Introduction

What is an ODE

Differential equations can be used to describe the time-dependent behaviour of a variable.

$$\frac{\mathrm{d}\vec{x}}{\mathrm{d}t} = f(\vec{x}, t)$$

The variable stands for a concentration or a number of individuals in a population.

In general, a first order ODE has two parts, the increasing (birth, production,...) and the decreasing (death, degradation, consumption,...) part:

$$\frac{\mathrm{d}\vec{x}}{\mathrm{d}t} = \sum \mathrm{Rates}_{\mathrm{production}} - \sum \mathrm{Rates}_{\mathrm{loss}}$$

You probably already know ways to solve a differential equation algebraically by 'separation of variables' (Trennung der Variablen) in the homogeneous case or 'variation of parameters' (Variation der Konstanten) in the inhomogeneous case. Here, we want to discuss the use of numerical methods to solve your ODE system.

Numerical integration

In principle, every numerical procedure to solve an ODE is based on the so-called "Euler" method. It's very easy to understand, you just have to read the $\frac{\mathrm{d}\vec{x}}{\mathrm{d}t}$ as a $\frac{\Delta\vec{x}}{\Delta t}$. Then you can multiply both sides of the equation with Δt and you have an equation describing the change of your variables during a certain time intervall Δt:

$$\Delta\vec{x} = f(\vec{x}, t) \cdot \Delta t$$

Of course, the smaller yoy choose the time intervall Δt, the more accurate your result will be in comparison to the analytical solution.
So it's clear, we chose a tiny one, right? Well, not exactly, the smaller your time intervall the longer the simulation will take. Therefore, we need a compromise and here the provided software will help us by constantly testing and observing the numerical solution and adapt the "step size" Δt automatically.

Lotka-Volterra: A prey-predator model

Model equations:

$$\frac{\mathrm{d}\,R}{\mathrm{d}\,t} = aR - bRW$$
$$\frac{\mathrm{d}\,W}{\mathrm{d}\,t} = cWR - dW$$

Variables:

R: Rabbit population

W: Wolf population

Parameters:

a: rabbit's birth rate

b: predation rate

c: wolf's benefit

d: wolf's death rate

Let's start

we write a small function **f**, that receives a list of the current values of our variables **x**, the current time **t** and parameters **p**. The function has to evaluate the equations of our system or $\frac{\mathrm{d}\vec{x}}{\mathrm{d}t}$, respectively. Afterwards, it returns the values of the equations as another list.
Important
*Since this function **f** is used by the solver, we are not allowed to change the input (arguments) or output (return value) of this function.*

In [4]:
```python
import numpy as np

# define ODE
def f(y, t, p):
    dydt = np.zeros(2)
    # dR/dt
    dydt[0] = p[0]*y[0] -p[1]*y[0]*y[1]
    # dW/dt
    dydt[1] = p[2]*y[1]*y[0] -p[3]*y[1]
    return dydt
```

Before we start the simulation of our model, we have to define our system.
We start with our static information:

1. Initial conditions for our variables
2. Values of the paramters
3. Simulation time and number of time points at which we want to have the values for our variables (the time grid). *Use numpy!!*

In [5]:
```
# initial values of variables
y0 = np.zeros(2)
# R(t=0)
y0[0] = 100
# W(t=0)
y0[1] = 1

# define p = [a, b, c, d]
p = np.array([1, 0.01, 0.01, 0.8])

# time grid
t = np.linspace(0,50,500)
```

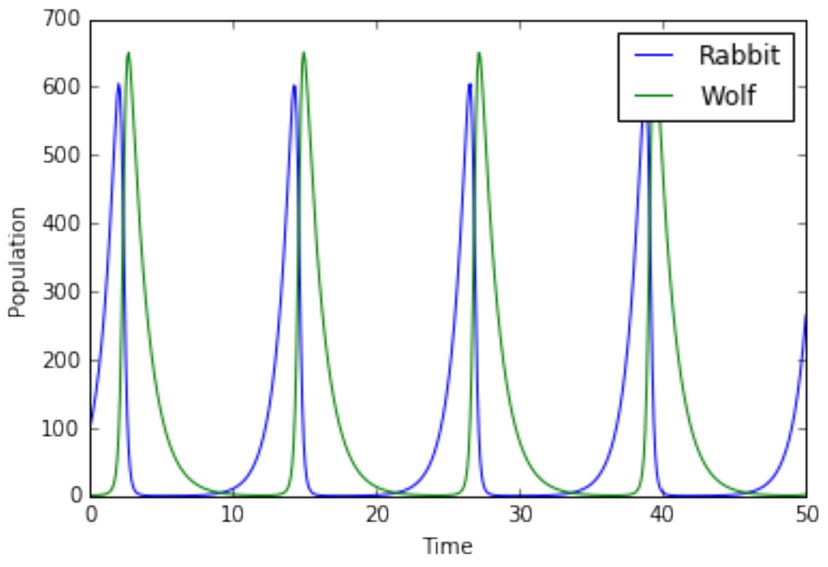
Last but not least, we need to import and call our solver. The result will be a matrix with our time courses as columns and the values at the specified time points. Since we have a values for every time point and every species, we can directly plot the results using *matplotlib*.

In [6]:
```
from scipy.integrate import odeint

# solve ODE using odeint
y = odeint(f, y0, t, (p,))
```

Plot results

In [7]:
```
%matplotlib inline
import matplotlib.pyplot as plt

plt.plot(t,y)
plt.xlabel('Time')
plt.ylabel('Population')
plt.legend(['Rabbit','Wolf'])
```

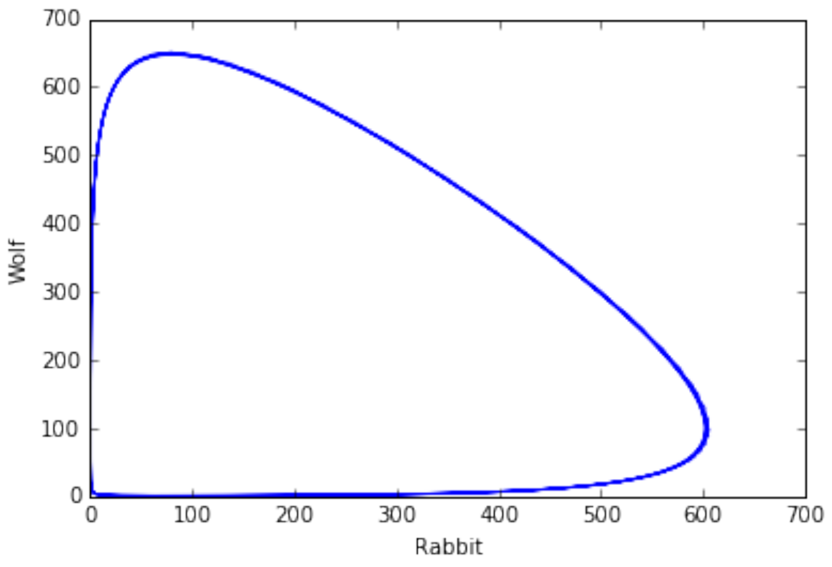
Out[7]:
```
<matplotlib.legend.Legend at 0x10a09b278>
```



Phase space diagram

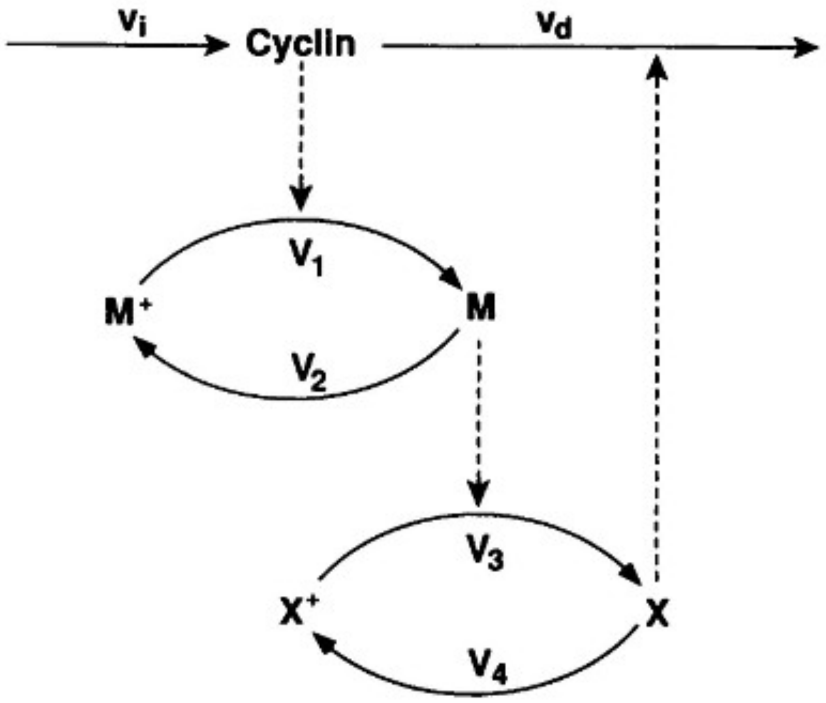
In [8]:
```
%matplotlib inline
import matplotlib.pyplot as plt

plt.plot(y[:,0],y[:,1])
plt.xlabel('Rabbit')
plt.ylabel('Wolf')
```

Out[8]:
```
<matplotlib.text.Text at 0x10ae2f400>
```



Exercises

1. Implement the Goldbeter model:



$$\frac{\mathrm{d}C}{\mathrm{d}t} = v_i - v_d X \frac{C}{K_d + C} - k_d C$$

$$\frac{\mathrm{d}M}{\mathrm{d}t} = V_1 \frac{1-M}{K_1 + (1-M)} - V_2 \frac{M}{K_2 + M}$$

$$\frac{\mathrm{d}X}{\mathrm{d}t} = V_3 \frac{1-X}{K_3 + (1-X)} - V_4 \frac{X}{K_4 + X}$$

with $V_1 = \frac{C}{K_c + C} V_{M1}$ and $V_3 = M V_{M3}$

and $M + M^* = 1$ and $X + X^* = 1$

Tasks:

- Plot the trajectories and the phase space diagrams
- Perform parameter sampling (20 random parameter sets for the interval [0, 1]) and plot trajectories and phase space diagrams for each parameter set
- Scan the parameter 'Kd' in a range where the solution oscillates (~20 values) and plot trajectories and phase space diagrams for each parameter set
- Determine frequency and amplitude of those solutions and plot them in dependency of the 'Kd'

Solution

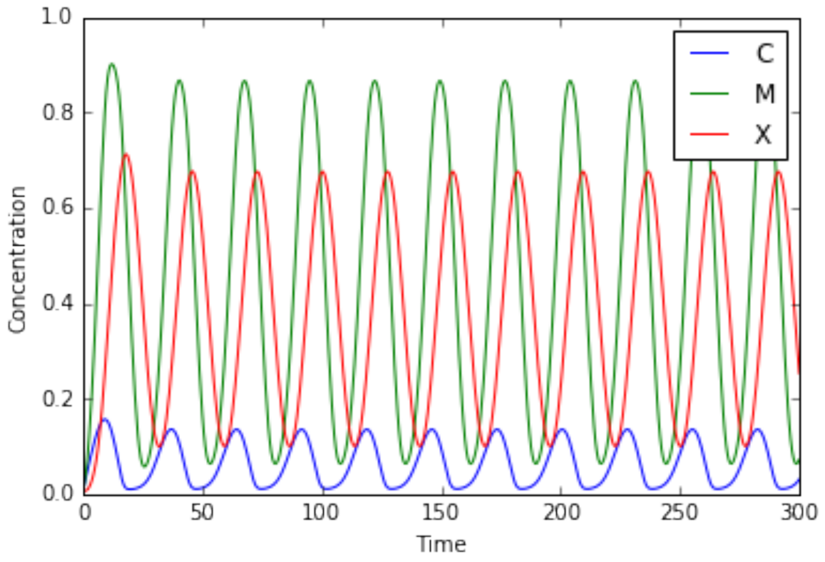
In [9]:
```
def f(y,t,p):

    V1 = y[0]/(p['Kc']+y[0])
    V3 = y[1]*p['V_M3']

    dydt = np.zeros(3)
    # dC/dt
    dydt[0] = p['vi'] -p['vd']*y[2]*y[0]/(p['Kd']+y[0]) -p['kd']*y[
0]
    # dM/dt
    dydt[1] = V1*(1-y[1])/(p['Ki']+(1-y[1])) -p['V2']*y[1]/(p['Ki']
+y[1])
    # dX/dt
    dydt[2] = V3*(1-y[2])/(p['Ki']+(1-y[2])) -p['V4']*y[2]/(p['Ki']
+y[2])

    return dydt
```

In [10]:
```
p = {'vi':0.023,
     'vd':0.1,
     'Kd':0.02,
     'kd':3e-3,
     'Ki':0.1,
     'V_M1':0.5,
     'V2':0.167,
     'V_M3':0.2,
     'V4':0.1,
     'Kc':0.3}

# initial values of variables
y0 = np.zeros(3)
# C(t=0)
y0[0] = 0.01
# M(t=0)
y0[1] = 0.01
# X(t=0)
y0[2] = 0.01

# time grid
t = np.linspace(0,300,500)
```

In [11]:
```
# solve ODE using odeint
y = odeint(f, y0, t, (p,))
```

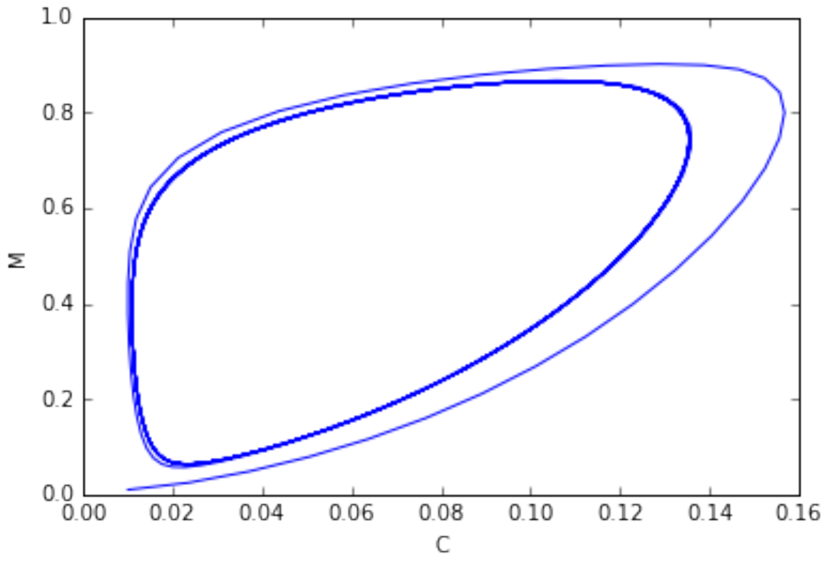
In [12]:
```
%matplotlib inline
import matplotlib.pyplot as plt

plt.plot(t,y)
plt.xlabel('Time')
plt.ylabel('Concentration')
plt.legend(['C','M','X'])
```

Out[12]:
```
<matplotlib.legend.Legend at 0x10a082dd8>
```

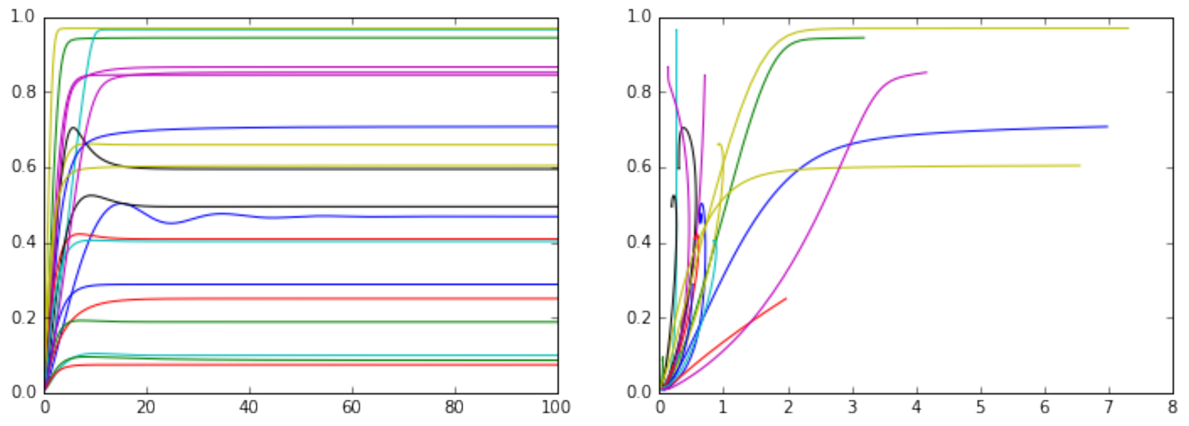


In [13]:
```
%matplotlib inline
import matplotlib.pyplot as plt

plt.plot(y[:,0],y[:,1])
plt.xlabel('C')
plt.ylabel('M')
```

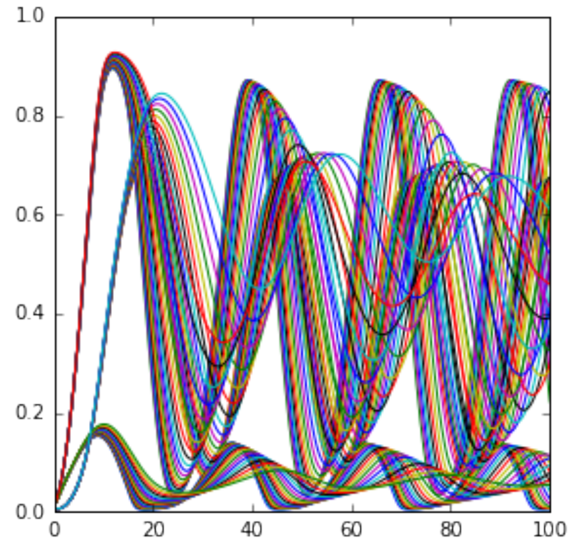
Out[13]:
```
<matplotlib.text.Text at 0x10b0d1e48>
```



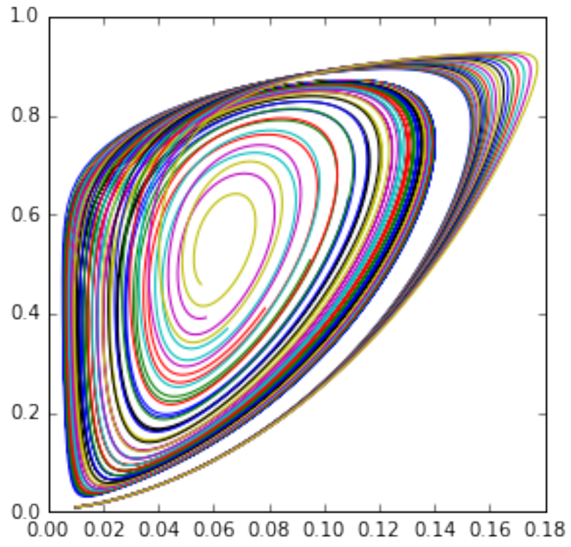
In [14]:
```
plt.figure(figsize=(12,4))

t = np.linspace(0,100,500)
for i in range(20):
    for pi in p:
        p[pi] = np.random.rand()
    y = odeint(f, y0, t, (p,))
    plt.subplot(1,2,1)
    plt.plot(t,y[:,1])
    plt.subplot(1,2,2)
    plt.plot(y[:,0],y[:,1])
```



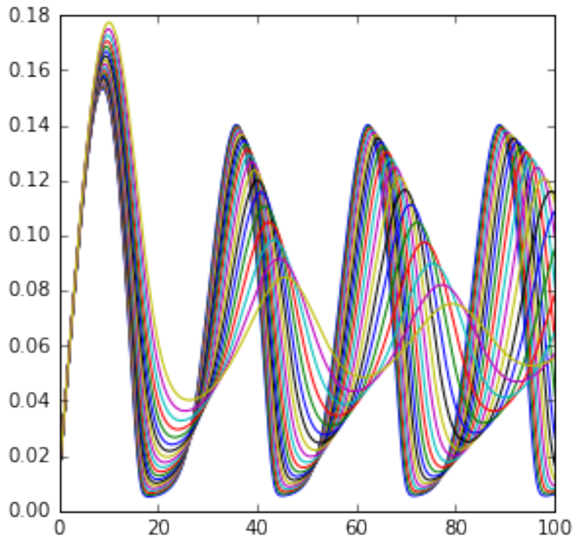
In [15]:
```
t = np.linspace(0,100,500)
p = {'vi':0.023,
     'vd':0.1,
     'Kd':0.02,
     'kd':3e-3,
     'Ki':0.1,
     'V_M1':0.5,
     'V2':0.167,
     'V_M3':0.2,
     'V4':0.1,
     'Kc':0.3}

p_scan = np.logspace(-2,-1,20)

plt.figure(figsize=(10,10))
for pi in p_scan:
    p['Kd'] = pi
    y = odeint(f, y0, t, (p,))

    plt.subplot(2,2,1)
    plt.plot(t,y)

    plt.subplot(2,2,2)
    plt.plot(y[:,0],y[:,1])

    plt.subplot(2,2,3)
    plt.plot(t,y[:,0])
```



1.0
0.8
0.6
0.4
0.2
0.0
0
20
40
60
80
100



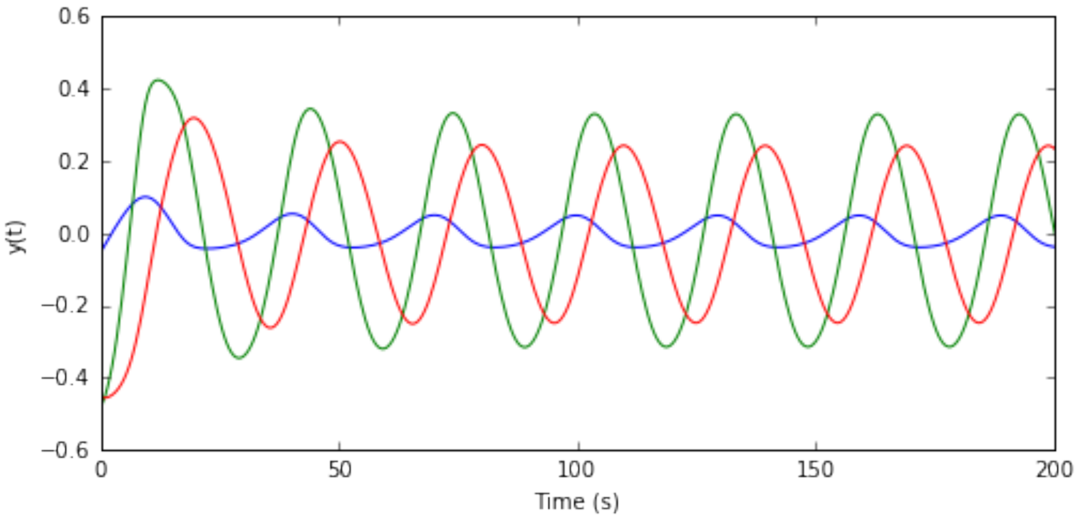
1.0
0.8
0.6
0.4
0.2
0.0
0.00
0.02
0.04
0.06
0.08
0.10
0.12
0.14
0.16
0.18

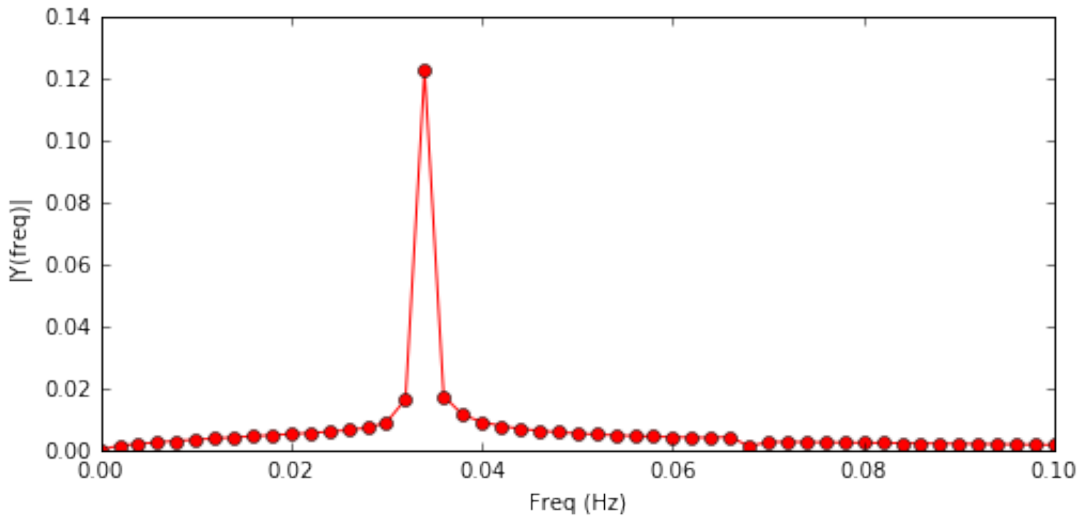


0.18
0.16
0.14
0.12
0.10
0.08
0.06
0.04
0.02
0.00
0
20
40
60
80
100

In [16]:
```
p['Kd'] = 0.05

T = 500
time_steps = 1000
t = np.linspace(0,T,time_steps)
y = odeint(f, y0, t, (p,))
centered_y = y - y.mean(axis=0)

Fs = time_steps/T
Ts = T/Fs

k = np.arange(time_steps)
frq = k/T # two sides frequency range
frq = frq[:int(time_steps/2)] # one side frequency range

Y = np.fft.fft(centered_y[:,2])/time_steps # fft computing and norm
alization
Y = Y[:int(time_steps/2)]

plt.figure(figsize=(8,8))

plt.subplot(2,1,1)
plt.plot(t,centered_y)
plt.xlim(0.0,200.0)
plt.xlabel('Time (s)')
plt.ylabel('y(t)')

plt.subplot(2,1,2)
plt.plot(frq,abs(Y),'-or') # plotting the spectrum
plt.xlim(0.0,0.1)
plt.xlabel('Freq (Hz)')
plt.ylabel('|Y(freq)|')

abs_Y = abs(Y)
max_frq_index = np.argmax(abs_Y)
max_frq = frq[max_frq_index]
print("Frequency with maximal amplitude: %2.2f" % max_frq)

T_period = 1.0/max_frq
print("Corresponding time period: %2.2f" % T_period)
```

```
Frequency with maximal amplitude: 0.03
Corresponding time period: 29.41
```

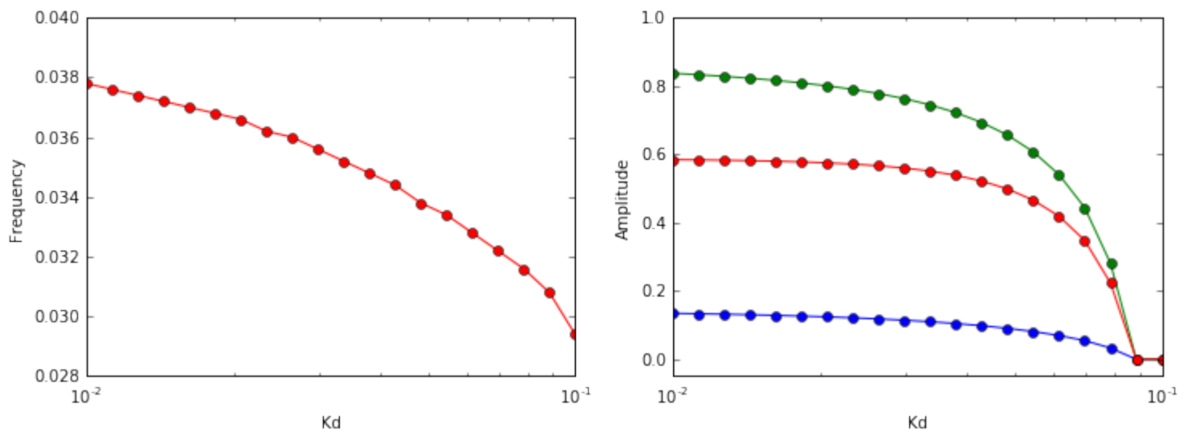




In [19]:
```
T = 5000
time_steps = 10000
t = np.linspace(0,T,time_steps)

p_scan = np.logspace(-2,-1,20)
frequency = np.zeros_like(p_scan)
amplitude = np.zeros([p_scan.shape[0],y.shape[1]])
for i, pi in enumerate(p_scan):
    p['Kd'] = pi
    y = odeint(f, y0, t, (p,))
    centered_y = y - y.mean(axis=0)

    Fs = time_steps/T
    Ts = T/Fs

    k = np.arange(time_steps)
    frq = k/T # two sides frequency range
    frq = frq[:int(time_steps/2)] # one side frequency range

    Y = np.fft.fft(centered_y[:,2])/time_steps # fft computing and
normalization
    Y = Y[:int(time_steps/2)]

    abs_Y = abs(Y)
    max_frq_index = np.argmax(abs_Y)
    max_frq = frq[max_frq_index]
    frequency[i] = max_frq
    amplitude[i,:] = abs(y[int(time_steps/2):].min(axis=0)-y[int(ti
me_steps/2):].max(axis=0))
```

```
In [20]: plt.figure(figsize=(12,4))

         plt.subplot(1,2,1)
         plt.semilogx(p_scan,frequency,'-or')
         plt.xlabel('Kd')
         plt.ylabel('Frequency')
         plt.ylim([0.028, 0.04])

         plt.subplot(1,2,2)
         plt.semilogx(p_scan,amplitude,'-o')
         plt.xlabel('Kd')
         plt.ylabel('Amplitude')
         plt.ylim([-0.05, 1])
```

```
Out[20]: (-0.05, 1)
```



```
In [ ]:
```