Project 1

ETH zürich

Computational Methods for
Engineering Applications
Last edited: November 25, 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?

Solution: It is an explicit scheme. We can deduce it from the Butcher tableau, noticing that, there, the coefficient matrix $(a_{i,j})_{i,j}$ is such that $a_{i,j} = 0$ for $i \leq j$.

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 $n = 0, 1, 2, \ldots, \frac{T}{\Delta t} - 1$.

Solution: According to the Butcher tableau, the system reads:

$$\begin{aligned} y_1 &= u_n \\ y_2 &= u_n + \Delta t f(t^n, y_1) \\ u_{n+1} &= u_n + \Delta t \left(\frac{1}{2} f(t^n, y_1) + \frac{1}{2} f(t^{n+1}, y_2) \right), \end{aligned}$$

for $n = 0, 1, \ldots, \frac{T}{\Delta t} - 1$.

Alternatively, the system can be written as follows:

$$\begin{aligned} k_1 &= f(t^n, u_n), \\ k_2 &= f(t^n + \Delta t, u_n + \Delta t k_1) = f(t^{n+1}, u_n + \Delta t f(t^n, u_n)), \\ u_{n+1} &= u_n + \Delta t \left(\frac{1}{2} k_1 + \frac{1}{2} k_2 \right), \end{aligned}$$

for $n = 0, 1, \ldots, \frac{T}{\Delta t} - 1$.

1c)

Show that Heun's method is *consistent*.

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

Solution: We have to check that $\sum_{j=1}^{s} a_{i,j} = \sum_{j=1}^{2} a_{i,j} = c_i$ for $i = 1, 2$, and $\sum_{j=1}^{s} b_j = \sum_{j=1}^{2} b_j = 1$. From the Butcher tableau, it is trivial to see that these equalities hold and thus the method is consistent.

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).

Solution: The consistent form of the method reads:

$$\frac{u_{n+1} - u_n}{\Delta t} = \frac{1}{2} f(t^n, u_n) + \frac{1}{2} f(t^{n+1}, u_n + \Delta t f(t^n, u_n)).$$

we obtain a truncation error that is $\mathcal{O}(\Delta t^k)$ The Taylor expansion for the last summand (with $u(t^n)$ instead of u_n) is:

$$\begin{aligned}
&f(t^{n+1}, u(t^n) + \Delta t f(t^n, u(t^n))) = f(t^{n+1}, u(t^n) + \Delta t u'(t^n)) \\
&= f(t^n, u(t^n)) + \Delta t \frac{\partial f}{\partial t}(t^n, u(t^n)) + \Delta t u'(t^n) \frac{\partial f}{\partial u}(t^n, u(t^n)) \\
&+ \frac{\Delta t^2}{2} \left(\frac{\partial^2 f}{\partial t^2}(t^n, u(t^n)) + (u'(t^n))^2 \frac{\partial^2 f}{\partial u^2}(t^n, u(t^n)) + 2u'(t^n) \frac{\partial^2 f}{\partial t \partial u}(t^n, u(t^n)) \right) + \mathcal{O}(\Delta t^3) \\
&= f(t^n, u(t^n)) + \Delta t f'(t^n, u(t^n)) \\
&+ \frac{\Delta t^2}{2} \left(\frac{\partial^2 f}{\partial t^2}(t^n, u(t^n)) + (u'(t^n))^2 \frac{\partial^2 f}{\partial u^2}(t^n, u(t^n)) + 2u'(t^n) \frac{\partial^2 f}{\partial t \partial u}(t^n, u(t^n)) \right) + \mathcal{O}(\Delta t^3) \\
&= u'(t^n) + \Delta t u''(t^n) \\
&+ \frac{\Delta t^2}{2} \left(\frac{\partial^2 f}{\partial t^2}(t^n, u(t^n)) + (u'(t^n))^2 \frac{\partial^2 f}{\partial u^2}(t^n, u(t^n)) + 2u'(t^n) \frac{\partial^2 f}{\partial t \partial u}(t^n, u(t^n)) \right) + \mathcal{O}(\Delta t^3)
\end{aligned}$$

where we have used the partial derivatives of f with repect to its first and second argument, and the fact that $f'(t, u(t)) := \frac{df}{dt}(t, u(t)) = \dfrac{\partial f}{\partial t}(t, u(t)) + \dfrac{\partial f}{\partial u}(t, u(t)) u'(t)$.

Inserting the Taylor expansion above and expanding the other summands too, we obtain:

$$\begin{aligned}
&\frac{u(t^{n+1}) - u(t^n)}{\Delta t} - \frac{1}{2} f(t^n, u(t^n)) - \frac{1}{2} f(t^{n+1}, u(t^n) + \Delta t f(t^n, u(t^n))) = \\
&= u'(t^n) + \frac{1}{2} \Delta t u''(t^n) + \frac{1}{6} \Delta t^2 u'''(t^n) + \mathcal{O}(\Delta t^3) - \frac{1}{2} u'(t^n) - \frac{1}{2} u'(t^n) - \frac{1}{2} \Delta t u''(t^n) \\
&- \frac{\Delta t^2}{4} \left(\frac{\partial^2 f}{\partial t^2}(t^n, u(t^n)) + (u'(t^n))^2 \frac{\partial^2 f}{\partial u^2}(t^n, u(t^n)) + 2u'(t^n) \frac{\partial^2 f}{\partial t \partial u}(t^n, u(t^n)) \right) + \mathcal{O}(\Delta t^3) \\
&= \Delta t^2 \left(\frac{1}{6} u'''(t^n) - \frac{1}{4} \frac{\partial^2 f}{\partial t^2}(t^n, u(t^n)) - \frac{1}{4} (u'(t^n))^2 \frac{\partial^2 f}{\partial u^2}(t^n, u(t^n)) - \frac{1}{2} u'(t^n) \frac{\partial^2 f}{\partial t \partial u}(t^n, u(t^n)) \right) \\
&+ \mathcal{O}(\Delta t^3) \\
&= \mathcal{O}(\Delta t^2),
\end{aligned}$$

which means that the method is second order.

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.

Solution: Heun's method applied to equation (3) gives, for a generic step $n \in \mathbb{N}$:

$$\begin{aligned} u_{n+1} &= u_n + \frac{1}{2}\Delta t\lambda u_n + \frac{1}{2}\Delta t\lambda(u_n + \Delta t\lambda u_n) \\ &= \left(1 + \lambda\Delta t + \frac{1}{2}(\lambda\Delta t)^2\right) u_n. \end{aligned}$$

Thus, the stability region in the complex plane is described by

$$\left\{ w \in \mathbb{C} : \left| \frac{1}{2}w^2 + w + 1 \right| \leq 1 \right\}.$$

For $\lambda \in \mathbb{R}$, we obtain that $w \in [-2, 0]$.

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, $i = 0, \ldots, \frac{T}{\Delta t}$. The size of the output vectors has to be initialized inside the function according to the number of time steps.

Solution: See listing 1.

Listing 1: Implementation of the function `Heun` for subproblem **2h)**.

```
#include <Eigen/Core>
#include <vector>
#include <iostream>
#include "writer.hpp"

/// Uses the Heun's method to compute u from time 0 to time T
/// for the ODE $u'=e^{-2t}-2u$
///
/// @param[out] u at the end of the call (solution at all time steps)
/// @param[out] time contains the time levels
/// @param[in] u0 the initial data
/// @param[in] dt the step size
/// @param[in] T the final time up to which to compute the solution.
///

void Heun(std::vector<double> & u, std::vector<double> & time,
          const double & u0, double dt, double T) {
    const unsigned int nsteps = ceil(T/dt);
    //// CMEA_START_TEMPLATE
    u.resize(nsteps+1);
    time.resize(nsteps+1);

    // generate time vector
    for(unsigned int i=0; i<=nsteps; i++) {
        time[i] = i*dt;
    }
    // be careful! T/dt might not be an integer, so maybe nsteps*dt > T!
    if(time.back() > T) {
        time.back() = T;
    }

    u[0] = u0;

    for(unsigned int i = 0; i < nsteps; i++) {
        dt = std::min(dt, T - time[i]);// last timestep? Make sure we don't overshoot T!
        double k1 = std::exp(-2.*time[i])-2.*u[i];
        double k2 = std::exp(-2.*time[i+1])-2.*(u[i]+dt*k1);
```

```
        u[i+1] = u[i] + dt*0.5*(k1+k2);
    }
    //// CMEA_END_TEMPLATE
}


int main(int argc, char** argv) {

    double T = 10.0;

    double dt = 0.2;

    // To make some plotting easier, we take the dt parameter in as an optional
    // parameter.
    if (argc == 2) {
        dt = atof(argv[1]);
    }

    const double u0 = 0.;
    std::vector<double> time;
    std::vector<double> u;
    Heun(u,time,u0,dt,T);

    writeToFile("solution.txt", u);
    writeToFile("time.txt",time);

    return 0;
}
```

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.

Solution: See Figure 1.

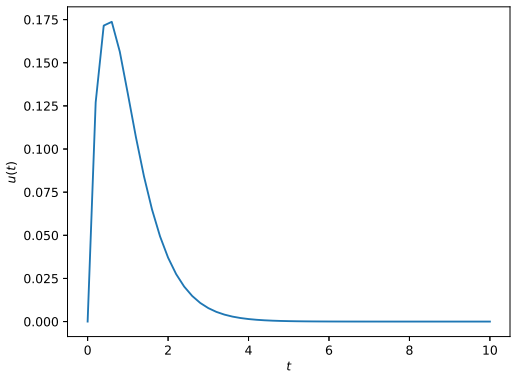


Figure 1: Plot for subproblem **1g)**.

1h)

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

Solution: From equation (5), we can see that the associated homogeneous equation has the eigenvalue $\lambda = -2$. We have to ensure that $|\lambda \Delta t| \in [0, 2]$, and thus the maximum timestep for which we still have stability is $\Delta t = 1$.

Indeed, Figure 2 shows that for $\Delta t = 0.5$, the numerical solution is stable, and, even more, $|u_n| \to 0$ for $n \to \infty$, or in other words, $\frac{|u_{n+1}|}{|u_n|} < 1$ for n big. For $\Delta t = 1$, the method is still stable, so the solution remains bounded as $t \to \infty$; in particular, in this case, $\frac{|u_{n+1}|}{|u_n|} = 1$ for n big. For bigger Δt, instead, the numerical solution is unstable and it tends to $-\infty$ as $t \to \infty$, that is $\frac{|u_{n+1}|}{|u_n|} > 1$ for n big.

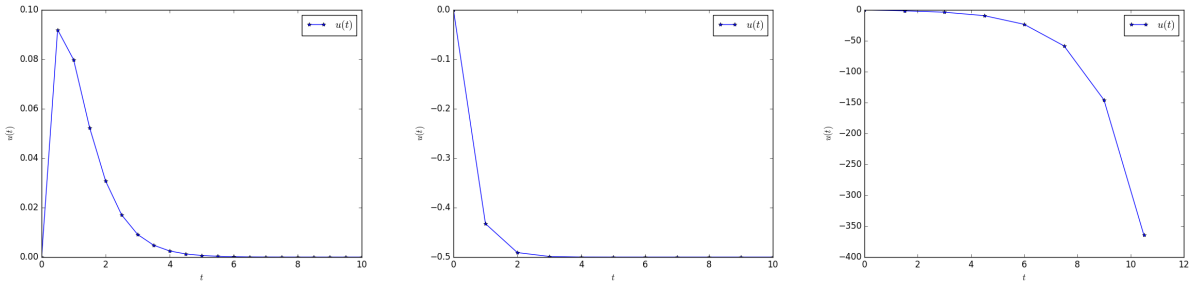


Figure 2: Plots for subproblem **1h)**. Left: $\Delta t = .5$, center: $\Delta t = 1$, right: $\Delta t = 1.5$.

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 $\Delta t_k = 2^{-k}$, $k = 1, \ldots, 8$, 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 $u(t) = te^{-2t}$, $t \in [0, T]$.

Solution: See listing 2 and Figure 3. To extrapolate the empirical order of convergence, we perform a linear fit of the data in the double logarithmic plot. The slope of the fitted line gives us the order of convergence. (Here, we neglect the data for the first time step, because there we are still in preasymptotic regime.) This results in ≈ 2.09, as expected from subproblem **2e)**.

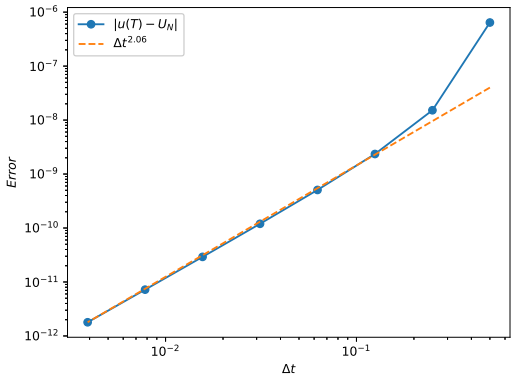


Figure 3: Convergence plot for subproblem **1i)**.

Listing 2: Implementation of the `main` function for subproblem **1i)**.

```
    for(unsigned int i = 0; i < nsteps; i++) {
        dt = std::min(dt, T - time[i]);// last timestep? Make sure we don't overshoot T!
        double k1 = std::exp(-2.*time[i])-2.*u[i];
        double k2 = std::exp(-2.*time[i+1])-2.*(u[i]+dt*k1);
        u[i+1] = u[i] + dt*0.5*(k1+k2);
    }
}

int main(int argc, char** argv) {

    double T = 10.0;
    std::vector<double> dt(8);
```

```
    std::vector<double> error(8);
    const double u0 = 0.;

    // For convenience of testing, we take the factor for dt from command line
    double dt0;
    if (argc == 2) {
        dt0 = atof(argv[1]);
    } else {
        dt0 = 1.;
    }

    for(int i=0; i<8; i++) {
        dt[i]=dt0*std::pow(0.5,i+1);
        std::vector<double> time;
        std::vector<double> u;
        Heun(u,time,u0,dt[i],T);
        double uex = T*std::exp(-2.*T);
        error[i]=std::abs(u.back()-uex);
    }
    writeToFile("dt.txt", dt);
    writeToFile("error.txt", error);
    return 0;
}
//// CMEA_BLANKFILE_TEMPLATE
//// CMEA_END_TEMPLATE
```

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?

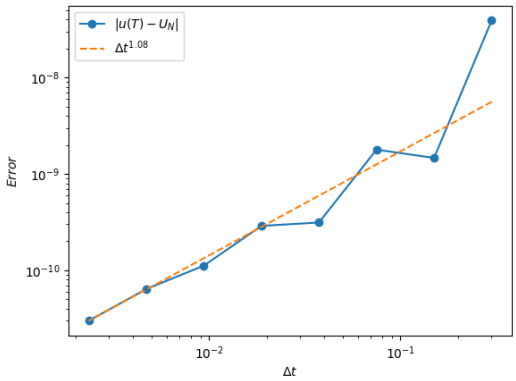


Figure 4: A possible convergence plot for subproblem **1j)**.

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

Solution: This should be shocking: a smaller time-step produces a larger error? The order of the method reduces to (approximately) 1? This is not at all what theory says should happen!

The explanation is subtle, but very simple. Let us write explicitly the grid for $k = 1$:

$$\{0,\ 0.3,\ 0.6,\ 0.9,\ 1.2, \ldots,\ 9.3,\ 9.6,\ 9.9,\ 10.2\};$$

depending on your implementation, your time-grid may stop at 9.9 instead; let us call the last time-point in the grid T' regardless. Note that, when one chooses Δt "freely", as is the case here, it doesn't necessarily happen that $T/\Delta t$ is an integer! Or in other words, if N is the number of time-steps, $T' := N\Delta t \neq T$. In other ways: we're computing the *correct* solution at the *wrong* time! Heun's method works perfectly, it is us who made a mistake.

Standard strategies to solve this problem are:

1. Always pick N integer, and compute Δt from it; not the other way as we did here.
2. Verify that the last entry in your time-grid is T. If it is $T' < T$, add one final time-step of length $T - T'$. If it is $T' > T$, shorten the last time-step appropriately. Either way, you will end up with a time-grid like

   $$\{0,\ 0.3,\ 0.6,\ 0.9,\ 1.2, \ldots,\ 9.3,\ 9.6,\ 9.9,\ \boxed{10}\}.$$

   Make sure you use the correct time-step in step `i` of Heun, e.g. by using:

   ```
   dt' = min(dt, T - time[i]).
   ```

The rest of this solution justifies the answer more rigorously, specifically: why do we still see convergence, but only first order?

As theory guarantees, we obtain a second-order approximation; i.e.,

$$u_N = u(N\Delta t) + O(\Delta t^2);$$

in our case, that means that $|u_N - u(T')| = O(\Delta t^2)$. However, the error is the distance to the solution $u(T)$, not to $u(T')$! Naturally, $u(T')$ and $u(T)$ are "close" to each other. Specifically, since u is smooth, and it must be that $|T' - T| < \Delta t$, Taylor analysis gives us that

$$|u(T) - u(T')| \leq u'(T)\Delta t + C\Delta t^2 = O(\Delta t).$$

This, then, very clearly explains that we still observe first-order convergence:

$$|u(T) - u_N| = |u(T) - u(T') + u(T') - u_N| \leq \underbrace{|u(T) - u(T')|}_{O(\Delta t)} + \underbrace{|u(T') - u_N|}_{O(\Delta t^2)} = O(\Delta t).$$

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

Solution:

$$F(t,\mathbf{U}) = \begin{bmatrix} \Pi S - \alpha(t)SI - \beta(t)SE - \delta S + dR \\ \alpha(t)SI + \beta(t)SE - eE - \gamma E - \delta E \\ eE - fI - (\sigma+\delta)I \\ fI + \gamma E - dR - \delta R \end{bmatrix}, \qquad H(\mathbf{U}) = \delta(S+E+I+R) + \sigma I.$$

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`.

Solution: The function $\overline{H}$ is readily given by the trapezoidal rule

$$\int_{t_{n-1}}^{t_n} H\left(\mathbf{U}(s)\right)\, ds \approx \frac{1}{2}\Big(H\left(\mathbf{U}(t_n)\right) + H\left(\mathbf{U}(t_{n-1})\right) \Big)\Delta t =: \overline{H}\left(\mathbf{U}(t_{n-1}), \mathbf{U}(t_n), \Delta t\right).$$

For its implementation, see Listing 3.

Listing 3: Dead scheme

```
void computeD(std::vector< std::vector<double> >& u, const double dt, int n) {
    //// CMEA_START_TEMPLATE
```

```
        double Sn = u[0][n-1];
        double En = u[1][n-1];
        double In = u[2][n-1];
        double Rn = u[3][n-1];

        double Snp1 = u[0][n];
        double Enp1 = u[1][n];
        double Inp1 = u[2][n];
        double Rnp1 = u[3][n];

        double Hn = delta * (Sn + En + Rn + In) + sigma * In;
        double Hnp1 = delta * (Snp1 + Enp1 + Rnp1 + Inp1) + sigma * Inp1;

        u[4][n] = u[4][n-1] + 0.5 * (Hn + Hnp1) * dt;
        //// CMEA_END_TEMPLATE
    }
```

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?

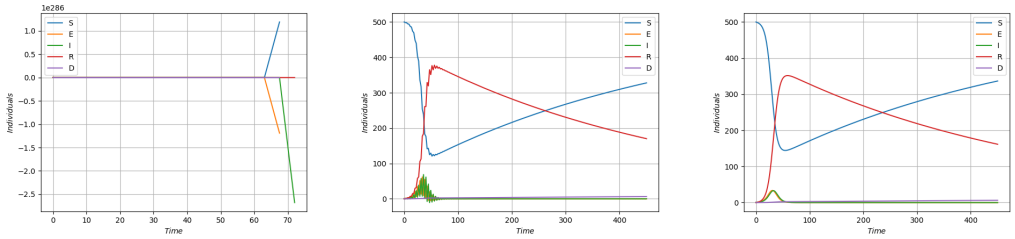
Hint: You only have to edit the following line in main that sets `N`:

```
int N = 100;
```

or you can run the program with a command line argument

```
./forward_euler 5e2
```

Solution: We run the program for the different values of N and get the plots in Figure 5. We clearly see that for small N, the numerical solution is nowhere near the analytical solution, and blows up to physically meaningless solutions: observe that the infected population in subfigure a) nears $-2.5 \cdot 10^{286}$. For values around 200, the solution is better, but high-frequency oscillations appear. For 500, however, a good approximation is found.



(a) $N = 100$. **(b)** $N = 200$. **(c)** $N = 500$.

Figure 5: Solution computed for (7) with Forward-Euler.

For this set of parameters, the virus gets rapidly defeated.

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!

Solution: Inserting $F(t, \mathbf{u})$ into the full Runge-Kutta formula, we get

$$\mathbf{y}_1 = \mathbf{u}_n + \Delta t \mu F(t^n + \mu \Delta t, \mathbf{y}_1) \tag{14}$$

$$\Rightarrow \quad \underbrace{\mathbf{u}_n + \Delta t \mu F(t^n + \mu \Delta t, \mathbf{y}_1) - \mathbf{y}_1}_{=:G_1} = 0 \tag{15}$$

$$\mathbf{y}_2 = \mathbf{u}_n + \Delta t \left[-\nu F(t^n + \mu \Delta t, \mathbf{y}_1) + \mu F(t^n + (\mu - \nu)\Delta t, \mathbf{y}_2)\right] \tag{16}$$

$$\Rightarrow \quad \underbrace{\mathbf{u}_n + \Delta t \left[-\nu F(t^n + \mu \Delta t, \mathbf{y}_1) + \mu F(t^n + (\mu - \nu)\Delta t, \mathbf{y}_2)\right] - \mathbf{y}_2}_{=:G_2} = 0 \tag{17}$$

$$\mathbf{u}_{n+1} = \underbrace{\mathbf{u}_n + \frac{\Delta t}{2}\left[F(t^n + \mu \Delta t, \mathbf{y}_1) + F(t^n + (\mu - \nu)\Delta t, \mathbf{y}_2)\right]}_{=:G} \tag{18}$$

Observe that (15) is only an equation of $\mathbf{y}_1$. Once that is solved, (17) only has $\mathbf{y}_2$ as an unknown; and finally (18) is simply a function evaluation with no unknowns.

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.

Solution: Let $L : \mathbb{R}^k \to \mathbb{R}^k$; we seek zeros for L, i.e. find $\mathbf{u}$ such that $L(\mathbf{u}) = 0$. Pick an initial point $\mathbf{u}_0$. One iteration of the multi-dimensional Newton method reads:

$$\mathbf{u}_{k+1} = \mathbf{u}_k - J_L(\mathbf{u}_k)^{-1} L(\mathbf{u}_k)$$

which we can rewrite as solving the linear system

$$J_L(\mathbf{u}_k)\,(\mathbf{u}_{k+1} - \mathbf{u}_k) = -L(\mathbf{u}_k) \tag{19}$$

whose solution lets us easily iterate with $\mathbf{u}_{k+1} = \mathbf{u}_k + (\mathbf{u}_{k+1} - \mathbf{u}_k)$. Observe that we use k to denote iterations in the Newton solver within a DIRK timestep; for this entire task, n, which indexes timesteps, is fixed.

We apply this to the functions $G_1(\cdot)$ and $G_2(\mathbf{y}_1, \cdot)$. First, it is immediate to find that the Jacobian for F, for fixed t, is

$$J_F\left(t, \begin{bmatrix} S \\ E \\ I \\ R \end{bmatrix}\right) = \begin{bmatrix} \Pi - \delta - \alpha(t)I - \beta(t)E & -\beta(t)S & -\alpha(t)S & d \\ \alpha(t)I + \beta(t)E & \beta(t)S - \gamma - e - \delta & \alpha(t)S & 0 \\ 0 & e & -f - \sigma - \delta & 0 \\ 0 & \gamma & f & -d - \delta \end{bmatrix}$$

And therefore, by linearity of the Jacobian,

$$J_{G_1}\left(t_1, \begin{bmatrix} S \\ E \\ I \\ R \end{bmatrix}\right) = \Delta t \mu J_F\left(t_1, \begin{bmatrix} S \\ E \\ I \\ R \end{bmatrix}\right) - \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

And

$$J_{G_2}\left(t_2, \begin{bmatrix} S \\ E \\ I \\ R \end{bmatrix}\right) = \Delta t \mu J_F\left(t_2, \begin{bmatrix} S \\ E \\ I \\ R \end{bmatrix}\right) - \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

where $t_1 = t^n + \mu\Delta t$, and $t_2 = t^n + (\mu - \nu)\Delta t$.

And so the linear system to solve for $\mathbf{y}_1$, with $\mathbf{u}_k = [S_k, E_k, I_k, R_k]^\intercal$, and $\mathbf{x}_k := \mathbf{u}_{k+1} - \mathbf{u}_k$, is:

$$J_{G_1}(t_1, \mathbf{u}_k)\,\mathbf{x}_k = -G_1(\mathbf{u}_k) \overset{(15)}{=} -\begin{bmatrix} S_n \\ E_n \\ I_n \\ R_n \end{bmatrix} - \Delta t \mu F(t_1, \mathbf{u}_k) + \mathbf{u}_k$$

and $\mathbf{u}_{k+1} = \mathbf{u}_k + \mathbf{x}_k$. Finally $\mathbf{y}_1 \approx \mathbf{u}_K$ for K large enough.

Now that $\mathbf{y}_1$ is known, we repeat the procedure for $\mathbf{y}_2$, with $\mathbf{v}_k = [S_k, E_k, I_k, R_k]^\intercal$:

$$J_{G_2}(t_2, \mathbf{v}_k)\,\mathbf{x}_k = -G_2(\mathbf{v}_k) \overset{(17)}{=} -\begin{bmatrix} S_n \\ E_n \\ I_n \\ R_n \end{bmatrix} + \nu\Delta t F(t_1, \mathbf{y}_1) - \mu\Delta t F(t_2, \mathbf{v}_k) + \mathbf{v}_k$$

with again $\mathbf{v}_{k+1} = \mathbf{v}_k + \mathbf{x}_k$.

Reasonable choices for initial points are $\mathbf{u}_0 := [S_n, E_n, I_n, R_n]^\intercal$, $\mathbf{v}_0 := \mathbf{y}_1$ (or $\mathbf{v}_0 = \mathbf{u}_0$).

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`.

Hint: Note that the values for the parameters are already defined, either by the constructor of `CoronaOutBreak` or by default. See the bottom of `coronaoutbreak.hpp`.

Solution: See listing 4

Listing 4: Jacobian of F

```
void computeJF(Eigen::MatrixXd& J, double t, Eigen::VectorXd U) {
    //// CMEA_START_TEMPLATE
    double S = U[0];
    double E = U[1];
    double I = U[2];
    double R = U[3];

    J.setConstant(0.0);

    J(0, 0) = Pi - delta - alpha(t) * I - beta(t) * E;
    J(0, 1) = -beta(t) * S;
    J(0, 2) = -alpha(t) * S;
    J(0, 3) = d;

    J(1, 0) = alpha(t) * I + beta(t) * E;
    J(1, 1) = - (e + gamma + delta) + beta(t) * S;
    J(1, 2) = alpha(t) * S;

    J(2, 1) = e;
    J(2, 2) = -f - sigma - delta;

    J(3, 1) = gamma;
    J(3, 2) = f;
    J(3, 3) = -(d + delta);
    //// CMEA_END_TEMPLATE
}
```

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).

Solution: We include some code snippets in listing 7; see `code/corona_dirk/dirksolver.hpp` for a full solution.

Listing 5: G_1

```
void computeG1(Eigen::VectorXd& G, Eigen::VectorXd y, double tn,
        Eigen::VectorXd Un, double dt) {
        //// CMEA_START_TEMPLATE
        int dim = y.size();
        Eigen::VectorXd Fy(dim);
        coronaOutbreak.computeF(Fy, tn + mu * dt, y);
        G = Un + dt * mu * Fy - y;
        //// CMEA_END_TEMPLATE
}
```

Listing 6: Newton method for G_2

```
void newtonSolveY2(Eigen::VectorXd& v, Eigen::VectorXd Un,
        Eigen::VectorXd y1, double dt, double tn, double tolerance, int
            ↪ maxIterations) {

   // Use newtonSolveY1 as a model for this
        //// CMEA_START_TEMPLATE
        int dim = Un.size();
        Eigen::VectorXd RHSG2(dim), x(dim);
        Eigen::MatrixXd JG2(dim,dim), JFv(dim,dim);
        v = y1;

        for (int iteration = 0; iteration < maxIterations; ++iteration) {
                coronaOutbreak.computeJF(JFv, tn + (mu - nu)*dt, v);
```

```
                Eigen::MatrixXd JG2 = dt * mu * JFv - Eigen::MatrixXd::
                    ↪ Identity(dim,dim);
                Eigen::VectorXd RHSG2;
                computeG2(RHSG2,v,tn,Un,dt,y1);

                x = JG2.lu().solve(-RHSG2);

                if ( x.norm() <= tolerance ) {
                        return;
                }

                v = v + x;
        }

        // If we reach this point, something wrong happened.
        throw std::runtime_error("Did not reach tolerance in Newton
            ↪ iteration in Y2");
        //// CMEA_END_TEMPLATE
}
```

Listing 7: Full DIRK timestepping

```
void solve(std::vector<std::vector<double> >& u, std::vector<double>&
    ↪ time,
        double T, int N) {

        const double dt = T / N;

        // Your main loop goes here. At iteration n,
        // 1) Find Y_1 with newtonSolveY1 (resp. Y2)
        // 2) Compute U^{n+1} with F(Y1), F(Y2)
        // 3) Write the values at u[...][n]
        // 4) Compute D and write time[n]

        //// CMEA_START_TEMPLATE
        int dim = u.size()-1;
        Eigen::VectorXd Fy1(dim), Fy2(dim);

        for (int i = 1; i < N + 1; ++i) {
                double tn = time[i - 1];
                Eigen::VectorXd uPrevious(dim);

                for(int k=0; k<dim; ++k){
                        uPrevious(k) = u[k][i - 1];
                }

                Eigen::VectorXd y1(dim), y2(dim);
```

```
                newtonSolveY1(y1, uPrevious, dt, time[i - 1], 1e-10, 100);
                newtonSolveY2(y2, uPrevious, y1, dt, time[i - 1], 1e-10,
                    ↪ 100);
                coronaOutbreak.computeF(Fy1, tn + mu * dt, y1);
                coronaOutbreak.computeF(Fy2, tn + (mu - nu)*dt, y2);
                Eigen::VectorXd uNext = uPrevious + (dt / 2.) * (Fy1 + Fy2);

                u[0][i] = uNext[0];
                u[1][i] = uNext[1];
                u[2][i] = uNext[2];
                u[3][i] = uNext[3];

                coronaOutbreak.computeD(u,dt,i);

                time[i] = time[i - 1] + dt;
        }

        //// CMEA_END_TEMPLATE
}
```

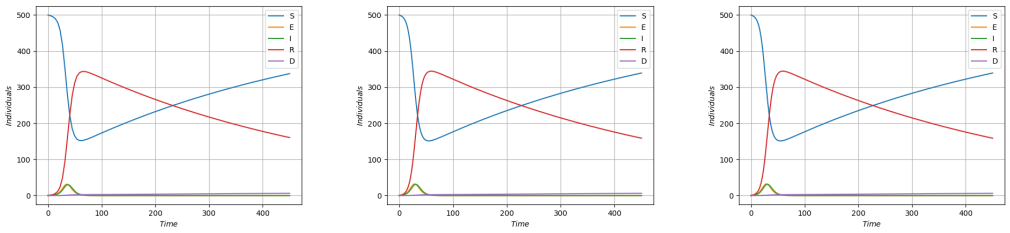
2h)

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?

Solution: We obtain the plots in Figure 6. We observe that the results approximate **u** well even for N as low as N_1. Remember that we needed several hundreds with forward Euler!



(a) $N = 100$. **(b)** $N = 200$. **(c)** $N = 500$.

Figure 6: Solution computed for (7) with DIRK.

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,

$$|D_i(T) - (D_i)_N| + \sum_{i=1}^{4} |u_i(T) - (u_i)_N|. \tag{20}$$

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`.

Solution: Please see the results for the next exercise.

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.

Solution: You can see the results in Fig. 7 and Listing 8, with time measured in nanoseconds. In particular, observe that error scales with (maybe surprisingly) *second* order with respect to runtime, and runtime is linear with number of cells. And thus error has to be second order with respect to runtime as well. Informally, if we denote R runtime, N number of cells, and e error, then:

$$e = O(N^{-2}),\ R = O(N),\ \Rightarrow e = O(R^{-2})$$

This is precisely what we observe.

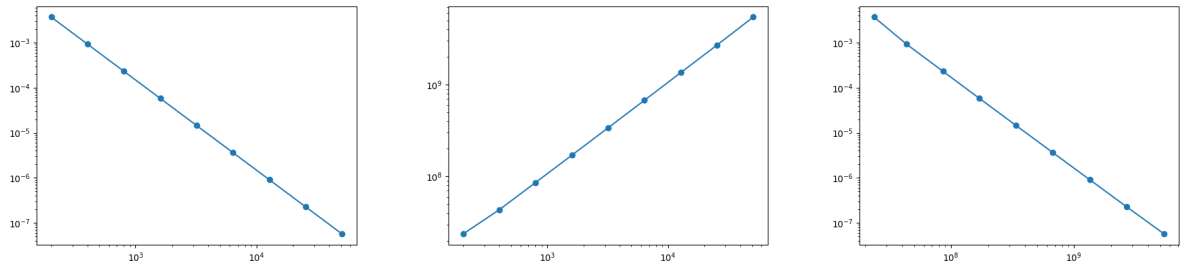


Figure 7: Plots for subproblem **2i)**. Left: L^1 error vs N, center: walltime vs N, right: L^1 error vs walltime.

Listing 8: Convergence study.

```
int main(int argc, char** argv) {

    double T = 101;
    CoronaOutbreak outbreak(0,0,0,0,0,0.03,0.02,0,0);
    std::vector<double> u0(5);
    u0[0] = 500;
    u0[1] = 0;
    u0[2] = 0;
    u0[3] = 0;
    u0[4] = 0;

    // Compute the exact solution for the parameters above
    std::vector<double> exact = outbreak.computeExactNoCorona(T, u0[0]);

    // Initialize solver object for the parameters above
    DIRKSolver dirkSolver(outbreak);

    int minExp = 0;
    int maxExp = 8;
    int countExponents = maxExp - minExp +1;
    std::vector<double> numbers(countExponents);
    std::vector<double> walltimes(countExponents);
    std::vector<double> errors(countExponents);

    //// CMEA_START_TEMPLATE
    std::vector<std::vector<double> > u(5);
    int baseN = 200;

    for (int i = 0; i < countExponents; i++) {
        int N = (1 << (i + minExp) ) * baseN;
        std::cout << "Running for N = " << N << "..." << std::endl;

        std::vector<double> time(N + 1, 0);
        u[0].resize(N + 1, 0);
        u[1].resize(N + 1, 0);
        u[2].resize(N + 1, 0);
        u[3].resize(N + 1, 0);
        u[4].resize(N + 1, 0);

        u[0][0] = u0[0];
        u[1][0] = u0[1];
        u[2][0] = u0[2];
        u[3][0] = u0[3];
        u[4][0] = u0[4];

        auto begin = std::chrono::high_resolution_clock::now();
        dirkSolver.solve(u, time, T, N);
        auto end = std::chrono::high_resolution_clock::now();

        int N_metric = u.size();
```

```
        std::cout << "Approx sol: ";
        for (int k = 0; k< N_metric; k++){
            errors[i] += std::abs(u[k].back() - exact[k]);
                std::cout << u[k].back() << " ";
        }
        std::cout << std::endl;

        numbers[i] = N;
        walltimes[i] = std::chrono::duration_cast<std::chrono::nanoseconds>
            (end - begin).count();
    }
    //// CMEA_END_TEMPLATE

    writeToFile("numbers.txt", numbers);
    writeToFile("errors.txt", errors);
    writeToFile("walltimes.txt", walltimes);

}
```

We know that DIRK(2,3) is **third** order accurate, then why do we observe only second order of the error with respect to mesh-size N? The reason resides in the splitting of our system of equations: as discussed in the lecture, the trapezoidal rule is only **second** order accurate. Heuristically, this means that (asymptotically) the error done when computing D is one order of magnitude higher than the one associated to $\mathbf{U}$. Hence, the global error gets dominated by the error associated to D, and what we are observing is the error of the trapezoidal rule.

If we omit the computation of D in our error metric (20), we recover the expected third order of convergence, see Fig. 8. Please notice that the third order of convergence is recovered not only due to the difference in discretization, but also to the fact that the evolution of all other variables are independent of D; even if the approximation of D is low-order, it will not affect the results of other variables. If we applied a trapezoidal rule to e.g. S, since it appears on the right hand side of eq. (7), we would observe second order of convergence for all variables, with respect to the metric (20).

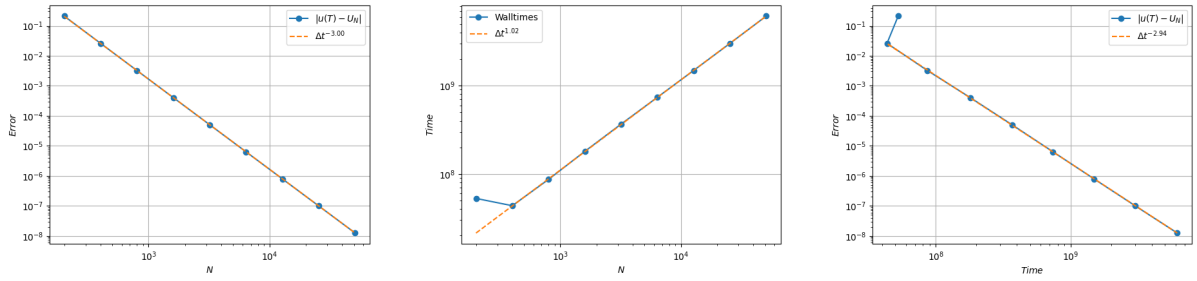


Figure 8: Plots for subproblem **2i)**. Left: L^1 error vs N, center: walltime vs N, right: L^1 error vs walltime.

This provides a very good illustration of the advantages, and cost, of higher order methods. We know that DIRK(2,3) is third order (i.e. the global error scales with Δt^3), and forward Euler is first order. In principle, this would make DIRK preferrable. However, it is also clear that DIRK is a more complex method than forward Euler: its implementation is not as straightforward, and it requires many more operations, due to the implicitness.

Fig. 9 compares the results obtained with the above `dirkconv` (when omitting computations of D!) and a similar study of convergence for forward Euler. The left plot (errors vs N) shows clearly what we already knew, that DIRK converges with a higher order than forward Euler.

The middle plot, walltime vs N, shows that the runtime of both DIRK and forward Euler grows linearly with the number of cells. However, observe how DIRK is about 100 times slower than forward Euler for a given N.

The most interesting plot is the third, error vs walltime. Given a fixed amount of computing power, this measures how small the error that one can obtain is. The plot here shows that, in the region of overlap, DIRK incurs in an error about 4 orders of magnitude lower for the same computational load! That is: if we are limited to 0.1 seconds of computations, Forward Euler could give us a rather poor error of about 10^1, while the error for DIRK would be 10^{-2}!

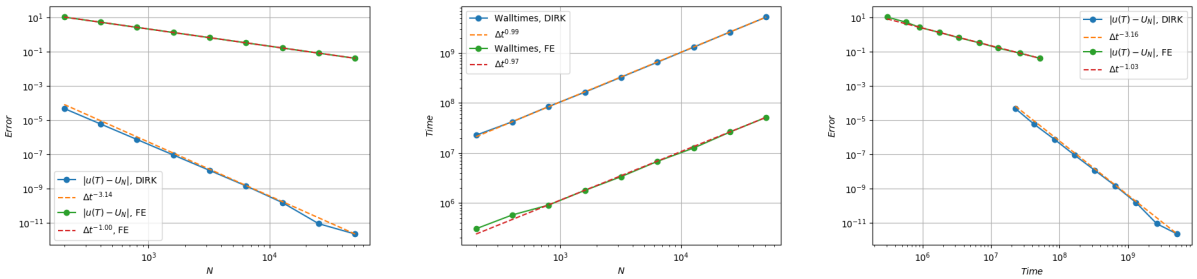


Figure 9: A comparison of forward Euler and DIRK(2,3). Left: L^1 error vs N, center: walltime vs N, right: L^1 error vs walltime.

2k)

Let us now exclude the computation of D from the error metric (20), 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 10.

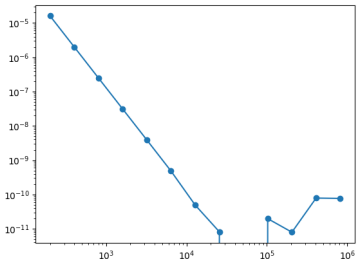


Figure 10: L^1 error vs N, N up to 819200

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

Solution: You can see that for $N \geq N_0 \approx 50000$, the error stops improving – in fact it apparently increases!

The reason can be read from the y-axis. For the right half of the plot, error is smaller than 10^{-10}. At this point, we run into *machine precision* issues: computers cannot accurately represent very small numbers, due to the finite amount of memory available and the nature of the floating point format.

This means that, in practice, we can think of any output "small enough" to be effectively zero. Naturally, what is "small enough" depends on many factors – computer architecture, data type used, purpose of the simulation, etc. But as a rule of thumb, values smaller than 10^{-10} in absolute value can be considered to be "machine zero", and any numerical results involving them should at the very least be taken with a grain of salt.

In this case, then, one shouldn't read it as "for fine enough meshes, the error becomes worse". Rather, for meshes finer than $N \approx 50000$, out approximation is accurate to machine precision; nothing can be gained by refining further.