Slides PHY 480 and PHY 905 Lectures: Ordinary differential equations

Morten Hjorth-Jensen[1,2]

[1]Department of Physics, University of Oslo
[2]Department of Physics and Astronomy and National Superconducting Cyclotron Laboratory, Michigan State University

Spring 2016

Differential equations program

- Ordinary differential equations, Runge-Kutta method,chapter 8
- Ordinary differential equations with boundary conditions: one-variable equations to be solved by shooting and Green's function methods, chapter 9
- We can solve such equations by a finite difference scheme as well, turning the equation into an eigenvalue problem. Still one variable. Done in projects 1 and 2.
- If we have more than one variable, we need to solve partial differential equations, see Chapter 10

The material on differential equations is covered by chapters 8, 9 and 10. Two of the final projects deal with ordinary differential equations.

Differential Equations, chapter 8

The order of the ODE refers to the order of the derivative on the left-hand side in the equation

$$\frac{dy}{dt} = f(t, y). \tag{1}$$

This equation is of first order and f is an arbitrary function. A second-order equation goes typically like

$$\frac{d^2y}{dt^2} = f(t, \frac{dy}{dt}, y). \tag{2}$$

A well-known second-order equation is Newton's second law

$$m\frac{d^2x}{dt^2} = -kx, \tag{3}$$

where k is the force constant. ODE depend only on one variable

Differential Equations

partial differential equations like the time-dependent Schrödinger equation

$$i\hbar\frac{\partial\psi(\mathbf{x},t)}{\partial t} = -\frac{\hbar^2}{2m}\left(\frac{\partial^2\psi(\mathbf{r},t)}{\partial x^2} + \frac{\partial^2\psi(\mathbf{r},t)}{\partial y^2} + \frac{\partial^2\psi(\mathbf{r},t)}{\partial z^2}\right) + V(\mathbf{x})\psi(\mathbf{x},t), \tag{4}$$

may depend on several variables. In certain cases, like the above equation, the wave function can be factorized in functions of the separate variables, so that the Schroedinger equation can be rewritten in terms of sets of ordinary differential equations. These equations are discussed in chapter 10. Involve boundary conditions in addition to initial conditions.

Differential Equations

We distinguish also between linear and non-linear differential equation where for example

$$\frac{dy}{dt} = g^3(t)y(t), \tag{5}$$

is an example of a linear equation, while

$$\frac{dy}{dt} = g^3(t)y(t) - g(t)y^2(t), \tag{6}$$

is a non-linear ODE.

Differential Equations

Another concept which dictates the numerical method chosen for solving an ODE, is that of initial and boundary conditions. To give an example, if we study white dwarf stars or neutron stars we will need to solve two coupled first-order differential equations, one for the total mass m and one for the pressure P as functions of ρ

$$\frac{dm}{dr} = 4\pi r^2\rho(r)/c^2,$$

and

$$\frac{dP}{dr} = -\frac{Gm(r)}{r^2}\rho(r)/c^2.$$

where ρ is the mass-energy density. The initial conditions are dictated by the mass being zero at the center of the star, i.e., when $r = 0$, yielding $m(r = 0) = 0$. The other condition is that the pressure vanishes at the surface of the star.

In the solution of the Schroedinger equation for a particle in a potential, we may need to apply boundary conditions as well, such as demanding continuity of the wave function and its derivative.

Differential Equations

In many cases it is possible to rewrite a second-order differential equation in terms of two first-order differential equations. Consider again the case of Newton's second law in Eq. (3). If we define the position $x(t) = y^{(1)}(t)$ and the velocity $v(t) = y^{(2)}(t)$ as its derivative

$$\frac{dy^{(1)}(t)}{dt} = \frac{dx(t)}{dt} = y^{(2)}(t), \tag{7}$$

we can rewrite Newton's second law as two coupled first-order differential equations

$$m\frac{dy^{(2)}(t)}{dt} = -kx(t) = -ky^{(1)}(t), \tag{8}$$

and

$$\frac{dy^{(1)}(t)}{dt} = y^{(2)}(t). \tag{9}$$

Differential Equations, Finite Difference

These methods fall under the general class of one-step methods. The algoritm is rather simple. Suppose we have an initial value for the function $y(t)$ given by

$$y_0 = y(t = t_0). \tag{10}$$

We are interested in solving a differential equation in a region in space $[a, b]$. We define a step h by splitting the interval in N sub intervals, so that we have

$$h = \frac{b-a}{N}. \tag{11}$$

With this step and the derivative of y we can construct the next value of the function y at

$$y_1 = y(t_1 = t_0 + h), \tag{12}$$

and so forth.

Differential Equations

If the function is rather well-behaved in the domain $[a, b]$, we can use a fixed step size. If not, adaptive steps may be needed. Here we concentrate on fixed-step methods only. Let us try to generalize the above procedure by writing the step y_{i+1} in terms of the previous step y_i

$$y_{i+1} = y(t = t_i + h) = y(t_i) + h\Delta(t_i, y_i(t_i)) + O(h^{p+1}), \tag{13}$$

where $O(h^{p+1})$ represents the truncation error. To determine Δ, we Taylor expand our function y

$$y_{i+1} = y(t = t_i + h) = y(t_i) + h(y'(t_i) + \cdots + y^{(p)}(t_i)\frac{h^{p-1}}{p!}) + O(h^{p+1}), \tag{14}$$

where we will associate the derivatives in the parenthesis with

$$\Delta(t_i, y_i(t_i)) = (y'(t_i) + \cdots + y^{(p)}(t_i)\frac{h^{p-1}}{p!}). \tag{15}$$

Differential Equations

We define

$$y'(t_i) = f(t_i, y_i) \tag{16}$$

and if we truncate Δ at the first derivative, we have

$$y_{i+1} = y(t_i) + hf(t_i, y_i) + O(h^2), \tag{17}$$

which when complemented with $t_{i+1} = t_i + h$ forms the algorithm for the well-known Euler method. Note that at every step we make an approximation error of the order of $O(h^2)$, however the total error is the sum over all steps $N = (b-a)/h$, yielding thus a global error which goes like $NO(h^2) \approx O(h)$.

Differential Equations

To make Euler's method more precise we can obviously decrease h (increase N). However, if we are computing the derivative f numerically by for example the two-steps formula

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

we can enter into roundoff error problems when we subtract two almost equal numbers $f(x+h) - f(x) \approx 0$. Euler's method is not recommended for precision calculation, although it is handy to use in order to get a first view on how a solution may look like. As an example, consider Newton's equation rewritten in Eqs. (8) and (9). We define $y_0 = y^{(1)}(t=0)$ an $v_0 = y^{(2)}(t=0)$. The first steps in Newton's equations are then

$$y_1^{(1)} = y_0 + hv_0 + O(h^2) \tag{18}$$

and

$$y_1^{(2)} = v_0 - hy_0 k/m + O(h^2). \tag{19}$$

Differential Equations

The Euler method is asymmetric in time, since it uses information about the derivative at the beginning of the time interval. This means that we evaluate the position at $y_1^{(1)}$ using the velocity at $y_0^{(2)} = v_0$. A simple variation is to determine $y_{n+1}^{(1)}$ using the velocity at $y_{n+1}^{(2)}$, that is (in a slightly more generalized form)

$$y_{n+1}^{(1)} = y_n^{(1)} + hy_{n+1}^{(2)} + O(h^2) \tag{20}$$

and

$$y_{n+1}^{(2)} = y_n^{(2)} + ha_n + O(h^2). \tag{21}$$

The acceleration a_n is a function of $a_n(y_n^{(1)}, y_n^{(2)}, t)$ and needs to be evaluated as well. This is the Euler-Cromer method.

Differential Equations

Let us then include the second derivative in our Taylor expansion. We have then

$$\Delta(t_i, y_i(t_i)) = f(t_i) + \frac{h}{2}\frac{df(t_i, y_i)}{dt} + O(h^3). \tag{22}$$

The second derivative can be rewritten as

$$y'' = f' = \frac{df}{dt} = \frac{\partial f}{\partial t} + \frac{\partial f}{\partial y}\frac{\partial y}{\partial t} = \frac{\partial f}{\partial t} + \frac{\partial f}{\partial y}f \tag{23}$$

and we can rewrite Eq. (14) as

$$y_{i+1} = y(t = t_i + h) = y(t_i) + hf(t_i) + \frac{h^2}{2}\left(\frac{\partial f}{\partial t} + \frac{\partial f}{\partial y}f\right) + O(h^3), \tag{24}$$

which has a local approximation error $O(h^3)$ and a global error $O(h^2)$.

Differential Equations

These approximations can be generalized by using the derivative f to arbitrary order so that we have

$$y_{i+1} = y(t = t_i+h) = y(t_i)+h(f(t_i, y_i)+\ldots f^{(p-1)}(t_i, y_i)\frac{h^{p-1}}{p!})+O(h^{p+1}). \tag{25}$$

These methods, based on higher-order derivatives, are in general not used in numerical computation, since they rely on evaluating derivatives several times. Unless one has analytical expressions for these, the risk of roundoff errors is large.

Differential Equations

The most obvious improvements to Euler's and Euler-Cromer's algorithms, avoiding in addition the need for computing a second derivative, is the so-called midpoint method. We have then

$$y_{n+1}^{(1)} = y_n^{(1)} + \frac{h}{2}\left(y_{n+1}^{(2)} + y_n^{(2)}\right) + O(h^2) \tag{26}$$

and

$$y_{n+1}^{(2)} = y_n^{(2)} + ha_n + O(h^2), \tag{27}$$

yielding

$$y_{n+1}^{(1)} = y_n^{(1)} + hy_n^{(2)} + \frac{h^2}{2}a_n + O(h^3) \tag{28}$$

implying that the local truncation error in the position is now $O(h^3)$, whereas Euler's or Euler-Cromer's methods have a local error of $O(h^2)$.

Differential Equations

Thus, the midpoint method yields a global error with second-order accuracy for the position and first-order accuracy for the velocity. However, although these methods yield exact results for constant accelerations, the error increases in general with each time step.

One method that avoids this is the so-called half-step method. Here we define

$$y_{n+1/2}^{(2)} = y_{n-1/2}^{(2)} + ha_n + O(h^2), \tag{29}$$

and

$$y_{n+1}^{(1)} = y_n^{(1)} + hy_{n+1/2}^{(2)} + O(h^2). \tag{30}$$

Note that this method needs the calculation of $y_{1/2}^{(2)}$. This is done using e.g., Euler's method

$$y_{1/2}^{(2)} = y_0^{(2)} + ha_0 + O(h^2). \tag{31}$$

As this method is numerically stable, it is often used instead of Euler's method.

Differential Equations

Another method which one may encounter is the Euler-Richardson method with

$$y_{n+1}^{(2)} = y_n^{(2)} + ha_{n+1/2} + O(h^2), \tag{32}$$

and

$$y_{n+1}^{(1)} = y_n^{(1)} + hy_{n+1/2}^{(2)} + O(h^2). \tag{33}$$

The Verlet method

Another set of popular algorithms, which are both numerically stable and easy to implement are the Verlet algorithms, with the velocity Verlet method as widely used in for example Molecular dynamics calculations. Consider again a second-order differential equation like Newton's second law, whose one-dimensional version reads

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

which we rewrite in terms of two coupled differential equations

$$\frac{dx}{dt} = v(x,t) \qquad \text{and} \qquad \frac{dv}{dt} = F(x,t)/m = a(x,t).$$

The Verlet method

If we now perform a Taylor expansion

$$x(t+h) = x(t) + hx^{(1)}(t) + \frac{h^2}{2}x^{(2)}(t) + O(h^3).$$

In our case the second derivative is known via Newton's second law, namely $x^{(2)}(t) = a(x,t)$. If we add to the above equation the corresponding Taylor expansion for $x(t-h)$, we obtain, using the discretized expressions

$$x(t_i \pm h) = x_{i\pm 1} \qquad \text{and} \qquad x_i = x(t_i),$$

we arrive at

$$x_{i+1} = 2x_i - x_{i-1} + h^2 x_i^{(2)} + O(h^4).$$

The velocity Verlet method

We note that the truncation error goes like $O(h^4)$ since all the odd terms cancel when we add the two Taylor expansions. We see also that the velocity is not directly included in the equation since the function $x^{(2)} = a(x,t)$ is supposed to be known. If we need the velocity however, we can compute it using the well-known formula

$$x_i^{(1)} = \frac{x_{i+1} - x_{i-1}}{2h} + O(h^2).$$

We note that the velocity has a truncation error which goes like $O(h^2)$. In for example so-called Molecular dynamics calculations, since the acceleration is normally known via Newton's second law, there is seldomly a need for computing the velocity.

We note also that the algorithm for the position is not self-starting since, for $i = 0$ it depends on the value of x at the fictitious value x_{-1}.

We can amend this by introducing the velocity Verlet method.

The velocity Verlet method

We have the Taylor expansion for the position given by

$$x_{i+1} = x_i + hx_i^{(1)} + \frac{h^2}{2}x_i^{(2)} + O(h^3).$$

The corresponding expansion for the velocity is

$$v_{i+1} = v_i + hv_i^{(1)} + \frac{h^2}{2}v_i^{(2)} + O(h^3).$$

Via Newton's second law we have normally an analytical expression for the derivative of the velocity, namely

$$v_i^{(1)} = \frac{d^2x}{dt^2}|_i = \frac{F(x_i,t_i)}{m},$$

The velocity Verlet method

If we add to this the corresponding expansion for the derivative of the velocity

$$v_{i+1}^{(1)} = v_i^{(1)} + hv_i^{(2)} + O(h^2),$$

and retain only terms up to the second derivative of the velocity since our error goes as $O(h^3)$, we have

$$hv_i^{(2)} \approx v_{i+1}^{(1)} - v_i^{(1)}.$$

We can then rewrite the Taylor expansion for the velocity as

$$v_{i+1} = v_i + \frac{h}{2}\left(v_{i+1}^{(1)} + v_i^{(1)}\right) + O(h^3).$$

The velocity Verlet method

Our final equations for the position and the velocity become then

$$x_{i+1} = x_i + hv_i + \frac{h^2}{2}v_i^{(1)} + O(h^3),$$

and

$$v_{i+1} = v_i + \frac{h}{2}\left(v_{i+1}^{(1)} + v_i^{(1)}\right) + O(h^3).$$

Note well that the term $v_{i+1}^{(1)}$ depends on the position at x_{i+1}. This means that you need to calculate the position at the updated time t_{i+1} before the computing the next velocity. Note also that the derivative of the velocity at the time t_i used in the updating of the position can be reused in the calculation of the velocity update as well.

Differential Equations, Runge-Kutta methods

Runge-Kutta (RK) methods are based on Taylor expansion formulae, but yield in general better algorithms for solutions of an ODE. The basic philosophy is that it provides an intermediate step in the computation of y_{i+1}.

To see this, consider first the following definitions

$$\frac{dy}{dt} = f(t, y), \tag{34}$$

and

$$y(t) = \int f(t, y)dt, \tag{35}$$

and

$$y_{i+1} = y_i + \int_{t_i}^{t_{i+1}} f(t, y)dt. \tag{36}$$

Differential Equations, Runge-Kutta methods

To demonstrate the philosophy behind RK methods, let us consider the second-order RK method, RK2. The first approximation consists in Taylor expanding $f(t,y)$ around the center of the integration interval t_i to t_{i+1}, that is, at $t_i+h/2$, h being the step. Using the midpoint formula for an integral, defining $y(t_i+h/2)=y_{i+1/2}$ and $t_i+h/2=t_{i+1/2}$, we obtain

$$\int_{t_i}^{t_{i+1}} f(t,y)dt \approx hf(t_{i+1/2},y_{i+1/2})+O(h^3). \tag{37}$$

This means in turn that we have

$$y_{i+1}=y_i+hf(t_{i+1/2},y_{i+1/2})+O(h^3). \tag{38}$$

Differential Equations, Runge-Kutta methods

However, we do not know the value of $y_{i+1/2}$. Here comes thus the next approximation, namely, we use Euler's method to approximate $y_{i+1/2}$. We have then

$$y_{(i+1/2)}=y_i+\frac{h}{2}\frac{dy}{dt}=y(t_i)+\frac{h}{2}f(t_i,y_i). \tag{39}$$

This means that we can define the following algorithm for the second-order Runge-Kutta method, RK2.

$$k_1=hf(t_i,y_i), \tag{40}$$

$$k_2=hf(t_{i+1/2},y_i+k_1/2), \tag{41}$$

with the final value

$$y_{i+i}\approx y_i+k_2+O(h^3). \tag{42}$$

The difference between the previous one-step methods is that we now need an intermediate step in our evaluation, namely $t_i+h/2=t_{(i+1/2)}$ where we evaluate the derivative f. This involves more operations, but the gain is a better stability in the solution.

Differential Equations, Runge-Kutta methods

The fourth-order Runge-Kutta, RK4, which we will employ in the solution of various differential equations below, has the following algorithm

$$k_1=hf(t_i,y_i) \quad k_2=hf(t_i+h/2,y_i+k_1/2)$$

$$k_3=hf(t_i+h/2,y_i+k_2/2) \quad k_4=hf(t_i+h,y_i+k_3)$$

with the final result

$$y_{i+1}=y_i+\frac{1}{6}\left(k_1+2k_2+2k_3+k_4\right).$$

Thus, the algorithm consists in first calculating k_1 with t_i, y_1 and f as inputs. Thereafter, we increase the step size by $h/2$ and calculate k_2, then k_3 and finally k_4. The global error goes as $O(h^4)$.

Building a code for the solar system, gravitational force and constants

We start with a simpler case first, the Earth-Sun system in two dimensions only. The gravitational force F_G is

$$F = \frac{GM_\odot M_E}{r^2},$$

where G is the gravitational constant,

$$M_E = 6 \times 10^{24}\text{Kg},$$

the mass of Earth,

$$M_\odot = 2 \times 10^{30}\text{Kg},$$

the mass of the Sun and

$$r = 1.5 \times 10^{11}\text{m},$$

is the distance between Earth and the Sun. The latter defines what we call an astronomical unit **AU**. From Newton's second law we have then for the x direction

$$\frac{d^2x}{dt^2} = \frac{F_x}{M_E},$$

and

$$\frac{d^2y}{dt^2} = \frac{F_y}{M_E},$$

for the y direction.

Building a code for the solar system, force equations

Introducing $x = r\cos(\theta)$, $y = r\sin(\theta)$ and

$$r = \sqrt{x^2 + y^2},$$

we can rewrite

$$F_x = -\frac{GM_\odot M_E}{r^2}\cos(\theta) = -\frac{GM_\odot M_E}{r^3}x,$$

and

$$F_y = -\frac{GM_\odot M_E}{r^2}\sin(\theta) = -\frac{GM_\odot M_E}{r^3}y,$$

for the y direction.

Building a code for the solar system, coupled equations

We can rewrite these two equations

$$F_x = -\frac{GM_\odot M_E}{r^2}\cos(\theta) = -\frac{GM_\odot M_E}{r^3}x,$$

and

$$F_y = -\frac{GM_\odot M_E}{r^2}\sin(\theta) = -\frac{GM_\odot M_E}{r^3}y,$$

as four first-order coupled differential equations

$$\frac{dv_x}{dt} = -\frac{GM_\odot}{r^3}x,$$

$$\frac{dx}{dt} = v_x,$$

$$\frac{dv_y}{dt} = -\frac{GM_\odot}{r^3}y,$$

$$\frac{dy}{dt} = v_y.$$

Building a code for the solar system, final coupled equations

The four coupled differential equations

$$\frac{dv_x}{dt} = -\frac{GM_\odot}{r^3}x,$$

$$\frac{dx}{dt} = v_x,$$

$$\frac{dv_y}{dt} = -\frac{GM_\odot}{r^3}y,$$

$$\frac{dy}{dt} = v_y,$$

can be turned into dimensionless equations (as we did in project 2) or we can introduce astronomical units with 1 AU $= 1.5 \times 10^{11}$.

Using the equations from circular motion (with $r = 1$AU)

$$\frac{M_E v^2}{r} = F = \frac{GM_\odot M_E}{r^2},$$

we have

$$GM_\odot = v^2 r,$$

and using that the velocity of Earth (assuming circular motion) is $v = 2\pi r/\text{yr} = 2\pi\text{AU/yr}$, we have

$$GM_\odot = v^2 r = 4\pi^2 \frac{(\text{AU})^3}{\text{yr}^2}.$$

Building a code for the solar system, discretized equations

The four coupled differential equations can then be discretized using Euler's method as (with step length h)

$$v_{x,i+1} = v_{x,i} - h\frac{4\pi^2}{r_i^3}x_i,$$

$$x_{i+1} = x_i + hv_{x,i},$$

$$v_{y,i+1} = v_{y,i} - h\frac{4\pi^2}{r_i^3}y_i,$$

$$y_{i+1} = y_i + hv_{y,i},$$

Building a code for the solar system, adding Jupiter

It is rather straightforward to add a new planet, say Jupiter. Jupiter has mass

$$M_J = 1.9 \times 10^{27}\text{kg},$$

and distance to the Sun of 5.2 AU. The additional gravitational force the Earth feels from Jupiter in the x-direction is

$$F_x^{EJ} = -\frac{GM_JM_E}{r_{EJ}^3}(x_E - x_J),$$

where E stands for Earth, J for Jupiter, r_{EJ} is distance between Earth and Jupiter

$$r_{EJ} = \sqrt{(x_E - x_J)^2 + (y_E - y_J)^2},$$

and x_E and y_E are the x and y coordinates of Earth, respectively, and x_J and y_J are the x and y coordinates of Jupiter, respectively. The x-component of the velocity of Earth changes thus to

$$\frac{dv_x^E}{dt} = -\frac{GM_\odot}{r^3}x_E - \frac{GM_J}{r_{EJ}^3}(x_E - x_J).$$

Building a code for the solar system, adding Jupiter

We can rewrite

$$\frac{dv_x^E}{dt} = -\frac{GM_\odot}{r^3}x_E - \frac{GM_J}{r_{EJ}^3}(x_E - x_J).$$

to

$$\frac{dv_x^E}{dt} = -\frac{4\pi^2}{r^3}x_E - \frac{4\pi^2M_J/M_\odot}{r_{EJ}^3}(x_E - x_J),$$

where we used

$$GM_J = GM_\odot\left(\frac{M_J}{M_\odot}\right) = 4\pi^2\frac{M_J}{M_\odot}.$$

Similarly, for the velocity in y-direction we have

$$\frac{dv_y^E}{dt} = -\frac{4\pi^2}{r^3}y_E - \frac{4\pi^2 M_J/M_\odot}{r_{EJ}^3}(y_E - y_J).$$

Similar expressions apply for Jupiter. The equations for x and y derivatives are unchanged. This equations are similar for all other planets and as we will see later, it will be convenient to object orient this part when we program the full solar system.

How can I get the initial velocities and positions of the planets

NASA has an excellent site at `http://www.nasa.gov/index.html`. From there you can extract initial conditions in order to start your differential equation solver.

For the first simple system involving the Earth and the Sun, you could just initialize the position with say $x = 1$ AU and $y = 0$ AU.

Simple Example, Block tied to a Wall

Our first example is the classical case of simple harmonic oscillations, namely a block sliding on a horizontal frictionless surface. The block is tied to a wall with a spring. If the spring is not compressed or stretched too far, the force on the block at a given position x is

$$F = -kx.$$

The negative sign means that the force acts to restore the object to an equilibrium position. Newton's equation of motion for this idealized system is then

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

or we could rephrase it as

$$\frac{d^2x}{dt^2} = -\frac{k}{m}x = -\omega_0^2 x,$$

with the angular frequency $\omega_0^2 = k/m$.

The above differential equation has the advantage that it can be solved analytically with solutions on the form

$$x(t) = Acos(\omega_0 t + \nu),$$

where A is the amplitude and ν the phase constant. This provides in turn an important test for the numerical solution and the development of a program for more complicated cases which cannot be solved analytically.

Simple Example, Block tied to a Wall

With the position $x(t)$ and the velocity $v(t) = dx/dt$ we can reformulate Newton's equation in the following way

$$\frac{dx(t)}{dt} = v(t),$$

and

$$\frac{dv(t)}{dt} = -\omega_0^2 x(t).$$

We are now going to solve these equations using the Runge-Kutta method to fourth order discussed previously.

Simple Example, Block tied to a Wall

Before proceeding however, it is important to note that in addition to the exact solution, we have at least two further tests which can be used to check our solution.

Since functions like *cos* are periodic with a period 2π, then the solution $x(t)$ has also to be periodic. This means that

$$x(t+T) = x(t),$$

with T the period defined as

$$T = \frac{2\pi}{\omega_0} = \frac{2\pi}{\sqrt{k/m}}.$$

Observe that T depends only on k/m and not on the amplitude of the solution.

Simple Example, Block tied to a Wall

In addition to the periodicity test, the total energy has also to be conserved.

Suppose we choose the initial conditions

$$x(t=0) = 1 \text{ m} \qquad v(t=0) = 0 \text{ m/s},$$

meaning that block is at rest at $t = 0$ but with a potential energy

$$E_0 = \frac{1}{2}kx(t=0)^2 = \frac{1}{2}k.$$

The total energy at any time t has however to be conserved, meaning that our solution has to fulfil the condition

$$E_0 = \frac{1}{2}kx(t)^2 + \frac{1}{2}mv(t)^2.$$

Simple Example, Block tied to a Wall

An algorithm which implements these equations is included below.

- Choose the initial position and speed, with the most common choice $v(t=0)=0$ and some fixed value for the position.
- Choose the method you wish to employ in solving the problem.
- Subdivide the time interval $[t_i, t_f]$ into a grid with step size

$$h = \frac{t_f - t_i}{N},$$

where N is the number of mesh points.

- Calculate now the total energy given by

$$E_0 = \frac{1}{2}kx(t=0)^2 = \frac{1}{2}k.$$

- The Runge-Kutta method is used to obtain x_{i+1} and v_{i+1} starting from the previous values x_i and v_i.
- When we have computed $x(v)_{i+1}$ we upgrade $t_{i+1} = t_i + h$.
- This iterative process continues till we reach the maximum time t_f.
- The results are checked against the exact solution. Furthermore, one has to check the stability of the numerical solution against the chosen number of mesh points N.

Simple Example, Block tied to a Wall

To run a c++ program using ipython notebook, you can use the following statements.

```
%%install_ext https://raw.github.com/dragly/cppmagic/master/cppmagic.py
%%load_ext cppmagic

%%cpp
/*     This program solves Newton's equation for a block
       sliding on a horizontal frictionless surface. The block
       is tied  to a wall with a spring, and Newton's equation
       takes the form
            m d^2x/dt^2 =-kx
       with k the spring tension and m the mass of the block.
       The angular frequency is omega^2 = k/m and we set it equal
       1 in this example program.

       Newton's equation is rewritten as two coupled differential
       equations, one for the position x  and one for the velocity v
            dx/dt = v    and
            dv/dt = -x   when we set k/m=1

       We use therefore a two-dimensional array to represent x and v
```

```
      as functions of t
      y[0] == x
      y[1] == v
      dy[0]/dt = v
      dy[1]/dt = -x

      The derivatives are calculated by the user defined function
      derivatives.

      The user has to specify the initial velocity (usually v_0=0)
      the number of steps and the initial position. In the programme
      below we fix the time interval [a,b] to [0,2*pi].

*/
#include <cmath>
#include <iostream>
#include <fstream>
#include <iomanip>
//#include "lib.h"
using namespace  std;
// output file as global variable
ofstream ofile;
// function declarations
void derivatives(double, double *, double *);
void initialise ( double&, double&, int&);
void output( double, double *, double);
void runge_kutta_4(double *, double *, int, double, double,
                   double *, void (*)(double, double *, double *));

int main(int argc, char* argv[])
{
//  declarations of variables
  double *y, *dydt, *yout, t, h, tmax, E0;
  double initial_x, initial_v;
  int i, number_of_steps, n;
  char *outfilename;
  // Read in output file, abort if there are too few command-line arguments
  if( argc <= 1 ){
    cout << "Bad Usage: " << argv[0] <<
      " read also output file on same line" << endl;
    //    exit(1);
  }
  else{
    outfilename=argv[1];
  }
  ofile.open(outfilename);
  //  this is the number of differential equations
  n = 2;
  //  allocate space in memory for the arrays containing the derivatives
  dydt = new double[n];
  y = new double[n];
  yout = new double[n];
  // read in the initial position, velocity and number of steps
  initialise (initial_x, initial_v, number_of_steps);
  //  setting initial values, step size and max time tmax
  h = 4.*acos(-1.)/( (double) number_of_steps);   // the step size
  tmax = h*number_of_steps;               // the final time
  y[0] = initial_x;                       // initial position
  y[1] = initial_v;                       // initial velocity
  t=0.;                                   // initial time
  E0 = 0.5*y[0]*y[0]+0.5*y[1]*y[1];       // the initial total energy
```

```
  // now we start solving the differential equations using the RK4 method
  while (t <= tmax){
    derivatives(t, y, dydt);   // initial derivatives
    runge_kutta_4(y, dydt, n, t, h, yout, derivatives);
    for (i = 0; i < n; i++) {
      y[i] = yout[i];
    }
    t += h;
    output(t, y, E0);   // write to file
  }
  delete [] y; delete [] dydt; delete [] yout;
  ofile.close();  // close output file
  return 0;
}   //  End of main function

//     Read in from screen the number of steps,
//     initial position and initial speed
void initialise (double& initial_x, double& initial_v, int& number_of_steps)
{
 cout << "Initial position = ";
 cin >> initial_x;
 cout << "Initial speed = ";
 cin >> initial_v;
 cout << "Number of steps = ";
 cin >> number_of_steps;
}  // end of function initialise

//   this function sets up the derivatives for this special case
void derivatives(double t, double *y, double *dydt)
{
  dydt[0]=y[1];    // derivative of x
  dydt[1]=-y[0]; // derivative of v
} // end of function derivatives

//    function to write out the final results
void output(double t, double *y, double E0)
{
  ofile << setiosflags(ios::showpoint | ios::uppercase);
  Ofile << setw(15) << setprecision(8) << t;
  ofile << setw(15) << setprecision(8) << y[0];
  ofile << setw(15) << setprecision(8) << y[1];
  ofile << setw(15) << setprecision(8) << cos(t);
  ofile << setw(15) << setprecision(8) <<
    0.5*y[0]*y[0]+0.5*y[1]*y[1]-E0 << endl;
}  // end of function output

/*   This function upgrades a function y (input as a pointer)
     and returns the result yout, also as a pointer. Note that
     these variables are declared as arrays.  It also receives as
     input the starting value for the derivatives in the pointer
     dydx. It receives also the variable n which represents the
     number of differential equations, the step size h and
     the initial value of x. It receives also the name of the
     function *derivs where the given derivative is computed
*/
void runge_kutta_4(double *y, double *dydx, int n, double x, double h,
                   double *yout, void (*derivs)(double, double *, double *))
{
  int i;
  double      xh,hh,h6;
  double *dym, *dyt, *yt;
```

```
  //   allocate space for local vectors
  dym = new double [n];
  dyt =  new double [n];
  yt =  new double [n];
  hh = h*0.5;
  h6 = h/6.;
  xh = x+hh;
  for (i = 0; i < n; i++) {
    yt[i] = y[i]+hh*dydx[i];
  }
  (*derivs)(xh,yt,dyt);     // computation of k2, eq. 3.60
  for (i = 0; i < n; i++) {
    yt[i] = y[i]+hh*dyt[i];
  }
  (*derivs)(xh,yt,dym); //  computation of k3, eq. 3.61
  for (i=0; i < n; i++) {
    yt[i] = y[i]+h*dym[i];
    dym[i] += dyt[i];
  }
  (*derivs)(x+h,yt,dyt);    // computation of k4, eq. 3.62
  //      now we upgrade y in the array yout
  for (i = 0; i < n; i++){
    yout[i] = y[i]+h6*(dydx[i]+dyt[i]+2.0*dym[i]);
  }
  delete []dym;
  delete [] dyt;
  delete [] yt;
}       //  end of function Runge-kutta 4
```

Simple Example, Block tied to a Wall, python code

The following python program performs essentially the same calculations as the previous c++ code.

```
#
# This program solves Newtons equation for a block sliding on
# an horizontal frictionless surface.
# The block is tied to the wall with a spring, so N's eq takes the form:
#
#  m d^2x/dt^2 = - kx
#
# In order to make the solution dimless, we set k/m = 1.
# This results in two coupled diff. eq's that may be written as:
#
#  dx/dt = v
#  dv/dt = -x
#
# The user has to specify the initial velocity and position,
# and the number of steps. The time interval is fixed to
# t \in [0, 4\pi) (two periods)
#
# Note that this is a highly simplifyed rk4 code, intended
# for conceptual understanding and experimentation.

import sys
import numpy, math

#Global variables
ofile = None;
```

```
E0    = 0.0

def sim(x_0, v_0, N):
    ts = 0.0
    te = 4*math.pi
    h = (te-ts)/float(N)

    t = ts;
    x = x_0
    v = v_0
    while (t < te):
        kv1 = -h*x
        kx1 = h*v

        kv2 = -h*(x+kx1/2)
        kx2 =  h*(v+kv1/2)

        kv3 = -h*(x+kx2/2)
        kx3 =  h*(v+kv2/2)

        kv4 = -h*(x+kx3/2)
        kx4 =  h*(v+kv3/2)

        #Write the old values to file
        output(t,x,v)

        #Update
        x = x + (kx1 + 2*(kx2+kx3) + kx4)/6
        v = v + (kv1 + 2*(kv2+kv3) + kv4)/6
        t = t+h

def output(t,x,v):
    de = 0.5*x**2+0.5*v**2 - E0;
    ofile.write("%15.8E %15.8E %15.8E %15.8E %15.8E\n"\
                %(t, x, v, math.cos(t),de));


#MAIN PROGRAM:

#Get input
if len(sys.argv) == 5:
    ofilename =       sys.argv[1];
    x_0       = float(sys.argv[2])
    v_0       = float(sys.argv[3])
    N         =   int(sys.argv[4])
else:
    print "Usage:", sys.argv[0], "ofilename x0 v0 N"
    sys.exit(0)

#Setup
ofile = open(ofilename, 'w')
E0    = 0.5*x_0**2+0.5*v_0**2

#Run simulation
sim(x_0,v_0,N)

#Cleanup
ofile.close()
```

The classical pendulum

The angular equation of motion of the pendulum is given by Newton's equation and with no external force it reads

$$ml\frac{d^2\theta}{dt^2} + mgsin(\theta) = 0, \tag{43}$$

with an angular velocity and acceleration given by

$$v = l\frac{d\theta}{dt}, \tag{44}$$

and

$$a = l\frac{d^2\theta}{dt^2}. \tag{45}$$

More on the Pendulum

We do however expect that the motion will gradually come to an end due a viscous drag torque acting on the pendulum. In the presence of the drag, the above equation becomes

$$ml\frac{d^2\theta}{dt^2} + \nu\frac{d\theta}{dt} + mgsin(\theta) = 0, \tag{46}$$

where ν is now a positive constant parameterizing the viscosity of the medium in question. In order to maintain the motion against viscosity, it is necessary to add some external driving force. We choose here a periodic driving force. The last equation becomes then

$$ml\frac{d^2\theta}{dt^2} + \nu\frac{d\theta}{dt} + mgsin(\theta) = Asin(\omega t), \tag{47}$$

with A and ω two constants representing the amplitude and the angular frequency respectively. The latter is called the driving frequency.

More on the Pendulum

We define

$$\omega_0 = \sqrt{g/l},$$

the so-called natural frequency and the new dimensionless quantities

$$\hat{t} = \omega_0 t,$$

with the dimensionless driving frequency

$$\hat{\omega} = \frac{\omega}{\omega_0},$$

and introducing the quantity Q, called the *quality factor*,

$$Q = \frac{mg}{\omega_0 \nu},$$

and the dimensionless amplitude

$$\hat{A} = \frac{A}{mg}$$

More on the Pendulum

We have

$$\frac{d^2\theta}{d\hat{t}^2} + \frac{1}{Q}\frac{d\theta}{d\hat{t}} + sin(\theta) = \hat{A}cos(\hat{\omega}\hat{t}).$$

This equation can in turn be recast in terms of two coupled first-order differential equations as follows

$$\frac{d\theta}{d\hat{t}} = \hat{v},$$

and

$$\frac{d\hat{v}}{d\hat{t}} = -\frac{\hat{v}}{Q} - sin(\theta) + \hat{A}cos(\hat{\omega}\hat{t}).$$

These are the equations to be solved. The factor Q represents the number of oscillations of the undriven system that must occur before its energy is significantly reduced due to the viscous drag. The amplitude $\hat{A}$ is measured in units of the maximum possible gravitational torque while $\hat{\omega}$ is the angular frequency of the external torque measured in units of the pendulum's natural frequency.

Adaptive methods

In case the function to integrate varies slowly or fast in different integration domains, adaptive methods are normally used. One strategy is always to decrease the step size. As we have seen earlier, this leads to more CPU cycles and may lead to loss or numerical precision. An alternative is to use higher-order RK methods for example. However, this leads again to more cycles, furthermore, there is no guarantee that higher-order leads to an improved error.

Adaptive methods

Assume the exact result is $\tilde{x}$ and that we are using an RKM method. Suppose we run two calculations, one with h (called x_1) and one with $h/2$ (called x_2). Then

$$\tilde{x} = x_1 + Ch^{M+1} + O(h^{M+2}),$$

and

$$\tilde{x} = x_2 + 2C(h/2)^{M+1} + O(h^{M+2}),$$

with C a constant. Note that we calculate two halves in the last equation. We get then

$$|x_1 - x_2| = Ch^{M+1}(1 - \frac{1}{2^M}).$$

yielding

$$C = \frac{|x_1 - x_2|}{(1 - 2^{-M})h^{M+1}}.$$

We rewrite

$$\tilde{x} = x_2 + \epsilon + O((h)^{M+2}),$$

with

$$\epsilon = \frac{|x_1 - x_2|}{2^M - 1}.$$

Adaptive methods

With RK4 the expressions become

$$\tilde{x} = x_2 + \epsilon + O((h)^6),$$

with

$$\epsilon = \frac{|x_1 - x_2|}{15}.$$

The estimate is one order higher than the original RK4. But this method is normally rather inefficient since it requires a lot of computations. We solve typically the equation three times at each time step. However, we can compare the estimate ϵ with some by us given accuracy ξ. We can then ask the question: what is, with a given x_j and t_j, the largest possible step size $\tilde{h}$ that leads to a truncation error below ξ? We want

$$C\tilde{h} \leq \xi,$$

which leads to

$$\left(\frac{\tilde{h}}{h}\right)^{M+1} \frac{|x_1 - x_2|}{(1 - 2^{-M})} \leq \xi,$$

meaning that

$$\tilde{h} = h\left(\frac{\xi}{\epsilon}\right)^{1+1/M}.$$

Adaptive methods

With

$$\tilde{h} = h\left(\frac{\xi}{\epsilon}\right)^{1+1/M}.$$

we can design the following algorithm:

- If the two answers are close, keep the approximation to h.

- If $\epsilon > \xi$ we need to decrease the step size in the next time step.
- If $\epsilon < \xi$ we need to increase the step size in the next time step.

A much used algorithm is the so-called RKF45 which uses a combination of fourth and fifth order RK methods.

Adaptive methods, RKF45

At each step, two different approximations for the solution are made and compared. If the two answers are in close agreement, the approximation is accepted. If the two answers do not agree to a specified accuracy, the step size is reduced. If the answers agree to more significant digits than required, the step size is increased. Each step requires the use of the following six values:

$$k_1 = hf(t_k, y_k),$$

$$k_2 = hf(t_k + \frac{1}{4}h, y_k + \frac{1}{4}k_1),$$

$$k_3 = hf(t_k + \frac{3}{8}h, y_k + \frac{3}{32}k_1 + \frac{9}{32}k_2),$$

$$k_4 = hf(t_k + \frac{12}{13}h, y_k + \frac{1932}{2197}k_1 + \frac{7200}{2197}k_2 + \frac{7296}{2197}k_3),$$

$$k_5 = hf(t_k + h, y_k + \frac{439}{216}k_1 - 8k_2 + \frac{3680}{513}k_3 + \frac{845}{4104}k_4),$$

$$k_6 = hf(t_k + \frac{1}{2}h, y_k - \frac{8}{27}k_1 + 2k_2 - \frac{3544}{2565}k_2 + \frac{1859}{4104}k_4 - +\frac{11}{40}k_5).$$

Adaptive methods, RKF45

An approximation to the solution of the ODE is made using a Runge-Kutta method of order 4:

$$y_{k+1} = y_k + \frac{25}{216}k_1 + \frac{1408}{2565}k_3 + \frac{2197}{4101}k_4 - \frac{1}{5}k_5,$$

where the four function values k_1 , k_3 , k_4 , and k_5 are used. Notice that k_2 is not used here. A better value for the solution is determined using a Runge-Kutta method of order 5:

$$z_{k+1} = y_k + \frac{16}{135}k_1 + \frac{6656}{12825}k_3 + \frac{28561}{56430}k_4 - \frac{9}{50}k_5 + \frac{2}{55}k_6.$$

The optimal time step αh is then determined by

$$\alpha = \left(\frac{\xi h}{2|z_{k+1} - y_{k+1}|}\right)^{1/4},$$

with ξ our defined tolerance.

Solar system code, main program

```
#include <iostream>
#include <solver.h>
#include <planet.h>
#include <cmath>
#include <armadillo>
using namespace arma;
using namespace std;

int main()
{
//  The solarsystem object inherits all functionality of the solver class
   solver solarsystem;
// Here we initialize all elements in the solar system (planets)
   planet  Sun(1,0,0,0,0,0,0);
   planet  Mercury(1.2e-7, 0.39, 0, 0,0,9.96,0);
   planet  Venus(2.4e-6, 0.72, 0, 0,0,7.36,0);
   planet  Earth(1.5e-6,1,0,0, 0, 6.26, 0);
   planet  Mars(3.3e-7, 1.52, 0, 0,0,5.06,0);
   planet  Jupiter(9.5e-4, 5.20, 0,0,0,2.75,0);
   planet  Saturn(2.75e-4, 9.54, 0, 0,0,2.04,0);
   planet  Uranus(4.4e-5, 19.19, 0, 0,0,1.43,0);
   planet  Neptune(5.1e-5, 30.06, 0, 0,0,1.14,0);
   planet  Pluto(5.6e-9, 39.53, 0, 0,0,0.99,0);
// Now we add all planets
   solarsystem.add(Sun);
   solarsystem.add(Mercury);
   solarsystem.add(Venus);
   solarsystem.add(Earth);
   solarsystem.add(Mars);
   solarsystem.add(Jupiter);
   solarsystem.add(Saturn);
   solarsystem.add(Uranus);
   solarsystem.add(Neptune);
   solarsystem.add(Pluto);

   int elements = solarsystem.number_planets;
   cout << "number of element" << elements<< endl;
   solarsystem.solverRK4(solarsystem.all_planets, 0.001, 100 );

}
```

The full code, with cpp files and header files, is at https://github.com/CompPhysics/ComputationalPhysicsMSU/tree/master/doc/Programs/OOExamples.

Solar system code, main program, header file

```
#ifndef PLANET_H
#define PLANET_H

class planet
{
public:

    double position[3];
    double velocity[3];
    double mass;
    planet(double M, double x,double y, double z, double vx, double vy, double vz);
```

```
    planet();

};

#endif // PLANET_H
```

Solar system code, main program, header file, cpp file for planet class, simple example

```
#include "planet.h"
// Simple constructor
planet::planet()
{
};
// Improved constructor
planet::planet(double M, double x,double y, double z, double vx, double vy, double vz){
mass = M;
position[0] = x;
position[1] = y;
position[2] = z;

velocity[0] = vx;
velocity[1] = vy;
velocity[2] = vz;
};
```