A Brief Introduction to Numerical Methods for Differential Equations

January 10, 2011

This tutorial introduces some basic numerical computation techniques that are useful for the simulation and analysis of complex systems modelled by differential equations. Such differential models, especially those partial differential ones, have been extensively used in various areas from astronomy to biology, from meteorology to finance. However, if we ignore the differences caused by applications and focus on the mathematical equations only, a fundamental question will arise: Can we predict the future state of a system from a known initial state and the rules describing how it changes? If we can, how to make the prediction?

This problem, known as Initial Value Problem(IVP), is one of those problems that we are most concerned about in numerical analysis for differential equations. In this tutorial, Euler method is used to solve this problem and a concrete example of differential equations, the heat diffusion equation, is given to demonstrate the techniques talked about. But before introducing Euler method, numerical differentiation is discussed as a prelude to make you more comfortable with numerical methods.

1 Numerical Differentiation

1.1 Basic: Forward Difference

Derivatives of some simple functions can be easily computed. However, if the function is too complicated, or we only know the values of the function at several discrete points, numerical differentiation is a tool we can rely on.

Numerical differentiation follows an intuitive procedure. Recall what we know about the definition of differentiation:

$$\frac{df}{dx} = f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

which means that the derivative of function $f(x)$ at point x is the difference between $f(x+h)$ and $f(x)$ divided by an infinitesimal h. Based on this definition, we can use an h that is small enough to compute the derivative as an approximation. And intuitively we believe that smaller h leads to more accurate derivative, which will be proved soon.

$$\frac{df}{dx} \approx f'(x)_{approx} = \frac{f(x+h) - f(x)}{h}$$

This approximation, known as *Forward Difference*, has, actually, suggested the program to compute derivatives: Pick a small h; evaluate $f(x+h)$ and $f(x)$; finally apply the above formula. What we want to discuss in the rest of this section are: Can we do better(more accurate)? Can we do more(derivatives of multi-dimension and high-order)?

1.2 Error Analysis

First we try to make sure that smaller h does generate better accuracy. To analyze the possible inaccuracy, which is called *error* in Numerical Computation, we have to find out what is the source of inaccuracy in the above computation.

An obvious source of error is the use of approximation to compute the derivative. Consider the Taylor series expansion of $f(x+h)$:

$$f(x+h) = f(x) + hf'(x) + \frac{1}{2}h^2 f''(x) + \frac{1}{6}h^3 f'''(x) + \dots$$

which implies,

$$f'(x)_{approx} = \frac{f(x+h) - f(x)}{h} = f'(x) + \frac{1}{2}hf''(x) + \frac{1}{6}h^2 f'''(x) + \dots \tag{1}$$

From equation (1) we can find that there is an error,

$$e_t = f'(x)_{approx} - f'(x) = \frac{1}{2}hf''(x) + \frac{1}{6}h^2 f'''(x) + \cdots = O(h) \tag{2}$$

between the approximation $f'(x)_{approx}$ and the accurate derivative $f'(x)$. Error e_t is called *truncation error*. As Figure 1 shows, We can conclude that smaller h does lead to smaller e_t, based on equation (2).

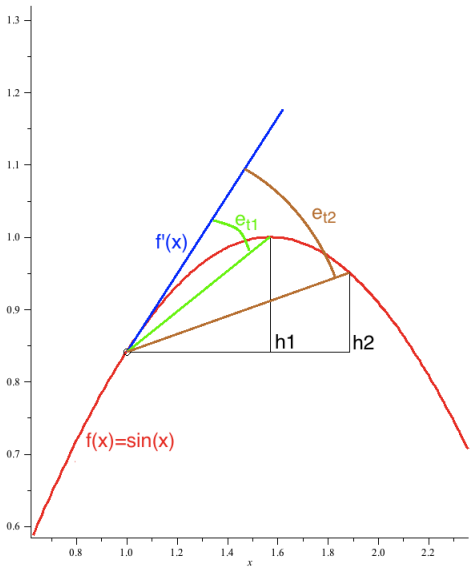

Figure 1: This is an example where function $f(x) = sin(x)$. $h_1 < h_2$ leads to truncation error $e_{t1} < e_{t2}$.

Another source of error is hidden in the way real numbers are stored in a computer's memory and registers[1]. Every real number is stored in a floating-point format with some fractional error e_m, which causes an error called *roundoff error* $e_r \approx e_m \cdot |f(x)/h|$. We can decrease e_r with a carefully chosen h, but we can not totally eliminate e_r. [2]

Note that roundoff error is a fractional error based on the accuracy of approximation. Generally, a smaller truncation error produces a smaller roundoff error.

Combining truncation error and roundoff error together, we get the total error:

$$e = e_t + e_r$$

[1]Please refer to Appendix A for details.

[2]Please refer to page 230, *Numerical Recipes* 3rd Edition for details.

1.3 Backward Difference and Central Difference

Now we try to find some computation techniques that can produce better accuracy.

First we try *Backward Difference*:

$$f'(x)_{approx-b} = \frac{f(x) - f(x-h)}{h}$$

Using Taylor series expansion again, we can get

$$e_{t-b} = f'(x)_{approx-b} - f'(x) = -\frac{1}{2}hf''(x) + \frac{1}{6}h^2 f'''(x) + \cdots = O(h)$$

It seems that there is no improvement.

Then we try *Central Difference*:

$$f'(x)_{approx-c} = \frac{f(x+h) - f(x-h)}{2h}$$

and by Taylor series expansion,

$$e_{t-c} = \frac{1}{6}h^2 f'''(x) + \frac{1}{120}h^4 f^{(5)}(x) + \cdots = O(h^2)$$

which gives us a smaller truncation error, and hence possibly, a smaller total error. Thus we usually use central difference instead of forward difference.

1.4 Multi-dimensional and High-Order Derivatives

For multi-dimensional and high-order derivatives, we can just apply the above central difference formula several times.

For example, the second order derivative:

$$\frac{d^2 f}{dx^2} = \frac{d}{dx}\frac{df}{dx} \approx \frac{f(x+2h) + f(x-2h) - 2f(x)}{4h^2}$$

and the second order partial derivative:

$$\frac{\partial^2 f}{\partial x \partial y} = \frac{\partial}{\partial y}\frac{\partial f}{\partial x} \approx \frac{f(x+h, y+h) + f(x-h, y-h) - f(x+h, y-h) - f(x-h, y+h)}{4h^2}$$

1.5 Further Reading

- A great introduction of numerical differentiation is in Chapter 5.7 of *Numerical Recipes*, 3rd Edition.
- A discussion about the truncation error and roundoff error can be found in

  `http://farside.ph.utexas.edu/teaching/329/lectures/node33.html`

2 Euler Method for Numerical Integration

2.1 Explicit and Implicit Methods

Before discussing Euler Method, we introduce the concepts of *Explicit Method* and *Implicit Method* first. These two methods are two approaches used in numerical analysis for differential equations.

Both explicit method and implicit method focus on solving the Initial Value Problem. The difference is that in explicit method, the state of the system in a later time, say, $y(t + \Delta t)$, is evaluated from the current state, $y(t)$, directly:

$$y(t + \Delta t) = f(y(t), \Delta t)$$

while in implicit method, there is only an equation describing the relationship between future state of the system, $y(t + \Delta t)$, and current state, $y(t)$:

$$g(y(t), y(t + \Delta t), \Delta t) = 0$$

To get the exact future state, we must solve this equation, which usually means an extra computation.

2.2 Explicit and Implicit Euler Methods

In this section we talk about Euler method. There are other methods that are faster and more accurate[3], but Euler method is simple to understand and easy to implement, which makes it a great starting point.

We start from Ordinary Differential Equations. First we need a formal definition of initial value problems. A differential equation describing the derivative of $y(t)$ and the initial value of $y(t_0)$ are given as follows:

$$\begin{cases} y'(t) = f(t, y(t)) \\ y(t_0) = y_0 \end{cases}$$

Assume that $y(t)$'s value at some point t_c is already known as current state, and we need to know future state $y(t_c + \Delta t)$. For small enough Δt, consider Forward Difference

$$y'(t_c) \approx \frac{y(t_c + \Delta t) - y(t_c)}{\Delta t}$$

which implies

$$y(t_c + \Delta t) \approx y(t_c) + \Delta t \cdot y'(t_c)$$

Ignore the approximation,

$$y(t_c + \Delta t) = y(t_c) + \Delta t \cdot f(t_c, y(t_c))$$

This is the explicit Euler method formula. If we fix Δt, use t_i to denote $t_0 + i\Delta t$ and y_i for $y(t_i)$, this formula can be written in the discrete form:

$$y_i = y_{i-1} + \Delta t \cdot f(t_{i-1}, y_{i-1}) \tag{3}$$

Based on equation (3) we can write a basic algorithm to solve IVP using the explicit Euler method:

Explicit-Euler-Method(f, y_0, t_0, t, dt)
$t_c \leftarrow t_0$
$y_c \leftarrow y_0$
while $t_c < t$ **do**
  $y_c \leftarrow y_c + dt \cdot f(t_c, y_c)$
  $t_c \leftarrow t_c + dt$
end
return y_c

Similarly, we can also use Backward Difference instead of Forward Difference.

$$y'(t_c) \approx \frac{y(t_c) - y(t_c - \Delta t)}{\Delta t}$$

which implies

$$y(t_c) \approx y(t_c - \Delta t) + \Delta t \cdot y'(t_c)$$

Ignore the approximation,

$$y_i = y_{i-1} + \Delta t \cdot f(t_i, yi)$$

This is the implicit Euler method formula.

[3]Euler method is a special case of Runge-Kutta methods. Other Runge-Kutta methods, especially the fourth-order Runge-Kutta method, are widely used in solving differential equations.

2.3 Error Analysis and Stability of Euler Method

However, a question remains: The Forward Difference is an approximation. Is that sound?

Generally, the answer is YES if Δt is small enough. Similar to numerical computation of differentiation, the truncation error e_{t-step} of each step of Euler method from t_c to $t_c + \Delta t$ is :

$$e_{t-step} = \frac{1}{2}\Delta t^2 y''(t_c) + \frac{1}{6}\Delta t^3 f'''(t_c) + \cdots = O(\Delta t^2)$$

If there is $n = \frac{t-t_0}{\Delta t}$ steps in the computation, a good assumption is that the total truncation error $e_{t-total} = ne_{t-setp}$, and $e_{t-total}$ is nearly proportional to Δt, which means that smaller Δt leads to smaller error. So, in order to get accurate solution, we need small Δt.

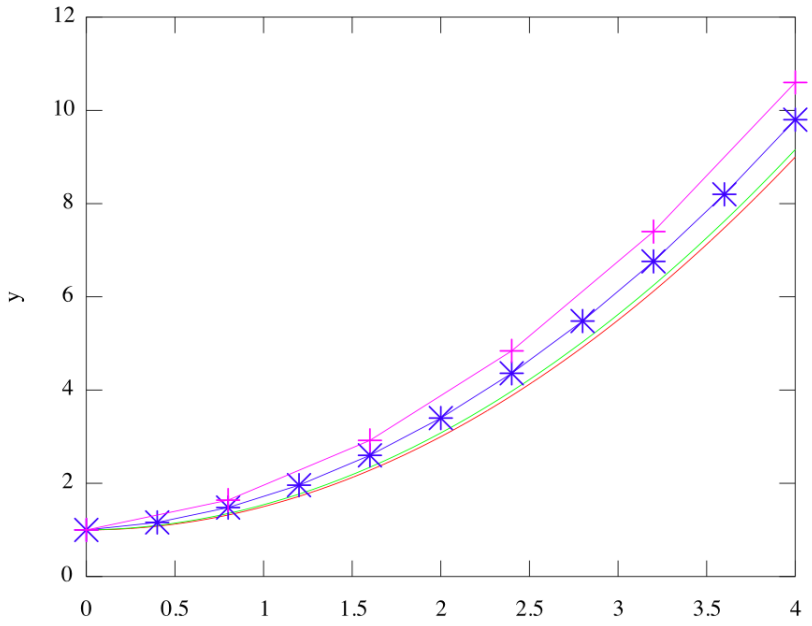

Figure 2: Smaller Δt leads to smaller error

In Figure 2 we show an example of using Euler method on differential equation

$$\begin{cases} y'(t) = t \\ y(0) = 1 \end{cases}$$

where the red line indicates the actual result $y = \frac{t^2}{2} + 1$, the green line indicates the result of choosing $\Delta t = 0.08$, the blue line indicates the result of choosing $\Delta t = 0.4$, and the magenta line indicates the result of choosing $\Delta t = 0.8$. It can be easily seen from the graph that smaller Δt does lead to smaller error.

However, when the Δt is not small enough, the explicit Euler method does not guarantee an accurate answer. In fact, the numerical integration becomes unstable.

For example, we take a look at the differential equation:

$$\begin{cases} y'(t) = -ky \\ y(0) = 1 \end{cases} \tag{4}$$

This is still a simple differential equation and we get the analytical solution is $y = e^{-kt}$. But if we use explicit Euler method to solve this problem when $k = 5$, the answer we get is as Figure 3.

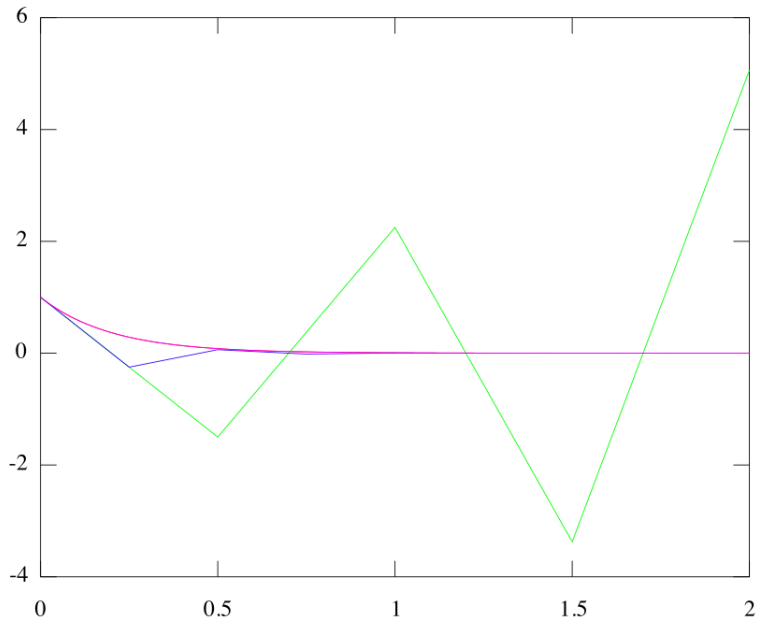

Figure 3: Instability of explicit Euler method

In Figure 3, the red line indicates the actual answer $y = e^{-5t}$, which overlaps with a magenta line that indicates the result of choosing $\Delta t = 0.0002$. The bue line indicates the result of choosing $\Delta t = 0.25$, and the green line indicates the result of choosing $\Delta t = 0.5$. The blue line oscillates at first, but it converges to the actual solution at last. For the green line, it seems that it will never converge to the right answer. The differences between those lines are the choices of Δt, which leads to the fact that the explicit Euler method will be instable when Δt is not small enough for equations like equation (4), which are called *stiff equations*.

Now we discuss the reason of this instability. Consider the explicit Euler method formula:

$$y_{i+1} = y_i - \Delta t \cdot k y_i = (1 - k\Delta t) y_i$$

When $\Delta t = 0.0002$, $0 < 1 - k\Delta t < 1$, the numerical integration converges to the actual solution. When $\Delta t = 0.25$, $-1 < 1 - k\Delta t < 0$, the numerical integration oscillates, but still converges to the right answer. When $\Delta t = 0.5$, $1 - k\Delta t < -1$, the numerical integration oscillates, and never converges to a fixed answer. Thus, the explicit Euler method requires a small Δt to make the numerical integration accurate.

The instability can be avoided by using implicit Euler method instead of the explicit one. But usually implicit Euler method needs an extra computation.

The wide application of Euler method is limited by the instability and the low speed. To solve the problem of instability, several improvements, such as Euler-Cromer method, are introduced as refinements of the original Euler method. However, it is hard to speed up the computation of Euler method. So some new methods, including Runge-Kutta methods, Richardson extrapolation and predictor-corrector methods, are discovered to solve large differential equations.

2.4 Further Reading

- An introduction to Euler Method can be retrieved from

  `http://tutorial.math.lamar.edu/Classes/DE/EulersMethod.aspx`

  This introduction contains several examples of solving differential equations using Euler method, including one that shows the instability of Euler method.

- A great discussion about the instability can be retrieved from

  `http://www.cs.cmu.edu/~baraff/sigcourse/slidese.pdf`

3 Heat Diffusion: An Example

In this section we take a look at a more concrete example to demonstrate what we discussed above.

3.1 The Problem

Heat diffusion equation:

$$\frac{\partial u}{\partial t} = \alpha \Delta u$$

describes the conduction of heat in a given region over time, in which u is the temperature distribution function, t is time, α is a positive constant called thermal diffusivity and Δ is the *Laplacian* operator.

The Laplacian operator is defined as the divergence of the gradient of a function. Recall the definition of divergence $\nabla\cdot$ and gradient ∇, the Laplacian operator is defined as:

$$\Delta f = \nabla^2 f = \nabla \cdot \nabla f$$

For function f in Cartesian coordinates $x_1, \ldots, x_n$, $\Delta f = \sum_{i=1}^{n} \frac{\partial^2 f}{\partial x_i^2}$

Knowing the definition of the Laplacian operator, the meaning of heat diffusion equation is clear: At some point, the rate of temperature change over time equals the thermal diffusivity times the divergence of the gradient of the temperature at that time.

3.2 Use Euler Method to Solve the Problem

Here we focus on the simplest case of the Heat Diffusion problem: heat diffusion of one dimensional space, i.e., a cable. The problem is: Assume we have a cable of length 10, and it is placed on the number axis from $x = 0$ to $x = 10$. The temperature $u(x, t)$ of a point x on the cable at initial time $t = 0$ is given as a function $u(x, 0) = 10 \cdot \sin(\pi x/10), 0 \leq x \leq 10$. What will the temperature distribution look like at time $t > 0$?

The formal form of this question is:

$$\begin{cases} \dfrac{\partial u}{\partial t} = \alpha \Delta u \\ u(x, 0) = 10 \cdot \sin(\pi x/10) \\ 0 \leq x \leq 10 \end{cases}$$

First, the Laplacian operator on one dimensional space is in the form

$$\Delta u = \frac{\partial^2 u}{\partial x^2}$$

in which $\dfrac{\partial^2 u}{\partial x^2}$ is similar to $\dfrac{d^2 u}{dx^2}$ that has already been discussed in Section 1.4. Use dx to denote $2h$, and we can get:

$$\Delta u(x, t) = \frac{u(x + dx, t) + u(x - dx, t) - 2u(x, t)}{dx^2}$$

Then, consider the temperature distribution function $u(x, t)$. Using Euler method, and let $dt = \Delta t$, we get:

$$u(x, t + dt) = u(x, t) + \alpha \cdot dt \cdot \frac{u(x + dx, t) + u(x - dx, t) - 2u(x, t)}{dx^2}$$

For boundaries of $x = 0$ and $x = 10$, we use equations

$$\begin{cases} u(0, t + dt) = u(0, t) + \alpha \cdot dt \cdot \dfrac{2u(dx, t) - 2u(0, t)}{(dx)^2} \\ u(10, t + dt) = u(10, t) + \alpha \cdot dt \cdot \dfrac{2u(10 - dx, t) - 2u(10, t)}{(dx)^2} \end{cases}$$

to make an approximation.

Given the equations above, we can write a program to solve the heat diffusion problem.

```
Euler-Method-Heat-Diffusion(t, dt)
▷ t is the total time of simulation, dt is the time step
x ← 10
▷ x is the length of the cable
dx ← 0.2
dt ← 0.02
nx ← x/dx
nt ← t/dt
▷ nx is the number of cells on the cable, nt is the number of time steps
α ← 0.9
dtdd_o_dx2 ← dt/dx/dx
allocate array u[1, ..., nx][0, ..., nt]
▷ initialize values of u at time 0
for i ← 1 to nx do
    u[i][0] ← 10sin(πi/nx)
end
▷ start simulation
for i ← 1 to nt do
    for j ← 1 to nx do
        if j = 1 then
            Laplacian ← α · dtdd_o_dx2 · (2u[2][i − 1] − 2u[1][i − 1)
        end
        else
            if j = nx then
                Laplacian ← dtdd_o_dx2 · (2u[nx − 1][i − 1] − 2u[nx][i − 1])
            end
            else
                Laplacian ←
                dtdd_o_dx2 · (u[j + 1][i − 1] + u[j − 1][i − 1] − 2u[j][i − 1])
            end
        end
        u[j][i] ← u[j][i − 1] + α · Laplacian
    end
end
```

The result of this program is drawn as a 3d graph in Figure 4, in which we can see that at initial time the temperature distribution follows a sine function and high temperature area (red) turns to cool down and low temperature areas (blue) turn to warm up as time passes.

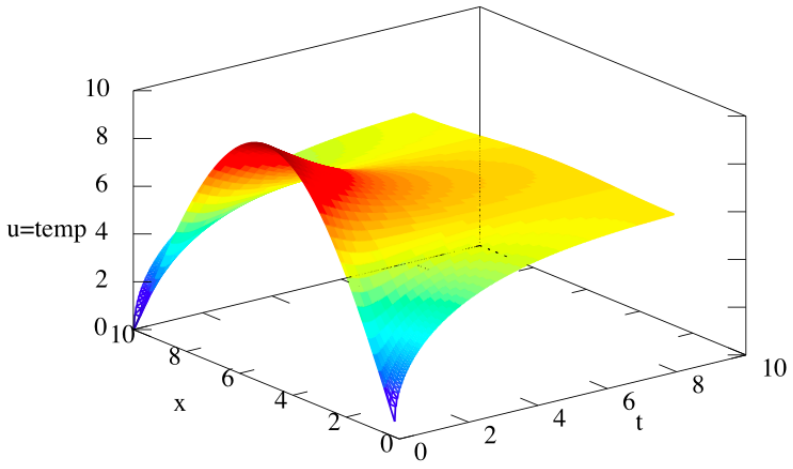

Figure 4: Heat Diffusion

3.3 Further Reading

- Here is a detailed description of numerical solution of the Heat Diffusion Problem. Several methods are introduced in it, including Euler method. There is also a discussion about the choosing of α, dt, dx, which is important to make the Euler method produce right answer. Retrieved from

  `http://mightylib.mit.edu/Course%20Materials/22.00/Spring%202002/Notes/lecture_16.pdf`

A How a real number is stored in a computer

Even though there are programming languages that contain a data type called *real*, there are no actual *real* data types with infinite accuracy in programming languages. This is easy to understand, because there are neither infinite memory, nor infinite time, to compute and store a real number with infinite accuracy.

Usually, real numbers are stored in floating point form that is similar to scientific notation. A typical floating point number is of the form:

$$\text{significant digits} \times \text{base}^{\text{exponent}}$$

The advantage of such representation is that a larger range of numbers can be represented using the same amount of memory, compared to fixed point form. But the tradeoff is that only the significant digits are precise and all information beyond the significant digits are lost, which is the source of roundoff error. Commonly, the base is set to 2 in modern computers. So to store the floating point number, only the significant digits and exponent are needed. An extra bit is used to store the sign of the number.

In C programming language, we use *float* and *double* type[4] to describe a real number. They both are floating point data types. Typically, *float* type is represented in one word(32 bits), and *double* type in two words (64 bits). We take float as an example.

[4]There are other floating-point types in C programming languages, such as *long double*, and third-party numerical computation libraries that provide data types with better precision.

Given that the base is always set to 2, there is no need to store it. The significant digits and exponent are in binary form. Exponent is set to be an integer. Significant digits are in the form of $1.x_1x_2x_3x_4...$, i.e., the radix point is just after the first digit. A trick is that the first digit is always 1, or we can change the exponent to make the first digit of significant digit 1. So the first digit is ignored and only the rest digits $x_1x_2x_3x_4...$, called *significand*, are stored.

Bit #	0 ... 22	23 ... 30	31
Meaning	Significand	Exponent	Sign

The difference between float and double is that double has a better accuracy.

Type	Total Bits	Sign	Exponent	Significand
Float	32	1	8	23
Double	64	1	11	52

The 23 bits significand of float type guarantees a precision of 6 decimal digits, which is usually not accurate enough for scientific computation. The 52 bits significand of double type guarantees a precision of 52 decimal bits, which is sufficient for some simple numerical computation.