MATH3204: Assignment 02

Due: 14-Sep-2020 @11:59pm

1. Spectral Properties

(a) Show that for any $\boldsymbol{A} \in \mathbb{C}^{n\times n}$, we have **[10 marks]**

$$\text{Trace}(\boldsymbol{A}\boldsymbol{A}^*) \geq \sum_{i=1}^{n} |\lambda_i|^2,$$

where $\{\lambda_i\}_{i=1}^n$ are the eigenvalues of $\boldsymbol{A}$.

(b) In the above, when does the equality hold? **[5 marks]**

2. Special Matrix Types **[5 marks each]**

(a) For any $\boldsymbol{A} \in \mathbb{C}^{m\times n}$, show that $\boldsymbol{A}\boldsymbol{A}^\dagger$ is the orthogonal projection onto Range($\boldsymbol{A}$) (so you have to show that not only is it an orthogonal projection, but it also projects onto the range of $\boldsymbol{A}$).[a]

(b) For any $\boldsymbol{A} \in \mathbb{C}^{m\times n}$, show that $\boldsymbol{A}\boldsymbol{A}^*$ is positive semi-definite.

(c) Suppose $\boldsymbol{A} \in \mathbb{C}^{n\times n}$ and $\boldsymbol{A} \succeq \boldsymbol{0}$. Show that, for the eigenvalues of $\boldsymbol{A}$, we have $\lambda_i(\boldsymbol{A}) \geq 0,\ \forall i$.

(d) Is the converse of the previous question also true? More specifically, for any $\boldsymbol{A} \in \mathbb{C}^{n\times n}$, is it true that if $\lambda_i(\boldsymbol{A}) \geq 0,\ \forall i$, then we must have $\boldsymbol{A} \succeq \boldsymbol{0}$. If so, prove it, otherwise, give a counter-example.

[a] Such projection is actually unique, but you will not need to show this.

3. Matrix Decomposition **[5 marks]**

Let $\boldsymbol{A} \in \mathbb{C}^{m\times n}$ where $m \geq n$. Prove that we can decompose $\boldsymbol{A}$ such that $\boldsymbol{A} = \boldsymbol{U}\boldsymbol{Q}$, where $\boldsymbol{Q}_{n\times n} \succeq \boldsymbol{0}$ and $\boldsymbol{U}_{m\times n}$ has orthonormal columns (**Hint:** You can use SVD!). [a]

[a] This decomposition is known as the Polar Decomposition, which like SVD, applies to *any* matrix $\boldsymbol{A} \in \mathbb{C}^{m\times n}$. However, for the purpose of this question, we only considered $m \geq n$. This decomposition is best motivated in analogy to the polar from of a non-zero complex number $z = re^{i\theta}$, where $r > 0$ and $e^{i\theta}$ is unit length. Intuitively, the polar decomposition separates $\boldsymbol{A}$ into a component that stretches

the space along a set of orthogonal axes, represented by $\boldsymbol{Q}$, and a rotation (with possible reflection) represented by $\boldsymbol{U}$.

4. Linear Systems and Explicit Matrix Inverse **[5 marks each]**

In this question, you will study one of the reasons why forming the matrix-inverse explicitly is so much frowned upon.

(a) For $n = 100, 200, 500, 1000, 2000, 5000, 10000$, consider a series of linear systems $\boldsymbol{A}_n\boldsymbol{x}_n = \boldsymbol{b}_n$ where $\texttt{A}_\texttt{n} = \texttt{randn}(\texttt{n}, \texttt{n}), \texttt{b}_\texttt{n} = \texttt{ones}(\texttt{n}, 1)$ if you are coding in `Matlab` or $\texttt{A}_\texttt{n} = \texttt{np.random.randn}(\texttt{n}, \texttt{n}), \texttt{b}_\texttt{n} = \texttt{np.ones}((\texttt{n}, 1))$ in `Python`. Plot the time it takes to solve $\boldsymbol{A}_n\boldsymbol{x}_n = \boldsymbol{b}_n$ using `Matlab`'s `inv` or `Python`'s `np.linalg.inv` as a function of n. On the same plot, do the same using `Matlab`'s celebrated "backslash" or `Python`'s `np.linalg.solve`. (**Note:** If your system's memory cannot handle $n = 10000$, then only do this up to $n = 5000$. However, the larger the size is, the better your can see the pattern.)

(b) Does your observation in comparing the time for these two implementations roughly match that predicted by the theory in terms of "flops"? Provide some explanation for your answer.

On paper, these functions are implementing the same thing, namely $\boldsymbol{x}_n = \boldsymbol{A}_n^{-1}\boldsymbol{b}_n$, but on computer, they perform differently! This is one of the places where numerical linear algebra differs from linear algebra.

(c) Now generate a similar plot and make similar comparisons as above. However, this time construct $\boldsymbol{A}_n$ using `sprandn(n,n,0.1)` or `sp.sparse.random(n,n,0.1, format='csc')` if you are using `Matlab` or `Python`, respectively. These commands construct random but sparse matrices. Feel free to check the relevant documentation and see what each argument of these functions is used for. The third argument controls the density of the matrix, i.e., what percentage of the entries of this matrix consists of non-zeros values. You can also compute its number of non-zero entries by the commands `nnz(A)` in `Matlab` or `A.nnz` in `Python`. Don't make it too sparse, as it might end up being singular; in this question `0.1` sparsity level is good to see what is going on. If you are coding in `Python`, your relevant functions for solving the linear systems are `sp.sparse.linalg.inv` and `sp.sparse.linalg.spsolve`.

(d) Does your observation in comparing the time for these two implementations roughly match that predicted by the theory in terms of "flops"? Provide some explanation for your answer.

Side note 1: You might have actually seen that for the same size matrix, sparse solvers take longer than when the matrix is densely stored. This is has in part to do with the way sparse matrices are stored and the issues related to the cache/memory access/locality, etc. So sparse matrices are easy to store and more efficient to perform many computations with, however, depending on the circumstances, it might be more time consuming to solve sparse linear systems.

Side note 2: In the same spirit as the side note above, for our question here, it might actually be a lot more efficient to first convert $\boldsymbol{A}_n$ to a dense matrix using `A.todense()`, and then use `sp.linalg.inv` and `sp.linalg.solve` (feel free to try this, but no need to reprt anything on this side note). But, for very large n depending on your memory, storing the dense matrix might not be possible at all, and we have to work with sparse matrix representations. We assume this is the situation here.

5. Stationary Linear System Solver **[5 marks each]**

For solving $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$, consider the stationary Richardson iteration

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \alpha\left(\boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_k\right).$$

For this question, we assume that all eigenvalues of $\boldsymbol{A}$ are real.

(a) Find the iteration matrix.

(b) Show that if $\lambda_{\min} < 0 < \lambda_{\max}$, the method will always be divergent for some initial iterate.

(c) So assuming that all eigenvalues are positive, find the spectral radius of the iteration matrix in terms of $\alpha, \lambda_{\min}$ and $\lambda_{\max}$?

(d) Find $\alpha_{\min} < \alpha_{\max}$ such that the method converges for any $\alpha_{\min} < \alpha < \alpha_{\max}$. In other words, find the largest range for α that ensures the method always stays convergent.

(e) What is the optimal value for α? In other words find α_{opt} that minimizes the spectral radius of the iteration matrix.

(f) Suppose $\boldsymbol{A} \succ \boldsymbol{0}$. Using this optimal step-size, obtain the optimal convergence rate in terms of the matrix condition number.

(g) Implement the stationary Richardson iteration and apply it to the matrix generated using the following code. For this assignment, we use `beta = gamma = 0`, `N = 50`, $\boldsymbol{x}_0 = \boldsymbol{0}$, and a right hand-side vector $\boldsymbol{b}$ containing all ones. Compare the performance using three step-sizes: $\alpha_{\text{opt}}, 0.5\alpha_{\text{opt}}$, and $1.1\alpha_{\text{opt}}$. What do you observe? Terminate the algorithms if the Euclidean norm of the residual vector

$\|\boldsymbol{r}_k\|_2 = \|\boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_k\|_2 \leq 10^{-12}$ or if a maximum number of iteration of $20,000$ is reached.

```
function A=lcd(beta,gamma,N)
%
% Simple code to generate test matrices for this question and maybe some later assignments
% Usage:
%   to generate symmetric positive definite, call with beta=gamma=0
% to generate nonsymmetric call, e.g., with beta=gamma=1/2
%
% Note: output matrix is of size N^2-by-N^2

ee=ones(N,1);
a=4; b=-1-gamma; c=-1-beta; d=-1+beta; e=-1+gamma;
t1=spdiags([c*ee,a*ee,d*ee],-1:1,N,N);
t2=spdiags([b*ee,zeros(N,1),e*ee],-1:1,N,N);
A=kron(speye(N),t1)+kron(t2,speye(N));
end

```

For `Matlab` use the above code snippet.

```
import numpy as np
import scipy as sp
def lcd(beta,gamma,N):
#   Simple code to generate test matrices for this question and maybe some later assignments
#   Usage:
#     to generate symmetric positive definite, call with beta=gamma=0
#     to generate nonsymmetric call, e.g., with beta=gamma=1/2
#
#   Note: output matrix is of size N^2-by-N^2

    ee = np.ones((1, N))
    a, b, c, d, e = 4, -1 - gamma, -1 - beta, -1 + beta, -1 + gamma
    t1 = sp.sparse.spdiags(np.vstack([c * ee, a * ee, d * ee]), np.arange(-1,2), N, N)
    t2 = sp.sparse.spdiags(np.vstack([b * ee, np.zeros((1, N)), e * ee]), np.arange(-1,2), N, N)
    A = sp.sparse.kron(sp.sparse.eye(N,N), t1) + sp.sparse.kron(t2, sp.sparse.eye(N,N))


```

For `Python` use the above code snippet.

(h) Implement the accelerated variant of the Richardson iteration and, on the same test problem and starting from the same $\boldsymbol{x}_0$ as above, compare its performance with the stationary version. For both methods, use the step-size $\alpha = 2/(\lambda_{\min} + \lambda_{\max})$.

Bonus Question

6. Eigenvalue/Eigenvector Computations

In lectures, we discussed eigenvalues and singular values of a matrix. However, due to time constraints, we are unfortunately unable to study in detail how we can actually compute these quantities. Recall that computing eigenvalues is equivalent to finding roots of polynomials, which for $n \geq 5$, can only be hoped to be approximated. In this question, you will get a flavor of some methods that can be used for approximating the spectrum of a matrix.

(a) Read Appendix A.1. You will not be marked for the questions in Appendix A.1: those questions are just there for you to experiment with the method and learn it better. As you see, the experiments in Appendix A.1 only consider the case where there is a simple dominant eigenvalue, i.e., when $|\lambda_1| > |\lambda_2| \geq |\lambda_3| \geq \ldots \geq |\lambda_n|$. Now you will investigate what happens when two dominant eigenvalues of equal magnitude exist, i.e., when $|\lambda_1| = |\lambda_2| > \lambda_3| \geq \ldots \geq |\lambda_n|$. Consider the three cases below. Comment on the convergence of the Power Method (Algorithm 1) in each case? Justify your answer. You can do this theoretically and/or by numerical demonstrations. **[4 marks each]**

i. $\lambda_1 = \lambda_2 = \lambda$ and the geometric multiplicity of λ is 2.

ii. $\lambda_1 = -\lambda_2$.

iii. $\lambda_1 = \overline{\lambda_2}$, where $\overline{x}$ denotes the conjugate of the complex number x.

(b) Now, read Appendix A.2. For the matrix $\boldsymbol{A}$ in Appendix **A.1-Q1**, run the inverse iteration with two fixed parameters: $\gamma = 33$ and $\gamma = 35$. Plot the absolute errors in estimating the dominant eigenvalue. Which shift parameter was more effective? **[5 marks]**

(c) Can inverse iteration using an appropriate shift overcome the challenges you identified above for the case where two or more dominant eigenvalues of equal magnitude exist? **[3 marks]**

100 marks in total + 20 extra marks for the bonus

Note:

- This assignment counts for 15% of the total mark for the course.

- You don't have to do the bonus question, but it may help bring up your mark if you miss some points on other questions.
- Your mark for this assignment will be calculated as

$$\min\{\text{Total marks obtained for all the questions}, 100\} \times 0.15.$$

So with or without bonus, you can only get a maximum of 15%.

- Although not mandatory, if you could type up your work, e.g., `LaTex`, it would be greatly appreciated.
- Show all your work and attach your code and all the plots (if there is a programming question).
- Combine your solutions, all the additional files such as your code and numerical results, along with your coversheet, **all in one single PDF file**.
- Please submit your single PDF file on Blackboard.

A Eigenvalue/Eigenvector Computations

Generally speaking, there are two main classes of numerical methods for approximating eigenvalues and eigenvectors of a given matrix $\boldsymbol{A}$: (i) *partial* methods, which compute the *extremal* eigenvalues of $\boldsymbol{A}$, i.e., those with maximum and minimum magnitude, and (ii) *global* methods, which approximate the whole spectrum of $\boldsymbol{A}$. In this practical, we only consider the former class of methods. The global methods require more work and analysis and cannot be given a fair treatment here.

A.1 Power Method

The power method is a classical method for approximating the extremal eigenvalues of a matrix, and hence is a partial method. It was the original algorithm that Google used to calculate the PageRank of documents in their search engine! Of course, now they use much more sophisticated methods. Twitter uses it to show users recommendations of who to follow. So it is an important algorithm that is actually used quite often. Let's see how it works.

Suppose $\boldsymbol{A} \in \mathbb{C}^{n\times n}$ is diagonalizable and let $\{\boldsymbol{v}_i\}_{i=1}^n$ be its eigenvectors. Suppose further that we have

$$|\lambda_1| > |\lambda_2| \geq |\lambda_3| \geq \ldots \geq |\lambda_n|. \tag{1}$$

So here λ_1 is the largest eigenvalue in magnitude, which is often called the *dominant* eigenvalue of $\boldsymbol{A}$ and its corresponding eigenvector is called the dominant eigenvector. Since $\boldsymbol{A}$ is diagonalizable, it has n linearly independent eigenvectors, and thus we can write any $\boldsymbol{x}_0$ as

$$\boldsymbol{x}_0 = \sum_{i=1}^{n} \alpha_i \boldsymbol{v}_i,\ \alpha_i \in \mathbb{C},\ i = 1, 2, \ldots, n.$$

So it follows that

$$\begin{aligned}\boldsymbol{A}^k\boldsymbol{x}_0 &= \sum_{i=1}^{n} \alpha_i \lambda_i^k \boldsymbol{v}_i \\ &= \alpha_1\lambda_1^k \left(\boldsymbol{v}_1 + \sum_{i=2}^{n} \left(\frac{\alpha_i}{\alpha_1}\right)\left(\frac{\lambda_i}{\lambda_1}\right)^k \boldsymbol{v}_i \right),\ k = 1, 2, \ldots.\end{aligned}$$

Since $|\lambda_i/\lambda_1| < 1,\ i = 2, \ldots, n$, as k increases, the vector $\boldsymbol{A}^k\boldsymbol{x}_0$ tends to be increasingly more aligned, i.e., parallel, to the dominant eigenvector of $\boldsymbol{A}$ and to have smaller and smaller components in other directions. Now let us define

$$\boldsymbol{x}_k \triangleq \frac{\boldsymbol{A}^k\boldsymbol{x}_0}{\|\boldsymbol{A}^k\boldsymbol{x}_0\|} = \operatorname{sign}(\alpha_1\lambda_1^k) \left(\frac{\boldsymbol{v}_1 + \sum_{i=2}^{n} \left(\frac{\alpha_i}{\alpha_1}\right)\left(\frac{\lambda_i}{\lambda_1}\right)^k \boldsymbol{v}_i}{\left\| \boldsymbol{v}_1 + \sum_{i=2}^{n} \left(\frac{\alpha_i}{\alpha_1}\right)\left(\frac{\lambda_i}{\lambda_1}\right)^k \boldsymbol{v}_i \right\|} \right).$$

Hence, as $k \to \infty$, the vector $\boldsymbol{x}_k$ must become completely parallel to $\boldsymbol{v}_1$. We have the following result (you may skip the theorem and its proof if you wish, but if you decided to look at it, then note that any vector norm can be used below).

Theorem 1. *Let $\boldsymbol{A} \in \mathbb{C}^{n\times n}$ be a diagonalizable matrix whose eigenvalues satisfy (1), and suppose that $\boldsymbol{x}_0$ is chosen such that $\alpha_1 \neq 0$. Then, there exists a constant $C > 0$ such that*

$$\|\tilde{\boldsymbol{x}}_k - \boldsymbol{v}_1\| \leq C \left|\frac{\lambda_2}{\lambda_1}\right|^k, \ k \geq 1,$$

where

$$\tilde{\boldsymbol{x}}_k \triangleq \left(\frac{\left\|\boldsymbol{A}^k \boldsymbol{x}_0\right\|}{\alpha_1 \lambda_1^k}\right) \boldsymbol{x}_k = \boldsymbol{v}_1 + \sum_{i=2}^{n} \left(\frac{\alpha_i}{\alpha_1}\right) \left(\frac{\lambda_i}{\lambda_1}\right)^k \boldsymbol{v}_i$$

Proof. Without loss of generality, we can consider all the eigenvectors of $\boldsymbol{A}$ to have unit Euclidean length, i.e., $\|\boldsymbol{v}_i\| = 1, \ i = 1, \ldots, n$. So we have

$$\begin{aligned}
\|\tilde{\boldsymbol{x}}_k - \boldsymbol{v}_1\| = \left\| \boldsymbol{v}_1 + \sum_{i=2}^{n} \left(\frac{\alpha_i}{\alpha_1}\right) \left(\frac{\lambda_i}{\lambda_1}\right)^k \boldsymbol{v}_i - \boldsymbol{v}_1 \right\| &= \left\| \sum_{i=2}^{n} \left(\frac{\alpha_i}{\alpha_1}\right) \left(\frac{\lambda_i}{\lambda_1}\right)^k \boldsymbol{v}_i \right\| \\
&\leq \sum_{i=2}^{n} \left(\left|\frac{\alpha_i}{\alpha_1}\right| \left|\frac{\lambda_i}{\lambda_1}\right|^k \|\boldsymbol{v}_i\| \right) \\
&\leq \left|\frac{\lambda_2}{\lambda_1}\right|^k \underbrace{\left(\sum_{i=2}^{n} \left|\frac{\alpha_i}{\alpha_1}\right| \right)}_{C}.
\end{aligned}$$

□

Remark 1. The assumption on diagonalizability is not really needed as power method converges on defective matrices as well. However, the convergence may be painfully slow, and definitely much slower than what the ratio of two dominant eigenvalues indicates for diagonalizable matrices.

Remark 2. But how can we ensure that $\alpha_1 \neq 0$, which is a priori impossible to check in practice (since we don't know $\boldsymbol{v}_1$)? It is a basic property from probability theory that if $\boldsymbol{x}_0$ is picked randomly, e.g., `rand` or `randn`, the condition $\alpha_1 \neq 0$ will be satisfied with probability one![a] In fact, even if it is not picked at random, the unavoidable round-off errors ensure that $\boldsymbol{x}_0$ will contain some component in the direction of $\boldsymbol{v}_1$ and hence

$\alpha_1 \neq 0$.

[a]Here, we have to be a bit careful and mention that the probability distribution from which we sample $\boldsymbol{x}_0$ has to satisfy certain loose conditions, e.g., the probability of picking a vector in the space orthogonal to the largest eigenvector has to be zero. This technical condition is satisfied for almost any "day-to-day" probability distributions.

Remark 3. The important thing to remember here is that the convergence rate is determined by the ratio $|\lambda_2/\lambda_1|$.

Since $\tilde{\boldsymbol{x}}_k$ converges to $\boldsymbol{v}_1$, we expect that the corresponding Rayleigh quotients, i.e.,

$$\frac{\langle \tilde{\boldsymbol{x}}_k, \boldsymbol{A}\tilde{\boldsymbol{x}}_k \rangle}{\|\tilde{\boldsymbol{x}}_k\|^2} = \langle \boldsymbol{x}_k, \boldsymbol{A}\boldsymbol{x}_k \rangle = \nu_k,$$

would approach λ_1. Indeed, we have $\lim_{k\to\infty} \nu_k = \lambda_1$, and this convergence is faster when the ratio $|\lambda_2/\lambda_1|$ is smaller.
If $\boldsymbol{A} \in \mathbb{R}^{n\times n}$ and symmetric, then one can show that under the same assumptions as in Theorem 1, we have

$$|\lambda_1 - \nu_k| \leq |\lambda_1 - \lambda_n| \tan^2(\theta_0) \left|\frac{\lambda_2}{\lambda_1}\right|^{2k},$$

where $\theta_0 = \arccos(|\langle \boldsymbol{x}_0, \boldsymbol{v}_1 \rangle|) \neq 0$.

Remark 4. So for a real symmetric matrix, the convergence of ν_k to λ_1 using power iterations is quadratically fast.

Algorithm 1 Power Method

1: **Input:** Initial matrix $\boldsymbol{A}$ and initial guess $\boldsymbol{x}_0$
2: **for** $k = 1, \ldots$ until convergence **do**
3: $\quad \boldsymbol{y} = \boldsymbol{A}\boldsymbol{x}_{k-1}$
4: $\quad \boldsymbol{x}_k = \boldsymbol{y}/\|\boldsymbol{y}\|$
5: $\quad \nu_k = \langle \boldsymbol{x}_k, \boldsymbol{A}\boldsymbol{x}_k \rangle$
6: **end for**
7: **Output:** Approximate eigenpair $(\nu_k, \boldsymbol{x}_k)$

The power method is as simple as it gets (hence not the best method out there). Also, note that power method does not require explicit knowledge of $\boldsymbol{A}$; a procedure providing matrix-vector-product (MVP), i.e., just being able to compute $\boldsymbol{A}\boldsymbol{x}$ for any $\boldsymbol{x}$, suffices.

Remark 5. Even though the convergence in Theorem 1 is given in terms of $\|\tilde{\boldsymbol{x}}_k - \boldsymbol{v}_1\|$, what we can actually compute in Algorithm 1 is $\boldsymbol{x}_k$ and not $\tilde{\boldsymbol{x}}_k$. Now, since the eigenvectors can be scaled arbitrarily, e.g., $\boldsymbol{v}_1$ and $-\boldsymbol{v}_1$ are both eigenvectors, an appropriate measure of error in the estimate $\boldsymbol{x}_k$ of the true eigenvector $\boldsymbol{v}_1$ should measure how "parallel" $\boldsymbol{x}_k$ is to $\boldsymbol{v}_1$. As a result, in what follows, we adopt the following error for measuring the accuracy of the eigenvector estimate $\boldsymbol{x}_k$,

$$\mathrm{Err}(\boldsymbol{x}_k, \boldsymbol{v}_1) = 1 - \frac{\langle \boldsymbol{x}_k, \boldsymbol{v}_1 \rangle}{\|\boldsymbol{x}_k\| \, \|\boldsymbol{v}_1\|}.$$

It is easy to see that $\mathrm{Err}(\boldsymbol{x}_k, \boldsymbol{v}_1) = \mathrm{Err}(\tilde{\boldsymbol{x}}_k, \boldsymbol{v}_1)$.

Form the following two matrices, $\boldsymbol{A}$ and $\boldsymbol{B}$,

$$\boldsymbol{A} = \mathrm{diag}\,(1, 2, 3, \ldots, 30, 31, 32)\,, \quad \boldsymbol{B} = \mathrm{diag}\,(1, 2, 3, \ldots, 30, 30, 32)\,,$$

where "diag($\boldsymbol{v}$)" creates a diagonal matrix with element of $\boldsymbol{v}$ on the diagonal.

Q.1 What are their eigenvalues (you don't need to use any code to find their eigenvalues)?

Q.2 Implement the power iteration and apply it to compute the corresponding largest eigenvalues of $\boldsymbol{A}$ and $\boldsymbol{B}$. Plot the absolute errors $|\nu_k - \lambda_1(\boldsymbol{A})|$ and $|\nu_k - \lambda_1(\boldsymbol{B})|$ as a function of iterations until the error in less than, say, 10^{-6}. What do you observe and why?

Now, consider the following matrix

$$\boldsymbol{A} = \begin{bmatrix} 15 & -2 & 2 \\ 1 & 10 & -3 \\ -2 & 1 & 0 \end{bmatrix}.$$

Q.3 Find its eigenvalues and eigenvectors using `eig` or `np.linalg.eig` in `Matlab` or `Python`, respectively.

Q.4 Run the power iteration with $\boldsymbol{x}_0 = [1, 1, 1]^\intercal$ until $|\nu_k - \lambda_1(\boldsymbol{A})| \leq 10^{-10}$. Record $\mathrm{Err}(\boldsymbol{x}_k, \boldsymbol{v}_1)$ for each k as well. Plot the errors in estimating the dominant eigenvalue and eigenvector.

Q.5 Repeat the above experiment with $\boldsymbol{x}_0 = \boldsymbol{v}_2 + \boldsymbol{v}_3$ where $\boldsymbol{v}_2, \boldsymbol{v}_3$ are the two eigenvectors corresponding to the two smallest eigenvalues. Clearly $\alpha_1 = 0$, but we still get convergence, even though with a lot more iterations. Here, round-off error worked in our favor!

Q.6 In your experiment using $\boldsymbol{x}_0 = [1, 1, 1]^\intercal$, plot the first component of $\boldsymbol{x}_k$. Does it converge to a value? Now using this $\boldsymbol{x}_0$ apply the power method to $-\boldsymbol{A}$ (note that now the largest eigenvalue is negative), and plot the first component of $\boldsymbol{x}_k$. What do you observe and why? How about the eigenvalue estimate: does it converge? How about the error $\mathrm{Err}(\boldsymbol{x}_k, \boldsymbol{v}_1)$?

Now consider the symmetric matrix

$$\boldsymbol{A} = \begin{bmatrix} 1 & 3 & 4 \\ 3 & 1 & 2 \\ 4 & 2 & 1 \end{bmatrix}.$$

Q.7 Find its eigenvalues and eigenvectors using using `eig` or `np.linalg.eig` in `Matlab` and `Python`, respectively.

For

$$\boldsymbol{S} = \begin{bmatrix} 8 & 1 & 6 \\ 3 & 5 & 7 \\ 4 & 9 & 2 \end{bmatrix}.$$

let $\boldsymbol{B} = \boldsymbol{S}^{-1}\boldsymbol{A}\boldsymbol{S}$. Clearly $\boldsymbol{A}$ and $\boldsymbol{B}$ are similar, although $\boldsymbol{B}$ is no longer symmetric.
Q.8 Run the power iteration on $\boldsymbol{A}$ and $\boldsymbol{B}$ with $\boldsymbol{x}_0 = [1, 1, 1]^\intercal$ until $|\nu_k - \lambda_1(\boldsymbol{A})| \leq 10^{-10}$ and $|\nu_k - \lambda_1(\boldsymbol{B})| \leq 10^{-10}$, and plot the errors across iterations for both $\boldsymbol{A}$ and $\boldsymbol{B}$. What do you observe and why?

A.2 Inverse Iteration

Power iteration as depicted in Algorithm 1 gives an estimate on the largest eigenvalue, however at a rate that is determined by $|\lambda_2/\lambda_1|$. If this ratio is close to 1, then power method can converge very slowly. The *inverse iteration* addresses this shortcoming by what is known as a *shift and invert technique*. This gives rise to a much faster convergence, however it comes at the considerable cost of having to solve a linear system in each iteration.
For a given shift parameter γ, the eigenvalues of $\boldsymbol{B} = (\boldsymbol{A} - \gamma\boldsymbol{I})^{-1}$ (assume it is invertible) are $\mu_i = 1/(\lambda_i - \gamma)$. Now the closer γ is to λ_1, the more dominant the largest eigenvalue of $\boldsymbol{B}$ is. The iterations of power method applied to $\boldsymbol{B}$ converge at a rate determined by $|(\lambda_1 - \gamma)/(\lambda_2 - \gamma)|$, which can be significantly smaller than $|\lambda_2/\lambda_1|$.
In fact, the same technique is possible if γ is picked to be close to any eigenvalue λ_i, and not necessarily the dominant one. So inverse iteration works for estimating any eigenvalue as long as we know roughly what its value is.

Remark 6. Beside the issue of "how to choose γ", the major computational cost here is that the inverse iteration requires solving a linear system at every iteration, i.e., $\boldsymbol{B}\boldsymbol{x}_k$ amounts to $(\boldsymbol{A}-\gamma\boldsymbol{I})^{-1}\boldsymbol{x}_k$, which can be obtained using linear system solvers (remember, we never form the inverse explicitly). This is in sharp contrast to the power method in which each iteration only involves a matrix-vector product. So one might argue that the convergence of the inverse iteration better be very fast for it to be worth it. In fact, for very large-scale problems such as those arising in Internet searching, using this method

may be too expensive and out of the question.