

# Scientific Computing: Solving Linear Systems

**Aleksandar Donev**  
*Courant Institute, NYU<sup>1</sup>*  
*donev@courant.nyu.edu*

<sup>1</sup>Course MATH-GA.2043 or CSCI-GA.2112, Fall 2020

September 17th and 24th, 2020

# Outline

- 1 Linear Algebra Background
- 2 Conditioning of linear systems
- 3 Gauss elimination and LU factorization
- 4 Beyond GEM
  - Symmetric Positive-Definite Matrices
- 5 Overdetermined Linear Systems
- 6 Sparse Matrices
- 7 Conclusions

# Outline

- 1 Linear Algebra Background
- 2 Conditioning of linear systems
- 3 Gauss elimination and LU factorization
- 4 Beyond GEM
  - Symmetric Positive-Definite Matrices
- 5 Overdetermined Linear Systems
- 6 Sparse Matrices
- 7 Conclusions

# Linear Spaces

- A **vector space**  $\mathcal{V}$  is a set of elements called **vectors**  $\mathbf{x} \in \mathcal{V}$  that may be multiplied by a **scalar**  $c$  and added, e.g.,

$$\mathbf{z} = \alpha\mathbf{x} + \beta\mathbf{y}$$

- I will denote scalars with lowercase letters and vectors with lowercase bold letters.
- Prominent examples of vector spaces are  $\mathbb{R}^n$  (or more generally  $\mathbb{C}^n$ ), but there are many others, for example, the set of polynomials in  $x$ .
- A **subspace**  $\mathcal{V}' \subseteq \mathcal{V}$  of a vector space is a subset such that sums and multiples of elements of  $\mathcal{V}'$  remain in  $\mathcal{V}'$  (i.e., it is closed).
- An example is the set of vectors in  $x \in \mathbb{R}^3$  such that  $x_3 = 0$ .

# Image Space

- Consider a set of  $n$  vectors  $\mathbf{a}_1, \mathbf{a}_2, \dots, \mathbf{a}_n \in \mathbb{R}^m$  and form a **matrix** by putting these vectors as columns

$$\mathbf{A} = [\mathbf{a}_1 \mid \mathbf{a}_2 \mid \dots \mid \mathbf{a}_n] \in \mathbb{R}^{m,n}.$$

- I will denote matrices with bold capital letters, and sometimes write  $\mathbf{A} = [m, n]$  to indicate dimensions.
- The **matrix-vector product** is defined as a **linear combination** of the columns:

$$\mathbf{b} = \mathbf{A}\mathbf{x} = x_1\mathbf{a}_1 + x_2\mathbf{a}_2 + \dots + x_n\mathbf{a}_n \in \mathbb{R}^m.$$

- The **image**  $\text{im}(\mathbf{A})$  or **range**  $\text{range}(\mathbf{A})$  of a matrix is the subspace of all linear combinations of its columns, i.e., the set of all  $\mathbf{b}$ 's. It is also sometimes called the **column space** of the matrix.

# Dimension

- The set of vectors  $\mathbf{a}_1, \mathbf{a}_2, \dots, \mathbf{a}_n$  are **linearly independent** or form a **basis** for  $\mathbb{R}^m$  if  $\mathbf{b} = \mathbf{A}\mathbf{x} = \mathbf{0}$  implies that  $\mathbf{x} = \mathbf{0}$ .
- The **dimension**  $r = \dim \mathcal{V}$  of a vector (sub)space  $\mathcal{V}$  is the number of elements in a basis. This is a property of  $\mathcal{V}$  itself and *not* of the basis, for example,

$$\dim \mathbb{R}^n = n$$

- Given a basis  $\mathbf{A}$  for a vector space  $\mathcal{V}$  of dimension  $n$ , every vector of  $\mathbf{b} \in \mathcal{V}$  can be uniquely represented as the vector of coefficients  $\mathbf{x}$  in that particular basis,

$$\mathbf{b} = x_1 \mathbf{a}_1 + x_2 \mathbf{a}_2 + \dots + x_n \mathbf{a}_n.$$

- A simple and common basis for  $\mathbb{R}^n$  is  $\{\mathbf{e}_1, \dots, \mathbf{e}_n\}$ , where  $\mathbf{e}_k$  has all components zero except for a single 1 in position  $k$ .  
With this choice of basis the coefficients are simply the entries in the vector,  $\mathbf{b} \equiv \mathbf{x}$ .

# Kernel Space

- The dimension of the column space of a matrix is called the **rank** of the matrix  $\mathbf{A} \in \mathbb{R}^{m,n}$ ,

$$r = \text{rank } \mathbf{A} \leq \min(m, n).$$

- If  $r = \min(m, n)$  then the matrix is of **full rank**.
- The **nullspace**  $\text{null}(\mathbf{A})$  or **kernel**  $\ker(\mathbf{A})$  of a matrix  $\mathbf{A}$  is the subspace of vectors  $\mathbf{x}$  for which

$$\mathbf{Ax} = \mathbf{0}.$$

- The dimension of the nullspace is called the **nullity** of the matrix.
- For a basis  $\mathbf{A}$  the nullspace is  $\text{null}(\mathbf{A}) = \{\mathbf{0}\}$  and the nullity is zero.

# Orthogonal Spaces

- An inner-product space is a vector space together with an **inner or dot product**, which must satisfy some properties.
- The standard dot-product in  $\mathbb{R}^n$  is denoted with several different notations:

$$\mathbf{x} \cdot \mathbf{y} = (\mathbf{x}, \mathbf{y}) = \langle \mathbf{x}, \mathbf{y} \rangle = \mathbf{x}^T \mathbf{y} = \sum_{i=1}^n x_i y_i.$$

- For  $\mathbb{C}^n$  we need to add complex conjugates (here  $\star$  denotes a complex conjugate transpose, or **adjoint**),

$$\mathbf{x} \cdot \mathbf{y} = \mathbf{x}^\star \mathbf{y} = \sum_{i=1}^n \bar{x}_i y_i.$$

- Two vectors  $\mathbf{x}$  and  $\mathbf{y}$  are **orthogonal** if  $\mathbf{x} \cdot \mathbf{y} = 0$ .



# Part I of Fundamental Theorem

- One of the most important theorems in linear algebra is that the sum of rank and nullity is equal to the number of columns: For  $\mathbf{A} \in \mathbb{R}^{m,n}$

$$\text{rank } \mathbf{A} + \text{nullity } \mathbf{A} = n.$$

- In addition to the range and kernel spaces of a matrix, two more important vector subspaces for a given matrix  $\mathbf{A}$  are the:
  - **Row space** or **coimage** of a matrix is the column (image) space of its transpose,  $\text{im } \mathbf{A}^T$ .  
*Its dimension is also equal to the the rank.*
  - **Left nullspace** or **cokernel** of a matrix is the nullspace or kernel of its transpose,  $\text{ker } \mathbf{A}^T$ .

# Part II of Fundamental Theorem

- The **orthogonal complement**  $\mathcal{V}^\perp$  or orthogonal subspace of a subspace  $\mathcal{V}$  is the set of all vectors that are orthogonal to every vector in  $\mathcal{V}$ .
- Let  $\mathcal{V}$  be the set of vectors in  $x \in \mathbb{R}^3$  such that  $x_3 = 0$ . Then  $\mathcal{V}^\perp$  is the set of all vectors with  $x_1 = x_2 = 0$ .
- Second fundamental theorem in linear algebra:

$$\text{im } \mathbf{A}^T = (\ker \mathbf{A})^\perp$$

$$\ker \mathbf{A}^T = (\text{im } \mathbf{A})^\perp$$

# Linear Transformation

- A function  $L : \mathcal{V} \rightarrow \mathcal{W}$  mapping from a vector space  $\mathcal{V}$  to a vector space  $\mathcal{W}$  is a **linear function** or a **linear transformation** if

$$L(\alpha \mathbf{v}) = \alpha L(\mathbf{v}) \text{ and } L(\mathbf{v}_1 + \mathbf{v}_2) = L(\mathbf{v}_1) + L(\mathbf{v}_2).$$

- Any linear transformation  $L$  can be represented as a multiplication by a matrix  $\mathbf{L}$

$$L(\mathbf{v}) = \mathbf{L}\mathbf{v}.$$

- For the common bases of  $\mathcal{V} = \mathbb{R}^n$  and  $\mathcal{W} = \mathbb{R}^m$ , the product  $\mathbf{w} = \mathbf{L}\mathbf{v}$  is simply the usual **matrix-vector product**,

$$w_i = \sum_{k=1}^n L_{ik} v_k,$$

which is simply the dot-product between the  $i$ -th row of the matrix and the vector  $\mathbf{v}$ .

# Matrix algebra

$$w_i = (\mathbf{L}\mathbf{v})_i = \sum_{k=1}^n L_{ik} v_k$$

- The composition of two linear transformations  $\mathbf{A} = [m, p]$  and  $\mathbf{B} = [p, n]$  is a **matrix-matrix product**  $\mathbf{C} = \mathbf{AB} = [m, n]$ :

$$\mathbf{z} = \mathbf{A}(\mathbf{B}\mathbf{x}) = \mathbf{A}\mathbf{y} = (\mathbf{AB})\mathbf{x}$$

$$z_i = \sum_{k=1}^n A_{ik} y_k = \sum_{k=1}^p A_{ik} \sum_{j=1}^n B_{kj} x_j = \sum_{j=1}^n \left( \sum_{k=1}^p A_{ik} B_{kj} \right) x_j = \sum_{j=1}^n C_{ij} x_j$$

$$C_{ij} = \sum_{k=1}^p A_{ik} B_{kj}$$

- Matrix-matrix multiplication is **not commutative**,  $\mathbf{AB} \neq \mathbf{BA}$  in general.

# The Matrix Inverse

- A square matrix  $\mathbf{A} = [n, n]$  is **invertible or nonsingular** if there exists a **matrix inverse**  $\mathbf{A}^{-1} = \mathbf{B} = [n, n]$  such that:

$$\mathbf{AB} = \mathbf{BA} = \mathbf{I},$$

where  $\mathbf{I}$  is the identity matrix (ones along diagonal, all the rest zeros).

- The following statements are equivalent for  $\mathbf{A} \in \mathbb{R}^{n,n}$ :
  - $\mathbf{A}$  is **invertible**.
  - $\mathbf{A}$  is **full-rank**,  $\text{rank } \mathbf{A} = n$ .
  - The columns and also the rows are linearly independent and form a **basis** for  $\mathbb{R}^n$ .
  - The **determinant** is nonzero,  $\det \mathbf{A} \neq 0$ .
  - Zero is not an eigenvalue of  $\mathbf{A}$ .

# Matrix Algebra

- Matrix-vector multiplication is just a special case of matrix-matrix multiplication. Note  $\mathbf{x}^T \mathbf{y}$  is a scalar (dot product).

$$\mathbf{C}(\mathbf{A} + \mathbf{B}) = \mathbf{CA} + \mathbf{CB} \text{ and } \mathbf{ABC} = (\mathbf{AB})\mathbf{C} = \mathbf{A}(\mathbf{BC})$$

$$(\mathbf{A}^T)^T = \mathbf{A} \text{ and } (\mathbf{AB})^T = \mathbf{B}^T \mathbf{A}^T$$

$$(\mathbf{A}^{-1})^{-1} = \mathbf{A} \text{ and } (\mathbf{AB})^{-1} = \mathbf{B}^{-1} \mathbf{A}^{-1} \text{ and } (\mathbf{A}^T)^{-1} = (\mathbf{A}^{-1})^T$$

- Instead of **matrix division**, think of multiplication by an inverse:

$$\mathbf{AB} = \mathbf{C} \Rightarrow (\mathbf{A}^{-1}\mathbf{A})\mathbf{B} = \mathbf{A}^{-1}\mathbf{C} \Rightarrow \begin{cases} \mathbf{B} & = \mathbf{A}^{-1}\mathbf{C} \\ \mathbf{A} & = \mathbf{CB}^{-1} \end{cases}$$

# Vector norms

- Norms are the abstraction for the notion of a length or **magnitude**.
- For a vector  $\mathbf{x} \in \mathbb{R}^n$ , the  $p$ -norm is

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

and special cases of interest are:

- 1 The 1-norm ( $L^1$  norm or Manhattan distance),  $\|\mathbf{x}\|_1 = \sum_{i=1}^n |x_i|$
  - 2 The 2-norm ( $L^2$  norm, **Euclidian distance**),  
 $\|\mathbf{x}\|_2 = \sqrt{\mathbf{x} \cdot \mathbf{x}} = \sqrt{\sum_{i=1}^n |x_i|^2}$
  - 3 The  $\infty$ -norm ( $L^\infty$  or maximum norm),  $\|\mathbf{x}\|_\infty = \max_{1 \leq i \leq n} |x_i|$
- 1 Note that all of these norms are inter-related in a finite-dimensional setting.

# Matrix norms

- Matrix norm **induced** by a given vector norm:

$$\|\mathbf{A}\| = \sup_{\mathbf{x} \neq \mathbf{0}} \frac{\|\mathbf{Ax}\|}{\|\mathbf{x}\|} \quad \Rightarrow \quad \|\mathbf{Ax}\| \leq \|\mathbf{A}\| \|\mathbf{x}\|$$

- The last bound holds for matrices as well,  $\|\mathbf{AB}\| \leq \|\mathbf{A}\| \|\mathbf{B}\|$ .

- Special cases of interest are:

- 1 The 1-norm or **column sum norm**,  $\|\mathbf{A}\|_1 = \max_j \sum_{i=1}^n |a_{ij}|$
- 2 The  $\infty$ -norm or **row sum norm**,  $\|\mathbf{A}\|_\infty = \max_i \sum_{j=1}^n |a_{ij}|$
- 3 The 2-norm or **spectral norm**,  $\|\mathbf{A}\|_2 = \sigma_1$  (largest singular value)
- 4 The Euclidian or **Frobenius norm**,  $\|\mathbf{A}\|_F = \sqrt{\sum_{i,j} |a_{ij}|^2}$   
(note this is not an induced norm)



# Outline

- 1 Linear Algebra Background
- 2 Conditioning of linear systems**
- 3 Gauss elimination and LU factorization
- 4 Beyond GEM
  - Symmetric Positive-Definite Matrices
- 5 Overdetermined Linear Systems
- 6 Sparse Matrices
- 7 Conclusions

# Matrices and linear systems

- It is said that 70% or more of applied mathematics research involves solving systems of  $m$  linear equations for  $n$  unknowns:

$$\sum_{j=1}^n a_{ij}x_j = b_i, \quad i = 1, \dots, m.$$

- Linear systems arise directly from **discrete models**, e.g., traffic flow in a city. Or, they may come through representing or more abstract **linear operators** in some finite basis (representation).

Common abstraction:

$$\mathbf{Ax} = \mathbf{b}$$

- Special case: Square invertible matrices,  $m = n$ ,  $\det \mathbf{A} \neq 0$ :

$$\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}.$$

- The goal: Calculate solution  $\mathbf{x}$  given data  $\mathbf{A}, \mathbf{b}$  in the most numerically stable and also efficient way.

# Stability analysis

Perturbations on **right hand side** (rhs) only:

$$\mathbf{A}(\mathbf{x} + \delta\mathbf{x}) = \mathbf{b} + \delta\mathbf{b} \quad \Rightarrow \quad \mathbf{b} + \mathbf{A}\delta\mathbf{x} = \mathbf{b} + \delta\mathbf{b}$$

$$\delta\mathbf{x} = \mathbf{A}^{-1}\delta\mathbf{b} \quad \Rightarrow \quad \|\delta\mathbf{x}\| \leq \|\mathbf{A}^{-1}\| \|\delta\mathbf{b}\|$$

Using the bounds

$$\|\mathbf{b}\| \leq \|\mathbf{A}\| \|\mathbf{x}\| \quad \Rightarrow \quad \|\mathbf{x}\| \geq \|\mathbf{b}\| / \|\mathbf{A}\|$$

the relative error in the solution can be bounded by

$$\frac{\|\delta\mathbf{x}\|}{\|\mathbf{x}\|} \leq \frac{\|\mathbf{A}^{-1}\| \|\delta\mathbf{b}\|}{\|\mathbf{x}\|} \leq \frac{\|\mathbf{A}^{-1}\| \|\delta\mathbf{b}\|}{\|\mathbf{b}\| / \|\mathbf{A}\|} = \kappa(\mathbf{A}) \frac{\|\delta\mathbf{b}\|}{\|\mathbf{b}\|}$$

where the **conditioning number**  $\kappa(\mathbf{A})$  depends on the matrix norm used:

$$\kappa(\mathbf{A}) = \|\mathbf{A}\| \|\mathbf{A}^{-1}\| \geq 1.$$

# Conditioning Number

- The full derivation, not given here, estimates the uncertainty or perturbation in the solution:

$$\frac{\|\delta\mathbf{x}\|}{\|\mathbf{x}\|} \leq \frac{\kappa(\mathbf{A})}{1 - \kappa(\mathbf{A}) \frac{\|\delta\mathbf{A}\|}{\|\mathbf{A}\|}} \left( \frac{\|\delta\mathbf{b}\|}{\|\mathbf{b}\|} + \frac{\|\delta\mathbf{A}\|}{\|\mathbf{A}\|} \right).$$

The **worst-case conditioning** of the linear system is determined by  $\kappa(\mathbf{A})$ .

- Best possible error with rounding unit  $u \approx 10^{-16}$ :

$$\frac{\|\delta\mathbf{x}\|_{\infty}}{\|\mathbf{x}\|_{\infty}} \lesssim 2u\kappa(\mathbf{A}),$$

- Solving an ill-conditioned system,  $\kappa(\mathbf{A}) \gg 1$  (e.g.,  $\kappa = 10^{15}!$ ), should only be done if something special is known.
- The conditioning number can only be **estimated** in practice since  $\mathbf{A}^{-1}$  is not available (see MATLAB's *rcond* function).

# Outline

- 1 Linear Algebra Background
- 2 Conditioning of linear systems
- 3 Gauss elimination and LU factorization**
- 4 Beyond GEM
  - Symmetric Positive-Definite Matrices
- 5 Overdetermined Linear Systems
- 6 Sparse Matrices
- 7 Conclusions

GEM: Eliminating  $x_1$ 

Step 1:

$$\begin{bmatrix} a_{11}^{(1)} & a_{12}^{(1)} & a_{13}^{(1)} \\ a_{21}^{(1)} & a_{22}^{(1)} & a_{23}^{(1)} \\ a_{31}^{(1)} & a_{32}^{(1)} & a_{33}^{(1)} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} b_1^{(1)} \\ b_2^{(1)} \\ b_3^{(1)} \end{bmatrix}$$

$\leftarrow$  Multiply FIRST row by  $l_{21} = \frac{a_{21}^{(1)}}{a_{11}^{(1)}}$   
 $\leftarrow$   $l_{31} = \frac{a_{31}^{(1)}}{a_{11}^{(1)}}$

Eliminate  $x_1$

$$\begin{bmatrix} a_{11}^{(1)} & a_{12}^{(1)} & a_{13}^{(1)} \\ 0 = a_{21}^{(1)} - l_{21} \cdot a_{11}^{(1)} & a_{22}^{(1)} - l_{21} \cdot a_{12}^{(1)} & a_{23}^{(1)} - l_{21} \cdot a_{13}^{(1)} \\ 0 & a_{32}^{(1)} - l_{31} \cdot a_{12}^{(1)} & a_{33}^{(1)} - l_{31} \cdot a_{13}^{(1)} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 - l_{21} \cdot b_1 \\ b_3 - l_{31} \cdot b_1 \end{bmatrix}$$

GEM: Eliminating  $x_2$ 

Step 2:

$$\begin{bmatrix} a_{11}^{(1)} & a_{12}^{(1)} & a_{13}^{(1)} \\ 0 & a_{22}^{(2)} & a_{23}^{(2)} \\ 0 & a_{32}^{(2)} & a_{33}^{(2)} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} b_1^{(2)} \\ b_2^{(2)} \\ b_3^{(3)} \end{bmatrix}$$

done row!  
 ← Multiply second row by  
 $l_{32} = \frac{a_{32}^{(2)}}{a_{22}^{(2)}}$

Eliminate  $x_2$

$$\begin{bmatrix} a_{11}^{(1)} & a_{12}^{(1)} & a_{13}^{(1)} \\ 0 & a_{22}^{(2)} & a_{23}^{(2)} \\ 0 & 0 & a_{33}^{(3)} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} b_1^{(3)} \\ b_2^{(3)} \\ b_3^{(3)} \end{bmatrix}$$

Upper triangular system  
 ← Solve  $x_3 = \frac{b_3^{(3)}}{a_{33}^{(3)}}$

## GEM: Backward substitution

Eliminate  $x_3$  entirely  $\rightarrow$

$$\begin{bmatrix} a_{11}^{(1)} & a_{12}^{(1)} \\ 0 & a_{22}^{(2)} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} b_1^{(3)} - a_{13}^{(1)} x_3 \\ b_2^{(3)} - a_{23}^{(2)} x_3 \end{bmatrix} \approx \tilde{b}$$

solve for  $x_2 = \frac{\tilde{b}}{a_{22}^{(2)}}$ , then  $x_1$ , and done!

Idea: Store the multipliers in the lower triangle of  $A$ :

Matrix at Step  $k$ :

$$\begin{array}{|c|c|} \hline & u^{(k)} \\ \hline L^{(k)} & A^{(k)} \\ \hline \end{array} \quad \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ l_{21} & a_{22}^{(2)} & a_{23}^{(2)} \\ l_{31} & a_{32}^{(2)} & a_{33}^{(2)} \end{bmatrix}$$

Example step 2

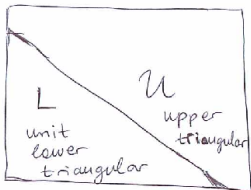
3



# GEM as an $LU$ factorization tool

At the end, we get

$u_{11}$	$u_{12}$	$u_{13}$
1		
$l_{21}$	1	$u_{23}$
$l_{31}$	$l_{32}$	1



- We have actually **factorized  $\mathbf{A}$**  as

$$\mathbf{A} = \mathbf{LU},$$

$\mathbf{L}$  is **unit lower triangular** ( $l_{ii} = 1$  on diagonal), and  $\mathbf{U}$  is **upper triangular**.

- GEM is thus essentially the same as the  $LU$  factorization method.

## GEM in MATLAB

```

% Sample MATLAB code (for learning purposes only, not
function A = MyLU(A)
% LU factorization in-place (overwrite A)
[n,m]=size(A);
if (n ~= m); error('Matrix not square'); end
for k=1:(n-1) % For variable x(k)
    % Calculate multipliers in column k:
    A((k+1):n,k) = A((k+1):n,k) / A(k,k);
    % Note: Pivot element A(k,k) assumed nonzero!
    for j=(k+1):n
        % Eliminate variable x(k):
        A((k+1):n,j) = A((k+1):n,j) - ...
            A((k+1):n,k) * A(k,j);
    end
end
end
end

```

## Pivoting

Zero diagonal entries (pivots) pose a problem  $\rightarrow$  PIVOTING (swapping rows and columns)

$$Ax = b$$

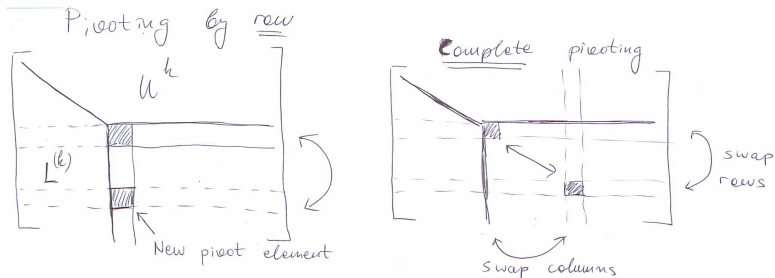
$$\begin{bmatrix} 1 & 1 & 3 \\ 2 & 2 & 2 \\ 3 & 6 & 4 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 5 \\ 6 \\ 13 \end{bmatrix} \Rightarrow \begin{bmatrix} 1 & 1 & 3 \\ \hline 2 & 0 & -4 \\ 3 & 3 & -5 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 1 & 3 \\ \hline 3 & 3 & -5 \\ 2 & 0 & -4 \end{bmatrix} \Rightarrow \begin{bmatrix} 1 & 1 & 3 \\ \hline 3 & 1 & 3 & -5 \\ \hline 2 & 0 & 1 & -4 \end{bmatrix}$$

OBSERVE  
PERMUTED  
LU = A

(4)

# Pivoting during LU factorization



- **Partial (row) pivoting** permutes the rows (equations) of  $\mathbf{A}$  in order to ensure sufficiently large pivots and thus numerical stability:

$$\mathbf{PA} = \mathbf{LU}$$

- Here  $\mathbf{P}$  is a **permutation matrix**, meaning a matrix obtained by permuting rows and/or columns of the identity matrix.
- **Complete pivoting** also permutes columns,  $\mathbf{PAQ} = \mathbf{LU}$ .

# Gauss Elimination Method (GEM)

- GEM is a **general** method for **dense matrices** and is commonly used.
- Implementing GEM efficiently and stably is difficult and we will not discuss it here, since others have done it for you!
- The **LAPACK** public-domain library is the main repository for excellent implementations of dense linear solvers.
- MATLAB uses a highly-optimized variant of GEM by default, mostly based on LAPACK.
- MATLAB does have **specialized solvers** for special cases of matrices, so always look at the help pages!

# Solving linear systems

- Once an  $LU$  factorization is available, solving a linear system is simple:

$$\mathbf{Ax} = \mathbf{LUx} = \mathbf{L}(\mathbf{Ux}) = \mathbf{Ly} = \mathbf{b}$$

so solve for  $\mathbf{y}$  using **forward substitution**.

This was implicitly done in the example above by overwriting  $\mathbf{b}$  to become  $\mathbf{y}$  during the factorization.

- Then, solve for  $\mathbf{x}$  using **backward substitution**

$$\mathbf{Ux} = \mathbf{y}.$$

- If row pivoting is necessary, the same applies but  $\mathbf{L}$  or  $\mathbf{U}$  may be permuted upper/lower triangular matrices,

$$\mathbf{A} = \tilde{\mathbf{L}}\mathbf{U} = (\mathbf{P}^T\mathbf{L})\mathbf{U}.$$

# In MATLAB

- In MATLAB, the **backslash operator** (see help on *mldivide*)

$$x = A \backslash b \approx A^{-1}b,$$

solves the linear system  $\mathbf{Ax} = \mathbf{b}$  using the LAPACK library.

Never use matrix inverse to do this, even if written as such on paper.

- Doing  $x = A \backslash b$  is **equivalent** to performing an *LU* factorization and doing two **triangular solves** (backward and forward substitution):

$$[\tilde{L}, U] = lu(A)$$

$$y = \tilde{L} \backslash b$$

$$x = U \backslash y$$

- This is a carefully implemented **backward stable** pivoted LU factorization, meaning that the returned solution is as accurate as the conditioning number allows.

## GEM Matlab example (1)

```
>> A = [ 1    2    3 ; 4    5    6 ; 7    8    0 ];
>> b=[2 1 -1]';
```

```
>> x=A^(-1)*b; x' % Don't do this!
ans =    -2.5556    2.1111    0.1111
```

```
>> x = A\b; x' % Do this instead
ans =    -2.5556    2.1111    0.1111
```

```
>> linsolve(A,b)' % Even more control
ans =    -2.5556    2.1111    0.1111
```



## GEM Matlab example (2)

```
>> [L,U] = lu(A) % Even better if resolving
```

```
L =    0.1429    1.0000         0
      0.5714    0.5000    1.0000
      1.0000         0         0
U =    7.0000    8.0000         0
        0    0.8571    3.0000
        0         0    4.5000
```

```
>> norm(L*U-A, inf)
```

```
ans =    0
```

```
>> y = L\b;
```

```
>> x = U\y; x'
```

```
ans =   -2.5556    2.1111    0.1111
```

# Backwards Stability

- Even though we cannot get  $\mathbf{x}$  correctly for ill-conditioned linear systems, we can still get an (not *the* one!)  $\mathbf{x}$  that is a solution of the equation to almost machine precision.
- This sort of **backward stability** means that there is a problem nearby the original problem such that the answer we compute  $\hat{\mathbf{x}}$  is the solution of that “perturbed” problem,

$$(\mathbf{A} + \delta\mathbf{A}) \hat{\mathbf{x}} = \mathbf{b} + \delta\mathbf{b}.$$

- A backwards stable method gives a **residual**  $\mathbf{r} = \mathbf{Ax} - \mathbf{b}$  that is zero to within the rounding unit  $u \approx 10^{-16}$ ,

$$\frac{\|\mathbf{Ax} - \mathbf{b}\|}{\|\mathbf{b}\|} \sim \frac{\|\mathbf{Ax} - \mathbf{b}\|}{\|\mathbf{Ax}\|} \sim u,$$

- Observe that the conditioning number of the matrix does not enter here, it can be large!

# Backwards Stability contd.

- Gaussian elimination with partial pivoting is almost always backwards stable in practice, but one can always check the residual after computing the answer (**always good practice** to confirm you solved the problem you thought you solved!)
- Specifically, if we compute the LU factorization we are guaranteed that

$$\mathbf{A} + \delta\mathbf{A} = \mathbf{LU} \quad \text{where} \quad \frac{\|\delta\mathbf{A}\|}{\|\mathbf{A}\|} \leq Cu$$

where  $C$  is some modest constant that depends *polynomially* on the number of unknowns (not exponentially).

- Complete pivoting is rarely used in practice because it is expensive, even though it will give better guarantees.

# Cost estimates for GEM

- For forward or backward substitution, at step  $k$  there are  $\sim (n - k)$  multiplications and subtractions, plus a few divisions.  
The total over all  $n$  steps is

$$\sum_{k=1}^n (n - k) = \frac{n(n - 1)}{2} \approx \frac{n^2}{2}$$

subtractions and multiplications, giving a total of  $O(n^2)$  **floating-point operations** (FLOPs).

- The LU factorization itself costs a lot more,  $O(n^3)$ ,

$$\text{FLOPS} \approx \frac{2n^3}{3},$$

and the triangular solves are negligible for large systems.

- When many linear systems need to be solved with the same  $\mathbf{A}$  the **factorization can be reused**.

# Outline

- 1 Linear Algebra Background
- 2 Conditioning of linear systems
- 3 Gauss elimination and LU factorization
- 4 Beyond GEM**
  - Symmetric Positive-Definite Matrices
- 5 Overdetermined Linear Systems
- 6 Sparse Matrices
- 7 Conclusions

# Matrix Rescaling and Reordering

- Pivoting is not always sufficient to ensure lack of roundoff problems. In particular, **large variations** among the entries in **A should be avoided**.
- This can usually be remedied by changing the physical units for **x** and **b** to be the **natural units**  $\mathbf{x}_0$  and  $\mathbf{b}_0$ .
- **Rescaling** the unknowns and the equations is generally a good idea even if not necessary:

$$\mathbf{x} = \mathbf{D}_x \tilde{\mathbf{x}} = \text{Diag}\{\mathbf{x}_0\} \tilde{\mathbf{x}} \text{ and } \mathbf{b} = \mathbf{D}_b \tilde{\mathbf{b}} = \text{Diag}\{\mathbf{b}_0\} \tilde{\mathbf{b}}.$$

$$\mathbf{Ax} = \mathbf{AD}_x \tilde{\mathbf{x}} = \mathbf{D}_b \tilde{\mathbf{b}} \quad \Rightarrow \quad (\mathbf{D}_b^{-1} \mathbf{AD}_x) \tilde{\mathbf{x}} = \tilde{\mathbf{b}}$$

- The **rescaled matrix**  $\tilde{\mathbf{A}} = \mathbf{D}_b^{-1} \mathbf{AD}_x$  should have a better conditioning.
- Also note that **reordering the variables** from most important to least important may also help.

# Efficiency of Solution

$$\mathbf{Ax} = \mathbf{b}$$

- The most appropriate algorithm really depends on the properties of the matrix  $\mathbf{A}$ :
  - General **dense matrices**, where the entries in  $\mathbf{A}$  are mostly non-zero and nothing special is known: Use  $LU$  factorization.
  - **Symmetric** ( $a_{ij} = a_{ji}$ ) and also **positive-definite** matrices.
  - General **sparse matrices**, where only a small fraction of  $a_{ij} \neq 0$ .
  - Special **structured sparse matrices**, arising from specific physical properties of the underlying system.
- It is also important to consider **how many times** a linear system with the same or related matrix or right hand side needs to be solved.

# Positive-Definite Matrices

- A real symmetric matrix  $\mathbf{A}$  is positive definite iff (if and only if):
  - ① All of its eigenvalues are real (follows from symmetry) and positive.
  - ②  $\forall \mathbf{x} \neq \mathbf{0}, \mathbf{x}^T \mathbf{A} \mathbf{x} > 0$ , i.e., the quadratic form defined by the matrix  $\mathbf{A}$  is convex.
  - ③ There exists a *unique* lower triangular  $\mathbf{L}$ ,  $L_{ii} > 0$ ,

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

termed the **Cholesky factorization** of  $\mathbf{A}$  (symmetric  $LU$  factorization).

- ① For Hermitian complex matrices just replace transposes with adjoints (conjugate transpose), e.g.,  $\mathbf{A}^T \rightarrow \mathbf{A}^*$  (or  $\mathbf{A}^H$  in the book).



# Cholesky Factorization

- The MATLAB built in function

$$R = chol(A)$$

gives the Cholesky factorization and is a good way to **test for positive-definiteness**.

- The cost of a Cholesky factorization is about half the cost of  $LU$  factorization,  $n^3/3$  FLOPS.
- Solving linear systems is as for  $LU$  factorization, replacing  $\mathbf{U}$  with  $\mathbf{L}^T$ .
- For Hermitian/symmetric matrices with positive diagonals MATLAB tries a Cholesky factorization first, *before* resorting to  $LU$  factorization with pivoting.

# Special Matrices in MATLAB

- MATLAB recognizes (i.e., tests for) some special matrices automatically: banded, permuted lower/upper triangular, symmetric, Hessenberg, but **not** sparse.
- In MATLAB one may specify a matrix **B** instead of a single right-hand side vector **b**.
- The MATLAB function

$$X = \text{linsolve}(A, B, \text{opts})$$

allows one to specify certain properties that speed up the solution (triangular, upper Hessenberg, symmetric, positive definite, none), and also estimates the condition number along the way.

- Use *linsolve* instead of backslash if you know (for sure!) something about your matrix.

# Outline

- 1 Linear Algebra Background
- 2 Conditioning of linear systems
- 3 Gauss elimination and LU factorization
- 4 Beyond GEM
  - Symmetric Positive-Definite Matrices
- 5 Overdetermined Linear Systems**
- 6 Sparse Matrices
- 7 Conclusions

# Non-Square Matrices

- In the case of **over-determined** (more equations than unknowns) or **under-determined** (more unknowns than equations), the solution to linear systems in general becomes **non-unique**.
- One must first define what is meant by a solution, and the common definition is to use a **least-squares formulation**:

$$\mathbf{x}^* = \arg \min_{\mathbf{x} \in \mathbb{R}^n} \|\mathbf{Ax} - \mathbf{b}\| = \arg \min_{\mathbf{x} \in \mathbb{R}^n} \Phi(\mathbf{x})$$

where the choice of the  $L_2$  norm leads to:

$$\Phi(\mathbf{x}) = (\mathbf{Ax} - \mathbf{b})^T (\mathbf{Ax} - \mathbf{b}).$$

- Over-determined systems,  $m > n$ , can be thought of as **fitting a linear model (linear regression)**:  
The unknowns  $\mathbf{x}$  are the coefficients in the fit, the input data is in  $\mathbf{A}$  (one column per measurement), and the output data (observables) are in  $\mathbf{b}$ .

# Normal Equations

- It can be shown that the least-squares solution satisfies:

$$\nabla\Phi(\mathbf{x}) = \mathbf{A}^T [2(\mathbf{Ax} - \mathbf{b})] = \mathbf{0} \text{ (critical point)}$$

- This gives the square linear system of **normal equations**

$$(\mathbf{A}^T \mathbf{A}) \mathbf{x}^* = \mathbf{A}^T \mathbf{b}.$$

- If  $\mathbf{A}$  is of full rank,  $\text{rank}(\mathbf{A}) = n$ , it can be shown that  $\mathbf{A}^T \mathbf{A}$  is positive definite, and Cholesky factorization can be used to solve the normal equations.
- Multiplying  $\mathbf{A}^T$  ( $n \times m$ ) and  $\mathbf{A}$  ( $m \times n$ ) takes  $n^2$  dot-products of length  $m$ , so  $O(mn^2)$  operations

# Problems with the normal equations

$$(\mathbf{A}^T \mathbf{A}) \mathbf{x}^* = \mathbf{A}^T \mathbf{b}.$$

- The conditioning number of the normal equations is

$$\kappa(\mathbf{A}^T \mathbf{A}) = [\kappa(\mathbf{A})]^2$$

- Furthermore, roundoff can cause  $\mathbf{A}^T \mathbf{A}$  to no longer appear as positive-definite and the Cholesky factorization will fail.
- If the normal equations are ill-conditioned, another approach is needed.

# The $QR$ factorization

- For nonsquare or ill-conditioned matrices of **full-rank**  $r = n \leq m$ , the  $LU$  factorization can be replaced by the  $QR$  factorization:

$$\mathbf{A} = \mathbf{QR}$$

$$[m \times n] = [m \times n][n \times n]$$

where  $\mathbf{Q}$  has **orthogonal columns**,  $\mathbf{Q}^T \mathbf{Q} = \mathbf{I}_n$ , and  $\mathbf{R}$  is a **non-singular upper triangular** matrix.

- Observe that orthogonal / unitary matrices are **well-conditioned** ( $\kappa_2 = 1$ ), so the  $QR$  factorization is numerically better (but also more expensive!) than the  $LU$  factorization.
- For matrices **not of full rank** there are modified  $QR$  factorizations but **the SVD decomposition is better** (next class).
- In MATLAB, the  $QR$  factorization can be computed using  $qr$  (with column pivoting).

Solving Linear Systems via  $QR$  factorization

$$(\mathbf{A}^T \mathbf{A}) \mathbf{x}^* = \mathbf{A}^T \mathbf{b} \text{ where } \mathbf{A} = \mathbf{QR}$$

- Observe that  $\mathbf{R}$  is the Cholesky factor of the matrix in the normal equations:

$$\mathbf{A}^T \mathbf{A} = \mathbf{R}^T (\mathbf{Q}^T \mathbf{Q}) \mathbf{R} = \mathbf{R}^T \mathbf{R}$$

$$(\mathbf{R}^T \mathbf{R}) \mathbf{x}^* = (\mathbf{R}^T \mathbf{Q}^T) \mathbf{b} \Rightarrow \mathbf{x}^* = \mathbf{R}^{-1} (\mathbf{Q}^T \mathbf{b})$$

which amounts to solving a triangular system with matrix  $\mathbf{R}$ .

- This calculation turns out to be much **more numerically stable** against roundoff than forming the normal equations (and has similar cost).



# Computing the $QR$ Factorization

- The  $QR$  factorization is closely-related to the **orthogonalization** of a set of  $n$  vectors (columns)  $\{\mathbf{a}_1, \mathbf{a}_2, \dots, \mathbf{a}_n\}$  in  $\mathbb{R}^m$ , which is a common problem in numerical computing.
- Classical approach is the **Gram-Schmidt method**: To make a vector  $\mathbf{b}$  orthogonal to  $\mathbf{a}$  do:

$$\tilde{\mathbf{b}} = \mathbf{b} - (\mathbf{b} \cdot \mathbf{a}) \frac{\mathbf{a}}{(\mathbf{a} \cdot \mathbf{a})}$$

- Repeat this in sequence: Start with  $\tilde{\mathbf{a}}_1 = \mathbf{a}_1$ , then make  $\tilde{\mathbf{a}}_2$  orthogonal to  $\tilde{\mathbf{a}}_1 = \mathbf{a}_1$ , then make  $\tilde{\mathbf{a}}_3$  orthogonal to  $\text{span}(\tilde{\mathbf{a}}_1, \tilde{\mathbf{a}}_2) = \text{span}(\mathbf{a}_1, \mathbf{a}_2)$ :

$$\tilde{\mathbf{a}}_1 = \mathbf{a}_1$$

$$\tilde{\mathbf{a}}_2 = \mathbf{a}_2 - (\mathbf{a}_2 \cdot \mathbf{a}_1) \frac{\mathbf{a}_1}{(\mathbf{a}_1 \cdot \mathbf{a}_1)}$$

$$\tilde{\mathbf{a}}_3 = \mathbf{a}_3 - (\mathbf{a}_3 \cdot \mathbf{a}_1) \frac{\mathbf{a}_1}{(\mathbf{a}_1 \cdot \mathbf{a}_1)} - (\mathbf{a}_3 \cdot \mathbf{a}_2) \frac{\mathbf{a}_2}{(\mathbf{a}_2 \cdot \mathbf{a}_2)}$$

# Gram-Schmidt Orthogonalization

- More efficient formula (**standard Gram-Schmidt**):

$$\tilde{\mathbf{a}}_{k+1} = \mathbf{a}_{k+1} - \sum_{j=1}^k (\mathbf{a}_{k+1} \cdot \mathbf{q}_j) \mathbf{q}_j, \quad \mathbf{q}_{k+1} = \frac{\tilde{\mathbf{a}}_{k+1}}{\|\tilde{\mathbf{a}}_{k+1}\|},$$

with cost  $\approx 2mn^2$  FLOPS but is **not numerically stable** against roundoff errors (**loss of orthogonality**).

- In the standard method we make each vector orthogonal to all previous vectors. A **numerically stable** alternative is the **modified Gram-Schmidt**, in which we take each vector and modify all following vectors (not previous ones) to be orthogonal to it (so the sum above becomes  $\sum_{j=k+1}^m$ ).
- As we saw in previous lecture, a small rearrangement of mathematically-equivalent approaches can produce a much more robust numerical method.

# Outline

- 1 Linear Algebra Background
- 2 Conditioning of linear systems
- 3 Gauss elimination and LU factorization
- 4 Beyond GEM
  - Symmetric Positive-Definite Matrices
- 5 Overdetermined Linear Systems
- 6 Sparse Matrices**
- 7 Conclusions

# Sparse Matrices

- A matrix where a substantial fraction of the entries are zero is called a **sparse matrix**. The difference with **dense matrices** is that only the nonzero entries are stored in computer memory.
- Exploiting sparsity is important for **large matrices** (what is large depends on the computer).
- The structure of a sparse matrix refers to the set of indices  $i, j$  such that  $a_{ij} > 0$ , and is visualized in MATLAB using *spy*.
- The structure of sparse matrices comes from the nature of the problem, e.g., in an inter-city road transportation problem it corresponds to the pairs of cities connected by a road.
- In fact, just counting the number of nonzero elements is not enough: the **sparsity structure** is the most important property that determines the best method.

# Banded Matrices

- **Banded matrices** are a very special but common type of sparse matrix, e.g., **tridiagonal matrices**

$$\begin{bmatrix} a_1 & c_1 & & \mathbf{0} \\ b_2 & a_2 & \ddots & \\ & \ddots & \ddots & c_{n-1} \\ \mathbf{0} & & b_n & a_n \end{bmatrix}$$

- There exist special techniques for banded matrices that are much faster than the general case, e.g, only  $8n$  FLOPS and no additional memory for tridiagonal matrices.
- A general matrix should be considered sparse if it has **sufficiently many zeros** that exploiting that fact is advantageous: usually only the case for **large matrices** (what is large?)!

# Sparse Matrices

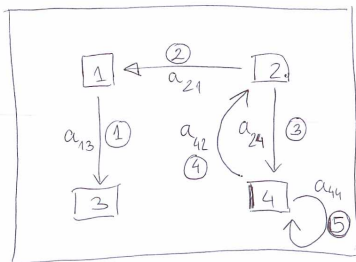
$$A = \begin{bmatrix} 0 & 0 & \textcircled{1} & 0 \\ \textcircled{2} & 0 & 0 & \textcircled{3} \\ 0 & 0 & 0 & 0 \\ 0 & \textcircled{4} & 0 & \textcircled{5} \end{bmatrix} \begin{matrix} \boxed{1} \\ \boxed{2} \\ \boxed{3} \\ \boxed{4} \end{matrix}$$

$\boxed{1}$      $\boxed{2}$      $\boxed{3}$      $\boxed{4}$

Sparse  
matrix

DIRECTED

Graph  
representation:



→ NODES are variables  
(VERTICES) or equations

→ ARCS (EDGES) are  
the non-zeros

UNDIRECTED GRAPH FOR  
SYMMETRIC MATRICES  $\textcircled{1}$

## Sparse matrices in MATLAB

```
>> A = sparse( [1 2 2 4 4], [3 1 4 2 3], 1:5 )
```

```
A =
```

```
(2,1)      2
```

```
(4,2)      4
```

```
(1,3)      1
```

```
(4,3)      5
```

```
(2,4)      3
```

```
>> nnz(A) % Number of non-zeros
```

```
ans =      5
```

```
>> whos A
```

```
  A          4x4          120  double  sparse
```

```
>> A = sparse([], [], [], 4, 4, 5); % Pre-allocate memory
```

```
>> A(2,1)=2; A(4,2)=4; A(1,3)=1; A(4,3)=5; A(2,4)=3;
```

# Sparse matrix factorization

```
>> B=sprand(4,4,0.25); % Density of 25%
```

```
>> full(B)
```

```
ans =
```

```

      0      0      0      0.7655
      0      0.7952      0      0
      0      0.1869      0      0
0.4898      0      0      0
```

```
>> B=sprand(100,100,0.1); spy(B)
```

```
>> [L,U,P]=lu(B); spy(L)
```

```
>> p = symrcm(B); % Permutation to reorder the rows and
```

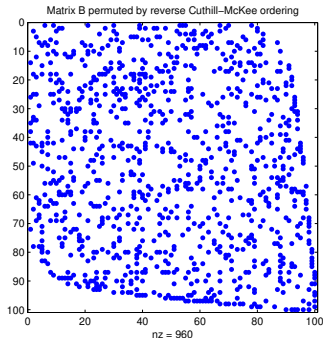
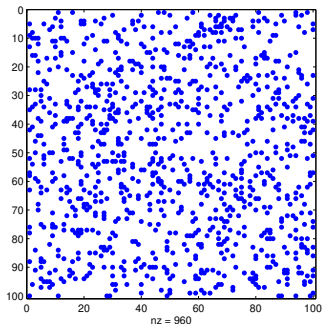
```
>> PBP=B(p,p); spy(PBP);
```

```
>> [L,U,P]=lu(PBP); spy(L);
```



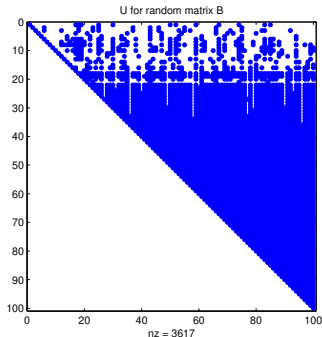
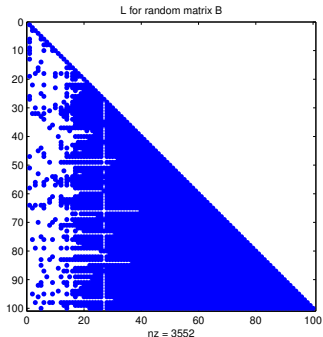
# Random matrix **B**

The MATLAB function `spy` shows where the nonzeros are (left), and what reordering does (right)



# $LU$ factors of random matrix **B**

Fill-in (generation of lots of nonzeros) is large for a random sparse matrix.  
Reordering helps only a bit.



# Fill-In

- There are general techniques for dealing with sparse matrices such as **sparse LU factorization**. How well they work depends on the structure of the matrix.
- When factorizing sparse matrices, the factors, e.g., **L** and **U**, can be much less sparse than **A**: **fill-in**.
- Pivoting (**reordering** of variables and equations) has a dual, sometimes conflicting goal:
  - 1 Reduce fill-in, i.e., **improve memory use**.
  - 2 Reduce roundoff error, i.e., **improve stability**. Typically some **threshold pivoting** is used only when needed.
- For many sparse matrices there is a large fill-in and **iterative methods** are required.

# Why iterative methods?

- Direct solvers are great for dense matrices and are implemented very well on modern machines.
- **Fill-in** is a major problem for certain sparse matrices and leads to extreme memory requirements.
- Some matrices appearing in practice are **too large** to even be represented explicitly (e.g., the Google matrix).
- Often linear systems only need to be **solved approximately**, for example, the linear system itself may be a linear approximation to a nonlinear problem.
- Direct solvers are much harder to implement and use on (massively) **parallel computers**.

# Stationary Linear Iterative Methods

- In iterative methods the core computation is **iterative matrix-vector multiplication** starting from an **initial guess**  $\mathbf{x}^{(0)}$ .
- Prototype is the **linear recursion**:

$$\mathbf{x}^{(k+1)} = \mathbf{B}\mathbf{x}^{(k)} + \mathbf{f},$$

where  $\mathbf{B}$  is an **iteration matrix** somehow related to  $\mathbf{A}$  (many different choices/algorithms exist).

- For this method to be **consistent**, we must have that the actual solution  $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$  is a **stationary point** of the iteration:

$$\mathbf{x} = \mathbf{B}\mathbf{x} + \mathbf{f} \quad \Rightarrow \quad \mathbf{A}^{-1}\mathbf{b} = \mathbf{B}\mathbf{A}^{-1}\mathbf{b} + \mathbf{f}$$

$$\mathbf{f} = \mathbf{A}^{-1}\mathbf{b} - \mathbf{B}\mathbf{A}^{-1}\mathbf{b} = (\mathbf{I} - \mathbf{B})\mathbf{x}$$

# Simple Fixed-Point Iteration

- If we just pick a matrix  $\mathbf{B}$ , in general we cannot easily figure out what  $\mathbf{f}$  needs to be since this requires knowing the solution we are after,

$$\mathbf{f} = (\mathbf{I} - \mathbf{B})\mathbf{x} = (\mathbf{I} - \mathbf{B})\mathbf{A}^{-1}\mathbf{b}$$

- But what if we choose  $\mathbf{I} - \mathbf{B} = \mathbf{A}$ ? Then we get

$$\mathbf{f} = \mathbf{A}\mathbf{A}^{-1}\mathbf{b} = \mathbf{b}$$

which we know.

- This leads us to this **fixed-point iteration** is an iterative method:

$$\mathbf{x}^{(k+1)} = (\mathbf{I} - \mathbf{A})\mathbf{x}^{(k)} + \mathbf{b}$$

## Side-note: Fixed-Point Iteration

- A naive but often successful method for solving

$$x = f(x)$$

is the **fixed-point iteration**

$$x_{n+1} = f(x_n).$$

- In the case of a linear system, consider rewriting  $\mathbf{Ax} = \mathbf{b}$  as:

$$\mathbf{x} = (\mathbf{I} - \mathbf{A})\mathbf{x} + \mathbf{b}$$

- Fixed-point iteration gives the consistent iterative method

$$\mathbf{x}^{(k+1)} = (\mathbf{I} - \mathbf{A})\mathbf{x}^{(k)} + \mathbf{b}$$

which is the same as we already derived differently.

# Convergence of simple iterative methods

- For this method to be **stable**, and thus **convergent**, the error  $\mathbf{e}^{(k)} = \mathbf{x}^{(k)} - \mathbf{x}$  must decrease:

$$\mathbf{e}^{(k+1)} = \mathbf{x}^{(k+1)} - \mathbf{x} = \mathbf{B}\mathbf{x}^{(k)} + \mathbf{f} - \mathbf{x} = \mathbf{B}(\mathbf{x} + \mathbf{e}^{(k)}) + (\mathbf{I} - \mathbf{B})\mathbf{x} - \mathbf{x} = \mathbf{B}\mathbf{e}^{(k)}$$

- We saw that the error propagates from iteration to iteration as

$$\mathbf{e}^{(k)} = \mathbf{B}^k \mathbf{e}^{(0)}.$$

- When does this converge? Taking norms,

$$\|\mathbf{e}^{(k)}\| \leq \|\mathbf{B}\|^k \|\mathbf{e}^{(0)}\|$$

which means that  $\|\mathbf{B}\| < 1$  is a **sufficient condition** for convergence.

- More precisely,  $\lim_{k \rightarrow \infty} \mathbf{e}^{(k)} = \mathbf{0}$  for any  $\mathbf{e}^{(0)}$  iff  $\mathbf{B}^k \rightarrow \mathbf{0}$ .



# Spectral Radius

- Theorem: The simple iterative method converges iff the **spectral radius** of the iteration matrix is less than unity:

$$\rho(\mathbf{B}) < 1.$$

- The **spectral radius**  $\rho(\mathbf{A})$  of a matrix  $\mathbf{A}$  can be thought of as the smallest consistent matrix norm

$$\rho(\mathbf{A}) = \max_{\lambda} |\lambda| \leq \|\mathbf{A}\|$$

- The spectral radius often **determines convergence of iterative schemes** for linear systems and eigenvalues and even methods for solving PDEs because it estimates the asymptotic rate of error propagation:

$$\rho(\mathbf{A}) = \lim_{k \rightarrow \infty} \|\mathbf{A}^k\|^{1/k}$$

# Termination

- The iterations of an iterative method can be terminated when:

- The **residual** becomes small,

$$\|\mathbf{r}^{(k)}\| = \|\mathbf{Ax}^{(k)} - \mathbf{b}\| \leq \varepsilon \|\mathbf{b}\|$$

This is good for well-conditioned systems.

- The solution  $\mathbf{x}^{(k)}$  stops changing, i.e., the **increment** becomes small,

$$[1 - \rho(\mathbf{B})] \|\mathbf{e}^{(k)}\| \leq \|\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}\| \leq \varepsilon \|\mathbf{b}\|,$$

which can be shown to be good if convergence is rapid.

- Usually a careful **combination** of the two strategies is employed along with some **safeguards**.

# Preconditioning

- The fixed-point iteration is consistent but it may not converge or may converge very slowly

$$\mathbf{x}^{(k+1)} = (\mathbf{I} - \mathbf{A})\mathbf{x}^{(k)} + \mathbf{b}.$$

- As a way to speed it up, consider having a **good approximate solver**

$$\mathbf{P}^{-1} \approx \mathbf{A}^{-1}$$

called the **preconditioner** ( $\mathbf{P}$  is the preconditioning matrix), and transform

$$\mathbf{P}^{-1}\mathbf{A}\mathbf{x} = \mathbf{P}^{-1}\mathbf{b}$$

- Now apply fixed-point iteration to this modified system:

$$\mathbf{x}^{(k+1)} = (\mathbf{I} - \mathbf{P}^{-1}\mathbf{A})\mathbf{x}^{(k)} + \mathbf{P}^{-1}\mathbf{b},$$

which now has an iteration matrix  $\mathbf{I} - \mathbf{P}^{-1}\mathbf{A} \approx \mathbf{0}$ , which means more **rapid convergence**.

# Preconditioned Iteration

$$\mathbf{x}^{(k+1)} = (\mathbf{I} - \mathbf{P}^{-1}\mathbf{A}) \mathbf{x}^{(k)} + \mathbf{P}^{-1}\mathbf{b}$$

- In practice, we solve linear systems with the matrix  $\mathbf{P}$  instead of inverting it:

$$\mathbf{P}\mathbf{x}^{(k+1)} = (\mathbf{P} - \mathbf{A})\mathbf{x}^{(k)} + \mathbf{b} = \mathbf{P}\mathbf{x}^{(k)} + \mathbf{r}^{(k)},$$

where  $\mathbf{r}^{(k)} = \mathbf{b} - \mathbf{A}\mathbf{x}^{(k)}$  is the **residual vector**.

- Finally, we obtain the usual form of a **preconditioned stationary iterative solver**

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \mathbf{P}^{-1}\mathbf{r}^{(k)}.$$

- Note that convergence will be faster if we have a **good initial guess**  $\mathbf{x}^{(0)}$ .

# Outline

- 1 Linear Algebra Background
- 2 Conditioning of linear systems
- 3 Gauss elimination and LU factorization
- 4 Beyond GEM
  - Symmetric Positive-Definite Matrices
- 5 Overdetermined Linear Systems
- 6 Sparse Matrices
- 7 Conclusions

# Conclusions/Summary

- The conditioning of a linear system  $\mathbf{Ax} = \mathbf{b}$  is determined by the condition number

$$\kappa(\mathbf{A}) = \|\mathbf{A}\| \|\mathbf{A}^{-1}\| \geq 1$$

- Gauss elimination can be used to solve general square linear systems and also produces a factorization  $\mathbf{A} = \mathbf{LU}$ .
- Partial pivoting is often necessary to ensure numerical stability during GEM and leads to  $\mathbf{PA} = \mathbf{LU}$  or  $\mathbf{A} = \tilde{\mathbf{L}}\mathbf{U}$ .
- MATLAB has excellent linear solvers based on well-known public domain libraries like LAPACK. Use them!

# Conclusions/Summary

- For symmetric positive definite matrices the Cholesky factorization  $\mathbf{A} = \mathbf{L}\mathbf{L}^T$  is preferred and does not require pivoting.
- The  $QR$  factorization is a numerically-stable method for solving **full-rank non-square systems**.
- **Sparse matrices** deserve special treatment but the details depend on the specific field of application.
- In particular, special sparse **matrix reordering** methods or iterative systems are often required.
- When **sparse direct methods** fail due to memory or other requirements, **iterative methods** are used instead.