Project2 FYS3150
Tiril Sørum Gransjøen

October 10, 2023

The code written to answer the problems in this project, can be viewed and copied in the GitHub repo, `https://github.com/tirilsg/FYS3150-Project2-secondtry`.

1 Problem 1

This deductinon is performed by assuming that L is in fact a constant, not dependant on x. We have the definintion $\vec{x} = x/L$, that we use to show that equation (4) from our assignment text can be written as equation (5). The equation (4) is

$$\gamma \frac{d^2u(x)}{dx^2} = -Fu(x) \tag{4}$$

We use our definintion for $\vec{x}$, and deduce:

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

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

$$\frac{Ld^2u(\vec{x})}{L^2d(\vec{x})^2} = -\frac{FLu(\vec{x})}{\gamma}$$

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

When equation is given

$$\frac{d^2u(\vec{x})}{d\vec{x}^2} = -\lambda u(\vec{x}) \tag{5}$$

where $\lambda = \frac{FL^2}{\gamma}$, we see that we have shown that equation (4) can be written as equation (5).

2 Problem 2

We write a program, *prob2.cpp*. The code is devided into four parts - first we declare our function $createA()$, which uses $vec()$ to define an arbitraty tridiagonal NxN matrix A, $A = tridiag(a, d, a)$. Thereafter we define two more functions *eigvec* and *eigval*, which uses the formulas from the assignment text and computes the eigenvalues and eigenvectors for a tridiagonal matrix. Finally it defines the function $solveMat$ that uses the three previosly mentioned functions to compute the eigenvalues and eigenvectors analyticlly, and compares them to the ones computed by the integrated armadillo-function eigsym. In our file *main.cpp* the code can be ran by calling the function $solveMat$ for arbitrary values N, a and d. We test that the function $solveMat$ works by calling the function for N=6, a=2 and d=5, and both the arguments *visual* and *match* set to be true. This allows $solveMat$ to print the eigenvalues and eigenvectors computed both analytically and by Armadillo's function into the terminal, as well as the statement that confirms the match between the methods.

Table 1: The code for problem 2 is ran, and in the terminal we get printed:

Calculation of eigenvalues using different methods printed in terminal

Eigval function:	Armadillo:
1.3961	1.3961
2.5060	2.5060
4.1099	4.1099
5.8901	5.8901
7.4940	7.4940
8.6039	8.6039

Calculations of eigenvectors using different methods printed in terminal

Eigvec function:

0.2319	0.4179	0.5211	0.5211	0.4179	0.2319
-0.4179	-0.5211	-0.2319	0.2319	0.5211	0.4179
0.5211	0.2319	-0.4179	-0.4179	0.2319	0.5211
-0.5211	0.2319	0.4179	-0.4179	-0.2319	0.5211
0.4179	-0.5211	0.2319	0.2319	-0.5211	0.4179
-0.2319	0.4179	-0.5211	0.5211	-0.4179	0.2319

Armadillo:

0.2319	-0.4179	-0.5211	-0.5211	0.4179	0.2319
-0.4179	0.5211	0.2319	-0.2319	0.5211	0.4179
0.5211	-0.2319	0.4179	0.4179	0.2319	0.5211
-0.5211	-0.2319	-0.4179	0.4179	-0.2319	0.5211
0.4179	0.5211	-0.2319	-0.2319	-0.5211	0.4179
-0.2319	-0.4179	0.5211	-0.5211	-0.4179	0.2319

Eigenvalues and eigenvectors match between the two methods.

From these results, we can indeed see that the functions implemented work as intended.

3 Problem 3

We write a program in our file *prob3.cpp*, that follows the recipe given in the assignment text; we define a function maxoffdiagsymmetric that runs through the upper elements in a symmetric matrix, and finds the largest absolute value it contains. The function simply returns this value to us. We use the function signature from the assignment text, double maxoffdiagsymmetric, which means it takes the matrix A, as well as two integers k and l as arguments. The integers k and l are used to navigate throug A. To test that the function works properly, it is called in *main.cpp* for the A in our assignment text. We see that it does in fact return the value -0.7 to us, so the function works as intended. Printed in the terminal when the code is ran:

```
Problem 3
The largest off-diagonal element is abs(0.7), at indices (1, 2).
```

4 Problem 4

We implement Jacobi's rotation algorithm to solve egquiation 6 in the assignment text, in three steps. The program does to a large extent follow the recipe presented in *CodeSnippets*. First, we define a function *JacobiRotation* that takes a matrix A, as well as integers k and l used to navigate the matrix, and an empty array for eigenvectors we call R as arguments. Then, it performs a single rotation for this matrix. This rotation is performed by following the recipe presented in the lecture notes 07-08 from 2022, that can be accessed from the course's main page. When called, this function will give us this rotated matrix containing eigenvectors for the instance. Then, we implement the function jacobieigensolver that uses both the function *JacobiRotation* and maxoffdiagsymmetric, and performs Jacobi rotations until our reslut is accurate, which is determined by setting the max-value for the off-diagonal small. The diagonal of the matrix will then contain the eigenvalues, which we extract. To check that the code works as intended, we write another funtion *checresults*, that takes the matrix A, as well as eigenvalues and eigenvectors computed by using jacobi's rotation algorithm as arguments. This function calculates the eigenvalues and eigenvectors itself by using Armadillo, and then compares the results with the ones we submitted as arguments. We call this function for the matrix C with a=2, d=5, N=6, eps=$1e-6$ (which is our tolerance), and an amout of max iterations set to 1000. In the terminal, we then get printed:

```
Problem 4
Eigenvalues and eigenvectors match between the two methods.
```

By running this code we see that the two results are in agreeace.

5 Problem 5

We implement another function iterTime in a .cpp file *prob5.cpp*. This function takes a vector containing different values of N we wish to test our Jacobi's rotation algorithm for as argument, as well as the name of the file we wish save our data in, an integer maxiterations, double-values a and d, as well as a bool-argument called notdense. The function checks if we wish to examine the scaling behaviour of a dense matrix or not. If we're not interested in a dense matrix, the function calls on *createA* to create our matrix for a and d, and if not it creates a random dense matrix of size N. Then, it opens a text-file with a name determined by the argument we gave, and writes each N-value, as well as the amount of iterations needed to get an adeque approximation to both eigenvalues and eigenvectors calculated by jacobieigensolver, as well as the time it took for the function to run into the file.

If A was a dense matrix we would expect the Jacobi algorithm to be slower compared to our tridiagonal matrix. The same would go for a larger matrix. One could further also expect that for a larger matrix, meaning a larger N, the more time it would take the function would take to run.

We visualize the difference efficiency due to size and denzity of the matrix A, by calling the function iterTime twice for the same vector of N-values - once for tridiagonal matrixes and once for dense matrixes. The data is stored by the program in two different .txt-files prob5notdense.txt and prob5dense.txt. We decide to call the functions for a max amount of iterations set to 110000 (the amount was adapted to contain the whole range of data we get), an a=2, d=5, and N-values [6, 10, 20, 30, 50, 80, 120, 150, 180, 200, 220]. We present the data from the files in tables:

In the terminal:

```
Problem 5
Data recorded in prob5notdense.txt
Data recorded in prob5dense.txt
```

We can from these data-sets clearly see that it takes longer time for the function to run when dealing with a denser matrix in general. When the size of the matrix grows, these differences become more notable. The reason for this efficiency-difference is fact that there are more iterations of jacobi-rotations needed to get an accurate enough answer.

Table 2: prob5notdense.txt

Matrix Size(N)	Iterations	Execution Time[ms]
6	31	0
10	133	0
20	546	0
30	1226	1
50	3390	8
80	8465	47
120	18848	96
150	29189	221
180	41579	447
200	51218	658
220	61274	947

Table 3: prob5dense.txt

Matrix Size(N)	Iterations	Execution Time[ms]
6	41	0
10	135	0
20	616	0
30	1396	0
50	3993	4
80	10443	25
120	23823	124
150	37425	289
180	53976	582
200	66861	885
220	80663	1222

6 Problem 6

In another file *prob*6.*cpp*, we implement a function disA that takes n as an argument, and generates the matrix with values varying from 0 to 1, with n steps inbetween. Thereafter we implement another function exportdata that takes two strings, that we wish to call file-names, as well as the matrix A and other relevant arguments. It uses jacobieigensolver to calculate eigenvectors and eigenvalues for the matrix it takes as argument, and saves it in two seperate files.

We call this function twice for discretization of $\vec{x}$ by n=10 and n=100. The information is stored in text files eigenvaluesn10.txt, eigenvectorsn10.txt, eigenvaluesn100.txt, eigenvectorsn100.txt all containing the data insinuated in the filenames. Thereafter, since plotting is needed, we import these text files which we create into a python-file *prob*6.*py*. This file simply uses the module *matplotlib.pyplot* to plot the three eigenvectors corresponding to the three lowest eigenvalues, as well as the boundary points, like the assignment asked us to.

The code saves the figures as $Problem6plot1.jpg$ and $Problem6plot2.jpg$ on our machine. In the terminal:

```
Problem 6
Eigenvalues and eigenvectors saved to file eigenvaluesn10.txt and eigenvectorsn10.txt
Eigenvalues and eigenvectors saved to file eigenvaluesn100.txt and eigenvec-
torsn100.txt
```

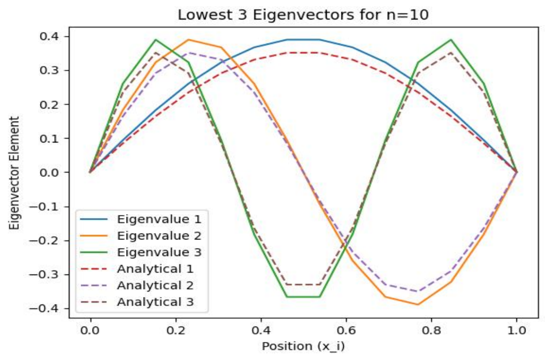

Figure 1: This figure shows the plot of the three lowest eigenvalues for n=10, as well as the boundary points

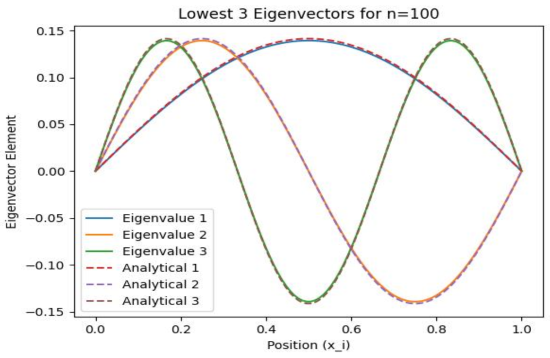

Figure 2: This figure shows the plot of the three lowest eigenvalues for n=100, as well as the boundary points

We notice that the functions are more accurate the for the higher n-value, both in the shape of the curves and the position of the boundary points. The accuracy of the shapes compared to the analytical solution is also good - also here the accuracy increases with a higher amount of steps n.