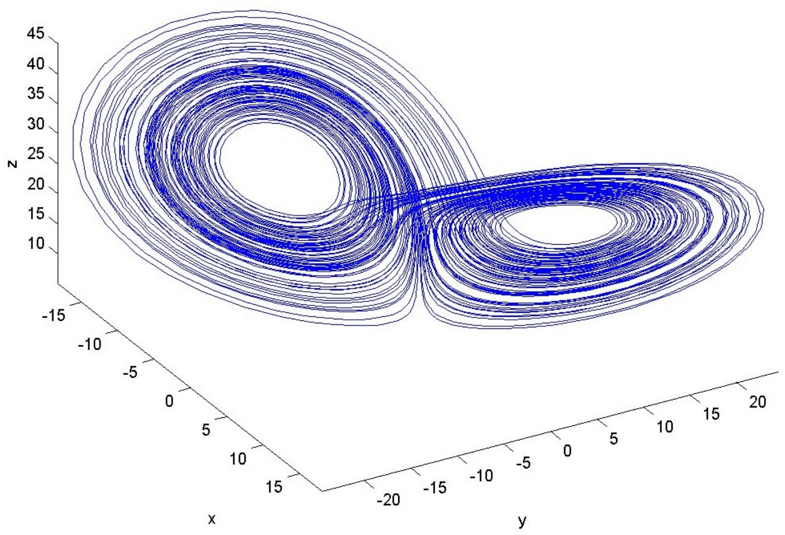
Physics Tutorial 2: Numerical Integration



Summary

The concept of numerically solving differential equations is explained, and applied to real time gaming simulation. Some objects are moved in a 3D space using numerical integration to resolve simple Newtonian mechanics in an iterative manner.

New Concepts

Integration Revision, Numerical Integration, Euler and Verlet Methods, Force-Based Physics Engine, Damping

Introduction

A physics engine needs to move items around the environment in a believable manner. The first tutorial in this series included a reminder of Newtonian mechanics, and described their relevance to real-time game simulation. This tutorial discusses how that simulation is achieved through implementation of a number of iterative techniques for resolving differential equations numerically.

The physics engine which we are developing is based around resolving the forces acting on a body, and calculating the acceleration instigated by the resultant force at every time-step of the simulation. From this acceleration, the velocity can then be calculated, which in turn allows the calculation of the position of the object for each frame of the game. The first tutorial described how multiple forces on an object are resolved, and showed that the relationship between force and acceleration is proportional according to Newton's second law of motion. In this tutorial we discuss how to translate an acceleration into a position in world space using integration.

We will first remind ourselves of what is meant by integration, and discuss why this is relevant to gaming simulation. We then look at a few techniques for implementing numerical integration.

Integration

As you will recall from school, integration is a major part of calculus, which allows us to mathematically describe the relationship between variables in terms of their rate of change.

Probably the most intuitive way to introduce integration is to talk more specifically about the relationship between acceleration, velocity and displacement – this is especially appropriate in the context of developing a physics engine for games, as these are the parameters in which we are interested. Note the use of the words *displacement* and *velocity* here, rather than distance and speed. This is because we are describing three-dimensional vectors, rather than one-dimensional scalars, as discussed in the previous tutorial.

Velocity v is the rate of change of displacement s over time, and acceleration a is the rate of change of velocity over time. In equation form, this is written as:

$$v = \frac{ds}{dt}$$

$$a = \frac{dv}{dt}$$

Conversely, velocity is the integral of acceleration over time, and displacement is the integral of velocity over time:

$$v = \int a\,dt$$

$$s = \int v\,dt$$

The graphs in Figure 1 show the relationship between displacement, velocity and acceleration for a body which accelerates smoothly from rest, then moves at a constant velocity, then decelerates smoothly back to being stationary.

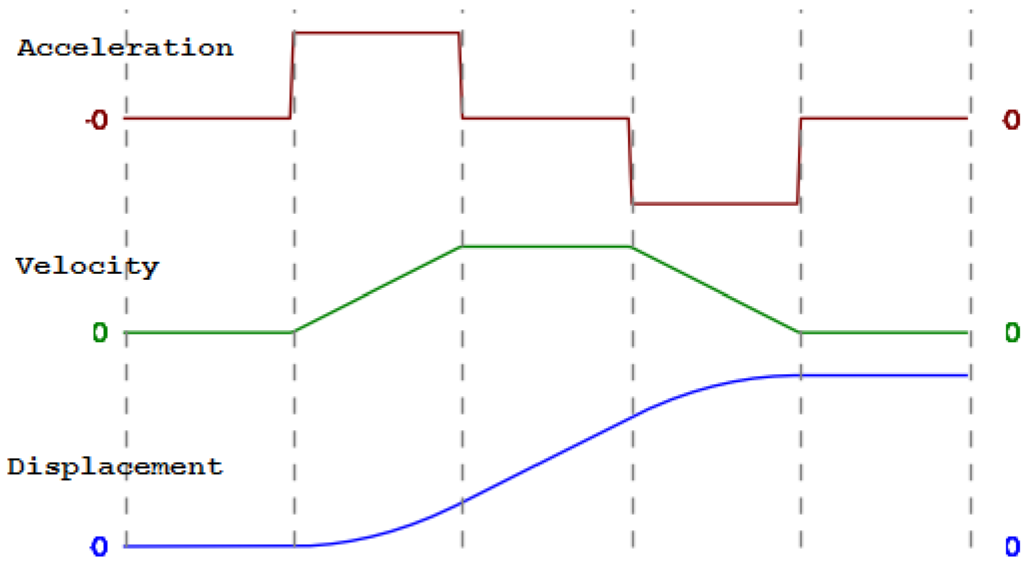


Figure 1: Relationship between acceleration, velocity and displacement

The first graph shows the acceleration, which is a series of discrete values – for the first segment there is no acceleration, then during the second there is a constant acceleration, during the third there is no acceleration, then the fourth has constant negative acceleration (i.e. deceleration), and the fifth again has zero acceleration. The second graph shows how this is translated into the velocity of the object – during the first segment with zero acceleration the velocity remains at zero, in the second portion with constant acceleration the velocity increases linearly, the third segment has zero acceleration so the velocity remains constant, in the fourth the velocity decreases linearly due to a constant deceleration, and the fifth segment maintains a constant velocity as acceleration is zero. The third trace shows how the results of the velocity on the displacement of the object (i.e. how far it has travelled in total) - again it is easy to see how the velocity graph contributes to the displacement, with the object getting further away while there is a positive velocity then remaining at the same displacement when the velocity returns to zero.

An informal definition of integration is that it measures the area under the trace of the value being integrated. Taking another look at the three traces in the diagram should make this more clear. The second trace (velocity) shows the integral of the first trace (acceleration); you should be able to see that it is a measure of the total area between the trace and the zero line – e.g. during the second segment when the acceleration is a constant positive value the velocity trace rises linearly as the area under the acceleration line increases linearly over time.

Force-Based Physics Engine

Okay, so now we have our mathematical model for calculating the positions of our simulated objects from the forces acting upon them:

- Resolve all forces on an object into a single force
- Calculate the acceleration caused by that force from Newton's second law $F = ma$.
- Integrate the acceleration over time to calculate the velocity.
- Integrate the velocity over time to calculate the position.

The next question to be addressed is how to integrate a function using a computer program. Integration is a continuous process, but a computer program can only work with discrete pieces of data. In fact the solution can be found in the informal description of integration as a running total of the area beneath the graph described earlier, as we shall see in the next section.

Numerical Integration

In this section we will discuss a number of increasingly complex, and increasingly accurate, methods of integrating a time series of data using numerical integration. They are all based on the idea that each iteration of the time series can be treated as an extra section of the area under the trace.

Time Series and the Time Step

Before discussing the specific integration methods, we must talk about how time is represented in our simulated system. As we know, computers deal with discrete numbers and states; if the simulation runs sufficiently quickly then we see these states as continuous movement. For each frame of the simulation, we need to calculate the state of each object (i.e. the velocity and position), based on the state of the object in the previous frame.

The record of a particular state of an object (for example the x coordinate of the object's position) is known as a *time series*. The trace of a graph where the x axis represents time (such as the three graphs in Figure 1) is a time series. The amount of simulated time that has elapsed between each step of the simulation is known as the *time step*. Note the use of the term *simulated time* – the time step does not have to be the same as the amount of time it takes to calculate each frame of the simulation; these numerical integration techniques are used in all areas of computer simulation, not just in real-time games.

The basic technique is best illustrated by considering a body travelling with constant velocity v as shown in Figure 2. The displacement s of this body is calculated at each time step, by numerically integrating the velocity. The displacement at the end of the current time step s_{n+1} is calculated by taking the displacement at the start of the time step s_n, and adding on the extra displacement achieved by the current velocity v_n.

$$s_{n+1} = s_n + v_n \Delta t$$

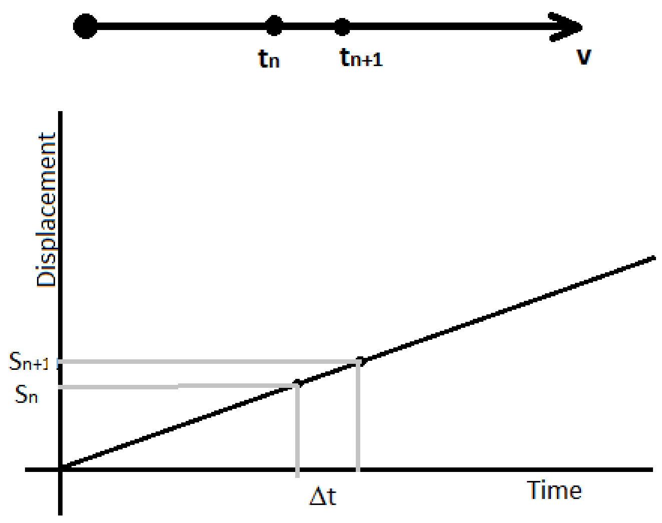


Figure 2: Timestep integration of constant velocity

Hence the displacement at each time step is calculated from the previous displacement and previous velocity. Similarly, the displacement at the next time step will be calculated from the current displacement and current velocity, and so on. This is therefore an *iterative* process.

In the stated example, the calculation of the displacement will be accurate if the velocity is a constant. However we are interested in simulating systems where the velocities are not constant. This is where numerical integration gets interesting. Because numerical integration is an attempt to represent a continuous process in a discrete manner, approximations need to be made, which means that the results become inaccurate. As it is an iterative process the inaccuracies are likely to get larger over time (as each result is calculated from the previously inaccurate result, and introduces its own inaccuracy). In the example given, when the velocity of the body is not constant, the assumption that the velocity remains constant throughout a specific time-step is no longer 100% accurate. Of course, increasing the number of samples (i.e. reducing the time step) should improve the accuracy. This is why a physics engine typically runs at a much faster frame-rate than the rest of the game.

We will now describe some algorithms commonly used for numerical integration, and discuss their appropriateness to games development. The algorithms are:

- Explicit Euler Integration
- Implicit Euler Integration
- Semi-Implicit Euler Integration (or Symplectic Euler Integration)
- Verlet Integration

Explicit Euler Integration

Explicit Euler Integration (often referred to as simply Euler Integration) is the direct implementation of the method already discussed. Explicit Euler Integration uses the first derivative and evaluates it at the current time.

The following equations are used to calculate the position of a body from its acceleration for each frame of the simulation:

$$v_{n+1} = v_n + a_n \Delta t$$

$$s_{n+1} = s_n + v_n \Delta t$$

Remember that we know the acceleration, as we have calculated it by resolving all the forces on the body, and dividing the resultant force by the mass of the body, according to Newton's second law.

In C++ these equations are encoded simply as:

```
  NextVelocity = ThisVelocity + ThisAcceleration * dt;
  NextPosition = ThisPosition + ThisVelocity * dt;
```

Explicit Euler

Explicit Euler Integration is the simplest and most intuitive numerical integration technique, but it also exhibits a tendency toward instability unless very short time steps are used. Consequently it tends not to be used in physics engines for games as the computational savings gained from the simplicity of each iteration, are outweighed by the increased frequency of the iterations that are required for accuracy.

Implicit Euler Integration

Implicit Euler Integration (also referred to as Backward Euler Integration) addresses the approximation issues inherent in Explicit Euler Integration by using a future state of the system. Implicit Euler Integration therefore uses the first derivative and evaluates it at the next time step.

The following equations are used to calculate the position of a body from its acceleration for each frame of the simulation:

$$v_{n+1} = v_n + a_{n+1}\Delta t$$

$$s_{n+1} = s_n + v_{n+1}\Delta t$$

Of course these equations rely on knowing the future value of the acceleration (i.e. a_{n+1}). While there are methods for approximating, or predicting, the future value of a time series, these are in general far too expensive to be appropriate for use in a physics engine for games. Remember that the physics engine is likely to be updating many many objects per iteration, so any additional expense in the calculations is repeated per object, multiplying the overall cost many times. Consequently the Implicit Euler Integration method tends not to be utilised in game simulation, and we will not consider it further.

Semi-Implicit Euler Integration (or Symplectic Euler Integration)

Semi-Implicit Euler Integration combines the ease of calculation of the Explicit approach with some of the increased accuracy of the Implicit approach. It is also often referred to as Symplectic Euler Integration. Semi-Implicit Euler Integration uses the first derivative and evaluates the acceleration at the current time step, and the velocity at the next time step.

The following equations are used to calculate the position of a body from its acceleration for each frame of the simulation:

$$v_{n+1} = v_n + a_n\Delta t$$

$$s_{n+1} = s_n + v_{n+1}\Delta t$$

In this case we know the acceleration (it is calculated from resolving the forces acting on the body, and dividing by the mass), and we use that to calculate the velocity at the subsequent time step. We then use that ”future” velocity to calculate the position at the next time step. This has the effect of introducing a stabilising factor, so the iterations are a lot less likely to become too inaccurate.

In C++ these equations are encoded as follows (note that the order of these two lines of code is very important, as the velocity must be calculated before it is used to calculate the position):

```
  NextVelocity = ThisVelocity + ThisAcceleration * dt;
  NextPosition = ThisPosition + NextVelocity * dt;
```

Semi–Implicit Euler

As Semi-Implicit Euler Integration is very fast to compute, and tends to retain accuracy over many iterations, this is a very common choice of algorithm for physics engines in games technology. This is the algorithm which we will utilise at the heart of the physics engine that we are developing in this tutorial series.

Verlet Integration

Verlet integration does not calculate the velocity directly, but bases its calculations on the last two positions of the body, and the acceleration. It therefore uses the second derivative and assesses it at the current time step.

If we take the equations given for Semi-Implicit Euler Integration above, and substitute for $v_{n=1}$ in the equation for s_{n+1}, then we get:

$$s_{n+1} = s_n + v_n \Delta t + a_n \Delta t^2$$

The velocity is calculated by subtracting the previous position from the current position, and dividing by the time step:

$$v_n = \frac{s_n - s_{n-1}}{\Delta t}$$

which gives us the equation used by Verlet Integration to calculate the position of a body from its acceleration for each frame of the simulation:

$$s_{n+1} = s_n + (s_n - s_{n-1}) + a_n \Delta t^2$$

In C++ these equations are encoded as:

```
 NextPosition = ThisPosition + ThisPosition - LastPosition +
    ThisAcceleration * dt * dt;
 LastPosition = ThisPosition;
```

Verlet

Note that in practice the multiplication of Δt with itself would not be calculated for every object. It would be calculated once per frame and the result (i.e. Δt^2) would be multiplied by the acceleration within the update loop, as this is more efficient.

Verlet integration is similarly efficient to compute as Semi-Implicit Euler Integration. It also has the advantage of being reversible, whereas the Euler approaches are not (this can be especially useful for games involving replays). Further to this, when we come to consider collision responses later in this tutorial series, we will see that there are distinct advantages to using a Verlet integration method, as we can set the post-collision position directly rather than having to worry about the velocity. One thing to bear in mind when using Verlet Integration is that it is not self-starting (i.e it needs the state of the simulated objects from one time step in the past), so care must be taken when setting the initial conditions of simulated objects.

More Advanced Numerical Integration Methods

Whereas the methods described so far in this section are most suited to game simulation, the applications of numerical integration are numerous and widespread throughout mathematics, engineering and science. In applications where performing in real-time is less important, or where fewer agents are to be simulated at much higher accuracies, there are more complex iterative methods available. These include the Runge-Kutta method, which divides the time step into a number of sections, and estimates each sub-time-step in order to arrive at a better estimate of the state at the end of the time step. Typically a fourth order Runge Kutta algorithm is used (referred to as *RK4*), which splits each time step into four subsections. Euler integration is actually a low order Runge Kutta implementation (i.e. with a single sub-step).

Numerically integrating a set of differential equations is an important part of many areas of scientific research. Most differential equations can't actually be solved mathematically, however they can be simulated by using one of these iterative methods to solve them numerically. The time series produced by the numerical integration can then be analysed further. For those interested, the three-dimensional time series shown in the figure at the start of this tutorial is generated by numerically integrating Lorenz' equations, and is known as a *Lorenz attractor*.

Damping and Rest

Before moving on to the implementation of the numerical integrator of your force-based physics engine, there is one more aspect to discuss. In the real world, objects gradually slow down due to friction, unless they are in a vacuum. Air resistance and friction from any contact surfaces provide a force acting against the direction of travel of an object. Whereas we could simulate these forces fully, and include them when resolving the rest of the forces acting on a body, it is more common (and more efficient) to cheat somewhat.

The observable effect of these frictional forces is to gradually slow down all moving bodies until they come to a rest. This is usually achieved in a physics engine simply be reducing the velocity by a constant factor every time step. When the velocity becomes less than a specified threshold, it is set explicitly to zero, which will have the effect of stopping an object jiggling around unrealistically.

In C++ this is encoded as:

```cpp
ThisVelocity *= DAMPING_FACTOR;
if (ThisVelocity < MINIMUM_VELOCITY) ThisVelocity = 0.0f;
```

Damping

Changing the value of the damping factor causes objects to come to a rest more or less quickly. This can be utilised to simulate such effects as sliding around on ice, or moving underwater. Consequently the damping factor may not be a constant – it may vary depending on the area of the game world, or the type of game object, that is being simulated. Changing the value of the minimum velocity before setting the object to rest is less common, and not really advisable – making the value too high will result in an abrupt looking halt, and setting it too low will just waste computation time on imperceptible velocities or contribute to some unwanted shuddering.

Implementation

The aim of this practical session is to implement the heart of your physics engine, in the form of a numerical integrator. We will construct a set of algorithms which resolve the forces acting on a body, calculate the acceleration caused by that resultant force, and then iteratively solve the differential equations governing the motion of the body, in order to calculate its position and velocity at each frame of the simulation. To demonstrate that the physics update is working, we will launch objects from the camera position by applying an initial force to them, and simulating their trajectory according to Newton's laws.

Tutorial Summary

We have expanded on the concept of a physics engine,and discussed how forces are used to calculate the motion of bodies. We have discussed how to implement a numerical integration scheme, and how to use it to calculate the velocity and displacement of an object from its acceleration. We have implemented this physics simulation in an efficient and well-engineered fashion so that it can be used by multiple game agents as we progress through the tutorial series. So far we have only considered linear motion; in the next tutorial we will expand our simulation further to cover angular motion.