# Project 1

ETH zürich

Computational Methods for
Engineering Applications
**Last edited:** October 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 all three 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 $\Delta t$ 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 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.

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 $\Delta t$, in the template called `dt`.
- The final time $T$, which we assume to be a multiple of $\Delta t$.

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 **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.

## 1h)

According to the discussion in subproblem **2e)**, 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 $\Delta t_k$, $k = 1, \ldots, 8$. Which rate of convergence do you observe?

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

## 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 $\delta$). 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 $\zeta$).
>
> Susceptibles can become zombies through transmission via an encounter with a zombie (transmission parameter $\beta$). 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 [...] $\Pi$. 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 $\alpha$). We also assume that zombies do not attack/defeat other zombies.

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

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

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 $\alpha$ and decreasing $\beta$ represent the improving ability of humans to fight off the undead, and survival of the fittest; $\zeta$ 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 $\alpha$ and $\beta$ have units in (thousand individuals)$^{-1}$day$^{-1}$, and $\Pi$, $\delta$ and $\zeta$ 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 poblem 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
```

## 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 $\Delta t$.

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

## 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.

## 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`.

## 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 $\alpha, \beta$, 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 $n \in \{0, 1, \ldots, N\}$ 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).

## 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?

## 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,

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

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 $i \in \{0, 1, \dots, 8\}$.

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.

[1] so far!

Which rate of convergence do you observe?

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

### 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.

### 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 1.

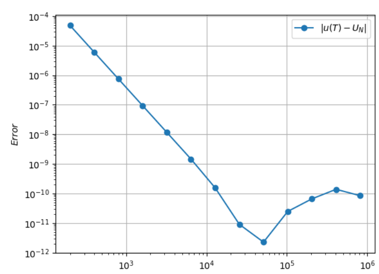


**Figure 1:** $L^1$ error vs $N$, $N$ up to 819200

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

## References


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


## Happy Halloween!