Systems Biology Project 3

Mathias D. Kallick

November 2, 2017

1 Forward Euler

1.1 Method

Forward Euler is the simplest and most intuitive of the three methods that we're considering for this project. The basic concept is the same as the principle behind Riemann sums - we evaluate the ordinary differential system at the initial condition, then use those slopes to "guess" the position of the system at time $t + \Delta t$. The equation for this is as follows:

$$\hat{y}(t_{i+1}) = \hat{y}(t_i) + \Delta t(f(t_i, \hat{y}_i))$$

1.2 Accuracy

This method is actually surprisingly accurate for low values of dt. At higher values, though, the curves become much more obviously combinations of straight lines, and for something slightly more complex, like the Lotka-Volterra ODE, we get problems - notably, it diverges massively. Figures 1 and 2 demonstrate this.

1.3 Performance

Using this method to evaluate one thousand seconds of two proteins degrading with time steps of .001, I found that the runtime was 11.4 seconds.

2 Explicit Trapezoid

2.1 Method

The Explicit Trapezoid method is a second-order ODE solving method. It relies on the assumption that the state of the system between the current time step and the next time step is closer to the average of those two states than either one of those two states. As long as our time steps are small enough, that is a very reasonable assumption. What this method does first is to find an approximation of the solution at the next step (my code uses the Forward Euler method to do

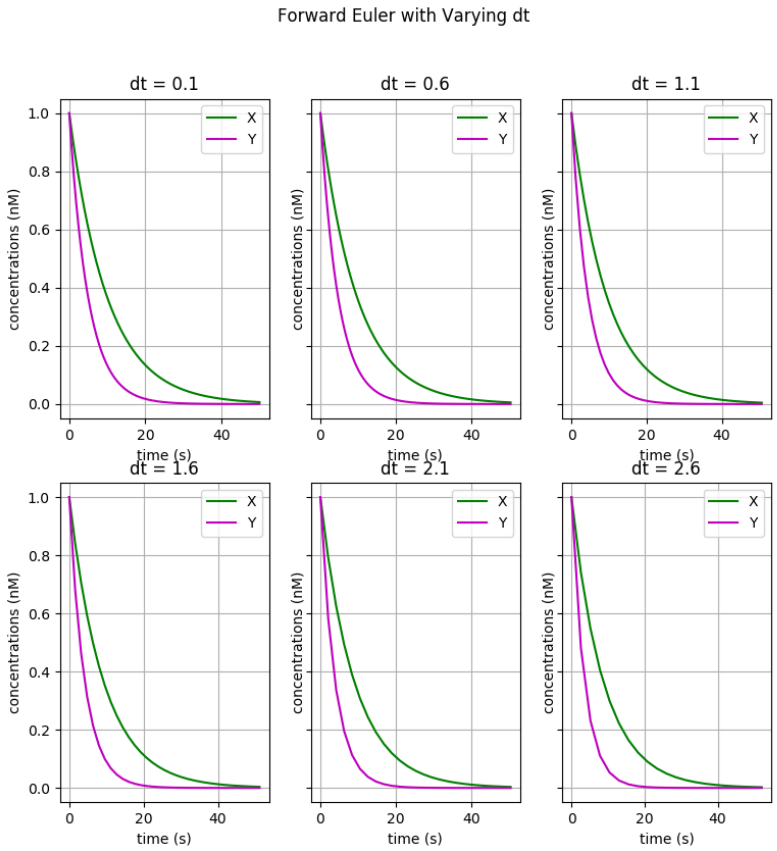

Figure 1: Degradation of two variables at rates $\alpha = .1$, $\beta = .2$. This is simulated with the Forward Euler method at various sizes of time steps.

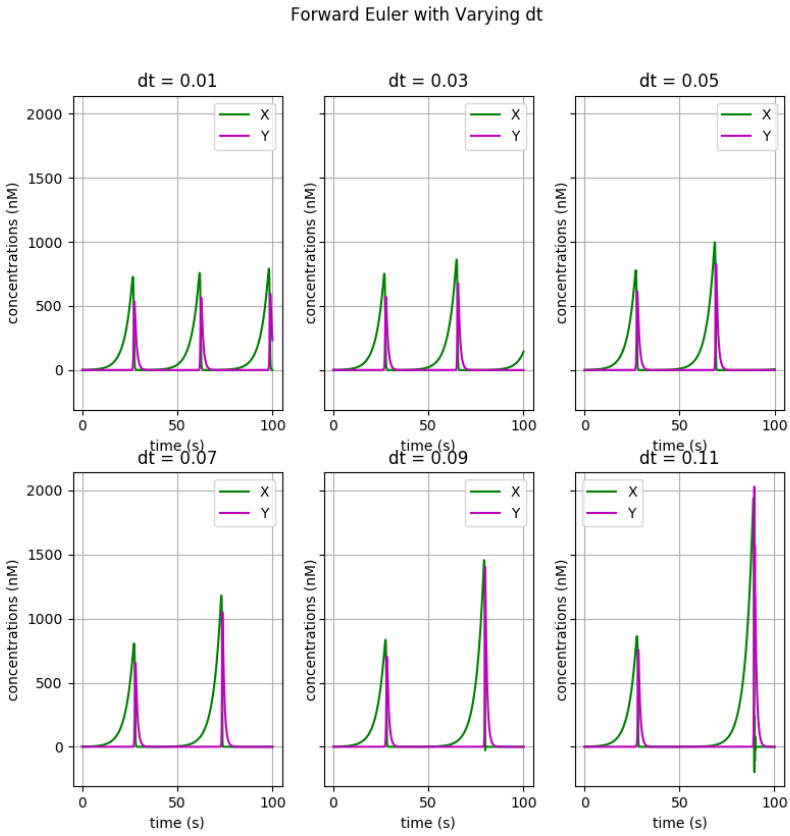

Figure 2: A simulation of the Lotka-Volterra ODE with parameters $\alpha = .25$, $\beta = .01$, $\gamma = 1$, $\delta = .01$. This is simulated with the Forward Euler method at various sizes of time steps.

this). It then takes an average of that approximate solution and the solution at the current time step, and uses that as the slope for the true "guess" at the next time step. The equation for this is as follows:

$$\hat{y}(t_{i+1}) = \hat{y}(t_i) + \frac{\Delta t}{2}(f(t_i, \hat{y}_i) + f(t_{i+1}, \hat{y}_{i+1}))$$

2.2 Accuracy

This method is generally more accurate than the Forward Euler method. It still falls apart at higher values of dt, though - the curves again become combinations of straight lines. For the Lotka-Volterra ODE, though, it doesn't seem to end up diverging anywhere as easily. Figures 3 and 4 demonstrate this.

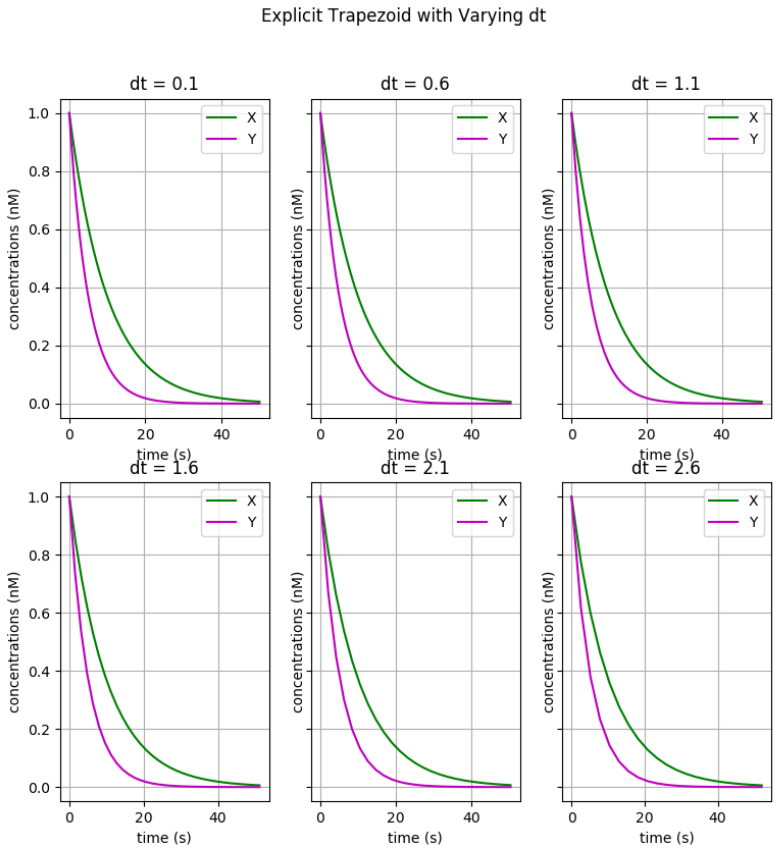

Figure 3: Degradation of two variables at rates $\alpha = .1$, $\beta = .2$. This is simulated with the Explicit Trapezoid method at various sizes of time steps.

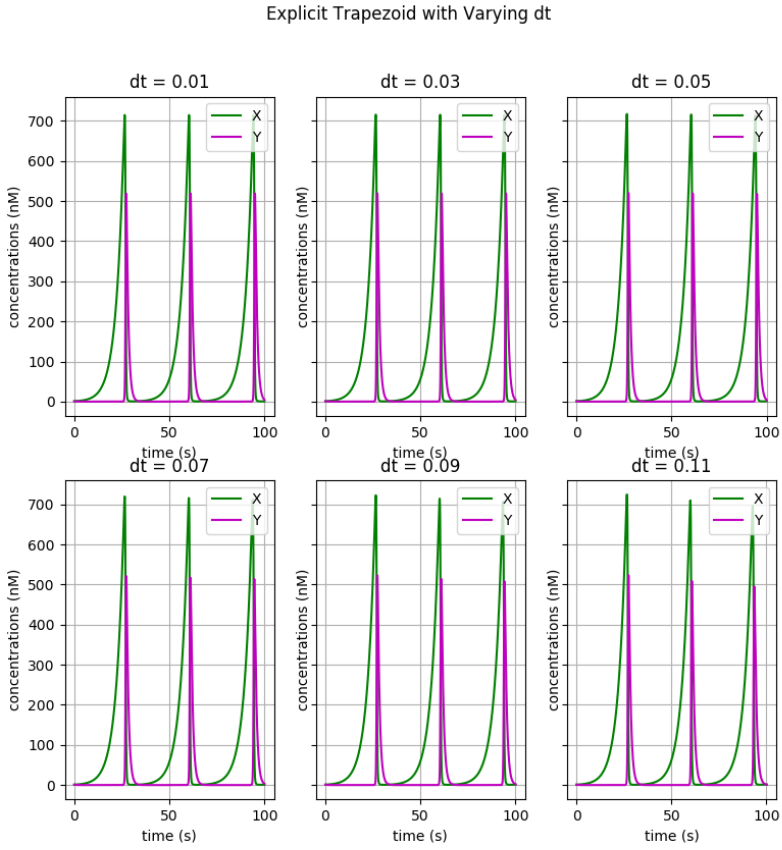

Figure 4: A simulation of the Lotka-Volterra ODE with parameters $\alpha = .25$, $\beta = .01$, $\gamma = 1$, $\delta = .01$. This is simulated with the Explicit Trapezoid method at various sizes of time steps.

2.3 Performance

Using this method to evaluate one thousand seconds of two proteins degrading with time steps of .001, I found that the runtime was 25.8 seconds.

3 Forward Euler with Predictive Time-Steps

3.1 Method

This method is an adaptation of the Forward Euler method. It is similar in every way to Forward Euler, but instead of taking a constant, input time step size (Δt), it continually calculates the error as the differential system is being evaluated, and adapts the size of the time steps as it is running. The way that it calculates the error is to use the Explicit Trapezoid method as a better approximation of the solution, and then uses the difference between that and the Forward Euler method as the error. The user inputs an error threshold, and the system adapts it's time steps to make sure that the error is never above that threshold - but keeps them as big as possible while staying below that threshold so that performance is increased, while accuracy remains the same. The equation used to update the step size as the system is running is as follows:

$$h_{new} = h\sqrt{0.9\frac{EST}{err}}$$

Where h_{new} is the new size of the time step, h is the old size, EST is the error tolerance, and err is the error. As we can see, if the error is above the error tolerance, the time step size goes down, and vice versa if the error is below the error tolerance.

3.2 Accuracy

The accuracy of this method is entirely dependent on the error tolerance that the user chooses. That said, we definitely get a very good looking graph, which we can see in Figure 5.

3.3 Performance

Obviously, comparing the runtime of this method to the others is slightly problematic, since the main determination of runtime is the number of time steps that we use. Using this method to evaluate one thousand seconds of two proteins degrading with an error tolerance of $5 \cdot 10^{-9}$, I found that the runtime was 1.4 seconds. Obviously this is a bad comparison to the other methods, but it is a fair point to make that it provides far higher error tolerance than would be normally used, and still runs far faster than either of the other solution methods. It is worth noting, of course, that the actual efficiency of this method will depend in large part on the actual ODE being solved, because it will take larger time steps if the ODE is changing slowly, so a slow ODE will have a faster solution

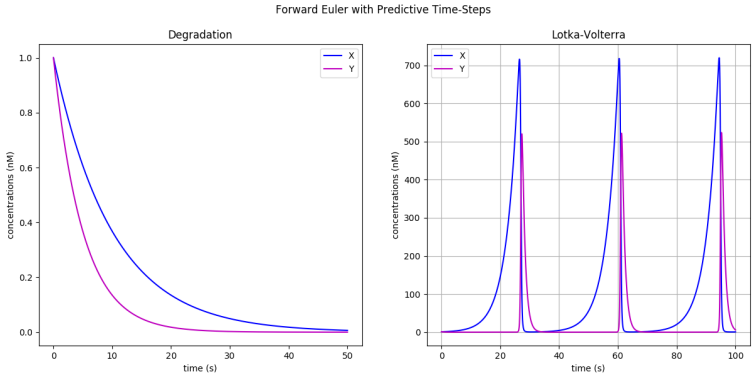

Figure 5: A demonstration of the Forward Euler method with predictive time steps. We used a degradation ODE with 2 variables and the Lotka-Volterra model with the parameters listed above.

time, while a fast ODE will have a slow solution time.

It is worth looking at the chosen time steps for the ODE that we calculated performance with - Figure 6 is a chart looking at the way these change over time as the simulation runs. Note that none of the error tolerances are as low as those we use for the performance run, and that at lower tolerances we tend to get a fairly consistent time step size.

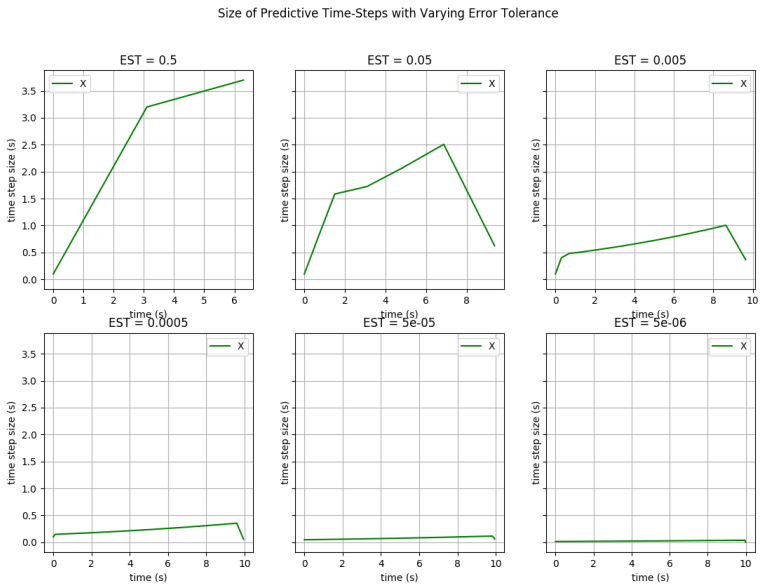

Figure 6: The size of time steps that our predictive algorithm chose for various error tolerances while simulating the simple degradation ODE we've been using in this paper.

4 Extensions

4.1 Runge-Kutta 4

This method is a fourth-order version of the Explicit Trapezoid method. Instead of using the average of two steps along the timeline, it uses the weighted average of four steps, according to this equation:

This method is more accurate but also more time-consuming than the lower-order Runge-Kutta methods. I implemented and quickly tested that it worked with both of the ODEs. Figure 7 shows that this method works well for simulating our ODEs. Also, with the same timing challenge as before, but with 100 instead of 1000 seconds, I get that this method takes 12.4 seconds to complete its simulation. The reason I used 100 instead of 1000 is that using 1000 was taking prohibitively long to run. It is worth noting that even though I wrote the method to be as efficient as possible, it is still running python, not something like Cython or C. This must be the reason that it is SO slow, but the point is that it is much slower than the other methods, as expected.

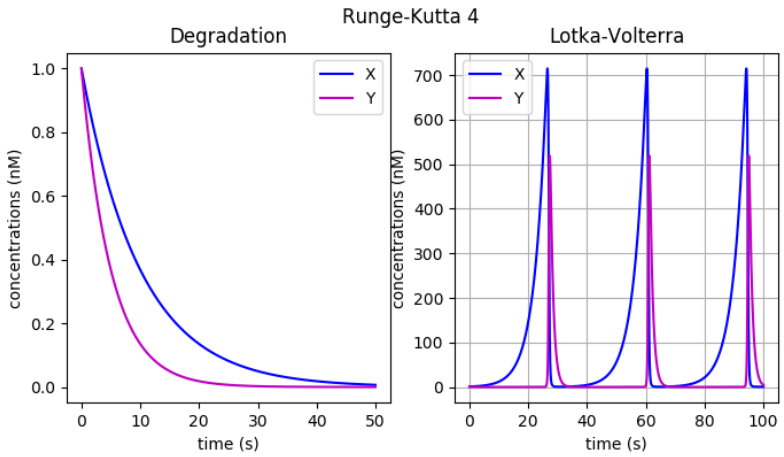

Figure 7: A demonstration of the Runge-Kutta 4 method. We used a degradation ODE with 2 variables and the Lotka-Volterra model with the parameters listed above.

4.2 Predictive Time-Steps with set output time-steps

I implemented the option for the user to set the size of the time-steps that the predictive algorithm outputs - the way this is done is by using a linear approximation between the larger time-steps to get the values at the time-steps that the user wants, so that the performance gains are sustained. The relevant code for this is shown here:

```
if (dt_out != None):
        while (tmp_t < t_cur - dt_out):
                slope = np.divide((np.array(sol[i + 1]) - np.array(sol[i])), h)
                tmpnxt = np.array(sol_out[j] + (dt_out * slope), ndmin=2)
                sol_out = np.concatenate([sol_out, tmpnxt], axis=0)
                j = j + 1
                tmp_t = tmp_t + dt_out
                t_out.append(tmp_t)
```