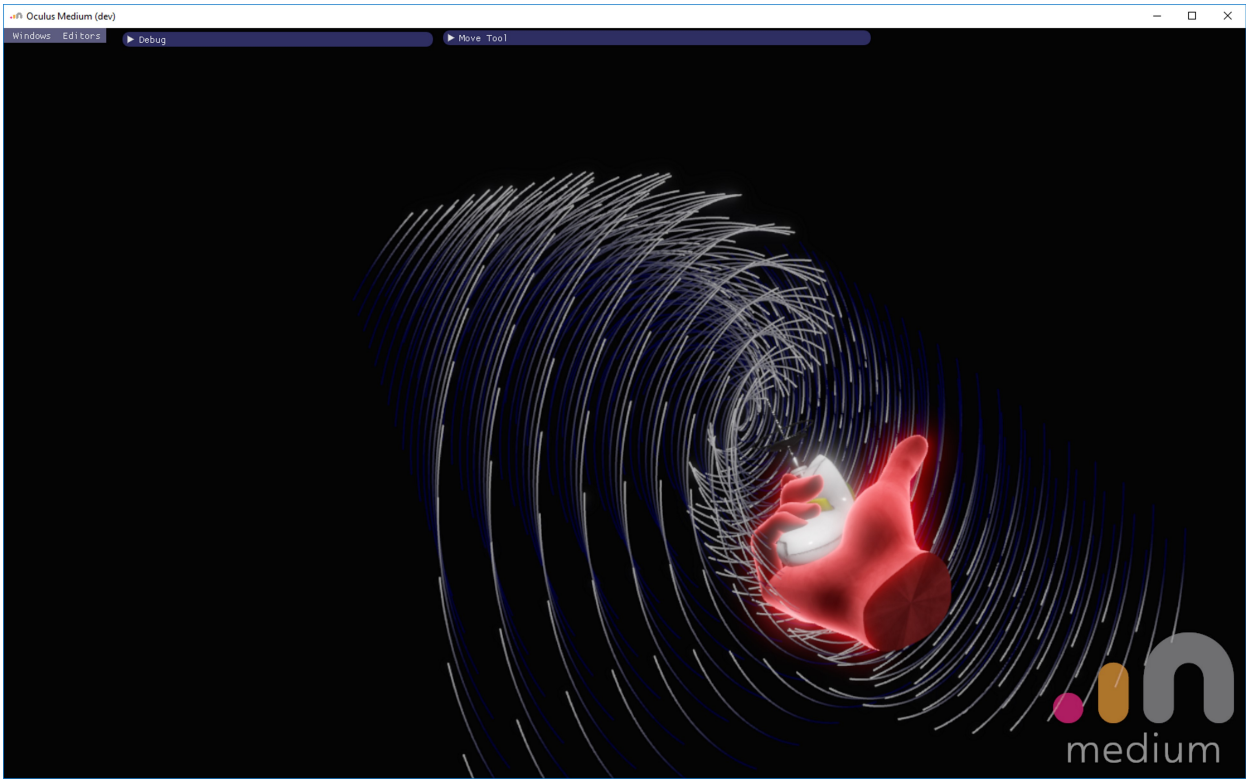
Notes on ODE Solvers

Sunday, March 17, 2019 8:01 AM

This document contains some notes about the differential equation solvers in the sample source code at https://github.com/fbsamples/sculpting-and-simulations-sample.



Adaptive Step Size

Monday, April 16, 2018 10:41 PM

An efficient way to numerically integrate ODE's is to use an adaptive time step solver. These solvers adaptively change the step size so that smaller steps are taken when the function is rapidly changing, and larger steps are taken when the function is closer to constant.

Adaptive step size solvers need an estimate of how much error was created by the last step size, and how much error is tolerable. All numeric ODE solvers create some error, and we want to adjust the step size so that the error stays small, but also that the step size stays large (so the numeric integration can be quickly computed). In the next section, we'll discuss methods to estimate this error.

To adaptively change the time step size, we relate the ratio of the new and current timesteps to the ratio of the desired and actual error. We know h (our current step size), as well as our desired and actual error, so we have:

$$\left(\frac{h_{new}}{h}\right)^{p+1} = \frac{error_{desired}}{error_{actual}}$$

The above formula uses p+1 because the error scales proportional to O(h^{p+1}). Euler's method has an error of O(h^2), and RK4 has an error of O(h^5).

That can be rearranged to find the new timestep size h_{new}:

$$h_{new} = h * safety * \left(\frac{error_{desired}}{error_{actual}}\right)^{\frac{1}{p+1}}$$

The $safety$ term is a fudge factor so that the new time step doesn't overshoot the edge of tolerable error. The above equation will advance time by as much as possible so that the new error is at the desired error. However, the actual error is just an estimate, and we don't know the future of the function, so it's possible for the above equation to overshoot (advance to a time where the error is greater than the desired error). We compensate for this by using a factor to pull the new time step back a bit. You don't want this factor to be too large (you will overshoot too often) or too small (you'll use a smaller timestep than necessary, wasting performance). In Oculus Medium, we use a safety factor of 0.9.

The $error_{desired}$ term is a constant value that should be chosen specifically for the problem. Every application has different needs for how much error can be tolerated. It's important to realize that these ODE solvers always contain error; the application must be designed to handle some amount of error.

The $error_{actual}$ term above is an estimate of how much error the integration step actually created. Finding the actual error can only be done accurately by running the integration twice: once with some very accurate way, and another with some other way, and comparing the results. There are two ways we'll look at to compute the error: step doubling (which takes two steps of half a time step each, and a full time step), and embedded Runge-Kutta methods (which compute two answers at the same time).

For Oculus Medium's Move Tool, we use a desired error of 0.1 in voxel space, meaning that the error should be no more than 10% the size of a single voxel. Using a larger error is faster to compute, but you can begin to see strange artifacts in the sculpt.

Step Doubling

Monday, April 16, 2018 10:41 PM

The simplest adaptive step solver uses step doubling. This solver finds two answers to the problem: the first answer takes a step of time h, and the second answer takes two steps of time h/2. Step doubling produces a smaller error than the embedded methods described in the next section, but uses 12 function evaluations per step.

In general, with step doubling, we find two answers y0 and y1. y0 is calculated from a single step of size h, and y1 is calculated from two steps of size h/2. The error is then computed as:

$$error = \frac{|y0 - y1|}{2^p - 1}$$

The Runge-Kutta integrator we use is order 4, so for a step doubling RK4 integrator, the error is |y0-y1|/15. This error can be used as the $error_{actual}$ term in the adaptive time stepping equation we used earlier.

DERIVING THE STEP DOUBLING ERROR

Where does the term $2^p - 1$ come from? Consider that we have two answers, y0 and y1, which equal the true answer y plus some error:

$$y = y0 + C\,h^{p+1} + O(h^{p+2})$$
$$y = y1 + 2\,C\,(h/2)^{p+1} + O(h^{p+2})$$

The first took a single step of size h, and the second took a two steps of size h/2.

We can then solve for the unknown error C:

$$y0 - y1 = 2\,C\,(h/2)^{p+1} - C\,h^{p+1}$$
$$y0 - y1 = C\left(\frac{h^{p+1}}{2^p} - h^{p+1}\right)$$
$$y0 - y1 = C\left((\frac{1}{2^p} - 1)h^{p+1}\right)$$
$$|y0 - y1| = C\left((1 - \frac{1}{2^p})h^{p+1}\right)$$
$$C = \frac{|y0 - y1|}{\left(1 - \frac{1}{2^p}\right)h^{p+1}}$$

We substitute that C into the second equation above (because the two half steps will generate a accurate answer than one full step):

$$y = y1 + 2\,\frac{|y0 - y1|}{\left(1 - \frac{1}{2^p}\right)h^{p+1}}\,(h/2)^{p+1} + O(h^{p+2})$$

Let's call the second term e (for error), and simplify it:

$$e = 2\ \frac{|y0 - y1|}{\left(1\ - \frac{1}{2^p}\right)h^{p+1}}\ (h/2)^{p+1}$$

$$e = 2\ \frac{|y0 - y1|}{1\ - \frac{1}{2^p}}\ \frac{1}{2^{p+1}}$$

$$e\ = \frac{|y0 - y1|}{1\ - \frac{1}{2^p}}\ \frac{1}{2^p}$$

$$e = \frac{|y0 - y1|}{2^p - 1}$$

So, our second equation becomes:

$y\ =\ y2\ +\ e\ +\ O(h^{p+2})$, where $e = \frac{|y0-y1|}{2^p-1}$

For fourth order Runge-Kutta, p is equal to 4, so our error estimate is |y0-y1|/15.

Embedded Methods

Monday, April 16, 2018

Embedded methods are a class of explicit ODE solvers that return a pair of solutions at each step, where one solution is order P and the other is order P+1. With two solutions, we can estimate how much error was made by the integration step. Embedded methods are cleverly constructed so that the two solutions share intermediate computations, making them much cheaper to compute than a step doubling solver, but possibly generating larger error per step.

To calculate the error for an embedded method, there are many suitable error metrics to choose from, such as the Euclidean difference between the pair of answers, or their Manhattan distance, or some other scheme. Oculus Medium uses the Euclidean distance. Given $\vec{a}$ and $\vec{b}$, the embedded method's pair of answers, the error is calculated as:

$$error = \frac{length\left(\vec{a} - \vec{b}\right)}{dt}$$

We examine three different embedded methods: Runga-Kutta-Fehlberg 4(5), Dormand-Prince 4(5), and Bogacki-Shampine 3(2).

RKF4(5) is a fifth order accurate method using six evaluations per step.

DP4(5) is also a fifth order accurate method, and has the First-Same-As-Last (FSAL) property. It uses seven evaluations per step (six if using FSAL).

BS3(2) is a third order accurate method and uses FSAL. It uses three evaluations per step (two if using FSAL). However, these fewer steps mean that the method has a lower order, and thus produces more error per step. Depending on the function that's being evaluated, more error per step can mean that you have to take more steps, and so BS3(2) is not always a win.

First Same As Last

Tuesday, October 23, 2018 6:06 PM

Embedded methods have been well studied by researchers, and a clever property of both the DP 4(5) and BS 3(2) methods is known as First Same As Last (FSAL). This property means that the last function evaluation at the end of a successful step is the same as the initial function evaluation at the next step, so we can reuse that evaluation instead of recomputing. This increases efficiency since fewer function evaluations are needed.

Let's look at BS 3(2) as an example, as it's simpler to examine than DP 5(4). The BS 3(2) method is computed as:

$$k_1 = f(\vec{p}(t), t)$$
$$k_2 = f\left(\vec{p}(t) + \frac{1}{2}h\,k_1, t + \frac{1}{2}h\right)$$
$$k_3 = f\left(\vec{p}(t) + \frac{3}{4}h\,k_2, t\ + \frac{3}{4}h\right)$$
$$\vec{p}(t+h) = \vec{p}(t) + \frac{2}{9}h\,k_1 + \frac{1}{3}h\,k_2 + \frac{4}{9}h\,k_3$$
$$k_4 = f(\vec{p}(t+h), t+h)$$
$$\hat{p}(t+h) = \vec{p}(t) + \frac{7}{24}h\,k_1 + \frac{1}{4}h\,k_2 + \frac{1}{3}h\,k_3 + \frac{1}{8}h\,k_4$$

Where $\vec{p}(t+h)$ and $\hat{p}(t+h)$ are the third and second order answers, respectively.

You can see how step size k_4 in this step is equal to k_1 in the next step. If the adaptive integrator can use the current step size (meaning, the error in this step is small enough), then we can reuse the k_4 value for the start of the next step. When this happens, the number of function evaluations for a BS 3(2) step is reduced from four to three.

The sample source code does not take advantage of the FSAL property, but it is a clear opportunity for optimization.

Implementation Details

Wednesday, April 18, 2018 12:18 PM

In the sample source code, the file odesolvers.h contains the ODE integrators and solvers.

The EVALUATE() function should be a pure function with no side effects. The adaptive techniques need to call EVALUATE(), and can throw away results.

The initial step size should be chosen with care. In Medium's Move Tool code, we tried to use an initial step size of 1.0, reasoning that the adaptive methods would reduce the step size if they found too much error. However, in practice, the adaptive methods aren't perfect, and a step size of 1.0 didn't work well because it skipped over too much of the function. Instead, we found that an initial step size of 0.1 worked well for our problems. There's no single right answer for the initial time step size that fits all equations.

There is also a minimum step size that should be considered. The solvers are designed to always return an answer, even if the adaptive timestepping starts to spiral into tiny timesteps. We clamp the minimum time step to a small size. This means that the solver is guaranteed to complete, but it is not guaranteed to return a very accurate answer. In development, we assert when this happens in Oculus Medium, and tune the settings so that this does not happen (increasing the max error, decreasing the initial step size, etc.) This works well for a sculpting application, and because we run the deformation in a vertex shader, we need to guarantee the solver will complete in a reasonable amount of time (or the GPU will time out). However, handling this error case is very application specific.

The max error is in Oculus Medium is set to 0.1 in voxel space (10% of the size of one voxel), which is about 0.00013 in world space. We set the max error proportional to voxel size because you can scale a sculpt (the voxel grid) up and down by a considerable amount-- but you want the same amount of error relative to the size of a voxel.

The max error setting is the most important part of the solver to tune. Too large of a max error and you'll see artifacts; too small of a max error will become performance prohibitive. This tradeoff is, again, very application specific.