Finding Few Largest Eigenvalues

Shiyi Chen
Tutor: Leonardo T. Rolla

January 21, 2021

Abstract

Power Method (PM) allows us to find the largest eigenvalue of real square matrices of real square matrices. By changing the algorithm of PM, the resulting algorithm - Inverse Power Method (IPM) enables us to find the eigenvalues other than the largest one. In this report, we firstly describe the IPM and code it in Matlab/Octave. To test the new function of IPM, we elaborate on how IPM allows us to find eigenvalues closest to a point for symmetric matrices. We then move on to elaborate on how we find the few largest eigenvalues for symmetric matrices using the IPM and the searching scheme we devised. The method succeeded in finding all the 10 largest eigenvalues of 30×30 random symmetric positive matrices in 98.4% sampled randomly.

Contents

1 Inverse Power Method

IPM is a variation of the Power Method which can be used to calculate the eigenvalues not limited to the one with the largest magnitude but all eigenvalues for the symmetric floating-point random matrices. PM starts by picking a vector, call it v_0, then apply the linear operator A to $v_k (k \geq 0)$ ($v'_{k+1} = Av_k$) and normalize ($v_{k+1} = v'_k / ||v'_k||$) iteratively. We keep on doing those two steps until v_k stays in the same line before and after applying A. PM finds the largest eigenvalue for any real square matrices. In the rest of the report, we mean small or large of the eigenvalues by referring to their absolute values.

By adapting PM, we can find all eigenvalues in the spectrum of the symmetric floating-point random matrices. Suppose λ is an eigenvalue of non-singular operator A corresponding to eigenvector v, in other words,

$$Av = \lambda v. \tag{1}$$

Suppose μ is a real number satisfying $\mu \in [\lambda_{min}, \lambda_{max}]$. μ is not equal to any of the eigenvalues of A. Thus, Equation (1) can be written as

$$\begin{aligned} (A - \mu I)v &= (\lambda - \mu)v \\ (A - \mu I)^{-1}v &= \frac{1}{\lambda - \mu}v. \end{aligned} \tag{2}$$

Compared to PM, IPM multiplies $(A - \mu I)^{-1}$ instead of A in each iteration. IPM finds the largest eigenvalue of $(A - \mu I)^{-1}$, this corresponds to the smallest $\lambda_i - \mu$ (λ_i belonging to the spectrum of A). As a result, IPM finds the eigenvalue λ_i closest to μ.

Let $\lambda_1, \lambda_2, \lambda_3, \cdots, \lambda_n$ be the eigenvalues listed from the largest to the smallest. How fast PM reach the eigenvalue and eigenvector depends on the ratio of the largest eigenvalue and the second largest eigenvalue ($\frac{|\lambda_1|}{|\lambda_2|}$) [2].

For IPM, the rate of convergence depends on

$$\frac{|\mu - \lambda'_2|}{|\mu - \lambda'_1|}, \tag{3}$$

while λ_1', λ_2' are two eigenvalues of A which are closest to μ in magnitude. By Equation (2), in each iteration, we have

$$\begin{aligned} x_{k+1} &= (A - \mu I)^{-1} x_k \\ (A - \mu I) x_{k+1} &= x_k. \end{aligned} \tag{4}$$

Equation (4) requires us to apply a linear solver in each iteration. We use the Matlab/Octave inbuilt linear solver to solve the linear equations in each step for optimal efficiency.

In each iteration, we measure the error by

$$error = ||x_{k+1} - x_k||_\infty, \tag{5}$$

while x_{k+1} and x_k are two normalized vectors in adjacent iterations. This measurement is equivalent to find the entry with largest magnitude of the difference vector $(x_{k+1} - x_k)$. We measure error in this way because it gives us the difference of the vectors in two adjacent iterations. This difference will be zero if both of them are eigenvectors.

The code for IPM is listed in Script 1. This Matlab/Octave function takes the square matrix `A`, starting vector `x0`, initial guess of the eigenvalue `mu`, allowed maximum iterations for IPM `m`, and the allowed error `e` as input. The result vector `x` and result value `eigen` as output.

2 Tests on IPM

We made two tests for the IPM function, the first one tests whether IPM can find the nearest eigenvalue around μ. The second one finds the few largest eigenvalues for A.

2.1 Random number generators

The two tests generate the random matrices in the same way. Before digging into the details of two tests, let first look at how we generate the matrices.

We use `rand` function to generate the random matrices for compatibility in Octave and Matlab. We set the key value of the generator to be `'twister'` in the `rand` that corresponds to the random number generator

Mersenne Twister[1]. Mersenne Twister is the updated random number generator implemented in both Octave and Matlab. Although we can set the random number generator in Matlab and Octave both to be Mersenne Twister, also set the value of `seed` to be the same in Matlab and Octave, the Octave random matrix sequences differ from the Matlab one. Since the range of each entry is from 0 to 1, the scales of eigenvalues are the same for Matlab and Octave.

As discussed above, we generate the random matrices use the following code:

```
rand('twister',seed);
a = rand(n,n);
A = a'*a;
```

In the first line, we set the generator to be `'twister'` and set the value of the seed to be `seed`. In the second line, we generator the n by n random matrix a with each entry range from 0 to 1. Finally, we let $A = a^T a$.

2.2 The first runtest

We name the first test `easy_runtest.m`. This test finds the nearest eigenvalue around μ. To achieve that, the first test does the following:

- If there are no variables with the same names in Table 1, create such variables in the workspace with default values as shown in that table.
- Generate random matrices as discussed in Section 2.1.
- Use Matlab/Octave function `eig` to find all eigenvalues for A.
- Use the IPM to find the nearest eigenvalue of μ. If the program fails to converge, i.e., `ERR>e` when the maximum iterations `m` reached, the prompt "`Maximum iteration reached, exiting the program.`" will appear in `Report.txt`.
- Write all the data into a file called `Report.txt` with detailed explanations, including eigenvalues found by IPM, eigenvalues found by `eig` function, and the iterations IPM before the program terminates in each trial of IPM.

Variable name	Description	Default values
`n`	The size of the square matrix.	100
`m`	Maximum iteration for IPM allowed.	500
`mu`	$x_{k+1} = (A-\mu I)^{-1}x_k$in each iteration of IPM.	n/2
`e`	If $\|\|x_{k+1} - x_k\|\|_\infty \leq e$ return x_{k+1}.	0.01
`display`	The number of eigenvalues of A found by `eig` function displayed.	3
`seed`	The seed in the random number generator	3

Table 1: The default values for `easy_runtest.m`

In order to run IPM on non-symmetric matrices, we let `A = rand(n,n)` directly in the `easy_runtest.m`. There may exist some pairs of non-real eigenvalues for non-symmetric matrices. We find that when the closest eigenvalues are two conjugate complex eigenvalues (we use `norm` function to calculate the norm of the complex eigenvalues, if they are conjugate, they have the same norm), the program fails to converge. The reason is: when a pair of complex eigenvalues is closest to μ than any other eigenvalues, $|\mu - \lambda_1'| = |\mu - \lambda_2'|$, since λ_1' and λ_2' are conjugate ($\lambda_1' = \bar{\lambda}_2'$). This implies $\dfrac{|\mu - \lambda_2'|}{|\mu - \lambda_1'|} = 1$.

The `easy_runtest.m` sometimes fails to converge. Using the default values as shown in Table 1(size of matrix is 100 by 100,) we find that when the value calculated by Equation (3) is close to 1 (for example 0.98), the method is likely to fail.

2.3 The second runtest

`PM.m` is the Matlab/Octave code of Power Method, we will use it in the process of finding the largest eigenvalue. In the second test `runtest.m`, we use IPM to find the few largest eigenvalues of given real symmetric square matrices A. Same as the first test described in the last section, `runtest.m` also has the default variables as listed in Table 2.

Real symmetric matrices have exactly n eigenvalues. Since we are using the floating-point random function to generate the matrices, the

probability to generate the eigenvalue with multiplicity more than one equals zero. Thus, all our choices should converge to exactly n points. In practice, we find those points IPM converged to are clustered around the accurate eigenvalues found by `eig` function. Due to this fact, we can start by finding the largest eigenvalue by calling PM function, also finding the smallest eigenvalue by using IPM with $\mu = 0$. There are two rounds for the `runtest.m`. In the first round, we start choosing our μ from the largest eigenvalue, then use IPM to search for other eigenvalues other than the largest one. This process terminates either μ reaches the smallest eigenvalue or we have found `findn` points where IPM converge to. Our choice of μ is equally distributed on a linear scale. We initially choose μ in log-linear scale, for example $\mu = 1.01, 1.01^2, 1.01^3 \cdots$. We find the log-linear scale does not correspond to the real distribution of the eigenvalues, it's too dense while μ is close to 0 and too sparse for μ close to largest eigenvalue, this scale may cause the repeating eigenvalues (two slightly different values which correspond to the same eigenvalue appear in the eigenvalue list) for the small eigenvalues and missing eigenvalues for the large eigenvalues.

We will consider two points where IPM converges corresponding to two different eigenvalues if the distance between them is larger than `e1` or `e2` depending on the position of the interval. For the measurement of distance, we take a relative measure: suppose we already find λ_1, λ_2 as the two largest eigenvalues. If p is the point where IPM converge to, then the relative distance between p and λ_2 can be calculated as:

$$|\lambda_2 - p|/|p|$$

The problem for the first round of searching is that some eigenvalues are missed in the eigenvalue list, for example, the program omits the second largest eigenvalue and considers the third-largest eigenvalue as the second largest eigenvalue. To solve this problem, we devise the second round of searching. Since we have `findn` points in the eigenvalue list, there are `(findn - 1)` intervals between `findn` eigenvalues. In each of those intervals, we put μ at positions 0.4 and 0.6 times the length of that interval. If IPM converges to different points other than the two ends of the interval, we will append those points to the end of the eigenvalue list. Finally, we will get a new eigenvalue list. We do above process until the new eigenvalue is no longer found in the new iteration.

Pseudo-code 1 includes more programming details for the process discussed above.

To get more information on which choices of μ fail to converge, we printed that μ on the terminal as long as we have the iterations for those numbers exceeds m.

3 Improved searching scheme for the second round for the second test

3.1 Measure the Cosine Values between the vectors

We mentioned that using the current searching scheme, we may encounter the situation when the tolerance is too small which causes repeated eigenvalues in the eigenvalue list. We try to solve this problem by changing the way of measuring the difference between the two eigenvalues, but we did not get the desired result.

Originally, we measure the difference directly through the difference between the two values. Later, we realize that the vectors IPM function returned could also be used as the way of measurement.

We begin by calculating the cosine values of angles between the vectors in eigenvalue list created in the first round of `runtest.m`. Here is the formula we used:

$$\cos\alpha = \frac{v_1 \cdot v_2}{||v_1|| \cdot ||v_2||}$$

From the testing result, the cosine value of adjacent vectors in the first eigenlist are close to 0, ranging from 10^{-6} to 10^{-11}. This can be explained by the real spectrum theorem, we generate the matrices by letting $A = a^T a$ which implies that A is symmetric and A is orthogonally diagonalizable. Hence, all eigenvectors of A are orthogonal to each other.

We do the trick in the second round of searching by adding this criterion in distinguishing the eigenvalues. In theory, there are two cases for the cosine value of the angles with two points on the end of the interval:
1) Both of the cosine values are close to 0, which corresponds to the situation when the new point IPM found is different from previous points in eigenlist.
2) One of the cosine value is close to 0 while the other one is close to 1. In this case, IPM finds the repeated eigenvalue either corresponds to the

Variable name	Description	Default value
`n`	Size of the square matrix.	100
`m`	Maximum iterations allowed for IPM.	500
`e`	If $\|\|x_{k+1} - x_k\|\|_\infty \leq e$ return x_{k+1}.	0.005
`e1`	The shortest relative distance where the program will consider two points as distinct eigenvalues in the first round.	0.1
`e2`	The shortest relative distance where the program will consider two points as distinct eigenvalues in the second round in the first 5 intervals.	0.07
`e3`	Same as e2 but for the intervals other than the first 5.	0.03
`findn`	The number of eigenvalues the program will find.	3
`e4`	If both cosine value larger than this, consider as a distinct pair of eigenvalue and eigenvector	0.01
`scalers`	The relative distance discussed in Section 3.1	[0.3,0.06]
`repeat`	The number of searching process in the second round of searching.	3
`seed`	The seed in the random number generator.	3

Table 2: The default values for `runtest.m`

left end or the right end.

In practice, there is another case for the missing eigenvalues. Suppose there is an eigenvalue list returned by the first iteration which has exactly one eigenvalue missing in some interval. Our intention for the second round is to find the missing eigenvalue in that interval. When our searching process arrives the interval corresponding to that eigenvalue, we try to pick up two points in that interval and perform IPM to the two points with hope that they can converge to one point different from those on the end of the interval. It can converge to one point different from the ends if we are lucky enough. However, for other cases, after applying IPM, the points we choose converge to the end points on the interval again.

One way to solve this problem is to add more points in the intervals in the second searching process. In order to achieve this, we add the variable `scalers` which enables tester deciding the distance those point away from the end of the interval. As usual, we take the relative measure for the variable `scalers`. The relative distance of p to the two ends λ_i, λ_j can be calculated as

$$d = \min\left(\frac{\lambda_i - p}{\lambda_i - \lambda_j}, \frac{p - \lambda_j}{\lambda_i - \lambda_j}\right)$$

while $\lambda_i > \lambda_j$, and $p \in (\lambda_i, \lambda_j)$. All the possible values of d is stored in the variable `scalers` and it can be specified by the tester. By default the value `scalers` is taken as `[0.3,0.01]`.

Another problem arises for the angle test is that if IPM discover the new eigenvalue corresponding to the new eigenvectors are not totally orthogonal. We encounter the cosine value between one of the end vector as 0.3 sometimes. We can solve this is by changing the tolerance for angle test $e4$ larger. We can enlarge the $e4$ to some value larger than 0.3. However, we choose to alter the eigenvalue list that the first searching round returned. Suppose we have ten values returned by the first round, we apply IPM again to the values in the list and we obtain a value list along with a vector list which has exactly same number of elements. We update the eigenvalue list by the new values IPM returned and keep the vector list with the same order with the eigenvalue list. These two lists are passed to the second round of searching.

3.2 Test for `runtest.m`

We test `runtest.m` by setting seeds with different values and test whether `runtest.m` finds `findn` largest eigenvalues or not. We do this process in MatLab for the optimal speed of execution. We have tested the seeds from 0 to 4,000 in MATLAB and there are 64 seeds (1.6%) which causes the program failing to find the eigenvalues. There are two types of situations for the method to fail - either repeated eigenvalues or missing eigenvalues. The repeated eigenvalues are caused by the tolerance (threshold value which we thought two values IPM found are different, either `e1`, `e2` or `e3`) is smaller than the actual distance between two adjacent eigenvalues. If the second round fails-all points we plugged in fail to converge to the eigenvalues, the program will miss the eigenvalue. If the parameter `repeat` is too small (for example 1), the second round of searching will not be done sufficiently, this will increase the chance for missing eigenvalues. However, if we do the second round of searching too many times, it will cause a longer elapse time. So the balance is needed for the best performance. Table 2 shows the well-adjusted parameters for the program.

3.3 Drawback

It is better to eliminate the variable `repeat` by implementing a mechanism such that we stop doing it if the new eigenvalue is not found. Thus, we do not need to manually adjust the variable `repeat`.

4 Conclusion

We start by describing IPM and implementing it in computer program. Then, we move on to introduce the tests we have run on the Inverse Power Method. The first test proves that the method can find the closest eigenvalue to a given point but the method fails when $\frac{|\mu - \lambda_2'|}{|\mu - \lambda_1'|}$ is close to 1. The second test proves that IPM can find the few largest eigenvalues for real matrices generated by $A = a' * a$ but fails for some cases.

There are some unsettled problems we encountered during this research. For example, in the first test, is there any better solution for the situation when $\frac{|\mu - \lambda_2'|}{|\mu - \lambda_1'|}$ is close to 1 which usually arises for non-symmetric matrices? In the second round of searching for the second

test, how can we ensure that the points we choose in the interval are different from the eigenvalues on two ends of the interval? Is there any better searching scheme which can avoid both the repeated eigenvalues and the missing eigenvalues in the final eigenlist? We hope future research could explore these questions. All code for this project can be found at https://github.com/Eleven7825/IPM.

References

[1] Mersenne twister. http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html. Accessed: 2020-10-26.

[2] Richard L Burden, J Douglas Faires, and Albert C Reynolds. Numerical analysis, 2001.

Algorithm 1 `Runtest.m`

procedure SETUP(n, m, e, $e1$,$e2$,$e3$,`seed`,`findn`,`repeat`)
    Generate random matrix A
    λ_{min} = IPM(A,0,x0,m,e)
    λ_{max} = PM(A,x0,m,e)
    Create empty lists `PList` and `eignList`
    Setting $i_{max} = \log \lambda_{max}$, $i_{min} = \log \lambda_{min}$, `Stepsize` = $(i_{min} - i_{max})/100$
end procedure

procedure THEFISTROUND
    while length(`eigenlist`) $<$ `findn` AND `eigen` $> \lambda_{min}$ **do**
        $\mu = 10^{\log(\mu) - \texttt{StepSize}}$
        p = IPM(A, μ,x0,m,e)
        Add p to `pList`
        `distaceList` = $|(\texttt{distanceList} - p)/p|$
        if any elements of `distanceList` $>$ e1 **then**
            Add to `eigenList`
        end if
    end while
    `eigenlist` = IPM(`eigenlist`)
end procedure

procedure THESECONDROUND
    while The new eigenvalue is found in the iteration **do**
        for `interval=(left:right)` between `eigenList` **do**
            step = length of the `interval`
            $p_1, \cdots, p_n$ = IPM(`left- scalers`*step,`right+scalers`*step)
            if the interval corresponds to the first 5 eigenvalues **then**
                TOL = e2*step
            else
                TOL = e3*step
            end if
            if any(p_i-`left`)>TOL or Cosine value $<$ e4 **then**
                Append p_i to `eigenList`
            else if any(`right`-p_i)>TOL or Cosine value $<$ e4 **then**
                Append p_i to `eignList`
            end if
            Reorder the `eigenList` from the largest to smallest
            Keep first `findn` elements of `eigenlist` while deletes others
        end for
    end while
end procedure

Script 1 IPM function in Octave/Matlab

```
function [eig_value,eig_vector,i] = IPM(A,mu,x0,m,e)
% initialize the method
x = x0;
B = A - mu*eye(size(A));
i = 0;
ERR = 1+e;
Xp = 0;

for j = 1:length(x)
    if abs(x(j)) > Xp
        Xp = x(j);
    end
end
x = x/Xp;

while i < m && ERR > e
    y = x/B;
    for j = 1 : length(y)
        Yp = 0;
        if abs(y(j)) > Yp
            Yp = y(j);
        end
    end

    ERR = 0;
    errvec = x - (y/Yp);
    for j = 1:length(errvec)
        if abs(errvec(j)) > ERR
            ERR = abs(errvec(j));
        end
    end

    x = y/Yp;
    i = i+1;
end

if i < m
    eig_value = 1/Yp+mu;
    eig_vector = x;
    return
else
    disp('max m exceeded!')
    return
end
end
```