NumCSE exercise sheet 6
Initial Value Problems and Non-Linear Equations

soumil.gurjar@sam.math.ethz.ch
oliver.rietmann@sam.math.ethz.ch

December 18, 2018

Exercise 6.1. *Explicit vs. Implicit Time Stepping.*

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

$$\ddot{x} + 2\zeta\dot{x} + x = 0, \quad t \in (0, T), \tag{1}$$

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

$$x(0) = x_0,\ \dot{x}(0) = v_0. \tag{2}$$

(a) Equations (1) and (2) can be rewritten as a linear system of first order differential equations with appropriate initial conditions, i.e.:

$$\dot{\mathbf{y}} = \mathbf{A}\mathbf{y} \tag{3}$$

$$\mathbf{y}(0) = \mathbf{y}_0. \tag{4}$$

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

Solution: Setting $y_1 = x$ and $y_2 = \dot{x}$, we get that (1) and (2) are equivalent to the following linear system:

$$\begin{aligned} \dot{y}_2 + 2\zeta y_2 + y_1 &= 0, && t \in (0, T), \\ \dot{y}_1 &= y_2, && t \in (0, T), \\ y_1(0) &= x_0, \\ y_2(0) &= v_0. \end{aligned}$$

In matrix form, we can write the system as in (3) with $\mathbf{y} = (x, \dot{x})^\top$, $\mathbf{y}_0 = (x_0, v_0)^\top$ and

$$\mathbf{A} = \begin{pmatrix} 0 & 1 \\ -1 & -2\zeta \end{pmatrix}.$$

(b) Compute the solution to (1) with initial conditions given by (2) with

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

Hint: Diagonalizing $\mathbf{A}$ is the key to finding an analytic solution. Recall that $\zeta < 1$ and that, for a real number α, it holds that $e^{\alpha i} = \cos\alpha + i \sin\alpha$, where $i = \sqrt{-1}$ denotes the imaginary unit.

Solution: First, note that $\mathbf{A}$ is diagonalizable. Indeed:

$$\begin{vmatrix} -\lambda & 1 \\ -1 & -2\zeta - \lambda \end{vmatrix} = \lambda^2 + 2\zeta\lambda + 1.$$

Thus $\mathbf{A}$ has two eigenvalues (complex, as $\zeta < 1$) given by

$$\lambda_1 = -\zeta + \sqrt{\zeta^2 - 1}, \quad \lambda_2 = -\zeta - \sqrt{\zeta^2 - 1}.$$

The associated eigenvectors are

$$\mathbf{v}_1 = s_1(1, \lambda_1)^\top,\ s_1 \in \mathbb{R}, \quad \mathbf{v}_2 = s_2(1, \lambda_2)^\top,\ s_2 \in \mathbb{R}.$$

Then $\mathbf{A} = \mathbf{R}\mathbf{\Lambda}\mathbf{R}^{-1}$, with $\mathbf{R} = (v_1|v_2)$ the matrix having as columns an eigenbasis and $\mathbf{\Lambda} = \text{diag}(\lambda_1, \lambda_2)$. Note that, since $\mathbf{A}$ is *not* symmetric, the eigenvectors associated to distinct eigenvalues are not necessarily orthogonal. We have:

$$\mathbf{\Lambda} = \begin{pmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{pmatrix}, \quad \mathbf{R} = \begin{pmatrix} 1 & 1 \\ \lambda_1 & \lambda_2 \end{pmatrix}, \quad \mathbf{R}^{-1} = \frac{1}{\lambda_2 - \lambda_1} \begin{pmatrix} \lambda_2 & -1 \\ -\lambda_1 & 1 \end{pmatrix}$$

Denote $\mathbf{w}(t) = \mathbf{R}^{-1}\mathbf{y}(t)$. Then immediately:

$$\mathbf{y}'(t) = \mathbf{A}\mathbf{y}(t) = \mathbf{R}\mathbf{\Lambda}\mathbf{R}^{-1}\mathbf{y}(t) \iff$$
$$\mathbf{R}^{-1}\mathbf{y}'(t) = \mathbf{\Lambda}\mathbf{R}^{-1}\mathbf{y}(t) \iff \mathbf{w}'(t) = \mathbf{\Lambda}\mathbf{w}(t)$$

Now observe that this defines two decoupled ODEs, with immediate solutions:

$$w_1'(t) = \lambda_1 w_1(t) \Rightarrow w_1(t) = C_1 e^{\lambda_1 t}$$
$$w_2'(t) = \lambda_1 w_2(t) \Rightarrow w_2(t) = C_2 e^{\lambda_2 t}$$

And then we have a solution

$$\mathbf{y}(t) = R\mathbf{w}(t) = \begin{pmatrix} 1 & 1 \\ \lambda_1 & \lambda_2 \end{pmatrix} \begin{pmatrix} C_1 e^{\lambda_1 t} \\ C_2 e^{\lambda_2 t} \end{pmatrix}$$

where $C_1, C_2 \in \mathbb{R}$ are such that the initial conditions are verified; i.e., for $t = 0$,

$$\mathbf{y}(0) = R\mathbf{w}(0) \Rightarrow \begin{pmatrix} x_0 \\ v_0 \end{pmatrix} = R \begin{pmatrix} C_1 \\ C_2 \end{pmatrix},$$

from which, using the expression for $\mathbf{R}^{-1}$: $C_1 = \frac{\lambda_2}{\lambda_2 - \lambda_1}$, $C_2 = \frac{-\lambda_1}{\lambda_2 - \lambda_1}$.

Remark 1: this is the only point at which we have used the explicit expression for $\mathbf{R}^{-1}$. For a large problem, one would typically solve this with a fast linear solver, e.g. with LU factorization, and never explicitly invert $\mathbf{R}$.

Remark 2: Remember that we are asked for a solution for $x : \mathbb{R}^+ \to \mathbb{R}$; our answer must be only the first component (i.e. x) of $\mathbf{y}$. So finally, our solution is:

$$x(t) = \frac{\lambda_2}{\lambda_2 - \lambda_1} e^{\lambda_1 t} - \frac{\lambda_1}{\lambda_2 - \lambda_1} e^{\lambda_2 t}$$

Finally, using that for $\alpha \in \mathbb{R}$, $e^{i\alpha} = \cos\alpha + i \sin\alpha$, and after some operations, we get to the solution:

$$x(t) = e^{-\zeta t} \left(\frac{\zeta}{K} \sin(Kt) + \cos(Kt) \right), \quad K := \sqrt{1 - \zeta^2}$$

(c) Recall the explicit Euler timestepping introduced in the lecture. Using `template_harmonic_oscill.cpp`, implement the function `explicitEuler` to compute the solution $\mathbf{y} = \mathbf{y}(t)$ to (3) up to the time $T > 0$. The function should take as input the following parameters:

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

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

Solution:

```cpp
void explicitEuler(std::vector<double> & y1, std::vector<double> &y2, std::vector<
    double> & time,
     const Eigen::Vector2d& y0,
            double zeta, double h, double T) {


  const unsigned int nsteps = T/h;
  y1.resize(nsteps+1);
  y2.resize(nsteps+1);
  time.resize(nsteps+1);
  Eigen::Vector2d yold;
  Eigen::Vector2d ynew;
  /* Initialize A */
  Eigen::Matrix2d A;
  A << 0,1,-1,-2*zeta;
  /* Initialize y */
  yold[0]=y0[0];
  yold[1]=y0[1];
  y1[0]=y0[0];
  y2[0]=y0[1];
  time[0]=0.;
  for(unsigned i=0; i<nsteps; i++)
    {
      Eigen::Matrix2d B=Eigen::MatrixXd::Identity(2,2)+h*A;
      ynew=B*yold;
      y1[i+1]=ynew[0];
      y2[i+1]=ynew[1];
      yold=ynew;
      time[i+1]=(i+1)*h;
    }

}
```

harmonic_oscill.cpp

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

Solution:

```
void implicitEuler(std::vector<double> & y1, std::vector<double> & y2, std::vector<
    double> & time,
     const Eigen::Vector2d& y0,
            double zeta, double h, double T) {

  const unsigned int nsteps = T/h;
  y1.resize(nsteps+1);
  y2.resize(nsteps+1);
  time.resize(nsteps+1);
  Eigen::Vector2d yold;
  Eigen::Vector2d ynew;
  /* Initialize A */
  Eigen::Matrix2d A;
  A << 0,1,-1,-2*zeta;
  /* Initialize y */
  yold[0]=y0[0];
  yold[1]=y0[1];
  y1[0]=y0[0];
  y2[0]=y0[1];
  time[0]=0.;
  for(unsigned i=0; i<nsteps; i++)
    {
      Eigen::Matrix2d B=Eigen::MatrixXd::Identity(2,2)-h*A;
      ynew=B.fullPivLu().solve(yold);
      y1[i+1]=ynew[0];
      y2[i+1]=ynew[1];
      yold=ynew;
      time[i+1]=(i+1)*h;
    }

}
```

harmonic_oscill.cpp

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

Solution:

```
void Energy(const std::vector<double> & v, std::vector<double> & energy)
{
  assert(v.size()==energy.size());

  for(unsigned i=0;i<v.size();i++)
    energy[i]=0.5*std::pow(v[i],2);

}
```

harmonic_oscill.cpp

(f) We consider two time steps $h_1 = 0.1$ and $h_2 = 0.5$. We choose $T = 20$, $\zeta = 0.2$.

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

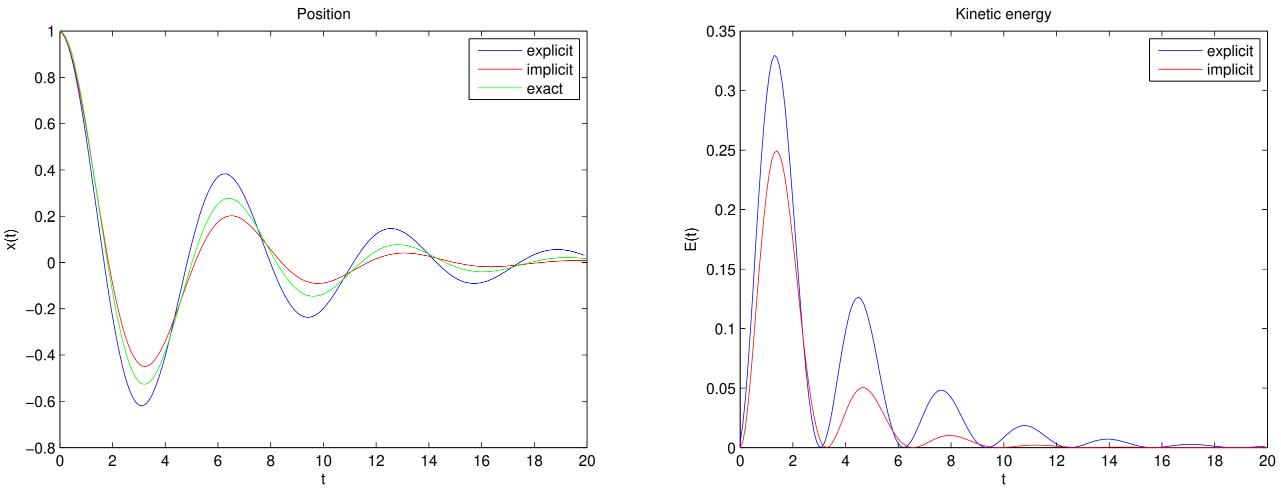


Figure 1: Plot of position and energy versus time for $h_1 = 0.1$.

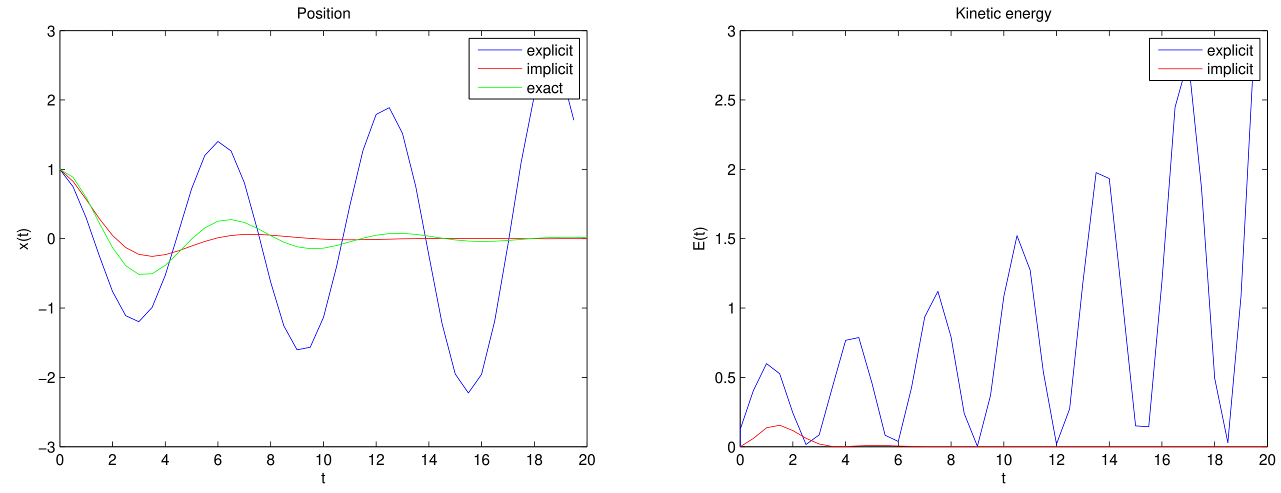


Figure 2: Plot of position and energy versus time for $h_2 = 0.5$.

Solution: See Figures 1 and 2 for the plots. As we can see, using $h_2 = 0.5$ the energy of the system explodes, which is not physical. For $h_1 = 0.1$, instead, the energy remains bounded and both schemes give an approximation to the exact solution. The reason is that for explicit timestepping schemes to be stable, the time step cannot be arbitrarily large and must be inside the so-called *stability* region. Implicit schemes, instead, are often unconditionally stable, and so the implicit Euler scheme worked for both timestep choices.

Exercise 6.2. *Heun's method.*

Heun's method is the Runge-Kutta method defined by the *Butcher tableau*

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

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

Solution: It is an explicit scheme since the coefficient matrix $(a_{i,j})$ is strictly lower triangular.

(b) Fix $t_0, T > 0$ and $y_0 \in \mathbb{R}$ and let $f : [t_0, T] \times \mathbb{R} \to \mathbb{R}$. Moreover, let $y : [t_0, T] \to \mathbb{R}$ solve the ODE

$$\begin{aligned} y'(t) &= f(t, y(t)), \quad t \in [t_0, T], \\ y(t_0) &= y_0. \end{aligned} \tag{5}$$

Fix $N \in \mathbb{N}$ and consider the time grid $t_0 < t_1 < \ldots < t_N = T$. For simplicity, we assume equidistant time steps, i.e. $h := t_{k+1} - t_k$ is the same for all $k \in \{0, \ldots, N-1\}$. Let $y_1, \ldots, y_N \in \mathbb{R}$ denote the approximations $y_k \approx y(t_k)$ according to Heun's method. For fixed $k \in \{0, \ldots, N-1\}$, give an expression to compute y_{k+1} from t_k, t_{k+1} and y_k.

Solution: Fix $k \in \{0, \ldots, N-1\}$ and let $h = t_{k+1} - t_k$ denote the (constant) time step. The Butcher tableau then yields

$$\begin{aligned} k_1 &:= f(t_k, y_k), \\ k_2 &:= f(t_k + h, y_k + hk_1), \\ y_{k+1} &= y_k + h\left(\frac{1}{2}k_1 + \frac{1}{2}k_2\right). \end{aligned}$$

(c) We keep the previous task. Implement a function

```
std::vector<double> Heun(const std::function<double(double, double)> &f,
                         const std::vector<double> &t, double y0);
```

that takes the quantities f and y_0 in Equation (5), and a time grid $t_0 < t_1 < \ldots < t_N$. It shall return a vector containing the approximations $y_0, \ldots, y_N$.

Solution:

```
std::vector<double> Heun(const std::function<double(double, double)> &f, const std::
    vector<double> &t, double y0) {
    int n = t.size();
    std::vector<double> y(n);
    y[0] = y0;

    for(int i = 0; i < n - 1; ++i) {
        double h = t[i + 1] - t[i];
        double k1 = f(t[i], y[i]);
        double k2 = f(t[i] + h, y[i] + h * k1);
        y[i + 1] = y[i] + h * 0.5 * (k1 + k2);
    }

    return y;
}
```

heun.cpp

(d) 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.

(e) Compute the stability function $S : \mathbb{C} \to \mathbb{C}$ and the stability region of Heun's method.

Solution: According to the Butcher tableau, the stability function is given by

$$S(z) = 1 + z \begin{pmatrix} \frac{1}{2} & \frac{1}{2} \end{pmatrix} \begin{pmatrix} 1 & 0 \\ -z & 1 \end{pmatrix}^{-1} \begin{pmatrix} 1 \\ 1 \end{pmatrix} = \frac{1}{2}z^2 + z + 1$$

for all $z \in \mathbb{C}$. The stability region is therefore

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

(f) From now on, we consider the special case of Equation (5) given by

$$\begin{aligned} y'(t) &= e^{-2t} - 2y(t), \quad t \in [0, T], \\ y(0) &= y_0. \end{aligned} \tag{6}$$

Which is the biggest time step $h > 0$ for which Heun's method is stable for the ODE (6)?

Hint: Consider the homogeneous version of (6) and use the stability region computed in the previous subproblem.

Solution: From Equation (6), we can see that the associated homogeneous equation has the eigenvalue $\lambda := -2$. We have stability if $z := \lambda h$ lies in the stability region. By the previous task, this holds if and only if $h < 1$.

(g) Compute the solution of Equation (6) on $[0, 1]$ (i.e. $T = 1$) using Heun's method with step size $h_j := 2^{-(j+2)}$ for $j = 0, 1, 2, 3$. In all four cases, compute the error to the exact solution $y(t) = te^{-2t}$ at time $t = 1$. Use this data to determine the order of convergence (e.g. by a log-log plot).

Solution: We obtain (at least) second order convergence (see plot below).

```
    std::vector<int> n = { 4, 8, 16, 32 };
    int J = n.size();
    std::vector<double> h(J);
    std::vector<double> error(J);
    for (int j = 0; j < J; ++j) {
        h[j] = 1. / n[j];
        std::vector<double> t = LinSpace(n[j] + 1, .0, T);
        std::vector<double> y = Heun(f, t, .0);
        error[j] = std::abs(Y(t.back()) - y.back());
    }
```

heun.cpp

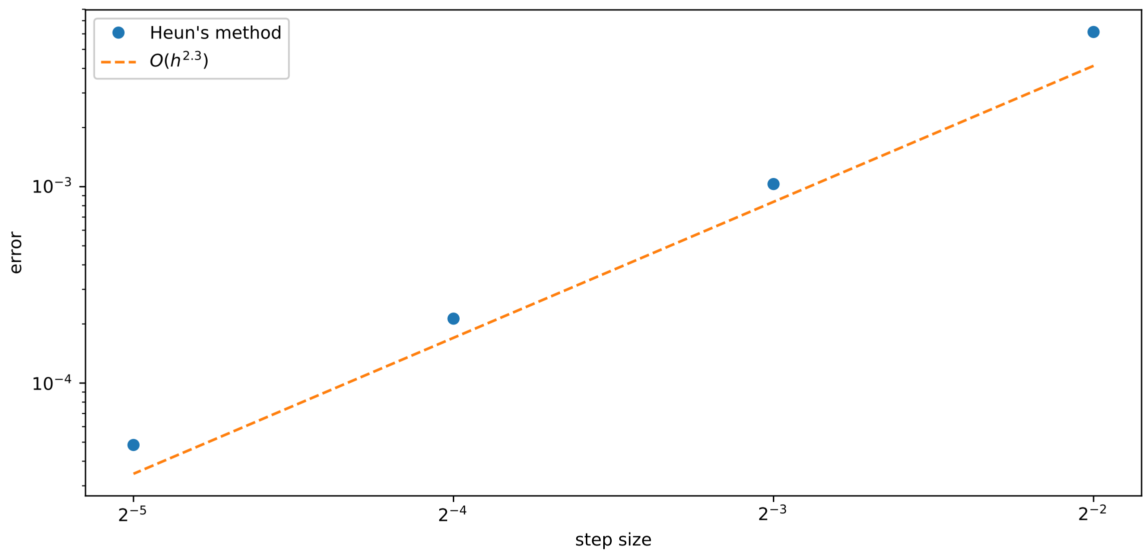


Figure 3: Heun's method is of second order.

Exercise 6.3. *Strong Stability Preserving RK3 Method for Time Stepping.*

In this exercise, we consider the Strong Stability Preserving, Runge-Kutta 3 method for time stepping, also known as SSPRK3, or the Shu-Osher method. The *Butcher tableau* for this scheme is:

$$\begin{array}{c|ccc} 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ \frac{1}{2} & \frac{1}{4} & \frac{1}{4} & 0 \\ \hline & \frac{1}{6} & \frac{1}{6} & \frac{2}{3} \end{array} \tag{7}$$

(a) Is the SSPRK3 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 the coefficient matrix $(a_{i,j})_{i,j}$ is such that $a_{i,j} = 0$ for $i \leq j$; i.e., the diagonal and above are all zeros.

(b) Consider the scalar ODE

$$u'(t) = f(t, u), \quad t \in (0, T), \tag{8}$$

for some $T > 0$.

Let us denote the time step by h and the time levels by $t^n = nh$ for $n = 0, 1, 2, \ldots, \frac{T}{h}$. Formulate explicitly the SSPRK3 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 $n = 0, 1, 2, \ldots, \frac{T}{h} - 1$.

Solution:

According to the Butcher tableau, the system reads:

$$\begin{aligned} k_1 &= f(t^n, u_n), \\ k_2 &= f(t^n + h, u_n + hk_1) \\ k_3 &= f\left(t^n + \frac{1}{2}h, u_n + \frac{1}{4}h(k_1 + k_2)\right), \\ u_{n+1} &= u_n + h\left(\frac{1}{6}k_1 + \frac{1}{6}k_2 + \frac{2}{3}k_3\right), \end{aligned}$$

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

(c) Show that SSPRK3 is *consistent*, and that it is *at least* third order accurate (i.e. its one-step error is fourth order).

Hint: Proving second and third order accuracy of an explicit, consistent, s-stage RK method by hand (using Taylor) is tedious; a simpler criteria is to check if the following holds:

$$\sum_{j=1}^{s} b_j c_j = \frac{1}{2} \text{ (for second order);}$$

$$\sum_{j=1}^{s} b_j c_j^2 = \frac{1}{3} \quad \text{and} \quad \sum_{j=1}^{s} \sum_{i=1}^{s} b_i a_{ij} c_j = \frac{1}{6} \text{ (for third order).}$$

Solution: **Consistent**: We have to check that $\sum_{j=1}^{s} a_{i,j} = \sum_{j=1}^{3} a_{i,j} = c_i$ for $i = 1, 2, 3$, and $\sum_{j=1}^{s} b_j = \sum_{j=1}^{3} b_j = 1$. From the Butcher tableau, it is trivial to see that these equalities hold and thus the method is consistent.

Third order: For an explicit, consistent, 3-stage RK method with the given Butcher tableau, these conditions reduce to:

$$\begin{aligned} b_2c_2 + b_3c_3 &= \frac{1}{2} \text{ (second order);} \\ b_2c_2^2 + b_3c_3^2 &= \frac{1}{3} \text{ and} \\ c_2a_{32}b_3 &= \frac{1}{6} \text{ (third order)} \end{aligned}$$

These can be immediately checked to hold. In fact, this bound is sharp - i.e. RK3 is an **exactly** third-order accurate method.

(d) 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. One approach to determine stability of a method is *absolute stability*. Recall that in order to study the absolute stability of a method, one considers the numerical method applied to the ODE

$$u'(t) = \lambda u(t), \quad t \in (0, +\infty), \tag{9}$$
$$u(0) = 0, \tag{10}$$

for $\lambda \in \mathbb{R}^-$.

Determine the inequality that the quantity $w := \lambda h$ has to satisfy so that SSPRK3 is absolutely stable.

Solution: SSPRK3 applied to equation $u'(t) = \lambda u(t),\ t \in (0, +\infty)$ gives, for a generic step $n \in \mathbb{N}$:

$$\begin{aligned} k_1 &= \lambda u_n \\ k_2 &= \lambda(1 + h\lambda)u_n \\ k_3 &= \lambda(1 + \frac{h\lambda}{2} + \frac{(h\lambda)^2}{4})u_n \\ u_{n+1} &= u_n + h\left(\frac{1}{6}\lambda u_n + \frac{1}{6}\lambda(1 + h\lambda)u_n + \frac{2}{3}\lambda(1 + \frac{h\lambda}{2} + \frac{(h\lambda)^2}{4})u_n\right) \end{aligned}$$

Which, using $w := \lambda h$ and operating, we can write as:

$$u_{n+1} = \left(1 + w + \frac{w^2}{2} + \frac{w^3}{6}\right)u_n$$

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

$$\left\{w \in \mathbb{C} : \left|1 + w + \frac{w^2}{2} + \frac{w^3}{6}\right| < 1\right\}.$$

Restricting to real values, the polynomial $p(w) = 1 + w + \frac{w^2}{2} + \frac{w^3}{6}$ can be easily checked to be strictly increasing, with value 1 at $w = 0$. Using a numerical solver, we can find that it takes the value -1 at $-\hat{w} \approx -2.51$. Thus, SSPRK3 is stable for $w \in [-\hat{w}, 0]$.

(e) From now on, we consider a particular case of (8), with some initial conditions. Namely, we take

$$u'(t) = e^{-2t} - 2u(t), \quad t \in (0, T), \tag{11}$$
$$u(0) = u_0. \tag{12}$$

Complete the template file `template_ssprk3.cpp` provided in the handout, implementing the function `SSPRK3` to compute the solution to (11) up to the time $T > 0$. The input arguments are:

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

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

Compute the solution to (11) at each time-step for $u_0 = 0$, $h = 0.2$ and $T = 10$ and plot it using the matlab or python script that has been provided.

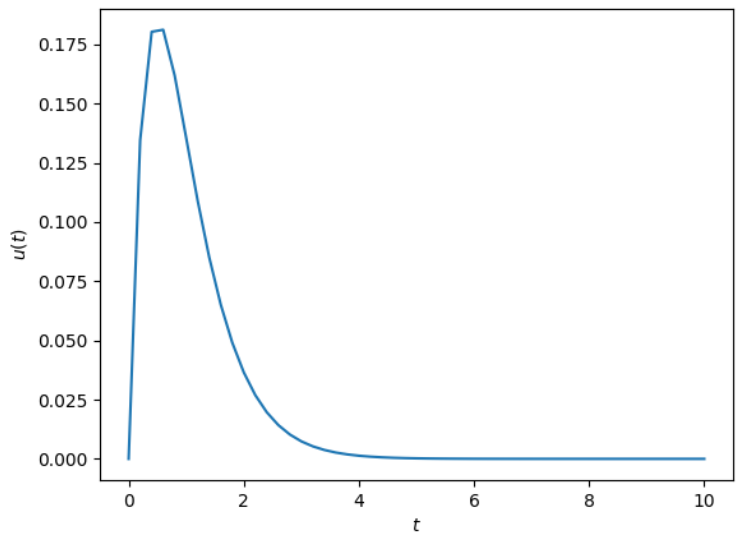
Solution:

```cpp
double f(double t, double u) {
    return std::exp(-2*t) - 2*u;
}
void SSPRK3(std::vector<double> & u, std::vector<double> & time,
          const double & u0, double dt, double T) {
    const unsigned int nsteps = std::round(T/dt);
    u.resize(nsteps+1);
    time.resize(nsteps+1);

    // Write your SSPRK3 code here
    time[0] = 0.;
    u[0] = u0;

    for(unsigned int i = 0; i < nsteps; i++)
    {
        time[i+1] = (i+1)*dt;
        double k1 = f(time[i], u[i]);
        double k2 = f(time[i+1], u[i] + dt*k1);
    double k3 = f(time[i] + 0.5*dt, u[i] + 0.25*dt*(k1 + k2));
        u[i+1] = u[i] + dt*( (1./6)*(k1 + k2) + (2./3)*k3);
    }
}
```

ssprk3.cpp



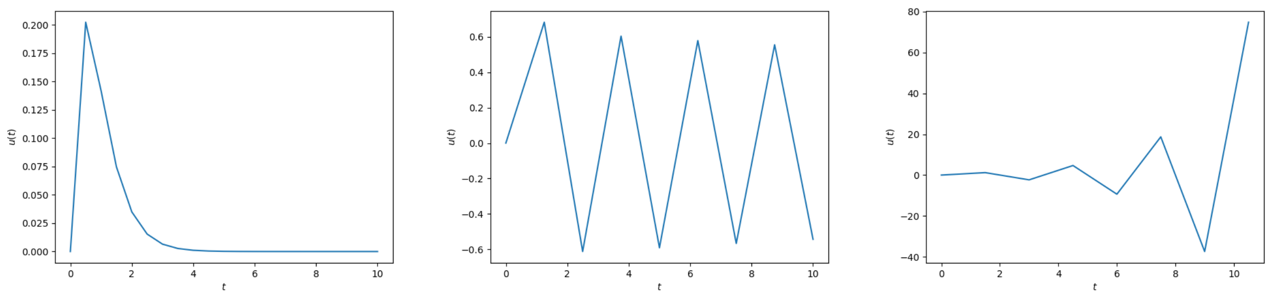
Plot for subproblem (e)

(f) According to the discussion in subproblem (d), which is the biggest timestep $h > 0$ for which SSPRK3 is stable for problem (11)?

Hint: Consider the homogeneous version of (11) and use the stability region computed in (d).

Solution: From equation (11), we can see that the associated homogeneous equation $u'(t) = -2u(t)$ has the eigenvalue $\lambda = -2$. We have to ensure that $|\lambda h| \in [0, \hat{w}]$, and thus the maximum timestep for which we still have stability is $h = \frac{\hat{w}}{2} \approx 1.25$.

Indeed, the figure below shows that for $h = 0.5$, 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 $h = 1.25$, the method is still stable, so the solution remains bounded as $t \to \infty$, although it is oscillatory. For bigger h, instead, the numerical solution is unstable and grows unbounded as $t \to \infty$, that is $\frac{|u_{n+1}|}{|u_n|} > 1$ for n big.



Plots for subproblem f. Left: $h = 0.5$, center: $h = 1.25$, right: $h = 1.5$.

(g) Create a copy of the completed file `template_ssprk3.cpp` and name it `template_ssprk3conv.cpp`. Modify the `main()` function to perform a convergence study for the solution to (11) computed using SSPRK3, with $u_0 = 0$ and $T = 10$. More precisely, consider the sequence of timesteps $h_k = 2^{-k}$, $k = 1, \dots, 8$, and for each of them, compute the numerical solution $u_{\frac{T}{h_k}} \approx u(T)$ and the error $|u_{\frac{T}{h_k}} - u(T)|$, where u denotes the exact solution to (11). Produce a double logarithmic plot of the error versus h_k, $k = 1, \dots, 8$ using the matlab or python script that has been provided. What rate of convergence do you observe?

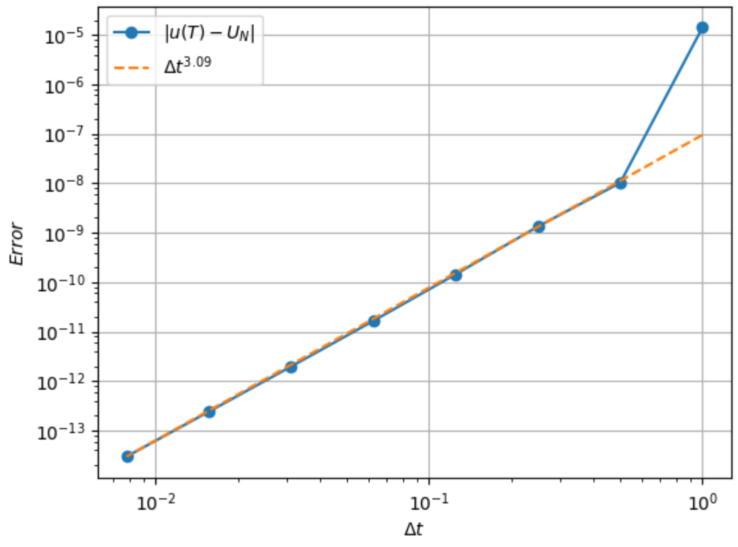
Hint: The exact solution to (11) is $u(t) = (t + u_0)e^{-2t}$, $t \in [0, T]$.

Solution: See code listing and the following figure. To estimate 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 pre-asymptotic regime.) This results in ≈ 3.09, as expected from subproblem (c).

```cpp
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);
        std::vector<double> time;
        std::vector<double> u;
        SSPRK3(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;
}
```

ssprk3conv.cpp



Convergence plot for subproblem (g).

Exercise 6.4. *Newton's method.*

Consider the following system of non-linear equations

$$\begin{aligned} f_0(x_0, x_1) &:= x_0^2 + 2x_1^2 - 1 = 0, \\ f_1(x_0, x_1) &:= x_1 - x_0^2 = 0, \end{aligned} \tag{13}$$

where $x_0, x_1 \in \mathbb{R}$. As shown in Figure 4 below, this system admits exactly two real solutions. One easily verifies that they are given by

$$x_0 = \pm\frac{1}{\sqrt{2}} \quad \text{and} \quad x_1 = \frac{1}{2}.$$

Our goal is to find these solutions numerically using *Newton's method.*

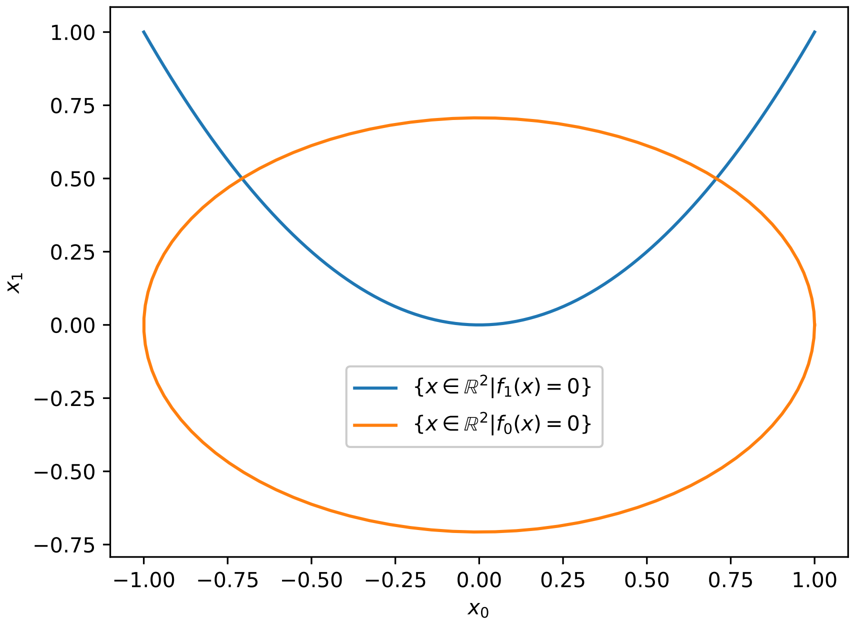


Figure 4: The real solutions are exactly the intersection points of these two curves.

(a) Write $x = (x_0, x_1)$ and let $f : \mathbb{R}^2 \to \mathbb{R}^2$ satisfy

$$f(x) = \begin{pmatrix} f_0(x_0, x_1) \\ f_1(x_0, x_1) \end{pmatrix}$$

for all $x_0, x_1 \in \mathbb{R}$. Write a `C++/Eigen` function

```
Eigen::Matrix2d Jacobian(const Eigen::Vector2d &x);
```

that returns the Jacobian of f evaluated at $x \in \mathbb{R}^2$.

Solution:

```
Eigen::Matrix2d Jacobian(const Eigen::Vector2d &x) {
    Eigen::Matrix2d J;
    J << 2. * x(0), 4. * x(1), -2. * x(0), 1.;
    return J;
}
```

newton.cpp

(b) Fix $n \in \mathbb{N}$ and $x \in \mathbb{R}^2$. Write a `C++/Eigen` function

```
Eigen::Vector2d Newton(Eigen::Vector2d x, int n);
```

that returns the approximate solution to (13) obtained from n iterations of Newton's method with starting point x.

Solution:

```
Eigen::Vector2d Newton(Eigen::Vector2d x, int n) {
    for (int i = 0; i < n; ++i) {
        x += Jacobian(x).fullPivLu().solve(-f(x));
    }
    return x;
}
```

newton.cpp

(c) Apply Newton's method to Equation (13) with $n = 100$ iterations. Try the starting points $(-1, 1), (1, 1)$ and $(-2, -2)$ and explain the results.

Solution: Newton's method converges to a solution when started at $(-1, 1)$ or $(1, 1)$. However, it does not converge when started at $(-2, -2)$. We need to start sufficiently close to a solution to ensure convergence to the latter. Compare our starting points to Figure 4 above.