Extra Credit: Rayleigh Quotient Iteration with Deflation

10 points

Rayleigh Quotient iteration seeks to compute a given eigenvalue/ eigenvector pair of a matrix A by taking an initial estimate of the eigenvalue/eigenvector pair, μ_0, and $\boldsymbol{b}_0$. The next approximate eigenvector, $\boldsymbol{b}_{i+1}$, is computed as

$$\begin{aligned}\boldsymbol{b}_{i+1} &= (A - \mu_i I)^{-1}\boldsymbol{b}_i \\ \boldsymbol{b}_{i+1} &= \frac{\boldsymbol{b}_{i+1}}{||\boldsymbol{b}_{i+1}||}\end{aligned}$$

while the eigenvalue is updated as

$$\mu_i = \frac{\boldsymbol{b}_i^T A \boldsymbol{b}_i}{\boldsymbol{b}_i^T \boldsymbol{b}_i}$$

Rayleigh Quotient iteration converges to the eigenvector for the eigenvalue "closest" to μ_0. However, deflation can be used to obtain further eigenpairs. Suppose we are trying to find the n eigenvector/eigenvalue pairs of a matrix A.

After eigenvalue λ_1 and corresponding eigenvector $\boldsymbol{x}_1$ are computed, a non-singular matrix H such that $H\boldsymbol{x}_1 = \alpha \boldsymbol{e}_1$ for some scalar multiple α can be used as a similarity transformation of A. In particular, H transforms A into the form

$$HAH^{-1} = \begin{bmatrix} \lambda_1 & \boldsymbol{b}^T \\ 0 & B \end{bmatrix}$$

where B is order $n-1$ with eigenvalues $\lambda_2, \ldots, \lambda_n$. Notice that H need not be a Householder reflector, but nevertheless, this is a common choice. Thus, it is possible to work with B to find the eigenvalue λ_2. Once an eigenvector/eigenvalue pair of B is found such that

$$B\boldsymbol{y} = \lambda_2 \boldsymbol{y}$$

then the original eigenvector for A corresponding to λ_2 may be computed as

$$\boldsymbol{x}_2 = H^{-1}\begin{bmatrix} \alpha \\ \boldsymbol{y} \end{bmatrix}$$

where $\alpha = \dfrac{\boldsymbol{b}^T \boldsymbol{y}}{\lambda_2 - \lambda_1}$. This process may be repeated to compute all of the eigenvector/eigenvalue pairs of A, as we can now apply this same process prescribed above to B in order to deflate λ_2 out of B. Once this occurs, one may apply this process to compute the corresponding eigenvctor of B, and then once again use that to compute the corresponding eigenvector of A.

In this problem, you are tasked with writing a program that, given a symmetric input matrix `A`, computes the eigenvectors and eigenvalues of `A` using Rayleigh Quotient iteration with deflation. Put the eigenvalues in a numpy vector `eig_vals`, along with the corresponds eigenvectors in a numpy array `eig_vecs`, where each column corresponds to an eigenvector. You are given a function `apply_householder` that, for a given eigenvector $\boldsymbol{x}$, computes a similarity of A using the Householder reflector matrix H such that $H\boldsymbol{x} = \alpha \boldsymbol{e}_1$, e.g. `apply_householder(A,x)` creates the similarity transform HAH^{-1} for a Householder reflector such that $H\boldsymbol{x} = \alpha \boldsymbol{e}_1$.

You are also given a function `householder_vec` which, for a given eigenvector $\boldsymbol{x}$ and vector $\boldsymbol{w}$, computes the Householder transformation $H\boldsymbol{w}$ using the Householder reflector such that $H\boldsymbol{x} = \alpha \boldsymbol{e}_1$.

For simplicity, you may assume that the number of eigenvalues and eigenvectors is equal to the size of the array. That is, all eigenvectors and eigenvalues have multiplicity 1.

INPUT:

- `A`: Numpy array of size $n \times n$
- `apply_householder(A,x)`: A function that takes a numpy array `A` and vector `x` and computes the similarity transform HAH^{-1} for a Householder reflector H such that $H\boldsymbol{x} = \alpha \boldsymbol{e}_1$. The return value is a 2D numpy array.
- `householder_vec(w,x)`: A function that takes a vector `w` and vector `x` and computes the Householder transformation $H\boldsymbol{w}$ for a Householder reflector H such that $H\boldsymbol{x} = \alpha \boldsymbol{e}_1$. The return value is a vector.

OUTPUT:

- `eig_vals`: Numpy vector of size n containing the eigenvalues.
- `eig_vecs`: Numpy array of size $n \times n$ containing all the eigenvectors where each column represents an eigenvector of `A`.

NOTE: You may not use any functions from `numpy.linalg` or `scipy.linalg`, with exception to `la.solve` and `la.norm`.

This problem is worth 10 extra credit points for this assignment.

Answer*

```
import numpy as np
import numpy.linalg as la
#get the eigenvalue of A with b
eig_vals = []
def eigv(A,b):
    return np.inner(b,A@b)/np.inner(b,b)

def geteigv(A):#get the first eigenvalue of A matrix
    n = A.shape[0]
    xnow = np.random.rand(n)
    vnow = eigv(A,xnow)
    vprev= 0
    I = np.eye(n)
```

```
    while la.norm(A@xnow - vnow*xnow) > 1e-8:
        xprev = xnow.copy()
        vprev = vnow
        xnow = la.solve(A - vprev*I,xprev)
        xnow = xnow/la.norm(xnow)
        vnow = eigv(A,xnow)
    return vnow,xnow

```

Press F9 to toggle full-screen mode. Set editor mode in user profile (/profile/).

Your answer is correct.

Here is some feedback on your code:

- Execution time: 0.5 s -- Time limit: 5.0 s

Your code ran on relate-05.cs.illinois.edu.

The following code is a valid answer:

```
import numpy as np
import numpy.linalg as la

class RayleighQuotientException(Exception):
    pass

def rq_iter(A, tol=1e-8, max_iterations=1000):
    n = A.shape[0]
    mu = 100*np.abs(np.random.rand())
    b = np.abs(np.random.randn(n))
    iteration = 0
    while la.norm(A@b-mu*b) > tol and iteration < max_iterations:
        iteration += 1
        try:
            b = la.solve(A-mu*np.eye(n),b)
        except la.LinAlgError:
            # restart with a new trial
            mu = 100 * np.abs(np.random.rand())
            b = np.abs(np.random.randn(n))
        b = b/la.norm(b)
        mu = np.inner(b,A@b)/la.norm(b)
    if iteration >= max_iterations:
        raise RayleighQuotientException("Rayleigh Quotient iteration did not conver
        return 0, np.zeros(n)
    return mu, b

def rq_iter_all(A):
    n=A.shape[0]
    eig = np.zeros(n)
    eig_vecs = np.zeros((n,n))
    B=A.copy()
    eig[0] ,eig_vecs[:,0]=rq_iter(A)
    last_eigen_vec = eig_vecs[:,0]
    v_list=[]
    b_list=[]
    for i in range(1,n):
        v_list.append(last_eigen_vec)
        B = apply_householder(B,last_eigen_vec)
        # Extract out b^T, new B from blocks
        b = B[0,1:n-i+1]
        b_list.append(b)
        B=B[1:n-i+1,1:n-i+1].copy()
        try:
            eig[i], last_eigen_vec = rq_iter(B)
        except RayleighQuotientException:
            # make a desparate attempt at correcting the solution
            eig[i], last_eigen_vec = rq_iter(B, 1e-4)
        eigen_vec = last_eigen_vec
        for j in range(0,i):
            index = i-1-j
            b = b_list[index]
            v = v_list[index]
            l = v.shape[0]
            alpha = np.inner(b,eigen_vec)/(eig[index+1]-eig[index])
            rhs = np.zeros(l)
            rhs[0]=alpha
            rhs[1:l]=eigen_vec
```

```
            eigen_vec = householder_vec(rhs,v)
        eig_vecs[:,i]=eigen_vec
    return eig, eig_vecs
eig_vals,eig_vecs = rq_iter_all(A.copy())
idx = eig_vals.argsort()[::-1]
eig_vals = eig_vals[idx]
eig_vecs = eig_vecs[:,idx]
```