Optimization Algorithms: Theory and Software Implementation

Prof. Thirumulanathan D

Department of Mathematics

Institute of IIT Kanpur

Lecture: 20

This is the fifth lecture in the fourth week of the course. In the previous lecture, we began our study of the Fletcher-Reeves formulation of the conjugate gradient algorithm.

The core objective of the conjugate gradient method for minimizing a quadratic function is to generate a set of n search directions, d_0, d_1, ..., d_{n-1}, that are H-conjugate.

This means they must satisfy the condition:

$d_i^T H d_j = 0$ for all $i \neq j$

where H is the positive definite Hessian matrix of the quadratic function.

Note : $g_k = g^k$, $\beta_k = \beta^k$, $d_k = d^k$, $g^{kT} = g^{k^T}$

We derived a recursive formula to construct these directions efficiently without explicitly computing the Hessian's eigenvectors.

The algorithm is initialized with the steepest descent direction:

$d_0 = -g_0$

Subsequent directions are constructed using the recurrence relation:

$d_{k+1} = -g_{k+1} + \beta_k d_k$

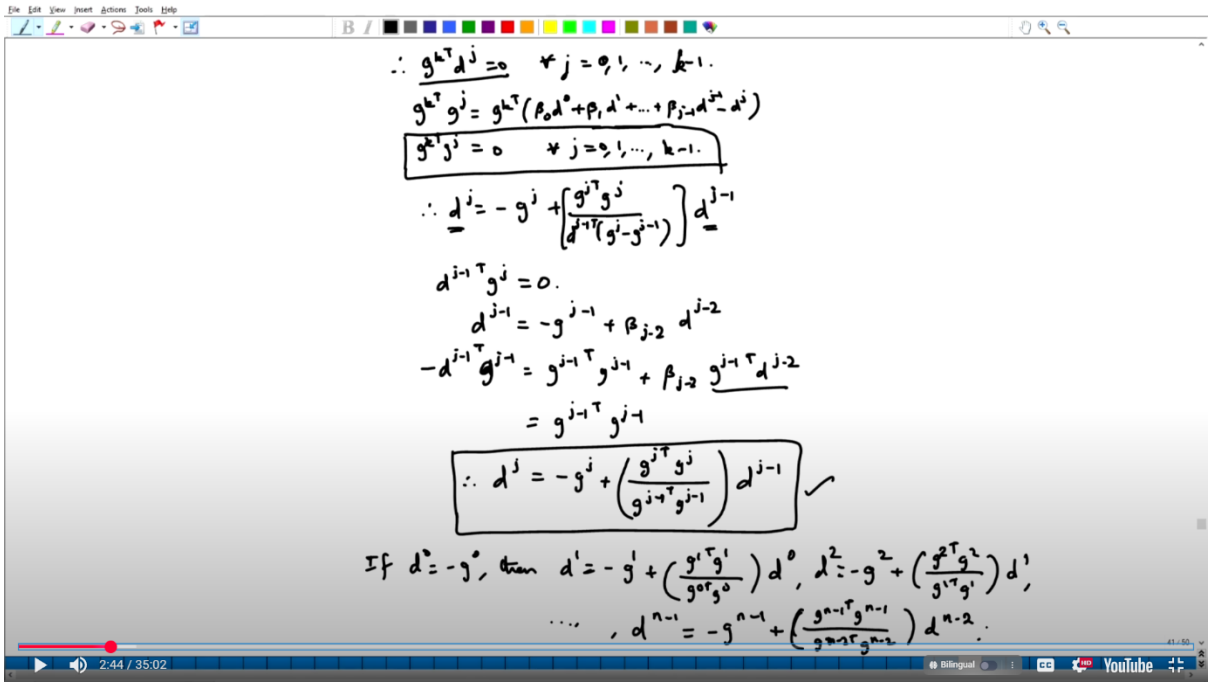
where the scalar β_k is calculated as:

$\beta_k = (g_{k+1}^T g_{k+1}) / (g_k^T g_k)$

This results in the specific updates:

* $d_1 = -g_1 + (g_1^T g_1 / g_0^T g_0) d_0$
* $d_2 = -g_2 + (g_2^T g_2 / g_1^T g_1) d_1$
* ...
* $d_{n-1} = -g_{n-1} + (g_{n-1}^T g_{n-1} / g_{n-2}^T g_{n-2}) d_{n-2}$

(Refer Slide Time 2:44)



The Complete Fletcher-Reeves Algorithm

The full algorithm can be summarized in the following steps:

1. Initialize:
   * Choose an initial point x_0
   * Set a tolerance tol
   * Set iteration counter k = 0
   * Compute the initial gradient $g_0 = \nabla f(x_0)$
   * Set the initial direction $d_0 = -g_0$
2. While $\|g_k\| >$ tol (the gradient norm is greater than the tolerance):
   a. Compute the step size via exact line search (for a quadratic function):

      $\alpha_k = - (g_k^T d_k) / (d_k^T H d_k)$

   b. Update the solution:

      $x_{k+1} = x_k + \alpha_k d_k$

   c. Compute the new gradient:

      $g_{k+1} = \nabla f(x_{k+1})$

   d. Compute the Fletcher-Reeves coefficient:

      $\beta_k = (g_{k+1}^T g_{k+1}) / (g_k^T g_k)$

   e. Update the search direction for the next iteration:

$d_{k+1} = -g_{k+1} + \beta_k d_k$

f. Increment the counter: $k = k + 1$

3. Output: The solution $x^* = x_k$.

The crucial addition to the standard gradient descent procedure is step 2.d, where the new search direction d_{k+1} is constructed as a combination of the new negative gradient $-g_{k+1}$ and the previous direction d_k, scaled by β_k. This step ensures the new direction is H-conjugate to all previous directions, leading to convergence in at most n steps for an n-dimensional quadratic function.

Of course. Here is the text revised for clarity and professionalism, with proper technical notation.

Implementing and Verifying the Fletcher-Reeves Algorithm

We will now implement the Fletcher-Reeves conjugate gradient algorithm in Python. The key claim is that for any *n*-dimensional quadratic function with a positive definite Hessian matrix H, this algorithm converges to the exact solution in at most n steps.

To verify this claim, we will test the algorithm using a 5×5 positive definite matrix. This requires generating a random symmetric positive definite matrix of a specified size.

Generating a Random Positive Definite Matrix

Creating an arbitrary positive definite matrix involves two steps:

1. Generate an arbitrary matrix: We can create a random matrix A with arbitrary elements. This matrix need not be symmetric or positive definite.

2. Ensure symmetry and positive definiteness: A reliable method to construct a symmetric positive definite matrix from an arbitrary matrix A is to compute:

$H = A A^T + I$

where I is the identity matrix.

Proof of Positive Definiteness:

For any non-zero vector x, we must show that $x^T H x > 0$.

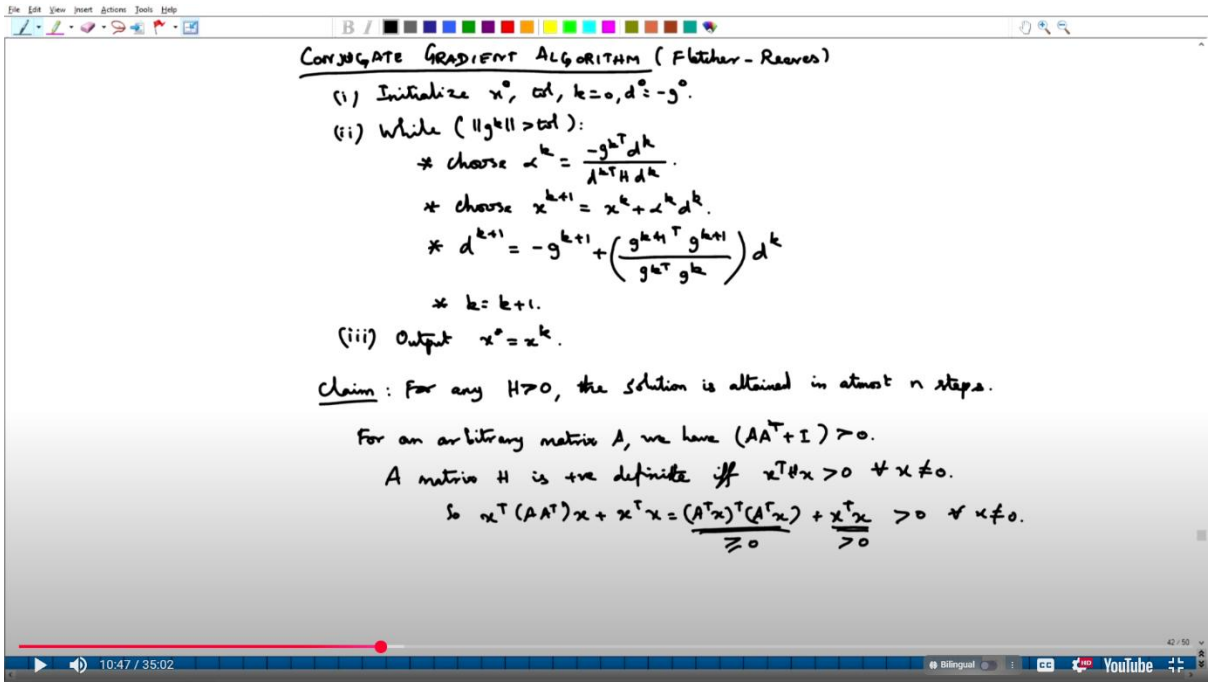
$x^T H x = x^T (A A^T + I) x = x^T A A^T x + x^T I x = (A^T x)^T (A^T x) + x^T x$

This simplifies to the sum of two squared norms:

$x^T H x = \|A^T x\|^2 + \|x\|^2$

Since $x \neq 0$, the term $\|x\|^2 > 0$. The term $\|A^T x\|^2$ is always non-negative. Therefore, the entire expression is strictly greater than zero for all non-zero x, confirming that H is positive definite.

(Refer Slide Time 11:00)



Proof of Symmetry:

The transpose of H is:

$H^T = (A\,A^T + I)^T = (A\,A^T)^T + I^T = (A^T)^T\,A^T + I = A\,A^T + I = H$

Thus, H is symmetric.

This method provides a straightforward way to generate a random symmetric positive definite matrix of any size for testing the algorithm.

Verification Plan

The implementation will proceed as follows:

1. Generate a random 5×5 matrix A.
2. Construct the symmetric positive definite Hessian $H = A\,A^T + I$.
3. Define a quadratic function $f(x) = \frac{1}{2}\,x^T H\,x$ (assuming a linear term of zero for simplicity).
4. Implement the Fletcher-Reeves algorithm as described.
5. Initialize the algorithm from a random starting point x_0.
6. Run the algorithm and verify that the norm of the gradient $\|g_k\|$ converges to zero (within a specified tolerance) in at most 5 iterations.

This experiment will serve as a practical verification of the algorithm's theoretical property of finite convergence for quadratic problems.

Of course. Here is the explanation, structured for clarity and free from conversational fillers.

Generating Random Matrices and the Fletcher-Reeves Implementation

We will now implement the Fletcher-Reeves conjugate gradient algorithm. A key step is generating a random symmetric positive definite matrix H of size 5×5 for testing. The method is to create an arbitrary random matrix A and then compute $H = A A^T + I$, which is guaranteed to be symmetric and positive definite.

Generating Random Numbers in NumPy

To create the matrix A, we use NumPy's random number generation functions. Here are the essential commands:

1. np.random.rand(n): This function generates an array of n random numbers. Each number is drawn from a uniform distribution over the half-open interval [0.0, 1.0). This means every value between 0 and 1 is equally likely to be chosen.

* Example: np.random.rand(100) creates an array of 100 such numbers. The empirical mean and median of a large sample from this distribution will be approximately 0.5.

2. Generating Uniform Numbers in a Custom Interval [a, b]: To generate numbers uniformly in any interval [a, b], you can scale and shift the output of np.random.rand().

* Formula: a + (b - a) * np.random.rand(n)

* How it works: np.random.rand(n) produces values in [0, 1). Multiplying by (b - a) scales the range to [0, b-a). Adding a then shifts the range to [a, b).

3. np.random.randn(n): This function generates an array of n random numbers from a standard normal distribution (Gaussian distribution with a mean of 0 and a standard deviation of 1). This is useful for generating matrices with values centered around zero.

Application: Creating the Positive Definite Matrix H

For the conjugate gradient test, we will generate the matrix A using np.random.randn(5, 5). This creates a 5×5 matrix with random, normally distributed entries. We then construct our test Hessian matrix as:

H = A @ A.T + np.eye(5)

This ensures H is symmetric and positive definite, providing a valid test case to verify that the Fletcher-Reeves algorithm converges to the solution of the quadratic function $f(x) = ½ x^T H x$ in at most 5 steps.

Of course. Here is the text revised for clarity and professionalism, with proper technical notation.

Generating Random Numbers and Matrices in NumPy

To construct a random symmetric positive definite matrix $H = A A^T + I$, we must first generate the arbitrary matrix A. This requires an understanding of NumPy's random number generation functions.

1. Uniform Distribution in [0, 1)

The function np.random.rand(n) generates an array of n random numbers. Each number is an independent sample from a uniform distribution over the interval [0.0, 1.0).

2. Uniform Distribution in a Custom Interval [a, b]

To generate numbers uniformly in any interval [a, b], scale and shift the output of np.random.rand():

a + (b - a) * np.random.rand(n)

This works because:

* np.random.rand(n) generates values in [0, 1).
* Multiplying by (b - a) scales the range to [0, b-a).
* Adding a shifts the range to [a, b).

3. Generating Random Matrices

To generate a matrix of random numbers, specify the dimensions. For example, np.random.rand(5, 5) creates a 5×5 matrix where each element is an independent uniform random variable from [0, 1).

4. Standard Normal (Gaussian) Distribution

The function np.random.randn(n) generates an array of n random numbers from a standard normal distribution (mean $\mu=0$, variance $\sigma^2=1$)

5. General Normal Distribution N(μ, σ²)

To generate numbers from a normal distribution with a specific mean μ and standard deviation σ, use the transformation:

μ + σ * np.random.randn(n)

This works because multiplying by σ scales the variance to σ^2, and adding μ shifts the mean.

6. Generating Gaussian Matrices

A random matrix with elements from N(0, 1) is generated using np.random.randn(5, 5). The resulting matrix will contain both positive and negative values, with most values clustered around 0. The empirical mean and variance of a large sample will converge to 0 and 1, respectively.

This method is ideal for creating the matrix A used to construct the positive definite Hessian H for testing the conjugate gradient algorithm. Using np.random.randn(5, 5) ensures the elements of A are centered around zero, providing a well-conditioned test case.

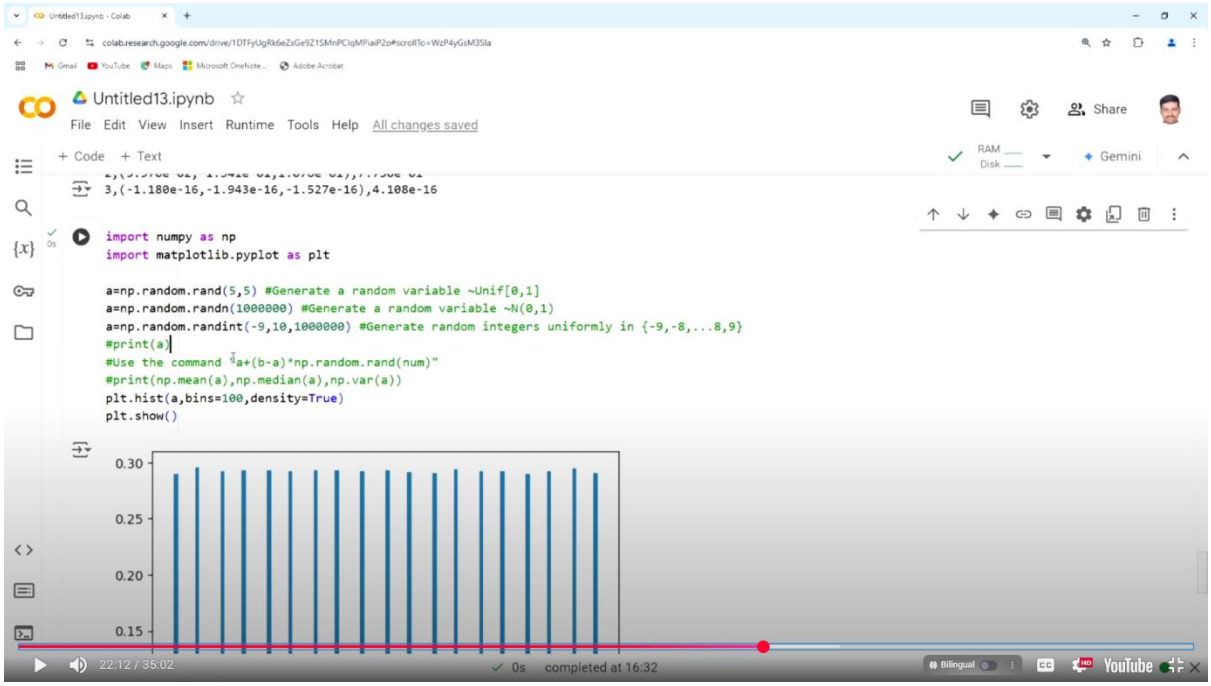
Generating the Test Hessian Matrix H

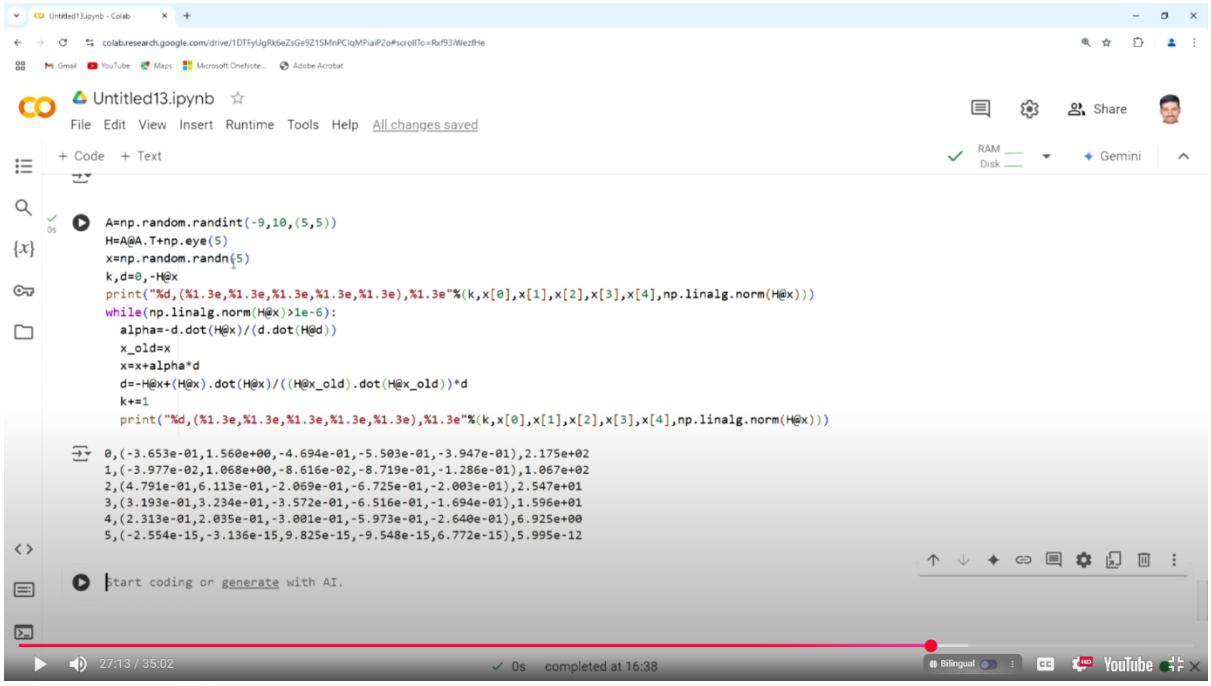
To test the Fletcher-Reeves algorithm, we generate a random 5×5 symmetric positive definite Hessian matrix H. This is achieved by:

1. Creating an arbitrary random matrix A using A = np.random.randint(-9, 10, (5, 5)). This generates a 5×5 matrix with integer elements uniformly distributed between -9 and 9.

2. Constructing the Hessian as H = A @ A.T + np.eye(5). This ensures H is symmetric and positive definite.

(Refer Slide Time 25:30)





2. Algorithm Implementation for Quadratic Functions

The Fletcher-Reeves algorithm for a quadratic function $f(x) = \frac{1}{2} x^T H x$ is implemented as follows:

1. Initialization:
   * Choose a random initial point x_0 (e.g., from a Gaussian distribution).
   * Set a tolerance tol = 1e-6.
   * Set iteration counter k = 0.
   * Compute the initial gradient $g_0 = H x_0$.
   * Set the initial search direction $d_0 = -g_0$.
2. Iteration Loop (while $\|g_k\| > tol$):
   a. Compute the optimal step size (exact line search for quadratic functions):

      $\alpha_k = - (d_k^T g_k) / (d_k^T H d_k)$

   b. Update the solution:

      $x_{k+1} = x_k + \alpha_k d_k$

   c. Compute the new gradient:

      $g_{k+1} = H x_{k+1}$

   d. Compute the Fletcher-Reeves coefficient:

      $\beta_k = (g_{k+1}^T g_{k+1}) / (g_k^T g_k)$

   e. Update the search direction:

      $d_{k+1} = -g_{k+1} + \beta_k d_k$

   f. Increment the counter: k = k + 1
3. Termination: The algorithm converges to the solution x* in at most n = 5 iterations, as verified by the gradient norm $\|g_k\|$ falling below the tolerance.

3. Generalization to Non-Quadratic Functions

The Fletcher-Reeves algorithm can be extended to minimize general non-quadratic functions. The key modifications are:

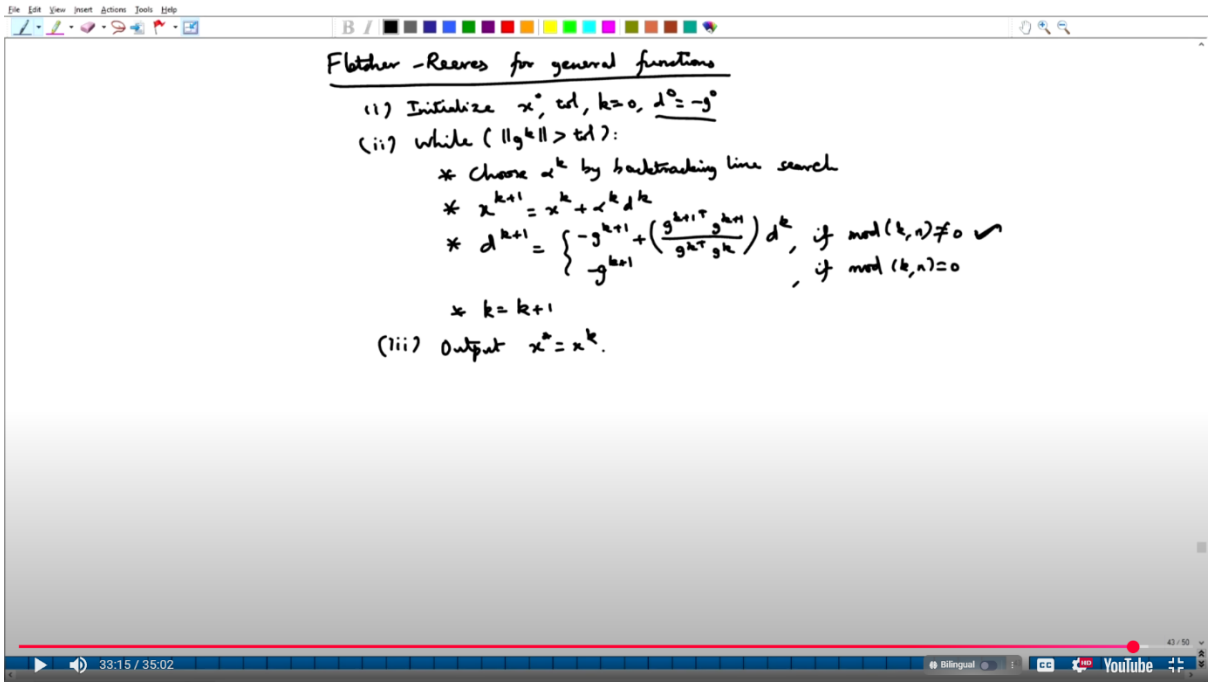
1. Line Search: The exact line search is replaced with a backtracking line search (or another inexact line search method) to find a step size α_k that satisfies the Wolfe conditions, as the function is no longer quadratic.

2. Cycle Restart: For a problem of dimension n, the algorithm is restarted every n iterations to mitigate the accumulation of numerical errors and to refresh the conjugacy property, which is theoretically exact only for quadratic functions. The direction update rule becomes:

* If (k mod n == 0): Restart the cycle by setting $d_k = -g_k$.
* Else: Update the direction using $d_k = -g_k + \beta_k d_{k-1}$, where $\beta_k = (g_k^T g_k) / (g_{k-1}^T g_{k-1})$.

This restarted version, often called the *Fletcher-Reeves method with periodic reset*, is more robust for general nonlinear optimization and typically converges faster than standard gradient descent.

(Refer Slide Time 33:15)



4. Summary of Weeks 3 and 4

Over the past two weeks, we have covered:

* Line Search Methods: Exact and backtracking line search algorithms.
* Descent Direction Methods:
  * Gradient (steepest) descent algorithm.
  * Conjugate gradient algorithms, specifically:
    * The theoretical eigenvector-based method.
    * The computationally efficient Fletcher-Reeves formulation.

The Fletcher-Reeves method is preferred for its efficiency and applicability beyond quadratic problems. Next week, we will begin our study of Newton's method. Thank you.