Lecture 2

Linear Systems I

CS328 - Numerical Methods for Visual Computing and Machine Learning

Prof. Wenzel Jakob

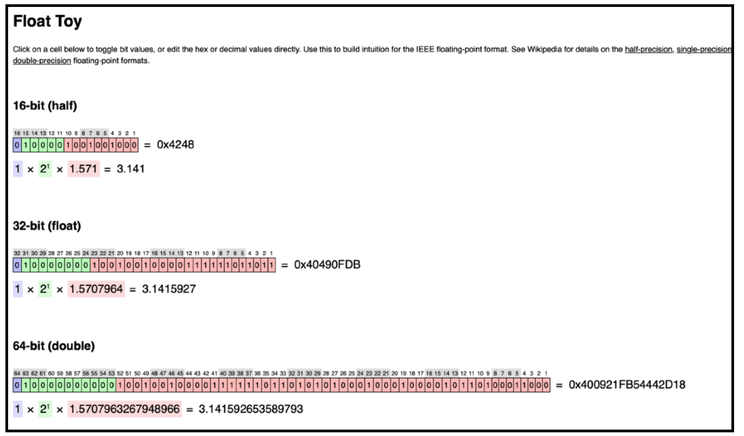
EPFL RGL Realistic Graphics Lab

Administrative bits

- Class and homework schedule posted: see https://rgl.epfl.ch/courses/NMVC25
- Homework 1 out today. **Deadline**: Oct. 2 at 9 pm.
- Will release exam-style equations roughly bi-weekly (starting next week). Ungraded.

IEEE 754 Standard— Computational foundation

Recap of lecture 1



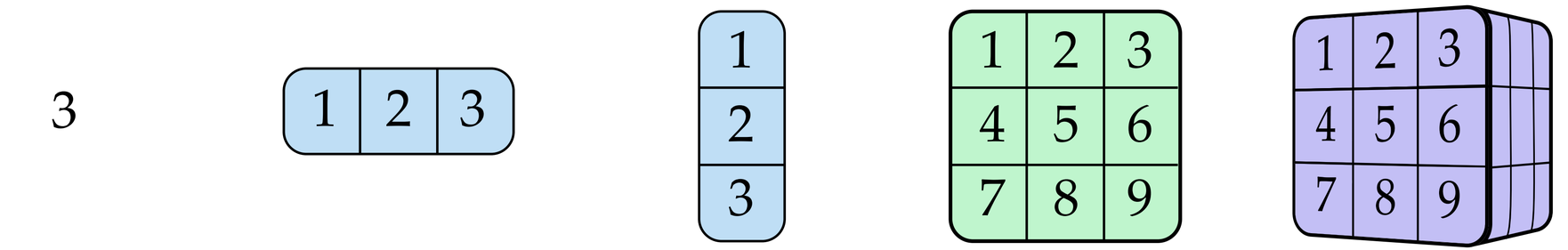
- Cool website to play with different FP formats
  - **Float Toy**: https://evanw.github.io/float-toy/

- We now understand how floating point arithmetic generally works.

- **But**: individual numbers *aren't enough* to solve interesting problems.
  - Real problems involve millions to billions of numbers arranged in specific ways.
  - Today, we will look at the most common arrangement: *vectors & matrices*.

Scalars, vectors, matrices, tensors, oh my..

Key numerical data structures organized by dimension



Scalar | Row vector | Column vector | Matrix | Tensor

Representations of *linear functions*

nd-array

General n-dimensional storage

What can a linear function do?

Turns out not very much!

f is a linear function *if and only if*

$$f(\mathbf{x}+\mathbf{y}) = f(\mathbf{x}) + f(\mathbf{y}) \qquad \textit{additivity}$$

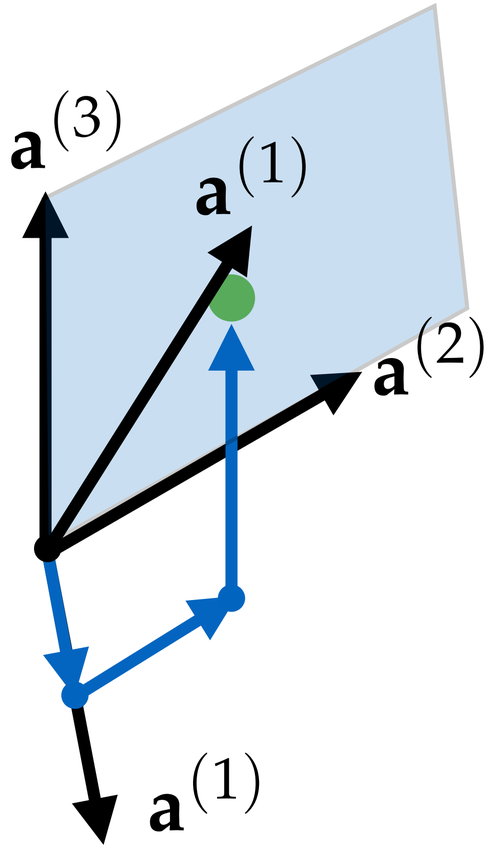
$$f(\alpha\mathbf{x}) = \alpha f(\mathbf{x}) \quad (\alpha \in \mathbb{R}) \qquad \textit{homogeneity}$$

In this case $f(\mathbf{x}) = \mathbf{A}\mathbf{x}$ fully encodes everything that f does!

Matrix-vector multiplication: $$(\mathbf{A}\mathbf{x})_i = \sum_{k=1}^{n} A_{ik} x_k$$

Geometric interpretation of matrix-vector product

$$(\mathbf{Ax})_i = \sum_{k=1}^{n} A_{ik} x_k \quad \text{where} \quad \mathbf{A} = \begin{pmatrix} | & | & | \\ \mathbf{a}^{(1)} & \mathbf{a}^{(2)} & \mathbf{a}^{(3)} \\ | & | & | \end{pmatrix}$$



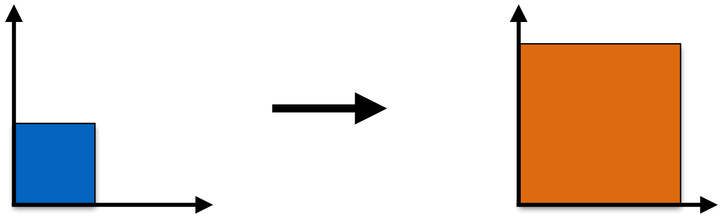
Example: linearly dependent columns span 2D subspace

$$\mathbf{Ax} = \mathbf{a}^{(1)} x_1 + \mathbf{a}^{(2)} x_2 + \mathbf{a}^{(3)} x_3$$

Basic matrix transformations in 2D

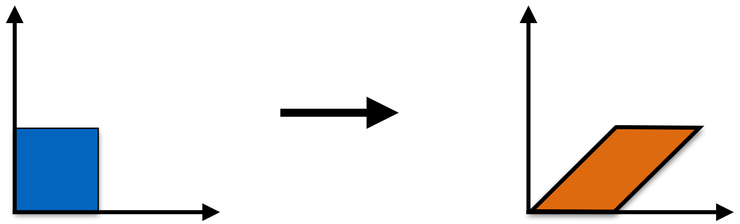
Scaling:

$$\begin{bmatrix} \mathbf{v}'_x \\ \mathbf{v}'_y \end{bmatrix} = \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix} \begin{bmatrix} \mathbf{v}_x \\ \mathbf{v}_y \end{bmatrix}$$



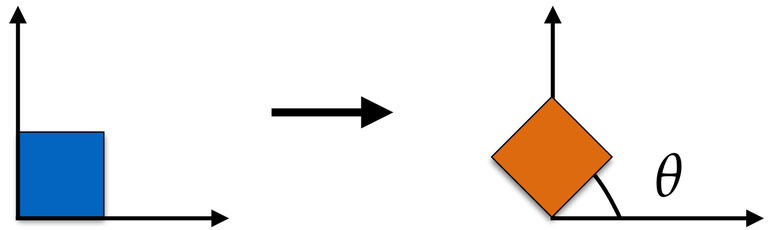
Shear:

$$\begin{bmatrix} \mathbf{v}'_x \\ \mathbf{v}'_y \end{bmatrix} = \begin{bmatrix} 1 & c \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \mathbf{v}_x \\ \mathbf{v}_y \end{bmatrix}$$



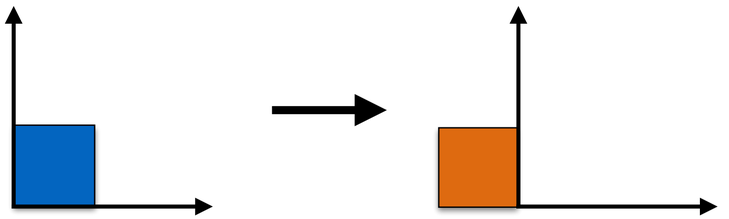
Rotation:

$$\begin{bmatrix} \mathbf{v}'_x \\ \mathbf{v}'_y \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} \mathbf{v}_x \\ \mathbf{v}_y \end{bmatrix}$$



Mirror:

$$\begin{bmatrix} \mathbf{v}'_x \\ \mathbf{v}'_y \end{bmatrix} = \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \mathbf{v}_x \\ \mathbf{v}_y \end{bmatrix}$$



Why should we care about linear functions?

(If that's really all they can do, isn't it better to focus on something more powerful?)

- Solving linear problems is one of the (few ☹) numerical problems that we know to solve **really well**.
  - When you can turn something into a linear system ↝ *good job, you are done*.
  - *Nonlinear* methods are *fragile* 😬.

- Almost everything boils down to a linear system at some point.

Matrix multiplication

Why is it defined in this particular way?

$$(\mathbf{AB})_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj}$$

Midjourney: *matrix multiplication*

$$f(\mathbf{x}) \quad g(\mathbf{x}) \quad g(f(\mathbf{x}))$$

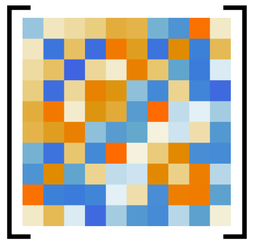
$$\updownarrow \qquad \updownarrow \qquad \updownarrow$$

$$\mathbf{Ax} \qquad \mathbf{Bx} \qquad \underbrace{\mathbf{BA}}_{\mathbf{C}}\mathbf{x}$$

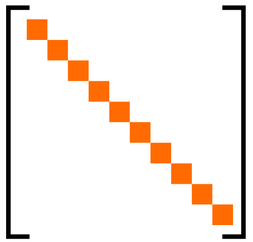
Matrix Zoo

A glimpse into the great variety of matrix flavors that arise in practice

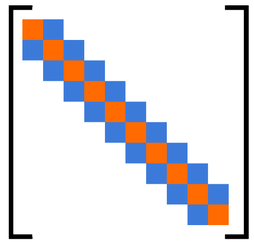
Dense



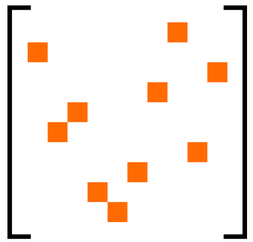
Symmetric



Diagonal

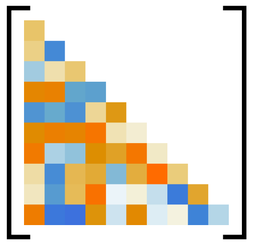


Tridiagonal

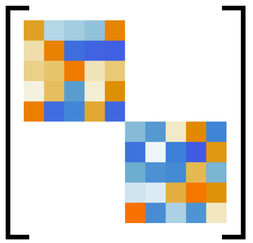


Permutation

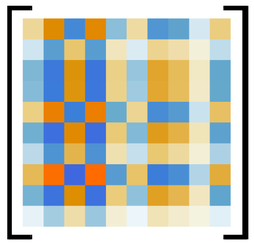
Upper triangular



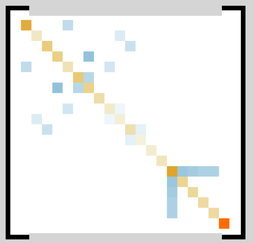
Lower triangular



Block



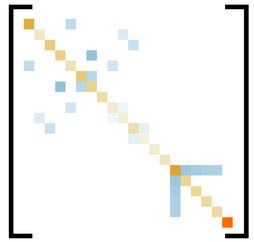
Low rank



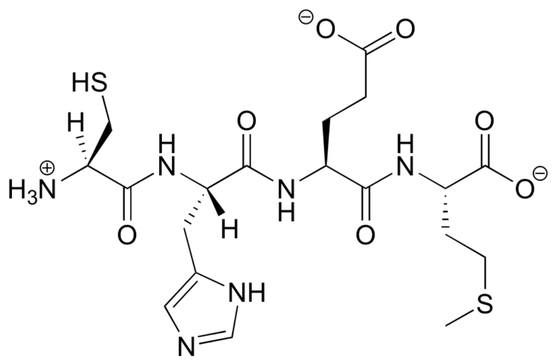
Sparse

Also important: positive definite & orthogonal matrices.

Sparse matrices

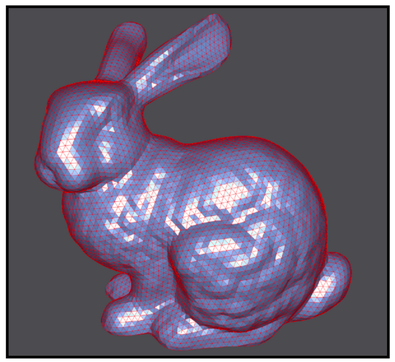


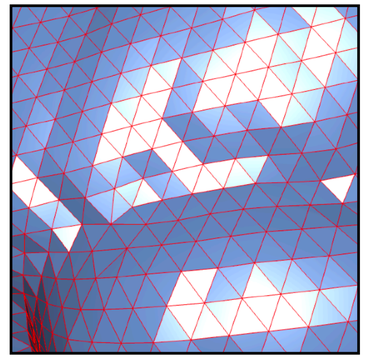
- In a **sparse matrix**, *almost all* (> 99%) entries are zero.
  - **Useful why?** Can avoid storing the zeros & skip them during computation.
  - Almost every matrix algorithm has an analogous *sparse* version (or several!)
- Encodings: *Compressed Sparse Row* (CSR) or *Compressed Sparse Column* (CSC) format
  - In NumPy/SciPy: `scipy.sparse.csr_matrix` & `scipy.sparse.csc_matrix`



Protein modeling

Social networks



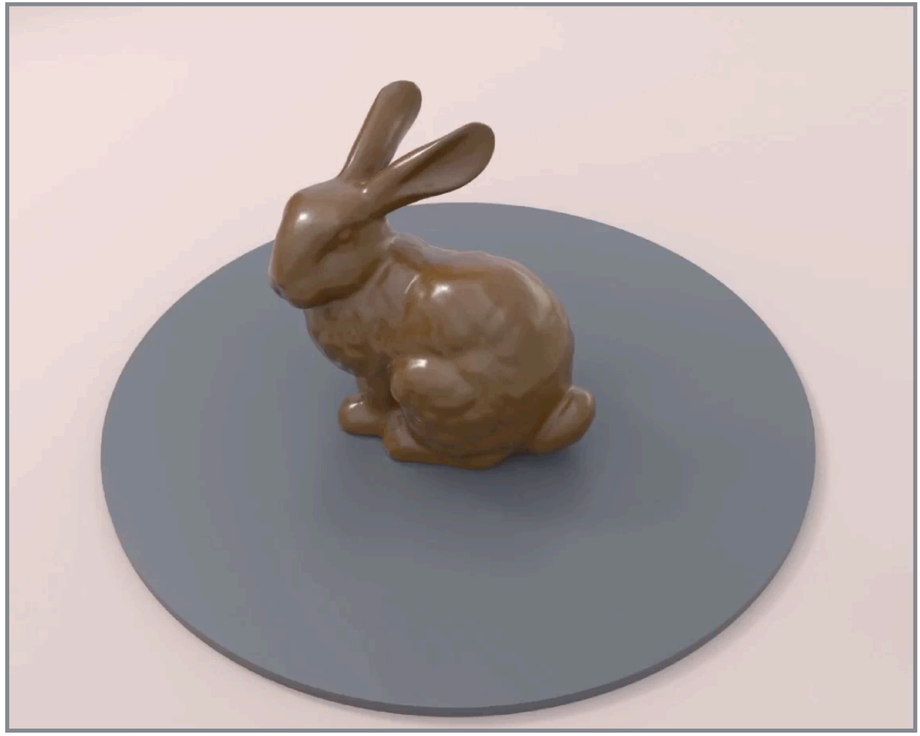


Triangle meshes of 3D objects

Sparsity arises here because most things *aren't* connected.

Sparsity, continued

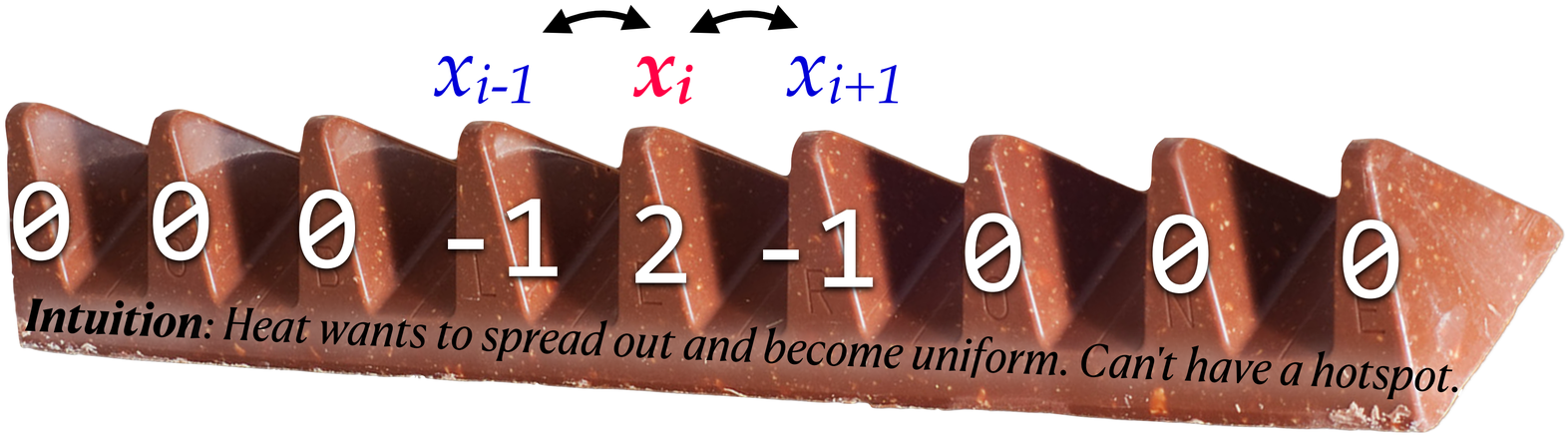
- Sparsity also occurs when one discretizes continuous equations.
- *What does that mean?* Let's look at an example!
- Heat equation (1D): $\dfrac{\partial^2 f(x)}{\partial x^2} = 0.$



[Christopher Batty]

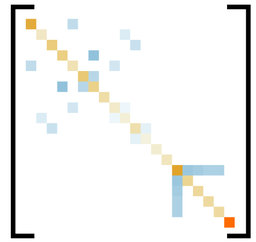
- Computer implementation: $-f(x_{i-1}) + 2f(x_i) - f(x_{i+1}) = 0.$

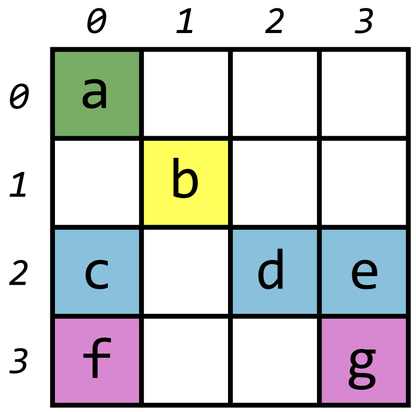
Turns into one row of a **HUGE** matrix:



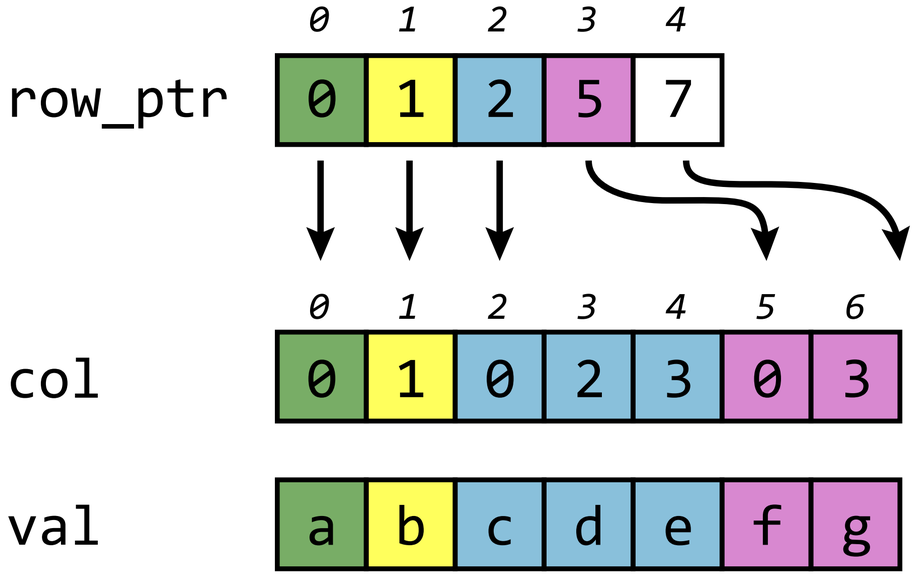
Intuition: *Heat wants to spread out and become uniform. Can't have a hotspot.*

Sparse matrix-vector multiplication





Original matrix



Compressed matrix (CSR format)

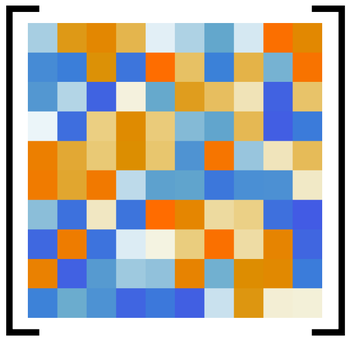
```
y = np.zeros(n)
for i in range(n):
    for j in range(n):
        y[i] += A[i, j] * x[j]
```

Dense matrix-vector multiplication

```
y = np.zeros(n)
for i in range(n):
    for k in range(A.rowptr[i], A.rowptr[i+1]):
        y[i] += A.val[k] * x[A.col[k]]
```

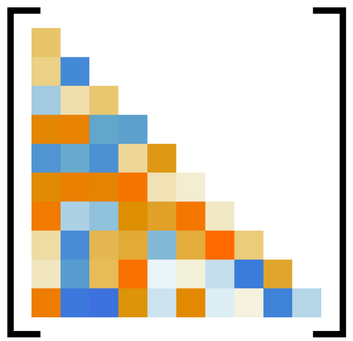
Sparse matrix-vector multiplication (CSR)

Today

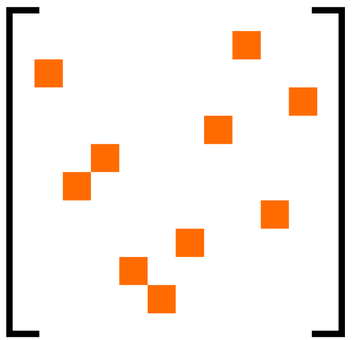


Dense

Upper triangular



Lower triangular



Permutation

Linear system

Recap

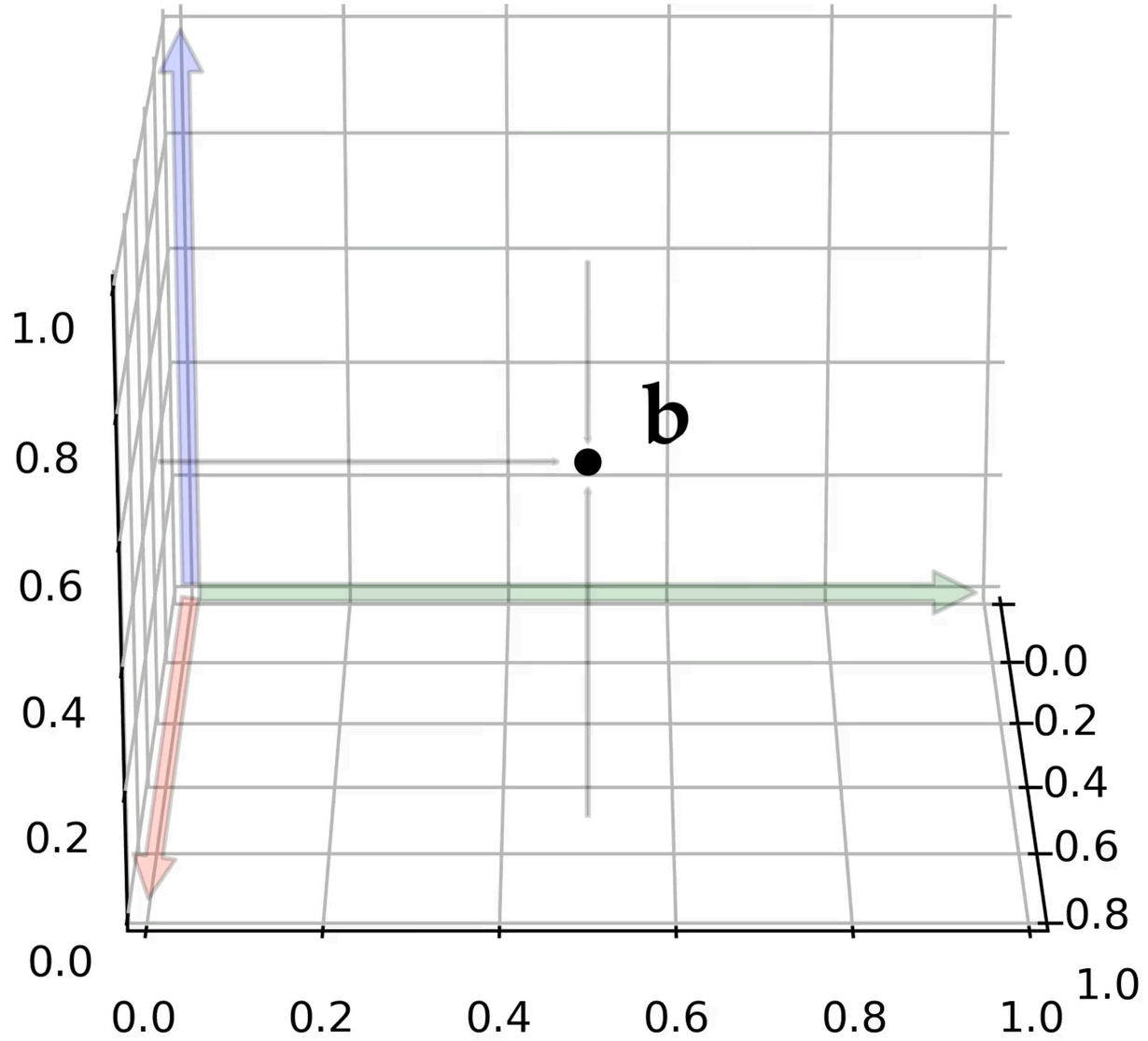
$$\mathbf{A}\mathbf{x} = \mathbf{b}$$

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

- Can $\mathbf{b}$ be expressed as a linear combination of the columns of $\mathbf{A}$?
- Solution may not exist or may not be unique!
- Will focus on *square* case in this lecture ($m = n$).

Geometric interpretation of linear systems

Column-based view



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

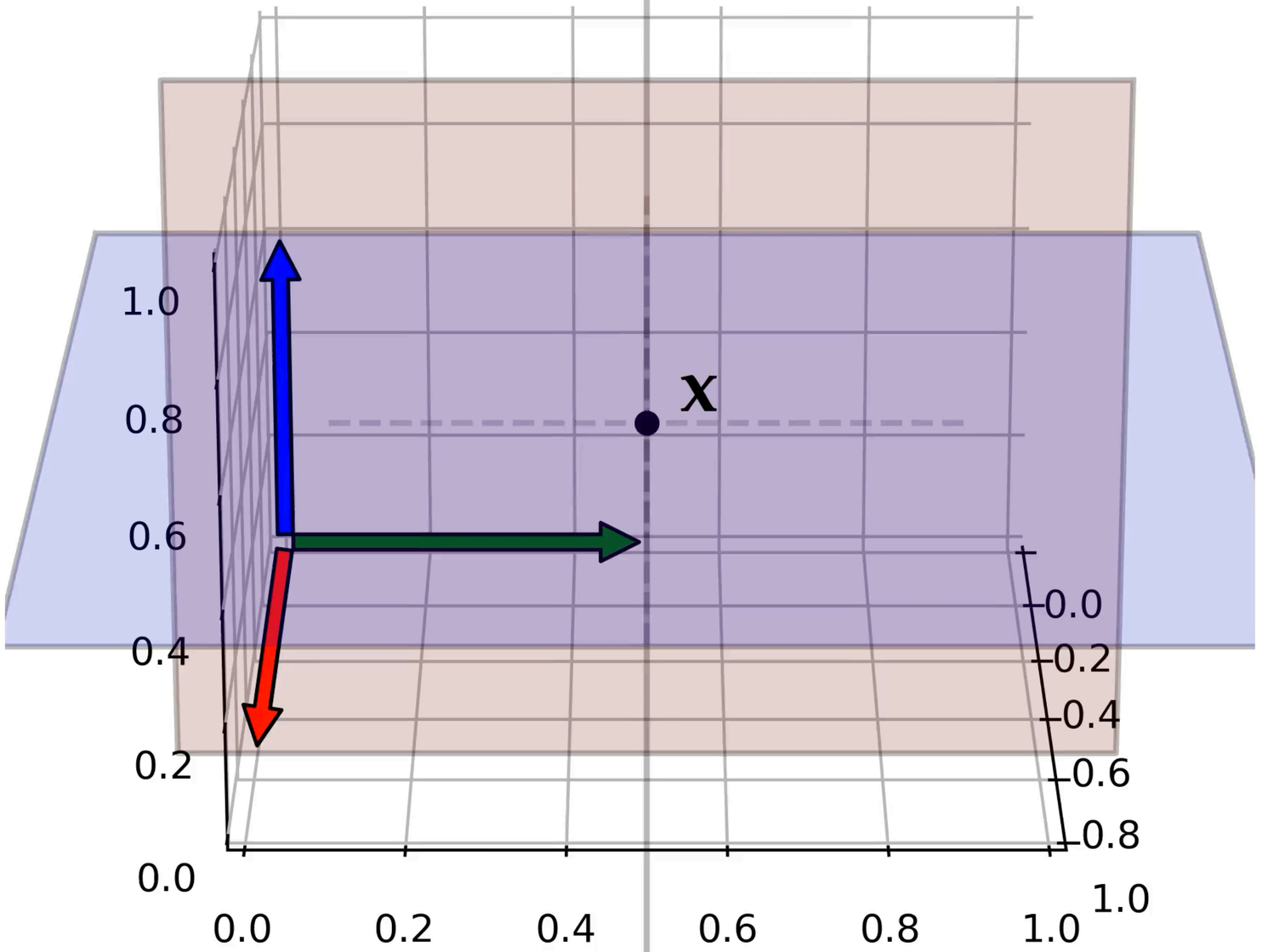
Interpretation: Try to reach point $\mathbf{b}$ using linear combination of basis vectors

$$\mathbf{A} = \begin{pmatrix} | & | & | \\ \mathbf{a}^{(1)} & \mathbf{a}^{(2)} & \mathbf{a}^{(3)} \\ | & | & | \end{pmatrix}$$

The resulting vector $\mathbf{x}$ tells us how much of each axis we need.

Geometric interpretation of linear systems

Row-based view

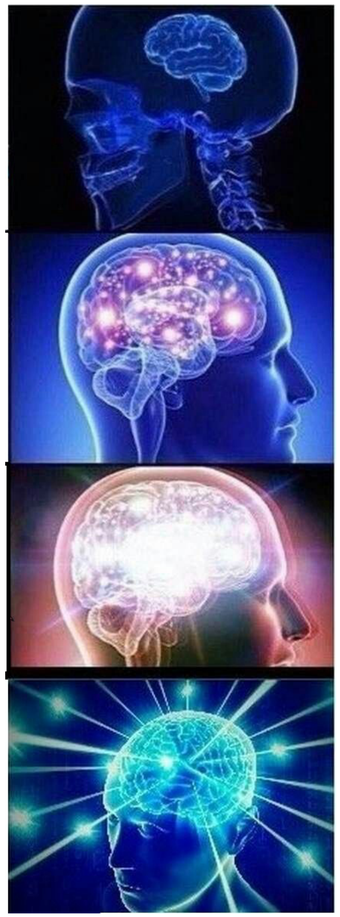


$$\mathbf{A} = \begin{pmatrix} — & \mathbf{a}^{(1)} & — \\ — & \mathbf{a}^{(2)} & — \\ — & \mathbf{a}^{(3)} & — \end{pmatrix}$$

Interpretation: Compute the intersection of planes with normals $\mathbf{a}^{(i)}$ and offset b_i

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

$$\Leftrightarrow \quad \begin{aligned} \mathbf{a}^{(1)} \cdot \mathbf{x} &= b_1 \\ \mathbf{a}^{(2)} \cdot \mathbf{x} &= b_2 \\ \mathbf{a}^{(3)} \cdot \mathbf{x} &= b_3 \end{aligned}$$



$$\mathbf{A} \in \mathbb{R}^{1\times 1}$$

$$\mathbf{A} \in \mathbb{R}^{2\times 2}$$

$$\mathbf{A} \in \mathbb{R}^{3\times 3}$$

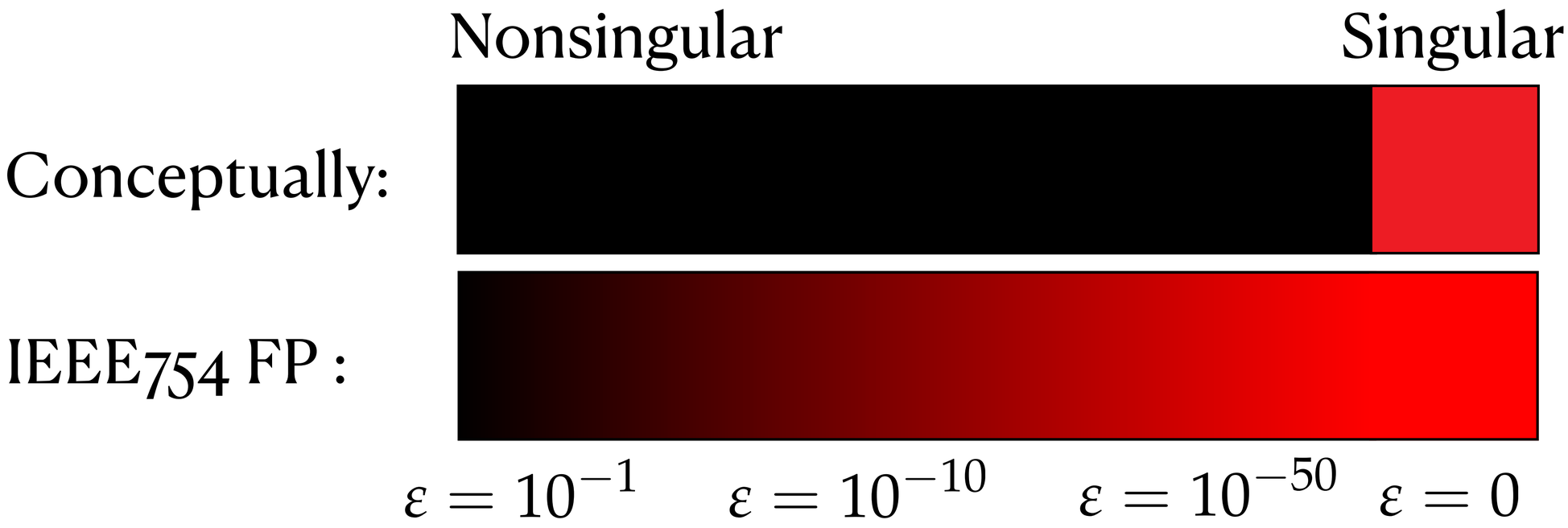
$$\mathbf{A} \in \mathbb{R}^{1000\times 10000}$$

Demo time

Pure versus numerical mathematics

Concepts like invertibility become more nuanced

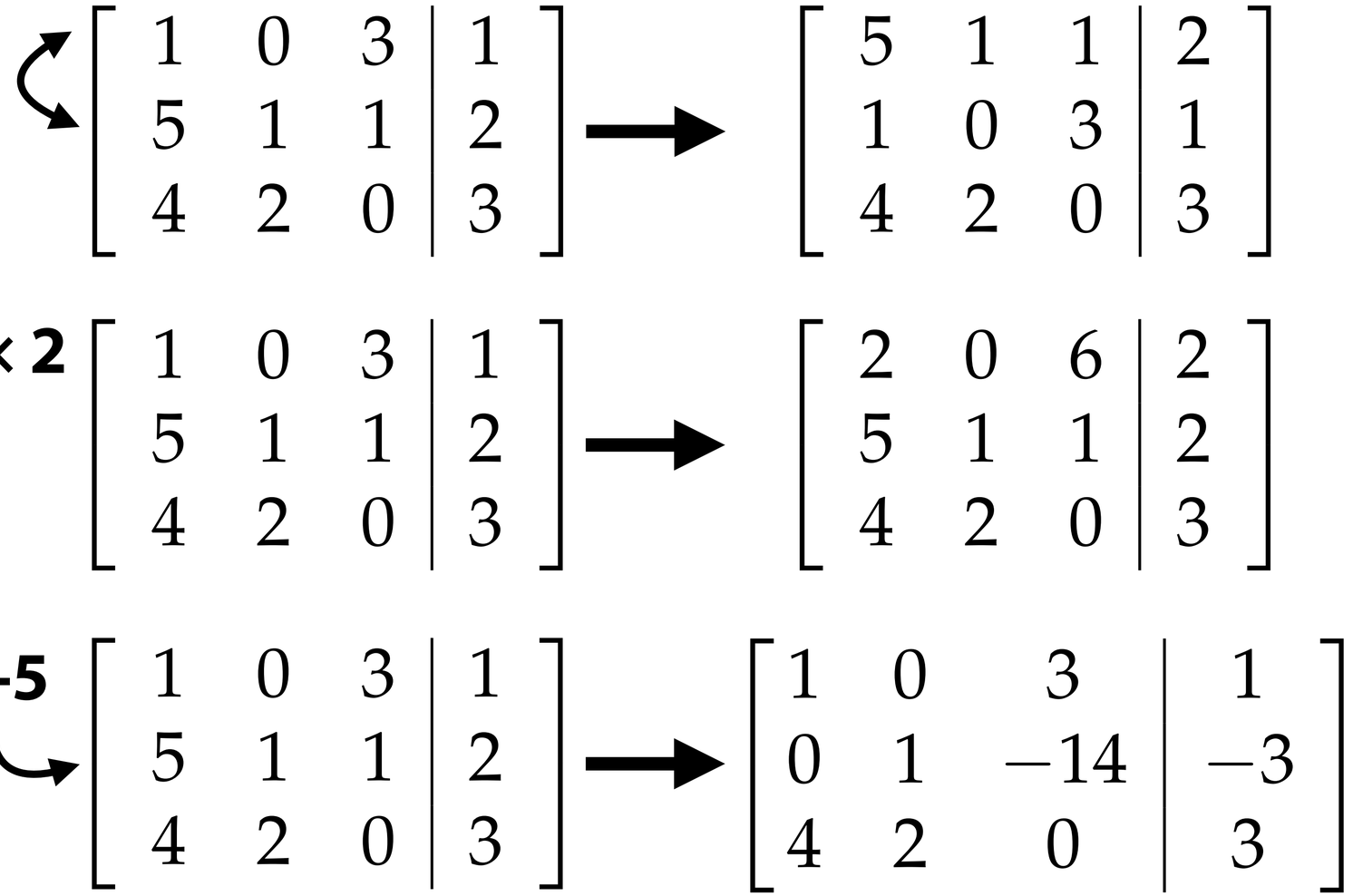
$$\begin{pmatrix} 1 & 0 \\ 1 & \epsilon \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} -1 \\ 1 \end{pmatrix}$$



Solving a linear system MATH-111 style

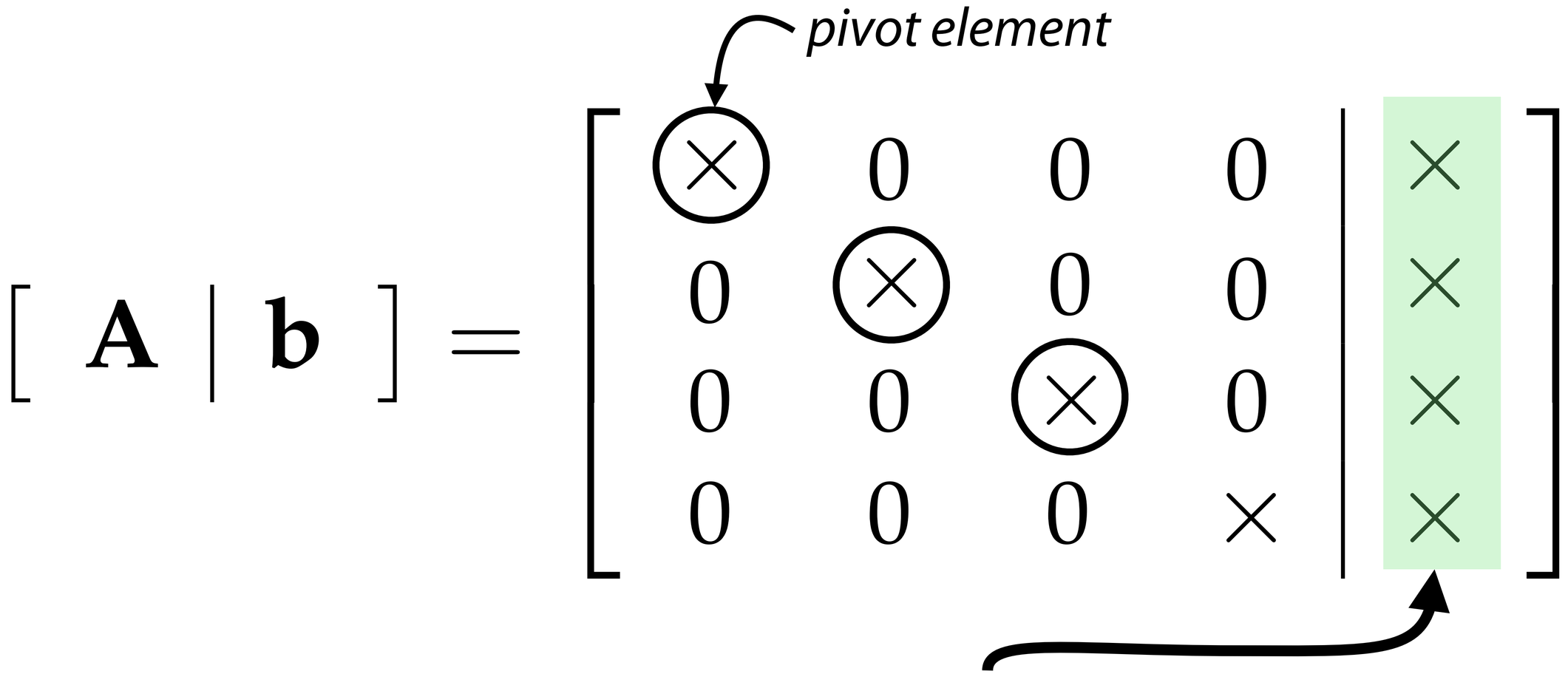
Gaussian Elimination makes a comeback

- **Ingredients:**
  - Permute rows
  - Scale rows
  - Add (scaled) version of row



Solving a linear system MATH-111 style

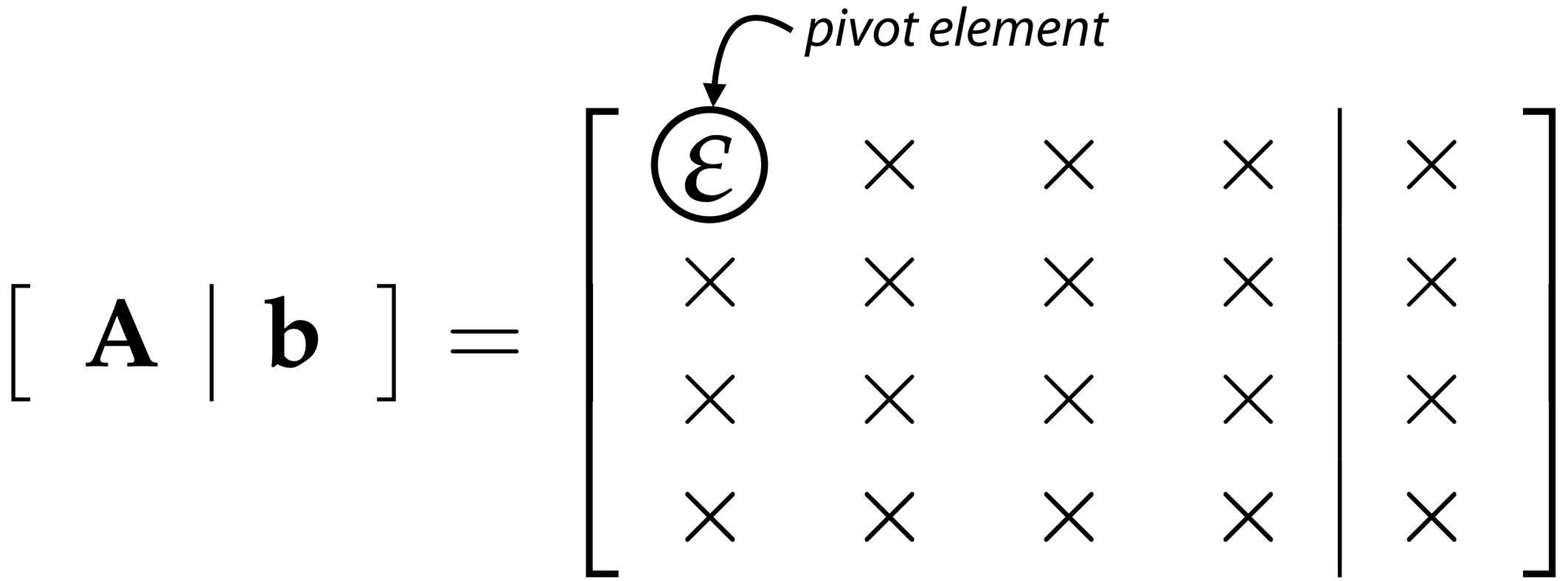
Gaussian Elimination makes a comeback



Observations: values here are updated but don't really influence the process.

Solving a linear system MATH-111 style

Instabilities can occur even for nonzero elements, must choose pivot element carefully.



Practical solution: pick element with largest absolute value in row.

A common situation

What if a linear system needs to be solved over and over again?

$$\mathbf{A}\mathbf{x}^{(1)} = \mathbf{b}^{(1)}$$

$$\mathbf{A}\mathbf{x}^{(2)} = \mathbf{b}^{(2)}$$

$$\mathbf{A}\mathbf{x}^{(3)} = \mathbf{b}^{(3)}$$

$$\vdots$$

This seems really inefficient. Can we *precompute* something to speed things up?

Anti-pattern — don't do this.

Precomputation: compute $\mathbf{A}^{-1}$ from $\mathbf{A}$

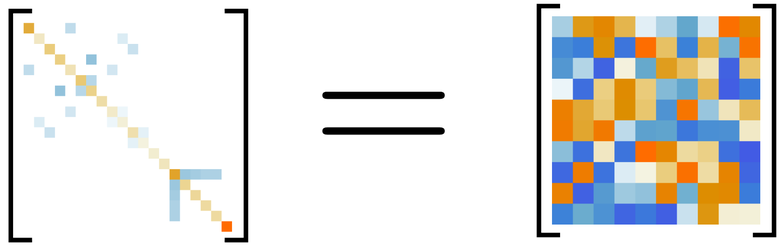
$$\mathbf{x}^{(1)} = \mathbf{A}^{-1}\mathbf{b}^{(1)}$$
$$\mathbf{x}^{(2)} = \mathbf{A}^{-1}\mathbf{b}^{(2)}$$
$$\mathbf{x}^{(3)} = \mathbf{A}^{-1}\mathbf{b}^{(3)}$$
$$\vdots$$

Idea: Precompute matrix inverse, then 1 matrix-vector multiplication per solve.

Why not compute the inverse?

1. Computation of the inverse can be numerically unstable.

2. Computation of inverse + matrix multiplies require slightly more FLOPs than alternative discussed next.

3. Also, generally A^{-1}  Sparse Dense

Will run out of memory 😕

(Alternative discussed next does not have these problems.)

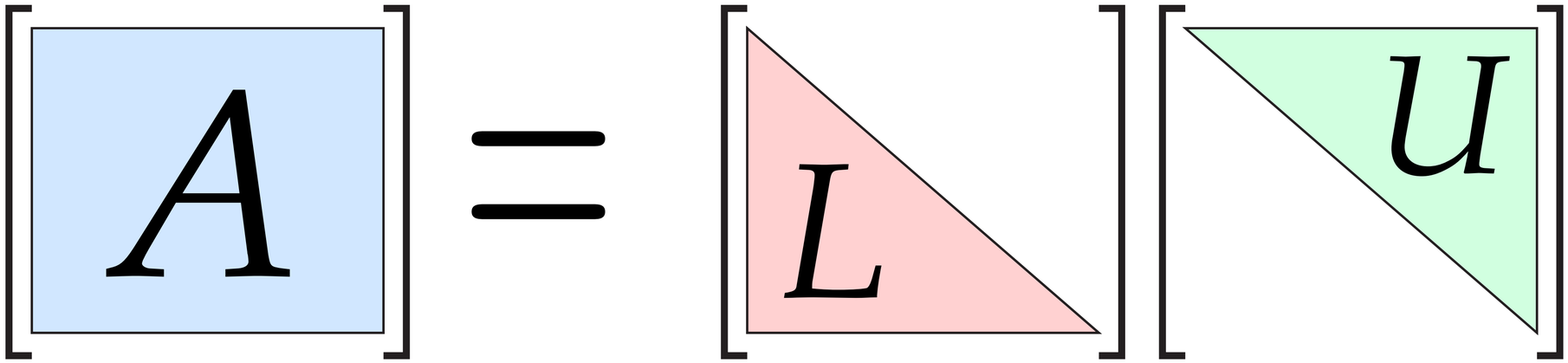
LU Factorization

Our first example of a decomposition.

LU Factorization

Basic motivation

- Run the Gaussian elimination algorithm on $\mathbf{A}$ alone (no right-hand side $\mathbf{b}$)
  - Keep track of what transformations it performs (e.g., adding scaled rows)
  - Those transformations can be written down as matrices $\mathbf{L}$ and $\mathbf{U}$.
  - We can then "apply" $\mathbf{L}$ and $\mathbf{U}$ to any right hand side $\mathbf{b}$ very efficiently.



Elimination, revisited

Adding a scaled row to another row can be expressed using a matrix

A single elimination step:

$$\begin{bmatrix} 1 & 0 & 0 \\ c & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} + cA_{11} & A_{22} + cA_{12} & A_{23} + cA_{13} \\ A_{31} & A_{32} & A_{33} \end{bmatrix}$$

The inverse of an elimination step is simple to compute:

$$\begin{bmatrix} 1 & 0 & 0 \\ c & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}^{-1} = \begin{bmatrix} 1 & 0 & 0 \\ -c & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

This is a **lower triangular matrix**. The *product* of lower-triangular matrices is *always* lower-triangular.

Putting things together

U is the traditional output of Gaussian elimination. L reconstructs A.

$$\mathbf{U} = \underbrace{\mathbf{E}_6\mathbf{E}_5\mathbf{E}_4\mathbf{E}_3\mathbf{E}_2\mathbf{E}_1\mathbf{A}}$$

$$\mathbf{A} = \mathbf{L}\mathbf{U}$$

$$\underbrace{\left(\mathbf{E}_6\mathbf{E}_5\mathbf{E}_4\mathbf{E}_3\mathbf{E}_2\mathbf{E}_1\right)^{-1}}$$

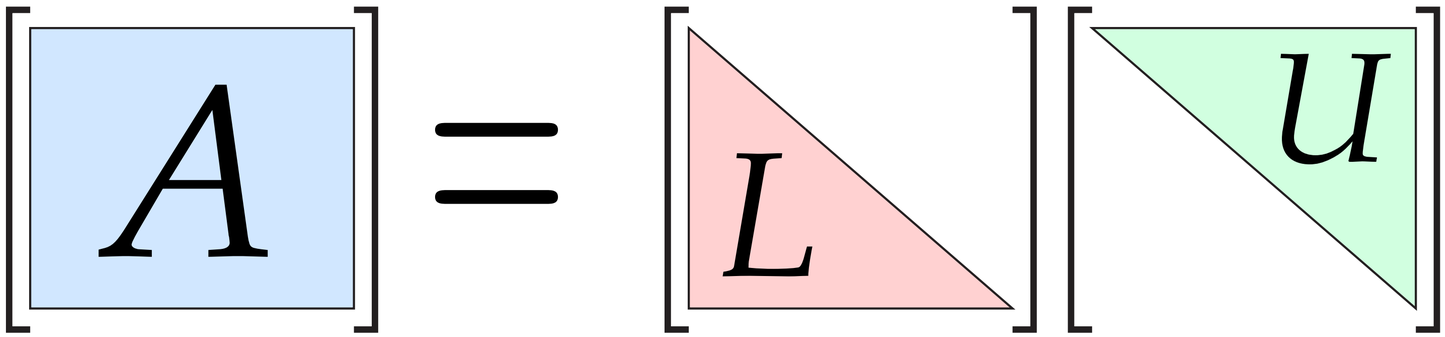
$$= \mathbf{E}_1^{-1}\mathbf{E}_2^{-1}\mathbf{E}_3^{-1}\mathbf{E}_4^{-1}\mathbf{E}_5^{-1}\mathbf{E}_6^{-1}$$

We won't look at low-level details of how the LU factorization is computed.

(ditto for others)

Using a LU decomposition

Once computed, how can it be applied to a right-hand side "b"?



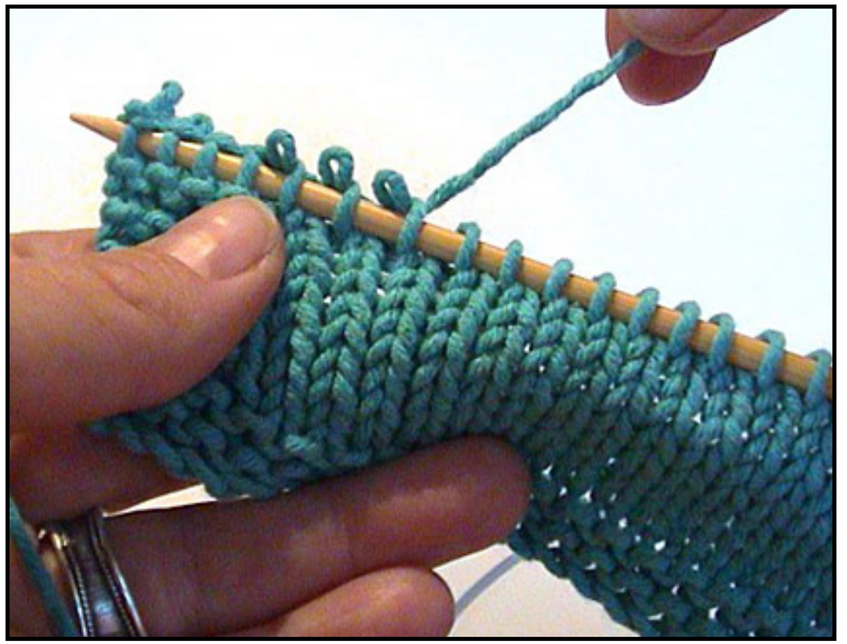
$$\mathbf{x} = \left[U \right]^{-1} \left(\left[L \right]^{-1} \mathbf{b} \right)$$

Important: Here, $..^{-1}$ does *not* mean: compute a matrix inverse.

Solving triangular linear systems

Forward/back-substitution have the same complexity as matrix-vector multiplication

$$\left[\begin{array}{c|c} \mathbf{A} & \mathbf{b} \end{array}\right] =$$



$$\left|\begin{array}{c} \times \\ \times \\ \times \\ \times \end{array}\right]$$

Solving a triangular system takes $O(n^2)$

i.e. *all the hard work is done already!!*

Version with pivoting

Column pivioting, also known as partial pivoting.

$$\mathbf{A} = \mathbf{PLU}$$

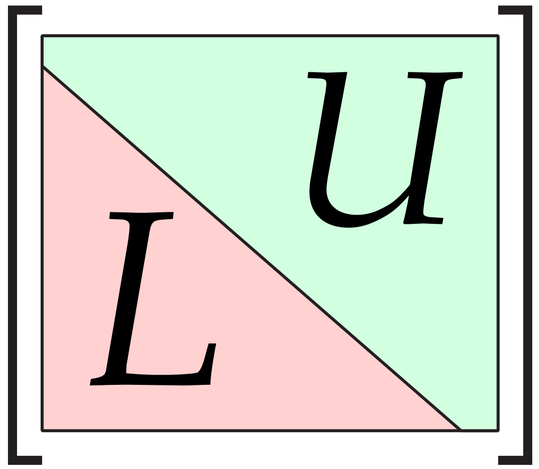
$$\mathbf{x} = \left[U \right]^{-1} \left(\left[L \right]^{-1} \left(\left[P \right]^{-1} \mathbf{b} \right) \right)$$

This is a **permutation matrix** (each row/column contains a single 1, the rest is zero)

In SciPy

```
import scipy.linalg as la
lu, piv = la.lu_factor(A)
x = la.lu_solve((lu, piv), b)
```

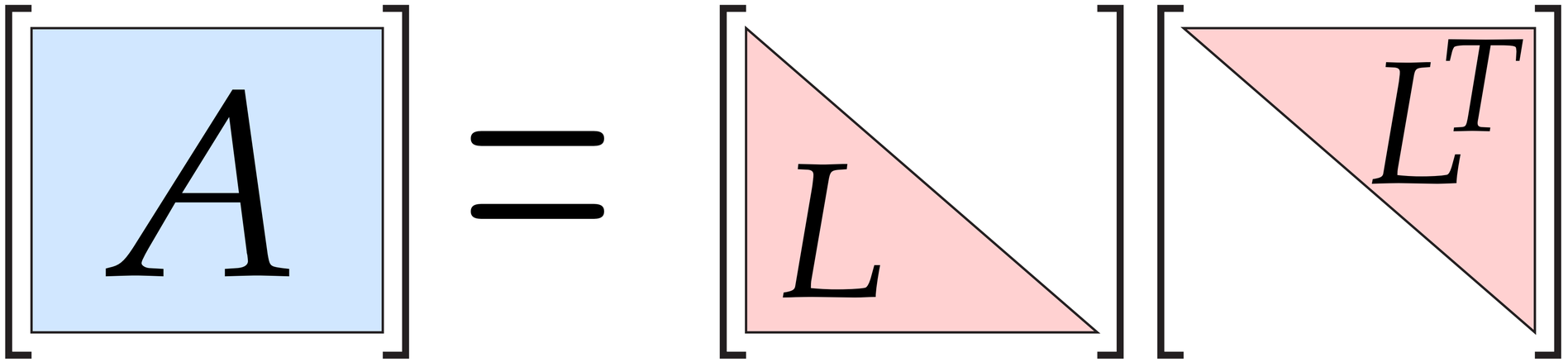
This returns a pivot element vector and a single 2D-array



Diagonal elements of **L** are 1. Can pack both components tightly.

Symmetric positive definite matrices: Cholesky decomposition

It's almost the same as an LU decomposition, but half as expensive (storage, compute)



A 2x2 example:

$$\begin{bmatrix} A_{11} & A_{21} \\ A_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{bmatrix} \begin{bmatrix} L_{11} & L_{21} \\ 0 & L_{22} \end{bmatrix}$$

$$L_{11} = \sqrt{A_{11}}$$

$$L_{21} = \frac{A_{21}}{L_{11}}$$

$$L_{22} = \sqrt{A_{22} - L_{21}^2}$$