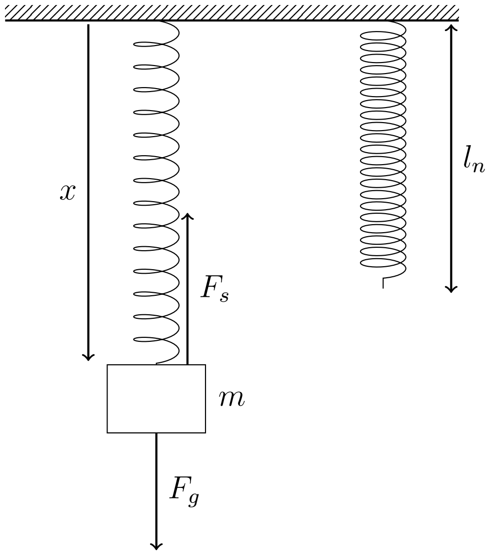
Christian Schilling
Jakob Ø. Hansen
Kim G. Larsen
Martijn Goorden
Max Tschaikowski
Milad Samim

AALBORG UNIVERSITY
DENMARK

Models and Tools for Cyber-Physical Systems
Exercise sheet 8
WITH SOLUTIONS

Exercise 1: Spring system



Consider the mass-spring system depicted above. A spring is attached to a fixed ceiling and carries a block of mass m. When a spring is not loaded, shown on the right, it has a neutral length l_n. The force a spring generates is proportional to the extension relative to its neutral position.
The two forces acting on the block are the gravitational force F_g and the spring force F_s, which can be expressed as

$$F_g = mg \qquad\qquad F_s = k(l - l_n),$$

where g is the gravitational constant, k is the spring constant, and l is the current length of the spring.

(a) Show, using Newton's law $F = ma$ and the two force equations above, that the dynamics of the block can be described with the following differential equation:

$$m\frac{d^2}{dt^2}x = mg - k(x - l_n).$$

(b) Create the continuous-time component for this system as a block diagram. What are the internal state variables, what are the inputs, and what are the outputs?

(c) Provide a formula for the steady-state position x_e of the block where the block is not moving.

.... Solution sketch ....

(a) First, note that the force of the spring is opposite to the direction in which the spring is extended, see figure. Second, note that the gravitational force is in the positive x-direction. Taking this into account, inserting the two force equations into Newton's law gives

$$\begin{aligned} F &= ma \\ F_g - F_s &= ma \\ mg - k(l - l_n) &= ma. \end{aligned}$$

Next, we have that acceleration a is the second derivative of the position x, so we get

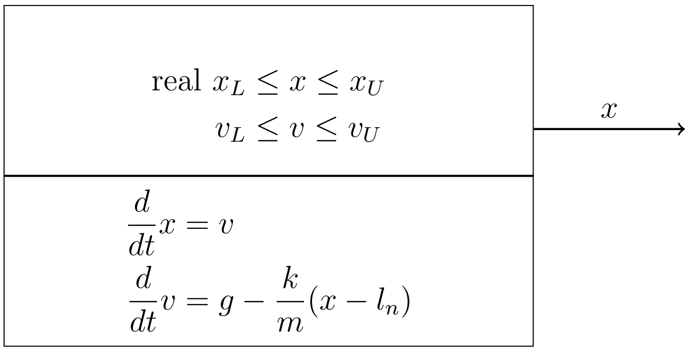
$$mg - k(l - l_n) = m\frac{d^2}{dt^2}x.$$

Finally, observe that in this system the length of the spring l equals the position of the block x. Inserting this results in

$$mg - k(x - l_n) = m\frac{d^2}{dt^2}x,$$

which equals the provided equation.

(b) For a continuous-time component, we can only have first-order differential equations. Therefore, we introduce the first-order derivative of x (which is velocity v) as a second state variable. The full block diagram is shown below. Note that the output is state variable x and that there is no input for this system.


real $x_L \leq x \leq x_U$
$v_L \leq v \leq v_U$
$\frac{d}{dt}x = v$
$\frac{d}{dt}v = g - \frac{k}{m}(x - l_n)$
x

(c) The system is in steady-state (equilibrium) when block does not move over time. This means that $\frac{d}{dt}x = 0$ and $\frac{d}{dt}v = 0$. From the first equation we have that $v_e = 0$. From the second equation, we can obtain, after some rewriting, that $x_e = \frac{mg}{k} + l_n$.

Exercise 2: Implementation of the Euler method

Many tools for cyber-physical systems, like Matlab, Mathematica, and Uppaal (coming in release version 5), have built-in implementations for solving differential equations numerically. In this exercise, you will create your own numerical solver that you can use in the other exercises. We provide you with Python code snippets to extend. Challenge: you can try to implement the Euler method without the guided steps below.

(a) Conceptually, the algorithm of the Euler method is as follows (see also slides)

```
1: procedure EULER($f, x_0, h, N$)
2:     $\tilde{x}_0 \leftarrow x_0$
3:     for $i \in \{0, 1, \ldots, N\}$ do
4:         print $\tilde{x}_i$
5:         $\tilde{x}_{i+1} \leftarrow \tilde{x}_i + h \cdot f(\tilde{x}_i)$
6:     end for
7: end procedure
```

Instead of printing the values directly, we would like to store them in an array and return the array. Implement the Euler method in the function framework below, where `f` will be function describing the ODE, `x0` the initial state value, `h` the step size, and `N` the number of iterations. You can assume, for the first implementation, that we only have a single state variable in our ODE.

```python
def euler_method(f, x0, h, N):
  ...
  return result
```

(b) We can have ODEs that explicitly depend on time, like $\frac{d}{dt}x = t$. While we can include t as another state variable in our continuous-timed component, it is easier to treat time separately in our implementation. Extend the Euler method function such that (1) it accepts an initial time `t0`, (2) it calls the ODE function `f` with the current state *and* current time, and (3) it returns an array of all the time stamps for which the value of x is calculated.

```python
def euler_method(f, x0, t0, h, N):
  ...
  return t, result
```

Also, implement the function `f` such that we can actually test the code. Compare the obtained values with the ones from example on the slides (where $\frac{d}{dt}x = t$).

(c) We will now extend the code to handle more than one state variable. Adjust the code such that (1) `euler_method` accepts that `x0` is an n-sized array of initial values, where n is the number of state variables, (2) `f` returns an array (again of size n) of derivatives, and (3) `result`, i.e., one of the return values of `euler_method`, should be an array where each element is again an array of the state values, so it is of size $(N + 1) \times n$.

(d) Solutions to ODEs are typically plot for visual inspection. In Python we can use the library `matplotlib` to handle plotting for us. With this library, we can plot data using `plot(x, y)` where `x` and `y` are arrays of the values for the x and y axes, respectively. Unfortunately, the data format returned by `euler_method` is in the wrong format. Finish the `convert` function that converts the data for us. Verify that the resulting plot matches the one from the slides.

```
import matplotlib.pyplot as plt

def convert(data):
   ...
   return result

def main():
  x0 = [0, 0]
  t0 = 0
  h = 1
  N = 5
  t, result = euler_method(f, x0, t0, h, N)
  result = convert(result)

  # The actual plotting of data.
  fig, ax = plt.subplots()
  for i in range(len(result)):
    ax.plot(t, result[i])
  plt.show()
```

(e) (*Optionally*) Modularize your code such that you can easily run experiments where you approximate the same function but where you vary the precision of the approximation by varying the step size h. Ideally, you should be able to plot the results in the same figure.

(f) (*Optionally*) Extend your code such that you can also plot the (provided) exact solution of the ODE. For the example $\frac{d}{dt}x = t$, the exact solution is given by $x(t) = \frac{1}{2}t^2$.

Solution sketch

(a) A solution could be

```
def euler_method(f, x0, h, N):
  result = [x0]
  for i in range(N):
    xi = result[-1]
    der = f(xi)
    result.append(xi + h * der)
  return result
```

(b) A solution could be

```
def euler_method(f, x0, h, N):
  t = [t0]
  result = [x0]
  for i in range(N):
    xi = result[-1]
    der = f(xi, t[-1])
    t.append(t[-1] + h)
    result.append(xi + h * der)
  return result

def f(x, t):
  return t

def main():
  x0 = [0, 0]
  t0 = 0
  h = 1
  N = 5
  t, result = euler_method(f, x0, t0, h, N)
  print result
```

(c) A solution could be the following one. You could make it cleaner by using an external library that can manipulate multi-dimensional arrays and matrices, like NumPy.

```
def euler_method(f, x0, t0 h, N):
  t = [t0]
  result = [x0]
  for i in range(N):
    xi = result[-1]
    der = f(xi, t[-1])
    t.append(t[-1] + h)
    step = []
    for j in range(len(der)):
      step.append(xi[j] + h * der[j])
    result.append(step)
  return t, result

def f(x, t):
  return [t]
```

(d) A solution could be the following one. Again, you could make it cleaner by using an external library that can manipulate multi-dimensional arrays and matrices, like NumPy.

```
def convert(data):
  result = []
  for i in range(len(data[0])):
    result.append([])

  for x in data:
    for i in range(len(x)):
      result[i].append(x[i])
  return result
```

(e) You could define an experiment as follows.

```
def experiment(x0, t0, h, T, ax):
  N = round(T / h)
  t, result = euler_method(f, x0, t0, h, N)
  result = convert(result)
  for i in range(len(result)):
        ax.plot(t, result[i], label=f'x_euler[{i}]_{h}(t)')
    return t

def main():
    x0 = [0]
    t0 = 0
    T = 5
    fig, ax = plt.subplots()
    t = experiment(x0, t0, 1, T, ax)
    t = experiment(x0, t0, 0.1, T, ax)
    t = experiment(x0, t0, 0.01, T, ax)
    ax.legend()
    plt.show()
```

(f) A solution could be the following one.

```
def f_exact(t):
    result = []
    for ti in t:
        result.append(0.5 * ti * ti)
    return result

def main():
    x0 = [0]
    t0 = 0
    T = 5
    fig, ax = plt.subplots()
    t = experiment(x0, t0, 1, T, ax)
    t = experiment(x0, t0, 0.1, T, ax)
    t = experiment(x0, t0, 0.01, T, ax)

    exact = f_exact(t)
    ax.plot(t, exact, label='Exact x(t)', color='red')
    ax.set_xlabel("t")
    ax.legend()
    plt.show()
```

Exercise 3: Implementation of the Runge-Kutta method
As discussed in the course, while the Euler method is the most basic numerical method for solving differential equations, it is seldom used in practice due to its inefficiency. A much better single-step family of methods is the Runge-Kutta methods. The most well-known version of it is the fourth-order Runge-Kutta scheme:

```
1:  procedure RUNGEKUTTA4($f, x_0, h, N$)
2:      $\tilde{x}_0 \leftarrow x_0$
3:      for $i \in \{0, 1, \dots, N\}$ do
4:          print $\tilde{x}_i$
5:          $k_1 \leftarrow f(\tilde{x}_i)$
6:          $k_2 \leftarrow f(\tilde{x}_i + (h/2)k_1)$
7:          $k_3 \leftarrow f(\tilde{x}_i + (h/2)k_2)$
8:          $k_4 \leftarrow f(\tilde{x}_i + hk_3)$
9:          $\tilde{x}_{i+1} \leftarrow \tilde{x}_i + h/6 \cdot (k_1 + 2k_2 + 2k_3 + k_4)$
10:     end for
11: end procedure
```

(a) Adjust your numerical solver from the previous exercise where you replace the Euler method by the fourth-order Runge-Kutta scheme. Test your implementation with the differential equation $\frac{d}{dt}x = t$, as the exact solution is given by $x(t) = \frac{1}{2}t^2$.

(b) Examine the difference in numerical precision of the Euler method and the fourth-order Runge-Kutta scheme. You can do this by plotting the error between the approximations and the exact solution.

.... Solution sketch ....

(a) A solution could be the following one. Again, you could make it cleaner by using an external library that can manipulate multi-dimensional arrays and matrices, like NumPy.

```python
def runge_kutta_4(f, x0, t0, h, N):
  t = [t0]
  result = [x0]
  for i in range(N):
    xi = result[-1]
    k1 = f(xi, t[-1])
    k2 = f(extend(xi, h/2, k1), t[-1] + h/2)
    k3 = f(extend(xi, h/2, k2), t[-1] + h/2)
    k4 = f(extend(xi, h, k3), t[-1] + h)
    step = []
    for j in range(len(xi)):
      step.append(xi[j] + h/6 * (k1[j] + 2*k2[j] + 2*k3[j] + k4[j])
                      )
    result.append(step)
    t.append(t[-1] + h)
  return t, result


def extend(x, h, der):
  assert(len(x)) == len(der)
  result = []
```

```
    for j in range(len(x)):
      result.append(x[j] + h * der[j])
    return result
```

(b) If you are using the differential equation $\frac{d}{dt}x = t$, where the exact solution is given by $x(t) = \frac{1}{2}t^2$, the fourth-order Runge-Kutta obtains the exact solution regardless of the step size.

Exercise 4: Lipschitz continuity

(a) Consider the differential equation $\frac{d}{dt}x(t) = g(x(t))$ with $g(x) = x^2 - x$. Using your own numerical solver from the previous exercise, compute and plot the numeric solution of $\frac{d}{dt}x(t) = g(x(t))$ in the case of initial value $x(0) = 2.0$ on time intervals $[0; 0.67]$, $[0; 0.68]$, $[0; 0.69]$ and $[0; 0.70]$. Can you explain what happens?

(b) To understand better what happens in (a), consider a function $f : [a; b] \to \mathbb{R}$ that is differentiable and recall that the derivative (i.e., slope) of f at x, denoted by $\frac{d}{dx}f(x)$, is given by

$$\frac{d}{dx}f(x) = \lim_{h\to 0}\frac{f(x+h) - f(x)}{h}$$

The mean-value theorem from calculus postulates that for any two $y_1, y_2 \in [a; b]$, there exists a x with $y_1 < x < y_2$ such that $f(y_2) - f(y_1) = \frac{d}{dx}f(x)(y_2 - y_1)$.

Using the above observation, compute the Lipschitz constant L_c of $g(x) = x^2 - x$ when $x \in [-c; c]$. That is, provide an $L_c < \infty$ that satisfies $|g(x_1) - g(x_2)| \leq L_c|x_1 - x_2|$ for all $x_1, x_2 \in [-c; c]$. What can be said about L_c if $c \to \infty$?

.... Solution sketch ....

(a) You can use the solution code from Exercise 2 where you define function `f` to be as follows. Note that you choose a value of h that is sufficiently small, for example $h = 0.005$.

```
def f(x, t):
  return [x[0] * x[0] - x[0]]
```

(b) We can rewrite the definition of the Lipschitz constant into

$$\frac{|g(x_1) - g(x_2)|}{|x_1 - x_2|} \leq L_c.$$

Note that this is the same as the mean-value theorem, except that the absolute brackets ensure that L_c is always a non-negative number. By relying on the mean-value theorem, it suffices to consider the derivative of $g(x) = x^2 - x$, which is $\frac{d}{dx}g(x) = 2x - 1$. Specifically, on $[-c; c]$, the Lipschitz constant L_c is

$$\max_{x\in[-c;c]} |2x - 1| = 2c + 1$$

This shows that the slopes of solutions of $\frac{d}{dt}x(t) = g(x(t))$ are not bounded on the entire $\mathbb{R}$. Hence, pathological behavior as observed in (a) can, at least in principle, happen.

(For those who are curios: One says that $\frac{d}{dx}x(t) = g(x(t))$ has a finite explosion time for $x(0) = 2$. Note however that there is no finite explosion when $x(0) = 0.5$. That is, the fact that slopes cannot be bounded does not necessarily imply a finite explosion time.)

Exercise 5: Car model

Consider the car model from the course

$$\frac{d}{dt}x(t) = v(t) \qquad\qquad \frac{d}{dt}v(t) = -\frac{k}{m}v(t) + \frac{1}{m}F,$$

where x is the position, v the velocity and F the applied force to a car of mass m with friction coefficient k.

(a) Assuming that a constant force F is applied, the velocity of the car will converge to steady-state velocity. Provide a formula for the steady-state velocity by assuming that the value of F is known.

(b) From now on, we assume that $F = 500$ (N), $m = 1000$ (kg), $k = 50$ $(\frac{m\,kg}{s^2})$. Using the initial condition $(x(0), v(0)) = (0, 0)$, compute and plot the numeric solution.

(c) Assuming that a steady-state velocity of $v = 20$ $(\frac{m}{s})$ is desired, compute the corresponding constant force F. Adjust your code and verify by plotting the underlying numeric solution that the system reaches the desired steady-state velocity.

.... Solution sketch ....

(a) To obtain the velocity attained by a car after some time, we set $\dot{v} = 0$, which is equivalent to $0 = -\frac{k}{m}v + \frac{1}{m}F$. This, in turn, implies $v = \frac{F}{k}$. Note that $\dot{x} \neq 0$, i.e., while the velocity reached an equilibrium, the overall system did not because the car is moving at constant speed.

(b) You can use the solution code from Exercise 2 where you define function `f` to be as follows.

```python
def f(x, t):
  F = 500
  m = 1000
  k = 50
  der_x = x[1]
  der_v = -k / m * x[1] + F / m
  return [der_x, der_v]
```

(c) Rearranging $v = \frac{F}{k}$ from (a) yields $F = vk$. With this, $v = 20$ $(\frac{m}{s})$ and $k = 50$ $(\frac{m\,kg}{s^2})$, we obtain $F = 1000$ (N). We can insert this new value of F into the script from (b) and verify using simulation on the time interval $[0; 100]$ that the steady-state velocity is indeed the desired one.
