Project 1

Computational Methods for
Engineering Applications
Last edited: November 21, 2019
Due date: November 3 at 23:59

Template codes are available on the course's webpage at `https://moodle-app2.let.ethz.ch/course/view.php?id=11356`.

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

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

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

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 $f'(t, u(t)) := \frac{df}{dt}(t, u(t)) = \frac{\partial f}{\partial t}(t, u(t)) + \frac{\partial f}{\partial u}(t, u(t)) u'(t)$.

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 seen in the lecture that the concept of convergence is necessary for a time stepping method to give accurate results, but it's not sufficient. Due to this, we introduced the concept of stability and in particular of *A-stability*.

We recall that 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 **2f)**.

```
#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 = round(T/dt);
    //// CMEA_START_TEMPLATE
    u.resize(nsteps+1);
    time.resize(nsteps+1);

    time[0] = 0.;
    u[0] = u0;

    for(unsigned int i = 0; i < nsteps; i++)
    {
        time[i+1] = (i+1)*dt;
        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 **2f)**, 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.

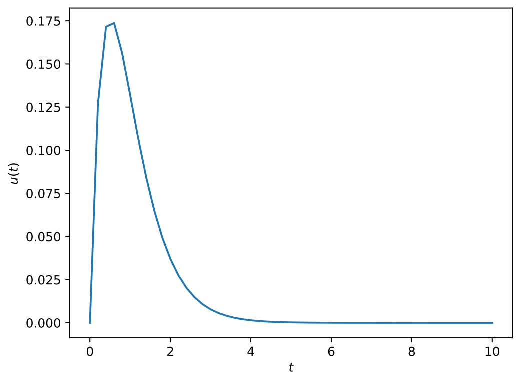


Figure 1: Plot for subproblem **1g)**.

1h)

According to the discussion in subproblem **2e)**, 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.

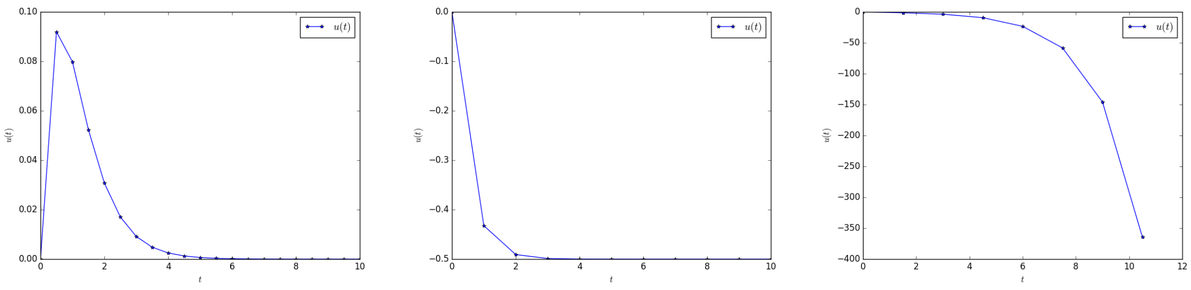


Figure 2: Plots for subproblem **1h)**. Left: $\Delta t = .5$, center: $\Delta t = 1$, right: $\Delta t = 1.5$.

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 **2c)**.

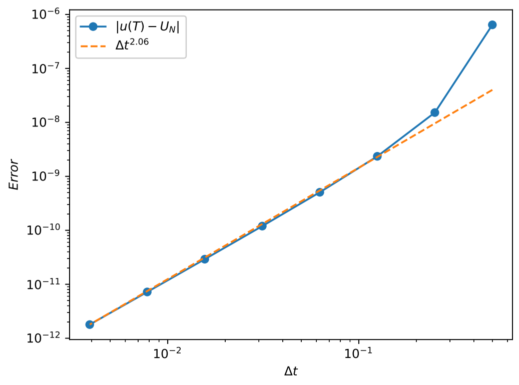


Figure 3: Convergence plot for subproblem **1i)**.

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

```
int main() {

   double T = 10.0;
   std::vector<double> dt(8);
   std::vector<double> error(8);
   const double u0 = 0.;

   for(int i=0; i<8; i++) {
      dt[i]=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);
      std::cout << time.back() << std::endl;
      error[i]=std::abs(u.back()-uex);
   }
   writeToFile("dt.txt", dt);
   writeToFile("error.txt", error);
   return 0;
}
//// CMEA_BLANKFILE_TEMPLATE
//// CMEA_END_TEMPLATE
```

Exercise 2 Dawn of the DIRK

Fiction: It all started in Hönggerberg. The biology students made a huge mistake in their latest lab experiment. They created a zombie virus! Quickly it infected the whole biology department.

Since exam session was upon us, no one noticed people turning into zombies. And now it is up to you to find how this will end.

In this exercise, you are tasked with modelling the fictional Hönggerberg Zombie Virus Zurich (HZVZ). We base our model on *When zombies attack!: Mathematical modelling of an outbreak of zombie infection* [1], which proposes:

> We consider three basic classes: Susceptible (S), Zombie (Z) and Removed (R).
>
> Susceptibles can become deceased through 'natural' causes, i.e. non-zombie-related death (parameter δ). The removed class consists of individuals who have died, either through attack or natural causes. Humans in the removed class can resurrect and become a zombie (parameter ζ).
>
> Susceptibles can become zombies through transmission via an encounter with a zombie (transmission parameter β). Only humans can become infected through contact with zombies, and zombies only have a craving for human flesh so we do not consider any other lifeforms in the model. New zombies can only come from two sources: the resurrected from the newly deceased (removed group), and susceptibles who have 'lost' an encounter with a zombie.
>
> In addition, we assume the birth rate is [...] Π. Zombies move to the removed class upon being 'defeated'. This can be done by removing the head or destroying the brain of the zombie (parameter α). We also assume that zombies do not attack/defeat other zombies.

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

$$\begin{aligned} S' &= \Pi S - \beta S Z - \delta S \\ Z' &= \beta S Z + \zeta R - \alpha S Z \\ R' &= \delta S + \alpha S Z - \zeta R \end{aligned} \tag{7}$$

for variables S, Z, $R : \mathbb{R}^+ \to \mathbb{R}^+$, which encode the population of a class at each time instant. Unlike in [1], we allow coefficients to be time-dependent. Increasing α and decreasing β represent the improving ability of humans to fight off the undead, and survival of the fittest; ζ allows us to delay the start of the event.

We consider the population units (for survivors, zombies and removed) to be in thousands of individuals, and time in days. We model up to time $T = 101$ days.

We choose the following values, where α and β have units in (thousand individuals)$^{-1}$day^{-1}, and

Π, δ and ζ in day^{-1}.

$$\begin{aligned}
\Pi &= 3 \cdot 10^{-5} \\
\delta &= 2 \cdot 10^{-5} \\
\alpha(t) &= 0.1 + \frac{0.1t}{1+t} \\
\beta(t) &= 0.2 - \frac{0.05t}{1+t} \\
\zeta(t) &= \begin{cases} 0 & \text{if } t \leq 5 \\ 0.086 & \text{if } t > 5 \end{cases}
\end{aligned}$$

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)

We have implemented a Forward-Euler solver in `zombie_dirk/forwardeulersolver.hpp`. The problem is written in terms of the variables

$$\mathbf{U}(t) = \begin{bmatrix} S(t) \\ Z(t) \\ R(t) \end{bmatrix}$$

Modify and run the program in `zombie_dirk/forward_euler.cpp` with the following number of timesteps:

$$\begin{aligned}
N_1 &= 1000 \\
N_2 &= 3000 \\
N_3 &= 5000.
\end{aligned}$$

and initial condition

$$\mathbf{U}(0) = \mathbf{U}_0 = \begin{bmatrix} 500 \\ 0 \\ 0 \end{bmatrix}$$

Plot the solution for the various N. What do you observe? Do humans survive this scenario?

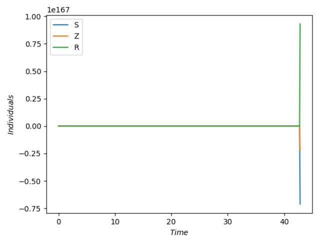
Hint: You only have to edit the following line in main that sets `N`:

```
int N = 5000;
```

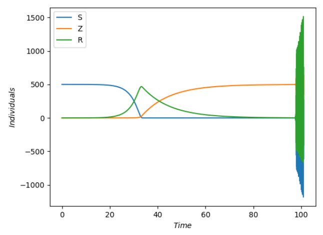
or you can run the program with a command line argument

```
./forward_euler 5e3
```

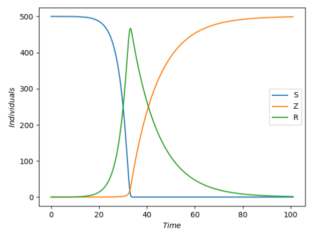
Solution: We run the program for the different values of N and get the plots in Figure 4. We clearly see that for small N, the numerical solution is nowhere near the analytical solution, and blows up to physically meaningless solutions: observe that the human population in subfigure a) nears $-1 \cdot 10^{167}$. For values around 3000, the solution is better, but eventually high-frequency oscillations appear. For 5000, however, a good approximation is found.



(a) $N = 1000$.



(b) $N = 3000$.



(c) $N = 5000$.

Figure 4: Solution computed for (7) with Forward-Euler.

Unfortunately, mankind gets wiped out around day 33. Note that the undead rise only with $\zeta > 0$ at day 5; thus humans die off 28 days later.

2b)

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:

$$\begin{array}{c|cc} \frac{1}{2}+\frac{1}{2\sqrt{3}} & \frac{1}{2}+\frac{1}{2\sqrt{3}} & 0 \\ \frac{1}{2}-\frac{1}{2\sqrt{3}} & -\frac{1}{\sqrt{3}} & \frac{1}{2}+\frac{1}{2\sqrt{3}} \\ \hline & \frac{1}{2} & \frac{1}{2} \end{array} \tag{8}$$

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

$$G_1(\mathbf{y}_1) = 0 \tag{9}$$
$$G_2(\mathbf{y}_1, \mathbf{y}_2) = 0 \tag{10}$$
$$\mathbf{u}_{n+1} = H(\mathbf{y}_1, \mathbf{y}_2) \tag{11}$$

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

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

Solution:

We name for convenience $\mu := {}^1\!/\!{}_2 + {}^1\!/\!{}_{2\sqrt{3}}$, $\nu := {}^1\!/\!{}_{\sqrt{3}}$. We denote the right hand side function of (7) by F,

$$F\left(t, \begin{bmatrix} S \\ Z \\ R \end{bmatrix}\right) = \begin{bmatrix} \Pi S - \beta(t)SZ - \delta S \\ \beta(t)SZ + \zeta(t)R - \alpha(t)SZ \\ \delta S + \alpha(t)SZ - \zeta(t)R \end{bmatrix} \tag{12}$$

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

$$\mathbf{y}_1 = \mathbf{u}_n + \Delta t \mu F(t^n + \mu\Delta t, \mathbf{y}_1) \tag{13}$$
$$\Rightarrow \quad \underbrace{\mathbf{u}_n + \Delta t \mu F(t^n + \mu\Delta t, \mathbf{y}_1) - \mathbf{y}_1}_{=:G_1} = 0 \tag{14}$$

$$\mathbf{y}_2 = \mathbf{u}_n + \Delta t \left[-\nu F(t^n + \mu\Delta t, \mathbf{y}_1) + \mu F(t^n + (\mu - \nu)\Delta t, \mathbf{y}_2)\right] \tag{15}$$
$$\Rightarrow \quad \underbrace{\mathbf{u}_n + \Delta t \left[-\nu F(t^n + \mu\Delta t, \mathbf{y}_1) + \mu F(t^n + (\mu - \nu)\Delta t, \mathbf{y}_2)\right] - \mathbf{y}_2}_{=G_2} = 0 \tag{16}$$

$$\mathbf{u}_{n+1} = \underbrace{\mathbf{u}_n + \frac{\Delta t}{2}\left[F(t^n + \mu\Delta t, \mathbf{y}_1) + F(t^n + (\mu - \nu)\Delta t, \mathbf{y}_2)\right]}_{H} \tag{17}$$

Observe that (14) is only an equation of $\mathbf{y}_1$. Once that is solved, (16) only has $\mathbf{y}_2$ as an unknown; and finally (17) is simply a function evaluation with no unknowns.

2c)

The non-linear systems from task **2b)** are not trivial to solve; therefore we need a numerical solver. Write explicitly the Newton method for the resolution of eq. (9). Do the same for eq. (10).

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

$$J_L(\mathbf{u}_k)\left(\mathbf{u}_{k+1} - \mathbf{u}_k\right) = -L(\mathbf{u}_k) \tag{18}$$

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

$$J_F\left(t, \begin{bmatrix} S \\ Z \\ R \end{bmatrix}\right) = \begin{bmatrix} \Pi - \delta - \beta(t)Z & -\beta(t)S & 0 \\ (\beta(t) - \alpha(t))Z & (\beta(t) - \alpha(t))S & \zeta(t) \\ \delta + \alpha(t)Z & \alpha S & -\zeta(t) \end{bmatrix}$$

And therefore, by linearity of the Jacobian,

$$J_{G_1}\left(t_1, \begin{bmatrix} S \\ Z \\ R \end{bmatrix}\right) = \Delta t \mu \begin{bmatrix} \Pi - \delta - \beta(t_1)Z & -\beta(t_1)S & 0 \\ (\beta(t_1) - \alpha(t_1))Z & (\beta(t_1) - \alpha(t_1))S & \zeta(t_1) \\ \delta + \alpha(t_1)Z & \alpha S & -\zeta(t_1) \end{bmatrix} - \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

And

$$J_{G_2}\left(t_2, \begin{bmatrix} S \\ Z \\ R \end{bmatrix}\right) = \Delta t \mu \begin{bmatrix} \Pi - \delta - \beta(t_2)Z & -\beta(t_2)S & 0 \\ (\beta(t_2) - \alpha(t_2))Z & (\beta(t_2) - \alpha(t_2))S & \zeta(t_2) \\ \delta + \alpha(t_2)Z & \alpha S & -\zeta(t_2) \end{bmatrix} - \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

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 $\mathbf{u}_k = [S_k, Z_k, R_k]^\intercal$, is:

$$J_{G_1}\left(t_1, \mathbf{u}_k\right)\mathbf{x}_k = -G_1\left(\mathbf{u}_k\right) \overset{(14)}{=} - \begin{bmatrix} S_n \\ Z_n \\ R_n \end{bmatrix} - \Delta t \mu F\left(t_1, \mathbf{u}_k\right) + \mathbf{u}_k$$

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 $\mathbf{v}_k = [S_k, Z_k, R_k]^\intercal$:

$$J_{G_2}\left(t_2, \mathbf{v}_k\right)\mathbf{x}_k = -G_2\left(\mathbf{v}_k\right) \overset{(16)}{=} - \begin{bmatrix} S_n \\ Z_n \\ R_n \end{bmatrix} + \nu \Delta t F\left(t_1, \mathbf{y}_1\right) - \mu \Delta t F(t_2, \mathbf{v}_k) + \mathbf{v}_k$$

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

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

2d)

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

Solution: See listing 3

Listing 3: Jacobian of F

```
void computeJF(Eigen::Matrix3d& J, double t, Eigen::Vector3d U) {
    //// CMEA_START_TEMPLATE
    double S = U[0];
    double Z = U[1];
    double R = U[2];
    J(0, 0) = Pi - delta - beta(t) * Z;
    J(0, 1) = -beta(t) * S;
    J(0, 2) = 0.;
    J(1, 0) = (beta(t) - alpha(t)) * Z;
    J(1, 1) = (beta(t) - alpha(t)) * S;
    J(1, 2) = zeta(t);
    J(2, 0) = delta + alpha(t) * Z;
    J(2, 1) = alpha(t) * S;
    J(2, 2) = -zeta(t);
    //// CMEA_END_TEMPLATE
}
```

2e)

(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 `zombieOutbreak.computeF(YOUR ARGUMENTS HERE)` for this. `zombieOutbreak` is an object of class `ZombieOutbreak` 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 $u_i(t^n)$, and `time[n]` must be $n\Delta t$, for $n \in \{0, 1, \dots, N\}$ and $i \in \{1, 2, 3\}$.

Hint: Mind capitalization! `ZombieOutbreak` is a class, and `zombieOutbreak` 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, `ZombieOutbreak.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 6; see `code/zombie_dirk/dirksolver.hpp` for a full solution.

Listing 4: G_1

```
void computeG1(Eigen::Vector3d& G, Eigen::Vector3d y, double tn,
    Eigen::Vector3d Un, double dt) {
    //// CMEA_START_TEMPLATE
    Eigen::Vector3d Fy;
    zombieOutbreak.computeF(Fy, tn + mu * dt, y);
    G = Un + dt * mu * Fy - y;
    //// CMEA_END_TEMPLATE
}
```

Listing 5: Newton method for G_2

```
void newtonSolveY2(Eigen::Vector3d& v, Eigen::Vector3d Un,
    Eigen::Vector3d y1, double dt, double tn, double tolerance, int
        ↪ maxIterations) {

       // Use newtonSolveY1 as a model for this
    //// CMEA_START_TEMPLATE
    Eigen::Vector3d RHSG2, Fv, Fy, x;
    Eigen::Matrix3d JG2, JFv;
    v = Un;

    for (int iteration = 0; iteration < maxIterations; ++iteration) {
        zombieOutbreak.computeJF(JFv, tn + (mu - nu)*dt, v);
        zombieOutbreak.computeF(Fy, tn + mu * dt, y1);
        zombieOutbreak.computeF(Fv, tn + (mu - nu)*dt, v);
        Eigen::Matrix3d JG2 = dt * mu * JFv - Eigen::Matrix3d::Identity();
        Eigen::Vector3d RHSG2;// = Un - dt * nu * Fy + dt * mu * Fv - v;
        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");
    //// CMEA_END_TEMPLATE
}
```

Listing 6: 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] and time[n]

    //// CMEA_START_TEMPLATE
    Eigen::Vector3d Fy1, Fy2;

    for (int i = 1; i < N + 1; ++i) {
        double tn = time[i - 1];
        Eigen::Vector3d uPrevious(u[0][i - 1], u[1][i - 1], u[2][i - 1]);
        Eigen::Vector3d y1, y2;
        newtonSolveY1(y1, uPrevious, dt, time[i - 1], 1e-10, 100);
        newtonSolveY2(y2, uPrevious, y1, dt, time[i - 1], 1e-10, 100);
        zombieOutbreak.computeF(Fy1, tn + mu * dt, y1);
        zombieOutbreak.computeF(Fy2, tn + (mu - nu)*dt, y2);
        Eigen::Vector3d uNext = uPrevious + (dt / 2.) * (Fy1 + Fy2);

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

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

    //// CMEA_END_TEMPLATE
}
```

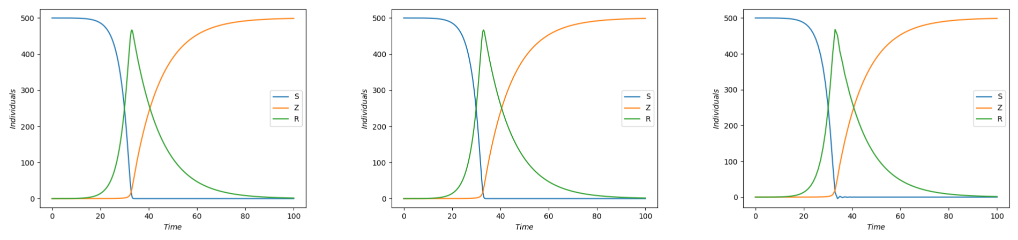
2f)

Use your function `dirk` to compute the solution up to $T = 101$ for the following number of timesteps:

$$\begin{aligned} N_1 &= 1000 \\ N_2 &= 500 \\ N_3 &= 100 \\ N_4 &= 10. \end{aligned}$$

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

Solution: We obtain the plots in Figure 5. We observe that the results approximate $\mathbf{u}$ well even for N as low as 100. Remember that we needed several thousands with forward Euler!



(a) $N = 1000$. **(b)** $N = 500$. **(c)** $N = 100$.

Figure 5: Solution computed for (7) with DIRK.

2g)

(Core problem) We want to study the convergence of the DIRK(2,3) scheme 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 $\zeta \equiv \alpha \equiv \beta \equiv 0$; and initial condition $(S_0, 0, 0)$. This means that we start with only humans, corpses don't return to life, and no one can become a zombie; i.e. the real-world scenario[1]. Therefore, we just have normal exponential growth for S, and thus for R, through natural mortality rate. Writing it formally,

[1]so far!

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

with $S_0 = 500$ and $T = 101$. 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,

$$\sum_{i=1}^{3} |u_i(T) - (u_i)_N|.$$

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?

Hint: $u_i(T)$ is already computed as `exact`.

Solution: Please see the results for the next exercise.

2h)

(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. 6 and Listing 7, with time measured in nanoseconds. In particular, observe that error scales with third order with respect to runtime, and runtime is linear with number of cells. And thus error has to be third order with respect to runtime as well. Informally, if we denote R runtime, N number of cells, and e error, then:

$$e = O(N^{-3}),\ R = O(N),\ \Rightarrow e = O(R^{-3})$$

This is precisely what we observe.

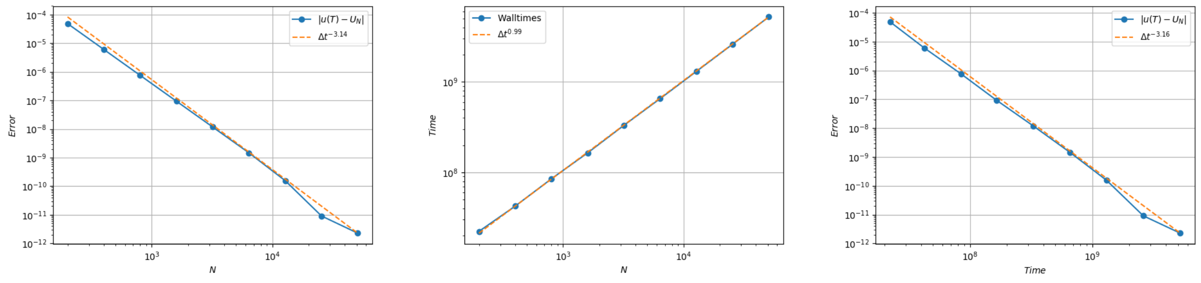


Figure 6: Plots for subproblem **2g)**. Left: L^1 error vs N, center: walltime vs N, right: L^1 error vs walltime.

Listing 7: Convergence study.

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

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

   // Compute the exact solution for the parameters above
   std::vector<double> exact = outbreak.computeExactNoZombies(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(3);
   int baseN = 200;

   for (int i = 0; i < countExponents; i++) {
      int N = (1 << (i + minExp) ) * baseN; // (1<<j) = pow(2,j)
      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[0][0] = u0[0];
        u[1][0] = u0[1];
        u[2][0] = u0[2];

        auto begin = std::chrono::high_resolution_clock::now();
        dirkSolver.solve(u, time, T, N);
        auto end = std::chrono::high_resolution_clock::now();
        errors[i] = std::abs(u[0].back() - exact[0]) + std::abs(
               u[1].back() - exact[1]) + std::abs(u[2].back() - exact[2]);
        std::cout << "Approx sol: " << u[0].back() << " " << u[1].back() << " " <<
           u[2].back() << 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);

}
```

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. 7 compares the results obtained with the above `dirkconv` 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^{-5}!

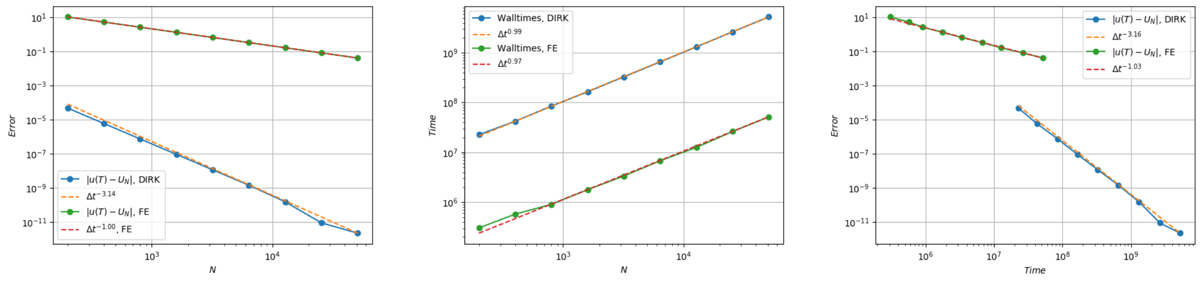


Figure 7: A comparison of forward Euler and DIRK(2,3). Left: L^1 error vs N, center: walltime vs N, right: L^1 error vs walltime.

2i)

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 8.

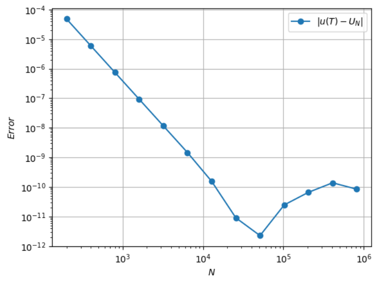


Figure 8: L^1 error vs N, N up to 819200

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.

References

[1] Munz, Hudea, Imad, Smith?, *When zombies attack!: mathematical modelling of an outbreak of zombie infection*, 2009

Happy Halloween!