Project 2

Contents

Practicalities

- **Deadline:** Wednesday, September 25, 23:59.
- **Format:**
  - A pdf document, typeset in LaTeX, with answers to all the problems below. You deliver the pdf on Canvas.
  - Use the template we have provided here.
  - Code (with comments, of course) on a UiO GitHub repo (github.uio.no), with the URL to you repo written in the pdf document.
  - You *must* deliver via your group on Canvas (even if you are working alone).
- **Not a complete report:** For project 2, we do not require you to write a proper scientific report — only a document with an answer for each problem. But the quality of the presentation still matters, of course. So pay attention to figures, figure captions, grammar, etc.
- **Collaboration:** We *strongly* encourage you to collaborate with others, in groups up to three students. The group hands in a single pdf. (So make sure you all join the same group on Canvas.) Remember to list everyone's name in the pdf.
- **Reproducibility:** Your code should be available on a GitHub repo. You can refer to relevant parts of your code in your answers. Make sure to include a README file in the repo that briefly explains how the code is organized, and how it should be compiled and run in order to reproduce your results.
- **Figures:** Figures included in your LaTeX document should be made as vector graphics (e.g. `.pdf` files), rather than raster graphics (e.g. `.png` files). If you are making plots with `matplotlib.pyplot` in Python, this is as simple as calling `plt.savefig("figure.pdf")` rather than `plt.savefig("figure.png")`.
- **We recommend using Armadillo:** For this project we recommend using Armadillo to work with matrices and vectors.

Introduction

The main topics of this project are

- scaling of equations
- eigenvalue problems
- code testing (*unit testing*).

Our example of an eigenvalue problem is a special case of a one-dimensional buckling beam. Consider the following situation:

- A horizontal beam.

- L is the horizontal length between the two beam endpoints. (So before any buckling, L is also the length of the beam.)
- We let $u(x)$ be the vertical displacement of the beam at horizontal position x, with $x \in [0, L]$.
- A force F is applied at the endpoint ($x = L$), *directed into the beam*, i.e. towards $x = 0$.
- The beam is fastened with *pin endpoints*, meaning that $u(0) = 0$ and $u(L) = 0$, but the endpoints are allowed to rotate ($u'(x) \neq 0$).
- We are *not* studying a time-dependent system here – we are simply interested in the different static shapes the beam can take due to the applied force.

We can then describe this situation with the second-order differential equation

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

where γ is some constant defined by material properties like rigidity. However, the equation we will actually be working with is the scaled equation

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

where $\hat{x} \equiv x/L$ is a dimensionless (unitless) varible, $\hat{x} \in [0, 1]$, and $\lambda = \frac{FL^2}{\gamma}$. (See problem 1.)

> **Note**
>
> Here we have been a bit sloppy with our notation. Technically, due to our change of variable, the functions $u(x)$ and $u(\hat{x})$ are two *different* functions, so we should have used e.g. notation like $u_x(x)$ for the original function and $u_{\hat{x}}(\hat{x})$ for the function after the variable change. The key thing is that the functions are related as $u_{\hat{x}}(\hat{x}(x)) = u_x(x)$.

Discretization: We discretize this by dividing our $\hat{x}$ range into n parts, i.e. we will have $n + 1$ points $\hat{x}_0$, $\hat{x}_1$, …, $\hat{x}_{n-1}$, $\hat{x}_n$. Thus we have a stepsize

$$h = \frac{\hat{x}_{\text{max}} - \hat{x}_{\text{min}}}{n} = \frac{1 - 0}{n} = \frac{1}{n}$$

and our $\hat{x}_i$ points are given by

$$\hat{x}_i = \hat{x}_0 + ih, \quad i = 0, 1, \ldots, n.$$

From this dicretization we get the following set of equations for v_i (our approximations to the exact u_i):

$$\frac{-v_{i+1} + 2v_i - v_{i-1}}{h^2} = \lambda v_i.$$

By inserting the boundary conditions $v_0 = 0$ and $v_n = 0$ we can write this as the linear algebra eigenvalue problem

$$\mathbf{A}\vec{v} = \lambda\vec{v} \tag{6}$$

where

- the elements of the vector $\vec{v}$ are $\vec{v} = [v_1, \ldots, v_{n-1}]$; and
- $\mathbf{A}$ is tridiag(a, d, a), with
  - $a = -1/h^2$
  - $d = 2/h^2$
- *Note:* In contrast to project 1, we here keep the factor $1/h^2$ as part of the definition of

$\mathbf{A}$.

To solve this matrix equation means finding the pairs of eigenvalues and eigenvectors, ($\lambda^{(j)}, \vec{v}^{(j)}$), that satify Eq. (6). These eigenvectors are then our discretized approximations to the true *eigenfunctions* $u^{(j)}(\hat{x})$ that are the solutions of the differential equation in Eq. (5).

Notation: We will use N to denote the size of the matrix, so that $\mathbf{A}$ is an $N \times N$ matrix, and $\vec{v}$ a column vector of length N. Make sure not to confuse N with the number of *steps* in our discretization of $\hat{x}$ (n steps), or the number of $\hat{x}_i$ points ($n+1$ points). They are related as $N = n - 1$.

Analytical solutions: For a given matrix size $N \times N$, the eigenvalue problem in Eq. (6) with the tridiagonal matrix $\mathbf{A} = \mathrm{tridiag}(a, d, a)$ has the following set of analytical eigenvalues and eigenvectors:

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

$$\vec{v}^{(j)} = \left[\sin\left(\frac{j\pi}{N+1}\right), \sin\left(\frac{2j\pi}{N+1}\right), \ldots, \sin\left(\frac{Nj\pi}{N+1}\right)\right]^T, \quad j = 1, \ldots, N$$

Scaling of eigenvectors: Remember that if $\vec{v}$ is an eigenvector of $\mathbf{A}\vec{v} = \lambda\vec{v}$, then a scaled vector $c\vec{v}$, where c is some constant, is an equally good eigenvector. (Remember that c can be negative.) When presenting your results, and when comparing to results from Armadillo, it will be useful to scale each eigenvector to have unit norm (i.e. vector length 1). You can do this easily using the Armadillo function `arma::normalise` described here.

Problems

Problem 1

With the definition $\hat{x} \equiv x/L$, show that Eq. (4) can be written as Eq. (5).

> **Note**
>
> See the note after Eq. (5) about some sloppy notation. Note that it's perfectly fine to use the simple notation $u(\hat{x})$ throughout this project, but keep in mind that $u(\hat{x})$ and $u(x)$ are *different* functions.

Problem 2

Before we get started with implementing the Jacobi rotation algorithm, let's make sure that we can set up the tridiagonal matrix $\mathbf{A}$ correctly. So, write a short program that:

- sets up the tridiagonal $\mathbf{A}$ for $N = 6$;
- solves $\mathbf{A}\vec{v} = \lambda\vec{v}$ using Armadillo's `arma::eig_sym`, described here;
- checks that the eigenvalues and eigenvectors from Armadillo agrees with the analytical result for $N = 6$. (Remember scaling of eigenvectors, as discussed above.)

Problem 3

An important part of the Jacobi algorithm is to have a function that can identify the largest (in absolute value) off-diagonal element of a matrix. Here we specialize to the case of a

symmetric matrix.

a) Write a C++ function that can identify the largest off-diagonal element of a matrix. A suggestion is to write a function that

- has return type `double`
- takes an Armadillo matrix as input
- takes *references* to two integers as input
- identifies the largest off-diagonal element (in absolute value) in the matrix, under the assumption of a symmetric matrix
- writes the matrix indices for this element to the two integer references
- returns the value of this matrix element

A simple function signature could then be

```
double max_offdiag_symmetric(arma::mat A, int& k, int &l),
```

Or, to avoid copying the (potentially large) matrix A every time we run the function, we could simply pass in a reference to A. Since the function `max_offdiag_symmetric` won't be modifying the matrix in any way, we should make this explicit by using a `const` reference, like this:

```
double max_offdiag_symmetric(const arma::mat& A, int& k, int &l)
```

b) Write a small test code that tests the above function using the matrix

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

Problem 4

a) Write a code implementation of Jacobi's rotation algorithm for solving Eq. (6). See the code snippets at the end of this page for a suggested code structure.

b) Let $\mathbf{A}$ be of size 6×6. Test your code by checking that the eigenvalues and eigenvectors you get agree with the analytical results for $N = 6$.

Problem 5

Now let's look at how many similarity transformations we need before we reach a result where all non-diagonal matrix elements are close to zero.

a) By running your program with different choices of N, try to estimate how the number of required transformations scale with the matrix size N when you use your code to solve Eq. (6). Present your scaling data either in a table or a plot (or both).

b) What scaling behaviour would you expect to see if $\mathbf{A}$ was a dense matrix?

First hint: Think about the result you got in problem a). Why is it that the algorithm is so slow, even when starting with a matrix with so many zero elements?

Second hint: While you're not required to do so, there's of course nothing stopping you from just testing the case in b) with your Jacobi code! Here's a quick way to generate a N*N dense and symmetric matrix with random entries in Armadillo:

```
// Generate random N*N matrix
arma::mat A = arma::mat(N, N).randn();

// Symmetrize the matrix by reflecting the upper triangle to lower
triangle
A = arma::symmatu(A);
```

Problem 6

a) For a discretization of $\hat{x}$ with $n = 10$ steps, solve Eq. (6) using your Jacobi code and make a plot of the three eigenvectors corresponding to the three lowest eigenvalues. The plot should show vector elements v_i against the corresponding positions $\hat{x}_i$.

Since we are effectively showing the solutions to a differential equation, your plot should also include the boundary points $(\hat{x}_0, v_0)$ and $(\hat{x}_n, v_n)$.

Plot the corresponding analytical eigenvectors (extended with the boundary points) in the same plot.

b) Make the same plot for discretization of $\hat{x}$ with $n = 100$ steps.

Code snippets

A suggested code structure: There are many ways of designing the code for this project. The three function declarations (and descriptions) below give a hint of one possible approach:

```
// Determine the the max off-diagonal element of a symmetric matrix A
// - Saves the matrix element indicies to k and l
// - Returns absolute value of A(k,l) as the function return value
double max_offdiag_symmetric(const arma::mat& A, int& k, int& l);

// Performs a single Jacobi rotation, to "rotate away"
// the off-diagonal element at A(k,l).
// - Assumes symmetric matrix, so we only consider k < l
// - Modifies the input matrices A and R
void jacobi_rotate(arma::mat& A, arma::mat& R, int k, int l);

// Jacobi method eigensolver:
// - Runs jacobo_rotate until max off-diagonal element < eps
// - Writes the eigenvalues as entries in the vector "eigenvalues"
// - Writes the eigenvectors as columns in the matrix "eigenvectors"
//   (The returned eigenvalues and eigenvectors are sorted using
arma::sort_index)
// - Stops if it the number of iterations reaches "maxiter"
// - Writes the number of iterations to the integer "iterations"
// - Sets the bool reference "converged" to true if convergence was
reached before hitting maxiter
void jacobi_eigensolver(const arma::mat& A, double eps, arma::vec&
eigenvalues, arma::mat& eigenvectors,
                        const int maxiter, int& iterations, bool&
converged);
```

When such helper functions are in place and working, writing a main program for some specific task is typically not too much work. (Famous last words.)

Finding the max off-diagonal element: Here is one possible sketch for the function `max_offdiag_symmetric` discussed above:

```
// A function that finds the max off-diag element of a symmetric matrix A.
// - The matrix indices of the max element are returned by writing to the
//   int references k and l (row and column, respectively)
// - The value of the max element A(k,l) is returned as the function
//   return value
double max_offdiag_symmetric(const arma::mat& A, int& k, int& l)
{
  // Get size of the matrix A. Use e.g. A.n_rows, see the Armadillo
documentation

  // Possible consistency checks:
  // Check that A is square and larger than 1x1. Here you can for instance
use A.is_square(),
  // see the Armadillo documentation.
  //
  // The standard function 'assert' from <assert.h> can be useful for
quick checks like this
  // during the code development phase. Use it like this: assert(some
condition),
  // e.g assert(a==b). If the condition evaluates to false, the program is
killed with
  // an assertion error. More info: https://www.cplusplus.com/reference/
cassert/assert/

  // Initialize references k and l to the first off-diagonal element of A

  // Initialize a double variable 'maxval' to A(k,l). We'll use this
variable
  // to keep track of the largest off-diag element.

  // Loop through all elements in the upper triangle of A (not including
the diagonal)
  // When encountering a matrix element with larger absolute value than
the current value of maxval,
  // update k, l and max accordingly.

  // Return maxval
}
```

Note that this function has some inefficiencies: In an optimized program, you don't want this function to always check that A is square and larger than 1x1 – perhaps you don't want this function to check this at all, but rather just trust the input is is given from your main code. But during development, it is often useful to first write code that is fairly self-contained and "safe" (makes it easier to debug), and then optimize it afterwards.

Helper functions for creating tridiagonal matrices: A standard task that you will often need is to create a tridiagonal matrix. Why not write a small helper function for doing precisely that? Here's an outline of such a function:

```
// Create a tridiagonal matrix tridiag(a,d,e) of size n*n,
// from scalar input a, d, and e. That is, create a matrix where
// - all n-1 elements on the subdiagonal have value a
// - all n elements on the diagonal have value d
// - all n-1 elements on the superdiagonal have value e
arma::mat create_tridiagonal(int n, double a, double d, double e)
{
  // Start from identity matrix
  arma::mat A = arma::mat(n, n, arma::fill::eye);

  // Fill the first row (row index 0), e.g.
  A(0,0) = d;
  A(0,1) = e;

  // Loop that fills rows 2 to n-1 (row indices 1 to n-2)

  // Fill last row (row index n-1)

  return A;
}
```

Again, there are ways to make this more efficient, e.g. by writing the diagonal elements directly when we create the matrix, e.g.

```
// Start from identity matrix
arma::mat A = arma::mat(n, n, arma::fill::eye) * d;
```

But this function will anyway not be the computational bottleneck of your code, so you might as well keep the code more explicit if you find that easier to read/understand.

Once you have a function `create_tridiagonal`, you could easily add a function specialized to the symmetric case, that simply uses the more general function:

```
// Create a symmetric tridiagonal matrix tridiag(a,d,a) of size n*n
// from scalar input a and d.
arma::mat create_symmetric_tridiagonal(int n, double a, double d)
{
  // Call create_tridiagonal and return the result
  return create_tridiagonal(n, a, d, a);
}
```

The above functions assume that all elements on a given sub-, super- or main diagonal are identical. In a more general case you would need to pass in three vectors with the relevant matrix elements. Then you could have a set of three helper functions as sketched here:

```
// Create tridiagonal matrix from vectors.
// - lower diagonal: vector a, lenght n-1
// - main diagonal:  vector d, lenght n
// - upper diagonal: vector e, lenght n-1
arma::mat create_tridiagonal(const arma::vec& a, const arma::vec& d, const
arma::vec& e)
{
  // Start from identity matrix
  arma::mat A = arma::mat(n, n, fill::eye);

  // Fill first row (row index 0)

  // Loop that fills rows 2 to n-1 (row indices 1 to n-2)

  // Fill last row (row index n-1)

  return A;
}


// Create a tridiagonal matrix tridiag(a,d,e) of size n*n
// from scalar input a, d and e
arma::mat create_tridiagonal(int n, double a, double d, double e)
{
  arma::vec a_vec = arma::vec(n-1, arma::fill::ones) * a;
  arma::vec d_vec = arma::vec(n, arma::fill::ones) * d;
  arma::vec e_vec = arma::vec(n-1, arma::fill::ones) * e;

  // Call the vector version of this function and return the result
  return create_tridiagonal(a_vec, d_vec, e_vec);
}


// Create a symmetric tridiagonal matrix tridiag(a,d,a) of size n*n
// from scalar input a and d.
arma::mat create_symmetric_tridiagonal(int n, double a, double d)
{
  // Call create_tridiagonal and return the result
  return create_tridiagonal(n, a, d, a);
}
```

Note:

- Having one function call another like this can help us avoid duplicating what would have been almost identical code in multiple functions.
- In C++ you can have multiple functions with the same function name (here `create_tridiagonal`) as long as the set of input arguments are different. This is known as **function overloading**.
