



10. NON-LINEAR LEAST SQUARE FITTING, OPTIMIZATION

10.1.1 Remember: what we saw during the last lesson

Iterative solution for sparse systems (Jacobi, Gauss Seidel, Matlab `help sparse`)

Non-linear systems Fixed point, Piccard and Newton iterations

Non-linear interpolation

10.1.2 Overview: what you will learn today

Non-linear fitting

Optimization searching for an extremum with-/ out using derivatives

Ordinary differential equations using Euler and Runge-Kutta methods

10.2 Non-linear least square fitting (NAM 6.10, H 6.6)

Example Fit the model function $F(t) = a + b \sin \omega(t - t_0)$ to experimental data.

| | | | | | | | | |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| t | 0.5 | 0.8 | 1.0 | 1.2 | 1.5 | 1.8 | 2.0 | 2.4 |
| y | 0.3 | 0.3 | 0.5 | 0.9 | 1.4 | 1.1 | 0.5 | 0.3 |

This yields an **overdetermined non-linear system** (4 unknowns, 8 equations)

$$\left\{ \begin{array}{l} a + b \sin \omega(0.5 - t_0) \approx 0.3 \\ a + b \sin \omega(0.8 - t_0) \approx 0.3 \\ a + b \sin \omega(1.0 - t_0) \approx 0.5 \\ a + b \sin \omega(1.2 - t_0) \approx 0.9 \\ a + b \sin \omega(1.5 - t_0) \approx 1.4 \\ a + b \sin \omega(1.8 - t_0) \approx 1.1 \\ a + b \sin \omega(2.0 - t_0) \approx 0.5 \\ a + b \sin \omega(2.4 - t_0) \approx 0.3 \end{array} \right. \quad \text{or} \quad \left\{ \begin{array}{l} a + b \sin \omega(0.5 - t_0) - 0.3 \approx 0 \\ a + b \sin \omega(0.8 - t_0) - 0.3 \approx 0 \\ a + b \sin \omega(1.0 - t_0) - 0.5 \approx 0 \\ a + b \sin \omega(1.2 - t_0) - 0.9 \approx 0 \\ a + b \sin \omega(1.5 - t_0) - 1.4 \approx 0 \\ a + b \sin \omega(1.8 - t_0) - 1.1 \approx 0 \\ a + b \sin \omega(2.0 - t_0) - 0.5 \approx 0 \\ a + b \sin \omega(2.4 - t_0) - 0.3 \approx 0 \end{array} \right.$$

which can be solved by demanding that residuals between the measurements y_i and the model $F(t_i)$ be small $\|y - \mathbf{F}\|_2 \approx 0$, i.e. $\|y - \mathbf{F}\|_2 = \mathbf{f}(\mathbf{c}) \approx 0$.

Gauss-Newton solution obtained from $\mathbf{c}^{\text{new}} = \mathbf{c}^{\text{old}} + \delta \mathbf{c}, \quad \mathbf{J}^T \mathbf{J} \delta \mathbf{c} = -\mathbf{J}^T \mathbf{f}(\mathbf{c}^{\text{old}})$

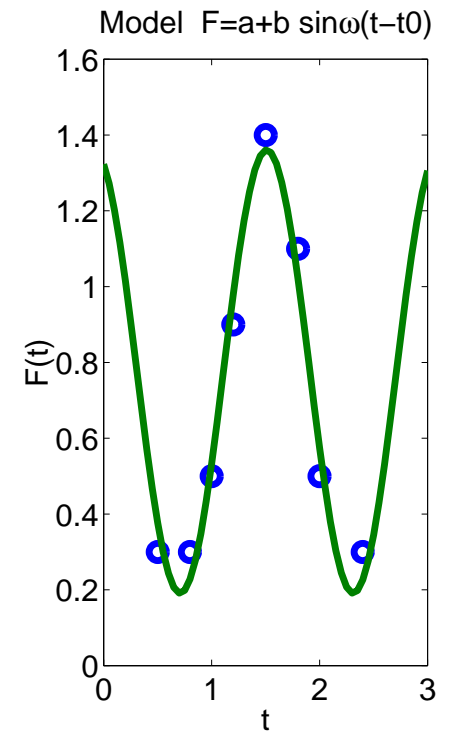
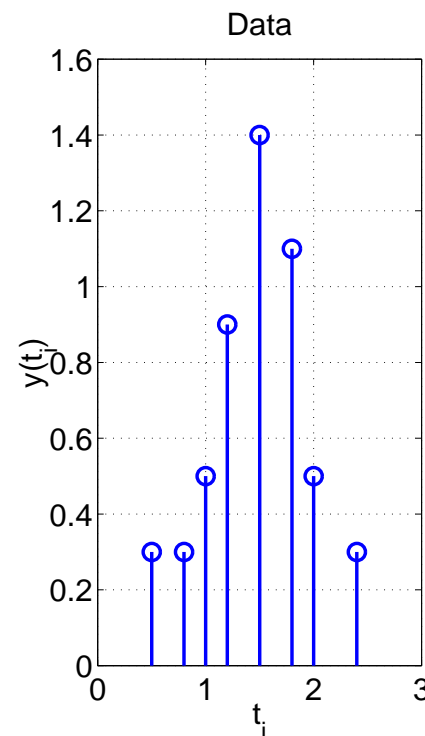
The coefficients \mathbf{c} are obtained from the Newton method for non-linear systems, with an increment $\delta \mathbf{c}$ solving the *overdetermined linear system* $\mathbf{J} \delta \mathbf{c} \approx -\mathbf{f}$. In Matlab, the solution of the normal equations can again be computed with $\delta \mathbf{c} = -\mathbf{f} \setminus \mathbf{J}$.

Example solution of the non-linear fit with Matlab

```

>> t=[0.5 0.8 1 1.2 1.5 1.8 2 2.4]';
>> y=[0.3 0.3 0.5 0.9 1.4 1.1 0.5 0.3]';
>> subplot(1,2,1), h=stem(t,y), title('Data')
>> a=0.7; b=0.7; w=pi; t0=1.2;
>> c=[a b w t0]'; %initial guess
>> n=size(t,1); iter=0; dcnorm=1.;
>> while dcnorm>1E-6 & iter<10
>>     u=w*(t-t0); f=a+b*sin(u)-y;
>>     Ji1=ones(n,1); Ji2=sin(u);
>>     Ji3=b*(t-t0).*cos(u); Ji4=-w*b*cos(u);
>>     J=[Ji1 Ji2 Ji3 Ji4]; % Jacobian
>>     dc=-J\f; c=c+dc; % Gauss-Newton
>>     dcnorm=norm(dc); iter=iter+1;
>>     a=c(1); b=c(2); w=c(3); t0=c(4);
>>     D=[iter ca b w t0 norm(f) norm(dc) ]
>> end
>> tt=(0:0.05:3)'; Ft=a+b*sin(w*(tt-t0));
>> subplot(1,2,2), plot(t,y,'o',tt,Ft)

```



A solution with an accuracy better than 10^{-6} is obtained with 6 steps:

| step | a | b | w | t0 | f | dc |
|------|--------|--------|--------|--------|--------|--------|
| 1 | 0.7246 | 0.4614 | 3.3935 | 1.1074 | 0.8034 | 0.3600 |
| 2 | 0.7772 | 0.5428 | 3.9476 | 1.1123 | 0.3688 | 0.5626 |
| 3 | 0.7762 | 0.5850 | 3.9219 | 1.1089 | 0.2117 | 0.0496 |
| 4 | 0.7761 | 0.5850 | 3.9225 | 1.1092 | 0.1928 | 0.0007 |
| 5 | 0.7761 | 0.5850 | 3.9225 | 1.1092 | 0.1928 | 0.0000 |
| 6 | 0.7761 | 0.5850 | 3.9225 | 1.1092 | 0.1928 | 0.0000 |

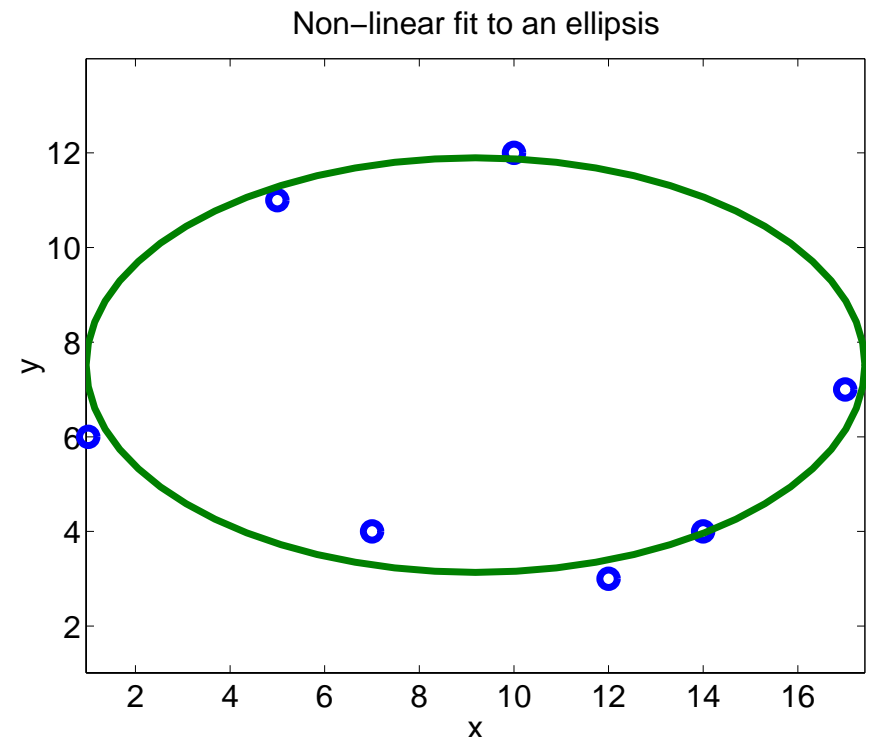
Example: fit an ellipsis $F(x, y) = \frac{(x-x_c)^2}{a^2} + \frac{(y-y_c)^2}{b^2} - 1 = 0$ to seven data points.

Set-up the overdetermined non-linear system with 7 equations for 4 unknowns (x_c, y_c, a, b) ; calculate the Jacobian $\partial F_i / \partial x_j$

$$\frac{\partial F}{\partial x_c} = -2 \frac{x - x_c}{a^2} \qquad \frac{\partial F}{\partial a} = -2 \frac{(x - x_c)^2}{a^3}$$

$$\frac{\partial F}{\partial y_c} = -2 \frac{y - y_c}{b^2} \qquad \frac{\partial F}{\partial b} = -2 \frac{(y - y_c)^2}{b^3}$$

```
>> x=[1 7 10 17 5 12 14]'; %data
>> y=[6 4 12 7 11 3 4]';
>> xc=10; yc=8; a=8; b=3; %initial guess
>> p=[xc yc a b]'; iter=0; dp=1;
>> while norm(dp)> 1e-6 & iter<10
>> iter=iter+1; xd=x-xc; yd=y-yc;
>> f=xd.^2/a^2+yd.^2/b^2-1;
>> J=-2*[xd/a^2 yd/b^2 xd.^2/a^3 yd.^2/b^3];
>> dp=-J\f; p=p+dp; %Gauss-Newton
>> end
>> xc=p(1), yc=p(2), a=p(3), b=p(4)
>> v=0:2*pi/60:2*pi;
>> plot(x,y,'o',xc+a*cos(v),yc+b*sin(v)),
>> axis equal,
xc = 9.1879 yc = 7.5159
a = 8.2298 b = 4.3817
```



Note that a fit to a circle can be written as a **linear** least square problem (lab 1.3).

10.3 Optimization in one dimension (NAM 7.1, H 6.4)

In Matlab use `fminbnd` for real arguments and `max` for integer arguments argument.

```
>> f=inline('1.3-exp(-(x-1).^2)-min(x/4,max(1./(x-1),0))');  
>> [xx,fx,flag]=fminbnd(f,0,3)  
>> x=0:0.001:3; plot(x,feval(f,x),xx,fx,'o')
```

```
xx = 1.1270    fxx = 0.0342    flag = 1
```

A variety of methods are combined to provide rapid convergence and robust answers; they require *unimodal functions* that are strictly increasing in all the direction from the minimum:

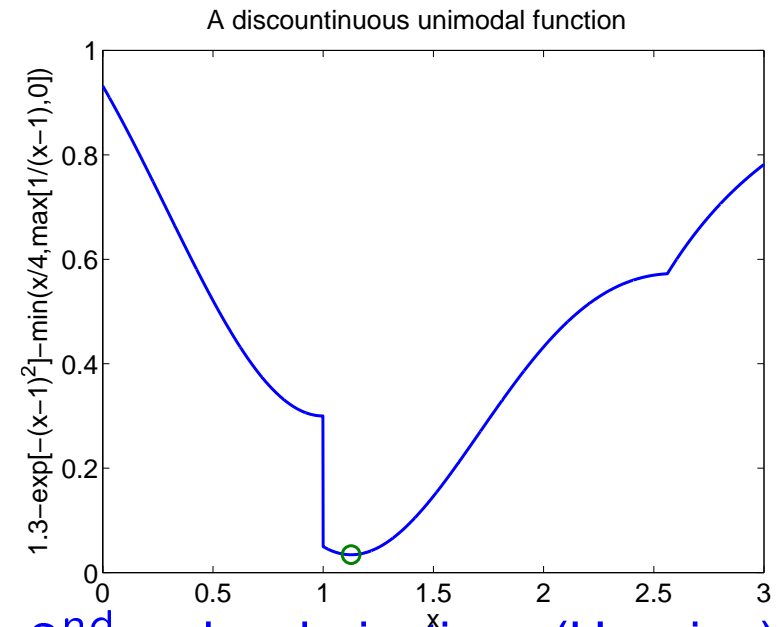
$$f(x) = 1.3 - e^{-(x-1)^2} - \min\left[\frac{x}{4}, \max\left[0, \frac{1}{x-1}\right]\right]$$

Methods that use 1st order (Jacobian) and even 2nd order derivatives (Hessian) converge faster... if defined! Check `help optimset` to set the properties.

Here is an example of an optimization where the argument is an integer and the function is cheap to evaluate:

```
>> n=1000000; f=randn(1,n); [fmin,indx]=min(f)
```

```
fmin = -4.6365    indx = 198438
```



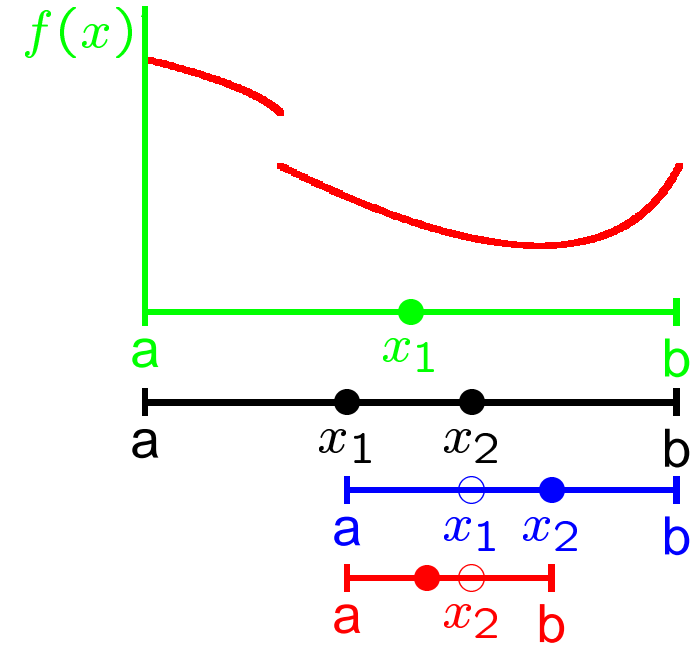
Golden section search of a minimum $x_* \in [a; b]$ without computing derivatives

Two evaluations are required to decide if the minimum is located in the left or right interval:

```
if  $f(x_1) > f(x_2)$  then  
   $x_* \in [x_1; b]$   
else  
   $x_* \in [a; x_2]$   
end
```

For efficiency, choose the proportions so that

$$\tau = \frac{b - x_1}{b - a} = \frac{b - x_2}{b - x_1}$$



in order to reuse the previous evaluations. By symmetry $(x_2 - a)/(b - a) = \tau$

$$\frac{b - x_2}{b - a} = 1 - \tau \quad \Rightarrow \quad \frac{b - x_2}{b - x_1} = \frac{1 - \tau}{\tau}$$

$$\tau = \frac{1 - \tau}{\tau} \quad \Rightarrow \quad \tau^2 + \tau - 1 = 0 \quad \Rightarrow \quad \tau = \frac{\sqrt{5} - 1}{2} \approx 0.6180$$

which reminds the *golden ratio* $1 + \sqrt{5}/2 \approx 1.6180$ from the antiquity. Finally

$$x_1 = a + (b - a)(1 - \tau)$$

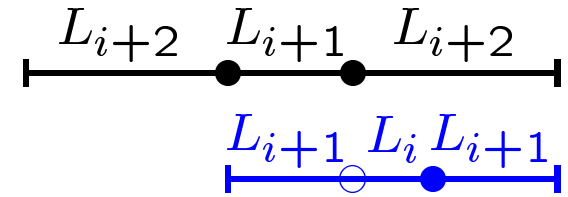
$$x_2 = a + (b - a)\tau$$

Fibonacci search can be used when the argument is an integer

Write a relation between steps in a golden search

$$L_{i+2} = L_i + L_{i+1}$$

This yields a recursion formula for the Fibonacci numbers



$$F \in \{1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, \dots\}$$

where consecutive integers approximate the golden section $34/55 = 0.6182 \approx \tau$. This suggests a more efficient algorithm than Matlab's `max` to find the maximum of $Ff(n)$ in the interval $i = 0 \dots 120$ starting from a larger Fibonacci number:

```
function f=Ff(n);
    f=exp(sqrt(n)/10) -cos(n/30+19); return
>> a=0; b=144; n2=89; n1=a+b-n2; F1=Ff(n1); F2=Ff(n2);
>> disp([a n1 n2 b])
>> while b-a>1
>>     if F1>F2, b=n2; n2=n1; F2=F1; n1=a+b-n2; F1=Ff(n1);
>>     else     a=n1; n1=n2; F1=F2; n2=a+b-n1; F2=Ff(n2);
>>     end
>>     disp([a n1 n2 b])
>> end; Fmax=[F1 F2]
```

Fmax = 3.6630 3.6629

| a | n1 | n2 | n |
|-----|-----|-----|-----|
| 0 | 55 | 89 | 144 |
| 55 | 89 | 110 | 144 |
| 89 | 110 | 123 | 144 |
| 89 | 102 | 110 | 123 |
| 89 | 97 | 102 | 110 |
| 97 | 102 | 105 | 110 |
| 97 | 100 | 102 | 105 |
| 100 | 102 | 103 | 105 |
| 100 | 101 | 102 | 103 |
| 101 | 102 | 102 | 103 |
| 102 | 102 | 103 | 103 |

10.4 Optimization in higher dimensions n (NAM 7.2, H 6.5)

Efficient methods such as *steepest descent*, *conjugate-gradients*) do not require evaluating the Jacobian: they are the subject of advanced courses.

Newton's method is however sufficient if the function can be differentiated analytically to calculate an extremum by solving

$$\nabla\Phi(x_1, x_2, \dots, x_n) = 0$$

using iterations

$$\mathbf{x}^{\text{new}} = \mathbf{x}^{\text{old}} + d\mathbf{x}, \quad \mathbf{J}d\mathbf{x} = -\mathbf{f}(\mathbf{x}^{\text{old}})$$

Example: maximum of $\Phi(x, y) = (x + \sin y)e^{-(x^2+y^2)}$ around $(x, y) = (\frac{1}{4}, \frac{1}{4})$.

$$\nabla\Phi = \begin{pmatrix} \frac{\partial\Phi}{\partial x} \\ \frac{\partial\Phi}{\partial y} \end{pmatrix} = \begin{pmatrix} (1 - 2x(x + \sin y))e^{-(x^2+y^2)} \\ (\cos y - 2x(x + \sin y))e^{-(x^2+y^2)} \end{pmatrix} = \begin{pmatrix} f_1(x, y) \\ f_2(x, y) \end{pmatrix} = 0$$

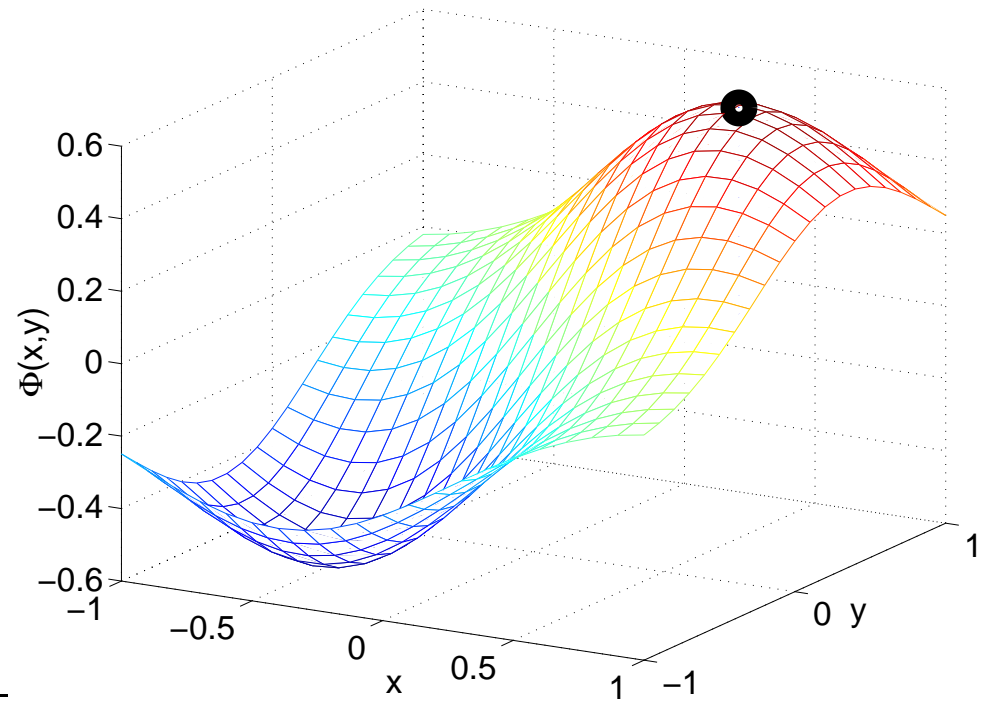
To solve this system of the form $\mathbf{f}(\mathbf{x}) = 0$, calculate the Jacobian $J_{ij} = \partial f_i / \partial x_j$

$$\mathbf{J} = \begin{pmatrix} 4x + 2 \sin y & 2x \cos y \\ 2y & 2x + 3 \sin y + 2y \cos y \end{pmatrix}$$

In matlab the solution is then obtained with

```
>> x=1/4; y=1/4;
>> F=(x+sin(y))*exp(-(x^2+y^2));
>> z=[x y]'; dznorm=1; iter=0;
>> disp([iter dznorm x y F])
>> while dznorm>1e-6 & iter < 30
>> s=x+sin(y);
>> f=[1-2*x*s; cos(y)-2*y*s];
>> J=[-4*x-2*sin(y) -2*x*cos(y);
>>     -2*y -2*x-3*sin(y)-2*y*cos(y)];
>> dz=-J\f; z=z+dz;
>> dznorm=norm(dz,inf); iter=iter+1;
>> x=z(1); y=z(2);
>> F=(x+sin(y))*exp(-(x^2+y^2));
>> disp([iter dznorm x y F])
>> end
```

| iter | dz | x | y | F |
|--------|--------|--------|--------|--------|
| 0 | 1.0000 | 0.2500 | 0.2500 | 0.4390 |
| 1.0000 | 0.4055 | 0.6555 | 0.5497 | 0.5666 |
| 2.0000 | 0.1240 | 0.5315 | 0.4695 | 0.5950 |
| 3.0000 | 0.0133 | 0.5182 | 0.4635 | 0.5953 |
| 4.0000 | 0.0002 | 0.5181 | 0.4634 | 0.5953 |
| 5.0000 | 0.0000 | 0.5181 | 0.4634 | 0.5953 |



11. ORDINARY DIFFERENTIAL EQUATIONS (ODEs)

11.2 Euler and Runge-Kutta methods (NAM 8.2, H 9.3)

Problem: numerically solve the ordinary differential equation for $y(t)$ for $t > t_0$

$$\frac{dy}{dt} = f(t, y), \quad \text{with } y(t_0) = y_0$$

using approximations with small steps in time t_0, t_1, \dots such that $t_{i+1} = t_i + h$.

Euler's method directly follows from the forward difference

$$\frac{y(t_i + h) - y(t_i)}{h} \approx f(t, y), \quad \Rightarrow \quad \boxed{y(t_{i+1}) \approx y_i + hf(t_i, y_i)}$$

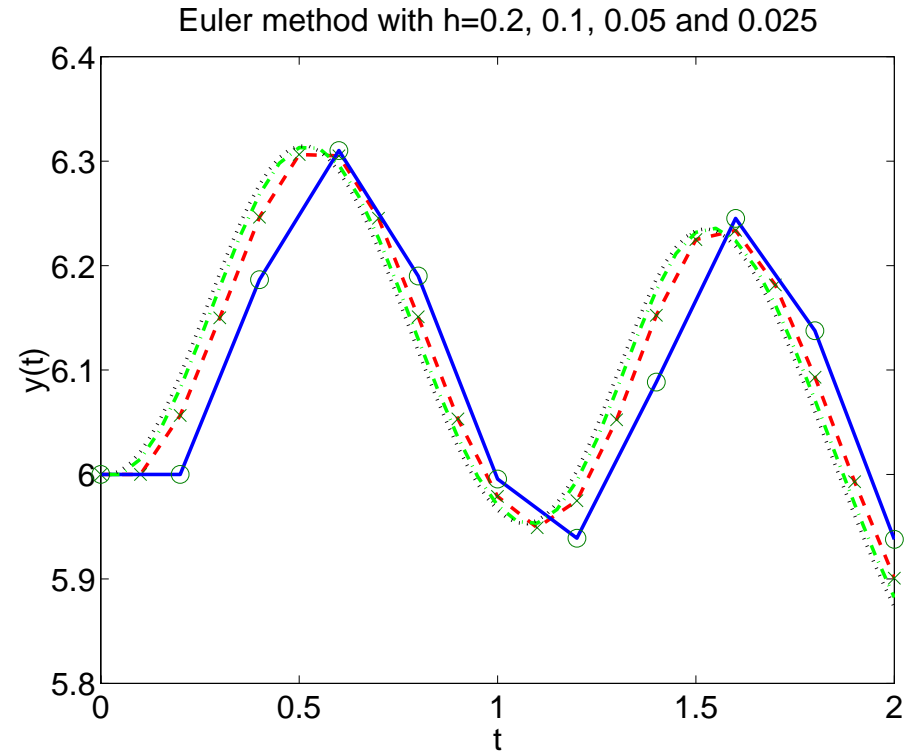
Starting from the *initial condition* (t_0, y_0) , one step with Euler's method produces an approximation (t_1, y_1) with a *local error* $\mathcal{O}(h^2)$; after n steps to reach the final time $t_n = t_0 + nh$, the solution (t_n, y_n) has a *global error* $n\mathcal{O}(h^2) \sim \mathcal{O}(h)$.

Example solve $y'(t) = \sin(ty)$ with $y(0) = 6$ using Euler's method

```
function fend=feuler(h)
y=6; t=0; tend=2; n=tend/h; T=t; Y=y;
for i=1:n
    f=sin(t*y); y=y+h*f; t=t+h;
    T=[T; t]; Y=[Y; y];
end
fend=y; plot(T,Y,T,Y,'o');
return

>> h=0.2; Y2=[];
>> for k=0:3, Y2=[Y2 feuler(h/2^k)], end;

Y2 = 5.9379  5.9007  5.8823  5.8731
```



A Richardson extrapolation can be used to cancel the leading (global) error in $\mathcal{O}(h)$ and calculate the value $y(2) \approx 5.8731 + \frac{5.8731 - 5.8823}{2^Q - 1} = 5.8639$.

Peer Teaching (2×1 minutes to think, explain to your neighbour and vote)

Richardson extrapolation. Which value for Q should you use here above?

← 0

↑ 1

→ 2

Runge-Kutta (RK2) achieves a better precision with a global error in $\mathcal{O}(h^2)$

$$y_{i+1} = y_i + \frac{h}{2}(k_1 + k_2), \quad \text{with} \quad \begin{cases} k_1 = f(t_i, y_i) \\ k_2 = f(t_i + h, y_i + hk_1) \end{cases}$$

Runge-Kutta (RK4) achieves a high precision with a global error in $\mathcal{O}(h^4)$

$$y_{i+1} = y_i + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4), \quad \text{with} \quad \begin{cases} k_1 = f(t_i, y_i) \\ k_2 = f(t_i + \frac{h}{2}, y_i + \frac{h}{2}k_1) \\ k_3 = f(t_i + \frac{h}{2}, y_i + \frac{h}{2}k_2) \\ k_4 = f(t_i + h, y_i + hk_3) \end{cases}$$

In Matlab use `help ode23`, `ode45`, `odeset` for RK methods with variable step size, where the step size h is continually adjusted to achieve a specified precision with a minimum number of steps.

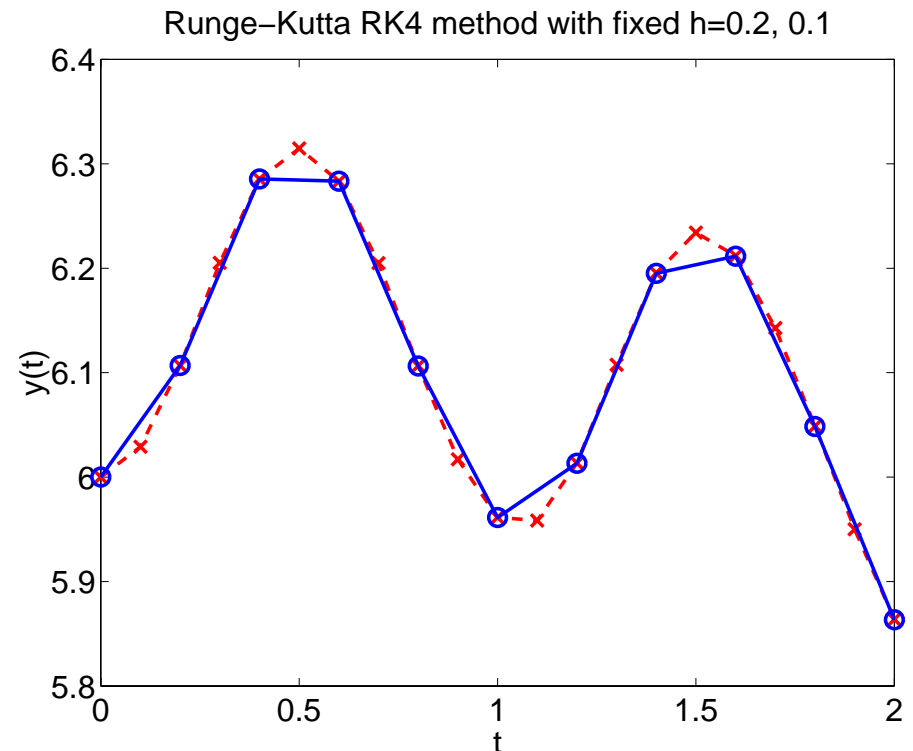
Example solve $y'(t) = \sin(ty)$ with $y(0) = 6$ using the RK4 method

```
function f=fsin(t,y),
    f=sin(t*y);
return

function fend=rk4(h)
    y=6; t=0; tend=2; n=tend/h;
    T=t; Y=y; h2=h/2;
    for i=1:n
        k1=fsin(t, y);
        k2=fsin(t+h2,y+h2*k1);
        k3=fsin(t+h2,y+h2*k2);
        k4=fsin(t+h ,y+h *k3);
        y=y+h/6*(k1+2*k2+2*k3+k4);
        t=t+h;
        T=[T; t]; Y=[Y; y];
    end
    plot(T,Y); fend=y;
return
```

```
>> h=0.2; YN=[];
>> for k=0:1, YN=[YN rk4(h/2^k)]; Y2=YN(end), end;
Y2 = 5.86346010701075    5.86390480621220
```

```
>> % --- Alternative using Matlab's solvers
>> tend=2; y0=6; options=odeset('RelTol',1e-6,'AbsTol',1e-4]);
>> [T,Y]=ode23(@fsin,[0 tend],y0,options);
>> [T,Y]=ode45(@fsin,[0 tend],y0,options);
```



A Richardson extrapolation again cancels the leading (global) error, here in $\mathcal{O}(h^4)$ with $y(2) \approx 5.86390 + \frac{5.86390 - 5.86346}{2^4 - 1} = 5.86392$.