MA 8463 - Homework 5
Jacob Kutch

5.1. a. Let for $n \geq 2$ and small $\varepsilon > 0$, $A = \begin{bmatrix} 0 & 1 & & \\ & \ddots & \ddots & \\ & & 0 & 1 \\ \varepsilon & & & 0 \end{bmatrix} \in \mathbb{R}^{n \times n}$
Verify that the characteristic polynomial of A is given by $\Phi(\lambda) = \lambda^n - \varepsilon = 0$.

Proof: Let $B = A - \lambda I$ and consider the LU decomposition with partial pivoting $PB = LU \Rightarrow B = P^{-1}LU$.

$$\Phi(\lambda) = \det(A - \lambda I) = \det(B) = \det(P^{-1}LU) = \frac{1}{\det(P)} \det(L)\det(U).$$

Since P is a permutation matrix, its determinant is $(-1)^k$ where k is the number of row interchanges performed. L is lower triangular, so $\det(L) = \prod_{i=1}^{n} l_{ii} = 1$, since $l_{ii} = 1 \ \forall i$ in L. Likewise, U is upper triangular, so $\det(U) = \prod_{i=1}^{n} u_{ii}$. So $\Phi(\lambda) = (-1)^k \prod_{i=1}^{n} u_{ii}$. The structure of A leads to a predictable pattern in row reductions, namely a row interchange $R_{2i-1} \leftrightarrow R_n$ for $\lfloor \frac{n}{2} \rfloor$ total interchanges. In order, sum between row n and row 1 multiplied by $\frac{\lambda}{\varepsilon}$, followed by a repeating pattern of $R_n \leftarrow R_n + \frac{1}{\lambda} R_{2i}$ for $2i < n-1$, then a row interchange $R_n \leftrightarrow R_{2i+1}$, then a row reduction $R_n \leftarrow R_n + \lambda^2 R_{2i+1}$. This pattern repeats until

$$U = \begin{bmatrix} \varepsilon & 0 & 0 & 0 & 0 & \cdots & -\lambda \\ 0 & -\lambda & 1 & 0 & 0 & \cdots & 0 \\ 0 & 0 & \frac{1}{\lambda} & 0 & 0 & \cdots & -\frac{\lambda^2}{\varepsilon} \\ 0 & 0 & 0 & -\lambda & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & & \vdots \\ 0 & 0 & 0 & 0 & & & \alpha \end{bmatrix}$$

where $\alpha = \frac{\varepsilon - \lambda^n}{\varepsilon}$ for even n and $\alpha = \frac{\varepsilon - \lambda^n}{\lambda \varepsilon}$.
Thus $\Phi(\lambda) = (-1)^{(k)}(\varepsilon)(\alpha) = 0$. $(-1)^k$ and any denominators may be divided so that

$$\Phi(\lambda) = \varepsilon - \lambda^n = 0 \Rightarrow \lambda^n - \varepsilon = 0.$$

b. Verify that $\left.\frac{d\lambda}{d\varepsilon}\right|_{\varepsilon=0} = \infty$

From (a), we have $\lambda = \sqrt[n]{\varepsilon}$, so $\frac{d\lambda}{d\varepsilon} = \frac{1}{n}\varepsilon^{\frac{1}{n}-1} = \frac{1}{n\varepsilon}\varepsilon^{\frac{1}{n}}$

Since the definition of a derivative of $f(x)$ at $x=c$ is $f'(c) = \lim_{x\to c}\frac{f(x)-f(c)}{x-c}$,

$$\lambda'(0) = \lim_{\varepsilon\to 0}\frac{\lambda(\varepsilon)-\lambda(0)}{\varepsilon-0} = \lim_{\varepsilon\to 0}\frac{\lambda(\varepsilon)}{\varepsilon} = \lim_{\varepsilon\to 0}\frac{\varepsilon^{\frac{1}{n}}}{\varepsilon} = \lim_{\varepsilon\to 0}\varepsilon^{\frac{1-n}{n}}$$

$$\Rightarrow \lim_{\varepsilon\to 0}\varepsilon^{-\frac{(n-1)}{n}} = \lim_{\varepsilon\to 0}\frac{1}{\varepsilon^{\frac{n-1}{n}}}. \text{ Since } \lim_{\varepsilon\to 0}\varepsilon^{\frac{n-1}{n}} \to 0, \text{ we know}$$

that $\forall n$, $\left.\frac{d\lambda}{d\varepsilon}\right|_{\varepsilon=0} = \lim_{\varepsilon\to 0}\frac{1}{\varepsilon^{\frac{n-1}{n}}} \to \infty$

2. Prove that similar matrices have the same eigenvalues.

Proof: Let $A, B \in \mathbb{C}^{n\times n}$ be similar matrices so that $\exists S \in \mathbb{C}^{n\times n}$ such that $B = S^{-1}AS$. Let the characteristic polynomials of A and B be given by $\Phi_A(\lambda)$ and $\Phi_B(\lambda)$.

$$\Phi_B(\lambda) = \det(B-\lambda I) = \det(S^{-1}AS - \lambda I) \qquad (1)$$

Since $S^{-1}S = I$, let $S^{-1}AS - \lambda I = S^{-1}AS - \lambda S^{-1}S = S^{-1}(AS-\lambda S) = S^{-1}(A-\lambda I)S$

Then (1) becomes $\Phi_B(\lambda) = \det(S^{-1}(A-\lambda I)S)$. Since determinants of matrix products are products of the determinants of each matrix,

$\Phi_B(\lambda) = \det(S^{-1})\det(A-\lambda I)\det(S)$. Since $\det(S^{-1}) = \frac{1}{\det(S)}$,

$\Phi_B(\lambda) = \det(A-\lambda I) = \Phi_A(\lambda)$, so the characteristic polynomials of A and B are the same. This implies that A and B also have the same eigenvalues.

5-3

Code:

```
import numpy as np

def invPowerIter(A, sigma, x0, max_iter = 50, TOL=10e-8):
    B = A - sigma*np.eye(np.shape(A)[0])
    i = 0
    x, y, eta = [x0], [], []
    flag = False
    while (not flag) and (i < max_iter):
        y.append(np.linalg.solve(B, x[i]))
        x.append(y[i]/np.linalg.norm(y[i]))
        eta.append(x[i+1].T @ (A @ x[i+1]))
        i += 1
        if (np.linalg.norm(x[i]-x[i-1]) < TOL): flag = True
    print(f'eigenvector for each iteration:\n{np.array(x)}')
    print(f'eigenvalue for each iteration:\n{np.array(eta)}\n')
    return np.array(x[1:]), np.array(eta)

def getEigVecConv(trueEigVec, appEigVec):
    numIter = len(appEigVec)
    absErr = np.zeros(numIter)
    convRates = np.zeros(numIter-1)
    for i in range(numIter):
        absErr[i] = np.linalg.norm(trueEigVec - appEigVec[i])
        if i < (numIter-1):
            convRates[i] = np.linalg.norm(trueEigVec - appEigVec[i+1])/absErr[i]
    print(f'absolute error in dominant eigenvector for each
iteration:\n{absErr}')
    print(f'convergence rate in dominant eigenvector for each
iteration:\n{convRates}\n')

def getEigValConv(trueEigVal, appEigVal):
    numIter = len(appEigVal)
    absErr = np.zeros(numIter)
    convRates = np.zeros(numIter-1)
    for i in range(numIter):
        absErr[i] = np.abs(trueEigVal-appEigVal[i])
        if i < (numIter-1):
            convRates[i] = np.abs(trueEigVal-appEigVal[i+1])/absErr[i]
    print(f'absolute error in dominant eigenvalue for each iteration:\n{absErr}')
    print(f'convergence rate in dominant eigenvalue for each
iteration:\n{convRates}\n')
```

```
if __name__ == '__main__':
    A = np.array([[1,1,1],[-1,9,2],[0,-1,2]])
    shift = 9
    initGuess = np.array([1,1,1])
    eigenVec, eigenVal = invPowerIter(A, shift, initGuess, 10)
    trueEigVal, trueEigVec = np.linalg.eig(A)
    eigIndices = np.argsort(trueEigVal)
    print(f'true eigenvectors sorted in order of largest singular
values:\n{trueEigVec[eigIndices][::-1]}')
    print(f'true eigenvalues sorted in order of largest singular
values:\n{trueEigVal[eigIndices][::-1]}\n')
    domEigVal = trueEigVal[eigIndices][-1]
    domEigVec = trueEigVec[eigIndices][-1]
    getEigVecConv(domEigVec, eigenVec)
    getEigValConv(domEigVal, eigenVal)
```

Output:

eigenvector for each iteration:
[[1. 1. 1.]
 [-0.15371887 -0.98380074 0.09223132]
 [0.11252711 0.98288984 -0.14582595]
 [-0.11004734 -0.98268937 0.14903418]
 [0.10989713 0.98267651 -0.14922966]
 [-0.10988801 -0.98267572 0.14924159]
 [0.10988746 0.98267567 -0.14924232]
 [-0.10988743 -0.98267567 0.14924236]
 [0.10988743 0.98267567 -0.14924237]
 [-0.10988743 -0.98267567 0.14924237]
 [0.10988743 0.98267567 -0.14924237]]
eigenvalue for each iteration:
[8.64650284 8.59010454 8.58478324 8.58445013 8.58442968 8.58442842
 8.58442835 8.58442834 8.58442834 8.58442834]

true eigenvectors sorted in order of largest singular values:
[[0.10988743 -0.98786135 0.55163424]
 [-0.14924237 -0.1225255   0.8186846]
 [0.98267567 -0.09548538 -0.15954682]]
true eigenvalues sorted in order of largest singular values:
[8.58442834 2.19488191 1.22068975]

absolute error in **dominant eigenvector** for each iteration:
[0.52967519 2.09053054 0.45878629 2.09146592 0.45852806 2.09146937

0.4585271 2.09146938 0.4585271 2.09146938]
convergence rate in **dominant eigenvector** for each iteration:
[3.94681602 0.21945926 4.55869312 0.21923764 4.56126802 0.21923682
 4.56127758 0.21923682 4.56127761]

absolute error in **dominant eigenvalue** for each iteration:
[6.20744952e-02 5.67619921e-03 3.54903717e-04 2.17876922e-05
 1.33543518e-06 8.18098016e-08 5.00969932e-09 3.06668468e-10
 1.87725391e-11 1.15285559e-12]
convergence rate in **dominant eigenvalue** for each iteration:
[0.09144173 0.06252489 0.06139043 0.0612931 0.06126078 0.06123593
 0.06121494 0.06121444 0.06141181]

5-4

Code:

```
import numpy as np

# reference was Wikipedia:
https://en.wikipedia.org/wiki/Gram%E2%80%93Schmidt_process#Numerical_stability
def modifiedGramSchmidt(A):
    m, n = np.shape(A)
    R = np.zeros((n,n))
    U = np.zeros((m,n))
    Q = np.zeros((m,n))
    projection = lambda a,b: (np.dot(a,b)/np.dot(b,b))*b
    for cols in range(n):
        U[:,cols] = A[:,cols] - (cols!=0)*projection(A[:,cols],A[:,0])
        for k in range(1,cols):
            U[:,cols] -= projection(U[:,cols],U[:,k])
        Q[:,cols] = U[:,cols]/np.linalg.norm(U[:,cols])
        for rows in range(cols+1):
            R[rows,cols] = np.dot(Q[:,rows],A[:,cols])
    return Q, R

def orthogonalIter(A, Z0, max_iter = 50, TOL=10e-8):
    Z, Y = [Z0], []
    i = 0
    flag = False
    while (not flag) and (i < max_iter):
        Y.append(A @ Z[i])
        Z.append(modifiedGramSchmidt(Y[i])[0])
```

```
        i += 1
        if np.allclose(Z[i], Z[i-1], rtol=TOL): flag = True
    return np.array(Z)


def testSubspace(A, Z):
    '''test if Z is an invariant subspace of A corresponding to the eigenspace of
the largest eigenvalues
        Here, Z is the final iteration returned by the orthogonal iteration
algorithm
    '''
    eigVal, eigVec = np.linalg.eig(A)
    # sort in order of largest singular values
    eigIndices = np.argsort(eigVal)[::-1]
    eigVec = eigVec[eigIndices]
    Z = Z[eigIndices]
    print(f'(sorted) eigenvectors of A:\n{eigVec}\n')
    print(f'(sorted) Z_k = \n{Z}\n')
    print(f'note: The absolute values of the elements in \n{eigVec[:,0]} and
\n{Z[:,0]} \n should be roughly the same\n')
    testArr = np.column_stack((eigVec[:,0],Z[:,0]))
    # test if the first column of Z is in an eigenspace of A
        # could probably also use np.allclose
    if (np.linalg.matrix_rank(testArr) < (eigVec.shape[1]+1)):
        return True


if __name__ == '__main__':
    A = np.array([[-31,-35,16],[-10,-8,4],[-100,-104,49]])
    n = A.shape[0]
    spectrum = np.array([9,3,-2]) # eigenvalues of A in descending order
    Z0 = np.eye(n)[:,:2] # first 2 standard basis vectors
    Z = orthogonalIter(A, Z0, 20, 10e-6)
    truthLabels = ['is not', 'is']
    idx = testSubspace(A, Z[-1])
    print(f'Z_k {truthLabels[idx]} an invariant subspace of A.')
```

Output:

(sorted) eigenvectors of A:
[[3.71390676e-01 -2.35702260e-01 -7.27606875e-01]
 [9.28476691e-01 -9.42809042e-01 -4.85071250e-01]
 [-6.54606271e-16 -2.35702260e-01 4.85071250e-01]]

(sorted) Z_k =
[[-3.71390676e-01 -6.63410367e-01]
 [-9.28476691e-01 2.65364147e-01]
 [-5.32409178e-11 6.99620293e-01]]

note: The absolute values of the elements in
[3.71390676e-01 9.28476691e-01 -6.54606271e-16] and
[-3.71390676e-01 -9.28476691e-01 -5.32409178e-11]
 should be roughly the same

Z_k is an invariant subspace of A.

5-5
Code:

```
import numpy as np

# reference was Wikipedia:
https://en.wikipedia.org/wiki/Gram%E2%80%93Schmidt_process#Numerical_stability
def modifiedGramSchmidt(A):
    m, n = np.shape(A)
    R = np.zeros((n,n))
    U = np.zeros((m,n))
    Q = np.zeros((m,n))
    projection = lambda a,b: (np.dot(a,b)/np.dot(b,b))*b
    for cols in range(n):
        U[:,cols] = A[:,cols] - (cols!=0)*projection(A[:,cols],A[:,0])
        for k in range(1,cols):
            U[:,cols] -= projection(U[:,cols],U[:,k])
        Q[:,cols] = U[:,cols]/np.linalg.norm(U[:,cols])
        for rows in range(cols+1):
            R[rows,cols] = np.dot(Q[:,rows],A[:,cols])
    return Q, R

def QRiter_noShift(A0, max_iter = 50, TOL = 10e-8):
    A = [A0]
    n = A0.shape[0]
    i = 0
    flag = False
    while (not flag) and (i < max_iter):
        Q, R = modifiedGramSchmidt(A[i])
        A.append(np.matmul(R, Q))
        i += 1
        if (np.abs(A[i][n-1][n-1]-A[i-1][n-1][n-1]) < TOL): flag = True
    return np.array(A)
```

```
def QRiter_withShift(A0, max_iter = 50, TOL = 10e-8):
    A = [A0]
    sigma = []
    n = A0.shape[0]
    i = 0
    flag = False
    while (not flag) and (i < max_iter):
        sigma.append(A[i][2][2]/2)
        diag = sigma[i]*np.eye(n)
        Q, R = modifiedGramSchmidt(A[i] - diag)
        A.append((R @ Q) + diag)
        i += 1
        if (np.abs(A[i][n-1][n-1]-A[i-1][n-1][n-1]) < TOL): flag = True
    return np.array(A)

def getConvRate(eigVals, A):
    numIter = A.shape[0]
    convRates = np.zeros(numIter-1)
    for i in range(numIter-1):
        appEigenVals_next = np.diag(A[i+1])
        appEigenVals_prev = np.diag(A[i])
        convRates[i] = np.linalg.norm(appEigenVals_next-
eigVals)/np.linalg.norm(appEigenVals_prev-eigVals)
    return convRates

def testEigenVals(trueEigVals, appEigVals):
    truthLabels = ['are not', 'are']
    print(f'The approximated eigenvalues,\n{appEigVals} \n' +
          f'{truthLabels[np.allclose(trueEigVals, appEigVals)]}' +
          f' roughly the same as the true eigenvalues, \n{trueEigVals}')
    print(f'Error: {np.linalg.norm(trueEigVals - appEigVals)}\n')

if __name__ == '__main__':
    A0 = np.array([[-31,-35,16],[-10,-8,4],[-100,-104,49]])
    spectrum = np.array([9,3,-2])
    A_noShift = QRiter_noShift(A0, 30)
    appEigenVals = np.diag(A_noShift[-1])
    testEigenVals(spectrum, appEigenVals)
    A_shift = QRiter_withShift(A0, max_iter = 30)
    appEigenVals = np.sort(np.diag(A_shift[-1]), axis=None)[::-1]
    testEigenVals(spectrum, appEigenVals)

    noShiftConvRates = getConvRate(spectrum, A_noShift)
```

```
    # error looked much larger when passing spectrum and sorting withing the
function
    shiftConvRates = getConvRate(np.array([9,-2,3]), A_shift)
    print(f'convergence rates with no shifting: v1 = \n{noShiftConvRates}\n')
    print(f'convergence rates with shifting: v2 = \n{shiftConvRates}\n')
    print(f'||v1|| = {np.linalg.norm(noShiftConvRates)}')
    print(f'||v2|| = {np.linalg.norm(shiftConvRates)}')
```

Output:

The approximated eigenvalues,
[9. 2.99999499 -1.99999499]
are roughly the same as the true eigenvalues,
[9 3 -2]
Error: 7.079608043996668e-06

The approximated eigenvalues,
[8.999995 3.00000002 -1.99999502]
are roughly the same as the true eigenvalues,
[9 3 -2]
Error: 7.059715061434967e-06

convergence rates with no shifting: v1 =
[0.03848308 0.31316222 0.28719607 1.09628519 0.68465936 0.6882024
 0.67542272 0.66399199 0.67151639 0.66350683 0.66927349 0.66486293
 0.66796515 0.66577491 0.66728188 0.66624888 0.66694999 0.66647574
 0.66679511 0.6665805 0.66672441 0.66662802 0.66669254 0.66664933
 0.66667833 0.66665872 0.66667223 0.66666256 0.66666992 0.66666355]

convergence rates with shifting: v2 =
[0.51770395 0.4603814 0.8437985 1.21412015 0.8830829 1.59489639
 0.61214681 1.27185265 0.28199619 0.78562462 0.35170951 0.52736545
 0.44038002 0.47967442 0.46108327 0.46965551 0.46561918 0.46746981
 0.46658135 0.46697197 0.46676838 0.46684349 0.46679038 0.46679825
 0.46677959 0.46677335 0.46676502]

||v1|| = 3.606315855986793
||v2|| = 3.5119770275392463

5-6
Code:

```
import numpy as np
from copy import deepcopy
```

```
# because numpy.matmul apparently defaults to np.dot for 1d arrays
def colByRowMult(u, v):
    n = np.size(u)
    if np.size(v) == n and u.ndim == 1 and v.ndim == 1:
        A = np.zeros((n,n))
        for i in range(n):
            A[i,:] = u[i]*v
        return A
    else:
        raise Exception('Arguments must be one dimensional vectors of the same
size')

# reference was Wikipedia:
https://en.wikipedia.org/wiki/Gram%E2%80%93Schmidt_process#Numerical_stability
def householder(x):
    n = len(x)
    sigma = np.dot(x[1:], x[1:])
    v = deepcopy(x)
    v[0] = 1
    beta = 0
    if sigma != 0:
        mu = np.sqrt(x[0]**2 + sigma)
        v[0] = x[0]-mu if x[0] <= 0 else -sigma/(x[0]+mu)
        beta = 2*(v[0]**2)/(sigma+v[0]**2)
        v /= v[0]
    return v, beta

def householderReflection(A):
    m, n = np.shape(A)
    R = deepcopy(A)
    Q = np.eye(m)
    for j in range(n):
        v, beta = householder(R[j:, j])
        bvvt = beta*colByRowMult(v, v)
        R[j:,:] -= (bvvt @ R[j:,:])
        Q[:,j:] -= (Q[:,j:] @ bvvt)
    return Q, np.triu(R[:m])

def HessReduction(A):
    n = max(np.shape(A))
    H = deepcopy(A)
    Q = np.eye(n)
    for k in range(n-2):
        v, beta = householder(H[(k+1):,k])
```

```
        bvvt = beta*colByRowMult(v, v)
        H[(k+1):,k:] -= (bvvt @ H[(k+1):,k:])
        H[:,(k+1):] -= (H[:,(k+1):] @ bvvt)
        Q[(k+1):,k:] -= (bvvt @ Q[(k+1):,k:])
        '''zeroMat = np.zeros((k,n-k))
        P = np.block([[np.eye(k), zeroMat],
                [zeroMat.T, np.eye(len(v))-bvvt]])
        print(P)
        Q = Q*P'''
    return Q, H

def singleShift_QR_Iter(A, init_shift, TOL = 10e-12, max_iter=100):
    Q0, H1 = HessReduction(A)
    n = A.shape[0]
    Qk = Q0
    H = [H1]
    k = 0
    flag = False
    mu = init_shift
    while (not flag) and (k < max_iter):
        shiftMatrix = mu*np.eye(n)
        Qk, Rk = householderReflection(H[k]-shiftMatrix)
        H.append((Rk @ Qk) + shiftMatrix)
        mu = H[k+1][-1][-1]
        k += 1
        if (np.allclose(H[k-1], H[k], atol=TOL)): flag = True
    return np.array(H)

def testEigenVals(trueEigVals, appEigVals):
    truthLabels = ['are not', 'are']
    isEqual = np.allclose(trueEigVals, appEigVals, atol=10e-12)
    print(f'The approximated eigenvalues,\n{appEigVals} \n' +
          f'{truthLabels[np.allclose(trueEigVals, appEigVals)]}' +
          f' roughly the same as the true eigenvalues, \n{trueEigVals}')
    print(f'Error: {np.linalg.norm(trueEigVals - appEigVals)}')

if __name__ == '__main__':
    A = np.array([[1,6,11,16,21],
                  [2,7,12,17,22],
                  [0,8,13,18,23],
                  [0,0,14,19,24],
                  [0,0,0,20,1]], dtype=np.float64)
    eigenVals, eigenVecs = np.linalg.eig(A)
    '''Q, H = HessReduction(A)
    print(H)
```

```
    print(Q)
    print((Q.T)@A@Q)'''
    H = singleShift_QR_Iter(A, init_shift=1, max_iter=30)
    finalH = H[-1]
    appEigVals = np.sort(np.diag(finalH))[::-1]
    testEigenVals(eigenVals, appEigVals)
```

Output:
The approximated eigenvalues,
[45.65319831 6.73399318 1.20203936 -1.25727713 -11.33195372]
are roughly the same as the true eigenvalues,
[45.65319831 6.73399318 1.20203936 -1.25727713 -11.33195372]
Error: 1.588946450819626e-09