Caleb Walter Bowden

U2166829

CHM2130 (Computational Mathematics 2)

University Of Huddersfield

Introduction

This report serves two primary objectives. Initially, it provides documentation of the testing strategy employed for the assigned tasks and the fully annotated MATLAB code PM1, PM2 and eigensolve. Subsequently, the focus extends to an exploration of Cramer's rule. This includes an overview of the rule, an example of its application, and a discussion of the algorithm's limitations and practicality in real-world contexts.

MatLab tasks

Initial vectors & Matrix

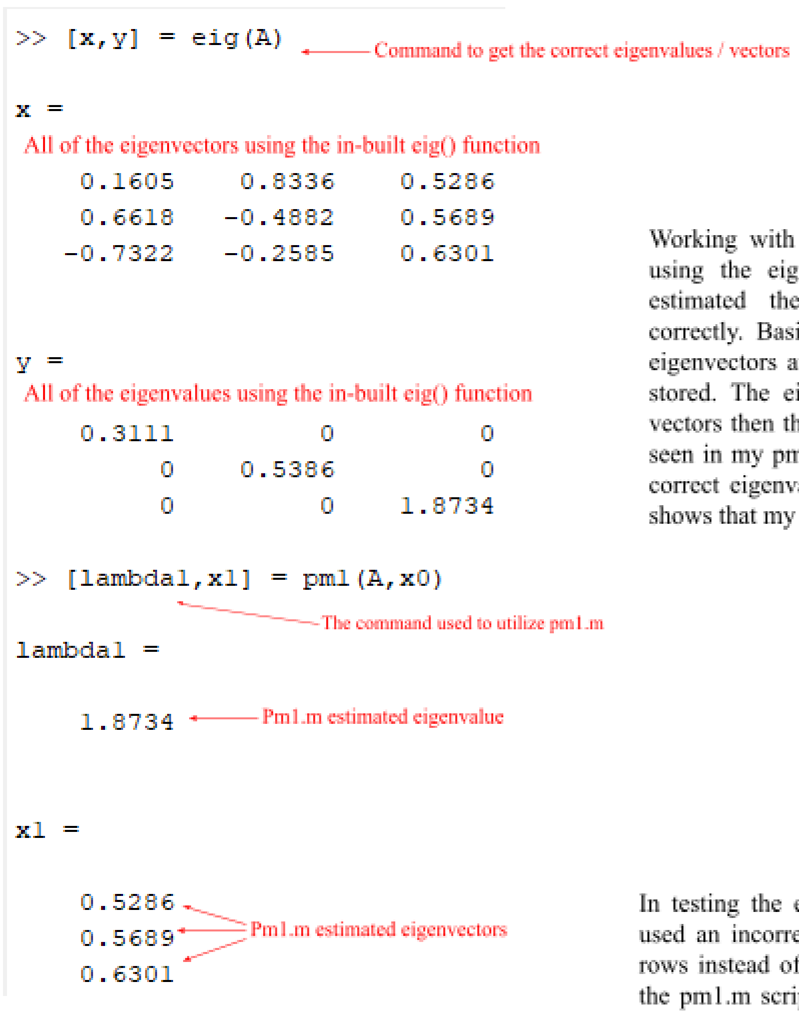
Using the rand function as follows x0 = rand(3,1) this will create a randomly generated vector. I have done this to make sure my guesses are randomly assigned. I also opted to create a random 3x3 symmetric matrix because it aids in the testing as it's a random matrix. To create this matrix I used the rand function A = rand(3x3) and to make it symmetric I used the following command A = (A + A') / 2 this command will update the matrix A and make it symmetric.

Note: During the testing phase different vectors / matrices have been used that is why eigenvalues do not match up for example pm1.m eig(A) function will be different from pm2.m eig(A) function. This is the reason.

Task 1 - pm1.m with testing

```
% This function will calculate a symmetric 3x3 matrices eigenvalue and
% eigenvector using Rayleigh quotient and the power method.
function [lambda1, x1] = pm1(A, x0) % Initial start of the PM1 function taking
in arguments A & x0 to spit out variables lambda1 (eigenvalue) and x1
(eigenvector)

% The following 4 lines below will check if the matrix is a 3x3 matrix.
% Firstly we need to get the row and column sizes of A we can use the size
% function to get the amount of rows & columns we then store them in a
% variable array. the row will store the amount of rows and the columns will
% store the amount of columns. the if statement line will then check using
% the operators '~=' (not equal to) , '||' (or) if the matrix is the
% correct size the code will carry on if not the code will throw an error to
% end the program if not the rest of the code can execute.
[row, column] = size(A); % Gets the size of the matrix
if row ~= 3 || column ~= 3 % Checks if the rows & columns equal to 3
    error('PM1: The matrix must be a 3x3 matrix.'); % Display an error message
if it's not a 3x3 matrix
end % Ends the if statement

% The following 3 lines below will check if the matrix is symmetric. The if
% statement will use the operator '~' which inverts the result of the "is
% equal" function so the statement basically says if not equal. To check if
% the matrix is symmetric we need to plug the original matrix in and then
% the transposed version using the ' operator which transposes the matrix.
% If the transpose of a matrix is not the same as the original matrix,
% then the matrix is not symmetric and we will throw an error to end the
% program. if not the rest of the code can execute.
if ~isequal(A, A') % Checks if the matrix is symmetric
    error('PM1: The matrix must be symmetric.'); % Display an error message if
it's not symmetric
end % Ends the if statement

% The following 4 lines below will check if the vector is actually a matrix
% vector. the size function will get the size of the vector and store the
% amount of rows in the rows & the amount of columns in the column section
% of the variable array. Then the if statement will check using the operators
% '~=' (not equal to) , '||' (or) if the vector is the correct size the
% code shall continue to execute if not we shall throw and error and exit
% the execution of the program.
[row, column] = size(x0); % Gets the size of the vector
if row ~= 3 || column ~= 1 % Checks if the vector is the correct size (3x1)
    error('PM1: The vector must be a 3x1') % Displays an error message if it's
not a 3x1 (not a vector)
end % Ends the if statement
```

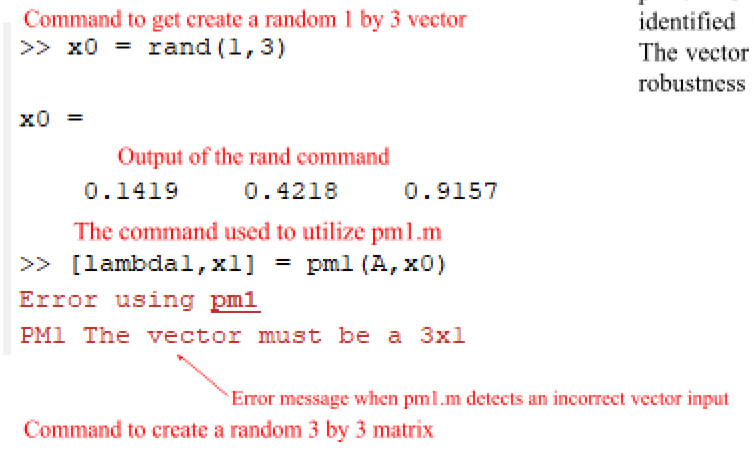
```
% Defining Variables
total_iterations = 100; % Total number of iterations the loop will perform
tolerance = 0.00001; % Tolerance variable used to set the threshold of the
acceptable error rate
iteration_counter = 0; % Counts and stores how many iterations the loop has
gone through
x1 = x0; % x1 initially stores the initial guess

% This while loop uses SMP1 (Scaled Power Method Version 1) To find the
% first eigenvalue and its corresponding eigenvector. The process begins by
% +1 the 'iteration_counter' making sure the loop will only run to the
% 'total_iterations' value. Inside the loop the calculation of the
% x is calculated by multiplying A by x1 which for the first
% iteration will be the initial guess. but will change over iterations.
% The new lambda1 (eigenvalue) is estimated using Rayleigh quotient, which
% is a method that multiplies the inverse of the current eigenvector
% estimate 'x1' by 'x' then divides by the inverse of 'x1'
% * 'x1' uses our current eigenvalue estimate. After this 'x1' is
% normalized by dividing 'x' by the normalized version of 'x' using matlabs
% in built norm() function. This makes sure the magnitude of 'x1' is constant.
% Then and if statement is created to check if the eigenvalue & vector are
close enough to the actual values within the tolerance of 0.00001.
% If so a break; is used to exit out of the while loop. if the if statement
% is false the loop will continue to iterate to either 100 or until it has
% an eigenvalue / vector estimate that fits the tolerance.
while iteration_counter < total_iterations % While loop for calculating the
first eigenvalue and eigenvector
   iteration_counter = iteration_counter + 1; % Increment the iteration counter

   x = A * x1; % Multiply matrix A by the current eigenvector estimate x1
   lambda1 = (x1' * x) / (x1' * x1); % Calculate new eigenvalue estimate using
Rayleigh quotient
   x1 = x / norm(x); % Normalize the matrix product to get the updated
eigenvector estimate x1
   if norm(A * x1 - lambda1 * x1) < tolerance % Check if the current eigenvalue
and eigenvector estimate is close enough to the tolerance also called
convergence
       break; % Exit the loop if the estimates are within the tolerance level
   end % Ends the if statement
end % Ends the while loop
end % End of of the pm1.m function
```



```
>> [x,y] = eig(A)
```
Command to get the correct eigenvalues / vectors

```
x =
```
All of the eigenvectors using the in-built eig() function
```
    0.1605    0.8336    0.5286
    0.6618   -0.4882    0.5689
   -0.7322   -0.2585    0.6301


y =
```
All of the eigenvalues using the in-built eig() function
```
    0.3111         0         0
         0    0.5386         0
         0         0    1.8734

>> [lambda1,x1] = pm1(A,x0)
```
The command used to utilize pm1.m
```
lambda1 =

    1.8734
```
Pm1.m estimated eigenvalue
```
x1 =

    0.5286
    0.5689
    0.6301
```
Pm1.m estimated eigenvectors

Working with my MATLAB script pm1.m I have been using the eig() function to check if I have correctly estimated the first eigenvalue and its eigenvectors correctly. Basically, x for the [x,y] output is where the eigenvectors are stored and y is were the eigenvalues are stored. The eig() function gets all of the eigenvalues / vectors then they are stored into x & y. From what I have seen in my pm1.m tests it looks like I have calculated the correct eigenvalues and vectors. The test is positive as it shows that my code can gather the first eigen pairs.



Command to get create a random 1 by 3 vector
```
>> x0 = rand(1,3)

x0 =
```
Output of the rand command
```
    0.1419    0.4218    0.9157
```
The command used to utilize pm1.m
```
>> [lambda1,x1] = pm1(A,x0)
Error using pm1
PM1 The vector must be a 3x1
```
Error message when pm1.m detects an incorrect vector input

Command to create a random 3 by 3 matrix

In testing the error handling of my code, I intentionally used an incorrectly formatted initial guess x0 now has 3 rows instead of 3 columns. To evaluate the robustness of the pm1.m script. This setup effectively tested the scripts ability to detect a vector input error. As expected, when pm1.m is executed with the incorrect input it correctly identified the issue and intilized and error message “PM1 The vector must be a 3x1”. This result confirms the scripts robustness and reliability to detect vector input errors.

```
>> A = rand(3,3)

A =
```

Output of the randomly generated 3x3 matrix

```
    0.7922    0.0357    0.6787
    0.9595    0.8491    0.7577
    0.6557    0.9340    0.7431
```

The command used to utilize pm1.m

```
>> [lambda1,x1] = pm1(A,x0)
Error using pm1
PM1 The matrix must be symmetric.
```

Error message when pm1.m detects an incorrect asymmetric matrix input

In this version of my error handling of my code, I needed to check if matrices that were getting inputted were in fact 3x3 symmetric matrices. To check this I had to intentionally create incorrectly formatted matrix A's. Firstly I created a 3x3 matrix that was not symmetric using the rand function. I then passed it into the function and as expected, when pm1.m is executed with the incorrect input it correctly detects the issue and initializes an error message "PM1: The matrix must be symmetric" This confirms that my script can detect asymmetric matrices.

Command to create a random 2 by 2 matrix

```
>> A = rand(2,2)

A =
```

Output of the randomly generated 2x2 matrix

```
    0.3922    0.1712
    0.6555    0.7060
```

Command to make a matrix symmetric

```
>> A = A + A'

A =
```

Output of the symmetric matrix command

```
    0.7845    0.8267
    0.8267    1.4121
```

The command used to utilize pm1.m

```
>> [lambda1,x1] = pm1(A,x0)
Error using pm1
PM1 The matrix must be a 3x3 matrix.
```

Error message when pm1.m detects a non 3x3 matrix input

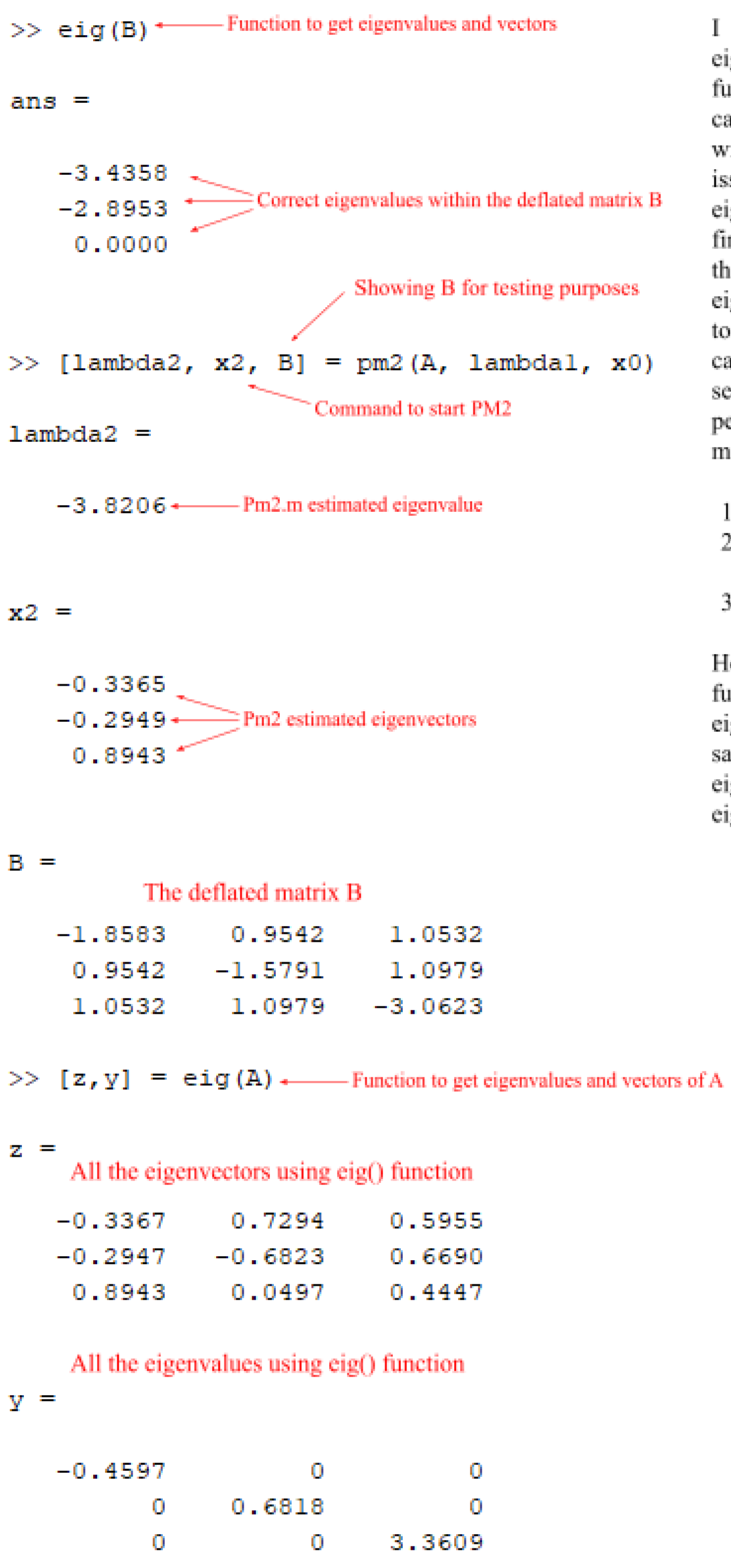
Here we are checking that the inputted matrix is infact a 3x3 matrix, As you can see we are creating a random 2 by 2 matrix that is symmetric to get a symmetric matrix we just plus matrix = A by the inverted matrix A' this should create a symmetric matrix. Once the matrix has been created it can then but pushed through into the function we call pm1.m and we how to see an error message. And once pm1.m is executed the error message appears "PM1: The matrix must be a 3x3 matrix." This shows that the function is robust can can detect symmetric non 3x3 matrixes.

Task 2 - pm2.m with testing

```
% This function will calculate a 3x3 symmetric matrixes eigenvalues and
% vectors using SPM2 method and a deflation process.
function [lambda2, x2, B] = pm2(A, lambda1, x0) % Initial start of the PM2.m
function takes in the following arguments A is the matrix, lambda1 is the first
eigenvalue, x0 is the initial guess
% The following 4 lines below will check if the matrix is a 3x3 matrix.
% Firstly we need to get the row and column sizes of A we can use the size
% function to get the amount of rows & columns we then store them in a
% variable array. the row will store the amount of rows and the columns will
% store the amount of columns. the if statement line will then check using
% the operators '~=' (not equal to) , '||' (or) if the matrix is the
% correct size the code will carry on if not the code will throw an error to
% end the program if not the rest of the code can execute.
[row, column] = size(A); % Gets the size of the matrix
```

```
if row ~= 3 || column ~= 3 % Checks if the rows & columns equal to 3
  error('PM2: The matrix must be a 3x3 matrix.'); % Display an error message if
it's not a 3x3 matrix
end % Ends the if statement
% The following 3 lines below will check if the matrix is symmetric. The if
% statement will use the operator '~' which inverts the result of the "is
% equal" function so the statement basically says if not equal. To check if
% the matrix is symmetric we need to plug the original matrix in and then
% the transposed version using the ' operator which transposes the matrix.
% If the transpose of a matrix is not the same as the original matrix,
% then the matrix is not symmetric and we will throw an error to end the
% program. if not the rest of the code can execute.
if ~isequal(A, A') % Checks if the matrix is symmetric
  error('PM2: The matrix must be symmetric.'); % Display an error message if
it's not symmetric
end % Ends the if statement
% The following 4 lines below will check if the vector is actually a matrix
% vector. the size function will get the size of the vector and store the
% amount of rows in the rows & the amount of columns in the column section
% of the variable array. Then the if statement will check using the operators
% '~=' (not equal to) , '||' (or) if the vector is the correct size the
% code shall continue to execute if not we shall throw and error and exit
% the execution of the program.
[row, column] = size(x0); % Gets the size of the vector
if row ~= 3 || column ~= 1 % Checks if the vector is the correct size (3x1)
  error('PM2: The vector must be a 3x1') % Displays an error message if it's
not a 3x1 (not a vector)
end % Ends the if statement
% The following 2 lines of code deflate matrix A into a new matrix called B
% The purpose of deflation is to remove the first eigenvalue from the
% matrix. Here we use the shifted power method to deflate the matrix. We
% start by creating a variable 'I' which stores an identity matrix the size
% of matrix A. An identity matrix is a square matrix with ones on the main
% diagonal and zeros elsewhere. To create this we use the built in eye()
% function within matlab. The next line of code starts by creating a
% variable 'B' which is the deflation matrix of A. the following code
% subtracts Lambda1 (first eigenvalue) * The identity matrix from A
% (initial matrix). This gets us our deflated matrix B
I = eye(size(A)); % Creating an Identity matrix the size of A
B = A - lambda1 * I; % Deflation process and storing result in variable B
% Defining Variables
x2 = x0; % setting x2 as the initial guess
tolerance = 0.00001; % settings the tolerance value
total_iterations = 100; % Sets the max iterations that can be used
lambda2 = 0; % Settings lambda2 as 0
iteration_counter = 0; % Stores the interactions of the while loop
% The following while loop uses SPM2 (Scaled Power Method 2) method to
calculate the second eigenvalue and
% vectors of a 3x3 symmetric matrix. The process begins by adding 1 to the
% iteration counter to make sure our loop only runs to the total iteration
% count. Then the processes multiplies the deflated matrix 'B' by x2 which
% is the guess variable and this is stored as 'x'. Next, theo code
% identifies the element in 'x' with the maximum absolute value. This is
% done by taking the absolute values of all elements in 'x', finding the
% maximum among these, and then recording the index of this maximum value.
% The value at this index in 'x' is then assigned to the variable 'm'. This
% step is crucial as 'm' represents the value used for normalizing the
% current eigenvector estimate and updating the eigenvalue estimate in the
% iterative process. the '~' operator basically means ignore this so the
% 'val' is the only thing that is stored within '[]', then x2 is updated by
% multiplying 'x' by m this process is also iterative. Then we create an if
% statement that uses the abs() function within MATLAB which gets the
% absolute value of lambda2 - m then compares with the tolerance and check
% if the difference is smaller than the tolerance if true the break
% function breaks out of the while loop. The next line of code stores m as
% lambda2.
while iteration_counter < total_iterations % While loop for calculating the
second eigenvalue and eigenvector
   iteration_counter = iteration_counter + 1; % Increment the iteration counter
   x = B * x2; % Multiplying the deflated matrix B by the current eigenvector
estimate
   [~, val] = max(abs(x)); % Finding the index of the maximum absolute value of
'x', '~' is used to remove the value its self as it is not needed.
   m = x(val); % setting variable 'm' as the element in 'x' with the highest
absolute magnitude.
   x2 = x / m; % Updating the eigenvector estimate
   if abs(lambda2 - m) < tolerance % check the difference between the current
eigenvalue and the previous eigenvalue to see if its within the tolerance level
       break; % If within the tolerance limit break out of the while loop
   end % End of the if statement
   lambda2 = m; % Updating lambda2 to the current value of m
end % End of the while loop
x2 = x2 / norm(x2); % Normalizing the eigenvector to fit with eig(A)
end % End of the PM2 function
```

I will not test to see if the error checking works correctly because it is the same error checking code that is within the PM1 Function so to check the same thing will be a waste of time as it already works.



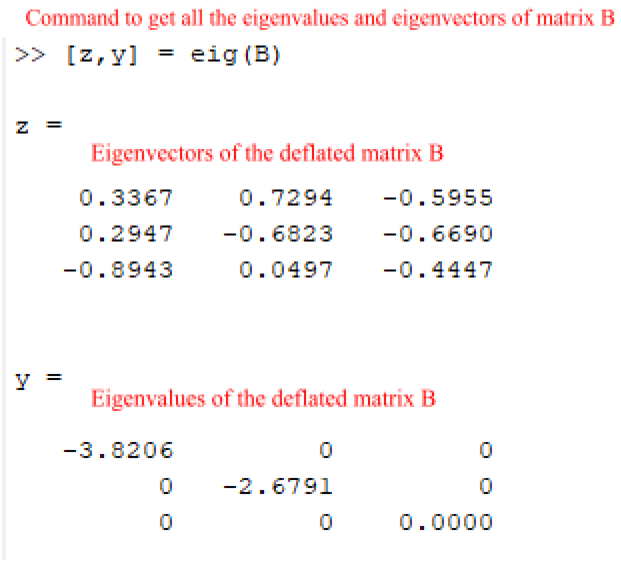
```
>> eig(B)

ans =

   -3.4358
   -2.8953
    0.0000

>> [lambda2, x2, B] = pm2(A, lambda1, x0)

lambda2 =

   -3.8206


x2 =

   -0.3365
   -0.2949
    0.8943


B =

   -1.8583    0.9542    1.0532
    0.9542   -1.5791    1.0979
    1.0532    1.0979   -3.0623

>> [z,y] = eig(A)

z =

   -0.3367    0.7294    0.5955
   -0.2947   -0.6823    0.6690
    0.8943    0.0497    0.4447


y =

   -0.4597         0         0
         0    0.6818         0
         0         0    3.3609
```

Function to get eigenvalues and vectors
Correct eigenvalues within the deflated matrix B
Showing B for testing purposes
Command to start PM2
Pm2.m estimated eigenvalue
Pm2 estimated eigenvectors
The deflated matrix B
Function to get eigenvalues and vectors of A
All the eigenvectors using eig() function
All the eigenvalues using eig() function

I have encountered some issues where eigenvectors and eigenvalues are not calculated accurately by my PM2.m function. Upon investigation the eigenvalues must be calculated incorrectly. Failing to stop either to not stop within the specified tolerance this could be the cause of the issue as the eigenvectors correspond to the third set of eigenvalues, indicating that the process does not stop after finding the second one. The deflated matrix B of PM2 gets the last eigenvalue which should represent the second eigenvalue of A after deflation. However this value appears to be incorrect. Infact PM2.m seems to only accurately calculate eigenvectors for the third eigenvalue. The code seem close to functioning correctly, but the issue seems to persist for reason I am unsure off. Here are a errors that it might be.

1) Incorrect deflation of matrix B
2) Inaccurate tolerance check, preventing the function from stopping at the right time
3) Calculating the eigenvalues incorrect.

However I would like to point out that the eigsolve function works correctly. It's just that if the second eigenvalue is incorrect the third one will also be incorrect same goes for the eigenvectors, because the last eigenvectors were found by pm2 there will be no more eigenvectors to find.



```
>> [z,y] = eig(B)

z =

    0.3367    0.7294   -0.5955
    0.2947   -0.6823   -0.6690
   -0.8943    0.0497   -0.4447


y =

   -3.8206         0         0
         0   -2.6791         0
         0         0    0.0000
```

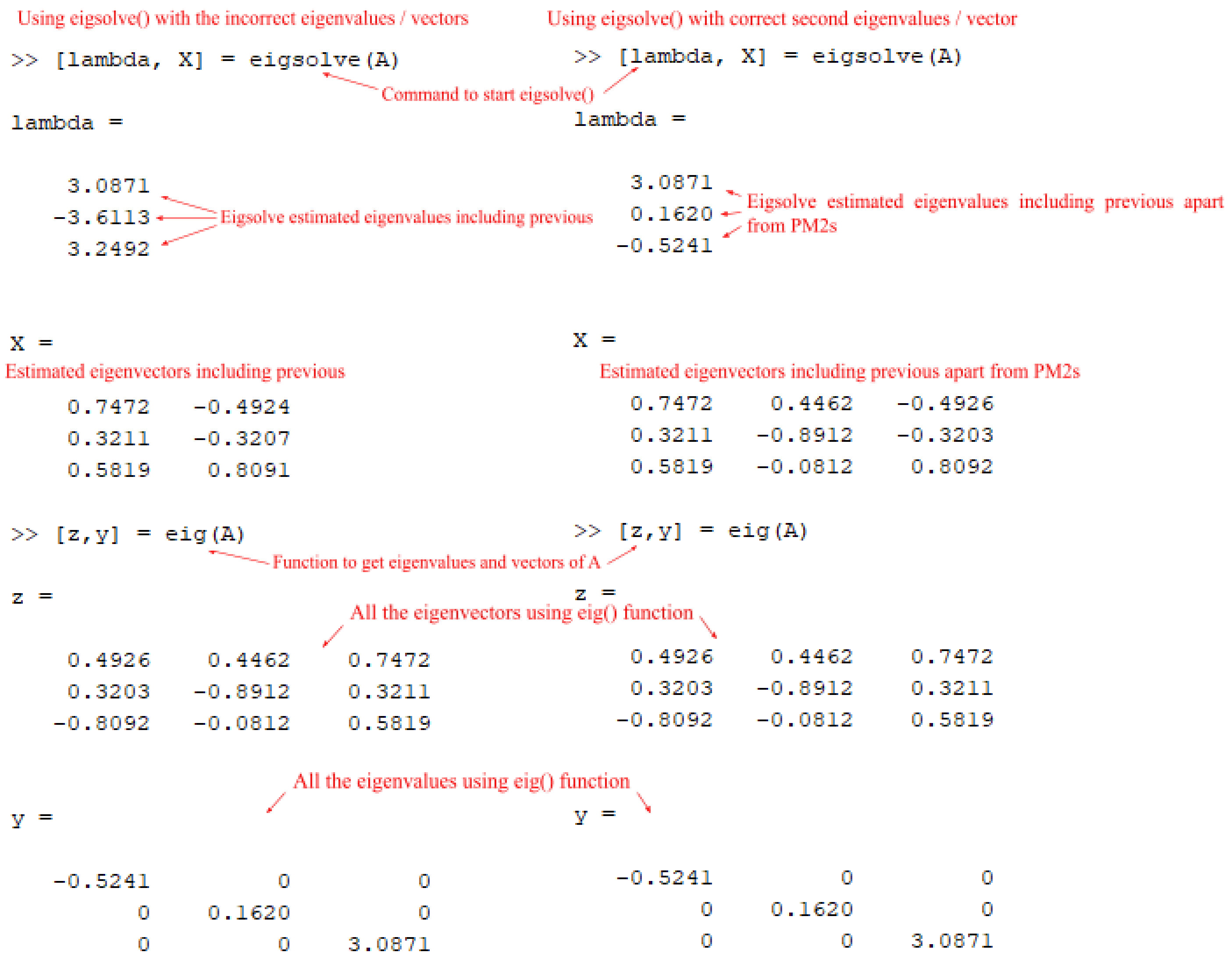
Command to get all the eigenvalues and eigenvectors of matrix B
Eigenvectors of the deflated matrix B
Eigenvalues of the deflated matrix B

Task 3 - eigsolve.m with testing

```
% This function pulls pm1.m and pm2.m together and runs them after that is
% will calculate the third eigenvalues and there corresponding eigenvectors
% using the trace and the null function within matlab.
function [lambda, X] = eigsolve(A)
x0 = rand(3,1); % Creating the initial guess vector
% The following 4 lines below will check if the vector is actually a matrix
% vector. the size function will get the size of the vector and store the
% amount of rows in the rows & the amount of columns in the column section
% of the variable array. Then the if statement will check using the operators
% '~=' (not equal to) , '||' (or) if the vector is the correct size the
% code shall continue to execute if not we shall throw and error and exit
% the execution of the program.
[row, column] = size(x0);
if row ~= 3 || column ~= 1
   error('eigsolve: The vector must be a 3x1.');
end
% The following 4 lines below will check if the matrix is a 3x3 matrix.
% Firstly we need to get the row and column sizes of A we can use the size
% function to get the amount of rows & columns we then store them in a
% variable array. the row will store the amount of rows and the columns will
% store the amount of columns. the if statement line will then check using
% the operators '~=' (not equal to) , '||' (or) if the matrix is the
% correct size the code will carry on if not the code will throw an error to
% end the program if not the rest of the code can execute.
[row, column] = size(A); % Gets the size of the matrix
if row ~= 3 || column ~= 3 % Checks if the rows & columns equal to 3
   error('eigsolve: The matrix must be a 3x3 matrix.'); % Display and error
message if it's not a 3x3 matrix
end % Ends the if statement
% The following 3 lines below will check if the matrix is symmetric. The if
% statement will use the operator '~' which inverts the result of the "is
% equal" function so the statement basically says if not equal. To check if
% the matrix is symmetric we need to plug the original matrix in and then
% the transposed version using the ' operator which transposes the matrix.
% If the transpose of a matrix is not the same as the original matrix,
% then the matrix is not symmetric and we will throw an error to end the
% program. if not the rest of the code can execute.
if ~isequal(A, A')  % Checks if the matrix is symmetric
   error('eigsolve: The matrix must be symmetric.'); % Display and error
message if the matrix is not symmetric
end % Ends the if statement
[lambda1, x1] = pm1(A, x0); % Using pm1.m to calculate the first eigenvalue /
vector
[lambda2, x2] = pm2(A, lambda1, x0); % Using pm2.m to calculate the second
eigenvalue / vector
% Calculating the last eigenvalue using the trace method
lambda3 = trace(A) - lambda1 - lambda2; % Determine the third eigenvalue by
subtracting the first two eigenvalues from the trace of A
```

```
% This code firstly adjusts the matrix A by subtracting the third
% eigenvalue multiplied by an identity matrix of the same size. Shifting
% alters the eigenvalues while preserving its eigenvectors. The null
% function finds the eigenvector associated with 'lambda3' by finding the
% null spaces of the shifted matrix the tolerance is then used to make
% sure the values are in the correct tolerance.
x3 = null(A - (lambda3 * eye(3)), 0.0001); % Calculate the eigenvector
corresponding to the eigenvalue lambda3 of matrix A.
x3 = x3 / norm(x3); % Normalizing the eigenvectors
lambda = [lambda1; lambda2; lambda3]; % Storing all of the calculated
eigenvalues in a vector named 'lambda'
X = [x1, x2, x3]; % Storing all of the calculated eigenvectors in a matrix
column named 'X'
end % End of the eigensolve function
```

I will not test to see if the error checking works correctly because it is the same error checking code that is within the PM1 Function so to check the same thing will be a waste of time as it already works.


Using eigsolve() with the incorrect eigenvalues / vectors
Using eigsolve() with correct second eigenvalues / vector
Command to start eigsolve()
Eigsolve estimated eigenvalues including previous
Eigsolve estimated eigenvalues including previous apart from PM2s
Estimated eigenvectors including previous
Estimated eigenvectors including previous apart from PM2s
Function to get eigenvalues and vectors of A
All the eigenvectors using eig() function
All the eigenvalues using eig() function

```
>> [lambda, X] = eigsolve(A)

lambda =

    3.0871
   -3.6113
    3.2492


X =

    0.7472   -0.4924
    0.3211   -0.3207
    0.5819    0.8091

>> [z,y] = eig(A)

z =

    0.4926    0.4462    0.7472
    0.3203   -0.8912    0.3211
   -0.8092   -0.0812    0.5819


y =

   -0.5241         0         0
         0    0.1620         0
         0         0    3.0871
```

```
>> [lambda, X] = eigsolve(A)

lambda =

    3.0871
    0.1620
   -0.5241


X =

    0.7472    0.4462   -0.4926
    0.3211   -0.8912   -0.3203
    0.5819   -0.0812    0.8092

>> [z,y] = eig(A)

z =

    0.4926    0.4462    0.7472
    0.3203   -0.8912    0.3211
   -0.8092   -0.0812    0.5819


y =

   -0.5241         0         0
         0    0.1620         0
         0         0    3.0871
```

As you can see upon testing that eigsolve does work if the correct second eigenvalue and vector were computed correctly. On the left you see the eigsolve function takes in a 3x3 symmetric matrix A and outputs all of its corresponding eigenvalues and eigenvectors. However the second eigenvalue and vector is incorrect PM2 actually calculates the third eigenvector and also calculates an incorrect eigenvalue. This means that eigsolve cannot solve for the third vector because the third vector has already been computed by PM2. However if we comment out the PM2 function like this

```
%[lambda2, x2] = pm2(A, lambda1, x0);
```

And instead manually input the correct second eigenvalue and eigenvectors like this

```
 lambda2 = 0.1620;
x2 = [0.4462;-0.8912;-0.0812];
```

Then run the eigsolve function the correct third eigenvalue and eigenvectors are displayed along with the second and first same goes for the eigenvectors as you can see in the image on the right.

Reasoning behind error checking all functions

The reason I have added error checking into all of the scripts is because if someone wants to only get the first eigenvalue for example and does not want to get all of them then they will just run pm1.m If I moved all of my error testing into eigsolve.m pm1.m and pm2.m would have no error checking. Making the person who just wants to run pm1 at risk of unintentional errors.

Cramer’s Rule

Task 1: Definition of Cramer’s Rule

Cramer's Rule is a formula you can use for solving variables in a system of equations with any number of unknowns (Song, Wang & Yu, 2018). Primarily, this rule is key in solving a system of linear equations, deriving general solutions, proving a theorem, and solving problems in differential geometry using determinants. Determinants are real numbers that can be used in multiple applications. One determinant comes from the coefficient matrix, while the other is created by replacing the column for a specific variable with the expected 'answer.' When using Cramer’s Rule, you are guided to form specific variables and divide them to find the appropriate values (Song et al., 2018). Ideally, the denominator of all divisions remains the determinant of the coefficient matrix. For numerators, you replace the coefficient of the said variable's entries in the coefficient matrix with the column vector containing the constants. In solving a square system, one variable is eliminated using row operations. For instance, in solving for x, the equation is multiplied by the coefficient of y, eliminating the variable y. The denominator for both x and y is the determinant of the coefficient matrix. The point of Cramer's Rule is that you do not have to solve the entire system to get one value that is needed (Song et al., 2018). Solving for one variable aids in calculating the others, saving a significant amount of time.

Cramer's Rule gives unique solutions to a system of equations. If the system has no solution or has infinite numbers, however, it is indicated by zero as a determinant (Justino & van den Berg, 2012). To find out, there is a need to perform elimination of the system. When two rows are interchanged, the determinant changes sign. In the case of a matrix with a column or rows of zero, the determinant’s value is zero (Gefter et al., 2020). In a system of linear equations with n variables, x_1, x_2, …, x□, denoted in the form AX=B, A stands for coefficient matrix, X for column matrix with variables, and B for column matrix with the constants. The determinant is calculated as $D = |A|$, Dx_1, Dx_3, Dx_n. in which I stands for 1, 2, 3, 4, 5, n. For linear equations, the solution is expressed in terms of one, two, or more conditions that can take unpredictable values. It means that the proof of the rule uses linearity concerning any given column. In a 3 by 3 matrix, for instance, the sum of the entries down each of the three diagonals is calculated. In such a case, when the coefficient determinant is zero and other determinants are real values

(1, 2, 3, 4…….), the system is described as incompatible. This does not mean that having all determinants equal to zero implies the system is indeterminate. It is not possible to divide by zero. *D* = 0 implies that the system of equations does not have a unique solution but rather is unbalanced. In this case, the solution is infinite. The key to the rule is replacing the variable column of focus and computing the determinants to get the required value. A system of linear equations cannot have exactly two solutions. However, there may be no solutions. Also, when D = 0, there can be two possibilities. The first possibility is that at least one of the numerator determinants is zero. It can also mean that none of the numerator determinants is zero (Song & Yu, 2019). The application of Cramer's Rule helps to determine whether the system has an infinite number of solutions or has no solution altogether.

Task 2: Cramer's Rule Solving Equations
Student number = U2166829
a = 2
b = 9
$2x_1 + x_2 = a$
$-4x_1 - x_2 = b$

$$\overset{a_1}{2}x_1 + \overset{b_1}{x_2} = 2 \;\; d_1 \qquad a_1 = 2,\ b_1 = 1,\ d_1 = 2$$
$$\underset{a_2}{-4}x_1 - \underset{b_2}{x_2} = 9 \;\; d_2 \qquad a_2 = -4,\ b_2 = -1,\ d_2 = 9$$

To begin, one variable is eliminated using row operations. When solving x_1, for instance, the equation is multiplied by the opposite of the coefficient of x_2. The two equations are then added to eliminate x_2. To solve using Cramer's Rule, there is a need to find the determinant of the coefficient Matrix *A* and represent it by *D*. *D* stands for determinant. It can be found by using the x, y, and z values in the equation. The properties of the determinants must be well understood to help in the computation process. The same formula applies in the computation of x_2.

$2x_1 + x_2 = 2$
$-4x_1 - x_2 = 9$

To solve for x_1, the x_1 is replaced with the constant column as shown below. The variable is eliminated using row operations. The denominator of both x_1 and x_2 remains the determinant of the coefficient matrix.

$$x_1 = \frac{\begin{vmatrix} d_1 & b_1 \\ d_2 & b_2 \end{vmatrix}}{\begin{vmatrix} a_1 & b_1 \\ a_2 & b_2 \end{vmatrix}} = \frac{\begin{vmatrix} 2 & 1 \\ 9 & -1 \end{vmatrix}}{\begin{vmatrix} 2 & 1 \\ -4 & -1 \end{vmatrix}} = \frac{2\times(-1) - (9\times1)}{2\times(-1) - (-4\times1)} = \frac{-2-9}{-2-(-4)} = \frac{-11}{2}$$

To solve for x_2, x_2 is replaced with the constant column.

$$x_2 = \frac{\begin{vmatrix} a_1 & d_1 \\ a_2 & d_2 \end{vmatrix}}{\begin{vmatrix} a_1 & b_1 \\ a_2 & b_2 \end{vmatrix}} = \frac{\begin{vmatrix} 2 & 2 \\ -4 & 9 \end{vmatrix}}{\begin{vmatrix} 2 & 1 \\ -4 & -1 \end{vmatrix}} = \frac{2\times 9 - (2\times(-4))}{2\times(-1) - (-4\times 1)} = \frac{18-(-8)}{-2-(-4)}$$

$$= \frac{\overset{13}{\cancel{26}}}{\underset{1}{\cancel{2}}}$$

$$= 13$$

The solution is
$x_1 = -11/2$ $x_2 = 13$

Task 3: Cramer’s Rule in Practice
In practice, Cramer's Rule is critical in solving one of the variables in a system of equations without having to compute the whole system of equations. It implies that if you need one variable from the system, the rule allows for the computation of just that variable (Justino & van den Berg, 2012). However, Cramer’s Rule is not always used in practice because a unique and more definite solution does not always exist. In calculations, there could be infinitely many answers, or no answers at all (Justino & van den Berg, 2012). Cramer’s Rule only works with square and nonsingular coefficient matrices (small systems) to give a definite answer. You are required to evaluate a determinant for each equation. The method becomes computationally more involved for larger systems (Song et al., 2018). In this regard, the rule is only applicable when exactly one solution exists and there is a need to calculate the other variable.

In the module, a range of topics are covered, including Linear Transformation and Determinants, LU Factorization, and Linear Equations (Rank and Dependence, and Gaussian Elimination). Cramer's Rule is specifically designed to solve systems of linear equations where the number of equations matches the number of variables (Song et al., 2018). It's often challenging to surpass the effectiveness of Gaussian Elimination, which is an algorithm for simplifying matrices into a more manageable form, facilitating the calculation of ranks. Essentially, the determinant, which is a special calculation in linear algebra that produces a single number from a matrix, is used in a unique way and form (Song & Yu, 2019). This means that using Cramer's Rule generally involves twice as many operations and maintains a similar level of numerical precision when dealing with the same matrices. Additionally, this rule is particularly useful when the solution for just one unknown variable is needed. In Gaussian Elimination, the goal is to reduce or change the original extended matrix into one primary form using row operations. Any zero rows that appear at the bottom of the matrix remain to the right of the first nonzero entry in the higher row (Grcar, 2011). Only two nonzero rows remain in the final matrix. In each row, the first three elements are the coefficients of x, y, and z in the equation. The fourth element is the right-hand side of the equation. In the process of Gaussian Elimination, the operation stops when zeros are produced in the first column below the diagonal and after adding the second row to the third. Primarily, the back substitution is used to obtain the values of the variables (Grcar, 2011). This is quite different from Cramer’s Rule in which determinants are used to find the value of the unknowns. For large systems, Cramer's rule is not very efficient because the determinant calculation becomes prohibitive, especially if done in a known manner. Gaussian Elimination, on the other hand, is more efficient and can handle non-square matrices as well as find the rank of the matrix in a formal operation.

Cramer's Rule can be sensitive to small numerical errors even in simple 2 2 systems, meaning that minor inaccuracies in data or variables can lead to significant changes in the outcome. One of the most crucial uses of Cramer's Rule is in cryptography, where secure messages are sent in encoded matrix forms. Decoding this information requires a matrix that has a specific property (a determinant) and is reversible (invertible) (Song & Yu, 2019). Moreover, Cramer's Rule is useful in solving certain optimization problems in mathematics (integer programming problems) where the constant matrix has a special characteristic of being composed of integers and its transformations maintain integer values. This property makes solving integer programs more straightforward. However, to achieve the correct and desired results, one needs to be adept at performing derivative calculations and interpreting geometric principles.

References

Gefter, S. L., Martseniuk, V. V., Goncharuk, A. B., & Piven, A. L. (2020). Analogue of the Cramer Rule for an Implicit Linear Second Order Difference Equation over the Ring of Integers. *Journal of Mathematical Sciences*, *244*(4). https://doi:10.1007/978-3-030-60107-2_16

Grcar, J. F. (2011). Mathematicians of Gaussian elimination. *Notices of the AMS*, *58*(6), 782-792. https://community.ams.org/journals/notices/201106/rtx110600782p.pdf

Justino, J., & van den Berg, I. (2012). Cramer's rule applied to flexible systems of linear equations. *Electronic Journal of Linear Algebra*, *24*, 126-152. https://doi.org/10.13001/1081-3810.1584

Song, G. J., & Yu, S. W. (2019). Cramer's rule for the general solution to a restricted system of quaternion matrix equations. *Advances in Applied Clifford Algebras*, *29*(5), 91. https://doi.org/10.1007/s00006-019-1000-1

Song, G. J., Wang, Q. W., & Yu, S. W. (2018). Cramer's rule for a system of quaternion matrix equations with applications. *Applied Mathematics and Computation*, *336*, 490-499. https://doi.org/10.1016/j.amc.2018.04.056