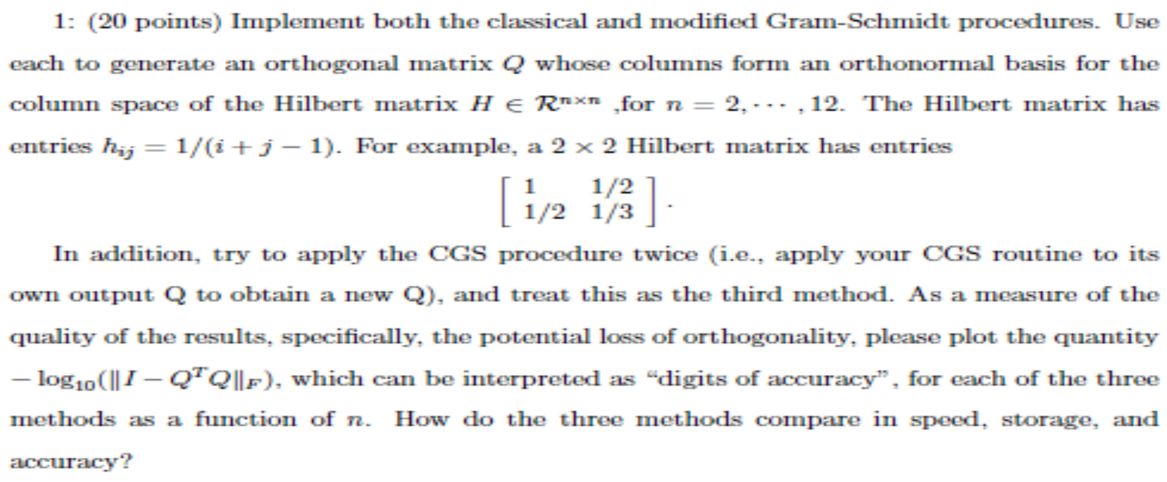
Programming Assignment -1

MATH 6601

Muhammed Emin Ozturk

09-28-2018

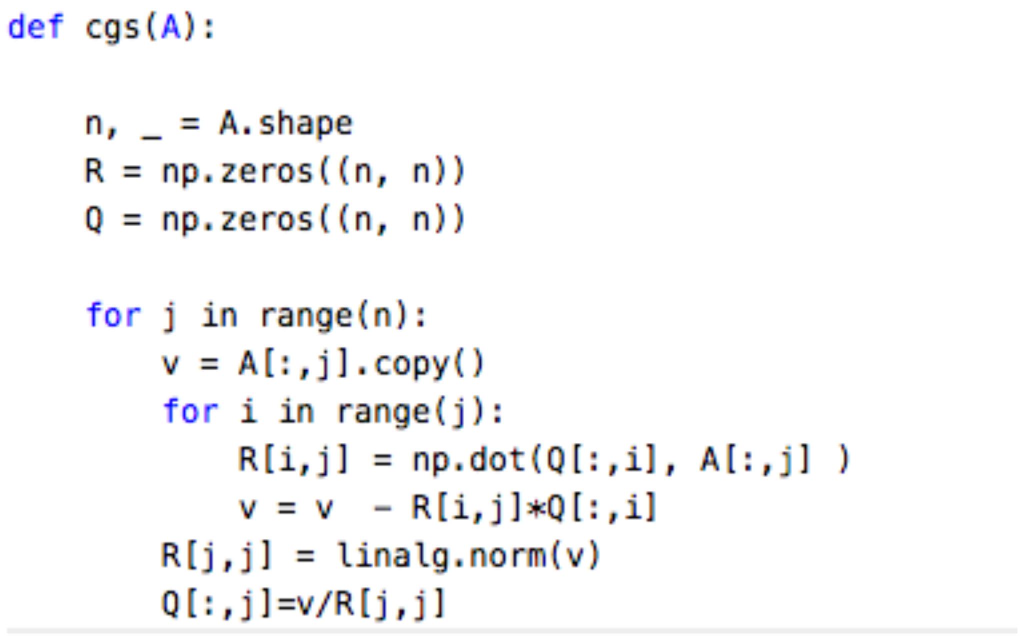

1: (20 points) Implement both the classical and modified Gram-Schmidt procedures. Use each to generate an orthogonal matrix Q whose columns form an orthonormal basis for the column space of the Hilbert matrix $H \in \mathcal{R}^{n\times n}$,for $n = 2, \cdots, 12$. The Hilbert matrix has entries $h_{ij} = 1/(i + j - 1)$. For example, a 2×2 Hilbert matrix has entries

$$\begin{bmatrix} 1 & 1/2 \\ 1/2 & 1/3 \end{bmatrix}.$$

In addition, try to apply the CGS procedure twice (i.e., apply your CGS routine to its own output Q to obtain a new Q), and treat this as the third method. As a measure of the quality of the results, specifically, the potential loss of orthogonality, please plot the quantity $-\log_{10}(\|I - Q^TQ\|_F)$, which can be interpreted as "digits of accuracy", for each of the three methods as a function of n. How do the three methods compare in speed, storage, and accuracy?

In this question, we are supposed to implement Classical Gram-Schdmit , Modified Gram-Schdmit and Double-Classical Gram Schdmit.

Classical Gram-Schdmit implemented in Python


```
def cgs(A):

    n, _ = A.shape
    R = np.zeros((n, n))
    Q = np.zeros((n, n))

    for j in range(n):
        v = A[:,j].copy()
        for i in range(j):
            R[i,j] = np.dot(Q[:,i], A[:,j] )
            v = v  - R[i,j]*Q[:,i]
        R[j,j] = linalg.norm(v)
        Q[:,j]=v/R[j,j]
```

Classical Gram–Schmidt algorithm (CGS):

In step k of CGS, the vector a_k is orthogonalized against $q_1, \dots, q_{k-1}$. The kth column of R is computed and only the first k columns are operated on.

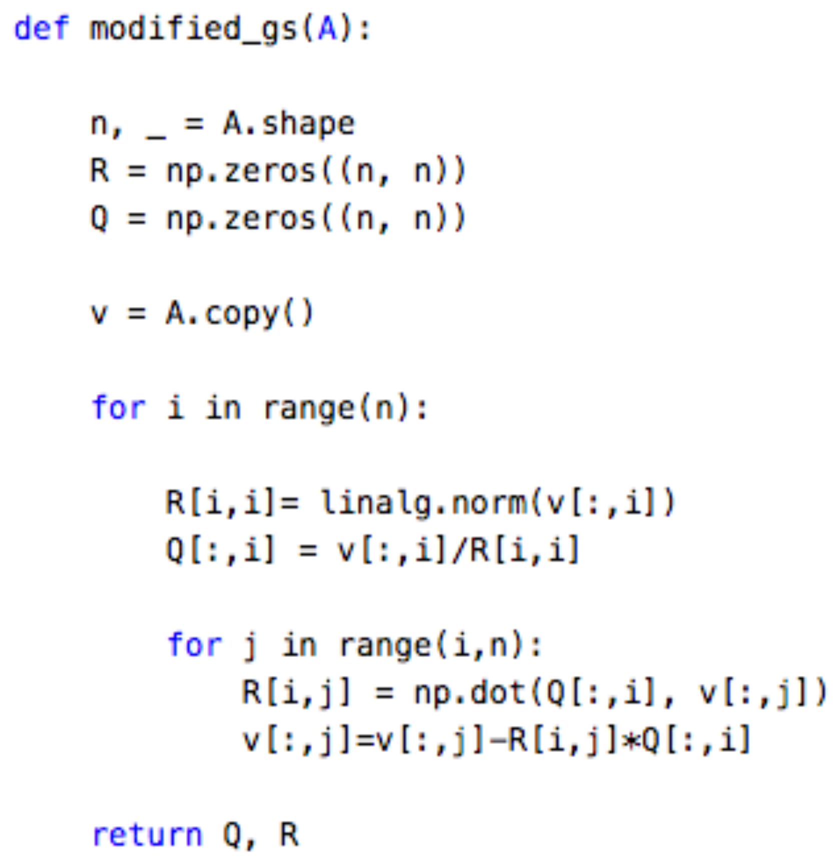
```
function [Q,R] = gschmidt(A);
[m,n] = size(A);
 Q = A; R = zeros(n);
 for k = 1:n
    R(1:k-1,k) = Q(:,1:k-1)'*A(:,k);
    Q(:,k) = A(:,k) - Q(:,1:k-1)*R(1:k-1,k);
    R(k,k) = norm(Q(:,k));
    Q(:,k) = Q(:,k)/R(k,k);
 end
```

Modified Gram–Schmidt algorithm (MGS):

```
function [Q,R] = mgs(A);
[m,n] = size(A);
Q = A; R=zeros(n);
for k = 1:n
   R(k,k) = norm(Q(:,k));
   Q(:,k) = Q(:,k)/R(k,k);
   R(k,k+1:n) = Q(:,k)'*Q(:,k+1:n);
   Q(:,k+1:n) = Q(:,k+1:n) - Q(:,k)*R(k,k+1:n);
end
```

Row-wise MGS has the advantage that a column pivoting strategy can be used, giving an upper triangular matrix R with non-increasing diagonal elements.

Modified Gram Schdmit Algorithm (Implemented in Python)

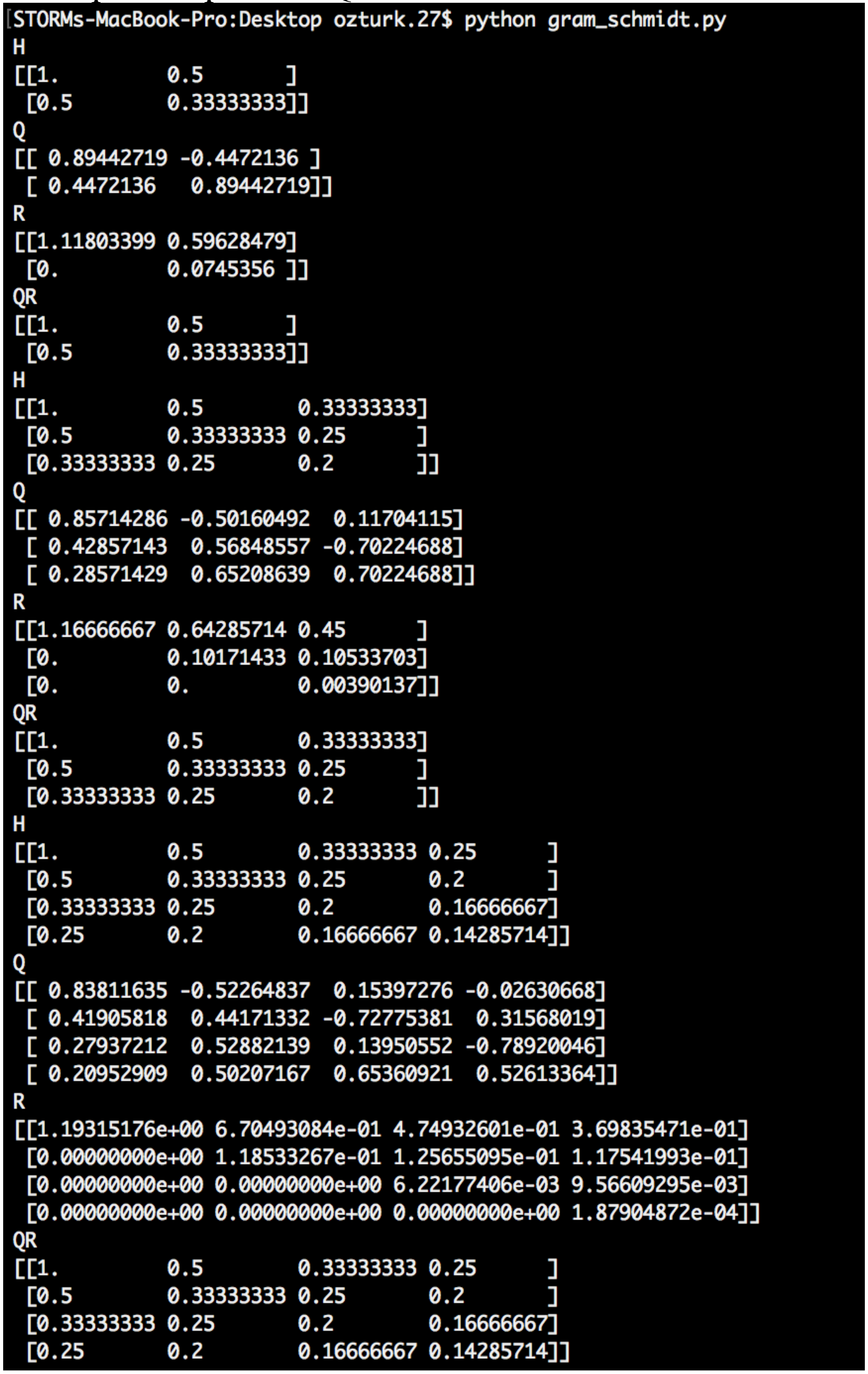


```
def modified_gs(A):

    n, _ = A.shape
    R = np.zeros((n, n))
    Q = np.zeros((n, n))

    v = A.copy()

    for i in range(n):

        R[i,i]= linalg.norm(v[:,i])
        Q[:,i] = v[:,i]/R[i,i]

        for j in range(i,n):
            R[i,j] = np.dot(Q[:,i], v[:,j])
            v[:,j]=v[:,j]-R[i,j]*Q[:,i]

    return Q, R
```

Code For Question – 1

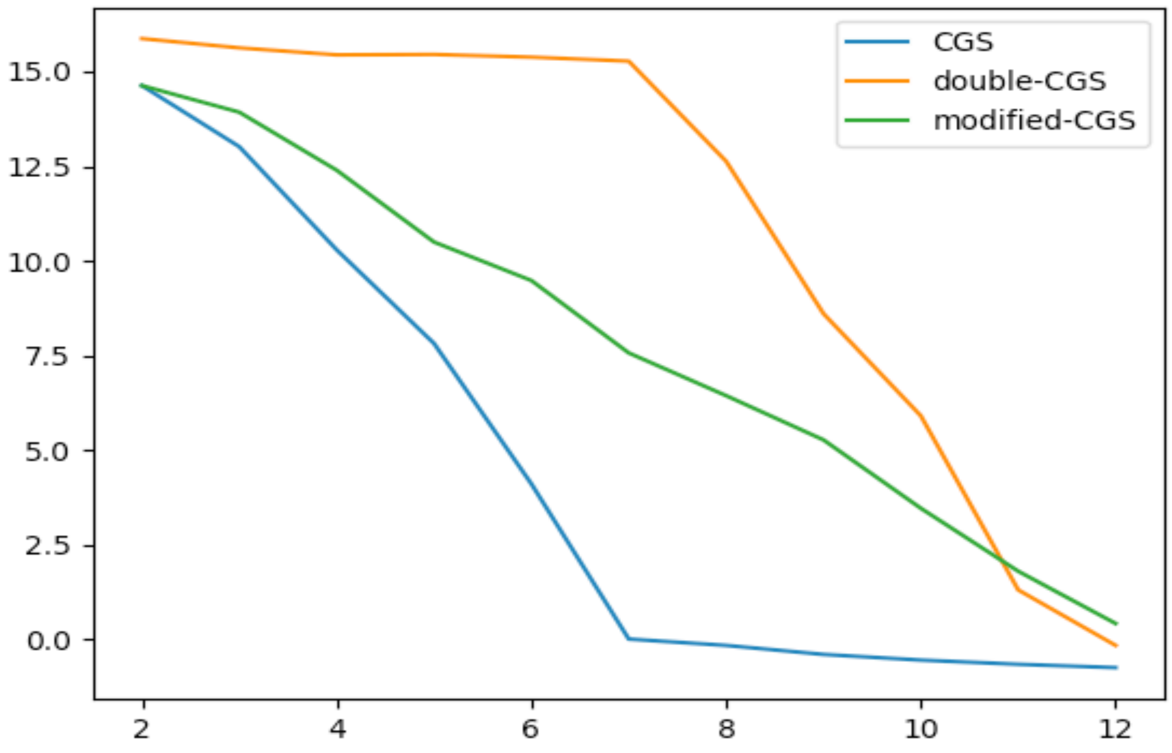
```
import numpy as np
from scipy import linalg
from time import time


import matplotlib.pyplot as plt

def cgs(A):

    n, _ = A.shape
    R = np.zeros((n, n))
    Q = np.zeros((n, n))

    for j in range(n):
        v = A[:,j].copy()
        for i in range(j):
            R[i,j] = np.dot(Q[:,i], A[:,j] )
            v = v  - R[i,j]*Q[:,i]
        R[j,j] = linalg.norm(v)
        Q[:,j]=v/R[j,j]

    return Q, R

def modified_gs(A):

    n, _ = A.shape
    R = np.zeros((n, n))
    Q = np.zeros((n, n))

    v = A.copy()

    for i in range(n):

        R[i,i]= linalg.norm(v[:,i])
        Q[:,i] = v[:,i]/R[i,i]

        for j in range(i,n):
            R[i,j] = np.dot(Q[:,i], v[:,j])
            v[:,j]=v[:,j]-R[i,j]*Q[:,i]

    return Q, R


gs_norms = []
modgs_norms = []
doublecgs_norms = []
for n in range(2,13):

    H = linalg.hilbert(n)

    Q, R  = cgs(H)

    the_norm = -np.log10(linalg.norm( np.eye(n) - np.dot(Q.T, Q) ))

    gs_norms.append(the_norm)

    Q, R  = cgs(Q)
    the_norm = -np.log10(linalg.norm( np.eye(n) - np.dot(Q.T, Q) ))
    doublecgs_norms.append(the_norm)

    Q, R  = modified_gs(H)
    print(np.dot(Q.T, Q))
    the_norm = -np.log10(linalg.norm( np.eye(n) - np.dot(Q.T, Q) ))

    modgs_norms.append(the_norm)


    # print(the_norm )

plt.plot(np.arange(2,13), gs_norms, label='CGS')
plt.plot(np.arange(2,13), doublecgs_norms, label='double-CGS')
```

Example Output For Question 1

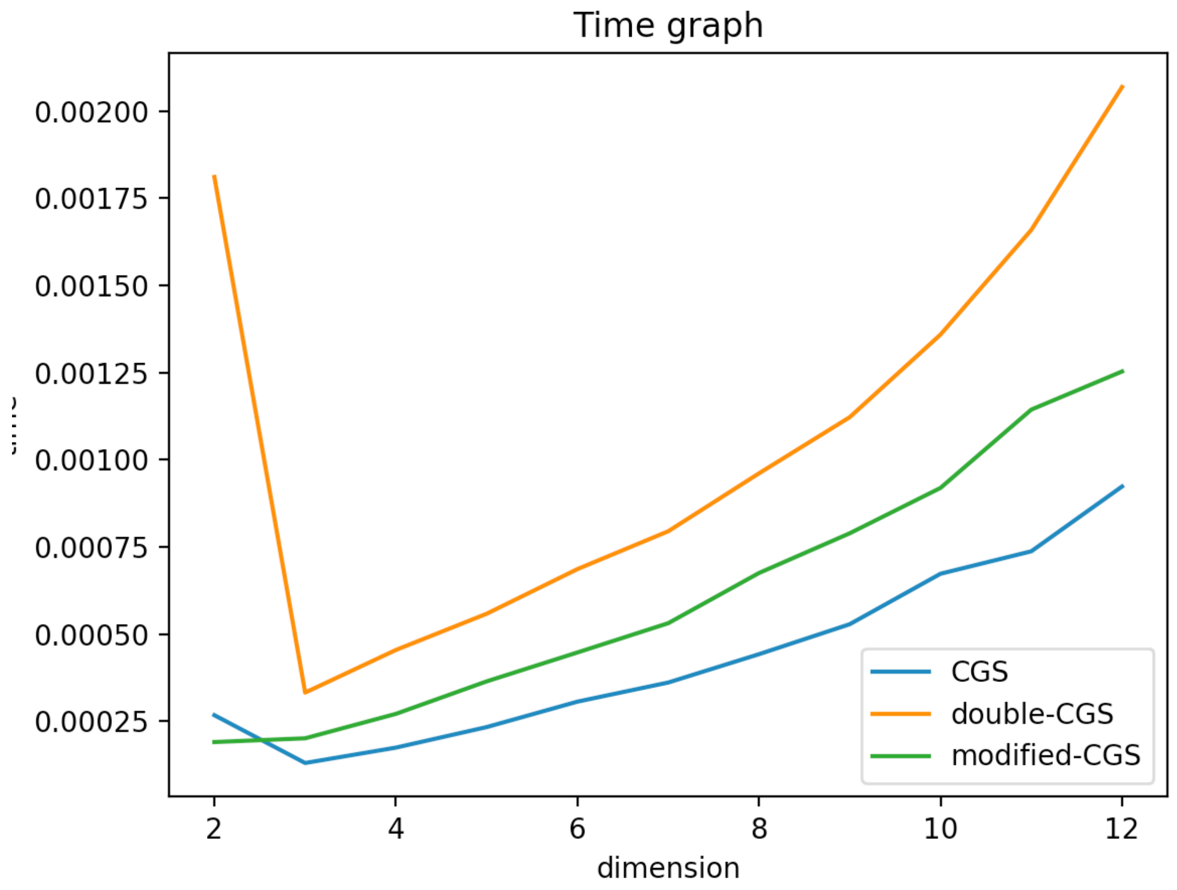


```
STORMs-MacBook-Pro:Desktop ozturk.27$ python gram_schmidt.py
H
[[1.         0.5       ]
 [0.5        0.33333333]]
Q
[[ 0.89442719 -0.4472136 ]
 [ 0.4472136   0.89442719]]
R
[[1.11803399 0.59628479]
 [0.         0.0745356 ]]
QR
[[1.         0.5       ]
 [0.5        0.33333333]]
H
[[1.         0.5        0.33333333]
 [0.5        0.33333333 0.25      ]
 [0.33333333 0.25       0.2       ]]
Q
[[ 0.85714286 -0.50160492  0.11704115]
 [ 0.42857143  0.56848557 -0.70224688]
 [ 0.28571429  0.65208639  0.70224688]]
R
[[1.16666667 0.64285714 0.45      ]
 [0.         0.10171433 0.10533703]
 [0.         0.         0.00390137]]
QR
[[1.         0.5        0.33333333]
 [0.5        0.33333333 0.25      ]
 [0.33333333 0.25       0.2       ]]
H
[[1.         0.5        0.33333333 0.25      ]
 [0.5        0.33333333 0.25       0.2       ]
 [0.33333333 0.25       0.2        0.16666667]
 [0.25       0.2        0.16666667 0.14285714]]
Q
[[ 0.83811635 -0.52264837  0.15397276 -0.02630668]
 [ 0.41905818  0.44171332 -0.72775381  0.31568019]
 [ 0.27937212  0.52882139  0.13950552 -0.78920046]
 [ 0.20952909  0.50207167  0.65360921  0.52613364]]
R
[[1.19315176e+00 6.70493084e-01 4.74932601e-01 3.69835471e-01]
 [0.00000000e+00 1.18533267e-01 1.25655095e-01 1.17541993e-01]
 [0.00000000e+00 0.00000000e+00 6.22177406e-03 9.56609295e-03]
 [0.00000000e+00 0.00000000e+00 0.00000000e+00 1.87904872e-04]]
QR
[[1.         0.5        0.33333333 0.25      ]
 [0.5        0.33333333 0.25       0.2       ]
 [0.33333333 0.25       0.2        0.16666667]
 [0.25       0.2        0.16666667 0.14285714]]
```

Digit Of Accuracy VS N



-n-



Conclusion :

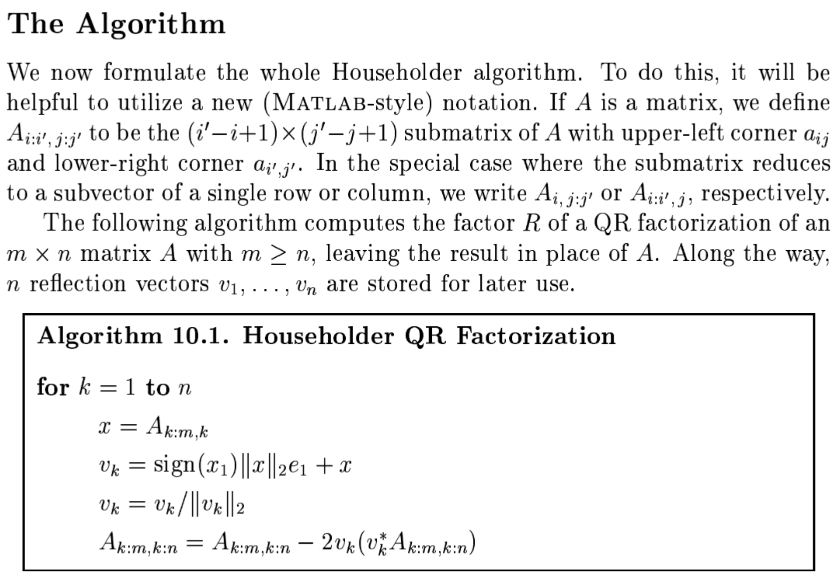
As it seen in the graph 1 and graph 2 , Double Classical Gram Schdmit has longest execution time. It is expected because it run classical Gram-Schdmit method twice.

According to graph 2, Modified Gram Schdmit method is slightly better than Classical Gram Schdmit in terms of time. The other point that we can realize is that time depends on dimension. Whenever we increase dimension , we should expect that execution time also will increase as well.

As it can be observed that DCG has better digit of accuracy that CGS and MGS. It might be because we run CGS algorithm twice which can decrease fault rate. MGS is bettern than CGS in terms of digit of accuracy. We know that MGS is stable. So it is mre reliable algorithm.

In terms of storage, DCG require more than CGS and MGS. I think CGS and MGS need approximately same storage.

2: (20 points) (a): Implement the Householder QR factorization (Algorithm 10.1), the implicit calculation of product $\hat{Q}^*b$ (by modifying Algorithm 10.2), and back substitution (Algorithm 17.1) by writing your own codes.



The Algorithm

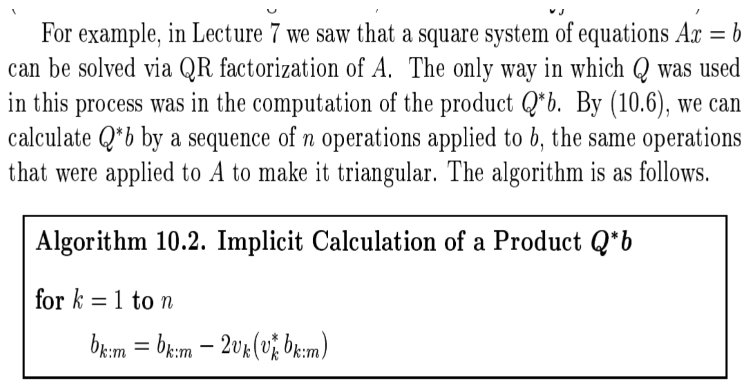
We now formulate the whole Householder algorithm. To do this, it will be helpful to utilize a new (MATLAB-style) notation. If A is a matrix, we define $A_{i:i',\,j:j'}$ to be the $(i'-i+1)\times(j'-j+1)$ submatrix of A with upper-left corner a_{ij} and lower-right corner $a_{i',j'}$. In the special case where the submatrix reduces to a subvector of a single row or column, we write $A_{i,\,j:j'}$ or $A_{i:i',\,j}$, respectively.

The following algorithm computes the factor R of a QR factorization of an $m \times n$ matrix A with $m \geq n$, leaving the result in place of A. Along the way, n reflection vectors $v_1, \ldots, v_n$ are stored for later use.

Algorithm 10.1. Householder QR Factorization

```
for $k = 1$ to $n$
    $x = A_{k:m,k}$
    $v_k = \text{sign}(x_1)\|x\|_2 e_1 + x$
    $v_k = v_k / \|v_k\|_2$
    $A_{k:m,k:n} = A_{k:m,k:n} - 2v_k(v_k^* A_{k:m,k:n})$
```

I have implemented The Householder QR Factorization in Python . In Householder method I have also added Implicit Calculation of a Product

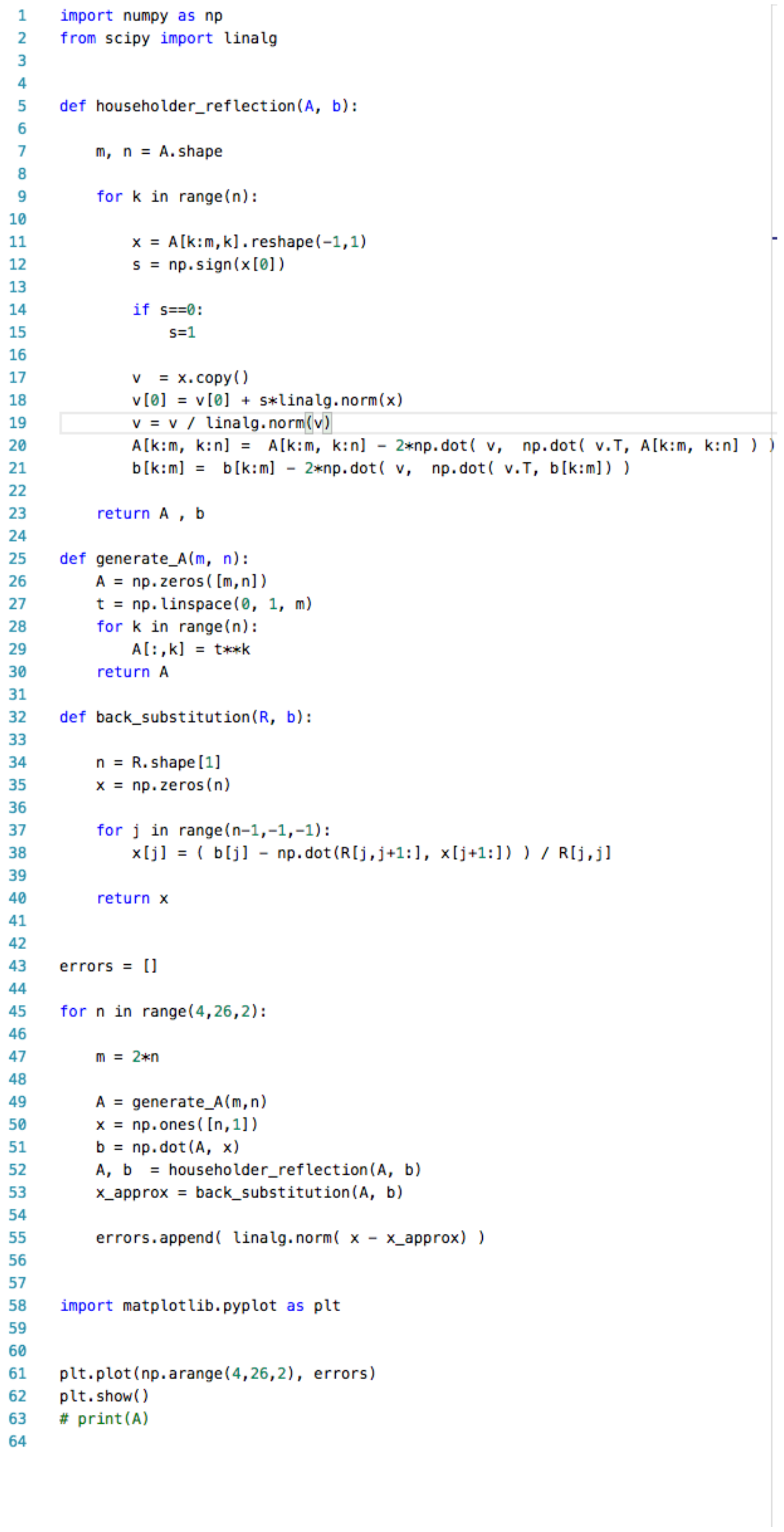


For example, in Lecture 7 we saw that a square system of equations $Ax = b$ can be solved via QR factorization of A. The only way in which Q was used in this process was in the computation of the product Q^*b. By (10.6), we can calculate Q^*b by a sequence of n operations applied to b, the same operations that were applied to A to make it triangular. The algorithm is as follows.

Algorithm 10.2. Implicit Calculation of a Product Q^*b

```
for $k = 1$ to $n$
    $b_{k:m} = b_{k:m} - 2v_k(v_k^* b_{k:m})$
```

The Implementation of Householder with Implicit Calculation of Product Q*b



```python
import numpy as np
from scipy import linalg


def householder_reflection(A, b):

    m, n = A.shape

    for k in range(n):

        x = A[k:m,k].reshape(-1,1)
        s = np.sign(x[0])

        if s==0:
            s=1

        v  = x.copy()
        v[0] = v[0] + s*linalg.norm(x)
        v = v / linalg.norm(v)
        A[k:m, k:n] =  A[k:m, k:n] - 2*np.dot( v,  np.dot( v.T, A[k:m, k:n] ) )
        b[k:m] =  b[k:m] - 2*np.dot( v,  np.dot( v.T, b[k:m]) )

    return A , b

def generate_A(m, n):
    A = np.zeros([m,n])
    t = np.linspace(0, 1, m)
    for k in range(n):
        A[:,k] = t**k
    return A

def back_substitution(R, b):

    n = R.shape[1]
    x = np.zeros(n)

    for j in range(n-1,-1,-1):
        x[j] = ( b[j] - np.dot(R[j,j+1:], x[j+1:]) ) / R[j,j]

    return x


errors = []

for n in range(4,26,2):

    m = 2*n

    A = generate_A(m,n)
    x = np.ones([n,1])
    b = np.dot(A, x)
    A, b  = householder_reflection(A, b)
    x_approx = back_substitution(A, b)

    errors.append( linalg.norm( x - x_approx) )


import matplotlib.pyplot as plt


plt.plot(np.arange(4,26,2), errors)
plt.show()
# print(A)
```

Back Substitution

$$\begin{bmatrix} r_{11} & r_{12} & \cdots & r_{1m} \\ & r_{22} & & \\ & & \ddots & \vdots \\ & & & r_{mm} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix},$$

Algorithm 17.1. Back Substitution

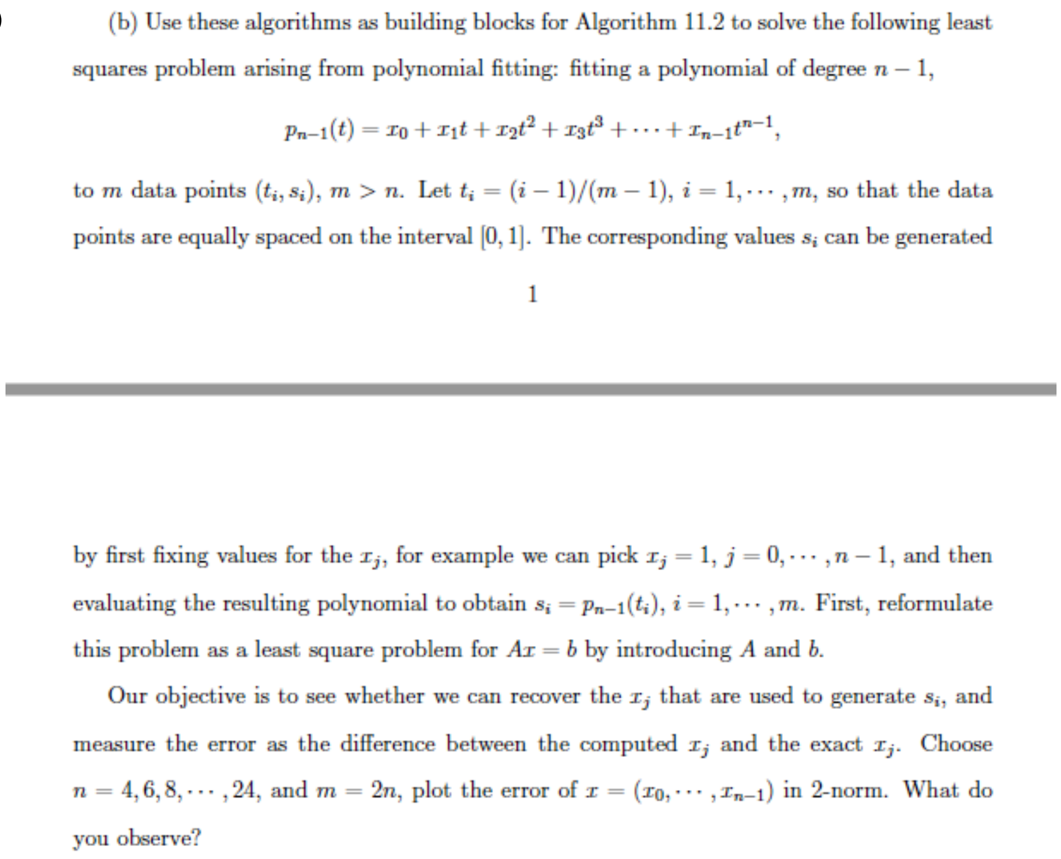
$$\begin{aligned} x_m &= b_m / r_{mm} \\ x_{m-1} &= (b_{m-1} - x_m r_{m-1,m}) / r_{m-1,m-1} \\ x_{m-2} &= (b_{m-2} - x_{m-1} r_{m-2,m-1} - x_m r_{m-2,m}) / r_{m-2,m-2} \\ &\vdots \\ x_j &= \left(b_j - \sum_{k=j+1}^{m} x_k r_{jk} \right) \Big/ r_{jj} \end{aligned}$$

The structure is triangular, with a subtraction and a multiplication at each position. The operation count is accordingly twice the area of an $m \times m$ triangle:

$$\text{Work for back substitution:} \quad \sim m^2 \text{ flops.} \qquad (17.2)$$

```
def back_substitution(R, b):

    n = R.shape[1]
    x = np.zeros(n)

    for j in range(n-1,-1,-1):
        x[j] = ( b[j] - np.dot(R[j,j+1:], x[j+1:]) ) / R[j,j]

    return x
```

2-b)



(b) Use these algorithms as building blocks for Algorithm 11.2 to solve the following least squares problem arising from polynomial fitting: fitting a polynomial of degree $n-1$,

$$p_{n-1}(t) = x_0 + x_1 t + x_2 t^2 + x_3 t^3 + \cdots + x_{n-1} t^{n-1},$$

to m data points (t_i, s_i), $m > n$. Let $t_i = (i-1)/(m-1)$, $i = 1, \cdots, m$, so that the data points are equally spaced on the interval $[0,1]$. The corresponding values s_i can be generated

1

by first fixing values for the x_j, for example we can pick $x_j = 1$, $j = 0, \cdots, n-1$, and then evaluating the resulting polynomial to obtain $s_i = p_{n-1}(t_i)$, $i = 1, \cdots, m$. First, reformulate this problem as a least square problem for $Ax = b$ by introducing A and b.

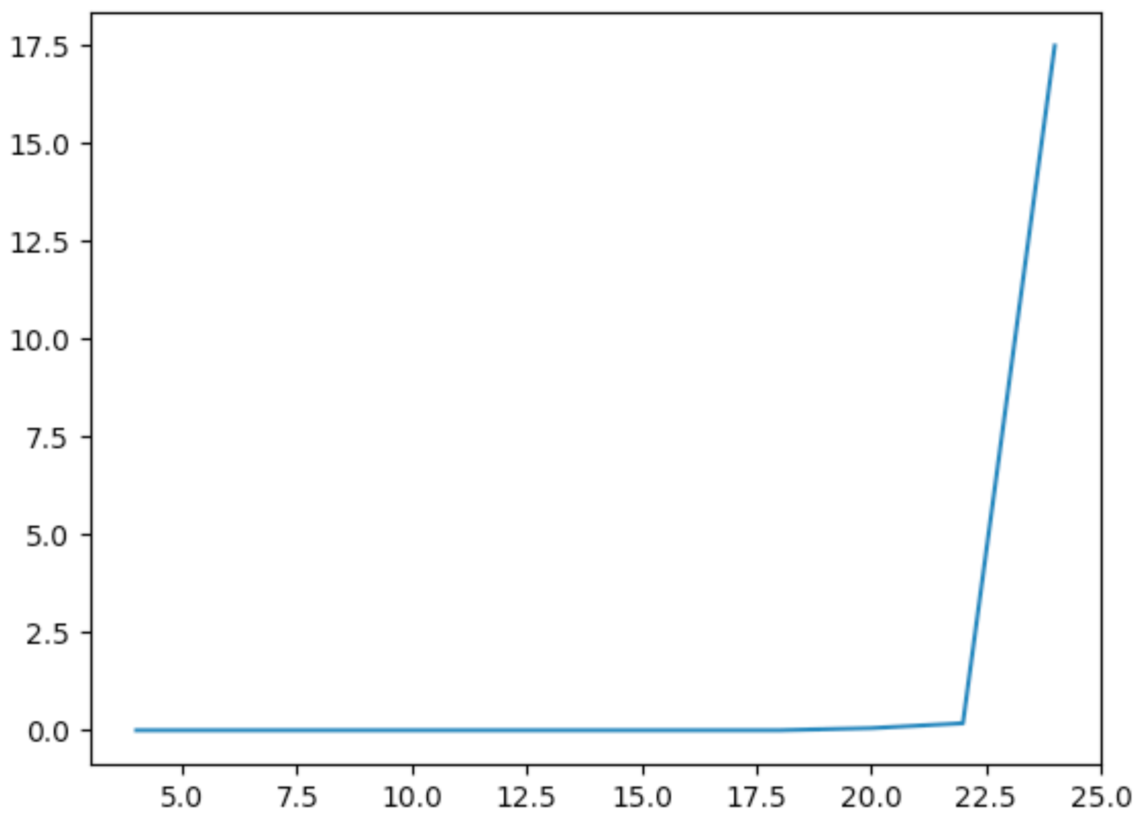
Our objective is to see whether we can recover the x_j that are used to generate s_i, and measure the error as the difference between the computed x_j and the exact x_j. Choose $n = 4, 6, 8, \cdots, 24$, and $m = 2n$, plot the error of $x = (x_0, \cdots, x_{n-1})$ in 2-norm. What do you observe?

To apply algorithm and code in a lets define matix such that $t_i = (i-1)/(m-1)$ $i=1,\ldots\ldots m$ so that data points are equally spaced on the inverval [0,1]

```
def generate_A(m, n):
   A = np.zeros([m,n])
   t = np.linspace(0, 1, m)
   for k in range(n):
      A[:,k] = t**k
   return A
```

Error of X in 2 norm

Error of X in 2 – Norm VS N



Conclusion

As you can see in graph above, till Size N = 22 error is approximately 0. However, after n size start to increase, error rate also increases sharply. We can conclude that for big variable unknow system may contain high error rate.

Appendix

```
import numpy as np
from scipy import linalg
from time import time


import matplotlib.pyplot as plt

def cgs(A):

    n, _ = A.shape
    R = np.zeros((n, n))
    Q = np.zeros((n, n))

    for j in range(n):
        v = A[:,j].copy()
        for i in range(j):
            R[i,j] = np.dot(Q[:,i], A[:,j] )
            v = v  - R[i,j]*Q[:,i]
        R[j,j] = linalg.norm(v)
        Q[:,j]=v/R[j,j]

    return Q, R

def modified_gs(A):

    n, _ = A.shape
    R = np.zeros((n, n))
    Q = np.zeros((n, n))

    v = A.copy()

    for i in range(n):

        R[i,i]= linalg.norm(v[:,i])
        Q[:,i] = v[:,i]/R[i,i]

        for j in range(i,n):
            R[i,j] = np.dot(Q[:,i], v[:,j])
            v[:,j]=v[:,j]-R[i,j]*Q[:,i]

    return Q, R
```

```
gs_norms = []
modgs_norms = []
doublecgs_norms = []

gs_time=[]
modgs_time=[]
doublecgs_time=[]

for n in range(2,13):

    H = linalg.hilbert(n)

    start1=time()

    Q, R  = cgs(H)

    end1=time()

    executeTime = abs(start1 -end1)

    the_norm = -np.log10(linalg.norm( np.eye(n) - np.dot(Q.T, Q) ))

    gs_norms.append(the_norm)
    gs_time.append(executeTime)

    start=time()

    Q, R  = cgs(Q)
    the_norm = -np.log10(linalg.norm( np.eye(n) - np.dot(Q.T, Q) ))
    doublecgs_norms.append(the_norm)

    end2=time()

    executeTime = abs(start1 -end2)
    doublecgs_time.append(executeTime)

    start3=time()
    Q, R  = modified_gs(H)
    #print(np.dot(Q.T, Q))
    the_norm = -np.log10(linalg.norm( np.eye(n) - np.dot(Q.T, Q) ))
    end3=time()

    executeTime = abs(start3 -end3)
    modgs_norms.append(the_norm)
    modgs_time.append(executeTime)
    print('H')
    print(H)
    print('Q')
    print(Q)
    print('R')
    print(R)

    print('QR')
    print( np.dot(Q, R) )


    # print(the_norm )
plt.figure()
plt.plot(np.arange(2,13), gs_norms, label='CGS')
plt.plot(np.arange(2,13), doublecgs_norms, label='double-CGS')
plt.plot(np.arange(2,13), modgs_norms, label='modified-CGS')

plt.legend()


plt.figure()

plt.plot(np.arange(2,13), gs_time, label='CGS')
plt.plot(np.arange(2,13), doublecgs_time, label='double-CGS')
plt.plot(np.arange(2,13), modgs_time, label='modified-CGS')

plt.title('Time graph')
plt.xlabel('dimension')
plt.ylabel('time')
plt.legend()

plt.show()
```

```
import numpy as np
from scipy import linalg


def householder_reflection(A, b):
    m, n = A.shape

    for k in range(n):

        x = A[k:m,k].reshape(-1,1)
        s = np.sign(x[0])

        if s==0:
            s=1

        v  = x.copy()
        v[0] = v[0] + s*linalg.norm(x)
        v = v / linalg.norm(v)
        A[k:m, k:n] =  A[k:m, k:n] - 2*np.dot( v,  np.dot( v.T, A[k:m, k:n] ) )
        b[k:m] =  b[k:m] - 2*np.dot( v,  np.dot( v.T, b[k:m]) )

    return A , b

def generate_A(m, n):
    A = np.zeros([m,n])
    t = np.linspace(0, 1, m)
    for k in range(n):
        A[:,k] = t**k
    return A

def back_substitution(R, b):

    n = R.shape[1]
    x = np.zeros(n)

    for j in range(n-1,-1,-1):
        x[j] = ( b[j] - np.dot(R[j,j+1:], x[j+1:]) ) / R[j,j]

    return x


errors = []

for n in range(4,26,2):

    m = 2*n

    A = generate_A(m,n)
    x = np.ones([n,1])
    b = np.dot(A, x)
    A, b  = householder_reflection(A, b)
    x_approx = back_substitution(A, b)

    errors.append( linalg.norm( x - x_approx) )


import matplotlib.pyplot as plt


plt.plot(np.arange(4,26,2), errors)
plt.show()
# print(A)
```