Modelling and Simulation EEEN 30150

Electronic, Electrical and Communications Engineering

Dr Paul Curran

Room 145, Engineering and Materials' Science Centre.

paul.curran@ucd.ie

+353-1-7161846

Overview of Topics

Solution of equations by iteration.

Optimisation.

Solution of system of linear equations.

Solution of ordinary differential equations: initial value problems.

Numerical Quadrature.

Solution of ordinary differential equations: boundary value problems.

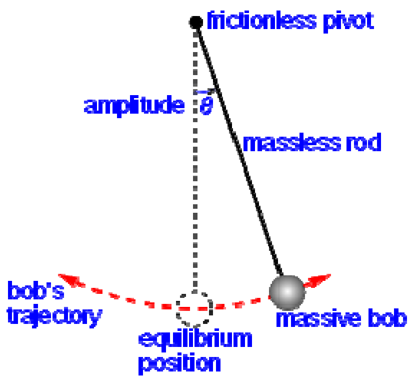
Solution of partial differential equations.

Section 4: Ordinary Differential Equations

- Elementary method: Euler Method
- Advanced methods: Fourth order Runge-Kutta and Adams-Bashforth/Moulton

Workload: 5 lectures + 1 laboratory + Autonomous learning

Example: Simple Gravity Pendulum



Example: Simple Gravity Pendulum

The simple gravity pendulum is an idealisation of a real pendulum. Effects such as friction and bowing of the rod are ignored. The resulting equation is the *pendulum equation*:

$$ml\frac{d^2\theta}{dt^2} + mg\sin(\theta) = 0 \qquad \frac{d^2\theta}{dt^2} + \left(\frac{g}{l}\right)\sin(\theta) = 0$$

where g is the acceleration due to gravity and l is the length of the rod. θ is the angle from the vertical in radians. This equation is an *ordinary differential equation*. The associated *initial value problem* is to solve this equation given $\theta(0)$ and $\theta'(0)$. In particular we are interested in $\theta(0) = \theta_0$, $\theta'(0) = 0$.

Example: Simple Gravity Pendulum

The pendulum equation is a second order equation since the highest derivative of the angle present is the second derivative. It is possible to translate the equation into a system of two first order differential equations by introducing a new variable, the angular velocity:

$$\theta' = \omega$$

$$\omega' = -\left(\frac{g}{l}\right)\sin(\theta)$$

It is generally the case that high order differential equations can be transformed to systems of first order differential equations and for this reason we focus upon first order differential equations.

First Order ODE: Euler Method

The simplest system of first order differential equations is one where there is just a single equation. It is customary, when taking a general view, to denote the dependent variable by y and take the equation to have the general form:

$$y' = f(x, y)$$

where y' denotes the derivative of y with respect to x and where y is a function of x sometimes more explicitly denoted $y(x)$. The problem to be solved is an *initial value problem*, meaning that $y(x_0) = y_0$ is given.

First Order ODE: Euler Method

The critical idea in the case of the Euler method (and in fact of any method) is that of taking a step at a time. Let the step size be h. We look to determine $y(x)$ for the special values of $x = x_0$, $x_0 + h$, x_0+2h, x_0+3h, *etc*. Let:

$$y(x_0) = y_0\ ,\ y(x_0 + h) = y_1\ ,\ y(x_0 + 2h) = y_2\ ,\ldots$$

Employing the Taylor series:

$$y_{n+1} = y(x_0 + (n+1)h) = y((x_0 + nh) + h) = y(x_0 + nh) + hy'(x_0 + nh) + \cdots$$

$$y_{n+1} \cong y_n + hf(x_0 + nh, y_n)$$

First Order ODE: Euler Method

So Euler's method reduces the problem to that of solving a *recursion* or *difference equation*. Although the error in each step depends upon h^2 and can therefore be quite small, error can propagate with erroneous values from earlier steps used as initial data for the present step. As one takes larger and larger numbers of steps the approximate solution can, and generally will, drift away from the true value. The Euler method is not a serious method for solving ODEs.

Example: Euler Method

Solve $y' = xy$ subject to the initial condition $x_0 = 0$, $y_0 = 1$.

$$y_{n+1} = y_n + h(x_n y_n)$$

Select a step size $h = 0.1$. Consider required Matlab commands.

```
>> h = 0.1;       set step size to chosen value
>> x = 0;         initialise x
>> y = 1;         initialise y
>> y = y + (h*x*y)    update y, using old x and old y  }
>> x = x + h      update x using old x                 } Repeat
```

Example: Euler Method

Solve $y' = xy$ subject to the initial condition $x_0 = 0$, $y_0 = 1$.

$$y_{n+1} = y_n + h(x_n y_n)$$

Select a step size $h = 0.1$.

n	x_n	y_n	**Analytic solution** $y = \exp(x^2/2)$
0	0	1	1
1	0.1	1	1.0050
2	0.2	1.0100	1.0202
3	0.3	1.0302	1.0460
4	0.4	1.0611	1.0833

First Order ODE: Euler Method

By employing a smaller step size *h* we can achieve a closer approximation for the first few steps. Ultimately however the accuracy degrades. There are modifications of the method which offer improved performance but they are hardly worth the effort. The method is simply not sufficiently accurate in the medium to long term.

Heun's or Second Order Runge-Kutta Method

I will view rescuing the Euler method as a hopeless cause for now. Let us consider instead a proper method, the Runge-Kutta method. At its heart the method is similar to the Euler method, indeed it includes it. We take steps and employ the Taylor series, but on this occasion we look to retain more terms in that series. Again we have

$$y(x_0) = y_0 \,,\; y(x_1) = y_1 \,,\; y(x_2) = y_2 \,,\ldots$$

and look to evaluate $y(x)$ at these special values of $x_n = x_0 + hn$ only.

Second Order Runge-Kutta Method

Again from the Taylor series we have

$$y_{n+1} = y(x_n) + hy'(x_n) + \frac{h^2}{2} y''(x_n) + O(h^3)$$

$$y_{n+1} = y_n + hy'(x_n) + \frac{h^2}{2} y''(x_n) + O(h^3)$$

In this equation of course $\quad y'(x_n) = f(x_n, y_n)$

The "big O" notation $O(h^3)$ is a standard notation which, in this case, denotes terms which involve powers of h of degree 3 or more.

Second Order Runge-Kutta Method

Note that

$$y''(x) = \frac{d}{dx} f(x,y) = \frac{\partial}{\partial x} f(x,y) + \frac{\partial}{\partial y} f(x,y) y'(x)$$

Accordingly

$$y_{n+1} = y_n + hf(x_n, y_n) + \frac{h^2}{2}\left(\left.\frac{\partial f}{\partial x}\right|_{x_n, y_n} + f(x_n, y_n) \left.\frac{\partial f}{\partial y}\right|_{x_n, y_n} \right) + O(h^3)$$

Runge and Kutta (and Heun before them) come up with a very good idea. They look for a quantity which is going to agree with the right hand side of this equation up to order h^2.

Second Order Runge-Kutta Method

Let $y_{n+1} = y_n + (w_1 k_1 + w_2 k_2)$

where $k_1 = hf(x_n, y_n)$

$$k_2 = hf(x_n + \alpha_1 h, y_n + \beta_1 k_1)$$

$$k_2 = hf(x_n, y_n) + \alpha_1 h^2 \left.\frac{\partial f}{\partial x}\right|_{x_n, y_n} + \beta_1 h^2 f(x_n, y_n) \left.\frac{\partial f}{\partial y}\right|_{x_n, y_n} + O(h^3)$$

which gives

$$y_{n+1} =$$

$$y_n + hw_1 f(x_n, y_n) + w_2 \left(hf(x_n, y_n) + \alpha_1 h^2 \left.\frac{\partial f}{\partial x}\right|_{x_n, y_n} + \beta_1 h^2 f(x_n, y_n) \left.\frac{\partial f}{\partial y}\right|_{x_n, y_n} + O(h^3) \right)$$

Second Order Runge-Kutta Method

i.e.

$$y_{n+1} =$$

$$y_n + h(w_1 + w_2) f(x_n, y_n) + h^2 w_2 \left(\alpha_1 \left.\frac{\partial f}{\partial x}\right|_{x_n, y_n} + \beta_1 f(x_n, y_n) \left.\frac{\partial f}{\partial y}\right|_{x_n, y_n} \right) + O(h^3)$$

which agrees up to terms of order h^2 with

$$y_{n+1} = y_n + hf(x_n, y_n) + \frac{h^2}{2} \left(\left.\frac{\partial f}{\partial x}\right|_{x_n, y_n} + f(x_n, y_n) \left.\frac{\partial f}{\partial y}\right|_{x_n, y_n} \right) + O(h^3)$$

if we choose

$$w_1 + w_2 = 1 \ , \ w_2 \alpha_1 = \tfrac{1}{2} \ , \ w_2 \beta_1 = \tfrac{1}{2}$$

Second Order Runge-Kutta Method

For example, if we choose the weights w_1 and w_2 to be equal, then we take

$$w_1 = w_2 = \tfrac{1}{2} \ , \ \alpha_1 = 1 \ , \ \beta_1 = 1$$

So the second order Runge-Kutta becomes

$$y_{n+1} = y_n + \left(\tfrac{1}{2} k_1 + \tfrac{1}{2} k_2 \right)$$

$$k_1 = hf(x_n, y_n) \qquad\qquad k_2 = hf(x_n + h, y_n + k_1)$$

a recursion/difference equation similar to that obtained with the Euler method but ensuring an error in each step depending on h^3, much smaller than that obtained in Euler method if h is small.

Example: Second Order Runge-Kutta

Solve $y' = xy$ subject to the initial condition $x_0 = 0$, $y_0 = 1$.

$$y_{n+1} = y_n + \left(\tfrac{1}{2}k_1 + \tfrac{1}{2}k_2\right) \qquad k_1 = hf\left(x_n, y_n\right)$$

$$k_2 = hf\left(x_n + h, y_n + k_1\right)$$

Select a step size $h = 0.1$.

```
>> h = 0.1;        set step size to chosen value
>> x = 0;          initialise x
>> y = 1;          initialise y
>> k1 = h*x*y;            evaluate k1 using old x and y
>> k2 = h*(x+h)*(y+k1);      evaluate k2 using old x and y
>> y = y + (1/2)*(k1+k2)     update y, using old y
>> x = x + h    update x using old x
```

Repeat

Example: Second Order Runge-Kutta

Solve $y' = xy$ subject to the initial condition $x_0 = 0$, $y_0 = 1$.

$$y_{n+1} = y_n + \left(\tfrac{1}{2}k_1 + \tfrac{1}{2}k_2\right) \qquad k_1 = hf\left(x_n, y_n\right)$$

$$k_2 = hf\left(x_n + h, y_n + k_1\right)$$

Select a step size $h = 0.1$.

n	$\boldsymbol{x_n}$	$\boldsymbol{y_n}$	**Analytic solution** $\boldsymbol{y = \exp(x^2/2)}$
0	0	1	1
1	0.1	1.0050	1.0050
2	0.2	1.0202	1.0202
3	0.3	1.0460	1.0460
4	0.4	1.0832	1.0833

Fourth Order Runge-Kutta Method

Clearly the idea can be extended. The fourth order Runge-Kutta produces a similar matching up to order h^4. The analysis is <u>considerably</u> more tedious but does not involve any additional concepts. We obtain

$$y_{n+1} = y_n + \left(w_1 k_1 + w_2 k_2 + w_3 k_3 + w_4 k_4\right)$$

where

$$k_1 = hf\left(x_n, y_n\right)$$

$$k_2 = hf\left(x_n + \alpha_1 h, y_n + \beta_1 k_1\right)$$

$$k_3 = hf\left(x_n + \alpha_2 h, y_n + \beta_2 k_2\right)$$

$$k_4 = hf\left(x_n + \alpha_3 h, y_n + \beta_3 k_3\right)$$

Fourth Order Runge-Kutta Method

Requiring that we have equation up to order h^4 does not uniquely define the weights w_i or the coefficients α_i or β_i, no more than the equivalent equation did in the second order case. However, a commonly employed solution is:

$$w_1 = \tfrac{1}{6} \ , \ w_2 = \tfrac{2}{6} \ , \ w_3 = \tfrac{2}{6} \ , \ w_4 = \tfrac{1}{6}$$

$$\alpha_1 = \tfrac{1}{2} \ , \ \alpha_2 = \tfrac{1}{2} \ , \ \alpha_3 = 1$$

$$\beta_1 = \tfrac{1}{2} \ , \ \beta_2 = \tfrac{1}{2} \ , \ \beta_3 = 1$$

Fourth Order Runge-Kutta Method

For this special solution we obtain:

$$y_{n+1} = y_n + \tfrac{1}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right)$$

where

$$k_1 = hf\left(x_n, y_n\right)$$

$$k_2 = hf\left(x_n + \tfrac{1}{2}h, y_n + \tfrac{1}{2}k_1\right)$$

$$k_3 = hf\left(x_n + \tfrac{1}{2}h, y_n + \tfrac{1}{2}k_2\right)$$

$$k_4 = hf\left(x_n + h, y_n + k_3\right)$$

Fourth Order Runge-Kutta Method

It should be noted that there is an inconsistency in the notation being employed concerning the Runge-Kutta methods. Some authors would rather write the fourth order Runge-Kutta as:

$$y_{n+1} = y_n + \left(w_1k_1 + w_2k_2 + w_3k_3 + w_4k_4\right)h$$

where

$$k_1 = f\left(x_n, y_n\right)$$

$$k_2 = f\left(x_n + \alpha_1 h, y_n + \beta_1 k_1 h\right)$$

$$k_3 = f\left(x_n + \alpha_2 h, y_n + \beta_2 k_2 h\right)$$

$$k_4 = f\left(x_n + \alpha_3 h, y_n + \beta_3 k_3 h\right)$$

Example: Radiative Cooling

Objects which are hot, i.e. above 0 K, radiate. The total power radiated by an object depends upon its temperature and is given by the Stefan-Boltzmann law:

$$P_{radiated} = \varepsilon\sigma A T^4$$

Here ε is a dimensionless parameter, called the *emissivity*, which determines the radiative efficiency of the object. Emissivity lies between 0 and 1. σ is a universal constant, the Stefan-Boltzmann constant = 5.67x10^{-8} W/m^{2}K^4. A is the surface area of the object. T is the temperature in degrees Kelvin.

Example: Radiative Cooling

If an object is at a temperature T and the surrounding area is at temperature T_{ambient} then the change in internal energy of the object is given by

$$P_{radiated} - P_{absorbed} = -\frac{dE}{dt} = \varepsilon\sigma A T^4 - \varepsilon\sigma A T_{ambient}^4$$

E being the internal energy. The principle of equipartition of energy, assuming three translational degrees of freedom only, gives

$$E = N\tfrac{3}{2}kT$$

where N is the number of molecules/atoms and k is Boltzmann's constant.

Example: Radiative Cooling

Finally, assuming heat loss is by radiation only

$$\frac{dT}{dt} = -\frac{\varepsilon\sigma A}{N\frac{3}{2}k}T^4 + \frac{\varepsilon\sigma A}{N\frac{3}{2}k}T^4_{ambient}$$

Suppose a ball is radiating heat to a surrounding region at 300 K. Let the initial temperature of the ball be 1200 K. Assume:

$$\frac{\varepsilon\sigma A}{N\frac{3}{2}k} = 2.2\times 10^{-12}\ \ \mathrm{s}^{-1}\mathrm{K}^{-2}$$

Example: Radiative Cooling

Solve $T' = -2.2\text{x}10^{-12}(T^4-300^4)$ subject to the initial condition $t_0 = 0$, $T_0 = 1200$ K.

Select a step size $h = 10$.

n	t_n	T_n
0	0	1,200.0
1	10	1,157.7
2	20	1,120.9
3	30	1,088.3
4	40	1,059.3

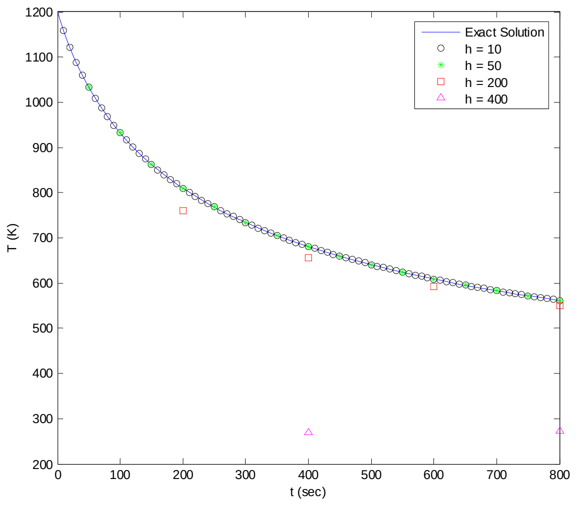
Step-size in Fourth Order Runge-Kutta Method

It is not surprising that the choice of the step size *h* has a significant influence upon the performance of the algorithm. If the step size is set very small then it takes a large number of iterations to acquire an approximate solution to the ODE over the desired time frame. If the step size is set very large then it may take a small number of iterations to obtain an approximate solution over the desired time frame, but this approximation may be very poor. We will investigate these issues by considering the example of radiative cooling.

Example: Radiative Cooling

Solve $T' = -2.2\text{x}10^{-12}(T^4 - 300^4)$ with $t_0 = 0$, $T_0 = 1200\text{K}$.

Here a step size of $h = 400$ is clearly too large, with significant error occurring. A step size of $h = 50$ appears to give good accuracy, good coverage and good efficiency.



Step-size in Fourth Order Runge-Kutta Method

The fundamental ideas in determining whether the step size is small enough are to use a Runge-Kutta of a higher order and compare predictions or to reduce (eg halve) the step size for a Runge-Kutta of the same order and again compare predictions. If the predicted next value of the solution does not significantly change from the original prediction in either case then we may conclude that the step size was in fact sufficiently small. However, the halving of the step size produces a numerically intensive algorithm.

Step-size in Fourth Order Runge-Kutta Method

There are some clever schemes which permit the testing of predicted values by application of higher order Runge-Kutta algorithms in a manner which is rather efficient numerically, meaning that we can reuse at least some of the data generated by the first, lower order Runge-Kutta calculation in the second higher order calculation. An example of such an algorithm is the *Runge-Kutta-Fehlberg method*.

Runge-Kutta Method for higher order ODEs

Although first order ODEs can fairly accurately describe some physical processes this is relatively uncommon. In general physical modelling leads to higher order ODEs. As noted in the case of the simple pendulum above however, these higher order ODEs can commonly be easily transformed to a system of first order ODEs, where in general there will be more than one equation in the system. Accordingly we must consider how to solve such equations by numerical means.

Second Order Runge-Kutta Method for systems of equations

Consider a general system of first order ODEs

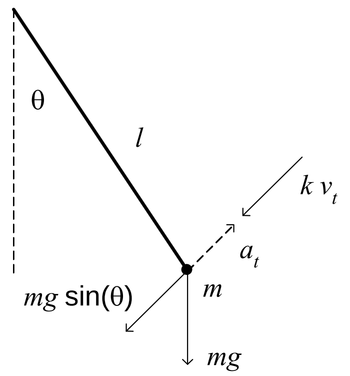
$$y'(x) = F(x,y) = \begin{bmatrix} f_{(1)}(x, y_{(1)}, \ldots, y_{(m)}) \\ \vdots \\ f_{(m)}(x, y_{(1)}, \ldots, y_{(m)}) \end{bmatrix}$$

To avoid clutter I am not explicitly using vector notation here, although y is a vector and F is a vector function.

Example: Pendulum

$$a_t = l\ddot{\theta} \quad , \quad v_t = l\dot{\theta}$$

$$ml\ddot{\theta} = -mg\sin(\theta) - kl\dot{\theta}$$



If we ignore friction then k = 0 so that we obtain the pendulum equation as above.

$$\ddot{\theta} + \left(\frac{g}{l}\right)\sin(\theta) = 0 \qquad \theta(0) = \theta_0 \ , \ \dot{\theta}(0) = 0$$

Example: Pendulum

The second order differential equation is transformed as above to two first order differential equations:

$$\theta' = \omega \qquad \theta(0) = \theta_0$$

$$\omega' = -\left(\frac{g}{l}\right)\sin(\theta) \qquad \omega(0) = 0$$

i.e. by the choice of vector:

$$y = \begin{bmatrix} \theta \\ \omega \end{bmatrix}$$

Second Order Runge-Kutta Method

$$y'(x) = F(x, y) = \begin{bmatrix} f_{(1)}(x, y_{(1)}, \ldots, y_{(m)}) \\ \vdots \\ f_{(m)}(x, y_{(1)}, \ldots, y_{(m)}) \end{bmatrix}$$

As before we let $x_n = x_0 + nh$ and look to evaluate vector y at the special values of x_n only, i.e. we look for $y(x_0) = y_0 \, , \, y(x_1) = y_1 \, , \, y(x_2) = y_2 \, , \ldots$

The Taylor series was the key tool above. But the Taylor series is perfectly well defined for vector functions of many variables, so it remains the key tool in this the more general case.

Second Order Runge-Kutta Method

From the Taylor series we have

$$y_{n+1} = y(x_n) + hy'(x_n) + \frac{h^2}{2} y''(x_n) + O(h^3)$$

$$y_{n+1} = y_n + hy'(x_n) + \frac{h^2}{2} y''(x_n) + O(h^3)$$

In this equation of course

$$y'(x_n) = F(x_n, y_n)$$

So far nothing substantial has changed.

Section: 4 Ordinary differential equations

Second Order Runge-Kutta Method

Note that

$$y''(x) = \frac{d}{dx}F(x,y) = \frac{d}{dx}\begin{bmatrix} f_{(1)}\left(x, y_{(1)}, \dots, y_{(m)}\right) \\ \vdots \\ f_{(m)}\left(x, y_{(1)}, \dots, y_{(m)}\right) \end{bmatrix}$$

$$y''(x) = \begin{bmatrix} \frac{\partial f_{(1)}}{\partial x} + \left(\frac{\partial f_{(1)}}{\partial y_{(1)}}\right) y'_{(1)}(x) + \dots + \left(\frac{\partial f_{(1)}}{\partial y_{(m)}}\right) y'_{(m)}(x) \\ \vdots \\ \frac{\partial f_{(m)}}{\partial x} + \left(\frac{\partial f_{(m)}}{\partial y_{(1)}}\right) y'_{(1)}(x) + \dots + \left(\frac{\partial f_{(m)}}{\partial y_{(m)}}\right) y'_{(m)}(x) \end{bmatrix}$$

Section: 4 Ordinary differential equations

Second Order Runge-Kutta Method

Rewrite as

$$y''(x) = \frac{\partial F(x,y)}{\partial x} + \begin{bmatrix} \frac{\partial f_{(1)}}{\partial y_{(1)}} & \cdots & \frac{\partial f_{(1)}}{\partial y_{(m)}} \\ \vdots & & \vdots \\ \frac{\partial f_{(m)}}{\partial y_{(1)}} & \cdots & \frac{\partial f_{(m)}}{\partial y_{(m)}} \end{bmatrix} \begin{bmatrix} y'_{(1)}(x) \\ \vdots \\ y'_{(m)}(x) \end{bmatrix} = \frac{\partial F(x,y)}{\partial x} + \frac{\partial F(x,y)}{\partial y} y'(x)$$

We obtain from $y'(x) = F(x,y)$

$$y_{n+1} = y_n + hF(x_n, y_n) + \frac{h^2}{2}\left(\left.\frac{\partial F}{\partial x}\right|_{x_n, y_n} + \left(\left.\frac{\partial F}{\partial y}\right|_{x_n, y_n} \right) F(x_n, y_n) \right) + O(h^3)$$

which in terms of notation is very similar to the previous case although more complicated mathematical objects are in fact involved.

Second Order Runge-Kutta Method

As before let $y_{n+1} = y_n + (w_1 k_1 + w_2 k_2)$

where $k_1 = hF(x_n, y_n)$

$$k_2 = hF(x_n + \alpha_1 h, y_n + \beta_1 k_1)$$

$$k_2 = hF(x_n, y_n) + \alpha_1 h^2 \left.\frac{\partial F}{\partial x}\right|_{x_n, y_n} + \beta_1 h^2 \left(\left.\frac{\partial F}{\partial y}\right|_{x_n, y_n} \right) F(x_n, y_n) + O(h^3)$$

which gives

$$y_{n+1} = y_n + hw_1 F(x_n, y_n)$$
$$+ w_2 \left(hF(x_n, y_n) + \alpha_1 h^2 \left.\frac{\partial F}{\partial x}\right|_{x_n, y_n} + \beta_1 h^2 \left(\left.\frac{\partial F}{\partial y}\right|_{x_n, y_n} \right) F(x_n, y_n) + O(h^3) \right)$$

Second Order Runge-Kutta Method

i.e. $y_{n+1} = y_n + h(w_1 + w_2) F(x_n, y_n)$

$$+ h^2 w_2 \left(\alpha_1 \left.\frac{\partial F}{\partial x}\right|_{x_n, y_n} + \beta_1 \left(\left.\frac{\partial F}{\partial y}\right|_{x_n, y_n} \right) F(x_n, y_n) \right) + O(h^3)$$

which agrees up to terms of order h^2 with

$$y_{n+1} = y_n + hF(x_n, y_n) + \frac{h^2}{2} \left(\left.\frac{\partial F}{\partial x}\right|_{x_n, y_n} + \left(\left.\frac{\partial F}{\partial y}\right|_{x_n, y_n} \right) F(x_n, y_n) \right) + O(h^3)$$

if we choose

$$w_1 + w_2 = 1 \ , \ w_2 \alpha_1 = \tfrac{1}{2} \ , \ w_2 \beta_1 = \tfrac{1}{2}$$

Second Order Runge-Kutta Method

$$w_1 + w_2 = 1 \;,\; w_2\alpha_1 = \tfrac{1}{2} \;,\; w_2\beta_1 = \tfrac{1}{2}$$

As before we note and use the special solution:

$$w_1 = w_2 = \tfrac{1}{2} \;,\; \alpha_1 = \beta_1 = 1$$

These are *exactly the same* equations as we obtained in the simple case considered previously. So the second order Runge-Kutta method does not change when we apply it to systems of equations. Remarkably the same statement holds for general Runge-Kutta methods, i.e. those of arbitrary order.

Example: Pendulum

Consider the second order Runge-Kutta for the pendulum equation.

$$y = \begin{bmatrix}\theta\\ \omega\end{bmatrix} = \begin{bmatrix}y_{(1)}\\ y_{(2)}\end{bmatrix} \qquad y' = \begin{bmatrix}y_{(2)}\\ -\frac{g}{l}\sin\left(y_{(1)}\right)\end{bmatrix} \qquad y(0) = \begin{bmatrix}\theta_0\\ 0\end{bmatrix}$$

$$y_{n+1} = y_n + \left(\tfrac{1}{2}k_1 + \tfrac{1}{2}k_2\right)$$

$$k_1 = hF(x_n, y_n) = h\begin{bmatrix}(y_n)_{(2)}\\ -\frac{g}{l}\sin\left((y_n)_{(1)}\right)\end{bmatrix}$$

$$k_2 = hF(x_n + h, y_n + k_1) = h\begin{bmatrix}(y_n + k_1)_{(2)}\\ -\frac{g}{l}\sin\left((y_n + k_1)_{(1)}\right)\end{bmatrix}$$

Example: Pendulum

Let initial angle be 5^o, i.e. 0.0873 rad. Select a step size of h = 0.1. Assume a length of 0.5 m.

```
>> h = 0.1;      set step size to chosen value
>> q = 9.81/0.5;      set parameter q = g/l
>> t = 0;      initialise t
>> y = [0.0873;0];      initialise vector y
>> k1 = h*[y(2);-q*sin(y(1))];      evaluate k1 using old t and y
>> z = y + k1; update intermediate z
>> k2 = h*[z(2);-q*sin(z(1))];      evaluate k2 using old t and z
>> y = y + (1/2)*(k1+k2)      update y, using old y
>> t = t + h      update t using old t
```

Example: Pendulum

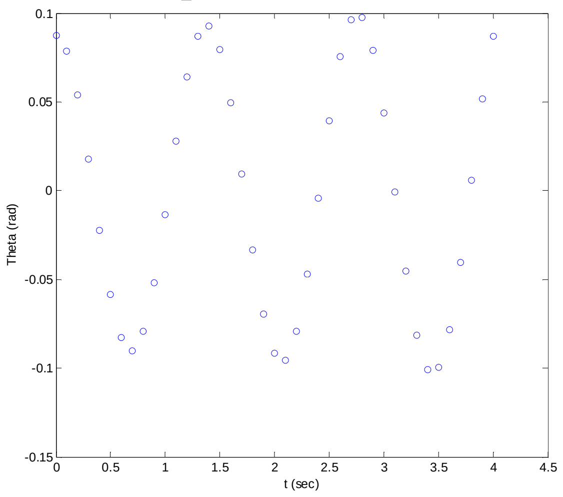
Let initial angle be 5^o, i.e. 0.0873 rad. Select a step size of h = 0.1. Assume a length of 0.5 m.

n	t_n	θ_n	ω_n
0	0	0.0873	0
1	0.1	0.0787	-0.1711
2	0.2	0.0539	-0.3087
3	0.3	0.0178	-0.3842
4	0.4	-0.0224	-0.3813

Example: Pendulum

Consider a longer time frame. Let initial angle be 5^o, i.e. 0.0873 rad. Select a step size of $h = 0.1$.

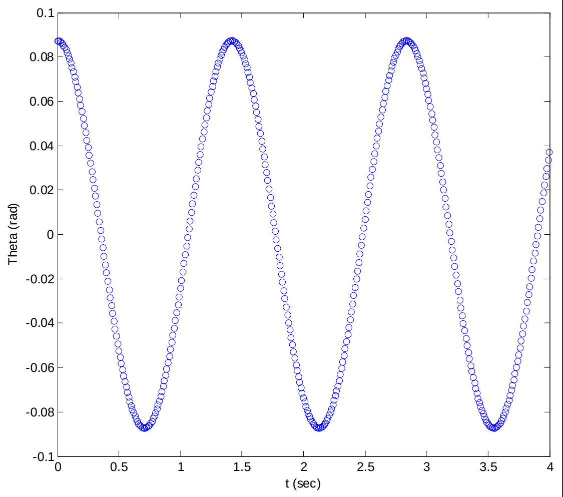
Clearly this step-size does not give suitable coverage. We also see a creep upwards in the peaks, this is false, so this step size is also giving errors.



Example: Pendulum

Again let initial angle be 5^o, i.e. 0.0873 rad. Select a step size of $h = 0.01$.

We have good coverage with this step size and see that the solution is apparently sinusoidal with an amplitude of 0.0873, the initial angle.



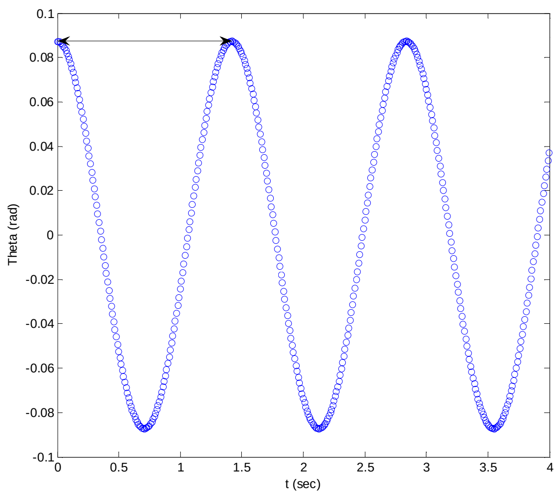
Example: Pendulum

An estimate of the period is 1.42 sec.

The corresponding frequency is

$$\omega_0 = \frac{2\pi}{1.42} \text{ rad/sec}$$

i.e. 4.4248 rad/sec



Example: Linearised Pendulum

The pendulum equation can be simplified through a process called *linearisation*.

$$\ddot{\theta} + \left(\frac{g}{l}\right)\sin(\theta) = 0$$

The pendulum equation obviously has the simple solution $\theta = 0$ where the pendulum hangs down in the vertical position and is unmoving. This type of solution, where nothing is changing, is called an *equilibrium* solution. Engineers might call this an *operating point*. The key idea is to see how the pendulum behaves close to this equilibrium.

Example: Linearised Pendulum

Well, close to equilibrium the angle is small (as is the angular velocity). It is known that for small angle θ

$$\sin(\theta) \cong \theta$$

and the approximation improves as the angle becomes even smaller. Accordingly for small angles, i.e. close to equilibrium, the pendulum equation becomes approximately

$$\ddot{\theta} + \left(\frac{g}{l}\right)\theta = 0$$

Example: Linearised Pendulum

Letting $\omega_0^2 = \frac{g}{l}$ the approximate equation becomes

$$\ddot{\theta} + \omega_0^2\theta = 0.$$

This equation is rather famous, being the equation of simple harmonic motion. Its general solution is also rather famous, being

$$\theta(t) = A\cos(\omega_0 t) + B\sin(\omega_0 t)$$

From the intial conditions $\theta(0) = \theta_0$, $\dot{\theta}(0) = 0$ we have

$\theta_0 = A$, $0 = \omega_0 B$ $\qquad$ $\theta(t) = \theta_0 \cos(\omega_0 t)$

Example: Linearised Pendulum

With $\omega_0^2 = \frac{g}{l}$ we find that the linearised, approximate theory suggests that the pendulum will oscillate sinusoidally with an amplitude equal to the magnitude of the initial angle and with a fixed frequency of oscillation

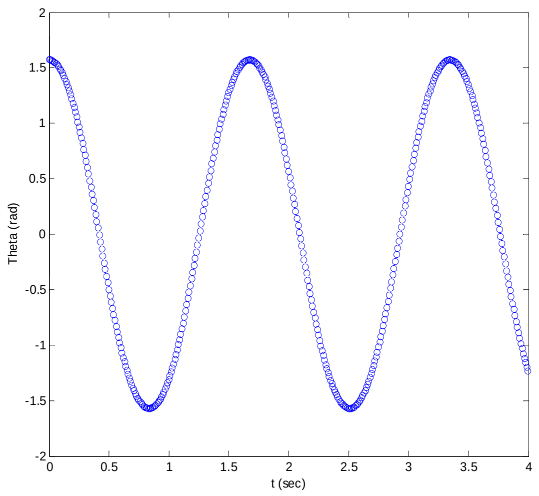
$$\omega_0 = \sqrt{\frac{g}{l}} = \sqrt{\frac{9.81}{0.5}} = 4.4294 \text{ rad/sec}$$

since we assume length of 0.5m. The theory based upon the nonlinear equation disagrees. Whereas it offers a solution which does appear sinusoidal with an amplitude equal to the magnitude of the initial angle the frequency of 4.4248 rad/sec differs.

Example: Pendulum

Let initial angle be 90°, i.e. 1.5708 rad. Select a step size of $h = 0.01$.

The estimated period is 1.67 sec corresponding to a frequency of 3.7624 rad/sec. The frequency is decreasing (very slowly at first) with increasing amplitude.



Example: Pendulum

We see here the first indication of the value of having numerical methods for solving ODEs. It is very commonly the case that we can approximate the equations by linear, constant coefficient equations and solve. But in the case of the pendulum the linear, constant coefficient approximation has failed to observe an *entire phenomenon*, namely that the frequency of oscillation increases as the amplitude decreases. There is a huge price to pay in employing approximate linear models: the set of possible behaviours of these models is not nearly as rich as that of the actual systems.

Example: Pendulum

We will consider the application of the fourth order Runge-Kutta to the pendulum problem, but this time we will give more consideration to visualisation of the solution. Recall the equations:

$$\theta' = \omega \qquad \omega' = -\left(\frac{g}{l}\right)\sin(\theta) \qquad \theta(0) = \theta_0 \qquad \omega(0) = 0 \qquad y = \begin{bmatrix} \theta \\ \omega \end{bmatrix}$$

With a step-size of h the fourth order Runge-Kutta with the special common choices for parameters becomes:

Example: Pendulum

$$y = \begin{bmatrix} \theta \\ \omega \end{bmatrix} = \begin{bmatrix} y_{(1)} \\ y_{(2)} \end{bmatrix} \qquad y' = \begin{bmatrix} y_{(2)} \\ -\frac{g}{l}\sin\left(y_{(1)}\right) \end{bmatrix} \qquad y(0) = \begin{bmatrix} \theta_0 \\ 0 \end{bmatrix}$$

$$y_{n+1} = y_n + \tfrac{1}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right)$$

$$k_1 = h\begin{bmatrix} (y_n)_{(2)} \\ -\frac{g}{l}\sin\left((y_n)_{(1)}\right) \end{bmatrix} \qquad k_2 = h\begin{bmatrix} (y_n + \frac{1}{2}k_1)_{(2)} \\ -\frac{g}{l}\sin\left((y_n + \frac{1}{2}k_1)_{(1)}\right) \end{bmatrix}$$

$$k_3 = h\begin{bmatrix} (y_n + \frac{1}{2}k_2)_{(2)} \\ -\frac{g}{l}\sin\left((y_n + \frac{1}{2}k_2)_{(1)}\right) \end{bmatrix} \qquad k_4 = h\begin{bmatrix} (y_n + k_3)_{(2)} \\ -\frac{g}{l}\sin\left((y_n + k_3)_{(1)}\right) \end{bmatrix}$$

Example: Pendulum

Let initial angle be 5°, i.e. 0.0873 rad. Select a step size of *h* = 0.1 and take a length of 0.5 m as above.

```
>> h = 0.1;       set step size to chosen value
>> q = 9.81/0.5;         set parameter q = g/l
>> t = 0;         initialise t
>> y = [0.0873;0];      initialise vector y
>> k1 = h*[y(2);-q*sin(y(1))];         evaluate k1 using y
>> z = y + 0.5*k1;      update intermediate z
>> k2 = h*[z(2);-q*sin(z(1))];         evaluate k2 using z
>> z = y + 0.5*k2;      update intermediate z
>> k3 = h*[z(2);-q*sin(z(1))];         evaluate k3 using z
```

Example: Pendulum

```
>> z = y + k3; update intermediate z
>> k4 = h*[z(2);-q*sin(z(1))];          evaluate k4 using z
>> y = y + (1/6)*(k1+(2*k2)+(2*k3)+k4);   update y, using old y
>> t = t + h;     update t using old t
```

n	t_n	θ_n	ω_n
0	0	0.0873	0
1	0.1	0.0789	-0.1655
2	0.2	0.0553	-0.2991
3	0.3	0.0210	-0.3751
4	0.4	-0.0173	-0.3788

Example: Pendulum

One thing to note is that the values *have changed* from those found by employing the second order Runge-Kutta. The changes are relatively small, nevertheless they indicate that one (or both) of the methods is (are) in error. A second thing to note is that determining the approximate solution in this manner, namely using the command line, is rapidly becoming tedious and will become unworkable as the number of time steps to be taken increases. There are two clear options for overcoming this growing problem.

For loop: Pendulum

Option 1: use a **for** loop.

```
>> h = 0.1; , q = 9.81/0.5; t = 0; , y = [0.0873;0];      initialise
>> N = 100;   set number of time-steps required
>> Storage = zeros(N+1,3);   set aside space (filled with zeros
initially) in memory to store N+1 triplets of numbers (t,y(1),y(2))
```

Note the use of the **,** command to separate individual commands entered on one line in the command window. This will save space on future slides and is a style adopted by many practitioners. Note also that I wish to generate 101 solutions at 101 times, starting with initial conditon at time t = 0. It is a *far more efficient* procedure to have Matlab allocate the space to store these 101 solutions at the outset.

For loop: Pendulum

Option 1: use a **for** loop.

```
>> Storage(1,:) = [0 y(1) y(2)];  place initial value in first row of Storage
>> for count = 1:N
k1 = h*[y(2);-q*sin(y(1))]; , z = y + 0.5*k1;
k2 = h*[z(2);-q*sin(z(1))]; , z = y + 0.5*k2;
k3 = h*[z(2);-q*sin(z(1))]; , z = y + k3;
k4 = h*[z(2);-q*sin(z(1))];
y = y + (1/6)*(k1+(2*k2)+(2*k3)+k4);
t = t + h;
Storage(count+1,:) = [t y(1) y(2)];
end
```

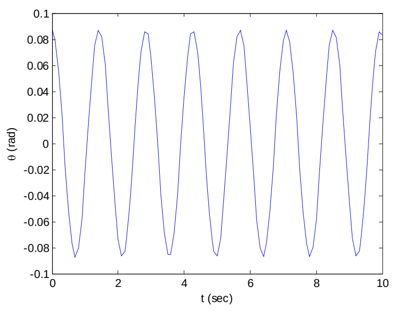
For loop: Pendulum

I note that when I enter the **for** command (which Matlab automatically colours blue) at the command line and hit return the prompt >> does not appear on the next line. This continues for all of the lines up to the point where I type the **end** command (which Matlab also automatically colours blue) . This command causes the for loop to be executed and, when complete, the prompt >> returns.

For loop: Pendulum

I note that the time values are stored in the first column of matrix **Storage** and the angle values in the second column.

```
>> plot(Storage(:,1), Storage(:,2);      plot angle vs time
>> xlabel('t (sec) ')  ,  ylabel('\theta (rad) ')
```



Note method for achieving θ symbol in y-axis label. Note also the rather “cornered” maxima and minima suggesting that step size is too large.

M-files

Option 2: use a **function M-file**. When the number of commands starts to become large it starts to make sense to consider collecting these commands together, in short it is time to use an **M-file**. M-files are collections of commands which can be grouped together and simply run in a batch. You can also take the opportunity to create a more general solution, allowing for example a different choice for *N* and *h*. The **New M-File** button on the toolbar opens up the text editor for writing new M-files. Alternatively select **New** from the **File** menu. The **Open file** button opens the text editor to faciliate editing of an old M-file. Alternatively select **Open…** from the **File** menu.

Function M-files

M-files can be *functions* which accept input arguments and produce as a result output arguments, or they can be *scripts* which execute a series of Matlab commands. We will consider function M-files only. Function M-files have the excellent property that, once written, they become indistinguishable from any other Matlab function. In short the user is permitted to extend the Matlab language. Just as Matlab is, you should therefore be careful in your choice of names for function M-files, picking names which at least somewhat describe what the function M-file does, rather than meaningless names like **f**. Likewise, just as regular Matlab functions have help files which are printed by means of the **help** command, you can and should write similar help files which will similarly be accessed for your functions.

Function M-files

I wrote a function M-file to collect all of the commands used for solving the pendulum equation with the special initial conditions considered above and gathering the solutions into an (N+1) x 3 matrix. Here it is:

```
function  [Storage]= PendulumEquation1(len,h,theta0,N)
%  Storage = PendulumEquation1 (len,h,theta0,N): produces matrix of solutions of idealised pendulum
%  equation for pendulum of length len, with initial angle theta0 and initial angular velocity of zero.
%  h is the time-step used. N+1 is number of time-steps for which the solution is generated.


g = 9.81;   % set acceleration due to gravity
q = g/len;   %  set parameter q
t = 0;
y = [theta0;0];     % initialise time and state vector
Storage = zeros(N+1,3);  % set aside space (filled with zeros initially) in memory to store N+1 triplets of
%  numbers (t,y(1),y(2))
Storage(1,:) = [0 y(1) y(2)];  % place initial value in first row of Storage
```

Function M-files

```
for count = 1:N
   k1 = h*[y(2);-q*sin(y(1))]; , z = y + 0.5*k1;
   k2 = h*[z(2);-q*sin(z(1))]; , z = y + 0.5*k2;
   k3 = h*[z(2);-q*sin(z(1))]; , z = y + k3;
   k4 = h*[z(2);-q*sin(z(1))];
   y = y + (1/6)*(k1+(2*k2)+(2*k3)+k4);
   t = t + h;
   Storage(count+1,:) = [t y(1) y(2)];
end
```

I saved this function M-file as PendulumEquation1.m in my work folder. In this way I can achieve everything which we did above to generate matrix **Storage** with the single command:

```
>> Storage = PendulumEquation1(0.5,0.1,0.0873,100);
```

Function M-files

Note how the function M-file is created using the **function** command. Note how the input arguments are passed to it and how the output arguments are passed from it. Note how the help file is created, it comprises all of the text that follows the function definition line up to the first blank line. This text is preceeded by the % symbol so that Matlab knows that it comprises commentary not commands. The text editor automatically colours text following the % symbol and the % symbol itself green. It also automatically colours the commands **function**, **for** and **end** blue. Lastly it automatically indents the commands between the **for** and **end** commands. The help file for my function is available as for any other Matlab function:

```
>> help PendulumEquation1
```

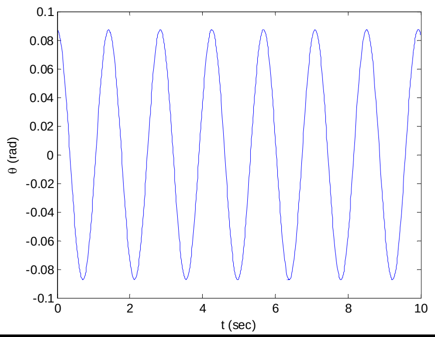
Function M-files

In creating this function M-file I took the opportunity afforded to write something more general than we considered above. It will work for any length, *len*, of pendulum, for any choice (within reason) of *N*, the number of time-steps over which the simulation is to be carried out, and for any choice (within reason) of time-step *h*. It does assume a particular form for the initial condition, namely that the initial angular velocity is zero, but the initial angle is not restricted to 0.0873. There is one special number embedded in the code: 9.81 is the acceleration in m/s^2 due to gravity. Although 9.81 is not a universally acceptable value the number is named and set at the very start of the function M-file.

Visualisation: Pendulum

As noted the plot achieved with a time step of 0.1 is a little course. Now that I have my function M-file I can reduce this:

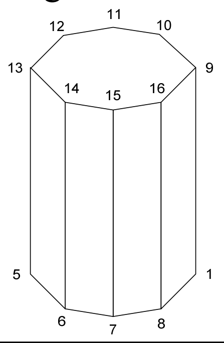
```
>> Storage = PendulumEquation1(0.5,0.01,0.0873,1000);
>> plot(Storage(:,1),Storage(:,2))
>> xlabel('t (sec) ') , ylabel('\theta (rad)')
```



Visualisation: Pendulum

This visualisation of the solution is fine, but it is not particularly interesting. One way of presenting the data which can look well (if done properly) is to use 3D models and to animate. Firstly we must create a 3D model and we will do this using the **patch** command. By way of example let us try to make a 3D model of the following:

The object is a solid polygon with 16 vertices (labeled 1-16) and 10 faces (including top and bottom). We also consider drawing this object in the second introductory laboratory.



Visualisation: Pendulum

To begin we must come up with (x,y,z) co-ordinates for the 16 vertices and we will place these in a 16x3 matrix which we will call vertex_matrix. Each vertex corresponds to one row of this matrix, where its x,y,z co-ordinates are recorded.

```
>> theta = [0:2*pi/8:(2*pi)-(2*pi/8)]';  angles of 8 points uniformly distributed around unit circle expressed as column vector

>> vertices_bottom = [cos(theta)  sin(theta)  zeros(8,1)];

>> vertices_top = [cos(theta)  sin(theta)  ones(8,1)];

>> vertex_matrix = [vertices_bottom; vertices_top];
```

Visualisation: Pendulum

Now we must describe the 10 faces of the object by listing which vertices belong to each face. We record this information in a 10x8 matrix, **face_matrix**. We require 10 rows because each row will correspond to one face and there are 10 faces. We require 8 columns because the faces with the largest number of vertices (the top and bottom) have 8 vertices. The faces are:

Note how we cope with the different number of vertices in different faces.

1	2	3	4	5	6	7	8
1	2	10	9	9	9	9	9
2	3	11	10	10	10	10	10
3	4	12	11	11	11	11	11
4	5	13	12	12	12	12	12
5	6	14	13	13	13	13	13
6	7	15	14	14	14	14	14
7	8	16	15	15	15	15	15
8	1	9	16	16	16	16	16
9	10	11	12	13	14	15	16

Visualisation: Pendulum

A relatively elegant description of the matrix, **face_matrix**, is available in Matlab code:

```
>> face_matrix = [[1:8];[1:8]' [[2:8] 1]' [(8+[2:8]) 9]'
((8+[1:8])')*ones(1,5);8+[1:8]];
```

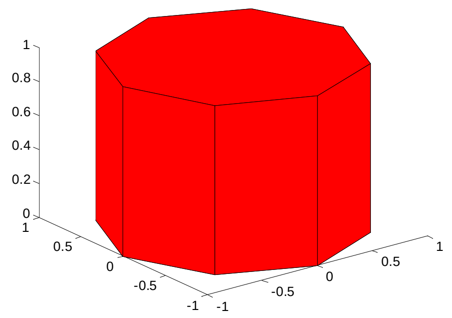
We now employ the **patch** command where I have opted for the colour red, [RGB] = [1 0 0].

```
>> patch('Vertices',vertex_matrix, 'Faces',face_matrix, 'FaceColor',[1
0 0]);

>> view(3)
```

The default view is 2D, so to see the object in 3D I use the **view(3)** command.

Visualisation: Pendulum

The result is as follows:



I do not like this. I dislike the lines showing the face boundaries and I dislike the cartoon-like colouring.

Visualisation: Pendulum

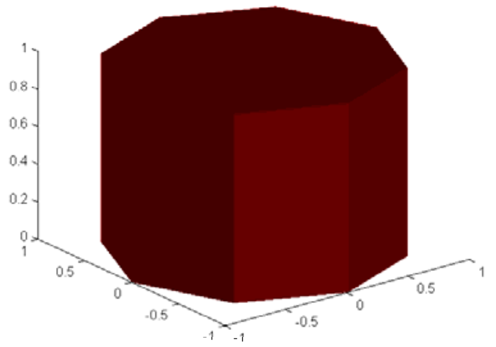
To improve matters I set a few parameters while using the **patch** command:

```
>> patch('Vertices',vertex_matrix, 'Faces',face_matrix, 'FaceColor',[1 0 0], 'LineStyle', 'none',);

>> view(3)

>> light('Position',[-1 -1 -1])
```

Setting LineStyle to none turns off the boundary lines. **view(3)** again shows the object in 3D. **light** lights the object from a single light source at infinity in the direction [-1,-1,-1], as specified by setting the Position.

Visualisation: Pendulum

The result is as follows (having set Copy Options … in the Edit menu of the Figure window to Force white background):



I find something like this to be more realistic. Realism can be improved by adding additional light sources and adjusting a variety of object properties which determine its reflectivity.

Visualisation: Pendulum

For the 3D model of the pendulum we will work with three patch objects, each of which is approximately a cylinder. Each is generated by code very similar to the previous, but with a much larger number of vertices (100) on each end and with the orientation of two of them (the bracket and the bob) being changed. Since there is rather a lot of code, we put it in a function M-file.

Visualisation: Pendulum

```
function  DrawPendulum(len)
% DRAWPENDULUM(len): Draw pendulum at angle 0.

g = 9.81;  % set universal gravitational constant
N = 100;  % number of points on circluar cross sections
r_rod = 0.05;
r_bob = 4*r_rod; , depth_bob = 0.3;
r_brack = 2*r_rod; , depth_brack = 0.4;

phi = [0:2*pi/N:(2*pi)-(2*pi/N)]';  % angle data
```

Visualisation: Pendulum

```
% create data for cylindrical patch representing rod
vertices1_rod = [r_rod*cos(phi) r_rod*sin(phi) zeros(N,1)];
vertices2_rod = [r_rod*cos(phi) r_rod*sin(phi) -len*ones(N,1)];
vertices_rod = [vertices1_rod;vertices2_rod];


% create data for cylindrical patch representing bob
vertices1_bob = [depth_bob*0.5*ones(N,1) r_bob*cos(phi) -
len+(r_bob*sin(phi))];
vertices2_bob = [-depth_bob*0.5*ones(N,1) r_bob*cos(phi) -
len+(r_bob*sin(phi))];
vertices_bob = [vertices1_bob;vertices2_bob];
```

Visualisation: Pendulum

```
% create data for cylindrical patch representing bracket
vertices1_brack = [depth_brack*0.5*ones(N,1) r_brack*cos(phi)
r_brack*sin(phi)];
vertices2_brack = [-depth_brack*0.5*ones(N,1) r_brack*cos(phi)
r_brack*sin(phi)];
vertices_brack = [vertices1_brack;vertices2_brack];


% all three objects have same faces
faces_all = [[1:N];[1:N]' [[2:N] 1]' [(N+[2:N]) N+1]'
((N+[1:N])')*ones(1,N-3);N+[1:N]];
```

Visualisation: Pendulum

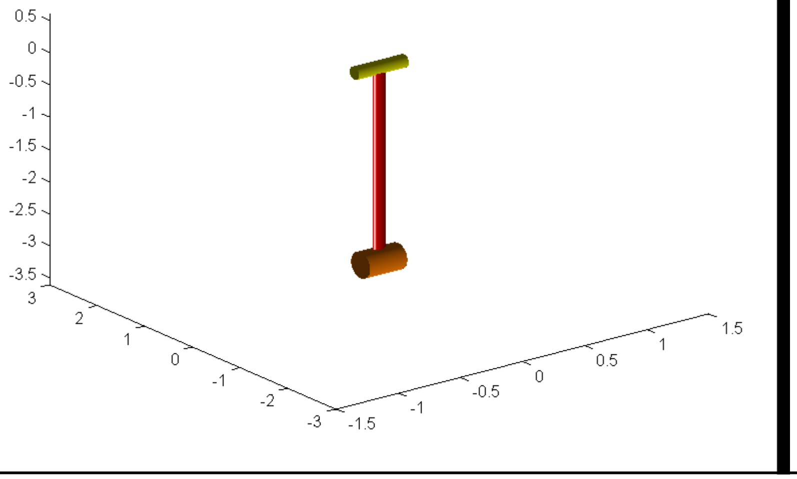
```
handlefig = figure(1);
axis([-1.5*len 1.5*len -1.5*len 1.5*len -1.5*len 1.5*len])
light('Position',[-1 -1 -1]);
handlepatch_brack =
patch('Vertices',vertices_brack,'Faces',faces_all,'FaceColor',[1 1
0],'LineStyle','none');
handlepatch_rod =
patch('Vertices',vertices_rod,'Faces',faces_all,'FaceColor',[1 0
0],'LineStyle','none');
handlepatch_bob =
patch('Vertices',vertices_bob,'Faces',faces_all,'FaceColor',[1 0.5
0],'LineStyle','none');
```

Visualisation: Pendulum

A few extra commands have been included here. The **figure(1)** command tells Matlab to create a new figure window. Within Matlab a figure window is an example of a *graphics object*. The three patch objects (bracket, rod and bob) are also graphics objects. It can often be the case that you will need to determine or change the properties of a graphics object (its colour for example). To do so you need the objects *handle*, which is just a long string of numbers which Matlab uses to identify the object. Although it is possible to discover the handle of a graphics object long after it has been created, it is much easier to record the handle at the same time as you create the object. This is what is being done in the previous slide with the various handles being stored as **handle_fig**, **handlepatch_brack**, *etc*.

Visualisation: Pendulum

```
>> Drawpendulum(3)
```

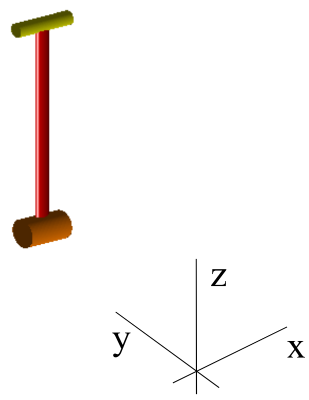


Visualisation: Pendulum

The **axis** command sets the x, y and z-axis ranges, morover since it specifies three axes it usually negates the requirement for the **view(3)** command which we previously employed. When drawn the pendulum looks fine in Matlab, but copying over to Powerpoint introduces the annoying artefact of a black line along the right hand edge. In general showing the axes produces a litany of similar annoyances. We may edit Drawpendulum.m, placing the command **axis off** immediately after the light command. This eliminates the artefact.

Visualisation: Pendulum

```
>> Drawpendulum(3)
```



Visualisation: Pendulum

To animate it will be necessary to modify the Drawpendulum.m function M-file so that it will draw the 3D model of the pendulum at different angles. The key observation here is to realize that to rotate the pendulum through θ in the y-z plane we have to rotate every vertex through θ in this plane. The faces do not change. Moreover, all vertices have been specified by their (x,y,z) co-ordinates so to rotate in the y-z plane we just multiply by the rotation matrix:

$$R_\theta = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\theta) & \sin(\theta) \\ 0 & -\sin(\theta) & \cos(\theta) \end{bmatrix} \qquad \begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = R_\theta \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

Visualisation: Pendulum

Given this observation it is seen that a relatively small modification to the function M-file, Drawpendulum.m, permits the drawing of the pendulum at any desired angle of rotation in the y-z plane. We create a new function M-file, Drawpendulum2.m to facilitate drawing the pendulum at any desired angle of rotation in the y-z plane.

Visualisation: Pendulum

```
function  DrawPendulum2(len,theta)
% DRAWPENDULUM2(len,theta): Draw pendulum at angle theta.


g = 9.81;  % set universal gravitational constant
N = 100;  % number of points on circluar cross sections
r_rod = 0.05;
r_bob = 4*r_rod; , depth_bob = 0.3;
r_brack = 2*r_rod; , depth_brack = 0.4;


phi = [0:2*pi/N:(2*pi)-(2*pi/N)]';  % angle data
```

Visualisation: Pendulum

```
% create data for cylindrical patch representing rod

vertices1_rod = [r_rod*cos(phi) r_rod*sin(phi) zeros(N,1)];

vertices2_rod = [r_rod*cos(phi) r_rod*sin(phi) -len*ones(N,1)];

vertices_rod = [vertices1_rod;vertices2_rod];


% create data for cylindrical patch representing bob

vertices1_bob = [depth_bob*0.5*ones(N,1) r_bob*cos(phi) -
len+(r_bob*sin(phi))];

vertices2_bob = [-depth_bob*0.5*ones(N,1) r_bob*cos(phi) -
len+(r_bob*sin(phi))];

vertices_bob = [vertices1_bob;vertices2_bob];
```

Visualisation: Pendulum

```
% create data for cylindrical patch representing bracket

vertices1_brack = [depth_brack*0.5*ones(N,1) r_brack*cos(phi)
r_brack*sin(phi)];

vertices2_brack = [-depth_brack*0.5*ones(N,1) r_brack*cos(phi)
r_brack*sin(phi)];

vertices_brack = [vertices1_brack;vertices2_brack];


% all three objects have same faces

faces_all = [[1:N];[1:N]' [[2:N] 1]' [(N+[2:N]) N+1]'
((N+[1:N])')*ones(1,N-3);N+[1:N]];
```

Visualisation: Pendulum

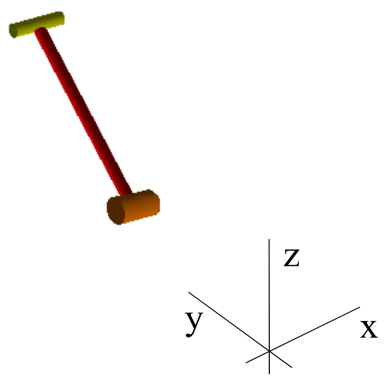
```
% create matrix of rotation in y-z plane through theta
Rtheta = [1 0 0; 0 cos(theta) sin(theta);0 –sin(theta) cos(theta)];
% rotate rod and bob by rotating their vertices
for count = 1:2*N
  vertices_rod_rotated(count,:) = (Rtheta*(vertices_rod(count,:)'))';
end

for count = 1:2*N
  vertices_bob_rotated(count,:) = (Rtheta*(vertices_bob(count,:)'))';
end
```

Visualisation: Pendulum

```
handlefig = figure(1);
axis([-1.5*len 1.5*len -1.5*len 1.5*len -1.5*len 1.5*len])
light('Position',[-1 -1 -1]); , axis off
handlepatch_rod =
patch('Vertices',vertices_rod_rotated,'Faces',faces_all,'FaceColor',[1 0
0],'LineStyle','none');
handlepatch_bob =
patch('Vertices',vertices_bob_rotated,'Faces',faces_all,'FaceColor',[1
0.5 0],'LineStyle','none');  % use rotated vertices
handlepatch_brack =
patch('Vertices',vertices_brack,'Faces',faces_all,'FaceColor',[1 1
0],'LineStyle','none'); % use rotated vertices
```

Visualisation: Pendulum

```
>> Drawpendulum2(3,pi/4)
```



Visualisation: Pendulum

We may combine the two function M-files, PendulumEquation1.m and Drawpendulum2.m to create a 3D animation. Although other methods are available I find the following to be effective. Create a new function M-file Drawpendulum3.m based upon Drawpendulum2.m. The new M-file will have an output M and two inputs, len, the length of the pendulum and Angles, a vector of the required angles at which the pendulum is to be drawn. Accordingly the first line of the M-file is:

```
function  M = DrawPendulum3(len,Angles)
```

Visualisation: Pendulum

As the first command after the help file we find the number of angles for which the pendulum is to be drawn thus:

```
FrameNumber = length(Angles);
```

The commands in Drawpendulum3.m are now identical to those of Drawpendulm2.m down to the setting up of rotation matrix Rtheta. We must now start a for loop which allows us to undertake the task of drawing the pendulum for each required angle in turn.

Visualisation: Pendulum

```
for  angle_count = 1:FrameNumber
  theta = Angles(angle_count);
  Rtheta = [1 0 0; 0 cos(theta) sin(theta);0 –sin(theta) cos(theta)];
  for count = 1:2*N
      vertices_rod_rotated(count,:) = (Rtheta*(vertices_rod(count,:)'))';
  end
  for count = 1:2*N
     vertices_bob_rotated(count,:) = (Rtheta*(vertices_bob(count,:)'))';
  end
```

Visualisation: Pendulum

```
        handlefig = figure('Position ',[100 100 850 600]);
        axis([-1.5*len 1.5*len -1.5*len 1.5*len -1.5*len 1.5*len])
        light('Position',[-1 -1 -1]); , axis off

        handlepatch_brack =
        patch('Vertices',vertices_brack,'Faces',faces_all,'FaceColor',[1 1
        0],'LineStyle','none');

        handlepatch_rod =
        patch('Vertices',vertices_rod_rotated,'Faces',faces_all,'FaceColor',[1 0
        0],'LineStyle','none');  % use rotated vertices

        handlepatch_bob =
        patch('Vertices',vertices_bob_rotated,'Faces',faces_all,'FaceColor',[1 0.5
        0],'LineStyle','none');  % use rotated vertices

        M(angle_count) = getframe(handlefig);

        close(handlefig)

end
```

Visualisation: Pendulum

The key command here is the **getframe** command. This gets the current figure and stores it as a frame in a movie. The **figure** command is also significant. Specifying exactly where the figure window is to open, by setting its 'Position' property, eliminates artefacts. Also finding the unique identifier for the figure window, the *figure handle*, and employing this with the **getframe** command ensures that we indeed get the correct frame. Finally the **close** command terminates the figure window once we are finished with it preventing the simultaneous opening of dozens (hundreds perhaps) of such windows which can lead to a significant number of problems.

Visualisation: Pendulum

Finally at the command line execute the commands:

```
>> Storage = PendulumEquation1(3,0.1,pi/4,100);
>> M = Drawpendulum3(3,Storage(count,2));
```

These commands create 101 movie frames for a Matlab movie which can be played using the **movie** command. To suppress artefacts I recommend:

```
>>  figure(' Position ',[100 100 850 600]); , axis off
>>  movie(M);     play movie once at 12 frames per second default
```

Visualisation: Pendulum

In older Matlab versions, although the **movie** command does play the movie once at the default 12 frames per second it first loads the frames, so it appears to play the movie twice, once rather quickly and once again at a better pace. If you wish to actually play the movie just the once you should use some other package (e.g. Windows Media Player). The movie must first be converted to a more universal format. Recent Matlab versions permit conversion to Audio/Video Interleave (.avi) format using the **movie2avi** command.

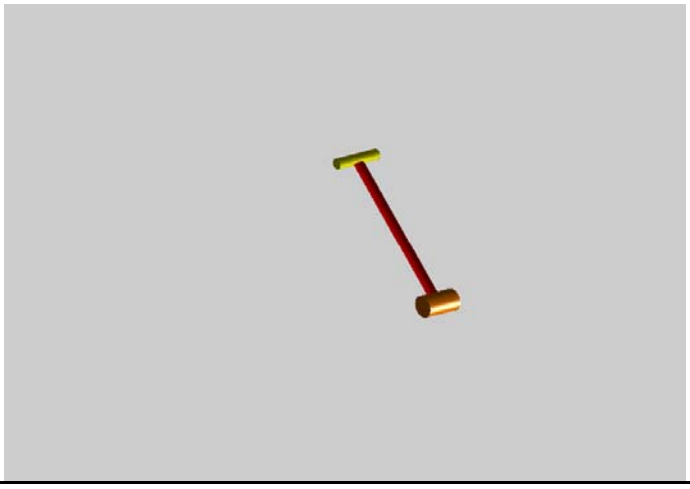
```
>> movie2avi(M, 'Pendulum1.avi')    convert Matlab movie to .avi
format and store in file Pendulum1.avi.
```

Windows Media Player will commonly have problems playing the file if compression is used because the necessary codec (**co**mpression/**dec**ompression software) is unavailable.

Visualisation: Pendulum

To avoid problems convert without compression:

```
>> movie2avi(M,'Pendulum1.avi', 'compression', 'none');
```



Multi-step Methods

Fundamentally the Euler and Runge-Kutta methods base their estimation of the value of the solution in the next step on information acquired in the previous step only. Information from steps preceding the previous step is essentially discarded. A rather general principle, not just in the area of numerical methods, is that if a method aspires to be the best possible then it should not discard useful information. Multi-step methods for solving ODEs strive to be better by employing information from more than one previous step in their estimation of the solution in the next step.

Linear Multi-step Methods

Let us start again by considering first order ODE $y' = f(x,y)$. A number of methods propose to offer an approximate solution to this ODE by letting $x_n = x_0+nh$, h being the step size and looking to approximately evaluate y at the special values of x_n only, i.e. looking for

$$y(x_0) = y_0 , y(x_1) = y_1 , y(x_2) = y_2 , \ldots$$

To approximately evaluate y_n the methods of this class offer the following general equation:

$$y_n + \alpha_1 y_{n-1} + \cdots + \alpha_s y_{n-s} = h\beta_1 f(x_{n-1}, y_{n-1}) + \cdots + h\beta_s f(x_{n-s}, y_{n-s})$$

Linear Multi-step Methods

The difference between the methods of this class is due to different choices of s and different choices for coefficients α_i and β_i . Although the ideas behind the methods are far more complicated it is possible to at least in part justify the choices made by a relatively simple procedure.

To this end consider the special case where $s = 1$.

$$y_n + \alpha_1 y_{n-1} = h\beta_1 f(x_{n-1}, y_{n-1})$$

We ask that this approximate expression for $y(x)$ be exact if y is a polynomial of degree 0 or 1 in x.

Linear Multi-step Methods

$$y_n + \alpha_1 y_{n-1} = h\beta_1 f(x_{n-1}, y_{n-1}) = h\beta_1 y'(x_{n-1})$$

So the formula must be exact if $y(x) = c_0 + c_1 x$, for arbitrary c_0 and c_1 .

$$(c_0 + c_1 x_n) + \alpha_1 (c_0 + c_1 (x_n - h)) = h\beta_1 c_1$$

$$c_0 + \alpha_1 c_0 = 0 \qquad c_1 x_n + \alpha_1 c_1 (x_n - h) = h\beta_1 c_1$$

$$1 + \alpha_1 = 0 \qquad x_n + \alpha_1 (x_n - h) = h\beta_1$$

These are the same equations obtained if we ask that the formula be exact for $y(x) = 1$ and $y(x) = x$.

Linear Multi-step Methods

$$\alpha_1 = -1 \qquad \alpha_1(-h) = h\beta_1 \qquad \beta_1 = 1$$

$$y_n - y_{n-1} = hf(x_{n-1}, y_{n-1})$$

$$y_n = y_{n-1} + hf(x_{n-1}, y_{n-1})$$

and we recognise that this formula is identical to the Euler method apart from a trivial shift by 1 of the index.

Linear Multi-step Methods

For general *s* it can become considerably more tedious to evaluate appropriate coefficients. A useful set of methods emerges however if we assume $\alpha_1 = -1$, $\alpha_2 = 0$, … , $\alpha_s = 0$, i.e.

$$y_n - y_{n-1} = h\beta_1 f(x_{n-1}, y_{n-1}) + \cdots + h\beta_s f(x_{n-s}, y_{n-s})$$

Again we obtain good choices for coefficients β_i by requiring that the formula be exact if *y* is a polynomial of degree 0, 1, 2, … , *s* in *x*. The linearity of the equation, as before, means that we may alternatively assume *y* to equal 1, *x*, x^2, … , x^s.

Linear Multi-step Methods

For $s = 2$

$$y_n - y_{n-1} = h\beta_1 y'(x_{n-1}, y_{n-1}) + h\beta_2 y'(x_{n-2}, y_{n-2})$$

For $y(x) = 1$ $\quad 1 - 1 = 0$

For $y(x) = x$ $\quad x_n - (x_n - h) = h\beta_1 + h\beta_2$

$$1 = \beta_1 + \beta_2$$

For $y(x) = x^2$

$$x_n^2 - (x_n - h)^2 = h\beta_1 2(x_n - h) + h\beta_2 2(x_n - 2h)$$

$$1 = 2\beta_1 + 4\beta_2$$

Linear Multi-step Methods

Solving gives $\beta_1 = \frac{3}{2}$, $\beta_2 = -\frac{1}{2}$

The resulting recursion becomes:

$$y_n = y_{n-1} + h\left(\tfrac{3}{2} f\left(x_{n-1}, y_{n-1}\right) - \tfrac{1}{2} f\left(x_{n-2}, y_{n-2}\right)\right)$$

This recursion is called the *2-step Adams-Bashforth method*. A virtually identical argument with $s = 4$ leads to the *4-step Adams-Bashforth method*:

$$y_n = y_{n-1} + h\left(\tfrac{55}{24} f\left(x_{n-1}, y_{n-1}\right) - \tfrac{59}{24} f\left(x_{n-2}, y_{n-2}\right) + \tfrac{37}{24} f\left(x_{n-3}, y_{n-3}\right) - \tfrac{9}{24} f\left(x_{n-4}, y_{n-4}\right)\right)$$

Linear Multi-step Methods

$$y_n = y_{n-1} + h\left(\tfrac{55}{24} f\left(x_{n-1}, y_{n-1}\right) - \tfrac{59}{24} f\left(x_{n-2}, y_{n-2}\right) + \tfrac{37}{24} f\left(x_{n-3}, y_{n-3}\right) - \tfrac{9}{24} f\left(x_{n-4}, y_{n-4}\right)\right)$$

It is clear that the 4-step Adams-Bashforth method is not *self-starting*. It can predict y_4 and subsequently y_5 *etc.* provided y_0, y_1, y_2 and y_3 are given. Of course only the first of these actually is given. Accordingly we must augment the 4-step Adams-Bashforth method with some other method which can offer us approximations to the missing starting values y_1, y_2 and y_3 . In this respect the Runge-Kutta methods are obvious choices.

Predictor-Corrector Methods

Another very good idea in the area of numerically solving ODEs by employing linear multi-step methods is the idea of employing *two such methods*. The first method is called the *predictor*. It offers an approximation or prediction for the value of y at x_n. Let us denote this predicted value by $y^{(p)}{}_n$. The key new idea is that, given this prediction we may evaluate $f(x_n, y^{(p)}{}_n)$. It offers the prospect of employing a new recursion to obtain a corrected approximation

$$y_n + \alpha_1 y_{n-1} + \cdots + \alpha_s y_{n-s} = h\beta_1 f\left(x_n, y_n^{(p)}\right) + \cdots + h\beta_s f\left(x_{n+1-s}, y_{n+1-s}\right)$$

Corrector Methods

Again the difference between corrector methods of this class is due to different choices of s and different choices for coefficients α_i and β_i. Again we may in part justify the choices made by a relatively simple procedure.

To this end consider the special case where $s = 2$.

$$y_n + \alpha_1 y_{n-1} + \alpha_2 y_{n-2} = h\beta_1 f\left(x_n, y_n^{(p)}\right) + h\beta_2 f\left(x_{n-1}, y_{n-1}\right)$$

We ask that this approximate expression for $y(x)$ be exact if y is a polynomial of degree 0, 1 or 2 in x and note that, as above, this is so if and only if it holds for $y(x) = 1$, $y(x) = x$ and $y(x) = x^2$.

Corrector Methods

$$y_n + \alpha_1 y_{n-1} + \alpha_2 y_{n-2} = h\beta_1 y'(x_n) + h\beta_2 y'(x_{n-1})$$

As before we simplify by assuming $\alpha_2 = 0$.

For $y(x) = 1$ $\quad 1 + \alpha_1 = 0 \quad \alpha_1 = -1$

For $y(x) = x$ $\quad x_n + \alpha_1(x_n - h) = h\beta_1 + h\beta_2$

$$\beta_1 + \beta_2 = 1$$

For $y(x) = x^2$

$$x_n^2 + \alpha_1(x_n - h)^2 = h\beta_1 2x_n + h\beta_2 2(x_n - h)$$

$$2\beta_2 = 1$$

Corrector Methods

Solving gives $\quad \beta_1 = \frac{1}{2} \quad , \quad \beta_2 = \frac{1}{2}$

The resulting recursion equation becomes

$$y_n = y_{n-1} + h\left(\tfrac{1}{2} f\left(x_n, y_n^{(p)}\right) + \tfrac{1}{2} f\left(x_{n-1}, y_{n-1}\right)\right)$$

This recursion is called the *2-step Adams-Moulton method*. A virtually identical argument with $s = 4$ leads to the *4-step Adams-Moulton method*:

$$y_n = y_{n-1} +$$
$$h\left(\tfrac{9}{24} f\left(x_n, y_n^{(p)}\right) + \tfrac{19}{24} f\left(x_{n-1}, y_{n-1}\right) - \tfrac{5}{24} f\left(x_{n-2}, y_{n-2}\right) + \tfrac{1}{24} f\left(x_{n-3}, y_{n-3}\right)\right)$$

Linear Multi-step Methods

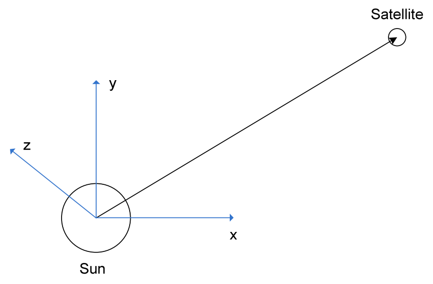
A powerful combination of techniques is to employ a 4th order Runge-Kutta to provide the additional required starting values y_1, y_2 and y_3 for a 4-step Adams-Bashforth predictor method and to employ this in conjunction with a 4 step Adams-Moulton corrector method.

$$y_n^{(p)} = y_{n-1} + h\left(\tfrac{55}{24} f\left(x_{n-1}, y_{n-1}\right) - \tfrac{59}{24} f\left(x_{n-2}, y_{n-2}\right) + \tfrac{37}{24} f\left(x_{n-3}, y_{n-3}\right) - \tfrac{9}{24} f\left(x_{n-4}, y_{n-4}\right)\right)$$

$$y_n = y_{n-1} + h\left(\tfrac{9}{24} f\left(x_n, y_n^{(p)}\right) + \tfrac{19}{24} f\left(x_{n-1}, y_{n-1}\right) - \tfrac{5}{24} f\left(x_{n-2}, y_{n-2}\right) + \tfrac{1}{24} f\left(x_{n-3}, y_{n-3}\right)\right)$$

Example: One Body Problem

A satellite (planet or comet) orbits the sun. The sun is considerably more massive than the satellite. We ignore fluctuations in the motion of the sun, expressing equations of motion in terms of a set of axes centred at the centre of mass of the sun.



$$M_{sat} \frac{d^2}{dt^2} \vec{r} = -\left(\frac{G M_{sat} M_{sun}}{r^3} \right) \vec{r}$$

Newton's Universal Law of Gravitation.

Example: One Body Problem

$$\frac{d^2}{dt^2}\begin{bmatrix} x \\ y \\ z \end{bmatrix} = -\left(\frac{GM_{sun}}{\left(x^2+y^2+z^2\right)^{\frac{3}{2}}}\right)\begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

By orienting the axes such that the motion of the satellite relative to the sun is initially in the x-y plane we obtain $z = 0$ and dz/dt = 0 at $t = 0$. It follows that z remains zero for all t, i.e. that motion, initially restricted to the x-y plane, remains thereafter in this plane.

$$\frac{d^2}{dt^2}\begin{bmatrix} x \\ y \end{bmatrix} = -\left(\frac{GM_{sun}}{\left(x^2+y^2\right)^{\frac{3}{2}}}\right)\begin{bmatrix} x \\ y \end{bmatrix}$$

Example: One Body Problem

A problem arises when we consider the values for G and M_{sun}.

$$G = 6.67384\times10^{-11}\ \text{m}^3\text{kg}^{-1}\text{s}^{-2}$$

$$M_{sun} = 1.98855\times10^{30}\ \text{kg} \qquad \text{Solar Mass}$$

The constant GM_{sun} is seen to be very large. Expressed in metres x and y will also be very large, whereas with t expressed in seconds we will need to take a very large number of time steps in order to simulate over a reasonable timeframe. The resolution of all of these problems is to choose more suitable time and distance scales.

Example: One Body Problem

An appropriate distance scale is the *Astronomical Unit* (AU). 1 AU is (approximately) the mean distance of the earth from the sun = 149,597,870,700 m.
An appropriate time scale is the *Day* = 86,400 s. However, it is my intention to use the year = 31,557,600 s, based on the definition of 1 year as 365.25 days. Of course this is not exactly equal to the time it takes for the earth to complete one full revolution about the sun, in large part because the effects of other planets cause this time to change from year to year.

Example: One Body Problem

$$\tau = \frac{t}{31{,}557{,}600} = \frac{t}{TU} \quad , \quad X = \frac{x}{AU} \quad , \quad Y = \frac{y}{AU}$$

$$\frac{d^2}{d\tau^2}\begin{bmatrix} X \\ Y \end{bmatrix} = -\left(\frac{C}{\left(X^2+Y^2\right)^{\frac{3}{2}}}\right)\begin{bmatrix} X \\ Y \end{bmatrix}$$

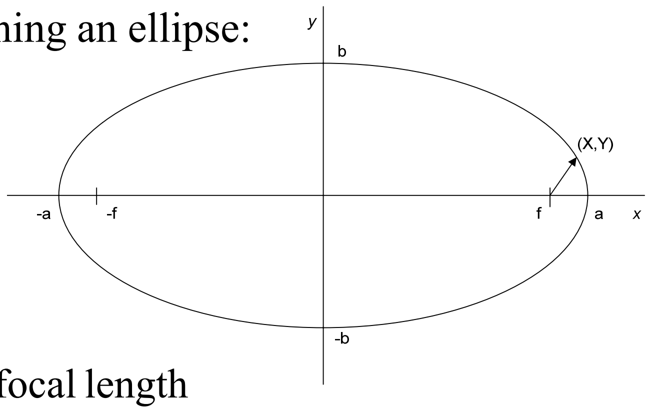
where

$$C = \frac{GM_{sun}TU^2}{AU^3} = 39.4526$$

a far more reasonable number.

Example: One Body Problem

Recall concerning an ellipse:



$f = \sqrt{a^2 - b^2}$ focal length

$$\left(\frac{x+f}{a}\right)^2 + \left(\frac{y}{b}\right)^2 = \left(\frac{X+F}{A}\right)^2 + \left(\frac{Y}{B}\right)^2 = 1$$

Example: One Body Problem

The specific details of the solution depend upon the initial conditions. We will assume that the satellite is initially at its *perihelion* (where it is closest to the sun) and we will align the x-y axes such that this corresponds to $X(0) = r_0$ and $Y(0) = 0$. Let us assume that the satellite commences in elliptical motion.

$$\left(\frac{X+F}{A}\right)^2 + \left(\frac{Y}{B}\right)^2 = 1 \qquad \left(\frac{r_0+F}{A}\right)^2 = 1 \qquad r_0 = A - F$$

$$\frac{2(X+F)X'}{A^2} + \frac{2YY'}{B^2} = 0 \qquad \frac{2AX'}{A^2} = 0 \qquad \frac{dX(0)}{d\tau} = 0$$

Example: One Body Problem

$$\frac{2(X+F)X''}{A^2}+\frac{2(X')^2}{A^2}+\frac{2YY''}{B^2}+\frac{2(Y')^2}{B^2}=0$$

$$-\frac{2C(X+F)X}{A^2r^3}+\frac{2(X')^2}{A^2}-\frac{2CY^2}{B^2r^3}+\frac{2(Y')^2}{B^2}=0$$

$$-\frac{2CAr_0}{A^2{r_0}^3}+\frac{2(Y')^2}{B^2}=0 \qquad \frac{dY(0)}{d\tau}=\frac{B}{\sqrt{A(A-F)}}\sqrt{\frac{C}{r_0}}$$

$$\frac{B}{A}=\frac{b}{a}=\sqrt{1-\varepsilon^2}$$

where ε is the *eccentricity* of the ellipse.

Example: One Body Problem

$$\frac{b}{a}=\sqrt{1-\varepsilon^2}$$

$$f=\sqrt{a^2-b^2}=a\varepsilon$$

$$\frac{dY(0)}{d\tau}=\frac{b}{\sqrt{a(a-f)}}\sqrt{\frac{C}{r_0}}=\frac{b/a}{\sqrt{1-\varepsilon}}\sqrt{\frac{C}{r_0}}=\sqrt{1+\varepsilon}\sqrt{\frac{C}{r_0}}$$

Example: One Body Problem

$$\begin{bmatrix} X'' \\ Y'' \end{bmatrix} = -\frac{C}{\left(X^2+Y^2\right)^{\frac{3}{2}}}\begin{bmatrix} X \\ Y \end{bmatrix}$$

$$X(0)=r_0 \,,\, Y(0)=0 \,,\, X'(0)=0 \,,\, Y'(0)=\sqrt{1-\varepsilon^2}\sqrt{\frac{C}{r_0}}$$

Wish to solve for general perihelion r_0 and eccentricity ε, using 4th order Runge-Kutta as starter for 4-step Adams-Bashforth predictor and 4-step Adams-Moulton corrector method.

Example: One Body Problem

Problem is that we do not have a system of first order differential equations. This is easily resolved:

$$v = \begin{bmatrix} X \\ Y \\ X' \\ Y' \end{bmatrix} \qquad v(0) = \begin{bmatrix} r_0 \\ 0 \\ 0 \\ \sqrt{1+\varepsilon}\sqrt{\frac{C}{r_0}} \end{bmatrix}$$

$$v' = \begin{bmatrix} X' \\ Y' \\ X'' \\ Y'' \end{bmatrix} = \begin{bmatrix} X' \\ Y' \\ -\frac{CX}{r^3} \\ -\frac{CY}{r^3} \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}\begin{bmatrix} X \\ Y \\ X' \\ Y' \end{bmatrix} - \frac{C}{\left(X^2+Y^2\right)^{\frac{3}{2}}}\begin{bmatrix} 0 \\ 0 \\ X \\ Y \end{bmatrix}$$

Starter

```
function  Storage = OneBodyProblemRK4Starter(r0,e,h)
% OneBodyProblemRK4Starter(r0,e): Use fourth order Runge-Kutta
% to start 4-step predictor-corrector for one body problem where
% satellite mass starts at perihelion, r0, of elliptical orbit of
% eccentricity e. Time step = h.
N = 4;    % Number of steps to be taken
G = 6.67*(10^(-11));  % set universal gravitational constant
Msun = 1.98855*(10^(30));  % set solar mass
AU = 149597870700;  % set 1 Astronomical unit (mean distance of
earth from sun)
TU = 31556600;  % set number of seconds in 1 year
C = G*Msun*(TU^2)/(AU^3);  % constant for equations in
normalised time and distance = 39.4526
```

Starter

```
Storage = zeros(4,N);   % assign store for N time steps of data
v = [r0;0;0;sqrt(1+e)*sqrt(C/r0)];  % initial conditions
Storage(:,1) = [v'];   % store initial time/position in Storage
A = [0 0 1 0;0 0 0 1;0 0 0 0;0 0 0 0];
for count = 2:N
   w = v;  % set temporary variable
   k1 = h*((A*w)-(C*[0;0;w(1);w(2)]/(((w(1)^2)+(w(2)^2))^(3/2))));
   w = v+((1/2)*k1);  % update w
   k2 = h*((A*w)-(C*[0;0;w(1);w(2)]/(((w(1)^2)+(w(2)^2))^(3/2))));
   w = v+((1/2)*k2);  % update w
   k3 = h*((A*w)-(C*[0;0;w(1);w(2)]/(((w(1)^2)+(w(2)^2))^(3/2))));
```

Starter

```
    w = v+k3;  % update w
    k4 = h*((A*w)-(C*[0;0;w(1);w(2)]/(((w(1)^2)+(w(2)^2))^(3/2))));
    v = v + ((1/6)*(k1+(2*k2)+(2*k3)+k4));   % update v
    Storage(:,count) = [v'];  % store new position
end
```

The perihelion of the orbit of Halley's comet is 0.586 AU. The eccentricity is 0.967. Let us use a step size of approximately 1 week.

```
>> Storage = OneBodyProblemRK4Starter(0.586,0.967,1/(52));
```

Starter

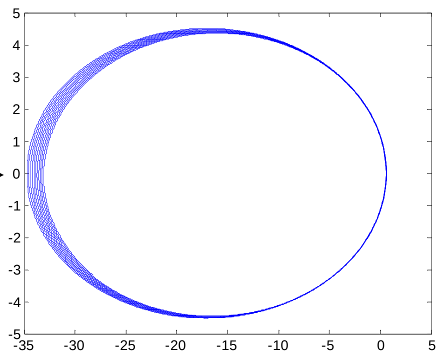
t_n	X_n	Y_n	X'_n	Y'_n
0	0.5860	0	0	11.5078
1/52	0.5652	0.2187	-2.1116	11.1132
2/52	0.5079	0.4237	-3.7483	10.1492
3/52	0.4249	0.6080	-4.7959	9.0081

Of course by editing this function M-file I can permit it to take a much larger number of steps. This is achieved by changing the assignment N = 4 to whatever you please. With N = 10000

```
>> Initial4 = OneBodyProblemRK4Starter(0.586,0.967,1/(52));
>> plot(Initial4(1,:),Initial4(2,:))
```

4th order Runge-Kutta

Note a problem here with accumulation of error.



Clearly the predicted orbit is not periodic. Instead, given the required initial data we employ a 4-step Adams-Bashforth predictor/4-step Adams-Moulton corrector.

Predictor/Corrector

```
function  Storage = OneBodyProblemABM4(Initial4,h)

% OneBodyProblemABM4(Initial4,h): Use 4-step Adams-Bashforth
% predictor/ 4-step Adams-Moulton corrector to solve one body
% problem where stellite mass starts at perihelion, r0, of elliptical
% orbit with initial data for first 4 time steps specified in Initial4 and
% with time step = h.

N = 4+40000;    % Number of steps to be taken

G = 6.67*(10^(-11));  % set universal gravitational constant

Msun = 1.98855*(10^(30));  % set solar mass

AU = 149597870700;  % set 1 Astronomical unit (mean distance of
earth from sun)

TU = 31556600;  % set number of seconds in 1 year

C = G*Msun*(TU^2)/(AU^3);
```

Predictor/Corrector

```
Storage = zeros(4,N);   % assign store for N time steps of data
Storage(:,[1:4]) = Initial4;   % store initial data
v1 = initial4(:,1);   % state vector at n-4
v2 = initial4(:,2);   % state vector at n-3
v3 = initial4(:,3);   % state vector at n-2
v4 = initial4(:,4);   % state vector at n-1
A = [0 0 1 0;0 0 0 1;0 0 0 0;0 0 0 0];
```

Predictor/Corrector

```
for count = 5:N
 % Predictor
   f1 = ((A*v1)-(C*[0;0;v1(1);v1(2)]/(((v1(1)^2)+(v1(2)^2))^(3/2))));
   f2 = ((A*v2)-(C*[0;0;v2(1);v2(2)]/(((v2(1)^2)+(v2(2)^2))^(3/2))));
   f3 = ((A*v3)-(C*[0;0;v3(1);v3(2)]/(((v3(1)^2)+(v3(2)^2))^(3/2))));
   f4 = ((A*v4)-(C*[0;0;v4(1);v4(2)]/(((v4(1)^2)+(v4(2)^2))^(3/2))));
   w = v4 + h*(((55/24)*f4)-((59/24)*f3)+((37/24)*f2)-((9/24)*f1));
% predicted value at n
% Corrector
   f5 = ((A*w)-(C*[0;0;w(1);w(2)]/(((w(1)^2)+(w(2)^2))^(3/2))));
   z = v4 + h*(((9/24)*f5)+((19/24)*f4)-((5/24)*f3)+((1/24)*f2));
```

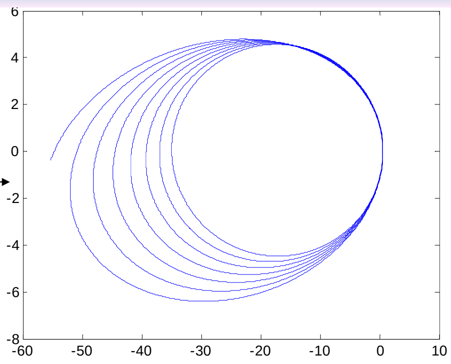
Predictor/Corrector

```
% update
  v1 = v2;
  v2 = v3;
  v3 = v4;
  v4 = z;
  Storage(:,count) = [v4];  % store new position
end
```

```
>> Storage = OneBodyProblemABM4(Initial4,1/(52));
>> plot(Storage(1,:),Storage(2,:))
```

Predictor/Corrector

Note major problem here



In this case the long term accumulation of error is even worse than when we employ the 4th order Runge-Kutta and indeed it appears that the solution is unstable. The problem, fundamentally, is the time frame being considered.

Section 4 - Conclusion

- The Runge-Kutta methods comprise a widely used and widely useful set of procedures for solving initial value problems for ODEs. In practice fourth order strikes a good balance between efficiency, stability and accuracy.
- Fourth order Runge-Kutta as a starter, 4-step Adams-Bashforth as predictor and 3- or 4-step Adams-Moulton as corrector amounts to a very good procedure in this field.

Overview of Topics

Solution of equations by iteration.

Optimisation.

Solution of system of linear equations.

Solution of ordinary differential equations: initial value problems.

Numerical Quadrature. ○

Solution of ordinary differential equations: boundary value problems.

Solution of partial differential equations.

Section 5: Numerical Quadrature

- Elementary method: Simpson's Rule
- Advanced methods: Gaussian Quadrature

Workload: 1 lectures + Autonomous learning

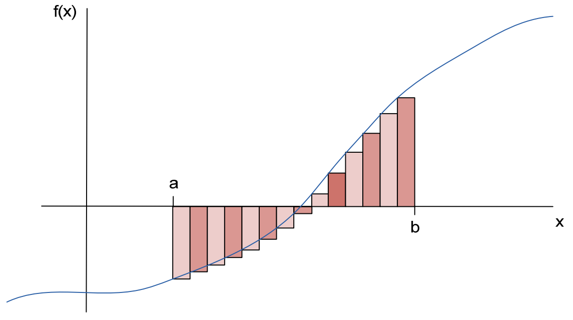
Quadrature

The classical problem of determining the square whose area is equal to that of a given two dimensional region is referred to as *quadrature*. In classical times it led to some very difficult problems such as "squaring the circle", i.e. constructing a square whose area equals that of a given disc. In fact if the construction of the square is to be by the favoured classical Greek method of elementary geometrical means then certain problems, such as squaring the circle, become impossible. Since the area of a square is known, the problem essentially amounts to that of finding the area of a given region. The problem of doing this numerically is therefore called either *numerical integration* or *numerical quadrature*.

Quadrature

The problem to be solved, in its simplest case, is to numerically evaluate the area under a curve, $f(x)$, between the values $x = a$ and b. The ancient idea is to break the interval $[a,b]$ up into sub-intervals of equal length $h = (b-a)/N$, where N is the number of sub-intervals.

Let x_n denote $a+nh$, the start of the nth sub-interval.

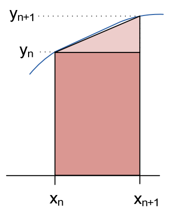


Quadrature: Rectangle Rule

Evidently one obtains a rather poor approximation in general if one approximates the area under the curve in the nth sub-interval by the area of the rectangle which coincides with the curve at the top left corner as shown. This approximation would be correct if the function was *constant* over this sub-interval.

Let y_n denote $f(a+nh)$, the y co-ordinate at the start of the nth sub-interval.

$$\text{Area} \cong hy_n$$



Quadrature: Rectangle Rule

Over the whole range of integration from *a* to *b* we have:

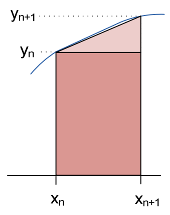
$$\begin{aligned}\text{Area} &\cong \\ &hy_0 + hy_1 + \cdots + hy_{N-1} \\ &= h(y_0 + y_1 + y_2 + \cdots + y_{N-1}) \\ &\cong h(y_1 + y_2 + y_3 + \cdots + y_N)\end{aligned}$$

where $y_0 = f(x_0) = f(a)$, $y_N = f(x_N) = f(b)$. This approximation is called the *rectangle rule* or *rectangle method*, a name which it takes from the shape approximating the region under the curve in each sub-interval.

Quadrature: Trapezoidal Rule

A better approximation is obtained by approximating the area by the sum of the areas of a rectangle and a triangle (i.e. the area of a *trapezoid*) as shown. This approximation would be correct if the function was constant or *linear* over this sub-interval.

$$\begin{aligned}\text{Area} &\cong hy_n + \tfrac{1}{2}h(y_{n+1} - y_n) \\ &= h(\tfrac{1}{2}y_n + \tfrac{1}{2}y_{n+1})\end{aligned}$$



Quadrature: Trapezoidal Rule

Over the whole range of integration from *a* to *b* we have:

$$\text{Area} \cong h\left(\tfrac{1}{2}y_0 + \tfrac{1}{2}y_1\right) + h\left(\tfrac{1}{2}y_1 + \tfrac{1}{2}y_2\right) + \cdots + h\left(\tfrac{1}{2}y_{N-1} + \tfrac{1}{2}y_N\right)$$
$$= \tfrac{h}{2}\left(y_0 + 2y_1 + 2y_2 + \cdots + 2y_{N-1} + y_N\right)$$

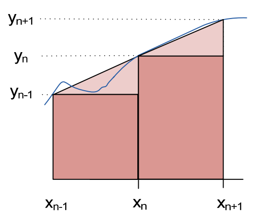
where $y_0 = f(x_0) = f(a)$, $y_N = f(x_N) = f(b)$. This approximation is called the *trapezoidal rule*, a name which it takes from the shape approximating the region under the curve in each sub-interval.

Quadrature: Simpson's Rule

A better approximation still is obtained by looking for an approximation to the area under the curve over *two* sub-intervals. The previous approximations have been correct for $f(x)$ constant or linear over sub-intervals. We will ask that the new approximation be correct for $f(x)$ being any quadratic function over the two sub-intervals.

Bearing in mind previous examples we suggest that:

$$\text{Area} \cong h\left(c_1 y_{n-1} + c_2 y_n + c_3 y_{n+1}\right)$$



Quadrature: Simpson's Rule

By linearity the formula is correct for any quadratic $f(x)$ if it is correct for 1, x and x^2. We have as a result the following conditions to assist in calculating appropriate values for coefficients c_i:

$$\int_{x_{n-1}}^{x_{n+1}} 1dx = h(c_1 + c_2 + c_3) = (x_{n-1} + 2h) - x_{n-1} = 2h$$

$$\int_{x_{n-1}}^{x_{n+1}} xdx = h(c_1 x_{n-1} + c_2(x_{n-1} + h) + c_3(x_{n-1} + 2h)) = \tfrac{1}{2}(x_{n-1} + 2h)^2 - \tfrac{1}{2}x_{n-1}^2$$

$$\int_{x_{n-1}}^{x_{n+1}} x^2 dx = h\left(c_1 x_{n-1}^2 + c_2(x_{n-1} + h)^2 + c_3(x_{n-1} + 2h)^2\right) = \tfrac{1}{3}(x_{n-1} + 2h)^3 - \tfrac{1}{3}x_{n-1}^3$$

Quadrature: Simpson's Rule

These may be simplified to obtain:

$$c_1 + c_2 + c_3 = 2$$
$$c_2 + 2c_3 = 2$$
$$c_2 + 4c_3 = \tfrac{8}{3}$$

which may be solved giving:

$$c_1 = \tfrac{1}{3}$$
$$c_2 = \tfrac{4}{3}$$
$$c_3 = \tfrac{1}{3}$$

$$\text{Area} \cong \tfrac{h}{3}\left(y_{n-1} + 4y_n + y_{n+1}\right)$$

Section: 5 Numerical Quadrature

Quadrature: Simpson's Rule

$$\text{Area} \cong \tfrac{h}{3}\left(y_{n-1} + 4y_n + y_{n+1}\right)$$

Assume that the range of integration from *a* to *b* is divided into an even number of sub-intervals, i.e. *N* is even. The approximation of the total area is obtained by using the previous approximation for the area over two adjacent sub-intervals.

Area ≅

$$\tfrac{h}{3}\left(y_0 + 4y_1 + y_2\right) + \tfrac{h}{3}\left(y_2 + 4y_3 + y_4\right) + \cdots + \tfrac{h}{3}\left(y_{N-2} + 4y_{N-1} + y_N\right)$$

$$= \tfrac{h}{3}\left(y_0 + 4y_1 + 2y_2 + 4y_3 + \cdots + 2y_{N-2} + 4y_{N-1} + y_N\right)$$

this approximation is called *Simpson's Rule*.

Section: 5 Numerical Quadrature

Example:

Numerically determine: $\int_1^{1.5} \sqrt{x}\,dx$

Select a step-size of $h = 0.05$, i.e. $N = 10$ and compare performances of three methods discussed.

x	$y = \sqrt{x}$	**x**	$y = \sqrt{x}$
1.00	1.0000	1.30	1.1402
1.05	1.0247	1.35	1.1619
1.10	1.0488	1.40	1.1832
1.15	1.0724	1.45	1.2042
1.20	1.0954	1.50	1.2247
1.25	1.1180		

Example:

Correct value: $\int_{1}^{1.5} \sqrt{x}dx = \frac{2}{3}x^{\frac{3}{2}}\Big|_{1}^{1.5} = \frac{2}{3}\left(1.5^{\frac{3}{2}}\right) - \frac{2}{3} = 0.5581$

10-point Rectangle Rule:

$$0.05(1.000 + 1.0247 + 1.0488 + 1.0724 + 1.0954 + 1.1180 + 1.1402 + 1.1619 + 1.1832 + 1.2042)$$

$$\cong 0.05(11.0488) = 0.5524$$

10-point Trapezoidal Rule:

$$\frac{0.05}{2}(1.000 + 2(1.0247 + 1.0488 + 1.0724 + 1.0954 + 1.1180 + 1.1402 + 1.1619 + 1.1832 + 1.2042) + 1.2247)$$

$$\cong \frac{0.05}{2}(22.3224) = 0.5581$$

Example:

Correct value: $\int_{1}^{1.5} \sqrt{x}dx = \frac{2}{3}x^{\frac{3}{2}}\Big|_{1}^{1.5} = \frac{2}{3}\left(1.5^{\frac{3}{2}}\right) - \frac{2}{3} = 0.5581$

10-point Simpson's Rule:

$$\frac{0.05}{3}(1.000 + 4(1.0247 + 1.0724 + 1.1180 + 1.1619 + 1.2042) + 2(1.0488 + 1.0954 + 1.1402 + 1.1832) + 1.2247)$$

$$\cong \frac{0.05}{3}(33.4847) = 0.5581$$

To 4 decimal places the Trapezoidal Rule and Simpson's Rule are correct when the range is broken into 10 sub-intervals.

Example:

To 14 digits:

Correct value: $\frac{2}{3}\left(1.5^{\frac{3}{2}}\right)-\frac{2}{3}=0.55807820472492$

10-point Trapezoidal Rule:

$$\cong \frac{0.05}{2}(22.32236367431802)=0.55805909185795$$

10-point Simpson's Rule:

$$\cong \frac{0.05}{3}(33.48469178744536)=0.55807819645742$$

Matlab: Trapezoidal Rule

```
>> N = 10;            select number of sub-intervals
>> h = (1.5-1)/N;          associated step size h = (b-a)/N
>> x = [1:h:1.5];     initialise vector x
>> y = sqrt(x);       compute associated values of vector y
```

Trapezoidal Rule:

```
>> Sum_trap = y(1) + (2*sum(y(2:N))) + y(N+1);  sum
equals first plus last plus twice sum of remaining
>> Int_trap = (h/2)*Sum_trap
```

Matlab: Simpson's Rule

```
>> N = 10;              select number of sub-intervals
>> h = (1.5-1)/N;       associated step size h = (b-a)/N
>> x = [1:h:1.5];       initialise vector x
>> y = sqrt(x);         compute associated values of vector y
```

Simpson's Rule:

```
>> Sum_simp = y(1) + (4*sum(y(2:2:N))) +
(2*sum(y(3:2:N-1)) )+ y(N+1); sum equals first plus last
plus four times sum of even plus twice sum of odd

>> Int_simp = (h/3)*Sum_simp
```

Example: Electrical Energy Usage

The electrical power demand is measured and recorded every fifteen minutes by Eirgrid (eirgrid.com). The demand for 18/09/2012 was:

time	***P* (MW)**	***time***	***P* (MW)**	***time***	***P* (MW)**
00:00	2279	02:30	1930	05:00	1943
00:15	2248	02:45	1916	05:15	1954
00:30	2198	03:00	1897	05:30	1999
00:45	2157	03:15	1881	05:45	2020
01:00	2113	03:30	1899	06:00	2077
01:15	2073	03:45	1870	06:15	2194
01:30	2043	04:00	1884	06:30	2273
01:45	2023	04:15	1935	06:45	2379
02:00	1986	04:30	1890	07:00	2462
02:15	1976	04:45	1911	07:15	2634

Example: Electrical Energy Usage

time	*P* (MW)	*time*	*P* (MW)	*time*	*P* (MW)
07:30	2779	10:45	3286	14:00	3261
07:45	2917	11:00	3294	14:15	3274
08:00	3040	11:15	3334	14:30	3288
08:15	3118	11:30	3297	14:45	3250
08:30	3142	11:45	3285	15:00	3287
08:45	3154	12:00	3318	15:15	3312
09:00	3180	12:15	3350	15:30	3281
09:15	3242	12:30	3334	15:45	3293
09:30	3284	12:45	3372	16:00	3318
09:45	3273	13:00	3368	16:15	3368
10:00	3261	13:15	3316	16:30	3393
10:15	3268	13:30	3287	16:45	3410
10:30	3274	13:45	3258	17:00	3418

Example: Electrical Energy Usage

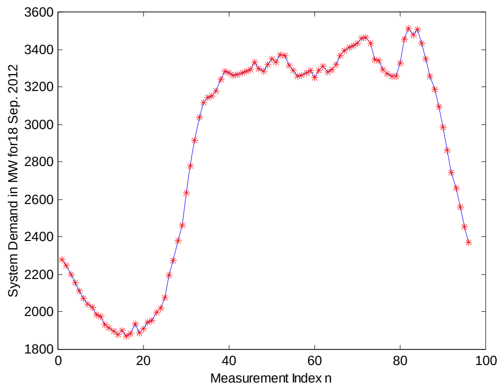
time	*P* (MW)	*time*	*P* (MW)	*time*	*P* (MW)
17:15	3431	19:30	3257	21:45	3187
17:30	3458	19:45	3328	22:00	3095
17:45	3464	20:00	3457	22:15	2985
18:00	3432	20:15	3512	22:30	2862
18:15	3346	20:30	3478	22:45	2743
18:30	3340	20:45	3508	23:00	2661
18:45	3294	21:00	3432	23:15	2561
19:00	3269	21:15	3348	23:30	2456
19:15	3256	21:30	3256	23:45	2372

The measurements at 15 minute time-intervals may be indexed with index n running from 1 (time 00:00) to 96 (time 23:45).

Example: Electrical Energy Usage

A plot of the system demand vs measurement index is as follows:

We require an estimate of the total electrical energy demand for the 18th of September, 2012.



Example: Electrical Energy Usage

Using all of the data to employ a 96-point Rectangle Rule with step-size $h = 0.25$ hr, gives an estimate:

68,754 MWhr

Using hourly data to employ a 24-point Rectangle Rule with step-size $h = 1$ hr, gives an estimate:

68,732 MWhr

Using hourly data to employ a 24-point Trapezoidal Rule with step-size $h = 1$ hr, gives an estimate:

66,262 MWhr

Example: Electrical Energy Usage

Using all of the data to employ a 96-point Trapezoidal Rule with step-size $h = 0.25$ hr, gives an estimate:

68,173 MWhr

Using hourly data to employ a 24-point Simpson's Rule with step-size $h = 1$ hr, gives an estimate:

65,283 MWhr

Using all data to employ a 96-point Simpson's Rule with step-size $h = 0.25$ hr, gives an estimate:

67,984 MWhr

Example: Electrical Energy Usage

The total electrical energy demand emerges as approximately 68,000 MWhr. No data is available for determining an exact value. The variability in the demand results in the various numerical quadrature rules giving rather poor estimates when the larger hourly step-size is employed. Simpson's Rule appears to be less effective in this case because it is based upon an approximation over two time intervals (i.e. a half hour) and the demand curve is simply not well approximated by a quadratic polynomial over this kind of time-frame.

Quadrature: Newton-Cotes Formulae

The Rectangle Rule, the Trapezoidal Rule and Simpson's Rule *essentially* form the first three methods in a class of numerical integration procedures called the *Newton-Cotes quadrature formulae*. Fundamentally these formulae are based upon approximating the function by polynomials of degree 0 (Rectangle Rule), 1 (Trapezoidal Rule), 2 (Simpson's Rule) or higher. In this list the next rule (based on degree 3 polynomial approximation) is called Simpson's 3/8 Rule and the next again after that (based on degree 4 polynomial approximation) is called Boole's Rule. Boole's rule is sometimes referred to as Bode's rule because of a misprint in a famous book on numerical analysis and its subsequent unquestioning reproduction.

Quadrature: Newton-Cotes Formulae

In principle it seems that approximations based upon higher order polynomials should be better. However, there exists a general phenomenon in the field of approximation by polynomials called *Runge's Phenomenon*. When one approximates a function *over a finite range* by a polynomial it can transpire that the error close to the boundary of that range is *larger* for higher order polynomials. The result of this phenomenon is that the hope that higher order Newton-Cotes formulae should give better estimates of integrals can be in vain.

Gaussian Quadrature

The Newton-Cotes formulae all share the property that the resulting approximation to the integral takes the form:

$$\int_a^b f(x)dx \cong \left(w_0 y_0 + w_1 y_1 + w_2 y_2 + \cdots + w_N y_N\right)$$

where the coefficients w_i comprise appropriate weights and where samples $y_i = f(x_i)$ comprise the function to be integrated evaluated at the special values x_i which are *equally* or *uniformly spaced*, i.e.

$$x_{i+1} - x_i = h$$

Gaussian Quadrature

If, as the Runge Phenomenon suggests, we find that errors when approximating by higher order polynomials can be large at the edges then it would appear that to mitigate this phenomenon we should take more samples at the edges, i.e. we should take non-uniformly spaced samples. Although not explicitly introduced for this purpose, Gaussian quadrature looks to approximate an integral by formulae of the form

$$\int_a^b f(x)dx \cong \left(w_0 f(x_0) + w_1 f(x_1) + w_2 f(x_2) + \cdots + w_N f(x_N)\right)$$

where on this occasion both weights w_i *and* points x_i must be chosen.

Gaussian Quadrature

The theory of Gaussian quadrature and its more recent variations (e.g. Gauss-Konrad quadrature, Clenshaw-Curtis quadrature, *etc*.) is considerably more difficult than that of the Newton-Cotes formulae. To see the value of the method I quote the *3-point Gaussian quadrature rule*:

$$w_0 = \tfrac{5}{9} \quad , \quad w_1 = \tfrac{8}{9} \quad , \quad w_2 = \tfrac{5}{9}$$

$$x_0 = -\sqrt{\tfrac{3}{5}} \quad , \quad x_1 = 0 \quad , \quad x_2 = \sqrt{\tfrac{3}{5}}$$

$$\int_{-1}^{1} f(x)dx \cong \left(\tfrac{5}{9} f\left(-\sqrt{\tfrac{3}{5}}\right) + \tfrac{8}{9} f(0) + \tfrac{5}{9} f\left(\sqrt{\tfrac{3}{5}}\right)\right)$$

Gaussian Quadrature

Note carefully the very special assumed range of the integration. I note also that the notation of the previous slide is not conventional. The indices are 0, 1 and 2 in that slide, where convention would have them as 1, 2 and 3. Of course this makes no difference at all to the formula:

$$\int_{-1}^{1} f(x)dx \cong \tfrac{5}{9} f\left(-\sqrt{\tfrac{3}{5}}\right) + \tfrac{8}{9} f(0) + \tfrac{5}{9} f\left(\sqrt{\tfrac{3}{5}}\right)$$

which is the 3-point Gaussian quadrature formula.

Example

I wish to consider the problem considered above, namely to numerically determine $\int_1^{1.5}\sqrt{x}dx$

First one must transform it to a problem to which the 3-point Gaussian quadrature formula can be applied. Let $z = 4x\text{-}5$. When $x = 1$, $z = -1$. When $x = 1.5$, $z = 1$.

$$\int_1^{1.5}\sqrt{x}dx = \tfrac{1}{4}\int_{-1}^{1}\sqrt{\frac{z+5}{4}}dz \cong \tfrac{1}{4}\left(\tfrac{5}{9}\sqrt{\frac{\left(-\sqrt{\tfrac{3}{5}}\right)+5}{4}}+\tfrac{8}{9}\sqrt{\frac{5}{4}}+\tfrac{5}{9}\sqrt{\frac{\left(\sqrt{\tfrac{3}{5}}\right)+5}{4}}\right)$$

Example:

To 14 digits:

Correct value: 0.55807820472492

(10-point) Trapezoidal Rule: 0.55805909185795

(10-point) Simpson's Rule: 0.55807819645742

(3-point) Gaussian Quadrature Rule:

0.55807822225438

3-point Gaussian Quadrature Rule is better than 10-point Trapezoidal and Simpson Rules.

2-point Gaussian Quadrature

To give at least some indication of how these quadrature formulae are derived consider the 2-point Gaussian quadrature formula in more conventional notation:

$$\int_{-1}^{1} f(x)dx \cong w_1 f(x_1) + w_2 f(x_2)$$

As in the Newton-Cotes formulae the idea is to require that this formula should hold exactly for every polynomial $f(x)$ of degree 3 or less. Again linearity establishes that this will hold provided the formula holds exactly for $f(x) = 1, x, x^2$ and x^3.

2-point Gaussian Quadrature

We obtain four equations:

$$\int_{-1}^{1} 1dx = 2 = w_1 + w_2 \qquad \int_{-1}^{1} xdx = 0 = w_1 x_1 + w_2 x_2$$

$$\int_{-1}^{1} x^2 dx = \tfrac{2}{3} = w_1 x_1^2 + w_2 x_2^2 \qquad \int_{-1}^{1} x^3 dx = 0 = w_1 x_1^3 + w_2 x_2^3$$

With four equations in four unknowns it should be possible to solve. At its heart Gaussian Quadrature is based upon a very clever method for solving equations of this kind.

2-point Gaussian Quadrature

Note that:

$$\int_{-1}^{1}\left(\tfrac{3}{2}x^2-\tfrac{1}{2}\right)dx = 0 = \tfrac{3}{2}\left(w_1x_1^2+w_2x_2^2\right)-\tfrac{1}{2}\left(w_1+w_2\right)$$

$$= w_1\left(\tfrac{3}{2}x_1^2-\tfrac{1}{2}\right)+w_2\left(\tfrac{3}{2}x_2^2-\tfrac{1}{2}\right)$$

$$\int_{-1}^{1}\left(\tfrac{3}{2}x^2-\tfrac{1}{2}\right)x\,dx = 0 = \tfrac{3}{2}\left(w_1x_1^3+w_2x_2^3\right)-\tfrac{1}{2}\left(w_1x_1+w_2x_2\right)$$

$$= w_1x_1\left(\tfrac{3}{2}x_1^2-\tfrac{1}{2}\right)+w_2x_2\left(\tfrac{3}{2}x_2^2-\tfrac{1}{2}\right)$$

2-point Gaussian Quadrature

Hence: $$\begin{bmatrix} w_1 & w_2 \\ w_1x_1 & w_2x_2 \end{bmatrix}\begin{bmatrix}\left(\tfrac{3}{2}x_1^2-\tfrac{1}{2}\right) \\ \left(\tfrac{3}{2}x_2^2-\tfrac{1}{2}\right)\end{bmatrix}=\begin{bmatrix}0\\0\end{bmatrix}$$

The matrix has determinant: $w_1w_2\left(x_2-x_1\right)$

Neither of the weights should be zero, else we are looking at a 1-point formula (they cannot both be zero since their sum is 2). The points x_1 and x_2 should differ, else again we are looking at a 1-point formula. It follows that the determinant of the matrix is non-zero. The matrix is non-singular so the only solution to the above equation is: $\tfrac{3}{2}x_1^2-\tfrac{1}{2}=0,\quad \tfrac{3}{2}x_2^2-\tfrac{1}{2}=0$

2-point Gaussian Quadrature

The solution method has seperated the weights and the points determining that the latter must be the roots of the polynomial: $\frac{3}{2}x^2 - \frac{1}{2}$

Accordingly: $x_1 = -\frac{1}{\sqrt{3}} \quad , \quad x_2 = \frac{1}{\sqrt{3}}$

Returning to the original four equations we find that two of them now automatically hold and the remaining two become:

$$w_1 + w_2 = 2$$
$$-w_1 + w_2 = 0$$

giving: $w_1 = 1 \quad , \quad w_2 = 1$

Gaussian Quadrature

Apparantly the key to the success of this method is down to the rather remarkable properties of the polynomial $\frac{3}{2}x^2 - \frac{1}{2}$

For the more general *m*-point Gaussian Quadrature a different, larger set of equations emerges and the remarkable properties of other, higher degree polynomials can be similarly employed to affect a solution. The polynomials involved fall into a set of polynomials having the name of the *Legendre polynomials of the first kind*. The theory of these polynomials explains how they come to have the properties which were so useful above.

Gaussian Quadrature: Caveat

By employing non-uniform sampling, i.e. points which are not uniformly spaced Gaussian quadrature achieves a superior performance for considerably less numerical effort. In engineering it is very rare to have non-uniformly spaced samples of a signal, it is far more common to have annual data or hourly data or second-by-second data, *etc.* Gaussian Quadrature is incredibly unlikely to be applicable given this kind of data.

Section 5 - Conclusion

- Simpson’s Rule comprises a reasonably accurate and efficient numerical quadrature rule applicable when uniformly-spaced samples only are available.
- If an equation is available, Gaussian quadrature comprises a far more efficient procedure.

Overview of Topics

Solution of equations by iteration.

Optimisation.

Solution of system of linear equations.

Solution of ordinary differential equations: initial value problems.

Numerical Quadrature.

Solution of ordinary differential equations: boundary value problems. ○

Solution of partial differential equations.

Section 6: Ordinary Differential Equations, Boundary Value Problems

- Elementary method: Finite Differences

Workload: 1 lecture + Autonomous learning

Section: 6 ODE, Boundary Value

Example: Catenary

The catenary is the curve which an idealised hanging chain assumes under its own weight when supported at both ends. Galileo observed that this curve is approximately a parabola and that this approximation improves as the curvature becomes less. The equation for the catenary was derived, essentially simultaneously, by Leibniz, Huygens and Johann Bernoulli in response to a mathematical challenge by Jacob Bernoulli.

$$y'' = \frac{\rho g A}{T_0}\sqrt{1+(y')^2}$$

ρA	mass per unit length
T_0	tension

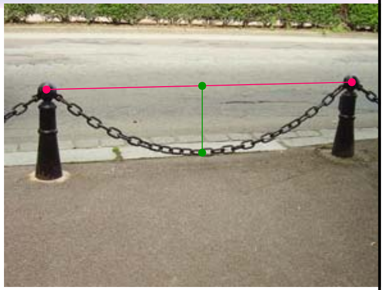
Section: 6 ODE, Boundary Value

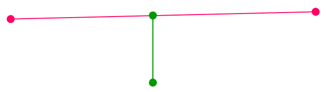
Example: Catenary

$$y'' = \frac{\rho g A}{T_0}\sqrt{1+(y')^2}$$

If the two ends are at equal height, let us measure from this height. Assume the ends are a distance l apart. Let $x = 0$ correspond to one end. We obtain as a result boundary conditions:

$$y(0) = 0\ ,\ y(l) = 0$$





Max droop about 22% of distance between fixed points.

Boundary-Value Problems

The catenary equation is an example of a boundary-value problem, where instead of having initial conditions only for an ODE we have conditions at *both* end-points. The methods which we have considered for solving ODEs with initial conditions do not apply.

Method of Finite differences

Consider the general second order boundary value problem:

$$y'' = f(x, y, y') \quad , \quad y(a) = y_a \quad , \quad y(b) = y_b$$

The range of values of x to be considered is finite being $[a,b]$. Break up this range into N equal sub-intervals so that each sub-interval has length $h = (b-a)/N$. Let y_i denote $y(a+ih)$, the value of $y(x)$ at the end-point $x_i = a+ih$ of the i^{th} sub-interval. The boundary conditions yield:

$$y_0 = y(a) = y_a \quad , \quad y_N = y(b) = y_b$$

Method of Finite differences

The key idea in the method of finite differences is to consider Taylor approximations to $y(x+h)$ and $y(x-h)$:

$$y(x+h) = y(x) + hy'(x) + \tfrac{1}{2}h^2 y''(x) + \tfrac{1}{6}h^3 y'''(x) + O(h^4)$$
$$y(x-h) = y(x) - hy'(x) + \tfrac{1}{2}h^2 y''(x) - \tfrac{1}{6}h^3 y'''(x) + O(h^4)$$

$$y(x+h) + y(x-h) = 2y(x) + h^2 y''(x) + O(h^4)$$

$$y(x+h) - y(x-h) = 2hy'(x) + O(h^3)$$

Method of Finite differences

Now evaluate at the special values $x = x_i$

$$y(x_{i+1}) + y(x_{i-1}) = 2y(x_i) + h^2 y''(x_i) + O(h^4)$$
$$y''(x_i) = \frac{y(x_{i+1}) - 2y(x_i) + y(x_{i-1})}{h^2} + O(h^2)$$

$$y(x_{i+1}) - y(x_{i-1}) = 2hy'(x_i) + O(h^3)$$
$$y'(x_i) = \frac{y(x_{i+1}) - y(x_{i-1})}{2h} + O(h^2)$$

Finally introduce notation:

$$y_i = y(x_i) \quad , \quad y_i' = y'(x_i) \quad , \quad y_i'' = y''(x_i)$$

Method of Finite differences

We achieve:

$$y_i' = \frac{y_{i+1} - y_{i-1}}{2h} + O\left(h^2\right) \qquad y_i'' = \frac{y_{i+1} - 2y_i + y_{i-1}}{h^2} + O\left(h^2\right)$$

or alternatively the *central difference* approximations:

$$y_i' \cong \frac{y_{i+1} - y_{i-1}}{2h} \qquad y_i'' \cong \frac{y_{i+1} - 2y_i + y_{i-1}}{h^2}$$

These approximations are rather good, the error being $O(h^2)$ rather than $O(h)$.

Method of Finite differences

Using the central difference approximations:

$$y_i' = \frac{y_{i+1} - y_{i-1}}{2h} \qquad y_i'' = \frac{y_{i+1} - 2y_i + y_{i-1}}{h^2}$$

the equation may therefore be replaced by:

$$\frac{y_{i+1} - 2y_i + y_{i-1}}{h^2} = f\left(x_i, y_i, \frac{y_{i+1} - y_{i-1}}{2h}\right)$$

for every *i* lying between 1 and *N*-1. In general this is a system of equations which can be solved.

Example: Catenary

$$y'' = \frac{\rho g A}{T_0}\sqrt{1+(y')^2} = \tfrac{1}{\alpha}\sqrt{1+(y')^2} \qquad \alpha = \frac{T_0}{\rho g A}$$

$$\frac{y_{i+1}-2y_i+y_{i-1}}{h^2} = \tfrac{1}{\alpha}\sqrt{1+\left(\frac{y_{i+1}-y_{i-1}}{2h}\right)^2}$$

$$y_0 = 0 \quad , \quad y_N = 0$$

If we take the unreasonably small value $N = 2$ we obtain $y_0 = 0 \quad , \quad y_2 = 0$

$$\frac{0-2y_1+0}{h^2} = \tfrac{1}{\alpha}\sqrt{1+\left(\frac{0-0}{2h}\right)^2} \qquad y_1 = -\frac{h^2}{2\alpha}$$

Example: Catenary

Of course $h = l/N = l/2$, so that

$$y_0 = 0 \quad , \quad y_1 = -\frac{l^2}{8\alpha} \quad , \quad y_2 = 0$$

The actual solution is:

$$y(x) = \alpha\left(\cosh\left(\frac{(x-\frac{l}{2})}{\alpha}\right) - \cosh\left(\frac{l}{2\alpha}\right)\right)$$

Employing the Taylor series for cosh:

$$y\left(\tfrac{l}{2}\right) = \alpha\left(1-\cosh\left(\frac{l}{2\alpha}\right)\right) = \alpha\left(1-1-\frac{1}{2!}\left(\frac{l^2}{4\alpha^2}\right)-\frac{1}{4!}\left(\frac{l^4}{16\alpha^4}\right)-\cdots\right)$$

$$y\left(\tfrac{l}{2}\right) = -\frac{l^2}{8\alpha^2}-\frac{l^4}{384\alpha^3}-\cdots$$

Method of Finite differences

So even with such a small number of steps the method has successfully determined the dominant term in the Taylor series for $y(l/2)$. Let us try $N = 3$.

$$y_0 = 0 \quad , \quad y_3 = 0$$

$$\frac{y_2 - 2y_1 + 0}{h^2} = \tfrac{1}{\alpha}\sqrt{1 + \left(\frac{y_2 - 0}{2h}\right)^2}$$

$$\frac{0 - 2y_2 + y_1}{h^2} = \tfrac{1}{\alpha}\sqrt{1 + \left(\frac{0 - y_1}{2h}\right)^2}$$

$$\begin{bmatrix} -2y_1 + y_2 - \frac{h^2}{\alpha}\sqrt{1 + \left(\frac{y_2}{2h}\right)^2} \\ y_1 - \frac{h^2}{\alpha}\sqrt{1 + \left(\frac{y_1}{2h}\right)^2} \quad -2y_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

Method of Finite differences

$$\begin{bmatrix} -2y_1 + y_2 - \frac{h^2}{\alpha}\sqrt{1 + \left(\frac{y_2}{2h}\right)^2} \\ y_1 - \frac{h^2}{\alpha}\sqrt{1 + \left(\frac{y_1}{2h}\right)^2} \quad -2y_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

We obtain as a result a system of nonlinear equations and of course we can employ the Newton-Raphson method to solve this system of equations provided we actually know the values of parameters α and $h = l/3$.

Example: Catenary

$$y(x) = \alpha\left(\cosh\left(\frac{(x-\frac{l}{2})}{\alpha}\right) - \cosh\left(\frac{l}{2\alpha}\right)\right)$$

Max droop occurs at $x = l/2$ where

$$y(\tfrac{l}{2}) = \alpha\left(1 - \cosh\left(\frac{l}{2\alpha}\right)\right)$$

If this is -0.22*l* (i.e. about 22% of distance between fixed points as in case of chain fence above) then

$$0.22 \cong \frac{\alpha}{l}\left(\cosh\left(\frac{l}{2\alpha}\right) - 1\right)$$

A nonlinear equation in l/α whose solution can be determined by Newton-Raphson method.

Example: Catenary

$$0 \cong \left(\cosh\left(\frac{l}{2\alpha}\right) - 1\right) - 0.22\left(\frac{l}{\alpha}\right)$$

Let $z = l/\alpha$ so that $0 \cong \left(\cosh\left(\frac{z}{2}\right) - 1\right) - 0.22z$

Newton-Raphson recursion is:

$$z_{n+1} = z_n - \frac{\left(\left(\cosh\left(\frac{z_n}{2}\right) - 1\right) - 0.22z_n\right)}{\left(\frac{1}{2}\sinh\left(\frac{z_n}{2}\right) - 0.22\right)}$$

$z_0 = 2$

$z_1 = 1.719,585,235$

$z_2 = 1.664,266,309$

$z_3 = 1.662,112,081$

$z_4 = 1.662,108,846$

Method of Finite differences

$$\begin{bmatrix} -2y_1 + y_2 - \frac{h^2}{\alpha}\sqrt{1+\left(\frac{y_2}{2h}\right)^2} \\ y_1 - \frac{h^2}{\alpha}\sqrt{1+\left(\frac{y_1}{2h}\right)^2} \quad -2y_2 \end{bmatrix} = \begin{bmatrix}0\\0\end{bmatrix}$$

Accordingly in the system of nonlinear equations obtained for the $N = 3$ case, assume $l = 1$m and take $z = l/\alpha = 1.662$. The equations become:

$$\begin{bmatrix} -2y_1 + y_2 - 0.1847\sqrt{1+2.25y_2^2} \\ y_1 - 0.1847\sqrt{1+2.25y_1^2} \quad -2y_2 \end{bmatrix} = \begin{bmatrix}0\\0\end{bmatrix}$$

Method of Finite differences

$$\begin{bmatrix} y_1(n+1) \\ y_2(n+1) \end{bmatrix} = \begin{bmatrix} y_1(n) \\ y_2(n) \end{bmatrix} -$$

$$\begin{bmatrix} -2 & 1-\frac{(0.1847)(4.5y_2(n))}{2\sqrt{1+2.25y_2^2(n)}} \\ 1-\frac{(0.1847)(4.5y_1(n))}{2\sqrt{1+2.25y_1^2(n)}} & -2 \end{bmatrix}^{-1} \begin{bmatrix} -2y_1(n)+y_2(n)-0.1847\sqrt{1+2.25y_2^2(n)} \\ y_1(n)-0.1847\sqrt{1+2.25y_1^2(n)} \quad -2y_2(n) \end{bmatrix}$$

```
>> y = [1;1];   initialise y

>> f = [-2*y(1)+y(2)-(0.1847*sqrt(1+(2.25*(y(2)^2))));-
2*y(2)+y(1)-(0.1847*sqrt(1+(2.25*(y(1)^2))))]      update function

>> J = [-2 1-((0.1847*4.5*y(2))/(2*sqrt(1+(2.25*(y(2)^2)))));1-
((0.1847*4.5*y(1))/(2*sqrt(1+(2.25*(y(1)^2))))) -2] update
Jacobian matrix J

>> y = y-(inv(J)*f)     update y
```

Method of Finite differences

It would be possible to incorporate all of the updates into a single Matlab command, but such is the length and complexity of this command that I find it is very easy to make a mistake, particularly with regard to the need to match brackets.

$$(y_1(1), y_2(1)) = (1,1)$$

$$(y_1(2), y_2(2)) = (-1.3330,-1.3330)$$

$$(y_1(3), y_2(3)) = (-0.0833,-0.0833)$$

$$(y_1(4), y_2(4)) = (-0.1898,-0.1898)$$

$$(y_1(5), y_2(5)) = (-0.1922,-0.1922)$$

Method of Finite differences

So in the case $l/\alpha = 1.662$, $l = 1$m the 4-point finite difference method gives:

$$y\left(\tfrac{1}{3}\right) = y\left(\tfrac{2}{3}\right) = -0.1922$$

whereas the actual values are:

$$y\left(\tfrac{1}{3}\right) = \tfrac{1}{1.662}\left(\cosh\left(1.662\left(\tfrac{1}{3}-\tfrac{1}{2}\right)\right) - \cosh\left(\frac{1.662}{2}\right)\right) = -0.1968$$

$$y\left(\tfrac{2}{3}\right) = \tfrac{1}{1.662}\left(\cosh\left(1.662\left(\tfrac{2}{3}-\tfrac{1}{2}\right)\right) - \cosh\left(\frac{1.662}{2}\right)\right) = -0.1968$$

a rather good approximation given the small number of steps employed.

Method of Finite differences

A larger value of N is more difficult to code. Let us take $N = 20$ for a 21-step finite difference algorithm.

$$\left.\begin{array}{l}
-2y_1 + y_2 - \frac{h^2}{\alpha}\sqrt{1+\left(\frac{y_2}{2h}\right)^2} = 0 \\
y_1 - 2y_2 + y_3 - \frac{h^2}{\alpha}\sqrt{1+\left(\frac{y_3-y_1}{2h}\right)^2} = 0 \\
y_2 - 2y_3 + y_4 - \frac{h^2}{\alpha}\sqrt{1+\left(\frac{y_4-y_2}{2h}\right)^2} = 0 \\
\qquad\vdots \\
y_{N-3} - 2y_{N-2} + y_{N-1} - \frac{h^2}{\alpha}\sqrt{1+\left(\frac{y_{N-1}-y_{N-3}}{2h}\right)^2} = 0 \\
y_{N-2} - 2y_{N-1} - \frac{h^2}{\alpha}\sqrt{1+\left(\frac{-y_{N-2}}{2h}\right)^2} = 0
\end{array}\right\} \quad F(y) = Ay - f(y) = \vec{0}$$

Method of Finite differences

The matrix A is:

$$A = \begin{bmatrix}
-2 & 1 & 0 & 0 & \cdots & 0 & 0 \\
1 & -2 & 1 & 0 & \cdots & 0 & 0 \\
0 & 1 & -2 & 1 & \cdots & 0 & 0 \\
0 & 0 & 1 & -2 & \cdots & 0 & 0 \\
\vdots & \vdots & \vdots & \vdots & & \vdots & \vdots \\
0 & 0 & 0 & 0 & \cdots & -2 & 1 \\
0 & 0 & 0 & 0 & \cdots & 1 & -2
\end{bmatrix}$$

This is a 19x19 matrix with the non-zero elements occurring on the main diagonal and on the super- and sub-diagonals only. Such a matrix is called a *tridiagonal matrix*.

Method of Finite differences

The Jacobian of $F(y)$ is to be found. $DF(y) = A - Df(y)$

$$Df(y) = \frac{h^2}{\alpha}\begin{bmatrix}
0 & \frac{\left(\frac{y_2}{4h^2}\right)}{\sqrt{1+\left(\frac{y_2}{2h}\right)^2}} & 0 & 0 & \cdots & 0 & 0 \\
\frac{\left(\frac{y_1-y_3}{4h^2}\right)}{\sqrt{1+\left(\frac{y_3-y_1}{2h}\right)^2}} & 0 & \frac{\left(\frac{y_3-y_1}{4h^2}\right)}{\sqrt{1+\left(\frac{y_3-y_1}{2h}\right)^2}} & 0 & \cdots & 0 & 0 \\
0 & \frac{\left(\frac{y_2-y_4}{4h^2}\right)}{\sqrt{1+\left(\frac{y_4-y_2}{2h}\right)^2}} & 0 & \frac{\left(\frac{y_4-y_2}{4h^2}\right)}{\sqrt{1+\left(\frac{y_4-y_2}{2h}\right)^2}} & \cdots & 0 & 0 \\
0 & 0 & \frac{\left(\frac{y_3-y_5}{4h^2}\right)}{\sqrt{1+\left(\frac{y_5-y_3}{2h}\right)^2}} & 0 & \cdots & 0 & 0 \\
\vdots & \vdots & \vdots & \vdots & & \vdots & \vdots \\
0 & 0 & 0 & 0 & \cdots & 0 & \frac{\left(\frac{y_{N-1}-y_{N-3}}{4h^2}\right)}{\sqrt{1+\left(\frac{y_{N-1}-y_{N-3}}{2h}\right)^2}} \\
0 & 0 & 0 & 0 & \cdots & \frac{\left(\frac{y_{N-2}}{4h^2}\right)}{\sqrt{1+\left(\frac{y_{N-2}}{2h}\right)^2}} & 0
\end{bmatrix}$$

Method of Finite differences

Although the Jacobian $Df(y)$ is complicated there is hope, like A it is a tridiagonal matrix. In fact $Df(y)$ is even simpler, since the elements on the main diagonal are all zero. In matrix A over 84% of the elements are zero and in matrix $Df(y)$ over 90% of the elements are zero. Such matrices are called a *sparse matrices.* Matlab has commands which permit the efficient creation, storage and usage of matrices of the tridiagonal kind because such matrices are examples of sparse matrices and Matlab has commands which permit the efficient creation, storage and usage of sparse matrices.

Method of Finite differences

```
>> N = 20        set parameter N
>> y = -ones(N-1,1);   initialise y as column vector
>> e = ones(N-1,1);
>> A = spdiags([e -2*e e], [-1 0 1], N-1, N-1);
```

combined commands create a tridiagonal matrix with -2s on the main diagonal and 1s on the super- and sub-diagonals. The matrix is stored in an efficient manner whereby only the non-zero elements and their indices are recorded. In Matlab such a matrix is called a sparse matrix.

Method of Finite differences

```
>> f = zeros(N-1,1)     initialise vector f
>> q = 1.662/(N^2)      set parameter q = h²/α
>> h = 1/N       set parameter h = l/N the step-size
>> f(1)  = q*sqrt(1+((y(2)^2)/((2*h)^2)));
>> f(N-1)  = q*sqrt((1+((y(N-2)^2))/((2*h)^2)));
>> for n = 2:N-2
 f(n)  = q*sqrt(1+(((y(n+1)-y(n-1))^2)/((2*h)^2)));
end
>> F = A*y – f;
```

Method of Finite differences

```
>> Dfsub = zeros(N-2,1);  initialise sub-diagonal vector of Df
>> Dfsup = zeros(N-2,1);   initialise super-diagonal vector of Df
>> Dfsub(N-2)  = q*(y(N-2)/((2*h)^2))/sqrt(1+(y(N-2)^2)/((2*h)^2));
>> Dfsup(1)  = q*(y(2)/((2*h)^2))/sqrt(1+(y(2)^2)/((2*h)^2));
>> for n = 1:N-3
Dfsub(n)  = -q*((y(n+2)-y(n))/((2*h)^2))/sqrt(1+((y(n+2)-y(n))^2)/((2*h)^2));
end
>> for n = 1:N-3
Dfsup(n+1)  = q*((y(n+2)-y(n))/((2*h)^2))/sqrt(1+((y(n+2)-y(n))^2)/((2*h)^2));
end
>> J = spdiags([[Dfsub;0] [0;Dfsup]], [-1 1], N-1,N-1);
>> y = y-inv(A-J)*F;
```

Method of Finite differences

After three iterations the solution converges to

$y_1 = y_{19} = -0.0437$ $y_8 = y_{12} = -0.2115$

$y_2 = y_{18} = -0.0820$ $y_9 = y_{11} = -0.2178$

$y_3 = y_{17} = -0.1152$ $y_{10} = -0.2198$

$y_4 = y_{16} = -0.1436$

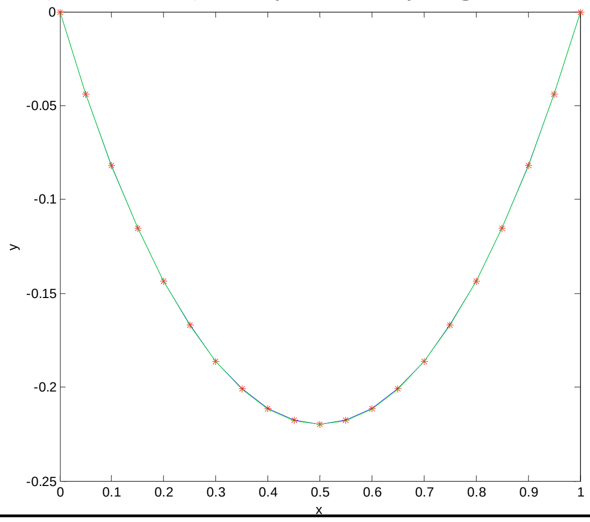
$y_5 = y_{15} = -0.1672$

$y_6 = y_{14} = -0.1863$

$y_7 = y_{13} = -0.2011$

Method of Finite differences

The actual solution (green solid) and the approximate solution (red stars) very closely agree.



Finite Element Method

There exists for boundary value problems (and more generally for PDEs) a second, more recent and more sophisticated method, called the *finite element method*. The method is based upon a considerably more advanced understanding of the differential equation itself. This understanding derives, first and foremost, from the works of Euler and Lagrange on the *calculus of variations*. Time does not permit consideration of this method.

Section 6 - Conclusion

- Boundary value problems for ODEs can be converted to systems of equations by means of the finite difference procedure. The resulting equations can be solved by the Newton-Raphson method where they generally give rise to sparse matrices.

- The finite difference method is not considered to be the best available.