```
import numpy as np
```

Week 3 Coding Lecture 1: Linear Systems

Matrix Multiplication

Remember from the first week of class that we have defined two different ways to multiply matrices. As an example, suppose that we have the two matrices

$$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \text{ and } B = \begin{pmatrix} -2 & 1 \\ 0 & 3 \end{pmatrix}.$$

In python we would write this as:

```
A = np.array([[1, 2], [3, 4]])
print(A)
```

```
[[1 2]
 [3 4]]
```

```
B = np.array([[-2, 1], [0, 3]])
print(B)
```

```
[[-2  1]
 [ 0  3]]
```

We have defined the elementwise multiplication of A and B as

$$C = A.*B = \begin{pmatrix} (1)(-2) & (2)(1) \\ (3)(0) & (4)(3) \end{pmatrix} = \begin{pmatrix} -2 & 2 \\ 0 & 12 \end{pmatrix},$$

or in python:

```
C = A * B
print(C)
```

```
[[-2  2]
 [ 0 12]]
```

More generally, as long as A and B are exactly the same shape, we can define $C \equiv A.*B$ by $c_{ij} = a_{ij}b_{ij}$. That is, the entry in the ith row, jth column of C is the product of the entry in the ith row, jth column of A and the entry in the ith row, jth column of B.

We also defined normal (linear algebra) multiplication of A and B as

$$D = AB = \begin{pmatrix} (1)(-2)+(2)(0) & (1)(1)+(2)(3) \\ (3)(-2)+(4)(0) & (3)(1)+(4)(3) \end{pmatrix} = \begin{pmatrix} -2 & 7 \\ -6 & 15 \end{pmatrix},$$

or in python:

```
D = A @ B
print(D)
```

```
[[-2  7]
 [-6 15]]
```

More generally, as long as the number of columns of A is the same as the number of rows of B, we define $D = AB$ as the matrix where d_{ij} (the entry in the ith row, jth column) is the dot product of the ith row of A with the jth column of B. If n is the number of columns of A and the number of rows of B, then we can write this as $d_{ij} = \sum_{k=1}^{n} a_{ik}b_{kj}$. You should confirm to yourself that this is equivalent to the dot product definition.

If you are not familiar with linear algebra, then this second definition probably seems a little strange. As we will see in a moment, the big advantage of this definition is that it makes it easy to write linear systems of equations.

Systems of equations

One of the most important skills we will cover in this class is how to efficiently solve systems of linear equations. It turns out that such systems arise when solving a wide variety of problems in applied mathematics, even if the original problem does not appear linear. What do we mean by a system of linear equations? You should already be familiar with the concept from a basic algebra class. For instance:

$$\begin{cases} 2x + y & = & 5, \\ -6x + y & = & -3, \end{cases}$$

or

$$\begin{cases} 2x_1 + x_2 + x_3 & = & 1, \\ 4x_1 + 3x_2 + 3x_3 & = & 1, \\ 8x_1 + 7x_2 + 9x_3 & = & -1. \end{cases}$$

(If we only have two variables, we will often call them x and y. If there are three variables, we will sometimes call them x, y and z. For anything more complicated, we will use a single letter with different subscripts like in the second example.) For starters, let's focus on the first pair of equations. We have two equations and two variables, so we refer to this as a 2×2 (read: two by two) system. The first number indicates the number of equations and the second number indicates the number of variables. You have probably already seen several ways to solve this system.

Substitution

One approach would be to solve the first equation for y, then substitute this into the second equation and solve for x. Solving the first equation for y gives us $y = 5 - 2x$. Substituting that into the second equation gives

$-6x + (5 - 2x) = -3$, so $-8x = -8$ and therefore $x = 1$.

Plugging this back into $y = 5 - 2x$ gives us $y = 3$. The final solution to the system is therefore $x = 1$, $y = 3$. This method is called *substitution*. It is an easy approach when we only have a couple of equations, but it rapidly becomes messy when we try to generalize to larger systems. We will therefore only use substitution in certain special cases.

Elimination

You have probably seen another approach, where we try to cancel out one of the variables from one of the equations by adding/subtracting different equations. For example, if we multiplied the first equation by 3 and then added it to the second equation, we would cancel out the x's. We then obtain

$$\begin{cases} 2x + y &= &5, \\ 0x + 4y &= &12. \end{cases}$$

We denote this step by:

$$3 \cdot R_1 + R_2 \to R_2.$$

You should read this as "multiply row 1 by 3 and add it to row 2, then change the second row to this result."

This process is called *elimination*. Notice that we have not solved the system yet, but we did make it much easier to solve by substitution. The second equation is now very easy to solve for y, and we find that $y = 3$. Now that we have this value, we can substitute it into the first equation to find that $x = 1$. The system we obtained from elimination is called *upper triangular* because the nonzero coefficents form a triangle in the upper right (and the zero coefficients are all in the lower left). When we use the substitution method on an upper triangular system, it is called *back substitution* (because we find the variables in backwards order).

Note that it would be just as easy to make the coefficient of y in the first equation zero. This would give us a system where all the nonzero coefficients are in the lower left corner and all the zero coefficients are in the upper right. Such a system is called *lower triangular* and we could solve it in the opposite order with *forward substitution*. Either method would work equally well. In general, we call these systems *triangular*.

This method generalizes much better when we start solving larger systems. For example, to solve the $3\times 3# system given at the beginning of this section, we want to eliminate (i.e., turn to zero) all the coefficients in the lower left. That is, we want to make a system of the form

$$\begin{cases} ax_1 + bx_2 + cx_3 &= &d, \\ \mathbf{0x_1} + ex_2 + fx_3 &= &g, \\ \mathbf{0x_1} + \mathbf{0x_2} + hx_3 &= &i. \end{cases}$$

Once we do this, it will be easy to solve the resulting system with back substitution. There are several approaches to this problem, but if you are not systematic then it is easy to accidentally undo your work. We will therefore always follow a standard order. This is easier to see with an example.

First, we will eliminate the x_1 coefficient from the second equation. We can do this by multiplying the first equation by -2 and adding it to the second equation. This gives us the new system

$$\begin{cases} 2x_1 + x_2 + x_3 & = & 1, \\ 0x_1 + x_2 + x_3 & = & -1, \\ 8x_1 + 7x_2 + 9x_3 & = & -1. \end{cases}$$

This step can be written as

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

Next, we will eliminate the x_1 coefficient from the third equation. We can do this by multiplying the first equation by -4 and adding it to the third equation. We obtain

$$\begin{cases} 2x_1 + x_2 + x_3 & = & 1, \\ 0x_1 + x_2 + x_3 & = & -1, \\ 0x_1 + 3x_2 + 5x_3 & = & -5. \end{cases}$$

This step can be written as

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

Finally, we eliminate the x_2 coefficient from the third equation. We can do this by multiplying the second equation by -3 and adding it to the third equation. (Note that it would not be a good idea to multiply the first equation by -3 and add it to the third equation, because that would undo the work that we did in the last step.) We get

$$\begin{cases} 2x_1 + x_2 + x_3 & = & 1, \\ 0x_1 + x_2 + x_3 & = & -1, \\ 0x_1 + 0x_2 + 2x_3 & = & -2. \end{cases}$$

This step can be written as

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

Now our system is in upper triangular form, so we can use back substitution to solve it. The third equation is easy to solve and gives us $x_3 = -1$. Plugging this into the second equation, we find that $x_2 = 0$. Plugging each of these into the first equation, we find that $x_1 = 1$.

The order in which we eliminated the coefficients is somewhat arbitrary, but it is important to be consistent so that we can turn this into code. (In addition, this particular order will make our life easier in the next section.) When we use this order (i.e., eliminate all of the x_1 coefficients, then all of the x_2 coefficients, etc.) the method is called *Gaussian elimination*. Hopefully you can see how this generalizes to larger systems. If you have trouble visualizing it, I encourage you to try experimenting with a 4×4 system until it becomes clear. That said, the amount of arithmetic and writing rapidly becomes cumbersome as the number of equations and variables increases, so I will never ask you to solve anything larger than a 3×3 by hand. Fortunately for us, lots of arithmetic in a prescribed order is exactly what computers are good at.

There is one problem with our method. To see what can go wrong, consider the system

$$\begin{cases} 0x + 4y &= 12, \\ 2x + y &= 5. \end{cases}$$

This is the same as one of the systems we solved above with back substitution except that the equations are in a different order, so we know that there is a solution. In fact, we know that the solution is $x = 1$ and $y = 3$. However, the system is not upper triangular, and we cannot apply our method because we cannot eliminate the coefficient for x in the second equation without dividing by zero. Of course, we can just reorder the equations to solve this issue.

Matrix Notation

Python has many predefined methods to solve systems of equations, but first we need to input our equations in a way that python will understand. To do so, we will rewrite our systems as matrix equations. We do this by putting all of the coefficients from our system in a matrix, putting all of the variables into a vector and putting all of the right-hand side values into another vector. For the first sytem, we have

$$A = \begin{pmatrix} 2 & 1 \\ -6 & 1 \end{pmatrix}, \mathbf{x} = \begin{pmatrix} x \\ y \end{pmatrix} \text{ and } \mathbf{b} = \begin{pmatrix} 5 \\ -3 \end{pmatrix}.$$

Now it will become apparent why we defined matrix multiplication the way we did. Since A is a 2×2 matrix and $\mathbf{x}$ is a 2×1 vector, it makes sense to multiply the two together. (That is, the number of columns of A is the same as the number of rows of $\mathbf{x}$. This is as it should be, because A has one column for every variable in our equations and $\mathbf{x}$ has one row for every variable in our equations.) When we multiply these two, we should get a 2×1 vector (the number of rows of A by the number of columns of $\mathbf{x}$). Using the dot product definition, we get

$$A\mathbf{x} = \begin{pmatrix} 2 & 1 \\ -6 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 2x + y \\ -6x + y \end{pmatrix}.$$

But we know from the original equations that $2x + y$ is 5 and that $-6x + y$ is -3, so we have

$$A\mathbf{x} = \begin{pmatrix} 2x + y \\ -6x + y \end{pmatrix} = \begin{pmatrix} 5 \\ -3 \end{pmatrix} = \mathbf{b}.$$

That is, we can rewrite our whole system as the single matrix equation $A\mathbf{x} = \mathbf{b}$.

Similarly, we can rewrite the 3×3 system from above as the matrix equation $A\mathbf{x} = \mathbf{b}$, where

$$A = \begin{pmatrix} 2 & 1 & 1 \\ 4 & 3 & 3 \end{pmatrix}, \mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \text{and } \mathbf{b} = \begin{pmatrix} 1 \\ 1 \end{pmatrix}.$$

Solving systems in python

Python has several useful predefined functions for solving systems of equations. All of these functions are designed to work with the matrix notation that we described in the previous section. To enter a system of equations into python, we therefore need to enter the matrix A and the vector $\mathbf{b}$ as 2D numpy arrays. For example, we could define the 2×2 system from above by writing

```
A = np.array([[2, 1], [-6, 1]])
b = np.array([[5], [-3]])
```

Notice that we made `b` a column vector, not a 1D array. Also notice that we did not explicitly write out the vector `[[x], [y]]`. We will never tell python the names of our unknown variables, but all of the builtin solvers assume that we are solving for an unknown vector such that A times that vector equals $\mathbf{b}$.

The easiest way to solve a linear system in python is with the command `solve`. This command is located in a package called `linalg`, which is itself located in a package called `scipy`. (The latter stands for "scientific python".) We will have to import this package in order to use the solve command.

```
import scipy.linalg
```

From now on, we can call functions from the linalg package with `scipy.linalg.<name of function>`. In particular, we can use the `solve` function by writing `scipy.linalg.solve()`. The function takes two arguments: The matrix A and the right hand side vector $\mathbf{b}$.

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

```
    [[1.]
     [3.]]
```

Note that we never explicitly told python the names of our variables, but the order is implicitly represented by our equations. In particular, the coefficients in A are always in order. The coefficients in the first column of A correspond to the first variable, and the coefficients in the second column of A correspond to the second variable. Since we saved our solution in a vector named `x`, you can refer to these solutions in python as `x[0]` and `x[1]`, but it is up to you to remember that `x[0]` corresponds to the variable x and `x[1]` corresponds to the variable y.

Similarly, we can enter the 3×3 example with

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

We can then solve this system with the command

```
x = scipy.linalg.solve(A, b)
```

```
print(x)
```

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

The `solve` function uses the Gaussian elimination method we just outlined above. (Well, it uses a closely related method called LU decomposition with pivoting, which we will explain in a couple lectures, but it is safe to assume that this is just Gaussian elimination with some of the equations re-ordered.) In particular, no matter what numbers are in the matrix `A`, python will always go through all of the elimination steps to make the system upper triangular and then use back substitution to solve the resulting triangular system.

This method always works, but it is not always ideal. In particular, if the matrix `A` is already triangular, then it is a waste of time to go through all of the elimination steps and we should just jump straight to back/forward substitution. Fortunately, there is another function in the linalg module that does exactly that: `scipy.linalg.solve_triangular`. As an example, we can solve the system

$$\begin{cases} 2x_1 + x_2 + x_3 & = & 1, \\ 0x_1 + x_2 + x_3 & = & -1, \\ 0x_1 + 0x_2 + 2x_3 & = & -2. \end{cases}$$

with the code

```
A = np.array([[2, 1, 1], [0, 1, 1], [0, 0, 2]])
b = np.array([[1], [-1], [-2]])
x = scipy.linalg.solve_triangular(A, b)
print(x)
```

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

The `solve_triangular` function assumes that `A` is upper triangular, but you can also solve a lower triangular system with the syntax `scipy.linalg.solve_triangular(A, b, lower=True)`. For example, we can solve the system

$$\begin{cases} 2x_1 + 0x_2 + 0x_3 & = & 2, \\ x_1 + x_2 + 0x_3 & = & 2, \end{cases}$$

```
A = np.array([[2, 0, 0], [1, 1, 0], [2, 1, 1]])
b = np.array([[2], [2], [2]])
x = scipy.linalg.solve_triangular(A, b, lower=True)
print(x)
```

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

It's important to remember that the `solve_triangular` function **does not check** if the system is actually triangular; it just ignores any entries that are supposed to be zero. If you accidentally use `solve_triangular` on a non-triangular system, python will not give any error messages or warnings but you will get the wrong answer.

Existence and Uniqueness

As you already know from high school algebra, not all systems of equations have a solution, and if they do have a solution it is not necessarily unique. For example, the following system of equations does not have any solutions:

$$\begin{cases} 3x_1 + 2x_2 & = & 5, \\ 3x_1 + 2x_2 & = & 4, \end{cases}$$

because $3x_1 + 2x_2$ cannot be both 5 and 4 simultaneously. However, the system

$$\begin{cases} 3x_1 + 2x_2 & = & 5, \\ 3x_1 + 2x_2 & = & 5, \end{cases}$$

has infinitely many solutions of the form $x_1 = (5 - 2z)/3$ and $x_2 = z$. We won't prove it in this class, but one can show that these are the only three possibilities for any system of equations: Either the system has exactly one solution, or it has no solutions, or it has infinitely many solutions. Moreover, one can determine whether or not a system has exactly one solution just by looking at the left hand side (i.e., the matrix A), not the right hand side (i.e., the vector $\mathbf{b}$).

We will primarily be interested in solving systems that have a unique solution. If the system has infinitely many solutions, then you need more context to decide which solution is best. Similarly, if there are no solutions then you might be willing to accept something that is "close to" a solution, but you would need more context to decide what "close to" means. We will discuss the latter situation when we talk about data fitting, but for now we will only worry about solving systems that have a unique solution.

The `solve` function (as well as `solve_triangular`) checks if the system has one solution and will raise an error if there are infinitely many or no solutions. For example,

```
A = np.array([[3, 2], [3, 2]])
b = np.array([[5], [4]])
x = scipy.linalg.solve(A, b)
```

```
---------------------------------------------------------------------------
LinAlgError                               Traceback (most recent call last)
<ipython-input-41-dbeeebd589c4> in <module>
      1 A = np.array([[3, 2], [3, 2]])
      2 b = np.array([[5], [4]])
----> 3 x = scipy.linalg.solve(A, b)

~/anaconda3/lib/python3.8/site-packages/scipy/linalg/basic.py in solve(a, b, sym_pos, lower, overwrite_a,
overwrite_b, debug, check_finite, assume_a, transposed)
    212                                                        (a1, b1))
    213         lu, ipvt, info = getrf(a1, overwrite_a=overwrite_a)
--> 214         _solve_check(n, info)
    215         x, info = getrs(lu, ipvt, b1,
    216                         trans=trans, overwrite_b=overwrite_b)

~/anaconda3/lib/python3.8/site-packages/scipy/linalg/basic.py in _solve_check(n, info, lamch, rcond)
     27                          '.'.format(-info))
     28     elif 0 < info:
---> 29         raise LinAlgError('Matrix is singular.')
     30 
     31     if lamch is None:

LinAlgError: Matrix is singular.
```

SEARCH STACK OVERFLOW

The phrase "Matrix is singular" means that there is not a unique solution to the equation $A\mathbf{x} = \mathbf{b}$. However, it is good to know how to check this yourself. There are two very useful functions for this purpose.

1) In the numpy package, there is a subpackage called `linalg` (this is different than the scipy.linalg package), and this subpackage has a function called `cond`. The `cond` function calculates the *condition number* of a matrix. We won't worry about the technical definition of the condition number, but the relevant feature for us is that the system $A\mathbf{x} = \mathbf{b}$ has a unique solution if and only if the condition number of A is finite. If the condition number of A is infinite, then the system either has no solutions or infinitely many solutions. Calculating the condition number perfectly is not always possible, so `cond` only gives an approximation. In particular, it might not give actual infinity even if the system does not have a solution. If A has a very large condition number then you should assume that there is no unique solution. (We won't worry about exactly what "very large" means. You can safely assume that condition numbers larger than 10^{10} are very large.) For example,

```
A = np.array([[3, 2], [3, 2]])
print(np.linalg.cond(A))
```

```
    3.5393555045047484e+16
```

```
B = np.array([[2, 1], [-6, 1]])
print(np.linalg.cond(B))
```

```
    5.0520609798684495
```

2) The scipy.linalg package has a function called `det` that calculates the *determinant* of a matrix. If you ever take a more theoretical linear algebra class, you will spend a lot of time discussing the determinant, so it might come as a surprise to find that we will barely mention it in this class. It turns out that the determinant is very useful theoretically, but very inefficient to calculate. It does have the following useful property: A square system (that is, when you have the same number of equations as variables) has exactly one solution if and only if the determinant of A is non-zero. Unfortunately, the determinant is not defined for non-square matrices. For example,

```
print(scipy.linalg.det(A))
```

```
    3.3306690738754696e-16
```

```
print(scipy.linalg.det(B))
```

```
    8.0
```

When using python to solve systems of equations, you don't usually need to check the condition number or determinant (because `solve` will do so for you), but many other languages do not have this feature. In particular, if you use MATLAB to solve systems of equations, you should always check the condition number of A beforehand. If the condition number is too large, then the correct answer is not to go ahead with one of the system solvers; the correct answer is to say "no."