LAB 9 - ITERATIVE METHODS FOR EIGENVALUES AND MARKOV CHAINS

1. Iterative methods for finding eigenvalues

Finding eigenvectors and eigenvalues of matrices is an important task in modern fields such as engineering, physics, computer science, and data science (in Lab 10 we will investigate one such application to Google's PageRank internet search algorithm). Unfortunately, often the matrices of interest are very large, with tens of thousands of rows and columns. In these cases, the standard method of finding eigenvalues (computing and factoring the characteristic polynomial) is far too slow to be of practical use. In these cases we will again turn to iterative methods for quicker, more practical algorithms. In this lab we will discuss the *power method* for computing eigenvalues and eigenvectors.

The power method is designed to find the 'dominant' eigenvalue and its corresponding eigenvector for an $n \times n$ matrix A. The *dominant eigenvalue* is the eigenvalue with the largest absolute value. This means if a 4×4 matrix has eigenvalues $-4,\ 3,\ 2, -1$ then the power method will converge to $\lambda = -4$ and identify its corresponding eigenvector as well. If, on the other hand, the eigenvalues are $-4,\ 3,\ 2,\ 4$ then there is no 'dominant' eigenvalue, and so power iteration will likely not converge, or may instead converge to either $\lambda = 4$ or $\lambda = -4$ depending on the initial guess for the eigenvector.

Standard power method. The power method begins by selecting a starting guess $\boldsymbol{x}_0$ for our dominant eigenvector. If the matrix A is diagonalizable (something that we will assume going forward) and we suppose that its eigenvalues satisfy $|\lambda_1| \geq |\lambda_2| \geq |\lambda_3| \geq \cdots \geq |\lambda_n|$, with corresponding eigenvectors $\boldsymbol{v}_1,\ \boldsymbol{v}_2,\ \boldsymbol{v}_3, \ldots, \boldsymbol{v}_n$, then any initial guess $\boldsymbol{x}_0$ can be written as:

$$\boldsymbol{x}_0 = c_1\boldsymbol{v}_1 + c_2\boldsymbol{v}_2 + c_3\boldsymbol{v}_3 + \cdots + c_n\boldsymbol{v}_n \tag{1}$$

for some scalars $c_1, \ldots, c_n \in \mathbb{R}$ (recall that any diagonalizable matrix gives us a basis of $\mathbb{R}^n$ consisting of eigenvectors). Power iteration simply applies matrix multiplication to the vector $\boldsymbol{x}_0$ repetitively, i.e.

$$\begin{aligned}
\boldsymbol{x}_1 &= A\boldsymbol{x}_0, \\
\boldsymbol{x}_2 &= A\boldsymbol{x}_1 = A^2\boldsymbol{x}_0, \\
\boldsymbol{x}_3 &= A\boldsymbol{x}_2 = A^3\boldsymbol{x}_0, \\
&\vdots \\
\boldsymbol{x}_k &= A\boldsymbol{x}_{k-1} = A^k\boldsymbol{x}_0.
\end{aligned}$$

Note, by using (1) we can see

$$\begin{aligned}\boldsymbol{x}_k &= A^k\boldsymbol{x}_0 \\ &= A^k(c_1\boldsymbol{v}_1 + c_2\boldsymbol{v}_2 + c_3\boldsymbol{v}_3 + \cdots + c_n\boldsymbol{v}_n) \\ &= c_1A^k\boldsymbol{v}_1 + c_2A^k\boldsymbol{v}_2 + \cdots + c_nA^k\boldsymbol{v}_n \\ &= c_1\lambda_1^k\boldsymbol{v}_1 + c_2\lambda_2^k\boldsymbol{v}_2 + \cdots + c_n\lambda_n^k\boldsymbol{v}_n \\ &= \lambda_1^k\left(c_1\boldsymbol{v}_1 + c_2\left(\frac{\lambda_2}{\lambda_1}\right)^k\boldsymbol{v}_2 + \cdots + c_n\left(\frac{\lambda_n}{\lambda_1}\right)^k\boldsymbol{v}_n\right).\end{aligned}$$

If $\lambda_1 \neq 0$, then since λ_1 is the dominant eigenvalue $\left|\frac{\lambda_j}{\lambda_1}\right| < 1$ for all $j \neq 1$, and hence all of these terms in the expansion of $\boldsymbol{x}_k$ above will get smaller and smaller as k gets larger and larger. All the terms that is, except the first one. This means that as $k \to \infty$ then $\boldsymbol{x}_k \to \lambda_1^k c_1\boldsymbol{v}_1$ so long as $c_1 \neq 0$. Notice that $\lambda_1^k c_1\boldsymbol{v}_1$ is a scalar multiple of the eigenvector $\boldsymbol{v}_1$, and hence it is an eigenvector as well with the same eigenvalue (i.e. it is a dominant eigenvector).

This leads us to the first version of the power method:

(1) For an $n \times n$ diagonalizable matrix A with real eigenvalues ordered as $|\lambda_1| \geq \ldots \geq |\lambda_n|$, define an initial guess $\boldsymbol{x}_0$.
(2) For j from 1 to k (a cutoff value chosen beforehand) compute $\boldsymbol{x}_j = A\boldsymbol{x}_{j-1}$.
(3) Return the vector $\boldsymbol{x}_k$ as the dominant eigenvector.

Problem 1. Define a function `evect_approx1(x_0,k)` which approximates the dominant eigenvector of

$$A = \begin{bmatrix} 1 & 1 \\ 2 & 0 \end{bmatrix}$$

using the power method as described above. Your function should take as input a 2-dimensional NumPy vector `x_0` which represents an initial guess, and an integer `k`. It should return the 2-dimensional NumPy vector `x_k` as described in the algorithm above for this specific 2×2 matrix.

For example, if you call `evect_approx1(np.array([1,9]),10)` it should return

```
array([3752, 3760])
```

Now that you have discovered how to approximate the dominant eigenvector $\boldsymbol{v}_1$ for a given matrix, the next natural question is to determine the dominant eigenvalue λ_1. To do this, we recall that $\boldsymbol{x}_{k+1} = A\boldsymbol{x}_k \approx \lambda_1\boldsymbol{x}_k$. It follows that we can approximate λ_1 by dividing one of the entries of $\boldsymbol{x}_{k+1} = A\boldsymbol{x}_k$ by the same entry of $\boldsymbol{x}_k$.

Problem 2. Define a function `eval_approx1(x_0,k)` which approximates the dominant eigenvalue of

$$A = \begin{bmatrix} 1 & 1 \\ 2 & 0 \end{bmatrix}$$

using the power method. Once again your function should take as input an initial guess 2-dimensional NumPy vector `x_0` and an integer `k` and return an estimate of λ_1. For continuity of grading, compute your estimate of λ_1 using the first entry of $\boldsymbol{x}_{k+1}$ and $\boldsymbol{x}_k$.

For example, if you call `eval_approx1(np.array([1,9]),10)` it should return

```
2.002132196162047
```

1.1. **Normalizing the power method.** The algorithm we introduced above works quite well, but there is a problem that arises as we continue the iterations, i.e. as we take $k \to \infty$.

- To see what happens, use your function `evect_approx1` to compute the $\boldsymbol{x}_k$ approximation of the dominant eigenvector for the matrix described in Problems 1 and 2 above using $k = 61$ and $k = 62$ and the starting guess `np.array([1,9])`.
- Compare the vectors $\boldsymbol{x}_{61}$ and $\boldsymbol{x}_{62}$ you obtain.
- Given that the dominant eigenvalue of the matrix A is $\lambda_1 = 2$, how would you expect the two approximations $\boldsymbol{x}_{61}$ and $\boldsymbol{x}_{62} = A\boldsymbol{x}_{61}$ to be related? Is that what actually happens?

The problem is that each time we multiply our current approximation $\boldsymbol{x}_j$ by the matrix A we are effectively multiplying the dominant eigenvector by the dominant eigenvalue λ_1. This is problematic after several iterations, i.e. λ_1^k can get very large and lead to roundoff errors that are unavoidable with this setup.

To avoid such issues we recall that an eigenvector is never unique, even for its specific eigenvalue. In particular, for the matrix A in Problem 2 the eigenspace corresponding to the dominant eigenvalue is a line in $\mathbb{R}^2$, and any nonzero vector on this line is another dominant eigenvector of A. Put more concisely, any nonzero scalar multiple of a dominant eigenvector is a dominant eigenvector. We can use this fact to our advantage in the power method by 'normalizing' the dominant eigenvector at each step. Because we are only concerned with the direction of the dominant eigenvector, and not its length, we can normalize $\boldsymbol{x}_k$ at each step. There are several possible choices for the normalization, some of which are more practical or computationally efficient than others, but the key is keeping the entries of $\boldsymbol{x}_k$ from growing exponentially with k.

The first approach to this idea is to divide at each step by the 'norm' or length of the eigenvector itself.

(1) For an $n \times n$ matrix A, define an initial guess $\boldsymbol{x}_0$.
(2) For j from 1 to k (a cutoff value chosen beforehand) compute $\boldsymbol{w}_j = A\boldsymbol{x}_{j-1}$.
(3) Define $\boldsymbol{x}_j = \frac{\boldsymbol{w}_j}{\|\boldsymbol{w}_j\|}$ and go to $j+1$. Here $\|\cdot\|$ denotes the norm of the vector.

(4) Return $\boldsymbol{x}_k$ as the dominant eigenvector.

Problem 3. Define a function `norm_evect_approx1(x_0,k)` which approximates the dominant eigenvector and eigenvalue of the same matrix A as above, but this time normalize the eigenvector at each step using the norm or length of the vector. Your function should take as inputs an initial guess `x_0` and an integer `k`, and return the dominant eigenvector estimate `x_k` and an estimate of the eigenvalue (to obtain your eigenvalue estimate divide the first entries of $\boldsymbol{w}_k$ and $\boldsymbol{x}_{k-1}$). You may want to use the `np.linalg.norm` function to calculate the norm of vectors.

For example, if you call `norm_evect_approx1(np.array([1,9]),10)` it should return

```
(array([0.70635334, 0.70785942]), 1.995744680851064)
```

Another option, which is more computationally efficient particularly for very large matrices/vectors, is to normalize by the maximal entry of the vector $\boldsymbol{x}_k$. This means that at each step in the iteration, $\boldsymbol{w}_j = A\boldsymbol{x}_{j-1}$ and then $\boldsymbol{x}_j = \boldsymbol{w}_j/m_j$, where m_j is the maximum absolute value of the entries of $\boldsymbol{w}_j$.

Problem 4. Define a function `norm_approx_gen(M,x_0,k)` that approximates the dominant eigenvector and eigenvalue of any square matrix. Your function should take as input a square matrix `M` of any size, as well as an initial guess `x_0` and integer `k`, and use the power iteration method described above to approximate the dominant eigenvector and eigenvalue of `M` up to the `k`th iterate.

Your function should normalize at each step by dividing the eigenvector approximation by the largest absolute value of its entries. You function should return the approximation $\boldsymbol{x}_k$ of the eigenvector obtained in this way, as well as the eigenvalue approximation obtained by dividing the first entries of $\boldsymbol{w}_k$ and $\boldsymbol{x}_{k-1}$.

Hint: The command `np.abs(a)` *returns a vector whose entries are the absolute value of the entries in the NumPy vector* `a`*. The command* `np.max(a)` *returns the maximum element of the NumPy vector* `a`*. You should also check this against simple examples for which you can compute eigenvalues by hand, perhaps a* 3×3 *matrix from the homework. Keep in mind that power iteration will only work for matrices that have a dominant eigenvalue.*

For example, if you call

```
norm_approx_gen(np.array([[2,4,6],[4,8,0],[1,2,9]]),np.array([1,5,-1]),10)
```

it should return

```
(array([0.98994349, 1.        , 0.98491523]), 12.01744017467949)
```

Extensions and potential issues with the power method. Once the dominant eigenvalue and eigenvector have been found, we can modify the power method to identify the next most dominant eigenvalue and corresponding eigenvector. This is done by searching for the eigenvalues of $A - \lambda_1 I$. This can be applied iteratively to approximate all of the eigenvalues of the matrix A starting with the largest in absolute value and moving to the smallest. Additionally we can use Gaussian elimination and the fact that if $\lambda_1 \neq 0$ is an eigenvalue of A then $\frac{1}{\lambda_1}$ is an eigenvalue of A^{-1} (provided A is invertible) to identify algorithms for computing eigenvalues that are more reliable and general than the standard power method seen here.

Another modification which provides more reliable estimates on the dominant eigenvalue is to note that because $A\boldsymbol{v}_1 = \lambda_1 \boldsymbol{v}_1$ then

$$\frac{(A\boldsymbol{v}_1)\cdot \boldsymbol{v}_1}{\boldsymbol{v}_1 \cdot \boldsymbol{v}_1} = \frac{(\lambda_1 \boldsymbol{v}_1)\cdot \boldsymbol{v}_1}{\boldsymbol{v}_1 \cdot \boldsymbol{v}_1} = \frac{\lambda_1(\boldsymbol{v}_1 \cdot \boldsymbol{v}_1)}{\boldsymbol{v}_1 \cdot \boldsymbol{v}_1} = \lambda_1.$$

The quotient $R(\boldsymbol{x}) = \frac{(A\boldsymbol{x})\cdot \boldsymbol{x}}{\boldsymbol{x}\cdot \boldsymbol{x}}$ is called the Rayleigh quotient and provides a more efficient way to approximate the dominant eigenvalue λ_1.

Problem 5. Define a function `ray_quotient(M,x_0,k)` that once again takes a generic $n \times n$ matrix `M`, initial guess `x_0`, and integer `k`, and applies power iteration, normalizing the current eigenvector approximation at each step by the maximum absolute value, and then applies the Rayleigh quotient at the end to $\boldsymbol{x}_k$ to estimate the dominant eigenvalue of `M`. Your function only needs to return the dominant eigenvalue approximation.

Hint: Since the first thing your function needs to do is to find an eigenvector approximation $\boldsymbol{x}_k$ using the exact same method as in Problem 4, your function `ray_quotient` *may begin by calling the function* `norm_approx_gen` *to compute $\boldsymbol{x}_k$. Then simply apply the Rayleigh quotient to $\boldsymbol{x}_k$ to find the eigenvalue approximation.*

For example, if you call

```
ray_quotient(np.array([[2,4,6],[4,8,0],[1,2,9]]),np.array([1,5,-1]),10)
```

it should return

```
12.001490204299047
```

Even with these extensions, the power method is not perfect and is known to fail for several different matrices and/or initial guesses $\boldsymbol{x}_0$. For instance, what happens if you try to

approximate either the dominant eigenvalue or dominant eigenvector of

$$\begin{bmatrix} 3 & 2 & -2 \\ -1 & 1 & 4 \\ 3 & 2 & -5 \end{bmatrix} \tag{2}$$

starting with an initial guess of $\boldsymbol{x}_0 = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$?

- Try computing different eigenvector/eigenvalue approximations for the above matrix and starting guess $\boldsymbol{x}_0$, using the function `norm_approx_gen` and different values of k. Does it appear as though the process is converging?

Problem 6. Compute $\boldsymbol{x}_3$ and $\boldsymbol{x}_4$ for the matrix and initial vector $\boldsymbol{x}_0$ defined in (2) using the normalized power method function `norm_approx_gen` you defined in Problem 4. Save the values of these vectors as `x_vect_3` and `x_vect_4` respectively.

There are several issues that may arise, such as an initial guess that has no component related to the dominant eigenvalue, meaning $c_1 = 0$ in (1). In practice this typically only serves to slow down convergence, as roundoff error will eventually allow for some nonzero c_1 to arise which will then dominate further iterations. In the meantime, it may appear that the algorithm is converging to the dominant eigenvector when it is in fact converging to the second most dominant eigenvector $\boldsymbol{v}_2$.

In summary, iterative techniques like the straightforward power method shown here are quite useful when approximating eigenvalues and eigenvectors of a given matrix. In particular, if you know the structure of the matrix (such as whether or not all eigenvalues are real) then one particular algorithm or another will work quite well. On the other hand it is important to keep in mind the shortcomings of each of these methods, and understand when they may fail. In practice inverse iteration (when the estimates of $1/\lambda_1$ are used) is coupled with the Rayleigh quotient to accelerate convergence to the dominant eigenvalue. The next couple of eigenvalues can also be sought via iterative techniques as discussed above, but there are other algorithms that can be used if all the eigenvalues/eigenvectors of a given matrix are sought.

2. Markov Chains

Markov Chains are a standard tool to model changes or evolution in a system with a discrete set of states. As we will see below the dominant eigenvector for a Markov Chain is a very important aspect of the system.

To motivate this topic we will use a specific example. Suppose Netflix is trying to determine how many customers they are losing to Hulu (for the sake of example suppose these are the only two streaming services available on the market). They send out a monthly survey to determine how frequently people switch streaming services. The results of the survey indicate that every month 80% of Netflix users continue to use Netflix while 20% switch to Hulu, and 70% of Hulu users continue to use Hulu while 30% of them switch to Netflix. If in January, Netflix had 80 million subscribers and Hulu had 120 million, then in February we would expect

Hulu to have:

$$(0.7)120 + (0.2)80 = 100$$

million subscribers, and Netflix would have

$$(0.3)120 + (0.8)80 = 100$$

million subscribers as well.

This can easily be described by the matrix-vector equation:

$$\begin{bmatrix} 0.7 & 0.2 \\ 0.3 & 0.8 \end{bmatrix} \begin{bmatrix} 120 \\ 80 \end{bmatrix} = \begin{bmatrix} 100 \\ 100 \end{bmatrix}.$$

If we let

$$P = \begin{bmatrix} 0.7 & 0.2 \\ 0.3 & 0.8 \end{bmatrix}$$

be the matrix above, which we call the *transition matrix*, and

$$\boldsymbol{x}_0 = \begin{bmatrix} 120 \\ 80 \end{bmatrix},$$

which we call the *initial state*, then we can see that the number of subscribers to the two streaming services in month k will be given by $\boldsymbol{x}_k = P\boldsymbol{x}_{k-1}$, with $\boldsymbol{x}_1 = P\boldsymbol{x}_0$. The transition matrix P defines a *discrete Markov chain*, i.e. a chain that describes how a system evolves from one state to another in a probabilistic sense. The entries in each column of a transition matrix for a Markov chain must add up to 1.

Problem 7. Write a function `subscriber_vals(x_0,k)` that takes as input an initial state `x_0` and a number of months `k` and returns the number of million subscribers for each service in the `k`th month using the model described above.

Notice that the total number of subscribers each month should add up to the number of subscribers total in month 0.

For example, if you input the code `subscriber_vals(np.array([95,102]),10)` it should return

```
array([ 78.81582031, 118.18417969])
```

There are clearly some issues with the way we have presented this model. We are neglecting any other competitors to the two streaming services mentioned, and we are ignoring the introduction of new customers into the system, i.e. we have ignored the fact that some people may choose to not stream anything (or to use their relative's login and not pay for their own) and that some people will want to get started on a streaming service when they haven't been using one in the past (their cousin finally changed their password). One way to avoid some of these issues with our current model is to consider the percentage of individuals using each service rather than the total number, i.e. rather than stating that Netflix had 80 million subscribers in January we would say that Netflix had 40% of the subscribers in January. This would mean

that our new $\boldsymbol{x}_0$ would be given by

$$\boldsymbol{x}_0 = \begin{bmatrix} 0.6 \\ 0.4 \end{bmatrix}.$$

Problem 8. What is the proportion of Netflix subscribers to total customers after 6 months with the specified model and initial state? Save this quantity as the variable `netflix_subs6` in your Colab notebook. (This should be a value between 0 and 1.) You may use previously defined functions to solve this problem.

After 12 months, what percentage of subscribers does Netflix now have? What about 24 months, or 36?

This example illustrates what happens for any Markov chain with a specified transition matrix P. For any initial vector $\boldsymbol{x}_0$ representing the probability of starting in each of the n potential states, $\boldsymbol{x}_k = P^k \boldsymbol{x}_0$ will converge to a single vector $\boldsymbol{v}$ which describes the steady state probability of the system, i.e. the probability of being in each state after a sufficiently long time.

For example, if our initial state is

$$\boldsymbol{x}_0 = \begin{bmatrix} 0.6 \\ 0.4 \end{bmatrix}$$

this means that at month 0 that 60% of customers will subscribe to Hulu, while 40% will subscribe to Netflix. As we go further and further into the future, we see that $\boldsymbol{x}_k = P^k \boldsymbol{x}_0$ begins getting closer and closer to the vector

$$\boldsymbol{v} = \begin{bmatrix} 0.4 \\ 0.6 \end{bmatrix}.$$

This means that in the long-run, we expect that about 40% of customers will subscribe to Hulu, while 60% subscribe to Netflix. Moreover, notice that

$$P\boldsymbol{v} = \begin{bmatrix} 0.7 & 0.2 \\ 0.3 & 0.8 \end{bmatrix} \begin{bmatrix} 0.4 \\ 0.6 \end{bmatrix} = \begin{bmatrix} 0.4 \\ 0.6 \end{bmatrix} = \boldsymbol{v},$$

so the vector $\boldsymbol{v}$ is a an eigenvector of the matrix P with eigenvalue 1. This implies that $\boldsymbol{v}$ is a steady state vector, meaning that if the proportions of subscribers are given by the vector $\boldsymbol{v}$, then the percentage of customers with each streaming service will stay constant from month to month afterwards (even though customers will be flowing back and forth between the streaming services, the total number with each will stay the same).

It turns out what is happening is exactly power iteration for a special type of matrix (the transition matrix for a Markov Chain). The steady state probability vector $\boldsymbol{v}$ is the dominant eigenvector of the transition matrix P with dominant eigenvalue $\lambda_1 = 1$, and every initial state will converge to this steady state.

We now consider a slightly more complicated problem. Suppose that in a remote region in the midwest there is a bullfrog who frequents 4 different ponds throughout the summer. The first pond (pond A) is his favorite, so if he is in pond A then he is 80% likely to stay there the next day and 5% likely to move to pond B, 10% likely to move to pond C, and 5% likely to move to pond D. Once the bullfrog is in pond B, he is very unsettled, and is only 20% likely to stay there the next day, 50% likely to go to pond A, 10% likely to go to pond C, and 20% likely to visit pond D. From pond C, he is 30% likely to stay, 30% likely to visit pond A, 10%

likely to visit pond B, and 30% likely to visit pond D. From pond D, he is 60% likely to stay in pond D, 10% likely to visit pond B, 10% likely to visit pond C, and 20% likely to visit pond A.

Problem 9. Construct the transition matrix describing the pond this bullfrog will be in, and save it as a NumPy array called `trans_matrix`. When ordering your variables to create the matrix, order the ponds as A, B, C, then D.

- Find the steady state probability vector for this transition matrix, and determine what percentage of the bullfrog's time will be spent in pond C in the long run (regardless of which pond he starts in). Recall that finding the steady state probability vector is the same things as finding the eigenvector corresponding to the dominant eigenvalue 1.