Lecture 13

Symmetric Factorizations & Iterative Methods

Owen L. Lewis

Department of Mathematics and Statistics
University of New Mexico

Oct. 1, 2024

Goals for today. . .

- Review LDL^T factorization
- Show Cholesky factorization.
- Discussion of Matrix solve costs.
- Iterative methods

Reminder: Goal of Factorization?

- Different solves have different costs:
  - $Ax = b$ costs $\mathcal{O}(n^3)$ via Gaussian elimination.
  - $Ly = b$ or $Uy = b$ costs $\mathcal{O}(n^2)$ via forward/backward substitution.
  - $\rightarrow A = LU$ costs $\mathcal{O}(n^3)$ to compute L and U (basically just Gaussian Elimination), but each successive solve is just 2 traingular solves.
- What if we know that $A = A^T$, can we speed up computation of L and U?

Use SYMMETRY ! YRTEMMYS esU

Suppose

$$A = A^T.$$

$$\Rightarrow LU = A = LDL^T.$$

LDL^T Factorization

If A is symmetric, we can factor A as

$$A = LDL^T,$$

where L is the same lower triangular matrix made during LU factorization (don't need to do any work to create L^T.

How do we make D?

If you multiply out $A = LDL^T$ diagonal by diagonal and take advantage of all the zeros, you get

$$d_{11} = a_{11},$$

$$d_{22} = a_{22} - d_{11}\ell_{21}^2,$$

$$d_{33} = a_{33} - d_{11}\ell_{31}^2 - d_{22}\ell_{32}^2,$$

$$\vdots$$

LDL^T Factorization

Listing 1: LDL^T Factorization

```
given $A$
output $L$, $D$

for $j = 1 \dots n$
    $\ell_{jj} = 1$

    $d_j = a_{jj} - \sum_{\nu=1}^{j-1} d_\nu \ell_{j\nu}^2$ . % COULD BE NEGATIVE

    for $i = j+1 \dots n$
        $\ell_{ji} = 0$
        $\ell_{ij} = \left( a_{ij} - \sum_{\nu=1}^{j-1} \ell_{i\nu} d_\nu \ell_{j\nu} \right) / d_j$
    end
end
```

- Special form of the LU factorization (for symmetric A).

$\tilde{L}\tilde{L}^T$: Cholesky Factorization

- A must be symmetric and positive definite (SPD)
- A is Positive Definite (PD) if for all $x \neq 0$ the following holds

$$x^T A x > 0$$

- Positive definite gives us an all positive D in $A = LDL^T$
- Define $D^{1/2}$ via $\sqrt{d_{ii}}$.
- $A = LDL^T = LD^{1/2}D^{1/2}L^T$.
- Define $\tilde{L} = LD^{1/2}$.
- $A = \tilde{L}\tilde{L}^T$, i.e. $\tilde{L} = \tilde{U}^T$ (note, L had ones on the diagonal, $\tilde{L}$ does not).
  - Half as many flops as LU!
  - Only calculate L not U

Notes

Cholesky Factorization

Listing 2: Cholesky

```
given $A$
output $L$

for $k = 1 \dots n$
    $\ell_{kk} = \left(a_{kk} - \sum_{s=1}^{k-1} \ell_{ks}^2\right)^{1/2}$   %SQUARE ROOT OF D ENDS UP ON DIAGONAL

    for $i = k+1 \dots n$
        $\ell_{ik} = \left(a_{ik} - \sum_{s=1}^{k-1} \ell_{is}\ell_{ks}\right) / \ell_{kk}$
    end
end
```

Why SPD?

In general, SPD gives us

- non singular
  - If $x^T A x > 0$, for all nonzero x
  - Then $\lambda > 0$ for all eigenvalues λ
- No pivoting
  - From algorithm, can derive that $|l_{kj}| \leqslant \sqrt{a_{kk}}$
  - Elements of $\tilde{L}$ do not grow with respect to A
  - *For short proof see book*
- Cholesky faster than LU
  - No pivoting
  - Only calculate $\tilde{L}$, not U

Why SPD?

A matrix is Positive Definite (PD) if for all $x \neq 0$ the following holds

$$x^T A x > 0$$

- For SPD matrices, use the Cholesky factorization, $A = \tilde{L}\tilde{L}^T$
- Cholesky Factorization
  - Requires no pivoting
  - Requires one half as many flops as LU factorization, that is only calculate $\tilde{L}$ not L and U.
  - Cholesky will be more than *twice* as fast as LU because no pivoting means no data movement
- Use MATLAB's built-in `chol` function for routine work

Motivation Revisted

Multiple right hand sides

- Solve $Ax = b$ for k different b vectors
- Using LU factorization, the cost is $\mathcal{O}(n^3) + \mathcal{O}(kn^2)$
- Using Gaussian Elimination, the cost is $\mathcal{O}(kn^3)$

If A is symmetric

- Save 50% of the flops with LDL^T factorization
- Save 50% of the flops and many memory operations with Cholesky ($\tilde{L}\tilde{L}^T$) factorization

See `time_LU_vs_Cholesky.m`

The Backslash Operator

Given an $n \times n$ matrix A, and an $n \times 1$ vector b the \ operator performs a sequence of tests on the A matrix. MATLAB attempts to solve the system with the method that gives the least roundoff and the fewest operations.
When A is an $n \times n$ matrix:

1. MATLAB examines A to see if it is a permutation of a triangular system

   If so, the appropriate triangular solve is used.

2. MATLAB examines A to see if it *appears* to be symmetric and positive definite.

   If so, MATLAB attempts a Cholesky factorization
   and two triangular solves.

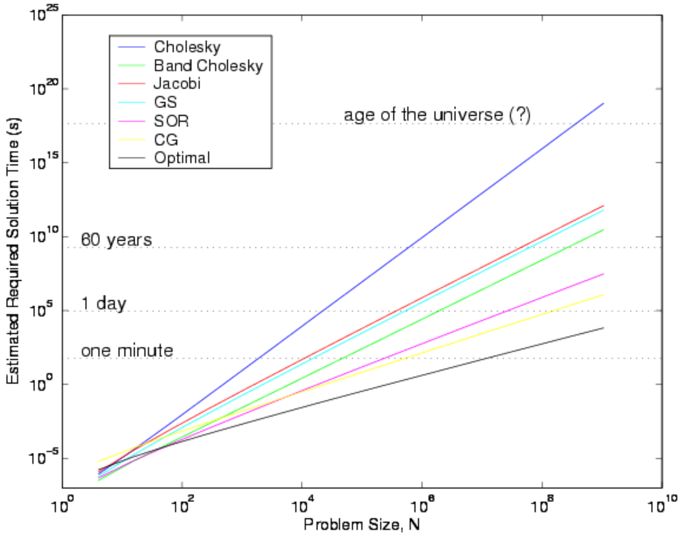
3. If the Cholesky factorization fails, or if A does not appear to be symmetric,

   MATLAB attempts an *LU* factorization
   and two triangular solves.

Summary: Complexity of Linear Solves

- $Ax = b$
- diagonal system: $\mathcal{O}(n)$
- tridiagonal system: $\mathcal{O}(n)$ (bigger constant)
- m-band system: $\mathcal{O}(m^2 n)$
- upper or lower triangular system: $\mathcal{O}(n^2)$
- full system with GE: $\mathcal{O}(n^3)$
- scaled partial pivoting adds $\mathcal{O}(n^2)$
- full system with LU: $\mathcal{O}(n^3)$
- LU back solve: $\mathcal{O}(n^2)$
- m different right-hand sides (when using LU): $\mathcal{O}(n^3 + mn^2)$

Summary: Complexity



Approximate solutions

So far, we are seeking “exact” (up to floating point error) solutions x^* to

$$Ax = b.$$

What if we only need an approximation $\hat{x}$ to x^*?

We would like some $\hat{x}$ so that $\|\hat{x} - x^*\| \leqslant \epsilon$, where ϵ is some tolerance.

The Residual

We can't actually evaluate

$$e = x^* - \hat{x}$$

But...

For $x = x^*$

$$b - Ax \equiv 0$$

For $x = \hat{x}$

$$b - Ax \not\equiv 0$$

We call $\hat{r} = b - A\hat{x}$ the *residual* (Recall from Lecture 7). It is a different way to measure the "error". In fact

$$\begin{aligned} \hat{r} &= b - A\hat{x} \\ &= Ax^* - A\hat{x} \\ &= A\hat{e} \end{aligned}$$

How big is the residual?

For a given approximation, $\hat{x}$ to x, how "big" is the residual $\hat{r} = b - A\hat{x}$?

- $\|r\|$ gives a magnitude
- $\|r\|_1 = \sum_{j=1}^{n} |r_i|$
- $\|r\|_2 = \left(\sum_{j=1}^{n} r_i^2\right)^{1/2}$
- $\|r\|_\infty = \max_{1 \leqslant j \leqslant n} |r_i|$

Notes

Approximating x...

Suppose we made a wild guess to the solution x of $Ax = b$:

$$x^{(0)} \approx x$$

How do I improve $x^{(0)}$?

Ideally:

$$x^{(1)} = x^{(0)} + e^{(0)}$$

but to obtain $e^{(0)}$, we must know x^*. Not a viable method.

Ideally (another way):

$$\begin{aligned} x^{(1)} &= x^{(0)} + e^{(0)} \\ &= x^{(0)} + (x^* - x^{(0)}) \\ &= x^{(0)} + (A^{-1}b - x^{(0)}) \\ &= x^{(0)} + A^{-1}(b - Ax^{(0)}) \\ &= x^{(0)} + A^{-1}r^{(0)} \end{aligned}$$

An iteration

Again, the method

$$x^{(1)} = x^{(0)} + A^{-1} r^{(0)}$$

is nonsense since A^{-1} is needed.

What if we approximate A^{-1}? Suppose $Q^{-1} \approx A^{-1}$ and is cheap to evaluate, then

$$x^{(1)} = x^{(0)} + Q^{-1} r^{(0)}$$

is a good step.

continuing...

$$x^{(k)} = x^{(k-1)} + Q^{-1} r^{(k-1)} \tag{1}$$

What kind of Q^{-1} do I need?

Rewrite:

$$x^{(k)} = x^{(k-1)} + Q^{-1}(b - Ax^{(k-1)})$$

This becomes

$$\begin{aligned} Qx^{(k)} &= Qx^{(k-1)} + (b - Ax^{(k-1)}) \\ &= (Q - A)x^{(k-1)} + b \end{aligned}$$

Or

$$x^{(k)} = Q^{-1}(Q - A)x^{(k-1)} + Q^{-1}b \tag{2}$$

Often (1) is more practical for programming, while (2) is useful to do math. There are exceptions however.

Choices, choices....

We want Q^{-1} to be simple/cheap to apply.
Ideally it will also be a “good” approximate of A^{-1}.

Let $A = D - C_L - C_U$ where D is diagonal, C_L and C_U are *strictly* lower and upper triangular.

Choices, choices....

We want Q^{-1} to be simple/cheap to apply.
Ideally it will also be a “good” approximate of A^{-1}.

Let $A = D - C_L - C_U$ where D is diagonal, C_L and C_U are *strictly* lower and upper triangular.

What if $Q = D$? Super cheap to invert, maybe a bad approximation... What if $Q = D - C_L$? Still kind of cheap, probably a better approximation ...

Two Popular Choices

Example

Jacobi iteration approximates A with $Q = diag(A)$.

```
$x = x^{(0)}$

Q = D

for $k = 1$ to $k_{max}$
    $r = b - Ax$
    if $\|r\| \leqslant tol$, stop

    $x = x + Q^{-1} r$
end
```

Two Popular Choices

Let $A = D - C_U - C_L$, where C_U and C_L are the negative of the strictly upper and lower triangular, respectively, of A.

Example

Gauss-Seidel iteration approximates A with $Q = lowertri(A)$.

```
$x = x^{(0)}$

$Q = D - C_L$

for $k = 1$ to $k_{max}$
    $r = b - Ax$
    if $\|r\| \leqslant tol$, stop

    $x = x + Q^{-1} r$
end
```

Why D and $D - C_L$?

Look again at the iteration

$$x^{(k)} = x^{(k-1)} + Q^{-1} r^{(k-1)}$$

Looking at the error:

$$x^* - x^{(k)} = x^* - x^{(k-1)} - Q^{-1} r^{(k-1)}$$

Gives

$$e^{(k)} = e^{(k-1)} - Q^{-1} A e^{(k-1)}$$

or

$$e^{(k)} = (I - Q^{-1} A) e^{(k-1)}$$

or

$$e^{(k)} = (I - Q^{-1} A)^k e^{(0)}$$

Why D and $D - C_L$?

We want

$$e^{(k)} = (I - Q^{-1}A)^k e^{(0)}$$

to converge.

Our iteration converges when $\|I - Q^{-1}A\| < 1$, because

$$\begin{aligned} \|e^{(k)}\| &= \|(I - Q^{-1}A)^k e^{(0)}\| \\ &\leqslant \|I - Q^{-1}A\|^k \|e^{(0)}\| \end{aligned}$$

goes to 0, as $k \to \infty$.

Notes

Matrix Norms (Recall Lecture 7)

What is $\|I - Q^{-1}A\|$?

- $\|A\|_1 = \max_{1 \leqslant j \leqslant n} \sum_{i=1}^{n} |a_{ij}|$
- $\|A\|_\infty = \max_{1 \leqslant i \leqslant n} \sum_{j=1}^{n} |a_{ij}|$

Other norms (you may be familiar with):

- $\|A\|_2 = \sqrt{\rho(A^T A)}$
- $\rho(A) = \max_{1 \leqslant j \leqslant n} |\lambda_i|$
- $\|A\|_2 = \rho(A)$ for symmetric A

Again, why do Jacobi and Gauss-Seidel work?

Jacobi, Gauss-Seidel (sufficient) Convergence Theorem

If A is diagonally dominant, then the Jacobi and Gauss-Seidel methods converge for any initial guess $x^{(0)}$.

Definition: Diagonal Dominance

A matrix is *diagonally dominant* if

$$|a_{ii}| > \sum_{j=1, j\neq i}^{n} |a_{ij}|$$

for all i.

Notes

Smart Jacobi Algorithm

The Jacobi algorithm uses the matrix representation:

$$x^{(k)} = D^{-1}(C_L + C_U)x^{(k-1)} + D^{-1}b$$

Or componentwise,

$$x_i^{(k)} = -\sum_{j=1, j\neq i}^{n} \left(\frac{a_{ij}}{a_{ii}}\right) x_j^{(k-1)} + \frac{b_i}{a_{ii}}$$

So each sweep (from $k-1$ to k) uses $\mathcal{O}(n)$ operations per vector element (potentially $\mathcal{O}(n^2)$ total).

If, for each row i, $a_{ij} \neq 0$ for *at most m* values, each sweep uses $\mathcal{O}(mn)$ operations.

Smart Gauss-Seidel Algorithm

The Gauss-Seidel algorithm uses the matrix representation:

$$x^{(k)} = (D - C_L)^{-1} C_U x^{(k-1)} + (D - C_L)^{-1} b$$

Component-wise:

$$x_i^{(k)} = -\sum_{j=1}^{i-1} \left(\frac{a_{ij}}{a_{ii}} \right) x_j^{(k)} - \sum_{j=i+1}^{n} \left(\frac{a_{ij}}{a_{ii}} \right) x_j^{(k-1)} + \frac{b_i}{a_{ii}}$$

So again each sweep (from $k - 1$ to k) uses $\mathcal{O}(n)$ operations per vector element.
If, for each row i, $a_{ij} \neq 0$ for *at most m* values, each sweep uses $\mathcal{O}(mn)$ operations.

The difference is that in the Jacobi method, updates are saved (and not used) in a new vector. With Gauss-Seidel, an update to an element $x_i^{(k)}$ is used immediately.

Example:

Lets do Jacobi and Gauss-Seidel:

Example

Consider

$$A = \begin{bmatrix} 2 & -1 & 0 \\ -1 & 3 & -1 \\ 0 & -1 & 2 \end{bmatrix}, \qquad b = \begin{bmatrix} 1 \\ 8 \\ -5 \end{bmatrix}, \qquad x^{(0)} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

See `iterative_methods.m`