Scientific Computing: Initial Value Problems

Nathan Browne ge18723

March 1, 2021

The components of this report are organised according to the same convention given in the Assignment Description.

1 Using the Euler method to solve dxdt = x

1.1 Euler Step

Function euler_step is built in ODE_Utils.py. It takes:

- a function 'func' which describes the first order differential system
- the nth value of the independent variable 't'
- the nth values of the dependent variables as a numpy array named 'vect'
- the stepsize 'h' to be used in this step of the solve

It returns:

- the (n+1)th value of the independent variable 't', incremented once by h
- the (n+1)th values of the dependent variables as a numpy array named 'vect'

1.2 Solve to function

Function solve_to is built in ODE_Utils.py. It takes:

- a function 'func' which describes the first order differential system
- the initial value of the independent variable 't1'
- the final value of the independent variable 't2'
- the initial values of the dependent variables as a numpy array named 'vect'
- the maximum stepsize 'deltat_max' to be used in any step of the solve
- a string 'method' which describes an integration method to use for the solve. At this point, the only valid input is 'Euler'

It returns:

- a list 'tl' of independent variable values between our integration bounds

- a list ‘vl’ of lists containing the dependent variable values for each corresponding independent variable value[1]

It is also worth mentioning that two steps have been taken within this function to handle manipulation of the step size. Firstly, I have determined it to be logical to re-scale all steps in the integration to equal sizes. So, before any solving occurs, the variable ‘deltat_max’ is manipulated according to the following example: We wish to solve between time 1 and time 11 in steps no greater than 0.6. First, we divide the size of the time bound by our stepsize: (11 - 1)/0.6 gives 16.666... We then take the ceiling function of this value to get 17. Our new stepsize is equal to (11 - 1)/16.666... which gives 0.58823. Of course, due to rounding errors in python, this will still not exactly hit 11 at the final step, but it will be very close. A final conditional statement means that if another step can’t fit exactly, the new step size is set to the exact value that brings us to the final bound.

1.3 Solve ODE function

Function solve_ode is built in ODE_Utils.py. It takes:

- a function ‘func’ which describes the first order differential system
- the initial value of the independent variable ‘t1’
- the final value of the independent variable ‘t2’
- the initial values of the dependent variables as a numpy array named ‘v0’
- a list of maximum step sizes ‘stepsizes’ to be used in each solving of the system
- a string ‘method’ which describes an integration method to use for the solve. At this point, the only valid input is ‘Euler’

It returns:

- a list ‘tls’ of independent variable values between our integration bounds for each max step size
- a list ‘sols’ of lists containing the dependent variable values for each corresponding independent variable value and each corresponding max step size

```
$ python Solver.py
Final sol by  Euler  with stepsize  1 :  2
Final sol by  Euler  with stepsize  0.1 :  2.5937424601
Final sol by  Euler  with stepsize  0.01 :  2.6780334944767583
Final sol by  Euler  with stepsize  0.001 :  2.7142097225133828
Final sol by  Euler  with stepsize  0.0001 :  2.7181459268252266
```

Figure 1: *Euler method values at t = 1*

[1]It may become beneficial later on to return a singular Pandas DataFrame object, to help with making calls to specific variables if we wish to graph them easily.

1.4 Error for step size

For this comparison, we have chosen step sizes varying by orders of magnitude, shown in Figure 2 leading to solutions in 1.

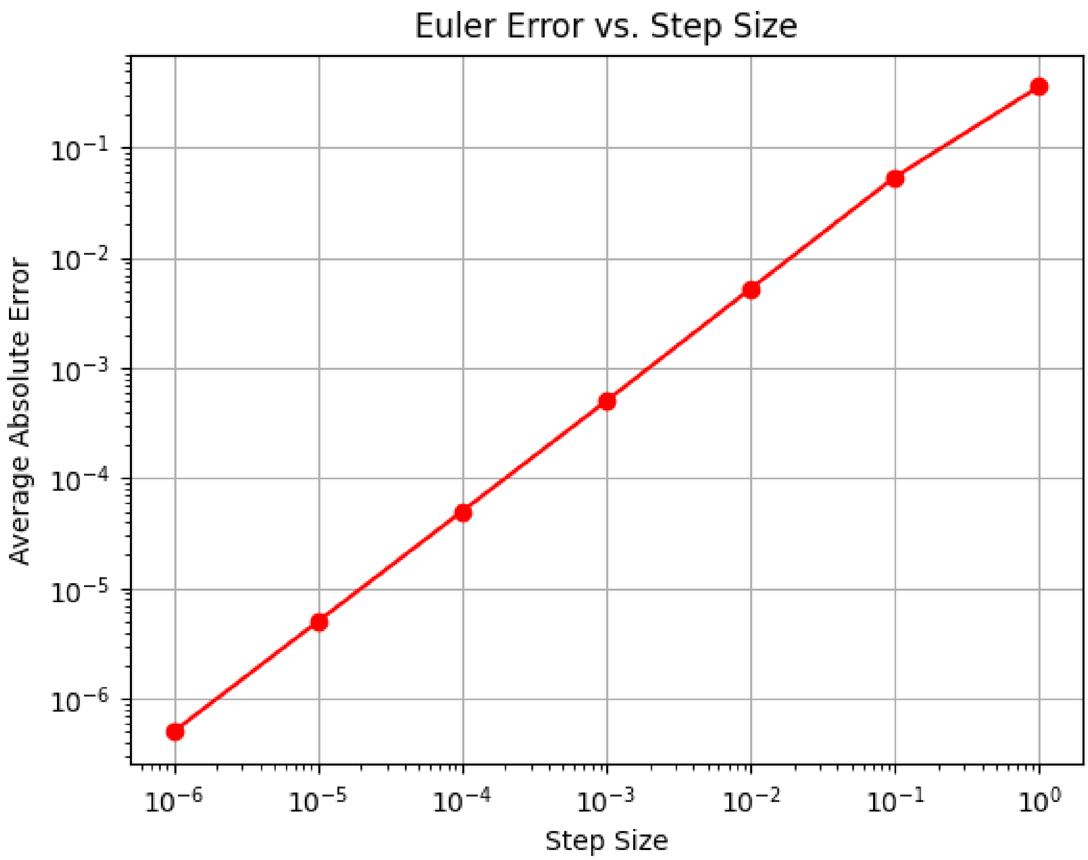
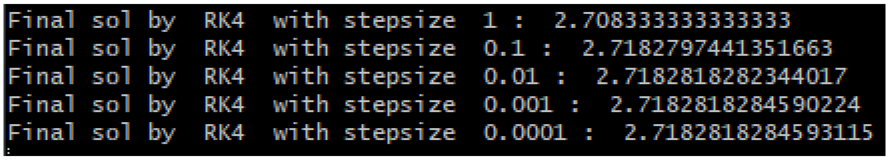


Figure 2: *Euler method error vs step size*

2 4th Order Runge-Kutta method

2.1 Preference handling

The function solve_ode now performs a different operation if the string 'RK4' is passed as a 'method' argument. If no argument is given, it defaults to Runge-Kutta anyway.



```
Final sol by  RK4  with stepsize  1 :  2.708333333333333
Final sol by  RK4  with stepsize  0.1 :  2.7182797441351663
Final sol by  RK4  with stepsize  0.01 :  2.7182818282344017
Final sol by  RK4  with stepsize  0.001 :  2.7182818284590224
Final sol by  RK4  with stepsize  0.0001 :  2.7182818284593115
```

Figure 3: *RK4 method values at $t = 1$, as we can see it converges much faster, as expected*

2.2 Error for step size

For this comparison, we have chosen step sizes varying by orders of magnitude, shown in Figure 4 leading to solutions in 1.

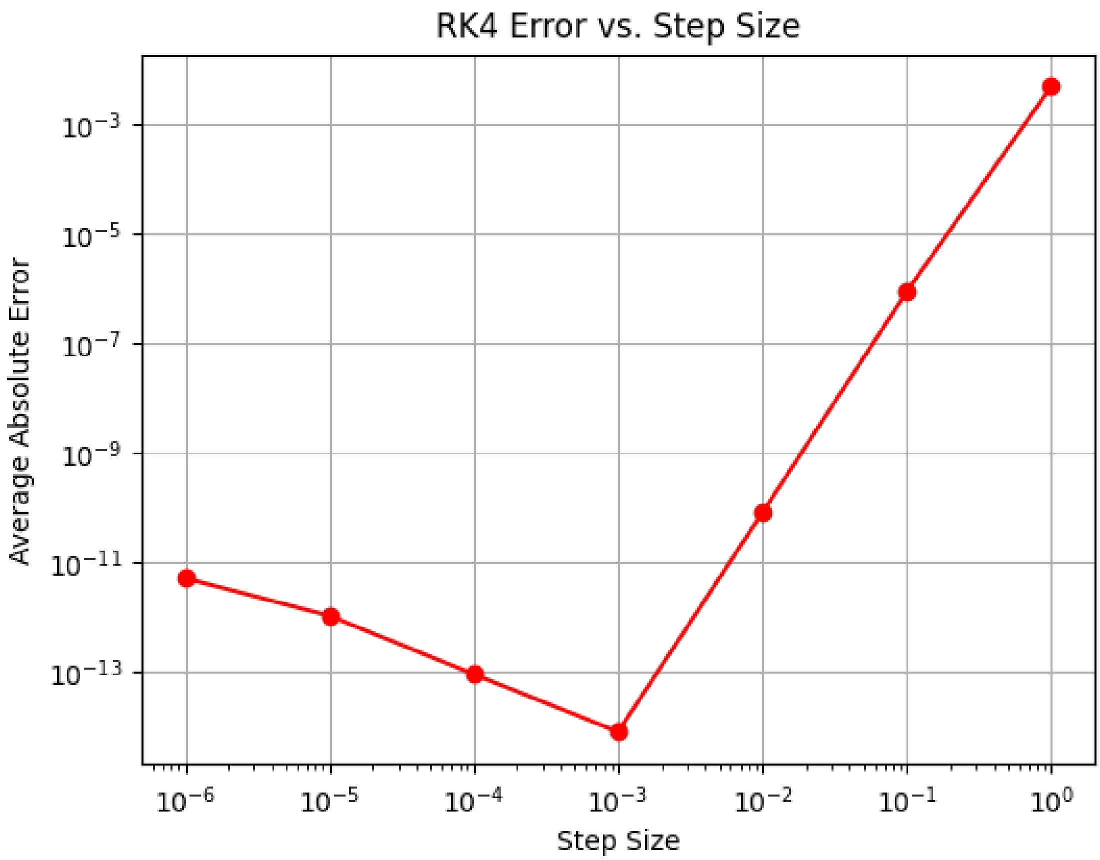


Figure 4: *RK4 method error vs step size. As we can see, it reaches a much smaller error than the Euler method, but truncation in the python operations leads to incorrect error calculation with steps smaller than 0.0001*

2.3 Error Comparison of Euler and Runge-Kutta methods

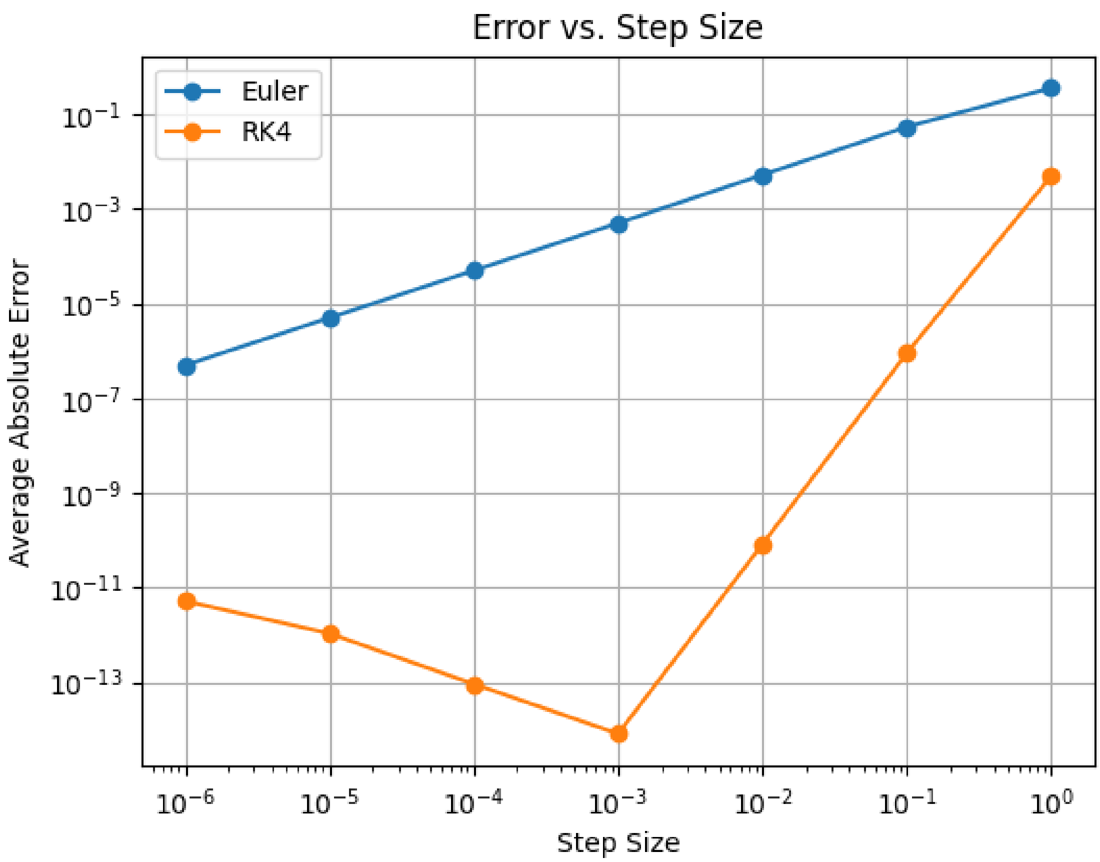
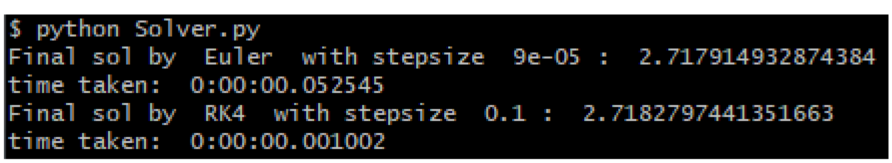


Figure 5: *Comparison of error for stepsize using each method*

As we can see, an Euler step of 9e-5 has an error similar to RK4 steps of 1e-1. Let's see how quickly they arrive at solutions:



```
$ python Solver.py
Final sol by  Euler  with stepsize  9e-05 :  2.717914932874384
time taken:  0:00:00.052545
Final sol by  RK4  with stepsize  0.1 :  2.7182797441351663
time taken:  0:00:00.001002
```

Figure 6: *As we can see, the Runge Kutta method is more efficient for reaching the same level of error. In this case, it is roughly 50 times faster.*

3 Solving ODE systems

At this point, our module is already created able to handle ODE systems so long as they are sufficiently defined in a singular python function.

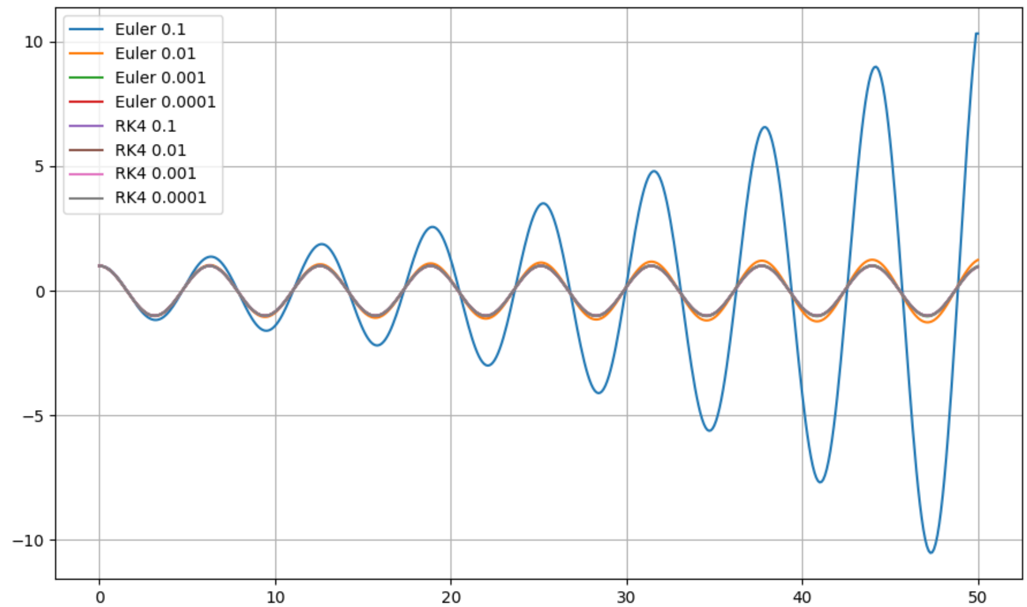


Figure 7: *When solving this second order differential equation, the errors increase through the solve, making the solution diverge from the analytic solution*

If we were to plot the equation of xdot versus x, we would expect a circle, making the error more apparent:

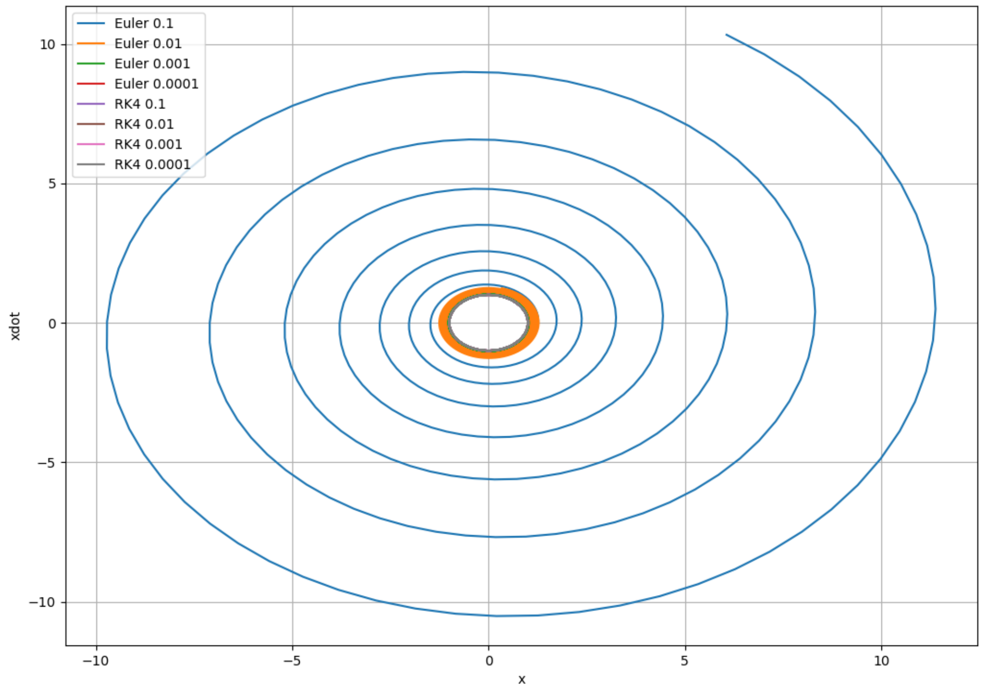


Figure 8: *The error becomes more obvious in this plot of xdot and x*