Project 2

Christophe Kristian Blomsen
c.k.blomsen@astro.uio.no

Eloi Martaillé Richard
e.m.richard@astro.uio.no

Vebjørn Øvereng
vebjoro@uio.no

Vetle Henrik Hvoslef
vetlehh@uio.no

September 27, 2022

Contents

Github page: https://github.com/christopheblomsen/project2

Problem 1

Will start from the definitions

$$\gamma \frac{\mathrm{d}u^2}{\mathrm{d}x^2} = -Fu \tag{1}$$

$$\frac{\mathrm{d}u^2}{\mathrm{d}x^2} = -\frac{F}{L}u \tag{2}$$

Now using

$$\hat{x} = \frac{x}{L} \tag{3}$$

$$\Downarrow$$

$$\frac{\mathrm{d}}{\mathrm{d}x} = \frac{\mathrm{d}}{\mathrm{d}\hat{x}}\frac{\mathrm{d}\hat{x}}{\mathrm{d}x} \tag{4}$$

$$\frac{\mathrm{d}}{\mathrm{d}x} = \frac{\mathrm{d}}{\mathrm{d}\hat{x}}\frac{1}{L} \tag{5}$$

$$\Downarrow$$

$$\frac{\mathrm{d}^2}{\mathrm{d}x^2} = \frac{\mathrm{d}^2}{\mathrm{d}\hat{x}^2}\frac{1}{L^2} \tag{6}$$

Putting (6) into (2)

$$\frac{1}{L^2}\frac{\mathrm{d}^2u}{\mathrm{d}\hat{x}^2} = -\frac{F}{\gamma}u \tag{7}$$

$$\frac{\mathrm{d}^2u}{\mathrm{d}\hat{x}^2} = -\frac{FL^2}{\gamma}u \tag{8}$$

$$\frac{\mathrm{d}^2u}{\mathrm{d}\hat{x}^2} = -\lambda u \tag{9}$$

We use the fact that since we both scale $u_x(x)$ and x so that they become $u_{\hat{x}}(\hat{x})$ and $\hat{x}$, we can say that $u_x(x) = u_{\hat{x}}(\hat{x})$. Because we have both scaled the function and its interval so $u_x(x)$ on the interval $x \in [0, L]$ and $u_{\hat{x}}(\hat{x})$ on $\hat{x} \in [0, 1]$ should look the same. Using this we have that

$$\frac{\mathrm{d}^2u}{\mathrm{d}\hat{x}^2} = -\lambda u \tag{10}$$

$$\frac{\mathrm{d}^2u_{\hat{x}}(\hat{x})}{\mathrm{d}\hat{x}^2} = -\lambda u_{\hat{x}}(\hat{x}) \tag{11}$$

Problem 2

In order to solve our system, we have to set up a tridiagonal matrix A so we have implemented a function in our `src/utils.cpp` called `create_tridiagonal`. This function takes 3 double numbers a, d, e representing the number on each line. We first initialize our matrix with setting it up as

our identity matrix multiplied by our diagonal number d. Next we iterate over every row until the last one, where for each row i we assigned the two values $a_{i,i+1} = e$ and $a_{i+1,i} = a$ creating our tridiagonal matrix $\boldsymbol{A}$.

To test if our matrix is correctly set up and can correspond to our system, we have calculated the analytical eigenvalues and eigenvectors given in the Project description and we used the armadillo `arma::eig_sym` on our matrix to compare with the analytical solution. We did obtain the same results with both methods showing that our matrix was successfully set up. We can find the test while running the `main.cpp` file.

Problem 3

Problem 3.a)

One main point in the Jacobi's rotation method is to find the maximum off-diagonal element of our matrix. So we created a `max_offdiag_symmetric` to find our maximum off-diagonal element assuming a symmetric matrix allowing us to only iterate over the upper part of our matrix considerably reducing the computational time.

We start off by defining the row size our matrix and initializing our maximum value b to be 0 so we are sure to replace in our first iteration since we consider the absolute value. Then we create a double for loop, the first one iterates on every row and the other will iterate on every upper element of that row, i.e, for a row i we will iterate over every $j > i$ and compare that element to our current maximum absolute value.

Problem 3.b)

We have implemented the function in `src/utils.cpp` and to do the test with the required matrix we can run the `test_3.cpp` file.

Problem 4

Problem 4.a)

A single Jacobi rotation is performed by `jacobi_rotate()`. First we calculate τ as

$$\tau = \frac{a_{\mathrm{ll}}^{\mathrm{m}} - a_{\mathrm{kk}}^{\mathrm{m}}}{2a_{\mathrm{kl}}^{\mathrm{m}}}, \tag{12}$$

then we find $\tan\theta$, $\sin\theta$ and $\cos\theta$. We choose the smallest of the two possible angles of rotation, that is

$$\tau = \begin{cases} \tan\theta = \frac{1}{\tau + sqrt1+\tau^2}, & \tau \geq 0 \\ \tan\theta = \frac{-1}{-\tau + \sqrt{1+\tau^2}}, & \tau < 0. \end{cases} \tag{13}$$

$\sin\theta$ and $\cos\theta$ are calculated found by

$$\cos\theta = \frac{1}{\sqrt{1+\tan^2\theta}} \tag{14}$$
$$\sin\theta = \cos\theta\tan\theta. \tag{15}$$
$$\tag{16}$$

We can now perform the rotation on the $\mathbf{A}$-matrix, $\mathbf{A}^{\mathrm{m}} \rightarrow \mathbf{A}^{\mathrm{m+1}}$, first we update the four elements a_{kk}, a_{ll}, a_{kl} and a_{lk}

$$a_{\mathrm{kk}}^{\mathrm{m+1}} = a_{\mathrm{kk}}^{m}\cos^2\theta - 2a_{\mathrm{kl}}^{m}\cos\theta\sin\theta + a_{\mathrm{ll}}^{m}\sin^2\theta \tag{17}$$
$$a_{\mathrm{ll}}^{m+1} = a_{\mathrm{ll}}^{m}\cos^2\theta + 2a_{\mathrm{kl}}^{m}\cos\theta\sin\theta + a_{\mathrm{kk}}^{m}\sin\theta \tag{18}$$
$$a_{\mathrm{kl}}^{m+1} = a_{\mathrm{lk}}^{m+1} = 0, \tag{19}$$
$$\tag{20}$$

and then the rows and columns given by k, l. For all $i \neq k, l$

$$a_{\mathrm{ik}}^{m+1} = a_{\mathrm{ik}}^{m}\cos\theta - a_{\mathrm{il}}^{m} \tag{21}$$
$$a_{\mathrm{ki}}^{m+1} = a_{\mathrm{ik}}^{m+1} \tag{22}$$
$$a_{\mathrm{il}}^{m+1} = a_{\mathrm{il}}^{m}\cos\theta + a_{\mathrm{ik}}^{m}\sin\theta \tag{23}$$
$$a_{\mathrm{li}}^{m+1} = a_{\mathrm{il}}^{m+1}. \tag{24}$$

We make sure to store a_{kk}^{m} and a_{ik}^{m} not to unintentionally use the next step $m+1$ in the updates. Now we update our rotation matrix $\mathbf{R}^{\mathrm{m}} \rightarrow \mathbf{R}^{\mathrm{m+1}} = \mathbf{R}^{\mathrm{m}}\mathbf{S}_{\mathrm{m}}$. For all i

$$r_{\mathrm{ik}}^{m+1} = r_{\mathrm{ik}}^{m}\cos\theta - r_{\mathrm{il}}^{m}\sin\theta \tag{25}$$
$$r_{\mathrm{il}}^{m+1} = r_{\mathrm{il}}^{m}\cos\theta - r_{\mathrm{ik}}^{m}\sin\theta \tag{26}$$
$$\tag{27}$$

For å implementere heile løysingsalgoritmen lager me funksjonen `jacobi_eigensolver()`, denne kallar på `max_offdiag_symmetric()` for å finne indeksane `k` og `l` og utførar Jacobi-rotasjonar iterativt for å iterere vekk desse elementa fram til det største elementet er mindre enn ein toleranse.

Problem 4.b)

We implement the analytical solution in the functions `analytical_eigenvalues()` and `analytical_eigenvectors()`. We solve for the eigenvectors of the 6×6-matrix given in the project description.

In Figure 1 we can see two plots comparing the first three eigenvectors. By eye they seem similar. If we divide the analytical solution by the numerical solution element wise we get a constant vector of ones. This is conclusive that the solutions are indeed identical, our implementation of Jacobi's rotation method is up and running.

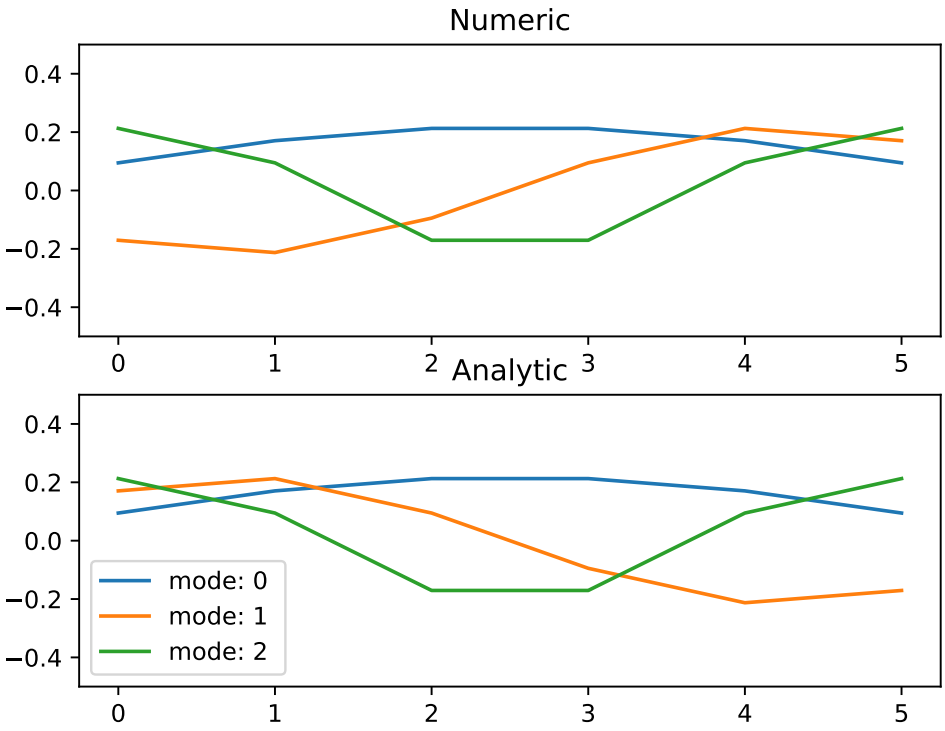

Figure 1: The first three eigenvectors of our 6x6 test matrix. With the exception that mode 2 is flipped they look similar.

Problem 5

Problem 5.a)

This is achieved with a for loop running over N from 10 to 100 with the iteration being multiplied by a factor 10 for each of them See in Figure 2 that we need roughly $N^{2.13}$ iterations for the algorithm to converge, and that this number is fairly constant over the range we tested.

This can also be seen in the table Reason why $N > 70$ is not included in both Figure 2 and Table 1 is that it did not converge.

Problem 5.b)

We will except the same behavior in the algorithm with a dense

matrix as with a matrix with lots zeros. In the `max_offdiag_symmetric()` we check all the values of the upper triangle regardless of zero or not, so that would stay the same. That is also because we assume that the dense matrix is still an symmetric matrix, so we still only need to care about, in our case, the upper triangle.

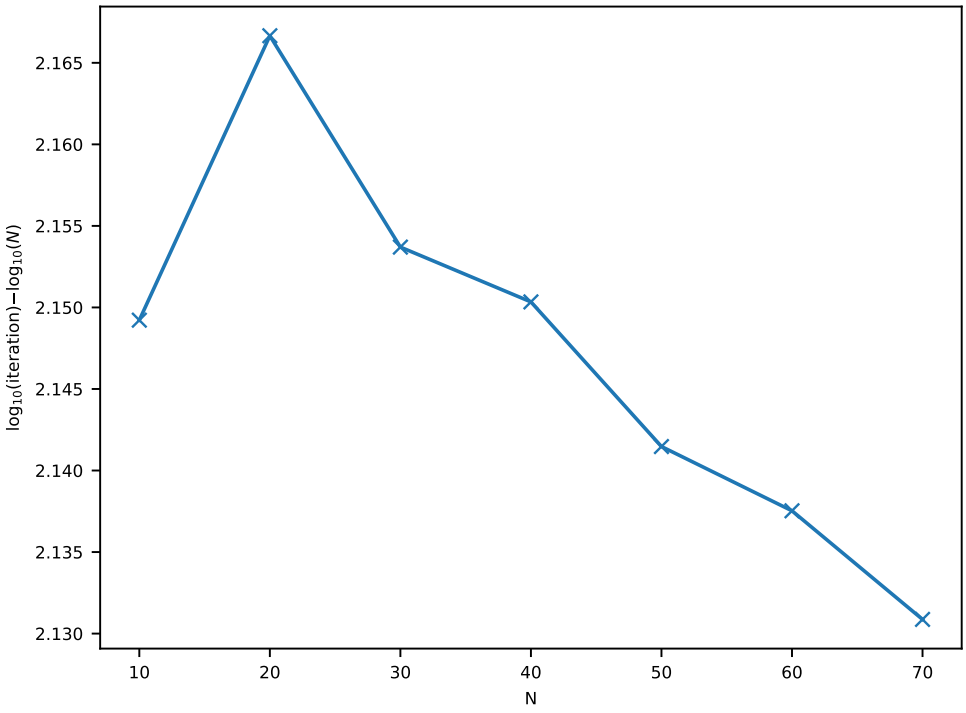

Figure 2: A logarithmic y axis showing the level of N needed to get to convergence with the Jacobi method

The reason for the Jacobi rotation algorithm to approximately take the same time is because of the fact that when we rotate one max value to be zero other values that previously were zero may then become not zero. Hjorth-Jensen

Problem 6

Problem 6.a)

The figure can be seen in Figure 3

Problem 6.b)

We have plotted in Figure 4 the numerical and the analytical solution for the first three mode with different step size. We can see that in both case the numerical behavior is similar to what the theory predict. For this plot we have normalised the eigenvectors for the analytical solution with the `arma::normalise` function introduce in the project description. We also had to multiply

Table 1: Table of the different N with corresponding iterations and the log difference.

N	Iterations	$\log_{10}$(Iter)-$\log_{10}$(N)
10	141	2.14922
20	659	2.16666
30	1518	2.1537
40	2786	2.15034
50	4348	2.14147
60	6322	2.13753
70	8544	2.13087

the second and third mode of the analytical solution by -1. This is not a problem because as we know if v is a solution to our eigen system then $c * v$, with c a constant, is also a solution.

References

Hjorth-Jensen, Morten. *Computational Physics*. 2015th ed. Department of Physics, University of Oslo, 2015.

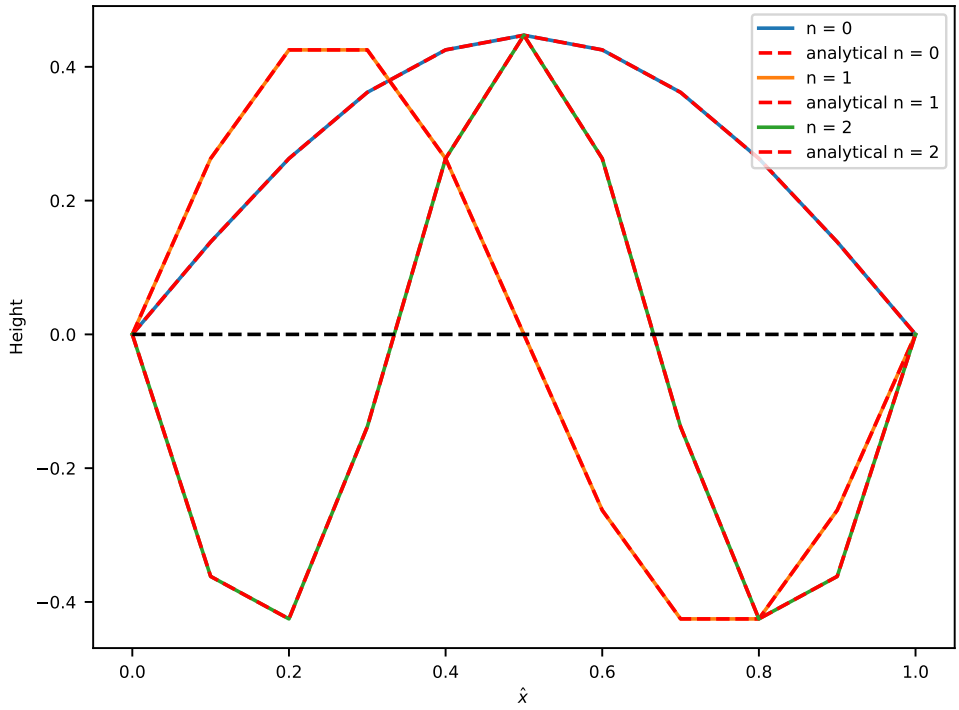

Figure 3: Numerical solution for the first three mode in plain line with the analytical solution for the same mode in red dashed line for N=10. The solution is presented with the dimensionless scale $\hat{x}$ is on the x-axis, and the amplitude is on the y-axis

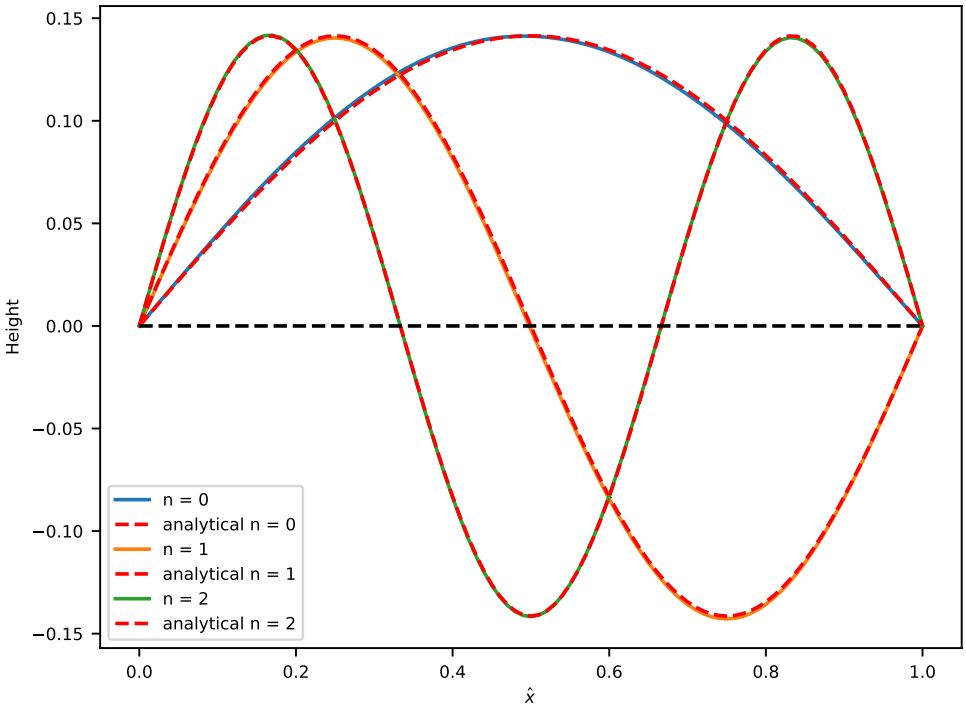

Figure 4: Numerical solution for the first three mode in plain line with the analytical solution for the same mode in red dashed line for N=100. The solution is presented with the dimensionless scale $\hat{x}$ is on the x-axis, and the amplitude is on the y-axis