Cauchy Timestep Adjustment in End-User Simulations

Ted Kaehler

VPRI Research Note RN-2017-001

Viewpoints Research Institute, 1025 Westwood Blvd 2nd flr, Los Angeles, CA 90024 t: (310) 208-0524

Cauchy Timestep Adjustment in End-User Simulations
April 2017
by Ted Kaehler

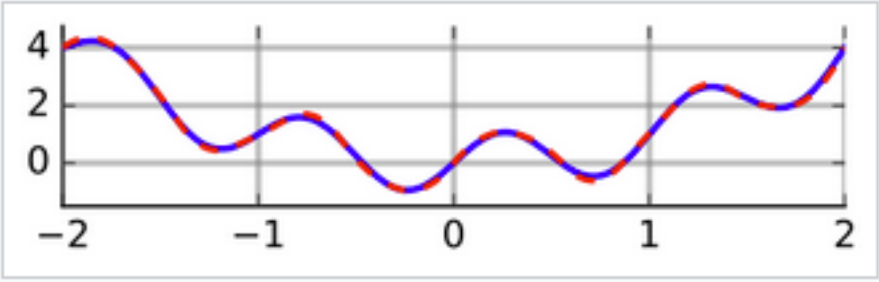
Abstract

This paper explains how Scratch, Etoys, or GP can have the Cauchy Criterion built in. With minimal effort, the user can have control of the time stepsize in simulations.

One of the problems in simulations is using a stepsize that is not correct. If the stepsize is too small, the simulation takes too long and uses too much compute time. If the stepsize is too big, the simulation does not capture detail. It skips over times where unusual things are happening. When a ball approaches a wall, a simulation of it has code to change the ball's velocity and direction when it contacts the wall. If the stepsize is too big, the ball will go right through the wall. The conditional that detects the wall will not run until the ball has already passed through it.

Background

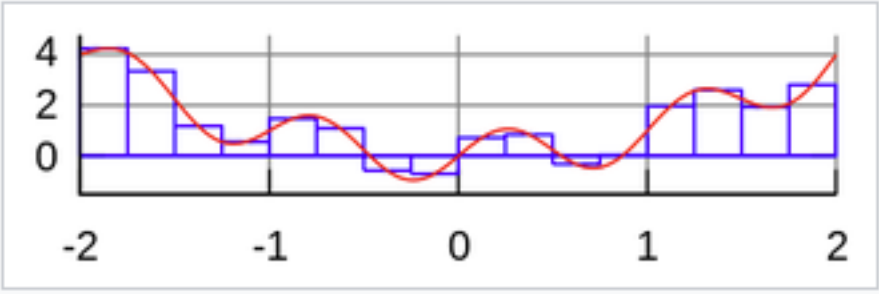
Numerical Integration in math -- finding the area under a curve -- has the same problem. A curve is divided into segments, and the height in the middle of each segment is used to compute the area under the curve. If there are too few divisions, many wiggles and changes in the curve are missed. The computed area is either too low or too high.



(Wikipedia, Numerical Integration)

If we choose divisions that are 2 units wide, height measurements will only be made at x = -2, 0, and 2. Adding up those values will be quite different than the true area under the curve.

One way to improve the accuracy is to make the divisions closer together. The error e is the absolute value if the true area minus the computed area. Using smaller and smaller divisions creates a series of error values, e(n). The distance between divisions is the stepsize. We know that this converges to an error of zero when there are infinitely many divisions.



(Wikipedia, Numerical Integration)

Here, the divisions are 0.25 wide, and the area under the curve is much more accurate.

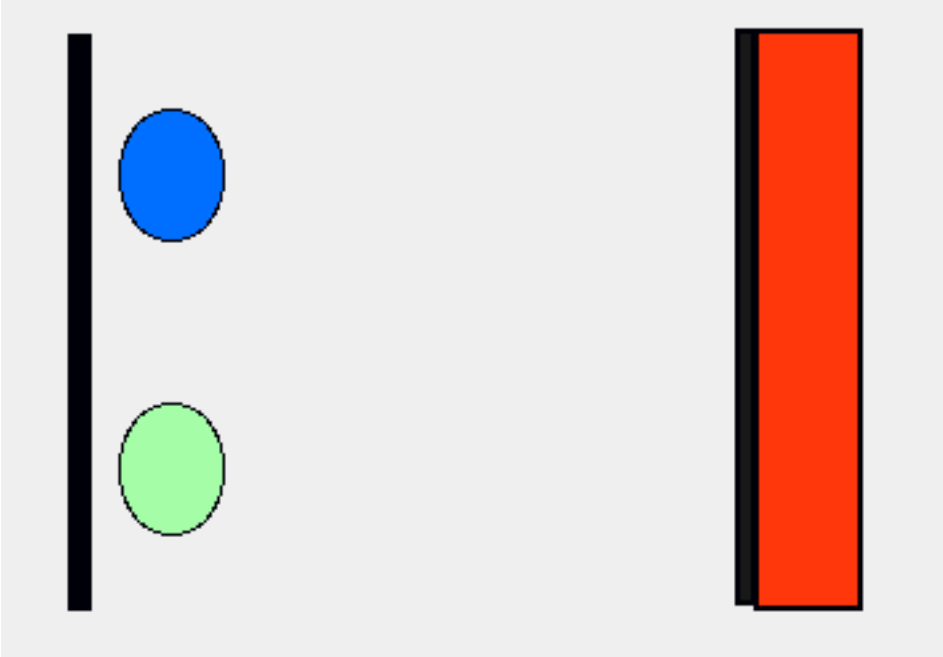
Intuitively, we can see that a smaller distance between divisions leads to a smaller error. Not only does this work for the area under any segment of a given curve, but it also works for any particular error calculation in a simulation.

"Given a desired error e, there is a stepsize S such that the answer using stepsize S will have an error smaller than e."

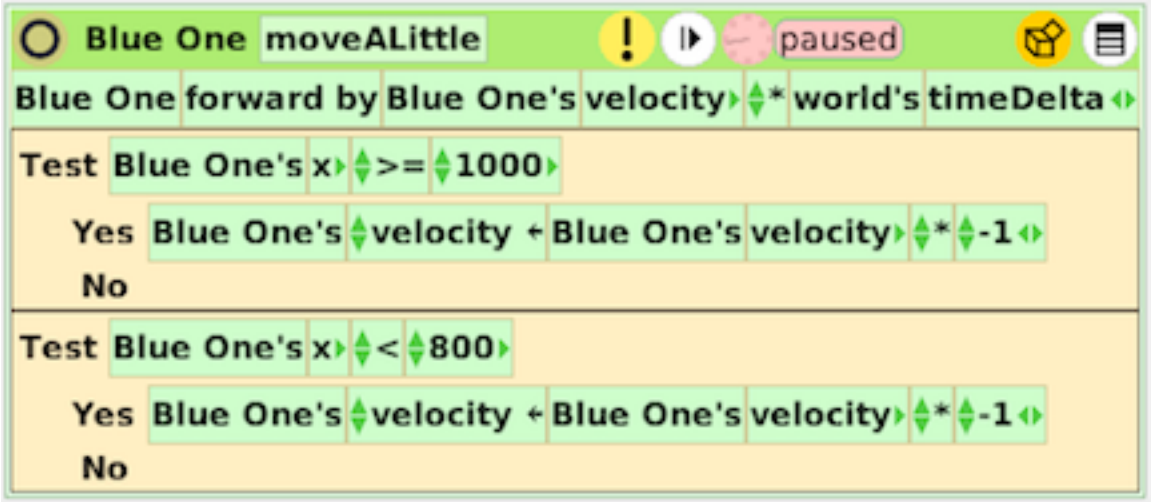
This is a version of the Cauchy Criterion for the convergence of a numerical series.

Controlling Errors in Simulations

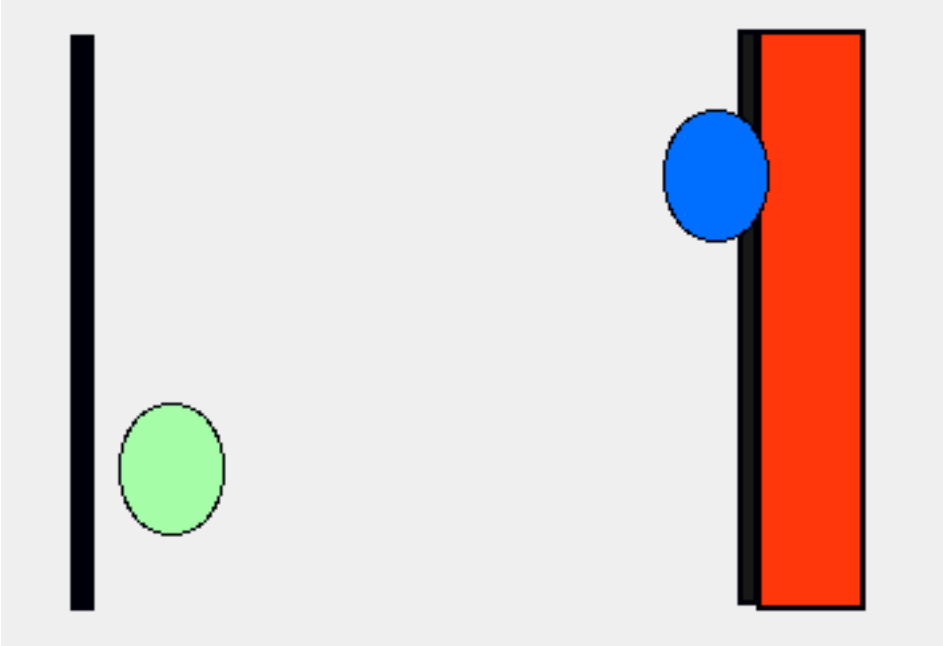
Consider two balls bouncing between two walls in the Squeak Etoys system.



The author uses a stepsize of 20 pixels. There is a ticking script to move each ball forward. Here is the script to make the blue ellipse move. The script says to reverse direction when the ellipse's X gets to the black wall at x=1000.



Using this script, the blue ellipse overshoots the wall and hits the red rectangle, which stops it. This is because the stepsize is too large. The X is incremented by 20 pixels at a time. The ellipse does not notice the wall until it has already gone beyond it.



We'd like the Etoys simulation system to help manage the stepsize. Only a small number of modifications to the system were needed to let the user easily get the proper stepsize. The things the user needs to do to her simulation are very simple and easy.

The first requirement is that the user write her motion script in terms of a timeDelta. The amount to go forward is (Blue One's velocity * world's timeDelta). The World object controls the increment in time for each step. When the ellipse is moving at a constant velocity, its movement on the screen will be the same, whether the timeDelta is large or small. Calling the moveALittle script a few times with a large timeDelta is the same as calling it many times with a small timeDelta. The system is free to choose any timeDelta to make the simulation work best.

The green ellipse is a copy of the blue ellipse. Its motion scripts is the same as before. But, the green ellipse has an additional script.

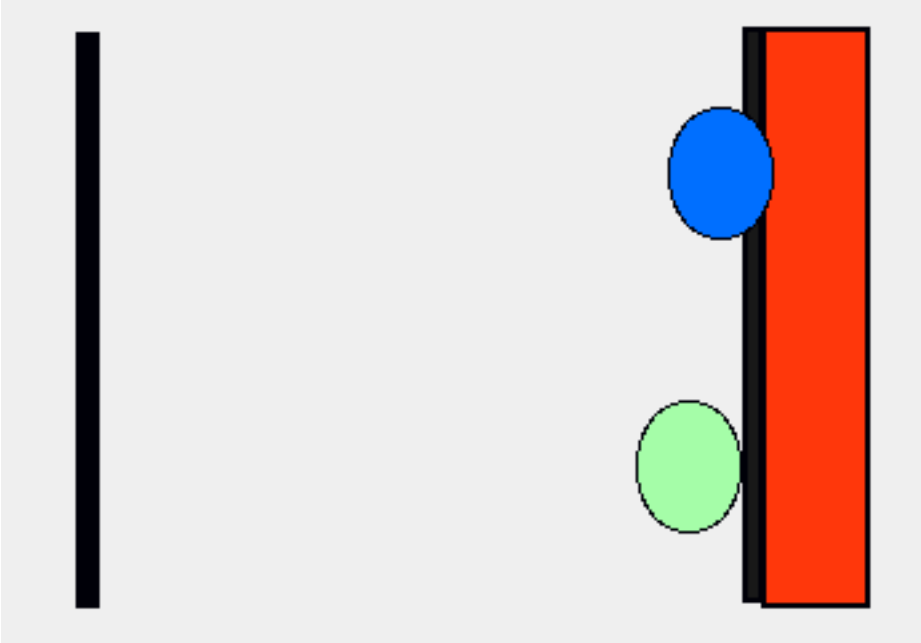
The script "cautionRight" measures how far the ellipse is from the turning point

at the right wall. It stores this quantity into the variable cautionNear0. That is a variable of the World, and can be set inside of any script.

The user sets cautionNear0 to an expression that she wants to be careful around. When cautionNear0 changes sign, the system reduces the time step. It forces the user's stepping scripts to be evaluated at a time when cautionNear0's sign has changed and is within "2" of zero.

If the number 2 is too big or small to be in the critical zone, the user can scale her expression by multiplying by some number before storing into cautionNear0.

Caution scripts ensure that the user's other scripts are run at the right time, and that important changes will be noticed.



Here is the green ellipse just touching the wall. The system has reduced the timeDelta and retried the motion several times until cautionNear0 is at an acceptable level.

How it Works

Every caution script that the user writes must have a name that begins with 'caution'. The system finds these scripts. At each time step, the system evaluates every caution script and saves the value of cautionNear0 for each. It then runs every ticking script using the current timeDelta. After that it runs every caution

script again to see if it changed sign. For the ones that did, is cautionNear0 < 2? If any are not, set all variables back to their values at the start of the time step, cut the timeDelta in half, and try again.

This procedure depends on both the caution scripts and the ticking scripts being able to be re-evaluated many times. The system fills a dictionary with the values of the variables at the start of a time step. After a trial evaluation, it uses the dictionary to set all of the variables back. The X and Y of each moving object are among the variables. To decide which variables to save, the system finds all setter calls in all user scripts and notes the variables being set.

This is very general. Any variable, not just positions can be changed, or can be used in a caution method. The system will apply the Cauchy Criterion to any variable.

After a time step is completed, the timeDelta is set back to its default value.

What the User Does

As mentioned above, everything that depends on time, such as motion, must be stated in terms of a timeDelta. In addition, the user must define caution scripts that set cautionNear0 to an expression. The value must be zero when something crucial is happening. That is all the user needs to do.

For a collision between two moving objects, a caution script needs to compute the distance between the objects. If there is a special technique for detecting collisions, a caution method can look at the output of it, and simply return the smallest distance between any two objects.

The System

It would be quite easy to make the changes to Scratch or GP to support this Cauchy-inspired control of time steps. The modifications to Etoys were minimal.

The changes are non-intrusive. If the user chooses not to define any caution scripts, or not to write motion in terms of timeDelta, her simulation will work exactly as before.

Conclusion

We have shown that an end-user system -- suitable for kids -- can incorporate the Cauchy Criterion. Using it does not place very much of a burden on the user.

Video demo: "Cauchy Timestep Adjustment" by Ted Kaehler, https://vimeo.com/214582477