Math 260: Python programming in math

Solving linear systems:
LU factorization

The problem

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

$$Ax = b, \qquad A = \textbf{invertible } n \times n \text{ matrix}, \quad b \in \mathbb{R}^n$$

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

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

a = [[1,2],[3,4]] #mat[0] = 0-th row
b = [5,11]
x = linsolve(a, b) # 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

$$Ax = b, \qquad A = \begin{bmatrix} a_{11} & 0 & \dots & 0 \\ a_{21} & a_{22} & \ddots & 0 \\ \vdots & \ddots & \ddots & 0 \\ a_{n1} & \dots & a_{n,n-1} & a_{nn} \end{bmatrix}$$

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

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

- 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 $x_1, \cdots, x_{i-1}$ are already computed).

The easy cases: lower triangular

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

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

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

    return x
```

$\Longrightarrow$

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

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

```
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

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

- 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]:

$$\begin{bmatrix} b_0 \\ b_1 \\ b_2 \end{bmatrix} \underbrace{\rightarrow}_{i=0} \begin{bmatrix} x_0 \\ b_1 \\ b_2 \end{bmatrix} \underbrace{\rightarrow}_{i=1} \begin{bmatrix} x_0 \\ x_1 \\ b_2 \end{bmatrix} \underbrace{\rightarrow}_{i=2} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix}$$

```
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.

in-place:

```
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),

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

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

code structure:

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

example:

$$A = \begin{bmatrix} 4 & 1 & 2 \\ 0 & 3 & 1 \\ 0 & 0 & 2 \end{bmatrix}, \; b = \begin{bmatrix} 1 \\ 5 \\ 4 \end{bmatrix}, \; x = \begin{bmatrix} -1 \\ 1 \\ 2 \end{bmatrix}$$

LU factorization

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

$$R_i \leftarrow R_i - mR_k, \qquad m = a_{ik}/a_{kk}$$

    -Store the multiplier in the (i, k) entry of L

The result is that the reduced matrix is U, so

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

(Does this always work? No - that we'll have to fix...)

Gaussian elimination

Example: Consider the LU factorization for

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

Two columns to reduce:

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

$$L: \quad \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \Longrightarrow \begin{bmatrix} 1 & 0 & 0 \\ 3/2 & 1 & 0 \\ 3/2 & 0 & 1 \end{bmatrix} \Longrightarrow \begin{bmatrix} 1 & 0 & 0 \\ 3/2 & 1 & 0 \\ 3/2 & 1 & 1 \end{bmatrix}$$

Result: $A = LU$ where

$$L = \begin{bmatrix} 1 & 0 & 0 \\ 3/2 & 1 & 0 \\ 3/2 & 1 & 1 \end{bmatrix}, \qquad U = \begin{bmatrix} 4 & -2 & 2 \\ 0 & 9 & 15 \\ 0 & 0 & -8 \end{bmatrix}.$$

Gaussian elimination (Aside: theory)

Why does this work?

- Elementary row operations are matrices, e.g.

$$E = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ \lambda & 0 & 1, \end{bmatrix}, \qquad EA = \text{adds } \lambda R_1 \text{ to } R_3$$

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

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

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

$$M_{n-1}M_{n-2}\cdots M_1 A = U,$$

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

These matrices just have λ's in corresponding entries, e.g.

$$M = \begin{bmatrix} 1 & 0 & 0 \\ -x & 1 & 0 \\ -y & 0 & 1 \end{bmatrix} \text{ does the row ops. } \begin{pmatrix} R_2 \leftarrow R_2 - xR_1 \\ R_3 \leftarrow R_3 - yR_1 \end{pmatrix}$$

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

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

Example:

$$A:\quad \begin{bmatrix} 4 & -2 & 2 \\ 6 & 6 & 18 \\ 6 & 6 & 10 \end{bmatrix} \xRightarrow[R_3 \leftarrow R_3 - \frac{3}{2}R_2]{R_2 \leftarrow R_2 - \frac{3}{2}R_1} \begin{bmatrix} 4 & -2 & 2 \\ 0 & 9 & 15 \\ 0 & 9 & 7 \end{bmatrix} \xRightarrow{R_3 \leftarrow R_3 - R_2} \begin{bmatrix} 4 & -2 & 2 \\ 0 & 9 & 15 \\ 0 & 0 & -8 \end{bmatrix}$$

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...

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

$$M = \text{row ops.} \begin{pmatrix} R_3 \leftarrow R_2 - xR_1 \\ R_3 \leftarrow R_3 - yR_1 \end{pmatrix} \implies M^{-1} = \text{row ops.} \begin{pmatrix} R_3 \leftarrow R_2 + xR_1 \\ R_3 \leftarrow R_3 + yR_1 \end{pmatrix}$$

Finally, we claim that

$$\begin{aligned} L &= M_1^{-1} \cdots M_{n-1}^{-1} \\ &= \text{ROs to reduce } A \text{ in reverse with opposite signs} \\ &= \text{matrix of multipliers} \end{aligned}$$

where the 'multiplier' for $R_j \to R_j - \lambda R_i$ is λ. 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.

$$\begin{aligned} L &= M_1^{-1} \cdots M_{n-1}^{-1} \\ &= \text{ROs to reduce } A \text{ in reverse with opposite signs} \end{aligned}$$

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

$$\begin{aligned} R_3 &\to R_3 + R_2 \\ R_3 &\to R_3 - \frac{3}{2} R_1 \\ R_2 &\to R_2 - \frac{3}{2} R_1 \end{aligned}$$

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)

```
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
```

$\Longrightarrow$

```
...
for k in range(n-1):
    for i in range(...):
        # get multiplier, store it
        # reduce i-th row with k-th
return ell, u
```

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

We want to make the code more compact...

$\Longrightarrow$ (Exercise)

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)

```
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?

Gaussian elimination

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):

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

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 `ellu` then `ellu[1][0]` is part of L but `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:

```
#(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.

```
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 How can this be done efficiently? 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

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

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**!):

$$n^2 \text{ is } O(n^3) \text{ but } n^3 \text{ is not } O(n^2)$$

Little o

Definition

Big-O (sequences) A sequence a_n is said to be litle-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 descibes 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`)

```
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.

Computational complexity

For simplicity, we count the number of mults. (multiplies) only.

Example - matrix-vector multiplication:

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

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

- For each i, there are n multiplies

$$\texttt{\# of mults} = n \cdot n \implies n^2.$$

- If you also count additions then

$$\texttt{\# of flops} = 2n^2 + O(n).$$

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

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

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

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

$O(n^2) \sim (300 \text{ million})^2 =$ way too much computation

Example: LU factorization

Recall that to solve $Ax = b$ we needed two separate parts:
Substitution (Forward or back): As a reminder, the forward formula is

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

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.

Example: LU factorization

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** $A = LU$,

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

and the factor step only has to be done once.

Since matrix *multiplication* is n^2 mults, this is quite good!

pivoting

Pivoting: theory

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, weeneed the **pivot element** a_{kk} to be non-zero.

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

$$A = \begin{bmatrix} \color{red}{1} & 1 & 1 & 1 \\ 1 & 1 & 2 & 1 \\ 0 & 5 & 2 & 1 \\ 0 & 3 & 0 & 7 \end{bmatrix} \underbrace{\Longrightarrow}_{\text{reduce col. 1}} \begin{bmatrix} 1 & 1 & 1 & 1 \\ 0 & \color{red}{0} & 1 & 0 \\ 0 & 5 & 2 & 1 \\ 0 & 3 & 0 & 7 \end{bmatrix} \underbrace{\Longrightarrow}_{k=1} ???$$

Pivoting: theory

To fix this, we must perform another row operation: **swapping rows**.

$$\cdots \begin{bmatrix} 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 3 & 2 & 1 \\ 0 & 3 & 0 & 7 \end{bmatrix} \underbrace{\Longrightarrow}_{R_2 \leftrightarrow R_3} \begin{bmatrix} 1 & 1 & 1 & 1 \\ 0 & 3 & 2 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 3 & 0 & 7 \end{bmatrix} \underbrace{\Longrightarrow}_{\text{reduce col 2}} \begin{bmatrix} 1 & 1 & 1 & 1 \\ 0 & 3 & 2 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -2 & 6 \end{bmatrix} \cdots$$

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

$$\begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix} \text{ swaps rows 1 and 3}$$

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

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

The net effect is that PA permutes rows $1 \to 2 \to 3 \to 1$.

Pivoting: Theory

In matrix form, Gaussian elimination then has the form

$$M_{n-1}P_{n-1}\cdots M_2P_2M_1P_1A = U \tag{G}$$

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

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

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 the list

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

For example:

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

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 the list

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

For example:

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

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

$$\text{the } i\text{-th row of } PA = \ p(i)\text{-th row of } A.$$

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:

$$R_i \leftarrow R_i - \underbrace{\frac{a_{ik}}{a_{kk}}}_{\text{div. by small!}} R_k$$

- 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 $a_{k+1,k}, \cdots a_{n,k}$ (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.

Pivoting: example

Here's an example: For $k = 1$ (first column), with actual swaps:

$$\begin{bmatrix} 1 & 2 & 4 \\ 1 & 0 & 1 \\ \boxed{\text{-2}} & 2 & 4 \end{bmatrix} \underbrace{\Longrightarrow}_{R_1 \leftrightarrow R_3} \begin{bmatrix} \boxed{\text{-2}} & 2 & 4 \\ 1 & 0 & 1 \\ 1 & 2 & 4 \end{bmatrix} \Longrightarrow \begin{bmatrix} -2 & 2 & 4 \\ \color{red}{-1/2} & 1 & 3 \\ \color{red}{-1/2} & 3 & 6 \end{bmatrix}, \quad p = \{3, 2, 1\}$$

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:

$$\begin{bmatrix} -2 & 2 & 4 \\ \color{red}{-1/2} & 1 & 3 \\ \color{red}{-1/2} & \boxed{3} & 6 \end{bmatrix} \underbrace{\Longrightarrow}_{R_2 \leftrightarrow R_3} \begin{bmatrix} -2 & 2 & 4 \\ \color{red}{-1/2} & \boxed{3} & 6 \\ \color{red}{-1/2} & 1 & 3 \end{bmatrix} \Longrightarrow \begin{bmatrix} -2 & 2 & 4 \\ \color{red}{-1/2} & 3 & 6 \\ \color{red}{-1/2} & \color{red}{1/3} & 1 \end{bmatrix}$$

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

$$Ly = Pb, \qquad Ux = y.$$

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

```
p, ellu = lu_factor(a)
x = lu_fwd_and_back(ellu, p, b)
```

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

```
x = linsolve(a, b)
```

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 `matlab`'s default solver and `scipy.linalg`).

Stray notes

Aside: commas and swaps

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 # x is now (a, b)
a, b = 1, 2 #now a = 1 and b = 2
```

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

```
# Example 1%
a = 1
b = 2

a = b
b = a
```

```
# Example 2
a = 1
b = 2

a, b = b, a
```

```
# Example 3
a = [1,2]
b = [3,4]

c = a
a = b
b = c
```

```
# Example 4
a = [1,2]
b = [3,4]

a, b = b, a
```

```
# Example 5
a = [1,2]
b = [3,4]

c = a[:]
a[:] = b
b[:] = c
```

Example: operation counts

Forward substitution to solve $Lx = b$:

```
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]  #**
```

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

$$i = 0, \cdots n-1.$$

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

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

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

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

Now calculate the sum:

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