



THE UNIVERSITY OF
CHICAGO

STAT 310 Lecture 3

Mihai Anitescu

Outline

- Homework Questions? Structure of an optimization code (**EXPAND**)
- Survey
- 2.3 Direct Linear Algebra – Factorization
- 2.4 Sparsity
- 3.1 Failure of vanilla Newton
- 3.2 Line Search Methods
- 3.3 Dealing with Indefinite Matrices
- 3.4 Quasi-Newton Methods

Some thoughts about coding

1. Think ahead of time what functionality your code will have, and define the interface properly
2. If portions of code are similar, try to define a function and “refactorize” (e.g the 3 different iterations).
3. Document your code.
4. Do not write long function files; they are impossible to debug (unless very experienced).

Example Encapsulation

```
function [x,gradNorm]=newtonLikeIteration(functionHandle,xStartPoint,
    iterationType,iterIndex)
% computes one iteration of Newton Like method with a diagonal
% perturbation
% INPUT:    functionHandle:    (pointer) Function defining problem
%           xStartPoint:      (vector) The Starting Point
%           iterIndex:        (integer) The index of the iterate
%           iterationType:    k=1:    Newton's Method
%                               k=2:    Hessian peturbed by identity, I
%                               k=3:    Hessian peturbed by o(iterInd)*I
% OUTPUT:   x:                (vector) Next iteration Point.
```

```
function [xout,iteratesGradNorms]=newtonLikeMethod(functionHandle,
    xStartPoint,iterationType,stopTolerance,maxIterations)
% [xout,iteratesGradNorms]=newtonLikeMethod(functionHandle,xStartPoint,
    iterationType,stopTolerance,maxIterations)
% PURPOSE:  computes the outcome of a Newton-like Method with a perturbed
    diagonal
% INPUT:    functionHandle:    (pointer) Handle to the optimization
%                               problem to be solved
%           xStartPoint:      (vector) The starting point
%           iterationType:    (integer) The type of the diagonal
%                               peturbation to be
%                               used
%           stopTolerance:    (scalar) the gradient size at which the
%                               iteration
%                               will stop.
%           maxIterations:    (integer) The maximum number of iterations
%                               for which
%                               the algorithm should be run
% OUTPUT:   xout:            (vector) The final output
%           iteratesGradNorms: (vector) The norms of the gradients
```

```
[xout,iteratesGradNorms]=newtonLikeMethod(@fenton_wrap,[3 4]',1,1e-12,200)
```

2.3 SOLVING SYSTEMS OF LINEAR EQUATIONS

2.3.1 DIRECT METHODS: THE ESSENTIALS

L and U Matrices

- Lower Triangular Matrix

$$[L] = \begin{bmatrix} l_{11} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ l_{21} & l_{22} & \mathbf{0} & \mathbf{0} \\ l_{31} & l_{32} & l_{33} & \mathbf{0} \\ l_{41} & l_{42} & l_{34} & l_{44} \end{bmatrix}$$

- Upper Triangular Matrix

$$[U] = \begin{bmatrix} u_{11} & u_{12} & u_{13} & u_{13} \\ \mathbf{0} & u_{22} & u_{23} & u_{24} \\ \mathbf{0} & \mathbf{0} & u_{33} & u_{34} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & u_{44} \end{bmatrix}$$

LU Decomposition for $Ax=b$

- **LU decomposition / factorization**

$$[A] \{x\} = [L] [U] \{x\} = \{b\}$$

- **Forward substitution**

$$[L] \{d\} = \{b\}$$

- **Back substitution**

$$[U] \{x\} = \{d\}$$

- **Q: Why might I do this instead of Gaussian elimination?**

Complexity of LU Decomposition

to solve $Ax=b$:

- decompose A into LU -- cost $2n^3/3$ flops
- solve $Ly=b$ for y by forw. substitution -- cost n^2 flops
- solve $Ux=y$ for x by back substitution -- cost n^2 flops

slower alternative:

- compute A^{-1} -- cost $2n^3$ flops
- multiply $x=A^{-1}b$ -- cost $2n^2$ flops

26 Sept 2000 11:45:03 Introduction to Scientific Computing
this costs about 3 times as much as LU

Cholesky LU Factorization

- If $[A]$ is **symmetric** and **positive definite**, it is convenient to use Cholesky decomposition.

$$[A] = [L][L]^T = [U]^T[U]$$

- No pivoting or scaling needed if $[A]$ is symmetric and positive definite (**all eigenvalues are positive**)
- If $[A]$ is not positive definite, the procedure may encounter the square root of a negative number
- Complexity is $\frac{1}{2}$ that of LU (due to symmetry exploitation)

Cholesky LU Factorization

- $[A] = [U]^T[U]$
- Recurrence relations

$$u_{ii} = \sqrt{a_{ii} - \sum_{k=1}^{i-1} u_{ki}^2}$$
$$u_{ij} = \frac{a_{ij} - \sum_{k=1}^{i-1} u_{ki}u_{kj}}{u_{ii}} \quad \text{for } j = i + 1, \dots, n$$

Pivoting in LU Decomposition

- **Still need pivoting in LU decomposition (why?)**
- **Messes up order of $[L]$**
- **What to do?**
- **Need to pivot both $[L]$ and a permutation matrix $[P]$**
- **Initialize $[P]$ as identity matrix and pivot when $[A]$ is pivoted. **Also pivot $[L]$****

LU Decomposition with Pivoting

- **Permutation matrix [P]**
 - permutation of identity matrix [I]
- **Permutation matrix performs “bookkeeping” associated with the row exchanges**
- **Permuted matrix [P] [A]**
- **LU factorization of the permuted matrix**

$$[P] [A] = [L] [U]$$

- **Solution**

$$[L] [U] \{ \mathbf{x} \} = [P] \{ \mathbf{b} \}$$

LU-factorization for real symmetric Indefinite matrix A (constrained optimization has saddle points)

$$LU - \text{factorization} \quad A = \left(\begin{array}{c|c} E & c^T \\ \hline c & B \end{array} \right) = \left(\begin{array}{c|c} I & \\ \hline cE^{-1} & I \end{array} \right) \left(\begin{array}{c|c} E & c^T \\ \hline - & B - cE^{-1}c^T \end{array} \right)$$

$$LDL^T - \text{factorization} \quad A = \left(\begin{array}{c|c} E & c^T \\ \hline c & B \end{array} \right) = \left(\begin{array}{c|c} I & \\ \hline cE^{-1} & I \end{array} \right) \left(\begin{array}{c|c} E & \\ \hline - & B - cE^{-1}c^T \end{array} \right) \left(\begin{array}{c|c} I & E^{-1}c^T \\ \hline & I \end{array} \right)$$

$$\text{where} \quad L = \left(\begin{array}{c|c} I & \\ \hline cE^{-1} & I \end{array} \right) \quad \text{and} \quad L^T = \left(\begin{array}{c|c} I & E^{-T}c^T \\ \hline & I \end{array} \right) = \left(\begin{array}{c|c} I & E^{-1}c^T \\ \hline & I \end{array} \right)$$

Question: 1) If A is not singular, can I be guaranteed to find a nonsingular principal block E after pivoting? Of what size?

2) Why not LU-decomposition?

History of LDL' decomposition: 1x1, 2x2 pivoting

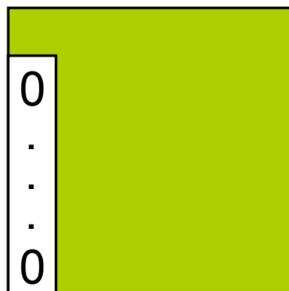
- diagonal pivoting method with **complete** pivoting:
Bunch-Parlett, “Direct methods for solving symmetric indefinite systems of linear equations,” SIAM J. Numer. Anal., v. 8, 1971, pp. 639-655
- diagonal pivoting method with **partial** pivoting:
Bunch-Kaufman, “Some Stable Methods for Calculating Inertia and Solving Symmetric Linear Systems,” Mathematics of Computation, volume 31, number 137, January 1977, page 163-179
- **DEMOS**

2.3.1 DIRECT METHODS: EXTRA DETAILS

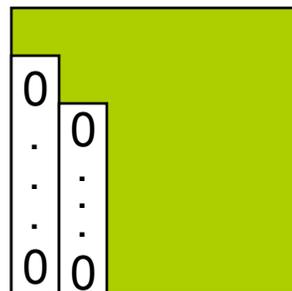
Gaussian Elimination (GE)

- Add multiples of each row to later rows to make A upper triangular
- Solve resulting triangular system $Ux = c$ by substitution

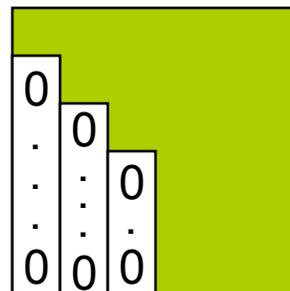
```
... for each column i
... zero it out below the diagonal by adding multiples of row i to later rows
for i = 1 to n-1
  ... for each row j below row i
  for j = i+1 to n
    ... add a multiple of row i to row j
    tmp = A(j,i);
    for k = i to n
      A(j,k) = A(j,k) - (tmp/A(i,i)) * A(i,k)
```



After i=1

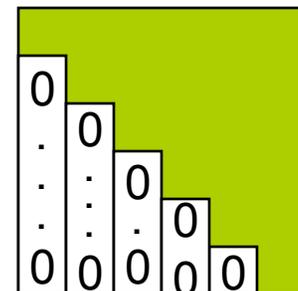


After i=2



After i=3

...



After i=n-1

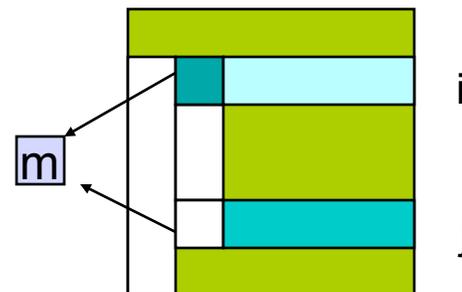
Refine GE (1/5)

- Initial Version

```
... for each column i
... zero it out below the diagonal by adding multiples of row i to later rows
for i = 1 to n-1
  ... for each row j below row i
  for j = i+1 to n
    ... add a multiple of row i to row j
    tmp = A(j,i);
    for k = i to n
      A(j,k) = A(j,k) - (tmp/A(i,i)) * A(i,k)
```

Remove computation of constant $\text{tmp}/A(i,i)$ from inner loop.

```
for i = 1 to n-1
  for j = i+1 to n
    m = A(j,i)/A(i,i)
    for k = i to n
      A(j,k) = A(j,k) - m * A(i,k)
```



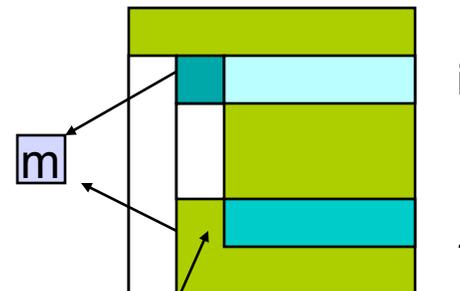
Refine GE (2/5)

- Last version

```
for i = 1 to n-1
  for j = i+1 to n
    m = A(j,i)/A(i,i)
    for k = i to n
      A(j,k) = A(j,k) - m * A(i,k)
```

- Don't compute what we already know:
zeros below diagonal in column i

```
for i = 1 to n-1
  for j = i+1 to n
    m = A(j,i)/A(i,i)
    for k = i+1 to n
      A(j,k) = A(j,k) - m * A(i,k)
```



Do not compute zeros

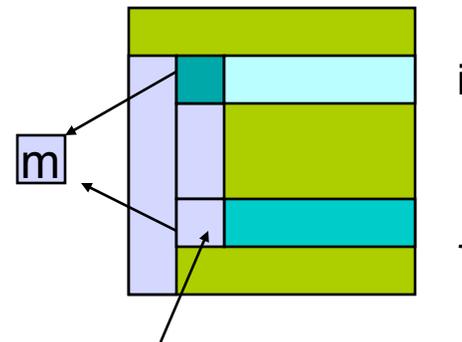
Refine GE Algorithm (3/5)

- Last version

```
for i = 1 to n-1
  for j = i+1 to n
    m = A(j,i)/A(i,i)
    for k = i+1 to n
      A(j,k) = A(j,k) - m * A(i,k)
```

- Store multipliers m below diagonal in zeroed entries for later use

```
for i = 1 to n-1
  for j = i+1 to n
     $A(j,i) = A(j,i)/A(i,i)$ 
    for k = i+1 to n
       $A(j,k) = A(j,k) - A(j,i) * A(i,k)$ 
```



Store m here

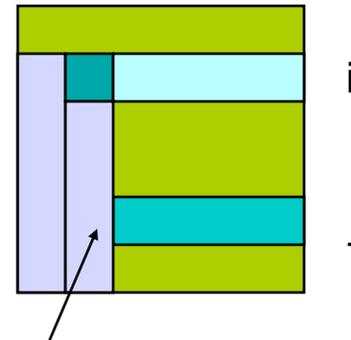
Refine GE Algorithm (4/5)

- Last version

```
for i = 1 to n-1
  for j = i+1 to n
    A(j,i) = A(j,i)/A(i,i)
    for k = i+1 to n
      A(j,k) = A(j,k) - A(j,i) * A(i,k)
```

- Split Loop

```
for i = 1 to n-1
  for j = i+1 to n
    A(j,i) = A(j,i)/A(i,i)
  for j = i+1 to n
    for k = i+1 to n
      A(j,k) = A(j,k) - A(j,i) * A(i,k)
```



Store all m's here before updating rest of matrix

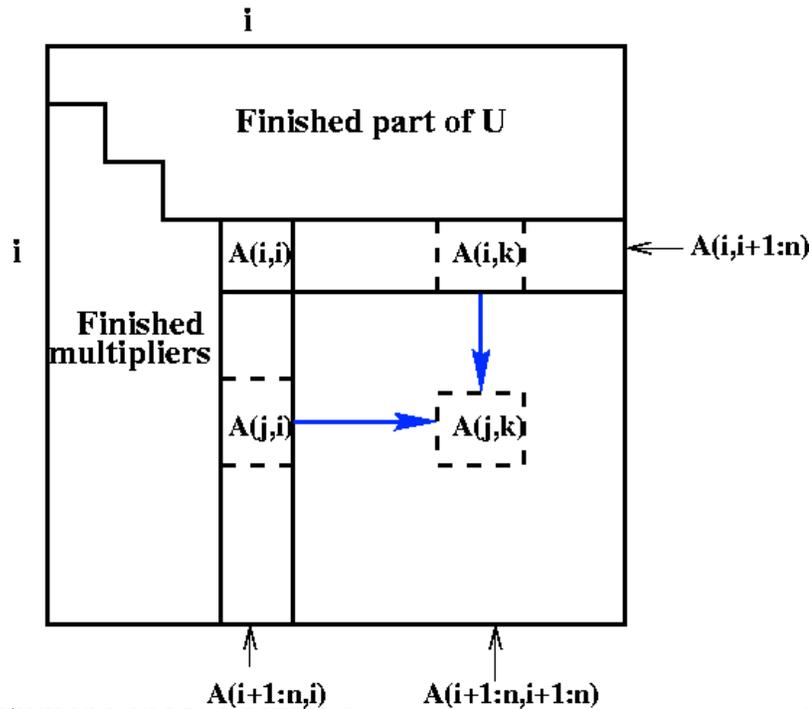
Refine GE Algorithm (5/5)

- Last version

```

for i = 1 to n-1
  for j = i+1 to n
    A(j,i) = A(j,i)/A(i,i)
  for j = i+1 to n
    for k = i+1 to n
      A(j,k) = A(j,k) - A(j,i) * A(i,k)
  
```

Work at step i of Gaussian Elimination



```

for i = 1 to n-1
  A(i+1:n,i) = A(i+1:n,i) * ( 1 / A(i,i) )
  ... BLAS 1 (scale a vector)
  A(i+1:n,i+1:n) = A(i+1:n, i+1:n )
  - A(i+1:n, i) * A(i, i+1:n)
  ... BLAS 2 (rank-1 update)
  
```

What GE really computes

for $i = 1$ to $n-1$

$A(i+1:n,i) = A(i+1:n,i) / A(i,i)$... BLAS 1 (scale a vector)

$A(i+1:n,i+1:n) = A(i+1:n, i+1:n) - A(i+1:n, i) * A(i, i+1:n)$... BLAS 2 (rank-1 update)

- Call the strictly lower triangular matrix of multipliers M , and let $L = I+M$
- Call the upper triangle of the final matrix U
- *Lemma (LU Factorization)*: If the above algorithm terminates (does not divide by zero) then $A = L*U$

What GE really computes

for $i = 1$ to $n-1$

$A(i+1:n,i) = A(i+1:n,i) / A(i,i)$... BLAS 1 (scale a vector)

$A(i+1:n,i+1:n) = A(i+1:n, i+1:n) - A(i+1:n, i) * A(i, i+1:n)$... BLAS 2 (rank-1 update)

- Solving $A^*x=b$ using GE
 - Factorize $A = L^*U$ using GE (cost = $2/3 n^3$ flops)
 - Solve $L^*y = b$ for y , using substitution (cost = n^2 flops)
 - Solve $U^*x = y$ for x , using substitution (cost = n^2 flops)
- Thus $A^*x = (L^*U)^*x = L^*(U^*x) = L^*y = b$ as

desired

Forward Substitution

- Once $[L]$ is formed, we can use forward substitution instead of forward elimination for different $\{b\}$'s

$$[L]\{d\} = \begin{bmatrix} l_{11} & 0 & 0 & 0 \\ l_{21} & l_{22} & 0 & 0 \\ l_{31} & l_{32} & l_{33} & 0 \\ l_{41} & l_{42} & l_{43} & l_{44} \end{bmatrix} \begin{Bmatrix} d_1 \\ d_2 \\ d_3 \\ d_4 \end{Bmatrix} = \begin{Bmatrix} l_{11}d_1 \\ l_{21}d_1 + l_{22}d_2 \\ l_{31}d_1 + l_{32}d_2 + l_{33}d_3 \\ l_{41}d_1 + l_{42}d_2 + l_{43}d_3 + l_{44}d_4 \end{Bmatrix} = \begin{Bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{Bmatrix}$$

$$\Rightarrow \begin{cases} d_1 = b_1 / l_{11} \\ d_2 = (b_2 - l_{21}d_1) / l_{22} \\ d_3 = (b_3 - l_{31}d_1 - l_{32}d_2) / l_{33} \\ d_4 = (b_4 - l_{41}d_1 - l_{42}d_2 - l_{43}d_3) / l_{44} \end{cases}$$

Very efficient for large matrices !

Back Substitution

$$[U]\{x\} = \begin{bmatrix} u_{11} & u_{12} & u_{13} & u_{14} \\ 0 & u_{22} & u_{23} & u_{24} \\ 0 & 0 & u_{33} & u_{34} \\ 0 & 0 & 0 & u_{44} \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{Bmatrix} = \begin{Bmatrix} u_{11}x_1 + u_{12}x_2 + u_{13}x_3 + u_{14}x_4 \\ u_{22}x_2 + u_{23}x_3 + u_{24}x_4 \\ u_{33}x_3 + u_{34}x_4 \\ u_{44}x_4 \end{Bmatrix} = \begin{Bmatrix} d_1 \\ d_2 \\ d_3 \\ d_4 \end{Bmatrix}$$

$$\Rightarrow \begin{cases} x_4 = d_4 / u_{44} \\ x_3 = (d_3 - u_{34}x_4) / u_{33} \\ x_2 = (d_2 - u_{23}x_3 - u_{24}x_4) / u_{22} \\ x_1 = (d_1 - u_{12}x_2 - u_{13}x_3 - u_{14}x_4) / u_{11} \end{cases}$$

**Identical to
Gauss
elimination**

(S) Forward Substitution

Example:

$$[L]\{d\} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -1 & 1 & 0 & 0 \\ 0 & 1/2 & 1 & 0 \\ 6 & 1 & 14 & 1 \end{bmatrix} \begin{Bmatrix} d_1 \\ d_2 \\ d_3 \\ d_4 \end{Bmatrix} = \begin{Bmatrix} 1 \\ -1 \\ 2 \\ 1 \end{Bmatrix} = \{b\}$$

$$\begin{cases} d_1 = 1 \\ d_2 = -1 + d_1 = -1 + 1 = 0 \\ d_3 = 2 - (1/2)d_2 = 2 \\ d_4 = 1 - 6d_1 + d_2 - 14d_3 = 1 - 6 - 14(2) = -33 \end{cases}$$

$$\{d\} = \begin{Bmatrix} 1 \\ 0 \\ 2 \\ -33 \end{Bmatrix}$$

(S) Back-Substitution

$$[U]\{x\} = \begin{bmatrix} 1 & 0 & 2 & 3 \\ 0 & 2 & 4 & 0 \\ 0 & 0 & -1 & 4 \\ 0 & 0 & 0 & -70 \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{Bmatrix} = \begin{Bmatrix} 1 \\ 0 \\ 2 \\ -33 \end{Bmatrix}$$

$$\begin{cases} x_4 = -33 / -70 = 33/70 \\ x_3 = 4x_4 - 2 = -4/35 \\ x_2 = -2x_3 = 8/35 \\ x_1 = 1 - 2x_3 - 3x_4 = -13/70 \end{cases}$$

$$\{x\} = \begin{bmatrix} -13/70 \\ 8/35 \\ -4/35 \\ 33/70 \end{bmatrix}$$

Forward and Back Substitutions

- **Forward-substitution**

$$d_i = b_i - \sum_{j=1}^{i-1} l_{ij} d_j \quad \text{for } i = 1, 2, \dots, n$$

- **Back-substitution (identical to Gauss elimination)**

$$x_n = d_n / a_{nn}$$

$$x_i = \frac{d_i - \sum_{j=i+1}^n u_{ij} x_j}{u_{ii}} \quad \text{for } i = n-1, n-2, \dots, 3, 2, 1$$

Example: Cholesky LU

$$[A] = \begin{bmatrix} 9 & -6 & 12 & -3 \\ -6 & 5 & -9 & 2 \\ 12 & -9 & 21 & 0 \\ -3 & 2 & 0 & 6 \end{bmatrix} = \begin{bmatrix} u_{11}^2 & u_{11}u_{12} & u_{11}u_{13} & u_{11}u_{14} \\ u_{11}u_{12} & u_{12}^2 + u_{22}^2 & u_{13}u_{12} + u_{23}u_{22} & u_{14}u_{12} + u_{24}u_{22} \\ u_{11}u_{13} & u_{13}u_{12} + u_{23}u_{22} & u_{13}^2 + u_{23}^2 + u_{33}^2 & u_{14}u_{13} + u_{24}u_{23} + u_{34}u_{33} \\ u_{11}u_{14} & u_{14}u_{12} + u_{24}u_{22} & u_{14}u_{13} + u_{24}u_{23} + u_{34}u_{33} & u_{14}^2 + u_{24}^2 + u_{34}^2 + u_{44}^2 \end{bmatrix}$$

1st column/row: $u_{11} = \sqrt{9} = 3$; $u_{12} = -6/3 = -2$; $u_{13} = 12/3 = 4$; $u_{14} = -3/3 = -1$

2nd column/row: $u_{12}^2 + u_{22}^2 = 5$; $u_{13}u_{12} + u_{23}u_{22} = -9$; $u_{14}u_{12} + u_{24}u_{22} = 2$

$$u_{22} = \sqrt{5 - (-2)^2} = 1; \quad u_{23} = (-9 - 4(-2)) / 1 = -1; \quad u_{24} = (2 - (-1)(-2)) / 1 = 0$$

3rd column/row: $u_{13}^2 + u_{23}^2 + u_{33}^2 = 21$; $u_{13}u_{14} + u_{23}u_{24} + u_{33}u_{34} = 0$

$$u_{33} = \sqrt{21 - (4)^2 - (-1)^2} = 2; \quad u_{34} = (0 - (-1)(4) - (0)(-1)) / 2 = 2$$

4th row: $u_{14}^2 + u_{24}^2 + u_{34}^2 + u_{44}^2 = 6 \Rightarrow u_{44} = \sqrt{6 - (-1)^2 - (0)^2 - (2)^2} = 1$

2.4 COMPLEXITY OF LINEAR ALGEBRA; SPARSITY

Complexity of LU Decomposition

to solve $Ax=b$:

- decompose A into LU -- cost $2n^3/3$ flops
- solve $Ly=b$ for y by forw. substitution -- cost n^2 flops
- solve $Ux=y$ for x by back substitution -- cost n^2 flops

slower alternative:

- compute A^{-1} -- cost $2n^3$ flops
- multiply $x=A^{-1}b$ -- cost $2n^2$ flops

26 Sept 2000 11:58:59 AM Introduction to Scientific Computing
this costs about 3 times as much as LU

Complexity of linear algebra

lesson:

- if you see A^{-1} in a formula, read it as “solve a system”, not “invert a matrix”

Cholesky factorization -- cost $n^3/3$ flops

LDL' factorization -- cost $n^3/3$ flops

26 Sept. 2000 15-859B - Introduction to Scientific Computing
Q: What is the cost of Cramer's rule (roughly)?

Sparse Linear Algebra

- Suppose you are applying matrix-vector multiply and the matrix has lots of zero elements
 - Computation cost? Space requirements?
- General sparse matrix representation concepts
 - Primarily only represent the nonzero data values (nnz)
 - Auxiliary data structures describe placement of nonzeros in “dense matrix”
- And ***MAYBE*** LU or Cholesky can be done in $O(\text{nnz})$, so not as bad as $(O(n^3))$; since very oftentimes $\text{nnz} = O(n)$

Sparse Linear Algebra

- Because of its phenomenal computational and storage savings potential, sparse linear algebra is a huge research topic.
- VERY difficult to develop.
- Matlab implements sparse linear algebra based on i,j,s format.
- DEMO
- Conclusion: Maybe I can SCALE well ... Solve $O(10^{12})$ problems in $O(10^{12})$.

SUMMARY SECTION 2

- The heaviest components of numerical software are Numerical differentiation (AD/DIVDIFF) and linear algebra.
- Factorization is always preferable to direct (Gaussian) elimination.
- Keeping track of sparsity in linear algebra can enormously improve performance.



THE UNIVERSITY OF
CHICAGO

Section 3: Line Search Methods

Mihai Anitescu STAT 310

Reference: Chapter 3 in Nocedal and
Wright.

3.1 FAILURE OF NEWTON METHODS

Problem definition

$$\min f(x)$$

$$f : R^n \rightarrow R$$

- continuously differentiable
- gradient is available
- Hessian is unavailable

Necessary optimality conditions: $\nabla f(x^*) = 0$

Sufficient optimality conditions: $\nabla^2 f(x^*) \succ 0$

DEMO

- Algorithm: Newton.
- Note: not only does the algorithm not converge, the function values go to infinity.
- So we should have known ahead of time we should have done something else earlier.

Ways of enforcing that thinks do not blow up or wander

- 1. Line-search methods.
 - Make a “guess” of a good direction.
 - Make good progress along that direction. At least know you will decrease f .
- 2. Trust region model.
 - Create a quadratic model of the function.
 - Define a region where we “believe”—”trust” the model and find a “good” point in that “region”.
 - If at that point the model is far from f , less trust—smaller region, if not, more –larger region.

3.2 LINE SEARCH METHODS

3.2.1 LINE SEARCH METHODS: ESSENTIALS

Line Search Methods Idea:

- At the current point x_k find a “Newton-like” direction d_k
- Along that direction d_k do 1-dimensional minimization (**simpler than over whole space**)
$$x_{k+1} \approx \arg \min_{\alpha} f(x_k + \alpha d_k)$$
- Because the line search always decreases f , we will have an accumulation point (cannot diverge if bounded below) – unlike Newton proper

Descent Principle

- Descent Principle: Carry Out a one-Dimensional Search Along a Line where I will decrease the function.

$$g(\alpha) = f(x_k + \alpha p_k) \text{ for } \nabla f(x_k)' p_k < 0$$

- If this happens, there exists an alpha (why?) such that.

$$f(x_k + \alpha p_k) < f(x_k)$$

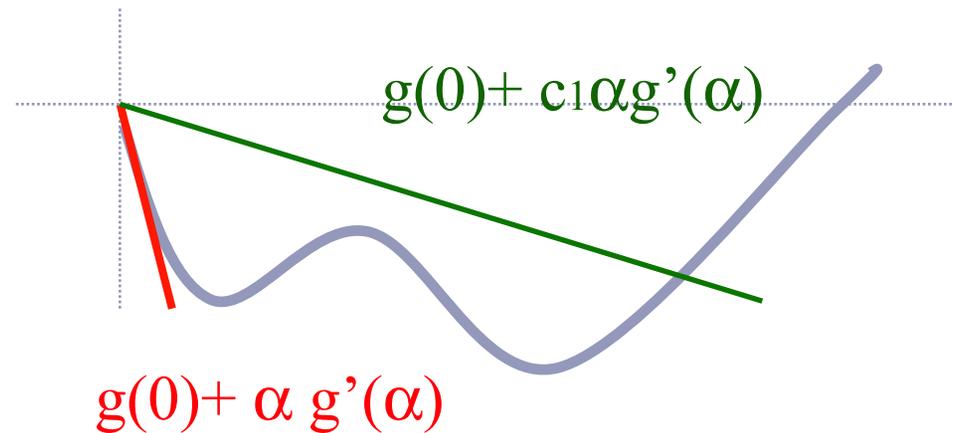
- So I will keep making progress.
- Typical choice (why)? $B_k p_k = -\nabla f(x_k)$; $B_k \succ 0$
- Newton may need to be modified (why?)

Line Search-Armijo

- I cannot accept just about ANY decrease, for I may NEVER converge (why , example of spurious convergence).
- IDEA: Accept only decreases PROPORTIONAL TO THE SQUARE OF GRADIENT. Then I have to converge (since process stops only when gradient is 0).
- Example: Armijo Rule. It uses the concept of BACKTRACKING.

$$f(x_k) - f(x_k + \beta^m \tau_k d_k) \geq -\rho \beta^m \tau_k \nabla f(x_k)^T d_k$$

$\beta \in (0,1) \quad \rho \in (0,1/2)$



Some Theory

Global Convergence:

Let f be twice continuously differentiable on an open set \mathcal{D} , and assume that the starting point x_0 of Algorithm 3.2 is such that the level set $\mathcal{L} = \{x \in \mathcal{D} : f(x) \leq f(x_0)\}$ is compact. Then if the bounded modified factorization property holds, we have that


$$\lim_{k \rightarrow \infty} \nabla f(x_k) = 0.$$

$$\kappa(B_k) = \|B_k\| \|B_k^{-1}\| \leq C, \quad \text{some } C > 0 \text{ and all } k = 0, 1, 2, \dots$$

Fast Convergence:

Newton is accepted by LS

Suppose that f is twice differentiable and that the Hessian $\nabla^2 f(x)$ is Lipschitz continuous (see (A.42)) in a neighborhood of a solution x^* at which the sufficient conditions (Theorem 2.4) are satisfied. Consider the iteration $x_{k+1} = x_k + p_k$, where p_k is given by (3.30). Then

- (i) if the starting point x_0 is sufficiently close to x^* , the sequence of iterates converges to x^* ;
- (ii) the rate of convergence of $\{x_k\}$ is quadratic; and
- (iii) the sequence of gradient norms $\{\|\nabla f_k\|\}$ converges quadratically to zero.

Extensions

- Line Search Refinements:
 - Use interpolation
 - Wolfe and Goldshtein rule
- Other optimization approaches
 - Steepest descent,
 - CG

3.2.2 LINE SEARCH METHODS: EXTRAS.

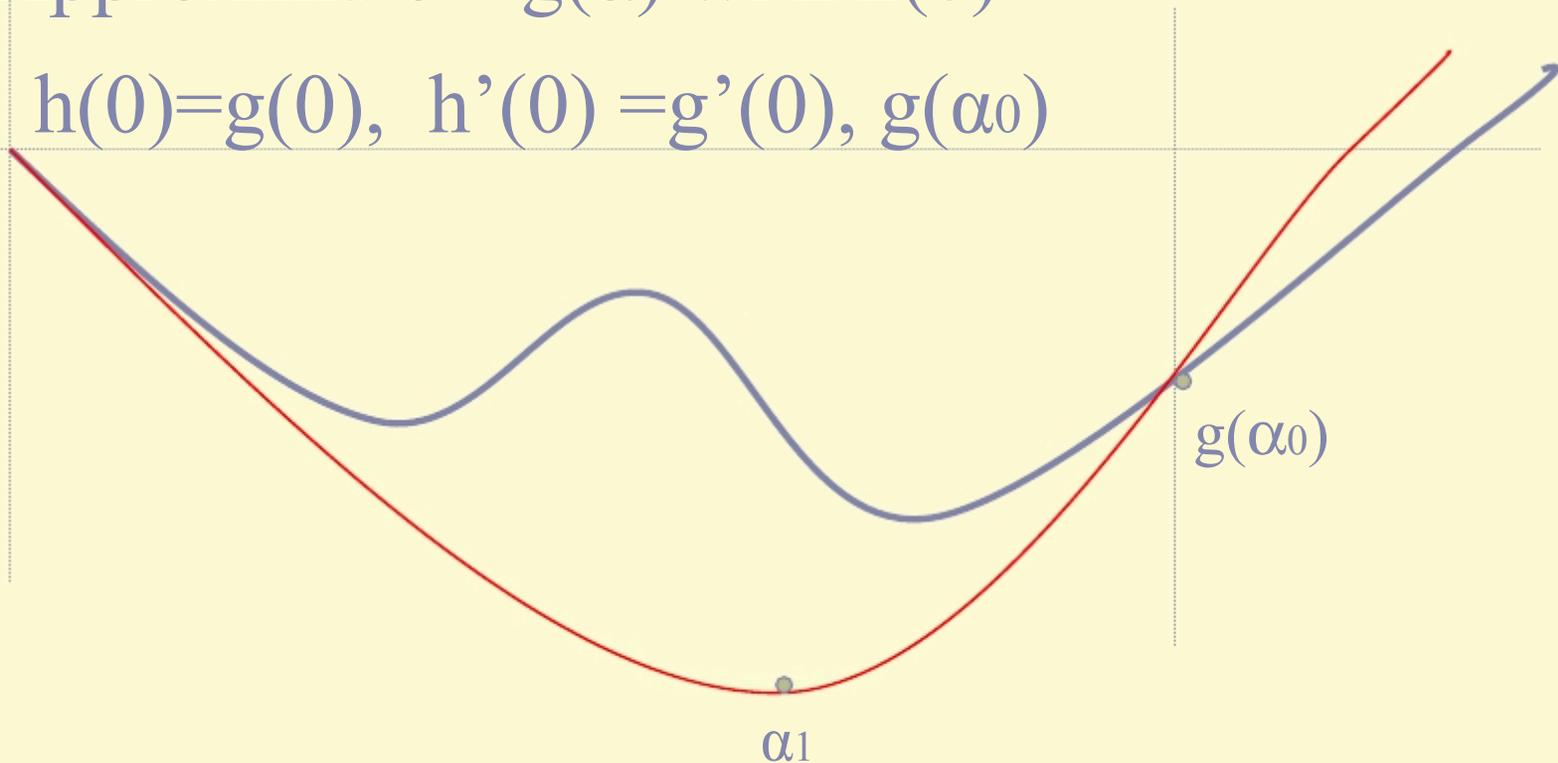
3.2.2.1 LINE SEARCH METHODS: USING INTERPOLATION IN LINE SEARCH

Quadratic Interpolation

- Approximate $g(\alpha)$ with $h(\alpha)$

$$h(0)=g(0), \quad h'(0)=g'(0), \quad g(\alpha_0)$$

α

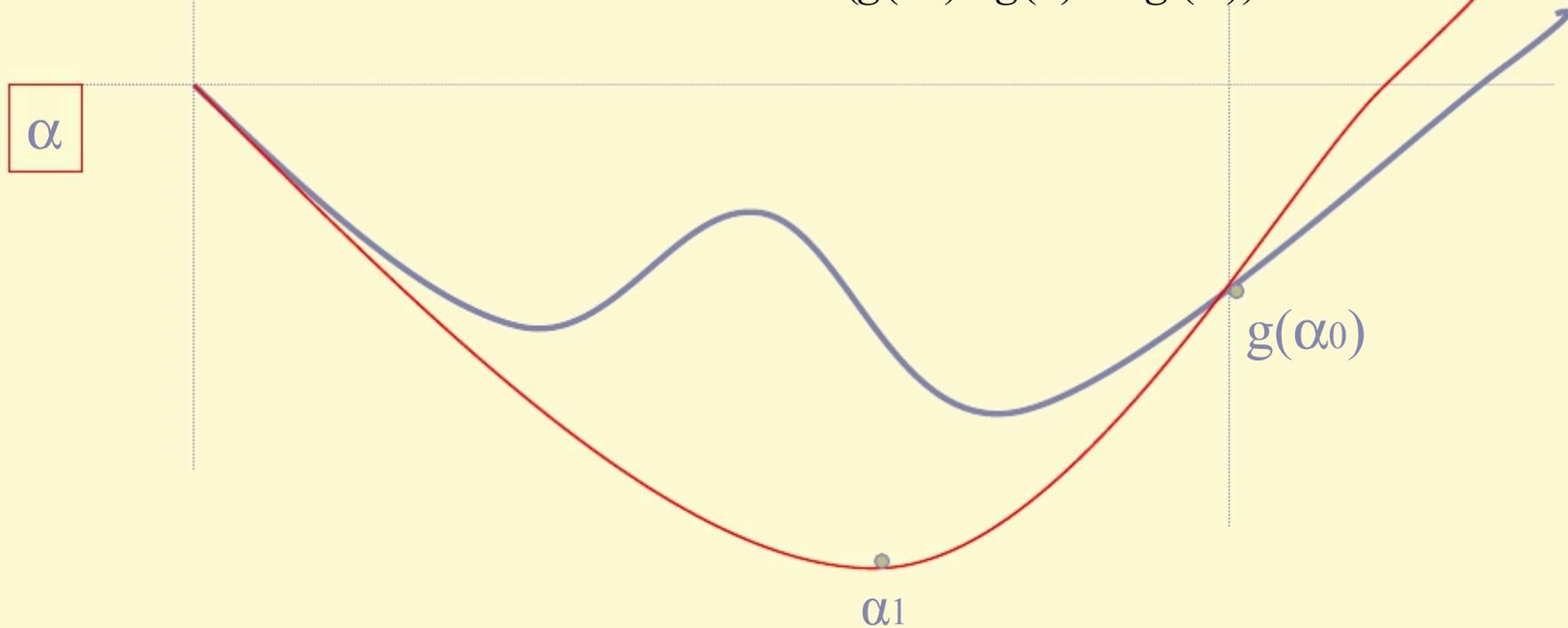


$$\left(\frac{g(\alpha_0) - g(0) - \alpha_0 g'(0)}{2} \right)^2$$

Quadratic Interpolation

Potential step

$$\alpha_1 = \frac{g'(\alpha)\alpha_0^2}{2(g(\alpha_0) - g(0) - \alpha_0 g'(\alpha))}$$



Cubic Interpolation

Cubic Interpolation

$$h(\alpha) = a\alpha^3 + b\alpha^2 + \alpha g'(0) + g(0)$$

$$\begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} 1 & \alpha_0^2 - \alpha_1^2 \\ \alpha_0^3 - \alpha_1^3 & \alpha_0^2 - \alpha_1^2 \end{bmatrix} \begin{pmatrix} g(\alpha_1) - g(0) - g'(0)\alpha_1 \\ g(\alpha_0) - g(0) - g'(0)\alpha_0 \end{pmatrix}$$

$$\alpha_2 = \frac{-b + \sqrt{b^2 - 3ag'(0)}}{3a}$$

3.2.2.2 LINE SEARCH

METHODS: OTHER LINE

SEARCH PRINCIPLES

Unconstrained optimization methods

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

α_k : Step length

d_k : Search direction

1) Line search

2) Trust-Region algorithms

Quadratic approximation

Influences

Step length computation:

1) Armijo rule:

$$f(x_k) - f(x_k + \beta^m \tau_k d_k) \geq -\rho \beta^m \tau_k \nabla f(x_k)^T d_k$$

$$\beta \in (0,1) \quad \rho \in (0,1/2)$$

$$\tau_k = -(\nabla f(x_k)^T d_k) / \|d_k\|^2$$

2) Goldstein rule:

$$\rho_1 \alpha_k g_k^T d_k \leq f(x_k + \alpha_k d_k) - f(x_k) \leq \rho_2 \alpha_k g_k^T d_k$$

$$0 < \rho_2 < \frac{1}{2} < \rho_1 < 1$$

3) Wolfe conditions:

$$f(x_k + \alpha_k d_k) - f(x_k) \leq \rho \alpha_k g_k^T d_k$$

$$\nabla f(x_k + \alpha_k d_k)^T d_k \geq \sigma g_k^T d_k$$

$$0 < \rho \leq \sigma < 1$$

Implementations:

Shanno (1978)

Moré - Thuente (1992-1994)

3.2.2.3 LINE SEARCH

METHODS: TAXONOMY OF METHODS

Methods for Unconstrained Optimization

1) Steepest descent (Cauchy, 1847)

$$d_k = -\nabla f(x_k)$$

2) Newton

$$d_k = -\nabla^2 f(x_k)^{-1} \nabla f(x_k)$$

3) Quasi-Newton (Broyden, 1965; and many others)

$$d_k = H_k \nabla f(x_k)$$

4) Conjugate Gradient Methods (1952)

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

β_k is known as the conjugate gradient parameter

$$s_k = x_{k+1} - x_k$$

5) Truncated Newton method (Dembo, et al, 1982)

$$d_k \cong -\nabla^2 f(x_k)^{-1} g_k \qquad \|r_k\| = \|\nabla^2 f(x_k) d_k + g_k\|$$

6) Trust Region methods

7) Conic model method (Davidon, 1980)

$$q(d) = f(x_k) + g_k^T d + \frac{1}{2} d^T B_k d$$

$$c(d) = f(x_k) + \frac{g_k^T d}{1 + b^T d} + \frac{1}{2} \frac{d^T A_k d}{(1 + b^T d)^2}$$

8) Tensorial methods (Schnabel & Frank, 1984)

$$m_T(x_c + d) = f(x_c) + \nabla f(x_c) \cdot d + \frac{1}{2} \nabla^2 f(x_c) \cdot d^2 \\ + \frac{1}{6} T_c \cdot d^3 + \frac{1}{24} V_c \cdot d^4$$

9) Methods based on systems of Differential Equations

Gradient flow Method (Courant, 1942)

$$\frac{dx}{dt} = -\alpha \nabla^2 f(x)^{-1} \nabla f(x)$$

$$x(0) = x_0$$

10) Direct searching methods

Hooke-Jeeves (form searching) (1961)

Powell (conjugate directions) (1964)

Rosenbrock (coordinate system rotation)(1960)

Nelder-Mead (rolling the simplex) (1965)

Powell –UOBYQA (quadratic approximation) (1994-2000)

N. Andrei, *Critica Retiunii Algoritmilor de Optimizare fara Restrictii*
Editura Academiei Romane, 2008.