Linear Systems

Lecture 6, Computational Astrophysics (ASTR660)

Hsiang-Yi Karen Yang, NTHU, 10/20/2022

Announcements

- ***HW3*** will be posted on eLearn TODAY. ***Due at 14:20 on 10/27 (Thu)***. Late submission within one week will receive ***75%*** of the credit
- Schedule for the midterm presentation has been posted on eLearn. Please let me know ASAP if you have any question. Here's the link to the spreadsheet:

https://docs.google.com/spreadsheets/d/16zh4Xw3bOGANhSBlbKo6is6DRghPC3Nul8tmeOVIl8c/edit?usp=sharing

Previous lecture...

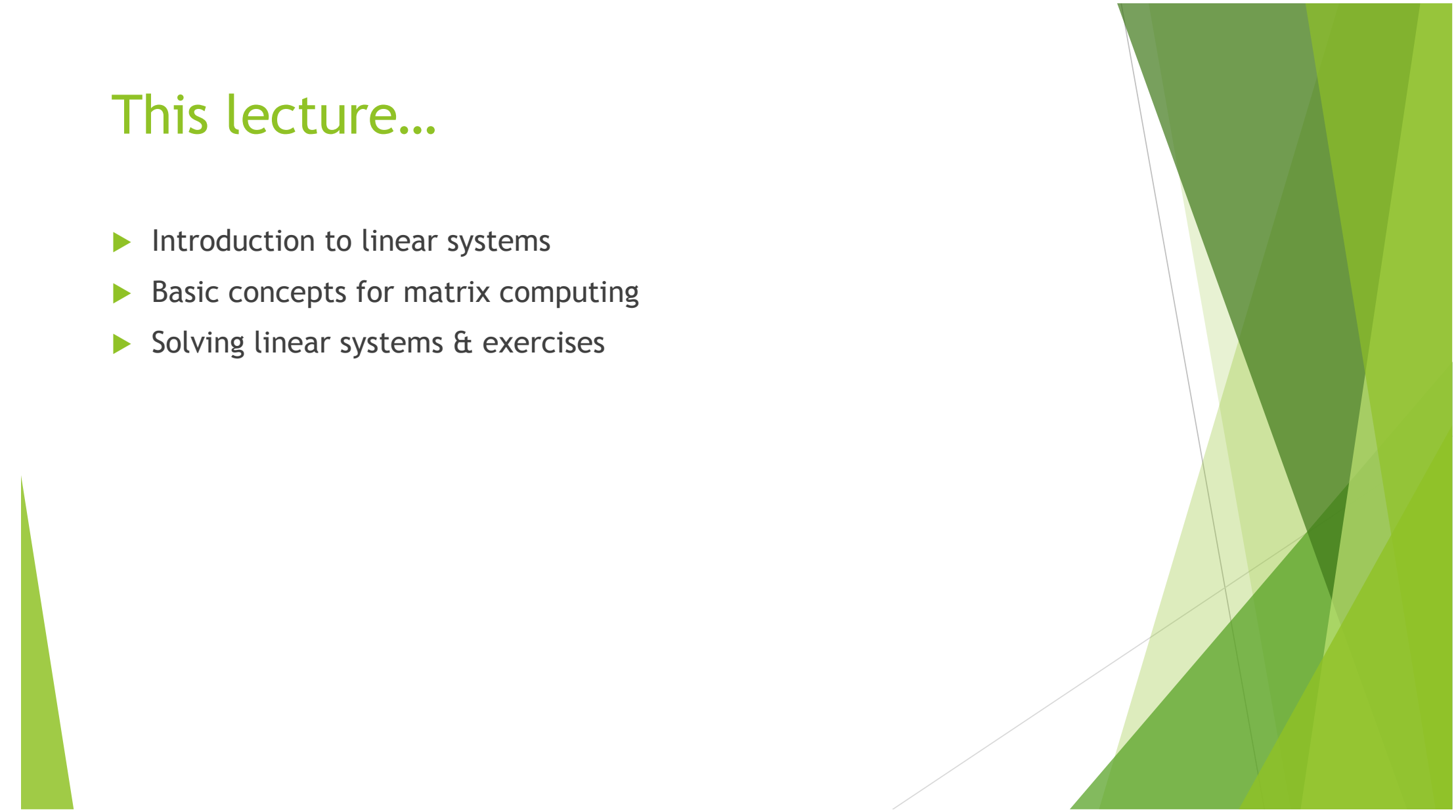
- In physics/astrophysics systems we often encounter equations that are nonlinear in the unknown variables.
- 1D nonlinear equations are easier to solve => ***root-finding*** problem
- Commonly used 1D root-finding algorithms are based on iterations:
  - ***Bisection***: bracket the solution and halve the search region
  - ***Newton-Raphson***: use intercept of tangent line as next guess
  - ***Secant***: similar to Newton-Raphson but use previous guesses to evaluate f'(x)
- Rapidly convergent methods (e.g., Newton-Raphson) only works when initial guess is close enough to solution
  - Need to combine with ***backtracking/safeguard/hybrid*** methods
- For *N*-dim nonlinear systems, Newton-Raphson & secant analogs are available

$f(x) = 0$ 1D

$\boldsymbol{f}(\boldsymbol{x}) = 0$ nD

This lecture…

- Introduction to linear systems
- Basic concepts for matrix computing
- Solving linear systems & exercises



Intro to linear systems

Linear equations

- The most basic task in linear algebra broadly used for scientific computing is to solve for unknowns in a set of *linear* equations

$$
\begin{aligned}
a_{11}x_1 + a_{12}x_2 + \ldots + a_{1N}x_N &= b_1 \\
a_{21}x_1 + a_{22}x_2 + \ldots + a_{2N}x_N &= b_2 \\
&\ldots \\
a_{M1}x_1 + a_{M2}x_2 + \ldots + a_{MN}x_N &= b_M
\end{aligned}
$$

- We could rewrite the equations in *matrix* form:

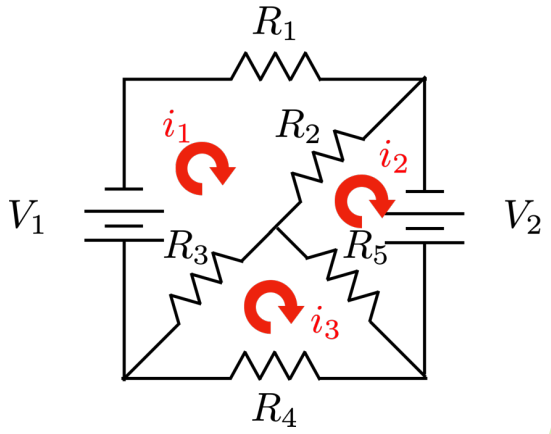
$$
\begin{bmatrix}
a_{11} & a_{12} & a_{13} & \ldots & a_{1N} \\
a_{21} & a_{22} & a_{23} & \ldots & a_{1N} \\
a_{31} & a_{32} & a_{33} & \ldots & a_{1N} \\
\ldots & & & & \\
a_{M1} & a_{M2} & a_{M3} & \ldots & a_{MN}
\end{bmatrix}
\cdot
\begin{bmatrix}
x_1 \\ x_2 \\ x_3 \\ \ldots \\ x_N
\end{bmatrix}
=
\begin{bmatrix}
b_1 \\ b_2 \\ b_3 \\ \ldots \\ b_M
\end{bmatrix}
$$

Or $\boldsymbol{A \cdot x = b}$

Example #1: electric circuit

- Given the voltages V_i and resistances R_i, solve for the electric currents i_i
- Using Ohm's law and Kirchhoff's law, one could write down a set of linear equations:

$$
\begin{aligned}
i_1 R_1 + (i_1 - i_2) R_2 + (i_1 - i_3) R_3 + V_1 &= 0 \\
(i_2 - i_1) R_2 + (i_2 - i_3) R_5 - V_2 &= 0 \\
(i_3 - i_1) R_3 + i_3 R_4 + (i_3 - i_2) R_5 &= 0
\end{aligned}
$$



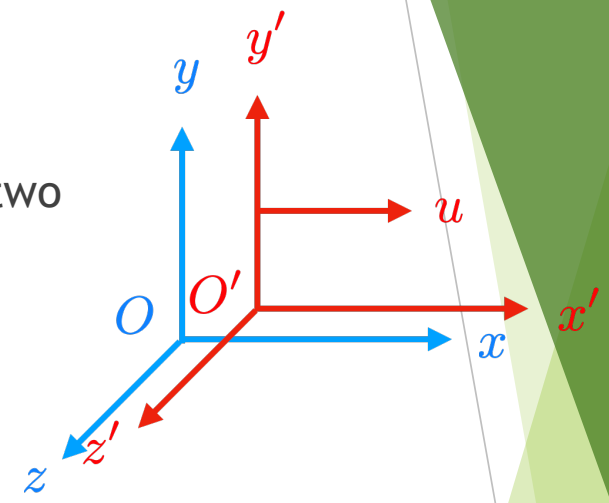
- Expressed in matrix form:

$$
\begin{bmatrix}
R_1 + R_2 + R_3 & -R_2 & -R_3 \\
-R_2 & R_2 + R_5 & -R_5 \\
-R_3 & -R_5 & R_3 + R_4 + R_5
\end{bmatrix}
\cdot
\begin{bmatrix} i_1 \\ i_2 \\ i_3 \end{bmatrix}
=
\begin{bmatrix} -V_1 \\ V_2 \\ 0 \end{bmatrix}
$$

Example #2: special relativity

- Special relativity describes the relationship between two inertial frames moving with a constant relative speed
- Their spacetime coordinates can be described by the Lorentz transformation:

$$t' = \gamma t - \gamma u/c^2 x$$
$$x' = \gamma x - \gamma u t$$
$$y' = y$$
$$z' = z$$



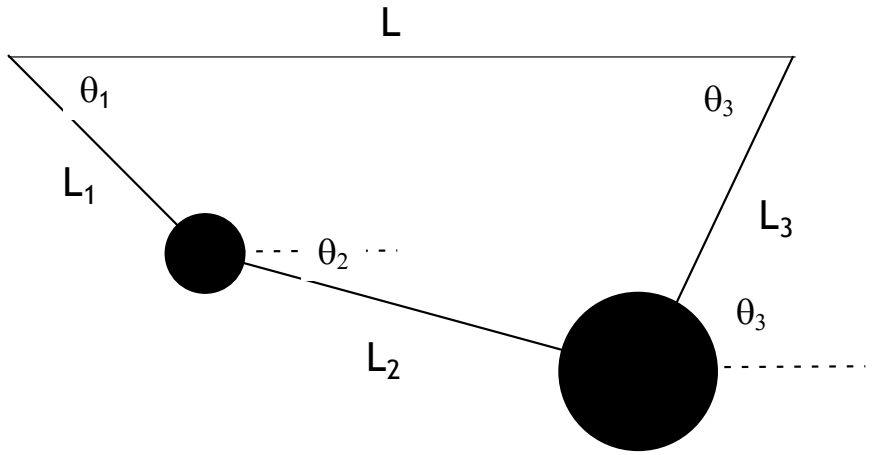
$$\gamma = \frac{1}{\sqrt{1 - u^2/c^2}}$$

- In matrix form:

$$\begin{bmatrix} \gamma & \gamma u/c^2 & 0 & 0 \\ \gamma u & \gamma & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} t' \\ x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} t \\ x \\ y \\ z \end{bmatrix}$$

Nonlinear equations can often be reduced to linear equations

- Recall the example of the multi-D nonlinear equations (last lecture)



$$
\begin{aligned}
L_1\cos\theta_1 + L_2\cos\theta_2 + L_3\cos\theta_3 &= L \\
L_1\sin\theta_1 + L_2\sin\theta_2 - L_3\sin\theta_3 &= 0 \\
\sin^2\theta_1 + \cos^2\theta_1 &= 1 \\
\sin^2\theta_2 + \cos^2\theta_2 &= 1 \\
\sin^2\theta_3 + \cos^2\theta_3 &= 1 \\
T_1\sin\theta_1 - T_2\sin\theta_2 - W_1 &= 0 \\
T_1\cos\theta_1 - T_2\cos\theta_2 &= 0 \\
T_2\sin\theta_2 + T_3\sin\theta_3 - W_2 &= 0 \\
T_2\cos\theta_2 - T_3\cos\theta_3 &= 0
\end{aligned}
$$

Nonlinear equations can often be reduced to linear equations

- One can use the Newton-Raphson method in N-dim

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k - \boldsymbol{J}(\boldsymbol{x}_k)^{-1}\boldsymbol{f}(\boldsymbol{x}_k)$$

where $\boldsymbol{J}(\boldsymbol{x})$ is *Jacobian matrix* of $\boldsymbol{f}$:

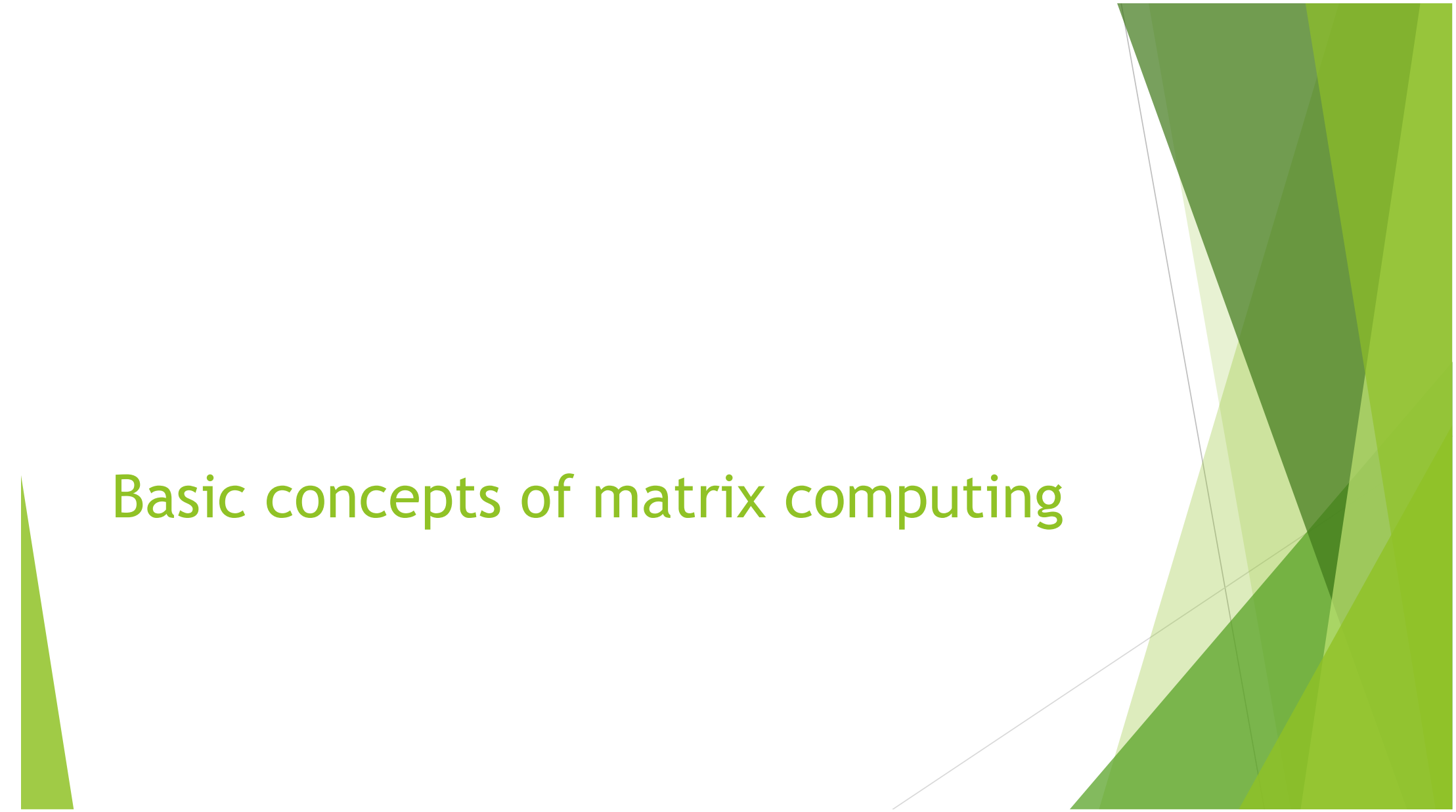
$$\{\boldsymbol{J}(\boldsymbol{x})\}_{ij} = \frac{\partial f_i(\boldsymbol{x})}{\partial x_j}$$

- In practice, the Jacobian matrix is not explicitly solved, but instead this *linear* system is solved for the step $\boldsymbol{s}_k$:

$$\boldsymbol{J}(\boldsymbol{x}_k)\boldsymbol{s}_k = -\boldsymbol{f}(\boldsymbol{x}_k)$$

Then the next guess is: $\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \boldsymbol{s}_k$

Basic concepts of matrix computing



Linear system

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

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1N} \\ a_{21} & a_{22} & a_{23} & \dots & a_{1N} \\ a_{31} & a_{32} & a_{33} & \dots & a_{1N} \\ \dots & & & & \\ a_{M1} & a_{M2} & a_{M3} & \dots & a_{MN} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \dots \\ x_N \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ \dots \\ b_M \end{bmatrix}$$

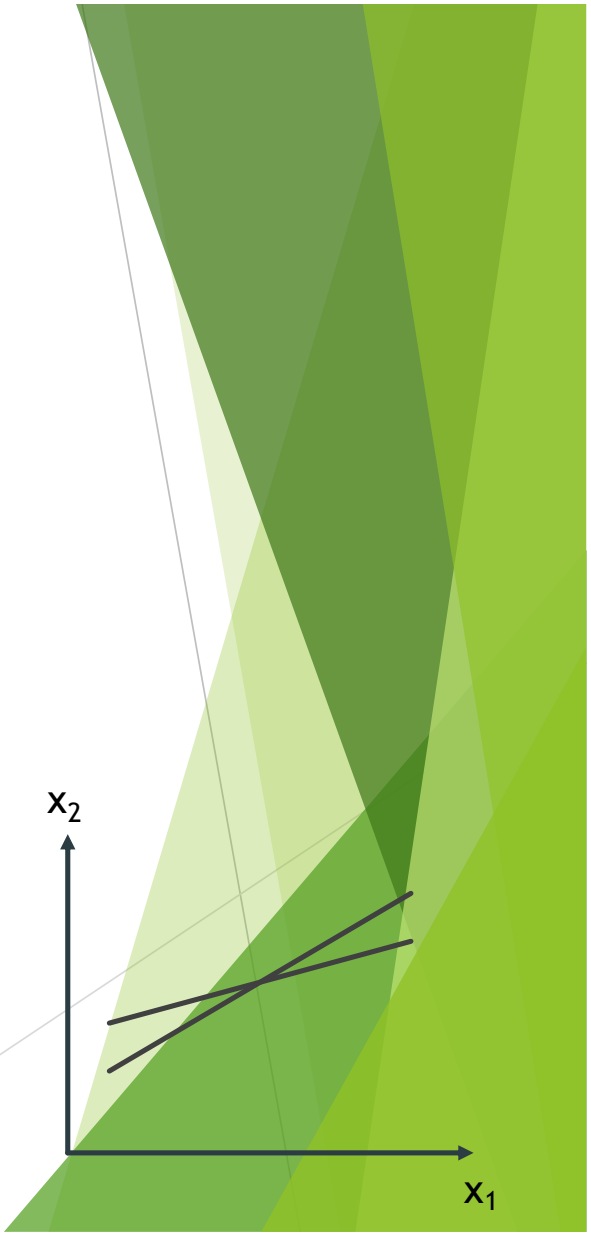
- ***A*** is a *M* x *N* matrix
  - *M* = # equations
  - *N* = # unknowns
- If *M* > *N*, the system is overdetermined => no solution in general
- If *M* < *N* => no solution or infinite solutions
- If *M* = *N* => solutions depend on whether ***A*** is ***singular*** or not. ***A*** is ***non-singular*** when it has any of the following equivalent properties:
  - ***A*** has an inverse such that $\boldsymbol{AA^{-1} = A^{-1}A = I}$
  - det(***A***) /= 0
  - rank(***A***) = *N*
  - For any vector z/=0, ***A***z /= 0

If A is nonsingular

- then a unique solution for ***x*** can be found
- **Example:**

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

- Since ***A*** is ***non-singular***, this system has a unique solution regardless of the values of *b*
- In 2D, the linear system is equivalent to solving the intersection of two straight lines

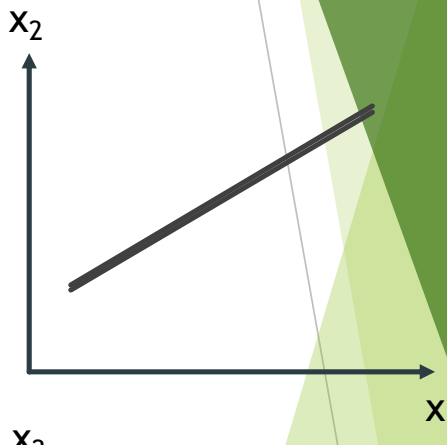


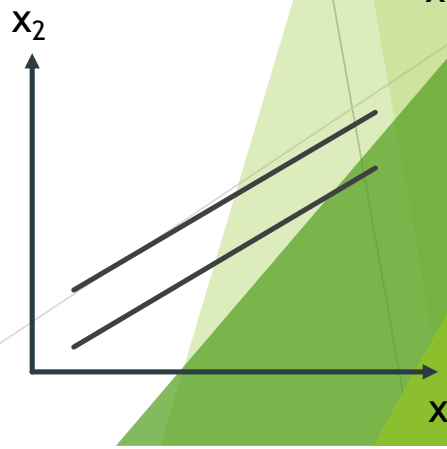
If A is singular

- then solutions depend on *b*
- **Example:**

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

- If b = [4 8]^T, then x = [c (4-2c)/3]^T => infinite solutions (two straight lines overlap)
- If b = [8 14]^T, then there is no solution => no intersection between two parallel straight lines





Error estimation using vector norms

- When solution ***x*** is a scalar, absolute error = $|x - x^*|$
- When the error is a vector, we often quantify its ***magnitude*** using ***vector norms***:

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

- Commonly used ones:

$$||x||_1 = \sum_{i=1}^{n} |x_i|,$$ (***L1-norm*** or Manhatten norm)

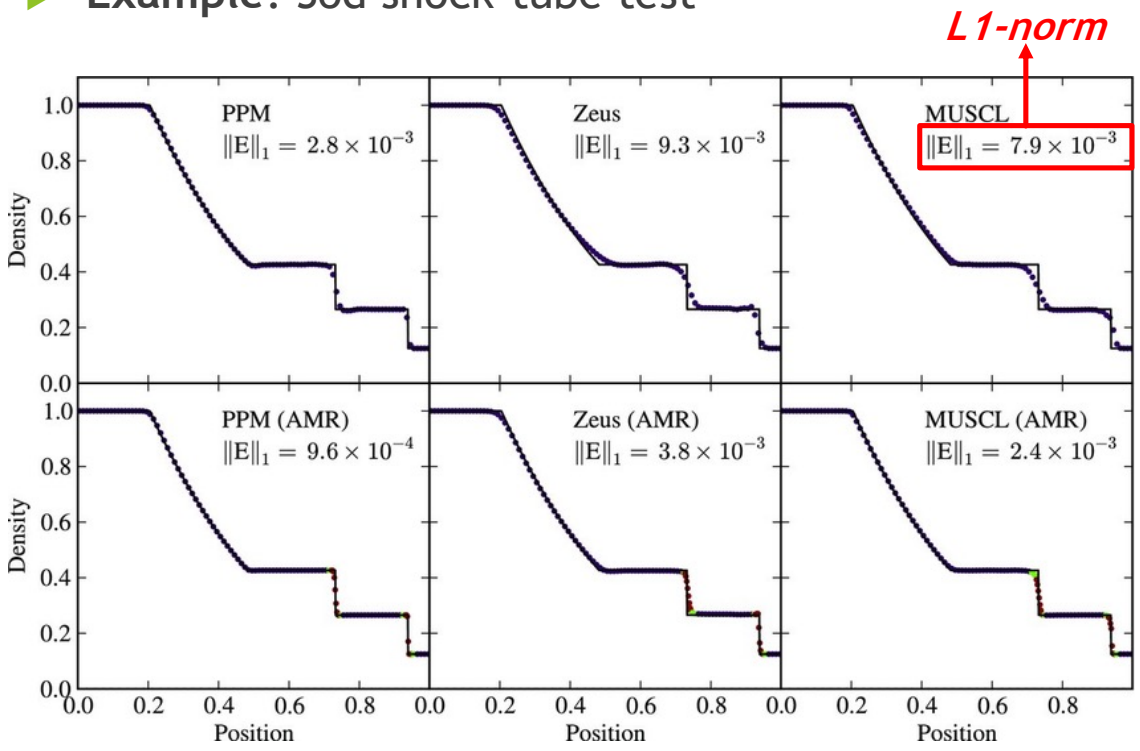
$$||x||_2 = \left(\sum_{i=1}^{n} |x_i|^2 \right)^{1/2}$$ (***L2-norm*** or Euclidean norm)

(most often used as length of a vector; also used in least-square fitting)

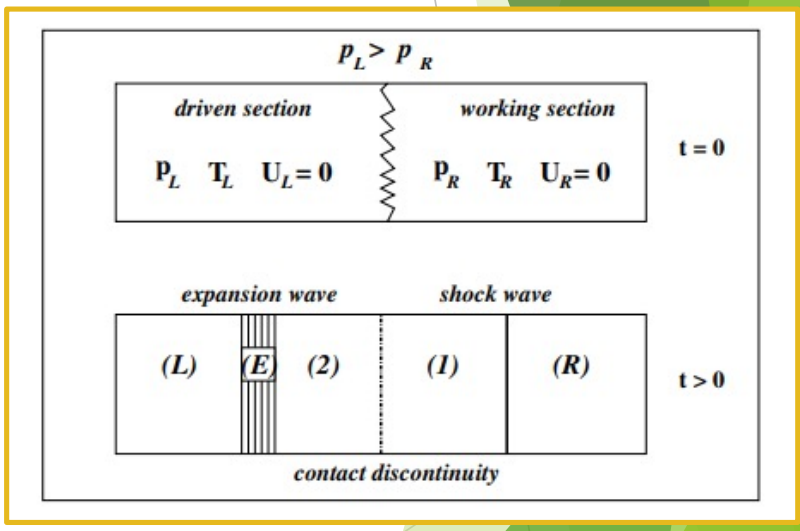
$$||x||_\infty = \max_{1 \le i \le n} |x_i|,$$ (***L∞-norm***)

Error estimation using vector norms

- **Example**: Sod shock-tube test



Schematic of the shock-tube test



Enzo collaboration (2013): comparing different hydrodynamic solvers

Properties of vector norms

L1-norm

$$||x||_1 = \sum_{i=1}^{n} |x_i|,$$

L2-norm

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

L∞-norm

$$||x||_\infty = \max_{1 \le i \le n} |x_i|,$$

- Example: ***x*** = (-1.6, 1.2) => $\|x\|_1 = 2.8, \|x\|_2 = 2.0, \|x\|_\infty = 1.6,$
- In general,

$$\|x\|_1 \geq \|x\|_2 \geq \|x\|_\infty$$

$$||x||_1 \leq \sqrt{n}||x||_2,\ ||x||_2 \leq \sqrt{n}||x||_\infty,\ \text{and}\ ||x||_1 \leq n||x||_\infty$$

- Since they differ only by a constant, they are essentially equivalent and could be chosen based on convenience

Properties of vector norms

L1-norm

$$||x||_1 = \sum_{i=1}^{n} |x_i|,$$

L2-norm

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

L∞-norm

$$||x||_\infty = \max_{1 \le i \le n} |x_i|,$$

$$||x|| > 0 \text{ if } x \neq 0$$

$$||\gamma x|| = |\gamma| \cdot ||x|| \text{ for any scalar } \gamma$$

$$||x + y|| \le ||x|| + ||y|| \text{ (triangle inequality)}$$

Matrix norms

- Similarly, one could measure the ***magnitudes*** of a matrix using ***matrix norms:***

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

- Effectively, the matrix norm measures the ***stretching*** effect of a matrix to a vector
- Analogous to the L1-norm, one could define the matrix 1-norm to be ***maximum absolute column sum***:

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

- Analogous to the L∞ -norm, one could define the matrix ∞-norm to be ***maximum absolute row sum***:

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

Properties of matrix norms

- Any matrix norm satisfies:

$$\|\boldsymbol{A}\| > 0 \text{ if } \boldsymbol{A} \neq \boldsymbol{0}$$

$$\|\gamma \boldsymbol{A}\| = |\gamma| \cdot \|\boldsymbol{A}\| \text{ for any scalar } \gamma$$

$$\|\boldsymbol{A} + \boldsymbol{B}\| \leq \|\boldsymbol{A}\| + \|\boldsymbol{B}\|$$

- Most matrix norms satisfy:

$$\|\boldsymbol{A}\boldsymbol{B}\| \leq \|\boldsymbol{A}\| \cdot \|\boldsymbol{B}\|$$

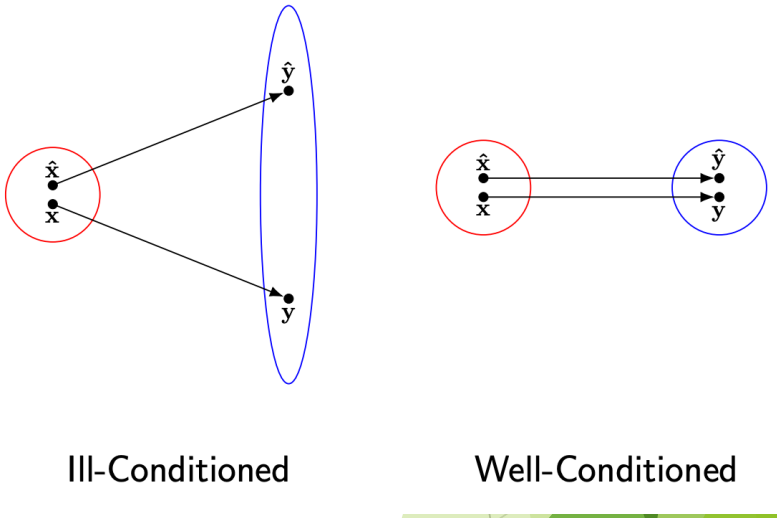
$$\|\boldsymbol{A}\boldsymbol{x}\| \leq \|\boldsymbol{A}\| \cdot \|\boldsymbol{x}\| \text{ for any vector } \boldsymbol{x}$$

Error estimation using matrix norms

- For $y = f(x)$, in order to estimate how ***sensitive*** the solution *y* depends on the input parameter *x*, one can define the ***condition number***:

$$\text{cond} = \frac{|[f(\hat{x}) - f(x)]/f(x)|}{|(\hat{x} - x)/x|} = \frac{|\Delta y/y|}{|\Delta x/x|}$$

- If cond ~ 1, problem is ***insensitive*** or ***well-conditioned***
- If cond >> 1, problem is ***sensitive*** or ***ill-conditioned***



Ill-Conditioned

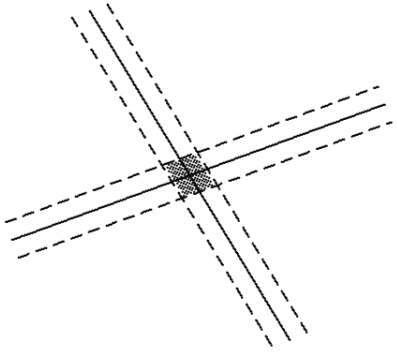
Well-Conditioned

Error estimation using matrix norms

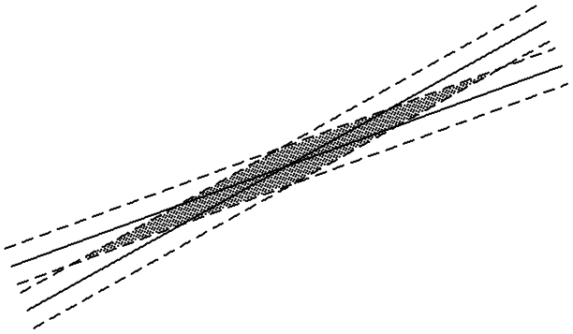
- In matrix computation, the condition number is related to the matrix norms:

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

- One can show that, for linear system $\boldsymbol{A} \cdot \boldsymbol{x} = \boldsymbol{b}$: $\dfrac{\|\Delta \boldsymbol{x}\|}{\|\boldsymbol{x}\|} \leq \mathrm{cond}(\boldsymbol{A}) \dfrac{\|\Delta \boldsymbol{b}\|}{\|\boldsymbol{b}\|}$
- If ***A*** is nearly singular, cond(***A***) would be large => solution is ***sensitive*** to perturbations in ***b***



well-conditioned



ill-conditioned

Practical aspects of matrix computing

Many programming bugs arise from improper use of arrays, so be sure to keep the following in mind!

1. ***Computers have finite memory***: A 4D array with 100 elements in one dimension would require $(100)^4$ x 64 bytes ~ 1GB memory
2. ***Processing time***: Computing inverse of a matrix of dimension *N* would take $\mathcal{O}(N^3)$ steps; doubling *N* would lead to 8x computing time
3. ***Array indices***: Python/C starts with 0 and Fortran starts with 1 by default

Practical aspects of matrix computing

4. ***Matrix storage***: the computer stores matrix elements not as multi-D blocks, but sequentially as a linear string of numbers

memory	*row-major*	*column-major*
Location	Python /C element	Fortran element
Lowest	`a[0][0]`	`a(1,1)`
	`a[0][1]`	`a(2,1)`
	`a[1][0]`	`a(3,1)`
	`a[1][1]`	`a(1,2)`
	`a[2][0]`	`a(2,2)`
Highest	`a[2][1]`	`a(3,2)`
		a(i,j)

Consequences:

- When outputing/reading arrays from files generated by another program, be careful about the row/column arrangements
- When using multi-layer for/do loops, avoid jumping across very different memory locations to access array elements
  - For Fortran programs, it's better to write:

```
do j = 1, N
  do i = 1, M
    ...
  end do
enddo
```

Solving linear systems

Solving linear systems $A \cdot x = b$

- General strategy: transform it into another linear system where the solution is the same but easier to compute
- Example: *pre-multiply* both sides by any *nonsingular* matrix ***M*** without affecting the solution, i.e., solution to ***MAx = Mb*** is same as original:

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

- What type of linear system is easy to solve? *=> triangular linear systems*

Lower triangular matrix

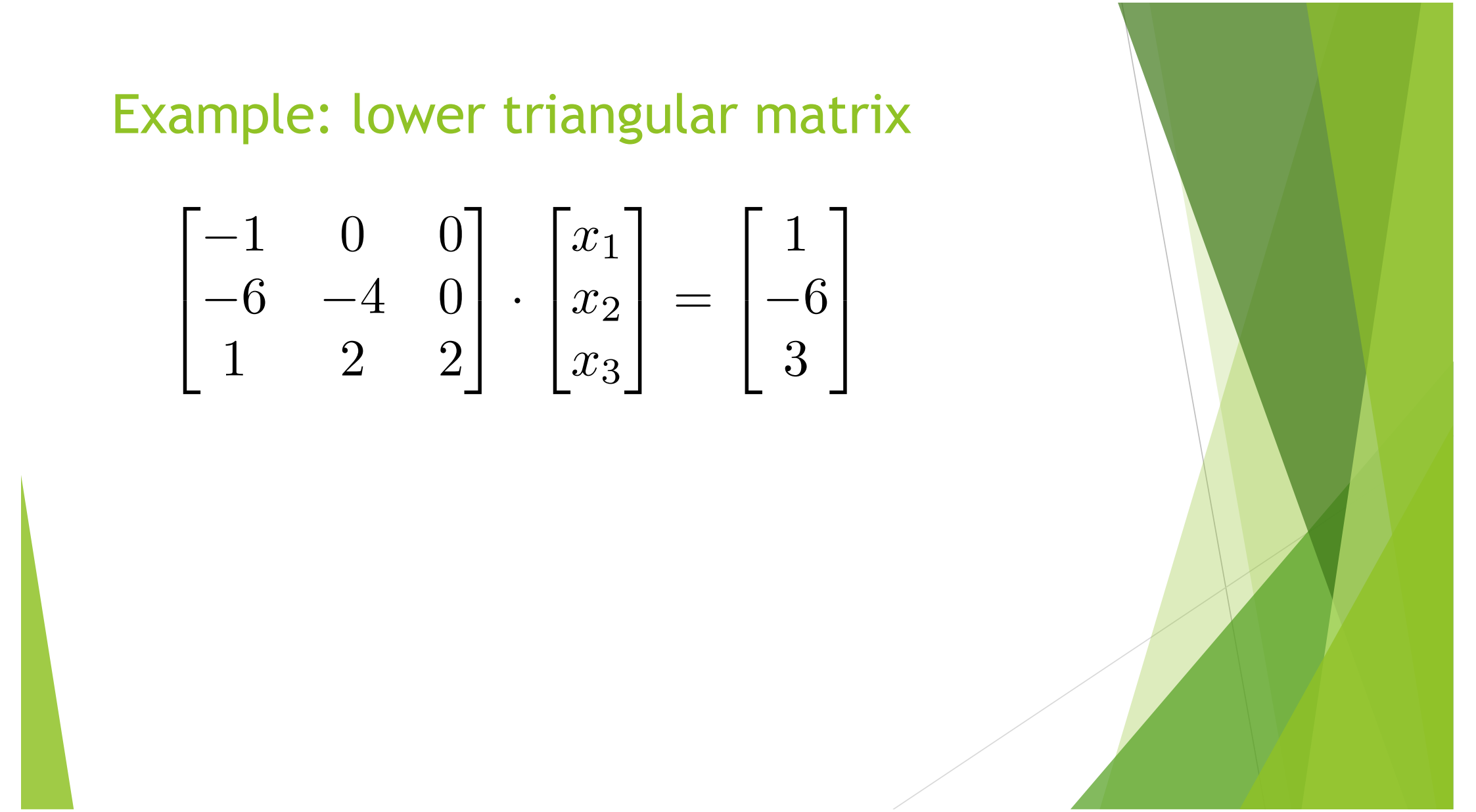
$$\begin{bmatrix} l_{11} & 0 & 0 \\ l_{21} & l_{22} & 0 \\ l_{31} & l_{32} & l_{33} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

Upper triangular matrix

$$\begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

Example: lower triangular matrix

$$\begin{bmatrix} -1 & 0 & 0 \\ -6 & -4 & 0 \\ 1 & 2 & 2 \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 1 \\ -6 \\ 3 \end{bmatrix}$$



Example: lower triangular matrix

$$\begin{bmatrix} -1 & 0 & 0 \\ -6 & -4 & 0 \\ 1 & 2 & 2 \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 1 \\ -6 \\ 3 \end{bmatrix}$$

$$x_1 = \frac{1}{-1} = -1$$

Example: lower triangular matrix

$$\begin{bmatrix} -1 & 0 & 0 \\ -6 & -4 & 0 \\ 1 & 2 & 2 \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 1 \\ -6 \\ 3 \end{bmatrix}$$

$$x_1 = \frac{1}{-1} = -1$$

$$-6x_1 - 4x_2 = -6$$

Example: lower triangular matrix

$$\begin{bmatrix} -1 & 0 & 0 \\ -6 & -4 & 0 \\ 1 & 2 & 2 \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 1 \\ -6 \\ 3 \end{bmatrix}$$

$$x_1 = \frac{1}{-1} = -1$$

$$x_2 = \left(-6 - \left(-6x_1\right)\right) / (-4)$$

Example: lower triangular matrix

$$\begin{bmatrix} -1 & 0 & 0 \\ -6 & -4 & 0 \\ 1 & 2 & 2 \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 1 \\ -6 \\ 3 \end{bmatrix}$$

$$x_1 = \frac{1}{-1} = -1$$

$$x_2 = 3$$

Example: lower triangular matrix

$$\begin{bmatrix} -1 & 0 & 0 \\ -6 & -4 & 0 \\ 1 & 2 & 2 \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 1 \\ -6 \\ 3 \end{bmatrix}$$

$$x_1 = \frac{1}{-1} = -1$$

$$x_2 = 3$$

$$x_1 + 2x_2 + 2x_3 = 3$$

Example: lower triangular matrix

$$\begin{bmatrix} -1 & 0 & 0 \\ -6 & -4 & 0 \\ 1 & 2 & 2 \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 1 \\ -6 \\ 3 \end{bmatrix}$$

$$x_1 = \frac{1}{-1} = -1$$

$$x_2 = 3$$

$$x_3 = (3 - (x_1 + 2x_2))/2$$

Example: lower triangular matrix

$$\begin{bmatrix} -1 & 0 & 0 \\ -6 & -4 & 0 \\ 1 & 2 & 2 \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 1 \\ -6 \\ 3 \end{bmatrix}$$

$$x_1 = \frac{1}{-1} = -1$$

$$x_2 = 3$$

$$x_3 = -1$$

Example: lower triangular matrix

$$\begin{bmatrix} l_{11} & 0 & 0 \\ l_{21} & l_{22} & 0 \\ l_{31} & l_{32} & l_{33} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

$$x_1 = b_1 / l_{11},$$

$$x_2 = (b_2 - (l_{21} x_1)) / l_{22}$$

$$x_3 = (b_3 - (l_{31} x_1 + l_{32} x_2)) / l_{33}$$

Forward-substitution method:

$$x_1 = b_1 / l_{11},$$

$$x_i = \left(b_i - \sum_{j=1}^{i-1} l_{ij} x_j \right) / l_{ii}, i = 2, ..., n$$

Exercise #1 – implement the forward substitution method for solving lower triangular systems

- Connect to CICA and load the Fortran compiler:

  ```
  module load pgi
  ```

- Go to your exercise directory on CICA:

  ```
  cd astr660/exercise
  ```

- Copy the template Fortran files to the current directory:

  ```
  cp -r /data/hyang/shared/astro660CompAstro/L6exercise ./
  ```

- Enter the directory and see what files are there:

  ```
  cd L6exercise
  ls
  ```

matrix.f90 – where the main program and arrays are initialized

```
program linear
   use linalg
    implicit none
    integer, parameter  :: N = 3
    real,dimension(N,N) :: lower, upper, A, P, Ainv
    real,dimension(N) :: b
    real,dimension(N) :: x
    real,dimension(4,4) :: aa,ll,uu,pp
    integer :: i,j

    ! lower triangle
    lower(1,1) = -1.0
    lower(1,2) =  0.0
    lower(1,3) =  0.0

    lower(2,1) = -6.0
    lower(2,2) = -4.0
    lower(2,3) =  0.0

    lower(3,1) =  1.0
    lower(3,2) =  2.0
    lower(3,3) =  2.0

    ! the vectore b
    b(1) =  1.0
    b(2) = -6.0
    b(3) =  3.0

    call solve_lower_triangular_matrix(N,lower,b,x)

    call mat_print("L",lower)
    print *, "vector   b = ",b
    print *, "solution x = ",x

end program linear
```

use the module defined in linalg.f90

linalg.f90 – where the *module* and subroutines are defined

For printing matrix elements to screen

For solving lower triangular systems using the forward-substitution method

```
module linalg

    ! ------------------------------------------------------------
    ! ref: https://rosettacode.org/wiki/LU_decomposition#Fortran
    ! ------------------------------------------------------------

    implicit none
    contains


        subroutine mat_print(amsg,a)
            character(*), intent(in) :: amsg
            class    (*), intent(in) :: a(:,:)
            integer                  :: i
            print*,' '
            print*,amsg
            do i=1,size(a,1)
                select type (a)
                    type is (real(8)) ; print'(100f8.3)',a(i,:)
                    type is (integer) ; print'(100i8  )',a(i,:)
                end select
            end do
            print*,' '
        end subroutine

        subroutine solve_lower_triangular_matrix(N,L,b,x)
            implicit none

            integer, intent(in)  :: N
            real, dimension(N,N), intent(in)  :: L  ! lower triangle
            real, dimension(N)  , intent(in)  :: b  ! vector
            real, dimension(N)  , intent(out) :: x  ! solution
            real, dimension(N)  :: bs

            integer :: i,j

            return
        end subroutine solve_lower_triangular_matrix
```

Exercise #1 – implement the forward substitution method for solving lower triangular systems

- Let's solve this lower triangular matrix:

$$\begin{bmatrix} -1 & 0 & 0 \\ -6 & -4 & 0 \\ 1 & 2 & 2 \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 1 \\ -6 \\ 3 \end{bmatrix}$$

- Fill in the subroutine `solve_lower_triangular_matrix` in `linalg.f90` (algorithm shown on the right)
- Compile and run the codes by

```
make
```

```
./matrix
```

- Verify that your code prints out the correct answer $\boldsymbol{x} = [-1 \;\; 3 \;\; -1]^T$

Pseudo code for forward-substitution method:

```
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
```

(please go through this code step by step and understand how it works)

Example: upper triangular matrix

$$\begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

Back-substitution method:

$$x_n = b_n / u_{nn},$$

$$x_i = \left(b_i - \sum_{j=i+1}^{n} u_{ij} x_j \right) / u_{ii}, i = 2, ..., n$$

Exercise #2 – implement the back-substitution method for solving upper triangular systems

- Let's solve this upper triangular matrix:

$$\begin{bmatrix} 1 & 2 & 2 \\ 0 & -4 & -6 \\ 0 & 0 & -1 \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 3 \\ -6 \\ 1 \end{bmatrix}$$

- Fill in the subroutine `solve_upper_triangular_matrix` in `linalg.f90` (algorithm shown on the right)
- Compile and run the codes by

  ```
  make
  ./matrix
  ```

- Verify that your code prints out the correct answer
- To get the bonus credit, submit your code and screen shot to the TAs by end of today (10/20/2022)

Pseudo code for back-substitution method:

```
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
```

How to transform a general matrix to a triangular matrix

Elimination (消去法)

- To transform a general matrix into triangular form, need to replace selected elements by zeros
- This can be done by taking linear combinations of rows
- **Example:** for $\mathbf{a} = [a_1 \;\; a_2]^T$ (if $a_1 \neq 0$),

"pivot" (樞鈕/支點; must be nonzero)

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

By choosing the right coefficients, a_2 is eliminated

"Elementary elimination matrix"

Elementary elimination matrix

- More generally, to eliminate all elements below kth position in a n-vector $\mathbf{a}$:

"pivot" (must be nonzero)

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

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

coefficients chosen to eliminate a_{k+1} to a_n

Properties of elementary elimination matrix

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

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

- M_k is unit *lower triangular matrix* and is non-singular
- $M_k = I - m_k e_k^T$, where $m_k = [0, \ldots, 0, m_{k+1}, \ldots, m_n]^T$ and e_k is kth column of identify matrix
- $M_k^{-1} = I + m_k e_k^T$, which means $M_k^{-1} = L_k$ is the same as M_k except the signs of m_k are reversed
- **Example:**

$$M_1 a = \begin{bmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} 2 \\ 4 \\ -2 \end{bmatrix} = \begin{bmatrix} 2 \\ 0 \\ 0 \end{bmatrix} \qquad M_2 a = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1/2 & 1 \end{bmatrix} \begin{bmatrix} 2 \\ 4 \\ -2 \end{bmatrix} = \begin{bmatrix} 2 \\ 4 \\ 0 \end{bmatrix}$$

$$L_1 = M_1^{-1} = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ -1 & 0 & 1 \end{bmatrix} \qquad L_2 = M_2^{-1} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -1/2 & 1 \end{bmatrix}$$

Properties of elementary elimination matrix

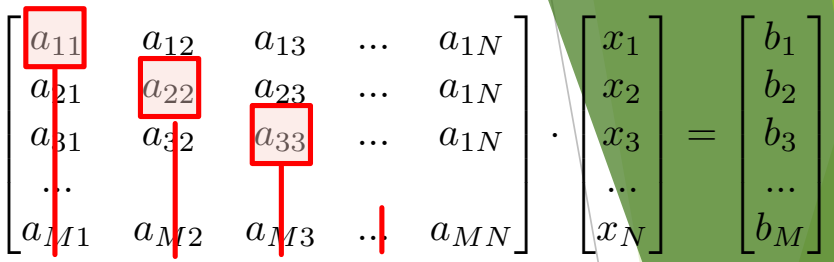
$$\begin{aligned} \boldsymbol{M}_k \boldsymbol{M}_j &= \boldsymbol{I} - \boldsymbol{m}_k \boldsymbol{e}_k^T - \boldsymbol{m}_j \boldsymbol{e}_j^T + \boldsymbol{m}_k \boldsymbol{e}_k^T \boldsymbol{m}_j \boldsymbol{e}_j^T \\ &= \boldsymbol{I} - \boldsymbol{m}_k \boldsymbol{e}_k^T - \boldsymbol{m}_j \boldsymbol{e}_j^T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

Example:

$$\boldsymbol{M}_1 \boldsymbol{a} = \begin{bmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} 2 \\ 4 \\ -2 \end{bmatrix} = \begin{bmatrix} 2 \\ 0 \\ 0 \end{bmatrix} \quad \boldsymbol{M}_2 \boldsymbol{a} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1/2 & 1 \end{bmatrix} \begin{bmatrix} 2 \\ 4 \\ -2 \end{bmatrix} = \begin{bmatrix} 2 \\ 4 \\ 0 \end{bmatrix} \quad \boldsymbol{M}_1 \boldsymbol{M}_2 = \begin{bmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ 1 & 1/2 & 1 \end{bmatrix}$$

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

Gaussian elimination



To transform the general linear system ***Ax = b*** to upper triangular form

- **Step 1** – apply M_1 (a_{11} is pivot) to both sides to eliminate 1st column of ***A*** below the first row
- **Step 2** – apply M_2 (a_{22} is pivot) to eliminate 2nd column of M_1A below 2nd row
- Continue...
- Finally getting an *upper triangular linear system*:
- This can be solved by *back-substitution* to obtain solution ***x***

$$M_1 Ax = M_1 b$$

$$M_2 M_1 Ax = M_2 M_1 b$$

$$\begin{aligned} M_{n-1} \cdots M_1 Ax &= M_{n-1} \cdots M_1 b \\ MAx &= Mb \end{aligned}$$

Gaussian elimination = LU decomposition

$$MAx = Mb$$

$$\left.\begin{aligned} MA &= U \\ M^{-1} &= L \end{aligned}\right\} \quad \boxed{\boldsymbol{A = LU}}$$

Original problem becomes: $\boldsymbol{LUx = b}$

which can be solved in two steps:

$$\boldsymbol{Ly = b} \quad \textit{(forward substitution)}$$

$$\boldsymbol{Ux = y} \quad \textit{(back substitution)}$$

Example: Gaussian elimination/LU decomposition

Solve this linear system:

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

$$\boldsymbol{M_1 A} = \begin{bmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} 2 & 4 & -2 \\ 4 & 9 & -3 \\ -2 & -3 & 7 \end{bmatrix} = \begin{bmatrix} 2 & 4 & -2 \\ 0 & 1 & 1 \\ 0 & 1 & 5 \end{bmatrix} \qquad \boldsymbol{M_1 b} = \begin{bmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} 2 \\ 8 \\ 10 \end{bmatrix} = \begin{bmatrix} 2 \\ 4 \\ 12 \end{bmatrix}$$

$$\boldsymbol{M_2 M_1 A} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -1 & 1 \end{bmatrix} \begin{bmatrix} 2 & 4 & -2 \\ 0 & 1 & 1 \\ 0 & 1 & 5 \end{bmatrix} = \begin{bmatrix} 2 & 4 & -2 \\ 0 & 1 & 1 \\ 0 & 0 & 4 \end{bmatrix} = U, \qquad \boldsymbol{M_2 M_1 b} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -1 & 1 \end{bmatrix} \begin{bmatrix} 2 \\ 4 \\ 12 \end{bmatrix} = \begin{bmatrix} 2 \\ 4 \\ 8 \end{bmatrix} = \boldsymbol{Mb}$$

Example: Gaussian elimination/LU decomposition

Problem reduced to:

$$\boldsymbol{U}\boldsymbol{x} = \begin{bmatrix} 2 & 4 & -2 \\ 0 & 1 & 1 \\ 0 & 0 & 4 \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 2 \\ 4 \\ 8 \end{bmatrix} = \boldsymbol{M}\boldsymbol{b}$$

Using back-substitution:

$$\boldsymbol{x} = \begin{bmatrix} -1 \\ 2 \\ 2 \end{bmatrix}$$

Example: Gaussian elimination/LU decomposition

Equivalently,

$$\boldsymbol{L_1 L_2} = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ -1 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ -1 & 1 & 1 \end{bmatrix} = \boldsymbol{L}$$

$(M_1^{-1} M_2^{-1})$

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

Pseudo code for LU decomposition/factorization

This algorithm can *decompose/factor (分解)* ***A*** into ***L*** and ***U***:

```
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
```

- ***L*** = resulting matrix ***M*** + identity matrix ***I***
- ***U*** = upper triangular part of resulting matrix ***A***

Special types of linear systems

Specialized algorithms are available for special matrices:

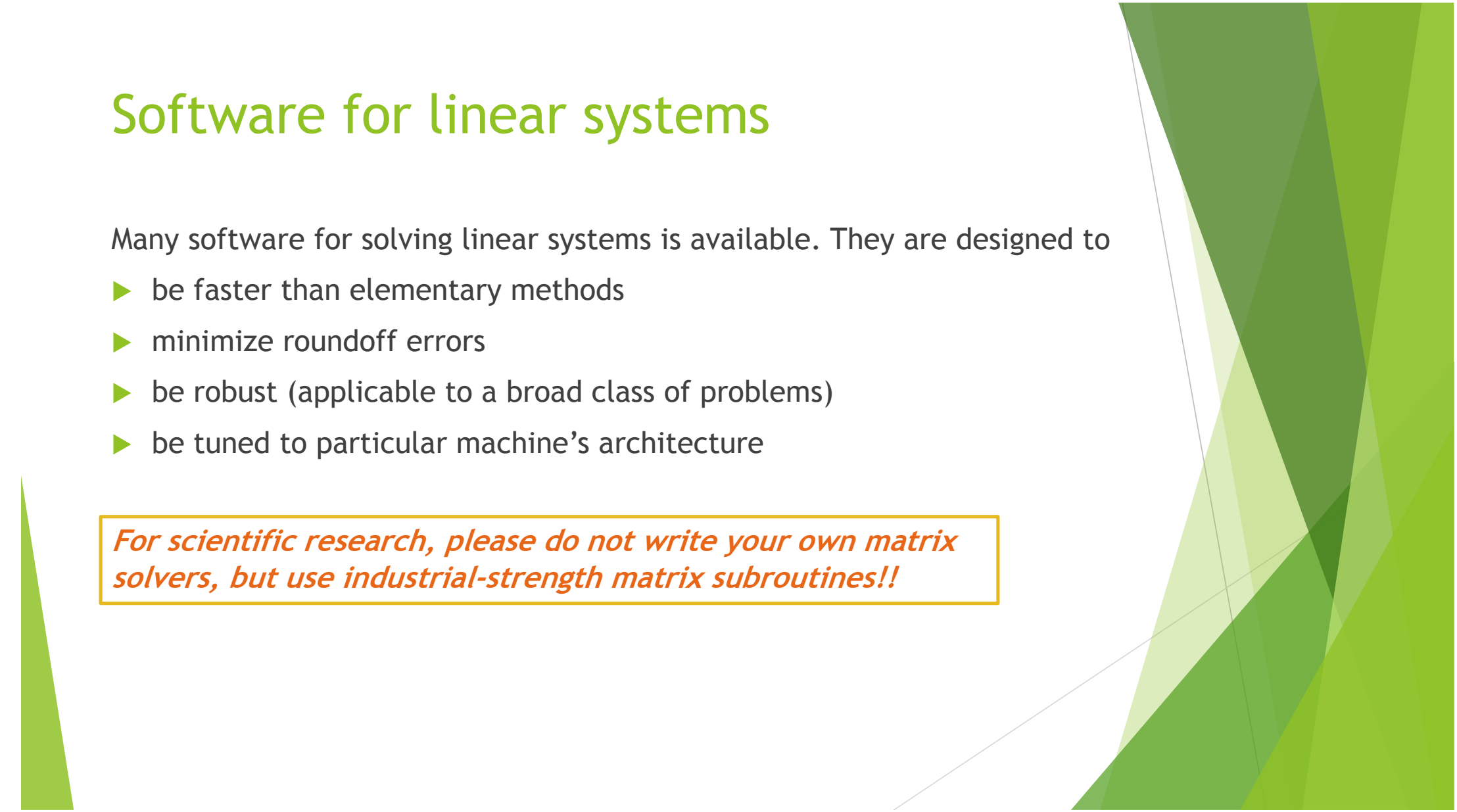
- *Symmetric*: $\boldsymbol{A} = \boldsymbol{A}^T$
- *Positive definite*: $\boldsymbol{x}^T \boldsymbol{A}\, \boldsymbol{x} > 0$ for all $\boldsymbol{x} \neq 0$
- *Banded*: $a_{ij} = 0$ for all $|i-j| > c$, where c is the bandwidth
  - If $c = 1$ => tridiagonal matrix
- *Sparse*: most entries of **A** are zero

Software for linear systems

Many software for solving linear systems is available. They are designed to

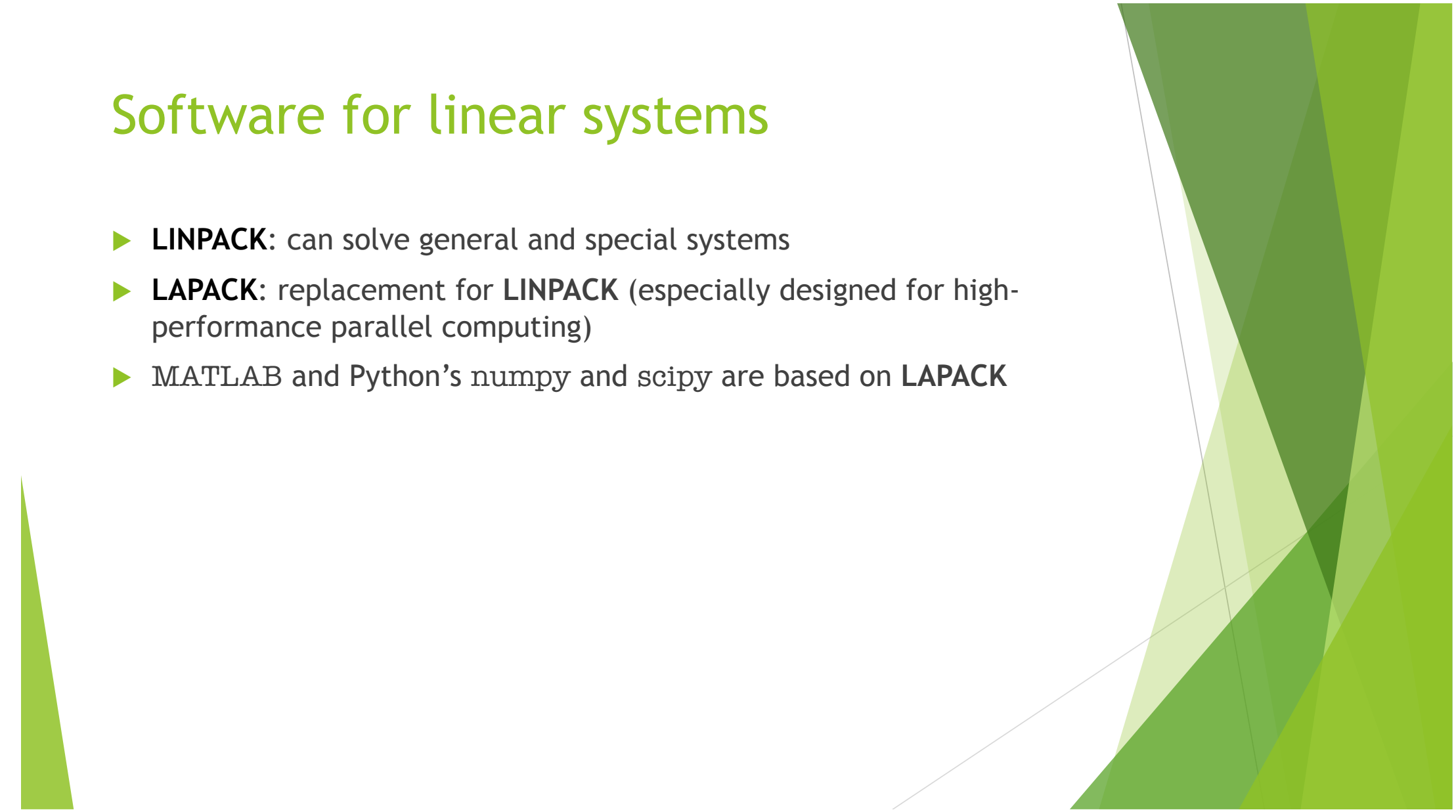
- be faster than elementary methods
- minimize roundoff errors
- be robust (applicable to a broad class of problems)
- be tuned to particular machine's architecture

For scientific research, please do not write your own matrix solvers, but use industrial-strength matrix subroutines!!



Software for linear systems

- **LINPACK**: can solve general and special systems
- **LAPACK**: replacement for **LINPACK** (especially designed for high-performance parallel computing)
- `MATLAB` and Python's `numpy` and `scipy` are based on **LAPACK**



Linear systems -- summary

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

- In physics/astrophysics, we often need to solve equations that are *linear* in the unknown variables
- Express ***A*** as a *M* x *N* matrix, # solutions depend on values of *M* and *N*, and whether ***A*** is singular or non-singular
- *Vector norms* (*L1*-norm, *L2*-norm, *L*∞-norm) can measure the magnitude of a vector and often are used for quantifying errors

  Matrix norms can measure magnitude of a matrix
- *Condition number* is used to see how *sensitive* the solutions are

$$\text{cond} = \frac{|[f(\hat{x}) - f(x)]/f(x)|}{|(\hat{x} - x)/x|} = \frac{|\Delta y/y|}{|\Delta x/x|}$$

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

Linear systems -- summary

$$A \cdot x = b$$

- Strategies for solving linear systems
  - Transform a general matrix to a triangular matrix: *Gaussian elimination* or *LU decomposition/factorization*
  - For lower triangular systems: *forward substitution*
  - For upper triangular systems: *back substitution*
- For research purposes, it is recommended to use industrial-strength matrix software

References & acknowledgements

- Course materials of Computational Astrophysics from Prof. Kuo-Chuan Pan (NTHU)
- Course materials of Computational Astrophysics from Prof. Hsi-Yu Schive (NTU)
- Course materials of Computational Astrophysics and Cosmology from Prof. Paul Ricker (UIUC)
- “Computational Physics” by Rubin H. Landau, Manuel Jose Paez and Cristian C. Bordeianu
- “Scientific Computing – An Introductory Survey” by Michael T. Heath