Scientific Computing & Advanced Programming Lab

SAURAV SAMANTARAY

Department of Mathematics
IIT Madras

Lecture-Slides-6

sauravsray@iitm.ac.in

Gaussian Elimination

- We want to solve

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

where

$$A \in \mathbb{R}^{n\times n}, \qquad \mathbf{x}, \mathbf{b} \in \mathbb{R}^{n}.$$

Gaussian Elimination

- We want to solve

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

where

$$A \in \mathbb{R}^{n \times n}, \qquad \mathbf{x}, \mathbf{b} \in \mathbb{R}^n.$$

- Write the system as an augmented matrix: $[A \mid \mathbf{b}]$.

Gaussian Elimination

- We want to solve

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

where

$$A \in \mathbb{R}^{n\times n}, \qquad \mathbf{x}, \mathbf{b} \in \mathbb{R}^n.$$

- Write the system as an augmented matrix: $[A \mid \mathbf{b}]$.

Goal: Transform $[A \mid \mathbf{b}] \longrightarrow [U \mid \mathbf{c}]$, where U is upper triangular:

$$U = \begin{pmatrix} u_{11} & u_{12} & \cdots & u_{1n} \\ 0 & u_{22} & \cdots & u_{2n} \\ \vdots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & u_{nn} \end{pmatrix}. \text{ Then solve } U\mathbf{x} = \mathbf{c}$$

using **back substitution**.

The Basic Elimination Step

- Suppose we are working with column k.
- The pivot is a_{kk}.

The Basic Elimination Step

- Suppose we are working with column k.
- The pivot is a_{kk}.
- For every row below the pivot i.e. $i = k+1, \ldots, n-1$

The Basic Elimination Step

- Suppose we are working with column k.
- The pivot is a_{kk}.
- For every row below the pivot i.e. $i = k+1, \ldots, n-1$ calculate the multiplier: $\boxed{m_{ik} = \dfrac{a_{ik}}{a_{kk}}}$

The Basic Elimination Step

- Suppose we are working with column k.
- The pivot is a_{kk}.
- For every row below the pivot i.e. $i = k+1, \ldots, n-1$ calculate the multiplier: $\boxed{m_{ik} = \dfrac{a_{ik}}{a_{kk}}}$
- and perform the row operation: $\boxed{R_i \leftarrow R_i - m_{ik} R_k.}$
- This makes $a_{ik} = 0$.

The Basic Elimination Step

- Suppose we are working with column k.
- The pivot is a_{kk}.
- For every row below the pivot i.e. $i = k+1, \ldots, n-1$ calculate the multiplier: $\boxed{m_{ik} = \dfrac{a_{ik}}{a_{kk}}}$
- and perform the row operation: $\boxed{R_i \leftarrow R_i - m_{ik} R_k.}$
- This makes $a_{ik} = 0$.
- Entries below the pivot are eliminated.

Gaussian Elimination: Pseudocode

```
GaussianElimination(A, b)
    n = number of rows in A
    for k = 0 to n-2
        for i = k+1 to n-1
            m = A[i,k] / A[k,k]
            for j = k to n-1
                A[i,j] = A[i,j] - m*A[k,j]
            b[i] = b[i] - m*b[k]
    return A, b
```

The algorithm transforms

$$[A \mid \mathbf{b}] \quad \longrightarrow \quad [U \mid \mathbf{c}].$$

Back Substitution

After forward elimination we have

$$U\mathbf{x} = \mathbf{c}.$$

For example,

$$\begin{aligned} u_{11}x_1 + u_{12}x_2 + u_{13}x_3 &= c_1, \\ u_{22}x_2 + u_{23}x_3 &= c_2, \\ u_{33}x_3 &= c_3. \end{aligned}$$

Start with the last equation and work upwards.

Back Substitution

After forward elimination we have

$$U\mathbf{x} = \mathbf{c}.$$

For example,

$$\begin{aligned} u_{11}x_1 + u_{12}x_2 + u_{13}x_3 &= c_1, \\ u_{22}x_2 + u_{23}x_3 &= c_2, \\ u_{33}x_3 &= c_3. \end{aligned}$$

Start with the last equation and work upwards.

For $i = n-1, n-2, \ldots, 0,$

$$\boxed{x_i = \frac{c_i - \sum_{j=i+1}^{n-1} u_{ij}x_j}{u_{ii}}}$$

$$\boxed{x_{n-1} \to x_{n-2} \to \cdots \to x_1 \to x_0}$$

Back Substitution: Pseudocode

```
BackSubstitution(U, c)
    n = number of rows
    create x = zeros(n)
    for i = n-1 down to 0
        s = 0
        for j = i+1 to n-1
            s = s + U[i,j] * x[j]
        x[i] = (c[i] - s) / U[i,i]
    return x
```

Why Pivoting?

- In Gaussian elimination, we divide by the pivot: $m = \frac{A_{ik}}{A_{kk}}$.

Why Pivoting?

- In Gaussian elimination, we divide by the pivot: $m = \frac{A_{ik}}{A_{kk}}$.
- If $A_{kk} = 0$, the algorithm fails.

Why Pivoting?

- In Gaussian elimination, we divide by the pivot: $m = \frac{A_{ik}}{A_{kk}}$.
- If $A_{kk} = 0$, the algorithm fails.
- try: Example:

$$\left[\begin{array}{cc|c} 0 & 1 & 1 \\ 1 & 1 & 2 \end{array}\right]$$

Why Pivoting?

- In Gaussian elimination, we divide by the pivot: $m = \frac{A_{ik}}{A_{kk}}$.
- If $A_{kk} = 0$, the algorithm fails.
- try: Example:

$$\left[\begin{array}{cc|c} 0 & 1 & 1 \\ 1 & 1 & 2 \end{array}\right]$$

- The first pivot is zero. Swap the rows:

$$\left[\begin{array}{cc|c} 1 & 1 & 2 \\ 0 & 1 & 1 \end{array}\right].$$

Partial Pivoting

- At elimination step k, examine the current column: $A_{kk}, A_{k+1,k}, \ldots, A_{n-1,k}$.

Partial Pivoting

- At elimination step k, examine the current column: $A_{kk}, A_{k+1,k}, \ldots, A_{n-1,k}$.
- Choose the entry with the largest absolute value:

$$\boxed{p = \underset{i=k,\ldots,n-1}{\operatorname{argmax}} |A_{ik}|}$$

Partial Pivoting

- At elimination step k, examine the current column: $A_{kk}, A_{k+1,k}, \ldots, A_{n-1,k}$.
- Choose the entry with the largest absolute value:

$$\boxed{p = \underset{i=k,\ldots,n-1}{\operatorname{argmax}} |A_{ik}|}$$

- Then swap:

$$R_k \leftrightarrow R_p.$$

Partial Pivoting

- At elimination step k, examine the current column: $A_{kk}, A_{k+1,k}, \ldots, A_{n-1,k}$.
- Choose the entry with the largest absolute value:

$$\boxed{p = \underset{i=k,\ldots,n-1}{\operatorname{argmax}} |A_{ik}|}$$

- Then swap:

$$R_k \leftrightarrow R_p.$$

- Example:

$$\begin{pmatrix} 0.001 \\ -5 \\ 2 \end{pmatrix} \quad \longrightarrow \quad \boxed{-5}$$

Partial Pivoting

- At elimination step k, examine the current column: $A_{kk}, A_{k+1,k}, \ldots, A_{n-1,k}$.
- Choose the entry with the largest absolute value:

$$\boxed{p = \underset{i=k,\ldots,n-1}{\operatorname{argmax}} |A_{ik}|}$$

- Then swap:

$$R_k \leftrightarrow R_p.$$

- Example:

$$\begin{pmatrix} 0.001 \\ -5 \\ 2 \end{pmatrix} \quad \longrightarrow \quad \boxed{-5}$$

Choose the largest available pivot in the column.

Gaussian Elimination with Partial Pivoting

```
Input: A, b
n = number of rows
FOR k = 0, ..., n-2:

    # Find pivot
    p = index of largest |A[i,k]|,
        i = k, ..., n-1

    # Swap rows
    swap rows k and p of A
    swap b[k] and b[p]

    # Eliminate below pivot
    # continue as usual
```

From Gaussian Elimination to LU

Gaussian elimination transforms

$$A\mathbf{x} = \mathbf{b}$$

into

$$U\mathbf{x} = \mathbf{c},$$

where U is upper triangular.

From Gaussian Elimination to LU

Gaussian elimination transforms

$$A\mathbf{x} = \mathbf{b}$$

into

$$U\mathbf{x} = \mathbf{c},$$

where U is upper triangular. But during elimination, we compute multipliers

$$m_{ik} = \frac{A_{ik}}{A_{kk}}.$$

From Gaussian Elimination to LU

Gaussian elimination transforms

$$A\mathbf{x} = \mathbf{b}$$

into

$$U\mathbf{x} = \mathbf{c},$$

where U is upper triangular. But during elimination, we compute multipliers

$$m_{ik} = \frac{A_{ik}}{A_{kk}}.$$

Key idea: Instead of throwing these multipliers away, store them in a lower triangular matrix L.

$$\boxed{A = LU}$$

Where Does L Come From?

Consider one elimination step:

$$R_i \leftarrow R_i - m_{ik} R_k.$$

The multiplier

$$m_{ik} = \frac{A_{ik}}{A_{kk}}$$

is exactly what is needed to construct L.

Where Does L Come From?

Consider one elimination step:

$$R_i \leftarrow R_i - m_{ik} R_k.$$

The multiplier

$$m_{ik} = \frac{A_{ik}}{A_{kk}}$$

is exactly what is needed to construct L. For example,

$$L = \begin{pmatrix} 1 & 0 & 0 \\ m_{21} & 1 & 0 \\ m_{31} & m_{32} & 1 \end{pmatrix}, \qquad U = \begin{pmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{pmatrix}.$$

Where Does L Come From?

Consider one elimination step:

$$R_i \leftarrow R_i - m_{ik} R_k.$$

The multiplier

$$m_{ik} = \frac{A_{ik}}{A_{kk}}$$

is exactly what is needed to construct L. For example,

$$L = \begin{pmatrix} 1 & 0 & 0 \\ m_{21} & 1 & 0 \\ m_{31} & m_{32} & 1 \end{pmatrix}, \qquad U = \begin{pmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{pmatrix}.$$

Thus, Gaussian elimination gives us both:

U from the row operations

L from the multipliers

Why Factorize A?

Once $A = LU$, the system $A\mathbf{x} = \mathbf{b}$ becomes

$$LU\mathbf{x} = \mathbf{b}.$$

Introduce

$$L\mathbf{y} = \mathbf{b}.$$

Then solve

$$U\mathbf{x} = \mathbf{y}.$$

Why Factorize A?

Once $A = LU$, the system $A\mathbf{x} = \mathbf{b}$ becomes

$$LU\mathbf{x} = \mathbf{b}.$$

Introduce

$$L\mathbf{y} = \mathbf{b}.$$

Then solve

$$U\mathbf{x} = \mathbf{y}.$$

So one system becomes two triangular systems:

$$A\mathbf{x} = \mathbf{b} \quad \Longrightarrow \quad \begin{cases} L\mathbf{y} = \mathbf{b}, \\ U\mathbf{x} = \mathbf{y}. \end{cases}$$

Why Factorize A?

Once $A = LU$, the system $A\mathbf{x} = \mathbf{b}$ becomes

$$LU\mathbf{x} = \mathbf{b}.$$

Introduce

$$L\mathbf{y} = \mathbf{b}.$$

Then solve

$$U\mathbf{x} = \mathbf{y}.$$

So one system becomes two triangular systems:

$$A\mathbf{x} = \mathbf{b} \quad \Longrightarrow \quad \begin{cases} L\mathbf{y} = \mathbf{b}, \\ U\mathbf{x} = \mathbf{y}. \end{cases}$$

Advantage: If A is fixed and many different $\mathbf{b}$'s are given, we factorize A only once.

LU Decomposition: Pseudocode

Input: Matrix $AU \leftarrow A, \qquad L \leftarrow I$

```
FOR k = 0,...,n-2:

    FOR i = k+1,...,n-1:

        m = U[i,k] / U[k,k]

        L[i,k] = m

        U[i,k:] = U[i,k:] - m U[k,k:]

RETURN L, U
```

LU Decomposition: Pseudocode

Input: Matrix $AU \leftarrow A, \qquad L \leftarrow I$

```
FOR k = 0,...,n-2:

    FOR i = k+1,...,n-1:

        m = U[i,k] / U[k,k]

        L[i,k] = m

        U[i,k:] = U[i,k:] - m U[k,k:]

RETURN L, U
```

$$\boxed{A = LU}$$

Using LU to Solve $A\mathbf{x} = \mathbf{b}$

Suppose

$$A = LU.$$

Then

$$A\mathbf{x} = \mathbf{b} \quad \Longrightarrow \quad LU\mathbf{x} = \mathbf{b}.$$

Introduce an intermediate vector $\mathbf{y}$:

$$L\mathbf{y} = \mathbf{b}.$$

Then solve

$$U\mathbf{x} = \mathbf{y}.$$

Using LU to Solve $A\mathbf{x} = \mathbf{b}$

Suppose

$$A = LU.$$

Then

$$A\mathbf{x} = \mathbf{b} \quad \Longrightarrow \quad LU\mathbf{x} = \mathbf{b}.$$

Introduce an intermediate vector $\mathbf{y}$:

$$L\mathbf{y} = \mathbf{b}.$$

Then solve

$$U\mathbf{x} = \mathbf{y}.$$

$L\mathbf{y} = \mathbf{b}$	(forward substitution)
$U\mathbf{x} = \mathbf{y}$	(back substitution)

LU: Solving a System

Consider

$$A = \begin{pmatrix} 2 & 1 \\ 4 & 3 \end{pmatrix}, \qquad \mathbf{b} = \begin{pmatrix} 5 \\ 11 \end{pmatrix}.$$

LU decomposition gives

$$L = \begin{pmatrix} 1 & 0 \\ 2 & 1 \end{pmatrix}, \qquad U = \begin{pmatrix} 2 & 1 \\ 0 & 1 \end{pmatrix}.$$

LU: Solving a System

Consider

$$A = \begin{pmatrix} 2 & 1 \\ 4 & 3 \end{pmatrix}, \qquad \mathbf{b} = \begin{pmatrix} 5 \\ 11 \end{pmatrix}.$$

LU decomposition gives

$$L = \begin{pmatrix} 1 & 0 \\ 2 & 1 \end{pmatrix}, \qquad U = \begin{pmatrix} 2 & 1 \\ 0 & 1 \end{pmatrix}.$$

Step 1: Forward substitution

$$L\mathbf{y} = \mathbf{b} \quad \Rightarrow \quad \begin{cases} y_1 = 5, \\ 2y_1 + y_2 = 11 \end{cases} \quad \Rightarrow \mathbf{y} = \begin{pmatrix} 5 \\ 1 \end{pmatrix}.$$

LU: Solving a System

Consider

$$A = \begin{pmatrix} 2 & 1 \\ 4 & 3 \end{pmatrix}, \qquad \mathbf{b} = \begin{pmatrix} 5 \\ 11 \end{pmatrix}.$$

LU decomposition gives

$$L = \begin{pmatrix} 1 & 0 \\ 2 & 1 \end{pmatrix}, \qquad U = \begin{pmatrix} 2 & 1 \\ 0 & 1 \end{pmatrix}.$$

Step 1: Forward substitution

$$L\mathbf{y} = \mathbf{b} \quad \Rightarrow \quad \begin{cases} y_1 = 5, \\ 2y_1 + y_2 = 11 \end{cases} \quad \Rightarrow \mathbf{y} = \begin{pmatrix} 5 \\ 1 \end{pmatrix}.$$

Step 2: Back substitution

$$U\mathbf{x} = \mathbf{y} \quad \Rightarrow \quad \begin{cases} 2x_1 + x_2 = 5, \\ x_2 = 1 \end{cases}$$

$$\boxed{\mathbf{x} = (2, 1)^T}$$

Computing the Inverse using LU Decomposition

- Suppose $A\mathbf{X} = I$.
- If $\mathbf{X} = A^{-1}$,then

$$A\mathbf{x}_j = \mathbf{e}_j, \qquad j = 1, \ldots, n,$$

where $\mathbf{e}_j$ is the j-th column of the identity matrix.

- Hence

$$\boxed{A^{-1} = \begin{bmatrix} \mathbf{x}_1 & \mathbf{x}_2 & \cdots & \mathbf{x}_n \end{bmatrix}}$$

Computing the Inverse using LU Decomposition

- Suppose $A\mathbf{X} = I$.
- If $\mathbf{X} = A^{-1}$,then

$$A\mathbf{x}_j = \mathbf{e}_j, \qquad j = 1, \ldots, n,$$

where $\mathbf{e}_j$ is the j-th column of the identity matrix.

- Hence

$$\boxed{A^{-1} = \begin{bmatrix} \mathbf{x}_1 & \mathbf{x}_2 & \cdots & \mathbf{x}_n \end{bmatrix}}$$

- Using the LU factorization

$$PA = LU,$$

each system becomes

$$L\mathbf{y}_j = P\mathbf{e}_j, \qquad U\mathbf{x}_j = \mathbf{y}_j.$$

- where P is the matrix which gives appropriate pivoted matrix PA.

The Algorithm

1. Compute the LU factorization
$$PA = LU.$$
2. For each column $\mathbf{e}_j$ of I:
$$L\mathbf{y}_j = P\mathbf{e}_j,$$
followed by
$$U\mathbf{x}_j = \mathbf{y}_j.$$
3. Store $\mathbf{x}_j$ as the j-th column of A^{-1}:
$$A^{-1} = [\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n].$$

Storing the Permutation Matrix P

- During Gaussian elimination with partial pivoting, rows of A are exchanged.
- Instead of physically constructing P at every step, we can store the row ordering in a **permutation vector**.

Storing the Permutation Matrix P

- During Gaussian elimination with partial pivoting, rows of A are exchanged.
- Instead of physically constructing P at every step, we can store the row ordering in a **permutation vector**.
- Start with $p = \begin{bmatrix} 0 & 1 & 2 & \cdots & n-1 \end{bmatrix}$.
- If rows k and p are exchanged, simply swap their entries:

```
p[[k, p]] = p[[p, k]]
```

Storing the Permutation Matrix P

- During Gaussian elimination with partial pivoting, rows of A are exchanged.
- Instead of physically constructing P at every step, we can store the row ordering in a **permutation vector**.
- Start with $p = \begin{bmatrix} 0 & 1 & 2 & \cdots & n-1 \end{bmatrix}$.
- If rows k and p are exchanged, simply swap their entries:

```
p[[k, p]] = p[[p, k]]
```

- For example, suppose

$$p = [0, 1, 2, 3]$$

and rows 0 and 2 are exchanged:

Storing the Permutation Matrix P

- $p = [2, 1, 0, 3]$.
- This means that the current row ordering is

$$A_{\text{current}} = \begin{bmatrix} A_2 \\ A_1 \\ A_0 \\ A_3 \end{bmatrix}.$$

- The corresponding permutation matrix is

$$P = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

Storing the Permutation Matrix P

- $p = [2, 1, 0, 3]$.
- This means that the current row ordering is

$$A_{\text{current}} = \begin{bmatrix} A_2 \\ A_1 \\ A_0 \\ A_3 \end{bmatrix}.$$

- The corresponding permutation matrix is

$$P = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

- $\boxed{PA = A_{\text{current}}}$

Saving the Permutation Matrix P

1. Initialize the permutation matrix: $P \leftarrow I$.

Saving the Permutation Matrix P

1. Initialize the permutation matrix: $P \leftarrow I$.
2. At the k-th elimination step, find the pivot row r: $r = \underset{i=k,\ldots,n-1}{\arg\max} |A_{ik}|$.

Saving the Permutation Matrix P

1. Initialize the permutation matrix: $P \leftarrow I$.
2. At the k-th elimination step, find the pivot row r: $r = \underset{i=k,\ldots,n-1}{\arg\max} |A_{ik}|$.
3. If $r \neq k$, exchange the same rows in both A and P:

```
Swap rows k and r of A
Swap rows k and r of P
```

Saving the Permutation Matrix P

1. Initialize the permutation matrix: $P \leftarrow I$.
2. At the k-th elimination step, find the pivot row r: $r = \underset{i=k,\ldots,n-1}{\arg\max} |A_{ik}|$.
3. If $r \neq k$, exchange the same rows in both A and P:

```
Swap rows k and r of A
Swap rows k and r of P
```

4. At the end of the elimination, $\boxed{PA = LU}$ where P records all row exchanges performed during partial pivoting.

Saving the Permutation Matrix P

1. Initialize the permutation matrix: $P \leftarrow I$.
2. At the k-th elimination step, find the pivot row r: $r = \underset{i=k,\ldots,n-1}{\arg\max} |A_{ik}|$.
3. If $r \neq k$, exchange the same rows in both A and P:

```
Swap rows k and r of A
Swap rows k and r of P
```

4. At the end of the elimination, $\boxed{PA = LU}$ where P records all row exchanges performed during partial pivoting.

```
P = np.eye(n)

# whenever rows k and r are exchanged:
P[[k,r], :] = P[[r,k], :]
```

Python Implementation

```
A = np.array([[4., 2., 1.],
              [2., 5., 2.],
              [1., 2., 4.]])
n = A.shape[0]
I = np.eye(n)
Ainv = np.zeros_like(A)
for j in range(n):
    b = I[:, j]

    # LU + forward/back substitution
    x = solve_LU(A, b)
    Ainv[:, j] = x
```

Condition Number

- The condition number measures how sensitive a problem is to small perturbations in the data.

Condition Number

- The condition number measures how sensitive a problem is to small perturbations in the data.
- For a nonsingular matrix, $\boxed{\kappa(A) = \|A\| \, \|A^{-1}\|}$

Condition Number

- The condition number measures how sensitive a problem is to small perturbations in the data.
- For a nonsingular matrix, $\boxed{\kappa(A) = \|A\| \, \|A^{-1}\|}$
- Interpretation:
  - $\kappa(A) \approx 1$: well-conditioned
  - $\kappa(A) \gg 1$: ill-conditioned

Condition Number

- The condition number measures how sensitive a problem is to small perturbations in the data.
- For a nonsingular matrix, $\boxed{\kappa(A) = \|A\| \, \|A^{-1}\|}$
- Interpretation:
  - $\kappa(A) \approx 1$: well-conditioned
  - $\kappa(A) \gg 1$: ill-conditioned
- For the linear system $Ax = b$, a small relative perturbation in b can produce an error in x amplified by approximately $\kappa(A)$:

$$\frac{\|\delta x\|}{\|x\|} \lesssim \kappa(A) \frac{\|\delta b\|}{\|b\|}.$$

Condition Number

- The condition number measures how sensitive a problem is to small perturbations in the data.
- For a nonsingular matrix, $\boxed{\kappa(A) = \|A\| \, \|A^{-1}\|}$
- Interpretation:
  - $\kappa(A) \approx 1$: well-conditioned
  - $\kappa(A) \gg 1$: ill-conditioned
- For the linear system $Ax = b$, a small relative perturbation in b can produce an error in x amplified by approximately $\kappa(A)$:

$$\frac{\|\delta x\|}{\|x\|} \lesssim \kappa(A) \frac{\|\delta b\|}{\|b\|}.$$

- **Important:** conditioning is a property of the *problem*, not of the numerical algorithm.

Approximate Inverse

- In exact arithmetic, $AA^{-1} = I$.

Approximate Inverse

- In exact arithmetic, $AA^{-1} = I$.
- In floating-point arithmetic, the computed inverse $\widehat{A^{-1}}$ is only an **approximate inverse**: $\boxed{A\,\widehat{A^{-1}} \approx I}$.

Approximate Inverse

- In exact arithmetic, $AA^{-1} = I$.
- In floating-point arithmetic, the computed inverse $\widehat{A^{-1}}$ is only an **approximate inverse**: $\boxed{A\,\widehat{A^{-1}} \approx I}$.
- Define the inverse error by $E = A\,\widehat{A^{-1}} - I$.

Approximate Inverse

- In exact arithmetic, $AA^{-1} = I$.
- In floating-point arithmetic, the computed inverse $\widehat{A^{-1}}$ is only an **approximate inverse**: $\boxed{A\,\widehat{A^{-1}} \approx I}$.
- Define the inverse error by $E = A\,\widehat{A^{-1}} - I$.
- For a well-conditioned matrix, typically $\|E\| = O(\epsilon_{\text{mach}})$, where for double precision

$$\epsilon_{\text{mach}} \approx 2.22 \times 10^{-16}.$$

Approximate Inverse

- In exact arithmetic, $AA^{-1} = I$.
- In floating-point arithmetic, the computed inverse $\widehat{A^{-1}}$ is only an **approximate inverse**: $\boxed{A\,\widehat{A^{-1}} \approx I}$.
- Define the inverse error by $E = A\,\widehat{A^{-1}} - I$.
- For a well-conditioned matrix, typically $\|E\| = O(\epsilon_{\text{mach}})$, where for double precision

$$\epsilon_{\text{mach}} \approx 2.22 \times 10^{-16}.$$

- For an ill-conditioned matrix, small round-off errors can be amplified, leading to a much larger inverse error.

Approximate Inverse

- In exact arithmetic, $AA^{-1} = I$.
- In floating-point arithmetic, the computed inverse $\widehat{A^{-1}}$ is only an **approximate inverse**: $\boxed{A\,\widehat{A^{-1}} \approx I}$.
- Define the inverse error by $E = A\,\widehat{A^{-1}} - I$.
- For a well-conditioned matrix, typically $\|E\| = O(\epsilon_{\text{mach}})$, where for double precision

$$\epsilon_{\text{mach}} \approx 2.22 \times 10^{-16}.$$

- For an ill-conditioned matrix, small round-off errors can be amplified, leading to a much larger inverse error.
- Thus,

$$\boxed{\text{small num. error} \quad + \quad \text{large } \kappa(A) \quad \Longrightarrow \quad \text{pot. large sol. err..}}$$

Computing the Condition Number

- The condition number depends on the choice of matrix norm:

$$\boxed{\kappa_p(A) = \|A\|_p \, \|A^{-1}\|_p}$$

Computing the Condition Number

- The condition number depends on the choice of matrix norm:

$$\boxed{\kappa_p(A) = \|A\|_p \, \|A^{-1}\|_p}$$

- Common choices include:
  - 1-norm:
$$\kappa_1(A) = \|A\|_1 \|A^{-1}\|_1$$
  - 2-norm:
$$\kappa_2(A) = \|A\|_2 \|A^{-1}\|_2$$
  - ∞-norm:
$$\kappa_\infty(A) = \|A\|_\infty \|A^{-1}\|_\infty$$

Computing the Condition Number

In NumPy:

```
np.linalg.cond(A)
```

Computing the Condition Number

In NumPy:

```
np.linalg.cond(A)
```

- By default, NumPy computes the 2-norm condition number.
- A large value of $\kappa(A)$ indicates an ill-conditioned problem.

Eigenvalue Decomposition

- For a square matrix $A \in \mathbb{R}^{n \times n}$, an eigenvector $\mathbf{v} \neq \mathbf{0}$ satisfies

$$\boxed{A\mathbf{v} = \lambda \mathbf{v}}$$

where λ is the corresponding eigenvalue.

Eigenvalue Decomposition

- For a square matrix $A \in \mathbb{R}^{n \times n}$, an eigenvector $\mathbf{v} \neq \mathbf{0}$ satisfies

$$\boxed{A\mathbf{v} = \lambda \mathbf{v}}$$

  where λ is the corresponding eigenvalue.

- If A has n linearly independent eigenvectors, we can write $\boxed{A = V \Lambda V^{-1}}$ where

$$V = \begin{bmatrix} | & | & & | \\ \mathbf{v}_1 & \mathbf{v}_2 & \cdots & \mathbf{v}_n \\ | & | & & | \end{bmatrix}, \qquad \Lambda = \begin{bmatrix} \lambda_1 & & 0 \\ & \ddots & \\ 0 & & \lambda_n \end{bmatrix}.$$

Interpretation

The eigenvectors provide special directions that are only **scaled** by the action of A.

Why Eigen decomposition?

- If $A = V\Lambda V^{-1}$, then many matrix operations become simpler.

Why Eigen decomposition?

- If $A = V\Lambda V^{-1}$, then many matrix operations become simpler.
- For example, $A^k = V\Lambda^k V^{-1}$, since Λ is diagonal,

$$\Lambda^k = \begin{bmatrix} \lambda_1^k & & 0 \\ & \ddots & \\ 0 & & \lambda_n^k \end{bmatrix}.$$

- This is useful for studying: powers of matrices, dynamical systems, stability, differential equations, iterative numerical methods, etc. .

Why Eigen decomposition?

- If $A = V\Lambda V^{-1}$, then many matrix operations become simpler.
- For example, $A^k = V\Lambda^k V^{-1}$, since Λ is diagonal,

$$\Lambda^k = \begin{bmatrix} \lambda_1^k & & 0 \\ & \ddots & \\ 0 & & \lambda_n^k \end{bmatrix}.$$

- This is useful for studying: powers of matrices, dynamical systems, stability, differential equations, iterative numerical methods, etc. .
- If A is symmetric i.e., $A = A^T$, the decomposition has the particularly nice form

$$\boxed{A = Q\Lambda Q^T}$$

where Q is orthogonal:$Q^TQ = I$.

Limitations of Eigendecomposition

Eigendecomposition is powerful, but it has important limitations.

Limitations of Eigendecomposition

Eigendecomposition is powerful, but it has important limitations.

- It applies directly only to **square matrices**.

Limitations of Eigendecomposition

Eigendecomposition is powerful, but it has important limitations.

- It applies directly only to **square matrices**.
- A matrix may not have enough linearly independent eigenvectors to form

$$A = V\Lambda V^{-1}.$$

Limitations of Eigendecomposition

Eigendecomposition is powerful, but it has important limitations.

- It applies directly only to **square matrices**.
- A matrix may not have enough linearly independent eigenvectors to form

$$A = V\Lambda V^{-1}.$$

- Eigenvalues and eigenvectors can be difficult to interpret for non-symmetric matrices.

Limitations of Eigendecomposition

Eigendecomposition is powerful, but it has important limitations.

- It applies directly only to **square matrices**.
- A matrix may not have enough linearly independent eigenvectors to form

$$A = V\Lambda V^{-1}.$$

- Eigenvalues and eigenvectors can be difficult to interpret for non-symmetric matrices.
- For numerical problems, eigenvectors can be sensitive when eigenvalues are close to one another.

Limitations of Eigendecomposition

Eigendecomposition is powerful, but it has important limitations.

- It applies directly only to **square matrices**.
- A matrix may not have enough linearly independent eigenvectors to form

$$A = V\Lambda V^{-1}.$$

- Eigenvalues and eigenvectors can be difficult to interpret for non-symmetric matrices.
- For numerical problems, eigenvectors can be sensitive when eigenvalues are close to one another.

Question

Can we find a decomposition that works for **every** $m \times n$ matrix and has similarly useful properties?

Limitations of Eigendecomposition

Eigendecomposition is powerful, but it has important limitations.

- It applies directly only to **square matrices**.
- A matrix may not have enough linearly independent eigenvectors to form

$$A = V\Lambda V^{-1}.$$

- Eigenvalues and eigenvectors can be difficult to interpret for non-symmetric matrices.
- For numerical problems, eigenvectors can be sensitive when eigenvalues are close to one another.

Question

Can we find a decomposition that works for **every** $m \times n$ matrix and has similarly useful properties?

This leads us to the Singular Value Decomposition.