In [2]:
```
"""
Arjun Srivastava
arj1
AMATH 301 B
"""

import numpy as np
import scipy.linalg
import pandas as pd
import time
```

In [3]:
```
# Problem 1

# a)

a, b = 1e21 - (1e21 - 1e5), (1e21 - 1e21) + 1e5

diff = abs(a-b)  # diff = 31072.0

"""
Considering the size of the numbers a and b, I would initially say that this r
esult is not very significant. 31072.0, however,
is still a large number, and there are many contexts in which this error would
be significant.
"""

# b)

A = np.array([[10e-20, 1], [1, 1]])

cond = np.linalg.cond(A)  # cond = 2.6180339887498953, which implies a unique
 solution

# c)

L, U = np.array([[1, 0], [10e20, 1]]), np.array([[10e-20, 1], [0, 1-10e20]])
print('='*30, 'c)', sep='\n')
print(A, (L @ U), sep='\n\n')

"""
L @ U is not equal to A. The first row is, and this is because we are not addi
ng values that involve catastrophic cancellation.
The bottom row, however, involves 10e20 + 10e-20, which should be equal to 0.
 As can be seen by the result printed below,
there is an extra 100. This makes sense due to the immense size of 10e20 and 1
0e-20. The second column of the second row
has 10e20 - 10e20. The result should be 1, but it is instead 0. Catastrophic c
ancellation is occurring in both of these cases
since 10e20 and 10e-20 are being operated on together.
"""

# d)

B = np.array([[1, 1], [10e-20, 1]])
L1, U1 = np.array([[1, 0], [10e-20, 1]]), np.array([[1, 1], [0, 1-10e-20]])
print('='*30, 'd)', sep='\n')
print(B, (L1 @ U1), sep='\n\n')
print('='*30, 'e)', sep='\n')

"""
By reordering the matrix A to B, we have avoided any operation that involves t
wo very large numbers (10e20, 10e-20). Since
10e20 and 10e-20 are only ever being added to or subtracted by 1, there is no
 case in which catastrophic cancellation occurs,
and our answers are much more accurate.
"""
```

```
# e)

P, L2, U2 = scipy.linalg.lu(A)
P = P.T
print(L2, U2, P, sep='\n')
print('\n')

P1, L3, U3 = scipy.linalg.lu(B)
P1 = P1.T
print(L3, U3, P1, sep='\n')

"""
The official SciPy LU yields the same results as part d). This implies that Py
thon is using the permutation matrix
to ensure that the matrix A can be more efficiently decomposed without issues.
As shown, P is different for A and B,
implying that Python wants to avoid catastrophic cancellation by setting up th
e equation in a more efficient manner. Since
A is not set up efficiently for an LU decomp, the permutation matrix converts
 it.
"""
```

```
==============================
c)
[[1.e-19 1.e+00]
 [1.e+00 1.e+00]]

[[1.e-19 1.e+00]
 [1.e+02 0.e+00]]
==============================
d)
[[1.e+00 1.e+00]
 [1.e-19 1.e+00]]

[[1.e+00 1.e+00]
 [1.e-19 1.e+00]]
==============================
e)
[[1.e+00 0.e+00]
 [1.e-19 1.e+00]]
[[1. 1.]
 [0. 1.]]
[[0. 1.]
 [1. 0.]]


[[1.e+00 0.e+00]
 [1.e-19 1.e+00]]
[[1. 1.]
 [0. 1.]]
[[1. 0.]
 [0. 1.]]
```

Out[3]: '\nThe official SciPy LU yields the same results as part d). This implies tha t Python is using the permutation matrix\nto ensure that the matrix A can be more efficiently decomposed without issues. As shown, P is different for A an d B,\nimplying that Python wants to avoid catastrophic cancellation by settin g up the equation in a more efficient manner. Since\nA is not set up efficien tly for an LU decomp, the permutation matrix converts it.\n'

In [4]:
```
# Problem 2

# a)

A = np.genfromtxt('example_matrix.csv', delimiter=',')
```

In [5]:
```
# b)

t0 = time.time()
rb = []
for i in range(100):
    b = np.random.rand(3000 , 1)
    x = scipy.linalg.solve(A, b)
    rb.append(np.max(np.abs((A @ x) - b)))
t1 = time.time()
final_time_b = t1 - t0  # 44.55109429359436 seconds
rb = np.mean(rb)  # 2.954359e-14
```

In [6]:
```
# c)

t0 = time.time()
P, L, U = scipy.linalg.lu(A)
P = P.T
rc = []
for i in range(100):
    b = np.random.rand(3000, 1)
    y = scipy.linalg.solve_triangular(L, P @ b, lower=True)
    x = scipy.linalg.solve_triangular(U, y)
    rc.append(np.max(np.abs((A @ x) - b)))
t1 = time.time()
final_time_c = t1 - t0  # 4.096189737319946 seconds
rc = np.mean(rc)  # 2.958356e-14
```

In [11]:
```
# d)

t0 = time.time()
inv = scipy.linalg.inv(A)
rd = []
for i in range(100):
    b = np.random.rand(3000, 1)
    x = inv @ b
    rd.append(np.max(np.abs((A @ x) - b)))
t1 = time.time()
final_time_d = t1 - t0  # 1.173656940460205 seconds
rd = np.mean(rd)  # 9.092505e-14

# e) modified the code above ^
```

In [8]:
```
# f)

data = {'Method': ['G.E.', 'LU', 'Inverse'], 'total time': [final_time_b, fina
l_time_c, final_time_d], 'average residual': [rb, rc, rd]}
table = pd.DataFrame(data)

# g)

"""
If I only cared about speed, I would choose the inverse method, as it is signi
ficantly faster than the Gaussian Elimination method
and fairly quicker than the LU decomposition. The LU decomposition is close be
hind (3 seconds slower).
"""

# h)

"""
If I only cared about accuracy, I would choose either the Gaussian Elimination
method or the LU decomposition. They have
near identical average residuals, and both are significantly lower than the in
verse method.
"""

# i)

"""
The most ideal method seems like the LU decomposition. It has a fast time and
 much less residual error than the inverse method.
Overall, it seems like the best tradeoff for guaranteeing fast and accurate re
sults, as the Gaussian Elimination method
takes significantly longer to execute.
"""

# j)

"""
The times are increased if these calculations are done inside the loop. For ex
ample, the inverse loop took 1.17 seconds before,
but now takes 65.18 seconds. This is a very important detail, especially if we
are trying to analyze the runtime of a specific
function or process. The LU decomposition and the inverse calculations all tak
e time, and we should always put pre and post
steps outside of the time calculation. In the inverse case, the time increase
 was extremely significant, proving that all steps
matter.
"""

# k)

"""
To better time the efficiency of the solve function, I would start and end my
 times() within the for loop itself, starting
before the solve function and ending right after. After the time is closed for
that particular iteration of the loop,
I would add the difference between the ending time and starting time to an int
```

```
(which was defined before the loop), and then
check the value of the int after the for loop is done. This way, I am only spe
nding time calculating the solve function, and not
the creation of the range() object or the creation of the random vector. If I
 were to implement this, I am sure the times would be smaller
across the board for each method.
"""

table  # show DataFrame
```

Out[8]:

	Method	total time	average residual
0	G.E.	42.274952	2.954359e-14
1	LU	4.819105	2.958356e-14
2	Inverse	1.956770	9.092505e-14

In [ ]: