Computational Mathematics I - Project 2

Foivos Lambadaridis Kokkinakis

December 2023

Contents

Abstract/Objective

Numerical methods aren't only used in solving simple equations. We often have to tackle more difficult problems that demand solving a big number of equations at once. When those equations are linear, we often use a mathematical construct called a matrix that allows us to write and perform calculations more compactly. Matrices follow their own mathematical rules and have their own processes and computable quantities that give us crucial information about themselves. In this paper, we will focus on the numerical inversion of a matrix A using the L-U decomposition method and the calculation of its largest eigenvalue and the corresponding eigenvector using the power method.

The matrix we will work with is:

$$A = \begin{bmatrix} 4 & 1 & 2 & 3 & 5 \\ 1 & 3 & 1 & 4 & 2 \\ 2 & 1 & 5 & 2 & 3 \\ 3 & 4 & 2 & 4 & 1 \\ 5 & 2 & 3 & 1 & 5 \end{bmatrix}$$

We will present the code we wrote and some extra calculations we did in two appendixes. The first one will be about the inversion of A with LU decomposition, while the second will be about the power method and the calculation of A's largest eigenvalue and its corresponding eigenvector.

Chapter 1

LU decomposition and inversion of a matrix

1.1 Theory

Matrices have many applications in mathematics and physics alike. Let's assume we have the $(n \times n)$ matrix A. One of the most commonly done calculations concerning a matrix A, is finding its inverse A^{-1}. We say that A^{-1} is the inverse of A only if [1]:

$$\boxed{AA^{-1} = A^{-1}A = I_{nxn}} \tag{1.1}$$

where I_{nxn} is the $(n \times n)$ identity matrix meaning it has its diagonal elements equal to one, while all its other elements equal to zero.

Unlike scalars (apart from zero) matrices aren't always invertible so we have to be careful. Before attempting to calculate A^{-1} we first must make sure that it exists, which means that for A, it must be true that [1]:

$$\boxed{det(A) \neq 0} \tag{1.2}$$

After calculating $det(A)$ the most common way to compute A^{-1} is by applying the formula: $A^{-1} = \frac{1}{det(A)} Adj(A)$, where $Adj(A)$ is a matrix made of n determinants of $(n-1) \times (n-1)$ dimensions. [1]. From the definition of the determinant, it's clear that one would have to make a very large number of computations to calculate one determinant, especially if n is large, since a $(n \times n)$ determinant is made from n,$(n-1) \times (n-1)$ determinants [1]. To be more specific, the order of computations one has to make to compute a $(n \times n)$ determinant is $O(n!)$! [2]. This makes the application of the formula shown above, very time-consuming and computationally costly, which in return makes it clear that we must approach the calculation of A^{-1} differently.

From matrix multiplication, we know that when two matrices A and B are multiplied and the result is a matrix C, the ith line of the A matrix and the jth column of the B matrix are going to be used to calculate the $c_{i,j}$ element of the C matrix [1]. Let's assume we are trying to solve a system of n linear equations $A \times x = b$ where A is the coefficient matrix with dimensions $(n \times n)$, x is the column matrix of the variables with dimensions $(n \times 1)$ and b is the column matrix of the constants with dimensions $(n \times 1)$. Now let's assume that the mth element of b, b_m is its only non-zero element and it's equal to one.

If we take into account eq 1.1 and that matrix multiplication works as explained earlier, we can see that if the only non-zero element of b is b_m, the solution of our system must be such that the matrix x contains the elements of the mth column of A^{-1}. [3]. This means that the calculation of the inverse matrix of a $(n \times n)$ matrix A, is reduced to the calculation of the solution of n number of linear systems, each containing n linear equations. By solving each one of these n linear systems we will calculate one column of A^{-1}. The only difference between these n linear systems of equations is going to be matrix b, since the position of b_m will determine the column of A^{-1} that will be calculated.

Let's assume that we're trying to solve the linear system:

$$A \times x = b \tag{1.3}$$

The most commonly used method of solving several systems where their only difference is the matrix of constants b, is L-U decomposition. L-U decomposition refers to the process of solving a linear system of equations by firstly calculating the elements of a lower diagonal matrix L and an upper diagonal matrix U [3], such as:

$$\boxed{L \times U = A}$$

After that, matrices L and U are used to solve the system. One can assume different things about matrices L and U which will influence the formulas they'd have to use. Here we chose to work with Crout decomposition, which assumes that the diagonal elements of U are equal to one [3].

1.1.1 Step 1: Crout decomposition

To explain how we can calculate the elements of L and U, let's assume that A is a (3×3) matrix. Then: $A = L \times U \Rightarrow A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \begin{bmatrix} l_{11} & 0 & 0 \\ l_{21} & l_{22} & 0 \\ l_{31} & l_{32} & l_{33} \end{bmatrix} * \begin{bmatrix} 1 & u_{12} & u_{13} \\ 0 & 1 & u_{23} \\ 0 & 0 & 1 \end{bmatrix}$
Multiplying the elements L's rows with the first column of U we get: $l_{11} = a_{11}$ $l_{21} = a_{21}$ $l_{31} = a_{31}$. Knowing the first column of L, we multiply its elements with those of the 2nd and 3rd column of U and we get: $l_{11}u_{12} = a_{12} \Rightarrow u_{12} = \frac{a_{12}}{l_{11}}$ and $l_{11}u_{13} = a_{13} \Rightarrow u_{13} = \frac{a_{13}}{l_{11}}$. We now know the first row of U. Using each element, one would then calculate the elements of the 2nd column of L and then the elements of the 2nd row of U and so on. This process is repeated until all elements of L and U have been calculated. Luckily for us, there are general formulas for the calculation of these elements, which make the L-U decomposition process quite easy to program. If A is a $(n \times n)$ matrix the elements of L and U are [3]:

$$\boxed{l_{ij} = a_{ij} - \sum_{k=1}^{j-1} l_{ik}u_{kj} \text{ for } j \leq i \text{ and } i = 1, 2, \ldots, n} \tag{1.4}$$

$$\boxed{u_{ji} = \frac{a_{ji} - \sum_{k=1}^{j-1} l_{jk}u_{ki}}{l_{jj}} \text{ for } j \leq i \text{ and } j = 2, 3, \ldots, n} \tag{1.5}$$

where in the special case that $j = 1$, we have that $\boxed{l_{i1} = a_{i1}}$ and when $i = 1$, we have that $\boxed{l_{1j} = \frac{a_{i1}}{l_{11}}}$.[3] What's important is that if we want to find A^{-1}, we don't have to repeat this step once L and U have been calculated. This step requires only $O(n^3)$ number of computations, which is a lot smaller than $O(n!)$ [3].

1.1.2 Step 2: Solving the System

Now that we have calculated L and U, we can write our system of linear equations as:

$$\boxed{L \times U \times x = b}$$

If we name $U \times x = b'$ we have that:

$$L \times b' = b \tag{1.6}$$

which is a linear system of equations, where the matrix of coefficients is now L and the variable matrix is b'. Since L is a lower diagonal matrix this system is already solved, meaning we can directly calculate the elements of b' using the forward substitution algorithm. This algorithm is called frorward substitution since when we apply it, the first variable b'_1 is calculated first, then b_2 and so on. We can show that [3]:

$$\boxed{b'_1 = \frac{b_1}{l_{11}} \text{ and } b'_i = \frac{b_i - \sum_{k=1}^{i-1} l_{ik} b'_k}{l_{ii}}, i = 2, 3, \ldots, n} \tag{1.7}$$

Now that we know the elements of the b' matrix we can solve the system of our linear equations $U \times x = b'$. Since U is an upper diagonal matrix this system is already solved, meaning we can directly calculate the elements of x using the backward substitution algorithm. This algorithm is called backward substitution since when we apply it, the last variable x_n is calculated first, then b_{n-1} and so on. We can show that [3]:

$$\boxed{x_n = \frac{b'_n}{u_{11}} \text{ and } x_i = \frac{b'_i - \sum_{k=i+1}^{n} u_{ik} x_k}{u_{ii}}, i = n-1, n-2, \ldots, 1} \tag{1.8}$$

This step also requires only $O(n^3)$ number of computations, which is a lot smaller than $O(n!)$ [3]. In other words, the power method is a lot quicker that the analytical one.

1.2 Methodology

1.2.1 Inversibility check

In this section, we will discuss how we applied the method mentioned in the previous section to calculate A^{-1}. The code we wrote is in the 1st appendix.

Our application revolves around the (5×5) matrix A that's mentioned in the abstract section. Before even attempting to calculate the inverse matrix of A, we should first check if A is invertible. To do so we will use criterion 1.2. Using a special function, we calculate the determinant of A with the result we get from our Python program being: $det(A) = -431.999 \neq 0$. This means that A is invertable and we can attempt to calculate A^{-1}.

1.2.2 Applying Crout decomposition

Our first step when using the L-U decomposition method is, as stated above, the calculation of the lower diagonal matrix L and the upper diagonal matrix U, based on our matrix A. To do this we have created an appropriate function named LU_decomposition. This function will take a square matrix A as an input and it will return:

1. The lower diagonal matrix L using the formula 1.4 and the special case for l_{1j}.

2. The upper diagonal matrix U using the formula 1.5 and the special case for u_{i1}.

Since we have our function we will use it in our specific problem. Here the matrices we want to calculate are L and U, out of A. Using L-U (Crout) decomposition, we must save all of its outputs in the proper variables:

1. The lower diagonal matrix L is saved in the variable name L_A

2. The upper diagonal matrix U is saved in the variable name U_A.

Now that we have calculated U and L and saved them in the appropriate variables we don't have to perform this calculation for A again!

1.2.3 Inverting A

To invert A, we've created a function named LU_Inverse that takes as input the two diagonal matrices named L and U coming from the L-U decomposition of A and tries to calculate A^{-1}. Its first step is to normalise the inverse of A, as a matrix that's the same dimension as L and U filled with zeros. Then it tries to calculate A^{-1} column by column.

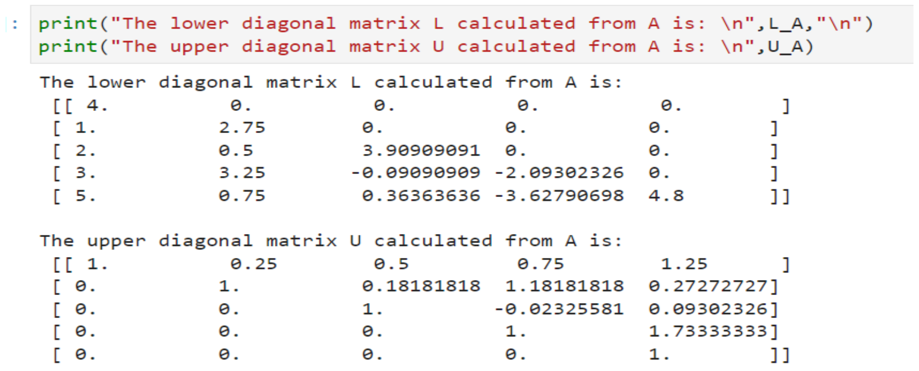
Let's say the ith column of A^{-1} is being calculated. Its' first step is to set the ith element of the matrix of constants $b_i = 1$ while it sets all its other elements $b_{\neq i} = 0$. This way, the calculation of A^{-1}'s' ith column is reduced to solving this system: $L \times U \times x = b$. Its' next step is to calculate the b' matrix using the elements of matrix L and the forward substitution formulas 1.7. Finally knowing b', using the elements of matrix U and the backward substitution formulas 1.8, it can calculate the x matrix, which is saved in the ith column of A^{-1}. This process is repeated for all columns of A^{-1} and in the end A^{-1} is returned and saved under the name A_inv.

1.3 Results/Analysis

In this section, we will present the results we got from our program and we will comment on them.

1.3.1 Crout decomposition results

Our LU_decomposition function returns us the two matrices L_A and U_A. Figure 1.1 is a picture that was taken by our program and it shows the lower and upper diagonal matrices L_A and U_A we get from our program:



```
print("The lower diagonal matrix L calculated from A is: \n",L_A,"\n")
print("The upper diagonal matrix U calculated from A is: \n",U_A)
```

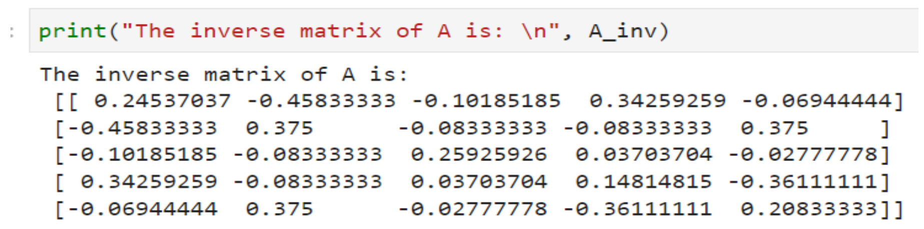
```
The lower diagonal matrix L calculated from A is:
 [[ 4.          0.          0.          0.          0.        ]
 [ 1.          2.75        0.          0.          0.        ]
 [ 2.          0.5         3.90909091  0.          0.        ]
 [ 3.          3.25       -0.09090909 -2.09302326  0.        ]
 [ 5.          0.75        0.36363636 -3.62790698  4.8       ]]

The upper diagonal matrix U calculated from A is:
 [[ 1.          0.25        0.5         0.75        1.25      ]
 [ 0.          1.          0.18181818  1.18181818  0.27272727]
 [ 0.          0.          1.         -0.02325581  0.09302326]
 [ 0.          0.          0.          1.          1.73333333]
 [ 0.          0.          0.          0.          1.        ]]
```

Figure 1.1: The upper and lower diagonal matrices that our LU_decomposition function returns if we input matrix A.

If the LU_decomposition function did its job correctly we expect that if we multiply L_A with U_A we would get back A. Indeed that result is shown in Appendix 1. This means that we have implemented the Crout decomposition step of our algorithm correctly.

1.3.2 The inverse of A

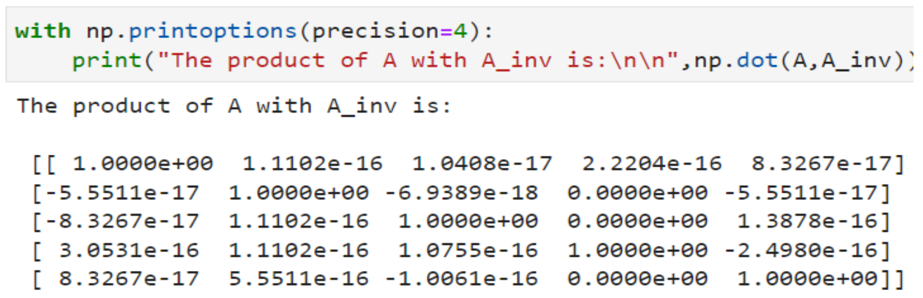
The second step is feeding L_A and U_A in our LU_Inverse function which produces A_inv. Figure 1.2 is a picture that was taken from our program and it shows the result we get for A^{-1}, A_inv.



```
print("The inverse matrix of A is: \n", A_inv)

The inverse matrix of A is:
 [[ 0.24537037 -0.45833333 -0.10185185  0.34259259 -0.06944444]
 [-0.45833333  0.375      -0.08333333 -0.08333333  0.375     ]
 [-0.10185185 -0.08333333  0.25925926  0.03703704 -0.02777778]
 [ 0.34259259 -0.08333333  0.03703704  0.14814815 -0.36111111]
 [-0.06944444  0.375      -0.02777778 -0.36111111  0.20833333]]
```

Figure 1.2: The numerical result we get for the inverse matrix of A after implementing our method.

To check if our result has any validity we will use eq 1.1 to our advantage. If A_inv is indeed A^{-1}, we expect that if we multiply A with A_inv we would get the identity matrix. The result of this multiplication is shown in figure 1.3.



```
with np.printoptions(precision=4):
    print("The product of A with A_inv is:\n\n",np.dot(A,A_inv)

The product of A with A_inv is:

 [[ 1.0000e+00  1.1102e-16  1.0408e-17  2.2204e-16  8.3267e-17]
 [-5.5511e-17  1.0000e+00 -6.9389e-18  0.0000e+00 -5.5511e-17]
 [-8.3267e-17  1.1102e-16  1.0000e+00  0.0000e+00  1.3878e-16]
 [ 3.0531e-16  1.1102e-16  1.0755e-16  1.0000e+00 -2.4980e-16]
 [ 8.3267e-17  5.5511e-16 -1.0061e-16  0.0000e+00  1.0000e+00]]
```

Figure 1.3: The result of the multiplication of A and A_inv.

The result we get is indeed not equal to the identity matrix, since some of its non-diagonal elements are close, but not equal to zero. This was expected because, as the number of calculations our computer has to perform increases, and the accuracy of our final result decreases. This is due to the round-off error and we can't do anything about it, other than finding another method of calculating A^{-1} which would require fewer calculations. This means that our algorithm worked as expected and that the L-U decomposition can indeed be used for calculating the inverse of a matrix A.

1.4 Conclusion

In conclusion, the L-U decomposition method is really important when trying to compute the inverse of a matrix A. The biggest strength of this method is that a big part of the calculation (computing L and U) has to be done one time. After that, we can use the lower and upper diagonal matrices L and U to compute each column of A expertly just by altering the constant matrix b. This makes the computation of A^{-1} a lot quicker than the analytical method.

Chapter 2

Power method and the largest eigenvalue and eigenvector

2.1 Theory

Another important calculation one may be required to do when working with matrices, is to compute its eigenvalues and the corresponding eigenvectors. In this project we will just focus on the computation of the largest eigenvalue of a given matrix, using the power method.

The definition of an eigenvector and eigenvalue is given by the following: If A is a $(n \times n)$ matrix we say that $y^{(i)}$ and λ_i is its eigenvector and eigenvalue if and only if [1]:

$$\boxed{A \times y^{(i)} = \lambda_i \times y^{(i)}} \tag{2.1}$$

Classically the computation of the eigenvalues of a $(n \times n)$ matrix A is given by its characteristical polynomial [1]:

$$det(A - \lambda \times I_{n\times n}) = 0 \tag{2.2}$$

As we can see, the above requires the computation of an $(n \times n)$ determinant which is very computationally costly since its complexity is $O(n!)$ [2]! This makes it clear the eigenvalues of a matrix should be calculated differently. Let's assume that the matrix we are working on is A and since it is an $(n \times n)$ matrix. Its characteristic polynomial could theoretically be an nth degree polynomial, meaning it could have n different eigenvalues λ_i. If we sort A's from large to smallest we get:

$$|\lambda_1| > |\lambda_2| > \ldots > |\lambda_n| \tag{2.3}$$

An important note to make here is that the eigenvectors of an $(n \times n)$ matrix A, create an $n$dimension orthocanonical system. This means that every other $n$dimension vector can be written as a linear combination of A's eigenvectors[1], [2]. In other words, we can write any vector x as:

$$x = a_1 y^{(1)} + a_2 y^{(2)} + \ldots + a_n y^{(n)} \tag{2.4}$$

where $y^{(i)}$ are A's eigenvectors that correspond to the eigenvalue λ_i. If we take the product of A with a random vector x and use eq.2.4 to represent the latter, we will have [2]:

$$x^{(1)} = A \times x = A \times \sum_{i=1}^{n} a_i y^{(i)} = \lambda_1 a_1 y^{(2)} + \lambda_2 a_2 y^{(i)} \ldots + \lambda_n a_n y^{(n)} \tag{2.5}$$

If we keep multiplying $x^{(1)}$ by A (let's say k times) we will get [2]:

$$x^{(k)} = A \times x^{(k-1)} = \lambda_1^k \left(a_1 y^{(1)} + \left(\frac{\lambda_2}{\lambda_1}\right)^k a_2 y^{(2)} + \ldots + \left(\frac{\lambda_n}{\lambda_1}\right)^k a_n y^{(n)} \right) \approx \lambda_1^k a_1 y^{(1)} \quad (2.6)$$

where the the terms with: $\frac{\lambda_i}{\lambda_1} < 1 \Rightarrow \left(\frac{\lambda_i}{\lambda_1}\right)^k \approx 1$ approach zero as k increases, since $|\lambda_1| > |\lambda_i|$ [2].

If we divide the elements of $x^{(k)}$ with the elements of $x^{(k-1)}$ assuming that $k \to \infty$ we will have:

$$u = \left(\frac{x_1^{(k)}}{x_1^{(k-1)}}, \frac{x_2^{(k)}}{x_2^{(k-1)}}, \ldots, \frac{x_n^{(k)}}{x_n^{(k-1)}} \right) \approx \left(\frac{\lambda_1^k a_1 y_1^{(1)}}{\lambda_1^{k-1} a_1 y_1^{(1)}}, \ldots, \frac{\lambda_1^k a_1 y_n^{(1)}}{\lambda_1^{k-1} a_1 y_n^{(1)}} \right) = (\lambda_1, \lambda_1, \ldots, \lambda_1) \quad (2.7)$$

In other words, the vector u will be a n long vector, where each one of its components is an approximation of A's largest eigenvalue, λ_1 [2].

The corresponding eigenvector of λ_1 will be $x^{(k)} = y^{(1)}$, since when we multiply it with A and use equations 2.7 and 2.6, we will get :$Ax^{(k)} = a_1 \lambda_1 y^{(1)} = b_a y^{(1)}$ [2].

2.2 Methodology

In this section, we will discuss how we applied the method mentioned in the previous section to calculate λ_1 and $y^{(1)}$. The code we wrote is in the 2nd appendix. To implement the power method we chose to create a function named power_method that takes as input:

1. The matrix whose eigenvector and eigenvalue we are trying to find, named A
2. How many significant figures of precision the user demands the value of the eigenvalue named significant_figures_value
3. How many significant figures of precision the user demands the value of the eigenvector named significant_figures_vector
4. The maximum amount of iteration the user is going to allow, named max_count

To begin the implementation we first have to make an initial guess x for the eigenvector of A's largest eigenvalue λ_1. We will always choose the n-long unitary vector as our initial guess. Since in our case A is a 5 by 5 matrix x, we choose $x_k = (1, 1, 1, 1, 1)$. In each iteration first, we normalize the old approximation of the eigenvector and then calculate the new one by multiplying the old one with A. Then we normalize that and save it in the proper list. After that, we divide the largest element of x_k_plus_1 with the largest element of x_k (non normalised eigenvectors) and we save that on the proper list. That value is going to be our approximation for the eigenvalue for this iteration as explained from eq. 2.7. After that, the approximation percentage error for the eigenvector and eigenvector is calculated and saved in the proper lists. The latter is calculated only from the first element from the eigenvector since we assume that every element is going to converge in a similar matter. Finally, the eigenvalue and eigenvector we got in this iteration become the old ones and the method is repeated as long as the necessary precision isn't met, and max amount of steps hasn't been reached.

In the end, we call our function. Here we want to calculate A's largest eigenvalue and the corresponding eigenvector, we ask for 10 significant figures of precision for both and we allow for 10000 iterations. Our function returns:

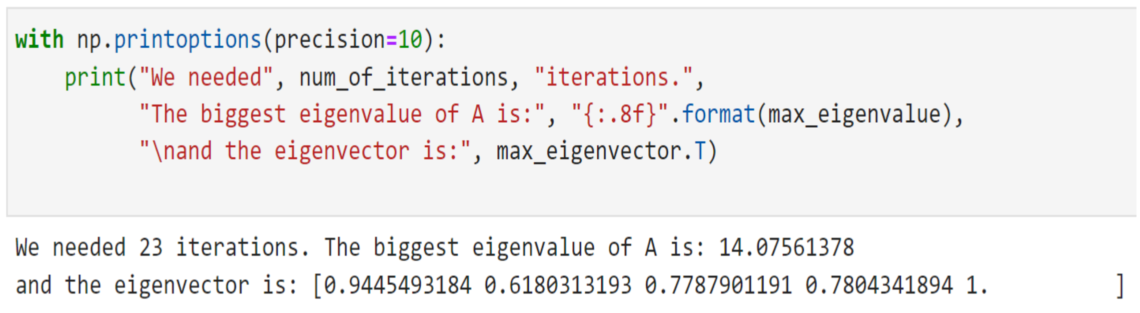
1. The biggest eigenvalue of matrix A with 10 significant figures of precision. It's saved in max_eigenvalue
2. The eigenvector that corresponds to the biggest eigenvalue. It's saved in max_eigenvector
3. The list with the approximate percentage error of the eigenvalue, calculated in each iteration. It's saved in approximate_error_value.
4. The list with the approximate percentage error of the eigenvector, calculated in each iteration. It's saved in approximate_error_vector.
5. The list with the calculated eigenvector in each iteration is saved in eigenvector_results
6. The list with the calculated eigenvalue in each iteration. It's saved in eigenvalue_results
7. The number of iterations the power method needed It's saved in num_of_iterations.

2.3 Results/Analysis

In this section, we will present our results from the power method, and its rate of convergence and we will comment on them.

2.3.1 Largest eigenvalue and eigenvector

Our power_method function returns everything that was mentioned in the previous section. Figure 2.1 is a picture that was taken from our program and it shows the results we get from the power method.



```
with np.printoptions(precision=10):
    print("We needed", num_of_iterations, "iterations.",
          "The biggest eigenvalue of A is:", "{:.8f}".format(max_eigenvalue),
          "\nand the eigenvector is:", max_eigenvector.T)
```

```
We needed 23 iterations. The biggest eigenvalue of A is: 14.07561378
and the eigenvector is: [0.9445493184 0.6180313193 0.7787901191 0.7804341894 1.          ]
```

Figure 2.1: The number of iterations the power method needed to reach the necessary precision num_of_iterations, the best approximation of largest eigenvalue max_eigenvalue and the best approximation of the corresponding eigenvector max_eigenvector. Note: Both approximations have 10 significant figures of precision so both are presented as such.

2.3.2 Power method convergence

Now we will focus on the rate of convergence of the power method. More specifically we will focus on the rate of convergence of the eigenvalue and eigenvector. To do so we have created the appropriate graph which is shown below:

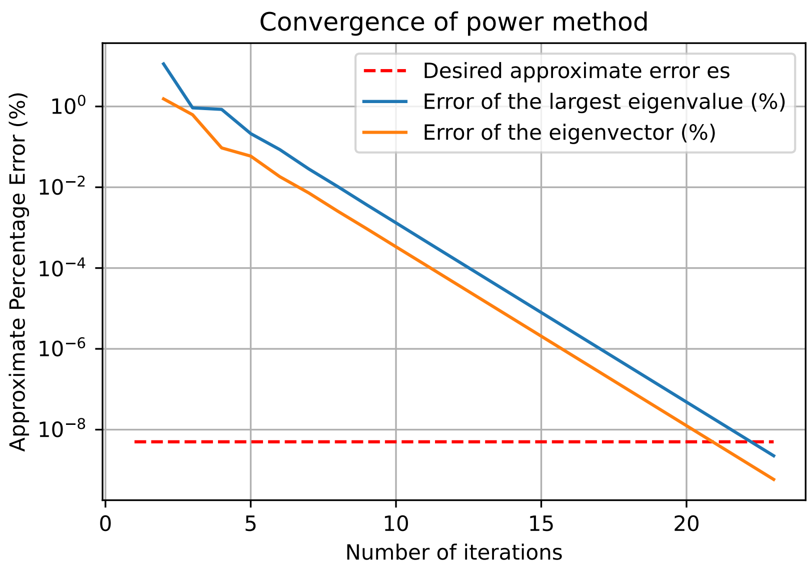


Figure 2.2: The approximate percentage error of the largest eigenvalue's approximation (blue curve), the approximate percentage error of the corresponding eigenvector's approximation (orange curve), and the Scarborough error when we ask for 10 significant figures of precision. Note: Both the eigenvalue's and eigenvector's approximate percentage errors must be lower than the Scarborough error value for us to stop implementing this method.

From figure 2.2 we can see that the eigenvector seems to need fewer iterations than the eigenvalue to converge. Another important note is that both seem to converge in a similar matter, meaning that the approximate error of both quantities falls linearly with the same slope in a semi-log graph.

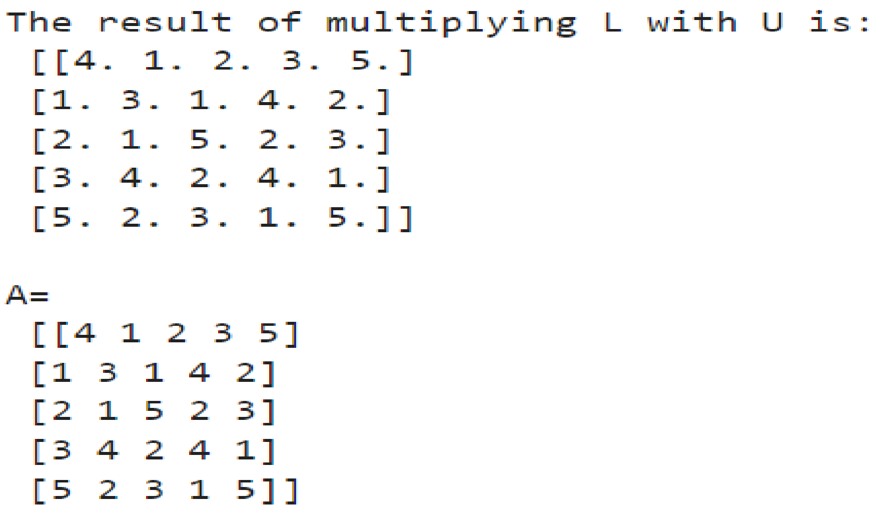
2.4 Conclusion

In conclusion, the power method can be an efficient and fast way to compute the largest eigenvalue of a matrix, something that would require a lot more calculations if we attempted the analytical way. From our study of its convergence rate, we concluded that the eigenvalue and eigenvector converge at the same rate, with the eigenvalue requiring more iteration to reach the same precision as the latter. This means that it would have been more efficient if we just calculated the approximate percentage error of the eigenvalue, since if it meets the Scarborough criterion, so would the components of the eigenvector.

Appendix I - Inversion

Verifivation of Crout decomposition

As stated in the 1st chapter, from our function LU_decomposition we got the lower and upper diagonal matrices L_A and U_A. If our function worked correctly one would expect $L\_A \times U\_A = A$. We checked if this expression holds in our program. We can see the results in figure 2.3.



```
The result of multiplying L with U is:
 [[4. 1. 2. 3. 5.]
 [1. 3. 1. 4. 2.]
 [2. 1. 5. 2. 3.]
 [3. 4. 2. 4. 1.]
 [5. 2. 3. 1. 5.]]

A=
 [[4 1 2 3 5]
 [1 3 1 4 2]
 [2 1 5 2 3]
 [3 4 2 4 1]
 [5 2 3 1 5]]
```

Figure 2.3: The result of the multiplication between the upper and lower diagonal matrices that our LU_decomposition function returns is printed. Under that matrix, A is printed for comparison.

As we can see from figure 2.3 by multiplying L_A with U_A we get matrix A. This means that our LU_decomposition performed the Crout decomposition of A perfectly.

Code of the LU_decomposition function

As we can see $A$ is invertible so we can attempt to calculate it!

Our first step when using the L-U decomposition method is calculating the lower diagonal matrix $L$ and the upper diagonal matrix $U$, based on our matrix $A$. To do so we have created an appropriate function named LU_decomposition. This function will take a square matrix $A$ as an input and it will return:

1) The lower diagonal matrix $L$ using the appropriate formulas stated in the theory section.

2) The upper diagonal matrix $U$ using the appropriate formulas stated in the theory section.

In [30]:
```
# Our function has as input a square matrix A
def LU_decomposition(A):

    n=len(A)#The dimensions of A

    L=np.zeros((n,n)) #L is normalized as the 0(nxn) matrix

    U=np.eye(n) #U is normalized as the identity matrix of n dimensions since its diagonal elements are equal to 1


    for i in range(0,n): #i from 0 to n-1 The bigger index. this means that for the L matrix, it's the row index while for U it's the column index.

        for j in range(0,i+1): # for j from 0 to i. The smaller index. this means that for the L matrix, it's the column index while for U it's the row index.

            #The sums that are used in the formula are normalized
            sumL=0
            sumU=0

            if j==0:#Special case: (The first column of L)
                L[i][0]=A[i][0] #The formula that gives us the first column of L (it's equal to the first column of A)


            if i==0: #Special case (first row of U)
                U[0][j]=A[0][j]/L[0][0] #The formula that gives us the first row of U


            #The sums used in the formulas are calculated for each value of j (sums up to j-1)
            for k in range(j):
                sumL=sumL+L[i][k]*U[k][j]
                sumU=sumU+L[j][k]*U[k][i]

            #The formulas that give us the elements of L and U when we're not in a special case
            L[i][j]=A[i][j]-sumL
            U[j][i]=(A[j][i]-sumU)/L[j][j]


    return L, U #We return
    #1) The Matrix L
    #2) The Matrix U
```

Since we have our function we will use it in our specific problem. Here the matrices we want to calculate $L$ and $U$ out of is $A$. Using L-U decomposition, we must save all its outputs in the proper variables:

1) The lower diagonal matrix $L$ is saved in the variable name L_A.

2) The upper diagonal matrix $U$ is saved in the variable name U_A.

Code of the LU_Inverse function

Now that we have calculated the matrices L_A and U_A we don't have to do that again!

We've created a function named LU_Inverse that takes as input the two diagonal matrices named L and U coming from the L-U decomposition of A and tries to calculate A^{-1}. Its first step is to normalize the inverse of A as a matrix that's the same dimension as L and U filled with zeros. Then it tries to calculate A^{-1} column by column.

Let's assume the ith column of A^{-1} is being calculated. Its first step is to set the ith element of the matrix of constants $b_i=1$ while it sets all its other elements $b_{\neq i}=0$. This way, the calculation of A^{-1}'s' ith column is reduced to solving this system: $L \times U \times x=b$.

Our next step is to calculate the b' matrix using the element of matrix L and the forward substitution formulas that are mentioned in the theory section. Finally knowing b', using the elements of matrix U and the backward substitution formulas that are mentioned in the theory section, we can calculate the x matrix, which is saved in the ith column of A^{-1}. This process is repeated for all columns of A^{-1}.

In [6]:
```
def LU_Inverse(L,U):

    n=len(L)#The dimensions of L and thus A

    A_inv=np.zeros((n,n)) #We normalize the inverse of A as the 0(nxn) matrix

    for c in range(n): #for every column of the Matrix A (or L or U)

        b_prime=np.zeros((n,1))#We normalize the b' matrix as the 0(nx1) matrix
        b=np.zeros((n,1)) #We normalize the b matrix as the 0(nx1) matrix

        x=np.zeros((n,1)) #X is normalized as 0(nx1) matrix where X[c] will give us the (c+1)th column of A inverse

        b[c]=1 #We set b[c]=1 so we can calculate the (c+1)-th columb of A inverse

        #SOLVING L*b'=b (forward substitution)
        for i in range(0,n): # for i from 0 to n-1 (i is an index so, in reality, it's from the 1st to the nth element)

            if i==0:#We check if we are working on the 1st element of b' (0 index)
                b_prime[i]=b[0]/L[0][0]# we calculate b'[i] using the appropriate forwards substitution formula

            #if we are working on the other elements of b' we calculate b'[i] using the forward substitution
            else:
                sum_b_prime=0 #We normalize the sum used in the calculation of b'[i] as 0

                #for k from 0 to i-1 (k is the limits of the sum used in the forward substitution formula)
                for k in range(i):
                    sum_b_prime=sum_b_prime+L[i][k]*b_prime[k] #We calculate the sum we need for the calculation of b'[i]

                b_prime[i]=(b[i]-sum_b_prime)/L[i][i] # we calculate b'[i] using the appropriate forwards substitution formula


        #SOLVING U*X=b' (backward substitution)
        for j in range(n-1,-1,-1): #for j from n-1 to 0 (j is an index so, in reality, it's from the nth  to the 1st element) / (using j we will calculate x[j])


            if j==n-1: #if we are working on the nth element of x ((n-1)th element of the list) we calculate it using the right formula
                x[j]=b_prime[n-1]/U[n-1][n-1]

            else:#if we are working on the other elements of x we calculate x[j] using the backward substitution formula.

                sum_x=0 #We normalize the sum used in the calculation of x[j] as 0

                for k in range(j+1,n): #for k from 0 to i-1 (k is the limits of the sum used in the formulas)
                    sum_x=sum_x+U[j][k]*x[k] #We calculate the sum we need for the calculation of x[j]


                x[j]=(b_prime[j]-sum_x)/(U[j][j]) # We apply the backwards substitution formula to calculate the element x[j]


        A_inv[:, c] = x.flatten() # We save the elements of x in the cth column of A inverse


    return A_inv #We return the inverse matrix of A
```

Since we have our function we will use it in our specific problem. Here the matrix we want to calculate A^{-1}.Using L_A and U_A that we got from L-U decomposition, we call our function. We must save all its outputs in the proper variables:

1) The inverse of matrix A is saved in a variable named A_inv.

In [7]:
```
A_inv=LU_Inverse(L_A,U_A)
```

Appendix II - Power method

Code of the power_method function

In the end our function returns:

1) The biggest eigenvalue of matrix A with significant_figures number of significant figures of precision

2) The eigenvector that correspond to the biggest eigen vector.

3) a list with the approximate percentage error of the eigenvalue, calculated in each iteration.

4) a list with the calculated eigenvector in each iteration

5) a list with the calculated eigenvalue in each iteration

6) the number of iterations it took.

The function power_method is defined below:

```
In [ ]: # Our function takes as input:
#1) A: The matrix whose bigger eigenvalue and the corresponding eigenvector the user wants to find
#2) significant_figures_value: The number of significant figures of precision the user demands the eigenvalue to have. After this precision is reached, the iteration stops.
#3) significant_figures_vector: The number of significant figures of precision the user demands the eigenvector to have. After this precision is reached, the iteration stops.
#4) max_count is the upper limit of iterations that are going to be performed. This is more of a safety measure in case the user makes a mistake, or something goes wrong.
def power_method(A,significant_figures_value,significant_figures_vector,max_count):

    ea_value=100 #The normalization of the approximate percentage error of the eigenvalue so the method can begin.
    ea_vector=100 #The normalization of the approximate percentage error of the eigenvector so the method can begin.

    #The Scarborough Formula. We use it to compute the limit of the percentage approximate error we require to reach the precision that the user asked for for the eigenvalue and vector.
    es_value=0.5*10**(2-significant_figures_value)
    es_vector=0.5*10**(2-significant_figures_vector)


    n=len(A) #The dimensions of matrix A

    x_k=np.ones((n,1)); #Our initial guess for the eigenvector corresponding to the largest eigenvalue
    eig_vec_results=[] #This is where our approximations of the eigenvector in each iteration will be saved

    eig_value_results=[] #This is where the vector u will be saved in each iteration (its elements approximate the largest eigenvalue)
    u_k=1 #The normalization of the vector u_k so the loop can be calculated the first time through (this value doesn’t matter)

    ea_results_value=[] # This is where the approximate percentage error of the largest eigenvalue will be saved in each iteration
    ea_results_vector=[] # This is where the approximate percentage error of the corresponding eigenvector will be saved in each iteration

    count=0 #The count of the iteration we're on is normalized at 0
    while abs(ea_value)>es_value or abs(ea_vector)>es_vector and count<=max_count:#While the approximate error or the eigenvalue or the approximate error or the eigenvector
        #is lower than what is given by the Scarborough Formula (the necessary precision isn't reached) and while we are under the maximum number of iterations, the method is applied.

        count=count+1 #We are in the next iteration

        eig_vec_k=(x_k/(np.max(x_k))).flatten() #The old approximation of the eigenvector is normalized and saved


        x_k_plus_1=np.dot(A,x_k) #The vector x_(k+1)=A*x_k is calculated
        eig_vec_k_plus_1=(x_k_plus_1/(np.max(x_k_plus_1))).flatten() #The eigenvector is calculated by dividing the x_k+1 vector by its larger component (normalization) and saved
        #in the appropriate list



        eig_vec_results.append(eig_vec_k_plus_1) #The vector normalized vector x_(k+1) is saved in the appropiate list

        u_k_plus_1=(np.max(x_k_plus_1)/np.max(x_k)) #We calculate the vector u, whose elements give us an estimate for the eigenvalue
        eig_value_results.append(u_k_plus_1) #The vector u is saved in the appropriate list

        ea_value=abs((u_k_plus_1-u_k)/(u_k_plus_1))*100 #We calculate the new value of the approximate percentage error of the eigenvalue by dividing the vectors
        #u_(k+1) and u_k by element
        ea_results_value.append(ea_value) #We save it in the proper list.

        ea_vector=abs((eig_vec_k_plus_1[0]-eig_vec_k[0])/eig_vec_k_plus_1[0])*100 #We calculate the new value of the approximate percentage error
        #of the eigenvector from its' first element
        ea_results_vector.append(ea_vector) #We save it in the proper list.



        u_k=u_k_plus_1 ##The best new approximation of the vector u, becomes the old one and the method is repeated
        x_k=x_k_plus_1 #The new vector x_(k+1) becomes the old one and the method is repeated

        if count==max_count:#If the max amount of iteration was reached
            print("Warning!!! the maximum number of iterations has been reached! The result isn't as precise as you've asked!") #Print a warning message to the user

    return u_k_plus_1,eig_vec_k_plus_1.flatten(), ea_results_value,ea_results_vector, eig_vec_results,eig_value_results, count #We return
    # 1) The biggest eigenvalue of matrix A with significant_figures number of significant figures of precision
    # 2) The eigenvector that corresponds to the biggest eigenvector.
    # 3) a list with the approximate percentage error of the eigenvalue, calculated in each iteration.
    # 4) a list with the approximate percentage error of the eigenvector, calculated in each iteration.
    # 5) a list with the calculated eigenvector in each iteration
    # 6) a list with the calculated eigenvalue in each iteration
    # 7) The number of iterations it took.
```

Since we have our function we will use it in our specific problem. Here we want to calculate A's largest eigenvalue and the corresponding eigenvector, we ask for 10 significant figures of precision for both, and we allow for 10000 iterations. We must save all its outputs in the proper variables:

1) The biggest eigenvalue of matrix $A$ with 10 significant figures of precision is saved in max_eigenvalue

2) The eigenvector that corresponds to the biggest eigenvalue is saved in max_eigenvector

3) The list with the approximate percentage error of the eigenvalue, calculated in each iteration is saved in approximate_error_value.

4) The list with the approximate percentage error of the eigenvector, calculated in each iteration is saved in approximate_error_vector.

5) The list with the calculated eigenvector in each iteration is saved in eigenvector_results

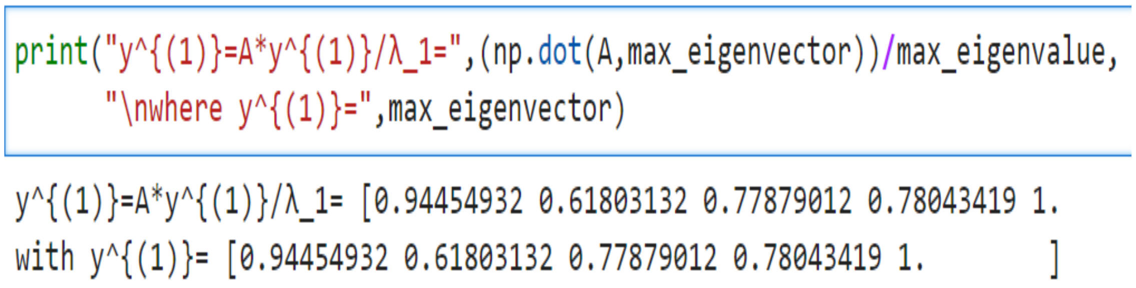
6) The list with the calculated eigenvalue in each iteration is saved in eigenvalue_results

7) The number of iterations the power method took to reach the desired precision is saved in num_of_iterations

```
In [ ]: max_eigenvalue, max_eigenvector, approximate_error_value, approximate_error_vector ,eigenvector_results, eigenvalue_results, num_of_iterations=power_method(A,10,10,10000)
```

Verification of power methods' results

In this section, we will check the validity of the results we got from our power_method function. If our results are correct, we expect our potential eigenvalue and eigenvalue to follow the relation of 2.1. In other words, if we multiply A with the potential eigenvector max_eigenvector ($y^{(1)}$) and divide the result with the potential eigenvalue max_eigenvalue (λ_1), we expect the result to be equal to the eigenvector. This calculation was done in our program and we can see the result of figure 2.4.



```
print("y^{(1)}=A*y^{(1)}/λ_1=",(np.dot(A,max_eigenvector))/max_eigenvalue,
      "\nwhere y^{(1)}=",max_eigenvector)
```

```
y^{(1)}=A*y^{(1)}/λ_1= [0.94454932 0.61803132 0.77879012 0.78043419 1.
with y^{(1)}= [0.94454932 0.61803132 0.77879012 0.78043419 1.        ]
```

Figure 2.4: $\dfrac{A \times eigen\_value}{eigen\_vector}$ and eigen_vector are compared. If they are equal then eigen_value and eigen_vector were calculated correctly.

From figure 2.4 we can see that the two vectors are equal, which means that the potential eigenvalue and eigenvector we calculated are indeed a real eigenvalue and eigenvector pair of matrix A.

Bibliography

[1]T. G. Ch. Moustakidis Efth. Meletlidou, *An introduction to linear algebra and analytical geometry* (Sofia, 2018).

[2]R. L. Burden, *Numerical analysis* (Brooks/Cole Cengage Learning, 2011).

[3]D. V. Griffiths and I. M. Smith, *Numerical methods for engineers* (CRC press, 2006).