Solving linear systems:
LU factorization
The problem

A fundamental equation in computational math is the **linear system**

\[ Ax = b, \quad A = \text{invertible } n \times n \text{ matrix, } b \in \mathbb{R}^n \]

- We will learn a good algorithm to solve it, and translate to python
- The end goal: write a function that looks like this:

```python
def linsolve(a, b):
    ...
    return x
```

\[ a = \begin{bmatrix} 1,2,3,4 \end{bmatrix} \quad \text{# mat[0] = 0-th row} \\
\text{b = [5,11]} \\
x = \text{linsolve(a, b)} \quad \text{# x = [1,2]}
```

- **efficiency** Is important - keep the number of operations low.

---

**Off-by-one?**

Math convention: \( A \) has entries \( a_{ij} \) with \( i,j \) starting at one

Code convention: indexing starts at zero

You will often have to translate from ‘starts at one’ to ‘starts at zero’, e.g. \( a_{12} \) might be \( a[0][1] \). Keep this in mind!
The easy cases: lower triangular

There are two ‘easy’ cases to look at first. Suppose

\[ Ax = b, \quad A = \begin{bmatrix}
  a_{11} & 0 & \ldots & 0 \\
  a_{21} & a_{22} & \ddots & 0 \\
  \vdots & \vdots & \ddots & 0 \\
  a_{n1} & \ldots & a_{n,n-1} & a_{nn}
\end{bmatrix} \]

i.e. \( A \) is a **lower triangular** (LT) matrix. Then

\[ a_{11}x_1 = b_1 \]
\[ a_{21}x_1 + a_{22}x_2 = b_2 \]
\[ \vdots \]
\[ \sum_{j=1}^{i} a_{ij}x_j = b_i \quad (\text{for row } i) \]

- Solve for \( x_1 \), then \( x_2 \), etc. (**forward substitution**):

\[ x_i = \frac{1}{a_{ii}} \left( b_i - \sum_{j=1}^{i-1} a_{ij}x_j \right) \]

(given that \( x_1, \ldots, x_{i-1} \) are already computed).
The easy cases: lower triangular

\[ x_i = \left( b_i - \sum_{j=1}^{i-1} a_{ij} x_j \right) / a_{ii} \]

- Direct 'translation' to python (just remember to index from zero)
- Start with \( i = 1 \) (first row), then \( i = 2 \) etc, computing (??)

```python
def fwd_solve(a, b):
    n = len(b)
    x = [0]*n
    for i in range(n):
        # compute xi here
        x[i] = b[i]
        for j in range(0,i):
            x[i] -= a[i][j]*x[j]
        x[i] /= a[i][i]
    return x
```

Example:

\[
A = \begin{bmatrix} 3 & 0 & 0 \\ 4 & 2 & 0 \\ 1 & 5 & 3 \end{bmatrix}, \quad b = \begin{bmatrix} 3 \\ 2 \\ -1 \end{bmatrix}, \quad x = \begin{bmatrix} 1 \\ -1 \\ 1 \end{bmatrix} \rightarrow \]

\[
a = [[3,0,0],[4,2,0],[1,5,3]]
\]

\[
b = [3,2,-1]
\]

\[
x = fwd_solve(a, b)
\]

\[
# x is [1,-1,1]
\]
The easy cases: lower triangular

\[ x_i = \left( b_i - \sum_{j=1}^{i-1} a_{ij} x_j \right) / a_{ii} \]

- A second option: do the work ‘in-place’:
  overwrite \( b \) with the result and have no return

Once \( b[i] \) is used, the space is free, so we can replace it with \( x[i] \):

\[
\begin{bmatrix}
  b_0 \\
  b_1 \\
  b_2
\end{bmatrix}
\rightarrow
\begin{bmatrix}
  x_0 \\
  b_1 \\
  b_2
\end{bmatrix}
\rightarrow
\begin{bmatrix}
  x_0 \\
  x_1 \\
  b_2
\end{bmatrix}
\rightarrow
\begin{bmatrix}
  x_0 \\
  x_1 \\
  x_2
\end{bmatrix}
\]

---

```python
def fwd_solve(a, b):
    n = len(b)
    x = [0]*n
    for i in range(n):
        x[i] = b[i]
        for j in range(i):
            x[i] -= a[i][j]*x[j]
        x[i]/= a[i][i]
    return x
```

**vs.**

```python
def fwd_solve(a, b):
    n = len(b)
    #b[0], ...b[i-1] contain x-values
    for i in range(n):
        for j in range(i):
            b[i] -= a[i][j]*b[j]
        b[i]/=a[i][i]
```

What are the benefits/disadvantages of each approach?
The easy cases

The second easy case - if $A$ is **upper triangular** (UT),

$$Ax = b,$$

$$A = \begin{bmatrix}
a_{11} & 0 & \ldots & 0 \\
a_{21} & a_{22} & \ddots & 0 \\
\vdots & \vdots & \ddots & 0 \\
a_{n1} & a_{n2} & \ldots & a_{n,n}
\end{bmatrix}$$

- Use **back-substitution** (same as forward, but start at $x_n$)
- Go backwards from $x_n$ down to $x_1$
- Exercise: implement this

```python
def back_solve(a, b):
    n = len(b)
    x = [0]*n
    ...
    return x
```

**example:**

$$A = \begin{bmatrix} 4 & 1 & 2 \\ 0 & 3 & 1 \\ 0 & 0 & 2 \end{bmatrix}, \quad b = \begin{bmatrix} 1 \\ 5 \\ 4 \end{bmatrix}, \quad x = \begin{bmatrix} -1 \\ 1 \\ 2 \end{bmatrix}$$
So how do we solve the problem

\[ Ax = b \]

for a general \( n \times n \) matrix \( A \)? One approach: break into an UT and a LT solve.

**Definition (LU factorization)**

Let \( A \) be an \( n \times n \) matrix. An **LU factorization** of \( A \) has the form

\[ A = LU \]

where \( L \) is **lower** triangular and \( U \) is **upper** triangular.

To solve \( Ax = b \) we can try to:

1) Find an LU factorization of \( A \); then \( LUx = b \).
2) Solve \( Ly = b \) with forward substitution.
3) Solve \( Ux = y \) with backward substitution.

That is, we solve \( L(Ux) = b \) for \( Ux \) then solve for \( x \) from that.

You already know how to do this from linear algebra - Gaussian elimination!
Gaussian elimination

Here’s the algorithm for reducing $A$ to upper triangular form (this will be $U$):

- Initialize $L$ to the identity matrix
- Reduce column 1, column 2, ... up to column $n-1$ of $A$
- To reduce the $k$-th column:
  - For all entries $(i, k)$ below $(k, k)$ in that column:
    - Zero out the $(i, k)$ entry using the row operation
      $$R_i \leftarrow R_i - mR_k, \quad m = \frac{a_{ik}}{a_{kk}}$$
  - Store the multiplier in the $(i, k)$ entry of $L$

The result is that the reduced matrix is $U$, so

$$A = LU, \quad L = \text{lower triangular}, \quad U = \text{upper triangular}.$$  

(Does this always work? No - that we’ll have to fix...)
Example: Consider the \( LU \) factorization for
\[
A = \begin{bmatrix}
4 & -2 & 2 \\
6 & 6 & 18 \\
6 & 6 & 10
\end{bmatrix}.
\]

Two columns to reduce:
\[
\begin{align*}
A : & \quad \begin{bmatrix}
4 & -2 & 2 \\
6 & 6 & 18 \\
6 & 6 & 10
\end{bmatrix} \quad \overrightarrow{R_2 \leftarrow R_2 - \frac{3}{2} R_1} \quad \begin{bmatrix}
4 & -2 & 2 \\
0 & 9 & 15 \\
0 & 9 & 7
\end{bmatrix} \quad \overrightarrow{R_3 \leftarrow R_3 - R_2} \quad \begin{bmatrix}
4 & -2 & 2 \\
0 & 9 & 15 \\
0 & 0 & -8
\end{bmatrix} \\
L : & \quad \begin{bmatrix}
1 & 0 & 0 \\
0 & 1 & 0 \\
0 & 0 & 1
\end{bmatrix} \quad \overrightarrow{3/2 \times R_2 \leftarrow R_2} \quad \begin{bmatrix}
1 & 0 & 0 \\
0 & 1 & 0 \\
3/2 & 0 & 1
\end{bmatrix} \quad \overrightarrow{3/2 \times R_3 \leftarrow R_3} \quad \begin{bmatrix}
1 & 0 & 0 \\
0 & 1 & 0 \\
3/2 & 1 & 1
\end{bmatrix}
\end{align*}
\]

Result: \( A = LU \) where
\[
L = \begin{bmatrix}
1 & 0 & 0 \\
3/2 & 1 & 0 \\
3/2 & 1 & 1
\end{bmatrix}, \quad U = \begin{bmatrix}
4 & -2 & 2 \\
0 & 9 & 15 \\
0 & 0 & -8
\end{bmatrix}.
\]
Gaussian elimination (Aside: theory)

Why does this work?
- Elementary row operations are matrices, e.g.

\[
E = \begin{bmatrix}
1 & 0 & 0 \\
0 & 1 & 0 \\
\lambda & 0 & 1,
\end{bmatrix}, \quad EA = \text{adds } \lambda R_1 \text{ to } R_3
\]

- The inverse of this RO is simple - subtract instead of add:

\[
E^{-1} = \begin{bmatrix}
1 & 0 & 0 \\
0 & 1 & 0 \\
-\lambda & 0 & 1,
\end{bmatrix}, \quad E^{-1}A = \text{subtracts } \lambda R_1 \text{ from } R_3
\]

The reduction process, in matrix form, then looks like:

\[
M_{n-1}M_{n-2}\cdots M_1A = U,
\]

\[
M_k = \text{product of RO's to reduce that column.}
\]

These matrices just have \(\lambda\)'s in corresponding entries, e.g.

\[
M = \begin{bmatrix}
1 & 0 & 0 \\
-x & 1 & 0 \\
-y & 0 & 1
\end{bmatrix} \text{ does the row ops. } \begin{cases}
R_2 \leftarrow R_2 - xR_1 \\
R_3 \leftarrow R_3 - yR_1
\end{cases}
\]
Gaussian elimination (Aside: theory)

The reduction process, in matrix form, is:

\[ M_{n-1}M_{n-2} \cdots M_1 A = U \]

\[ M_k = \text{row ops to reduce the } k\text{-th column} \]

It follows that \( A = LU \) where

\[ L = M_1^{-1} \cdots M_{n-1}. \]

Example:

\[
\begin{pmatrix}
4 & -2 & 2 \\
6 & 6 & 18 \\
6 & 6 & 10
\end{pmatrix}
\xrightarrow{R_2 \leftarrow R_2 - \frac{3}{2}R_1}
\xrightarrow{R_3 \leftarrow R_3 - \frac{3}{2}R_2}
\begin{pmatrix}
4 & -2 & 2 \\
0 & 9 & 15 \\
0 & 9 & 7
\end{pmatrix}
\xrightarrow{R_3 \leftarrow R_3 - R_2}
\begin{pmatrix}
4 & -2 & 2 \\
0 & 9 & 15 \\
0 & 0 & -8
\end{pmatrix}
\]

Row reduction matrices:

\[
M_1 = \begin{bmatrix}
1 & 0 & 0 \\
-3/2 & 1 & 0 \\
-3/2 & 0 & 1
\end{bmatrix}, \quad M_2 = \begin{bmatrix}
1 & 0 & 0 \\
0 & 1 & 0 \\
0 & -1 & 1
\end{bmatrix}
\]
Observe that

\[ M_k^{-1} = \text{same as } M_k, \text{ but with multiplier signs reversed} \]

since \( M_k \) is inverted by adding instead of subtracting rows...

\[
M = \begin{bmatrix}
1 & 0 & 0 \\
-x & 1 & 0 \\
-y & 0 & 1
\end{bmatrix} \implies M^{-1} = \begin{bmatrix}
1 & 0 & 0 \\
x & 1 & 0 \\
y & 0 & 1
\end{bmatrix}
\]

\[ M = \text{row ops.} \left( \begin{array}{c}
R_3 \leftarrow R_2 - xR_1 \\
R_3 \leftarrow R_3 - yR_1
\end{array} \right) \implies M^{-1} = \text{row ops.} \left( \begin{array}{c}
R_3 \leftarrow R_2 + xR_1 \\
R_3 \leftarrow R_3 + yR_1
\end{array} \right) \]

Finally, we claim that

\[ L = M_1^{-1} \cdots M_{n-1}^{-1} = \text{ROs to reduce } A \text{ in reverse with opposite signs} \]

\[ = \text{matrix of multipliers} \]

where the ‘multiplier’ for \( R_j \rightarrow R_j - \lambda R_i \) is \( \lambda \). To show this...
Gaussian elimination (Aside: theory)

... requires a bit of work; each $M$ deposits its multipliers into $L$, and later $M$’s do not affect existing columns.

\[ L = M_1^{-1} \cdots M_{n-1}^{-1} \]

= ROs to reduce $A$ in reverse with opposite signs

Example:

\[
M_1 = \begin{bmatrix}
1 & 0 & 0 \\
-3/2 & 1 & 0 \\
-3/2 & 0 & 1
\end{bmatrix}, \quad M_2 = \begin{bmatrix}
1 & 0 & 0 \\
0 & 1 & 0 \\
0 & -1 & 1
\end{bmatrix}
\]

\[
L = \begin{bmatrix}
1 & 0 & 0 \\
3/2 & 1 & 0 \\
3/2 & 0 & 1
\end{bmatrix} \begin{bmatrix}
1 & 0 & 0 \\
0 & 1 & 0 \\
0 & 1 & 1
\end{bmatrix} = \begin{bmatrix}
1 & 0 & 0 \\
3/2 & 1 & 0 \\
3/2 & 1 & 1
\end{bmatrix}
\]

$L$ can be computed by applying the ROs

\[
R_3 \rightarrow R_3 + R_2
\]

\[
R_3 \rightarrow R_3 - \frac{3}{2}R_1
\]

\[
R_2 \rightarrow R_2 - \frac{3}{2}R_1
\]
Gaussian elimination

- $k$: index of column to reduce
- $i$: row to reduce
- $j$: element of that row

(Assume you can copy a and create a zero matrix)

```python
def ge_lu(a):
    u = a.copy()
    ell = zeros([n,n])
    for k in range(n-1):
        # reduce k-th column of u
        # (rows k+1 through n-1)
        return ell, u
    return ell, u
```

But this leaves empty space in `ell` and `u`!

We want to make the code more compact...
Gaussian elimination

Storage:

• The algorithm can be written ‘in-place’, overwriting $A$
• Regardless, we can store $L$ in the unused half of $U$
• This works even if in-place (‘zeroed’ entries of $A$ are free space)

```python
def ge_lu(a):
    for k in range(n-1):
        for i in range(...):
            # replace zeroed entry with mult.
            # update rows in a directly
            # (no return, both $L$ and $U$ in a)
```

• Typical: ‘return’ one matrix containing both $L$ and $U$ (compact form)
• But $L$ and $U$ both have diagonal entries?
Compact version of previous example:

\[
A = \begin{bmatrix}
4 & -2 & 2 \\
6 & 6 & 18 \\
6 & 6 & 10
\end{bmatrix}.
\]

Store multipliers in the zeroed entries (shown in red):

\[
A : \begin{bmatrix}
4 & -2 & 2 \\
6 & 6 & 18 \\
6 & 6 & 10
\end{bmatrix} \overset{R_2 \leftarrow R_2 - \frac{3}{2} R_1}{\longrightarrow} \begin{bmatrix}
4 & -2 & 2 \\
3/2 & 9 & 15 \\
3/2 & 9 & 7
\end{bmatrix} \overset{R_3 \leftarrow R_3 - \frac{3}{2} R_2}{\longrightarrow} \begin{bmatrix}
4 & -2 & 2 \\
3/2 & 9 & 15 \\
3/2 & 1 & -8
\end{bmatrix}
\]

Result: a single matrix storing \( L \) and \( U \):

\[
\text{result} = \begin{bmatrix}
4 & -2 & 2 \\
3/2 & 9 & 15 \\
3/2 & 1 & -8
\end{bmatrix}
\]

Note: if used to solve \( LUx = b \), be careful with the indexing - e.g. if the result is \( \text{ellu} \) then \( \text{ellu}[1][0] \) is part of \( L \) but \( \text{ellu}[0][1] \) is part of \( U \).
LU factorization

Back to solving $Ax = b$... recall our algorithm had two parts:

1) The ‘factor’ step: Find an LU factorization of $A$; then $LUx = b$.
2) The ‘solve’ step:
   - Solve $Ly = b$ with forward substitution.
   - Solve $Ux = y$ with backward substitution.

• The actual code for solving $Ax = b$ will then look like:

```python
#(given a matrix a, vector b)
fact = lu_factor(a)
y = fwd_solve_lu(fact, b)
x = back_solve_lu(fact, y)
```

• Note that the ‘solve’ functions are specialized and not general forward/back solve routines; they assume `fact` is $LU$ in compact form

• Note that a new array $y$ is unnecessary (we can do some overwriting!)
Why factor?

- The factor/solve split lets us quickly solve with the same \( A \) repeatedly, e.g.
  
  ```python
  lu = lu_factor(a)  # expensive
  x1 = lu_solve(lu, b1)  # cheap!
  x2 = lu_solve(lu, b2)  # cheap!
  ```

- This is (almost always) better than computing the inverse \( A^{-1} \)

**Key point: no inverses!**

In numerical linear algebra, you should think:

\[
A^{-1} b \text{ means to solve } Ax = b
\]

i.e. you almost never actually compute \( A^{-1} \) to compute \( A^{-1} b \).
Your ‘\( Ax = b \)’ solver is then also a ‘multiply \( b \) by \( A^{-1} \)’ routine.

**Question:** suppose we want to solve

\[
A^2 x = b
\]

where is \( A \) an \( n \times n \) matrix

**Answer:**
- Compute \( L, U \) so that \( A = LU \)
- Use this to solve \( Ay = b \), then \( Ax = y \)

(Effectively: \( x = A^{-1}(A^{-1}b) \))
Big-O
Big-O notation

- We want to express the computational cost of an algorithm as it scales
- Big-O notation describes size to ‘leading order’

**Definition (Big-O (sequences))**

A sequence $a_n$ is said to be Big-O of a sequence $b_n$, written

$$a_n = O(b_n)$$

if it holds that

$$|a_n| \leq C|b_n| \text{ as } n \to \infty$$

for some constant $C$. (That is, it holds for $n$ large enough).

- Measures how fast a sequence grows. Typical rates:
  $$O(1), \quad O(n), \quad O(n \log n), \quad O(n^2), \quad O(n^3), \cdots$$
  
- ‘Leading order’ behavior, e.g.
  $$a_n = 2n^3 + 4n^2 + 1 \implies a_n = O(n^3) \text{ or } a_n = 2n^3 + O(n^2)$$

- Caution: the ‘equals’ here is not really equals (**not symmetric**!):
  $$n^2 \text{ is } O(n^3) \text{ but } n^3 \text{ is not } O(n^2)$$
**Definition**

**Big-O (sequences)** A sequence $a_n$ is said to be little-o of a sequence $b_n$, written

$$a_n = o(b_n)$$

if it holds that

$$\lim_{n \to \infty} \frac{a_n}{b_n} = 0.$$ 

Similarly, we say that $a_n$ is asymptotic to $b_n$ (written $a_n \sim b_n$) if

$$\lim_{n \to \infty} \frac{a_n}{b_n} = 1.$$ 

- Asymptotic-to precisely describes leading order behavior, e.g.

  $$a_n = 2n^3 + 4n^2 + 1 \implies a_n \sim 2n^3 \text{ as } n \to \infty.$$ 

- Little-o describes 'smaller terms', e.g.

  $$a_n = 2n^3 + o(n^3).$$ 

- Note that $a_n \sim b_n$ if and only if $a_n = b_n + o(b_n)$. 
Computational complexity

A simple way to measure computational cost: count the steps.

- What operations take time?
  - flop (floating point operation): add/subtract, mult/divide
- Assignment is cheaper than arithmetic (omitted here for simplicity)
- Other issues: loading/unloading in memory, conditionals... (also omitted)

Empirical approach: test and time directly (use e.g. `time.perf_counter()`)

```python
import time
start = time.perf_counter()
# ...some code
elapsed = time.perf_counter() - start
```

Best practices

Using actual time (called clock time) to measure your program is unreliable - it may vary due to internal memory, other processes on your cpu, etc.

To get a good measure, take a large number of runs and average the result! In addition, see how it scales with problem size - only relative times matter since computer power varies.
For simplicity, we count the number of mults. (multiplies) only.

Example - matrix-vector multiplication:

\[ y = Ax \implies y_i = \sum_{j=1}^{n} a_{ij} x_j \text{ for } 1 \leq i \leq n. \]

Note: here we do not take into account relative costs of operations!

• For each \( i \), there are \( n \) multiplies

\[ \text{# of mults} = n \cdot n \implies n^2. \]

• If you also count additions then

\[ \text{# of flops} = 2n^2 + O(n). \]
Computational complexity

Now suppose that $A$ is tridiagonal:

$$
A = \begin{bmatrix}
    a_1 & b_1 & 0 & \cdots & 0 \\
    c_2 & a_2 & b_2 & \ddots & \vdots \\
    0 & \ddots & \ddots & \ddots & 0 \\
    \vdots & \ddots & c_{n-1} & a_{n-1} & b_{n-1} \\
    0 & \cdots & 0 & c_n & a_n
\end{bmatrix}
$$

That is only, only one diagonal above/below have non-zero entries. How many multiplies are needed to compute $Ax$?

Answer: three per row for rows $i = 2, \cdots, n-1$ so

$$
\text{# of mults} = 3n + O(1).
$$

This is an example of a sparse matrix (a matrix with mostly zeros). For such matrices, linear algebra operations are fast. Example:

$$
twitter \text{ users } i = 1, \cdots n, \quad a_{ij} = \begin{cases} 
1 & \text{i follows j} \\
0 & \text{otherwise}
\end{cases}
$$

$O(n^2) \sim (300 \text{ million})^2 = \text{ way too much computation}$
Recall that to solve $Ax = b$ we needed two separate parts:

**Substitution** (Forward or back): As a reminder, the forward formula is

$$x_i = \left( b_i - \sum_{j=1}^{i-1} a_{ij} x_j \right) / a_{ii}$$

**Gaussian elimination** (to compute the $L$ and $U$):

- For $k$ from 1 to $n - 1$:
  - For each row $i$ from $k + 1$ to $n$:
    - zero out the $a_{ik}$ entry with
      $$R_i \leftarrow R_i - \frac{a_{ik}}{a_{kk}} R_k,$$

You can show (exercise) that

- Forward/back substitution take $\frac{1}{2}n^2 + O(n)$ mults.
- The LU step (Gaussian elimination) takes $\frac{1}{3}n^3 + O(n^2)$ mults.
So to solve $Ax = b$,

- **Factor**: $A = LU$ (steps: $\sim n^3/3$)
- **Forward solve**: $Ly = b$ (steps: $\sim n^2/2$)
- **Back solve**: $Ux = y$ (steps: $\sim n^2/2$)

Most of the work happens in the factor step; the rest is (relatively) faster.

Thus, **given** $A = LU$,

computing $A^{-1}b$ takes $n^2 + O(n)$ mults.

and the factor step only has to be done once.

Since matrix *multiplication* is $n^2$ mults, this is quite good!
pivoting
Now let's return to Gaussian elimination (with math indexing):

- For \( k \) from 1 to \( n - 1 \):
  - For each row \( i \) from \( k + 1 \) to \( n \):
    - zero out the \( a_{ik} \) entry with
      \[
      R_i \leftarrow R_i - \frac{a_{ik}}{a_{kk}} R_k,
      \]

Call the partially reduced matrix that we update the ‘working matrix’.

Question: When does this algorithm work?

Answer: At the \( k \)-th step, we need the pivot element \( a_{kk} \) to be non-zero.

This means GE can fail for invertible matrices - not good!

\[
A = \begin{bmatrix}
1 & 1 & 1 & 1 \\
1 & 1 & 2 & 1 \\
0 & 5 & 2 & 1 \\
0 & 3 & 0 & 7
\end{bmatrix}
\begin{bmatrix}
1 & 1 & 1 & 1 \\
0 & 0 & 1 & 0 \\
0 & 5 & 2 & 1 \\
0 & 3 & 0 & 7
\end{bmatrix}
\begin{bmatrix}
1 & 1 & 1 & 1 \\
0 & 0 & 1 & 0 \\
0 & 5 & 2 & 1 \\
0 & 3 & 0 & 7
\end{bmatrix}
\begin{bmatrix}
1 & 1 & 1 & 1 \\
0 & 0 & 1 & 0 \\
0 & 5 & 2 & 1 \\
0 & 3 & 0 & 7
\end{bmatrix}
\]
To fix this, we must perform another row operation: **swapping rows**.

\[
\begin{bmatrix}
1 & 1 & 1 & 1 \\
0 & 0 & 1 & 0 \\
0 & 3 & 2 & 1 \\
0 & 3 & 0 & 7
\end{bmatrix}
\Rightarrow
\begin{bmatrix}
1 & 1 & 1 & 1 \\
0 & 3 & 2 & 1 \\
0 & 0 & 1 & 0 \\
0 & 3 & 0 & 7
\end{bmatrix}
\Rightarrow
\begin{bmatrix}
1 & 1 & 1 & 1 \\
0 & 3 & 2 & 1 \\
0 & 0 & 1 & 0 \\
0 & 0 & \_2 & 6
\end{bmatrix}
\]

Each row swap can be written in matrix form, e.g.

\[
\begin{bmatrix}
0 & 0 & 1 \\
0 & 1 & 0 \\
1 & 0 & 0
\end{bmatrix}
\]

swaps rows 1 and 3

A **permutation matrix** is a product of swaps, e.g.

\[
P = \begin{bmatrix}
0 & 0 & 1 \\
1 & 0 & 0 \\
0 & 1 & 0
\end{bmatrix}
\leftrightarrow
\begin{pmatrix}
\text{swap } R_1 \leftrightarrow R_3 \\
\text{then } R_2 \leftrightarrow R_3
\end{pmatrix}
\]

The net effect is that \( PA \) permutes rows 1 \(\rightarrow\) 2 \(\rightarrow\) 3 \(\rightarrow\) 1.
In matrix form, Gaussian elimination then has the form
\[ M_{n-1}P_{n-1} \cdots M_2P_2M_1P_1A = U \] (G)

where the \( M \)'s and \( P \)'s are the row reductions and row swaps.

- GE with pivoting always works if \( A \) is invertible (lin. al. exercise: why?)
- Some work required to simplify the mess of \( P \)'s inside...

**Theorem (Gaussian elimination)**

The row-swaps can be ‘factored’ out of (G).
The result is that if \( A \) is invertible, then

\[ PA = LU \]

where
\[ P = P_{n-1} \cdots P_1 \] is the product of the row swaps
\[ U \] is the UT reduced matrix from GE
\[ L \] is the LT matrix of multipliers

In short: if you knew the row swaps in advance, you could apply them to \( A \) first (to get \( PA \)) then apply GE without pivoting to \( PA \) to get \( L, U \).
Pivoting: in practice

\[ M_{n-1}P_{n-1} \cdots M_2P_2M_1P_1A = U \]

- The GE algorithm still has to do the row swaps as it goes
- We do **not want to store** \( P \) as a matrix

**Definition**

For a permutation matrix \( P \), the corresponding **permutation vector** is the result of applying the row swaps to the list \( p = \{1, 2, \cdots, n\} \)

For example:

\[
\begin{pmatrix}
\text{swap } R_1 \leftrightarrow R_3, \\
\text{then } R_2 \leftrightarrow R_3
\end{pmatrix} \rightarrow P = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \rightarrow p = \{3, 1, 2\}
\]

by swapping \( p_1 \) and \( p_3 \), then \( p_2 \) and \( p_3 \). This completely describes \( P \)!
Pivoting: in practice

Definition
For a permutation matrix $P$, the corresponding **permutation vector** is the result of applying the row swaps to the list

$$p = \{1, 2, \cdots, n\}$$

For example:

$$\left( \begin{array}{c} \text{swap } R_1 \leftrightarrow R_3, \\ \text{then } R_2 \leftrightarrow R_3 \end{array} \right) \rightarrow P = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \rightarrow p = \{3, 1, 2\}$$

by swapping $p_1$ and $p_3$, then $p_2$ and $p_3$. This completely describes $P$! A useful rule is that

the $i$-th row of $PA = p(i)$-th row of $A$.

For example, for the $P$ above,

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}, \quad PA = \begin{bmatrix} 7 & 8 & 9 \\ 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}, \quad p = \{3, 1, 2\}$$

and row 3 of $PA$ is row $p(3) = 2$ of $A$. 
Pivoting: in practice

Implementation:

- It is important to swap even if the entry is non-zero
- Small pivot elements can amplify error in row reduction:

\[ R_i \leftarrow R_i - \frac{a_{ik}}{a_{kk}} R_k \]

- To keep the algorithm stable (minimize accumulation of error), we need to swap rows to keep the pivot element \( a_{kk} \) large
- We don’t want to waste too much time - \( O(n^2) \) is okay, \( O(n^3) \) is not

Partial pivoting

A typical pivoting scheme is **partial pivoting**:

- Look at \( a_{k+1,k}, \ldots, a_{n,k} \) (entries below the diagonal in col. \( k \))
- Find the index \( r > k \) such that \( a_{r,k} \) has the largest magnitude
- Swap rows \( r \) and \( k \)

While not perfect, it is usually good enough.
Here's an example: For $k = 1$ (first column), with actual swaps:

\[
\begin{bmatrix}
1 & 2 & 4 \\
1 & 0 & 1 \\
-2 & 2 & 4
\end{bmatrix}
\rightarrow
\begin{bmatrix}
-2 & 2 & 4 \\
1 & 0 & 1 \\
1 & 2 & 4
\end{bmatrix}
\Rightarrow
\begin{bmatrix}
-2 & 2 & 4 \\
-1/2 & 1 & 3 \\
-1/2 & 3 & 6
\end{bmatrix}, \quad p = \{3, 2, 1\}
\]

with row operations $R_2 \leftarrow R_2 - (-1/2)R_1$ and $R_3 \leftarrow R_3 - (-1/2)R_1$.

Now for $k = 2$ with actual swaps:

\[
\begin{bmatrix}
-2 & 2 & 4 \\
-1/2 & 1 & 3 \\
-1/2 & 3 & 6
\end{bmatrix}
\rightarrow
\begin{bmatrix}
-2 & 2 & 4 \\
-1/2 & 3 & 6 \\
-1/2 & 1 & 3
\end{bmatrix}
\Rightarrow
\begin{bmatrix}
-2 & 2 & 4 \\
-1/2 & 3 & 6 \\
-1/2 & 1/3 & 1
\end{bmatrix}
\]

with row op $R_3 \leftarrow R_3 - \frac{1}{3}R_2$ and $p$ updated from $\{3, 2, 1\}$ to $\{3, 1, 2\}$. 
Pivoting

How do we swap rows? Assume that the matrix is a list of rows like

```
# row 1  row 2  row 3
a = [[1,2,4],[1,0,1],[-2,2,4]]
```

for

```
A = \[
\begin{bmatrix}
1 & 2 & 4 \\
1 & 0 & 1 \\
2 & 2 & 4
\end{bmatrix}
\]
```

(This is called **row major** because rows are the first index. The other ordering is called **column major**).

To swap two rows we only need to **swap the references** to the data, e.g.

```
tmp = a[0]
a[0] = a[2]
a[2] = tmp
```

makes `a[0]` point to the old data of `a[2]` and vice versa. This is much faster than copying the data!
Pivoting: implementation

To use this factorization, we need to update the back/forward solves also since

\[ Ax = b \implies LUx = Pb. \]

So we need to solve

\[ Ly = Pb, \quad Ux = y. \]

Thus, the solve part has to take \( p \) in as well, e.g.

\[
\begin{align*}
p, \text{ellu} &= \text{lu_factor}(a) \\
x &= \text{lu_fwd_and_back}(\text{ellu}, p, b)
\end{align*}
\]

Note that \((Pb)_i = b_{p(i)}\), so \(Pb\) can be accessed from knowing \(b\) and \(p\).

Now all that’s left is to put it all together and write, (i) factor, (ii) forward/backward solve and (iii) a general function

\[
\begin{align*}
x &= \text{linsolve}(a, b)
\end{align*}
\]

that a user can call to solve \(Ax = b\) without worrying about all the details.

**Remark:** This isn’t just a practice algorithm; it’s a good method for a general linear system when \(n\) is not too large. (A version is used by \texttt{matlab}'s default solver and \texttt{scipy.linalg}).
Stray notes
In discussing function returns we came across commas on the LHS of an equals:

t = (1,2)  
a, b = t  # a=1 and b=2

The ‘comma’ syntax works on the right hand side, too; it is short for ‘make a tuple with these elements’ - for instance:

\[
x = a, b  \quad \# \text{ x is now } (a, b)
\]
\[
a, b = 1, 2  \quad \#\text{ now a = 1 and b = 2}
\]

What do the following snippets do? How are they different (if at all)?

<table>
<thead>
<tr>
<th># Example 1%</th>
<th># Example 2</th>
<th># Example 3</th>
<th># Example 4</th>
<th># Example 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>a = 1</td>
<td>a = 1</td>
<td>a = [1,2]</td>
<td>a = [1,2]</td>
<td>a = [1,2]</td>
</tr>
<tr>
<td>a = b</td>
<td>a, b = b, a</td>
<td>c = a</td>
<td>a = b</td>
<td>c = a[::]</td>
</tr>
<tr>
<td>b = a</td>
<td>a, b = b, a</td>
<td>a = b</td>
<td>a = b</td>
<td>a[:] = b</td>
</tr>
<tr>
<td></td>
<td></td>
<td>b = c</td>
<td>b = c</td>
<td>b[:]= c</td>
</tr>
</tbody>
</table>
Example: operation counts

Forward substitution to solve $Lx = b$:

```python
for i in range(n):
    x[i] = b[i]
    for j in range(i):
        x[i] -= a[i][j]*x[j]  #*
        x[i] /= a[i][i]  #**
```

$$x_i = \frac{1}{a_{ii}} \left( b_i - \sum_{j=0}^{i-1} a_{ij}x_j \right), \quad i = 0, \ldots, n - 1.$$  

To count multiplications/divisions: sum # ops over each for loop:

$$\sum_{i=0}^{n-1} (\cdots)$$

$$\sum_{i=0}^{n-1} \left( 1 + \sum_{j=0}^{i-1} \cdots \right) \quad \text{one div. at (**)}$$

$$\sum_{i=0}^{n-1} \left( 1 + \sum_{j=0}^{i-1} 1 \right) \quad \text{one mult. at (*)}$$

Now calculate the sum:

$$\#\text{ops} = \sum_{i=0}^{n-1} (1 + i) = n + \frac{n(n - 1)}{2} = \frac{1}{2}n^2 + \frac{1}{2}n.$$