# Numerical Methods for ODEs

D. Stott Parker  Luis Ángel Larios Cárdenas

December 5, 2014

## 1 Numerical Approximation: Euler's Method

It is the exception rather than the rule when a differential equation of the general form

$$\frac{\mathrm{d}y}{\mathrm{d}x} = f(x, y)$$

can be solved exactly and explicitly by elementary methods. For example, consider the simple equation

$$\frac{\mathrm{d}y}{\mathrm{d}x} = e^{-x^2} \tag{1}$$

A solution for (1) is simply an antiderivative of $e^{-x^2}$. But it is known that every integral of $f(x) = e^{-x^2}$ is a **nonelementary** function.

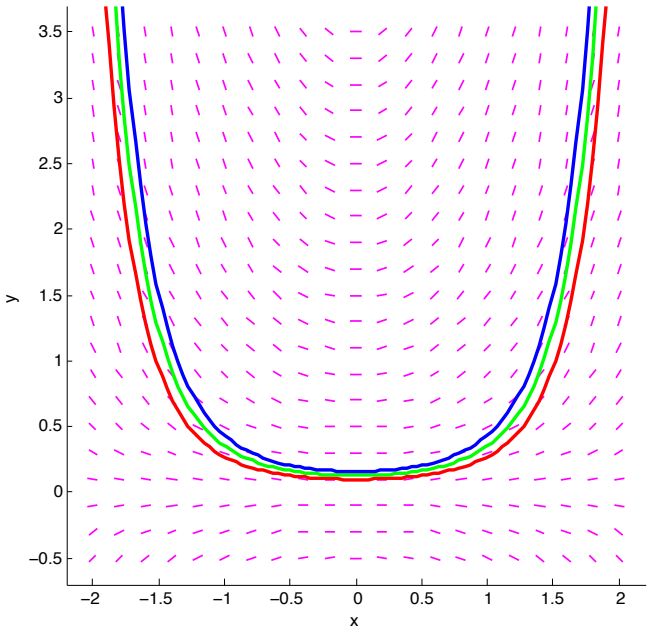


Figure 1: Slope field for the ODE $\frac{\mathrm{d}y}{\mathrm{d}x} = xy$ illustrating three solution courves for the initial value problems $y(0) = 0.1$, $y(0) = 0.13$, and $y(0) = 0.16$, in red, green, and blue respectively

As a possible alternative, we can write a computer program to draw a solution curve that starts at the initial point $(x_0, y_0)$ and attempts to thread its way through the *slope field* (figure 1) of a given differential equeation $y' = f(x, y)$. The procedure we would carry out can be described as follows:

- We start at the initial point $(x_0, y_0)$ and move a tiny distance along the slope segment through $(x_0, y_0)$. This takes us to the point $(x_1, y_1)$.
- At $(x_1, y_1)$ we change direction, and now move a tiny distance along the slope segment through this new starting point $(x_1, y_1)$. This takes us to the next starting point $(x_2, y_2)$.

- At $(x_2, y_2)$ we change direction again, and now move a tiny distance along the slope segment through $(x_2, y_2)$. This takes us to the next starting point $(x_3, y_3)$.

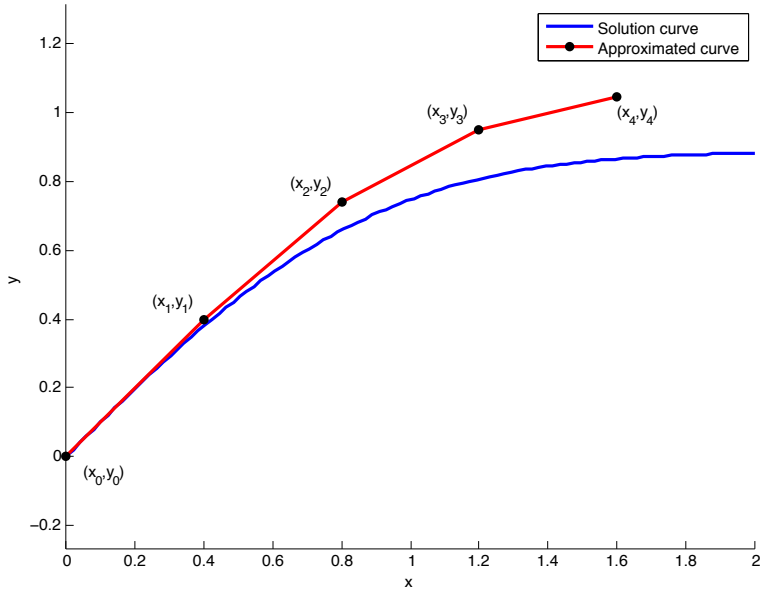


Figure 2: The first few steps in approximating the solution curve

Figure 2 illustrates the result of continuing in this fashion. In this figure, we see a polygonal curve consisting of line segments that connect the successive points. However, suppose that each "tiny distance" we travel along a slope segment is so very small that the naked eye cannot distinguish the invidivual line segments constituting the polygonal curve. Then, the resulting polygonal curve looks like a smooth, continuously turning solution curve of the differential equation.

*Leonhard Euler* did not have a computer, and his idea was to do all this numerically rather than graphically. In order to approximate the solution of the initial value problem

$$\frac{\mathrm{d}y}{\mathrm{d}x} = f(x, y), \qquad y(x_0) = y_0 \tag{2}$$

we first choose a fixed (horizontal) **step size $h$** to use in making each step from one point to the next. Then, the step from $(x_k, y_k)$ to $(x_{k+1}, y_{k+1})$ is illustrated in figure 3. The slope of the direction segment through $(x_k, y_k)$ is $m = f(x_k, y_k)$. Hence, a horizontal change of $h$ from $x_k$ to $x_{k+1}$ corresponds to a vertical change of $m \cdot h = h \cdot f(x_k, y_k)$ from $y_k$ to $y_{k+1}$. Therefore, the coordinates of the new point $(x_{k+1}, y_{k+1})$ are given in terms of the old coordinates by

$$x_{k+1} = x_k + h, \qquad y_{k+1} = y_k + h \cdot f(x_k, y_k)$$

Given the *initial value problem* in (2), **Euler's method** with step size $h$ consists of applying the formulas

$$\begin{array}{ll} x_1 = x_0 + h & y_1 = y_0 + h \cdot f(x_0, y_0) \\ x_2 = x_1 + h & y_2 = y_1 + h \cdot f(x_1, y_1) \\ x_3 = x_2 + h & y_3 = y_2 + h \cdot f(x_2, y_2) \\ \vdots & \vdots \\ \vdots & \vdots \end{array}$$

to calculate successive points on an approximate solution curve. The numerical result of applying Euler's method is the sequence of *approximations*

$$y_1,\ y_2,\ y_3,\ \ldots,\ y_k,\ \ldots$$

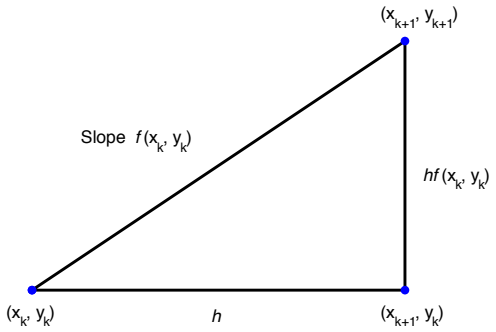


Figure 3: The step from $(x_k, y_k)$ to $(x_{k+1}, y_{k+1})$

to the true values

$$y(x_1),\ y(x_2),\ y(x_3),\ \ldots,\ y(x_k),\ \ldots$$

which are the *exact* (though unknown) solution $y(x)$ of the initial value problem.

## 1.1 The Euler's Method

Given the initial value problem in (2), **Euler's method with step size $h$** consists of applying the iterative formula

$$y_{k+1} = y_k + h \cdot f(x_k, y_k), \qquad (k \leqslant 0) \tag{3}$$

to calculate successive approximations $y_1,\ y_2,\ y_3,\ \ldots$ to the [true] values $y(x_1),\ y(x_2),\ y(x_3),\ \ldots$ of the [exact] solution $y = y(x)$ at the points $x_1,\ x_2,\ x_3,\ \ldots$, respectively. Euler's method can be implemented in Matlab in a functional manner as follows:

```
% Function to compute Forward Euler integration.
% Input arguments.
%   x -- initial value for x.
%   y -- initial value for y.
%   f -- anonymous function of the form f(x,y) that represents derivative.
%   h -- time step.
%   evalNum -- number of evaluations (including initial point).
function [X, Y] = ForwardEuler( x, y, f, h, evalNum )
X = zeros( evalNum, 1 );
Y = zeros( evalNum, 1 );
X(1) = x;
Y(1) = y;

for I = 2:evalNum
   y = y + h * f(x,y);
   x = x + h;
   X(I) = x;              % Store approximations.
   Y(I) = y;
end;
```

## Example

Apply the Euler's method to approximate the solution of the initial value problem

$$\frac{\mathrm{d}y}{\mathrm{d}x} = x + \frac{1}{5}y, \qquad y(0) = -3 \tag{4}$$

(a) first with step size $h = 1$ on the interval $[0,\ 5]$.

(b) then with step size $h = 0.2$ on the interval $[0,\ 1]$.

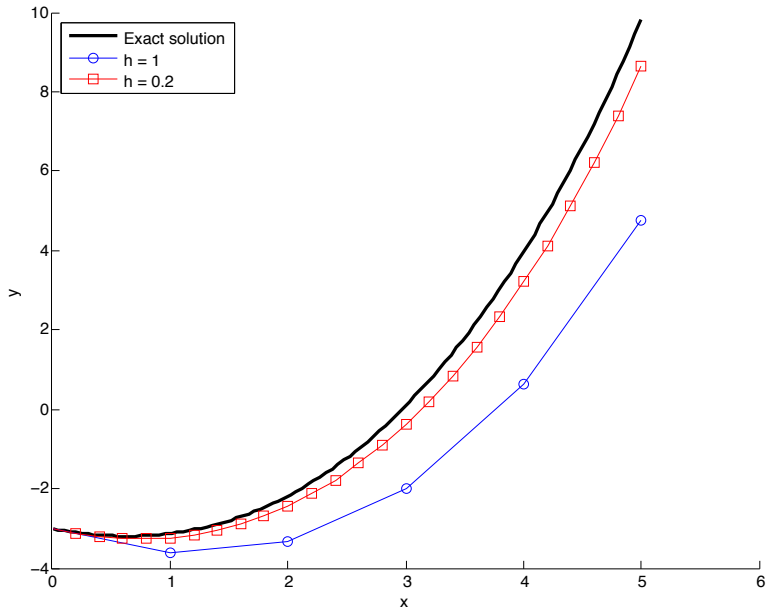


Figure 4: Graphs of Euler approximations with step sizes $h = 1$ and $h = 0.02$

**Solution**

**(a)** With $x_0 = 0$, $y_0 = -3$, $f(x, y) = x + \frac{1}{5}y$, and $h = 1$, the iterative formula in (3) yields the values:

| $x$ | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| *Approx.* $y$ | -3 | -3.6 | -3.32 | -1.984 | 0.6912 | 4.7430 |

Figure 4 shows the graph of this approximation, together with the graphs of the Euler aproximation obtained with step size $h = 0.2$, as well as the graph of the exact solution

$$y(x) = 22e^{x/5} - 5x - 25$$

We can see that **decreasing the step size increases the accuracy**, but with any single approximation, the accuracy decreases with distance from the initial point.

**(b)** Starting afresh with $x_0 = 0$, $y_0 = -3$, $f(x, y) = x + \frac{1}{5}y$, and $h = 0.2$, we ge the approximate values

| $x$ | 0 | 0.2 | 0.4 | 0.6 | 0.8 | 1 |
|---|---|---|---|---|---|---|
| *Approx.* $y$ | -3 | -3.12 | -3.205 | -3.253 | -3.263 | -3.234 |

High accuracy with Euler's method usually requires a *very small step size and hence a larger number of steps* than can reasonably be carried out by hand — it's always better to use any software package or programming language to implement Euler's method, like the one we presented in previous paragraphs.

## 1.2 Local and Cumulative Errors

There are several sources of error in Euler's method that may make the approximation $y_k$ to $y(x_k)$ unreliable for large values of $k$; those for which $x_k$ is not sufficiently close to $x_0$. The error in the linear approximation formula

$$y(x_{k+1}) \approx y_k + h \cdot f(x_k, y_k) = y_{n+1} \tag{5}$$

is the amount by which the tangent line at $(x_k, y_k)$ departs from the solution curve through $(x_k, y_k)$ as illustrated in figure 5. This error, introduced at each step in the process, is called **local error**.

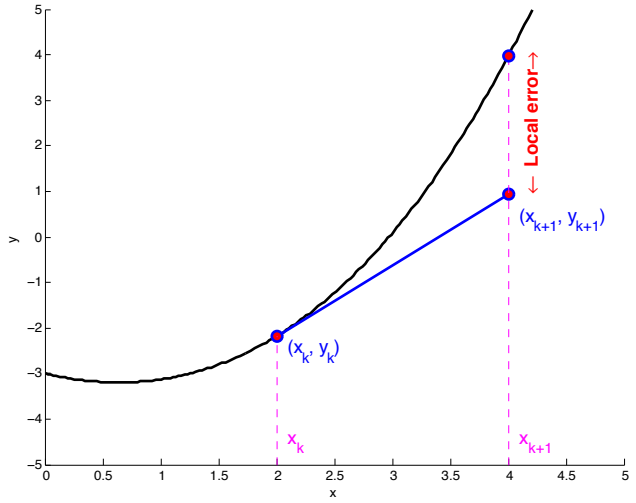


Figure 5: The local error in Euler's method

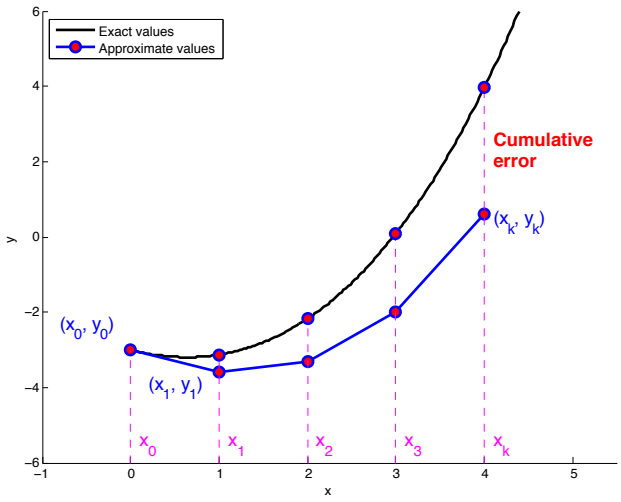


Figure 6: The cumulative error in Eurler's method

The local error indicated in figure 5 *would be* the total error in $y_{k+1}$ *if* the starting point $y_k$ were an exact value, rather than merely an approximation to the actual value $y(x_k)$. But $y_k$ itself suffers from the accumulated effects of all the local error introduced at the previous steps. Thus the tangent line in figure 5 is tangent to the "wrong" solution curve — the one through $(x_k, y_k)$ rather than the actual solution curve through the initial point $(x_0, y_0)$. Figure 6 illustrates this **cumulative error** in Euler's method; it is the amount by which the polygonal stepwise path from $(x_0, y_0)$ departs from the actual solution curve through $(x_0, y_0)$.

As we saw in figure 4, the usual way to reduce the cumulative error in Euler's method is to decrease the step size $h$. Thus, the smaller the step size, the more slowly does the error grow with increasing distance from the starting point.

Why, then, do we not simply choose an exceedingly small step size, with the expectation that very great accuracy will result? There are two reasons for not doing so. The first is obvious: the time required for the computation. The second reason is more subtle. In addition to local and cumulative errors, the computer itself will contribute **roundoff error** at each stage because only finitely many significant digits can be used in each calculation.

The "best" choice of $h$ is difficult to determine in practice as well as in theory. It depends on the nature of the function $f(x, y)$ in the initial value problem in (2), on the exact code in which the program is written, and on the specific computer that is used.

### The Error in the Euler's Method

The Euler's method has the advantage of simplicity, and a careful study of this method yields insights into the workings of more accurate frameworks (such as the Runge-Kutta Method), because many of the latter are extensions or refinements of the Euler's.

Suppose that the initial value problem in (2) has a unique solution $y(x)$ on the closed interval $[a, b]$ with $a = x_0$, and assume that $y(x)$ has a continuous second derivative on $[a, b]$. Then, there exists a constant $C$ such that the following is true:

*If the approximations $y_1, y_2, y_3, \ldots, y_n$ to the actual values $y(x_1), y(x_2), y(x_3), \ldots, y(x_n)$ at points of $[a, b]$ are computed using the Euler's method with step size $h > 0$, then*

$$|y_k - y(x_k)| \leqslant Ch \tag{6}$$

*for each $k = 1, 2, 3, \ldots n$.*

# 2 Case Study: A Particle System

We next consider the simplest scenario where numerical solutions of ODEs (by the Euler's method) take place: **particle systems**. Since particles can thought of infinitesimal and "compact" objects, we usually find them to be suitable models to produce *physics-based animations*.

## 2.1 Basic Physics — A Review

The physics most useful to animators is based on *Newton's Laws of Motion*. The most fundamental equation relates force to the acceleration of a mass

$$\mathbf{f} = m \cdot \mathbf{a} \tag{7}$$

where $\mathbf{f}$ and $\mathbf{a}$ are vectors, usually in $\mathbb{R}^3$, representing force and acceleration, respectively, while $m$ refers to the objects's (scalar) mass.

When setting up a physically based animation, the animator will specify all the forces acting in that environment. In calculating a *frame* of the animation, each object will be considered and all the forces acting on that object will be determined. Once that is done, the object acceleration can be calculated based on its mass and (7), yielding

$$\mathbf{a} = \frac{\mathbf{f}}{m} \tag{8}$$

The purpose of an animation is to compute a *new position* for the object in a series of discretized *time steps*. This new position will result from moving the object a small distance in the direction that its *velocity* vector $\mathbf{v}$ points to. The forces acting on the object will induce a new accelaration (8), which will ultimately change the object's velocity each time. So, then, how can we calculate a new velocity $\mathbf{v}$ from $\mathbf{a}$? Recall that *acceleration is the change of velocity over time*. We can express this relation as follows

$$\mathbf{a} = \frac{\Delta \mathbf{v}}{\Delta t} \tag{9}$$

and, as $\Delta t \to \infty$ (i.e. the change in time is instantaneous), equation (9) becomes

$$\mathbf{a} = \frac{\mathrm{d}\mathbf{v}}{\mathrm{d}t} \tag{10}$$

which is nothing else than an *ordinary differential equation*. However, attempting to solve it analytically is pretty much (really) hard because it depends on a variety of factors — among them, the random forces $\mathbf{f}$ acting on the object every time step. Fortunately, this is where Euler's method comes in handy. If we compare (10) with the initial value problem of (2), we can see that $x = t$, $y = \mathbf{v}$, and $f(x, y) = f(t, \mathbf{v}) = \mathbf{a}$. Thus, choosing and appropriate step size $h = \Delta t$, we can use the Euler's method iteration in (3) to compute a new velocity

$$\mathbf{v}_{new} = \mathbf{v}_{old} + \mathbf{a} \cdot \Delta t \tag{11}$$

Finally, we can update the object's new position from its previous location by using the newly computed velocity and performing yet another *integration step* via the Euler's method

$$\mathbf{p}_{new} = \mathbf{p}_{old} + \mathbf{v}_{new} \cdot \Delta t \tag{12}$$

### May the Force Be with You

There are a variety of forces that we can consider to obtain acceleration from (8). Typically, forces act on the object because of its position and velocity. The simplest force that all *physics-based* animations use is *weight*, due to the Earth's **gravity** field

$$\mathbf{f}_g = m \cdot \mathbf{g}, \qquad \mathbf{g} = \begin{pmatrix} 0 \\ -9.81 \end{pmatrix} \in \mathbb{R}^2 \tag{13}$$

A **spring** is also a common tool for modeling flexible objects or to keep two objects at a prescribed distance (such as a distance between two spheres). When attached to an object, a spring imparts a force depending on the object's location relative to the other end of the spring. The spring force $\mathbf{f}_s$ is proportional to the difference between

Figure 7: Diney's © 2012 Brave's starring character Mérida exhibits her red hair made of strands of particles interconnected with springs

the spring's rest length $\ell_r$ and its current length $\ell_c$. The constant of proportionality $k_s$, also called *spring constant*, determines how much the spring reacts to a change in length (a.k.a. *stiffness*)

$$\mathbf{f}_s = k_s \cdot (\ell_c - \ell_r) \tag{14}$$

A **damper**, like a spring, is also attached to two objects and works against their relative velocity. The damping force of a spring $\mathbf{f}_d$ is negatively proportional to the velocity of the spring length $\mathbf{v}_s$. The damper constant $k_d$ determines how much resistance there is to a change in spring length

$$\mathbf{f}_d = -k_d \cdot \mathbf{v}_s \tag{15}$$

There are many more types of forces that an animator can take into account – each of them adding up to the details of the simulation, such as viscosity, torque, and material stress. However, gravity, springs, and dampers, by themselves, already give us a decent framework to work with particles.

## 2.2 Particle Systems

A particle system is a collection of a large number of point-like elements. They are often animated as a simple physical simulation. Because of the large number of elements typically found in a particle system, simplifying assumptions are used in both rendering and calculation of their motion. Some of the common assumptions are

Particles do not collide with one another.

Particles do not cast shadows, except in an aggregate sense.

Particles do not reflect light — they are modeled as point light sources.

Particles are often modeled as having a finite life span, so that during an animation there may be hundred of thousands of particles used (figure 7), but only thousands active at any one time. *Randomness* is introduced into most of the modeling and processing of particles to avoid excessive orderlines. In computing a frame of motion (at a given time step) for a particle system, we take the following actions:

1. Any new particles that are born during this frame are generated.
2. Each new particle is assigned attributes.
3. Any particles that have exceeded their allocated life span are terminated.
4. The remaining particles are animated and their shading parameters changed according to the controlling processes.
5. The particles are rendered.

The steps are then repeated for each frame of the animation. Figure 8 shows an extraordinary use of thousands of particles in the computer-animated movie industry.

Figure 8: "The arrival of the Rohirrim from the North" — an epic battle in a videogame for the motion picture *The Lord of the Rings: The Return of the King* (WingNut Films © 2003), using thousands of "particle-like" knights to enhance realism

### Particle Attributes

The attributes of a particle determine its motion status, its appearance, and its life in the particle system. Typical attributes are

Position

Velocity

Color

Transparency

Shape parameters

Lifetime

Each of the attributes is initialized when the particle is created. To avoid uniformity, the user typically randomizes values in some controlled way. The position and velocity are updated according to the particle's motion. The shape parameters, color, and transparency control the particle's appearance. The lifetime attribute is a count of how many frames or time steps the particle will exist.

A common (short) representation of a particle in Matlab is a tuple `[p, v, f, m]` which holds its position, velocity, force accumulation, and mass:

```
%% A particle structure (defining the first element of an array of particles)

Particle = [];
Particle.p = zeros(3,1);         % Position 3D vector.
Particle.v = zeros(3,1);         % Velocity 3D vector.
Particle.f = zeros(3,1);         % Force 3D vector.
Particle.m = 0;                  % Scalar mass.
```

### Particle Animation

Usually, each active particle in a system is animated throughout its life span. This activation includes not only its position and velocity, but also its display attributes. To animate the particle's position in space, we take into account forces and compute the resultant particle acceleration. The velocity of the particle is updated from its acceleration, and then its new velocity is used to update the position. As we mentioned above (cfr. section 2.1), gravity, other

global force fields (like wind), local force fields (vortices), and collisions with objects in the environment are typical forces that affect the particle motion.

### Updating the Particle System Status

Using the Euler's method, we can integrate velocity from acceleration (11), and position from velocity (12) after we have found $\mathbf{a}$ in (8) for each particle in our system. If we solve this set of equations every time step $\Delta t$, for a finite number of times within a loop, we can basically "animate" particles obeying the Newton's Laws of Motion!

The following script shows how to conduct, in a general way, a particle system simulation, where the life span is not considered (i.e. particles are "immortal"):

```
%% Particle simulation

% Define time control variables.
deltaT = 0.01;                  % Time step (h step size).
simulationTime = 0.0;           % Accumulate simulation time.
maxSimulationTime = 10.0;       % Threshold to finish simulation.

while( simulationTime < maxSimulationTime )

   % Prepare plotting.
   ...

   % Accumulate forces in all particles.
   for I = 1:particlesCount
     particles(I).f = accumulateForce( particles(I) );   % Internal and external forces.
   end;

   % Find each particle's new position from accumulated force.
   for I = 1:particlesCount

      % Equations of motion.
      a = particles(I).f / particles(I).m;                % Acceleration.
      oldV = particles(I).v;                              % Keep old velocity.
      particles(I).v = particles(I).v + deltaT * a;       % New velocity.
      particles(I).p = particles(I).p + deltaT * oldV;    % New position.

      % Correct new position if collisions are modeled.
      ...

      % Change particle appearance and draw.
      particles(I) = updateAppearance( particles(I) );
      renderParticle( particles(I) );

   end;

   simulationTime = simulationTime + deltaT;               % Advance time.
end;
```

Computer animation and numerical methods are disciplines on their own, and the space here is not enough to study them in detail. One last word, though, reminds us there exist other packages, (e.g. OpenGL), which offer a lot of flexibility and control over visualization/shading. Additionally, languages such as C++, Java, or the newly "convert" Javascript by means of HTML5 and WebGL are incredibly powerful in terms of Computer Graphics. We can stop at this point since the tools we have already covered are decent enough to scratch a little bit the realm of physics-based simulations (e.g. videogames, Artificial Life, virtual reality, etc.). You can now show off your fellows that you are able to understand physics at such a level that you can actually write simulation software that uses it!

# References

[1] Edwards, C. Henry; and Penney, David E. (2008) *Elementary Differential Equations*. Pearson - Prentice Hall, Sixth Edition.

[2] Parent, Rick. (2012) *Computer Animation, Algorithms and Techniques*. Morgan Kaufmann, 3rd Edition