Y2 Computational Physics Laboratory

UNIVERSITY OF BIRMINGHAM

Exercise 2

Numerical solution of ordinary differential equations: from radioactive decay to a skydiver

Aims:

- To use the Euler and Runge-Kutta methods for the numerical solution of ordinary differential equations
- To use C++ to solve some real, physical problems
- To get more practice using C++ functions
- **1 mark given for overall presentation**

Duration: **3** weeks

Hand-in deadline: **End of session #5**

Introduction

This exercise will apply your programming skills to the numerical solution of ordinary differential equations (ODEs). This is a key numerical method that is widely used in computer modelling in both academic and industrial research.

The ODEs in this exercise describe two very different physical systems: (*i*) the decay of radioactive nuclei, and (*ii*) a skydiver falling under gravity taking account of air drag. In the first problem, you will write a program to find the number of undecayed nuclei as a function of time using a simple numerical technique called *Euler's method*. In the second problem, you will use the more advanced (and far more accurate) *Runge-Kutta* method to find the velocity of a skydiver as a function of time before he opens his parachute. In a third problem, you will find the skydiver's velocity both before and after the parachute opens. **You should allow one session for each of the three problems.**

Numerical solution is especially useful for ODEs that we cannot solve analytically. However, the ODEs for the two systems in this exercise actually have analytical solutions. Hence, you can test the accuracy of the Euler and Runge-Kutta methods by comparing your numerical solutions with the analytical solutions.

Euler's Method

In differential calculus, the first derivative of a function $y(x)$ with respect to x is defined as

$$\frac{dy}{dx} = \underset{\delta x \to 0}{\text{Lim}}\left[\frac{y(x+\delta x) - y(x)}{\delta x}\right], \qquad (1)$$

where δx is a small interval, as illustrated in Figure 1 below. When the interval δx is sufficiently small, we may re-write equation (1) approximately as

$$y(x+\delta x) = y(x) + \left(\frac{dy}{dx}\right)\delta x\,. \qquad (2)$$

You should recognise this equation as the Taylor expansion for $y(x)$, in which we have neglected terms of order δx^2 and above on the right-hand side on the assumption that δx is sufficiently small.

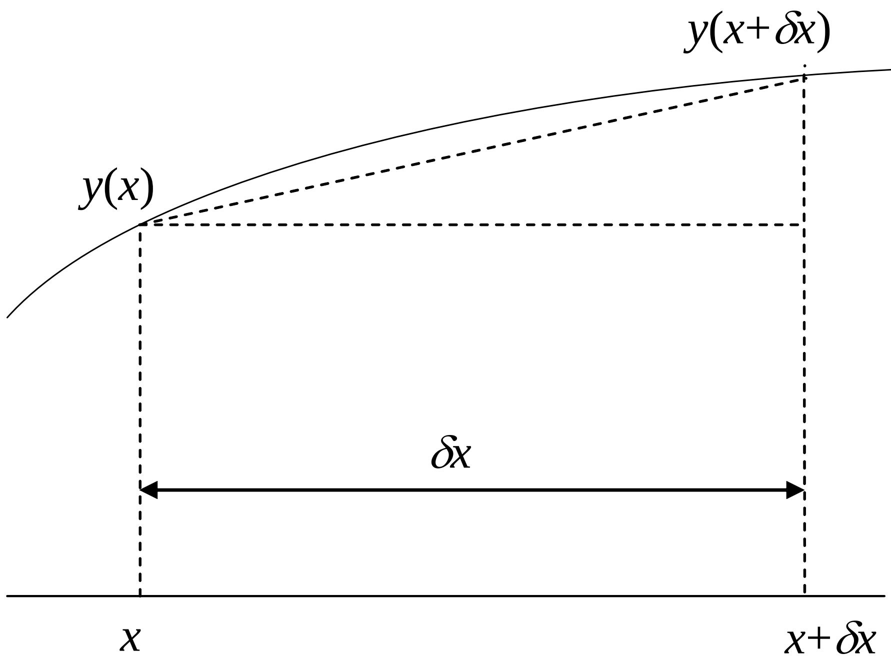

Figure 1

Suppose that the function $y(x)$ satisfies a first-order differential equation of the form

$$\frac{dy}{dx} = f(x,y), \tag{3}$$

where $f(x,y)$ is some function of x and y. This differential equation provides us with an expression for calculating the derivative dy/dx in equation (2).

If we start at some point x_0 where we know the initial value $y(x_0)$, then we can use equations (2) and (3) to estimate $y(x_0+\delta x)$ at a nearby point $x_0+\delta x$. We can then repeat this procedure time and again to produce a numerical solution for y as a function of x. This iterative procedure is the basis of Euler's method for the numerical solution of a first-order differential equation.

Problem 1: Radioactive Decay

Radioactive decay is modelled by the simple differential equation

$$\frac{dN}{dt} = -\lambda N, \tag{4}$$

where N is the average number of nuclei that remain at time t and λ is the decay constant.

We can solve equation (4) numerically using Euler's method. Instead of equation (2), we write

$$N(t+\delta t) = N(t) + \left(\frac{dN}{dt}\right)\delta t, \tag{5}$$

where δt is a small time interval. It then follows from equations (4) and (5) that

$$N(t+\delta t) = N(t) - \lambda N(t)\delta t = N(t)(1-\lambda\delta t). \tag{6}$$

Hence, if we know the number of undecayed nuclei $N(t)$ at time t (starting, say, at $t = 0$), we can use equation (6) to estimate the number $N(t + \delta t)$ at time $t + \delta t$. We then use $N(t + \delta t)$ on the right-hand side to estimate $N(t + 2\delta t)$; and so on.

We can translate the iteration formula in Equation (6) into C++ as

```
N = N * (1. - lambda * dt);                                                    (7)
```

where the double-precision variables `N`, `lambda` and `dt` represent *N*, λ and δt, respectively, in the theory. Notice that `N` appears on both sides of the equals sign in (7). This would make no sense as an equation (unless N = 0). However, you must remember that the equals sign in C++ actually indicates an *assignment statement*, not an equation. The assignment statement in (7) is an instruction to the computer to take the current value of `N` on the RHS, multiply it by `(1-lambda*dt)`, and then save the result as the new value of `N` on the LHS.

In the numerical solution, you can incorporate the iteration statement (7) inside a `for` loop to make the time steps, as in Exercise 1. In addition, Equation (4) has a simple analytical solution that you can compare with the numerical solution to assess the accuracy of the Euler method.

The problem—what's required

1. Write a C++ program to solve for *N* as a function of time from $t = 0$ to 100 s for time steps of 0.1 s using Euler's method taking the initial value of $N = 1000$ with $\lambda = 0.1\ \mathrm{s}^{-1}$. Repeat the calculations with time steps of 1 s and 5 s.
2. How do the results from Euler's method compare with the analytical solution?

Your solution—what's needed

1. Solve equation (4) analytically and include a clearly laid-out solution. **[2 marks]**
2. Include a clearly commented printout of your code. The marker must be able to follow the code easily and to understand its logic and function. **[5 marks]**
3. Plot your results (Euler and analytical solution). Briefly explain why the accuracy of the method depends on the size of the time step. **[3 marks]**

The Runge-Kutta Method

Euler's method demonstrates the basic principle for the numerical solution of an ordinary differential equation. It can handle problems where the derivative *dy/dx* changes slowly, so that Equation (2) is a good approximation. One might argue that the method should *always* work, provided that we make the step size δx small enough. However, a very small step size entails a large number of iterations, which slows down execution. Even worse, the large number of repeated floating-point additions can produce significant rounding errors that make the numerical solution totally unreliable.

For solving real problems, we use more accurate methods. The error in truncating the Taylor expansion in the Euler formula in equation (2) is of order δx^2 . Hence, it is generally more accurate to use a higher-order formula with a truncation error of order δx^4 or δx^5, such as the popular Runge-Kutta algorithm, as described in the authoritative book *Numerical Recipes* by Press *et al*. You can consult the *C* version of this book online at www.nrbook.com/a/bookcpdf.php - see Chapter 16. The Runge-Kutta method evaluates the derivative *dy/dx* at several points inside the interval δx to produce a far more accurate interpolation formula that can handle problems where *dy/dx* changes rapidly. (For the mathematical details, see *Numerical Recipes* and the book by Ralston & Rabinowitz.) Practical implementations also incorporate an adaptive step size that decreases (or increases) automatically in order to achieve a specified level of accuracy.

The sample project RungeKuttaDemo1 uses a simplified Runge-Kutta routine to integrate Equation (4) for the radioactive-decay problem. It contains the following global declarations:

```
inline double deriv(double t, double N);        //Function prototype for the derivative
void rk1(double& N, double t1, double t2);      //Function prototype for the RK algorithm
const double lambda=0.1;                        //Radioactive-decay constant (/s)
```

Points to note:

1. The `deriv()` function uses Equation (4) to calculate *dN*/*dt* with `t` and `N` as parameters Its function definition is `inline double deriv(double t, double N) {return -lambda*N;}`. To speed up execution, this short function is declared to be inline.
2. The radioactive-decay constant `lambda` is declared as a global constant so that we can use it in both `deriv()` and `main()`.
3. The Runge-Kutta routine `rk1()` integrates the `deriv()` function from time `t1` to `t2`. The ampersand in '`double& N`' indicates that a *reference* to `N` (rather than the actual *value* of `N`) is passed as the first argument to `rk1()`. When `rk1()` integrates the `deriv()` function from `t1` to `t2`, it can then change the value of `N` in the `main()` routine. (The reference declaration provides `rk1()` with the memory address of variable `N` in main() so that `rk1()` can change the value of `N`. Another way to do this would be to pass the address of `N` as a *pointer*, which is a type of variable that holds the address of another variable. However, a pointer can be changed to point to another part of memory, which can produce run-time errors. Since a reference cannot be changed, the code is intrinsically safer.)
4. The result of `rk1()` is to change the value of `N` in `main()` using call-by-reference. Hence, the `rk1()` function does not need to return a value, and its return type is `void`. The function call to `rk1()` from `main()` is therefore the following statement:

   ```
   rk1(N,t1,t2); //Calls rk1() to integrate deriv() from t1 to t2
   ```
5. The `rk1()` function is a simplified version of the `odeint()` Runge-Kutta routine in Chapter 16 of *Numerical Recipes in C*. It evaluates the derivative six times. That is why we put the derivative in the `deriv()` function so that it can be called repeatedly by `rk1()`. (Since `rk1()` and `deriv()` are compiled together, we can speed up execution by declaring `deriv()` to be `inline`.)
6. The adaptive step size in `rk1()` is held in a variable `h` that is declared as `static double h=t2-t1`. The `static` storage type means that the value of `h` is preserved between function calls (in contrast to normal 'automatic' variables, whose value is lost after each function call). The first call to `rk1()` initializes the step size to `t2-t1`. The routine estimates the truncation error for this step size and reduces `h` as necessary to achieve the required accuracy (set as 10^{-8} in the code). That provides the starting value of `h` for the next integration step. Hence, the routine adjusts itself very quickly to the correct step size.
7. Run the RungeKuttaDemo1 project to compare the numerical and analytical solutions. Even though the integration steps are large (10 steps in 100 s), the numerical solution is very accurate, in contrast to the simple Euler method in Problem 1.

Problem 2: Free-fall motion with air resistance

Consider the motion of a skydiver in free fall. To simplify the problem, we ignore any horizontal motion and suppose that the skydiver falls vertically from rest under the influence of gravity and air resistance. Assuming that the drag force from air resistance is proportional to the square of the vertical velocity *v* (a more accurate description raises it to the power 9/5), we may write the skydiver's equation of motion as

$$m\frac{dv}{dt} = mg - \alpha v^2, \qquad (8)$$

where *m* is the mass of the skydiver, *g* is the acceleration due to gravity, and α is a constant of proportionality. Re-arranging equation (8), we find

$$\frac{dv}{dt} = g - kv^2, \qquad (9)$$

where $k = \alpha/m$. We can solve this equation numerically using the Runge-Kutta algorithm `rk1()`, as in the RungeKuttaDemo1 project, which you can use as the starting point for coding your solution.

The problem—what's required

1. Write a C++ program to compute the vertical component of velocity v of a skydiver as a function of time t, starting with $v = 0$ at $t = 0$. The program should use the `rk1()` Runge-Kutta routine with equation (9) to calculate v. You may assume that the skydiver weighs 100 kg and that the drag coefficient $\alpha = 0.3$ kg m^{-1}.

2. Your program should compute v as a function of time at 200 points for the first 30 seconds of the motion. Compare your results with the analytical solution to equation (9), which is

$$v(t) = v_\infty \tanh\left(t/\tau\right), \qquad (10)$$

where $v_\infty = \sqrt{g/k}$ and $\tau = 1/kv_\infty$. Hence, your program will need to output both the numerical and the analytical solutions for each time t.

Your solution—what's needed

1. Include a clearly commented printout of your code with a brief description. The marker must be able to follow the code easily and to understand its logic and function. **[6 marks]**
2. Tabulate and plot the estimated velocity, together with the actual velocity from equation (10), as a function of time. **[2 marks]**
3. Explain the physical significance of v_∞ and τ in equation (10). **[2 marks]**

Problem 3: Opening the parachute

Suppose that, after 25 seconds of free fall, our intrepid skydiver actually opens the parachute. This immediately increases the drag coefficient α to 30 kg m^{-1}.

The problem—what's required

1. Modify the `deriv()` function in your C++ program to allow for the parachute being closed for $t < 25$ s and then open for $t \geq 25$ s. A simple way to do this is to use a conditional expression of the form:

```
        (condition) ? x1 : x2;
```

The value of this expression is equal to `x1` when `condition` is `true` and `x2` when it is `false`.

2. Use your revised C++ program to find the numerical solution for v as in Problem 2 at 200 points from $t = 0$ to 30 s. This period now includes the opening of the parachute at $t = 25$ s.

Your solution—what's needed

1. Include a clearly commented printout of your code with a brief description. The marker must be able to follow the code easily and to understand its logic and function. **[5 marks]**
2. Plot your results on an Excel chart with those from Problem 2 for comparison. **[2 marks]**
3. Compare your computed velocity at $t = 30$ s with the expected terminal velocity for the skydiver when the parachute is open. Approximately how long does is take the skydiver to reach this velocity? **[2 marks]**

Further reading

Ammeraal, L. (2000) *C++ for Programmers* (Wiley).
Press, W.H. *et al*. (1992) *Numerical Recipes in C*, Ch. 16. (Cambridge)
Ralston, A. & Rabinowitz, P. (1978) *A First Course in Numerical Analysis*, Ch. 5 (McGraw-Hill).

Neil Thomas
n.thomas@bham.ac.uk

More advanced techniques (optional and non-assessed)

The sample project RungeKuttaDemo2 on Canvas demonstrates a more powerful Runge-Kutta routine `rkint()` based on the `odeint()` routine in *Numerical Recipes*. Whereas `rk1()` integrates just a single first-order ODE, `rkint()` is designed to integrate a *system* of coupled first-order ODEs. Since a second-order ODE can be written as two coupled first-order ODEs, this means that `rkint()` can integrate second-order (and higher-order) ODEs. As an example, to integrate the equation of motion

$$m\frac{d^2x}{dt^2}=F(t), \tag{11}$$

we would re-write it as

$$\frac{dx}{dt}=v \quad \text{and} \quad \frac{dv}{dt}=\frac{F(t)}{m}. \tag{12}$$

There are now two variables in the problem, x and v. They can be held as elements `x[1]` and `x[2]` in a C++ *array*, whilst *dx/dt* and *dv/dt* are held in `dery[1]` and `dery[2]`. RungeKuttaDemo2 uses this method to integrate the equation of motion for a particle undergoing damped SHM.

Points to note:

1. The function prototype for the `derivs()` function (which now contains more than one derivative) is `void derivs(double t,double x[],double dery[])`. The entries '`double x[]`' and '`double dery[]`' mean that parameters `x` and `dery` are the *names* of arrays of type `double`, which hold the variables for the problem and their respective derivatives. In C and C++, the name of an array is a *pointer* to (*i.e.*, the memory address of) the first element in the array, which is always indexed as element 0. (Since the variables used in the Runge-Kutta demonstration are `x[1]` and `x[2]`, element `x[0]` is not actually being used in this particular case.)
2. The function call `derivs(t,pcopy,dery)` in `rkint()` passes the addresses of arrays `pcopy` and `dery` to the `derivs()` function. It can then read the values of the array elements and, crucially, it can change the values of the derivatives in the `dery` array. Passing an array name as a pointer is fast and convenient, but it is also potentially dangerous: there is nothing to stop us from assigning a value to an array element that is out of bounds, which causes a run-time error.
3. An alternative declaration for `derivs()` is `void derivs(double t,double* x,double* dery)`. Here, the notation '`double* x`' can be read backwards as '`x` is a pointer to a variable of type `double`'. In this case, that variable is `x[0]`, the first element in the array. However, in general, we can declare a pointer to *any* type of variable using the star notation.
4. Pointers are also used in *dynamic memory allocation*. We can create an array of `N` elements of type `double` with a statement of the form: `double* x = new double[N]`. This asks the operating system to allocate memory to the array at run-time from the 'heap' of free memory (which can be 1 GB or more on a modern PC). This is particularly useful for large arrays or when the number of elements is not known at compile time. Operator `new` returns a pointer to the address of the memory allocated, which in this case is the start address of the array. Before the program finishes, we must delete the dynamically allocated array with a statement of the form `delete[] x`. This frees up the memory for other programs to use. (To create and delete single objects, rather than arrays, omit the square brackets in the `new` and `delete` statements.)
5. Just as the name of an array is a pointer to its start address, the name of a function is actually a pointer to the start of the function's code. This allows us to pass function names as arguments. In the case of `rkint()`, passing the name of the `derivs()` function as a parameter allows us to use `rkint()` to solve more than one set of ODEs in the same program. The function prototype for `rkint()`, even without optional parameter names, is quite long: `void rkint(double[], int, double,double,double,double&,double,int&,int&, void(*derivs)(double,double[],double[]))`. The final parameter here specifies a *function pointer* to a function of return type `void` and with an argument list containing three parameters of types `double`, `double[]` and `double[]`.

You don't need to understand all of the above details at this stage. However, if you have time, you can try `rkint()`, out of interest, on the skydiver problem, where you can calculate both the skydiver's height *z* and his vertical velocity $v = -dz/dt$ as a function of time *t*.

The Runge-Kutta method works well for many problems. However, as discussed in *Numerical Recipes*, the solution of *stiff* (or *ill-conditioned*) equations, which contain coefficients of very different size, requires special techniques.