Numerics 2 - HW 2

Cooper Simpson

Problem 1

[1]:
```
import numpy as np
import scipy.linalg as spl
```

a).

We want to show that the Hilbert matrix is positive definite. This matrix is defined as follows:

$$H_{ij} = \frac{1}{i+j-1}$$

So the entries are the unit fractions, it is symmetric, and it has these anti-diagonal bands of the same entry.

Proof: Let $\mathbf{H} \in \mathbb{R}^{n\times n}$ be a Hilbert matrix as defined above.

We want to show that $\mathbf{H}$ is positive definite, so take any $\mathbf{z} \neq \mathbf{0} \in \mathbb{R}^n$ and we want to show $\mathbf{z}^T\mathbf{H}\mathbf{z} > 0$.

We can write out what this expression means at the element level.

$$\mathbf{z}^T\mathbf{H}\mathbf{z} = \sum_{i=1}^{n} z_i \cdot (\mathbf{H}\mathbf{z})_i$$

We can then do the same thing for the ith element of the $\mathbf{H}\mathbf{z}$ term.

$$(\mathbf{H}\mathbf{z})_i = \sum_{j=1}^{n} H_{ij} \cdot z_j$$

Putting this together we have the following:

$$\mathbf{z}^T\mathbf{H}\mathbf{z} = \sum_{i=1}^{n} z_i \cdot \sum_{j=1}^{n} H_{ij} \cdot z_j$$

Next, we note that $H_{ij} = \int_0^1 x^{i+j-2}dx$, so we replace this in our sum and pull the integral to the outside because the sums are finite.

$$\implies \mathbf{z}^T\mathbf{H}\mathbf{z} = \int_0^1 \Big(\sum_{i=1}^{n} z_i \cdot \sum_{j=1}^{n} x^{i+j-2} \cdot z_j\Big)dx$$

Splitting up the x term and matching it with its i and j counterparts we can get the following:

$$\mathbf{z}^T\mathbf{H}\mathbf{z} = \int_0^1 \Big(\sum_{i=1}^{n} z_i x^{i-1} \cdot \sum_{j=1}^{n} z_j \cdot x^{j-1}\Big)dx = \int_0^1 \Big(\sum_{k=1}^{n} z_k x^{k-1}\Big)^2 dx$$

The sums are of the same form and thus we can write it as the square of one sum. We assumed that $\mathbf{z}$ was not the zero vector, and thus we can say the following for $x \in [0,1]$:

$$\Big(\sum_{k=1}^{n} z_k x^{k-1}\Big)^2 > 0 \implies \int_0^1 \Big(\sum_{k=1}^{n} z_k x^{k-1}\Big)^2 dx > 0 \implies \mathbf{z}^T\mathbf{H}\mathbf{z} > 0$$

$\therefore$ the Hilbert matrix $\mathbf{H}$ is positive definite.

b).

We implement the Power Method for finding the largest eigenvalue. We then apply this to a Hilbert Matrix.

[3]:
```
#Compute dominant eigenvalue of A
#A is complex nxn matrix
def eigPower(A, tol=1E-6, maxI=1000):
    n = A.shape[0] #Dimension

    q = np.random.randn(n,1) #Initial random vector
    l = q.conj().T@A@q #First e-val estimate

    for i in range(maxI):
        z = A@q
        q = z/np.linalg.norm(z, 2) #Update q
        l_new = q.conj().T@A@q #Update e-val

        if np.abs(l_new-l)/np.abs(l_new) < tol:
            return l_new

        l = l_new

    raise ValueError('Maximum number of iterations exceeded.')
```

Let's try a simple toy example to test our method.

[4]:
```
A = np.array([[0,1],
              [-2,-3]]) #Has e-vals -1 and -2

e = eigPower(A)
print(e)
```

```
[[-1.99999802]]
```

Great! Everything seems to be working fine. Now let's apply this to an order 16 Hilbert matrix.

[5]:
```
#Define Hilbert matrix
n = 16
H = spl.hilbert(n)

e = eigPower(H)
print('Dominant eigenvalue: ',e[0,0])
```

```
Dominant eigenvalue:  1.8600364244729357
```

To check our answer we will also examine the result that Numpy gives us.

[6]:
```
print('Dominant eigenvalue (Numpy): ', max(np.linalg.eig(H)[0]))
```

```
Dominant eigenvalue (Numpy):  1.8600364427433274
```

We see that our method has worked quite well, our dominant eigenvalue is $\boxed{\lambda_{max} \approx 1.86}$.

c).

We modify our power method to find the smallest eigenvalue of a Hilbert matrix of size 16. We then investigate the accuracy of this approach.

[7]:
```
#Compute eigenvalue of A closest to mu
#A is nxn complex matrix
def eigInvPower(A, mu, tol=1E-6, maxI=1000):
    n = A.shape[0] #Dimension

    AI = A-mu*np.eye(n) #A-muI

    q = np.random.randn(n,1) #Initial random vector
    l = q.conj().T@A@q #First e-val estimate

    for i in range(maxI):
        z = np.linalg.solve(AI, q)

        q = z/np.linalg.norm(z, 2) #Update q
        l_new = q.conj().T@A@q #Update e-val

        if np.abs(l_new-l)/np.abs(l_new) < tol:
            return l_new

        l = l_new

    raise ValueError('Maximum number of iterations exceeded.')
```

We will use our Inverse Power method on a Hilber matrix again of size 16. By choosing $\mu = 0$ we will be finding the eigenvalue closest to 0 and thus the smallest eigenvalue.

[16]:
```
#Define Hilbert matrix
n = 16
H = spl.hilbert(n)

e = eigInvPower(H, 0, maxI=1000000)[0,0]
print('Smallest eigenvalue: ', e)
```

```
Smallest eigenvalue:  -6.064627183294988e-18
```

It took a large number of iterations, but eventually our Inverse Power method produces a value on the order of 10^{-18} which is within the set tolerence. However, we note that this eigenvalue

is negative and we just showed that all Hilbert matrices were positive definite (i.e. have positive eigenvalues). Clearly something is going wrong here numerically.

[17]:
```
e_np = min(np.linalg.eig(H)[0])

print('Smallest eigenvalue (Numpy): ', e_np)
```

```
Smallest eigenvalue (Numpy):  -6.966678610511967e-18
```

With Numpy we again see that we are getting a negative value for our eigenvalue which is not possible. To determine the true eigenvalue we can use an exact inverse of our Hilbert matrix.

[18]:
```
H_inv = spl.invhilbert(16)
e_exact = 1/max(np.linalg.eig(H_inv)[0])

print('Smallest eigenvalue (Inverse): ', e_exact)
```

```
Smallest eigenvalue (Inverse):  (9.197419820651449e-23+0j)
```

That seems better. At least we are getting a positive value, although it is very small (on the order of 10^{-23}). Next we look at the error as compared with our calculated eigenvalue.

[19]:
```
print('Absolute Error: ', np.abs(e_exact-e))
print('Relative Error: ', np.abs(e_exact-e)/np.abs(e_exact))
```

```
Absolute Error:  6.0647191574931944e-18
Relative Error:  65939.35338121418
```

Well that is an absolutely horrible relative error. We can also see whether or not our approximation is consistent with the following estimate:

$$\min_{\lambda \in \sigma(H)} |\lambda - \mu| \leq ||E||_2$$

In our case μ is our approximation, λ is the exact value, and E is the perturbation due to machine imprecision (about 10^{-16}). Looking at the absolute error above we see that indeed our estimate is within this bound as it is two orders of magnitude smaller.

What we have learned is that we need to be careful when trusting numerical output from a computer as it is certainly not always accurate.

d)

We assume that a real symmetric matrix $\mathbf{A}$ has eigenvalues such that $\lambda_1 = -\lambda_2$ and are ordered by their magnitude. To find the eigenvectors $\mathbf{v}_1$ and $\mathbf{v}_2$ corresponding to these eigenvalues we suggest a modification of the Power method.

In short our method is as follows: in the standard Power method let $\mathbf{q}_1$ be the $2k$ (i.e. even) iterate, and $\mathbf{q}_2$ be the $2k-1$ (i.e. odd) iterate. Then we have the following:

$$\mathbf{v}_1 \approx \frac{\mathbf{q}_1 + \mathbf{q}_2}{2}$$

$$\mathbf{v}_2 \approx \frac{\mathbf{q}_1 - \mathbf{q}_2}{2}$$

To see why this follows consider the following form for $\mathbf{q}^{(k)}$ where $\mathbf{q}^{(0)} = a_1\mathbf{x}_1 + \cdots + a_n\mathbf{x}_n$ for $\mathbf{x}_i$ being the columns of the similarity transform for the diagonalization of $\mathbf{A}$.

$$\mathbf{q}^{(k)} = \mathbf{A}^k\mathbf{q}^{(0)} = a_1\lambda_1^k\big(\mathbf{x}_1 + \frac{a_2}{a_1}(\frac{\lambda_2}{\lambda_1})^k\mathbf{x}_2 + \sum_{j=3}^{n}\frac{a_j}{a_1}(\frac{\lambda_j}{\lambda_1})^k\mathbf{x}_j\big)$$

We can see that the sum term will go to zero as we iterate, so eventually we will be left with something of the form...

$$\mathbf{q}^{(k)} = \mathbf{A}^k\mathbf{q}^{(0)} = a_1\lambda_1^k\big(\mathbf{x}_1 + \frac{a_2}{a_1}(\frac{\lambda_2}{\lambda_1})^k\mathbf{x}_2\big)$$

Noting that $\lambda_1 = -\lambda_2$ and distributing we have the following:

$$\mathbf{q}^{(k)} = \mathbf{A}^k\mathbf{q}^{(0)} = a_1\lambda_1^k\mathbf{x}_1 + (-1)^k a_2\lambda_1^k\mathbf{x}_2$$

Thus we see that for odd k the last term will be negative and for even k it will be positive. This is what we leverage by combining them in our method above to obtain the two eigenvectors.

e).

We assume that a real symmetric matrix has an eigenvalue of 1 with multiplicity 8, and the remaining eigenvalues are magnitude ≤ 0.1.

We propose a method for finding an orthogonal basis for the 8 dimensional eigenspace corresponding to this eigenvalue of 1. The core of this approach relies on using the Power method to find eigenvectors corresponding to the largest eigenvalue (1), and then creating an othogonal basis from this. There are a number of ways to approach this, but we will outline the most stable.

Take 8 randomly initialized vectors of the same dimension as $\mathbf{A}$ and form a matrix $\mathbf{X}$. Apply the power method to this matrix which is essentially applying it to all 8 vectors individually. At each iteration of the Power method after updating the matrix $\mathbf{X}$, orthogonalize the vectors using Gram-Schmidt, QR, or some other method. Once the matrix $\mathbf{X}$ has converged to a given tolerance the process is complete. You now have 8 orthogonal vectors that correspond to the eigenvalue 1.

We want to determine how many iterations of our process it would take to achieve double precison (i.e. 10^{-16}) accuracy, so we only need to look at the power method. We can see that the rate of convergence at least $\mathcal{O}((0.1)^k)$, given the other eigenvalues are smaller than 0.1. Thus we would need about 16 iterations to achieve double precision.