CQF Exam 2 - Numerical Linear Algebra

October 5, 2020

[294]:
```
import numpy as np
from numpy.linalg import inv
```

[336]:
```
# Question 1, Strictly Diagonally Dominant Test.
# Accuracy is guaranteed, and this runs in a linear complexity depending on the␣
↪number of rows.
A = np.array([[0,3,-1,8],[-1,11,-1,3],[2,-1,10,-1],[10,-1,2,0]])
def isStrictlyDiagonallyDominant(A):
    for i, row in enumerate(A):
        x = sum(abs(m) for j, m in enumerate(row) if i != j)
        if x > abs(row[i]):
            return False
    return True

isStrictlyDiagonallyDominant(A)
```

[336]: False

[926]:
```
# Question 2, Doolittle's Method.
# We need to pivot the matrix to make sure it is decomposable. After that the␣
↪accuracy is guaranteed.
# Pivot complexity is in O(n^2)
# The Crout and Doolittle are just variants of each other, and have the same␣
↪complexity.
# If we have symmetric positive definite, we should use Cholesky's since it is␣
↪twice as fast.

A = np.array([[0,3,-1,8],[-1,11,-1,3],[2,-1,10,-1],[10,-1,2,0]])
def pivot(A):
    for i in range(0,3):
        if A[i,i] < np.max(A[i:4,i]):
            j = np.argmax(A[i:4,i])
            A[[i,j]]=A[[j,i]]
    return A
def Doolittle(A):
    A_p = pivot(A)
    L = np.zeros((4,4))
```

```python
    U = np.zeros((4,4))
    for i in range(0,4):
        L[i,i]=1
        for j in range(0,4):
            if i==0:
                continue
            if j>=i:
                continue
            sum2 = sum(L[i,k]*U[k,j] for k in range(0,4))
            L[i,j] = (A_p[i,j]-sum2)/U[j,j]
        for j in range(0,4):
            if j<i:
                continue
            sum1 = sum(L[i,k]*U[k,j] for k in range(0,4))
            U[i,j] = A_p[i,j] - sum1
    return L,U
L,U=Doolittle(A)
print(L)
print(U)
np.matmul(L,U)
b = np.array([6,25,-11,15])
def forward_subs(L,b):
    y=[]
    for i in range(len(b)):
        y.append(b[i])
        for j in range(i):
            y[i]=y[i]-(L[i,j]*y[j])
        y[i]=y[i]/L[i,i]
    return y
def back_subs(U,y):
    x=np.zeros_like(y)
    for i in range(len(x),0,-1):
        x[i-1]=(y[i-1]-np.dot(U[i-1,i:],x[i:]))/U[i-1,i-1]
    return x
def solve_system_LU(L,U,b):
    y=forward_subs(L,b)
    x=back_subs(U,y)
    return x
print(solve_system_LU(L,U,b))
```

```
[[ 1.          0.          0.          0.        ]
 [-0.1         1.          0.          0.        ]
 [ 0.2        -0.0733945   1.          0.        ]
 [ 0.          0.27522936 -0.08173077  1.        ]]
[[10.         -1.          2.          0.        ]
 [ 0.         10.9        -0.8         3.        ]
 [ 0.          0.          9.5412844  -0.77981651]
```

```
 [ 0.          0.          0.          7.11057692]]
[ 1.  2. -1.  1.]
```

[927]:
```python
# Question 2, Crout's Method.
def Crout(A):
    A_p = pivot(A)
    L = np.zeros((4,4))
    U = np.zeros((4,4))
    for i in range(0,4):
        U[i,i]=1
        for j in range(0,4):
            if j>i:
                continue
            sum2 = sum(L[i,k]*U[k,j] for k in range(0,4))
            L[i,j] = A_p[i,j] - sum2
        for j in range(0,4):
            if j<i:
                continue
            if i==j:
                continue
            sum1 = sum(L[i,k]*U[k,j] for k in range(0,4))
            U[i,j] = (A_p[i,j]-sum1) / L[i,i]
    return L,U
L,U=Crout(A)
print(L)
print(U)
print(np.matmul(L,U))
b = np.array([6,25,-11,15])
def forward_subs(L,b):
    y=[]
    for i in range(len(b)):
        y.append(b[i])
        for j in range(i):
            y[i]=y[i]-(L[i,j]*y[j])
        y[i]=y[i]/L[i,i]
    return y
def back_subs(U,y):
    x=np.zeros_like(y)
    for i in range(len(x),0,-1):
        x[i-1]=(y[i-1]-np.dot(U[i-1,i:],x[i:]))/U[i-1,i-1]
    return x
def solve_system_LU(L,U,b):
    y=forward_subs(L,b)
    x=back_subs(U,y)
    return x
print(solve_system_LU(L,U,b))
```

```
[[10.          0.          0.          0.        ]
```

```
 [-1.         10.9         0.          0.        ]
 [ 2.         -0.8         9.5412844   0.        ]
 [ 0.          3.         -0.77981651  7.11057692]]
[[ 1.         -0.1         0.2         0.        ]
 [ 0.          1.         -0.0733945   0.27522936]
 [ 0.          0.          1.         -0.08173077]
 [ 0.          0.          0.          1.        ]]
[[10. -1.  2.  0.]
 [-1. 11. -1.  3.]
 [ 2. -1. 10. -1.]
 [ 0.  3. -1.  8.]]
[ 1.  2. -1.  1.]
```

[1024]:

```
# Question 3, for x = [0,0,0,0]
# Results converge in this case. This is around twice as fast as Jacobi Method.
def seidel(A, x ,b):
    A_p = pivot(A)
    for j in range(0, 4):
        d = b[j]
        sum1 = sum(A_p[j,k] * x[k] for k in range(0,4)) - A_p[j,j]*x[j]
        x[j] = (d-sum1) / A_p[j][j]
    return x

x = [0, 0, 0, 0]
A = np.array([[0,3,-1,8],[-1,11,-1,3],[2,-1,10,-1],[10,-1,2,0]])
b = np.array([6,25,-11,15])
x_1 = [0,0,0,0]
for i in range(0,25):
    x = seidel(A,x,b)
    for j in range(0,4):
        x[j] = round(x[j], 4)
    print(x)
    if x == x_1:
        print(i)
        break
    else:
        x_1 = x.copy()
```

```
[0.6, 2.3273, -0.9873, 0.8789]
[1.0302, 2.0369, -1.0145, 0.9843]
[1.0066, 2.0036, -1.0025, 0.9983]
[1.0009, 2.0003, -1.0003, 0.9998]
[1.0001, 2.0, -1.0, 1.0]
[1.0, 2.0, -1.0, 1.0]
[1.0, 2.0, -1.0, 1.0]
6
```

[1061]:
```
# Question 3, for x = [1,1,1,1]
def seidel(A, x ,b):
    A_p = pivot(A)
    for j in range(0, 4):
        d = b[j]
        sum1 = sum(A_p[j,k] * x[k] for k in range(0,4)) - A_p[j,j]*x[j]
        x[j] = (d-sum1) / A_p[j][j]
    return x

x = [1,1,1,1]
A = np.array([[0,3,-1,8],[-1,11,-1,3],[2,-1,10,-1],[10,-1,2,0]])
b = np.array([6,25,-11,15])
x_1 = [0,0,0,0]
for i in range(0,25):
    x = seidel(A,x,b)
    for j in range(0,4):
        x[j] = round(x[j], 4)
    print(x)
    if x == x_1:
        print(i)
        break
    else:
        x_1 = x.copy()
```

```
[0.5, 2.1364, -0.8864, 0.9631]
[0.9909, 2.0196, -0.9999, 0.9927]
[1.0019, 2.0022, -1.0009, 0.9991]
[1.0004, 2.0002, -1.0002, 0.9999]
[1.0001, 2.0, -1.0, 1.0]
[1.0, 2.0, -1.0, 1.0]
[1.0, 2.0, -1.0, 1.0]
6
```

[1080]:
```
# Question 4, Successive Over-Relaxation
# We get a faster rate of convergence than from normal Gauss-seidel.
def seidel(A, x ,b):
    omega = 1.1
    A_p = pivot(A)
    for j in range(0, 4):
        x_current = x
        d = b[j]
        sum1 = sum(A_p[j,k] * x[k] for k in range(0,4)) - A_p[j,j]*x[j]
        x[j] = omega*((d -sum1) / A_p[j][j]) - (1-omega)*x_current[j]
        x_current += x_current + x[j]
    return x_current

x = [0, 0, 0, 0]
```

```
A = np.array([[0,3,-1,8],[-1,11,-1,3],[2,-1,10,-1],[10,-1,2,0]])
b = np.array([6,25,-11,15])
x_1 = [0,0,0,0]
seidel(A,x,b)
```

[1080]:
```
array([ 2.17649575,  5.98849575, -1.28938425,  2.56948725])
```