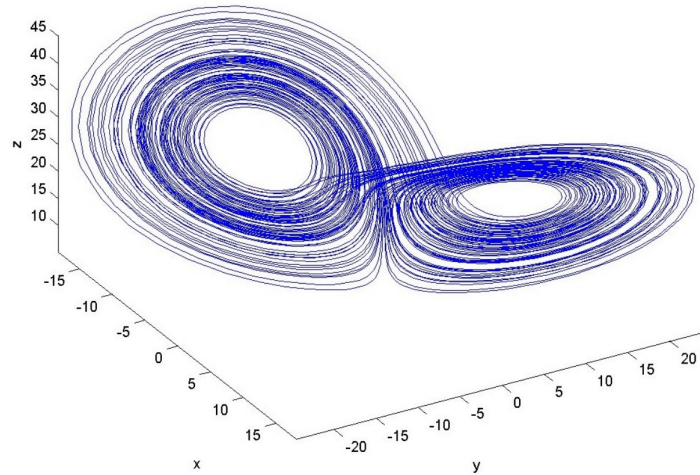


# Physics Tutorial 2: Numerical Integration Methods



## 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 for Linear Motion, Integration for Angular Motion, Force-Based Physics Simulation, Numerical Integration through the Euler, Verlet and RK2 Midpoint Methods

## 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 may recall from school or undergraduate study, integration is a major part of calculus, which allows us to mathematically describe the relationship between variables in terms of their rate of change. In this section, we'll first present integration in the context of linear motion, before showing how these principles extend into angular motion.

## Linear Motion

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.

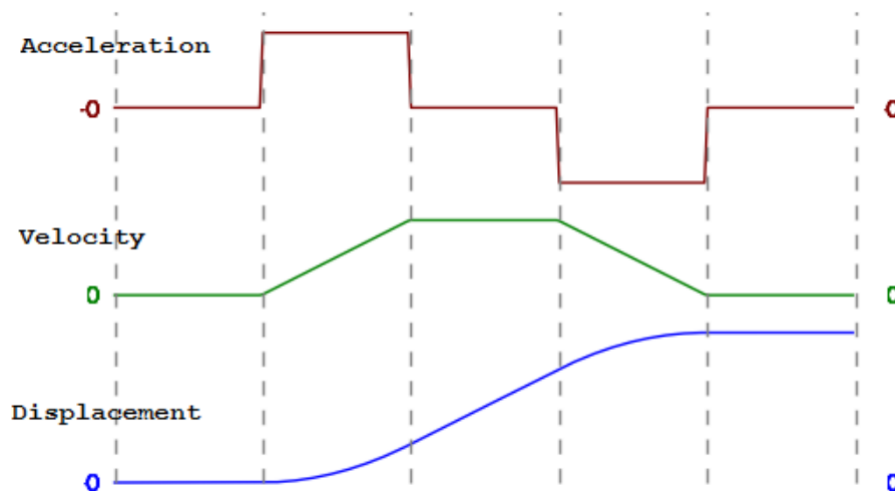


Figure 1: Relationship between acceleration, velocity and displacement

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.

## Angular Motion

We have discussed in an earlier lecture how to calculate the angular acceleration of a rigid body caused by forces acting on that body. The next step is to translate that acceleration into an angular velocity and orientation, in order to spin and rotate our objects in the game world. Our approach here still has its roots in the mathematics discussed above.

Remembering the analogues between angular and linear motion, we can integrate angular velocity  $\omega$  with respect to the angular acceleration  $\alpha$  as follows:

$$\omega_{n+1} = \omega_n + \alpha_n \Delta t$$

However, orientation ( $\Theta$ ) needs special care in order to ensure behaviour is consistent across all three axes. To that end, our equation connecting  $\Theta_n$  to  $\Theta_{n+1}$ , which is a fast approximation, can be defined:

$$\Theta_{n+1} = \Theta_n + \Theta_n \omega_n \frac{\Delta t}{2}$$

$\Theta_{n+1}$  must then be *normalised* to remove errors introduced by this approximation. The reason this approximation has that condition is we bypass the computation which explicitly connects  $\theta$  to  $\Theta$ .

The origination of these discrete equations is discussed below.

## 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.

```
1 for (PhysicsNode* obj : physicsNodes)
2   obj->IntegrateForVelocity(updateTimestep);
```

PhysicsEngine::UpdatePhysics

- Resolve Constraints.
- Integrate the velocity over time to calculate the position.

```
3 for (PhysicsNode* obj : physicsNodes)
4   obj->IntegrateForPosition(updateTimestep);
```

PhysicsEngine::UpdatePhysics

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.

## A Note on Inverse Mass

Efficiency is at the heart of real-time physics simulation. As a result, we tend to try and avoid the costlier floating point operations. Computing our acceleration requires us to divide the applied force by the mass of the object; division is more computationally expensive than multiplication and, as a result, we often store the inverse-mass of our object as an element of its physical data. This is basically  $1/m$  for mass  $m$ , allowing us to multiply by inverse mass to obtain acceleration.

## 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.

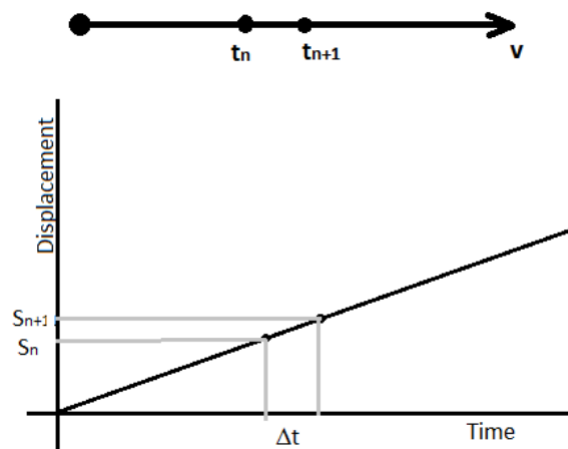


Figure 2: Timestep integration of constant velocity

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. Rather, it is common (and recommended) practise that the time step be a fixed value (e.g., 8ms or 120fps).

It should be noted, too, that these numerical integration techniques are used in all areas of computer simulation, not just in real-time games. And, even within games, not just in real-time physics simulation. Consider the example of a city or civilisation simulator where population increases over time (be that in 'real-time', or turn-based). This means that there may be a variable  $x$  representing

the population of a city at any given time step, and a variable  $y$ , where  $y = \frac{dx}{dt}$ , which is the rate at which population changes. Perhaps the rate of population change is connected to the presence of local wildlife, which connects  $y$  to another function - e.g., a lack of available animals to hunt makes  $y$  negative for a turn, and the availability of animals is related back to just how many people are hunting them ( $x$ ).

You can (hopefully!) see how this interconnection is both complex, and connected to the very principles we're talking about. It's also a fundamental aspect of all gaming - real-time or otherwise - because if nothing changes over time, why on earth are you playing?

The basic techniques are best illustrated with a physical example, however wide-ranging their applicability. Consider 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  $v_n$ .

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

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.

- Explicit Euler Integration
- Implicit Euler Integration
- Semi-Implicit Euler Integration (or Symplectic Euler Integration)
- Second Order Approaches (e.g., 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:

```
1 NextVelocity = ThisVelocity + ThisAcceleration * dt;  
2 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:

$$\begin{aligned}v_{n+1} &= v_n + a_{n+1}\Delta t \\s_{n+1} &= s_n + v_{n+1}\Delta t\end{aligned}$$

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:

$$\begin{aligned}v_{n+1} &= v_n + a_n\Delta t \\s_{n+1} &= s_n + v_{n+1}\Delta t\end{aligned}$$

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. Also note that in our framework's engine structure, we update position after constraints have been resolved - this has significant benefits in terms of accuracy:

```
1 NextVelocity = ThisVelocity + ThisAcceleration * dt;  
2 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.

## Second Order Approaches

### 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. It should be noted that the Verlet approach does not mesh well with the provided framework code, and requires extensive rewrites to integrate - pun intended.

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:

```
1 NextPosition = ThisPosition + ThisPosition - LastPosition +
2   ThisAcceleration * dt * dt;
3 LastPosition = ThisPosition;
```

Verlet

Note that in practice the multiplication of  $\Delta t$  with itself would not be calculated for every object. It would be calculated once per frame and the result (i.e.  $\Delta t^2$ ) 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.

### Runge-Kutta "Midpoint" Method

A second-order approach which more easily slots into the framework is provided through the Runge-Kutta methods. The Runge-Kutta methods are actually a collection of implicit and explicit iterative methods for resolving calculus. We have already considered one Runge-Kutta implementation - Euler integration is essentially a first-order Runge-Kutta approach.

In the Euler approach, we consider the variable and its derivative at time  $t$ , and time  $t + 1$  - e.g.,  $s_t$  and  $v_t$  are used to determine  $s_{t+1}$ . We know that an object acting under changing forces might experience a change to its acceleration value over the course of the time-step - we just accept that inaccuracy as part of the price for a fast simulation. The same is true of changing velocity under constant acceleration - we know that velocity halfway through a time-step will have changed, we just overlook it. The premise of RK2 is that the derivative at the midpoint of the time-step is a more accurate representation than the derivative at the beginning (Explicit Euler) or the derivative at the end (Semi-Implicit Euler).

Under this scheme, you need to predict the derivative (e.g., velocity at constant acceleration) halfway through the timestep, and then apply that in computation of the change in position, e.g.  $s_{n+1} = s_n + v_{n+0.5} \Delta t$ .

## 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 higher orders of the Runge-Kutta method, which divide the time step into an even larger number of sections. Typically a fourth order Runge Kutta algorithm is used (referred to as *RK4*), which splits each time step into four subsections.

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*.

## Implementation

Consider the first couple of tasks for today from the Practical Tasks handout. Think about them during your free time today, so you have some ideas of how to go about implementing them before the practical session begins. Today covers a lot of material, so make good use of your time.

## Tutorial Summary

We have reviewed basic calculus and its applicability to linear and angular motion, 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, which can be extended towards the angular case.

```
1 void PhysicsNode::IntegrateForVelocity(float dt)
2 {
3 //Apply Gravity
4 // Technically gravity here is calculated by formula:
5 // ( gravity / invMass * invMass * dt )
6 // So even though the divide and multiply cancel out, we still
7 // need to handle the possibility of divide by zero.
8
9     if (invMass > 0.0f)
10         linVelocity += PhysicsEngine::Instance()->GetGravity() * dt;
11
12
13 //Semi-Implicit Euler Intergration
14 // - See "Update Position" below
15     linVelocity += force * invMass * dt;
16
17
18 //Apply Velocity Damping
19 // - This removes a tiny bit of energy from the simulation each update
20 // to stop slight calculation errors accumulating and adding force
21 // from nowhere.
22 // - In it's present form this can be seen as a rough approximation
23 // of air resistance, albeit (wrongly?) making the assumption that
24 // all objects have the same surface area.
25     linVelocity =
26         linVelocity * PhysicsEngine::Instance()->GetDampingFactor();
27
28
```



```

29 //Angular Rotation
30 // - These are the exact same calculations as the three lines above,
31 // except for rotations rather than positions.
32 //   - Mass      -> Torque
33 //   - Velocity  -> Rotational Velocity
34 //   - Position  -> Orientation
35   angVelocity += invInertia * torque * dt;
36
37
38 //Apply Velocity Damping
39   angVelocity =
40     angVelocity * PhysicsEngine::Instance()->GetDampingFactor();
41 }
42
43 void PhysicsNode::IntegrateForPosition(float dt)
44 {
45 //Update Position
46 // - Euler integration; works on the assumption that linearvelocity
47 // does not change over time (or changes so slightly it doesnt make
48 // a difference).
49 // - In this scenario, gravity /will/ be increasing velocity over
50 // time. The in-accuracy of not taking into account of these changes
51 // over time can be visibly seen in tutorial 1.. and thus how better
52 // integration schemes lead to better approximations by taking into
53 // account of curvature.
54   position += linVelocity * dt;
55
56
57 //Update Orientation
58 // - This is slightly different calculation due to the wierdness
59 // of quaternions. It does the same thing as position update
60 // (with a slight error) but from i've seen, is generally the best
61 // way to update orientation.
62   orientation = orientation +
63     Quaternion(angVelocity * dt * 0.5f, 0.0f) * orientation;
64
65 //invInertia = invInertia *
66 // (Quaternion(angVelocity * dt * 0.5f, 0.0f)
67 // * orientation).ToMatrix3());
68 // As the above formulation has slight approximation error, we need
69 // to normalize our orientation here to stop them accumulating
70 // over time.
71   orientation.Normalise();
72 }

```

PhysicsNode.cpp