16-299 Spring 2021: State Space Numerical Integration

George Kantor

11 February 2021

1 Numerical Integration

Finding an explicit solution to ODEs is hard. Fortunately for the most part we can generate approximate (arbitrarily accurate) solutions to state space ODEs numerically with a computer. That's what this section is all about.

There is a great description of this stuff in (everything you will ever want to know):
Press, W.H., Teukolsky, S.A., Vetterling, W.T, and Flannery, B.P., Numerical Recipes in C: The Art of Scientific Computing, Second Edition, Cambridge University Press, 1992.

1.1 Example Problem

Recall the mass-spring-damper system:

$$m\ddot{y} + \mu\dot{y} + ky = u,$$

or in state space form:

$$\dot{x} = \begin{bmatrix} x_2 \\ \frac{1}{m}(u - kx_1 - \mu x_2) \end{bmatrix}$$

If we consider a"lossless, unforced" mass-spring-damper system (i.e., $\mu = 0$, $u = 0$) with $m = 1$ and $k_s = 1$, the original ODE becomes:

$$\ddot{y} + y = 0.$$

The initial conditions $y(0) = 1$, $\dot{y}(0) = 0$ corresponds to holding them mass out at 1 with zero velocity. "Unforced" means that we let go of the mass at $t = 0$ and let the dynamics play out. This is one of the rare cases when we can get the exact solution:

$$y(t) = \cos(t).$$

To check that, we first confirm that $y(t)$ satisfies the ODE:

$$\dot{y}(t) = -\sin(t)$$

$$\ddot{y}(t) = -\cos(t).$$

$$\ddot{y} + y = -\cos(t) + \cos(t) = 0.$$

Next we check to see if $y(t)$ satisfies the initial conditions:

$$y(0) = cos(0) = 1$$

$$\dot{y}(0) = -\sin(0) = 0$$

So we have confirmed that $y(t)$ satisfies both the ODE and the initial conditions, which means it is the unique solution to the system, which means it predicts how the system will behave for $t > 0$.

Now that we have the exact solution, we can compute some approximate solutions with a computer to see how they compare.

1.2 Euler's Method

Euler's method assumes that the derivative is constant between the time t and $t + T$.

$$x(t + T) = x(t) + T\dot{x}(t) + O(T^2).$$

Here the $O(T^2)$ indicates that the approximation

$$x(t + T) \approx x(t) + T\dot{x}(t)$$

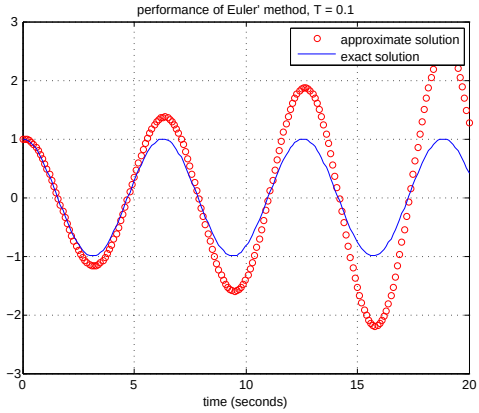
will introduce an error on the order of T^2 at each timestep. Substituting $\dot{x} = f(x, u)$ yields

$$x(t + T) \approx x(t) + Tf(x(t), u(t)).$$

This approximation and the fact that we know the initial condition $x(0)$ suggests an iterative method of approximating the signal $x(t)$ over longer time scales. We know $x(0)$, so:

$$\begin{aligned}
x(T) &\approx x(0) + Tf(x(0), u(0)) \\
x(2T) &\approx x(T) + Tf(x(T), u(T)) \\
x(3T) &\approx x(2) + Tf(x(2T), u(2T)) \\
&\vdots
\end{aligned}$$

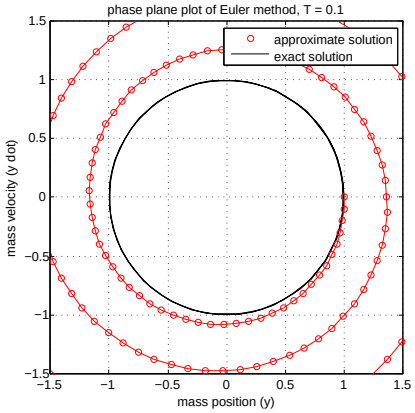
Let's just see how this works for an example (run `ex2`):

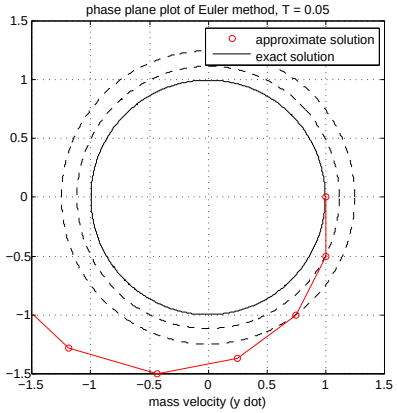


So, numerical solutions are approximations of the actual solution in two fundamental ways:

1. The solution is a sequence $\{x(0), x(T), x(2T, \ldots\}$ that approximates as continuous function $x(t)$. Sometimes we will adopt notation like $x[k] = x(kT)$ to make this notion more explicit.
2. The solution has error, and it accumulates with time.

In order to see why there is error, its easiest to look at the "phase-plane" plot for the system, i.e., the plot of $\dot{y}$ vs y. For a lossless mass-spring system (no damping) this plot should be a circular (or elliptical) trajectory, i.e., it should be periodic. So over each little timestep, the actual path is a curve that follows the arc of the circle. But when we use Euler's method, that arc is approximated by a straight line tangent to the circle at the starting point, causing the approximate solution to go outside the circle. As the integration is integrated, this error is compounded at each step, causing the approximate solution to spiral away from the actual solution.





1.3 The Midpoint Method

The Euler method assumes that the derivative over the entire time step is the same as at the beginning of the time step. The Midpoint method improves this approach by estimating the derivative in the middle of the timestep and using the result in an Euler-like step.

Notation: define $x_k = x(t)$ and $x_{k+1} = x(t+T)$.

The idea:

1. Compute that the Euler step would be. Call the Euler step k_1.

$$k_1 = Tf(x_k, u(t)).$$

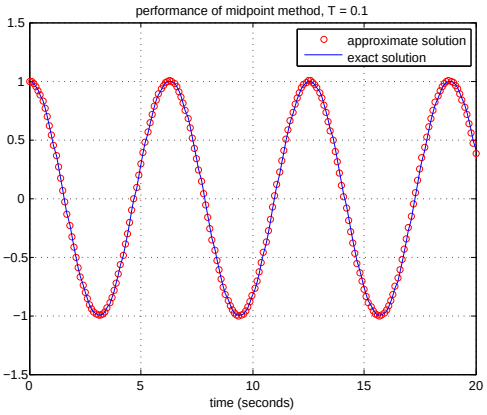
2. compute the step that results when you approximate the derivative at the midpoint between x_k and $x_k + k_1$. This will be a more accurate estimate of the derivative than Euler alone. Call this step k_2.

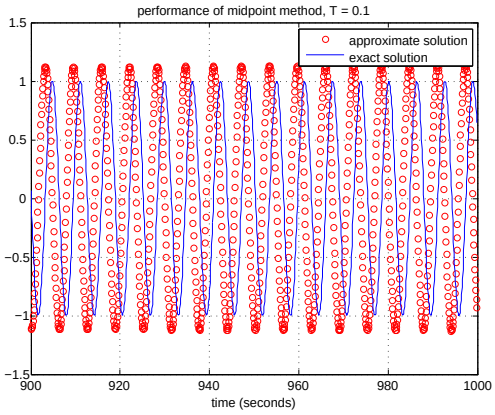
$$k_2 = Tf(x_k + \frac{1}{2}k_1, u(t + 0.5T)).$$

3. use this step to compute x_{k+1}

$$x_{k+1} = x_k + k_2 + O(T^3).$$

The midpoint method is also called *second order Runge-Kutta* because is it accurate to the second order. Here's a figure showing the performance of the midpoint method on the mass-spring example (run `ex3.m` and `ex4.m`):





1.4 Fourth Order Runge-Kutta

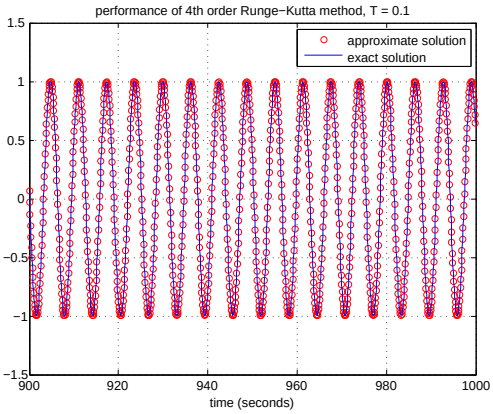
This is the most common because it has the best tradeoff between computational efficiency and accuracy:

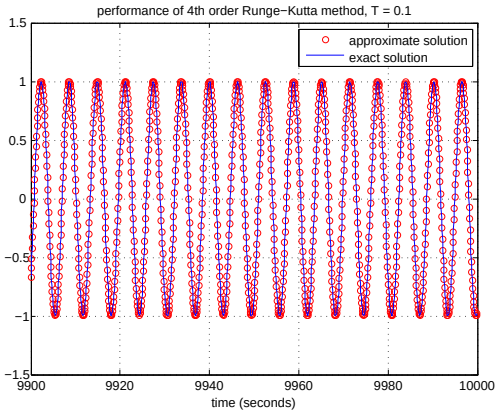
$$k_1 = Tf(x_k, u(t))$$
$$k_2 = Tf(x_k + 0.5k_1, u(t + 0.5T))$$
$$k_3 = Tf(x_k + 0.5k_2, u(t + 0.5T))$$
$$k_4 = Tf(x_k + k_3, u(t + T))$$

Then

$$x_{k+1} = x_k + \frac{k_1}{6} + \frac{k_2}{3} + \frac{k_3}{3} + \frac{k_4}{6} + O(T^5).$$

Here are the plots (run `ex5.m` and `ex6.m`.)





1.5 Other Stuff You Can Do

variable stepsize Runge-Kutta methods.
MATLAB solvers (*e.g.*, `ode45`).