Primer on Ordinary Differential Equations (ODEs)

For the DynamicsLab project: Sprint 2 (3D Motion)

Written by Gavin Fielder. gavinfielder@gmail.com

Basic Ideas

A **function** takes one or more inputs and produces one output. Ex: $x(t) = 5t^2 + 3$.

An **independent variable** is an input into a function. Ex: t

A **state variable** is the output of a function. Ex: $x(t)$, or just x.

A **derivative** is the rate of change of a state variable over time (or more generally, the instantaneous slope over its independent variable).

For example, for the function $x(t) = 5t^2 + 3$, the derivative is $x'(t) = 10t$. This means that at $t = 0, x'(0) = 10(0) = 0$ meaning there is no instantaneous change or direction, but at $t = 2, x'(2) = 10(2) = 20$, the function is increasing rapidly. Negative values would indicate the function is decreasing.

Since derivatives are also functions, derivatives have derivatives—this would be the **second derivative**. Many functions have an infinite number of derivatives. Some don't, but we won't have to worry about that.

While the most explicit expression of a derivative function is $\frac{dx}{dt}$, (which specifies the derivative of function x with respect to t), this type of notation is not important for what we're doing right now. More simply, derivatives can be expressed as $x'(t)$ or $\dot{x}(t)$, or simply as x' or $\dot{x}$, the independent variable being understood in context. The second derivatives can be expressed $x''(t)$ or $\ddot{x}(t)$, or simply as x'' or $\ddot{x}$.

Ordinary Differential Equations (ODEs)

A **differential equation** is an equation which relates a state variable with its derivative.

Ordinary differential equations (ODEs) involve equations of *only one independent variable*—this is in contrast to **partial differential equations (PDEs)**, which can analyze derivatives of functions with *multiple independent variables*.

We're only doing ordinary differential equations right now. Henceforth, I will say "differential equation" but specifically, I will mean an ODE.

One differential equation could be:

$$\dot{x} = 7x + 5t$$

In the above example, the independent variable shall be understood to be t, so even if we don't write it, we know x and $\dot{x}$ are functions of t. Notice how the differential equation can involve both the state variable x and independent variable t.

A **first-order differential equation** relates a state variable and it's first derivative. A **second-order DE** relates a state variable and its first and second derivatives. The spring-mass system is a second-order DE, while the example above is a first-order DE.

We are interested in knowing what the actual state variable function is. That is, we wish to know $x(t)$. For this problem, we can solve to find:

$$x(t) = -\frac{5}{7}t - \frac{5}{49} + ke^{7t}$$

Notice there's a new element: the symbol k. Here k is an arbitrary constant, meaning any value of k makes this a valid solution for the differential equation. This means this is not a single solution, but an entire *family of solutions*. By setting certain conditions in the problem, we can determine a particular value for k.

Initial Value Problems

An **initial value problem (IVP)** specifies an *initial state* that allows us to calculate a particular value of k. For example, an initial value problem could be:

$$\dot{x} = 7x + 5t$$

$$x(0) = 1$$

Here we have the same equation as the previous section's example, but we've also specified that at time 0, the value of the state variable should be 1. By imposing this condition, we can determine that $k = \frac{54}{49}$, making the solution to this initial value problem:

$$x(t) = -\frac{5}{7}t - \frac{5}{49} + \frac{54}{49}e^{7t}$$

It doesn't have to be time 0—we can also specify an "initial condition" for time 5, were we to see $t = 5$ as our "starting point" of solution.

Second-order equations will produce *two* arbitrary constants. Therefore a valid second-order initial value problem needs to specify the initial value of the state variable, but also the initial value of its first derivative. For example, a valid second-order initial value problem could be:

$$\ddot{x} = 2\dot{x} + 5x$$

$$x(0) = 1$$

$$\dot{x}(0) = -1$$

The spring-mass system is also a second-order initial value problem: it needs an equation, but in order to start generating a specific solution, it also needs to know its initial position and initial velocity.

Systems of Ordinary Differential Equations

A **system of ODEs** involves more than one state variable. For example,

$$\begin{cases} \dot{x} = 2x + 3y \\ \dot{y} = x - 2y + t \end{cases}$$

Note that $\dot{x}$ depends on both x and y. $\dot{y}$ depends on both x and y, and in its case, also t.

An initial value problem (IVP) for a system of ODEs could look like this:

$$\begin{cases} \dot{x} = 2x + 3y \\ \dot{y} = x - 2y + t \end{cases}$$

$$x(0) = 1$$

$$y(0) = 2$$

Notice that in order to have a particular solution to this IVP that involves a *2-dimensional system*, we need to specify *two* initial conditions, one for each of the state variables.

Expanding this, were we to have an IVP involving a ***three**-dimensional system of **second** order ODEs*, we would then need ***6*** initial conditions. This will be the case for 3D motion: we need initial values for x position, y position, and z position, and also initial values for x velocity, y velocity, and z velocity.

Numerical Methods

Many differential equations and IVPs cannot be solved analytically (i.e., we cannot find an explicit solution as I have done in the examples above. In this case, numerical methods involve approximating a solution—essentially generating a table of values that the solution will follow very closely.

One important concept in numerical methods is what I call the **resolution** of the numerical solution, meaning how finely populated is the data. The measure I generally use for this is h. h determines the amount of time (or whatever the independent variable is) between data points in the solution. I'll normally say h=0.1 to be a "resolution of 0.1". If time is the independent variable and its units are in seconds, the frequency would then be 10 data points per second. The reason resolution is important is that it determines a balance between runtime complexity and memory usage, and solution accuracy—but the Runge-Kutta method that we're using for IVPs is very accurate even for coarse solutions, so we shouldn't need to worry about this tradeoff unless we start doing very complicated problems that makes the system lag.

The **Runge-Kutta method** is a numerical algorithm which operates on a system of first order ODEs—the first order part is important to note—and generates a numerical solution for an initial value problem.

From its initial conditions, it iteratively generates values of the state variables over time. The way it does this is by accepting a state vector (the conditions or state at the current time value) and generating the *next* state vector—for instance, if h=0.1, we would pass the t=5.6 s data to the Runge-Kutta method in order to generate the t=5.7 s data.

I can explain how the Runge-Kutta method works in more detail, but I'm not expecting anyone else to really need to understand it beyond what I've said here. In general, simply know that the InitialValueProblem class I've created "solves" IVPs by this iterative process of passing in state vectors to get the vectors at the next time step, and that we have the option to lower the resolution to speed up computations at the expense of some accuracy.

Turning a second-order equation into a system of first-orders

The Runge-Kutta method only works on first order systems. But the spring-mass is a second-order differential equation. In order to solve it, we need to make substitutions to turn a second-order equation into a system of first orders.

Here is the second order differential equation for the spring mass system. Here b, m, k are constants (damping, mass, and spring stiffness)

$$\ddot{x} = -\frac{b}{m}\dot{x} - \frac{k}{m}x$$

By making these substitutions:

$$u = x$$

$$v = \dot{x}$$

Note that the derivative u will also be the derivative of x, and also the derivative of v is the derivative of $\dot{x}$, i.e., it would actually be the second derivative of x, $\ddot{x}$. In other words:

$$\dot{u} = \dot{x} = v$$

$$\dot{v} = \ddot{x} = -\frac{b}{m}\dot{x} - \frac{k}{m}x = -\frac{b}{m}v - \frac{k}{m}u$$

Thus by using these substitutions, we can rewrite the second order as a system of first orders:

$$\begin{cases} \dot{u} = v \\ \dot{v} = -\frac{b}{m}v - \frac{k}{m}u \end{cases}$$

In general, this is what we'll have to do for any second-order differential equation in order for it to be solvable by the Runge-Kutta method:

$$\ddot{x} = F(\dot{x}, x) \rightarrow \begin{cases} u = x \\ v = \dot{x} \end{cases} \rightarrow \begin{cases} \dot{u} = v \\ \dot{v} = F(v, u) \end{cases}$$

From the second order equation, assign these new symbols, and then generate the system on the right.

Turning a system of second orders into a system of first orders

For 3D motion, we will need to enable the user define 3 equations of motion: one which determines x acceleration, one which determines y acceleration, and one which determines z acceleration. Each of these equations are second order equations, and to make it general we'll let the user write equations which depend on all of the state variables.

This is the transformation of a system of second order equations into a system of a first-orders.

$$\begin{cases} \ddot{x} = F(x, \dot{x}, y, \dot{y}, z, \dot{z}, t) \\ \ddot{y} = G(x, \dot{x}, y, \dot{y}, z, \dot{z}, t) \\ \ddot{z} = H(x, \dot{x}, y, \dot{y}, z, \dot{z}, t) \end{cases} \rightarrow \begin{cases} u_1 = x \\ u_2 = \dot{x} \\ u_3 = y \\ u_4 = \dot{y} \\ u_5 = z \\ u_6 = \dot{z} \end{cases} \rightarrow \begin{cases} \dot{u}_1 = u_2 \\ \dot{u}_2 = F(u_1, u_2, u_3, u_4, u_5, u_6, t) \\ \dot{u}_3 = u_4 \\ \dot{u}_4 = G(u_1, u_2, u_3, u_4, u_5, u_6, t) \\ \dot{u}_5 = u_6 \\ \dot{u}_6 = H(u_1, u_2, u_3, u_4, u_5, u_6, t) \end{cases}$$

So for the user input fields for each object they create for the simulation, the user simply needs to specify the functions F, G, and H, and in each function they have access to 7 symbols: $x, \dot{x}, y, \dot{y}, z, \dot{z}, t$. How the derivatives can be represented in the string is for investigation—hopefully our parser will allow them to use `x'`, `y'`, and `z'`.

Without redefining each symbol, we could express the transformation as:

$$\begin{cases} \ddot{x} = F(x, \dot{x}, y, \dot{y}, z, \dot{z}, t) \\ \ddot{y} = G(x, \dot{x}, y, \dot{y}, z, \dot{z}, t) \\ \ddot{z} = H(x, \dot{x}, y, \dot{y}, z, \dot{z}, t) \end{cases} \rightarrow \begin{cases} \dot{u}_1 = \dot{x} \\ \dot{u}_2 = F(x, \dot{x}, y, \dot{y}, z, \dot{z}, t) \\ \dot{u}_3 = \dot{y} \\ \dot{u}_4 = G(x, \dot{x}, y, \dot{y}, z, \dot{z}, t) \\ \dot{u}_5 = \dot{z} \\ \dot{u}_6 = H(x, \dot{x}, y, \dot{y}, z, \dot{z}, t) \end{cases}$$

That should save on our workload. This way the 3 user input fields will end up (in the code) being the 2nd, 4th, and 6th dimensions of the system, and the 1st, 3rd, and 5th dimensions will be predefined.

We may later need transformations for third-order and higher equations, or higher dimensionality systems, but I'm not going to bother trying to do that in general right now. For sprint 2, the above transformation is what we'll have to implement.