Programming 4-1 Solution

전기공학전공 홍종혁

Problem 1

전체 코드

```
###Setting up###
import numpy as np                                          #numpy np라는 이름으로 import
n=int(input())                                              #n 입력
A = np.random.normal(0, 1, size=(n, n)) + 10000*np.identity(n)  #matrix A 설정
b = np.random.normal(0, 1, size=(n, 1))                     #matrix b 설정
U=np.copy(A)                                                #A를 복사해 U에 저장
L=np.eye(n)                                                 #L을 n by n 행렬로 설정
bb = np.copy(b)                                             #b를 복사해 bb에 저장

###LU Decomposition###
for i in range(n):                      #for문 이용 n회 반복
    for j in range(i+1,n):              #for문 이용 j가 i+1에서 n-1 될 때까지 반복
        p=U[j,i]/U[i,i]                 #pivot 설정
        L[j,i]=p                        #L[j,i]에 pivot 기록
        U[j,:]-=p*U[i,:]                #행 연산 수행
        bb[j]-=p*bb[i]                  #행 연산 수행

###Check the uniquness of the solution###
rank=np.linalg.matrix_rank(A)           #rank에 A의 rank 저장
det=np.linalg.det(A)                    #det에 A의 determinant 저장
eig,eigv=np.linalg.eig(A)               #eig,eigv에 A의 eigenvalue,eigenvector 저장
if rank!=n:print("not unique")          #rank가 n이 아니면 not unique 출력
else:print("Unique, By rank")          #n이면 Unique,By rank 출력
if det==0:print("not unique")           #det가 0이면 not unique 출력
else:print("Unique, By determinant")    #0이 아니면 Unique,By determinant 출력
if 0 in eig:print("not unique")         #eig에 0이 있으면 not unique출력
else:print("Unique, By eigenvalue")     #아니라면 Unique,By eigenvalue출력

###Back substitution###
x1=np.zeros((n,1))                              #x1 n by 1로 정의
x1[n-1]=bb[n-1]/U[n-1,n-1]                      #최초의 back substitution 수행
for i in range(n):                              #for문 이용 n회 반복
    k=n-1-i                                     #k에 n-1-i 저장
    x1[k]=(bb[k]-U[k,k+1:n]@x1[k+1:n])/U[k,k]   #x1[k]에 해당 연산 수행한 값 저장
```

```
###Solution By LU Decomposition###
invL=np.linalg.inv(L)                   #invL에 L의 inverse matrix 저장
invU=np.linalg.inv(U)                   #invU에 U의 inverse matrix 저장
y=invL@b                                #y=L^-1b 연산 수행
x2=invU@y                               #x2=U^-1y 연산 수행

###Verify###
solve=np.linalg.solve(A,b)              #solve에 선형방정식 solution 저장
if np.allclose(x1,solve,1e-9):          #오차 값이 10^-9 이내면 해당 메시지 출력
    print(f"Back substitution converges to the solution in very small error : 10^-9")
if np.array_equal(x1,solve):            #두 array 동일할 시 해당 메시지 출력
    print(f"Back substitution matched perfectly to the solution")
if np.allclose(x2,solve,1e-9):          #오차 값이 10^-9 이내면 해당 메시지 출력
    print(f"LU Decomposition converges to the solution in very small error : 10^-9")
if np.array_equal(x2,solve):            #두 array 동일할 시 해당 메시지 출력
    print(f"LU Decomposition matched perfectly to the solution")
```

알고리즘 설명

알고리즘 수행에 필요한 Setting

```
###Setting up###
import numpy as np
n=int(input())
A = np.random.normal(0, 1, size=(n, n)) + 10000*np.identity(n)
b = np.random.normal(0, 1, size=(n, 1))
U=np.copy(A)
L=np.eye(n)
bb = np.copy(b)
```

numpy를 np라는 이름으로 import한다. n을 입력 받은 후, A와 **b**를 주어진 조건대로 설정해주고, Upper Triangular matrix를 저장할 변수 U와 Lower Triangular matrix를 저장할 변수 L, Back substitution에 사용할 변수 **bb**를 설정해준다.

Pivoting을 통한 Lower Triangular, Upper Triangular matrix 얻기

```
###LU Decomposition###
for i in range(n):
    for j in range(i+1,n):
        p=U[j,i]/U[i,i]
        L[j,i]=p
        U[j,:]-=p*U[i,:]
        bb[j]-=p*bb[i]
```

반복문을 통해 LU Decomposition을 Gauss Elimination을 통해 수행하여 Lower Triangular matrix L과 Upper Triangular matrix U를 구하고, 첨가행렬 [A|**b**]를 [U|**b**]의 형태로 변형하여 Back substitution에 사용할 **bb**를 얻을 것이다.

for문 진입 전의 U는 A와 동일한 성분을 가지고 있는 n by n matrix로 설정되어 있다. 변수 p에는 U[j,i]/U[i,i]가 저장된다. 이 값을 pivot으로 설정한다. 이 때, U의 대각성분은 $N(0,1^2)$을 따르는 랜덤 난수에 10000이 더해진 값이다. 그 외의 성분은 $N(0,1^2)$을 따르는 랜덤 난수인데, 이 값에서 10000보다 큰 값이 나올 확률은 0에 수렴한다. 따라서 본 문제에서 Gauss Elimination을 수행하기 위해서 Permutation step을 수행할 필요는 없다.

구한 pivot을 U[j,i]에 저장해준다. 그리고 U의 i번째 행에 pivot을 곱한 값을 U의 j번째 행에서 빼준다. **bb**에도 똑같은 연산을 수행한다. 이를 반복하면 L, U, **bb**를 얻을 수 있다.

(1) Show that the above linear equation has a unique solution.

해당 선형 방정식의 해가 Unique하다는 것을 보이는 방법은 여러가지가 있는데, 이 중 숫자를 통해서 알아볼 수 있는 방법을 선택했다. 해가 Unique할 경우엔 eigenvalue의 값들이 모두 non-zero거나, $\det(A) \neq 0$이거나, $\mathrm{rank}(A)=n$인 경우이다.

```
###Check the uniquness of the solution###
rank=np.linalg.matrix_rank(A)
det=np.linalg.det(A)
eig,eigv=np.linalg.eig(A)
if rank!=n:print("not unique")
else:print("Unique, By rank")
if det==0:print("not unique")
else:print("Unique, By determinant")
if 0 in eig:print("not unique")
else:print("Unique, By eigenvalue")
```

출력된 결과는 다음과 같았다.

① $n = 150$

```
Unique, By rank
Unique, By determinant
Unique, By eigenvalue
```

② $n = 250$

```
Unique, By rank
Unique, By determinant
Unique, By eigenvalue
```

determinant 계산 과정에서 overflow관련한 메시지가 나오지만, 계산은 정상적으로 이루어진다.

(2) Use Python to construct the upper triangular matrix and back-substitution algorithm to find the solution.

```
###Back substitution###
x1=np.zeros((n,1))
x1[n-1]=bb[n-1]/U[n-1,n-1]
for i in range(n):
    k=n-1-i
    x1[k]=(bb[k]-U[k,k+1:n]@x1[k+1:n])/U[k,k]
```

Back substitution을 활용하여 solution을 구하는 algorithm이다. 결과값을 담을 **x1**을 선언하고, **x1**의 마지막 index, 즉 U의 마지막 행과 동일한 행의 값을 미리 계산해준다. 그 후 for문을 통해서 행을 거슬러 올라가면서 연산을 수행한다. 이전 행에서 구한 결과 값들이 `x1[k+1:n]`에 들어 있기 때문에, `U[k,k+1:n]`와 행렬 곱 한 값을 `bb[k]`에서 빼 준 후 `U[k,k]`로 나눠 주기만 하면 k행의 해를 구할 수 있다. 해당 algorithm을 통해서 구한 solution은 **x1**에 저장되었다.

(3) Use Python to construct the LU decomposition algorithm to find the solution.

```
###Solution By LU Decomposition###
invL=np.linalg.inv(L)
invU=np.linalg.inv(U)
y=invL@b
x2=invU@y
```

invL에 L의 inverse matrix를, invU에 U의 inverse matrix를 저장해준다. $A\mathbf{x} = \mathbf{b}$ 형태의 방정식은 LU

Decomposition을 수행하면 $L\mathbf{y} = \mathbf{b}, U\mathbf{x} = \mathbf{y}$의 두 가지 방정식으로 나누어 생각할 수 있다. 따라서 $\mathbf{y} = L^{-1}\mathbf{b}$, $\mathbf{x} = U^{-1}\mathbf{y}$ 를 수행하면, solution을 구할 수 있다. 이 solution은 **x2**에 저장되었다.

(4) Verify your solution using numpy.linalg.solve

```
###Verify###
solve=np.linalg.solve(A,b)
if np.allclose(x1,solve,1e-9):
    print(f"Back substitution converges to the solution in very small error : 10^-9")
if np.array_equal(x1,solve):
    print(f"Back substitution matched perfectly to the solution")
if np.allclose(x2,solve,1e-9):
    print(f"LU Decomposition converges to the solution in very small error : 10^-9")
if np.array_equal(x2,solve):
    print(f"LU Decomposition matched perfectly to the solution")
```

numpy.linalg.solve를 통해서 구한 solution과, Back substitution을 통해 구한 **x1**, LU decomposition을 통해서 구한 **x2**를 비교하여, algorithm이 옳았는지 확인할 것이다. **solve** 변수에 numpy.linalg.solve를 통해 구한 solution을 저장한다. numpy의 allclose 함수를 이용하여, 일정 오차 이내에서 근사하는지 확인할 수 있고, numpy의 array_equal 함수를 활용하여 완전히 동일한 지 확인할 수 있다. 통상적으로 오차는 10^{-6}으로 설정하나, 더 정확한 확인을 위해 범위를 10^{-9}로 좁혀보았다. 출력된 결과는 다음과 같았다.

① $n = 150$

```
Back substitution converges to the solution in very small error : 10^-9
LU Decomposition converges to the solution in very small error : 10^-9
```

② $n = 250$

```
Back substitution converges to the solution in very small error : 10^-9
LU Decomposition converges to the solution in very small error : 10^-9
```

완벽히 일치하지는 않았으나, 매우 작은 오차 내에서 solution 값으로 수렴하는 것을 확인할 수 있었다. 따라서 Back substitution과 LU decomposition을 통해 구한 solution은 올바른 solution임이 확인되었다.

또한 Permutation을 수행하는 알고리즘은 따로 작성하여 부록에 첨부하였다.

Problem 2

전체 코드

```
###Setting up###
import numpy as np                                                  #numpy np라는 이름으로 import
from matplotlib import pyplot as plt                                #pyplot plt라는 이름으로 import
n=int(input())                                                      #n 입력
A = np.random.normal(0, 1, size=(n, n)) + 10000*np.identity(n)      #matrix A 설정
b = np.random.normal(0, 1, size=(n, 1))                             #matrix b 설정
solve=np.linalg.solve(A,b)                                          #선형방정식 solution solve에 저장

###Jacobi###
def Jacobi(A,b,xJ,prevxJ,n):                            #A,b,xJ,prevxJ,n을 입력받는 함수 Jacobi 정의
    temp=np.zeros((n,1))                                #temp는 n by 1 vector
    prevxJ=np.copy(xJ)                                  #xJ를 복사해 prevxJ에 저장
    for i in range(n):                                  #for문 활용 n회 반복
        temp[i]=-(A[i,:i]@xJ[:i]+A[i,i+1:]@xJ[i+1:])#Jacobi 연산 수행한 값 temp[i]에 저장
        temp[i]=(temp[i]+b[i])/A[i,i]                   #Jacobi 연산 수행한 값 temp[i]에 저장
    xJ=np.copy(temp)                                    #xJ에 temp 복사해 저장
    return xJ,prevxJ                                    #xJ, prevxJ 반환
xJ=np.zeros((n,1))                          #xJ는 n by 1 vector
cntJ=0                                      #변수 cntJ 선언
cntJ_index=[0]                              #변화하는 cntJ 값 저장할 리스트 cntJ_index 선언
J=np.copy(xJ)                               #J는 n by 1 vector
prevJ=np.copy(xJ)                           #prevJ는 n by 1 vector
estJ=np.zeros((n,1))                        #estJ는 n by 1 vector
while True:                                 #while문 무한 반복
    cntJ+=1                                 #cntJ 1 증가
    cntJ_index.append(cntJ)                 #cntJ 값을 cntJ_index에 추가
    J,prevJ=Jacobi(A,b,J,prevJ,n)           #A,b,xJ,prevxJ,n을 입력받아 반환값을 J,prevJ에 저장
    if np.allclose(J,prevJ,1e-100):         #J와 prevJ의 오차 범위가 10^-100 이내라면
        xJ=J                                #J를 xJ에 저장
        break                               #while문 종료
    estJ=np.hstack((estJ, J))               #estJ에 J 이어붙임
estJ=np.hstack((estJ, xJ))                  #estJ에 J 이어붙임
```

```
###Gauss-Seidel###
def Gauss(A,b,xG,prevxG,n):                             #A,b,xG,prevxG,n을 입력받는 함수 Gauss 정의
    temp=np.zeros((n,1))                                #temp는 n by 1 vector
    prevxG=np.copy(xG)                                  #xG를 복사해 prevxG에 저장
    for i in range(n):                                  #for문 활용 n회 반복
        temp[i]=-(A[i,:i]@xG[:i]+A[i,i+1:]@prevxG[i+1:])#Gauss-Seidel 연산 수행한 값 temp[i]에 저장
        temp[i]=(temp[i]+b[i])/A[i,i]                   #Gauss-Seidel 연산 수행한 값 temp[i]에 저장
        xG=np.copy(temp)                                #xG에 temp 복사해 저장
    return xG,prevxG                                    #xG, prevxG 반환
xG=np.zeros((n,1))                          #xG는 n by 1 vector
cntG=0                                      #변수 cntG 선언
cntG_index=[0]                              #변화하는 cntG 값 저장할 리스트 cntG_index 선언
G=np.copy(xG)                               #G는 n by 1 vector
prevG=np.copy(xG)                           #prevG는 n by 1 vector
estG=np.zeros((n,1))                        #estG는 n by 1 vector
while True:                                 #while문 무한 반복
    cntG+=1                                 #cntG 1 증가
    cntG_index.append(cntG)                 #cntG 값을 cntG_index에 추가
    G,prevG=Gauss(A,b,G,prevG,n)            #A,b,xG,prevxG,n을 입력받아 반환값을 G,prevG에 저장
    if np.allclose(G,prevG,1e-100):         #G와 prevG의 오차 범위가 10^-100 이내라면
        xG=G                                #G를 xG에 저장
        break                               #while문 종료
    estG=np.hstack((estG, G))               #estG에 G 이어붙임
estG=np.hstack((estG, xG))                  #estG에 G 이어붙임
```

```
###Plot the Jacobi Convergence###
plt.figure                              #figure 생성
for i in range(n):                      #for문 활용 n회 반복
    plt.plot(cntJ_index,estJ[i,:])      #cntJ_index,estJ 사용하여 plot
plt.title("Jacobi Convergence")         #제목 설정
plt.xticks(np.arange(0, cntJ+1))        #x축 눈금 설정
plt.show()                              #plot 수행

###Plot the Gauss-Seidel Convergence###
plt.figure                              #figure 생성
for i in range(n):                      #for문 활용 n회 반복
    plt.plot(cntG_index,estG[i,:])      #cntG_index,estG 사용하여 plot
plt.title("Gauss-Seidel Convergence")           #제목 설정
plt.xticks(np.arange(0, cntG+1))        #x축 눈금 설정
plt.show()                              #plot 수행

###Plot square error for Jacobi###
estSEJ=(solve-estJ[:,0].reshape(-1,1))**2                           #estSEJ의 초기값 설정
for i in range(1,len(estJ[0,:])):                                   #for문 사용하여 반복
    estSEJ=np.hstack((estSEJ,(solve-estJ[:,i].reshape(-1,1))**2))   #estSEJ에 square error 저장
plt.figure                                      #figure 생성
for i in range(n):                              #for문 활용 반복
    plt.plot(cntJ_index,estSEJ[i,:])            #cntJ_index, estSEJ 사용하여 plot
plt.title("Jacobi square error Convergence")    #제목 설정
plt.xticks(np.arange(0, cntJ+1))                #x축 눈금 설정
plt.show()                                      #plot 수행
```

```
###Plot square error for Gauss-Seidel###
estSEG=(solve-estG[:,0].reshape(-1,1))**2                           #estSEG의 초기값 설정
for i in range(1,len(estG[0,:])):                                   #for문 사용하여 반복
    estSEG=np.hstack((estSEG,(solve-estG[:,i].reshape(-1,1))**2))   #estSEG에 square error 저장
plt.figure                                      #figure 생성
for i in range(n):                              #for문 활용 반복
    plt.plot(cntG_index,estSEG[i,:])            #cntG_index, estSEG 사용하여 plot
plt.title("Gauss-Seidel square error Convergence")  #제목 설정
plt.xticks(np.arange(0, cntG+1))                #x축 눈금 설정
plt.show()                                      #plot 수행
```

알고리즘 설명

알고리즘 수행에 필요한 Setting

```
###Setting up###
import numpy as np
from matplotlib import pyplot as plt
n=int(input())
A = np.random.normal(0, 1, size=(n, n)) + 10000*np.identity(n)
b = np.random.normal(0, 1, size=(n, 1))
solve=np.linalg.solve(A,b)
```

numpy를 np로, matplotlib의 pyplot을 plt로 import한다. n을 입력받은 후, 주어진 식에 따라 A와 **b**를 설정한다. 그리고 $A\mathbf{x} = \mathbf{b}$를 numpy.linalg.solve를 이용하여 푼 값을 solve에 저장한다.

(1) Use Python to construct the Jacobi algorithm and find the solution

```
###Jacobi###
def Jacobi(A,b,xJ,prevxJ,n):
    temp=np.zeros((n,1))
    prevxJ=np.copy(xJ)
```

```
    for i in range(n):
        temp[i]=-(A[i,:i]@xJ[:i]+A[i,i+1:]@xJ[i+1:])
        temp[i]=(temp[i]+b[i])/A[i,i]
    xJ=np.copy(temp)
    return xJ,prevxJ
```

A, **b, xJ, prevxJ,** n을 입력받아, Jacobi algorithm을 수행하는 함수를 먼저 정의하였다. 함수 내부에서, n by 1인 벡터 **temp**를 임시로 정의해 주고, 입력 받은 **xJ**을 복사하여 **prevxJ**에 저장한다. 그 후 for문을 활용하여, 각 행의 성분에 대해서 $x_i = \sum_{\substack{j=1 \\ j\neq i}}^{n}\left(-\frac{a_{ij}x_j}{a_{ii}}\right)+\frac{b_i}{a_{ii}}$를 수행한다. 그 후 **temp**의 값을 **xJ**에 저장해주고, **xJ**과 **prevxJ**을 반환한다.

```
xJ=np.zeros((n,1))
cntJ=0
cntJ_index=[0]
J=np.copy(xJ)
prevJ=np.copy(xJ)
estJ=np.zeros((n,1))
while True:
    cntJ+=1
    cntJ_index.append(cntJ)
    J,prevJ=Jacobi(A,b,J,prevJ,n)
    if np.allclose(J,prevJ,1e-100):
        xJ=J
        break
    estJ=np.hstack((estJ, J))
estJ=np.hstack((estJ, xJ))
```

solution을 구하기 위해, 초기값들을 설정한 후 while문을 이용하여 반복시켰다. cntJ는 반복문 시행 횟수를, cntJ_index는 이 cntJ 값들을 저장한 list, **J**와 **prevJ**는 Jacobi 함수에 대입시킬 벡터, estJ는 매 반복마다 도출되는 값을 저장할 변수이다. while문 내부에선, 반복당 cntJ값을 증가시키고 이를 cntJ_index에 저장한다. **J, prevJ**를 입력 받아 Jacobi함수를 수행한 값을 **J, prevJ**에 다시 저장해준다. numpy의 allclose함수를 활용하여 **J**와 **prevJ**를 비교하여, 오차범위 10^{-100}내에서 근사한다면 **xJ**에 **J**를 저장해주고 while문을 종료한다. 그렇지 않다면 estJ에 **J**를 이어 붙인다. while문이 종료되었다면, estJ에 **xJ**을 이어 붙인다. **xJ**에 solution이 저장되었다.

(2) Use Python to construct the Gauss-Seidel algorithm and find the solution.

```
###Gauss-Seidel###
def Gauss(A,b,xG,prevxG,n):
    temp=np.zeros((n,1))
    prevxG=np.copy(xG)
    for i in range(n):
        temp[i]=-(A[i,:i]@xG[:i]+A[i,i+1:]@prevxG[i+1:])
        temp[i]=(temp[i]+b[i])/A[i,i]
        xG=np.copy(temp)
    return xG,prevxG
```

A, **b, xG, prevxG,** n을 입력받아, Gauss-Seidel algorithm을 수행하는 함수를 먼저 정의하였다. 함수 내

부에서, n by 1인 벡터 **temp**를 임시로 정의해 주고, 입력 받은 **xG**을 복사하여 **prevxG**에 저장한다. 그 후 for문을 활용하여, 각 행의 성분에 대해서 $x_i = \frac{1}{a_{ii}}\left[-\sum_{j=1}^{i-1}(a_{ij}x_j) - \sum_{j=i+1}^{n}(a_{ij}x_j) + b_i\right]$를 수행한다. 그 후 **temp**의 값을 **xG**에 저장해주고, **xG**과 **prevxG**을 반환한다.

```
xG=np.zeros((n,1))
cntG=0
cntG_index=[0]
G=np.copy(xG)
prevG=np.copy(xG)
estG=np.zeros((n,1))
while True:
    cntG+=1
    cntG_index.append(cntG)
    G,prevG=Gauss(A,b,G,prevG,n)
    if np.allclose(G,prevG,1e-100):
        xG=G
        break
    estG=np.hstack((estG, G))
estG=np.hstack((estG, xG))
```

solution을 구하기 위해, 초기값들을 설정한 후 while문을 이용하여 반복시켰다. cntG는 반복문 시행 횟수를, cntG_index는 이 cntG 값들을 저장한 list, **G**와 **prevG**는 Gauss-Seidel 함수에 대입시킬 벡터, estG는 매 반복마다 도출되는 값을 저장할 변수이다. while문 내부에선, 반복당 cntG값을 증가시키고 이를 cntG_index에 저장한다. **G, prevG**를 입력 받아 Gauss-Seidel함수를 수행한 값을 **G, prevG**에 다시 저장해준다. numpy의 allclose함수를 활용하여 **G**와 **prevG**를 비교하여, 오차범위 10^{-100}내에서 근사한다면 **xG**에 **G**를 저장해주고 while문을 종료한다. 그렇지 않다면 estG에 **G**를 이어 붙인다. while문이 종료되었다면, estG에 **xG**을 이어 붙인다. **xG**에 solution이 저장되었다.

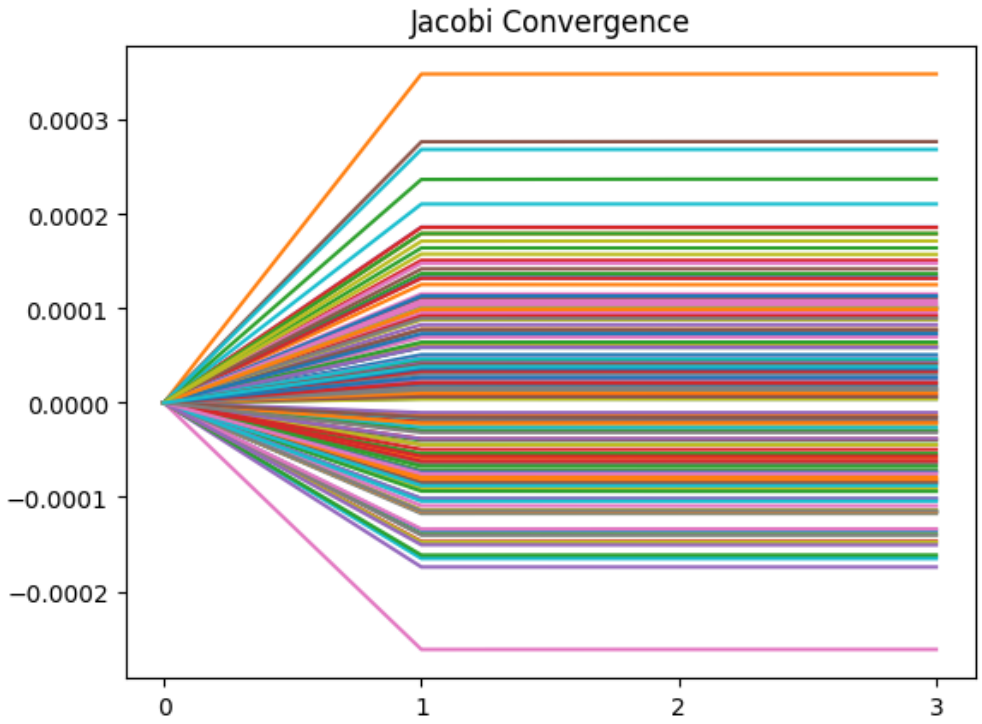
(3) For both algorithms, obtain the convergence plot of the solution.

(3)-(a) Jacobi algorithm

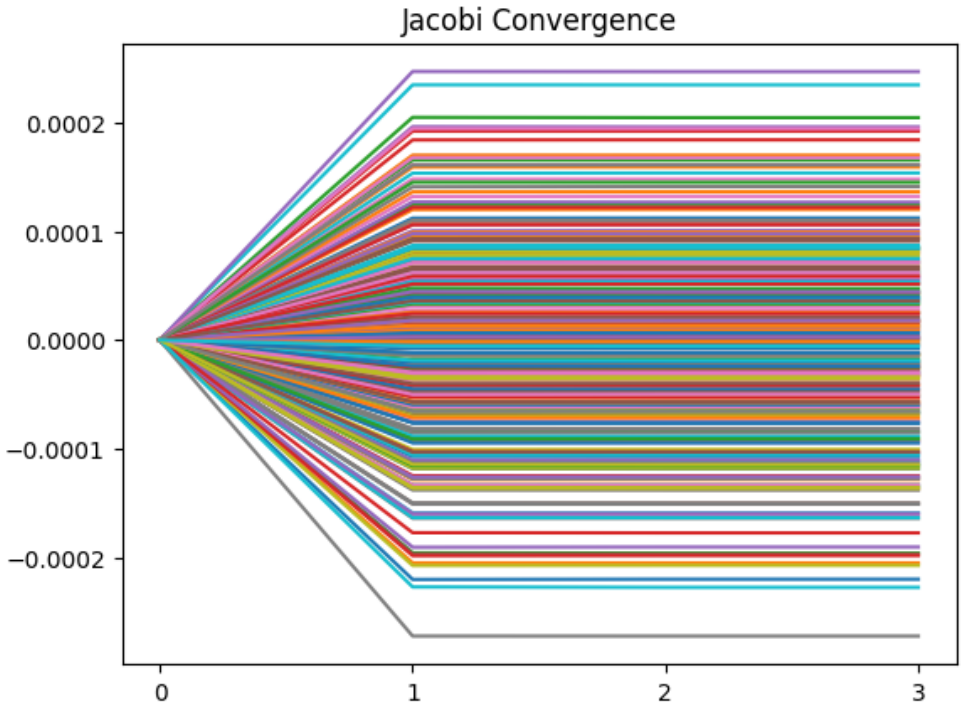
```
###Plot the Jacobi Convergence###
plt.figure
for i in range(n):
    plt.plot(cntJ_index,estJ[i,:])
plt.title("Jacobi Convergence")
plt.xticks(np.arange(0, cntJ+1))
plt.show()
```

figure를 생성한 후, for 문을 이용하여 차례대로 n회 반복하여, cntJ_index를 x축으로, estJ의 각 행을 y축으로 하여 plot한다. title은 "Jacobi Convergence", x축은 0부터 1씩 증가하며 cntJ까지 표기 시켜준다. plot을 수행한다. plot 결과는 다음과 같다.

① n=150



② n=250



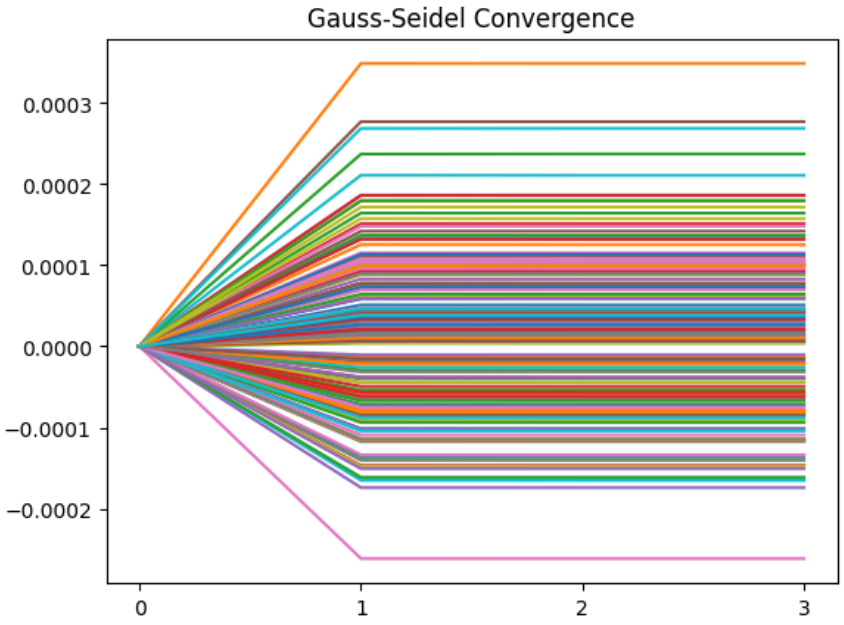
(3)-(b) Gauss-Seidel algorithm

```
###Plot the Gauss-Seidel Convergence###
plt.figure
```

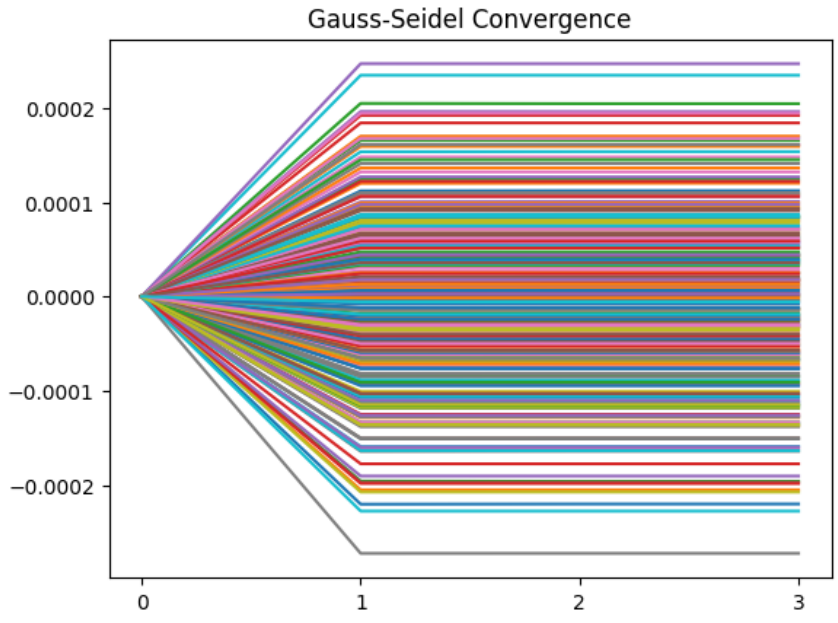
```
for i in range(n):
    plt.plot(cntG_index,estG[i,:])
plt.title("Gauss-Seidel Convergence")
plt.xticks(np.arange(0, cntG+1))
plt.show()
```

figure를 생성한 후, for 문을 이용하여 차례대로 n회 반복하여, cntG_index를 x축으로, estG의 각 행을 y축으로 하여 plot한다. title은 "Gauss-Seidel Convergence", x축은 0부터 1씩 증가하며 cntG까지 표기한다. plot을 수행한다. plot 결과는 다음과 같다.

① n=150



② n=250



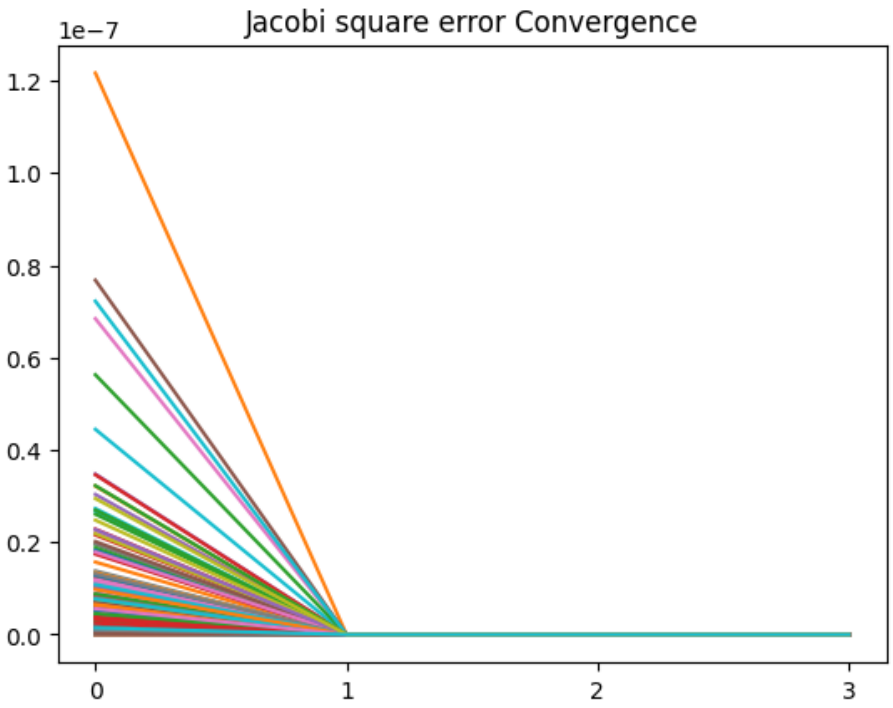
(4) Plot $\|x - x_k\|^2$ for each k where x is the true solution(by using numpy.linalg.solve) and x_k is the solutions obtained from the two algorithms in each steps

(4)-(a) Jacobi algorithm

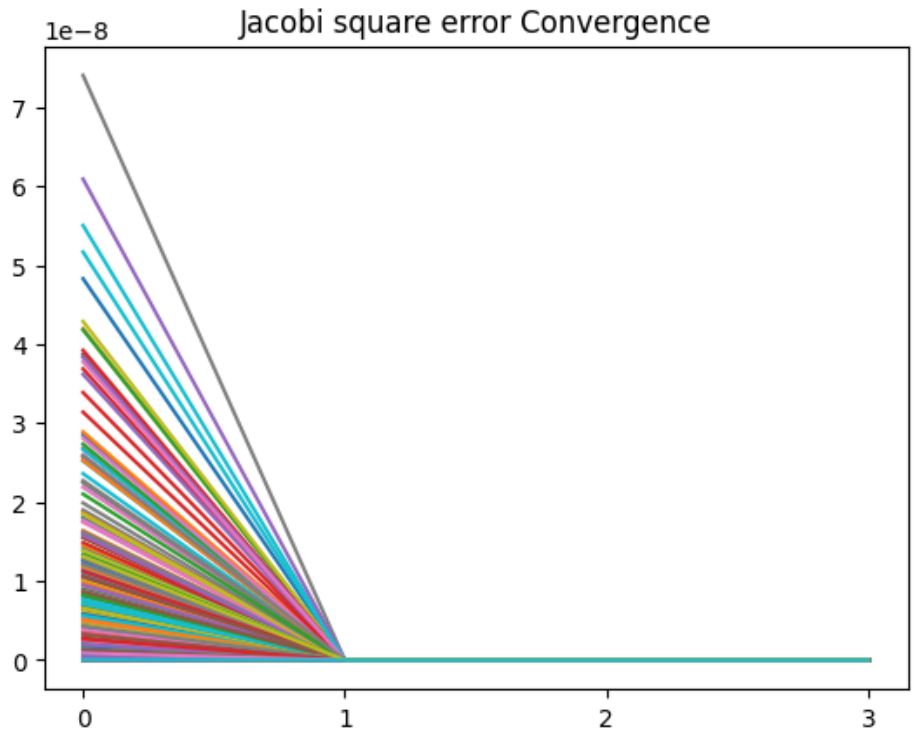
```
###Plot square error for Jacobi###
estSEJ=(solve-estJ[:,0].reshape(-1,1))**2
for i in range(1,len(estJ[0,:])):
    estSEJ=np.hstack((estSEJ,(solve-estJ[:,i].reshape(-1,1))**2))
plt.figure
for i in range(n):
    plt.plot(cntJ_index,estSEJ[i,:])
plt.title("Jacobi square error Convergence")
plt.xticks(np.arange(0, cntJ+1))
plt.show()
```

estSEJ에 첫 $\|x - x_k\|^2$ 값을 저장해준다. 이 후 for문을 활용하여 square error 값들을 estSEJ에 이어붙인다. figure를 생성하고, for문을 활용하여 n번 반복하여 cntJ_index를 x축, estSEJ의 한 행을 y축으로 하여 plot한다. 제목을 Jacobi square error Convergence로 설정하고, x축을 0부터 cntJ까지 표시해준다. plot을 수행한다. plot 결과는 다음과 같다.

① n=150



② n=250

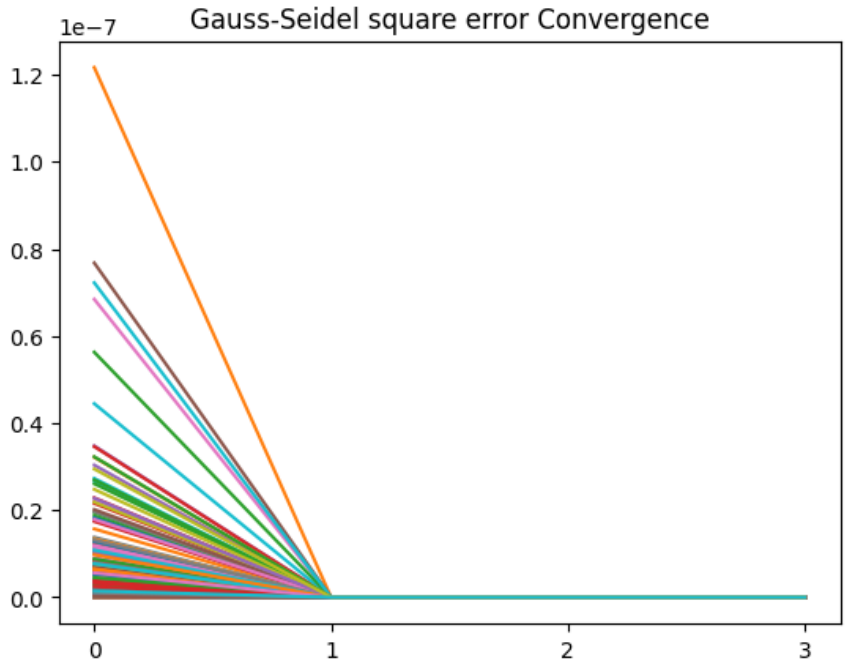


(4)-(b) Gauss-Seidel algorithm

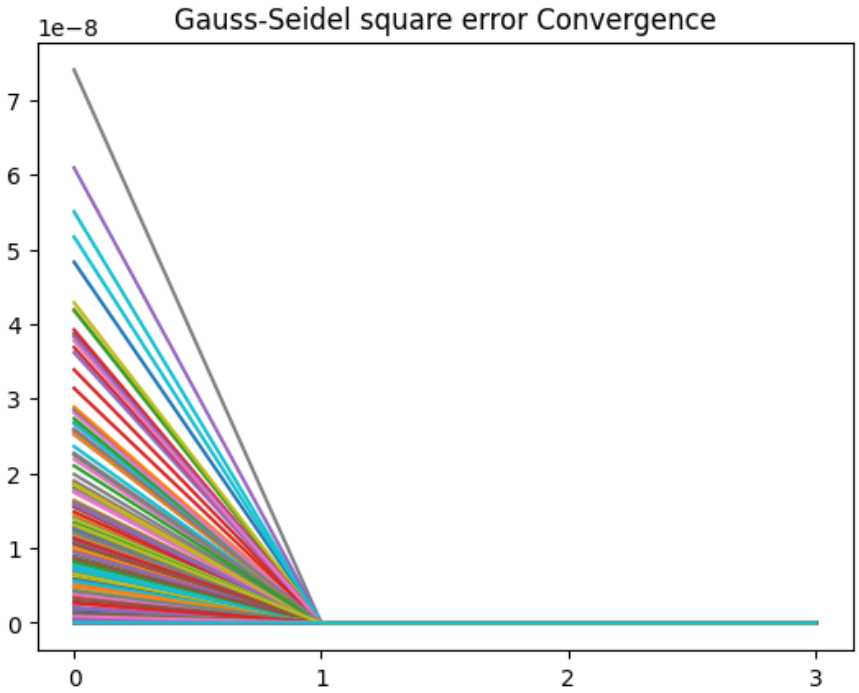
```
###Plot square error for Gauss-Seidel###
estSEG=(solve-estG[:,0].reshape(-1,1))**2
for i in range(1,len(estG[0,:])):
    estSEG=np.hstack((estSEG,(solve-estG[:,i].reshape(-1,1))**2))
plt.figure
for i in range(n):
    plt.plot(cntG_index,estSEG[i,:])
plt.title("Gauss-Seidel square error Convergence")
plt.xticks(np.arange(0, cntG+1))
plt.show()
```

estSEG에 첫 $\|x - x_k\|^2$ 값을 저장해준다. 이 후 for문을 활용하여 square error 값들을 estSEG에 이어붙인다. figure를 생성하고, for문을 활용하여 n번 반복하여 cntG_index를 x축, estSEG의 한 행을 y축으로 하여 plot한다. 제목을 Gauss-Seidel square error Convergence로 설정하고, x축을 0부터 cntG까지 표시해준다. plot을 수행한다. plot 결과는 다음과 같다.

① n=150



② n=250



각 시행 간의 오차가 크지 않아 잘 알아볼 수는 없지만, 대개 Gauss-Seidel Algorithm의 수렴이 Jacobi Algorithm보다 빠른 것으로 알려져 있다.

또한 부록에 np.allclose가 아닌 np.array_equal을 사용하여 Problem 2의 각 method를 수행한 결과를 첨부하였다.

Problem 3

전체 코드

```
###Setting up###

import numpy as np                                          #numpy np라는 이름으로 import
import time                                                 #time import
n=int(input())                                              #n 입력
A = np.random.normal(0, 1, size=(n, n)) + 10000*np.identity(n)  #matrix A 설정
b = np.random.normal(0, 1, size=(n, 1))                     #matrix b 설정
U=np.copy(A)                                                #A를 복사해 U에 저장
L=np.eye(n)                                                 #L을 n by n 행렬로 설정
bb = np.copy(b)                                             #b를 복사해 bb에 저장

###LU Decomposition###
for i in range(n):                      #for문 이용 n회 반복
    for j in range(i+1,n):              #for문 이용 j가 i+1에서 n-1 될 때까지 반복
        p=U[j,i]/U[i,i]                 #pivot 설정
        L[j,i]=p                        #L[j,i]에 pivot 기록
        U[j,:]-=p*U[i,:]                #행 연산 수행
        bb[j]-=p*bb[i]                  #행 연산 수행

###Jacobi###
def Jacobi(A,b,xJ,prevxJ,n):                        #A,b,xJ,prevxJ,n을 입력받는 함수 Jacobi 정의
    temp=np.zeros((n,1))                            #temp는 n by 1 vector
    prevxJ=np.copy(xJ)                              #xJ를 복사해 prevxJ에 저장
    for i in range(n):                              #for문 활용 n회 반복
        temp[i]=-(A[i,:i]@xJ[:i]+A[i,i+1:]@xJ[i+1:])#Jacobi 연산 수행한 값 temp[i]에 저장
        temp[i]=(temp[i]+b[i])/A[i,i]               #Jacobi 연산 수행한 값 temp[i]에 저장
    xJ=np.copy(temp)                                #xJ에 temp 복사해 저장
    return xJ,prevxJ                                #xJ, prevxJ 반환

###Gauss-Seidel###
def Gauss(A,b,xG,prevxG,n):                             #A,b,xG,prevxG,n을 입력받는 함수 Gauss 정의
    temp=np.zeros((n,1))                                #temp는 n by 1 vector
    prevxG=np.copy(xG)                                  #xG를 복사해 prevxG에 저장
    for i in range(n):                                  #for문 활용 n회 반복
        temp[i]=-(A[i,:i]@xG[:i]+A[i,i+1:]@prevxG[i+1:])#Gauss-Seidel 연산 수행한 값 temp[i]에 저장
        temp[i]=(temp[i]+b[i])/A[i,i]                   #Gauss-Seidel 연산 수행한 값 temp[i]에 저장
        xG=np.copy(temp)                                #xG에 temp 복사해 저장
    return xG,prevxG                                    #xG, prevxG 반환

###Vectors###
x1=np.zeros((n,1))              #x1은 n by 1 vector
x2=np.zeros((n,1))              #x2는 n by 1 vector
xJ=np.zeros((n,1))              #xJ는 n by 1 vector
xG=np.zeros((n,1))              #xG는 n by 1 vector
J=np.copy(xJ)                   #xJ 복사하여 J에 저장
prevJ=np.copy(xJ)               #prevJ에 xJ 복사하여 저장
G=np.copy(xG)                   #G에 xG 복사하여 저장
prevG=np.copy(xG)               #prevG에 xG 복사하여 저장


###back substitution###
start_time1=time.time()                         #시작 시간 저장
x1=np.zeros((n,1))                              #x1 n by 1로 정의
x1[n-1]=bb[n-1]/U[n-1,n-1]                      #최초의 back substitution 수행
for i in range(n):                              #for문 이용 n회 반복
    k=n-1-i                                     #k에 n-1-i 저장
    x1[k]=(bb[k]-U[k,k+1:n]@x1[k+1:n])/U[k,k]   #x1[k]에 해당 연산 수행한 값 저장
print(f"Back substitution took {(time.time() - start_time1)*(10)**3} ms for computing")
                                                #소요 시간 출력

###LU Decomposition###
start_time2=time.time()                 #시작 시간 저장
invL=np.linalg.inv(L)                   #invL에 L의 inverse matrix 저장
invU=np.linalg.inv(U)                   #invU에 U의 inverse matrix 저장
y=invL@b                                #y=L^-1b 연산 수행
x2=invU@y                               #x2=U^-1y 연산 수행
print(f"LU Decomposition Algorithm took {(time.time() - start_time2)*(10)**3} ms for computing")
                                        #소요 시간 출력
```

```
###Jacobi###
start_time3=time.time()                 #시작 시간 저장
while True:                             #while문 무한 반복
    J,prevJ=Jacobi(A,b,J,prevJ,n)       #A,b,xJ,prevxJ,n을 입력받아 반환값을 J,prevJ에 저장
    if np.allclose(J,prevJ,1e-100):     #J와 prevJ의 오차 범위가 10^-100 이내라면
        xJ=J                            #J를 xJ에 저장
        break                           #while문 종료
print(f"Jacobi Algorithm took {(time.time() - start_time3)*(10)**3} ms for computing")
                                        #소요 시간 출력
###Gauss-Seidel###
start_time4=time.time()                 #시작 시간 저장
while True:                             #while문 무한 반복
    G,prevG=Gauss(A,b,G,prevG,n)        #A,b,xG,prevxG,n을 입력받아 반환값을 G,prevG에 저장
    if np.allclose(G,prevG,1e-100):     #G와 prevG의 오차 범위가 10^-100 이내라면
        xG=G                            #G를 xG에 저장
        break                           #while문 종료
print(f"Gauss-Sediel Algorithm took {(time.time() - start_time4)*(10)**3} ms for computing")
                                        #소요 시간 출력
###Numpy###
start_time5=time.time()                 #시작 시간 저장
solve=np.linalg.solve(A,b)              #선형방정식의 solution 저장
print(f"numpy.linalg.solve took {(time.time() - start_time5)*(10)**3} ms for computing")
                                        #소요 시간 출력
```

```
###Check Accuracy###
def accuracy(x,solution):                   #x, solution을 입력 받는 함수 선언
    i=1                                     #변수 i 선언
    while True:                             #while문 무한 반복
        error=10**(-i)                      #error는 10^(-i)
        if np.array_equal(x,solution):      #x와 solution이 같다면 i=-1 저장하고 while문 종료
            i=-1
            break
        if np.allclose(x,solution,error):   #x와 solution이 오차범위 error 내에서 근사하면 i에 1 더함
            i+=1
        else:                               #범위 내에서 근사하지 않는다면 while문 종료
            break
        if i>=10000:                        #i가 10000보다 크거나 같으면 while문 종료
            break
    return i                                #i 반환
acc1=accuracy(x1,solve)                                 #acc1에 x1,solve로 함수 수행한 값 저장
acc2=accuracy(x2,solve)                                 #acc2에 x2,solve로 함수 수행한 값 저장
accJ=accuracy(xJ,solve)                                 #accJ에 xJ,solve로 함수 수행한 값 저장
accG=accuracy(xG,solve)                                 #accG에 xG,solve로 함수 수행한 값 저장
print(f"Back substitution's Accuracy point is {acc1}")  #해당 method의 Accuracy point 값 출력
print(f"LU Decomposition's Accuracy point is {acc2}")   #해당 method의 Accuracy point 값 출력
print(f"Jacobi Algorithm's Accuracy point is {accJ}")   #해당 method의 Accuracy point 값 출력
print(f"Gauss-Sediel Algorithm's Accuracy point is {accG}") #해당 method의 Accuracy point 값 출력
```

알고리즘 설명

초기값 설정

```
###Setting up###
import numpy as np
import time
n=int(input())
A = np.random.normal(0, 1, size=(n, n)) + 10000*np.identity(n)
b = np.random.normal(0, 1, size=(n, 1))
U=np.copy(A)
L=np.eye(n)
bb = np.copy(b)

###LU Decomposition###
for i in range(n):
    for j in range(i+1,n):
```

```
        p=U[j,i]/U[i,i]
        L[j,i]=p
        U[j,:]-=p*U[i,:]
        bb[j]-=p*bb[i]

###Jacobi###
def Jacobi(A,b,xJ,prevxJ,n):
    temp=np.zeros((n,1))
    prevxJ=np.copy(xJ)
    for i in range(n):
        temp[i]=-(A[i,:i]@xJ[:i]+A[i,i+1:]@xJ[i+1:])
        temp[i]=(temp[i]+b[i])/A[i,i]
    xJ=np.copy(temp)
    return xJ,prevxJ

###Gauss-Seidel###
def Gauss(A,b,xG,prevxG,n):
    temp=np.zeros((n,1))
    prevxG=np.copy(xG)
    for i in range(n):
        temp[i]=-(A[i,:i]@xG[:i]+A[i,i+1:]@prevxG[i+1:])
        temp[i]=(temp[i]+b[i])/A[i,i]
        xG=np.copy(temp)
    return xG,prevxG

###Vectors###
x1=np.zeros((n,1))
x2=np.zeros((n,1))
xJ=np.zeros((n,1))
xG=np.zeros((n,1))
J=np.copy(xJ)
prevJ=np.copy(xJ)
G=np.copy(xG)
prevG=np.copy(xG)
```

각 계산을 위한 알고리즘과 초기값들을 설정해주었다. 시간 측정을 위해 time을 import 했다.

Compare the computation time and accuracy of

- Upper triangular matrix and back - substitution algorithm

- LU decomposition algorithm

- Jacobi algorithm

- Gauss – Seidel algorithm

- numpy.linalg.solve in Python

(1) Computation time

```
###back substitution###
start_time1=time.time()
x1=np.zeros((n,1))
x1[n-1]=bb[n-1]/U[n-1,n-1]
for i in range(n):
    k=n-1-i
    x1[k]=(bb[k]-U[k,k+1:n]@x1[k+1:n])/U[k,k]
print(f"Back substitution took {(time.time() - start_time1)*(10)**3} ms for computing")

###LU Decomposition###
start_time2=time.time()
invL=np.linalg.inv(L)
invU=np.linalg.inv(U)
y=invL@b
x2=invU@y
print(f"LU Decomposition Algorithm took {(time.time() - start_time2)*(10)**3} ms for computing")

###Jacobi###
start_time3=time.time()
while True:
    J,prevJ=Jacobi(A,b,J,prevJ,n)
    if np.allclose(J,prevJ,1e-100):
        xJ=J
        break
print(f"Jacobi Algorithm took {(time.time() - start_time3)*(10)**3} ms for computing")

###Gauss-Seidel###
start_time4=time.time()
while True:
    G,prevG=Gauss(A,b,G,prevG,n)
    if np.allclose(G,prevG,1e-100):
        xG=G
        break
print(f"Gauss-Sediel Algorithm took {(time.time() - start_time4)*(10)**3} ms for computing")

###Numpy###
start_time5=time.time()
solve=np.linalg.solve(A,b)
print(f"numpy.linalg.solve took {(time.time() - start_time5)*(10)**3} ms for computing")
```

Back substitution과 LU decomposition은 L과 U, **bb**를 이미 구한 상태에서 solution을 구하는 계산을 수행하는 시간만을 측정하였다. 알고리즘 수행 전의 시간 정보는 각각의 start_time 변수에 저장된다. 출력된 결과는 다음과 같다.

① n=150

```
Back substitution took 1.0051727294921875 ms for computing
LU Decomposition Algorithm took 18.941640853881836 ms for computing
Jacobi Algorithm took 2.989530563354492 ms for computing
Gauss-Sediel Algorithm took 4.321575164794922 ms for computing
numpy.linalg.solve took 1.9927024841308594 ms for computing
```

② n=250

```
Back substitution took 0.9946823120117188 ms for computing
LU Decomposition Algorithm took 11.83319091796875 ms for computing
Jacobi Algorithm took 4.991292953491211 ms for computing
Gauss-Sediel Algorithm took 5.415439605712891 ms for computing
numpy.linalg.solve took 4.150390625 ms for computing
```

대체로 Back substitution이 가장 빨랐고 Jacobi와 Gauss-seidel은 비슷한 속도를 보였다. LU decomposition을 통해 구하는 시간이 대체적으로 가장 느리게 나오는 경우가 많았는데, 이는 inverse matrix를 사용하기 때문인 것으로 보인다. 단, 직접 작성한 L과 U를 구하는 알고리즘을 Back substitution의 계산시간 계산에 포함시켰을 경우에는 계산 시간이 오래 걸렸는데, 이는 해당 알고리즘을 이중 for문으로 작성하였기 때문으로 보인다.

(2) Accuracy

다음과 같은 정확도 검사 함수를 작성하였다.

```
###Check Accuracy###
def accuracy(x,solution):
    i=1
    while True:
        error=10**(-i)
        if np.array_equal(x,solution):
            i=-1
            break
        if np.allclose(x,solution,error):
            i+=1
        else:
            break
        if i>=10000:
            break
    return i
acc1=accuracy(x1,solve)
acc2=accuracy(x2,solve)
accJ=accuracy(xJ,solve)
accG=accuracy(xG,solve)
print(f"Back substitution's Accuracy point is {acc1}")
print(f"LU Decomposition's Accuracy point is {acc2}")
print(f"Jacobi Algorithm's Accuracy point is {accJ}")
print(f"Gauss-Sediel Algorithm's Accuracy point is {accG}")
```

x와 **solution**을 입력 받아 while문을 수행한다. numpy의 array_equal함수를 이용하여, **x**와 **solution**이 같다면 i에 -1을 할당하고 즉시 반복문을 종료한다. 그렇지 않다면 allclose 함수를 활용하여 error 값 만큼의 오차 범위 내에서 해당 값이 근사하는지 확인한다. 만약 그렇다면 i값을 1 증가시킨다. i는 증가하는 연산만 존재하므로 i가 -1이 나올 경우는 무조건 **x**와 **solution**이 같은 경우 밖에 없다. 무한 반복을 막기 위해서, i의 boundary는 10000으로 설정하였다. 이 함수는 i 값을 반환한다. i 값을 Accuracy point라는 이름으로 명명하였다. 출력 결과는 다음과 같다.

① n=150

```
Back substitution's Accuracy point is 10000
LU Decomposition's Accuracy point is 10000
Jacobi Algorithm's Accuracy point is 10000
Gauss-Sediel Algorithm's Accuracy point is 10000
```

② n=250

```
Back substitution's Accuracy point is 10000
LU Decomposition's Accuracy point is 10000
Jacobi Algorithm's Accuracy point is 10000
Gauss-Sediel Algorithm's Accuracy point is 10000
```

비록 완전히 같지는 않지만, 각 algorithm으로 구한 solution들과 true solution은 10^{-10000}의 오차범위 이내에서 근사한다. 이렇게만 보면 Accuracy point가 -1인 경우가 절대 안 나올 것으로 보이기도 하지만, n이 2나 3처럼 아주 작으면 -1이 검출되는 경우도 있었다.

Problem 4

전체 코드

```
###Setting up###

import numpy as np                                                  #numpy np라는 이름으로 import
n=int(input())                                                      #n 입력
A = np.random.normal(0, 1, size=(n, n)) + 0.0001*np.identity(n) #matrix A 설정
b = np.random.normal(0, 1, size=(n, 1))                             #matrix b 설정
solve=np.linalg.solve(A,b)                                          #선형방정식 solution solve에 저장
eig,eigv=np.linalg.eig(A)                                           #eig,eigv에 각각 A의 eigenvalue,eigenvector 저장

###To justify###
for i in range(len(eig)):       #eig의 길이만큼 반복
    if np.abs(eig[i])>=1:       #eig[i]의 절댓값이 1보다 크거나 같다면 not converge 출력
        print("not converge")

###Jacobi###
def Jacobi(A,b,xJ,prevxJ,n):                        #A,b,xJ,prevxJ,n을 입력받는 함수 Jacobi 정의
    temp=np.zeros((n,1))                            #temp는 n by 1 vector
    prevxJ=np.copy(xJ)                              #xJ를 복사해 prevxJ에 저장
    for i in range(n):                              #for문 활용 n회 반복
        temp[i]=-(A[i,:i]@xJ[:i]+A[i,i+1:]@xJ[i+1:])#Jacobi 연산 수행한 값 temp[i]에 저장
        temp[i]=(temp[i]+b[i])/A[i,i]               #Jacobi 연산 수행한 값 temp[i]에 저장
    xJ=np.copy(temp)                                #xJ에 temp 복사해 저장
    return xJ,prevxJ                                #xJ, prevxJ 반환
xJ=np.zeros((n,1))                      #xJ는 n by 1 vector
J=np.copy(xJ)                           #J는 n by 1 vector
prevJ=np.copy(xJ)                       #prevJ는 n by 1 vector
while True:                             #while문 무한 반복
    J,prevJ=Jacobi(A,b,J,prevJ,n)       #A,b,xJ,prevxJ,n을 입력받아 반환값을 J,prevJ에 저장
    if np.allclose(J,prevJ,1e-100):     #J와 prevJ의 오차 범위가 10^-100 이내라면
        xJ=J                            #J를 xJ에 저장
        break                           #while문 종료
```

```
###Gauss-Seidel###
def Gauss(A,b,xG,prevxG,n):                             #A,b,xG,prevxG,n을 입력받는 함수 Gauss 정의
    temp=np.zeros((n,1))                                #temp는 n by 1 vector
    prevxG=np.copy(xG)                                  #xG를 복사해 prevxG에 저장
    for i in range(n):                                  #for문 활용 n회 반복
        temp[i]=-(A[i,:i]@xG[:i]+A[i,i+1:]@prevxG[i+1:])#Gauss-Seidel 연산 수행한 값 temp[i]에 저장
        temp[i]=(temp[i]+b[i])/A[i,i]                   #Gauss-Seidel 연산 수행한 값 temp[i]에 저장
        xG=np.copy(temp)                                #xG에 temp 복사해 저장
    return xG,prevxG                                    #xG, prevxG 반환
xG=np.zeros((n,1))                      #xG는 n by 1 vector
G=np.copy(xG)                           #G는 n by 1 vector
prevG=np.copy(xG)                       #prevG는 n by 1 vector
while True:                             #while문 무한 반복
    cntG+=1                             #cntG 1 증가
    cntG_index.append(cntG)             #cntG 값을 cntG_index에 추가
    G,prevG=Gauss(A,b,G,prevG,n)        #A,b,xG,prevxG,n을 입력받아 반환값을 G,prevG에 저장
    if np.allclose(G,prevG,1e-100):     #G와 prevG의 오차 범위가 10^-100 이내라면
        xG=G                            #G를 xG에 저장
        break                           #while문 종료
```

알고리즘 설명

초기 설정

```
###Setting up###
import numpy as np
from matplotlib import pyplot as plt
n=int(input())
A = np.random.normal(0, 1, size=(n, n)) + 0.0001*np.identity(n)
b = np.random.normal(0, 1, size=(n, 1))
solve=np.linalg.solve(A,b)
```

numpy를 np라는 이름으로 import, matplotlib의 pyplot을 plt라는 이름으로 import하고 n을 입력 받아, A와 **b**를 설정한다. solve에는 $A\mathbf{x} = \mathbf{b}$의 solution을 저장한다.

(1) Use Python to construct the Jacobi algorithm and find the solution

```
###Jacobi###
def Jacobi(A,b,xJ,prevxJ,n):
    temp=np.zeros((n,1))
    prevxJ=np.copy(xJ)
    for i in range(n):
        temp[i]=-(A[i,:i]@xJ[:i]+A[i,i+1:]@xJ[i+1:])
        temp[i]=(temp[i]+b[i])/A[i,i]
    xJ=np.copy(temp)
    return xJ,prevxJ
xJ=np.zeros((n,1))
J=np.copy(xJ)
prevJ=np.copy(xJ)
while True:
    J,prevJ=Jacobi(A,b,J,prevJ,n)
    if np.allclose(J,prevJ,1e-100):
        xJ=J
        break
```

A, **b, xJ, prevxJ,** n을 입력 받아, Jacobi algorithm을 수행하는 함수를 정의하였다. 그 후 while문을 반복하여 true solution으로 수렴시켰다. (Problem 2와 동일)

(2) Use Python to construct the Gauss-Seidel algorithm and find the solution.

```
###Gauss-Seidel###
def Gauss(A,b,xG,prevxG,n):
    temp=np.zeros((n,1))
    prevxG=np.copy(xG)
    for i in range(n):
        temp[i]=-(A[i,:i]@xG[:i]+A[i,i+1:]@prevxG[i+1:])
        temp[i]=(temp[i]+b[i])/A[i,i]
        xG=np.copy(temp)
    return xG,prevxG
xG=np.zeros((n,1))
G=np.copy(xG)
prevG=np.copy(xG)
while True:
    G,prevG=Gauss(A,b,G,prevG,n)
    if np.allclose(G,prevG,1e-100):
        xG=G
        break
```

A, **b, xG, prevxG,** n을 입력 받아, Gauss-Seidel algorithm을 수행하는 함수를 정의하였다. 그 후 while문을 반복하여 true solution으로 수렴시켰다. (Problem 2와 동일)

(3) If the solution cannot be found using the two algorithms, why? Justify your answer.

코드를 실행하면, 다음과 같은 메시지가 표시된다.

```
###Jacobi###
C:\Users\Jonghyuk\AppData\Local\Temp\ipykernel_6356\4815892.py:15:
RuntimeWarning: overflow encountered in divide
  temp[i]=(temp[i]+b[i])/A[i,i]
C:\Users\Jonghyuk\AppData\Local\Temp\ipykernel_6356\4815892.py:14:
RuntimeWarning: overflow encountered in matmul
  temp[i]=-(A[i,:i]@xJ[:i]+A[i,i+1:]@xJ[i+1:])
C:\Users\Jonghyuk\AppData\Local\Temp\ipykernel_6356\4815892.py:14:
RuntimeWarning: invalid value encountered in matmul
  temp[i]=-(A[i,:i]@xJ[:i]+A[i,i+1:]@xJ[i+1:])
C:\Users\Jonghyuk\AppData\Local\Temp\ipykernel_6356\4815892.py:14:
RuntimeWarning: invalid value encountered in add
  temp[i]=-(A[i,:i]@xJ[:i]+A[i,i+1:]@xJ[i+1:])
###Guasss-Sediel###
C:\Users\Jonghyuk\AppData\Local\Temp\ipykernel_6356\2804751892.py:33:
RuntimeWarning: overflow encountered in divide
  temp[i]=(temp[i]+b[i])/A[i,i]
C:\Users\Jonghyuk\AppData\Local\Temp\ipykernel_6356\2804751892.py:32:
RuntimeWarning: invalid value encountered in matmul
    temp[i]=-(A[i,:i]@xG[:i]+A[i,i+1:]@prevxG[i+1:])
```

오류 메시지를 확인해보면, divide연산, 행렬 곱 연산에서 overflow가 발생했음을 의미하는 RuntimeWarning 메시지와 행렬 곱과 덧셈 연산에서 유효하지 않은 값이 존재하여 연산을 수행할 수 없음을 알리는 RuntimeWarning 메시지임을 알 수 있다. 이는 A가 대각지배행렬이 아니기 때문에 발생하는 오류라고도 볼 수 있다. Note 6를 참고하면 A가 대각지배행렬일 경우 $A\mathbf{x}=\mathbf{b}$가 unique solution을 가진다고 설명하는데, 이 경우 A의 eigenvalue의 절댓값은 1보다 작다. 이를 확인하기 위해 다음과 같은 코드를 작성한다.

```
eig,eigv=np.linalg.eig(A)
for i in range(len(eig)):
    if np.abs(eig[i])>=1:
        print("not converge")
```

A의 eigenvalue를 eig에, eigenvector를 eigv에 저장하고, for문을 활용하여 eigenvalue의 절댓값이 1보다 크거나 같으면 "not converge"라는 메시지를 출력하도록 하였다. 결과는 다음과 같았다.

```
not converge
not converge
not converge
not converge
...
```

따라서 A가 대각지배행렬(Strictly diagonally dominant matrix)가 아니기 때문에, 해를 찾을 수 없다.

부록

1. Permutation Steps

Permutation Step 과정의 코드를 따로 작성하였다. Permutation을 거칠 경우에는 Problem 1의 code는 다음과 같이 변경된다.

```
###Setting up###
import numpy as np
n=int(input())
A = np.random.normal(0, 1, size=(n, n)) + 10000*np.identity(n)
b = np.random.normal(0, 1, size=(n, 1))
U=np.copy(A)
L=np.eye(n)
P=np.eye(n)
bb = np.copy(b)
```

Permutation matrix P를 추가로 설정해준다.

```
###row_exchange###
def row_exchange(A,B,C,i,j):
    temp1 = np.copy(A[i, :])
    A[i, :] = np.copy(A[j, :])
    A[j, :] = np.copy(temp1)
    temp2 = np.copy(B[i, :])
    B[i, :] = np.copy(B[j, :])
    B[j, :] = np.copy(temp2)
    temp3 = np.copy(C[i, :])
    C[i, :] = np.copy(C[j, :])
    C[j, :] = np.copy(temp3)
    return A,B,C
```

행을 바꾸어 주는 함수를 새로 정의해 주었다.

```
###PA=LU decomposition###
for i in range(n):
    if U[i,i]==0:
        k=np.argmax(abs(U[i:n,i]))+i
        U,P,bb=row_exchange(U,P,bb,i,k)
    for j in range(i+1,n):
        L[j,i]=U[j,i]/U[i,i]
        U[j,:]-=L[j,i]*U[i,:]
        bb[j]-=L[j,i]*bb[i]
```

PA=LU Decomposition을 진행해 주는 알고리즘이다. pivot으로 설정할 `U[i,i]=0`인 경우에 절댓값이 가장 큰 값의 인덱스를 기록하여 row exchange를 수행해준다. 그 외에는 LU와 동일하다.

```
###Back substitution###
x1=np.zeros((n,1))
x1[n-1]=bb[n-1]/U[n-1,n-1]
```

```
for i in range(n):
    k=n-1-i
    x1[k]=(bb[k]-U[k,k+1:n]@x1[k+1:n])/U[k,k]
```

Problem1과 차이는 없다.

```
###Solution By LU Decomposition###
invL=np.linalg.inv(L)
invU=np.linalg.inv(U)
invP=np.linalg.inv(P)
pb=P@b
y=invL@pb
x2=invU@y
```

$L\mathbf{y} = P\mathbf{b}, U\mathbf{x} = \mathbf{y}$의 두 가지 방정식으로 나누어 생각할 수 있다. 따라서 $\mathbf{y} = L^{-1}P\mathbf{b}$, $\mathbf{x} = U^{-1}\mathbf{y}$ 를 수행하면, solution을 구할 수 있다. 이 solution은 **x2**에 저장되었다.

```
###Check the uniquness of the solution###
rank=np.linalg.matrix_rank(A)
det=np.linalg.det(A)
eig,eigv=np.linalg.eig(A)
if rank!=n:print("not unique")
else:print("Unique, By rank")
if det==0:print("not unique")
else:print("Unique, By determinant")
if 0 in eig:print("not unique")
else:print("Unique, By eigenvalue")
```

Problem1과 동일하다.

```
###Verify###
solve=np.linalg.solve(A,b)
if np.allclose(x1,solve,1e-9):
    print(f"Back substitution converges to the solution in very small error : 10^-9")
if np.array_equal(x1,solve):
    print(f"Back substitution matched perfectly to the solution")
if np.allclose(x2,solve,1e-9):
    print(f"LU Decomposition converges to the solution in very small error : 10^-9")
if np.array_equal(x2,solve):
    print(f"LU Decomposition matched perfectly to the solution")
if np.allclose(P@A,L@U,1e-9):
    print(f"PA and LU converges in very small error : 10^-9")
if np.array_equal(P@A,L@U):
    print(f"PA and LU  matched perfectly")
```

PA와 LU가 같은지 확인하는 조건문을 추가하였다.

출력된 결과는 다음과 같다.

(1) n=150

```
Unique, By rank
Unique, By determinant
Unique, By eigenvalue
Back substitution converges to the solution in very small error : 10^-9
LU Decomposition converges to the solution in very small error : 10^-9
PA and LU converges in very small error : 10^-9
```

(2) n=250

```
Unique, By rank
Unique, By determinant
Unique, By eigenvalue
Back substitution converges to the solution in very small error : 10^-9
LU Decomposition converges to the solution in very small error : 10^-9
PA and LU converges in very small error : 10^-9
```

A는 애초에 Permutation을 하지 않아도 되는 matrix기 때문에, 결과가 크게 달라진 것은 없었다. PA=LU Decomposition은 정상적으로 이루어진다는 것을 확인할 수 있었다.

2. Expanded Convergence Plot

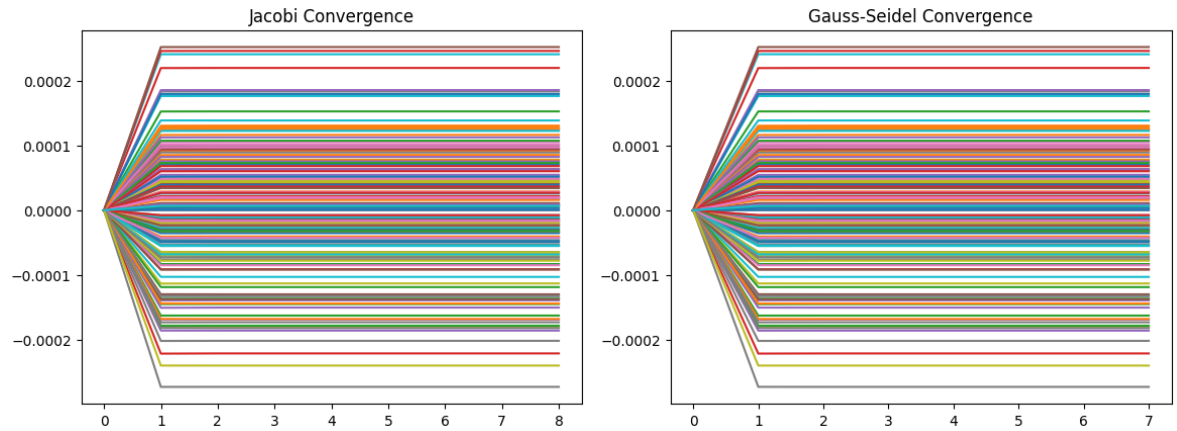
Problem2에서 Jacobi Algorithm과 Gauss-Seidel Algorithm의 수행 조건을 다음과 같이 변경하면, convergence 플롯의 x축 범위를 조금 더 늘려서 볼 수 있다.

`if np.allclose(G,prevG,1e-100):`	→	`if np.array_equal(G,prevG):`
`if np.allclose(G,prevG,1e-100):`		`if np.array_equal(G,prevG):`

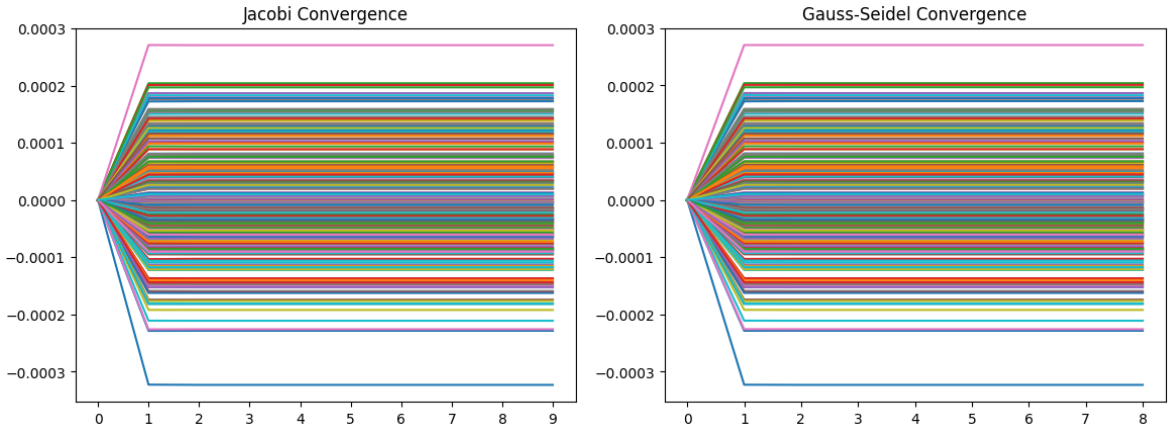
코드를 수정한 후 실행하여 보면 다음과 같은 결과를 볼 수 있다.

(1) Convergence Plot

n=150

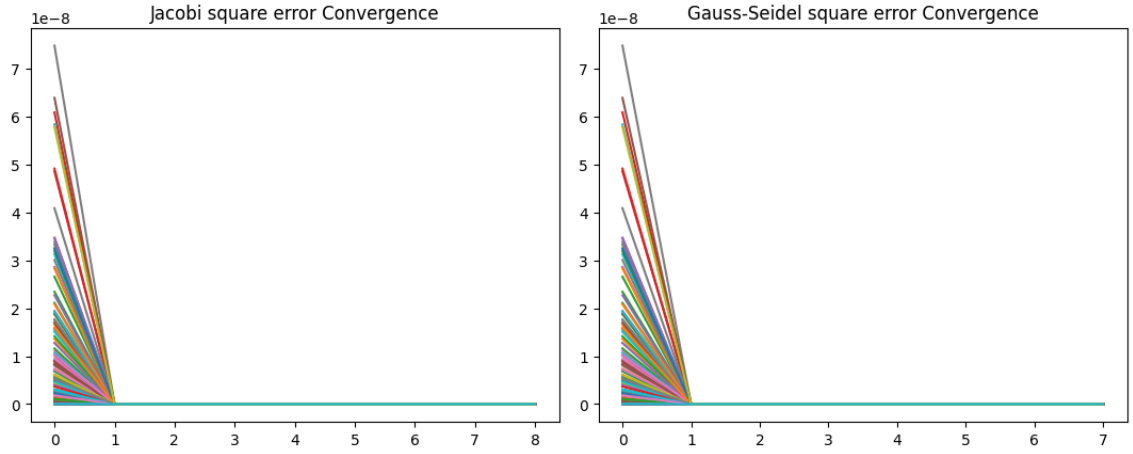


n=250



(2) Square Error Plot

n=150



n=250