Coding Report 2

1 Problem Description

In this coding report, we explore the implementation and performance of gaussian elimination and LU factorization. We first demonstrate the methods by solving linear systems from homework 2

$$\begin{bmatrix} 2 & 4 & 5 \\ 7 & 6 & 5 \\ 9 & 11 & 3 \end{bmatrix} x = \begin{bmatrix} 3 \\ 2 \\ 1 \end{bmatrix}.$$

Then we compare two methods:

1. gaussian elimination solve with back substitution (GE)
2. LU factorization then solve with forward and back substitution (LU)

on size $n = 50, 100, 250, 500$ matrices. The linear system $Ax = b$ to solve will have the following setup

$$A = 5 \times I + R$$

where random entries $r_{ij}, b_{ij} \sim N(0, 1)$. We record the errors of two methods in a table and then plot their execution time v.s. n.

2 Results

2.1 Homework Problem

By using LU factorization method on A, we get

$$L = \begin{bmatrix} 1 & 0 & 0 \\ 3.5 & 1 & 0 \\ 4.5 & 0.875 & 1 \end{bmatrix} U = \begin{bmatrix} 2 & 4 & 5 \\ 0 & -8 & -12.5 \\ 0 & 0 & -8.5625 \end{bmatrix}$$

which are the same as the results from homework.

By calling 'gauss_solve' and 'lu_solve', both methods get the same result for 8 digits

$$x = \begin{bmatrix} -0.25547445 \\ 0.13868613 \\ 0.59124088 \end{bmatrix}.$$

2.2 Comparison

To make the comparison fairer, for each n we call 'gauss_solve' and 'lu_solve' 10 times and record the average error and execution time. Since LU provides the option to solve with same A matrix and different b, we also record the time for using the different and same A matrix on two methods.

	GE	LU
50	1.549863e-11	9.336112e-12
100	1.532049e-11	1.424910e-11
250	8.491823e-11	9.370805e-11
500	9.946900e-10	1.124538e-09

Table 1: Errors of $\|Ax - b\|_2$ for GE and LU methods

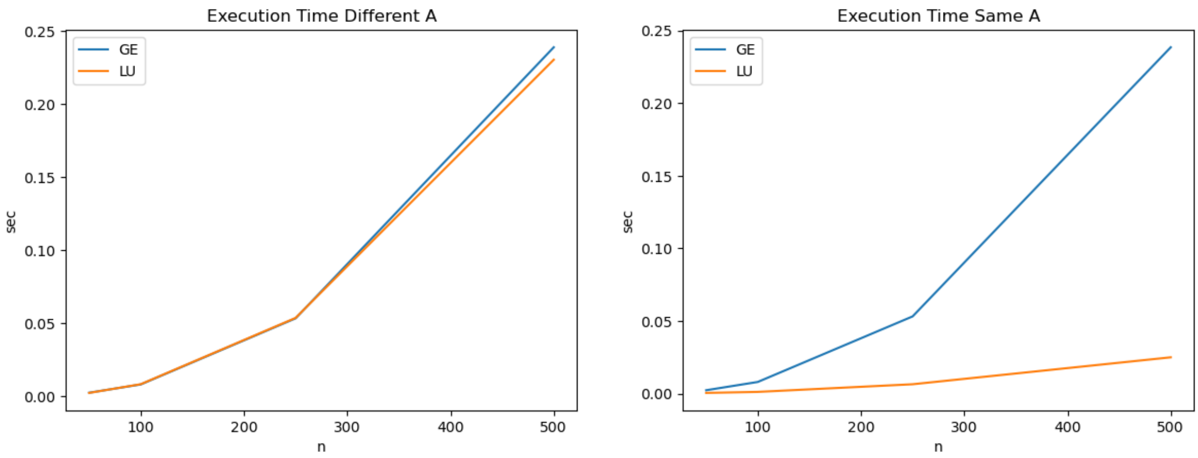

Figure 1: Execution time of GE and LU with different (left) and same (right) A.

From the table, we cen see that then n is small, i.e. $n = 50, 100$, LU results have smaller error than GE results. When n gets larger to $250, 500$, LU lose some accuracy, but GE's error is relatively stable. However, there is no clear evidence to determine the difference in errors for two methods. From the above graph, we can see that for different A matrix every time, LU and GE has roughly the same execution time. When the size of the matrix goes up, LU solves the linear system a litter faster than GE. If we use the same A matrix for the 10 repetition and only solve with different b, we can see that LU is much faster than GE since it only do gaussian elimination the first time and the rest is just forward and backward substitution.

2.3 Experiments on matrix family $\hat{A} = R$

	GE	LU
50	2.427530e-12	3.156321e-12
100	4.067419e-11	4.183250e-11
250	2.874165e-10	3.586288e-10
500	3.865731e-08	1.125961e-08

Table 2: Errors of $\|\hat{A}x - b\|_2$ for GE and LU methods

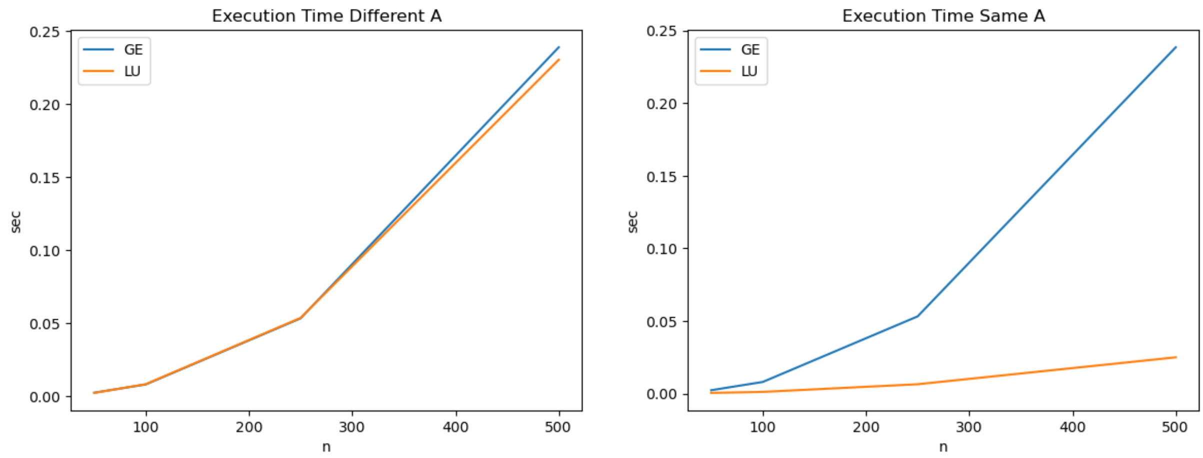

Figure 2: Execution time of GE and LU with different (left) and same (right) $\hat{A}$.

In this case, we still cannot detect any clear trend in errors. In terms of execution time, solving $\hat{A}$ shows similar trends as solving A.

3 Collaboration

No collaboration on this project.

4 Academic Integrity

On my personal integrity as a student and member of the UCD community, I have not given nor received any unauthorized assistance on this assignment.

5 Appendix

Note that there are two source files used for this report, each's name are stated in the title of the file listing. For the following code, please use **python3.10** and have the following packages installed

- numpy
- returns.

```python
import numpy as np
from returns.maybe import Maybe, Nothing, Some
from typing import Tuple


def _select_p(A: np.matrix, i: int, pivot: str) -> Maybe[int]:
    """compute p for gaussian elimination

    Args:
        A (np.matrix): current matrix
        i (int): iteration
```

```
        pivot (str): pivot policy

    Returns:
        Maybe[int]: p
    """
    xs = A[i:, i]
    p = i + 1

    match pivot:
        case "none":
            # find smallest non-zero entry
            idx = np.nonzero(xs)[0]
            # if no integer p can be found
            if len(idx) == 0:
                return Nothing
            p = np.min(idx) + i  # NOTE i is the offset
        case "partial":
            # find max entry
            p = np.argmax(xs) + i
        case _:
            return Nothing

    return Some(p)


def gaussian_elimination(
    A: np.matrix, pivot: str = "none"
) -> Maybe[Tuple[np.matrix, np.matrix]]:
    """Gaussian Elimination without backward substitution
    Args:
        A (np.matrix): matrix to do gaussian elimination
        pivot (str, optional): pivot policy in ["none", "partial"].
   Defaults to "none".

    Returns:
        Maybe[Tuple[np.matrix, np.matrix]]: (elimination result,
   multipliers used)
    """

    if A is None:
        return Nothing

    A = np.ndarray.copy(A)
    n = A.shape[0]
    m = np.identity(A.shape[0])

    # elimination process
    for i in range(n - 1):

        # search for valid p based on pivoting
        p = i + 1
        match _select_p(A, i, pivot):
            case Some(next_p):
                p = next_p
            case Nothing:
                return Nothing
        # no unique solution
        if A[p, i] == 0:
```

```python
                return Nothing
            # swap
            if i != p:
                A[[p, i]] = A[[i, p]]

            # column elimination
            for j in range(i + 1, n):
                m[j, i] = A[j, i] / A[i, i]
                A[j, :] = A[j, :] - m[j, i] * A[i, :]

    # no unique solution
    if A[-1, -1] == 0:
        return Nothing

    return Some((A, m))


def back_substitution(A: np.matrix, b: np.array) -> np.array:
    """back substitution step of gaussian elimination,
    this is assumed to be used upon success of `gaussian_elimination`

    Args:
        mat (np.matrix): matrix from result of gaussian elimination

    Returns:
        np.array: solved values
    """

    n = A.shape[0]
    x = np.zeros(shape=b.shape)

    x[-1, :] = b[-1, :] / A[-1, -1]
    for i in range(n - 2, -1, -1):
        x[i, :] = (b[i, :] - A[i, i + 1 :] @ x[i + 1 :, :]) / A[i, i]

    return x


def forward_substitution(A: np.matrix, b: np.array) -> np.array:
    """forward substitution step,
    this is assumed to be used upon success of `gaussian_elimination`

    Args:
        mat (np.matrix): a square matrix

    Returns:
        np.array: solved values
    """

    n = A.shape[0]
    x = np.zeros(shape=b.shape)

    x[0, :] = b[0, :] / A[0, 0]
    for i in range(1, n):
        x[i, :] = (b[i, :] - A[i, :i] @ x[:i, :]) / A[i, i]

    return x
```

```python
def lu_factorization(A: np.matrix) -> Maybe[Tuple[np.matrix, np.matrix
    ]]:
     """LU factorization

     Args:
         A (np.matrix): square matrix to factorize

     Returns:
         Maybe[Tuple[np.matrix, np.matrix]]: (U, L)
     """
     if A is None:
         return Nothing
     # only decompose square matrix
     if A.shape[0] != A.shape[1]:
         return Nothing
     return gaussian_elimination(A, pivot="none")


def gauss_solve(A: np.matrix, b: np.array) -> Maybe[np.array]:
     """solve linear system Ax = b by gaussian elimination and back
    substitution.
     result depends on the success of gaussian_elimination

     Args:
         A (np.matrix): coefficient matrix
         b (np.array): value vector

     Returns:
         Maybe[np.array]: result
     """
     return gaussian_elimination(np.hstack((A, b))).map(
         lambda p: back_substitution(p[0][:, :-1], p[0][:, [-1]])
     )


def lu_solve(L: np.matrix, U: np.matrix, b: np.array) -> np.array:
     """solve linear system with output from LU factorization
     this is assumed to be used upon the succuss of `lu_factorization`

     Args:
         L (np.matrix): L matrix
         U (np.matrix): U matrix
         b (np.array): value vector that linear system equals

     Returns:
         np.array: solved x unknown variables
     """
     y = forward_substitution(L, b)
     x = back_substitution(U, y)
     return x
```

numerical_methods/linear_direct_methods.py

```python
#!/usr/bin/env python
# coding: utf-8

# In[1]:
```

```python
import numpy as np
from returns.maybe import Maybe, Some, Nothing
from numerical_methods.linear_direct_methods import (
    gaussian_elimination,
    back_substitution,
    lu_factorization,
    lu_solve,
    gauss_solve,
)
import time
import pandas as pd
from returns.curry import partial


# ## Part 1
#
# check hw2 results
#

# In[2]:


A = np.matrix([[2, 4, 5], [7, 6, 5], [9, 11, 3]], dtype=float)
b = np.array([3, 2, 1]).reshape((3, 1))
U, L = lu_factorization(A).unwrap()
L, U


# In[3]:


gaussian_elimination(A, pivot="partial").unwrap()


# In[4]:


mat, _ = gaussian_elimination(np.hstack((A, b))).unwrap()
back_substitution(mat[:, :-1], mat[:, [-1]])


# In[5]:


gauss_solve(A, b)


# In[6]:


lu_solve(L, U, b)


# ## Part 2
#
# error analysis and time analysis
#
```

```python
# In[7]:


def get_err(n: int, repeat: int, changeA: bool, onlyRandomA: bool):
    x1_errs = []
    x2_errs = []
    x3_errs = []
    x1_time = []
    x2_time = []
    x3_time = []

    def get_A():
        A = np.random.normal(size=(n, n))
        if not onlyRandomA:
            A += 5 * np.eye(n)
        return A

    A = get_A()
    U, L = None, None

    for _ in range(repeat):
        if changeA:
            A = get_A()
        b = np.random.normal(size=(n, 1))

        x1_start = time.time()
        x1 = gauss_solve(A, b).unwrap()
        x1_time.append(time.time() - x1_start)

        x2_start = time.time()
        if changeA or U is None or L is None:
            U, L = lu_factorization(A).unwrap()
        x2 = lu_solve(L, U, b)
        x2_time.append(time.time() - x2_start)

        x3_start = time.time()
        x3 = np.linalg.solve(A, b)
        x3_time.append(time.time() - x3_start)

        x1_errs.append(np.linalg.norm(A @ x1 - b))
        x2_errs.append(np.linalg.norm(A @ x2 - b))
        x3_errs.append(np.linalg.norm(A @ x3 - b))

    return map(np.mean, [x1_errs, x2_errs, x3_errs, x1_time, x2_time,
   x3_time])


def test(ns, repeat=10, changeA=True, onlyRandomA=False):
    ts = list(zip(*map(lambda n: get_err(n, repeat, changeA, onlyRandomA
   ), ns)))
    err_df = pd.DataFrame(
        {
            "n": ns,
            "GE": ts[0],
            "LU": ts[1],
        }
    )
```

```
    exec_time = pd.DataFrame({"n": ns, "GE": ts[3], "LU": ts[4], "NP":
    ts[5]})
    return err_df, exec_time


ns = [50, 100, 250, 500]


# $$
# A = 5 \times I + R
# $$
# for different A every time
#

# In[8]:


err_df, exec_time = test(ns)


# In[9]:


print(err_df.to_latex(index=False))
err_df


# In[10]:


exec_time.plot(
    x="n",
    y=["GE", "LU"],
    title="Execution Time Different A",
    legend=True,
    ylabel="sec",
)


# for same $A$ different $b$ every time
#

# In[11]:


_, exec_time_same_A = test(ns, changeA=False)


# In[12]:


exec_time_same_A.plot(
    x="n",
    y=["GE", "LU"],
    title="Execution Time Same A",
    legend=True,
    ylabel="sec",
)
```

```
# $$
# \hat{A} = R
# $$
# different $\hat{A}$ every time
#

# In[13]:


err_df_R, exec_time_R = test(ns, onlyRandomA=True)


# In[14]:


print(err_df_R.to_latex(index=False))
err_df_R


# In[15]:


exec_time_R.plot(
    x="n",
    y=["GE", "LU"],
    title="Execution Time Different A",
    legend=True,
    ylabel="sec",
)


# same $\hat{A}$ different $b$
#

# In[16]:


_, exec_time_R_same_A = test(ns, changeA=False, onlyRandomA=True)


# In[17]:


exec_time_R_same_A.plot(
    x="n",
    y=["GE", "LU"],
    title="Execution Time Same A",
    legend=True,
    ylabel="sec",
)
```

main.py