16 Explicit Euler's method

16.1 Theoretical part

It is finally time to tackle numerically ordinary differential equations. In particular, I start with the following initial value problem (IVP):

$$x' = f(t, x), \quad x(t_0) = x_0, \tag{16.1}$$

note the choice of the notation, my independent variable is denoted by t, and the dependent variable is x, i.e., I am looking for a function $x = x(t)$ that also satisfies the given initial condition $x(t_0) = x_0$, where t_0, x_0 are given numbers. In the following I assume that the conditions for the existence and uniqueness theorem are satisfied, which is the case, e.g., if function f is continuous in variable t and continuously differential in variable x.

As a simple example consider the IVP

$$x' + x = \sin t, \quad x(0) = 0. \tag{16.2}$$

This is a linear equation, which can be solved, for instance, by using the method of integrating factor, and I invite the reader to check that the solution to this problem is given by

$$x(t) = \frac{1}{2}e^{-t} - \frac{1}{2}\cos t + \frac{1}{2}\sin t. \tag{16.3}$$

In most cases, however, I cannot write down the explicit solution (it still exists but frequently cannot be expressed in terms of elementary functions and their integrals). That is, natural approach to this problem is numerical.

Arguably, the simplest method to solve problem (16.1) numerically is to recall that geometrically x' gives the slope of function f, and therefore the slope of my (yet unknown) solution at the point t_0 is $x'(t_0) = f(t_0, x_0)$, which I know. Therefore, I can use this slope to approximate my unknown solution with the straight line, passing through (t_0, x_0), which is given by

$$x(t_1) = x_0 + f(t_0, x_0)(t_1 - t_0), \tag{16.4}$$

at some "future" time moment t_1 (make a picture!). Denoting $x_1 = x(t_1)$ I can find now the slope at the point (t_1, x_1), which is given by $f(t_1, x_1)$ and use to find the next approximation x_2, and so on. If I choose a fixed step size $h = t_1 - t_0$, which will be the same for all the future steps, I obtain

$$x_{k+1} = x_k + hf(t_k, x_k), \quad t_k = t_0 + kh, \quad x(t_0) = x_0, \tag{16.5}$$

which allows me to build two sequences: sequence of time moments $\{t_0, t_1, \ldots, t_n\}$, and sequence of my approximations $\{x_0, x_1, \ldots, x_n\}$, if I need to make n steps, thus solving my IVP on the time interval $[t_0, t_0 + nh]$.

Formulas (16.5) are called an *explicit Euler's method*, and were used by Leonard Euler in 1768. Let us now consider how it works.

Math 488/688: Numerical Analysis
e-mail: artem.novozhilov@ndsu.edu. Fall 2021

16.2 Implementation

Implementation of Euler's method is very simple. All we need is to clearly understand that the input data for our algorithm must contain 1) the right hand side f of our ODE, the initial condition x_0 and *the time span* on which we solve the problem, $t \in [t_0, t_{end}]$. Usually, the problem itself does not indicate what t_{end} must be, and it is the user's task to submit a reasonable value. Without t_{end} no software would know how far the code should proceed. Since we just starting, I will be also submitting n, the number of steps I expect to be made during the process of solution. To test the code, I will use example (16.2) with the known explicit solution (16.3). I will also try to present most of the comparisons in a graphical form.

```
import numpy as np
import matplotlib.pyplot as plt

def f(t: float, x: float):#test example
    return -x + np.sin(t)

def exact(t: float):
    return 1/2*np.exp(-t)-1/2*np.cos(t)+1/2*np.sin(t)

def ode_explicit_euler_v1(f, t_span, x0, n):
    '''accepts function f of two arguments t and x, the aaray t_span=[t0,t_end], the
        initial condition x0,
    and the number of steps n. Returns two arrays: first the array of time moments t0,
        t1,..., t_end, and second the array
    of approximations x0, x1,...,x_n of the solution to the given IVP.
    '''

    t = np.linspace(t_span[0], t_span[1], n) #create the array of time moments
    x = t.copy() #allocate the memory for array of x values
    h = (t_span[1]-t_span[0])/(n - 1) #compute the step size
    x[0], t[0] = x0, t_span[0]
    for i in range(1, n):
        x[i] = x[i-1] + h * f(t[i-1],x[i-1])

    return t, x

a = 0
b = 1
n = 16
x0 = 0

sol = ode_explicit_euler_v1(f, [a, b], x0, n)

t = np.linspace(a, b, 100)
x = exact(t)
plt.rc('text', usetex=True)
plt.plot(t, x,'g-')
plt.plot(sol[0], sol[1],'ro--')
plt.xlabel(r'\LARGE $t$')
```

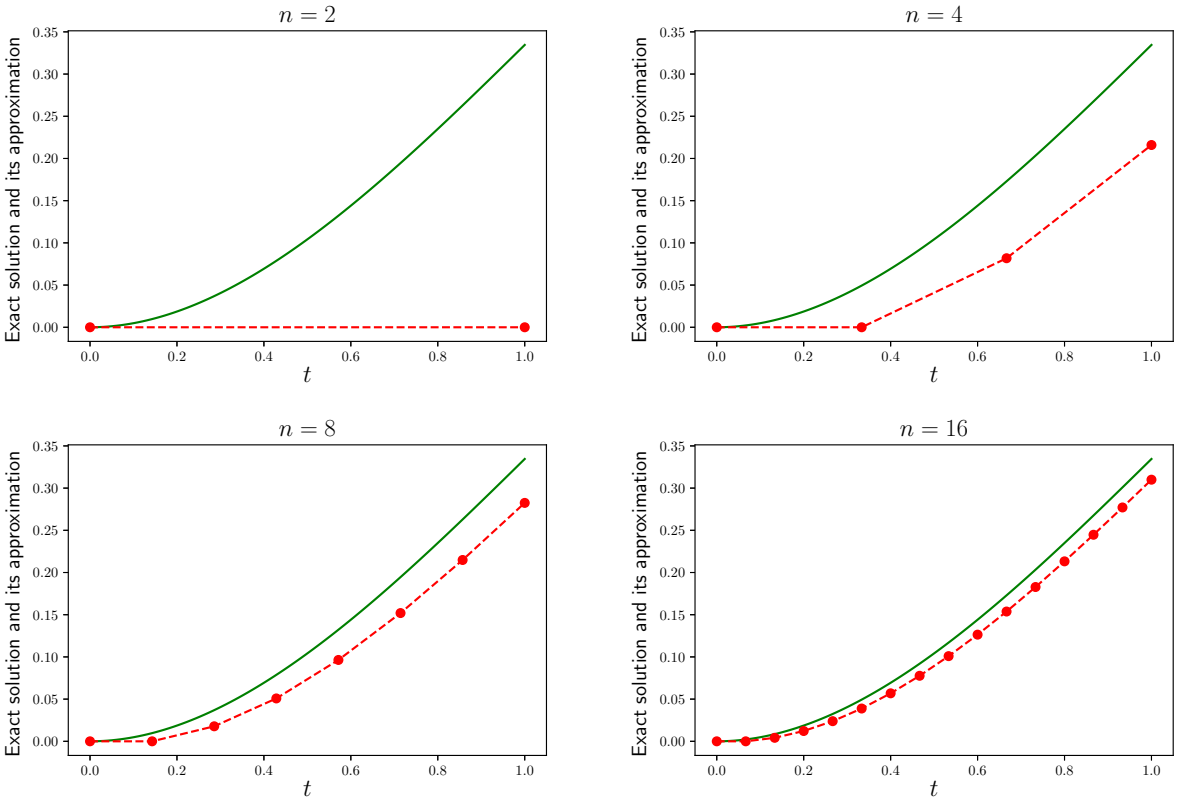

Figure 1: Comparison of the exact solution (green) and its approximation (red dots) on $[0, 1]$ with Euler's method for different number of steps.

```
plt.ylabel('\Large Exact solution and its approximation');
plt.title('\LARGE $n=16$');
plt.savefig('fig16_4.eps', format='eps')
```

It is clearly seen that increasing the number of steps leads to a better approximation. Moreover, if I increase the number of steps more significantly, I get almost exactly my solution, see the next figure.

16.3 Solving systems and equations of arbitrary order

If you remember your math 266 class, you should recall that you spend a significant amount of time trying to solve either systems of ordinary differential equations or one equation of higher order (recall that the order of an ODE is the order of the highest derivative). Can I modify my code above to include these cases? The answer is resounding "yes," and moreover it turns out that almost no changes are required if I use the possibility of the `numpy` package for vectorized calculations.

I start with an example:

$$\begin{aligned} x' &= -y, \\ y' &= x, \end{aligned} \tag{16.6}$$

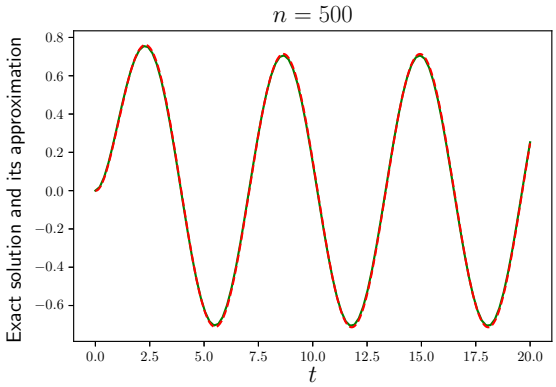

Figure 2: Comparison of the exact solution (green) and its approximation (red dots) on $[0, 20]$ with Euler's method for 500 steps.

with the initial conditions $x(0) = 1$, $y(0) = 1$. It is not difficult to solve this system analytically, but I will leave it as an exercise for the student. My goal here is to modify the basic Euler's method to include the possibility to solve systems of equations.

So, in general, I want to solve

$$\begin{aligned} x' &= f(t, x, y), \\ y' &= g(t, x, y), \end{aligned} \tag{16.7}$$

with $x(t_0) = x_0, y(t_0) = y_0$. Arguing exactly as in the scalar case, I claim that my formulas for Euler's method are

$$\begin{aligned} x_{k+1} &= x_k + hf(t_k, x_k, y_k), \\ y_{k+1} &= y_k + hg(t_k, x_k, y_k), \end{aligned} \tag{16.8}$$

with the starting values t_0, x_0, y_0. More importantly, however, I can rewrite equations (16.7) in the vector form

$$\boldsymbol{x}' = \boldsymbol{f}(t, \boldsymbol{x}),$$

where $\boldsymbol{x} = (x, y)^\top$ ($^\top$ means transposition, i.e., my vector $\boldsymbol{x}$ is a column vector) and $\boldsymbol{f} = (f, g)^\top$. Moreover, the same vector notation cab be used for systems with 3, 4, or any number of equations. Using the same vector notation, Euler's method becomes simply

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + h\boldsymbol{f}(t_k, \boldsymbol{x}_k), \tag{16.9}$$

where now $\boldsymbol{x}_k$ is a vector of values. Here you should also recall that multiplying a vector by a scalar means I multiply every element of this vector by this scalar. Here is a straightforward implementation of (16.9) in Python. It will work for any dimension, including also dimension 1, covered in the previous section. I also assume that all the required packages are loaded.

```
def f(t, x):#problem (16.6)
    f1 = - x[1]
    f2 = x[0]
```

```
    return np.array([f1, f2])


def ode_explicit_euler(f, t_span, x0, n):
    '''takes vector function f, the array t_span=[t0,t_end], and array of initial values x0.
    Even is x0 is one-dimensional, it should be an array,
    i.e., x0=[a]. n is the number of steps. The dimensions of f and x0 must coincide,
    and f must accept two variables: one-dimensional t and array x of the same dimension as
        x0.
    The function returns one the tuple (t,x), where t is one dimensional array and x is k
        by n array,
    where k is the dimension of x0 such that x[0] gives the first solution, x[1] gives the
        second solution, etc.
    '''
    k = len(x0) #dimension of the system to be solved
    t = np.linspace(t_span[0], t_span[1], n) #compute the array of times
    x = np.zeros((n, k)) #create n by k array where the approximations of solutions are
        stored
    h = (t_span[1] - t_span[0])/(n - 1) #compute the stepsize
    x[0], t[0] = x0, t_span[0]
    for i in range(1, n):
        x[i] = x[i-1] + h * f(t[i-1], x[i-1]) #all the computations use numpy arrays

    return (t, np.transpose(x)) #note that transposition returns k by n array of x as
        required

t, sol = ode_explicit_euler(f,[0, 10],[1, 1], 5)

>>> t, sol #output
(array([ 0. , 2.5, 5. , 7.5, 10. ]),
 array([[ 1.       , -1.       ,  2.19694429, -4.11479284, 5.99079279],
        [ 1.       , -1.       ,  2.19694429, -4.11479284, 5.99079279]]))
```

While solving systems of ODE, there are more ways to represent the solutions graphically. First, of course, one can plot $x(t), y(t)$ versus time t in the same (or different) coordinate axis.

One can actually plot the same solution in three dimensional space as follows.

Finally, especially in the case when the right hand side $\boldsymbol{f}$ does not depend on t (in this case the system is called *autonomous*), it is often representative to plot the solution in the coordinates (x, y), parameterized by the time t, which also specifies the direction along the curves, which is frequently indicated by an arrow.

In the last three figures I presented the same solution. I encourage the student to analyze all three of them and see clearly how they are connected.

Finally, let me explain how to solve using Euler's method a higher order equation. For instance, consider the famous van der Pol equation has the form

$$x'' - \mu(1 - x^2)x' + x = 0,$$

i.e., of the second order. Here $\mu > 0$ is a parameter. To solve this equation one would need two initial conditions $x(t_0) = x_0$ and $x'(t_0) = x_1$ (in general, solving k-th order equation requires k

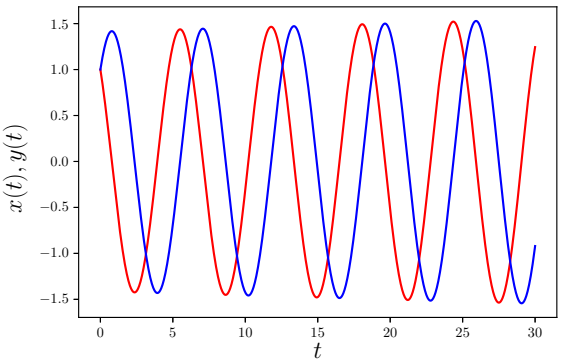

Figure 3: Solutions by Euler's method to problem (16.6).

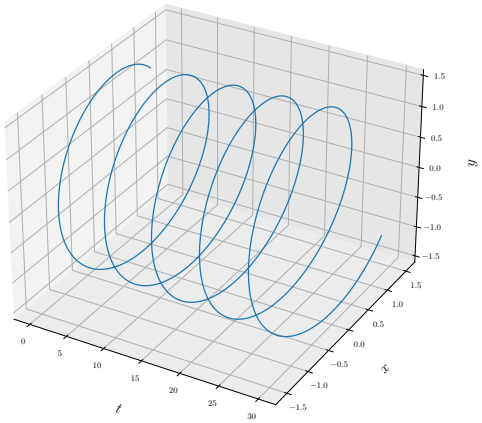

Figure 4: Solutions by Euler's method to problem (16.6) plotted in (t, x, y) coordinates.

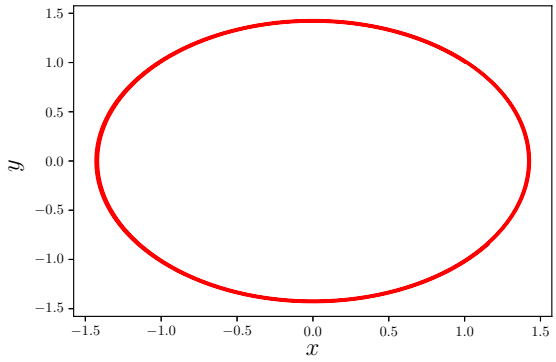

Figure 5: Solutions by Euler's method to problem (16.6) in coordinates (x, y) parameterized by the time t.

initial conditions). At the first glance it is not clear how to use considered above Euler's method to solve this equation numerically. The trick is to rewrite this equation as a system of two first order equations! Such modification is always possible and is required for many professional software (e.g., you would need to always rewrite your equation as a system in matlab or if you intend to use Python's `scipy.integrate.solve_ivp`).

Here is how it is done. We introduce new variables $x_1(t) = x(t)$ and $x_2(t) = x'(t)$. Note that $x_1' = x_2$ and $x_2' = x''$. Hence I have

$$\begin{aligned} x_1' &= x_2, \\ x_2' &= \mu(1 - x_1^2)x_2 - x_1, \end{aligned}$$

which is now a system of two first order equations to which Euler's method is applied. The initial conditions become $x_1(t_0) = x_0, x_2(t_0) = x_1$.

In general, if one has the k-th order equation

$$x^{(k)} = f(t, x, x', x'', \ldots, x^{(k-1)}),$$

then an equivalent system takes the form

$$\begin{aligned} x_1' &= x_2, \\ x_2' &= x_3, \\ &\vdots \\ x_{k-1}' &= x_k, \\ x_k' &= f(t, x_1, x_2, \ldots, x_k). \end{aligned}$$

So, great, Euler's method clearly works, all I need is to request a reasonable number of steps. Or not? Let me look at Euler's method from a slightly more theoretical point of view.

16.4 Error analysis