Solving a buckling beam problem with the Jacobi rotation algorithm

Aline, Even Tobias and Vilde

(Dated: September 26, 2022)

In this project, we implemented the Jacobi rotation algorithm for solving eigenvalue problems for symmetric matrices. In particular, to solve a second-order differential equation representing a "buckling beam" problem. Our `C++` program uses the `Armadillo` library in order to represent the matrices. We used this program to simulate and visualize the "buckling" of a one-dimensional beam of length L with fixed endpoints after a horizontal force has been applied to it. We found that in our implementation of the algorithm, the number of iterations required scales with matrix size N by $\mathcal{O}(N^2)$.

GitHub: *https://github.com/ungvilde/FYS4150/tree/main/Project%202*

PROBLEM 1

We have the second-order differential equation given by

$$\gamma \frac{\mathrm{d}^2 u(x)}{\mathrm{d}x^2} = -F u(x), \tag{1}$$

and we want to show that it can be expressed as the scaled equation

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

where $\hat{x} \equiv \frac{x}{L}$ is a dimensionless variable. We have the differential operator

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

which implies that

$$\begin{aligned}\frac{\mathrm{d}^2}{\mathrm{d}x^2} &= \left(\frac{\mathrm{d}}{\mathrm{d}x}\right)^2 \\ &= \left(\frac{\mathrm{d}\hat{x}}{\mathrm{d}x}\frac{\mathrm{d}}{\mathrm{d}\hat{x}}\right)^2 \\ &= \left(\frac{1}{L}\frac{\mathrm{d}}{\mathrm{d}\hat{x}}\right)^2 \\ &= \frac{1}{L^2}\frac{\mathrm{d}^2}{\mathrm{d}\hat{x}^2}.\end{aligned}$$

Inserting this into (1) gives

$$\begin{aligned}\frac{\mathrm{d}^2 u(x)}{\mathrm{d}x^2} &= -\frac{F}{\gamma}u(x) \\ \frac{1}{L^2}\frac{\mathrm{d}^2 u(\hat{x})}{\mathrm{d}\hat{x}^2} &= -\frac{F}{\gamma}u(\hat{x}) \\ \frac{\mathrm{d}^2 u(\hat{x})}{\mathrm{d}\hat{x}^2} &= -\frac{FL^2}{\gamma}u(\hat{x}).\end{aligned}$$

Now let $\lambda = \frac{FL^2}{\gamma}$. Then we finally have the scaled equation given in (2).

PROBLEM 2

We let **A** be a N x N tridiagonal matrix, where N = 6 determines the size of the matrix. To solve the eigenvalue equation (3) we will use the function `arma::eig_sym` from the Armadillo library.

$$\mathbf{A}\vec{v} = \lambda\vec{v} \tag{3}$$

To make sure that we check both signs for the eigenvectors we use the armadillo function `arma::normalise`. Then, to compare the armadillo solution to the analytical solution we use the function `arma::approx_equal`, which compares the two with a tolerance of $\epsilon = 10^{-10}$.

PROBLEM 3

a) In order to identify the largest off-diagonal element, we created a function `max_offdiag_symmetric` that takes in a reference to an `Armadillo` matrix as an argument and returns the maximum absolute element of the matrix, as well as its indices k and ℓ.

This function loops over each off-diagonal element in the matrix, assuming a symmetric matrix and thus only checking for indices $k < \ell$. The details of this implementation can be found in Algorithm 1. Still, this method of finding the maximum element uses significant computational power once the matrix is big. That is because each element will be iterated over regardless of whether it is 0 or not. See Problem 5b for further discussion on the computational cost of this implementation.

Algorithm 1 Function for finding the largest absolute off-diagonal element of a symmetric matrix.

```
function MAX_OFFDIAG_SYMMETRIC($\mathbf{A}$)
    max ← 0
    for i = 0, 1, ..., $N$ do
        for j = i+1, ..., $N$ do
            if $|\mathbf{A}(i,j)| >$ max then
                $\ell \leftarrow i$
                $k \leftarrow j$
                max ← $|\mathbf{A}(i,j)|$
            end if
        end for
    end for
    return max, $k, \ell$
end function
```

b) To verify our `max_offdiag_symmetric` function, we test it by sending in the matrix:

$$A = \begin{bmatrix} 1 & 0 & 0 & 0.5 \\ 0 & 1 & -0.7 & 0 \\ 0 & -0.7 & 1 & 0 \\ 0.5 & 0 & 0 & 1 \end{bmatrix} \tag{4}$$

where we expect our function to return 0.7, as this is the absolute value of the largest of the off-diagonal values. When we run our test, we get that the maximum off-diagonal matrix element is 0.7, and that the indices are $k = 1$ and $l = 2$, where `k` is the column index and l is the row. Thus we see that our `max_offdiag_symmetric` function produces valid results.

PROBLEM 4

a) To solve the eigenvalue problem presented by (3), we implement the Jacobi rotation algorithm. The Jacobi rotation algorithm is an iterative method, where we want to diagonalize **A** using similarity transformations. That is, we want to find a matrix **S** such that $\mathbf{SAS}^T = \mathbf{D}$. We will do this by performing a series of transformations making **A** approximately diagonal.

Let $\mathbf{A}^{(1)} = \mathbf{A}$ denote our original matrix. We then find a rotation matrix $\mathbf{S}_1$ which makes the largest off-diagonal element of $\mathbf{A}^{(1)}$ rotated to zero. Then,

$$\mathbf{A}^{(2)} = \mathbf{S}_1 \mathbf{A}^{(1)} \mathbf{S}_1^T.$$

We do this iteratively, such that at iteration $m+1$ we have

$$\mathbf{A}^{(m+1)} = \mathbf{S}_m \mathbf{A}^{(m)} \mathbf{S}_m^T,$$

until finally $\mathbf{A}^M \approx \mathbf{D}$. The rotation matrix $\mathbf{S}_m$ is completely defined by the parameters (k, ℓ, θ), where k and ℓ are the indices of the largest off-diagonal element of $\mathbf{A}^{(m)}$, and θ specifies the rotation.

Before presenting our algorithm, we clarify some notation. In our algorithm, we denote rotation $m+1$ by

$$\mathbf{R}^{(m+1)} = \mathbf{R}^{(m)} \mathbf{S}_m = \mathbf{S}_1 \mathbf{S}_2 \cdots \mathbf{S}_{m-1} \mathbf{S}_m$$

to keep track of the total rotation. We denote an element of the updated matrix $\mathbf{A}^{(m)}$ as $a_{i,j}^{(m)}$, and analogically for $\mathbf{R}^{(m)}$. Finally, to simplify our notation, we write $\sin\theta = s$, $\cos\theta = c$ and $\tan\theta = t$. A full overview our implementation of the Jacobi algorithm can be found in Algorithm 2. After iterating M times – that is, until the largest off-diagonal element of $\mathbf{A}^M$ is smaller than the chosen tolerance value ϵ – then the eigenvalues of $\mathbf{A}$ can be found in the diagonal of $\mathbf{A}^M$, and the eigenvectors can be found in our rotation matrix $\mathbf{R}^M$. We have also included a break statement in our algorithm, such that if the number of iterations m exceeds the maximum number of allowed iterations, then the while-loop terminates. We also normalise and sort the resulting eigenvectors by their corresponding eigenvalues in increasing order.

Algorithm 2 Jacobi rotation algorithm

```
$\epsilon \leftarrow 10^{-10}$                                   ▷ Choose tolerance
$m \leftarrow 1$
$\mathbf{A}^{(m)} \leftarrow \mathbf{A}$
$\mathbf{R}^{(m)} \leftarrow \mathbf{I}$
max, $k, \ell \leftarrow$ MAX_OFFDIAG_SYMMETRIC($\mathbf{A}^{(m)}$)     ▷ See Algorithm 1 for details.
while max $> \epsilon$ do
    $\tau \leftarrow \frac{a_{\ell,\ell}^{(m)} - a_{k,k}^{(m)}}{2a_{\ell,k}^{(m)}}$
    if $\tau > 0$ then
        $t \leftarrow \frac{1}{\tau + \sqrt{1+t^2}}$
    else
        $t \leftarrow \frac{-1}{-\tau + \sqrt{1+t^2}}$
    end if
    $c \leftarrow \frac{1}{\sqrt{1+t^2}}$
    $s \leftarrow ct$                ▷ Now we know our rotation matrix $\mathbf{S}_m$, since we have $k, \ell$ and $\theta$.
    $a_{k,k}^{(m+1)} \leftarrow a_{k,k}^{(m)} c^2 - 2a_{k,\ell}^{(m)} cs + a_{\ell,\ell}^{(m)} s^2$      ▷ Here we update current $\mathbf{A}^{(m)}$
    $a_{\ell,\ell}^{(m+1)} \leftarrow a_{\ell,\ell}^{(m)} c^2 + 2a_{k,\ell}^{(m)} cs + a_{k,k}^{(m)} s^2$
    $a_{k,\ell}^{(m+1)} \leftarrow 0$
    $a_{\ell,k}^{(m+1)} \leftarrow 0$
    for $i \neq k, \ell$ do
        $a_{i,k}^{(m+1)} \leftarrow a_{i,k}^{(m)} c - a_{i,\ell}^{(m)} s$
        $a_{k,i}^{(m+1)} \leftarrow a_{i,k}^{(m+1)}$
        $a_{i,\ell}^{(m+1)} \leftarrow a_{i,\ell}^{(m)} c + a_{i,k}^{(m)} s$
        $a_{\ell,i}^{(m+1)} \leftarrow a_{i,\ell}^{(m+1)}$
    end for
    for $i = 1, \ldots, N$ do
        $r_{i,k}^{(m+1)} \leftarrow r_{i,k}^{(m)} c - r_{i,\ell}^{(m)} s$
        $r_{i,\ell}^{(m+1)} \leftarrow r_{i,\ell}^{(m)} c + r_{i,k}^{(m)} s$
    end for
    max, $k, \ell \leftarrow$ MAX_OFFDIAG_SYMMETRIC($\mathbf{A}^{(m+1)}$)   ▷ Update max. off-diagonal element and indices $k$ and $\ell$.
    $m \leftarrow m + 1$                                  ▷ Update iteration number
end while
```

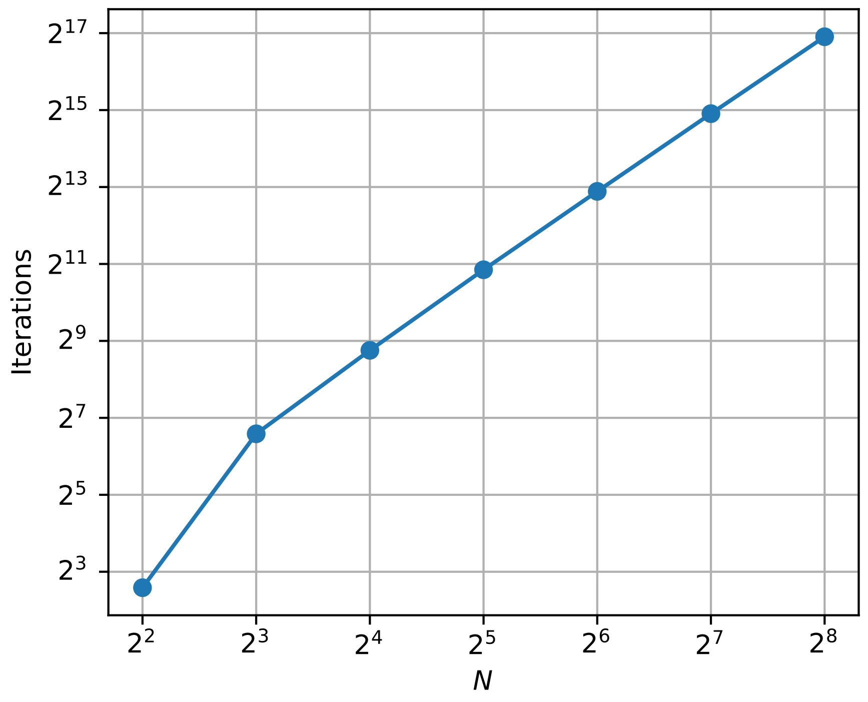

Figure 1. The plot shows the number of iterations required as a function of the matrix size N. This is a log-log plot on a $\log_2$-scale. We note that the number of iterations seems to follow a second-order relationship with N. That is, the number of iterations required is in the order $\mathcal{O}(N^2)$.

b) We can verify that our implementation of the Jacobi rotation algorithm is correct, by comparing the resulting eigenvalues and eigenvectors with the analytical solution for a 6×6 matrix $\mathbf{A}$. We use `approx_equal` from the `Armadillo` library to do a floating number comparison with a tolerance of $\epsilon = 10^{-10}$, making sure to also check if the eigenvectors and eigenvalues are the same in the case where the vector is scaled by -1.

PROBLEM 5

a) Looking at Fig. 1, we see that for matrices of sizes ranging from $N = 2^3$ to $N = 2^8$, the iterations increase log-linearly by 2. In other words, the number of iterations required seem to scale with N by order $\mathcal{O}(N^2)$.

b) If we use a dense matrix as opposed to our tridiagonal one, we would expect the number of iterations required to increase, as there are many more non-zero elements. However, when we check the number of iterations required for dense, symmetric $N \times N$ matrices ranging from $N = 2^2$ to $N = 2^8$, we find that there is little difference in the number of iterations required compared to what we found with tridiagonal matrices. This is due to our `max_offdiag_symmetric` method. This function loops over every upper triangle matrix element in order to find the largest off-diagonal element, and is thus the main bottleneck of our algorithm. This loop costs us the most computational power, and is the same regardless of whether we have a dense or a tridiagonal matrix.

PROBLEM 6

a) We used a discretization of $\hat{x}$ with $n = 10$ steps and solved equation (3) using our Jacobi rotation algorithm. The eigenvectors corresponding to the three lowest eigenvalues are shown in Fig. 2. We ended up scaling our numerical eigenvector solution corresponding to λ_3 with -1, in order to make it match with the analytical solution when plotting. This, however, is an equally valid solution.

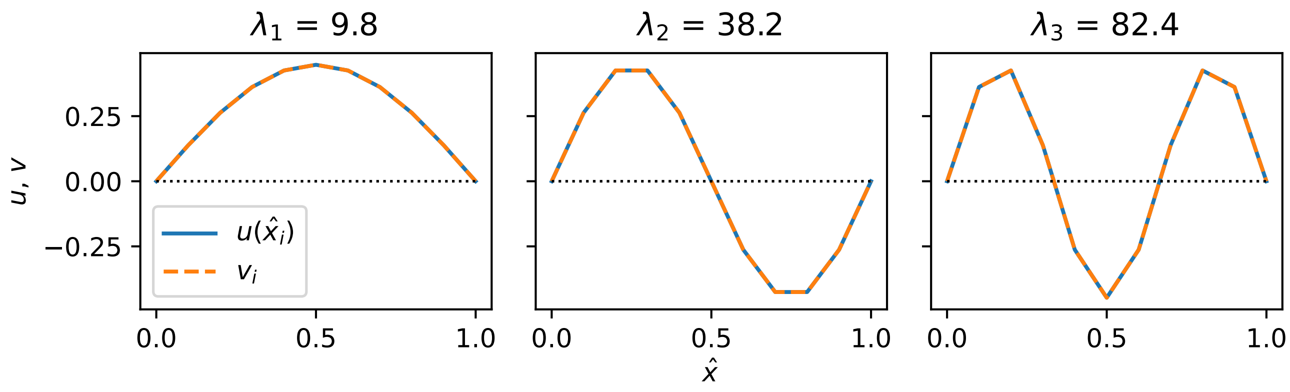

Figure 2. The figure shows the numerical and analytical eigenvectors plotted as a function of $\hat{x}$, for a discretization of $n = 10$ steps. We have plotted the eigenvectors corresponding to the three lowest eigenvalues λ_i for $i = 1, 2, 3$. The analytical solution is denoted by $u(\hat{x}_i)$, and the numerical solution is denoted by v_i. The black dotted line shows the original position of the beam.

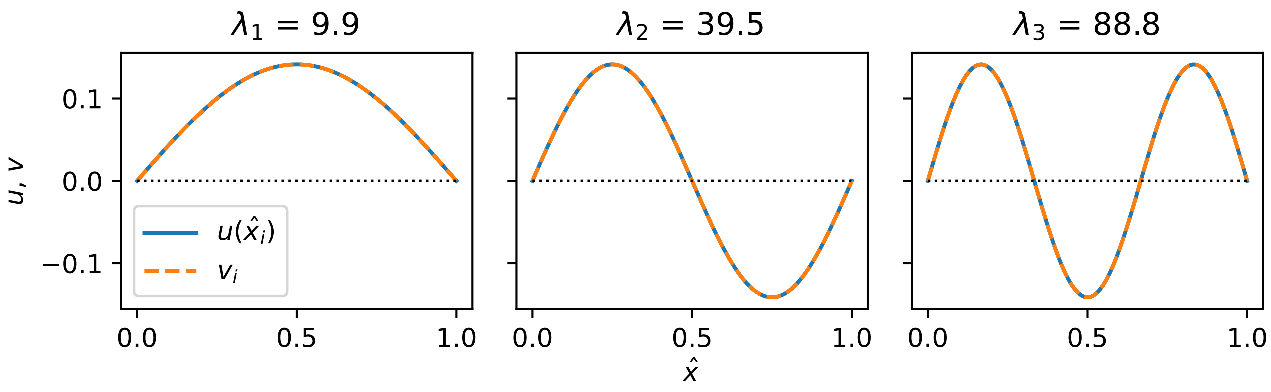

Figure 3. The figure shows the eigenvectors as a function of $\hat{x}$ for $n = 100$ steps. We have included both the numerical and the analytical solutions. The analytical solution is denoted by $u(\hat{x})$ and the numerical solution is denoted by v_i. The plot shows the eigenvectors corresponding to the three lowest eigenvalues λ_i for $i = 1, 2, 3$. The black dotted line shows the original position of the beam.

We can look at the vertical displacement of the buckling beam as a standing wave on a string. With this logic, since the eigenvalues correspond to the energy, we see that for lower eigenvalues, the resulting standing wave has a lower frequency.

b) Similarly we can plot the eigenvectors of the problem solved with $n = 100$ steps, which yields Fig. 3. Again, we scaled the numerical solution for λ_3 with -1 in order to make the plotted lines overlap. We see that the eigenvectors have the same general shape as for $n = 10$, but that the eigenvalues are consistently larger. We also note that the solutions have different range along the y-axis.