Physics 607 Project 1 Report: Errors in Integrations

John Batarekh

Oct 3 2023

1 ODE's

1.1 Damped Oscillator

To analyze the the errors in integration of ODE's I have chosen the problem of the damped oscillator. The damped oscillator takes the form $\ddot{x} + 2\gamma\dot{x} + \omega_0^2 x = 0$ where ω_0 is the natural frequency of oscillation and γ is the damping factor. I chose this problem in part because there are not a lot of ODE's with simple analytic solutions, but the damped oscillator has analytic solution:

$$x(t) = A_1 e^{\left[-\gamma+\sqrt{\gamma^2-\omega_0^2}\right]t} + A_2 e^{\left[-\gamma-\sqrt{\gamma^2-\omega_0^2}\right]t} \tag{1}$$

A_1 and A_2 are constants determined by the initial conditions. This problem is also nice because there are several distinct domains of γ that can be analyzed.

1.2 Euler's Method

Euler's method is a type of step-wise manual integration by which we calculate values of x by expanding it in a taylor's series such that $x(t) \approx x(t_0) + (t - t_0)\dot{x}(t_0) + 1/2(t - t_0)^2\ddot{x}(t_0)$ since our differential equation is of second order we will also expand $\dot{x}$ as $\dot{x}(t) \approx \dot{x}(t_0) + (t - t_0)\ddot{x}(t_0)$. We will also require an initial condition for both the position and the velocity. The procedure for calculating each consecutive step will go as follows. Starting with our initial conditions, calculate $\ddot{x}(t_0) = -2\gamma\dot{x}(t_0) - \omega_0^2 x(t_0)$. With $\ddot{x}(t_0)$, we can then calculate $x(t)$ and $\dot{x}(t)$ where $t - t_0 = dt$ and dt is a small step size. Then we change t into t_0 and calculate $\ddot{x}(t_0)$ again the same way. After that we progress the system by dt again and we can repeat these two steps until we reach an end time.

1.3 4th Order Runge-Kutta Method

Without explaining the theory behind the Runge-Kutta method, the general procedure goes as follows. We first calculate 4 slopes at different locations

$$\begin{aligned}
k_1 &= \dot{x}(t_0, x(t_0)) \\
k_2 &= \dot{x}(t_0 + dt/2, x(t_0) + k_1 \cdot dt/2) \\
k_3 &= \dot{x}(t_0 + dt/2, x(t_0) + k_2 \cdot dt/2) \\
k_4 &= \dot{x}(t_0 + dt/2, x(t_0) + k_3 \cdot dt)
\end{aligned} \tag{2}$$

Using these slopes we can calculate $x(t_0+dt) = x(t_0)+\frac{1}{6}(k_1+2k_2+2k_3+k_4)$. For the problem of oscillators, the differential equation is second order. Since the definition of runge-kutta only solves first order problems, I defined $y(t, x) = \frac{dx}{dt}$, this creates a pair of coupled differential equations at first order:

$$\begin{aligned}
\frac{dx}{dt} &= y \\
\frac{dy}{dt} &= -2\gamma y - \omega_0^2 x
\end{aligned} \tag{3}$$

In this form, we will be able to apply 4th order runge-kutta, unfortunately, it means we need to calculate twice as many slopes. We also need to generalize these slopes in terms of x and y.

$$\begin{aligned}
&k_1 = \dot{x}(t_0, x(t_0), y(t_0)) = y(t_0) \\
&l_1 = \dot{y}(t_0, x(t_0), y(t_0)) = -2\gamma y(t_0) - \omega_0^2 x(t_0) \\
&k_2 = \dot{x}(t_0 + dt/2, x(t_0) + k_1 \cdot dt/2, y(t_0) + l_1 \cdot dt/2) = y(t_0) + l_1 \cdot dt/2 \\
&l_2 = \dot{y}(t_0 + dt/2, x(t_0) + k_1 \cdot dt/2, y(t_0) + l_1 \cdot dt/2) = -2\gamma(y(t_0) + l_1 \cdot dt/2) - \omega_0^2(x(t_0) + k_1 \cdot dt/2) \\
&k_3 = \dot{x}(t_0 + dt/2, x(t_0) + k_2 \cdot dt/2, y(t_0) + l_2 \cdot dt/2) = y(t_0) + l_2 \cdot dt/2 \\
&l_3 = \dot{y}(t_0 + dt/2, x(t_0) + k_2 \cdot dt/2, y(t_0) + l_2 \cdot dt/2) = -2\gamma(y(t_0) + l_2 \cdot dt/2) - \omega_0^2(x(t_0) + k_2 \cdot dt/2) \\
&k_4 = \dot{x}(t_0 + dt, x(t_0) + k_3 \cdot dt, y(t_0) + l_3 \cdot dt) = y(t_0) + l_3 \cdot dt \\
&l_4 = \dot{y}(t_0 + dt, x(t_0) + k_3 \cdot dt, y(t_0) + l_3 \cdot dt) = -2\gamma(y(t_0) + l_3 \cdot dt) - \omega_0^2(x(t_0) + k_3 \cdot dt)
\end{aligned} \tag{4}$$

So in practice, we would start with our initial conditions and use them to calculate the k's and l's. With the slopes, we can move forward step dt

$$\begin{aligned}
x(t_0 + dt) &= x(t_0) + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) \\
y(t_0 + dt) &= y(t_0) + \frac{1}{6}(l_1 + 2l_2 + 2l_3 + l_4)
\end{aligned} \tag{5}$$

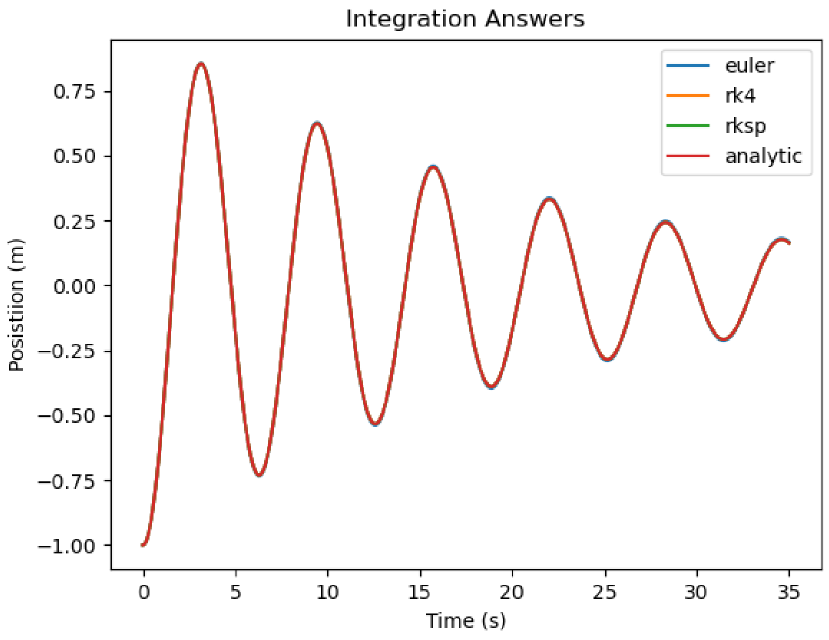
We can then set $x(t_0 + dt) = x(t_0)$ and $y(t_0 + dt) = y(t_0)$ and calculate new slopes and progress another step dt, we can repeat this procedure for N steps until we reach the end time.

1.4 SciPy Runge-Kutta

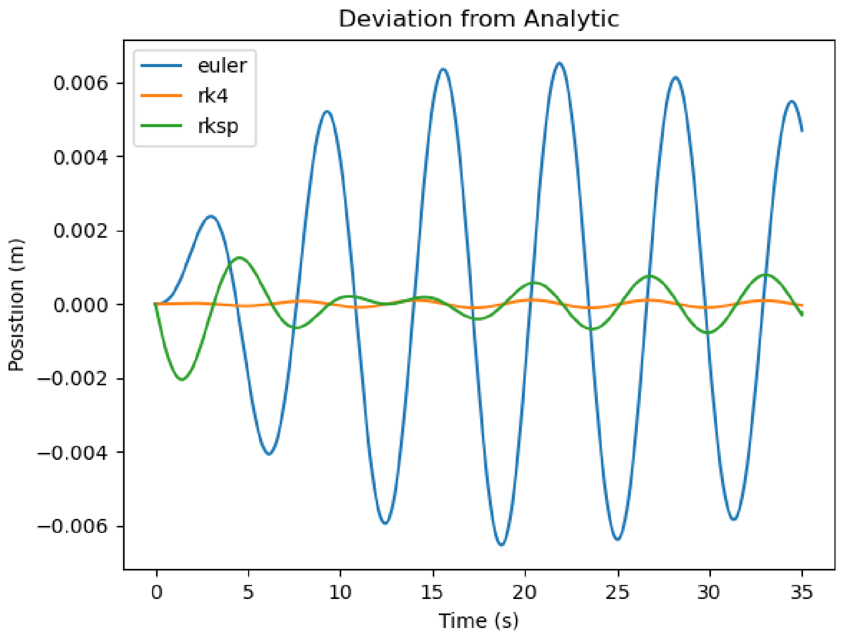
It is also possible to use the scipy python library to do the 4th order Runge-Kutta using the scipy.integrate.solve_ivp function. The solve_ivp function defaults its integration type to 4th order Runge-Kutta, it takes 3 required arguments: function(s), time boundaries, and initial condition(s). For multiple functions, solve_ivp accepts a list of differential equations, so I defined a function that created a list that looks like eq. (3). An optional argument that I also included was max_step. This argument allowed me to add resolution to the answer.

1.5 Oscillator Analysis

After creating a method for calculating the analytic expression, I created an oscillator with $2\gamma = .1$, $\omega_0^2 = 1$. I then plotted all the manual integration solutions and analytic solution on the same plot.

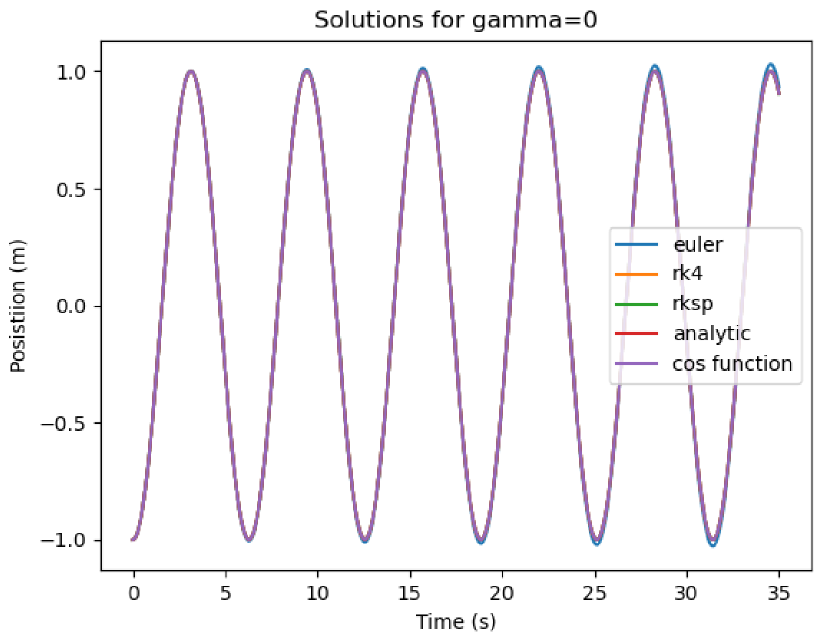


This plot shows that the solutions to the manual integrations are all overlapped with the analytic answer. The y-axis of this plot is on the order of a meter so to check for small deviations I subtracted each integration method from the analytic expression to see the magnitudes of the errors.

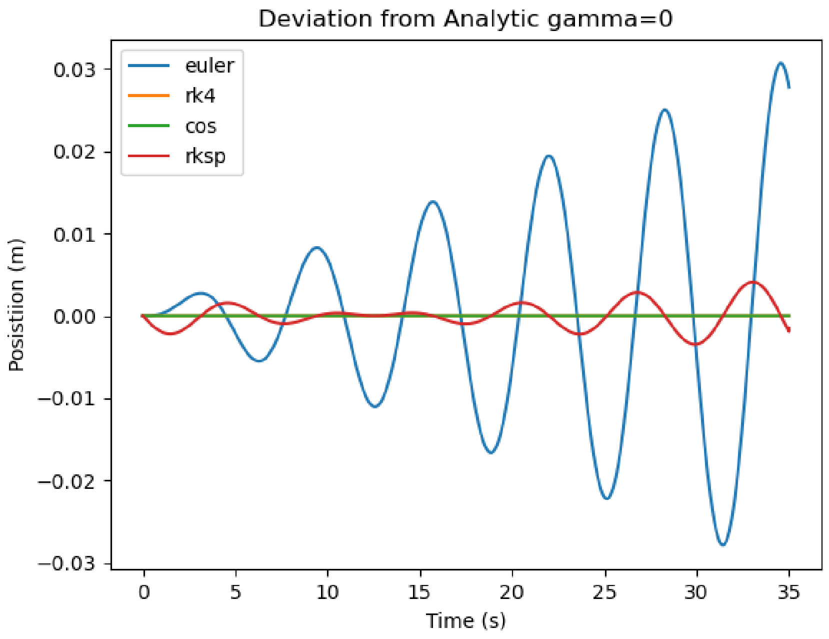


This plot shows, unsurprisingly, that the Euler method of integration had the largest errors. This is to be expected since Euler's method is the most naive integration approach of the three. I was, however, surprised to see that the 4th order Runge-Kutta I implemented (yellow) had smaller errors than the one implemented from scipy (green). I would have expected them to be the same. In my experience with scipy, I have used the optimize.curve_fit function and found it to give exact identical results to when I coded a curve fitting function.

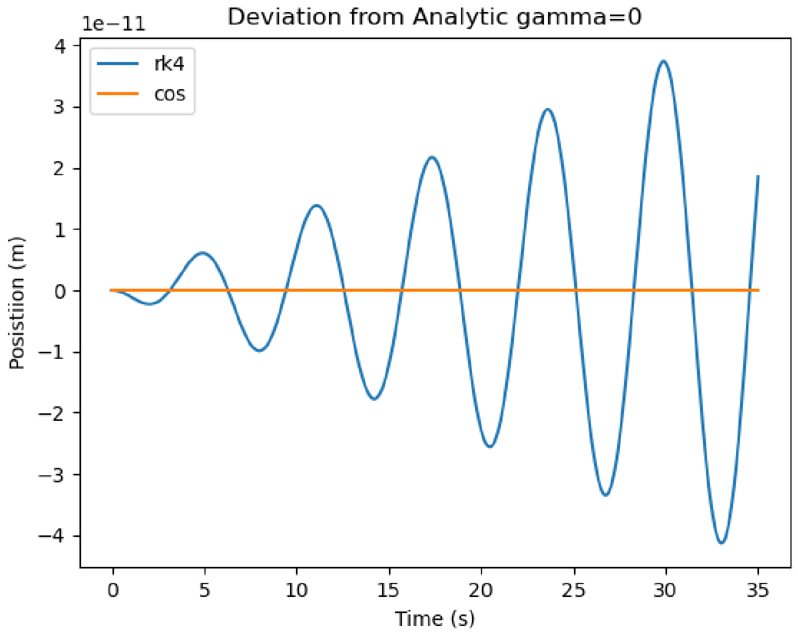
Next, I want to look at a trivial case where $\gamma = 0$. In this case the analytic solution will trivially be $x(t) = -cos(\omega_0 t)$. In this example, I will keep $\omega_0^2 = 1$. What I find is



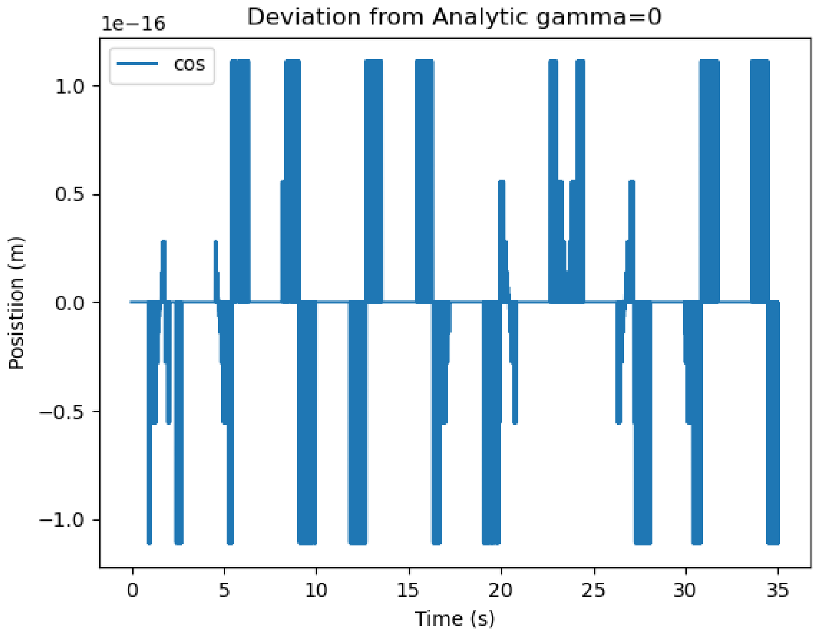
If you look closely, you can see the euler method diverging from the rest of the solutions. I also manually added a cos function to see if there will be a difference between the analytic solution described by eq. (3) since it has to convert from imaginary numbers, I was wondering if that would also cause any noticeable errors. Next, I did the same thing where I took the difference between the analytic solution and the other solutions.



This graph shows that the error in the Euler method integration gets large which we already knew, and it also shows that the scipy 4th order Runge-Kutta method has a similar error even with $\gamma = 0$. It isn't possible to see the manual 4th order Runge-Kutta method on the graph (or at least I could not see it), but plotting the same thing with out the Euler or scipy methods revealed very small error in the manual Runge-Kutta method.

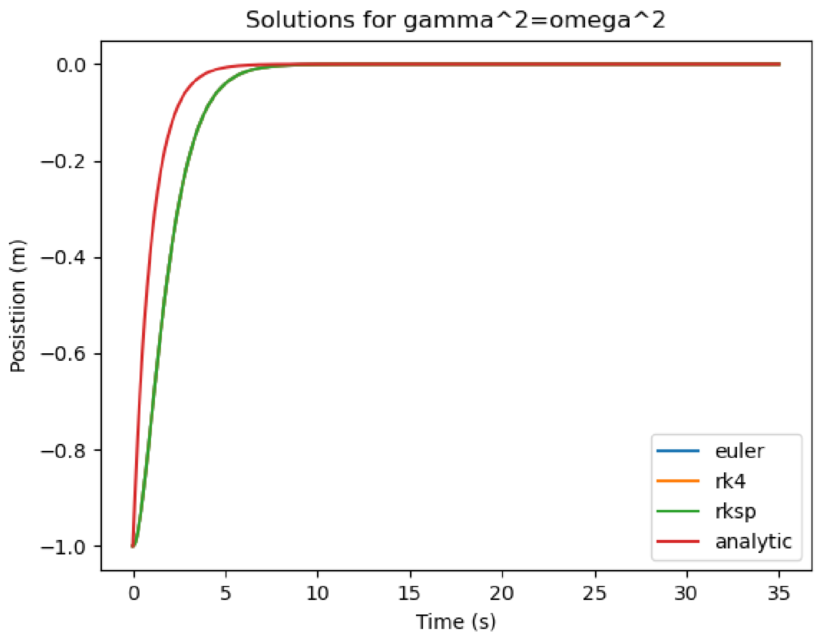


Notice the scale on the y-axis is 10^{-11}. As a sanity check, I also plotted the difference between the cos function and analytic function.

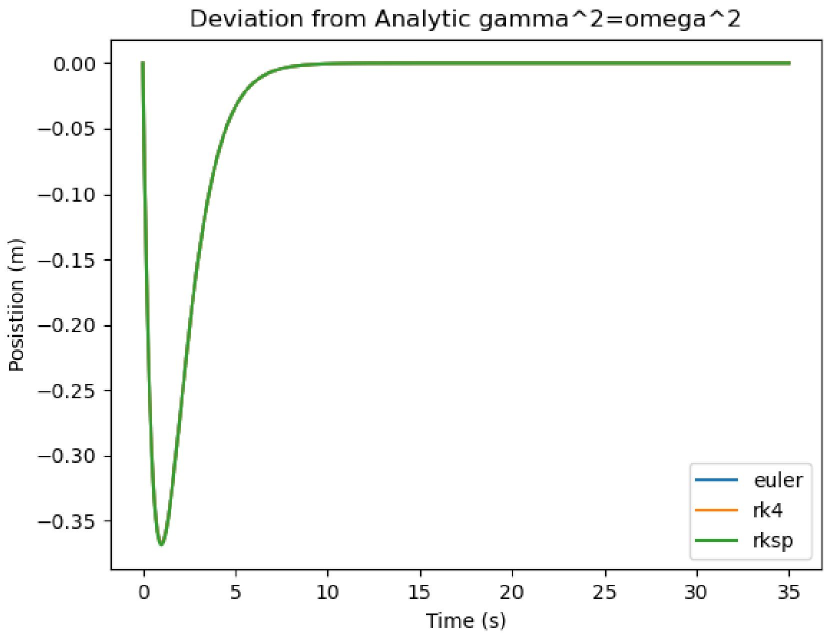


This graph shows that all the errors between the cosine function and analytic function only include rounding errors which implies that python's treatment of imaginary numbers is done well. I can rest easy knowing it doesn't impact any earlier analysis.

Finally, I am going to check when $\gamma^2 = \omega_0^2$. In this case, the solution to the differential equation will look like an exponential decay starting at the initial position. Using the same plotting type as before and creating an oscillator with $\gamma^2 = \omega_0^2 = 1$, I got:



and



These graphs are showing that the all three integration methods are overlapping which means they are all equally poor at approximating the exponential decay function at small time, but get better as time increases. I think, at least part of the problem, is since $\gamma = 1$, the all higher order derivatives are proportional to e^{-t} which is why all of the integration methods look the same because the next higher order derivative that they don't correct are identical. I did check when $\gamma^2 = \omega_0^2 < 1$ and $\gamma^2 = \omega_0^2 > 1$, and the errors are much more similar to the other example. I won't include the other graphs since this report is getting long.

2 Integrals

2.1 The Gamma Function

Prior to taking a statistical physics course, I would have never guessed that the gamma function could be written as a standard integral problem. I learned about the gamma function when discussing an N-dimensional sphere which the professor showed that the integral $\int_0^\infty t^{(z-1)}e^{-t}dt = \Gamma(z)$. My understanding of the gamma function is that it is the explicit version of the factorial function that can be applied to any real number. For any integer $n \in \mathbb{Z}$, $\Gamma(n) = (n-1)!$. An interesting part of this integral is that it goes to infinity, since it is impossible to manually integrate to infinity, there will be some additional error that we will have to integrate. To begin my code, I created an object called "Gamma" that has a single argument z which is any real number.

2.2 Reimann Sum

The simplest way to describe a Reimann sum is a discretized integral where $dt \rightarrow \delta t$ where δt is a small but finite step size such that $t_{n+1} = t_n + \delta t$. Technically, δt does not need to be a constant step size but in all integral methods, I write I take it to be the same at every t. The approximation looks as follows: $\int_a^b f(t)dt \approx \sum_{n=0}^{(b-a)/\delta t} f(a + n * \delta t)\delta t$. For the gamma function,

$$\Gamma(z) = \int_0^{\infty} t^{(z-1)} e^{-t} dt \approx \sum_{n=0}^{\infty} (n\delta t)^{(z-1)} e^{-n\delta t} \delta t \tag{6}$$

Once again, it is impossible to sum manually to infinity, but we can take a large number instead of infinity. I will discuss this more in the analysis section.

2.3 Trapezoidal Sum

If you think about it, the Reimann sum is just summing the area of rectangles with width δt and height $f(t)$. This means that if $f(t)$ is increasing, the Reimann sum will underestimate the integral and if $f(t)$ is decreasing it will overestimate the integral. The trapezoidal sum is really just a Reimann sum where the height of the rectangle is the average of the function at its current and next value (i.e. $\frac{f(t)+f(t+\delta t)}{2}$). This approximation is slightly better than the Reimann sum because it accounts for the area created by the instantaneous slope. The trapezoid sum will, however, overestimate the integral if $f(t)$ is concave up and underestimate it if $f(t)$ is concave down. The explicit form of this approximation for the gamma function is,

$$\Gamma(z) \approx \sum_{n=0}^{\infty} \frac{(n\delta t)^{(z-1)} e^{-n\delta t} + ((n+1)\delta t)^{(z-1)} e^{-(n+1)\delta t}}{2} \delta t \tag{7}$$

2.4 Simpson's Rule

Simpson's rule is prescribed as:

$$\int_a^b f(x) \approx \frac{b-a}{6}(f(a) + 4f(\frac{a+b}{2}) + f(b)) \tag{8}$$

This rule is also sometimes referred to as Simpson's 1/3 rule since it works by breaking up a segment from the curve into two parts using 3 points and weighting the middle point higher than the outside points. If you have been paying attention to the order of sections 2.2, 2.3, and 2.4, you may have noticed a pattern. The Reimann sum is accurate up until the second derivative of f, the trapezoidal sum is accurate up until the third derivative, and Simpson rule is accurate until the fourth derivative. For the gamma function, my version Simpson's rule will look like.

$$\Gamma(z) \approx \sum_{n=0}^{\infty} \frac{\delta t}{6}(f(n\delta t) + 4f(\frac{(2n+1)\delta t}{2}) + f((n+1)\delta t)) \tag{9}$$

2.5 Integration with SciPy

The SciPy library includes a integration method which has several types. I will be comparing my answers for the trapezoidal rule and Simpson's rules with their respective scipy versions. Both the scipy.integrate.trapezoid and scipy.integrate.simpson are both functions that take one mandatory argument, y, which is a list-like element of the function you want integrated. I also passed these functions optional argument, x, which is positional, or step list-like.

2.6 Integration Analysis

$\Gamma(z)$	z=3	z=8	z= 6.07
Analytic	2	5040	135.2821176204179915
Reimann	1.9999991673276956	5040.000000000035	135.28211761707576
Trapezoidal	1.9999991673276931	5040.000000000033	135.28211761707593
Simpson	2.0000002081267656	5039.999999999979	135.28211762146572
scipy Trapezoidal	1.9999991673276956	5040.000000000041	135.28211761707587
scipy Simpson	2.000074570445865	5040.000000000794	135.28211768125433

For the analytic solutions to $\Gamma(z)$, recall that if $z \in \mathbb{Z}$ then $\Gamma(z) = (z-1)!$. For z=6.07, I used wolfram alpha which gave me hundreds of digits, then truncated it down to 15. I'm not quite sure how they calculate it, but clearly they have a much better approximation function than I do. First I want to compare z=3 and z=8, both these have explicit answers. All the approximations for z=8 are more precise than their corresponding approximation for z=3. If you imagine what the function $f(z,t) = t^{z-1}e^{-t}$ this function looks like a mound centered around z, which terminates at 0 and has a long tail in the $+\infty$ t-direction. Since we are not integrating to infinity, we are losing a large part of the tail, but the tail contributes less and less to $\Gamma(z)$ as z increases. I've been thinking of another correction that I could have made to the time domain that I believe might increase precision of answer. My idea is that we can weight the density of points in the t domain such that when the earlier defined function $f(z,t)$ is larger there are more points per unit time. This would allow for more resolution in the more important parts and also allow the integral to go further out in the time domain with out linearly increasing the number of calculations. Take this with a grain of salt though, because I only just thought about it.