```
import numpy as np
import scipy.linalg
```

Week 3 Coding Lecture 2: LU Factorization

We have two different methods of solving systems of equations: Forward/back substitution and Gaussian elimination. We just saw that, at least for large systems, forward/back substitution is vastly faster than Gaussian elimination. We would therefore prefer to use forward/back substitution for all of our problems. Unfortunately, forward/back substitution only work in special cases. If our system isn't lower/upper triangular, then we can't use this faster method. We have already seen several examples of non-triangular systems, so we know that we can't hope that all systems will be triangular in general. However, it is possible that we could write all systems in some simple form so that we didn't have to use the full Gaussian elimination method. In particular, suppose that we could always rewrite a system $A\mathbf{x} = \mathbf{b}$ in the form

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

where L is an $N \times N$ lower triangular matrix and U is an $N \times N$ upper triangular matrix. If this were true, it would be relatively easy to solve the system. To see how, note that $U\mathbf{x}$ is an (unknown) $N \times 1$ vector (because it is the product of an $N \times N$ matrix U and an $N \times 1$ vector $\mathbf{x}$). If we give this vector a new name $\mathbf{y}$, then we have

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

where

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

Notice that the equation $L\mathbf{y} = \mathbf{b}$ is easy to solve! L is a lower triangular matrix and $\mathbf{b}$ is a known vector, so we can just use forward substitution, which takes $\mathcal{O}(N^2)$ flops. Once we do this, we know the vector $\mathbf{y}$, which means that we can also solve $U\mathbf{x} = \mathbf{y}$. Again, this is easy to solve! Since U is upper triangular, we can just use back substitution, which also takes $\mathcal{O}(N^2)$ flops. We can therefore solve the original system in two $\mathcal{O}(N^2)$ steps. Since big-oh notation ignores constant multiples, this is essentially the same as $\mathcal{O}(N^2)$. This means that if we are given a system in the form $LU\mathbf{x} = \mathbf{b}$, we can just use substitution twice instead of Gaussian elimination and therefore solve our system much faster.

Of course, it is unlikely that someone will simply hand you a system in this convenient form, so we need to find a method that calculates L and U from A. Through a somewhat lucky coincidence, it turns out that (almost) every matrix A can be written in this way, and that we can find L and U through Gaussian elimination. We will go through an example by hand and then turn to python.

Remember our 3×3 system from earlier in the week:

$$A\mathbf{x} = \begin{pmatrix} 2 & 1 & 1 \\ 4 & 3 & 3 \\ 8 & 7 & 9 \end{pmatrix} \mathbf{x} = \begin{pmatrix} 1 \\ 1 \\ -1 \end{pmatrix} = \mathbf{b}.$$

After performing the row operations

$$-2 \cdot R_1 + R_2 \to R_2,$$

$$-4 \cdot R_1 + R_3 \to R_3,$$

$$-3 \cdot R_2 + R_3 \to R_3,$$

we obtained the new system

$$\begin{pmatrix} 2 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 2 \end{pmatrix} \mathbf{x} = \begin{pmatrix} 1 \\ -1 \\ -2 \end{pmatrix}.$$

This new system is upper triangular, and we will use the resulting matrix as U. That is,

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

The matrix L is somewhat more complicated, but we can create it by looking at the row operations we employed. L is always of the form

$$L = \begin{pmatrix} 1 & 0 & 0 \\ \ell_{21} & 1 & 0 \\ \ell_{31} & \ell_{32} & 1 \end{pmatrix},$$

where the entries ℓ_{ij} are numbers that we have to determine. It turns out that these entries are just the coefficients we used in our row operations with the signs reversed. For instance, we used the row operation $-2 \cdot R_1 + R_2 \to R_2$ to zero out the 2nd row, 1st column of A, so the entry $\ell_{21} = 2$ (note that the sign has flipped). Likewise, we used the row operation $-4 \cdot R_1 + R_3 \to R_3$ to change the 3rd row, 1st column of A, so $\ell_{31} = 4$. Finally, we used the row operation $-3 \cdot R_2 + R_3 \to R_3$ to change the 3rd row, 2nd column of A, so $\ell_{32} = 3$. This means that

$L = \begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 4 & 3 & 1 \end{pmatrix}$. We can use python to check that $LU = A$:

```
A = np.array([[2, 1, 1], [4, 3, 3], [8, 7, 9]])
b = np.array([[1], [1], [-1]])

L = np.array([[1, 0, 0], [2, 1, 0], [4, 3, 1]])
U = np.array([[2, 1, 1], [0, 1, 1], [0, 0, 2]])

print(A)
```

```
[[2 1 1]
 [4 3 3]
 [8 7 9]]
```

```
print(L @ U)
```

```
[[2 1 1]
 [4 3 3]
 [8 7 9]]
```

The process of finding L and U is called LU decomposition.

Once we have L and U, we can solve the original system with two steps of forward/back substitution. We first solve the equation $L\mathbf{y} = \mathbf{b}$ with forward substitution:

```
y = scipy.linalg.solve_triangular(L, b, lower=True)
```

```
print(y)
```

```
[[ 1.]
 [-1.]
 [-2.]]
```

Then we use that result to solve $U\mathbf{x} = \mathbf{y}$ with back substitution:

```
x = scipy.linalg.solve_triangular(U, y)
print(x)
```

```
[[ 1.]
 [ 0.]
 [-1.]]
```

This is the same solution we found with Gaussian elimination originally.

```
print(scipy.linalg.solve(A, b))
```

```
[[ 1.]
 [-0.]
 [-1.]]
```

Permutation matrices

We said above that almost every matrix could be written in the form $A = LU$. The "almost" is important, and it is related to the fact that Gaussian elimination does not always work. We established earlier in the week that Gaussian elimination could fail if there were a zero on the main diagonal of your matrix so that you couldn't continue eliminating coefficients. We also established that you could always solve this issue by reordering your equations.

Python expresses "reordering your equations" through something called a *permutation matrix*. A permutation matrix is just the identity matrix with some of the rows reordered. (Remember, the identity matrix is a square matrix with 1's on the diagonal and 0's everywhere else.) For instance,

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

is a permutation matrix because it is the 3×3 identity matrix with the last row moved to the top.

If you multiply a permutation matrix by another matrix or vector, it just reorders the rows of the matrix/vector. For instance,

```
print(A)
```

```
[[2 1 1]
 [4 3 3]
 [8 7 9]]
```

```
P = np.array([[0, 0, 1], [1, 0, 0], [0, 1, 0]])
print(P @ A)
```

```
[[8 7 9]
 [2 1 1]
 [4 3 3]]
```

That is, PA is just A with the last row moved on to the top.

If you have a system of equations $A\mathbf{x} = \mathbf{b}$ and you want to reorder the equations, you need to multiply *both sides* of the equation by P. So, for example, if we have the same A and b as before:

```
print(A)
```

```
[[2 1 1]
 [4 3 3]
 [8 7 9]]
```

```
print(b)
```

```
[[ 1]
```

```
       [ 1]
       [-1]]
```

then you could reorder the system by changing them to PA and $P\mathbf{b}$:

```
print(P @ A)
```

```
     [[8 7 9]
      [2 1 1]
      [4 3 3]]
```

```
print(P @ b)
```

```
     [[-1]
      [ 1]
      [ 1]]
```

A matrix A can't always be written as $A = LU$, but if you reorder the rows of A first, then you can always write it in this form. In mathematical notation, this means that there is always a permutation matrix P, a lower triangular matrix L and an upper triangular matrix U such that

$$PA = LU.$$

We already saw how to compute L and U by hand. We won't worry about how to find P by hand, because it is somewhat more complicated and python will do it for us.

You can calculate these three matrices in python with the function `lu` from the `scipy.linalg` package. The general syntax is `P, L, U = scipy.linalg.lu(A)`. The latter two return values (L and U) will be the lower and upper triangular matrices that we want. Unfortunately, python formats the matrix P somewhat differently than we are looking for. As an example:

```
P, L, U = scipy.linalg.lu(A)
print(L)
```

```
     [[1.         0.         0.        ]
      [0.25       1.         0.        ]
```

```
       [0.5        0.66666667 1.        ]]

print(U)

     [[ 8.          7.          9.        ]
      [ 0.         -0.75       -1.25      ]
      [ 0.          0.         -0.66666667]]


print(P)

     [[0. 1. 0.]
      [0. 0. 1.]
      [1. 0. 0.]]
```

The authors of scipy decided to format these answers so that $A = PLU$ instead of $PA = LU$. We can confirm this by trying:

```
print("PA = ")
print(P @ A)
print("LU = ")
print(L @ U)

     PA =
     [[4. 3. 3.]
      [8. 7. 9.]
      [2. 1. 1.]]
     LU =
     [[8. 7. 9.]
      [2. 1. 1.]
      [4. 3. 3.]]


print("A = ")
print(A)
print("PLU = ")
print(P @ L @ U)

     A =
     [[2 1 1]
      [4 3 3]
```

```
 [8 7 9]]
PLU =
[[2. 1. 1.]
 [4. 3. 3.]
 [8. 7. 9.]]
```

This is easy to fix. The matrix that we called P is actually the transpose of the matrix that scipy calls P, so we should really use the code

```
P, L, U = scipy.linalg.lu(A)
P = P.T
```

(Strictly speaking, we don't have to do this, but it will make the results more consistent with our mathematical presentation. If you later decide to switch programming languages, or even to use a different python package to solve systems, you should check what format the output is in. There are many different ways to express these concepts in code, and no two LU decomposition functions are exactly the same.)

It's worth noting that python still didn't find the same L and U that we did. That is because we didn't reorder the rows of A, but python did. (You can tell by looking at our new P - it is not just the identity matrix.) We can, however, confirm that $PA = LU$ as desired.

```
print("PA = ")
print(P @ A)
print("LU = ")
print(L @ U)
```

```
PA =
[[8. 7. 9.]
 [2. 1. 1.]
 [4. 3. 3.]]
LU =
[[8. 7. 9.]
 [2. 1. 1.]
 [4. 3. 3.]]
```

Once you have these matrices, it is straightforward to solve for $\mathbf{x}$. We know that

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

so we can multiply both sides by P to reorder the equations:

$PA\mathbf{x} = P\mathbf{b}$.

We know that $PA = LU$ (where this P is the transpose of the matrix we got from `scipy.linalg.lu`), so we can rewrite this as

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

If we rename $U\mathbf{x} = \mathbf{y}$, then we have

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

This is a lower triangular system, so we can solve it with forward substitution to find $\mathbf{y}$. Once we have $\mathbf{y}$, we can use back substitution to solve $U\mathbf{x} = \mathbf{y}$, which gives us our final answer.

In python, the process looks like this:

```
P, L, U = scipy.linalg.lu(A)
P = P.T
y = scipy.linalg.solve_triangular(L, P @ b, lower=True)
x = scipy.linalg.solve_triangular(U, y)
print(x)
```

```
[[ 1.]
 [-0.]
 [-1.]]
```

Speed of LU decomposition

The `lu` function uses essentially the same algorithm as Gaussian elimination, so we know that it takes $\mathcal{O}(N^3)$ flops. We then have to use forward substitution to solve $L\mathbf{y} = P\mathbf{b}$, which takes $\mathcal{O}(N^2)$ flops, and then we have to use back substitution to solve $U\mathbf{x} = \mathbf{y}$, which takes another $\mathcal{O}(N^2)$ flops. The whole process therefore takes $\mathcal{O}(N^3) + \mathcal{O}(N^2) + \mathcal{O}(N^2)$ flops, but since we only care about the largest power this means that it takes $\mathcal{O}(N^3)$ flops.

This is essentially the same speed as Gaussian elimination. (Which should make sense, since it's the same process, plus one more forward substitution step.) It therefore looks like we haven't actually made any improvements. The key thing to notice, though, is that the LU decomposition step (i.e., finding the matrices P, L and U) only depends on A and not on $\mathbf{b}$. This means that if we have to solve two systems with the same left hand side, we only have to use the `lu` function once. For example, we can solve the system

$$A\mathbf{x} = \begin{pmatrix} 2 & 1 & 1 \\ 4 & 3 & 3 \\ 8 & 7 & 9 \end{pmatrix} \mathbf{x} = \begin{pmatrix} 4 \\ 10 \\ 24 \end{pmatrix} = \mathbf{c}$$

with the code

```
c = np.array([[4], [10], [24]])
y = scipy.linalg.solve_triangular(L, P @ c, lower=True)
x = scipy.linalg.solve_triangular(U, y)
print(x)
```

```
[[1.]
 [1.]
 [1.]]
```

Since we already have P, L and U, we don't have to use the `lu` function (which takes $\mathcal{O}(N^3)$ flops); we only have to use forward and back substitution (which both take $\mathcal{O}(N^2)$ flops).

It turns out that this is an extremely common situation. Very often, the matrix A describes the permanent structure of a problem, while the right hand side of the system describes some temporary features. As an example, the left hand side might represent the location and orientation of different girders in a bridge, while the right hand side represents the loads from vehicles on the bridge. If we want to see how the bridge reacts to different traffic patterns, we will need to repeatedly solve linear systems with the same left hand side, but with different right hand sides. In such a situation, we can use the `lu` function once, and then solve all the other problems much more quickly.

Inverses

There is one more solution method that you may see in textbooks or other classes. If you want to solve the system $A\mathbf{x} = \mathbf{b}$, then one possible approach is to multiply both sides of the equation by some matrix that will cancel out the A. Such a matrix is called the *inverse* of A and denoted by A^{-1}. You would then solve the system by writing:

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

$A^{-1}A\mathbf{x} = A^{-1}\mathbf{b}$, and so

$\mathbf{x} = A^{-1}\mathbf{b}$.

We will essentially never compute an inverse matrix in this class, but python does have a function for it in the `scipy.linalg` package called `inv`. This means that you could solve the system by writing

```
x = scipy.linalg.inv(A) @ b
print(x)
```

```
[[ 1.00000000e+00]
 [-1.11022302e-16]
 [-1.00000000e+00]]
```

You should not use this code. The `inv` function is both slower and more prone to rounding error than Gaussian elimination. (This method is still technically $\mathcal{O}(N^3)$, but it is worse than Gaussian elimination on every front.)

We will frequently use the notation $A^{-1}\mathbf{b}$ in this class, but you should always mentally translate that into "the solution of the equation $A\mathbf{x} = \mathbf{b}$". Mathematically, they are the same thing, but in code you should **never** use inverses to solve a system.

Summary of system solvers

We now know several different ways to solve a system of equations $A\mathbf{x} = \mathbf{b}$.

1) If the system is lower/upper triangular, you can use forward/back substitution. The code for this in python is `x = scipy.linalg.solve_triangular(A, b, lower=True)` for forward substitution and `x = scipy.linalg.solve_triangular(A, b)` for back substitution. This process is $\mathcal{O}(N^2)$.

2) If you have to solve multiple systems with the same A, but different right hand sides, you can use LU decomposition. The first system will take $\mathcal{O}(N^3)$ flops, but subsequent systems will only take $\mathcal{O}(N^2)$ flops. You can find the LU decomposition with the `scipy.linalg.lu` function.

3) You can always fall back on Gaussian elimination. The code for this in python is `x = scipy.linalg.solve(A, b)`. This process is $\mathcal{O}(N^3)$.

4) You should not use matrix inverses.