Damian Franco

dfranco24@unm.edu

101789677

CS-575

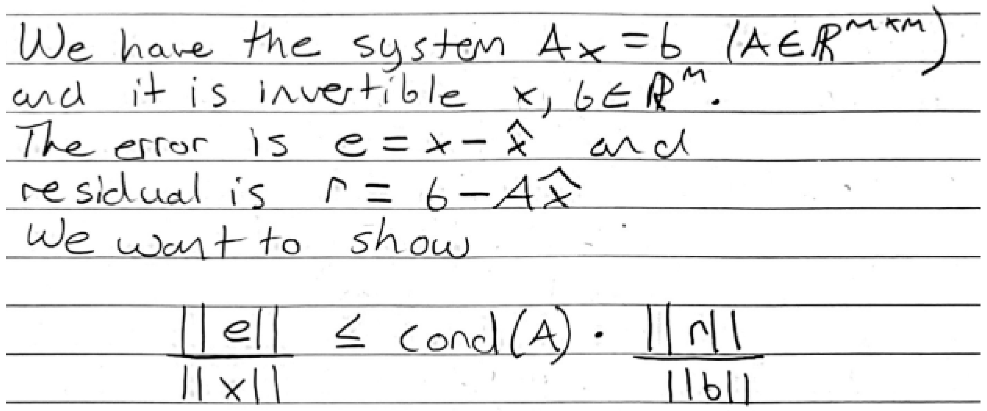
Homework 4

1) For the first problem were given a few hypothetical problems with an input *mxm* non-singular matrix *A*. Non-singular matrices are generally considered good for numerical computations because they have unique solutions to linear systems of equations while singular matrices do not have unique solutions and can lead to numerical instability and inaccuracies which is important to note because we are dealing with a non-singular matrix here. I think something that is very interesting to note that I learned is that the most appropriate method for computation will depend on the specific properties of the given matrix.
    a) The first question asks us to to find a effective solution to solving the linear system *A^k*x = b*. The approach I would take to effectively solving this system is by using the matrix power method. First, we would need to compute the matrix power *A^k* using any available method in which I chose the exponentiating by squaring approach. Exponentiating by squaring is a general method for fast computation of large positive integer powers of a number, or more generally of an element of a semigroup, of which a square matrix is a perfect example. Next, we would have to solve the system by using an effective method for solving such as Gaussian elimination, LU factorization, or QR decomposition. I would personally choose GEPP for solving the system because I am most familiar with it, but I know that LU and QR may have unknown computational advantages when it comes to square non-singular matrices. Any of these approaches are all numerically efficient for solving the system. Overall, the matrix power method is efficient in this case because it requires only one matrix multiplication of *A* with itself for each power of *k*, as opposed to explicitly multiplying *A* by itself *k* times. This indicates that computational time would be very efficient as well.

b) Much similar to the first problem, solving this equation relates to the structure of our A matrix and will be split into multiple parts for understandable and efficient computation. The computation we are doing for this question is $\alpha = c^T A^{-1} b$ where we are solving alpha. First, we want to compute the inverse of the matrix A. This can be done using a matrix inversion algorithm such as Gaussian elimination or LU decomposition. Depending on the size m of the matrix A, LU may be more efficient than GEPP, but I would opt in to using GE because I am more familiar with that. Next, we want to focus on computing the matrix-vector product of $A^{-1}b$. This can be done using matrix-vector multiplication, which involves multiplying each row of A^{-1} by the corresponding element of b and summing the results. Lastly, we want to perform multiplication of c^T and the product of $A^{-1}b$ found in the previous step. Instead of doing matrix-vector multiplication we would use the dot product here. We do so because matrix multiplication is basically a matrix version of the dot product and here we have two vectors. This is the most efficient way I would solve the computation of these questions with an efficient and effective approach.

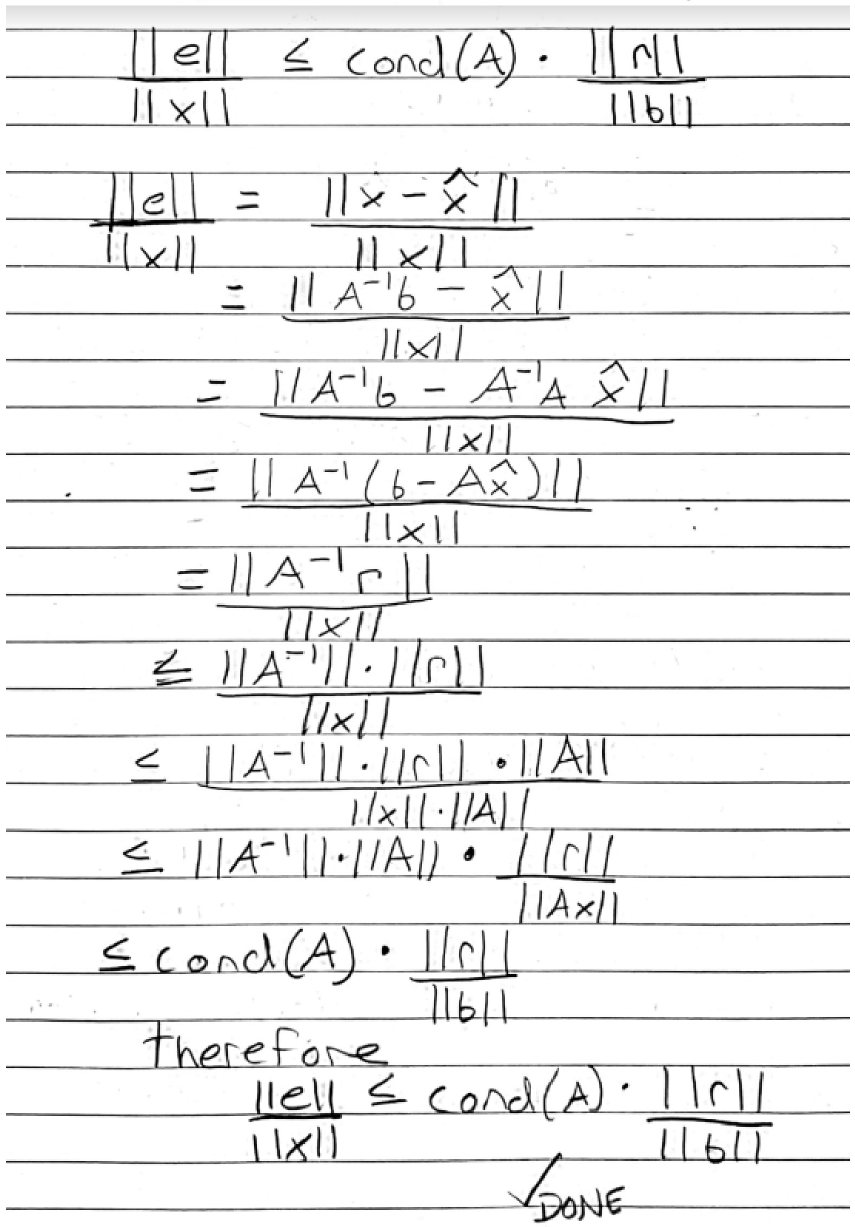
c) Next we are given the matrix equation $AX = B$, where A is a square non-singular matrix and B is an mxn matrix. Once again, I will be taking the approach of splitting this problem into various steps where there are two steps and algorithms applied at each step. The first step involves computing the inverse of A since we will be solving for X and our equation becomes $X = A^{-1}B$. This can be done efficiently using matrix factorization methods and the most familiar one that I will choose is the Gaussian elimination. Next we want to calculate the solution X within involves matrix-matrix multiplication of A^{-1} and B. That is all that needs to be done to compute this equation. The overall performance should be efficient and accurate since we know that GE is effective as well as matrix-matrix multiplication algorithms are somewhat efficient. This would be the most computationally dense out of all three computations in this problem because of the matrix-matrix multiplication, but should still perform well.

2) The second problem asks us to prove a relative residual perturbed matrix theorem that was shown in class. Below you can view the steps to my proof which I believe was relatively straightforward.



We have the system $Ax = b$ $(A \in \mathbb{R}^{m \times m})$ and it is invertible $x, b \in \mathbb{R}^m$.
The error is $e = x - \hat{x}$ and residual is $r = b - A\hat{x}$
We want to show

$$\frac{\|e\|}{\|x\|} \leq \operatorname{cond}(A) \cdot \frac{\|r\|}{\|b\|}$$



$$\frac{\|e\|}{\|x\|} \leq \operatorname{cond}(A) \cdot \frac{\|r\|}{\|b\|}$$

$$\frac{\|e\|}{\|x\|} = \frac{\|x - \hat{x}\|}{\|x\|}$$

$$= \frac{\|A^{-1}b - \hat{x}\|}{\|x\|}$$

$$= \frac{\|A^{-1}b - A^{-1}A\hat{x}\|}{\|x\|}$$

$$= \frac{\|A^{-1}(b - A\hat{x})\|}{\|x\|}$$

$$= \frac{\|A^{-1}r\|}{\|x\|}$$

$$\leq \frac{\|A^{-1}\| \cdot \|r\|}{\|x\|}$$

$$\leq \frac{\|A^{-1}\| \cdot \|r\| \cdot \|A\|}{\|x\| \cdot \|A\|}$$

$$\leq \|A^{-1}\| \cdot \|A\| \cdot \frac{\|r\|}{\|Ax\|}$$

$$\leq \operatorname{cond}(A) \cdot \frac{\|r\|}{\|b\|}$$

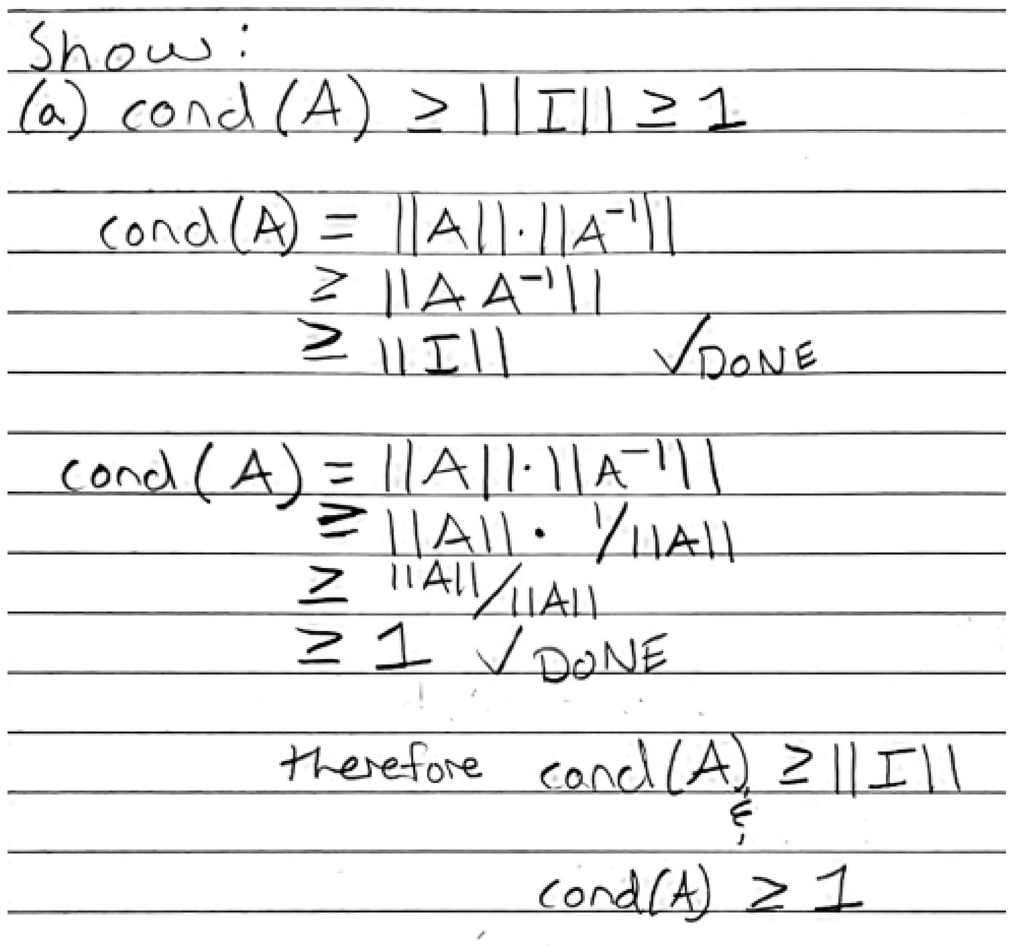
Therefore

$$\frac{\|e\|}{\|x\|} \leq \operatorname{cond}(A) \cdot \frac{\|r\|}{\|b\|}$$

✓DONE

3) Next, we were given two proofs for the condition number of a *mxm* matrix *A*. The two proofs are below.

   a) The first part of the question asks us to prove that the condition number of *A* is less than or equal to both ||*I*|| and *1*. I split this problem into two parts and used various induced matrix norm rules and applied those to find that both proofs were true. Below is my proof.



Show:

(a) $\text{cond}(A) \geq ||I|| \geq 1$

$$\begin{aligned}\text{cond}(A) &= ||A|| \cdot ||A^{-1}|| \\ &\geq ||A A^{-1}|| \\ &\geq ||I|| \quad \checkmark \text{DONE}\end{aligned}$$

$$\begin{aligned}\text{cond}(A) &= ||A|| \cdot ||A^{-1}|| \\ &\geq ||A|| \cdot 1/||A|| \\ &\geq ||A||/||A|| \\ &\geq 1 \quad \checkmark \text{DONE}\end{aligned}$$

therefore $\text{cond}(A) \geq ||I||$

&

$\text{cond}(A) \geq 1$

b) This proof is somewhat similar to proofs we have done in class and on the previous homework that involved norms. This problem utilized the norm rules and the equivalence of the condition number on the inverse norm multiplied to the norm of *A*, but this question is using two matrices *A* and *B* instead of one. Below is my proof.

(b) $\text{cond}(AB) \leq \text{cond}(A)\,\text{cond}(B)$

$$\begin{aligned}\text{cond}(AB) &= \|AB\| \cdot \|B^{-1}A^{-1}\| \\ &\leq \|A\| \cdot \|B\| \cdot \|B^{-1}\| \cdot \|A^{-1}\| \\ &\leq \|A\| \cdot \|A^{-1}\| \cdot \|B\| \cdot \|B^{-1}\| \\ &\leq \text{cond}(A) \cdot \text{cond}(B)\end{aligned}$$

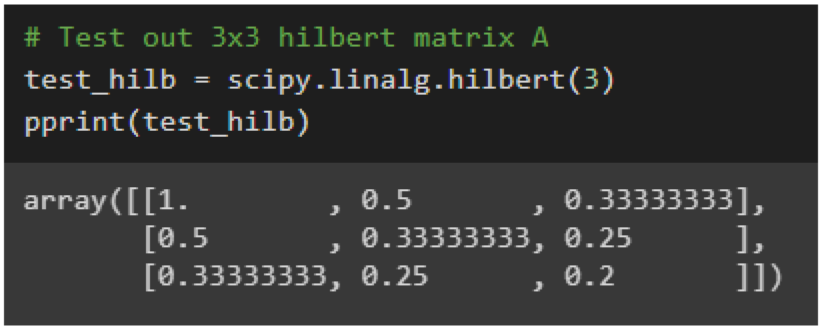
therefore

$\text{cond}(AB) \leq \text{cond}(A)\,\text{cond}(B)$

✓ DONE

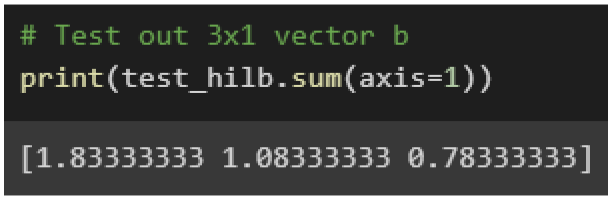
4) The next problem involves solving a linear system and consists of six sub-questions (a)-(f). Each question asks us to solve a system involving Hilbert matrices which are notoriously ill-conditioned matrices. The right hand side or *b* vector for our system is simply the current *i-th* row/column summed together. Below you can see some testing code for the creation of a 3x3 Hilbert matrix and a 3x1 *b* vector for our system that we are solving.

Testing out Hilbert Matrix creation in Python:



```
# Test out 3x3 hilbert matrix A
test_hilb = scipy.linalg.hilbert(3)
pprint(test_hilb)
```

```
array([[1.        , 0.5       , 0.33333333],
       [0.5       , 0.33333333, 0.25      ],
       [0.33333333, 0.25      , 0.2       ]])
```

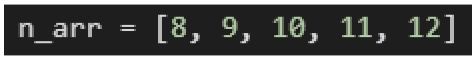


```
# Test out 3x1 vector b
print(test_hilb.sum(axis=1))
```

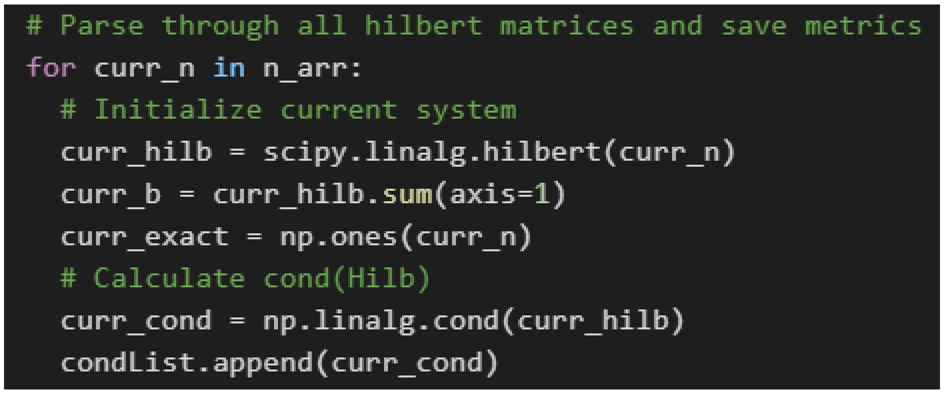
```
[1.83333333 1.08333333 0.78333333]
```

a) For the first question, we were asked to create Hilbert matrices and tabulate the condition numbers of each. As we learned, the condition number can indicate whether a matrix is well or ill-conditioned. A large condition number is a large indicator that a matrix is ill-conditioned and will cause some errors. Here I found that all of my matrices were ill-conditioned but the 12x12 matrix was especially bad. I know when I try to solve this system, it will give me a huge error just by looking at the condition number. Below is my code for calculating my condition number of the matrices as well as the code I used to initialize the matrices to begin with.

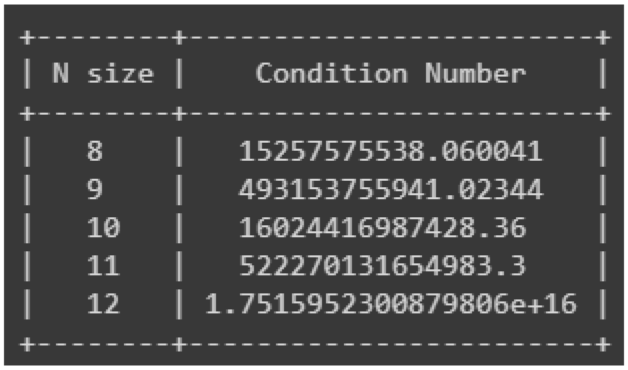
Creating Hilbert matrices and condition numbers:



```
n_arr = [8, 9, 10, 11, 12]
```



```
# Parse through all hilbert matrices and save metrics
for curr_n in n_arr:
  # Initialize current system
  curr_hilb = scipy.linalg.hilbert(curr_n)
  curr_b = curr_hilb.sum(axis=1)
  curr_exact = np.ones(curr_n)
  # Calculate cond(Hilb)
  curr_cond = np.linalg.cond(curr_hilb)
  condList.append(curr_cond)
```

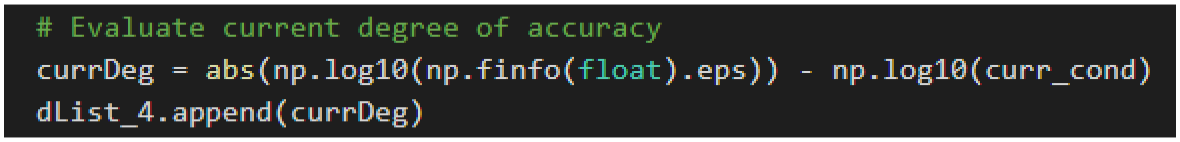


```
+--------+------------------------+
| N size |    Condition Number    |
+--------+------------------------+
|   8    |   15257575538.060041   |
|   9    |   493153755941.02344   |
|   10   |   16024416987428.36    |
|   11   |   522270131654983.3    |
|   12   | 1.7515952300879806e+16 |
+--------+------------------------+
```

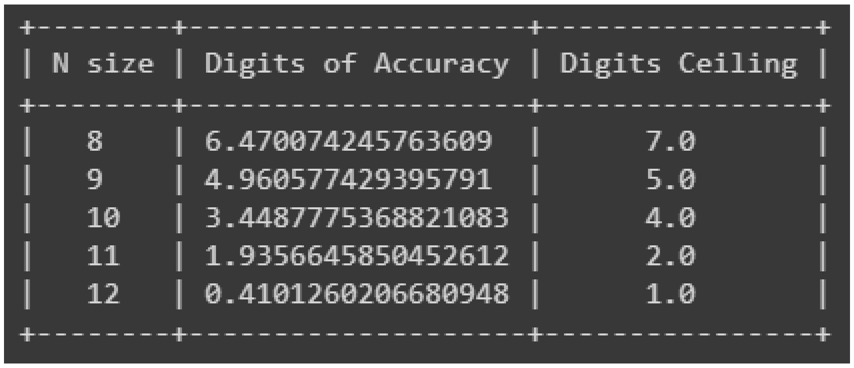
b) As I stated early, the *b* or right-hand side vector in our system is equal to the sum of the current column/row that we are parsing through. For instance, for the first element in *b*, b_1 would be row/column *1* in our Hilbert matrix summed together to get the value placed at b_1 . With that being said, we know that the linear system would be solved if all the unknowns did not change the value in the Hilbert matrices because we need the same sum value for our column/row. This means that our unknown will be 1 for x_1, which also indicates that the vector *x* would be a vector full of value 1 in each position. Our exact or true solution to the linear system in this case will be a vector of size Nx1 with all values being set to one. I did this by using the NumPy ones vector creation function call.

c) This problem asks us to make sure that this is a stable variant of GE by finding the number of accurate decimal digits *d*. I was able to find some interesting results. After making matrices/vectors of sizes 8x8, 9x9, 10x10, 11x11, and 12x12, I found that there was only a maximum 5 digits of accuracy between all of the systems. This is interesting and will be very notable for the computation that is going to take place in the system solving. You can see below the digits of accuracy I found for these systems with the sizes. I used the ceiling function here to see if the whole number of digits rounded up has any correlation to the actual number of precision that we have and found that it does (*somewhat*).

Calculating digits of accuracy for each system:



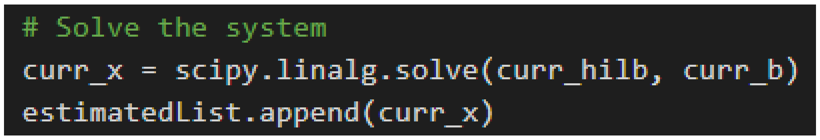
```python
# Evaluate current degree of accuracy
currDeg = abs(np.log10(np.finfo(float).eps)) - np.log10(curr_cond)
dList_4.append(currDeg)
```



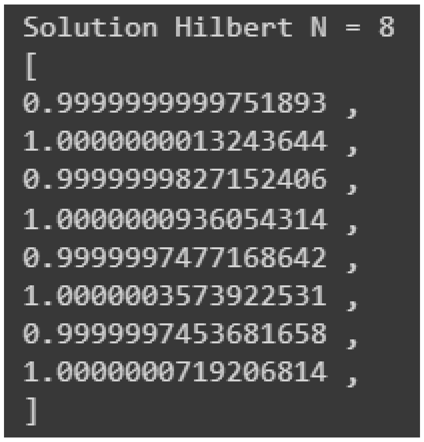
```
+--------+--------------------+----------------+
| N size | Digits of Accuracy | Digits Ceiling |
+--------+--------------------+----------------+
|   8    | 6.470074245763609  |      7.0       |
|   9    | 4.960577429395791  |      5.0       |
|   10   | 3.4487775368821083 |      4.0       |
|   11   | 1.9356645850452612 |      2.0       |
|   12   | 0.4101260206680948 |      1.0       |
+--------+--------------------+----------------+
```

d) We now needed to solve the linear system with a Python/Matlab operator or function and in my case it was the NumPy linear algebra solve function. For each system, we were to take note of our approximation of the solution and display 16 digits of the approximate solution to check if our digits of accuracy were relevant to the system. I found that the digits of accuracy correlated very well with the amount of error that we have in each approximation which you will see in a future part of this problem. It is very interesting that even from just viewing the outcome here, we are able to indicate that there is something very wrong. Luckily for us, we know that the algorithm is stable, but our matrix is ill-conditioned which is where our issues with errors occur. Below you can see my approximations for each system of size N.

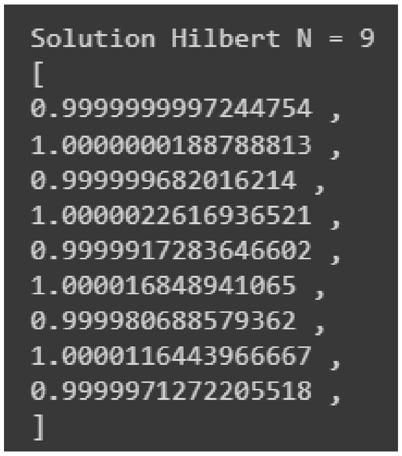
Solving the system for all size N systems:



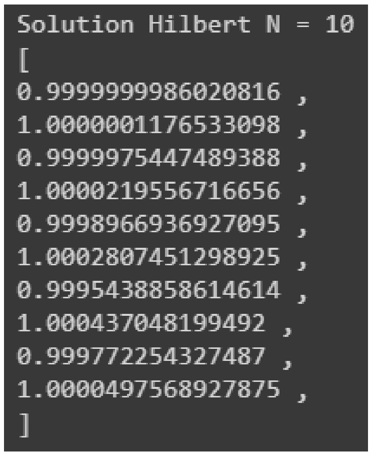
```
# Solve the system
curr_x = scipy.linalg.solve(curr_hilb, curr_b)
estimatedList.append(curr_x)
```



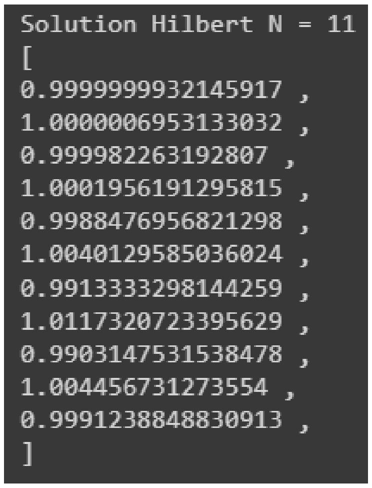
```
Solution Hilbert N = 8
[
0.9999999999751893 ,
1.0000000013243644 ,
0.9999999827152406 ,
1.0000000936054314 ,
0.9999997477168642 ,
1.0000003573922531 ,
0.9999997453681658 ,
1.0000000719206814 ,
]
```



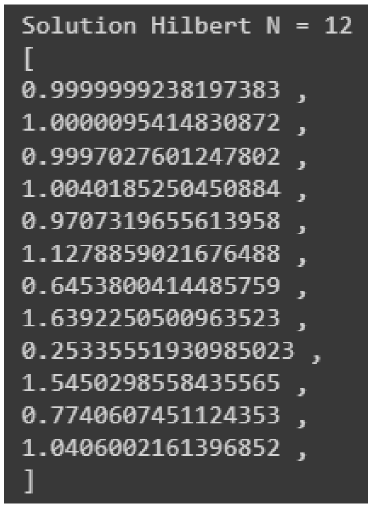
```
Solution Hilbert N = 9
[
0.9999999997244754 ,
1.0000000188788813 ,
0.999999682016214 ,
1.0000022616936521 ,
0.9999917283646602 ,
1.000016848941065 ,
0.999980688579362 ,
1.0000116443966667 ,
0.9999971272205518 ,
]
```



```
Solution Hilbert N = 10
[
0.9999999986020816 ,
1.0000001176533098 ,
0.9999975447489388 ,
1.0000219556716656 ,
0.9998966936927095 ,
1.0002807451298925 ,
0.9995438858614614 ,
1.000437048199492 ,
0.999772254327487 ,
1.0000497568927875 ,
]
```



```
Solution Hilbert N = 11
[
0.9999999932145917 ,
1.0000006953133032 ,
0.999982263192807 ,
1.0001956191295815 ,
0.9988476956821298 ,
1.0040129585036024 ,
0.9913333298144259 ,
1.0117320723395629 ,
0.9903147531538478 ,
1.004456731273554 ,
0.9991238848830913 ,
]
```

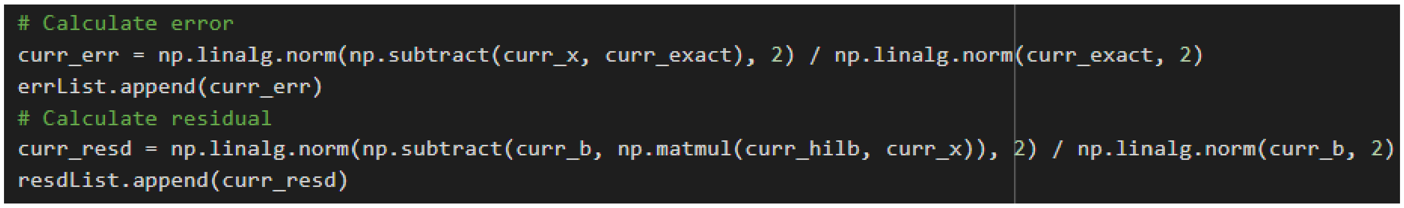


```
Solution Hilbert N = 12
[
0.9999999238197383 ,
1.0000095414830872 ,
0.9997027601247802 ,
1.0040185250450884 ,
0.9707319655613958 ,
1.1278859021676488 ,
0.6453800414485759 ,
1.6392250500963523 ,
0.2533555193098502 3 ,
1.5450298558435565 ,
0.7740607451124353 ,
1.0406002161396852 ,
]
```

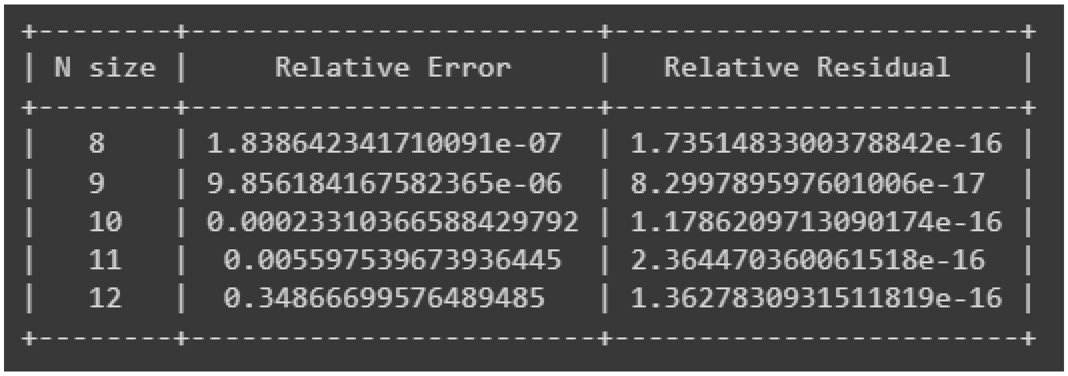
e) For the most part, the digits of accuracy in the approximate solutions above seem to correlate somewhat with my findings in the part above. Some of these systems seem to have more accuracy than I thought. For instance in the N = 8 Hilbert system, it looks like some approximations have about 7 digits of accuracy which is 2 more digits than I calculate while in the N = 12 Hilbert matrix, the digits of accuracy go down to mostly none with one outlier of an approximation with 5 digits of accuracy and another with 1 digit of accuracy, but overall the system was not a single digit accurate. This is interesting to note because the digits that I calculated in the previous part seems to be consistent somewhat when looking at the majority of approximations but some approximations are extremely accurate compared to others which I really do not know why this is occurring. This might be due to some numerical computation bugs or optimization that the NumPy library may have or it could be that my approximation just works better every once and a while if I have good luck. Overall, it is very interesting to note nonetheless.

f) For the last part, the question asks us to tabulate our relative error and the residual of our system and explain what we see. It is interesting to note that the relative residual values are always relatively small while the errors increase as the size of the matrix increases. The error increasing makes perfect sense and the digits of accuracy seem to correlate somewhat with the relative error that is shown here which is very interesting. I believe that the residual for Hilbert matrices is small and stays small regardless of the matrix size growing because the Hilbert matrix has a specific pattern which makes it somewhat symmetrical and orthogonal to a degree. This means that the residual will actually stay consistently small which is what we see here. Overall, this was a very informative question and shows exactly how ill-conditioned matrices can lead to bad results, even in stable algorithms.

Table of relative error and residual:



```
# Calculate error
curr_err = np.linalg.norm(np.subtract(curr_x, curr_exact), 2) / np.linalg.norm(curr_exact, 2)
errList.append(curr_err)
# Calculate residual
curr_resd = np.linalg.norm(np.subtract(curr_b, np.matmul(curr_hilb, curr_x)), 2) / np.linalg.norm(curr_b, 2)
resdList.append(curr_resd)
```

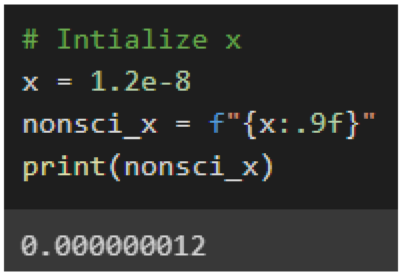


```
+--------+------------------------+------------------------+
| N size |     Relative Error     |   Relative Residual    |
+--------+------------------------+------------------------+
|   8    | 1.838642341710091e-07  | 1.7351483300378842e-16 |
|   9    | 9.856184167582365e-06  | 8.299789597601006e-17  |
|   10   | 0.00023310366588429792 | 1.1786209713090174e-16 |
|   11   |  0.005597539673936445  | 2.364470360061518e-16  |
|   12   |  0.34866699576489485   | 1.3627830931511819e-16 |
+--------+------------------------+------------------------+
```

5) This problem will be a question on our next homework assignment and was moved to accommodate the topics covered this upcoming week by Professor Zeb.

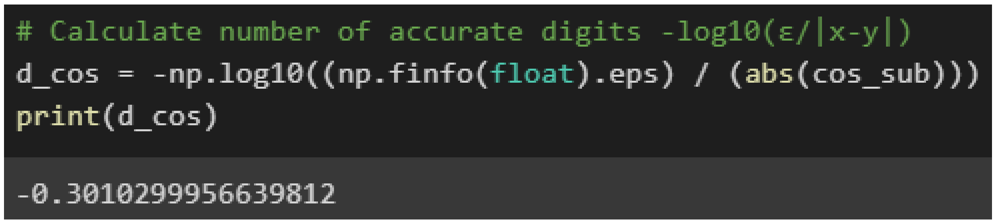
6) For the last problem in the homework, we were asked to look into the precision and loss of precision when subtracting two numbers that are very close in value but not exactly equal to one another.
   a) For the first part of this question, we used the value *x* given to use to solve the function *f(x)*. This was to be done in Python or Matlab and we were to take note of the accurate digits for this approximation. I was able to use the Loss of Precision Theorem to approximate exactly the number of accurate digits in this problem and found that this computation actually has no digits that are accurate when solving this function according to the Loss of Precision Theorem. This is interesting because when I solved the function, it solved with full 15-16 digits of accuracy. I am not sure if this is a bug on my part, but I believe that the value should be very inaccurate due to the subtracting of 1 and *cos(x).* These two values are very close, but not exactly equal which could make computation very inaccurate which is why we see no accurate digits when computing the number of accurate digits. Below you can see how I calculated the function and the digits of accuracy.

Accuracy of original f(x) function with cos(x):



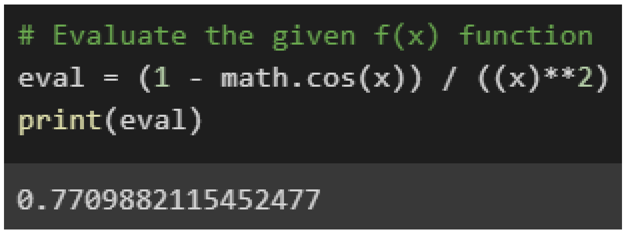
```
# Intialize x
x = 1.2e-8
nonsci_x = f"{x:.9f}"
print(nonsci_x)
```

```
0.000000012
```



```
# Calculate number of accurate digits -log10(ε/|x-y|)
d_cos = -np.log10((np.finfo(float).eps) / (abs(cos_sub)))
print(d_cos)
```

```
-0.3010299956639812
```



```
# Evaluate the given f(x) function
eval = (1 - math.cos(x)) / ((x)**2)
print(eval)
```

```
0.7709882115452477
```

b) Next, we were asked to use the half angle formula in place of *cos(x)* and show that the limit as x approaches 0 of the function is equal to 1/2. Below you can see how I found that it is in fact equal to 1/2.

b) $$\lim_{x \to 0} \frac{1-(1-2\sin^2(x/2))}{x^2}$$

$$= \lim_{x \to 0} \frac{1-1+2\sin^2(x/2)}{x^2}$$

$$= \lim_{x \to 0} \frac{2\sin^2(x/2)}{x^2}$$

$$= 2 \lim_{x \to 0} \frac{\sin^2(x/2)}{x^2}$$

$$= 2 \lim_{x \to 0} \frac{d/dx(\sin^2(x/2))}{d/dx(x^2)}$$ $\leftarrow$ L-Hopital's rule

$$= 2 \lim_{x \to 0} \frac{\cos^2(x/2)}{2}$$

$$= 2 \cdot \left(\frac{\cos(0/2)}{2}\right)^2$$

$$= 2 \cdot \left(\frac{\cos(0)}{2}\right)^2$$

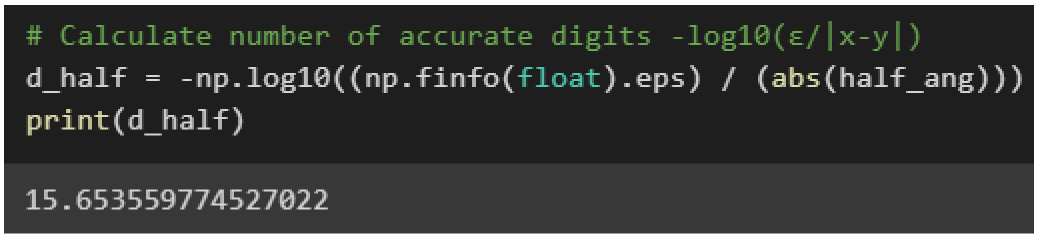
$$= 2 \cdot (1/2)^2$$

$$= 2 \cdot 1/4$$

$$= \underline{1/2}$$ ✓DONE

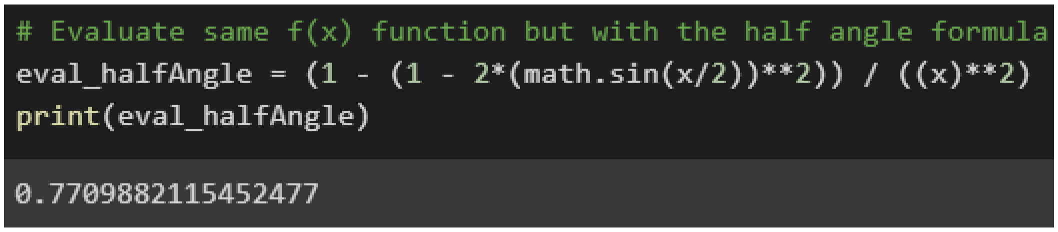
c) Lastly, we were asked to replace *cos(x)* with the half angle formula. I found that this function and small change gave me full 15-16 digits of accuracy which is a surprising but welcomed change to the first implementation of the function with *cos(x)*. I believe this is due to the half-angle formula being close to the value 1, but not extremely like how *cos(x)* was which caused the inaccuracy to begin with. The result and accuracy is different because of this and we see that this function is the right way to implement an accurate version of this function. Below you can see my code and my output for this problem. Still no change in final value computation compared to the first iteration of the function, but nevertheless, the digits of accuracy show a different story than what is being displayed on the evaluation.

Accuracy of updated half-angle f(x) function without cos(x):



```
# Calculate number of accurate digits -log10(ε/|x-y|)
d_half = -np.log10((np.finfo(float).eps) / (abs(half_ang)))
print(d_half)
```

```
15.653559774527022
```



```
# Evaluate same f(x) function but with the half angle formula
eval_halfAngle = (1 - (1 - 2*(math.sin(x/2))**2)) / ((x)**2)
print(eval_halfAngle)
```

```
0.7709882115452477
```