COMS 3251 Spring 2021: Lab 4

YOUR NAME(s): Vicente Farias

YOUR UNI(s): vf2272

Numerical Accuracy

In this lab you will explore some of the considerations for implementing Gaussian elimination in practice. While the theory we covered in lecture is provably correct, in reality we have to worry about Gaussian elimination on large matrices on machines with finite precision. We now need to worry about computational efficiency and accuracy.

In [ ]:
```
import numpy as np

def GE(A):
  A = A.astype(float)
  pr = 0
  pc = 0

  while pr < A.shape[0] and pc < A.shape[1]:  # run until either last row or last colu
    for i in range(pr, A.shape[0]):           # search downward for nonzero pivot
      if A[i, pc] != 0: break

    if A[i, pc] == 0:                         # no pivot in this column, move to next
      pc += 1

    else:
      A[[pr, i]] = A[[i, pr]]                          # swap rows with the found pivot
      A[pr, :] /= A[pr, pc]                            # rescale row to make leading er
      A[pr+1:, :] -= np.outer(A[pr+1:, pc], A[pr, :]) # row-reduce all rows below pivo
      A[0:pr, :] -= np.outer(A[0:pr, pc], A[pr, :])   # row-reduce all rows above pivo
      pr += 1                                          # move to next pivot position
      pc += 1

  return A
```

Example: First Try

Let's consider the following linear system:

$$10^{-20}x_1 + x_2 = 1 \quad (1)$$
$$x_1 - x_2 = 0 \quad (2)$$

The precise solution is $\mathbf{x} = \left(\frac{1}{1+10^{-20}}, \frac{1}{1+10^{-20}}\right)$. This is very close to $(1, 1)$. Here's what Gaussian elimination would do. First, we form the following augmented matrix:

$$\begin{bmatrix} 10^{-20} & 1 & 1 \\ 1 & -1 & 0 \end{bmatrix}$$

The algorithm performs the following steps:

1. Rescale the first row to $\begin{bmatrix} 1 & 10^{20} & 10^{20} \end{bmatrix}$.
2. Subtract the first row from the second row and replace the latter: $\begin{bmatrix} 0 & -1 - 10^{20} & -10^{20} \end{bmatrix}$.

Now suppose our machine has finite precision: $-1 - 10^{20}$ is replaced with -10^{20}, as both quantities are essentially "negative infinity". The reduced matrix is the following:

$$\begin{bmatrix} 1 & 10^{20} & 10^{20} \\ 0 & -10^{20} & -10^{20} \end{bmatrix}$$

Back substitution would then give us the solution $\mathbf{x} = (0, 1)$. This is completely wrong!

Example: Second Try

Now let's try solving the system again, but with the equation order reversed.

$$x_1 - x_2 = 0 \tag{3}$$
$$10^{-20} x_1 + x_2 = 1 \tag{4}$$

The augmented matrix is now

$$\begin{bmatrix} 1 & -1 & 0 \\ 10^{-20} & 1 & 1 \end{bmatrix}$$

The elimination algorithm subtracts 10^{-20} times the first row from the second row and replaces the latter: $\begin{bmatrix} 0 & 1 + 10^{-20} & 1 \end{bmatrix}$.

Again, our machine has finite precision, so $1 + 10^{-20}$ is replaced with 1. The reduced matrix is the following:

$$\begin{bmatrix} 1 & -1 & 0 \\ 0 & 1 & 1 \end{bmatrix}$$

Back substitution gives us the solution $\mathbf{x} = (1, 1)$, which is as close as we can get to the true solution on our machine.

Finally, just to convince you that what you saw here is very much a real problem, run the code cell below to show that the "order" of the linear equations does make a difference.

In [ ]:
```
print(GE(np.array([[1e-20,1,1], [1,-1,0]])))
print(GE(np.array([[1,-1,0], [1e-20,1,1]])))
```

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

Explanation

Why do we get two different solutions for the same system simply from switching the equation order? And why is one of the solutions completely wrong?

The answer lies in the first pivot entry, or the first entry of the first row. In our first try, this value is 10^{-20}, an extremely small number that is very close to 0. In the second try, this value is simply 1. The smaller the magnitude of the pivot entry, the more likely that we will run into trouble with roundoff errors. This example may seem very generic, but you can hopefully imagine how the potential for errors can arise when we are working with large matrices and real, messy data.

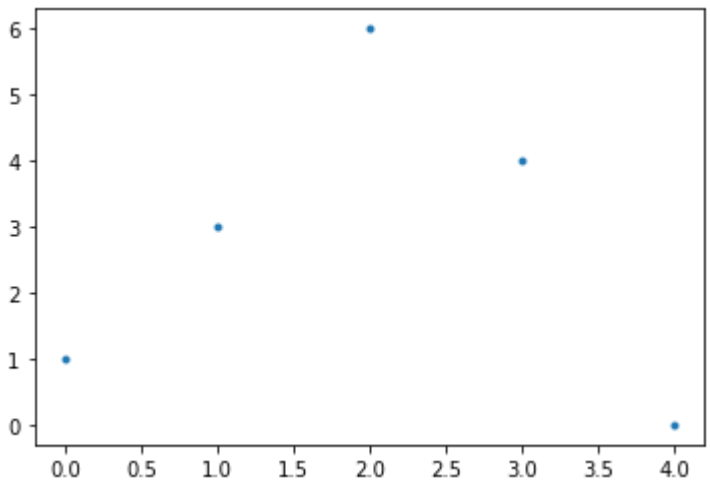
Exercises

The problems below will demonstrate more issues that can arise with numerical accuracy and Gaussian elimination. At the end, you will "fix" the elimination algorithm with a small modification to ensure that all these problems are correctly solved.

You will have to generate plots for some of the problems here. In terms of grading, we will only be looking at your plots and responses, as well as code for Problem 3. You should still leave all code that you used for other problems, but you don't have to worry about making it neat and presentable. As a reminder, you can use `matplotlib.pyplot.plot` as follows:

In [ ]:
```
import matplotlib.pyplot as plt
x_points = np.array([0, 1, 2, 3, 4])
y_points = np.array([1, 3, 6, 4, 0])
plt.plot(x_points, y_points, '.')
```

Out[ ]:
```
[<matplotlib.lines.Line2D at 0x7f54917ea650>]
```



PROBLEM 1 (20 points)

Consider the linear system $A\mathbf{x} = \mathbf{b}$, where

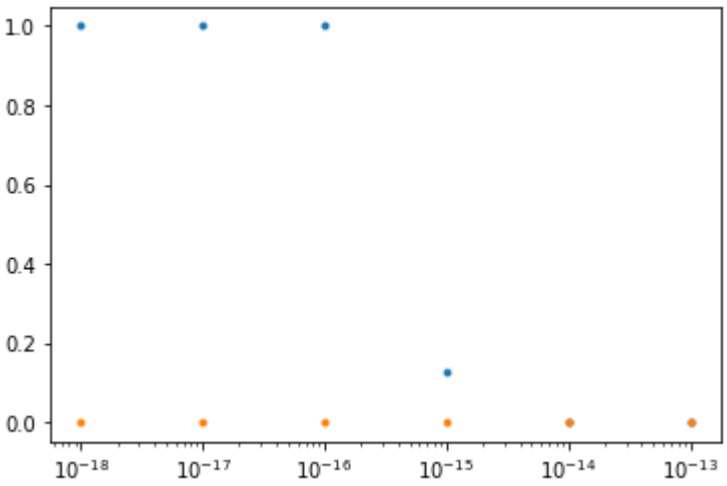
$$A = \begin{bmatrix} \varepsilon & 1 \\ 1 & 1 \end{bmatrix}, \qquad \mathbf{b} = \begin{bmatrix} 1+\varepsilon \\ 2 \end{bmatrix}.$$

The true solution is $\mathbf{x} = (1, 1)$, although you may get something different depending on the value of ε. For each of the values $\varepsilon = 10^{-13}, 10^{-14}, 10^{-15}, 10^{-16}, 10^{-17}, 10^{-18}$, find the solutions that would be returned by Gaussian elimination. Generate a plot of the norm of the difference between the actual solution and computed solution as a function of ε.

Lastly, repeat the above experiment and generate a second plot, but after swapping the order of the two rows of the augmented matrix.

In [ ]:
```
# YOUR CODE HERE
epsilons = [10**-13, 10**-14, 10**-15, 10**-16, 10**-17, 10**-18]
sol = [1, 1]
y1 = []
y2 = []
for eps in epsilons:
  A = np.array([[eps, 1, 1+eps], [1, 1, 2]])
  B = np.array([[1, 1, 2], [eps, 1, 1+eps]])
  X = GE(A)
  Y = GE(B)
  calcSol1 = np.array([X[0, 2], X[1,2]])
  calcSol2 = np.array([Y[0, 2], Y[1,2]])
  norm1 = np.linalg.norm(calcSol1-sol)
  norm2 = np.linalg.norm(calcSol2-sol)
  y1.append(np.linalg.norm(calcSol1-sol))
  y2.append(np.linalg.norm(calcSol2-sol))
plt.semilogx(epsilons, y1, '.')
plt.semilogx(epsilons, y2, '.')
```

Out[ ]:
```
[<matplotlib.lines.Line2D at 0x7f547e546a50>]
```



PROBLEM 2 (20 points)

A Hilbert matrix has the (i, j)th entry equal to $\frac{1}{i+j-1}$. For example, the 3 by 3 Hilbert matrix is

$$\begin{bmatrix} 1 & \frac{1}{2} & \frac{1}{3} \\ \frac{1}{2} & \frac{1}{3} & \frac{1}{4} \\ \frac{1}{3} & \frac{1}{4} & \frac{1}{5} \end{bmatrix}.$$

There are built-in procedures in `scipy` to both construct a Hilbert matrix and to compute the inverse of a Hilbert matrix, both shown below.

In [ ]:
```
from scipy.linalg import hilbert
from scipy.linalg import invhilbert
from scipy.linalg import inv

print(hilbert(3))
print(invhilbert(3))
```

```
[[1.         0.5        0.33333333]
 [0.5        0.33333333 0.25      ]
 [0.33333333 0.25       0.2       ]]
[[   9.  -36.   30.]
 [ -36.  192. -180.]
 [  30. -180.  180.]]
```

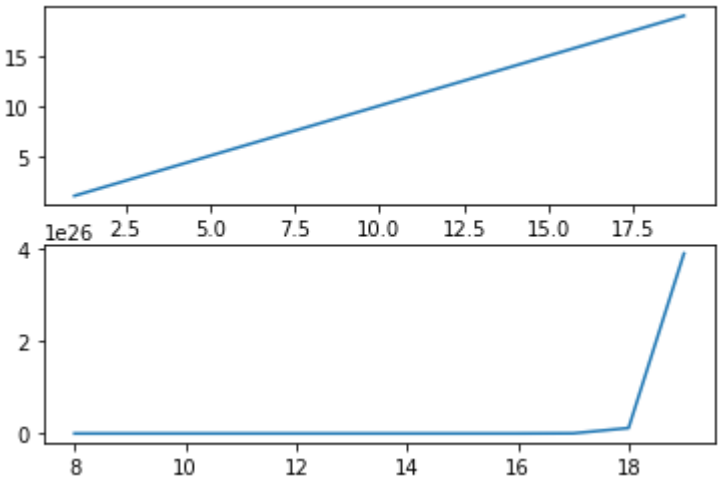
Let's first verify the rank of a Hilbert matrix. For n ranging from 1 to 20, run Gaussian elimination on a $n \times n$ Hilbert matrix and count the number of pivot entries. Generate a plot of these counts as a function of n.

1. What do your observations suggest about the invertibility of an arbitrary Hilbert matrix?

2. Now actually try computing the inverse of a larger Hilbert matrix. Compare the output of `inv` on a Hilbert matrix with the output of `invhilbert` given the same size. You can think of the first method as an output of the Gaussian elmination method for finding an inverse, while the second method produces the "correct" inverse of a Hilbert matrix. What do you observe, especially as n gets larger (you can start around $n = 8$ or so)?

In [ ]:
```
# YOUR CODE HERE
a = []
norms  = []
size = []
for n in range(19):
  x = n + 1
  A = hilbert(x)
  X = GE(A)
  if x>7:
      size.append(x)
      inv1 = inv(A)
      inv2 = invhilbert(x)
      norms.append(np.linalg.norm(inv1-inv2))
  a.append(x)
fig, axs = plt.subplots(2)
axs[0].plot(a, a)
axs[1].plot(size, norms)
```

Out[ ]: [<matplotlib.lines.Line2D at 0x7f547b4688d0>]



YOUR RESPONSES HERE

1. An arbitrary Hilbert matrix is invertible. The result of gaussian elimination on an n x n Hilbert Matrix is an n x n identity matrix, which is invertible with itself.

2. The norm of the difference between the matrices computed with the inverse function and the inverse Hilbert function grows exponentially as n gets larger.

PROBLEM 3 (20 points)

We will now provide a fix, at least for the problems in the example and in Problem 1. The implementation is called **partial pivoting**, which involves using not the first nonzero entry in the current pivot column, but the entry with the largest magnitude. Thus, in each iteration we search all entries below the current pivot row. If row k contains the largest entry, we end up swapping the current row with row k.

As an example, take a look at the following matrix:

$$A = \begin{bmatrix} 0.01 & 0 & 3 \\ 1 & -2 & 5 \\ -4 & 1 & 0.1 \end{bmatrix}$$

When turning the first column into a proper pivot column, we search downward from row 1 and find that row 3 has the largest magnitude entry (-4). We thus swap row 1 and row 3 before proceeding to zero out the entries below with row operations.

Implement this change below. We are providing the original GE code for you. The change should not take more than one or two lines of code. If you like, you can even avoid using a loop by using `numpy.argmax`.

In [ ]:
```
def GE_new(A):
  A = A.astype(float)
  pr = 0
```

```
  pc = 0

  while pr < A.shape[0] and pc < A.shape[1]:  # run until either last row or last colu
    i = pr + np.argmax(abs(A[pr:, pc]))

    if A[i, pc] == 0:                         # no pivot in this column, move to next
      pc += 1

    else:
      A[[pr, i]] = A[[i, pr]]                          # swap rows with the found pivot
      A[pr, :] /= A[pr, pc]                            # rescale row to make leading er
      A[pr+1:, :] -= np.outer(A[pr+1:, pc], A[pr, :]) # row-reduce all rows below pivo
      A[0:pr, :] -= np.outer(A[0:pr, pc], A[pr, :])   # row-reduce all rows above pivo
      pr += 1                                          # move to next pivot position
      pc += 1

  return A
```

Verify that `GE_new` correctly solves the first two systems in this notebook without us having to change the order of the equations ourselves. Then come up with a larger matrix, at least 4 by 4, such that the "old" and "new" elimination implementations give different results (and the latter should be the correct one!). Show these results in your code cell below, and also type out the matrix you came up with in text.

In [ ]:
```
print(GE_new(np.array([[1e-20,1,1], [1,-1,0]])))
print(GE_new(np.array([[1,-1,0], [1e-20,1,1]])))
print(GE(np.array([[10**-20, 1, 1, 0],[1, 10**-20, 1, 1],[1, 0, 10**-20, 0],[1, 1, 1,
print(GE_new(np.array([[10**-20, 1, 1, 0],[1, 10**-20, 1, 1],[1, 0, 10**-20, 0],[1, 1,
```

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

$$\begin{bmatrix} 10^{-20} & 1 & 1 & 0 \\ 1 & 10^{-20} & 1 & 1 \\ 1 & 0 & 10^{-20} & 0 \\ 1 & 1 & 1 & 10^{-20} \end{bmatrix}$$