Project 1

ETH zürich

Computational Methods for
Engineering Applications
Last edited: October 18, 2021
Due date: November 10 at 23:59

Template codes are available on the course's webpage at `https://moodle-app2.let.ethz.ch/course/view.php?id=15799`.

This project contains some tasks marked as **Core problems**. If you hand them in before the deadline above, these tasks will be corrected and graded. After a successful interview with the assistants (to be scheduled after the deadline), extra points will be awarded. Full marks for the all core problems in all assignments will give a 20% bonus on the total points in the final exam. This is really a bonus, which means that at the exam you can still get the highest grade without having the bonus points (of course then you need to score more points at the exam).

You only need to hand in your solution for tasks marked as core problems for full points, and the interview will only have questions about core problems. However, in order to do them, you may need to solve the previous non-core tasks.

The total number of points for the Core problems of this project is **40 points**. The total number of points over both projects will be 100.

Exercise 1 Heun's Method for Time Stepping

In this exercise, we consider Heun's method for time stepping, a particular Runge-Kutta method. The *Butcher tableau* for this scheme is

$$\begin{array}{c|cc} 0 & 0 & 0 \\ 1 & 1 & 0 \\ \hline & \frac{1}{2} & \frac{1}{2} \end{array}. \tag{1}$$

1a)

Is Heun's method an implicit or an explicit scheme? How can you see it?

1b)

Consider the scalar ODE

$$u'(t) = f(t, u), \quad t \in (0, T), \tag{2}$$

for some $T > 0$.

Let us denote the time step by Δt and the time levels by $t^n = n\Delta t$ for $n = 0, 1, 2, \ldots, \frac{T}{\Delta t}$. Formulate Heun's method, i.e. write down how to perform the time stepping from $u_n \approx u(t^n)$ to $u_{n+1} \approx u(t^{n+1})$ for $n = 0, 1, 2, \ldots, \frac{T}{\Delta t} - 1$.

1c)

Show that Heun's method is *consistent*.

Hint: Check that the consistency conditions for Runge-Kutta methods seen in class hold.

1d)

Show that Heun's method is a *second* order method. We recall that a time stepping method is of order $k \in \mathbb{N}$ if we obtain a truncation error that is $\mathcal{O}(\Delta t^k)$ when inserting the exact solution $u(t)$ in the consistent form of the method (e.g. $u(t^n)$ instead of u_n).

1e)

We have briefly discussed in the lecture that the concept of convergence is necessary for a time stepping method to give accurate results, but it's not sufficient. E.g. Forward Euler is a *convergent* method, but unless the discretization is "fine enough", it may output nonsense. To quantify this, one often uses the concept of **stability** and in particular of *A-stability*. Chapter 5 in the lecture notes contains a detailed introduction to this very important topic.

To study the A-stability of a method, one considers the numerical method applied to the ODE

$$u'(t) = \lambda u(t), \quad t \in (0, +\infty), \tag{3}$$

$$u(0) = 0, \tag{4}$$

for $\lambda \in \mathbb{C}$ (with the primary focus on $\mathrm{Re}(\lambda) < 0$), and analyses for which values of $\lambda\Delta t \in \mathbb{C}$ it holds that u_n remains bounded as $n \to \infty$, or, in other words, that $\frac{|u_{n+1}|}{|u_n|} \leq 1$, $n \in \mathbb{N}$. This analysis allows to identify the so-called *stability region* in the complex plane. (We suggest you to revise the lecture material to recall why studying A-stability for (3) is sufficient also for A-stability of linear systems of equations.)

Determine the inequality that the quantity $w := \lambda\Delta t$ has to satisfy so that Heun's method is stable. Solve the aforementioned inequality for $\lambda \in \mathbb{R}$ and draw the restriction of the stability region on the real line.

From now on, we consider a particular case of (2), with some initial conditions. Namely, we take

$$u'(t) = e^{-2t} - 2u(t), \quad t \in (0, T), \tag{5}$$
$$u(0) = u_0. \tag{6}$$

1f)

(Core problem) Complete the template file `heun.cpp` provided in the handout, implementing the function `Heun` to compute the solution to (5) up to the time $T > 0$. The input arguments are:

- The initial condition u_0.
- The step size Δt, in the template called `dt`.
- The final time T, which we assume to be a multiple of Δt.

In output, the function returns the vectors `u` and `time`, where the i-th entry contains, respectively, the solution u and the time t at the i-th iteration, $i = 0, \dots, \frac{T}{\Delta t}$. The size of the output vectors has to be initialized inside the function according to the number of time steps.

1g)

Using the code from subproblem **2h)**, plot the solution to (5) for $u_0 = 0$, $\Delta t = 0.2$ and $T = 10$. Note that the function `main` is already implemented in the template.

1h)

According to the discussion in subproblem **2g)**, which is the biggest timestep $\Delta t > 0$ for which Heun's method is stable?

1i)

(Core problem) Make a copy of the file `heun.cpp` and call it `heunconv.cpp`. Modify the file `heunconv.cpp` to perform a convergence study for the solution to (5) computed using Heun's method, with $u_0 = 0$ and $T = 10$. More precisely, consider the sequence of timesteps $\Delta t_k = 2^{-k}$, $k = 1, \dots, 8$, and for each of them, compute the numerical solution $u_{\frac{T}{\Delta t_k}} \approx u(T)$ and the error

$|u_{\frac{T}{\Delta t_k}} - u(T)|$, where u denotes the exact solution to (5). Produce a double logarithmic plot of the error versus Δt_k, $k = 1, \ldots, 8$. Which rate of convergence do you observe?

Hint: The exact solution to (5) is $u(t) = te^{-2t}$, $t \in [0, T]$.

1j)

This exercise highlights a very frequent mistake. Repeat the experiment above, now with the sequence of timesteps $\Delta t_k = 0.6 \cdot \left(2^{-k}\right)$, $k = 1, \ldots, 8$. If you have not been careful, your implementation may now only produce first-order convergence, as in the figure below. Why do you think this could happen? How can you fix this?

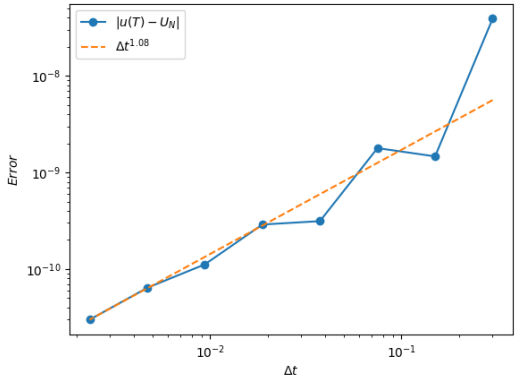

Figure 1: A possible convergence plot for subproblem **1j)**.

Hint: consider $k = 1$ for simplicity, and think carefully about the time-grid.

Exercise 2 Modeling a pandemic

Fiction: During the current SARS-CoV-2 (Corona) pandemic, many people were tested to identify the presence of the virus within them. Like many experiments, these tests do not guarantee 100% reliability. In fact, a number of people have been identified who, after contracting the virus and being cured (i.e. tested negative), have subsequently tested positive again. Due to the substantial lack of knowledge of the virus, some scientists have suggested that the SARS-CoV-2 virus may not be actually "immunising": a part of the population could return to host the virus even after having already contracted it once. If this was the case, how would mankind deal with the presence of such a virus? It seems intuitive to assume that under the "non-immunising" assumption, the human kind will start having a recurring number of infectious people. However, what if just a small percentage would be actually non-immune to the virus after contraction? Would this percentage substantially affect the spread of the pandemic (thus requiring strict social measures)?

In this exercise, you are tasked with modelling the fictional non-immunising coronavirus (and find how this will end), which we model as follows:

We consider five basic classes:

1. Susceptibles (S) : People who do not host the virus and can contract it;
2. Exposed (E) : People who host the virus but do not show symptoms;
3. Infected (I) : People who host the virus and show symptoms;
4. Removed (R) : People who got immune to the virus;
5. Dead (D) : Dead people.

People in each class can become deceased through 'natural' causes, i.e. non Corona-related death (parameter δ).

Susceptibles can contract the virus through the encounter with either an Infected (parameter α) or an Exposed (parameter β).

Exposed people can either develop symptoms becoming Infected (parameter e) or being completely asymptomatic – i.e. never showing symptoms and naturally becoming immune – (parameter γ).

Infected people will either die (parameter σ) or recover (parameter f) from Corona-related complications.

A small percentage of people in the Removed class can contract the virus again after recovery, entering again the Susceptibles class (parameter d).

Since very young people do not seem to be affected by the virus, we denote by Π the rate at which humans are introduced to the class of Susceptibles, as a consequence of growing up to an age where Corona virus can potentially be contracted.

We can rewrite the model above as a non-linear system of ODEs:

$$\begin{aligned} S' &= \Pi S - \alpha(t)SI - \beta(t)SE - \delta S + dR \\ E' &= \alpha(t)SI + \beta(t)SE - eE - \gamma E - \delta E \\ I' &= eE - fI - (\sigma + \delta)I \\ R' &= fI + \gamma E - dR - \delta R \\ D' &= \delta(S + E + I + R) + \sigma I \end{aligned} \tag{7}$$

for variables S, E, I, $R, D : \mathbb{R}^+ \to \mathbb{R}^+$, which encode the population of a class at each time instant. We allow some coefficients to be time-dependent: increasing α and decreasing β represent the ability of humans to fight off the virus spread (by social measures like social distancing, hand washing, etc). We model such measures by a time-dependent, decaying function which decreases the "natural" infectious rates a and b for Infected and Exposed, respectively, to a fraction of their effectiveness (parameter r):

$$\alpha(t) = a\left(1 - r\frac{t}{1+t}\right)$$
$$\beta(t) = b\left(1 - r\frac{t}{1+t}\right)$$

We consider the population units (for susceptible, exposed, infected and removed) to be in thousands of individuals, and time in days. We model up to 450 days from the outbreak of the pandemic, i.e. time $T = 450$ days.

We choose the following values for Π, δ, γ, σ, a, b, d, e, f, and r:

$$\Pi = 3 \cdot 10^{-5}, \quad \delta = 2 \cdot 10^{-5}, \quad \gamma = 1.5 \cdot 10^{-4} \cdot 10^{-4}, \quad \sigma = 2.5 \cdot 10^{-3}$$
$$a = 1.5 \cdot 10^{-3}, \quad b = 0.3 \cdot 10^{-3}, \quad d = 2 \cdot 10^{-3}, \quad e = 0.5, \quad f = 0.5, \quad r = 0.01$$

Hint: This exercise has *unit tests* which can be used to test your solution. To run the unit tests, run the executable `unittest`. Note that correct unit tests are *not* a guarantee for a correct solution. In some rare cases, the solution can be correct even though the unit tests do not pass (always check the output values, and if in doubt, ask the teaching assistant!)

2a)

Write down the explicit form of the functions $F : \mathbb{R} \times \mathbb{R}^4 \to \mathbb{R}$ and $H : \mathbb{R}^4 \to \mathbb{R}$ such that the system of ODEs (7) takes the form

$$\mathbf{U}' = F(t, \mathbf{U})$$
$$D' = H(\mathbf{U})$$

for the vector of variables $\mathbf{U}(t) := [S(t), E(t), I(t), R(t)]^T$.

2b)

Notice that the last equation in our system of ODEs (7) does not depend on the variable D, so that, upon integration, we can get an explicit update formula for the variable D, as a function of the other four. Hence, we would like to integrate this latter equation using a (composite) trapezoidal rule: given a time grid $0 = t_0 < t_1 < \ldots < t_N = T$, one can find the value of D at the time level t_n by integrating the equation over the time interval $[0, t_n]$

$$\int_0^{t_n} H\left(\mathbf{U}(s)\right)\, ds = \int_0^{t_n} D'(s)\, ds = D(t_n) - D(0) = D(t_n) - D_0$$

for some given initial value $D_0 = D(0)$. Since

$$D(t_n) := D_0 + \int_0^{t_n} H\left(\mathbf{U}(s)\right)\, ds = D_0 + \int_0^{t_{n-1}} H\left(\mathbf{U}(s)\right)\, ds + \int_{t_{n-1}}^{t_n} H\left(\mathbf{U}(s)\right)\, ds,$$

we deduce the updating formula for the variable D

$$D(t_n) = D(t_{n-1}) + \int_{t_{n-1}}^{t_n} H\left(\mathbf{U}(s)\right)\, ds. \tag{8}$$

By approximating the integral of (8) using the trapezoidal rule, one ends up with the following updating scheme

$$D(t_n) = D(t_{n-1}) + \overline{H}\left(\mathbf{U}(t_{n-1}), \mathbf{U}(t_n), \Delta t\right). \tag{9}$$

Write down the form of the function $\overline{H}$ and implement the scheme (9) in the function `computeD` in template `corona_dirk/coronaoutbreak.hpp`.

Given the update formula (9), for any of the following methods, we first compute the unknown $\mathbf{U}(t_n)$ and then update the dead variable D according to (9).

2c)

We have implemented a Forward-Euler solver in `corona_dirk/forwardeulersolver.hpp`. The problem is written in terms of the variables $\mathbf{U}(t)$ for the update of the system of ODEs $\mathbf{U}' = F(t, \mathbf{U})$.

Modify and run the program in `corona_dirk/forward_euler.cpp` with the following number of time steps:

$$N_1 = 100$$
$$N_2 = 200$$
$$N_3 = 500.$$

and initial condition

$$\mathbf{U}(0) = \mathbf{U}_0 = \begin{bmatrix} 500 \\ 0 \\ 1 \\ 0 \end{bmatrix}, \qquad D(0) = D_0 = 0.$$

Plot the solution for the various N. What do you observe? Do humans manage to defeat the virus?

Hint: You only have to edit the following line in main that sets `N`:

```
int N = 100;
```

or you can run the program with a command line argument

```
./forward_euler 5e2
```

2d)

In the exercise above, we saw that we need a high number of timesteps in order to get anything close to the exact solution. In this exercise, we will test a Diagonally Implicit Runge-Kutta (DIRK) method.

We will employ the **2-stage, 3rd order accurate** DIRK method, denoted DIRK(2,3). This is given by the following Butcher tableau:

$$\begin{array}{c|cc} \mu & \mu & 0 \\ \mu-\nu & -\nu & \mu \\ \hline & \mu-\frac{1}{2}\nu & \mu-\frac{1}{2}\nu \end{array} \tag{10}$$

where $\mu := \frac{1}{2} + \frac{1}{2\sqrt{3}}$ and $\nu := \frac{1}{\sqrt{3}}$.

Write down the non-linear equations for u_{n+1} for DIRK(2,3) for (7) in the following form

$$G_1(\mathbf{y}_1) = 0 \tag{11}$$

$$G_2(\mathbf{y}_1, \mathbf{y}_2) = 0 \tag{12}$$

$$\mathbf{u}_{n+1} = G(\mathbf{y}_1, \mathbf{y}_2) \tag{13}$$

for functions $G_1 : \mathbb{R}^3 \to \mathbb{R}^3$, $G_2,\ G : \mathbb{R}^3 \times \mathbb{R}^3 \to \mathbb{R}^3$ which may depend on $\mathbf{u}_n = [S_n, E_n, I_n, R_n]^\intercal$, t^n and Δt.

Hint: You do not have to solve the non-linear equations by hand!

2e)

The non-linear systems from task **2d)** are not trivial to solve; therefore we need a numerical solver. Write explicitly the Newton method for the resolution of eq. (11). Do the same for eq. (12).

Hint: You don't have to invert any matrix by hand. In fact, one iteration of the method can be rewritten as a linear system of equations; this means we will be able to later use an LU factorization instead of inverting a matrix, with all the associated benefits.

2f)

(Core problem) In file `coronaoutbreak.hpp`, implement the Jacobian matrix of the right hand side function for eq. (7) in `CoronaOutBreak::computeJF`.

Hint: You can test your code by running the unit tests (`./unittest/unittest` from the command line). The relevant unit tests are those marked as `TestJacobian`.

2g)

(Core problem) In file `dirksolver.hpp`, complete a C++ program that implements the DIRK(2,3) method. For this you need to:

1. In `DIRKSolver::computeG1` (resp. `DIRKSolver::computeG2`), implement the evaluation of G_1 (resp. G_2).

   Hint: Use `coronaOutbreak.computeF(YOUR ARGUMENTS HERE)` for this. `coronaOutbreak` is an object of class `CoronaOutbreak` which is already initialized for you. This contains the parameters α, β, etc; as well as functions `computeF` and your `computeJF`.

2. Implement the Newton solver to determine $\mathbf{y}_1$ (resp. $\mathbf{y}_2$) in `DIRKSolver::newtonSolveY1` (resp. `DIRKSolver::newtonSolveY2`).

3. Compute the full evolution of the problem in `DIRKSolver::solve`. At the end of the program, `u[i][n]` must contain an approximation to $u_i(t^n)$, $D_i(t^n)$, and `time[n]` must be $n\Delta t$, for $n \in \{0, 1, \ldots, N\}$ and $i \in \{1, 2, 3, 4\}$.

Hint: Mind capitalization! `CoronaOutbreak` is a class, and `coronaOutbreak` an object. If one has a `double x = 4.0;`, and does `sqrt(x);`, everything makes sense. But doing `sqrt(double);` is nonsense. For this same reason, `CoronaOutbreak.computeF(YOUR ARGUMENTS HERE)` will not work.

Hint: You can test your code by running the unit tests (`./unittest/unittest` from the command line). The relevant unit tests are those marked as `TestGFunctions` (step 1), `TestNewtonMethod` (step 2), and `TestDirkSolver` (step 3).

2h)

Use your function `dirk` to compute the solution up to T for the following number of timesteps:

$$N_1 = 100$$
$$N_2 = 200$$
$$N_3 = 500$$

Plot the solution for the different simulations. How does this compare against the results using Forward-Euler?

2i)

(Core problem) We want to study the convergence of our scheme when using DIRK(2,3) for system (7). Complete `dirkconv.cpp` to perform a convergence study of the solution to (7). **Use your implementation** of `solve` in file `dirksolver.cpp`; you can do this by calling

```
dirkSolver.solve(/*your parameters here*/).
```

To find the convergence rate, first we need a test case for which we know an exact solution, in order to compare our approximation. Let us choose $\alpha \equiv \beta \equiv 0$ and $d = e = f = \gamma = \sigma = 0$; and initial condition $(S_0, 0, 0, 0)$ and $D_0 = 0$. This means that we start with only susceptibles, no infectious person is present and nobody can then "immunise" and become susceptible again; i.e. the Corona-free scenario. Therefore, we just have normal exponential growth for S, and thus for D, through birth and natural mortality rates. Writing it formally,

$$\begin{aligned}
&S' = (\Pi - \delta)S, && S(0) = S_0 && \Rightarrow \quad S(t) = S_0 e^{(\Pi-\delta)t} \\
&E' = I' = R' = 0, && E(0) = I(0) = R(0) = 0 && \Rightarrow \quad E(t) = I(t) = R(t) = 0 \\
&D' = \delta S, && D(0) = 0 && \Rightarrow \quad D(t) = \frac{S_0\delta}{\Pi - \delta}\left(e^{(\Pi-\delta)t} - 1\right)
\end{aligned}$$

with $S_0 = 500$ and $t = T = 450$. In order to see results more clearly, we will use larger values for the natality/mortality rate, $\Pi = 0.03$ and $\delta = 0.02$. Use $N = 200 \cdot 2^i$, for $i \in \{0, 1, \ldots, 8\}$.

For now, `dirkconv` should generate two `.txt` files: `numbers.txt` containing the number of time-steps, and `errors.txt` containing the L^1 error of the approximation with respect to the exact solution at time T; that is,

$$|D_i(T) - (D_i)_N| + \sum_{i=1}^{4} |u_i(T) - (u_i)_N|. \tag{14}$$

A third file, `walltimes.txt`, will be generated from the contents of vector `walltimes`; you can ignore it for this task.

Which rate of convergence do you observe? Do results match what you expected to see? Justify your answer.

Hint: $u_i(T)$ and $D_i(T)$ are already computed as `exact`.

2j)

(Core problem) The study of the convergence above tells us how good our results get *as we refine the mesh*. For real-world problems, usually we have limited resources, and we need to figure out whether our solution is cost-effective. This means that, often, the really interesting question is: how good do our results get *as we increase the cost of the simulation*? And the simplest measure of cost is: "how long did the simulation take to run?".

We are going to finish the program `dirkconv.cpp` by making it measure runtime. For that, you need to save the time the simulation took to run, for each resolution, in vector `walltimes`. Class `std::chrono::high_resolution_clock`, contained in library `<chrono>` can be useful.

2k)

Let us now exclude the computation of D from the error metric (14), and fix $T = 101$. With the same parameters as above, we increase the number of meshpoints further, $N = 200 \cdot 2^i$, for $i \in \{0, 1, \ldots, 12\}$. We plot error versus number of points, and we obtain Figure 2.

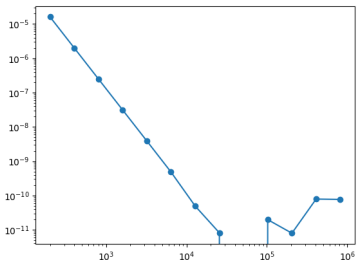

Figure 2: L^1 error vs N, N up to 819200

What do you observe? Why do you think this happens?