Computer Lab Instructions for ODE I

Joel Dahne

Spring 2022

Contents

1 Overview

The goal with this computer lab is to implement some simple numerical methods for approximating solutions to ordinary differential equations. We will look at two different methods, the forward Euler method as well as a fourth order Runge-Kutta method.

The first part of these instructions is a warm-up problem where we implement these methods for a 1-dimensional system. The second part discussed these methods for a 2-dimensional system.

In this case no code is given and it is part of the project to implement this. The third part is the actual project, it considers a second order ODE which we want to numerically approximate by rewriting it as a system of first order ODEs.

The code examples are given in both Python and in Octave[1]/Matlab. You are free to do your own implementation in any language you seem fit.

2 A warm-up problem

Consider the initial value problem

$$y' = f(t, y), \quad y(t_0) = y_0$$

where the function f and the constants t_0 and y_0 are given. We want to compute an approximation if $y(a)$ where a is a given point such that $a > t_0$. The goal will be to get an approximation for which the first two decimals are correct.

2.1 A sample problem

As an example we consider the ODE

$$y' = 1 - t + 4y, \quad y(0) = 1 \tag{1}$$

and look for an approximation of $y(1)$. This is a problem on form above with $t_0 = 0$, $y_0 = 1$, $f(t, y) = 1 - t + 4y$ and $a = 1$.

Since this is a first order linear ODE it can be solved explicitly. The general solution is given by

$$y(t) = C_1 e^{4t} + \frac{t}{4} - \frac{3}{16}$$

and the initial value $y(0) = 1$ gives $C_1 = \frac{19}{16}$ and hence

$$y(t) = \frac{19}{16} e^{4t} + \frac{t}{4} - \frac{3}{16}.$$

This gives us $y(1) = \frac{19}{16} e^4 + \frac{1}{4} - \frac{3}{16} \approx 64.8978$. A plot of y on the interval $[0, 1]$ is given in Figure 1. Having access to the explicit solution is good for testing since it makes it easier to check that our implementation is correct.

2.2 Methods

Before implementing any code we recall the Euler method as well as the fourth order Runge-Kutta method we will use.

[1] A free and open source version of Matlab `https://www.gnu.org/software/octave/index`

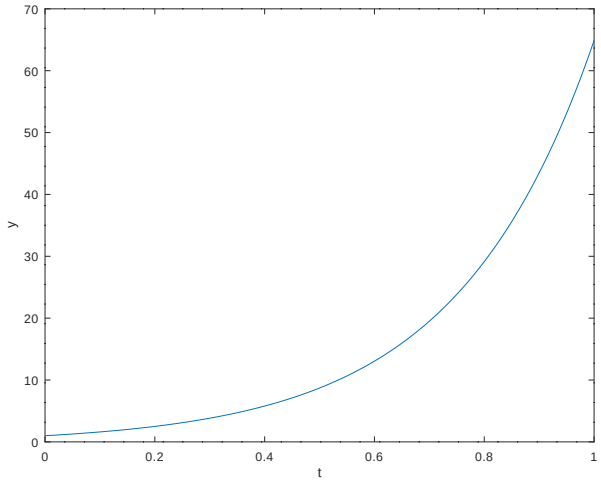

Figure 1: Plot of exact solution to 1

2.2.1 Euler method

1. Subdivide the interval $[t_0, a]$ into n pieces of equal length $h = \frac{a-t_0}{n}$:

$$t_0 < t_1 < t_2 < \cdots < t_{n-1} < t_n = a.$$

2. Given the initial values t_0 and y_0 we can compute

$$y_{i+1} = y_i + f(t_i, y_i)h$$

for $0 \leq i \leq n-1$.

3. For each i, y_i is an approximation of the actual value $y(t_i)$. In particular, y_n is the approximation of $y(t_n) = y(a)$ that we are interested in.

4. In order to improve the accuracy of the approximation we will be increasing n until the desired precision is reached.

2.2.2 Runge-Kutta

The procedure for the Runge-Kutta method is very similar to that for Euler, the only difference is in step 2.

1. Subdivide the interval $[t_0, a]$ into n pieces of equal length $h = \frac{a-t_0}{n}$:

$$t_0 < t_1 < t_2 < \cdots < t_{n-1} < t_n = a.$$

2. Given the initial values of t_0 and y_0 we can compute

$$\begin{aligned} k_{i,1} &= f(t_i, y_i), \\ k_{i,2} &= f(t_i + \frac{1}{2}h, y_i + \frac{1}{2}hk_{i,1}), \\ k_{i,3} &= f(t_i + \frac{1}{2}h, y_i + \frac{1}{2}hk_{i,2}), \\ k_{i,4} &= f(t_i, y_i + hk_{i,3}), \\ y_{i+1} &= y_i + \frac{1}{6}(k_{i,1} + 2k_{i,2} + 2k_{i,3} + k_{i,4})h \end{aligned}$$

   for $0 \leq i \leq n-1$.

3. For each i, y_i is an approximation of the actual value $y(t_i)$. In particular, y_n is the approximation of $y(t_n) = y(a)$ that we are interested in.

4. In order to improve the accuracy of the approximation we will be increasing n until the desired precision is reached.

2.3 Implementation

We demonstrate how to implement both methods and apply them to the sample problem given in Section 2.1 above. We will improve the quality of the approximation by increasing n, the number of subdivision points, as mentioned in Section 2.2.

We give implementation details in Python and Matlab/Octave. You are free to use the one you prefer, or any other programming language you seem fit.

2.3.1 Euler method - Python

An implementation of the Euler method in Python could look something like this:

```
1  def euler(f, a, t0, y0, n):
2    h = (a - t0) / n
3    ts = np.linspace(t0, a, n + 1)
4    ys = np.zeros(n + 1)
5    ys[0] = y0
6    for i in range(n):
7      k = f(ts[i], ys[i])
8      ys[i + 1] = ys[i] + k * h
9
10   return ts, ys
```

Some remarks:

- It takes as input `f, a, t0, y0, n`, corresponding to f, a, t_0, y_0 and n in the Euler method.
- `h = (a - t0) / n` computes the value of h used in the method.
- `ts = np.linspace(t0, a, n + 1)` creates an array of $n+1$ equally spaced points starting at t_0 and ending at a. The value of `ts[i]` corresponds to t_i for $0 \leq i \leq n$.

- `ys = np.zeros(n + 1)` creates an array of length $n+1$, initially filled with zeros. The value of `ys[i]` will after the computations correspond to y_i for $0 \leq i \leq n$.
- `ys[0] = y0` sets the first value in the array equal to y_0.
- `for i in range(n):` is a loop which starts at `i = 0` and ends with `i = n - 1` (notice that it doesn't go all the way up to `n` but stops one step before).
- `k = f(ts[i], ys[i])` computes the value $f(t_i, y_i)$.
- `ys[i + 1] = ys[i] + k * h` corresponds to $y_{i+1} = y_i + f(t_i, y_i)h$.
- Finally `return ts, ys` returns the computed values.

We put this function in a file together with some more code related to our specific sample problem.

```
#!/usr/bin/env python3
import numpy as np
import matplotlib.pyplot as plt

def euler(f, a, t0, y0, n):
  h = (a - t0) / n
  ts = np.linspace(t0, a, n + 1)
  ys = np.zeros(n + 1)
  ys[0] = y0
  for i in range(n):
    k = f(ts[i], ys[i])
    ys[i + 1] = ys[i] + k * h
  return ts, ys

f = lambda t, y: 1 - t + 4 * y
a = 1
t0 = 0
y0 = 1
n = 2

ts, ys = euler(f, a, t0, y0, n)

print("For n = " + str(n) + " Euler gives y(a) = " + str(ys[n]))
plt.plot(ts, ys)
plt.show()
```

Save this in a file called, for example, `warmup-euler.py`. Some remarks:

- `import numpy as np` and `import matplotlib.pyplot as plt` imports the modules Numpy and Matplotlib which we make use of.

- `f = lambda t, y: 1 - t + 4 * y` defines a function `f` which takes `t` and `y` as parameters and returns `1 - t + 4 * y`. This corresponds to our function $f(t, y) = 1 - t + 4y$. We also set the values of a, t_0 and y_0 as given in the sample problem. Finally we set $n = 2$, in order to increase the accuracy of the method we will need to make `n` larger, but for now we start with `n = 2`.
- `ts, ys = euler(f, a, t0, y0, n)` calls our Euler method and returns the times and values it computed.
- Line 24 prints the last computed value.
- `plt.plot(ts, ys)` and `plt.show()` plots the results.

2.3.2 Euler method - Octave/Matlab

Create a new `function` file in Octave/Matlab and type this:

```
1  function [ts, ys] = euler(f, a, t0, y0, n)
2    h = (a - t0) / n;
3    ts = linspace(t0, a, n + 1);
4    ys = zeros(n + 1);
5    ys(1) = y0;
6    for i = 1:n
7      k = f(ts(i), ys(i));
8      ys(i + 1) = ys(i) + k * h;
9    end
10 end
```

Then save the file naming it `euler.m`, note that the name of the file has to be the same as the name of the function. Some general things:

1. Doing this creates a new function called `euler` which Matlab will remember. After saving it into a file we can use this function in any of the script that we will write later on. The function takes `f, a, t0, y0, n` (corresponding to f, a, t_0, y_0, n from Section 2.2.1) as inputs and produces two lists of numbers `ts` (which is the list of $t_0, t_1, \ldots, t_n$) and `ys` (which is the list of $y_0, y_1, \ldots, y_n$).
2. Do not forget the ";" signs in the end of the lines, it makes sure that Matlab doesn't print all the results in the console.

Some remarks about the code:

- It takes as input `f, a, t0, y0, n`, corresponding to f, a, t_0, y_0 and n in the Euler method.
- `h = (a - t0) / n` computes the value of h used in the method.
- `ts = linspace(t0, a, n + 1)` creates an array of $n + 1$ equally spaced points starting at t_0 and ending at a. The vector is indexed from `1` to `n + 1`, the value of `ts(i)` corresponds to t_{i-1} for $1 \leq i \leq n + 1$.

- `ys = zeros(n + 1)` creates an array of length $n+1$, initially filled with zeros. The value of `ys(i)` will after the computations correspond to y_{i-1} for $1 \leq i \leq n+1$.
- `ys(1) = y0` sets the first value in the array equal to y_0.
- `for i = 1:n` is a loop which starts at `i = 1` and ends with `i = n`.
- `k = f(ts(i), ys(i))` computes the value $f(t_{i-1}, y_{i-1})$.
- `ys(i + 1) = ys(i) + k * h` corresponds to $y_i = y_{i-1} + f(t_{i-1}, y_{i-1})h$.

Then create a new `script` file in Matlab and type this:

```
1  f = @(t, y) 1 - t + 4 * y;
2  a = 1;
3  t0 = 0;
4  y0 = 1;
5  n = 2;
6
7  [ts, ys] = euler(f, a, t0, y0, n);
8
9  disp(['for n = ', num2str(n), ' Euler gives ', num2str(ys(end))]);
10 figure(1);
11 plot(ts, ys)
```

Some remarks:

- `f = @(t, y) 1 - t + 4 * y;` defines a function `f` which takes `t` and `y` as parameters and returns `1 - t + 4 * y`. This corresponds to our function $f(t, y) = 1 - t + 4y$. We also set the values of a, t_0 and y_0 as given in the sample problem. Finally we set $n = 2$, in order to increase the accuracy of the method we will need to make `n` larger, but for now we start with `n = 2`.
- `[ts, ys] = euler(f, a, t0, y0, n)` calls our Euler method and returns the times and values it computed.
- Line 9 prints the last computed value. The `disp` function can be used to print anything you like you separate various outputs with commas; any text (string) should be contained in `'`; to output any number you need to convert it into a string using the `num2str` function. Note that y_n (the approximation of $y(a)$) is the last element of `ys`, this we can get with `ys(end)`.
- `plt.plot(ts, ys)` and `plt.show()` plots the results.
- `figure(1)` opens a window for the output of our plot and `plot(ts, ys)` plots the approximation.

2.3.3 Runge-Kutta method - Python

An implementation of the Euler method in Python could look something like this:

```python
def rungekutta(f, a, t0, y0, n):
  h = (a - t0) / n
  ts = np.linspace(t0, a, n + 1)
  ys = np.zeros(n + 1)
  ys[0] = y0
  for i in range(n):
    k1 = f(ts[i], ys[i])
    k2 = f(ts[i] + h / 2, ys[i] + k1 * h / 2)
    k3 = f(ts[i] + h / 2, ys[i] + k2 * h / 2)
    k4 = f(ts[i] + h, ys[i] + k3 * h)
    ys[i + 1] = ys[i] + (k1 + 2 * k2 + 2 * k3 + k4) * h / 6

  return ts, ys
```

The only difference to the Euler method above is what's being done inside the loop. We compute the values of k_1, k_2, k_3 and k_4 first and then use this to set y_{i+1}.

You can put this in a script similar to that for the Euler method, switching the implementation for the Euler method with this implementation of Runge-Kutta, replace the line `ts, ys = euler(f, a, t0, y0, n)` with `ts, ys = rungekutta(f, a, t0, y0, n)` and change to "Runge-Kutta" in the output. You can save this in for example `warmup-rungekutta.py`

2.3.4 Runge-Kutta method - Matlab

Create a new `function` file in Matlab and type this:

```
function [ts, ys] = rungekutta(f, a, t0, y0, n)
    h = (a - t0) / n;
    ts = linspace(t0, a, n + 1);
    ys = zeros(n + 1);
    ys(1) = y0;
    for i = 1:n
      k1 = f(ts(i), ys(i));
      k2 = f(ts(i) + h / 2, ys(i) + k1 * h / 2);
      k3 = f(ts(i) + h / 2, ys(i) + k2 * h / 2);
      k4 = f(ts(i) + h, ys(i) + k3 * h);
      ys(i + 1) = ys(i) + (k1 + 2 * k2 + 2 * k3 + k4) * h / 6;
    end
end
```

Then save the file naming it `rungekutta.m`. The only difference to the Euler method above is what's being done inside the loop. We compute the values of k_1, k_2, k_3 and k_4 first and then use this to set y_{i+1}.

Then create a new `script` file in Matlab and use the same code as in Section but with `rungekutta` instead of `euler`.

2.4 Analysis of sample problem

Running the (Euler and Runge-Kutta) script with $n = 2$ will give you some (terrible) approximation of $y(1)$. We can improve the approximation by increasing n. For both Euler and Runge-Kutta do the following:

1. Run the script with $n = 2$ and record the result.
2. Now change `n = 2` to `n = 2**2` in Python or `n = 2^2` in Octave/Matlab, run the script and record the new result. Notice the change in the graph and the change in the resulting approximation.
3. Then run the script with `n = 2**3` in Python or `n = 2^3` in Octave/Matlab and so on - repeat until you notice that the first two decimals of your resulting approximation stopped changing. This is a good enough sign for our purposes that the approximation has these digits correct.
4. Compare the quality of the Euler and Runge-Kutta methods: which of them produces an accurate approximation quicker?.
5. **Bonus:** Since we know the explicit solution (see Section 2.1) we can compute the error for the different values of n. Do this for the Euler method and look at the quotient between successive errors. Does this quotient tend to anything? How does this compare to the theory?

If you prefer it is likely more convenient to write a loop which automatically tests the methods for a few different values of n.

3 Euler and Runge-Kutta for a system of ODE's

Now consider the initial value problem for the system of two ODEs:

$$\begin{aligned} x' &= f(t, x, y), \quad x(t_0) = x_0, \\ y' &= g(t, x, y), \quad y(t_0) = y_0 \end{aligned}$$

where the functions f, g and the constants t_0, x_0 and y_0 are given. We want to approximate $x(a)$ and/or $y(a)$ up to s decimals, where a is a given point such that $a > t_0$.

3.1 Methods

Again we implement a (forward) Euler method as well as a fourth order Runge-Kutta method. Below is a brief reminder about these methods in the context of systems with two equations.

3.1.1 Euler method

1. Subdivide the interval $[t_0, a]$ into n pieces of equal length $h = \frac{a-t_0}{n}$:

$$t_0 < t_1 < t_2 < \cdots < t_{n-1} < t_n = a.$$

2. Given the initial values t_0, x_0 and y_0 we can compute

$$x_{i+1} = x_i + f(t_i, x_i, y_i)h$$
$$y_{i+1} = y_i + g(t_i, x_i, y_i)h$$

for $0 \leq i \leq n-1$.

3. For each i, x_i is an approximation of the actual value $x(t_i)$. Similarly for each i, y_i is an approximation of the actual value $y(t_i)$. In particular, x_n and y_n are the approximations of $x(t_n) = x(a)$ and $y(t_n) = y(a)$ that we are interested in.

4. In order to improve the accuracy of the approximation we will be increasing n until the desired precision is reached.

3.1.2 Runge-Kutta method

1. Subdivide the interval $[t_0, a]$ into n pieces of equal length $h = \frac{a-t_0}{n}$:

$$t_0 < t_1 < t_2 < \cdots < t_{n-1} < t_n = a.$$

2. Given the initial values t_0, x_0 and y_0 we can

$$\begin{aligned}
k_{i,1} &= f(t_i, x_i, y_i), \\
l_{i,1} &= g(t_i, x_i, y_i), \\
k_{i,2} &= f(t_i + \frac{1}{2}h, x_i + \frac{1}{2}hk_{i,1}, y_i + \frac{1}{2}hl_{i,1}), \\
l_{i,2} &= g(t_i + \frac{1}{2}h, x_i + \frac{1}{2}hk_{i,1}, y_i + \frac{1}{2}hl_{i,1}), \\
l_{i,3} &= g(t_i + \frac{1}{2}h, x_i + \frac{1}{2}hk_{i,2}, y_i + \frac{1}{2}hl_{i,2}), \\
k_{i,3} &= f(t_i + \frac{1}{2}h, x_i + \frac{1}{2}hk_{i,2}, y_i + \frac{1}{2}hl_{i,2}), \\
k_{i,4} &= f(t_i, x_i + hk_{i,3}, y_i + hl_{i,3}), \\
l_{i,4} &= g(t_i, x_i + hk_{i,3}, y_i + hl_{i,3}), \\
x_{i+1} &= x_i + \frac{1}{6}(k_{i,1} + 2k_{i,2} + 2k_{i,3} + k_{i,4})h \\
y_{i+1} &= y_i + \frac{1}{6}(l_{i,1} + 2l_{i,2} + 2l_{i,3} + l_{i,4})h
\end{aligned}$$

for $0 \leq i \leq n-1$. Be **careful** with $k_{i,j}$ and $l_{i,j}$ so that you don't mix them up.

3. For each i, x_i is an approximation of the actual value $x(t_i)$. Similarly for each i, y_i is an approximation of the actual value $y(t_i)$. In particular, x_n and y_n are the approximations of $x(t_n) = x(a)$ and $y(t_n) = y(a)$ that we are interested in.

4. In order to improve the accuracy of the approximation we will be increasing n until the desired precision is reached.

3.2 Implementation

This is part of the project. Use the implementation of the scalar case in Section 2.3 as a reference.

4 Project: Euler and Runge-Kutta for a second order ODE

Consider the following initial value problem

$$\begin{aligned} \theta'' + \sin\theta &= 0 \\ \theta(0) &= \frac{\pi}{4} \\ \theta'(0) &= 0 \end{aligned}$$

which describes the motion of a pendulum as it swings back and forth. Your task is to estimate the value of $\theta(a)$ up to two decimals using both the Euler and the Runge-Kutta method. Take $a = 30$.

4.1 Reduction to a system

By introduction $x(t) = \theta(t)$ and $y(t) = \theta'(t)$ we obtain that the second order ODE is equivalent to the following systems of ODE's

$$\begin{aligned} x' &= y, \\ y' &= -\sin x, \\ x(0) &= \frac{\pi}{4}, \\ y(0) &= 0. \end{aligned}$$

So we are in the setting of Section 3 with $f(t, x, y) = y$ and $g(t, x, y) = -\sin x$.

Your task is to approximate $x(a)$ using the Euler and Runge-Kutta method from Section and respectively.

4.2 Analysis

For the above system and the above value of a, with both the Euler and the Runge-Kutta method, do the following:

1. Run the script with $n = 2$ and record the result.
2. Now change `n = 2` to `n = 2**2` in Python or `n = 2^2` in Octave/Matlab, run the script and record the new result. Notice the change in the graph and the change in the resulting approximation.
3. Then run the script with `n = 2**3` in Python or `n = 2^3` in Octave/Matlab and so on - repeat until you notice that the first two decimals of your resulting approximation stopped changing. This is a good enough sign for our purposes that the approximation has these digits correct.
4. Compare the quality of the Euler and Runge-Kutta methods: which of them produces an accurate approximation quicker?.

5. **Bonus** (no extra points): Compute an approximation of the error for the Euler method for each n by comparing to the result from the Runge-Kutta method using $n = 2^10$. Look at the quotient between successive errors. Does this quotient tend to anything? Compare with the known theory for the Euler method.

4.3 Report structure

Submit a report via Studium, it should be **one** pdf file containing the following:

1. Full code for your function and your script (for both Euler and Runge-Kutta).
2. All of your values of n ($n = 2, 4, 8, \ldots$) and the corresponding approximations of $\theta(a)$ (for both Euler and Runge-Kutta).
3. A plot of your last approximation (for both Euler and Runge-Kutta).
4. Compare the two methods and conclude which one is better (1-2 sentences is enough).
5. **Bonus** (no extra points): Discuss the bonus part mentioned above.

4.4 Common mistakes

Be careful to avoid the following common problems:

1. We are interested in the approximation of $\theta(a)$, **not** of $\theta'(a)$ (what does that mean in terms of x and y?).
2. Be consistent with the order of variables in your function: if $f(t, x, y)$ is defined to have t first, then x and then y, make sure that you use the same order of t, x and y everywhere else.
3. From the theory we know that both Runge-Kutta and Euler methods should eventually give us good approximations (the only question is how fast this happens). So the last approximation for $\theta(a)$ with the Euler method should be fairly close to the last approximation for $\theta(a)$ with the Runge-Kutta method. If this doesn't happen chances are that you have made some mistake.