CSE 380: Final Project

Canned Project

Evan Ott

December 9, 2016

Overview

Third-Party Libraries

These are the TPLs I relied on for this project:

Catch	unit testing, header-defined
Eigen	working with vectors and matrices easily, header-defined
HDF5	output format to store data
GSL	scientific library
INIReader	way to define configuration options easily, header-defined

Problem 1

For this simple problem, I solved the simple ODE

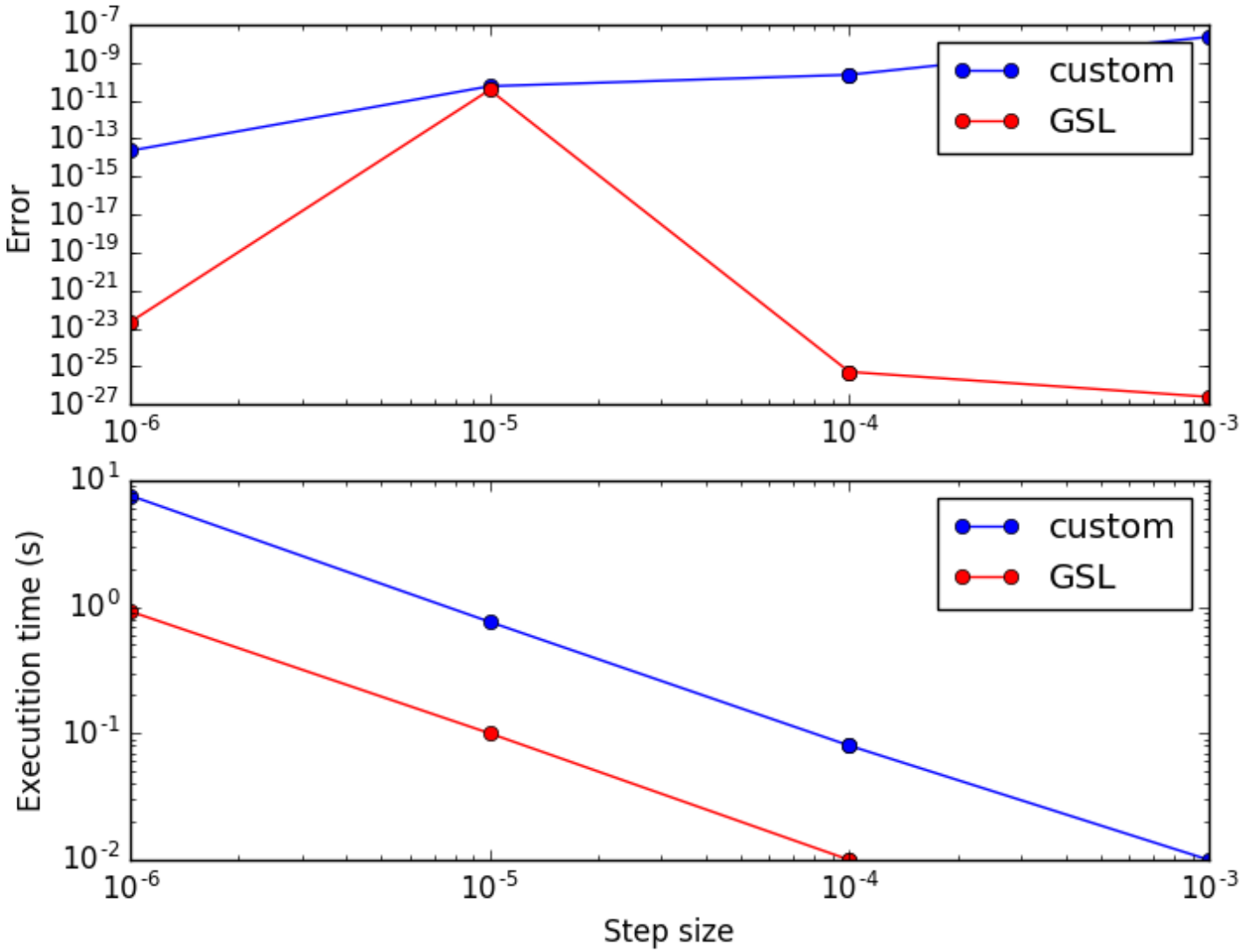
$$\dot{y}(t) = -0.1 \cdot y(t)$$

which of course has the solution

$$y(t) = y(t_0)e^{-0.1(t-t_0)}$$

for an initial condition specified at t_0.

Results


Error
10^{-7}
10^{-9}
10^{-11}
10^{-13}
10^{-15}
10^{-17}
10^{-19}
10^{-21}
10^{-23}
10^{-25}
10^{-27}
10^{-6}
10^{-5}
10^{-4}
10^{-3}
custom
GSL
Executition time (s)
10^{1}
10^{0}
10^{-1}
10^{-2}
10^{-6}
10^{-5}
10^{-4}
10^{-3}
custom
GSL
Step size

Results

- My implementation of forward Euler is slower than GSL's RK4 method on the same step size
- My implementation is also less accurate than GSL's RK4 on the same step size

Results

- My implementation of forward Euler is slower than GSL's RK4 method on the same step size
- My implementation is also less accurate than GSL's RK4 on the same step size
- This shouldn't be surprising, as forward Euler is in general less accurate than more general Runge-Kutta methods. Furthermore, I decided to use `Eigen` for handling vectors for problem 2 and the same overhead is present here.
- Not entirely sure why GSL does relatively poorly for $h = 10^{-5}$, but perhaps the larger step sizes were spuriously accurate

Problem 2

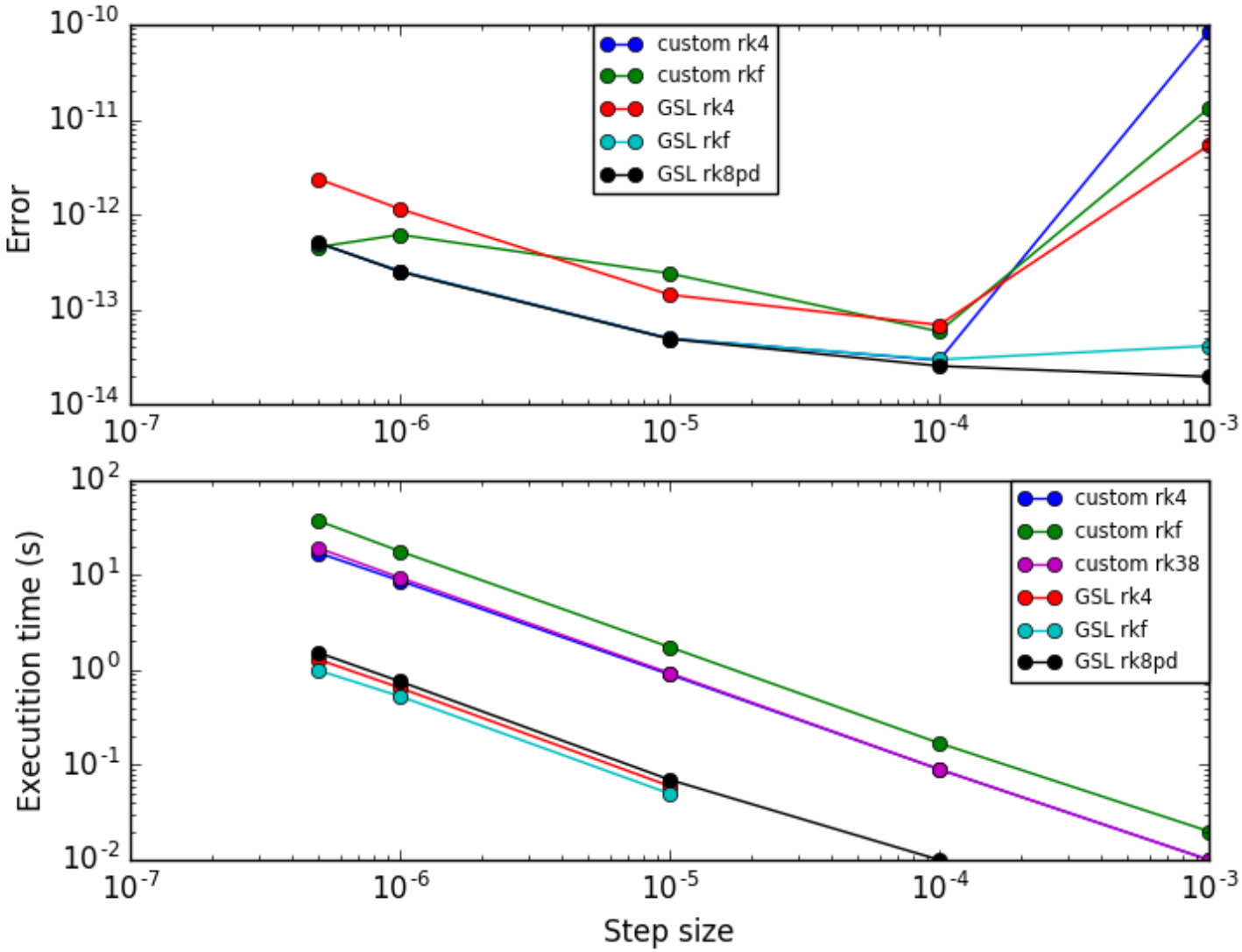
I used *Mathematica* to solve the system of ODEs for the following analytical solutions

$$x(t) = \frac{20\tau e^{-\frac{t}{\tau}}\left(\tau\omega\sin(t\omega) + e^{t/\tau} - \cos(t\omega)\right)}{\tau^2\omega^2 + 1}$$

$$y(t) = -\frac{20\tau e^{-\frac{t}{\tau}}\left(\tau\omega e^{t/\tau} - \tau\omega\cos(t\omega) - \sin(t\omega)\right)}{\tau^2\omega^2 + 1}$$

$$z(t) = 2\tau e^{-\frac{t}{\tau}}\left(e^{t/\tau} - 1\right)$$

Results


Error
custom rk4
custom rkf
GSL rk4
GSL rkf
GSL rk8pd
Executition time (s)
custom rk4
custom rkf
custom rk38
GSL rk4
GSL rkf
GSL rk8pd
Step size

Results I

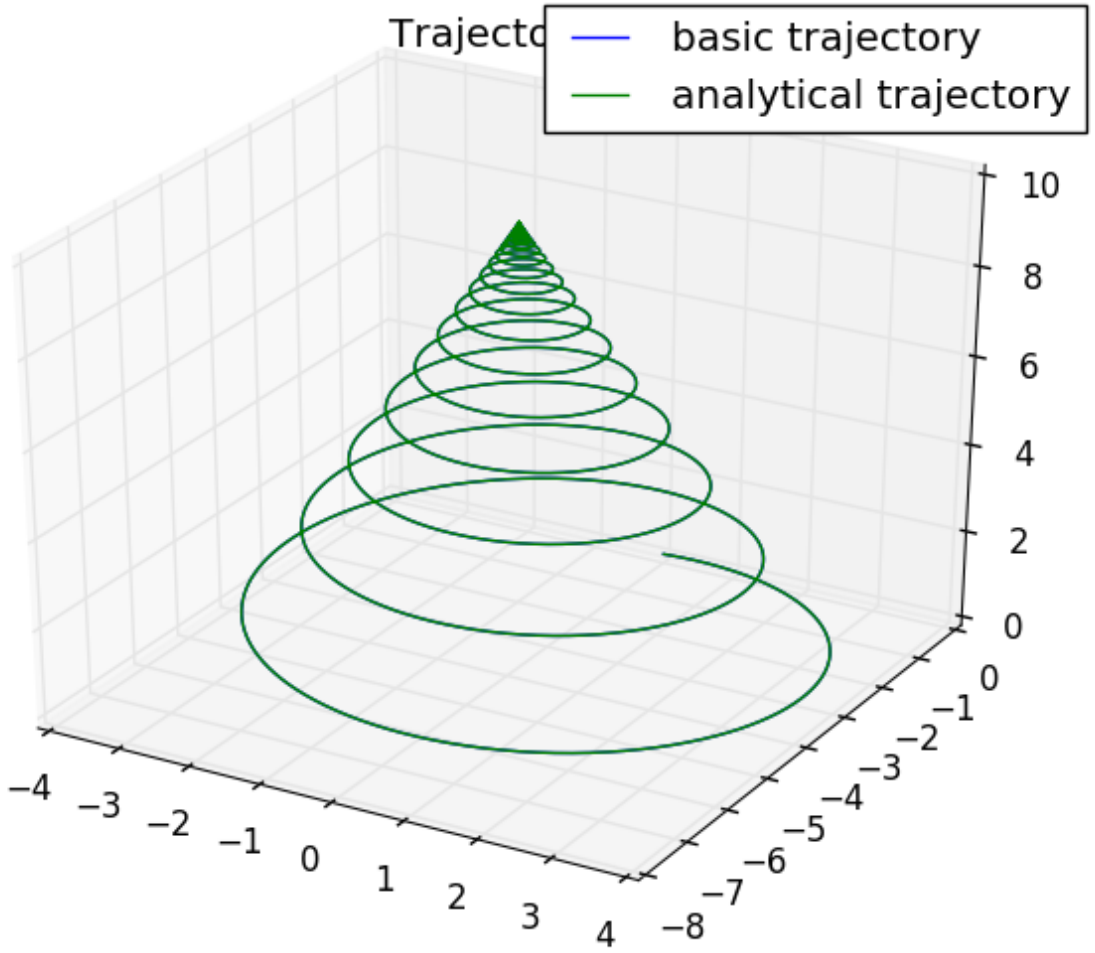
- Due to initially misreading the project assignment, I implemented my own RK solvers ("custom" in previous slide).
- Similar to problem 1, my solutions are slower in execution time (again, likely due to Eigen).
- Accuracy above $h = 10^{-4}$ behaves as expected, with my solutions being less accurate, and all solutions worsening.
- I can't explain why the accuracy gets worse (simulation was fixed total time), but maybe the numerical error levels out at some point?
- Interestingly, the GSL implementation of Runge-Kutta-Fehlberg was the fastest GSL, followed by RK4 then Runge-Kutta Prince-Dormand (a higher order solver)
- For my solutions, RK4 and the "3/8" rule RK method showed similar timing, while Runge-Kutta-Fehlberg was slowest.

Results II

- In terms of accuracy, RK4 as implemented by GSL was the worst overall at small step size, and the worst out of the GSL options at all step sizes.
- The RK4 and "3/8" RK methods I implemented performed nearly identically in time and in accuracy.

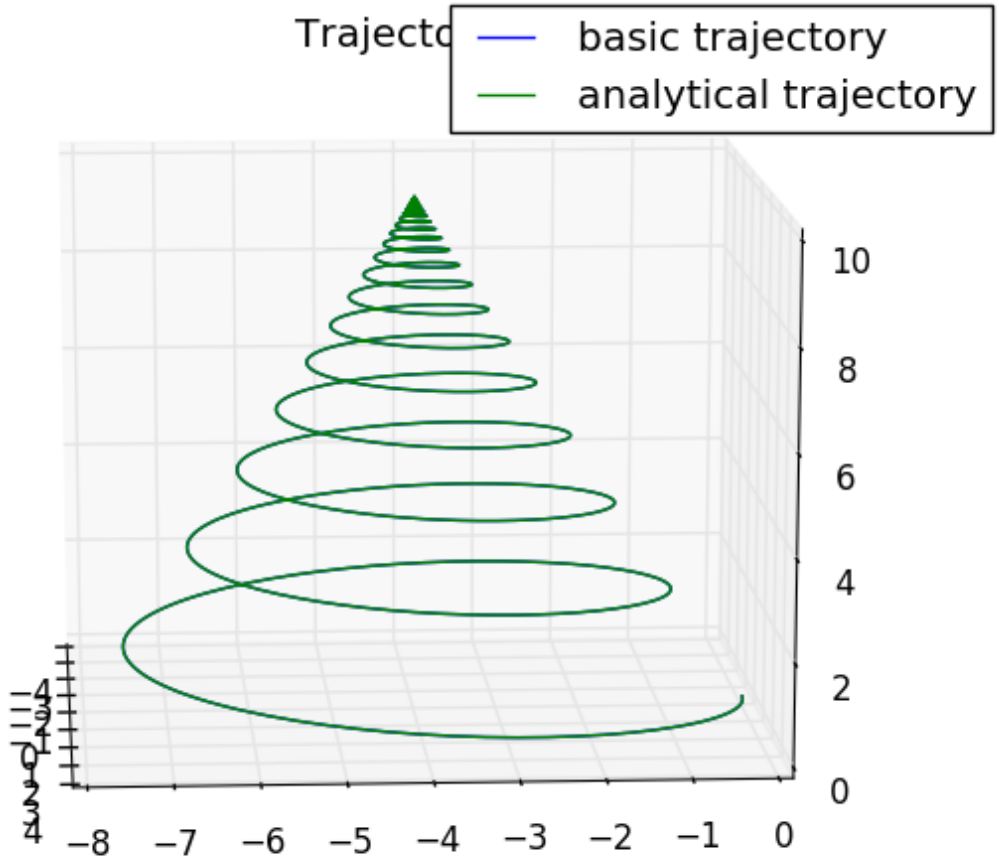
Simulation Results

This is a high-resolution result computed on Stampede for 1M iterations with a step size of 0.0001 (RK4).



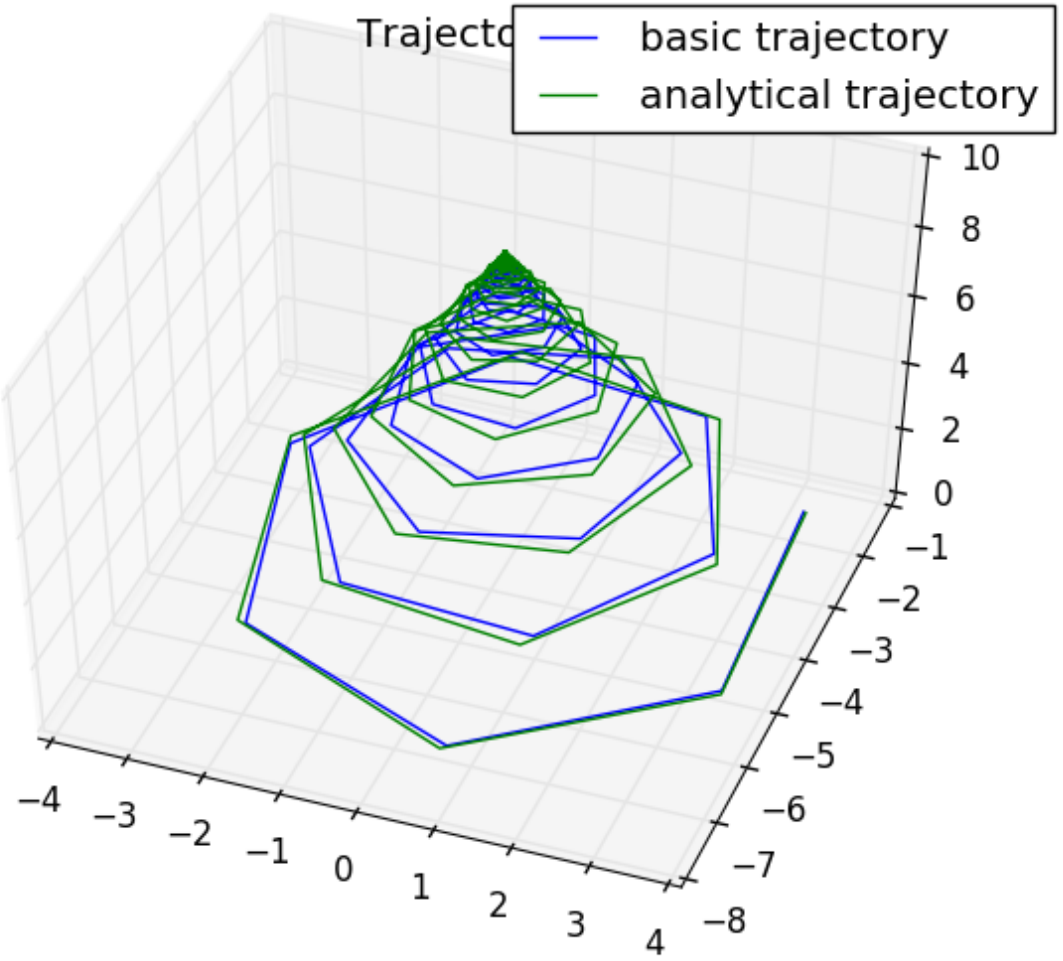
Simulation Results

This is a lower-resolution result computed on Stampede for 10K iterations with a step size of 0.01 (RK4).



Simulation Results

This is a minimal-resolution result computed on Stampede for 500 iterations with a step size of 0.2 (RK4).



Simulation Results

This is a ludicrous-resolution result computed on Stampede with a step size of 0.5, and is here mainly to celebrate that the semester is now over and Christmas is upon us.