Computational Physics – Week 3

Andrei Alexandru

Feb 4, 2019

1 Introduction

This week will discuss numerical integration of ordinary differential equations. The simplest case is a one dimensional equation:

$$\dot{x} = f(x,t)\,, \qquad x(0) = x_0\,. \tag{1}$$

This problem can be solved approximately using Euler method. We will solve this iteratively using a step-size Δt. Our solution will offer an approximation x_n for $x(n\Delta t)$, which will be determined iteratively by solving for x_n using x_{n-1} as an input. The Euler method consist in the following

$$x_n = x_{n-1} + f(x_{n-1}, t_{n-1}) \times \Delta t\,, \tag{2}$$

where $t_n \equiv n\Delta t$. This process starts with $x_0 = x(0)$ and compute x_1, then compute x_2 using x_1, and so on.

This method has an error of the order Δt, that is $x(t)$ and x_n (with $n = t/\Delta t$) for a fixed t will differ by a correction that vanishes linerly with Δt. In many cases it is useful to setup integrations routines that have better convergence rates. Runge-Kutta methods can be designed to produce very accurate integration routines. Here we will present the simplest of them, RK2:

$$\begin{aligned} k_1 &= f(x_{n-1}, t_{n-1})\,, \\ k_2 &= f(x_{n-1} + k_1 \times \Delta t/2, t_{n-1} + \Delta t/2)\,, \\ x_n &= x_{n-1} + k_2 \times \Delta t\,. \end{aligned} \tag{3}$$

This process involves two evaluations of the function f per step, but it generates an approximation that vanishes as $(\Delta t)^2$.

The equations above apply to the case where there is only one differential equation. This can be easily extended to the case where we have more than one variable:

$$\begin{aligned} \dot{x}_1 &= f_1(x_1, x_2, \ldots, x_k, t)\,, \qquad x_1(0) = (x_1)_0 \\ \dot{x}_2 &= f_2(x_1, x_2, \ldots, x_k, t)\,, \qquad x_2(0) = (x_2)_0 \\ &\vdots \\ \dot{x}_k &= f_k(x_1, x_2, \ldots, x_k, t)\,, \qquad x_k(0) = (x_k)_0\,. \end{aligned} \tag{4}$$

This equations can be written more compactly as

$$\dot{\boldsymbol{x}} = \boldsymbol{f}(\boldsymbol{x}, t)\,, \qquad \boldsymbol{x}(0) = \boldsymbol{x}_0\,. \tag{5}$$

Above $\boldsymbol{x} = \{x_1, \ldots, x_k\}$ and similarly for the initial conditions. The integration of the system of equations can be solved the same as before: For example the Euler method consists of

$$\boldsymbol{x}_n = \boldsymbol{x}_{n-1} + \boldsymbol{f}(\boldsymbol{x}_{n-1}, t_{n-1}) \times \Delta t\,. \tag{6}$$

The discussion above applies to ODE's of the first order. However, equations involving higher order derivatives can be transformed into first order equations. To see how this works consider the second order equations encountered in classical mechanics $m\ddot{\boldsymbol{x}} = \boldsymbol{F}(\boldsymbol{x}, t)$ with $\boldsymbol{x}(0) = \boldsymbol{x}_0$ and $\dot{\boldsymbol{x}}(0) = \boldsymbol{v}_0$. We introduce the velocities $\boldsymbol{v}$ as auxiliary variables. We have then the following system of *first order* equations

$$\begin{aligned} \dot{\boldsymbol{x}} &= \boldsymbol{v}\,, \\ \dot{\boldsymbol{v}} &= \boldsymbol{F}/m\,, \end{aligned} \tag{7}$$

which can be solved using the methods described above.

2 Example problem

We will solve three problems in class. First a very simple exponential decay problem:

$$\dot{x} = -x\,, x(0) = 1\,. \tag{8}$$

The solution for this is $x(t) = e^{-t}$, so we can compare the Euler and RK2 method results with the exact answer.

The second problem involves two variables, in this case the fox and rabbits problems:

$$\begin{aligned} \dot{f} &= 0.1 \times r\,, \\ \dot{r} &= -0.2 \times f\,. \end{aligned} \tag{9}$$

The idea is that the populations of foxes, f, increases proportionally with the population of rabbits r, since more food is available. For the same reason the population of rabbits decreases at a rate proportional to the number of foxes. As initial conditions we take the a rather unreasonable $r(0) = 1$ and $f(0) = 0$.

The third problem to solve is a two dimensional mechanics problem. We have a particle of mass $m = 1$ moving under the influence of a potential $V(x, y) = x^2y^2$.

3 Teaching objectives

- Numerical integration of ODEs using Euler and RK2 methods.
- Using command line to control parameters.

4 Notes on C programming

All variables declared in a program, for example `double x`, have to be stored in memory. This is done automatically by the compiler at least if the amount required can be determined at *compile time*. When the required memory is only known at *run time*, we need to allocate memory explicitly. We will discuss *dynamical memory allocation* latter in the semester.

To address locations in memory we use *pointers*. At the base level pointers are just memory addresses, but the C language provides some facilities to manipulate the pointers and the data they point to easily. First of all, the pointers have a type: when the underlying data is of type `type`, then the pointer has type `type*`. This helps the compiler compute the *pointer arithmetic* correctly. For example if we have `double* p`, a pointer to data of type `double`, then `p+1` is a pointer to the location in memory that stores the next `double`. Since the size of a `double` is 8 bytes, `p+1` corresponds to a memory address offset by 8 bytes with respect to `p`. The memory required for the underlying data type is needed to figure these offsets correctly.

To connect the pointers and the data they point to, we use two related operators: `*` and `&`. For a variable `type x`, the pointer to its location is `type* p = &x`. Assuming that we have the pointer `p` we can manipulate the data it points to using `*p`, for example to set it to value 2 we would use `*p=2`.

Pointers are necessary when we want to pass variables to functions that change their values. In C all parameters to functions are passed *by value*, that is for a variable passed to a function a copy is made. Inside the function body we have access to the value through its copy, but any modification on the copy will be inaccessible to the caller. For the function to affect the value of the variables we have to pass to the function pointers to the variables. For example to write a function that swaps the data between two variables we use

```c
void swap(double* a, double* b)
{
  double c=*a;
  *a = *b;
  *b = c;
}
```

Inside the function we modify the values using the pointers. If we want to swap the values of two variables `double x,y` we call `swap(&x, &y)`. When the function return the variables `x, y` will have their values swapped.

We also encounter pointers when dealing with arrays. In C an array represents a list of data of the same type that is stored in a contiguous region of memory. An array called `vec` of length `n` of data of type `type` is declared using `type vec[n]`. To access the elements of the array we use `vec[0]`, for the first, `vec[1]` for the second, etc. The last element of the array is `vec[n-1]`. To set the value of the second element to 2 we use `vec[1]=2`. The array type includes its length, so `double[3]`, an array of doubles of length 3, is different than `double[4]`. However, when you pass the array to a function the length of the array is lost and the type is implicitly converted from `type[n]` to `type*`, a pointer type. This pointer variable is pointing to the first element of the array, thus if we call `function(double vec[3])`, `vec` will be copied into a pointer that stores `&vec[0]`. The array `vec` is a *name* not a variable, so we cannot assign to it. For example `vec=p` for `p` a pointer of type `type*` is not allowed. However we can use it to assign to a pointer variable, so `p=vec` is allowed and is equivalent to `p=&vec[0]`, thus `vec` in this instance behaves as a read-only pointer to the beginning of the array. Incidentally, this means that `vec+i` is the pointer to `vec[i]` and we can also access the elements of the array using either `vec[i]` or `*(vec+i)` – these two forms are equivalent.

Text (or string) variables in C are represented as array of `char` type. Each `char` holds one character in the string. For example `char c='a'` sets the variable `c` to a value corresponding to a. Internally this `char` variable holds these values by using the codes associated with the characters corresponding to the ASCII table (for example the code for 'a' is 97.) One important thing to know is that the last character of each string should have code 0. This allows us to pass strings to functions without specifying their length (recall that arrays passed to functions convert to pointers to the first element.) The function will then read all the characters until it encounters a code 0 character. If we forget to add it the program will likely behave erroneously. However, it is often unnecessary to worry about this since the compiler does this for us when initializing strings. For example setting `char name[]="Andrei"` creates an array `name` with *7* characters, with the last one set to code 0.