P346 : DIY Project
Orbits with Adaptive Step Size RK4

Abhinav Roy
Roll Number 2011003
SPS, Batch 20

Ordinary differential equations (ODEs) can be used to model a wide variety of fascinating and complex physical systems. Thus the challenge of modelling the physical system can be reduced to solving a set of coupled first-order differential equations since it is always possible to convert an ordinary differential equation into a set of these equations. In this project, we aim to code the numerical integrator with adaptive step-size RKF45 using the Cash-Karp parameters in order to model our physical system, the motion of a planet around a sun.

I. Introduction

The trajectory of orbits can be determined over time by propagating them by use of numerical integrators. The numerical integrators approximate a solution to a nonlinear ODE initial value problem. As an example, consider a general second order equation,

$$p(x)\frac{d^2y}{dx^2} + q(x)\frac{dy}{dx} = r(x)$$

This can be rewritten as,

$$\frac{dy}{dx} = z(x)$$

$$\frac{dz}{dx} = \frac{r(x) - q(x)z(x)}{p(x)}$$

Therefore, the general set of equations to solve will be

$$\frac{dy_i(x)}{dx} = f_i(x, y_1, ...y_i, ...y_N)$$

where i runs from 1 to N.
In the model presented here, we used the motion of a body around another as our physical system. We assume an initial state and evolve it forward in time until a specified final time. In this example, Newton's equation of motion is

$$\frac{d^2r}{dt^2} = -\frac{GM}{r^2}$$

Rewriting them and working in the x, y and z components, we obtain

$$\frac{dx}{dt} = v_x(t) \ ; \ \frac{dv_x}{dt} = -\frac{GMx}{r^3}$$

$$\frac{dy}{dt} = v_y(t) \ ; \ \frac{dv_y}{dt} = -\frac{GMy}{r^3}$$

$$\frac{dz}{dt} = v_z(t) \ ; \ \frac{dv_z}{dt} = -\frac{GMz}{r^3}$$

where,

$$r = \sqrt{x^2 + y^2 + z^2}$$

Now that we have our equations of motion, we need to convert them to a state form that can be numerically integrated. The state form includes the position and velocity vector at a certain time. Here is the state and its time derivative.

$$\begin{Bmatrix} x \\ y \\ z \\ \dot{x} \\ \dot{y} \\ \dot{z} \end{Bmatrix} = \begin{Bmatrix} \dot{x} \\ \dot{y} \\ \dot{z} \\ \ddot{x} \\ \ddot{y} \\ \ddot{z} \end{Bmatrix}$$

This state time derivative will be made into a function that will be numerically integrated. For the integration, we will write a fourth-order Runge-Kutta with variable step size and a fourth-order Runge-Kutta with fixed step size. The methods will be fed the same initial conditions.The Runge-Kutta methods are described below along with how they are implemented in the code.

II. Method

The first method implemented is the fourth-order Runge-Kutta method with fixed step size, which is a method that allows us to solve complicated ODE with relative fast computation and with high accuracy. The form for the fourth-order Runge-Kutta for any variable y_i in our system will be

$$k1_i = \Delta t\ f_i(x, y_1, ...y_i, ...y_n)$$

$$k2_i = \Delta t\ f_i(x + \frac{\Delta t}{2}, y_1 + \frac{k1_1}{2}, ...y_i + \frac{k1_i}{2}, ...y_n + \frac{k1_n}{2})$$

$$k3_i = \Delta t\ f_i(x + \frac{\Delta t}{2}, y_1 + \frac{k2_1}{2}, ...y_i + \frac{k2_i}{2}, ...y_n + \frac{k2_n}{2})$$

$$k4_i = \Delta t\ f_i(x + \Delta t, y_1 + k3_1, ...y_i + k3_i, ...y_n + k3_n)$$

$$y_i += \frac{k1_i}{6} + \frac{k2_i}{3} + \frac{k3_i}{3} + \frac{k4_i}{6}$$

This is represented graphically in Figure 1.

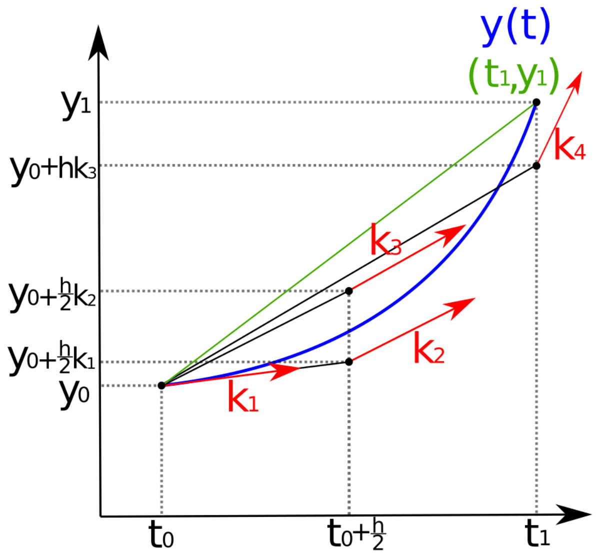
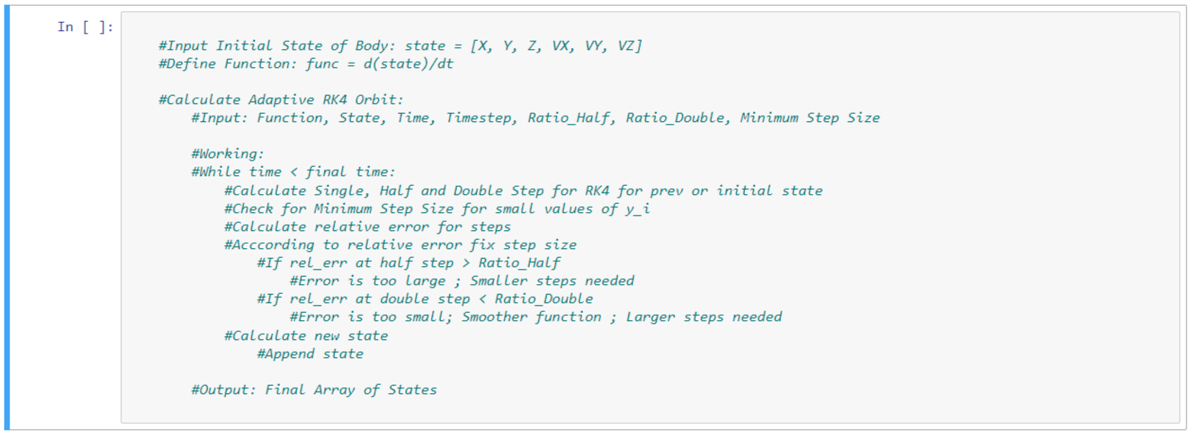


Figure 1. The fourth-order Runge-Kutta evaluates the derivative four times over a given time step. By choosing the weights for each derivative, lower orders of error can be canceled out.

In adaptive method, we shall calculate a half step for $\frac{dt}{2}$ and a double step for $2 * dt$, and then adjust the time step dt according to the relative error of the half and double steps.
Smaller step size are used in regions where the solution is tricky and error's magnitude is low, but one has to be careful because if smaller steps are used, then more steps are needed and hence the error will pile up. While a bigger step size could miss an important detail. Therefore, the adaptive step is a game of balancing between accuracy and time of computing. This in practise means that calculating the error on the steps and compare that error to desire accuracy and either increase or decrease the step to achieve the desired accuracy.

III. Pseudo-Code

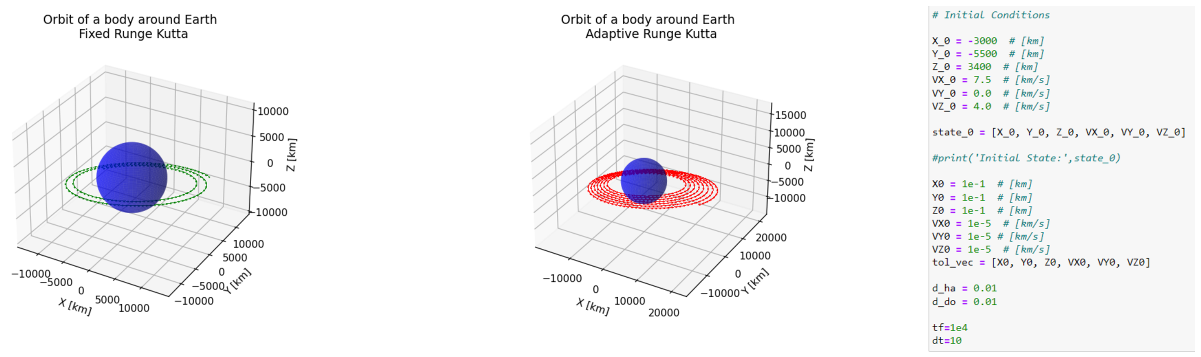


```
In [ ]:
#Input Initial State of Body: state = [X, Y, Z, VX, VY, VZ]
#Define Function: func = d(state)/dt

#Calculate Adaptive RK4 Orbit:
    #Input: Function, State, Time, Timestep, Ratio_Half, Ratio_Double, Minimum Step Size

    #Working:
    #While time < final time:
        #Calculate Single, Half and Double Step for RK4 for prev or initial state
        #Check for Minimum Step Size for small values of y_i
        #Calculate relative error for steps
        #Acccording to relative error fix step size
            #If rel_err at half step > Ratio_Half
                #Error is too large ; Smaller steps needed
            #If rel_err at double step < Ratio_Double
                #Error is too small; Smoother function ; Larger steps needed
        #Calculate new state
            #Append state

    #Output: Final Array of States
```

Figure 2. The pseudo-code for the solution of the adaptive step size orbital motion in multiple dimensions.

IV. Results

For the first run the initial conditions were set to X = -3000 kms, Y = -5500 kms, Z = 3400 kms, VX = 7.5 km/s, VY = 0.0 km/s, VZ = 4.0 km/s and T_i = 0.0. $Ratio_{Double}$ and $Ratio_{Half}$ were set to 0.01. The simulation was run until T_f = 1000. The step size was set to 10 for the fixed step Runge-Kutta. The same step size was suggested for the adaptive step size Runge Kutta. The plots produced are shown below.



```
# Initial Conditions

X_0 = -3000  # [km]
Y_0 = -5500  # [km]
Z_0 = 3400  # [km]
VX_0 = 7.5  # [km/s]
VY_0 = 0.0  # [km/s]
VZ_0 = 4.0  # [km/s]

state_0 = [X_0, Y_0, Z_0, VX_0, VY_0, VZ_0]

#print('Initial State:',state_0)

X0 = 1e-1  # [km]
Y0 = 1e-1  # [km]
Z0 = 1e-1  # [km]
VX0 = 1e-5  # [km/s]
VY0 = 1e-5 # [km/s]
VZ0 = 1e-5  # [km/s]
tol_vec = [X0, Y0, Z0, VX0, VY0, VZ0]

d_ha = 0.01
d_do = 0.01

tf=1e4
dt=10
```

Figure 3. Plots generated for Adaptive and Fixed Runge Kutta with the initial parameters anda tolerance parameters.

The plots were similar and further runs for the adaptive step size were not successfully obtained due to mismatch in error and tolerance values. Any further plots, if obtained will be attached in the github folder.

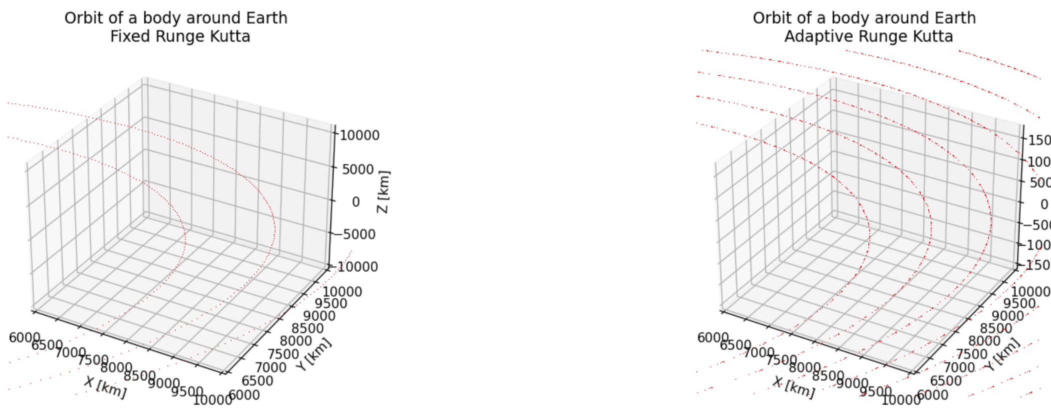


Figure 4. Closer look at the plots generated for Adaptive and Fixed Runge Kutta.

V. Unused RK Method

In this unused method, a fifth-order Runge-Kutta that requires six evaluations of the function is also used to calculate a fourth order Runge-Kutta with the same six evaluations, but in a different combination. The form for the fifth order Runge-Kutta for any variable y_i in our system will be

$$k1_i \;=\; \Delta t \; f_i(x, y_1, ...y_i, ...y_n)$$

$$k2_i \;=\; \Delta t \; f_i(x + a_2\Delta t, \; ...y_i + b_{21}k1_i, ...)$$

$$...$$

$$k6_i \;=\; \Delta t \; f_i(x + a_6\Delta t, y_i + b_{61}k1_i + b_{62}k2_i + b_{63}k3_i + b_{64}k4_i + b_{65}k5_i, ...)$$

$$y_i \mathrel{+}= \; c_1k1_i + c_2k2_i + c_3k3_i + c_4k4_i + c_5k5_i + c_6k6_i$$

The embedded fourth-order formula for the variable y_i will be

$$y*_i \;\; += \;\; c*_1\;k1_i + c*_2\;k2_i + c*_3\;k3_i + c*_4\;k4_i + c*_5\;k5_i + c*_6\;k6_i$$

The difference between these two answers gives an estimate of the error. The coefficients are listed in Table 1.

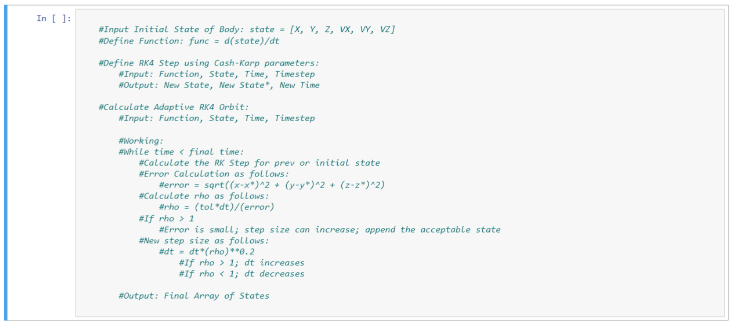
Table 1. List of the coefficients for the embedded Runge-Kutta method

Cash-Karp Parameters for Runge-Kutta Method								
i	a_i	b_{ij}					c_i	$c*_i$
1							37/378	2825/27648
2	1/5	1/5					0	0
3	3/10	3/40	9/40				250/621	18575/48384
4	3/5	3/10	-9/10	6/5			125/594	13525/55296
5	1	-11/54	5/2	-70/27	35/27		0	277/14336
6	7/8	1631/55296	175/512	575/13824	44275/110592	253/4096	512/1771	1/4
	j =	1	2	3	4	5		

Once the error $\Delta = y_{n+1} - y*_{n+1}$ has been calculated, the step is rescaled. The relation between Δ_1 corresponding to a step size Δt_1 and Δ_2 corresponding to a step size Δt_2 is given by

$$\Delta t_1 \;=\; \Delta t_2 \left| \frac{\Delta_1}{\Delta_2} \right|^{0.2}$$

The time steps are then adjusted to maintain Δ within a prescribed level of accuracy. I originally hoped to use this method to model my physical system however I was not able to figure out the error parameters and tolerance values for this method.



```
In [ ]:
#Input Initial State of Body: state = [X, Y, Z, VX, VY, VZ]
#Define Function: func = d(state)/dt

#Define RK4 Step using Cash-Karp parameters:
    #Input: Function, State, Time, Timestep
    #Output: New State, New State*, New Time

#Calculate Adaptive RK4 Orbit:
    #Input: Function, State, Time, Timestep

    #Working:
    #While time < final time:
        #Calculate the RK Step for prev or initial state
        #Error Calculation as follows:
            #error = sqrt((x-x*)^2 + (y-y*)^2 + (z-z*)^2)
        #Calculate rho as follows:
            #rho = (tol*dt)/(error)
        #If rho > 1
            #Error is small; step size can increase; append the acceptable state
        #New step size as follows:
            #dt = dt*(rho)**0.2
                #If rho > 1; dt increases
                #If rho < 1; dt decreases

    #Output: Final Array of States
```

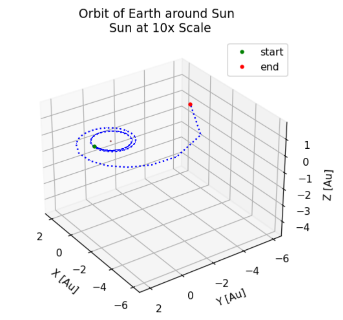


Figure 5. A look at the pseudo code and the plot generated by the unused Adaptive Runge Kutta.

VI. Conclusion

The plots for the orbits of the body about Earth matched the expected outcomes. However a few discrepancies I could note down was higher step count and time taken by Adaptive Runge Kutta method. It might be due to the stricter tolerance values for the method. However a higher number of orbits were observed when tolerance values were loosened. It is puzzling and I'm still trying to make sense of it.
I tried to compensate for the step doubling adaptive Runge Kutta method with a more efficient embedded Runge Kutta however it still needs to be debugged. The methods were fairly easy to implement, figuring out the vector form of the state and functions was a bit tricky. However due to large floating point errors, (due to very small values of error and tolerence being used and a large number of steps due to which errors piled up), debugging the code was a bit tricky. I tried to compensate for it by artificially introducing error in distance calculations which was a little successful, however still not presentable.

References

- https://en.wikipedia.org/wiki/Runge-Kutta_methods
- https://en.wikipedia.org/wiki/Cash-Karp_method
- Hruby, Robert. (2019). Cometary Orbits With Runge Kutta 4nd order adaptive step size.