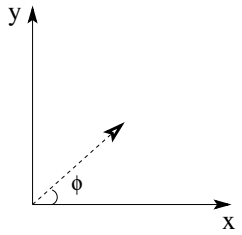
BERN01
HT23

Lecture 1, BERN01
Modelling in Computational Science

1 ODE-Projectile Motion Example

Consider firing a cannon shell, New Year fireworks or kicking a fotball into the air. We may, for instance, be interested where the rocket or ball lands. Any realistic attempt at modeling such a process we need to include air resistance, which in general complicates the equations sufficiently to make sure that no analytic solutions are available. In contrast to the calculation from the previous lecture we are thus forced to use a computer.

Let us write down the equations for the problem: A ball of mass m is ejected at an angle ϕ with an initial velocity $\mathbf{v}_0 = (v_{x,0}, v_{y,0})$ in a gravitational field with gravitational constant g.



Newton's equations of motion are:

$$\begin{aligned} m\frac{d^2x}{dt^2} &= F_{\text{drag},x} \\ m\frac{d^2y}{dt^2} &= -mg + F_{\text{drag},y} \end{aligned} \tag{1}$$

where $F_{\text{drag},x}$ ($F_{\text{drag},y}$) is the x-component (y-component) of the drag force due to the air. We may in general write (see G&N)

$$\mathbf{F}_{\text{drag}} = -(B_1 v + B_2 v^2)\hat{\mathbf{v}} \tag{2}$$

where $v = \sqrt{v_x^2 + v_y^2}$ with v_x being the x- (y)-component of the velocity, and $\hat{\mathbf{v}} = (v_x\hat{x} + v_y\hat{y})/v$ is a unit vector in the direction of the velocity. The first term above is the Stoke's drag - for a sphere we have: $B_1 = 6\pi\eta R$, where R is the radius and η the viscosity of the medium in which the object travels. Also, quite generally, we have:

$$B_2 = -\frac{1}{2}C\rho A \tag{3}$$

where A is the frontal area of the object, ρ the air density, and C a shape-dependent constant, see for instance *http://en.wikipedia.org/wiki/Drag_coefficient* for experimentally determined C-values for different shapes. For most macroscopic objects it is safe to neglect the Stoke's drag (show this!), i.e., set $B_1 = 0$. Due to the non-linearity caused by the 2nd term in the drag force, we cannot solve the equations above analytically.

The type of problem specified above is called an *initial value problem*, since the initial position and velocity is known at time $t = 0$. A useful trick to solve equations of the type above is rewrite them as a set of first order equations according to:

$$\begin{aligned}
\frac{dx}{dt} &= v_x \\
\frac{dv_x}{dt} &= -\frac{B_1 v_x}{m} - \frac{B_2 v v_x}{m} \\
\frac{dy}{dt} &= v_y \\
\frac{dv_y}{dt} &= -g - \frac{B_1 v_y}{m} - \frac{B_2 v v_y}{m}
\end{aligned} \tag{4}$$

The solution to the equations above can be generated using the Euler algorithm as in the previous lecture. We then noticed that the solution approached the true solution only occurs for small step sizes. We are thus lead to the question: How do we estimate numerical errors? And, can we improve on the Euler algorithm without decreasing the step size?

2 Estimating Errors

There are two types of errors we need to consider when doing numerical computations: round-off errors and truncation errors. In addition there may be errors in the input data (we may not know exactly the mass of the ball in the example above, for instance). We will only consider the first two types of errors here.

2.1 Round-off Errors

Numerical calculations are done in integer or floating-point arithmetic. Integer arithmetic is exact, whereas floating-point arithmetic has roundoff errors.

The floating-point representation looks like

$$s \cdot M \cdot 2^p \qquad \begin{cases} s & sign \\ M & mantissa \\ p & integerexponent\ (L \leq p \leq U) \end{cases}$$

Any real number can be represented this way, and the representation becomes unique if we require eg. that $1 \leq M < 2$ for any non-zero number.

Example. How is the number 5.625 represented in a binary way? We have:

$$(5.625)_{10} = 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 + 1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3} = (101.101)_2$$

Thus, this particular number can be represented exactly. Note, however in general this is not the case.

On the computer, each floating-point number becomes a string of bits, each bit being 0 or 1. Suppose the number of bits per floating-point number (the "wordlength") is 32, with 1 sign bit, 8 bits for p and t=23 bits for M. p and M may then (for example) be represented as

$$\begin{aligned} p &= -2^7, -2^7+1, ..., 2^7 - 1 \underbrace{x_7 \cdot 2^7 + x_6 \cdot 2^6 + ... + x_0 \cdot 2^0 - 2^7}_{} \qquad x_i \in \{0,1\} \\ M &= 1 + y_1 \cdot 2^{-1} + y_2 \cdot 2^{-2} + ... + y_{23} \cdot 2^{-23} \qquad y_i \in \{0,1\} \end{aligned}$$

In the example above we have $L = -2^7$ and $U = 2^7 - 1$. In *floating point* (in contrast to *fixed point*) arithmetics the U (and L) are adjusted according to the number we want to store. *Roundoff errors* arise because the number of y_i's is finite. Thus, in the example from the previous section, we actually make a small error already before starting our simulations(!), simply by storing the initial velocity and positions on the form above. Suppose we want to store a real number x on the computer. The representation of x is denoted by x_r. The relative error can then be estimated as

$$\frac{|x_r - x|}{|x|} = \mu \sim 2^{-t},$$

where t is the number of bits used to represent the mantissa M. The typical *relative precision* μ of 32-bit floating-point numbers is thus: $\mu \sim 2^{-23} \sim 10^{-7}$. For 64-bit double precision representation (mostly used today) usually 52 bits are reserved

for M, this gives a precision $\mu \sim 2^{-52} \sim 10^{-16}$. Roundoff errors are especially troublesome when subtracting two numbers with small relative difference. Also when adding many numbers care must be taken, see examples below.

Example. Consider one of the solutions of the equation $ax^2 + bx + c = 0$:

$$x = -\frac{b - \sqrt{b^2 - 4ac}}{2a}$$

for $\sqrt{ac} \ll b$. Direct evaluation of the expression above would involve subtracting two large numbers with small difference. Here the trick is simple. Just multiply with $b + \sqrt{b^2 - 4ac}$ in both denominator and nominator:

$$x = -\frac{b - \sqrt{b^2 - 4ac}}{2a} \times \frac{b + \sqrt{b^2 - 4ac}}{b + \sqrt{b^2 - 4ac}} = -\frac{2c}{b + \sqrt{b^2 - 4ac}}.$$

Example. Consider adding two floating point numbers a and $a\delta$. If $\delta < \mu$ then $a + a\delta = a(1 + \delta)$ will be rounded off to a. Thus, when adding many numbers it is usually a good idea to group the numbers in such a way as to avoid accumulation of this type of round-off errors. For instance, if one performs a sum of numbers of decaying magnitude (such as the sum $\sum_{n=1}^{N} 1/n^2$) it is advantageous (why?) to sum the smallest numbers first – backward summation.

2.2 Truncation errors

In addition to roundoff errors, most numerical calculations contain errors due to approximations, such as discretization or truncation of an infinite series. Such errors are called *truncation errors*. Truncation errors are errors that would persist even on a hypothetical computer with infinite precision.

3 Numerical derivatives and Richardson extrapolation

As a first demonstration of how to estimate, and reduce, errors we will in this section consider numerical derivatives. We will also introduce Richardson extrapolation, which is a clever technique for reducing truncation errors in numerical computations.

3.1 Numerical derivatives

Suppose we want to numerically compute $f'(x)$. Three different alternatives are:

$$\begin{aligned} f'(x) &\approx D_+(h) = f(x+h) - f(x)h \\ & \quad D_-(h) = f(x) - f(x-h)h \\ & \quad D_0(h) = f(x+h) - f(x-h)2h \end{aligned} \tag{5}$$

The first (second) approximation is called a forward (backward) difference, and the third expression is a central difference approximation.

- The truncation error for $D_+(h)$ and $D_-(h)$ is $\varepsilon_t \sim h\,|f''(x)|$, since

$$\begin{aligned} f(x+h) &= f(x) + hf'(x) + h^2 2f''(x) + h^3 3! f'''(x) + ... \\ f(x-h) &= f(x) - hf'(x) + h^2 2f''(x) - h^3 3! f'''(x) + ... \end{aligned} \tag{6}$$

- The roundoff errors in $f(x+h)$ and $f(x)$ are $\gtrsim \varepsilon|f(x)|$, where ε is the relative floating-point precision. Therefore, a lower bound on the total roundoff error ε_r is $\varepsilon_r \gtrsim \varepsilon|f(x)|/h$.

Note that h large $\Rightarrow \varepsilon_t$ large, whereas h small $\Rightarrow \varepsilon_r$ large. For the total relative error, we get the estimate

$$\varepsilon_r + \varepsilon_t|f'| \gtrsim h\,|f''| + \varepsilon\,|f|\,/h|f'| = 1|f'|\left[\left(\sqrt{h\,|f''|} - \sqrt{\varepsilon\,|f|\,/h}\right)^2 + 2\sqrt{\varepsilon\,|f''f|}\right]$$

With an optimal choice of h (so that the square vanishes), this becomes

$$\varepsilon_r + \varepsilon_t|f'| \gtrsim \sqrt{\varepsilon}\sqrt{|f''f|f'^2} \tag{7}$$

So, the error is $\mathcal{O}(\varepsilon^{1/2})$ rather than $\mathcal{O}(\varepsilon)$, which does make a difference: $\varepsilon = 10^{-7} \Rightarrow \varepsilon^{1/2} \approx 3 \cdot 10^{-4}$.

For the central difference approximation we have [using Eq. (6)]

$$f(x+h) - f(x-h)2h = f'(x) + h^2 3! f'''(x) + h^4 5! f^{(5)}(x) + ... \tag{8}$$

The truncation error using $D_0(h)$ is thus one order better than that of $D_+(h)$ and $D_-(h)$ above (as a result, the total error scales as $\varepsilon^{2/3}$ for the central difference, as can be easily verified). A more systematic approach to reduce the truncation error is to use Richardson extrapolation (see below).

3.2 Richardson Extrapolation

Suppose we want to determine a quantity a_0 that satisfies

$$a_0 = h \to 0 \text{lim} a\,(h)$$

where h may be thought of as a step-size parameter. Suppose further that

1. the functional form of $a(h)$ is known to be $a\,(h) = a_0 + a_1 h^2 + a_2 h^4 + ...$
2. the value $a\,(h)$ is known for $h = h_0, 2h_0, 2^2 h_0, \ldots$

a_0 could, for instance, be a derivative and $a(h)$ the central-difference approximation. We can then obtain an improved estimator $\hat{a}(h)$ of a_0 in the following way:

$$\begin{cases} a(h) = a_0 + a_1 h^2 + a_2 h^4 + \mathcal{O}(h^6) \\ a(2h) = a_0 + 4a_1 h^2 + 16 a_2 h^4 + O\,(h^6) \end{cases} \quad \Rightarrow \quad 4a\,(h) - a\,(2h) = 3a_0 - 12 a_2 h^4 + O\left(h^6\right)$$

so that

$$\hat{a}\,(h) \equiv 4a\,(h) - a\,(2h)3 = a_0 - 4a_2 h^4 + O\left(h^6\right) \tag{9}$$

$\hat{a}(h)$ is an improved estimator of a_0 because the truncation error is $\mathcal{O}(h^4)$ instead of $\mathcal{O}(h^2)$.

We can now go on to eliminate the h^4 term by forming

$$\hat{\hat{a}}\,(h) \equiv 16\hat{a}\,(h) - \hat{a}\,(2h)15 = a_0 + O\left(h^6\right) \qquad \text{(verify this!)} \tag{10}$$

and so on. To calculate $\hat{\hat{a}}(h)$ for one value $h = h_0$, we need to know $a(h)$ for $h = h_0$, $2h_0$ and $4h_0$. The scheme can thus be illustrated:

$$\begin{array}{ccccc} a(4h_0) & & & & \\ & \searrow & & & \\ a(2h_0) & \to & \hat{a}(2h_0) & & \\ & \searrow & & \searrow & \\ a(h_0) & \to & \hat{a}(h_0) & \to & \hat{\hat{a}}(h_0) \end{array}$$

Comments

The same procedure can be applied for more general versions of the assumptions 1 and 2. The precise form of the expressions for $\hat{a}$ and $\hat{\hat{a}}$ will then change.

Examples of methods that use this technique are Romberg's integration method and the Burlirsh-Stoer method for ordinary differential equations.

4 Ordinary Differential Equations, initial value problems

[G&N: appendix A]
[NR: chap. 16 (2nd ed.), chap. 17 (3rd ed.)]

In this section we go through different improvements of the Euler method for solving ordinary differential equations with specified initial conditions (i.e., problems of the type discussed in the first part of this lecture).

4.1 Introduction

This section deals with numerical integration of Ordinary Differential Equations (ODEs). We will discuss methods for solving systems of first-order ODEs

$$\begin{cases} dy_1 dt = f_1\left(t, y_1, \ldots, y_N\right) \\ \quad \vdots \\ dy_N dt = f_N\left(t, y_1, \ldots, y_N\right) \end{cases} \tag{11}$$

for given initial values of all the components y_i at some time t_0 (for a discussion of boundary value problems, see lecture 4). In vector notation, this initial value problem can be written as

$$d\mathbf{y}dt = \mathbf{f}(t, \mathbf{y}) \qquad \mathbf{y}(x_0) = \mathbf{y}_0\,. \tag{12}$$

Example. Eq. (4) can be written on the form above, where:

$$\mathbf{y} = \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{pmatrix} = \begin{pmatrix} x \\ v_x \\ y \\ v_y \end{pmatrix} \tag{13}$$

and

$$\mathbf{f}(t, \mathbf{y}) = \mathbf{f}(\mathbf{y}) = \begin{pmatrix} y_2 \\ B_1 y_2/m - B_2 y_2 \sqrt{y_2^2 + y_4^2}/m \\ y_4 \\ -g - B_1 y_4/m - B_2 y_4 \sqrt{y_2^2 + y_4^2}/m \end{pmatrix} \tag{14}$$

Looking only at first-order ODEs is not a strong limitation, because any N^{th} order ODE of the form

$$d^N y dt^N = f\left(t, y, dydt, \ldots, d^{N-1} y dt^{N-1}\right) \tag{15}$$

can be written as a system of first-order ODEs by the transformation

$$\begin{array}{l} y_1 = y \\ y_2 = dydt \\ \vdots \\ y_N = d^{N-1}ydt^{N-1} \end{array} \quad \Rightarrow \quad \left\{ \begin{array}{l} dy_1 dt = y_2 \\ dy_2 dt = y_3 \\ \vdots \\ dy_N dt = f\left(t, y_1, \ldots, y_N\right) \end{array} \right. \tag{16}$$

4.2 Euler's Method

The Euler method, which we encountered in Lecture 1 (for the case $N = 1$) is the simplest way to solve to solve Eq. (12). We discretize t using a step size h:

$$\begin{array}{l} t_n = t_0 + nh \quad n = 0, 1, \ldots \\ \mathbf{y}_n = \mathbf{y}\left(t_n\right) \end{array} \tag{17}$$

Approximating the derivative in Eq. (12) with a simple forward difference $D_+(h)$, we obtain the recursion formula

$$\mathbf{y}_{n+1} = \mathbf{y}_n + h\mathbf{f}(t_n, \mathbf{y}_n)\,. \tag{18}$$

which can be used to calculate $\mathbf{y}_n$ for arbitrary n, starting from a given $\mathbf{y}_0$. This is called the Euler method. The Euler method is a first-order method: its local truncation error is $\sim h^2$ (one step), which makes the global truncation error $\sim h$. How can we improve on this? One way is to use include higher an expansion of $\mathbf{y}(t_n + h)$ to higher orders in h - one such method is the Verlot method described in chap. 9 and appendix A of G&N. Another way is to use a Runge-Kutta approach, see next subsection.

4.3 The Runge-Kutta Method

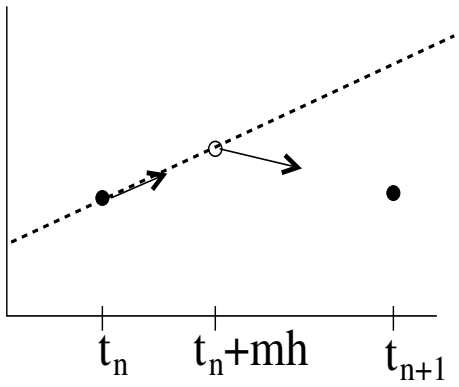
In the Euler method we used the value of the function $\mathbf{f}$ at "the beginning of the interval", i.e. at t_n and $\mathbf{y}_n$ to estimate $\mathbf{y}_{n+1}$. In the The Runge-Kutta method, the estimator for $\mathbf{y}_{n+1}$ is instead constructed using values of $\mathbf{f}$ itself at carefully chosen points between t_n and t_{n+1}. By increasing the number of points, the order of the method can be increased.

4.3.1 Second-Order Runge-Kutta (local error $\sim h^3$)

Consider the ansatz

$$\begin{aligned} \mathbf{k}_1 &= h\mathbf{f}\left(t_n, \mathbf{y}_n\right) && (19)\\ \mathbf{k}_2 &= h\mathbf{f}\left(t_n + mh, \mathbf{y}_n + m\mathbf{k}_1\right) && (20)\\ \mathbf{y}_{n+1} &= \mathbf{y}_n + a\mathbf{k}_1 + b\mathbf{k}_2 && (21) \end{aligned}$$

which combines two different f values and has three parameters: a, b and m. The idea is to determine these three parameters so that the error becomes as small as possible. For this purpose, we make a Taylor expansion of $\mathbf{y}_{n+1}$ about $t = t_n$:



$$\begin{aligned} \mathbf{k}_2 &= h\mathbf{f}(t_n, \mathbf{y}_n) + mh^2\partial_t\mathbf{f}\left(t_n, \mathbf{y}_n\right) + mh^2\mathbf{f}\left(t_n, \mathbf{y}_n\right)\nabla_y\mathbf{f}\left(t_n, \mathbf{y}_n\right) + \mathcal{O}\left(h^3\right) \quad \Rightarrow\\ \mathbf{y}_{n+1} &= \mathbf{y}_n + (a+b)\, h\mathbf{f}\left(t_n, \mathbf{y}_n\right) + bmh^2\left(\partial_t\mathbf{f}\left(t_n, \mathbf{y}_n\right) + \mathbf{f}\left(t_n, \mathbf{y}_n\right)\nabla_y\mathbf{f}\left(t_n, \mathbf{y}_n\right)\right) + \mathcal{O}(h^3) \end{aligned} \quad (22)$$

We now want to compare the expression above to the Taylor expansion of $\mathbf{y}(t_n + h)$ of the solution of Eq. (12). We have

$$\mathbf{y}\left(t_n + h\right) = \mathbf{y}_n + h\, d\mathbf{y}dt\big|_{t=t_n} + h^2 2\, d^2\mathbf{y}dt^2\big|_{t=t_n} + \ldots$$

Utilizing the equation of motion, $d\mathbf{y}/dt = \mathbf{f}$, and keeping terms to second order in h we can rewrite the expression above as:

$$\mathbf{y}\left(t_n + h\right) = \mathbf{y}_n + h\mathbf{f}\left(t_n, \mathbf{y}_n\right) + h^2 2\left[\partial_t\mathbf{f}\left(t_n, \mathbf{y}_n\right) + \mathbf{f}\left(t_n, \mathbf{y}_n\right)\nabla_y\mathbf{f}\left(t_n, \mathbf{y}_n\right)\right] + O\left(h^3\right)$$

By comparing this expression with Eq. (22) we see that the local error becomes $\mathcal{O}(h^3)$ [global error $\mathcal{O}(h^2)$] if we choose

$$a + b = 1 \qquad \text{and} \qquad bm = 1/2 \quad (23)$$

This is the best that can be achieved with this ansatz — to eliminate the $\mathcal{O}(h^3)$ error term we would need more points. The two equations for a, b and m have a one-parameter family of solutions. We can, for example, take

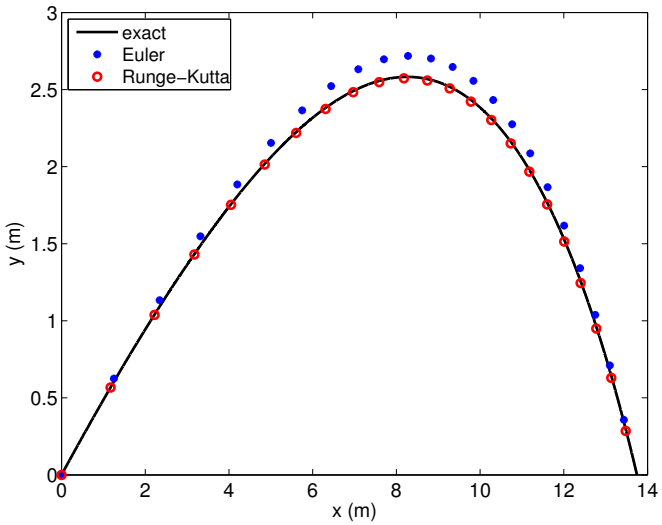
$$a = 0,\ b = 1,\ m = 12$$

as in *Numerical Recipes* book, or

$$a = b = 12,\ m = 1$$

which is called Heun's method [see Eqs. (A.7) and (A.8) in G&N)].

In the figure below we compare simulations using the Euler algorithm and the second-order Runge-Kutta method for the same step size h (=1/16 s). We set $B_2/m = 0.01m^{-1}$, $g = 9.82$ m/s^2, $v_x(t = 0) = 20$ m/s and $v_y(t = 0)$=10 m/s. We see that indeed the Runge-Kutta method performs better. The solid line is an "exact" solution, obtained by using the Runge-Kutta method with a very small step size.



In the table below we give the difference between the "exact" solution for height y =1 m, and compare to the approximate numerically generated result for the two methods for different h. From the numbers we see that indeed the global error scales as h for the Euler method, whereas the error scales as h^2 for the Runge-Kutta method (if we double h, the error increases by a factor 4).

h (units s)	$\lvert y_{\text{exact}} - y_{\text{Euler}} \rvert$	$\lvert y_{\text{exact}} - y_{\text{RungeKutta}} \rvert$
1/8	0.20	0.042
1/16	0.11	0.0084
1/32	0.056	0.0019
1/64	0.024	0.00045

4.3.2 Fourth-Order Runge-Kutta

Higher-order Runge-Kutta methods can be obtained by adding more points. This gives improved accuracy at the cost of increased complexity. A very popular compromise is the fourth-order scheme given by Eq. (A.10) in G&N, which requires four $\mathbf{f}$ evaluations. One reason that this scheme is so popular is that in order to get a fifth-order scheme, it turns out that six (and not five) points are needed.

$$\begin{aligned}
\mathbf{k}_1 &= h\mathbf{f}(t_n, \mathbf{y}_n) \\
\mathbf{k}_2 &= h\mathbf{f}(t_n + h/2, \mathbf{y}_n + \mathbf{k}_1/2) \\
\mathbf{k}_3 &= h\mathbf{f}(t_n + h/2, \mathbf{y}_n + \mathbf{k}_2/2) \\
\mathbf{k}_4 &= h\mathbf{f}(t_n + h, \mathbf{y}_n + \mathbf{k}_3) \\
\mathbf{y}_{n+1} &= \mathbf{y}_n + \mathbf{k}_1/6 + \mathbf{k}_2/3 + \mathbf{k}_3/3 + \mathbf{k}_4/6 + \mathcal{O}(h^5)
\end{aligned} \tag{24}$$

4.4 Adaptive Step Size*

So far, we have assumed that the step size h is held constant throughout the integration. However, the function $\mathbf{f}(t, \mathbf{y})$ may be very different in different parts of space. As a result, there may exist both "easy" regions where a large step can be used and "difficult" ones where a much smaller step size is required. Therefore, it is often very useful to vary the step size.

A natural choice in an adaptive step size algorithm is to vary the step size in such a way that the local error is kept at a constant level. For that, we need to have an estimate of the local error. A convenient way to estimate the local error is by step doubling. This means that we repeat all calculations using a step size twice as large. By comparing the two calculations, we get an estimate of the local error. To see how this works, consider fourth-order Runge-Kutta (and use $N = 1$ for simplicity), for which the local error scales as h^5. Let $y(t + h; h/2)$ and $y(t + h; h)$ denote the estimates of the exact solution $y(t + h)$ that are obtained by using two steps of size $h/2$ and one step of size h, respectively. For small h, these estimates should behave as

$$\begin{aligned}
y(t + h; h/2) - y(t + h) &\sim \left(\frac{h}{2}\right)^5 c(t) + \mathcal{O}(h^6) \\
y(t + h; h) - y(t + h) &\sim h^5 c(t) + \mathcal{O}(h^6)
\end{aligned}$$

where $c(t)$ is some unknown function. The quantity $\Delta = |y(t + h; h) - y(t + h; h/2)|$ provides a rough estimate of the error in $y(t + h; h)$. To keep the local error under

control, we may require that $\Delta < \Delta_{\rm tol}$, where $\Delta_{\rm tol}$ is a predetermined tolerance level. If $\Delta > \Delta_{\rm tol}$, we redo the calculation using a smaller step size h'. How small the new step size h' should be can be estimated by using that $\Delta \sim h^5$. This gives

$$h' \approx S \left(\frac{\Delta_{\rm tol}}{\Delta} \right)^{1/5} h$$

where h is the step size that gave us Δ. S is a "safety" factor that is supposed to be less than 1. If, on the other hand, $\Delta \ll \Delta_{\rm tol}$, the step size should be increased. How much it can be increased can also be estimated by using that $\Delta \sim h^5$.

The gain from using an adaptive step size can be huge compared to the cost of the additional calculations that are needed to keep track of the local error.

4.5 The Modified Midpoint Method*

Consider the same first-order ODE as before (here, $N = 1$ for simplicity)

$$dydt = f(t, y)\,.$$

A central-difference approximation $D_0(h)$ [see Eq. (5)] of the derivative gives us the so-called midpoint method,

$$y_{n+1} = y_{n-1} + 2hf(t_n, y_n)\,,$$

with a local error of $\mathcal{O}(h^3)$. This is a "two-point" method, in which both y_n and y_{n-1} are needed in order to obtain y_{n+1}. [Compare this with the second order Runge-Kutta method with $a = 0$: $y_{n+1} = y_n + hf(t_n + h/2, y_n + hf(t_n, y_n)/2)$ "=" $y_n + hf(t_{n+1/2}, y_{n+1/2})$].

The *modified* midpoint method has three components:

1. An initial Euler step to get a second starting value:

$$y_1 = y_0 + hf(t_0, y_0)$$

2. $N - 1$ steps with the midpoint method:

$$y_{n+1} = y_{n-1} + 2hf(t_n, y_n) \qquad n = 1, \ldots, N-1$$

3. A "correction" of the last value y_N:

$$y(t_0 + H; h) = 12\,(y_N + y_{N-1} + hf\,(t_N, y_N))$$

where $H = Nh$. $y(t_0 + H; h)$ is the final estimate, for step size h, of the exact solution $y(t_0 + H)$.

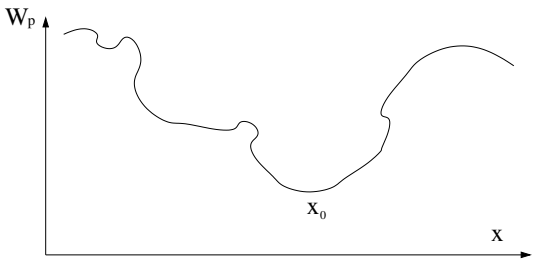
For fixed t_0 and H and even N, it has been shown that

$$y\left(t_0 + H; h\right) - y\left(t_0 + H\right) = c_1 h^2 + c_2 h^4 + ...$$

In other words, the truncation error contains only even powers of h, a situation we have encountered before. This property makes the method well suited for Richardson extrapolation. The so-called Bulirch-Stoer method is based on the modified midpoint method and a Richardson-like extrapolation (but with rational functions rather than polynomials). This method can be a good choice if high accuracy is needed, see *Numerical Recipes* for further details.

5 Instability

Physical systems tend towards a state where the potential energy $W_p(x)$ is at minimum, i.e. towards the position x_0 in the cartoon below.



Since a minimum is characterized by the first derivative of the energy being zero we can expand $W_p(x)$ around the equilibrium position x_0 as:

$$W_p(x) \approx W_p(x_0) + \frac{1}{2} W''(x_0)(x - x_0)^2 \tag{25}$$

We notice that the expression above is the same as the potential energy for a spring [recall Hooke's law: $F = -dW_p(x)/dx = -\kappa(x - x_0)$] where the spring constant here is $\kappa = W''(x_0)$. The argument above can be generalized to problems of more than one variable. This is the reason why physicists likes springs - most processes close to equilibrium or potential energy minimum behave like harmonic springs!

Let us now consider dynamics. Newton's equation of motion (setting $x_0 = 0$) becomes:

$$M \frac{d^2 x}{dt^2} = -\kappa x \tag{26}$$

or equivalently

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

where

$$\mathbf{y} = \begin{pmatrix} x \\ v_x \end{pmatrix}$$

and

$$\mathbf{f}(\mathbf{y}) = \begin{pmatrix} v_x \\ -(\kappa/M)x \end{pmatrix}$$

This system of equation has the same form as what we considered in the previous lecture. In fact, these equations are so simple that we can solve them analytically. For instance, if $v_0 = 0$ then

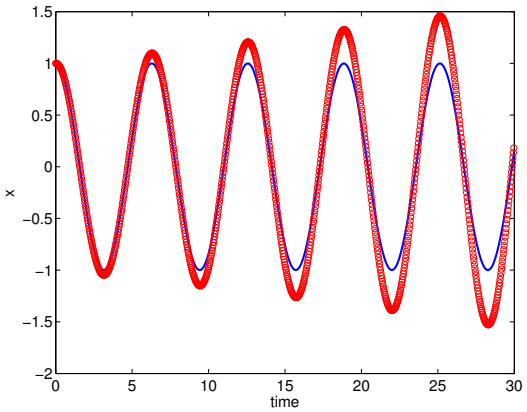
$$x(t) = A\cos(\omega_0 t) \tag{27}$$

with a characteristic frequency $\omega_0 = \kappa/M$, and $A = x(0)$. In G& N chap. 3 the dynamics of a pendulum is considered – for small angles these equation are identical to the ones above.

Let us now compare the analytic solution with numerics. The Euler algorithm is: $\mathbf{y}_{n+1} = \mathbf{y}_n + hf(\mathbf{y}_n)$, with $t_n = nh$ and $\mathbf{y}_n = \mathbf{y}(t_n)$, see Lecture 2. Explicitly we thus have the recurrence relation:

$$\begin{pmatrix} v_{n+1} \\ x_{n+1} \end{pmatrix} = \begin{pmatrix} v_n \\ x_n \end{pmatrix} + h \begin{pmatrix} -(\kappa/M)x_n \\ v_n \end{pmatrix} \tag{28}$$

Below a simulation (red circles), using the algorithm above, is shown ($x_0 = 1$, $v_0 = 0$, $\kappa/M = 1$, and $h = 0.03$). The solid line is the analytic result.



We notice that the amplitude in the simulations increase with time in contrast to the analytic result, i.e. the Euler algorithm is *unstable* for the present problem.

Let us now use the Runge-Kutta method to see if this will resolve the instability seen above. The second order Runge-Kutta is: $\mathbf{y}_{n+1} = \mathbf{y}_n + a\mathbf{k}_1 + b\mathbf{k}_2$, where $\mathbf{k}_1 = hf(\mathbf{y}_n)$, and $\mathbf{k}_2 = hf(\mathbf{y}_n + m\mathbf{k}_1)$. Explicitly we have:

$$\begin{pmatrix} x_{n+1} \\ v_{n+1} \end{pmatrix} = \begin{pmatrix} x_n \\ v_n \end{pmatrix} + a \begin{pmatrix} x^{(1)}_{\text{temp}} \\ v^{(1)}_{\text{temp}} \end{pmatrix} + b \begin{pmatrix} x^{(2)}_{\text{temp}} \\ v^{(2)}_{\text{temp}} \end{pmatrix}$$

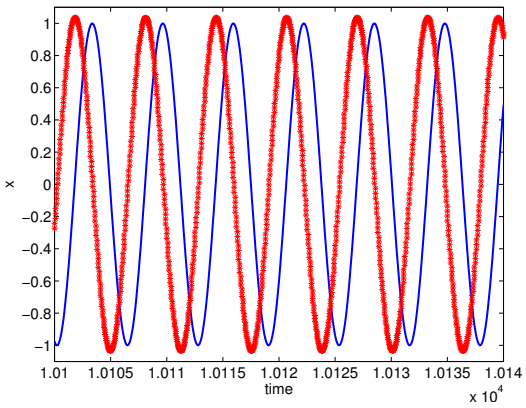
where

$$\mathbf{k}_1 = \begin{pmatrix} x^{(1)}_{\text{temp}} \\ v^{(1)}_{\text{temp}} \end{pmatrix} = h \begin{pmatrix} v_n \\ -(\kappa/M)x_n \end{pmatrix}$$

and

$$\mathbf{k}_2 = \begin{pmatrix} x^{(2)}_{\text{temp}} \\ v^{(2)}_{\text{temp}} \end{pmatrix} = h \begin{pmatrix} v_n + mv^{(1)}_{\text{temp}} \\ -(\kappa/M)(x_n + mx^{(1)}_{\text{temp}}) \end{pmatrix}$$

Below are simulations (red marks) using the second order Runge-Kutta method with $a = b = 1/2$ and $m = 1$. The solid blue curve is the analytic result.



We see that also with this algorithm, solutions are unstable, although we have to wait much longer for the instability to set in (notice the units on the time-axis). What is wrong?

6 ODE stability, the Euler-Cromer method

Let us now address the stability problem we had in the previous section. The origin of the found instability is easily understood if one considers the energy difference

between steps. Let us consider the Euler method for simplicity. The energy at a given step is

$$E_n = W_p + E_{\text{kinetic}} = \frac{\kappa x_n^2}{2} + \frac{M v_n^2}{2}$$

and the energy difference between steps is then [see Eq. (28)] after a little algebra:

$$\Delta E_n = E_{n+1} - E_n = \frac{\kappa}{2}\Big(v_n^2 + \frac{\kappa}{M} x_n^2\Big) h^2 \tag{29}$$

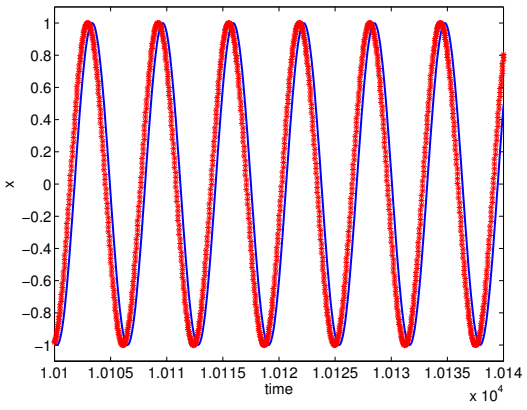
i.e. the energy increases for each time step!

Cromer suggested a simple solution to the problem above. The idea to use Euler's method, but when updating the position at step $n+1$ one uses v_{n+1} instead of v_n. The method thus becomes [compare to Eq. (28)]

$$v_{n+1} = v_n + f_n h \tag{30}$$

$$x_{n+1} = x_n + h v_{n+1} \tag{31}$$

where $f_n = -(\kappa/M)x_n$. The algorithm above is called the Euler-Cromer algorithm. As shown in *A.Cromer, Am. J. Phys. vol 49, pp. 455 (1981)*, the change in energy at step n compared to the initial energy is (for $v_0 = 0$) given by $-Mhf_n v_n/2$. Thus the maximum deviation from the true energy conservation scales as h and the error is zero whenever the force is zero. Also, it can be shown that the error averaged over half a period is zero. A simulation using the Euler-Cromer algorithm is shown below.



We notice that the amplitude does not diverge, as it did for the Euler and Runge-Kutta methods.

As we saw above the Euler-Cromer algorithm preserves the average energy. For other types of problems there may be other conservation laws or symmetries which we want

to be satisfied, for instance: the Verlet algorithm (see G&N chapter 9 and appendix A) preserves the time-reversal of Newton's equations, the leap frog method (see Lecture 4) preserves phase-space volume, and the Crank-Nicholson algorithm (see Lectures on PDEs) preserves normalization of the wavefunction in quantum mechanics problems.