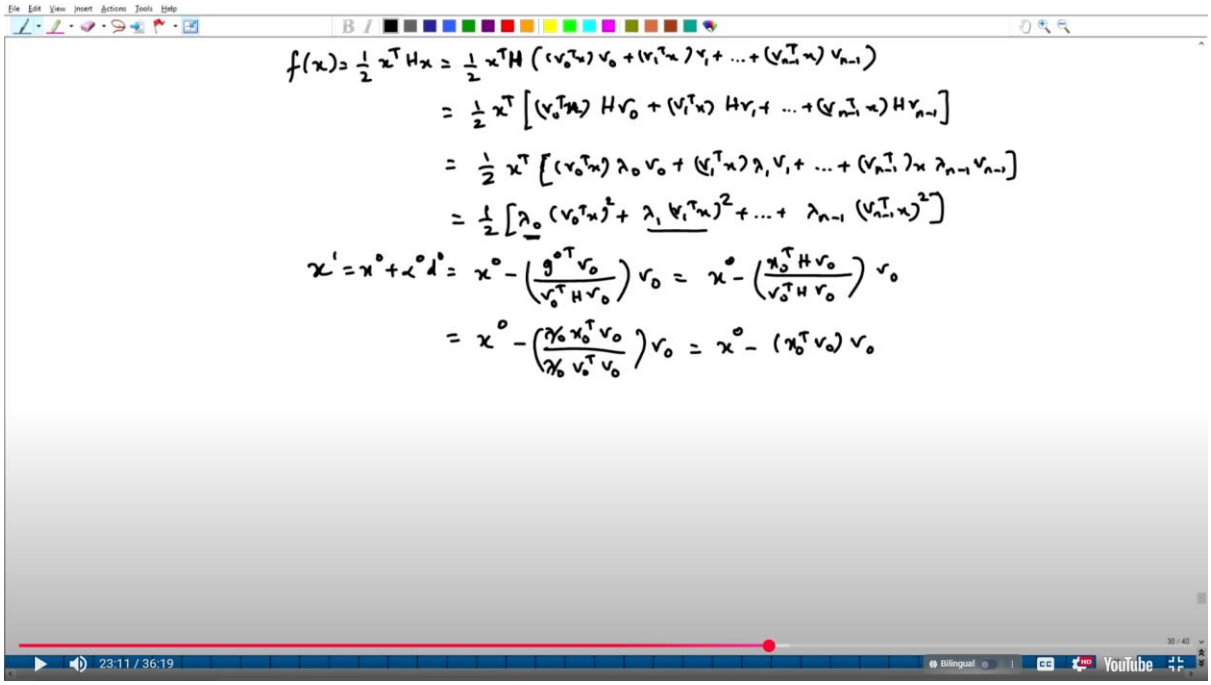
Because v_0 is an eigenvector, $H v_0 = \lambda_0 v_0$.

Substituting this in, the expression becomes $\alpha^0 = (x^{0T} (\lambda_0 v_0)) / (v_0^T (\lambda_0 v_0)) = (\lambda_0 (x^{0T} v_0)) / (\lambda_0 (v_0^T v_0))$.

Assuming the eigenvector is normalized ($v_0^T v_0 = 1$), the λ_0 terms cancel, leaving $\alpha^0 = (x^{0T} v_0)$. For minimization, we use the negative step, so $\alpha^0 = - (x^{0T} v_0)$.

Thus, the update is $x^1 = x^0 - (x^{0T} v_0) v_0$.

(Refer Slide Time 23:11)



This means we have subtracted the projection of x^0 onto the v_0 axis, just like in coordinate descent we subtract the projection onto the x_1 axis.

So, the component along v_0 has vanished. This process repeats for each eigenvector. In the next step, using $d^1 = v_1$, the projection along v_1 is removed, and so on.

After n steps, the projections along all eigenvectors v_0, v_1, ..., v_{n-1} are gone, and we reach the exact solution, which is $x = 0$ for the function $\frac{1}{2} x^T H x$.

This proves that for an n-dimensional quadratic function, using the eigenvectors of H as search directions guarantees convergence in n steps. The problem is practical: finding all eigenvectors for a large matrix is computationally expensive.

It is as expensive as inverting the matrix to solve $Hx = -b$ directly, which defeats the purpose of using an iterative method. Although this method works if we accept the cost, it is not efficient for large problems. This leads to the need for methods that achieve similar results without explicitly computing eigenvectors, such as the conjugate gradient method.

To find eigenvalues and eigenvectors in Python, we can use NumPy. First, generate a random matrix. The command np.random.randint(low, high, size) generates a matrix of random integers.

The low value is inclusive, and high is exclusive, so np.random.randint(-9, 10, size=(3,3)) creates a 3x3 matrix with integers from -9 to 9.

This matrix is not symmetric. To make it symmetric, use $(H + H^T)/2$. Then, to find eigenvalues and eigenvectors, use np.linalg.eig(H), which returns the eigenvalues and a matrix where each column is an eigenvector.

Now you can print the matrix H again to see the result. The original matrix was not symmetric, but after applying $(H + H^T)/2$, it has become symmetric.

It's as simple as that. Now, to find the eigenvalues and eigenvectors of this matrix, you use the command np.linalg.eig(H). This function returns the eigenvalues and eigenvectors. For example, you might see eigenvalues like 10.152, 0.675, and -5.828.

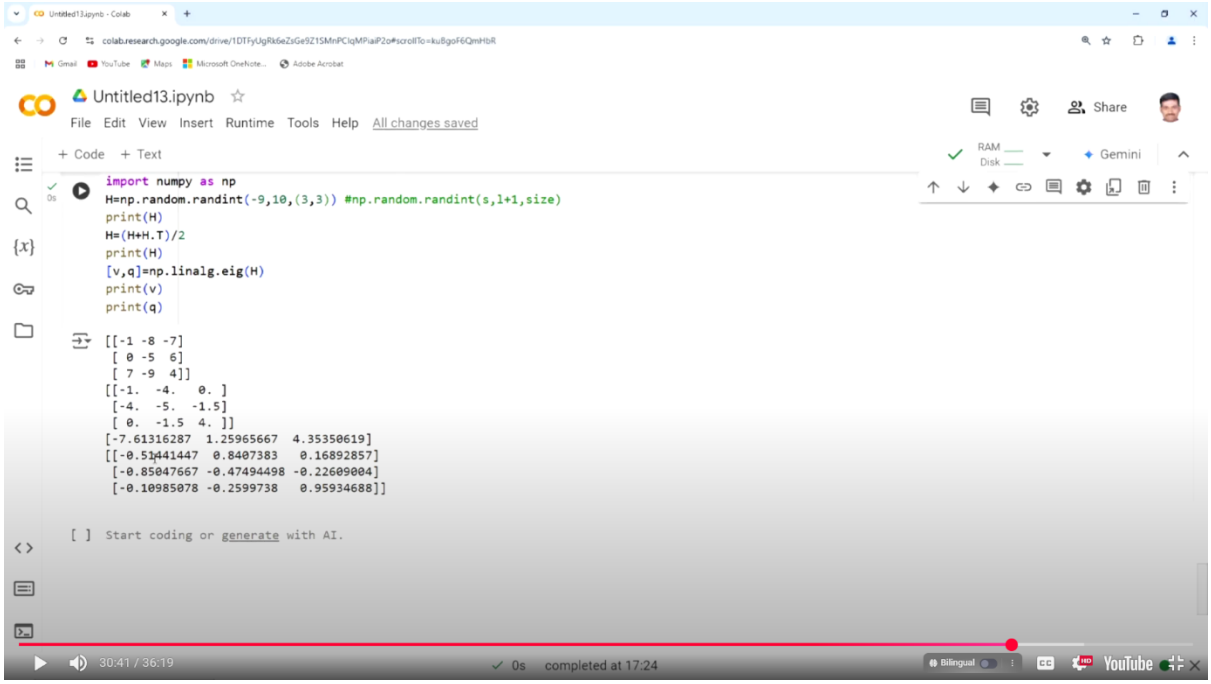
The eigenvectors are returned as columns in a matrix; the first column corresponds to the first eigenvalue (10.152), the second to the second (0.675), and the third to the third (-5.828).

To access them separately, you can write: eigenvalues, eigenvectors = np.linalg.eig(H).

Here, 'eigenvalues' (often stored in a variable like v) is the array of eigenvalues, and 'eigenvectors' (often stored in a variable like q) is the matrix where each column is the eigenvector corresponding to the eigenvalue at the same index.

So, v[0] corresponds to the eigenvector in q[:,0], v[1] to q[:,1], and so on.

(Refer Slide Time 30:40)



Now, to implement the algorithm we discussed, we start with an initial guess for x, say (1, 1, 1). The matrix H must be symmetric and positive definite.

You can verify this by checking that all eigenvalues from np.linalg.eig(H) are positive.

The algorithm proceeds as follows:

initialize k=0.

While the norm of the gradient (H @ x) is greater than a tolerance (e.g., 1e-6),

set the search direction d to the k-th column of the eigenvector matrix (d = eigenvectors[:, k]). Calculate the step size

$\alpha = - (d.T @ H @ x) / (d.T @ H @ d)$.

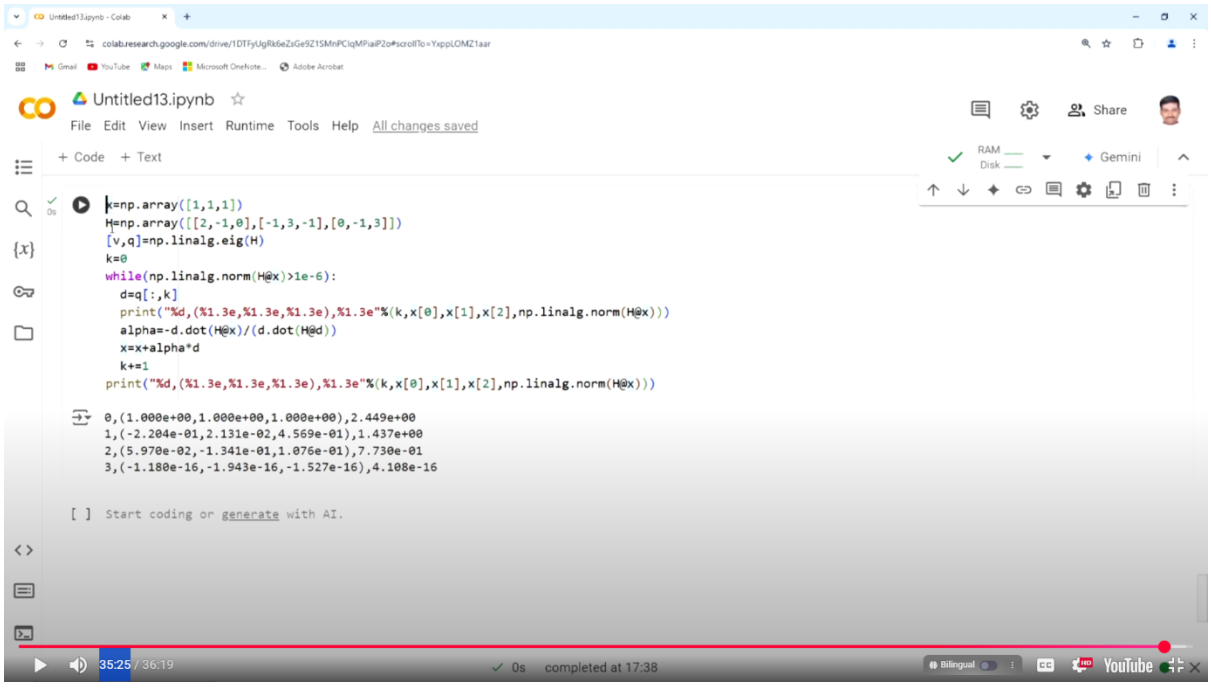
Then update $x = x + \alpha * d$.

Increment k by 1. After the loop, print the final x.

For a 3x3 positive definite matrix, this algorithm will converge in exactly 3 steps, giving the solution x = (0, 0, 0) for the function $\frac{1}{2} x^T H x$.

You can use this code for any dimension by changing the initial x and the matrix H accordingly. This concludes the lecture.

(Refer Slide Time 35:50)



In the next lecture, we will discuss an algorithm that achieves the same result in n steps without explicitly computing the eigenvalues and eigenvectors. Thank you.