INITIAL VALUE PROBLEM

Comp768: Physically-Based Modeling, Simulation and Animation

Computer Simulation

A computer simulation of a physical system can be considered as the evaluation of the following function:

$$f(\boldsymbol{x_o}, t)$$

This is also known as the *initial value problem*. The parameter $\boldsymbol{x_0}$ is a vector, which describes the state of the system you are simulating at a certain initial time, and t is lapse time from the initial time. The state vector may include many different parameters such as positions, velocities and temperature of particles depending on the system you simulate. And the value of the function is the state vector at time t. f is the mathematical model of the natural laws that govern the system. Because the natural laws should be consistent over time, we get

$$f(\boldsymbol{x_0}, t_a + t_b) = f(f(\boldsymbol{x_0}, t_a), t_b)$$

For a special case in which we can use a fixed time step Δt, we can recursively call a function to get results as:

$$f(\boldsymbol{x_0}, n\Delta t) = \bar{f}(\cdots\bar{f}(\bar{f}(\bar{f}(\boldsymbol{x_0})))\cdots)$$

where $\bar{f}(\boldsymbol{x}) = f(\boldsymbol{x}, \Delta t)$ for any choice of positive integer n. This *discrete recursion* is often used for computer simulation.

By the limitation of current digital electric computers, all we can use are 4 arithmetic operations to implement f. Furthermore, we are allowed to use only a finite number of operations. As we can see later, it is quite challenging to simulate natural phenomena with these limitations.

Differential Equation Solution

A Simple Example

Classical dynamics states that *the world is governed by differential equations*. Let's see a very simple example, the one dimensional free fall of a particle:

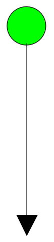
$$m\frac{d^2}{dt^2}x = -mg$$
where

t is time (1)

x is hieght of the paricle

g is gravitational acceleration

m is mass of the particle



This is an ***Ordinary Differential Equation (ODE),*** because it includes only ordinary differentiation as opposed to other kinds such as partial differentiation. This ODE is ***second order*** because the *maximum order of derivative* is two.

Now, we want to simulate the behavior of this particle using computers. We immediately get into problems. Differential equations are defined *in continuous space and time*. But the current computer is limited to discrete computation. Computers do not understand differential operators. It can execute only arithmetic operations finite times. Actually, we can only hope to get approximate solution.[1]

Let's try to apply the discrete recursion to this problem. First of all, we need to define the ***state*** of this system. What state would fully describe a particle like this? It is helpful to remember what Laplace said 177 years ago:

> "If we knew the position and velocities of all the particles in the universe, then we would know the future for all time, and the past as well."

Thus, the state we are looking for is the position and velocity of the particle. Let's denote them as

x: position
$\dot{x}$: velocity

Now, you may wonder why we can treat x and $\dot{x}$ as if they are independent variables. They are *not*. $\dot{x}$ is time derivative of x. But, we can pretend so if we explicitly state it:

$$\dot{x} = \frac{d}{dt}x$$

In this way, we can rewrite the previous equation into two equations:

$$\begin{aligned}\frac{d}{dt}x &= \dot{x}\\ m\frac{d}{dt}\dot{x} &= -mg\end{aligned} \qquad (2)$$

[1] Of course, you can integrate equation (1) twice to get a closed form solution of the differential equation ($x(t) = -1/2gt^2 + \dot{x}_0 t + x_0$). But closed form solution does not always exist for general differential equation, so let's pretend that there is no closed form solution for this case, too.

Equation (2)'s formulation converts the equation (1) into a first order differential equation[2]. Next, we discretize the state variables in time dimension.

$$x_i = x(i\Delta t)$$
$$\dot{x}_i = \dot{x}(i\Delta t)$$

Δt is a time step. It does not have to be a constant. But for the time being let's consider only a fixed time step. i is a nonnegative integer. $i = 0$ (i.e. $t = 0$) is the starting time of the simulation. So, x_0 and $\dot{x}_0$ comprise the initial state.

Now, all we need to do is to find a way to compute $(x_{i+1}, \dot{x}_{i+1})$ from $(x_i, \dot{x}_i)$ based on (2). The simplest way is ***Euler's method***:

$$x_{i+1} = x_i + \Delta t\left(\left.\frac{dx}{dt}\right|_{t=i\Delta t}\right)$$
$$\dot{x}_{i+1} = \dot{x}_i + \Delta t\left(\left.\frac{d\dot{x}}{dt}\right|_{t=i\Delta t}\right) \qquad (3)$$

Applying this to (2), we get

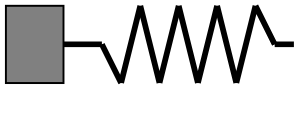
$$x_{i+1} = x_i + \Delta t \dot{x}_i$$
$$\dot{x}_{i+1} = \dot{x}_i - \Delta t g$$

The first equation says that if the particle *keeps* the current velocity $\dot{x}_i$ for Δt, it will move $\Delta t\dot{x}_i$ and reach $\dot{x}_i + \Delta t\dot{x}_i$. The second equation can be interpreted similarly. If the particle is under the gravitational acceleration $-g$ for Δt, then the velocity will increase $-\Delta t\, g$. In this particular case, the acceleration stays constant, so there is no error in the numerical integration of the acceleration. The numerical integration of velocity, however, is not accurate because velocity changes during the interval Δt. We can make Δt very small so that we can assume constant velocity in the small duration. But an overly small Δt will cause *catastrophic cancellation* in floating point computation. Furthermore, a small Δt implies many iteration steps, which are not desirable for high performance computation. We want a nice and large stride for each step.

Runge-Kutta Method

Runge-Kutta method is one of the most widely used numerical integrator for differential equations. Here we consider the oscillation of a linear spring. It is a little more complicated example than the free fall of a particle.

$$m\frac{d^2}{dt^2}x = -kx$$



where

t is time

x is horizontal position of the paricle

m is mass of the particle

[2] Alternatively we can use only x as state. In this case, we can deduce the current velocity from successive values of x in the past. We will discuss this method in the ***boundary value problem*** section.

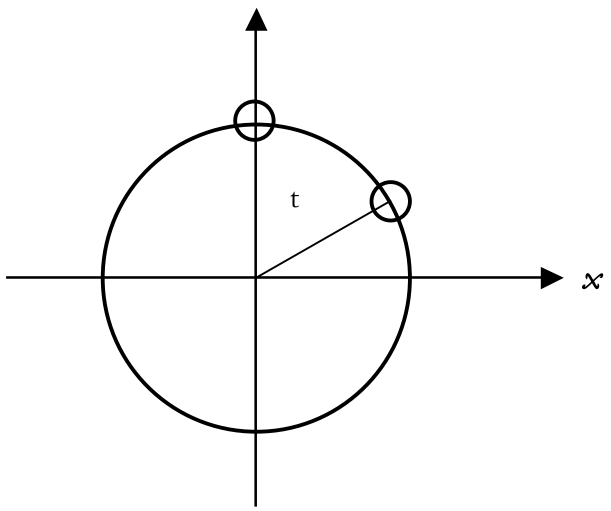
For the sake of simplicity, let's consider a case in which *m=k*.

$$\frac{d^2}{dt^2}x = -x$$

We can convert it to a first order ODE:

$$\frac{d}{dt}x = \dot{x}$$

$$\frac{d}{dt}\dot{x} = -x$$



And let's pick initial values:

$$x\big|_{t=0} = 0$$

$$\dot{x}_{t=0} = 1$$

We can easily get the closed form solution:

$$x(t) = \sin(t)$$

$$\dot{x}(t) = \cos(t)$$

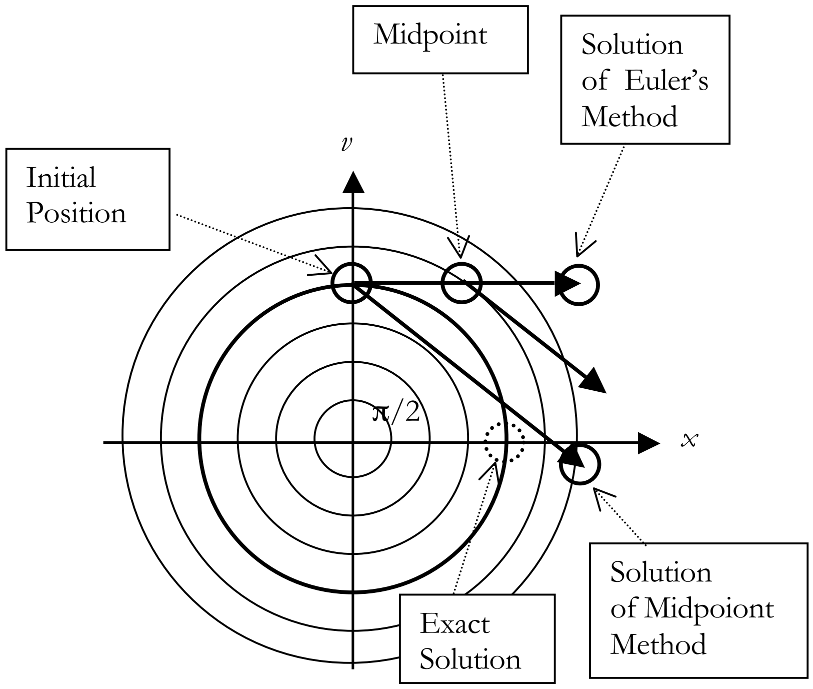
If we plot (x(t), ẋ(t)) as the advance of *t*, we get a circular trajectory.
Note also that, at any given point (x, ẋ), we can evaluate the derivatives as (x, -ẋ). In other words, the derivatives form a 2D vector field in the space (x, ẋ).

Again we consider that we do not know this exact solution, and attempt to solve this in numerical way. Let's try Euler's method with time step Δt = π/2 (≈1.57) (see the figure below). The exact solution is (x(π/2), ẋ(π/2)) = sin(π/2), cos(π/2))=(1,0). But, Euler's method gives us:

$$x(\pi/2) = x(0) + \dot{x}(0)\Delta t = 0 + 1\times\pi/2 \approx 1.57$$

$$\dot{x}(\pi/2) = \dot{x}(0) + \ddot{x}(0)\Delta t = 1 + 0\times\pi/2 \approx 1.00$$

This is not very accurate. The problem is that Euler's method ignores the change of derivatives during the interval Δt. To get the exact answer, we need the *average* value of derivatives in the duration. The ***midpoint method*** approximates the average by evaluating the derivatives at half way between the start point and the destination point. But,



because we do not know the destination (it is the answer we are looking for), we must use the approximation for it. The midpoint method uses the answer of Euler's method as the approximate destination point. Let's see a numerical example.

The initial postion is

$(0.0, 1.0)$

The solution of Euler's method is

$(\pi/2, 1.0)$

So the approximated midpoint is

$$\left(\frac{0.0+\pi/2}{2}, \frac{1.0+1.0}{2}\right) = (\pi/4, 1.0)$$

The derivatives at the midpoint are

$(1.0, -\pi/4)$

Therefore the final solution is

$$x(\pi/2) = x(0) + 1.0 \times \Delta t = \pi/2 \approx 1.57$$

$$v(\pi/2) = v(0) + (-\pi/4) \times \Delta t = 1.0 - \pi^2/8 \approx -0.23$$

As shown in the figure, there is great improvement over Euler's method.
The midpoint method is also called the **second order Runge-Kutta method**.

It is convenient to use general notation for ODE:

$$\frac{d}{dt}\boldsymbol{y} = f(\boldsymbol{y})$$

where $\boldsymbol{y}$ is a state vector of a system, and $f(\boldsymbol{y})$ is a derivative function.
In more general form, f is a function of t as well (i.e. denoted as $f(t,\boldsymbol{y})$), but in most cases of physical simulation, f is independent of time. Therefore t is omitted for the sake of simplicity.

For example, for the oscillation problem we are solving, $\boldsymbol{y}$ and f are defined as:

$$\boldsymbol{y} = \begin{pmatrix} x \\ \dot{x} \end{pmatrix}$$

$$f(\boldsymbol{y}) = \begin{pmatrix} \dot{x} \\ -x \end{pmatrix}$$

Using this notation, the second order Runge-Kutta method can be written in a compact form:

$$\boldsymbol{k_1} = \Delta t\, f(\boldsymbol{y_n})$$

$$\boldsymbol{k_2} = \Delta t\, f(\boldsymbol{y_n} + \tfrac{1}{2}\boldsymbol{k_1})$$

$$\boldsymbol{y_{n+1}} = \boldsymbol{y_n} + \boldsymbol{k_2} + O(\Delta t^3)$$

where

$$\boldsymbol{y_n} = \boldsymbol{y}(n\Delta t)$$

This method is known to have third order local error. That is why *$O(\Delta t^{3})$*. By evaluating derivatives at more points, we can get more accurate solution (in most cases). The following formula is the ***forth-order Runge-Kutta*** method.

$$\boldsymbol{k_1} = \Delta t\, f(\boldsymbol{y}_n)$$
$$\boldsymbol{k_2} = \Delta t\, f(\boldsymbol{y}_n + \tfrac{1}{2}\boldsymbol{k}_1)$$
$$\boldsymbol{k}_3 = \Delta t\, f(\boldsymbol{y}_n + \tfrac{1}{2}\boldsymbol{k}_2)$$
$$\boldsymbol{k}_4 = \Delta t\, f(\boldsymbol{y}_n + \boldsymbol{k}_3)$$
$$\boldsymbol{y}_{n+1} = \boldsymbol{y}_n + \frac{\boldsymbol{k}_1}{6} + \frac{\boldsymbol{k}_2}{3} + \frac{\boldsymbol{k}_3}{3} + \frac{\boldsymbol{k}_4}{6} + O(\Delta t^5)$$

The local error of the **forth-order Runge-Kutta** is *$O(\Delta t^{5})$*. In most cases higher order implies more accuracy, but it is true only if lower degree terms are dominant (in other word the function *$f(\boldsymbol{y})$* is sufficiently smooth). We can assume so in many cases because Δt is smaller than 1[3]. However, if f has a very large coefficient for a higher degree term, the term would have higher absolute value than lower degree terms. Thus higher order methods do not guarantee high accuracy in general.

Adaptive Step Size

Now how can we decide an appropriate step size? If the method is integrating a smooth part of function, a large step size can be safely used, while if it is going through a bumpy part, the step size must be small.
Our mission is maximizing the step size while keeping the error within preset tolerance. For each step, we should estimate error. If we find the error is larger than the tolerance, we must make the step size smaller and integrate the step again. If the error is within the limit, recompute the step size (make the step size larger), and go on to the next step.

Step doubling is a simple method to estimate error. Let's see how it works for the forth-order Runge-Kutta method. First, we take a normal step from t to $t+\Delta t$. The error is *$O(\Delta t^{5})$*, or we can write it as *$\phi\Delta t^5+O(\Delta t^{6})$*, where ϕ is an unknown coefficient[4]. Therefore

$$\boldsymbol{y}(t+\Delta t) - \boldsymbol{y}_1 = \phi\,\Delta t^5 + O(\Delta t^6)$$

computed solution $\boldsymbol{y_1}$ and the exact solution $\boldsymbol{y}(t+\Delta t)$ satisfy the following relationship.

Then we divide the step into half, and take two steps. The error of each step is *$\phi(\Delta t/2)^5+O(\Delta t^{6})$*, so the total error is *$2\phi(\Delta t/2)^5+O(\Delta t^{6})$*. Denoting the solution by 2 steps $\boldsymbol{y_2}$ we get

$$\boldsymbol{y}(t+\Delta t) - \boldsymbol{y}_2 = 2\phi(\Delta t/2)^5 + O(\Delta t^6) = \phi\Delta t^5/16 + O(\Delta t^6)$$

3 What happens if you have Δt larger than 1? For example, if you happen to pick msec for time unit, and your time step is 10msec, would a higher order method have larger errors?

4 By using Taylor expansion, we can show ϕ is $y^{(5)}(t)/5!$. It is assumed to be constant around the vicinity of t.

Subtracting the second equation from the first, we get

$$\boldsymbol{y}_2 - \boldsymbol{y}_1 = (15/16)\phi\Delta t^5 + O(\Delta t^6)$$

Therefore estimated error $\Delta\boldsymbol{y}$ is obtained as

$$\Delta\boldsymbol{y} = \phi\Delta t^5 = \frac{16}{15}(\boldsymbol{y}_2 - \boldsymbol{y}_1)$$

Given the error tolerance Δy_{max}, the new step size Δt_{new} should satisfy

$$\Delta y_{\max} > |\,\phi\Delta t_{new}{}^5\,|$$

Therefore

$$\frac{\Delta y_{\max}}{|\Delta\boldsymbol{y}|} > \frac{|\,\phi\Delta t_{new}{}^5\,|}{|\,\phi\Delta t^5\,|}$$

$$i.e.$$

$$\left(\frac{\Delta y_{\max}}{|\Delta\boldsymbol{y}|}\right)^{1/5}\Delta t > \Delta t_{new}$$

Thus we can compute the upper bound of the new time step that guarantees the maximum error Δy_{max}.[5]

Other ODE Solution Methods

Runge-Kutta method requires many evaluations of derivative per step. ***The multipoint methods*** exploit derivatives of previous steps to achieve higher order accuracy. At each step, the derivative is evaluated only once. Several derivatives of already determined steps are interpolated by a polynomial function, and by integrating the polynomial, we can get the solution of the next step.
Multipoint methods are known to be accurate and computationally less expensive than Runge-Kutta methods. Multipoint methods use derivatives of a few previous steps, so we have to use ***self-starting*** methods, such as Runge-Kutta methods, to compute those steps. In physically based simulation that involves frequent collision (or other kind of discontinuous events), multipoint methods have to be reinitialized many times, which may undermine the efficiency of this method.

[5] $\Delta\boldsymbol{y}$ and $\boldsymbol{\phi}$ are vectors. So the upper bound should be evaluated component wise, and maximum value should be taken.