Connor Burns
12/13/20

Ideal Spring Integration Report

Introduction:

This report will demonstrate and compare multiple numerical integration methods by calculating the x position of an object in an ideal spring system. Although there are many integration methods, the three under examination are Euler's Method, Leapfrog, and Runge-Kutta 4(RK4). The methods calculate the position of the object for n steps until the end of the simulation (H). Δt is the time between each step and Δt = H/n.

Euler's method works by calculating the new position (x_i) and velocity (v_i) at each time step using the previous position (x_{i-1}), velocity (v_{i-1}) and acceleration (a_{i-1}). The leapfrog method instead uses velocity at half steps, calculating the position (x_i) and half-step ahead velocity ($v_{i+delta\_t/2}$) at each time step using the previous position (x_{i-1}), half-step behind velocity ($v_{i-delta\_t/2}$) and half-step behind acceleration ($a_{i-delta\_t/2}$). The Runge-Kutta 4 method is similar but combines four different estimation methods to find the new position and velocity each step.

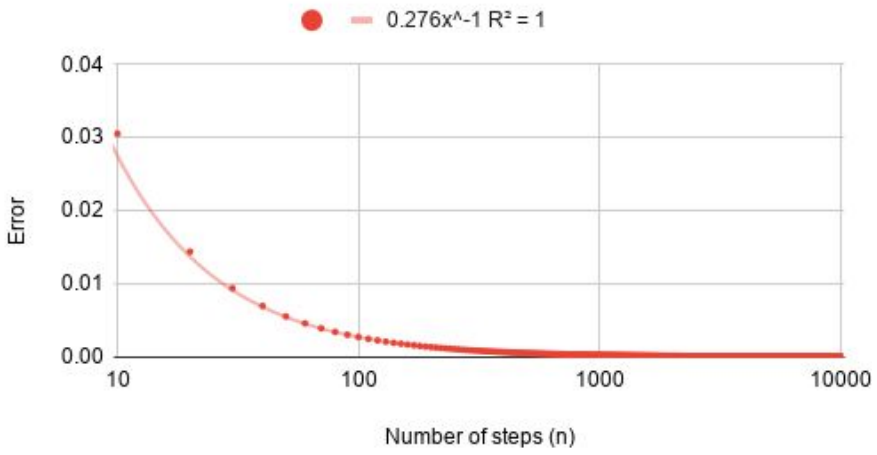
Each of these methods have an important comparative property called "order" which is represented with the variable "k". The error for each is about Δt^k. Euler's is first order (k=1), Leapfrog is second order (k=2), and Runge-Kutta 4 is fourth order (k=4). To demonstrate this we will use a system where an object with mass = 1 is attached to a spring with stiffness $k_s = 1$. The demonstration will start at t=0 where the position is $x_0 = 1$. The initial velocity (v_0) is 0. Each method will estimate the position of the object until the end of the simulation (H=end of simulation time). Meaning they will return the position of the object at x_H.

The force is calculated with Hooke's law where $F = -k_s*x$ and the acceleration is calculated with Newton's second law F = ma. To view the order we must compare the error between the estimation and an exact calculation. The exact calculation is found using the general solution for a simple oscillator: $x(H) = x_0 \cos(\text{sqrt}(k_s/m)H) + (v_0/\text{sqrt}(k_s/m))*\sin(\text{sqrt}(k_s/m)H)$.
Solved for example H=1:
x(1) = cos(1) = .5403...

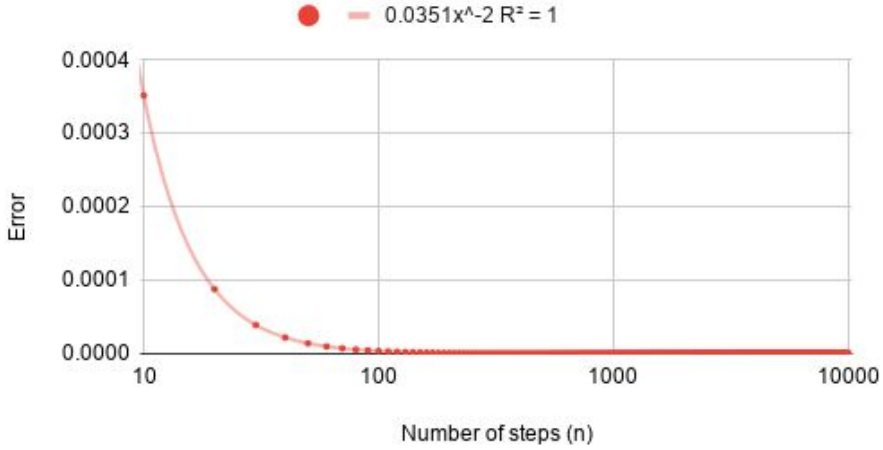
Part 1 Method Order Demonstration:

To demonstrate the order I ran 1,000 estimations of Euler's method and Leapfrog starting with 10 steps (n=10) increasing by 10 ten each time until n=10,000 (n = 10, 20, 30… 10,000) for total time H=1. To be able to visualize the small error for RK4 I ran by increments of 5s starting at n=5 going to n=1,000 (n = 5, 10, 15… 1,000) for total time H=1. The error of each simulation is graphed in red.

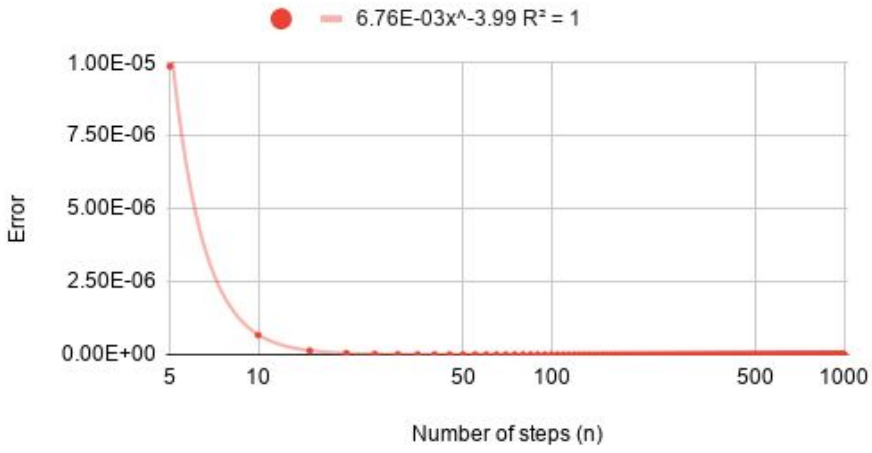
Euler's Method (k=1)



Leapfrog (k=2)



RK4 (k=4)



In the following graphs the line of the generic equation x^{-k} is in blue. This line demonstrates the order property (error = Δt^k) as follows:

$x = n$

$\Delta t = 1/n$

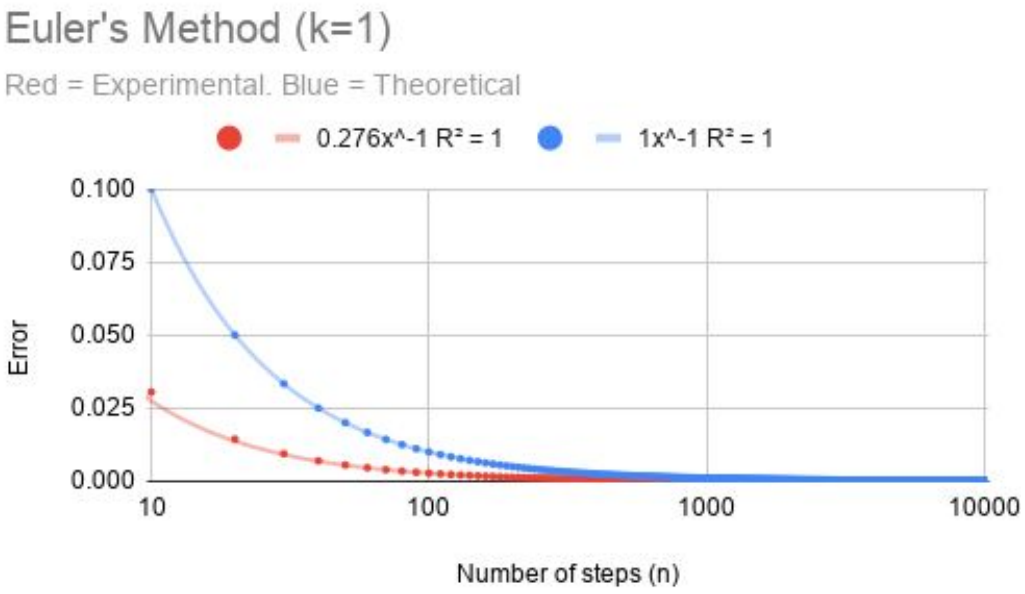
$f(x) = \Delta t^k$

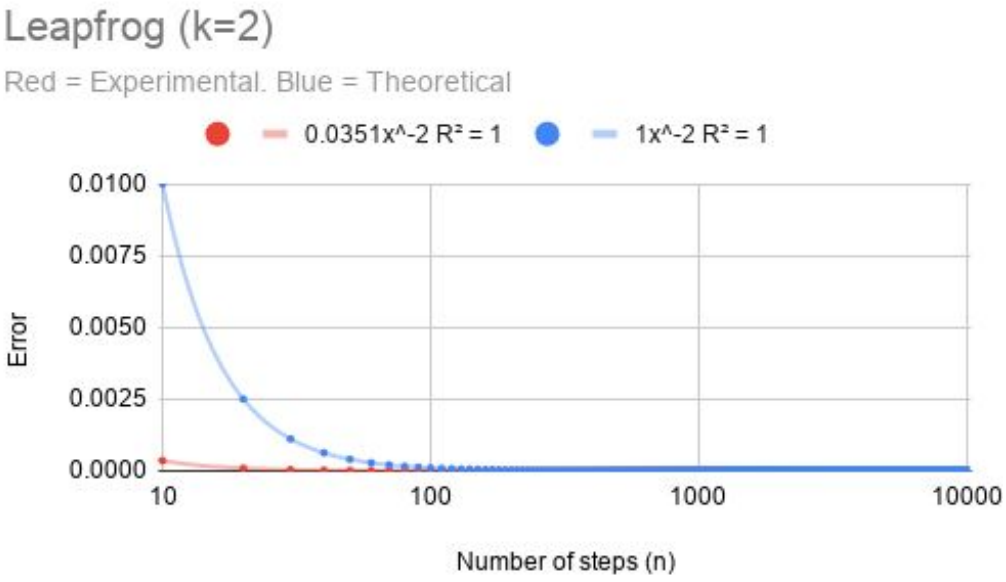
$f(x) = (1/n)^k$

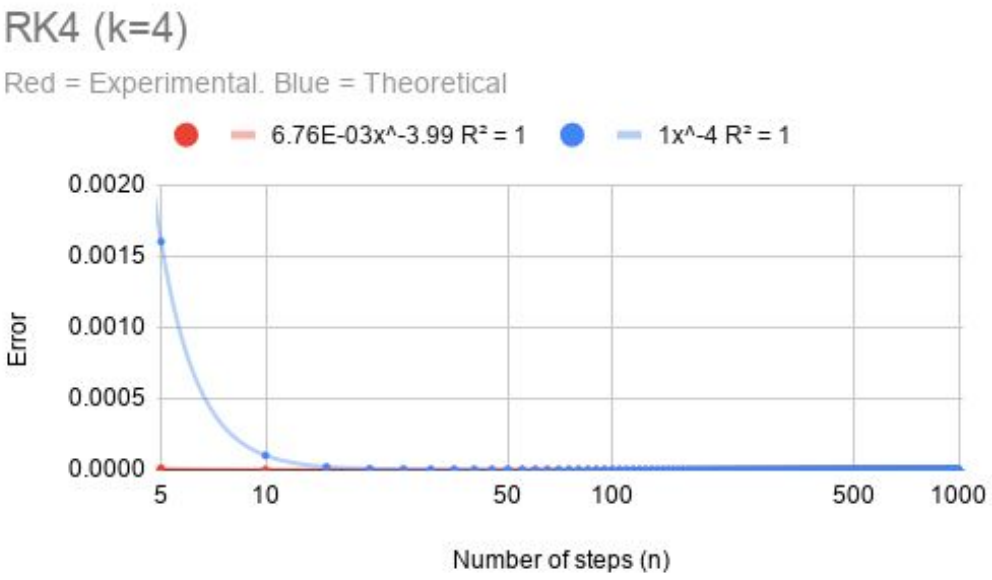
$f(x) = (1/x)^k$

$f(x) = x^{-k}$

If the order property is to be satisfied then the two lines should grow at a similar rate, which can be demonstrated by their equations being equal to each other using Big-O notation.







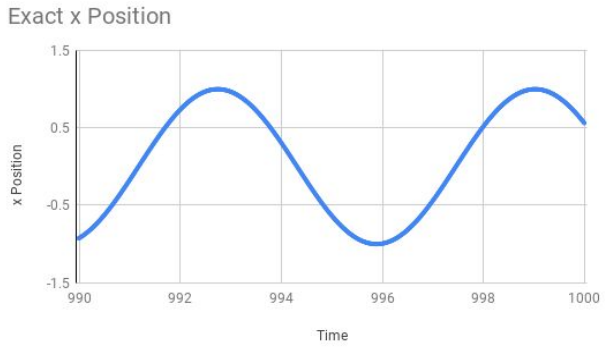
Clearly all three methods grow at rates very similar to x^{-k}. The experimental equation is the power series with the best fit. In each of the graphs the equations fits the data very well evidenced by the R^2 value being very close to 1.

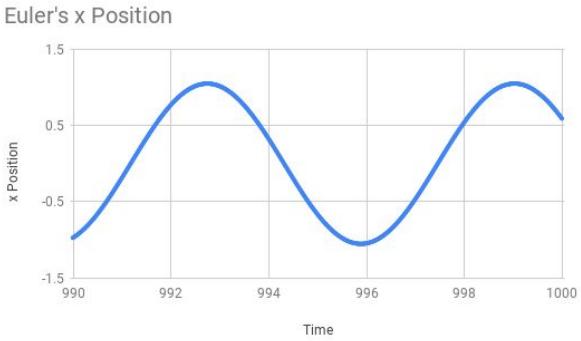
For Euler's method $O(0.276x^{-1}) = O(x^{-1})$ which is $O(x^{-k})$ for k=1.
For Leapfrog $O(0.0351x^{-2}) = O(x^{-2})$ which is $O(x^{-k})$ for k=2.
For Runge-Kutta 4 $O(6.76E\text{-}3x^{-3.99}) = O(x^{-4})$ which is $O(x^{-k})$ for k=4.

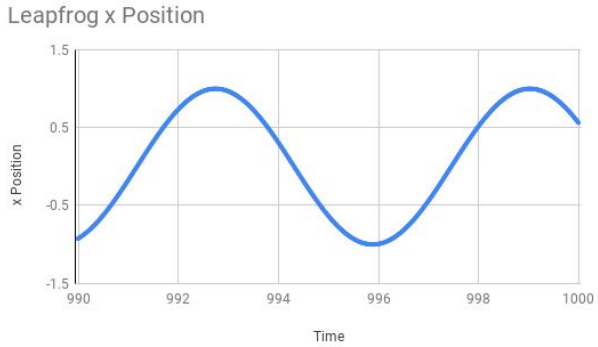
O(equation of best fit) is equal to $O(x^{-k})$ for the data sets of each of the methods. Because as shown $f(x) = x^{-k}$ is equal to $f(x) = \Delta t^k$ the order property holds for each method

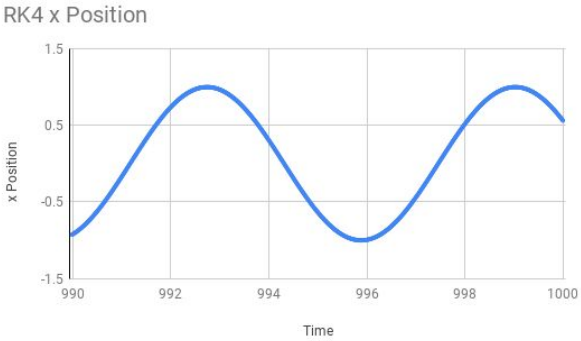
Part 2 Longer Duration Integration :

Below are the graphs of an integration of the same system but for time 1,000 (H=1,000). The timestep used is $\Delta t = 0.0001$ (1e-4), meaning the number of steps was n = 10,000,000 (1e7) over a time of 1,000 (1e3). The exact solution is about x = 0.562379. The following are graphs of the last 10 units of time (t=990 -> 1,000) where the total error is the greatest. First I included a graph of the exact x position over that time, to compare with the integration methods.

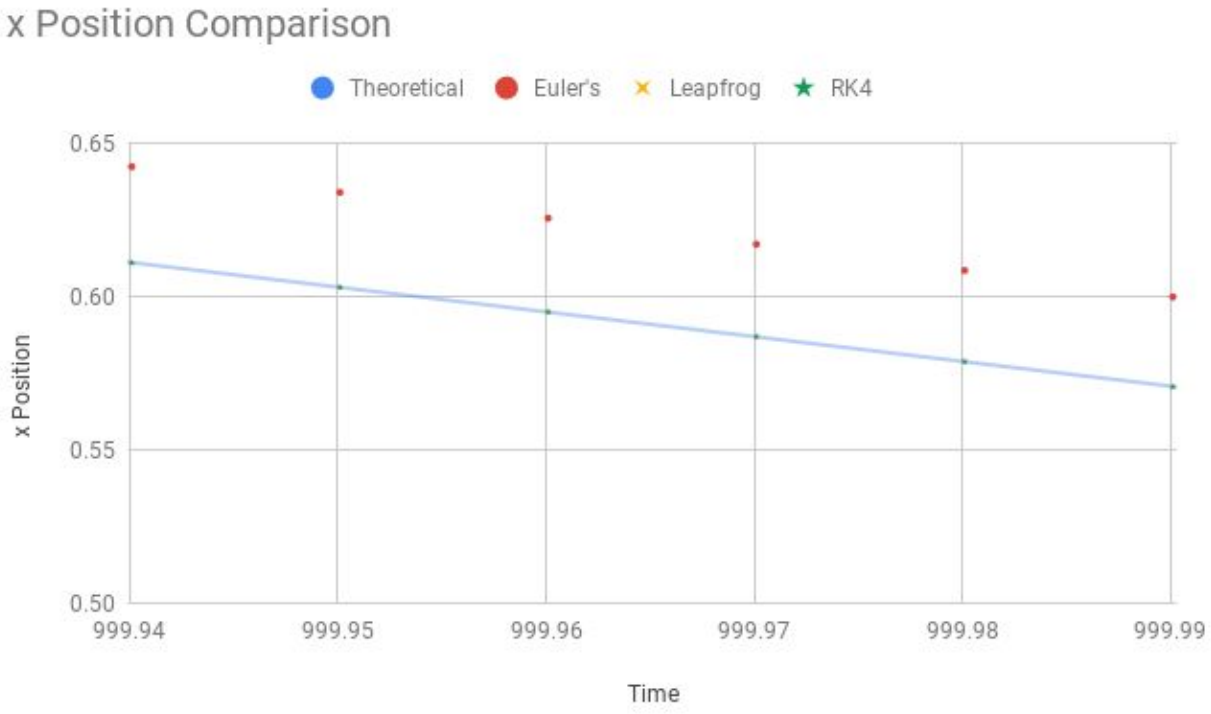






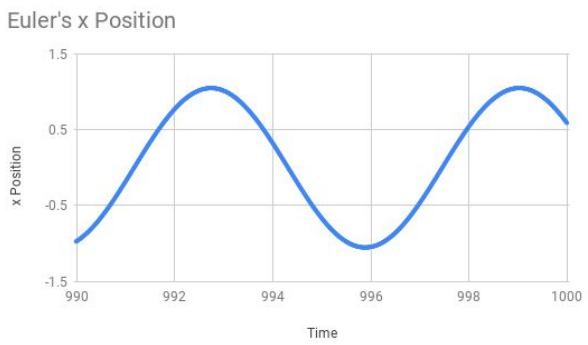


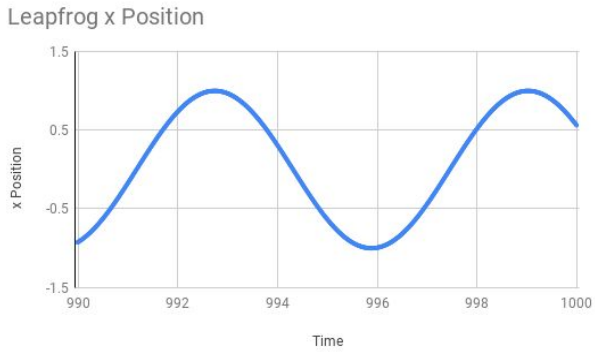
To really see the difference between the methods, the following graph shows the last 6 steps (t=999.94 -> 1,000) of each one. The Leapfrog and RK4 points are nearly on top of each other.

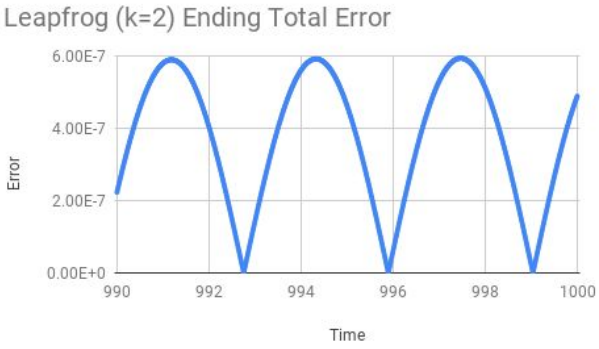


To view error over time, below are graphs of the total error for each step of the three integration methods. Clearly over time the max error is increasing. For each except the start of RK4, the error seems to linearly increase for each method.

However in actuality none of the methods errors increase linearly. The points are actually rising and dropping in a cycle. Comparing the following graphs of the last 10 units of time (t=990 -> 1,000) with their respective position graphs show the relation of the x position to the error for each method. For Euler's method in the small scale as the x position is closer to inflection points (x=0) the error decreases, and as the x position is closer to local minimums and maximums the error increases. Leapfrog and RK4 both have the opposite, greater error at inflection points, and lower at mins and maxs.This error property of each method works in conjunction with the general increase shown above. So when the exact x position is the same at two points in time, the error will be greater for the point later in time.







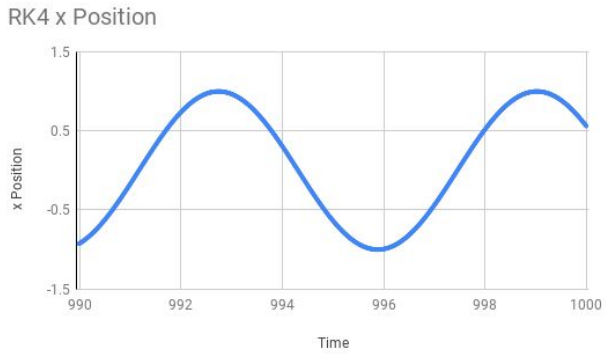


Chart of the runtime and error of each method:

Method (n = 1e7)	CPU Runtime (seconds)	Error
Euler	62.87	2.88367e-2
Leapfrog	68.52	3.44533e-7
RK4	77.04	2.39808e-14

To get the most accurate solution in the least amount of CPU time I would recommend RK4. For the same sample size RK4 is slower than both Euler and leapfrog, but because RK4 is significantly more accurate than the other two methods, one needs a much lower sample size to achieve the same level of accuracy.

Chart of RK4 and leapfrog runtime to get error similar to Euler's error:

Method	Samples (n)	Error	CPU Runtime (s)	Time Saved (compared to Euler)	% of Runtime (compared to Euler)
Euler	1e7	2.88e-2	6.76344e1	0	100.%
Leapfrog	3.5e4	2.84e-2	2.279e-1	6.74065e1	0.34%
RK4	3.7e3	2.94e-2	4.30e-2	6.75914e1	0.06%

The percentage of runtime above is the easiest metric to view the proportion to which RK4 beats its competition.

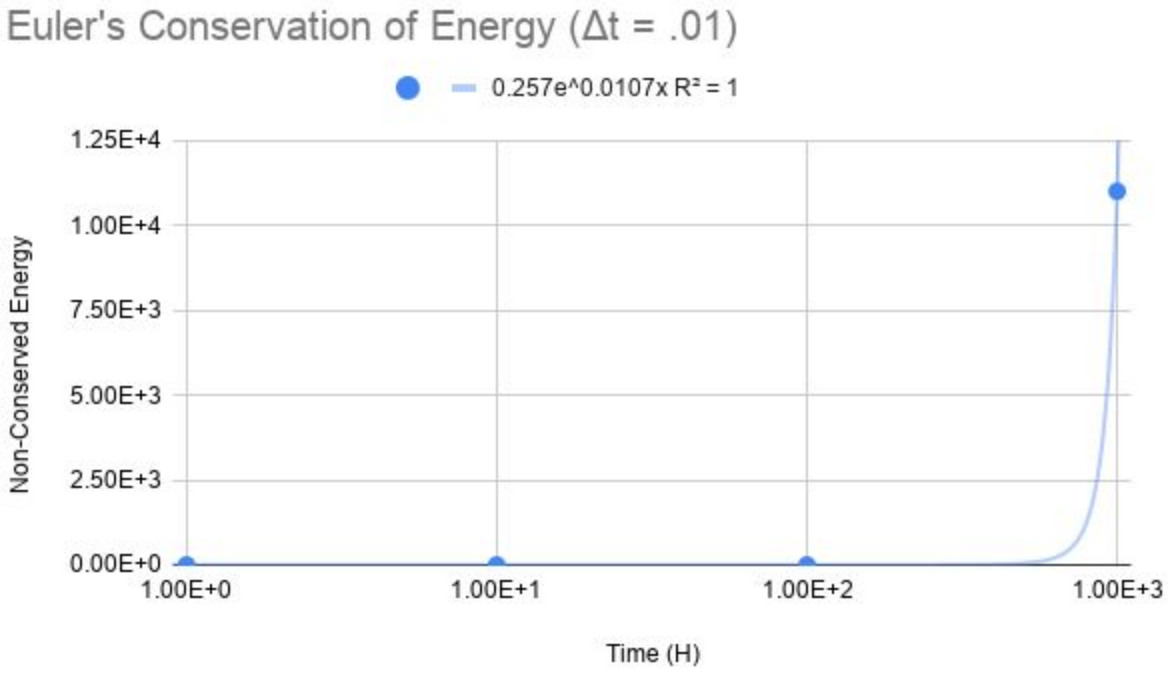
Part 3 Conservation of Energy and Integration Error:

Because I am simulating real world physics, the simulated mass-spring system should follow the law of conservation of energy. However because Euler's method, leapfrog, and Runge-Kutta 4 all do stepwise approximation of position and velocity, each will violate the law of conservation of energy. Even in the absence of knowing the exact solution, viewing the non-conserved energy is a measurement of error for each of the methods. The Net Energy of

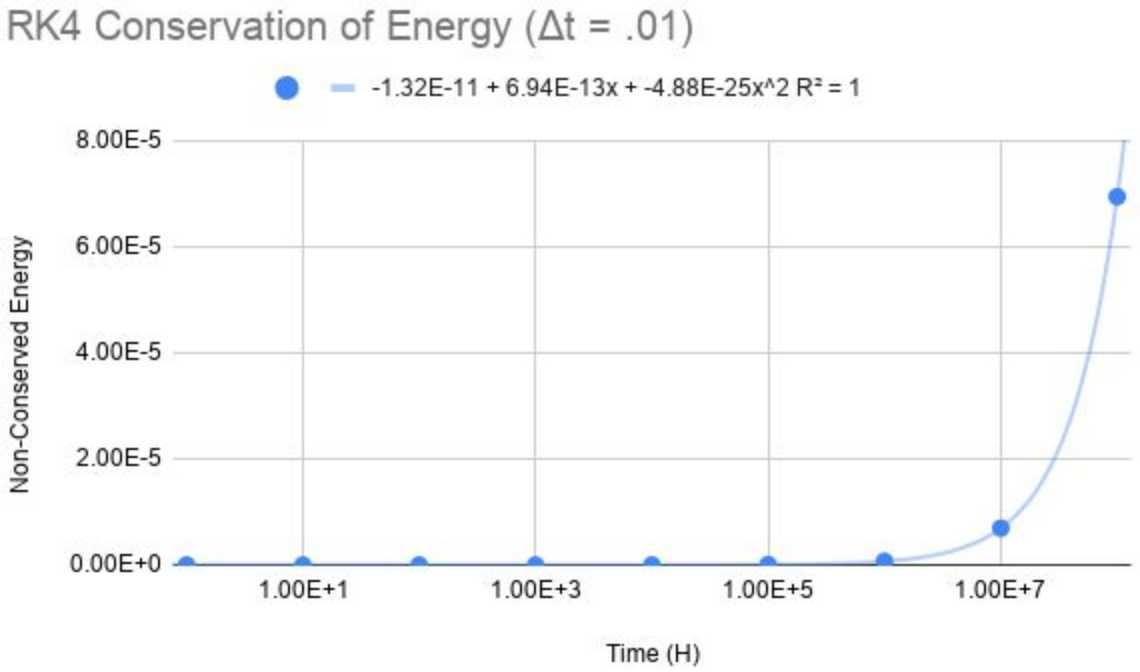
the system can be calculated at any point in time by summing Kinetic Energy($\frac{1}{2}mv^2$) and Potential Energy ($\frac{1}{2}k_s x^2$). Because the initial velocity is 0, we can find the theoretical energy of the system by calculating the potential energy of the system.

$\frac{1}{2}k_s x_0^2 = \frac{1}{2}*1*1 = .5$

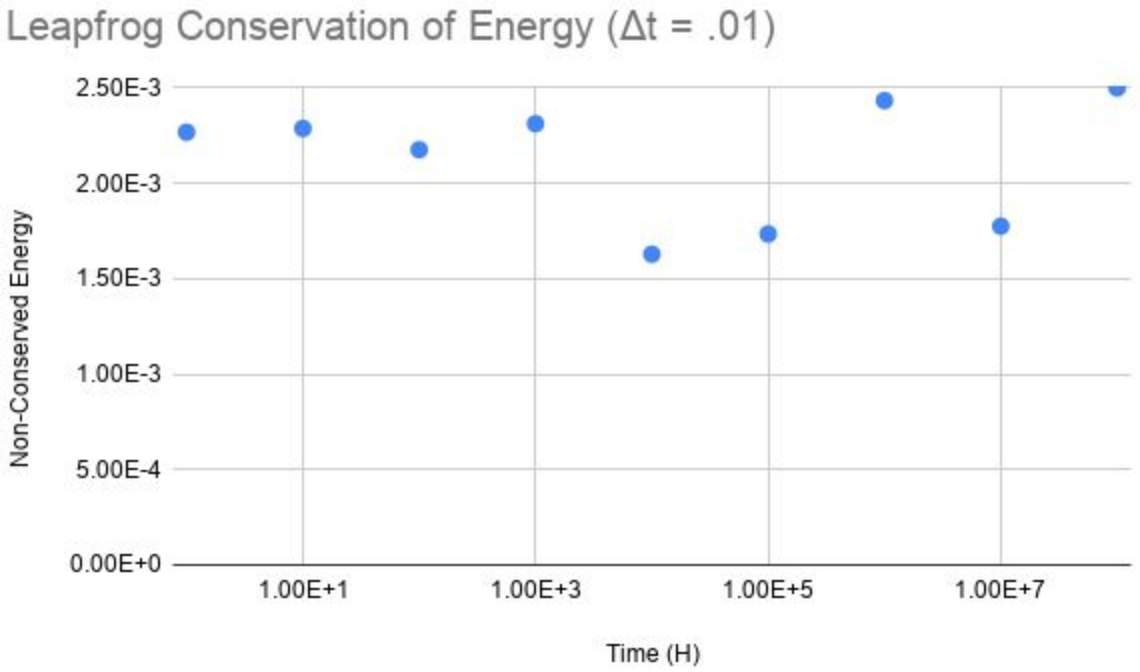
I ran simulations of increasing duration (H = 1 -> 1,00,000,000) with timesteps of $\Delta t = .01$ for each of the methods and have graphed their conservation of energy violation.



Euler's non-conservation of energy is exponential. In fact the growth is so quick that by $H=10^4$ the conservation of energy error was 1.33736E+43, and after that it exceeded the max float value. This is in line with what we expect for Euler's method, the error is relatively low for small intervals, but as they get larger you need a huge factor more steps to keep the error small. If you don't then Euler's solution will be so inaccurate that it has no reflection of the true position value.



RK4 conservation of energy error is polynomial, but nearly linear. The error start very small, and increases by about a factor of 10, for every time increment where $H_{i+1} = H_i*10$. There is almost a 1 to one correlation between non-conserved energy increase and time increase. The slow increase is in line with the order of RK4 being k=4 where the actual error grows very slowly.



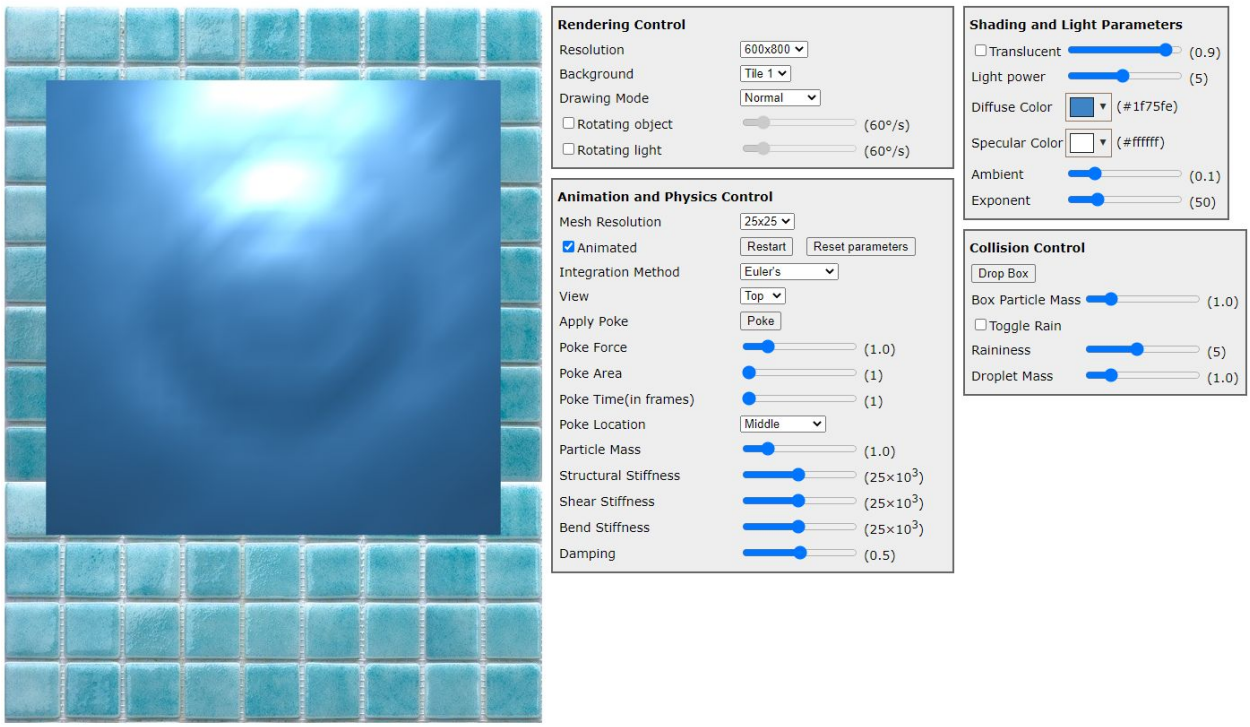
Leapfrog's conservation of energy graph is neither exponential nor linear. The energy error generally increases, but goes up and down by large proportions during integrations of increased durations. This anomaly can be explained. At times when the acceleration is high, the velocity at time t+(Δt/2) is much greater than the velocity at time t. This leapfrogging makes the extra kinetic energy in the future counteract the non-conserved potential energy, a function of x position. When acceleration is low, the opposite is the case. So the non-conserved energy is greatly influenced by the true x position at the end (t=H).

The main factor influencing leapfrogs conservation of energy is the velocity at H+Δt/2. This is what makes the leapfrog's energy error correlate so poorly with its position error. Whereas for both RK4 and Euler's it is the numerical error in calculating the position and velocity at time H and their energy errors correlate with their position errors. Leapfrog's velocity leapfrogging property results in a better long-term conservation of energy compared to both RK4 and Euler's method.

Part 4 Relevance in Computer Graphics:

Stepwise integration is particularly applicable to physics based animation. In physics based animation graphical objects have positions, velocities, and accelerations. A new timstep is done every time a new frame is shown to the viewer, and once can actually see the results of each step. Small errors in physics based animation can be imperceivable, especially if there are a lot of objects moving at a large scale. However, if there is a large amount of error it is extremely visible, and often can be viewed as an explosion of particles by the user, ruining the animation.

Water Physics Simulation - Connor Burns



I have animated a real-time physics based water simulation using WebGL and JavaScript, in which one can change between the multiple integration methods to view them for themselves. You can see that the error here is not really visible, so Euler's method is a fine choice as it is much easier to implement and requires less iterations through all the objects. RK4 requires every point to be visited 4 times as often, and requires 4 times the space as in Euler's method. This is because it gets and stores the acceleration at multiple x positions for every iteration. The link is in the "Code Repository Links Section".

Part 5 Conclusion:

The most important factor in choosing which integration method is best for a system is based on what the purpose of the system is. If one wants little error in their data as quickly as possible, then Runge-Kutta 4 is the best choice. It's order results in errors at such a massively smaller scale than the other methods that the overhead of a more complicated algorithm is worth it. If one wants the simplest integration method then Euler's method is a great choice. Euler's method should be the go-to for starting to learn about stepwise integration estimation, or in a system where speed of implementation is a main factor. When one wants both somewhat easy implementation and reducing error is a factor, then leapfrog is the best choice of integration method. At order k=2, its error is a large reduction from Euler's method, and the implementation only has 1 line different, compared to combining four estimations with RK4. To pick an integration method, decide what you care about as a mathematician, computer graphics engineer, physicist, teacher, or other profession and use whichever is inline with those goals.

Part 6 Code and Repository Links:

Ideal Spring Simulation:
https://github.com/ConnorBurnsCoder/IntegrationMethodSim

Water Physics Simulation:
https://github.com/ConnorBurnsCoder/WaterPhysicsSimulation