Series 1

Computational Methods for Engineering Applications
Last edited: September 28, 2020
Due date: October 14 at 23:59

Template codes are available on the course's webpage at `https://moodle-app2.let.ethz.ch/course/view.php?id=13412`.

This is an exercise sheet, not a project, and therefore awards **no bonus points**.

Exercise 1 Short questions: Basic concepts

1a)

Let $u, v : \mathbb{R} \to \mathbb{R}$. For each of the following ODEs, decide whether they are:

a) Scalar
b) Linear
c) Autonomous

Rewrite the non-autonomous equations into an autonomous form.

1. $u'(t) = \cos(u(t))$
2. $u'(t) = u(t)^2 + t$
3. $u'(t) = \exp(t)u(t)$
4. $u'(t) = 4u(t)$
5. $\begin{pmatrix} u'(t) \\ v'(t) \end{pmatrix} = \begin{pmatrix} 3u(t) + v(t) \\ v(t) \end{pmatrix}$
6. $u'(t) = \exp u(t) + \sin(u(t))$

1b)

We are given the ODE
$$u'(t) = -u(t),$$
with initial value $u(0) = A > 0$. What will be the first value u_1, when we solve the ODE using...?

1. The *Forward Euler* method with step size Δt
2. The *Backward Euler* method with step size Δt

Exercise 2 Chemical concentrations

In this exercise we will study the evolution of chemical concentrations. Consider a chemical system consisting of four substances A, B, C and D. We have a reversible chemical reaction of the form
$$A + B \underset{k_2}{\overset{k_1}{\rightleftarrows}} C + D.$$

At time $t \geq 0$, we let $u_1(t), u_2(t), u_3(t)$ and $u_4(t)$ denote the concentration of A, B, C and D respectively. The kinetics are given by the following system
$$\begin{cases} u_1'(t) = -k_1 u_1(t)u_2(t) + k_2 u_3(t)u_4(t) \\ u_2'(t) = -k_1 u_1(t)u_2(t) + k_2 u_3(t)u_4(t) \\ u_3'(t) = -k_2 u_3(t)u_4(t) + k_1 u_1(t)u_2(t) \\ u_4'(t) = -k_2 u_3(t)u_4(t) + k_1 u_1(t)u_2(t) \end{cases}. \tag{1}$$

2a)

Write the system (1) in the form
$$\boldsymbol{u}'(t) = \mathbf{F}(\boldsymbol{u}(t)) \tag{2}$$
where $\boldsymbol{u} \in \mathbb{R}^4$ and $\mathbf{F} : \mathbb{R}^4 \to \mathbb{R}^4$.

2b)

We will use the *trapezoidal rule* (or implicit second order Runge-Kutta) to solve the system (2) numerically. Recall that the trapezoidal rule is given as
$$\boldsymbol{v}_{n+1} = \boldsymbol{v}_n + \frac{\Delta t}{2}\left(\mathbf{F}(\boldsymbol{v}_n) + \mathbf{F}(\boldsymbol{v}_{n+1})\right) \qquad \text{for } n \geq 0, \tag{3}$$
$$\boldsymbol{v}_0 = \boldsymbol{u}_0, \tag{4}$$
where $\Delta t > 0$ is the step size.

Determine the one-step error of (3).

2c)

Notice that (3) is a non-linear equation in $\boldsymbol{v}_{n+1}$, therefore we will use the Newton method to solve for $\boldsymbol{v}_{n+1}$. Recall that the Newton method for solving

$$\mathbf{G}(\boldsymbol{x}) = 0$$

is formulated as

$$D\mathbf{G}(\boldsymbol{x}_k)\Delta\boldsymbol{x}_k = -\mathbf{G}(\boldsymbol{x}_k) \tag{5}$$

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \Delta\boldsymbol{x}_k \qquad k \geq 0, \tag{6}$$

where $\boldsymbol{x}_0$ is some initial guess. Write the Newton step for (3), solving for $\boldsymbol{v}_{n+1}$. What is $\mathbf{G}$ and $D\mathbf{G}$ in this case?

Hint: You should *not* compute $[D\mathbf{G}(\boldsymbol{x}_k)]^{-1}$ by hand.

2d)

Write a function in C++ that computes $\boldsymbol{v}_{n+1}$ using the Newton method. The function should take the following parameters:

- The previous value $\boldsymbol{v}_n$.
- An initial guess.
- The constants k_1 and k_2.
- The step size Δt.
- A tolerance for when to exit the Newton iteration.
- A maximum number of iterations to use for the Newton method.

See the function `newtonSolve` in `template_code1/trapezoidal/trapezoidal.cpp` for a template.

Hint: We strongly suggest that you use the Eigen library for solving the linear system $D\mathbf{G}(\boldsymbol{x}_k)\Delta\boldsymbol{x}_k = -\mathbf{G}(\boldsymbol{x}_k)$. Refer to the course (moodle-) page.

Hint: Use $\|\mathbf{G}(\boldsymbol{x}_k)\|$ as a measure on the error.

2e)

Implement the trapezoidal scheme (3), using the Newton function from the previous exercise. Compute the approximate solution up to time T, using the following constants:

- $\boldsymbol{u}_0 = \begin{pmatrix} 0.2 & 0.3 & 0.1 & 0.4 \end{pmatrix}$
- $k_1 = 0.4$, $k_2 = 0.1$
- $T = 20$
- $\Delta t = T/1000$

Plot u_1, u_2, u_3 and u_4 as a function of time. Also plot the sum $u_1 + u_2 + u_3 + u_4$ as a function of time.

See the function `main` in `template_code1/trapezoidal/trapezoidal.cpp` for a template.

Exercise 3 Explicit vs. Implicit Time Stepping

The universal oscillator equation with no forcing term is given by:

$$\ddot{x} + 2\zeta\dot{x} + x = 0, \quad t \in (0, T), \tag{7}$$

for $T > 0$. In (7), $x : \mathbb{R}^+ \to \mathbb{R}$ denotes the position of the oscillator, and $\dot{x}$ its velocity. The real parameter $\zeta > 0$ determines the damping behavior in the transient regime. For $\zeta > 1$ we have overdamping, for $\zeta = 1$ the so-called critical damping and for $\zeta < 1$ underdamping. In this exercise we consider the case $\zeta < 1$, $\zeta \neq 0$.

As initial conditions we impose

$$x(0) = x_0,\ \dot{x}(0) = v_0. \tag{8}$$

3a)

Equations (7) and (8) can be rewritten as a linear system of first order differential equations with appropriate initial conditions, i.e.:

$$\dot{\boldsymbol{y}} = \mathbf{A}\boldsymbol{y} \tag{9}$$

$$\boldsymbol{y}(0) = \boldsymbol{y}_0. \tag{10}$$

Specify $\mathbf{A} \in \mathbb{R}^{2\times 2}$ and $\boldsymbol{y}, \boldsymbol{y}_0 \in \mathbb{R}^2$.

3b)

Note: *This exercise introduces a technique which has not yet been covered in class, but will be useful in the later part of the course.*

Compute the solution to (7) with initial conditions given by (8) with

$$x_0 = 1,\ v_0 = 0.$$

Hint: Diagonalizing **A** is the key to finding an analytic solution. **Example 1.5** in the lecture notes can be an useful model to follow.

Hint: Recall that $\zeta < 1$ and that, for a real number α, it holds that $e^{\alpha i} = \cos\alpha + i\sin\alpha$, where $i = \sqrt{-1}$ denotes the imaginary unit.

3c)

Recall the explicit Euler timestepping introduced in the lecture. Using the template file `harmonic_oscill.cpp` provided in the handout, implement the function `explicitEuler` to compute the solution $\boldsymbol{y} = \boldsymbol{y}(t)$ to (9) up to the time $T > 0$. The function should take as input the following parameters:

- The initial position x_0 and initial velocity v_0, stored in the 2×1 vector `y0`.
- The damping parameter ζ.
- The step size Δt, in the template called `dt`.
- The final time T, that we assume to be a multiple of Δt.

In output, the function returns the vectors `y1`, `y2` and `time`, where the i-th entry contains the particle position, the particle velocity, and the time, respectively, at the i-th iteration, $i = 1, \ldots, \frac{T}{\Delta t}$. The size of the output vectors has to be initialized inside the function according to the number of time steps.

3d)

Recall the implicit Euler timestepping introduced in the lecture. Using the template file `harmonic_oscill.cpp` provided in the handout, implement the function `implicitEuler` to compute the solution $\boldsymbol{y} = \boldsymbol{y}(t)$ to (9) up to the time $T > 0$. The input and output parameters are as in the function `explicitEuler` from subproblem **3c)**.

3e)

In the template file `harmonic_oscill.cpp`, complete the function `Energy` that, given in input a vector containing velocities at different time steps, returns the kinetic energy $E(t) = \frac{1}{2}v^2(t)$, where $v(t)$ denotes the velocity of the particle at time t.

3f)

We consider two time steps $\Delta_1 t = 0.1$ and $\Delta_2 t = 0.5$. We choose $T = 20$, $\zeta = 0.2$.

Using the `main` already implemented in the template file `harmonic_oscill.cpp`, plot the positions and the energies obtained with the explicit Euler time stepping and the implicit Euler for the two choices of time steps. For the position, plot the exact solution from subproblem **4b)**, too. What do you observe?

Exercise 4 Numerical resolution of linear systems

The goal of this exercise is to show that, in order to solve a linear system of equations, computing a matrix inverse is in general a poor strategy; and we will showcase more efficient alternatives.

Let $\mathbf{A} \in \mathbb{R}^{n,n}$ invertible, and $\boldsymbol{b} \in \mathbb{R}^n$. We seek $\boldsymbol{x} \in \mathbb{R}^n$ such that $\mathbf{A}\boldsymbol{x} = \boldsymbol{b}$.

Clearly, since $\mathbf{A}$ is invertible, the solution exists and is unique, $\boldsymbol{x} = \mathbf{A}^{-1}\boldsymbol{b}$. However, in practice, this is not the optimal way of solving the system.

4a)

Find the exact solution to the following linear system of equations:

$$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 2 & 1 \\ 1 & 2 & 2 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}$$

Did you compute an inverse to get to the solution?

4b)

Let $\mathbf{H}_n$ be the Hilbert matrix in $\mathbb{R}^{n,n}$:

$$\mathbf{H}_n = (H_{ij})_{i,j=1}^n \in \mathbb{R}^{n,n}, \qquad H_{ij} = \frac{1}{i+j-1}$$

where the top left corner of the matrix is $H_{1,1} = 1$. For all $n \geq 1$, $\mathbf{H}_n$ is an invertible matrix, so for any right hand side $\boldsymbol{b}$, the system $\mathbf{H}_n\boldsymbol{x} = \boldsymbol{b}$ has a unique solution.

Let $\boldsymbol{e}_n = (1, 1, \ldots, 1) \in \mathbb{R}^n$. We want to compute the solution to

$$\mathbf{H}_n\boldsymbol{x} = \boldsymbol{e}_n. \tag{11}$$

In file `syssolve.cpp`, complete the implementation[1] of function `solve_inverse`, which takes a matrix $\mathbf{A}$ and a right hand side $\boldsymbol{b}$, and solves the system $\mathbf{A}\boldsymbol{x} = \boldsymbol{b}$ by returning $\boldsymbol{x} = \mathbf{A}^{-1}\boldsymbol{b}$.

4c)

For the exact solution $\boldsymbol{x}$ of (11), it must hold that $\mathbf{H}_n\boldsymbol{x} - \boldsymbol{e}_n = 0$. Due to numerical and round-off errors, this need not be true of an approximate numerical solution $\tilde{x}$. However, if we denote the residual vector $R_n := \mathbf{H}_n\tilde{x} - \boldsymbol{e}_n$, we can use it as a measure of how accurate our solution is: heuristically, the closer R_n is to zero, the better our solution is.

Fill in the blank in `main` to call your function `solve_inverse`, and measure the accuracy of the solution. For that, use the relative norm of R_n respect to $\boldsymbol{e}_n$, i.e., $\|R_n\|/\|\boldsymbol{e}_n\|$. We choose this relative norm, rather than absolute, to obtain a meaningful comparison among different matrix sizes.

Test your program with different values of n. (e.g. 10, 100, 1000). You can do this by running `./linalg 100`, or otherwise by changing the appropriate line in the code.

What happens to the error as the matrix becomes larger?

4d)

The previous exercise shows that we need a different approach: some numerical solver for systems of equations. For those of you unfamiliar with these techniques, the idea is to decompose the matrix into a product of matrices which are easier to operate with, in some sense. The Eigen documentation at `https://eigen.tuxfamily.org/dox/group__TutorialLinearAlgebra.html` lists all such decomposition algorithms provided by Eigen, with their advantages and disadvantages.

For the purposes of this course, **LU decomposition** is usually a good choice. Given a matrix $\mathbf{A}$, its LU factorization are two matrices, $\mathbf{L}$ lower-triangular and $\mathbf{U}$ upper-triangular, such that $\mathbf{A} = \mathbf{LU}$.

If we have the LU factorization of a matrix $\mathbf{A}$, solving the system $\boldsymbol{b} = \mathbf{A}\boldsymbol{x} = \mathbf{LU}\boldsymbol{x}$ is very easy:

1. Solve the system $\mathbf{L}\boldsymbol{y} = \boldsymbol{b}$.
2. Solve the system $\mathbf{U}\boldsymbol{x} = \boldsymbol{y}$.

Clearly then, $\mathbf{A}\boldsymbol{x} = \mathbf{LU}\boldsymbol{x} = \mathbf{L}\boldsymbol{y} = \boldsymbol{b}$; so $\boldsymbol{x}$ solves the system. Since $\mathbf{L}$ and $\mathbf{U}$ are triangular matrices, solving these systems is trivial by simple substitution, and therefore very fast. Furthermore, this approach is much more stable numerically than inverting a matrix.

[1]Observe how the template measures how long the code took to run, using C++ library `chrono`; this may be useful for future exercises.

Fill in the blanks in `main` and `solve_lu` to solve the same system (11) using LU factorization, and compute the norm of the residual as in the previous exercise.

Test with different values of n. How do runtime and accuracy compare with `solve_inverse`?

Hint: The Eigen functions `lu()` and `solve()` can be useful here.