Exercise 6. Numerical methods

Šimun Šopar

Goal of this exercise is to take a look at a more complex method of solving ordinary differential equations, Runge-Kutta method, solve the problem of harmonic oscillator and compare results to Euler method. This exercise is very similar to Exercise 5, so I will rely on some results written in the last PDF. There are two C++ codes, one for method RK2 (EX6-2.cpp), one for RK4 (EX6.cpp). Also, there are 5 Jupyter Notebooks, where results are plotted and compared to analytical solutions.

Runge-Kutta method

Runge-Kutta method is a numerical method for solving problems $\frac{dy}{dt} = f(y,t)$. In case of higher order differential equation, it can be re-written as a system of first order ODEs, for which RK method is also aplicable. The method is quite similiar to Euler method, they both use Taylor expansion of the function *f* to find unknown function *y* one step at a time. Euler methods approach is to just expand function *y* at the next step (y_{i+1}) over the previous step up to the linear term, which is given by function *f*. RK isn't as straight-forward. The idea is to turn this differential equation into an integral equation, $y(t) = \int_{t_0}^{t} f(y,t')dt'$. Then, by discretizing time by step h, we get the following:

$$y(t + (i+1)h) = y_{i+1} = y_i + \int_{t_i}^{t_{i+1}} f(y,t')dt',$$

and calculate the integral. We do that by using trapezoidal (RK2) or Simpson (RK4) rule around a midpoint $t_i + h/2 = t_{i+1/2}$. This is the main difference between RK and Euler, use of intermediate step.

In case of RK2 method, to calculate the integral we need value of *f* at the midpoint. We can get the midpoint with Euler method, by expanding it in Taylor series: $y_{i+1/2} = y_i + h/2 \cdot f(y_i, t_i)$. Once we get that, we can calculated value of *f* at the midpoint and compute the integral.

RK4 takes it a step further. In Simpson rule, we need value of *f* at the midpoint and at point y_{i+1}. We do the following: first we obtain midpoint value and value of *f* at that point using Taylor expansion. The value of *f* at the midpoint represents the slope at the midpoint. Using that slope we can get an even better estimate of midpoint, from which we calculate a more correct value of *f*. This new value is a better slope at midpoint. Using this slope, we can estimate the value of y_{i+1}. Now that we know those values, we simply use them to calculate the integral.

We expect that Euler method is the least precise, with error proportional to h. RK2 method has error proportional to h^2, while RK4 has one proportional to h^4. We can also expect some slight increase in computation time for RK methods.

Code and results

Like in the previous exercise, I chose initial conditions to be x(0) = 1, v(0) = 0, leading to analytical solution $x(t) = cos(\omega t)$. Also, like in the last exercise, I timed computation number for varying N, those results can be seen in Table 1.

N	Euler	RK2	RK4
1000	0.0s	0.0s	0.0s
5000	0.0s	0.0s	0.0s
10000	0.001s	0.001s	0.002s
50000	0.005s	0.006s	0.007s

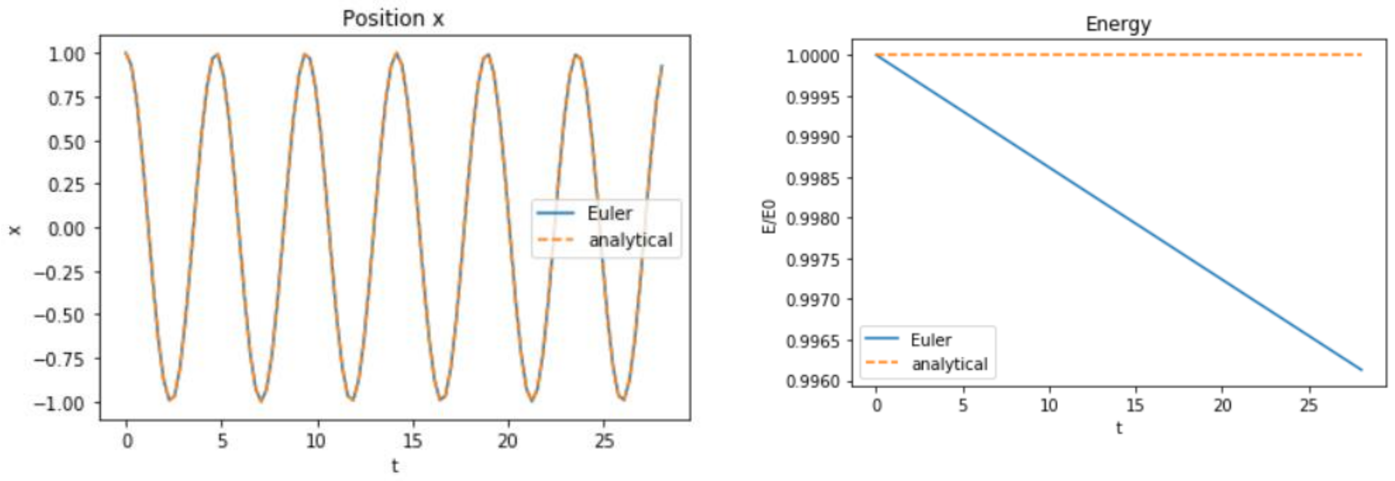
Table 1. Computation time for different methods

We can see, neither of the methods is significantly faster. As expected, RK4 is the slowest, but the difference is not by much, meaning that when it comes to choosing which method of these to use, time doesn't play an important role.

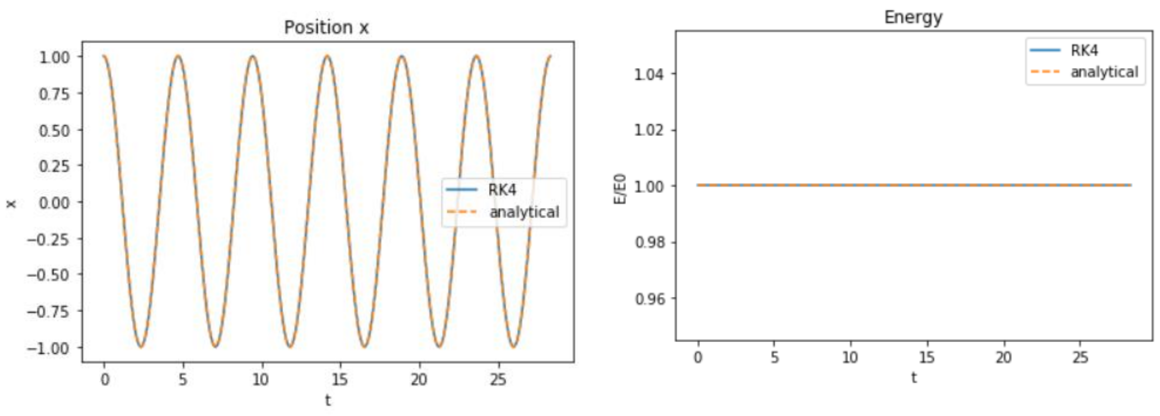
The results of computing are saved in a text file and copy-pasted to Python Notebook where they are plotted and compared to analytical solution.

Unlike the last exercise, there are now 5 .ipynb files. First three are for comparing RK4 values to analytical for values of N = 1000, 5000, 10000. As we will see below, results are very satisfying for these N, so I decided to calculate and plot the solution for one small N, N = 100, plots are given in fourth file. In fifth file I compare Euler, RK2 and RK4 method for N = 1000.

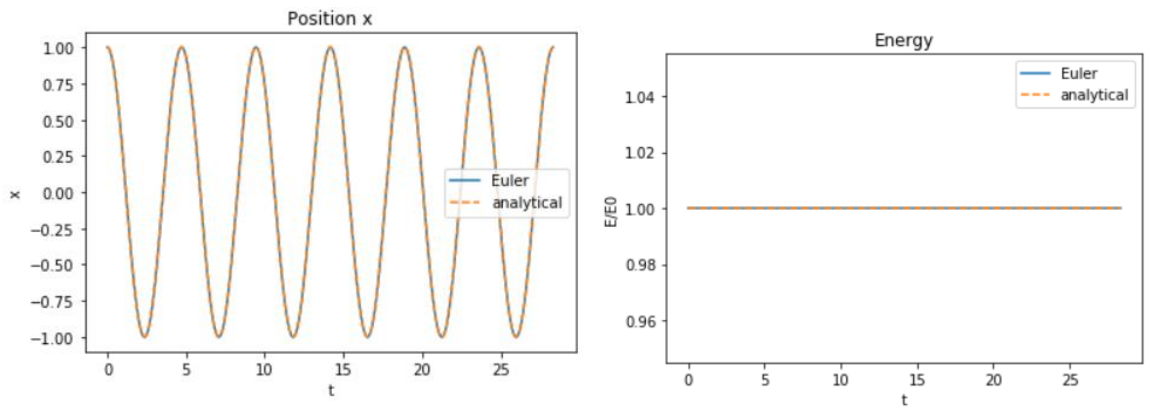
Next graphs show position and energy for N = 100, N = 1000 and N = 10000. We can see that for 1000 and 10000 the results look the same. The energy is conserved and the graphical solution looks the same as analytical solution. Since these results were so good, I decided not to plot for N = 50000, since it wouldn't be much of a difference. Instead; I calculated for a small N. We remember from previous exercise that Euler method for 1000 steps was quite imprecise, so for 100 steps it would give a very bad solution. In case RK4, while the solution isn't correct, we can see that the energy isn't conserved, but the results are still pretty precise. Comparing RK4 for 100 steps and Euler for 1000, we can say that RK4 is better, with such a large difference in steps. This goes to show just how imprecise Euler method is and how much a better method can help in searching for solution.



Graph 1. Positon and energy for N=100 (RK4)

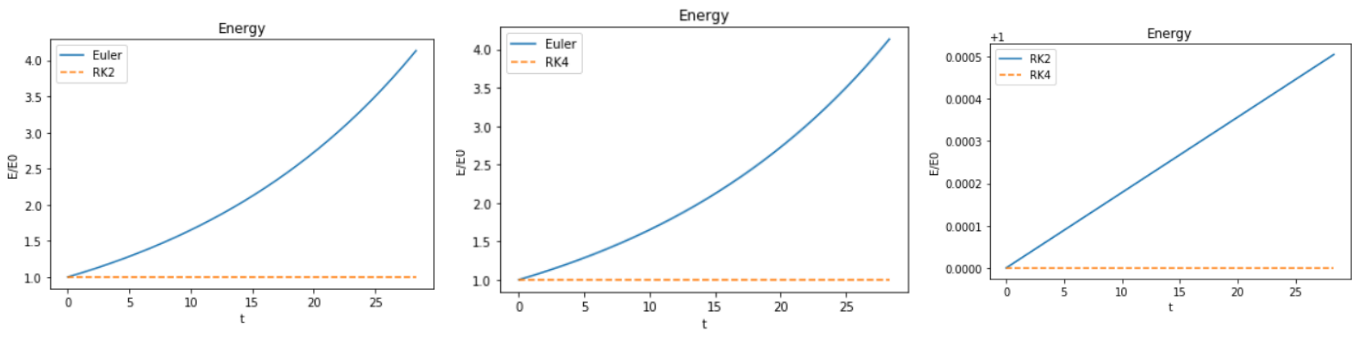


Graph 2. Position and energy for *N* = 1000 (RK4)



Graph 3. Position and energy for *N* = 10000 (RK4)

Graphs 4 show comparison in energy between the methods for 1000 steps. As expected, Euler method is the least precise and RK4 is the most precise. However, RK2 proved to be sufficiently precise as well, which we can see from the graphs. Although RK4 always gives the same initial energy, RK2 doesn't differ that much, it has a rise in energy, but that rise is small.



Graph 4. Comparsion between methods for *N* = 1000. More can be seen in Python file EX6-5.ipynb

To conclude, RK methods are way better than Euler method in computing ODEs. RK2 proved to be quite precise, and RK4 is even more precise. For the best solution, we should use RK4. We could define higher order RK methods, but for most problems, RK4 would be enough.