Project 2

Malin Eriksen
(Dated: October 11, 2022)

https://github.com/malineri/fys3150/tree/main/project_2

PROBLEM 1

In this project we will be looking at a horizontal beam. The beam will be of length L, and a force F will be applied in the endpoint of this beam. This will be described by the second-order differential equation (1).

$$\gamma \frac{d^2u(x)}{dx^2} = -Fu(x). \tag{1}$$

We begin with scaling this formula into a dimensionless equation, by changing the x to a unitless variable $\hat{x} = \frac{x}{L}$. The derivation term then becomes,

$$\frac{d}{dx} = \frac{d\hat{x}}{dx}\frac{d}{d\hat{x}} = \frac{1}{L}\frac{d}{d\hat{x}}.$$

We have a second order differential equation, so we have to do this operation twice. Adding this to equation(1) we get the formula,

$$\frac{\gamma}{L^2}\frac{d^2u(\hat{x})}{d\hat{x}^2} = -Fu(\hat{x}).$$

Which we rearrange slightly,

$$\frac{d^2u(\hat{x})}{d\hat{x}^2} = -\frac{FL^2}{\gamma}u(\hat{x}).$$

Now we introduce a new variable lambda $\lambda = \frac{FL^2}{\gamma}$ then we find that the dimensionless equation of our first equation can be written as,

$$\frac{d^2u(\hat{x})}{d\hat{x}^2} = -\lambda u(\hat{x}). \tag{2}$$

TABLE I. Eigenvalues and corresponding eigenvectors of a tridiagonal matrix A

Eigenvalues λ	Eigenvectors $\vec{v}$
0.1981	[0.2319, 0.4179, 0.5211, 0.5211, 0.4179, 0.2319]
0.7530	[-0.4179, -0.5211, -0.2319, 0.2319, 0.5211, 0.4179]
1.5550	[0.5211, 0.2319, -0.4179, -0.4179, 0.2319, 0.5211]
2.4450	[0.5211, -0.2319, -0.4179, 0.4179, 0.2319, -0.5211]
3.2470	[0.4179, -0.5211, 0.2319, 0.2319, -0.5211, 0.4179]
3.8019	[-0.2319, 0.4179, -0.5211, 0.5211, -0.4179, 0.2319]

PROBLEM 2

To solve this problem we will be using matrices. We write a short program to set up a tridiagonal NxN matrix A, when N = 6. We want this matrix to solve the classic eigenvector, eigenvalue problem $\mathbf{A}\vec{v} = \lambda\vec{v}$ in the same way we formulated our dimensionless equation(2).[**?**] We being with creating an algorithm with a general function that determines a tridiagonal matrix. This algorithm we see in Algorithm 1.

Algorithm 1 Creating function that takes values from diagonal and makes a tridiagonal matrix.

```
  int main(){
  arma::mat create tridiagonal(int n, double a, double d, double e)
  arma::mat A = arma::mat(n, n, arma::fill::eye);                    ▷ n x n identity matrix
  int N = 6                                      ▷ We begin with setting the length of the matrix N = 6
  A(0, 0) = d; A(1, 0) = e; A(0, 1) = a; ▷ Manually filling in values of A, Could potentially be done more efficiantly in a loop.
  return A;
  }
```

We now want to put in values of N, a, d and e, to determine our exact values. We want those to be a, $e = -1/h^2$ and $d = -2/h^2$. And we also want the program to print the eigenvalues and eigenvectors. A program returning the matrix, its eigenvalue and its corresponding eigenvectors is written in Algorithm 2.

Algorithm 2 Our main containing the values of our matrix, and printing the matrix, its eigenvector and eigenvalues.

```
  int main(){
  int N = 6.; float h = 1.; float a = (-1.)/(h*h); float d = (2.)/(h*h);      ▷ Setting values we use in the matrix
  arma::mat A = create tridiagonal(N, a, d, a);                              ▷ Create matrix A from function
  int N = 6                                      ▷ We begin with setting the length of the matrix N = 6
  arma::vec eigval;
  arma::mat eigvec;
  arma::eig sym(eigval, eigvec, A);
  int width = 18; int prec = 10;                                        ▷ Parameters for output formatting
  std::cout << "#" << std::setw(width-1) << A
  << std::endl;
  std::cout << "#" << std::setw(width-1) << eigvec
  << std::endl;
  std::cout << "#" << std::setw(width-1) << eigval
  << std::endl;
  return 0;
  }
```

Now that we have found the eigenvalues and corresponding eigenvectors we print the results, to make it easier to compare the analytical and numerical solutions we remove the /h2 part of the a, d and e values, so that we use the simple tridiagonal matrix with 2 on the diagonal and -1 on the upper and lower diagonal. This gives us the results we can see in table(1). We can check if these values are correct by comparing the results to the analytical results with the formulas

$$\lambda^i = d + 2a\cos\left(\frac{i\pi}{N+1}\right)$$

TABLE II. Eigenvalues and corresponding eigenvectors of a tridiagonal matrix A

Eigenvalues λ	Eigenvectors $\vec{v}$
0.1981	[0.43388374, -0.78183148, 0.97492791, -0.97492791,0.78183148, -0.43388374]
0.7530	[0.78183148, -0.97492791, 0.43388374, 0.43388374, -0.97492791, 0.78183148]
1.5550	[0.97492791, -0.43388374, -0.78183148, 0.78183148, 0.43388374, -0.97492791]
2.4450	[0.97492791, 0.43388374, -0.78183148, -0.78183148, 0.43388374, 0.97492791]
3.2470	[0.78183148, 0.97492791, 0.43388374, -0.43388374, -0.97492791, -0.78183148]
3.8019	[0.43388374, 0.78183148, 0.97492791, 0.97492791, 0.78183148, 0.43388374]

$$\vec{v}^i = \left[\sin\left(\frac{i\pi}{N+1}\right), \sin\left(\frac{2i\pi}{N+1}\right), ..., \sin\left(\frac{Ni\pi}{N+1}\right)\right]^T$$

Where i = 1, ... , N. We make a program that solves this algorithm for our N = 6 case. This program is called *analytical 2.py*, and the algorithm is added to the github repo. We get the same values on the eigenvalues, but we get different eigenvectors, something that tells us there must be a small mistake in one of the programs. If time I will go back and look at potential mistakes in the program. In table(2) we see the values we get from the analytical solution.

PROBLEM 3

a)

In general we would like to be able to take any matrix and find the operations needed to create this into a identity matrix using the Jacobi rotation method. And then later from this method finding the eigenvectors and values of the matrix. To be able to create a program that uses the Jacobi rotation method we first write a program that takes a matrix and finds the biggest off diagonal element. The algorithm(3) is used for this function.

Algorithm 3 Creating function that finds the biggest off diagonal element in a matrix A

```
double max_offdiag_symmetric(const arma::mat& A){
double m = INT_MIN;
for (int i = 0; i < A.n_rows; i++){
for (int j = 0; j < i; j++){
if (abs(A(i, j)) > m){
m = A(i, j);
}
}
}
return m;
}
```

b)

Now we apply this function to a known matrix to see if we get the output we are looking for. We use the matrix A from equation(3).

$$A = \begin{bmatrix} 1 & 0 & 0 & 0.5 \\ 0 & 1 & -0.7 & 0 \\ 0 & -0.7 & 1 & 0 \\ 0.5 & 0 & 0 & 1 \end{bmatrix} \tag{3}$$

We check only under the diagonal for our program since our matrix is symmetric over and under the diagonal. We know already from our matrix that the biggest absolute value is 0.7, and that the biggest plus value is 0.5. In our program we have the print of 0.5, unless we take the absolute value of the matrix. If we do so we find the greatest value to be 0.7. This we can test with other matrices aswell, and if it turnes out that the matrix has a negative value that is bigger than the positive value we may change the algorithm(3) to contain $m = A(A_n.rows - 1, 0)$, to have the biggest negative value. Otherwise we will have the same result in eigenvalues and vectors, only that this will also then be the absolute value. The algorithm implementing the matrix to the function above is represented in algorithm(4).

Algorithm 4 Our main containing the values of our matrix, and printing the matrix, its eigenvector and eigenvalues.

```
int main(){
int N = 4;                                                        ▷ Setting values we use in the matrix
arma::mat A = arma::mat(N, N, arma::fill::eye);
A(0, 3) = 0.5; A(1, 2) = -0.7; A(2, 1) = -0.7; A(3, 0) = 0.5;      ▷ filling in the exact matrix elements
arma::mat B = abs(A);          ▷ taking the absolute value of the matrix, otherwise the element -0.7 will not be counted
double max_value = max_offdiag_symmetric(B);
std::cout¡¡ max_value;
return 0;
}
```

PROBLEM 4

a)

The Jacobi rotation method is a method where diagonalize A, so that we have the equation $S^T AS = D$. The method is based on doing a series of transformations until we end up with A being something close enough to the diagonal, D. The number of iterations we have to do will then be described as m. And for each step we apply the S^T and S for the step before M-1. In general this can be described by equation(4).

$$S_{M-1}T...S_2^T S_1^T AS_1 S_2...S_{M-1} = A^m = D \tag{4}$$

The elements in the S vector will then be the eigenvectors of A, and the diagonal D will contain the eigenvalues. Rewriting this to an algorithm we will use equation(5).

$$A^{m+1} = S_M^T A^{(M)} S_M \tag{5}$$

The matrix that does the rotation is the S_M matrix, this will rotate the matrix clockwise at four important points $(k, k) = \cos\theta$, $(k, l) = \sin\theta$, $(l, k) = -\sin\theta$ and $(l, l) = \cos\theta$. These points will rotate the matrix to make our biggest off-diagonal value in A go to zero. The algorithm we make will find the biggest value, do a rotation, then search for a new biggest value, do another rotation, and do this on repeat till we are close enough to the diagonal matrix. See equation(6).

$$S_M = \begin{bmatrix} 1 & & & \\ & \cos\theta & \sin\theta & \\ & -\sin\theta & \cos\theta & \\ & & & 1 \end{bmatrix} \tag{6}$$

We will also introduse the vector R, that together with S_M creates the span of eigenvectors. The first element of R is the identity matrix.

$$R^{[M]} = R^{[M-1]} S_{M-1} = IS_1, S_2, ...S_{M-1} \tag{7}$$

From this matrix we can put up a set of equations, for the different elements in the matrix A, that we can use to make the algorithm.

$$A^{[2]}(l, l) = A^{[1]}(l, l)\cos^2\theta - 2A(k, l)\cos\theta\sin\theta + A^{[1]}(k, k)\sin^2\theta \tag{8}$$

$$A^{[2]}(k, k) = A^{[1]}(k, k)\cos^2\theta + 2A(k, l)\cos\theta\sin\theta + A^{[1]}(l, l)\sin^2\theta \tag{9}$$

$$A^{[2]}(k, l) = (A^{[1]}(l, l) - A^{[1]}(k, k))\cos\theta\sin\theta + A^{[1]}(k, l)(\cos^2\theta - \sin^2\theta) \to 0. \tag{10}$$

We reqire the last equation to be zero, and from these equations we get the formula,

$$\left(\frac{A^{[1]}(l,l) - A^{[1]}(k,k)}{A^{[1]}(k,l)}\right)\cos\theta\sin\theta + \cos^2\theta - \sin^2\theta = 0 \tag{11}$$

To make the algorithm simpler we introduce the notation $\tan\theta = t = \frac{s}{c}$, $\tau = \frac{A(k,k)-A(l,l)}{2(k,l)}$. A is a symmetric matrix, which means the element (k, l) = (l, k). This gives us,

$$\sin^2\theta + 2\epsilon\cos\theta\sin\theta - \cos^2\theta = 0 \tag{12}$$

$$t^2 + 2\epsilon t - 1 = 0. \tag{13}$$

$$-\tau\sqrt{1+\tau^2} = t \tag{14}$$

From this we can compute $c = \frac{1}{\sqrt{1+t^2}}$ and $s = ct$. And from all this we will now be able to compute $A^{[2]}(l,l)$ and $A^{[2]}(k,k)$. And we can make loops of iterations to do this for the next orders as well. This Algorithm is slightly longer, so I will leave this program in the github repo with the name *problem_4.cpp*.

b)

If I would be able to make this program work I would be able to do it for a matrix with N = 6. And then I would analyse the results we get from the jacobi rotation method with the analytical results we got from problem 2. I would also see how it compares to the eigenvectors and eigenvalues we found from the armadillo implementation in problem 2.

PROBLEM 5

a)

Whenever we have made the program above function, we may use it to figure out the number of iterations needed for a specific matrix before it has reached an almost diagonal matrix. We can run the program with different sizes on the matrix. We choose N = 10, 100 and 1000 for example, and then we see the number of iterations corresponding to this before out matrix becomes close to diagonal. This number is expected to increase more than linearly with N. This is because the rotation matrix will rotate one number at the time, and when rotating, another number that has previous been rotated and now is zero, may be rotated back, and now again be some value. This means that the greater the matrix, the higher the possibility to change back an already fixed number will increase, and therefore the number of iterations will increase more compared to the increasing of the size of the matrix itself. I did not get my program working in the previous task, and therefore making a plot or table with the numbers for these two will be hard to find. But I have just made some random guesses to be able to make some sort of a plot of something that may be something close to the actual graph. The numbers I have guessed, and used in the plot is possible to see in table(3) Most probably the matrix A is not as big as N here, but probably the number of transformations, or iterations in the

TABLE III. Number of iterations/tranformations before the matrix D is close to diagonal, for different sized A NxN matrices.

Iterations [i]	Size of matrix [N]
10	100
20	1000
50	3000
100	15000
500	50000
1000	1000000

program, may be even bigger for smaller N, this may end up giving the program troubles and making it a slow one. In figure(1) we see the plot of this table. Since I had very little values the plot is not very smooth, but it gives an indication of what I thought.

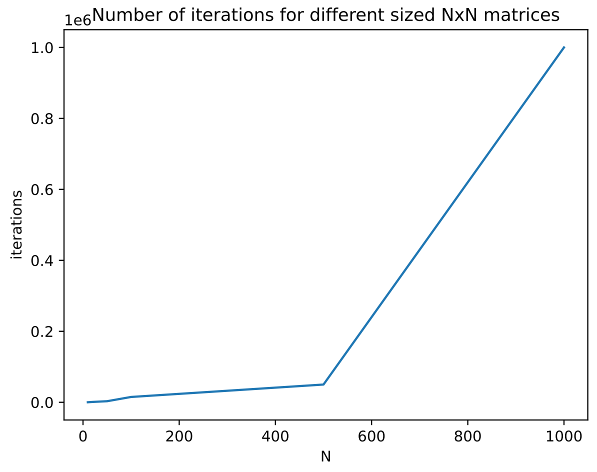

FIG. 1. Plot of the values descrived in table(3).

b)

From Wikipedia I find that the definition of a dense matrix is a matrix where most of the elements are non-zero. Which means that we would have to have even more iterations in the method before we are able to end up with a diagonal matrix.

PROBLEM 6

a)

With the discretized x we will use n = 10 steps, which means that our matrix A will be of size n-1, when we remove the endpoints. We will then use the jacobi iteration to make a plot of the three eigenvectors corresponding to the three lowest eigenvalues. From our S matrix we then find the eigenvectors S[i] corresponding to the lowest eigenvalues that we find in the diagonal D. We can then make a program that finds the lowest value in the matrix D, and prints the column and row. From this information we find the corresponding x values. And then we can plot the x against the vectors F[i] for the 3 lowest values.

b)

We can repeat the same operation with changing n = 10 steps to n = 100 steps.