# Ordinary differential equations

Differential equations are of central importance in physics- most of physics since Newton's times is based on them!

We start by considering the first-order initial value problem, i.e.

$$y' = dy/dx = f(x,y), \quad y(x_0)=y_0$$

later we also look at higher order problems (e.g. y''=...), as well as systems of differential equations.

We will not look at boundary value problems (y values fixed at different points) and partial differential equations in this course, but this is a big and important topic for future study (e.g. Poisson's equation, Maxwell's equations,...).

# Example: population models

Assume that on an island we have a certain population of rabbits. How will their number evolve?

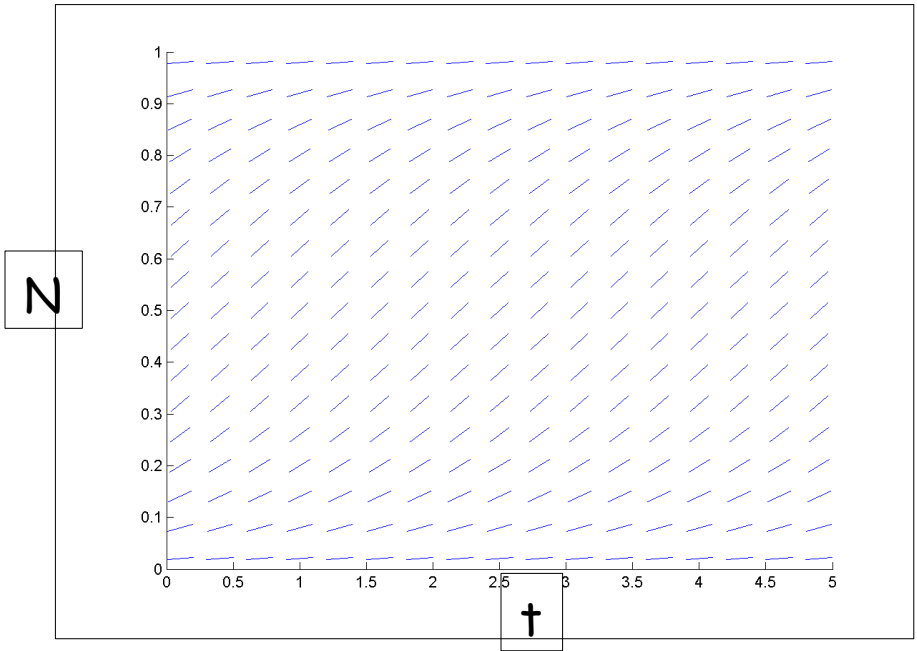
rabbits produce more rabbits → dy/dt ~ y

rabbits eat food, which is finite → dy/dt ~ 1-y

→ dy/dt = y(1-y)

"logistic equation"

the diff. eq. defines a slope-field (could also depend on t!), and we want to start with a given N at t=0 and follow the field.



# The Euler method

Consider the equation y'=f(x,y). A formal solution is given by:

$$y(x_1) = y(x_0) + \int_{x_0}^{x_1} f(x, y(x))dx$$

If f(x,y) is only a function of x this converts the problem to an integration. More typically, that is not the case, however.

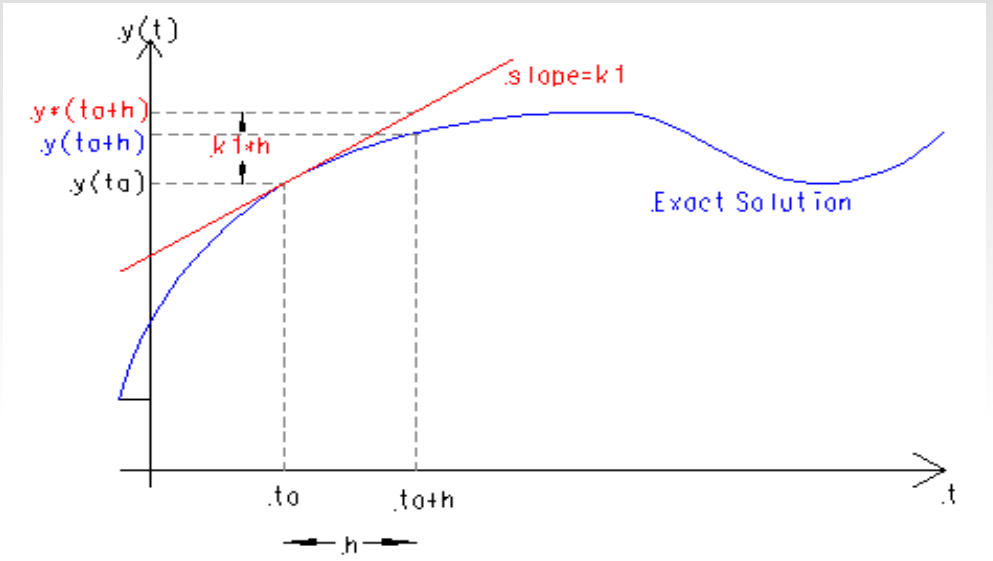
More generically, let's instead expand the function into its Taylor series, limiting ourselves to the first term:

$$y_1 \equiv y(x_1) = y(x_0 + h) \approx y(x_0) + hy'(x_0) = y_0 + hf(x_0, y_0)$$
$$y_2 = y_1 + hf(x_1, y_1); \ldots y_{k+1} = y_k + hf(x_k, y_k)$$

This gives the Euler method.

# The Euler method:graphical interpretation

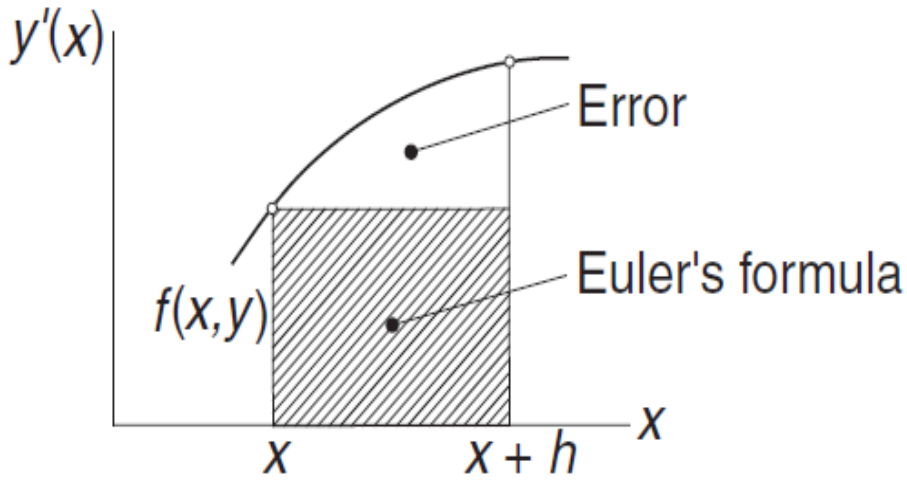
The curves that follow the slope field are curved, not straight, thus our solution will start to “lag behind” the true solution




The change in the solution of the equation is

$$y(x+h) - y(h) = \int_x^{x+h} y'\,dx = \int_x^{x+h} f(x,y)\,dx$$

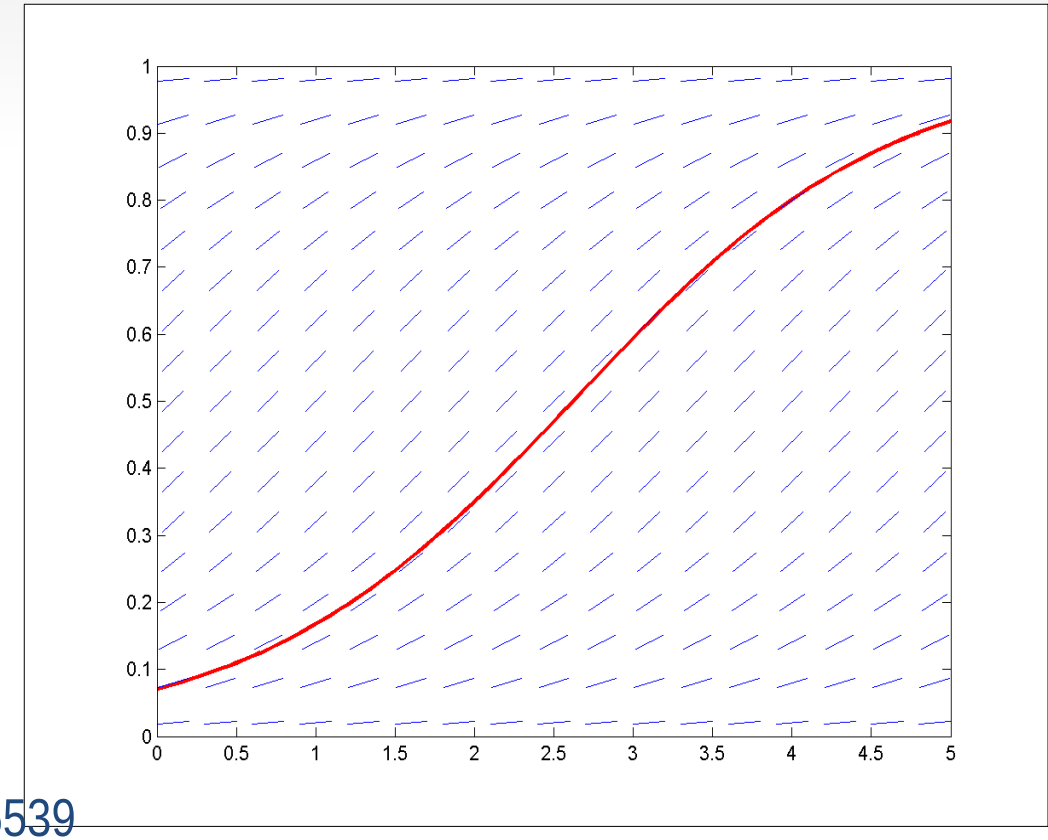
i.e. area under curve, while Euler's method gives the shaded region, with the error proportional to the slope (i.e. y”).



# The Euler method: example implementation

Implementation of the Euler method:

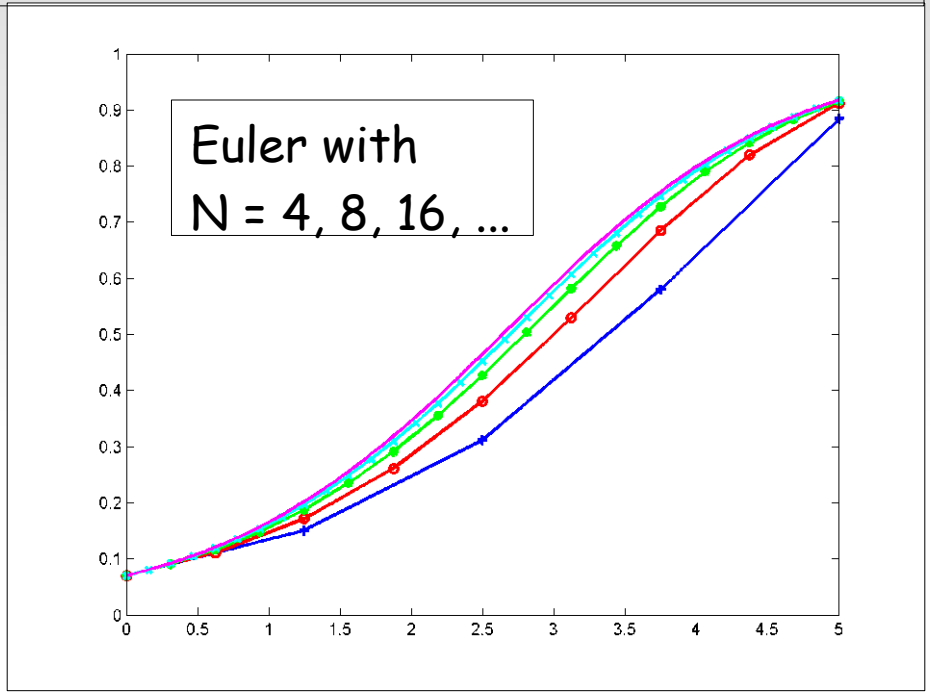
```
def odeeuler(fxy,  x0,  y0,  h,  N):
    x = x0;
    y = y0;
    for i in range(1,  N+1):
        y  +=  h * fxy( x, y)
        x  +=  h
    return y
```




http://adorio-research.org/wordpress/?p=6539

# Error in Euler's method

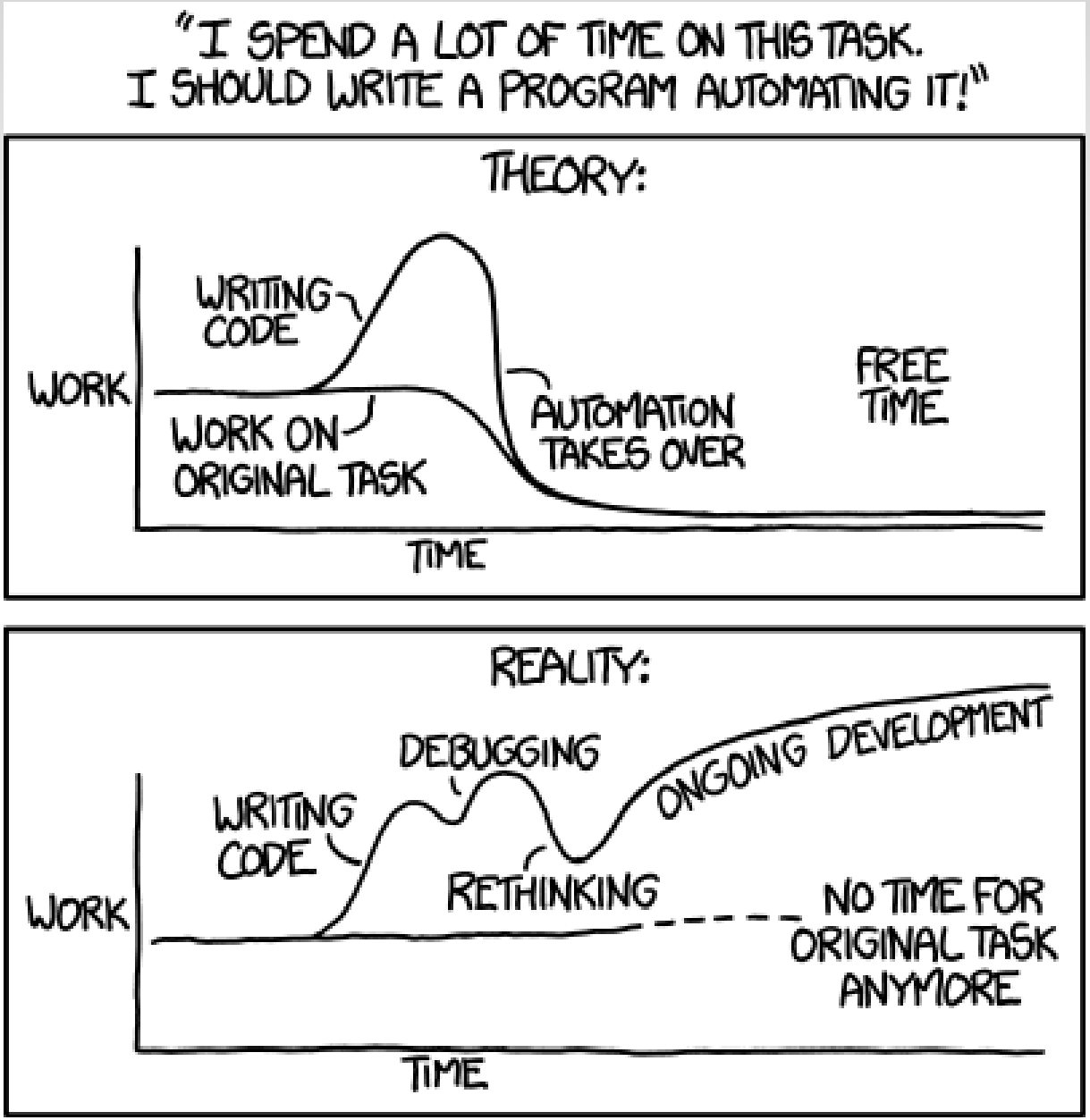
The graph shows that for smaller h (steps) the solution converges to the exact one, but also shows how the solution "lags behind". The error per step is:




$$y(x_1) = y_0 + hy_0' + \frac{h^2}{2}y''(\xi) = y_1 + h^2\frac{y''(\xi)}{2}$$

Is this method of 2nd order? No - total error after N steps is $N*h^2 = (b-a)*h$, or $O(h) = O(1/N)$, i.e. method is just 1st order.

# The life of a computing PhD student ...




XKCD comics:
http://xkcd.com/

# Runge-Kutta methods

Improve accuracy by including next term in Taylor expansion, to get $O(h^3)$:

$$y(x_1) = y_0 + hy_0' + \frac{h^2}{2}y_0'' + O(h^3)$$

Problem: how to get $y_0''$? We only need it to an accuracy of $O(h)$ since it is already multiplied by $h^2$

Use finite difference approximation, step size $\delta=\alpha h$

$$y_0'' \approx \frac{y'(x_0+\delta) - y'(x_0)}{\delta}$$

y' at $x_0+\delta$ is evaluated with a "sub"-Euler step of length $\delta$ which is (locally) accurate to $O(\delta^2)$.

# Runge-Kutta methods of second order (RK2)

Basically, we assume an integration formula given by

$$\mathbf{y}(x+h) = \mathbf{y}(x) + c_0\mathbf{F}(x,\mathbf{y})h + c_1\mathbf{F}\left[x+ph, \mathbf{y}+qh\mathbf{F}(x,\mathbf{y})\right]h$$

And try to determine the coefficients $c_0$, $c_1$, p and q so that the Taylor expansion formula is satisfied. By expanding the third term again in Taylor series and comparing coefficients (see book for details), we find the system of equations:

$$c_0 + c_1 = 1 \qquad c_1 p = \frac{1}{2} \qquad c_1 q = \frac{1}{2}$$

Clearly, we have only 3 equations for 4 unknowns, so there are multiple possible solutions. Some popular choices are:

| | | | | |
|---|---|---|---|---|
| $c_0 = 0$ | $c_1 = 1$ | $p = 1/2$ | $q = 1/2$ | Modified Euler's method |
| $c_0 = 1/2$ | $c_1 = 1/2$ | $p = 1$ | $q = 1$ | Heun's method |
| $c_0 = 1/3$ | $c_1 = 2/3$ | $p = 3/4$ | $q = 3/4$ | Ralston's method |

All these are referred to as 2nd order Runge-Kutta methods.

# Runge-Kutta methods of second order (RK2)

For example, for the modified Euler method, we have

$$\mathbf{y}(x+h) = \mathbf{y}(x) + \mathbf{F}\left[x + \frac{h}{2}, \mathbf{y} + \frac{h}{2}\mathbf{F}(x, \mathbf{y})\right] h$$

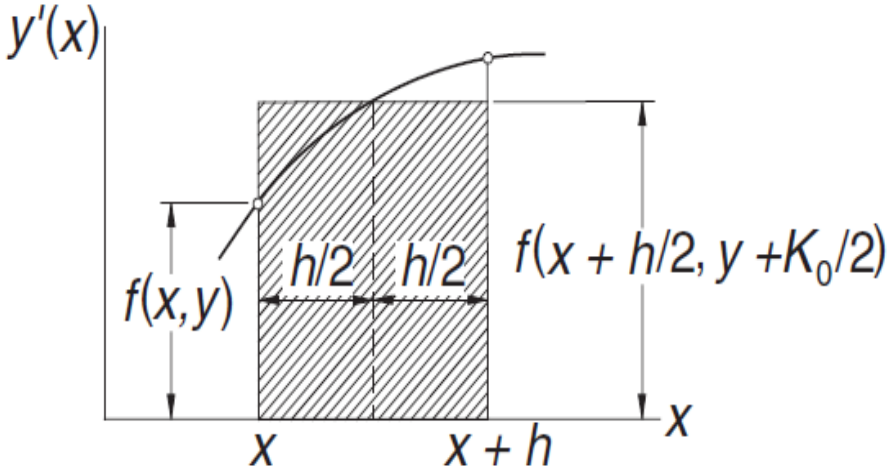
Where we evaluate the expressions by the following steps:

$$\mathbf{K}_0 = h\mathbf{F}(x, \mathbf{y})$$

$$\mathbf{K}_1 = h\mathbf{F}\left(x + \frac{h}{2}, \mathbf{y} + \frac{1}{2}\mathbf{K}_0\right)$$

$$\mathbf{y}(x+h) = \mathbf{y}(x) + \mathbf{K}_1$$

or geometrically:




and similarly for the other methods.

# Runge-Kutta methods: predictor-corrector

Another variation of second-order Runge-Kutta methods is the 2-step Heun method:

$$\begin{aligned} \tilde{y}_{n+1} &= y_n + h f(x_n, y_n) \\ y_{n+1} &= y_n + \frac{h}{2}\left[f(x_n, y_n) + f(x_{n+1}, \tilde{y}_{n+1}\right] \end{aligned}$$

which combines an Euler (predictor) step and trapezoidal (corrector) step – a simple example of a large class of predictor-corrector methods.

# RK4

Normally we use higher order Runge-Kutta methods, typically 4th order, i.e. $O(h^4)$. They are derived in the same way as RK2 (but derivation is longer and rather boring). The result is:

$$\mathbf{y}(x+h) = \mathbf{y}(x) + \frac{1}{6}(\mathbf{K}_0 + 2\mathbf{K}_1 + 2\mathbf{K}_2 + \mathbf{K}_3)$$

which is very similar to the Simpson integration method, with the slopes approximated by:

$$\mathbf{K}_0 = h\mathbf{F}(x, \mathbf{y})$$

$$\mathbf{K}_1 = h\mathbf{F}\left(x + \frac{h}{2}, \mathbf{y} + \frac{\mathbf{K}_0}{2}\right)$$

$$\mathbf{K}_2 = h\mathbf{F}\left(x + \frac{h}{2}, y + \frac{\mathbf{K}_1}{2}\right)$$

$$\mathbf{K}_3 = h\mathbf{F}(x + h, \mathbf{y} + \mathbf{K}_2)$$

Calculated in steps, as RK2.

# RK4: implementation

Implementation in Python is quite straightforward.

RK4 is very robust and widely-used method, but has one important drawback – there is no way to estimate the errors.

Solution: use adaptive step size or RK4+RK5 to estimate the errors and control the tolerance.

```
## module run_kut4
''' X,Y = integrate(F,x,y,xStop,h).
    4th-order Runge--Kutta method for solving the
    initial value problem {y}' = {F(x,{y})}, where
    {y} = {y[0],y[1],...y[n-1]}.
    x,y   = initial conditions.
    xStop = terminal value of x.
    h     = increment of x used in integration.
    F     = user-supplied function that returns the
            array F(x,y) = {y'[0],y'[1],...,y'[n-1]}.
'''
from numpy import array
def integrate(F,x,y,xStop,h):

    def run_kut4(F,x,y,h):
  # Computes increment of y from Eqs. (7.10)
        K0 = h*F(x,y)
        K1 = h*F(x + h/2.0, y + K0/2.0)
        K2 = h*F(x + h/2.0, y + K1/2.0)
        K3 = h*F(x + h, y + K2)
        return (K0 + 2.0*K1 + 2.0*K2 + K3)/6.0

    X = []
    Y = []
    X.append(x)
    Y.append(y)
    while x < xStop:
        h = min(h,xStop - x)
        y = y + run_kut4(F,x,y,h)
        x = x + h
        X.append(x)
        Y.append(y)
    return array(X),array(Y)
```

# Error estimates

Methods for RK error estimation where we compare 2 different orders are often called RK 23, RK45, etc.

e.g. for RK23 we have:

$$
\begin{aligned}
s_1 &= f(t_n, y_n), \\
s_2 &= f\left(t_n + \frac{h}{2}, y_n + \frac{h}{2}s_1\right), \\
s_3 &= f\left(t_n + \frac{3}{4}h, y_n + \frac{3}{4}hs_2\right), \\
t_{n+1} &= t_n + h, \\
y_{n+1} &= y_n + \frac{h}{9}(2s_1 + 3s_2 + 4s_3), \\
s_4 &= f(t_{n+1}, y_{n+1}), \\
e_{n+1} &= \frac{h}{72}(-5s_1 + 6s_2 + 8s_3 - 9s_4).
\end{aligned}
$$

All professional-level, robust ODE solvers do this (and more).

# Systems of ODE's

Often we have more than one variable and we need to follow their coupled evolution.

Example: In addition to the N rabbits there are also P foxes. The foxes eat the (poor) rabbits:

$dN/dt = N(a-b*P)$

$dP/dt = P(c*N-d)$

with e.g. a=c=1, b=d=2

Furthermore, higher order differential equations can be transformed into systems of 1st order ODE's:

$y''+a*y'+b*y=2$ -> introduce $z=y'$, then $y''=z'$ and

the equation becomes a system: $y'=z$ & $z'=-a*z-b*y+2$

# Systems of ODE's

This just means that we have a system of equations

$$y'_1 = f_1(x, y_1, y_2, \ldots, y_n), \quad y_1(x_0) = y_{1,0}$$

$$y'_2 = f_2(x, y_1, y_2, \ldots, y_n), \quad y_2(x_0) = y_{2,0}$$

$$\ldots$$

$$y'_n = f_n(x, y_1, y_2, \ldots, y_n), \quad y_n(x_0) = y_{n,0}$$

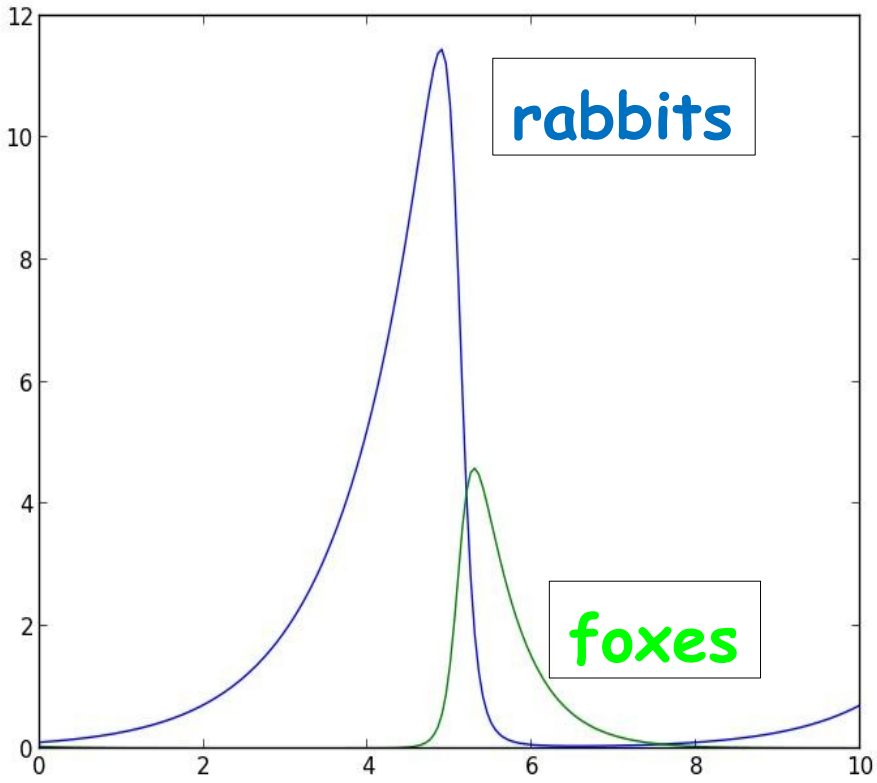
that we want to solve simultaneously. We then end up with prescriptions like

$$y'_{k+1,i} = y_{k,i} + h f_i(x, y_{k,1}, y_{k,2}, \ldots, y_{k,n})$$

for the i-th equation and the k-th step of the Euler method (and similar expressions for RK4, etc.). All methods we will consider work also for systems, but the RHS function should be a vector.

# Predator-prey example:

```
def F(x,y,a=1.0,b=2.0,c=1.0,d=2.0):
    F = zeros(2)
    F[0] = y[0]*(a-b*y[1])
    F[1] = y[1]*(c*y[0]-d)
    return F
x = 0.0 # Start of integration
xStop = 10.0 # End of integration
y = array([0.1, 0.03]) # Initial values of {y}
h = 0.05 # Step size
freq = 1 # Printout frequency
X,Y = integrate(F,x,y,xStop,h)
printSoln(X,Y,freq)
pl.plot(X,Y[:,0])
pl.plot(X,Y[:,1])
pl.show()
```



# Python (Scipy) commands

scipy.integrate.odepack: Solve a system of ordinary differential equations using lsoda from the FORTRAN library odepack.

Solves the initial value problem for stiff (to be discussed next time) or non-stiff (i.e. 'standard') systems of first order ODEs:

dy/dt = func(y,t0,...)

where y can be a vector.

scipy.integrate.ode: Interface to a number of ODE solvers, e.g.

Dopri5: an explicit Runge-Kutta method of order (4)5 due to Dormand & Prince (with stepsize control).

In either of the above cases one can also specify options (like tolerances) and extra parameters to pass to differential equation function, look at help!