S. Mishra
K. O. Lye
L. Scarabosio

Autumn Term 2015

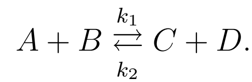
Computational Methods for Engineering Applications II

ETH Zürich
D-MATH

Homework Problem Sheet 1

Problem 1.1 Chemical concentrations

In this exercise we will study the evolution of chemical concentrations. Consider a chemical system consisting of four substances A, B, C and D. We have a reversible chemical reaction on the form



$$A + B \underset{k_2}{\overset{k_1}{\rightleftarrows}} C + D.$$

At time $t \geq 0$, we let $u_1(t), u_2(t), u_3(t)$ and $u_4(t)$ denote the concentration of A, B, C and D respectively. The kinetics are given by the following system

$$\begin{cases} u_1'(t) = -k_1 u_1(t) u_2(t) + k_2 u_3(t) u_4(t) \\ u_2'(t) = -k_1 u_1(t) u_2(t) + k_2 u_3(t) u_4(t) \\ u_3'(t) = -k_2 u_3(t) u_4(t) + k_1 u_1(t) u_2(t) \\ u_4'(t) = -k_2 u_3(t) u_4(t) + k_1 u_1(t) u_2(t) \end{cases} . \tag{1.1.1}$$

(1.1a) Write the system (1.1.1) in the form

$$\boldsymbol{u}'(t) = \vec{F}(\boldsymbol{u}(t)) \tag{1.1.2}$$

where $\boldsymbol{u} \in \mathbb{R}^4$ and $\vec{F} : \mathbb{R}^4 \to \mathbb{R}^4$.

Solution: We set

$$\boldsymbol{u}(t) = \begin{pmatrix} u_1(t) \\ u_2(t) \\ u_3(t) \\ u_4(t) \end{pmatrix},$$

and

$$\vec{F}(\boldsymbol{u}) = \begin{pmatrix} -k_1 u_1 u_2 + k_2 u_3 u_4 \\ -k_1 u_1 u_2 + k_2 u_3 u_4 \\ -k_2 u_3 u_4 + k_1 u_1 u_2 \\ -k_2 u_3 u_4 + k_1 u_1 u_2 \end{pmatrix},$$

then the system (1.1.1) turns into

$$\boldsymbol{u}'(t) = \vec{F}(\boldsymbol{u}(t)).$$

(1.1b) We will use the *trapezoidal rule* (or implicit second order Runge-Kutta) to solve the system (1.1.2) numerically. Recall that the trapezoidal rule is given as

$$\boldsymbol{v}_{n+1} = \boldsymbol{v}_n + \frac{\Delta t}{2}\Big(\vec{F}(\boldsymbol{v}_n) + \vec{F}(\boldsymbol{v}_{n+1})\Big) \qquad \text{for } n \geq 0, \tag{1.1.3}$$

$$\boldsymbol{v}_0 = \boldsymbol{u}_0, \tag{1.1.4}$$

where $\Delta t > 0$ is the step size.

Determine the truncation error of (1.1.3).

Solution: The truncation error is given as

$$\tau_n = \frac{\boldsymbol{u}(t^{n+1}) - \boldsymbol{u}(t^n)}{\Delta t} - \frac{1}{2}\Big(\vec{F}(\boldsymbol{u}(t^n)) + \vec{F}(\boldsymbol{u}(t^{n+1})\Big).$$

By (1.1.2) we have

$$\begin{aligned}\tau_n &= \frac{\boldsymbol{u}(t^{n+1}) - \boldsymbol{u}(t^n)}{\Delta t} - \frac{1}{2}\Big(\vec{F}(\boldsymbol{u}(t^n)) + \vec{F}(\boldsymbol{u}(t^{n+1})\Big)\\ &= \frac{\boldsymbol{u}(t^{n+1}) - \boldsymbol{u}(t^n)}{\Delta t} - \frac{1}{2}\big(\boldsymbol{u}'(t^n) + \boldsymbol{u}'(t^{n+1})\big).\end{aligned}$$

We insert the Taylor expansions

$$\boldsymbol{u}(t^{n+1}) = \boldsymbol{u}(t^n) + \boldsymbol{u}'(t^n)\Delta t + \frac{\boldsymbol{u}''(t^n)}{2}\Delta t^2 + \mathcal{O}(\Delta t^3),$$

and

$$\boldsymbol{u}'(t^{n+1}) = \boldsymbol{u}'(t^n) + \vec{u''}(t^n)\Delta t + \mathcal{O}(\Delta t^2),$$

to obtain

$$\begin{aligned}\tau_n &= \frac{\boldsymbol{u}(t^n) + \boldsymbol{u}'(t^n)\Delta t + \frac{\boldsymbol{u}''(t^n)}{2}\Delta t^2 + \mathcal{O}(\Delta t^3) - \boldsymbol{u}(t^n)}{\Delta t} - \frac{1}{2}\big(\boldsymbol{u}'(t^n)) + \boldsymbol{u}'(t^n) + \boldsymbol{u}''(t^n)\Delta t + \mathcal{O}(\Delta t^2)\big)\\ &= \mathcal{O}(\Delta t^3)/\Delta t - \frac{1}{2}\mathcal{O}(\Delta t^2)\\ &= \mathcal{O}(\Delta t^2).\end{aligned}$$

(1.1c) Notice that (1.1.3) is a non-linear equation in $\boldsymbol{v}_{n+1}$, therefore we will use the Newton method to solve for $\boldsymbol{v}_{n+1}$. Recall that the Newton method for solving

$$\vec{G}(\boldsymbol{x}) = 0$$

is formulated as

$$D\vec{G}(\boldsymbol{x}_k)\Delta\boldsymbol{x}_k = -\vec{G}(\boldsymbol{x}_k) \tag{1.1.5}$$

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \Delta\boldsymbol{x}_k \qquad k \geq 0, \tag{1.1.6}$$

where $\boldsymbol{x}_0$ is some initial guess. Write the Newton step for (1.1.3), solving for $\boldsymbol{v}_{n+1}$. What is $\vec{G}$ and $D\vec{G}$ in this case?

HINT: You should *not* compute $\left[D\vec{G}(\boldsymbol{x}_k)\right]^{-1}$ by hand.

Solution: We rewrite (1.1.3) into the following equation:

$$\boldsymbol{v}_n + \frac{\Delta t}{2}\left(\vec{F}(\boldsymbol{v}_n) + \vec{F}(\boldsymbol{v}_{n+1})\right) - \boldsymbol{v}_{n+1} = \mathbf{0},$$

so we put

$$\vec{G}(\boldsymbol{v}) = \boldsymbol{v}_n + \frac{\Delta t}{2}\left(\vec{F}(\boldsymbol{v}_n) + \vec{F}(\boldsymbol{v})\right) - \boldsymbol{v}$$

and we have

$$D\vec{G} = \frac{\Delta t}{2} D\vec{F} - \mathbf{I},$$

where $\mathbf{I}$ is the identity matrix, and

$$D\vec{F} = \begin{pmatrix} -k_1u_2 & -k_1u_1 & k_2u_4 & k_2u_3 \\ -k_1u_2 & -k_1u_1 & k_2u_4 & k_2u_3 \\ k_1u_2 & k_1u_1 & -k_2u_4 & -k_2u_3 \\ k_1u_2 & k_1u_1 & -k_2u_4 & -k_2u_3 \end{pmatrix}.$$

The Newton iteration is then given as

$$\begin{aligned} D\vec{G}(\boldsymbol{x}_k)\Delta\boldsymbol{x}_k &= -\vec{G}(\boldsymbol{x}_k) \\ \boldsymbol{x}_{k+1} &= \boldsymbol{x}_k + \Delta\boldsymbol{x}_k \qquad k \geq 0, \end{aligned}$$

(1.1d) Write a function in C++ that computes $\boldsymbol{v}_{n+1}$ using the Newton method. The function should take the following parameters:

- The previous value $\boldsymbol{v}_n$.
- An initial guess.
- The constants k_1 and k_2.
- The step size Δt.
- A tolerance for when to exit the Newton iteration.
- A maximum number of iterations to use for the Newton method.

See the function `newtonSolve` in `template_code1/exercise1/trapezoid.cpp` for a template.

HINT: We strongly suggest that you use the Eigen library for solving the linear system $D\vec{G}(\boldsymbol{x}_k)\Delta\boldsymbol{x}_k = -\vec{G}(\boldsymbol{x}_k)$. Refer to https://www2.math.ethz.ch/education/bachelor/lectures/hs2015/.. other/cmea2/series/eigen_example_dense.zip for an introduction to using Eigen. Further documentation on Eigen can be obtained on the Eigen homepage http://eigen.tuxfamily.org/, and in particular the section http://eigen.tuxfamily.org/dox/group__QuickRefPage.html.

HINT: Use $\|\vec{G}(\boldsymbol{x}_k)\|$ as a measure on the error.

Solution: See the listing 1.1.

(1.1e) Implement the trapezoidal scheme (1.1.3), using the Newton function from the previous exercise. Compute the approximate solution up to time T, using the following constants:

- $\boldsymbol{u}_0 = \begin{pmatrix} 0.2 & 0.3 & 0.1 & 0.4 \end{pmatrix}$
- $k_1 = 0.4$, $k_2 = 0.1$
- $T = 20$
- $\Delta t = T/1000$

Plot u_1, u_2, u_3 and u_4 as a function of time. Also plot the sum $u_1 + u_2 + u_3 + u_4$ as a function of time.

See the function `main` in `template_code1/exercise1/trapezoid.cpp` for a template.

Solution: See listing 1.1 for a solution, and `exercise1e_plot.py` provided in the handout for how to plot the solution (MATLAB version coming soon!).

We obtain the plots found in 1.1 and 1.2. Notice that the concentrations always sum to 1, which is what we should expect.

Listing 1.1: Implementation for the functions in subproblems (1.1d) and (1.1e).

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

///
/// Computes F(u) where F is as in the exercise
///
void computeF(Eigen::Vector4d& F, const Eigen::Vector4d& u, double
    k1, double k2) {
     F << -k1 * u[0] * u[1] + k2 * u[2] * u[3],
      -k1 * u[0] * u[1] + k2 * u[2] * u[3],
      k1 * u[0] * u[1] - k2 * u[2] * u[3],
      k1 * u[0] * u[1] - k2 * u[2] * u[3];
}

///
/// Computes DF and stores it to DF
///
void computeDF(Eigen::Matrix4d& DF, const Eigen::Vector4d& v,
    double k1, double k2) {
  DF << -k1 * v[1], -k1 * v[0], k2 * v[3], k2 * v[2],
    -k1 * v[1], -k1 * v[0], k2 * v[3], k2 * v[2],
    k1 * v[1], k1 * v[0], -k2 * v[3], -k2 * v[2],
    k1 * v[1], k1 * v[0], -k2 * v[3], -k2 * v[2];
```

```
26 }
27
28 ///
29 /// Uses the newton method to solve
30 ///
31 /// vNext = v + (h/2) [F(v) - F(vNext)]
32 ///
33 /// for vNext given the initial guess initialGuess.
34 ///
35 /// The results will be stored to vNext.
36 ///
37 /// @param[out] vNext at the end of the call, this will have vNext
38 /// @param[in] initialGuess the initial guess to use for newton
39 /// @param[in] v the previously obtained v (ie.  vn)
40 /// @param[in] k1 the reaction factor k1 (see exercise)
41 /// @param[in] k2 the reaction factor k2 (see exercise)
42 /// @param[in] dt the step size
43 /// @param[in] tolerance the limit on when to stop the newton
       iterations
44 /// @param[in] maxIterations maximum number of iterations to
       perform.
45 ///
46 void newtonSolve(Eigen::Vector4d& vNext,
47              const Eigen::Vector4d& initialGuess,
48              const Eigen::Vector4d& v,
49              double k1, double k2, double dt, double tolerance,
50              int maxIterations) {
51
52     // F(v), notice that we only have to compute Fv once
53     Eigen::Vector4d Fv;
54     computeF(Fv, v, k1, k2);
55
56     vNext = initialGuess;
57
58     for (int iteration = 0; iteration < maxIterations; ++iteration)
           {
59      // F(vNext)
60      Eigen::Vector4d FvNext;
61      computeF(FvNext, vNext, k1, k2);
62
63      // Compute the residual, this gives us the error.  (residual =
           G(u))
64      Eigen::Vector4d residual = v + (dt / 2.0) * (Fv + FvNext) -
           vNext;
65      if (residual.norm() <= tolerance) {
66          return;
67      }
68
69      // To store the jacobi matrix
70      Eigen::Matrix4d DF;
71      computeDF(DF, vNext, k1, k2);
```

```
72
73          Eigen::Matrix4d DG = 0.5 * dt * DF -
               Eigen::Matrix4d::Identity();
74
75          // Solves DG deltaX = G(u) for deltaX
76          Eigen::Vector4d deltaX = DG.lu().solve(residual);
77
78          vNext = vNext - deltaX;
79        }
80
81        throw std::runtime_error("Did not reach tolerance in Newton
             iteration");
82    }
83
84    int main() {
85
86        double T = 20.0;
87        size_t N = 1000;
88        double dt = T / N;
89
90        // This makes saving the results a lot easier.  u1[n] will hold
             u1 at timestep n
91        std::vector<double> u1(N+1), u2(N+1), u3(N+1), u4(N+1),
             time(N+1);
92
93        // Set initial start values
94        u1[0] = 0.2;
95        u2[0] = 0.3;
96        u3[0] = 0.1;
97        u4[0] = 0.4;
98        time[0] = 0;
99
100       // From the exercise
101       double k1 = 0.4;
102       double k2 = 0.1;
103
104       for (int n = 0; n < N; ++n) {
105         Eigen::Vector4d vPrevious;
106         vPrevious << u1[n], u2[n], u3[n], u4[n];
107
108         Eigen::Vector4d vNext;
109
110         newtonSolve(vNext, vPrevious, vPrevious, k1, k2, dt, 1e-15,
               100);
111
112         u1[n + 1] = vNext[0];
113         u2[n + 1] = vNext[1];
114         u3[n + 1] = vNext[2];
115         u4[n + 1] = vNext[3];
116
```

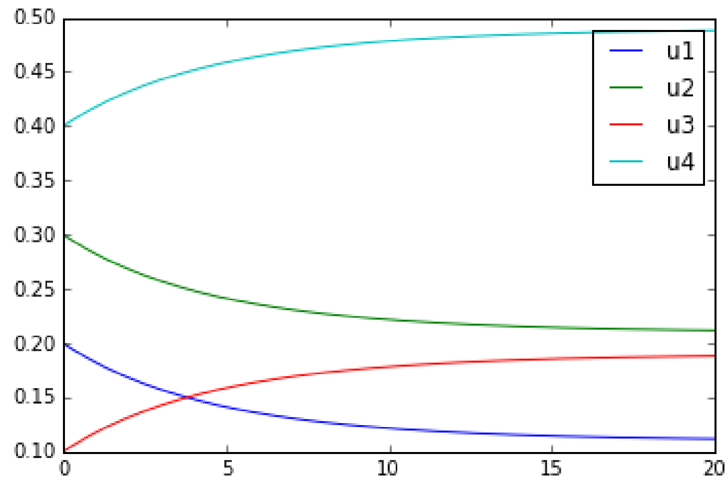


Figure 1.1: Plot of $\boldsymbol{u}$ versus time.

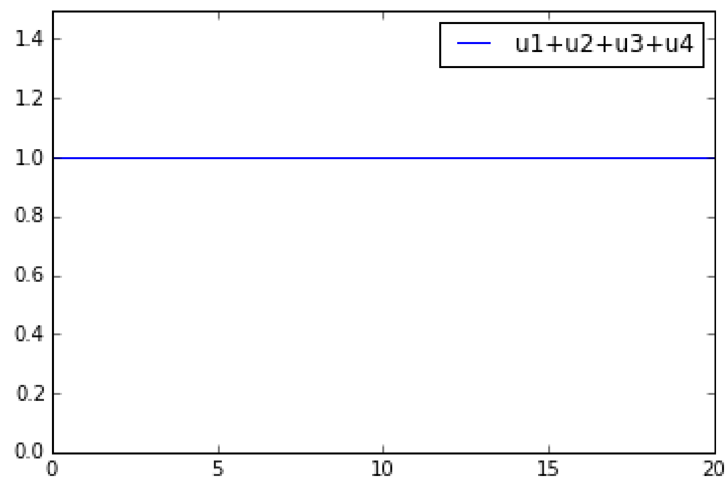


Figure 1.2: Plot of $u_1 + u_2 + u_3 + u_4$ versus time.

```
117        time[n + 1] = time[n] + dt;
118      }
119      writeToFile("u1.txt", u1);
120      writeToFile("u2.txt", u2);
121      writeToFile("u3.txt", u3);
122      writeToFile("u4.txt", u4);
123      writeToFile("time.txt", time);
124      return 0;
125  }
```

Problem 1.2 Explicit vs. Implicit Time Stepping

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

$$\ddot{x} + 2\zeta\dot{x} + x = 0, \quad t \in (0, T), \tag{1.2.1}$$

for $T > 0$. In (1.2.1), x 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

$$x(0) = x_0,\ \dot{x}(0) = v_0. \tag{1.2.2}$$

(1.2a) Equations (1.2.1) and (1.2.2) can be rewritten as a linear system of first order differential equations with appropriate initial conditions, i.e.:

$$\dot{\boldsymbol{y}} = \mathbf{A}\boldsymbol{y} \tag{1.2.3}$$

$$\boldsymbol{y}(0) = \boldsymbol{y}_0. \tag{1.2.4}$$

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

Solution: Setting $y_1 = x$ and $y_2 = \dot{x}$, we get that (1.2.1) and (1.2.2) are equivalent to the following linear system:

$$\begin{aligned} \dot{y}_2 + 2\zeta y_2 + y_1 &= 0, && t \in (0,T), \\ \dot{y}_1 &= y_2, && t \in (0,T), \\ y_1(0) &= 1, \\ y_2(0) &= 0. \end{aligned}$$

In matrix form, we can write the system as in (1.2.3) with $\boldsymbol{y} = (x, \dot{x})^\top$, $\boldsymbol{y}_0 = (1,0)^\top$ and

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

(1.2b) Compute the solution to (1.2.1) with initial conditions given by (1.2.2) with $x_0 = 1$ and $v_0 = 0$.

HINT: Starting from (1.2.3), use the matrix exponential technique.

HINT: Recall that $\zeta < 1$ and that, for a real number α, it holds that $e^{\alpha i} + e^{-\alpha i} = 2\cos\alpha$, where $i = \sqrt{-1}$ denotes the imaginary unit.

Solution: The solution is given by $\boldsymbol{y}(t) = e^{\mathbf{A}t}\boldsymbol{y}_0$.

We thus compute the matrix exponential $e^{\mathbf{A}t}$. 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 complex eigenvalues given by $\lambda_{1,2} = -\zeta \pm \sqrt{1-\zeta^2}i$. The associated eigenvalues are $\boldsymbol{v}_1 = (1, -\zeta + \sqrt{1-\zeta^2}i)^\top s_1$, $s_1 \in \mathbb{R}$, and $\boldsymbol{v}_2 = (1, -\zeta - \sqrt{1-\zeta^2}i)^\top s_2$, $s_2 \in \mathbb{R}$. Then $\mathbf{A} = \mathbf{R}\Lambda\mathbf{R}^{-1}$ and we can compute the matrix exponential as $e^{\mathbf{A}t} = \mathbf{R}e^{\Lambda t}\mathbf{R}^{-1}$, with $\mathbf{R}$ the matrix having as columns an eigenbasis and $e^{\Lambda t} = \begin{pmatrix} e^{\lambda_1} & 0 \\ 0 & e^{\lambda_2} \end{pmatrix}$. Note that, since $\mathbf{A}$ is *not* symmetric, the eigenvectors associated to distinct eigenvalues are not necessarily orthogonal. We have:

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

In the end, the solution to the homogeneous equation is given by:

$$\begin{aligned}
\boldsymbol{y}(t) = e^{\mathbf{A}t}\boldsymbol{y}_0 &= \frac{1}{\lambda_2-\lambda_1}\begin{pmatrix}1 & 1\\ \lambda_1 & \lambda_2\end{pmatrix}\begin{pmatrix}e^{\lambda_1 t} & 0\\ 0 & e^{\lambda_2 t}\end{pmatrix}\begin{pmatrix}\lambda_2 & -1\\ -\lambda_1 & 1\end{pmatrix}\boldsymbol{y}_0\\
&= \frac{1}{\lambda_2-\lambda_1}\begin{pmatrix}1 & 1\\ \lambda_1 & \lambda_2\end{pmatrix}\begin{pmatrix}\lambda_2 e^{\lambda_1 t} & -e^{\lambda_1 t}\\ -\lambda_1 e^{\lambda_2 t} & e^{\lambda_2 t}\end{pmatrix}\boldsymbol{y}_0\\
&= \frac{1}{\lambda_2-\lambda_1}\begin{pmatrix}\lambda_2 e^{\lambda_1 t}-\lambda_1 e^{\lambda_2 t} & -e^{\lambda_1 t}+e^{\lambda_2 t}\\ \lambda_1\lambda_2\big(e^{\lambda_1 t}-e^{\lambda_2 t}\big) & -\lambda_1 e^{\lambda_1 t}+\lambda_2 e^{\lambda_2 t}\end{pmatrix}\boldsymbol{y}_0\\
&= \frac{1}{\lambda_2-\lambda_1}\begin{pmatrix}\lambda_2 e^{\lambda_1 t}-\lambda_1 e^{\lambda_2 t} & -e^{\lambda_1 t}+e^{\lambda_2 t}\\ \lambda_1\lambda_2\big(e^{\lambda_1 t}-e^{\lambda_2 t}\big) & -\lambda_1 e^{\lambda_1 t}+\lambda_2 e^{\lambda_2 t}\end{pmatrix}\begin{pmatrix}1\\ 0\end{pmatrix}\\
&= \frac{1}{\lambda_2-\lambda_1}\begin{pmatrix}\lambda_2 e^{\lambda_1 t}-\lambda_1 e^{\lambda_2 t}\\ \lambda_1\lambda_2\big(e^{\lambda_1 t}-e^{\lambda_2 t}\big)\end{pmatrix},
\end{aligned}$$

and in particular, looking at the first entry, we obtain that

$$\begin{aligned}
x(t) &= \frac{1}{-2\sqrt{1-\zeta^2}i}\Big(-\zeta(e^{\sqrt{1-\zeta^2}it}-e^{-\sqrt{1-\zeta^2}it})-\sqrt{1-\zeta^2}i(e^{\sqrt{1-\zeta^2}it}+e^{-\sqrt{1-\zeta^2}it})\Big)\\
&= e^{-\zeta t}\left(\frac{\zeta}{\sqrt{1-\zeta^2}}\sin(\sqrt{1-\zeta^2}t)+\cos(\sqrt{1-\zeta^2}t)\right).
\end{aligned}$$

(1.2c) Recall the explicit Euler timestepping introduced in the lecture. Using the template file `harmonic_oscill.cpp` provided in the handout, implement the function `explicitEuler` to compute the solution $\boldsymbol{y} = \boldsymbol{y}(t)$ to (1.2.3) 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 Δt, in the template called `dt`.
- The final time T, that we assume to be a multiple of Δt.

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, $i = 1, \ldots, \frac{T}{\Delta t}$. The size of the output vectors has to be initialized inside the function according to the number of time steps.

Solution: See listing 1.2.

(1.2d) Recall the implicit Euler timestepping introduced in the lecture. Using the template file `harmonic_oscill.cpp` provided in the handout, implement the function `implicitEuler` to compute the solution $\boldsymbol{y} = \boldsymbol{y}(t)$ to (1.2.3) up to the time $T > 0$. The input and output parameters are as in the function `explicitEuler` from subproblem (1.2c).

Solution: See listing 1.2.

(1.2e) In the template file `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: See listing 1.2.

(1.2f) We consider two time steps $\Delta_1 t = 0.1$ and $\Delta_2 t = 0.5$.

Using the `main` already implemented in the template file `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 (1.2b), too. What do you observe?

Solution: See listing 1.2 and Figures 1.3 and 1.4 for the plots. As we can see, using $\Delta_2 t = 0.3$ the energy of the system explodes, which is not physical. For $\Delta_1 t = 0.1$, instead, the energy remains bounded and both schemes give an approximation to the exact solution. The reason for this behavior 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, can be unconditionally stable, and so the implicit Euler scheme worked for both timestep choices.

Listing 1.2: Implementation for the functions in subproblems (1.2c), (1.2d) and (1.2e).

```
1  #include <Eigen/Core>
2  #include <Eigen/LU>
3  #include <vector>
4  #include <iostream>
5  #include "writer.hpp"
6  #include <assert.h>
7
8  /// Uses the explicit Euler method to compute y from time 0 to time
       T
9  /// where y is a 2x1 vector solving the linear system of ODEs as in
       the exercise
10 ///
11 /// @param[out] yT at the end of the call, this will have vNext
12 /// @param[in] y0 the initial conditions
13 /// @param[in] zeta the damping factor (see exercise)
14 /// @param[in] dt the step size
15 /// @param[in] T the final time at which to compute the solution.
16 ///
17 /// The first component of y (the position) will be stored to y1,
       the second component (the velocity) to y2.  The i-th entry of y1
       (resp.  y2) will contain the first (resp.  second) component of y
       at time i*dt.
18 ///
19
20 void explicitEuler(std::vector<double> & y1, std::vector<double>
       &y2, std::vector<double> & time,
21     const Eigen::Vector2d& y0,
22            double zeta, double dt, double T) {
23   const unsigned int nsteps = T/dt;
24   y1.resize(nsteps+1);
25   y2.resize(nsteps+1);
26   time.resize(nsteps+1);
27   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)+dt*A;
      ynew=B*yold;
      y1[i+1]=ynew[0];
      y2[i+1]=ynew[1];
      yold=ynew;
      time[i+1]=(i+1)*dt;
    }
}

// Implements the implicit Euler. Analogous to explicit Euler, same input and output parameters
void implicitEuler(std::vector<double> & y1, std::vector<double> & y2, std::vector<double> & time,
    const Eigen::Vector2d& y0,
            double zeta, double dt, double T) {
  const unsigned int nsteps = T/dt;
  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)-dt*A;
      ynew=B.fullPivLu().solve(yold);
      y1[i+1]=ynew[0];
      y2[i+1]=ynew[1];
      yold=ynew;
      time[i+1]=(i+1)*dt;
```

```
76      }
77  }
78
79  // Energy computation given the velocity.  Assume the energy vector
        to be already initialized with the correct size.
80  void Energy(const std::vector<double> & v, std::vector<double> &
        energy)
81  {
82    assert(v.size()==energy.size());
83    for(unsigned i=0;i<v.size();i++)
84      energy[i]=0.5*std::pow(v[i],2);
85  }
86
87  int main() {
88
89        double T = 20.0;
90        double dt = 0.5; // Change this for explicit / implicit time
              stepping comparison
91        const Eigen::Vector2d y0(1,0);
92        double zeta=0.2;
93        std::vector<double> y1;
94        std::vector<double> y2;
95        std::vector<double> time;
96        explicitEuler(y1,y2,time,y0,zeta,dt,T);
97        writeToFile("position_expl.txt", y1);
98        writeToFile("velocity_expl.txt",y2);
99        writeToFile("time_expl.txt",time);
100       std::vector<double> energy(y2.size());
101       Energy(y2,energy);
102       writeToFile("energy_expl.txt",energy);
103       y1.assign(y1.size(),0);
104       y2.assign(y2.size(),0);
105       time.assign(time.size(),0);
106       energy.assign(energy.size(),0);
107       implicitEuler(y1,y2,time,y0,zeta,dt,T);
108       writeToFile("position_impl.txt", y1);
109       writeToFile("velocity_impl.txt",y2);
110       writeToFile("time_impl.txt",time);
111       Energy(y2,energy);
112       writeToFile("energy_impl.txt",energy);
113
114       return 0;
115 }
```

Problem 1.3 Multiple Choice: Basic concepts

Each question may have *multiple correct answers*.

(1.3a) Which of the following ODEs are *autonomous*?

1. $u'(t) = \cos(u(t))$

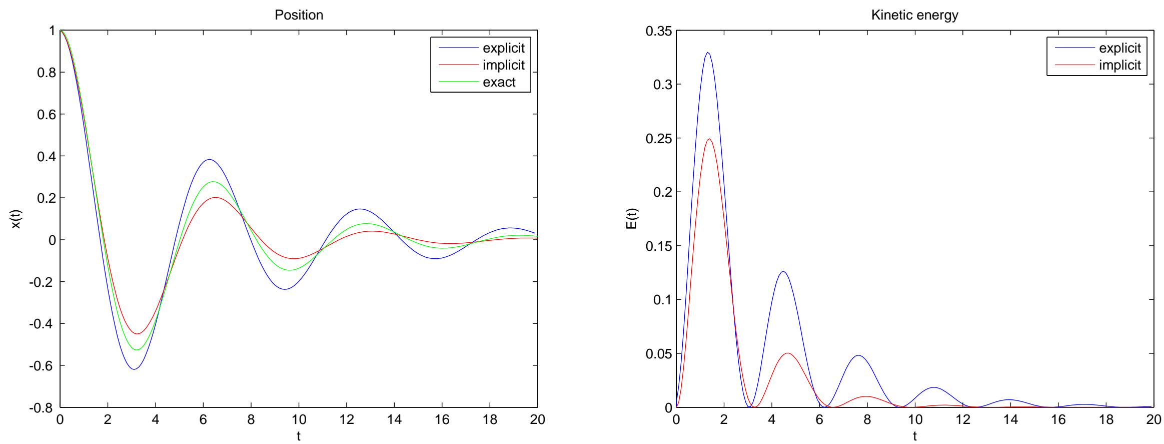


Figure 1.3: Plot of position and energy versus time for $\Delta_1 t = 0.1$.

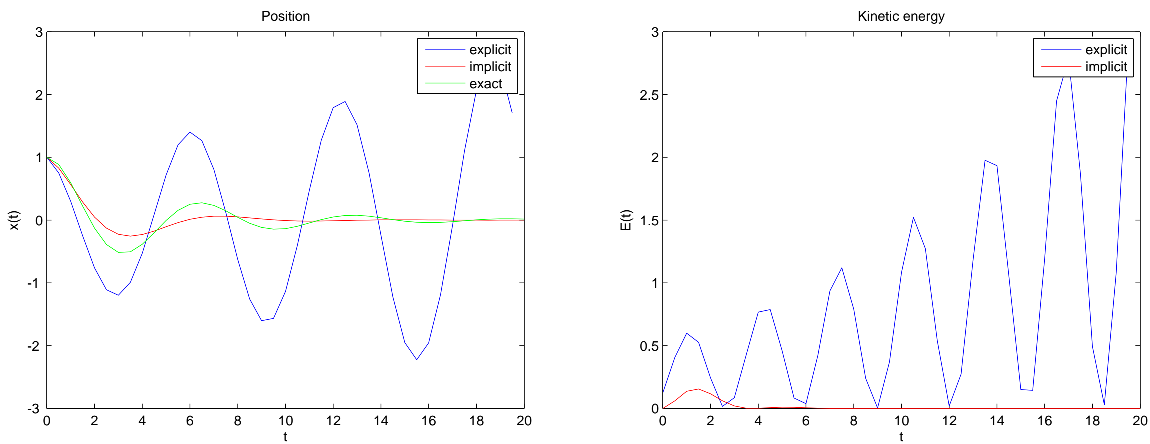


Figure 1.4: Plot of position and energy versus time for $\Delta_2 t = 0.5$.

2. $u'(t) = u(t)^2 + t$

3. $u'(t) = \exp(t)u'(t)$

4. $u'(t) = \sin(u(t) + 2\pi t)$

Solution: **Correct answer:** 1.
An autonomous ODE is an ODE where the right hand side does not explicitly depend on the time t. We see that the only option satisfying this is *1.)*

(1.3b) Which of the following ODEs are *scalar*?

1. $u'(t) = 4u(t)$

2. $u'(t) = u(t)^2 + u(t)$

3. $\begin{pmatrix} u_1'(t) \\ u_2'(t) \end{pmatrix} = \begin{pmatrix} 3u_1(t) + u_2(t) \\ u_2(t) \end{pmatrix}$

4. $u'(t) = \cos(u(t))$

Solution: **Correct answer:** 1., 2., 4.
ODE is scalar when $u(t) \in \mathbb{R}$ (ie. when u is not a vector).

(1.3c) Which of the following ODEs are *linear*?

1. $u'(t) = 4u(t)$

2. $u'(t) = \exp u(t) + \sin(u(t))$

3. $\begin{pmatrix} u_1'(t) \\ u_2'(t) \end{pmatrix} = \begin{pmatrix} 3u_1(t) + u_2(t) \\ u_2(t) \end{pmatrix}$

4. $u'(t) = u(t)^2 + u(t)$

Solution: **Correct answer:** 1., 3.
An autonomous ODE is linear when any linear combinations of solutions to the ODE is also a solution. You can check if the ODE is linear if the right hand side is linear.

(1.3d) We are given the ODE

$$u'(t) = -u(t),$$

with initial value $u(0) = A > 0$. We solve the ODE using the *Forward-Euler* method with step size Δt. What will be the first value u_1?

1. Method will crash because of the minus sign

2. $(1 - \Delta t)A$

3. $A - \Delta t A + \Delta t^2 A$

4. $A \exp(-\Delta t)$

Solution: **Correct answer:** 2.
The Forward-Euler is given as

$$u_{n+1} = u_n + \Delta t F(u_n) = u_n - \Delta t u_n = (1 - \Delta t) u_n$$

with $u_0 = A$, we get

$$u_1 = (1 - \Delta t)A.$$

(1.3e) We are given the ODE

$$u'(t) = -u(t),$$

with initial value $u(0) = A > 0$. We solve the ODE using the *Backward-Euler* method with step size Δt. What will be the first value u_1?

1. Method will crash because of the minus sign
2. $(1 + \Delta t)A$
3. $A + \Delta t A + \Delta t^2 A$
4. $A/(1 + \Delta t)$

Solution: **Correct answer:** 4.
The Backward-Euler is given as

$$u_{n+1} = u_n + \Delta t F(u_{n+1}) = u_n - \Delta t u_{n+1},$$

solving for u_{n+1} we get

$$u_{n+1} = \frac{u_n}{1 + \Delta t},$$

which if we insert $u_0 = A$ we get

$$u_1 = \frac{A}{1 + \Delta t}.$$

Published on September 29.

To be submitted on October 13.

Last modified on October 19, 2015