CS 450 – Numerical Analysis

Chapter 2: Systems of Linear Equations †

Prof. Michael T. Heath

Department of Computer Science
University of Illinois at Urbana-Champaign
heath@illinois.edu

January 28, 2019

†Lecture slides based on the textbook *Scientific Computing: An Introductory Survey* by Michael T. Heath, copyright © 2018 by the Society for Industrial and Applied Mathematics. http://www.siam.org/books/cl80

Systems of Linear Equations

Review: Matrix-Vector Product

$$\boldsymbol{Ax} = \begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$$

$$= \begin{bmatrix} a_{1,1}x_1 + a_{1,2}x_2 + \cdots + a_{1,n}x_n \\ \hline a_{2,1}x_1 + a_{2,2}x_2 + \cdots + a_{2,n}x_n \\ \hline \vdots \\ \hline a_{m,1}x_1 + a_{m,2}x_2 + \cdots + a_{m,n}x_n \end{bmatrix}$$

$$= x_1 \begin{bmatrix} a_{1,1} \\ a_{2,1} \\ \vdots \\ a_{m,1} \end{bmatrix} + x_2 \begin{bmatrix} a_{1,2} \\ a_{2,2} \\ \vdots \\ a_{m,2} \end{bmatrix} + \cdots + x_n \begin{bmatrix} a_{1,n} \\ a_{2,n} \\ \vdots \\ a_{m,n} \end{bmatrix}$$

Definition: For $\boldsymbol{A} \in \mathbb{R}^{m \times n}$, $\text{span}(\boldsymbol{A}) = \{\boldsymbol{Ax} : \ \boldsymbol{x} \in \mathbb{R}^n\}$

System of Linear Equations

$$\boldsymbol{A}\,\boldsymbol{x} = \boldsymbol{b}$$

- Given $m \times n$ matrix $\boldsymbol{A}$ and m-vector $\boldsymbol{b}$, find unknown n-vector $\boldsymbol{x}$ satisfying $\boldsymbol{Ax} = \boldsymbol{b}$
- System of equations asks whether $\boldsymbol{b}$ can be expressed as linear combination of columns of $\boldsymbol{A}$, or equivalently, is $\boldsymbol{b} \in \text{span}(\boldsymbol{A})$?
- If so, coefficients of linear combination are components of solution vector $\boldsymbol{x}$
- Solution may or may not *exist*, and may or may not be *unique*
- For now, we consider only *square* case, $m = n$

Singularity and Nonsingularity

$n \times n$ matrix $\boldsymbol{A}$ is *nonsingular* if it has any of following equivalent properties

1. Inverse of $\boldsymbol{A}$, denoted by $\boldsymbol{A}^{-1}$, exists such that $\boldsymbol{A}\boldsymbol{A}^{-1} = \boldsymbol{A}^{-1}\boldsymbol{A} = \boldsymbol{I}$
2. $\det(\boldsymbol{A}) \neq 0$
3. $\text{rank}(\boldsymbol{A}) = n$
4. For any vector $\boldsymbol{z} \neq \boldsymbol{0}$, $\boldsymbol{A}\boldsymbol{z} \neq \boldsymbol{0}$

Existence and Uniqueness

- Existence and uniqueness of solution to $\boldsymbol{Ax} = \boldsymbol{b}$ depend on whether $\boldsymbol{A}$ is singular or nonsingular
- Can also depend on $\boldsymbol{b}$, but only in singular case
- If $\boldsymbol{b} \in \text{span}(\boldsymbol{A})$, system is *consistent*

$\boldsymbol{A}$	$\boldsymbol{b}$	# solutions
nonsingular	arbitrary	1
singular	$\boldsymbol{b} \in \text{span}(\boldsymbol{A})$	∞
singular	$\boldsymbol{b} \notin \text{span}(\boldsymbol{A})$	0

Geometric Interpretation

- In two dimensions, each equation determines straight line in plane
- Solution is intersection point of two straight lines, if any
- If two straight lines are not parallel (nonsingular), then their intersection point is unique solution
- If two straight lines are parallel (singular), then they either do not intersect (no solution) or else they coincide (any point along line is solution)
- In higher dimensions, each equation determines hyperplane; if matrix is nonsingular, intersection of hyperplanes is unique solution

Example: Nonsingularity

- 2×2 system

$$\begin{aligned} 2x_1 + 3x_2 &= b_1 \\ 5x_1 + 4x_2 &= b_2 \end{aligned}$$

or in matrix-vector notation

$$\boldsymbol{Ax} = \begin{bmatrix} 2 & 3 \\ 5 & 4 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} = \boldsymbol{b}$$

is nonsingular and thus has unique solution regardless of value of $\boldsymbol{b}$

- For example, if $\boldsymbol{b} = \begin{bmatrix} 8 & 13 \end{bmatrix}^T$, then $\boldsymbol{x} = \begin{bmatrix} 1 & 2 \end{bmatrix}^T$ is unique solution

Example: Singularity

- 2×2 system

$$\boldsymbol{Ax} = \begin{bmatrix} 2 & 3 \\ 4 & 6 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} = \boldsymbol{b}$$

  is singular regardless of value of $\boldsymbol{b}$

- With $\boldsymbol{b} = \begin{bmatrix} 4 & 7 \end{bmatrix}^T$, there is no solution

- With $\boldsymbol{b} = \begin{bmatrix} 4 & 8 \end{bmatrix}^T$, $\boldsymbol{x} = \begin{bmatrix} \gamma & (4 - 2\gamma)/3 \end{bmatrix}^T$ is solution for any real number γ, so there are infinitely many solutions

Norms and Condition Number

Vector Norms

- Magnitude (absolute value, modulus) for scalars generalizes to *norm* for vectors

- We will use only p-norms, defined by

$$\|\boldsymbol{x}\|_p = \left(\sum_{i=1}^{n} |x_i|^p \right)^{1/p}$$

  for integer $p > 0$ and n-vector $\boldsymbol{x}$

- Important special cases

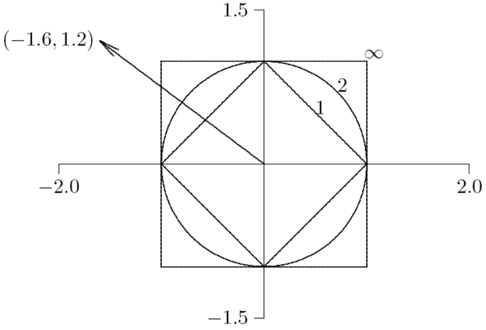
  - 1-norm: $\|\boldsymbol{x}\|_1 = \sum_{i=1}^{n} |x_i|$

  - 2-norm: $\|\boldsymbol{x}\|_2 = \left(\sum_{i=1}^{n} |x_i|^2 \right)^{1/2}$

  - ∞-norm: $\|\boldsymbol{x}\|_\infty = \max_i |x_i|$

Example: Vector Norms

- Drawing shows unit "circle" in two dimensions for each norm



- Norms have following values for vector shown

$$\|\boldsymbol{x}\|_1 = 2.8, \quad \|\boldsymbol{x}\|_2 = 2.0, \quad \|\boldsymbol{x}\|_\infty = 1.6$$

⟨ interactive example ⟩

Equivalence of Norms

- In general, for any vector $\boldsymbol{x}$ in $\mathbb{R}^n$, $\|\boldsymbol{x}\|_1 \geq \|\boldsymbol{x}\|_2 \geq \|\boldsymbol{x}\|_\infty$
- However, we also have
  - $\|\boldsymbol{x}\|_1 \leq \sqrt{n} \cdot \|\boldsymbol{x}\|_2$
  - $\|\boldsymbol{x}\|_2 \leq \sqrt{n} \cdot \|\boldsymbol{x}\|_\infty$
  - $\|\boldsymbol{x}\|_1 \leq n \cdot \|\boldsymbol{x}\|_\infty$
- For given n, norms differ by at most a constant, and hence are equivalent: if one is small, all must be proportionally small
- Consequently, we can use whichever norm is most convenient in given context

Properties of Vector Norms

- For any vector norm
  - $\|\boldsymbol{x}\| > 0$ if $\boldsymbol{x} \neq \boldsymbol{0}$
  - $\|\gamma \boldsymbol{x}\| = |\gamma| \cdot \|\boldsymbol{x}\|$ for any scalar γ
  - $\|\boldsymbol{x} + \boldsymbol{y}\| \leq \|\boldsymbol{x}\| + \|\boldsymbol{y}\|$ (triangle inequality)
- In more general treatment, these properties taken as *definition* of vector norm
- Useful variation on triangle inequality
  - $|\,\|\boldsymbol{x}\| - \|\boldsymbol{y}\|\,| \leq \|\boldsymbol{x} - \boldsymbol{y}\|$

Matrix Norms

- *Matrix norm* induced by a given vector norm is defined by

$$\|\boldsymbol{A}\| = \max_{\boldsymbol{x} \neq \boldsymbol{0}} \frac{\|\boldsymbol{A}\boldsymbol{x}\|}{\|\boldsymbol{x}\|}$$

- Norm of matrix measures maximum relative stretching matrix does to any vector in given vector norm

Example Matrix Norms

- Matrix norm induced by vector 1-norm is maximum absolute *column* sum

$$\|\boldsymbol{A}\|_1 = \max_j \sum_{i=1}^{n} |a_{ij}|$$

- Matrix norm induced by vector ∞-norm is maximum absolute *row* sum

$$\|\boldsymbol{A}\|_\infty = \max_i \sum_{j=1}^{n} |a_{ij}|$$

- Handy way to remember these is that matrix norms agree with corresponding vector norms for $n \times 1$ matrix

- No simple formula for matrix 2-norm

Properties of Matrix Norms

- Any matrix norm satisfies
  - $\|\boldsymbol{A}\| > 0$ if $\boldsymbol{A} \neq \mathbf{0}$
  - $\|\gamma \boldsymbol{A}\| = |\gamma| \cdot \|\boldsymbol{A}\|$ for any scalar γ
  - $\|\boldsymbol{A} + \boldsymbol{B}\| \leq \|\boldsymbol{A}\| + \|\boldsymbol{B}\|$
- Matrix norms we have defined also satisfy
  - $\|\boldsymbol{AB}\| \leq \|\boldsymbol{A}\| \cdot \|\boldsymbol{B}\|$
  - $\|\boldsymbol{Ax}\| \leq \|\boldsymbol{A}\| \cdot \|\boldsymbol{x}\|$ for any vector $\boldsymbol{x}$

Condition Number

- *Condition number* of square nonsingular matrix $\boldsymbol{A}$ is defined by

$$\text{cond}(\boldsymbol{A}) = \|\boldsymbol{A}\| \cdot \|\boldsymbol{A}^{-1}\|$$

- By convention, $\text{cond}(\boldsymbol{A}) = \infty$ if $\boldsymbol{A}$ is singular

- Since

$$\|\boldsymbol{A}\| \cdot \|\boldsymbol{A}^{-1}\| = \left(\max_{\boldsymbol{x} \neq \boldsymbol{0}} \frac{\|\boldsymbol{A}\boldsymbol{x}\|}{\|\boldsymbol{x}\|} \right) \cdot \left(\min_{\boldsymbol{x} \neq \boldsymbol{0}} \frac{\|\boldsymbol{A}\boldsymbol{x}\|}{\|\boldsymbol{x}\|} \right)^{-1}$$

  condition number measures ratio of maximum stretching to maximum shrinking matrix does to any nonzero vectors

- Large $\text{cond}(\boldsymbol{A})$ means $\boldsymbol{A}$ is *nearly singular*

Properties of Condition Number

- For any matrix $\boldsymbol{A}$, $\text{cond}(\boldsymbol{A}) \geq 1$
- For identity matrix $\boldsymbol{I}$, $\text{cond}(\boldsymbol{I}) = 1$
- For any matrix $\boldsymbol{A}$ and scalar γ, $\text{cond}(\gamma \boldsymbol{A}) = \text{cond}(\boldsymbol{A})$
- For any diagonal matrix $\boldsymbol{D} = \text{diag}(d_i)$, $\text{cond}(\boldsymbol{D}) = \dfrac{\max |d_i|}{\min |d_i|}$

⟨ interactive example ⟩

Computing Condition Number

- Definition of condition number involves matrix inverse, so it is nontrivial to compute
- Computing condition number from definition would require much more work than computing solution whose accuracy is to be assessed

- In practice, condition number is estimated inexpensively as byproduct of solution process
- Matrix norm $\|\boldsymbol{A}\|$ is easily computed as maximum absolute column sum (or row sum, depending on norm used)
- Estimating $\|\boldsymbol{A}^{-1}\|$ at low cost is more challenging

Computing Condition Number, continued

- From properties of norms, if $\boldsymbol{Az} = \boldsymbol{y}$, then

$$\frac{\|\boldsymbol{z}\|}{\|\boldsymbol{y}\|} \leq \|\boldsymbol{A}^{-1}\|$$

  and this bound is achieved for optimally chosen $\boldsymbol{y}$

- Efficient condition estimators heuristically pick $\boldsymbol{y}$ with large ratio $\|\boldsymbol{z}\|/\|\boldsymbol{y}\|$, yielding good estimate for $\|\boldsymbol{A}^{-1}\|$

- Good software packages for linear systems provide efficient and reliable condition estimator

- Condition number useful in assessing accuracy of approximate solution

Assessing Accuracy

Error Bounds

- Condition number yields error bound for approximate solution to linear system
- Let $\boldsymbol{x}$ be solution to $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$, and let $\hat{\boldsymbol{x}}$ be solution to $\boldsymbol{A}\hat{\boldsymbol{x}} = \boldsymbol{b} + \Delta\boldsymbol{b}$
- If $\Delta\boldsymbol{x} = \hat{\boldsymbol{x}} - \boldsymbol{x}$, then

$$\boldsymbol{b} + \Delta\boldsymbol{b} = \boldsymbol{A}(\hat{\boldsymbol{x}}) = \boldsymbol{A}(\boldsymbol{x} + \Delta\boldsymbol{x}) = \boldsymbol{A}\boldsymbol{x} + \boldsymbol{A}\Delta\boldsymbol{x}$$

which leads to bound

$$\frac{\|\Delta\boldsymbol{x}\|}{\|\boldsymbol{x}\|} \leq \text{cond}(\boldsymbol{A})\frac{\|\Delta\boldsymbol{b}\|}{\|\boldsymbol{b}\|}$$

for possible relative change in solution $\boldsymbol{x}$ due to relative change in right-hand side $\boldsymbol{b}$

⟨ interactive example ⟩

Error Bounds, continued

- Similar result holds for relative change in matrix: if $(\boldsymbol{A} + \boldsymbol{E})\hat{\boldsymbol{x}} = \boldsymbol{b}$, then

$$\frac{\|\Delta \boldsymbol{x}\|}{\|\hat{\boldsymbol{x}}\|} \leq \text{cond}(\boldsymbol{A}) \frac{\|\boldsymbol{E}\|}{\|\boldsymbol{A}\|}$$

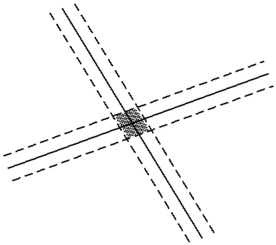
- If input data are accurate to machine precision, then bound for relative error in solution $\boldsymbol{x}$ becomes

$$\frac{\|\hat{\boldsymbol{x}} - \boldsymbol{x}\|}{\|\boldsymbol{x}\|} \leq \text{cond}(\boldsymbol{A})\, \epsilon_{\text{mach}}$$

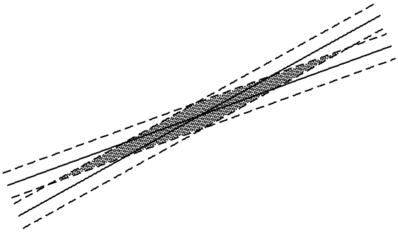
- Computed solution loses about $\log_{10}(\text{cond}(\boldsymbol{A}))$ decimal digits of accuracy relative to accuracy of input

Error Bounds – Illustration

- In two dimensions, uncertainty in intersection point of two lines depends on whether lines are nearly parallel



well-conditioned



ill-conditioned

⟨ interactive example ⟩

Error Bounds – Caveats

- Normwise analysis bounds relative error in *largest* components of solution; relative error in smaller components can be much larger
  - Componentwise error bounds can be obtained, but are somewhat more complicated
- Conditioning of system is affected by relative scaling of rows or columns
  - Ill-conditioning can result from poor scaling as well as near singularity
  - Rescaling can help the former, but not the latter

Residual

- *Residual vector* of approximate solution $\hat{\boldsymbol{x}}$ to linear system $\boldsymbol{Ax} = \boldsymbol{b}$ is defined by

$$\boldsymbol{r} = \boldsymbol{b} - \boldsymbol{A}\hat{\boldsymbol{x}}$$

- In theory, if $\boldsymbol{A}$ is nonsingular, then $\|\hat{\boldsymbol{x}} - \boldsymbol{x}\| = 0$ if, and only if, $\|\boldsymbol{r}\| = 0$, but they are not necessarily *small* simultaneously

- Since

$$\frac{\|\Delta \boldsymbol{x}\|}{\|\hat{\boldsymbol{x}}\|} \leq \text{cond}(\boldsymbol{A}) \frac{\|\boldsymbol{r}\|}{\|\boldsymbol{A}\| \cdot \|\hat{\boldsymbol{x}}\|}$$

small relative residual implies small relative error in approximate solution *only if* $\boldsymbol{A}$ is well-conditioned

Residual, continued

- If computed solution $\hat{\boldsymbol{x}}$ exactly satisfies

  $$(\boldsymbol{A} + \boldsymbol{E})\hat{\boldsymbol{x}} = \boldsymbol{b}$$

  then

  $$\frac{\|\boldsymbol{r}\|}{\|\boldsymbol{A}\| \; \|\hat{\boldsymbol{x}}\|} \le \frac{\|\boldsymbol{E}\|}{\|\boldsymbol{A}\|}$$

  so large *relative residual* implies large backward error in matrix, and algorithm used to compute solution is *unstable*

- Stable algorithm yields small relative residual regardless of conditioning of nonsingular system

- Small residual is easy to obtain, but does *not* necessarily imply computed solution is accurate

Example: Small Residual

- For linear system

$$\boldsymbol{A}\boldsymbol{x} = \begin{bmatrix} 0.913 & 0.659 \\ 0.457 & 0.330 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 0.254 \\ 0.127 \end{bmatrix} = \boldsymbol{b}$$

consider two approximate solutions

$$\hat{\boldsymbol{x}}_1 = \begin{bmatrix} 0.6391 \\ -0.5 \end{bmatrix}, \qquad \hat{\boldsymbol{x}}_2 = \begin{bmatrix} 0.999 \\ -1.001 \end{bmatrix}$$

- Norms of respective residuals are

$$\|\boldsymbol{r}_1\|_1 = 7.0 \times 10^{-5}, \qquad \|\boldsymbol{r}_2\|_1 = 2.4 \times 10^{-2}$$

- Exact solution is $\boldsymbol{x} = [1, -1]^T$, so $\hat{\boldsymbol{x}}_2$ is much more accurate than $\hat{\boldsymbol{x}}_1$, despite having much larger residual
- $\boldsymbol{A}$ is ill-conditioned ($\text{cond}(\boldsymbol{A}) > 10^4$), so small residual does *not* imply small error

Solving Linear Systems

Solving Linear Systems

- General strategy: To solve linear system, transform it into one whose solution is same but easier to compute
- What type of transformation of linear system leaves solution unchanged?
- We can *premultiply* (from left) both sides of linear system $\boldsymbol{Ax} = \boldsymbol{b}$ by any *nonsingular* matrix $\boldsymbol{M}$ without affecting solution
- Solution to $\boldsymbol{MAx} = \boldsymbol{Mb}$ is given by

$$\boldsymbol{x} = (\boldsymbol{MA})^{-1}\boldsymbol{Mb} = \boldsymbol{A}^{-1}\boldsymbol{M}^{-1}\boldsymbol{Mb} = \boldsymbol{A}^{-1}\boldsymbol{b}$$

Example: Permutations

- *Permutation matrix* $\boldsymbol{P}$ has one 1 in each row and column and zeros elsewhere, i.e., identity matrix with rows or columns permuted
- $\boldsymbol{P}^T$ reverses permutation, so $\boldsymbol{P}^{-1} = \boldsymbol{P}^T$
- Premultiplying both sides of system by permutation matrix, $\boldsymbol{PAx} = \boldsymbol{Pb}$, reorders rows, but solution $\boldsymbol{x}$ is unchanged
- Postmultiplying $\boldsymbol{A}$ by permutation matrix, $\boldsymbol{APx} = \boldsymbol{b}$, reorders columns, which permutes components of original solution

$$\boldsymbol{x} = (\boldsymbol{AP})^{-1}\boldsymbol{b} = \boldsymbol{P}^{-1}\boldsymbol{A}^{-1}\boldsymbol{b} = \boldsymbol{P}^T(\boldsymbol{A}^{-1}\boldsymbol{b})$$

Example: Diagonal Scaling

- Row scaling: premultiplying both sides of system by nonsingular diagonal matrix $\boldsymbol{D}$, $\boldsymbol{DAx} = \boldsymbol{Db}$, multiplies each row of matrix and right-hand side by corresponding diagonal entry of $\boldsymbol{D}$, but solution $\boldsymbol{x}$ is unchanged

- Column scaling: postmultiplying $\boldsymbol{A}$ by $\boldsymbol{D}$, $\boldsymbol{ADx} = \boldsymbol{b}$, multiplies each column of matrix by corresponding diagonal entry of $\boldsymbol{D}$, which rescales original solution

$$\boldsymbol{x} = (\boldsymbol{AD})^{-1}\boldsymbol{b} = \boldsymbol{D}^{-1}\boldsymbol{A}^{-1}\boldsymbol{b}$$

Triangular Linear Systems

- What type of linear system is easy to solve?
- If one equation in system involves only one component of solution (i.e., only one entry in that row of matrix is nonzero), then that component can be computed by division
- If another equation in system involves only one additional solution component, then by substituting one known component into it, we can solve for other component
- If this pattern continues, with only one new solution component per equation, then all components of solution can be computed in succession.
- System with this property is called *triangular*

Triangular Matrices

- Two specific triangular forms are of particular interest
  - *lower triangular*: all entries *above* main diagonal are zero, $a_{ij} = 0$ for $i < j$
  - *upper triangular*: all entries *below* main diagonal are zero, $a_{ij} = 0$ for $i > j$
- Successive substitution process described earlier is especially easy to formulate for lower or upper triangular systems
- Any triangular matrix can be permuted into upper or lower triangular form by suitable row permutation

Forward-Substitution

- *Forward-substitution* for lower triangular system $\boldsymbol{Lx} = \boldsymbol{b}$

$$x_1 = b_1/\ell_{11}, \quad x_i = \left(b_i - \sum_{j=1}^{i-1} \ell_{ij} x_j \right) / \ \ell_{ii}, \quad i = 2, \ldots, n$$

```
for j = 1 to n                          { loop over columns }
    if ℓ_jj = 0 then stop               { stop if matrix is singular }
    x_j = b_j / ℓ_jj                    { compute solution component }
    for i = j + 1 to n
        b_i = b_i − ℓ_ij x_j            { update right-hand side }
    end
end
```

Back-Substitution

- *Back-substitution* for upper triangular system $\boldsymbol{Ux} = \boldsymbol{b}$

$$x_n = b_n / u_{nn}, \quad x_i = \left(b_i - \sum_{j=i+1}^{n} u_{ij} x_j \right) / \, u_{ii}, \quad i = n-1, \ldots, 1$$

for $j = n$ **to** 1	{ loop backwards over columns }
if $u_{jj} = 0$ **then** stop	{ stop if matrix is singular }
$x_j = b_j / u_{jj}$	{ compute solution component }
for $i = 1$ **to** $j - 1$	
$b_i = b_i - u_{ij} x_j$	{ update right-hand side }
end	
end	

Example: Triangular Linear System

$$\begin{bmatrix} 2 & 4 & -2 \\ 0 & 1 & 1 \\ 0 & 0 & 4 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 2 \\ 4 \\ 8 \end{bmatrix}$$

- Using back-substitution for this upper triangular system, last equation, $4x_3 = 8$, is solved directly to obtain $x_3 = 2$
- Next, x_3 is substituted into second equation to obtain $x_2 = 2$
- Finally, both x_3 and x_2 are substituted into first equation to obtain $x_1 = -1$

Elementary Elimination Matrices

Elimination

- To transform general linear system into triangular form, need to replace selected nonzero entries of matrix by zeros
- This can be accomplished by taking linear combinations of rows
- Consider 2-vector $\boldsymbol{a} = \begin{bmatrix} a_1 \\ a_2 \end{bmatrix}$
- If $a_1 \neq 0$, then

$$\begin{bmatrix} 1 & 0 \\ -a_2/a_1 & 1 \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \end{bmatrix} = \begin{bmatrix} a_1 \\ 0 \end{bmatrix}$$

Elementary Elimination Matrices

- More generally, we can annihilate *all* entries below kth position in n-vector $\boldsymbol{a}$ by transformation

$$\boldsymbol{M}_k\boldsymbol{a} = \begin{bmatrix} 1 & \cdots & 0 & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & 1 & 0 & \cdots & 0 \\ 0 & \cdots & -m_{k+1} & 1 & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & -m_n & 0 & \cdots & 1 \end{bmatrix} \begin{bmatrix} a_1 \\ \vdots \\ a_k \\ a_{k+1} \\ \vdots \\ a_n \end{bmatrix} = \begin{bmatrix} a_1 \\ \vdots \\ a_k \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

  where $m_i = a_i/a_k$, $i = k+1, \ldots, n$

- Divisor a_k, called *pivot*, must be nonzero

- Matrix $\boldsymbol{M}_k$, called *elementary elimination matrix*, adds multiple of row k to each subsequent row, with *multipliers* m_i chosen so that result is zero

Elementary Elimination Matrices, continued

- $\boldsymbol{M}_k$ is unit lower triangular and nonsingular
- $\boldsymbol{M}_k = \boldsymbol{I} - \boldsymbol{m}_k \boldsymbol{e}_k^T$, where $\boldsymbol{m}_k = [0, \ldots, 0, m_{k+1}, \ldots, m_n]^T$ and $\boldsymbol{e}_k$ is kth column of identity matrix
- $\boldsymbol{M}_k^{-1} = \boldsymbol{I} + \boldsymbol{m}_k \boldsymbol{e}_k^T$, which means $\boldsymbol{M}_k^{-1} = \boldsymbol{L}_k$ is same as $\boldsymbol{M}_k$ except signs of multipliers are reversed
- If $\boldsymbol{M}_j$, $j > k$, is another elementary elimination matrix, with vector of multipliers $\boldsymbol{m}_j$, then

$$\begin{aligned}\boldsymbol{M}_k \boldsymbol{M}_j &= \boldsymbol{I} - \boldsymbol{m}_k \boldsymbol{e}_k^T - \boldsymbol{m}_j \boldsymbol{e}_j^T + \boldsymbol{m}_k \boldsymbol{e}_k^T \boldsymbol{m}_j \boldsymbol{e}_j^T \\ &= \boldsymbol{I} - \boldsymbol{m}_k \boldsymbol{e}_k^T - \boldsymbol{m}_j \boldsymbol{e}_j^T\end{aligned}$$

  which means their product is essentially their "union" and similarly for product of inverses, $\boldsymbol{L}_k \boldsymbol{L}_j$

Example: Elementary Elimination Matrices

- For $\boldsymbol{a} = \begin{bmatrix} 2 \\ 4 \\ -2 \end{bmatrix}$,

$$\boldsymbol{M}_1 \boldsymbol{a} = \begin{bmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} 2 \\ 4 \\ -2 \end{bmatrix} = \begin{bmatrix} 2 \\ 0 \\ 0 \end{bmatrix}$$

and

$$\boldsymbol{M}_2 \boldsymbol{a} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1/2 & 1 \end{bmatrix} \begin{bmatrix} 2 \\ 4 \\ -2 \end{bmatrix} = \begin{bmatrix} 2 \\ 4 \\ 0 \end{bmatrix}$$

Example, continued

- Note that

$$\boldsymbol{L}_1 = \boldsymbol{M}_1^{-1} = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ -1 & 0 & 1 \end{bmatrix}, \quad \boldsymbol{L}_2 = \boldsymbol{M}_2^{-1} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -1/2 & 1 \end{bmatrix}$$

and

$$\boldsymbol{M}_1 \boldsymbol{M}_2 = \begin{bmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ 1 & 1/2 & 1 \end{bmatrix}, \quad \boldsymbol{L}_1 \boldsymbol{L}_2 = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ -1 & -1/2 & 1 \end{bmatrix}$$

LU Factorization by Gaussian Elimination

Gaussian Elimination

- To reduce general linear system $\boldsymbol{Ax} = \boldsymbol{b}$ to upper triangular form, first choose $\boldsymbol{M}_1$, with a_{11} as pivot, to annihilate first column of $\boldsymbol{A}$ below first row
  - System becomes $\boldsymbol{M}_1\boldsymbol{Ax} = \boldsymbol{M}_1\boldsymbol{b}$, but solution is unchanged
- Next choose $\boldsymbol{M}_2$, using a_{22} as pivot, to annihilate second column of $\boldsymbol{M}_1\boldsymbol{A}$ below second row
  - System becomes $\boldsymbol{M}_2\boldsymbol{M}_1\boldsymbol{Ax} = \boldsymbol{M}_2\boldsymbol{M}_1\boldsymbol{b}$, but solution is still unchanged
- Process continues for each successive column until all subdiagonal entries have been zeroed
- Resulting upper triangular linear system

$$\begin{aligned} \boldsymbol{M}_{n-1}\cdots\boldsymbol{M}_1\boldsymbol{Ax} &= \boldsymbol{M}_{n-1}\cdots\boldsymbol{M}_1\boldsymbol{b} \\ \boldsymbol{MAx} &= \boldsymbol{Mb} \end{aligned}$$

  can be solved by back-substitution to obtain solution to original linear system $\boldsymbol{Ax} = \boldsymbol{b}$
- Process just described is called *Gaussian elimination*

LU Factorization

- Product $\boldsymbol{L}_k \boldsymbol{L}_j$ is unit lower triangular if $k < j$, so

  $$\boldsymbol{L} = \boldsymbol{M}^{-1} = \boldsymbol{M}_1^{-1} \cdots \boldsymbol{M}_{n-1}^{-1} = \boldsymbol{L}_1 \cdots \boldsymbol{L}_{n-1}$$

  is unit lower triangular

- By design, $\boldsymbol{MA} = \boldsymbol{U}$ is upper triangular

- So we have

  $$\boldsymbol{A} = \boldsymbol{L}\,\boldsymbol{U}$$

  with $\boldsymbol{L}$ unit lower triangular and $\boldsymbol{U}$ upper triangular

- Thus, Gaussian elimination produces *LU factorization* of matrix into triangular factors

LU Factorization, continued

- Having obtained LU factorization $\boldsymbol{A} = \boldsymbol{LU}$, equation $\boldsymbol{Ax} = \boldsymbol{b}$ becomes

$$\boldsymbol{LUx} = \boldsymbol{b}$$

  which can be solved by

  - solving lower triangular system $\boldsymbol{Ly} = \boldsymbol{b}$ for $\boldsymbol{y}$ by forward-substitution
  - then solving upper triangular system $\boldsymbol{Ux} = \boldsymbol{y}$ for $\boldsymbol{x}$ by back-substitution

- Note that $\boldsymbol{y} = \boldsymbol{Mb}$ is same as transformed right-hand side in Gaussian elimination

- Gaussian elimination and LU factorization are two ways of expressing same solution process

LU Factorization by Gaussian Elimination

```
for k = 1 to n − 1                      { loop over columns }
    if a_kk = 0 then stop               { stop if pivot is zero }
    for i = k + 1 to n                  { compute multipliers
        m_ik = a_ik / a_kk                  for current column }
    end
    for j = k + 1 to n
        for i = k + 1 to n              { apply transformation to
            a_ij = a_ij − m_ik a_kj         remaining submatrix }
        end
    end
end
```

Example: Gaussian Elimination

- Use Gaussian elimination to solve linear system

$$\boldsymbol{A}\boldsymbol{x} = \begin{bmatrix} 2 & 4 & -2 \\ 4 & 9 & -3 \\ -2 & -3 & 7 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 2 \\ 8 \\ 10 \end{bmatrix} = \boldsymbol{b}$$

- To annihilate subdiagonal entries of first column of $\boldsymbol{A}$,

$$\boldsymbol{M}_1\boldsymbol{A} = \begin{bmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} 2 & 4 & -2 \\ 4 & 9 & -3 \\ -2 & -3 & 7 \end{bmatrix} = \begin{bmatrix} 2 & 4 & -2 \\ 0 & 1 & 1 \\ 0 & 1 & 5 \end{bmatrix},$$

$$\boldsymbol{M}_1\boldsymbol{b} = \begin{bmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} 2 \\ 8 \\ 10 \end{bmatrix} = \begin{bmatrix} 2 \\ 4 \\ 12 \end{bmatrix}$$

Example, continued

- To annihilate subdiagonal entry of second column of $\boldsymbol{M}_1\boldsymbol{A}$,

$$\boldsymbol{M}_2\boldsymbol{M}_1\boldsymbol{A} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -1 & 1 \end{bmatrix} \begin{bmatrix} 2 & 4 & -2 \\ 0 & 1 & 1 \\ 0 & 1 & 5 \end{bmatrix} = \begin{bmatrix} 2 & 4 & -2 \\ 0 & 1 & 1 \\ 0 & 0 & 4 \end{bmatrix} = \boldsymbol{U},$$

$$\boldsymbol{M}_2\boldsymbol{M}_1\boldsymbol{b} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -1 & 1 \end{bmatrix} \begin{bmatrix} 2 \\ 4 \\ 12 \end{bmatrix} = \begin{bmatrix} 2 \\ 4 \\ 8 \end{bmatrix} = \boldsymbol{Mb}$$

- We have reduced original system to equivalent upper triangular system

$$\boldsymbol{Ux} = \begin{bmatrix} 2 & 4 & -2 \\ 0 & 1 & 1 \\ 0 & 0 & 4 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 2 \\ 4 \\ 8 \end{bmatrix} = \boldsymbol{Mb}$$

which can now be solved by back-substitution to obtain $\boldsymbol{x} = \begin{bmatrix} -1 \\ 2 \\ 2 \end{bmatrix}$

Example, continued

- To write out LU factorization explicitly,

$$\boldsymbol{L}_1 \boldsymbol{L}_2 = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ -1 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ -1 & 1 & 1 \end{bmatrix} = \boldsymbol{L}$$

so that

$$\boldsymbol{A} = \begin{bmatrix} 2 & 4 & -2 \\ 4 & 9 & -3 \\ -2 & -3 & 7 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ -1 & 1 & 1 \end{bmatrix} \begin{bmatrix} 2 & 4 & -2 \\ 0 & 1 & 1 \\ 0 & 0 & 4 \end{bmatrix} = \boldsymbol{L}\boldsymbol{U}$$

Pivoting

Row Interchanges

- Gaussian elimination breaks down if leading diagonal entry of remaining unreduced matrix is zero at any stage
- Easy fix: if diagonal entry in column k is zero, then interchange row k with some subsequent row having nonzero entry in column k and then proceed as usual
- If there is no nonzero on or below diagonal in column k, then there is nothing to do at this stage, so skip to next column
- Zero on diagonal causes resulting upper triangular matrix $\boldsymbol{U}$ to be singular, but LU factorization can still be completed
- Subsequent back-substitution will fail, however, as it should for singular matrix

Partial Pivoting

- In principle, any nonzero value will do as pivot, but in practice pivot should be chosen to minimize error propagation
- To avoid amplifying previous rounding errors when multiplying remaining portion of matrix by elementary elimination matrix, multipliers should not exceed 1 in magnitude
- This can be accomplished by choosing entry of largest magnitude on or below diagonal as pivot at each stage
- Such *partial pivoting* is essential in practice for numerically stable implementation of Gaussian elimination for general linear systems

$\langle$ interactive example $\rangle$

LU Factorization with Partial Pivoting

- With partial pivoting, each $\boldsymbol{M}_k$ is preceded by permutation $\boldsymbol{P}_k$ to interchange rows to bring entry of largest magnitude into diagonal pivot position

- Still obtain $\boldsymbol{MA} = \boldsymbol{U}$, with $\boldsymbol{U}$ upper triangular, but now

$$\boldsymbol{M} = \boldsymbol{M}_{n-1}\boldsymbol{P}_{n-1}\cdots\boldsymbol{M}_1\boldsymbol{P}_1$$

- $\boldsymbol{L} = \boldsymbol{M}^{-1}$ is still triangular in general sense, but not necessarily *lower* triangular

- Alternatively, we can write

$$\boldsymbol{PA} = \boldsymbol{L}\,\boldsymbol{U}$$

  where $\boldsymbol{P} = \boldsymbol{P}_{n-1}\cdots\boldsymbol{P}_1$ permutes rows of $\boldsymbol{A}$ into order determined by partial pivoting, and now $\boldsymbol{L}$ is lower triangular

Complete Pivoting

- *Complete pivoting* is more exhaustive strategy in which largest entry in entire remaining unreduced submatrix is permuted into diagonal pivot position

- Requires interchanging columns as well as rows, leading to factorization

$$\boldsymbol{PAQ} = \boldsymbol{L}\,\boldsymbol{U}$$

  with $\boldsymbol{L}$ unit lower triangular, $\boldsymbol{U}$ upper triangular, and $\boldsymbol{P}$ and $\boldsymbol{Q}$ permutations

- Numerical stability of complete pivoting is theoretically superior, but pivot search is more expensive than for partial pivoting

- Numerical stability of partial pivoting is more than adequate in practice, so it is almost always used in solving linear systems by Gaussian elimination

Example: Pivoting

- Need for pivoting has nothing to do with whether matrix is singular or nearly singular

- For example,

$$\boldsymbol{A} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

  is nonsingular yet has no LU factorization unless rows are interchanged, whereas

$$\boldsymbol{A} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$$

  is singular yet has LU factorization

Example: Small Pivots

- To illustrate effect of small pivots, consider

$$\boldsymbol{A} = \begin{bmatrix} \epsilon & 1 \\ 1 & 1 \end{bmatrix}$$

  where ϵ is positive number smaller than ϵ_{mach}

- If rows are not interchanged, then pivot is ϵ and multiplier is $-1/\epsilon$, so

$$\boldsymbol{M} = \begin{bmatrix} 1 & 0 \\ -1/\epsilon & 1 \end{bmatrix}, \quad \boldsymbol{L} = \begin{bmatrix} 1 & 0 \\ 1/\epsilon & 1 \end{bmatrix},$$

$$\boldsymbol{U} = \begin{bmatrix} \epsilon & 1 \\ 0 & 1 - 1/\epsilon \end{bmatrix} = \begin{bmatrix} \epsilon & 1 \\ 0 & -1/\epsilon \end{bmatrix}$$

  in floating-point arithmetic, but then

$$\boldsymbol{L}\,\boldsymbol{U} = \begin{bmatrix} 1 & 0 \\ 1/\epsilon & 1 \end{bmatrix} \begin{bmatrix} \epsilon & 1 \\ 0 & -1/\epsilon \end{bmatrix} = \begin{bmatrix} \epsilon & 1 \\ 1 & 0 \end{bmatrix} \neq \boldsymbol{A}$$

Example, continued

- Using small pivot, and correspondingly large multiplier, has caused loss of information in transformed matrix

- If rows interchanged, then pivot is 1 and multiplier is $-\epsilon$, so

$$\boldsymbol{M} = \begin{bmatrix} 1 & 0 \\ -\epsilon & 1 \end{bmatrix}, \quad \boldsymbol{L} = \begin{bmatrix} 1 & 0 \\ \epsilon & 1 \end{bmatrix},$$

$$\boldsymbol{U} = \begin{bmatrix} 1 & 1 \\ 0 & 1-\epsilon \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$$

  in floating-point arithmetic

- Thus,

$$\boldsymbol{LU} = \begin{bmatrix} 1 & 0 \\ \epsilon & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ \epsilon & 1 \end{bmatrix}$$

  which is correct after permutation

Pivoting, continued

- Although pivoting is generally required for stability of Gaussian elimination, pivoting is *not* required for some important classes of matrices

- *Diagonally dominant*

$$\sum_{i=1,\, i\neq j}^{n} |a_{ij}| < |a_{jj}|, \quad j = 1, \ldots, n$$

- *Symmetric positive definite*

$$\boldsymbol{A} = \boldsymbol{A}^T \quad \text{and} \quad \boldsymbol{x}^T \boldsymbol{A} \boldsymbol{x} > 0 \ \text{ for all } \ \boldsymbol{x} \neq \boldsymbol{0}$$

Residual

Residual

- Residual $\boldsymbol{r} = \boldsymbol{b} - \boldsymbol{A}\hat{\boldsymbol{x}}$ for solution $\hat{\boldsymbol{x}}$ computed using Gaussian elimination satisfies

$$\frac{\|\boldsymbol{r}\|}{\|\boldsymbol{A}\| \; \|\hat{\boldsymbol{x}}\|} \leq \frac{\|\boldsymbol{E}\|}{\|\boldsymbol{A}\|} \leq \rho \; n^2 \; \epsilon_{\text{mach}}$$

  where $\boldsymbol{E}$ is backward error in matrix $\boldsymbol{A}$ and *growth factor* ρ is ratio of largest entry of $\boldsymbol{U}$ to largest entry of $\boldsymbol{A}$

- Without pivoting, ρ can be arbitrarily large, so Gaussian elimination without pivoting is *unstable*

- With partial pivoting, ρ can still be as large as 2^{n-1}, but such behavior is extremely rare

Residual, continued

- There is little or no growth in practice, so

$$\frac{\|\boldsymbol{r}\|}{\|\boldsymbol{A}\|\ \|\hat{\boldsymbol{x}}\|} \le \frac{\|\boldsymbol{E}\|}{\|\boldsymbol{A}\|} \lessapprox n\ \epsilon_{\mathrm{mach}}$$

  which means Gaussian elimination with partial pivoting yields small relative residual regardless of conditioning of system

- Thus, small relative residual does *not* necessarily imply computed solution is close to “true” solution unless system is well-conditioned

- Complete pivoting yields even smaller growth factor, but additional margin of stability is not usually worth extra cost

Example: Small Residual

- Use 4-digit decimal arithmetic to solve

$$\begin{bmatrix} 0.913 & 0.659 \\ 0.457 & 0.330 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 0.254 \\ 0.127 \end{bmatrix}$$

- Gaussian elimination with partial pivoting yields triangular system

$$\begin{bmatrix} 0.9130 & 0.6590 \\ 0 & 0.0002 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 0.2540 \\ -0.0001 \end{bmatrix}$$

- Back-substitution then gives solution

$$\hat{\boldsymbol{x}} = \begin{bmatrix} 0.6391 & -0.5 \end{bmatrix}^T$$

- Exact residual norm for this solution is 7.04×10^{-5}, as small as we can expect using 4-digit arithmetic

Example, continued

- But exact solution is

$$\boldsymbol{x} = \begin{bmatrix} 1.00 & 1.00 \end{bmatrix}^T$$

  so error is almost as large as solution

- Cause of this phenomenon is that matrix is nearly singular ($\text{cond}(\boldsymbol{A}) > 10^4$)

- Division that determines x_2 is between two quantities that are both on order of rounding error, and hence result is essentially arbitrary

- When arbitrary value for x_2 is substituted into first equation, value for x_1 is computed so that first equation is satisfied, yielding small residual, but poor solution

Implementing Gaussian Elimination

Implementing Gaussian Elimination

- Gaussian elimination has general form of triple-nested loop

```
for ______________
    for ______________
        for ______________
            a_ij = a_ij − (a_ik/a_kk) a_kj
        end
    end
end
```

- Indices i, j, and k of **for** loops can be taken in any order, for total of $3! = 6$ different arrangements
- These variations have different memory access patterns, which may cause their performance to vary widely on different computers

Uniqueness of LU Factorization

- Despite variations in computing it, LU factorization is unique up to diagonal scaling of factors
- Provided row pivot sequence is same, if we have two LU factorizations $\boldsymbol{PA} = \boldsymbol{LU} = \hat{\boldsymbol{L}}\hat{\boldsymbol{U}}$, then $\hat{\boldsymbol{L}}^{-1}\boldsymbol{L} = \hat{\boldsymbol{U}}\boldsymbol{U}^{-1} = \boldsymbol{D}$ is both lower and upper triangular, hence diagonal
- If both $\boldsymbol{L}$ and $\hat{\boldsymbol{L}}$ are unit lower triangular, then $\boldsymbol{D}$ must be identity matrix, so $\boldsymbol{L} = \hat{\boldsymbol{L}}$ and $\boldsymbol{U} = \hat{\boldsymbol{U}}$
- Uniqueness is made explicit in LDU factorization $\boldsymbol{PA} = \boldsymbol{LDU}$, with $\boldsymbol{L}$ unit lower triangular, $\boldsymbol{U}$ unit upper triangular, and $\boldsymbol{D}$ diagonal

Storage Management

- Elementary elimination matrices $\boldsymbol{M}_k$, their inverses $\boldsymbol{L}_k$, and permutation matrices $\boldsymbol{P}_k$ used in formal description of LU factorization process are *not* formed explicitly in actual implementation

- $\boldsymbol{U}$ overwrites upper triangle of $\boldsymbol{A}$, multipliers in $\boldsymbol{L}$ overwrite strict lower triangle of $\boldsymbol{A}$, and unit diagonal of $\boldsymbol{L}$ need not be stored

- Row interchanges usually are not done explicitly; auxiliary integer vector keeps track of row order in original locations

Complexity of Solving Linear Systems

- LU factorization requires about $n^3/3$ floating-point multiplications and similar number of additions
- Forward- and back-substitution for single right-hand-side vector together require about n^2 multiplications and similar number of additions
- Can also solve linear system by matrix inversion: $\boldsymbol{x} = \boldsymbol{A}^{-1}\boldsymbol{b}$
- Computing $\boldsymbol{A}^{-1}$ is tantamount to solving n linear systems, requiring LU factorization of $\boldsymbol{A}$ followed by n forward- and back-substitutions, one for each column of identity matrix
- Operation count for inversion is about n^3, three times as expensive as LU factorization

Inversion vs. Factorization

- Even with many right-hand sides $\boldsymbol{b}$, inversion never overcomes higher initial cost, since each matrix-vector multiplication $\boldsymbol{A}^{-1}\boldsymbol{b}$ requires n^2 operations, similar to cost of forward- and back-substitution

- Inversion gives less accurate answer; for example, solving $3x = 18$ by division gives $x = 18/3 = 6$, but inversion gives $x = 3^{-1} \times 18 = 0.333 \times 18 = 5.99$ using 3-digit arithmetic

- Matrix inverses often occur as convenient notation in formulas, but explicit inverse is rarely required to implement such formulas

- For example, product $\boldsymbol{A}^{-1}\boldsymbol{B}$ should be computed by LU factorization of $\boldsymbol{A}$, followed by forward- and back-substitutions using each column of $\boldsymbol{B}$

Gauss-Jordan Elimination

- In Gauss-Jordan elimination, matrix is reduced to diagonal rather than triangular form
- Row combinations are used to annihilate entries above as well as below diagonal
- Elimination matrix used for given column vector $\boldsymbol{a}$ is of form

$$\begin{bmatrix} 1 & \cdots & 0 & -m_1 & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & 1 & -m_{k-1} & 0 & \cdots & 0 \\ 0 & \cdots & 0 & 1 & 0 & \cdots & 0 \\ 0 & \cdots & 0 & -m_{k+1} & 1 & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & -m_n & 0 & \cdots & 1 \end{bmatrix} \begin{bmatrix} a_1 \\ \vdots \\ a_{k-1} \\ a_k \\ a_{k+1} \\ \vdots \\ a_n \end{bmatrix} = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ a_k \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

where $m_i = a_i / a_k$, $i = 1, \ldots, n$

Gauss-Jordan Elimination, continued

- Gauss-Jordan elimination requires about $n^3/2$ multiplications and similar number of additions, 50% more expensive than LU factorization
- During elimination phase, same row operations are also applied to right-hand-side vector (or vectors) of system of linear equations
- Once matrix is in diagonal form, components of solution are computed by dividing each entry of transformed right-hand side by corresponding diagonal entry of matrix
- Latter requires only n divisions, but this is not enough cheaper to offset more costly elimination phase

⟨ interactive example ⟩

Updating Solutions

Solving Modified Problems

- If right-hand side of linear system changes but matrix does not, then LU factorization need not be repeated to solve new system

- Only forward- and back-substitution need be repeated for new right-hand side

- This is substantial savings in work, since additional triangular solutions cost only $\mathcal{O}(n^2)$ work, in contrast to $\mathcal{O}(n^3)$ cost of factorization

Sherman-Morrison Formula

- Sometimes refactorization can be avoided even when matrix *does* change

- *Sherman-Morrison formula* gives inverse of matrix resulting from rank-one change to matrix whose inverse is already known

$$(\boldsymbol{A} - \boldsymbol{u}\boldsymbol{v}^T)^{-1} = \boldsymbol{A}^{-1} + \boldsymbol{A}^{-1}\boldsymbol{u}(1 - \boldsymbol{v}^T\boldsymbol{A}^{-1}\boldsymbol{u})^{-1}\boldsymbol{v}^T\boldsymbol{A}^{-1}$$

  where $\boldsymbol{u}$ and $\boldsymbol{v}$ are n-vectors

- Evaluation of formula requires $\mathcal{O}(n^2)$ work (for matrix-vector multiplications) rather than $\mathcal{O}(n^3)$ work required for inversion

Rank-One Updating of Solution

- To solve linear system $(\boldsymbol{A} - \boldsymbol{u}\boldsymbol{v}^T)\boldsymbol{x} = \boldsymbol{b}$ with new matrix, use Sherman-Morrison formula to obtain

$$\begin{aligned} \boldsymbol{x} &= (\boldsymbol{A} - \boldsymbol{u}\boldsymbol{v}^T)^{-1}\boldsymbol{b} \\ &= \boldsymbol{A}^{-1}\boldsymbol{b} + \boldsymbol{A}^{-1}\boldsymbol{u}(1 - \boldsymbol{v}^T\boldsymbol{A}^{-1}\boldsymbol{u})^{-1}\boldsymbol{v}^T\boldsymbol{A}^{-1}\boldsymbol{b} \end{aligned}$$

  which can be implemented by following steps
  - Solve $\boldsymbol{A}\boldsymbol{z} = \boldsymbol{u}$ for $\boldsymbol{z}$, so $\boldsymbol{z} = \boldsymbol{A}^{-1}\boldsymbol{u}$
  - Solve $\boldsymbol{A}\boldsymbol{y} = \boldsymbol{b}$ for $\boldsymbol{y}$, so $\boldsymbol{y} = \boldsymbol{A}^{-1}\boldsymbol{b}$
  - Compute $\boldsymbol{x} = \boldsymbol{y} + ((\boldsymbol{v}^T\boldsymbol{y})/(1 - \boldsymbol{v}^T\boldsymbol{z}))\boldsymbol{z}$

- If $\boldsymbol{A}$ is already factored, procedure requires only triangular solutions and inner products, so only $\mathcal{O}(n^2)$ work and no explicit inverses

Example: Rank-One Updating of Solution

- Consider rank-one modification

$$\begin{bmatrix} 2 & 4 & -2 \\ 4 & 9 & -3 \\ -2 & -1 & 7 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 2 \\ 8 \\ 10 \end{bmatrix}$$

  (with $3, 2$ entry changed) of system whose LU factorization was computed in earlier example

- One way to choose update vectors is

$$\boldsymbol{u} = \begin{bmatrix} 0 \\ 0 \\ -2 \end{bmatrix} \quad \text{and} \quad \boldsymbol{v} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$$

  so matrix of modified system is $\boldsymbol{A} - \boldsymbol{u}\boldsymbol{v}^T$

Example, continued

- Using LU factorization of $\boldsymbol{A}$ to solve $\boldsymbol{Az} = \boldsymbol{u}$ and $\boldsymbol{Ay} = \boldsymbol{b}$,

$$\boldsymbol{z} = \begin{bmatrix} -3/2 \\ 1/2 \\ -1/2 \end{bmatrix} \quad \text{and} \quad \boldsymbol{y} = \begin{bmatrix} -1 \\ 2 \\ 2 \end{bmatrix}$$

- Final step computes updated solution

$$\boldsymbol{x} = \boldsymbol{y} + \frac{\boldsymbol{v}^T \boldsymbol{y}}{1 - \boldsymbol{v}^T \boldsymbol{z}} \boldsymbol{z} = \begin{bmatrix} -1 \\ 2 \\ 2 \end{bmatrix} + \frac{2}{1 - 1/2} \begin{bmatrix} -3/2 \\ 1/2 \\ -1/2 \end{bmatrix} = \begin{bmatrix} -7 \\ 4 \\ 0 \end{bmatrix}$$

- We have thus computed solution to modified system without factoring modified matrix

Improving Accuracy

Scaling Linear Systems

- In principle, solution to linear system is unaffected by diagonal scaling of matrix and right-hand-side vector
- In practice, scaling affects both conditioning of matrix and selection of pivots in Gaussian elimination, which in turn affect numerical accuracy in finite-precision arithmetic
- It is usually best if all entries (or uncertainties in entries) of matrix have about same size
- Sometimes it may be obvious how to accomplish this by choice of measurement units for variables, but there is no foolproof method for doing so in general
- Scaling can introduce rounding errors if not done carefully

Example: Scaling

- Linear system
$$\begin{bmatrix} 1 & 0 \\ 0 & \epsilon \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1 \\ \epsilon \end{bmatrix}$$
has condition number $1/\epsilon$, so is ill-conditioned if ϵ is small

- If second row is multiplied by $1/\epsilon$, then system becomes perfectly well-conditioned

- Apparent ill-conditioning was due purely to poor scaling

- In general, it is usually much less obvious how to correct poor scaling

Iterative Refinement

- Given approximate solution $\boldsymbol{x}_0$ to linear system $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$, compute residual

$$\boldsymbol{r}_0 = \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_0$$

- Now solve linear system $\boldsymbol{A}\boldsymbol{z}_0 = \boldsymbol{r}_0$ and take

$$\boldsymbol{x}_1 = \boldsymbol{x}_0 + \boldsymbol{z}_0$$

as new and "better" approximate solution, since

$$\begin{aligned} \boldsymbol{A}\boldsymbol{x}_1 &= \boldsymbol{A}(\boldsymbol{x}_0 + \boldsymbol{z}_0) = \boldsymbol{A}\boldsymbol{x}_0 + \boldsymbol{A}\boldsymbol{z}_0 \\ &= (\boldsymbol{b} - \boldsymbol{r}_0) + \boldsymbol{r}_0 = \boldsymbol{b} \end{aligned}$$

- Process can be repeated to refine solution successively until convergence, potentially producing solution accurate to full machine precision

Iterative Refinement, continued

- Iterative refinement requires double storage, since both original matrix and its LU factorization are required
- Due to cancellation, residual usually must be computed with higher precision for iterative refinement to produce meaningful improvement
- For these reasons, iterative improvement is often impractical to use routinely, but it can still be useful in some circumstances
- For example, iterative refinement can sometimes stabilize otherwise unstable algorithm

Special Types of Linear Systems

Special Types of Linear Systems

- Work and storage can often be saved in solving linear system if matrix has special properties
- Examples include
  - *Symmetric*: $\boldsymbol{A} = \boldsymbol{A}^T$, $a_{ij} = a_{ji}$ for all i, j
  - *Positive definite*: $\boldsymbol{x}^T \boldsymbol{A} \boldsymbol{x} > 0$ for all $\boldsymbol{x} \neq \boldsymbol{0}$
  - *Band*: $a_{ij} = 0$ for all $|i - j| > \beta$, where β is *bandwidth* of $\boldsymbol{A}$
  - *Sparse*: most entries of $\boldsymbol{A}$ are zero

Symmetric Positive Definite Matrices

- If $\boldsymbol{A}$ is symmetric and positive definite, then LU factorization can be arranged so that $\boldsymbol{U} = \boldsymbol{L}^T$, which gives *Cholesky factorization*

$$\boldsymbol{A} = \boldsymbol{L}\,\boldsymbol{L}^T$$

  where $\boldsymbol{L}$ is lower triangular with positive diagonal entries

- Algorithm for computing it can be derived by equating corresponding entries of $\boldsymbol{A}$ and $\boldsymbol{L}\boldsymbol{L}^T$

- In 2×2 case, for example,

$$\begin{bmatrix} a_{11} & a_{21} \\ a_{21} & a_{22} \end{bmatrix} = \begin{bmatrix} l_{11} & 0 \\ l_{21} & l_{22} \end{bmatrix} \begin{bmatrix} l_{11} & l_{21} \\ 0 & l_{22} \end{bmatrix}$$

  implies

$$l_{11} = \sqrt{a_{11}}, \quad l_{21} = a_{21}/l_{11}, \quad l_{22} = \sqrt{a_{22} - l_{21}^2}$$

Cholesky Factorization

- One way to write resulting algorithm, in which Cholesky factor $\boldsymbol{L}$ overwrites lower triangle of original matrix $\boldsymbol{A}$, is

```
for k = 1 to n                          { loop over columns }
    a_kk = sqrt(a_kk)
    for i = k + 1 to n
        a_ik = a_ik / a_kk              { scale current column }
    end
    for j = k + 1 to n                  { from each remaining column,
        for i = j to n                    subtract multiple
            a_ij = a_ij − a_ik · a_jk     of current column }
        end
    end
end
```

Cholesky Factorization, continued

- Features of Cholesky algorithm for symmetric positive definite matrices
  - All n square roots are of positive numbers, so algorithm is well defined
  - No pivoting is required to maintain numerical stability
  - Only lower triangle of $\boldsymbol{A}$ is accessed, and hence upper triangular portion need not be stored
  - Only $n^3/6$ multiplications and similar number of additions are required
- Thus, Cholesky factorization requires only about half work and half storage compared with LU factorization of general matrix by Gaussian elimination, and also avoids need for pivoting

⟨ interactive example ⟩

Symmetric Indefinite Systems

- For symmetric indefinite $\boldsymbol{A}$, Cholesky factorization is not applicable, and some form of pivoting is generally required for numerical stability
- Factorization of form

$$\boldsymbol{PAP}^T = \boldsymbol{LDL}^T$$

  with $\boldsymbol{L}$ unit lower triangular and $\boldsymbol{D}$ either tridiagonal or block diagonal with 1×1 and 2×2 diagonal blocks, can be computed stably using symmetric pivoting strategy
- In either case, cost is comparable to that of Cholesky factorization

Band Matrices

- Gaussian elimination for band matrices differs little from general case — only ranges of loops change
- Typically matrix is stored in array by diagonals to avoid storing zero entries
- If pivoting is required for numerical stability, bandwidth can grow (but no more than double)
- General purpose solver for arbitrary bandwidth is similar to code for Gaussian elimination for general matrices
- For fixed small bandwidth, band solver can be extremely simple, especially if pivoting is not required for stability

Tridiagonal Matrices

- Consider tridiagonal matrix

$$\boldsymbol{A} = \begin{bmatrix} b_1 & c_1 & 0 & \cdots & 0 \\ a_2 & b_2 & c_2 & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & a_{n-1} & b_{n-1} & c_{n-1} \\ 0 & \cdots & 0 & a_n & b_n \end{bmatrix}$$

- Gaussian elimination without pivoting reduces to

$d_1 = b_1$
for $i = 2$ **to** n
    $m_i = a_i / d_{i-1}$
    $d_i = b_i - m_i c_{i-1}$
end

Tridiagonal Matrices, continued

- LU factorization of $\boldsymbol{A}$ is then given by

$$\boldsymbol{L} = \begin{bmatrix} 1 & 0 & \cdots & \cdots & 0 \\ m_2 & 1 & \ddots & & \vdots \\ 0 & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & m_{n-1} & 1 & 0 \\ 0 & \cdots & 0 & m_n & 1 \end{bmatrix}, \quad \boldsymbol{U} = \begin{bmatrix} d_1 & c_1 & 0 & \cdots & 0 \\ 0 & d_2 & c_2 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & 0 \\ \vdots & & \ddots & d_{n-1} & c_{n-1} \\ 0 & \cdots & \cdots & 0 & d_n \end{bmatrix}$$

General Band Matrices

- In general, band system of bandwidth β requires $\mathcal{O}(\beta n)$ storage, and its factorization requires $\mathcal{O}(\beta^2 n)$ work
- Compared with full system, savings is substantial if $\beta \ll n$

Iterative Methods for Linear Systems

- Gaussian elimination is direct method for solving linear system, producing exact solution in finite number of steps (in exact arithmetic)
- Iterative methods begin with initial guess for solution and successively improve it until desired accuracy attained
- In theory, it might take infinite number of iterations to converge to exact solution, but in practice iterations are terminated when residual is as small as desired
- For some types of problems, iterative methods have significant advantages over direct methods
- We will study specific iterative methods later when we consider solution of partial differential equations

Software for Linear Systems

LINPACK and LAPACK

- `LINPACK` is software package for solving wide variety of systems of linear equations, both general dense systems and special systems, such as symmetric or banded

- Solving linear systems is of such fundamental importance in scientific computing that `LINPACK` has become standard benchmark for comparing performance of computers

- `LAPACK` is more recent replacement for `LINPACK` featuring higher performance on modern computer architectures, including many parallel computers

- Both `LINPACK` and `LAPACK` are available from `Netlib.org`

- Linear system solvers underlying `MATLAB` and Python's `NumPy` and `SciPy` libraries are based on `LAPACK`

BLAS – Basic Linear Algebra Subprograms

- High-level routines in LINPACK and LAPACK are based on lower-level Basic Linear Algebra Subprograms (BLAS)
- BLAS encapsulate basic operations on vectors and matrices so they can be optimized for given computer architecture while high-level routines that call them remain portable
- Higher-level BLAS encapsulate matrix-vector and matrix-matrix operations for better utilization of memory hierarchies such as cache and virtual memory with paging
- Generic versions of BLAS are available from Netlib.org, and many computer vendors provide custom versions optimized for their particular systems

Examples of BLAS

Level	Data	Work	Examples	Function
1	$\mathcal{O}(n)$	$\mathcal{O}(n)$	`saxpy`	Scalar $\times$ vector $+$ vector
			`sdot`	Inner product
			`snrm2`	Euclidean vector norm
2	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	`sgemv`	Matrix-vector product
			`strsv`	Triangular solution
			`sger`	Rank-one update
3	$\mathcal{O}(n^2)$	$\mathcal{O}(n^3)$	`sgemm`	Matrix-matrix product
			`strsm`	Multiple triang. solutions
			`ssyrk`	Rank-k update

Level-3 `BLAS` have more opportunity for data reuse, and hence higher performance, because they perform more operations per data item than lower-level `BLAS`

Summary - Solving Linear Systems

- Solving linear systems is fundamental in scientific computing
- Sensitivity of solution to linear system is measured by cond($\boldsymbol{A}$)
- Triangular linear system is easily solved by successive substitution
- General linear system can be solved by transforming it to triangular form by Gaussian elimination (LU factorization)
- Pivoting is essential for stable implementation of Gaussian elimination
- Specialized algorithms and software are available for solving particular types of linear systems