FACHKURS

Modellierung Biochemischer Netzwerke
Institut für Theoretische Biologie
31 Januar.-10. Februar 2017
Teil I

Ralf Steuer, Marjan Faizi
Humboldt-Universität zu Berlin
Institut für Theoretische Biologie
Invalidenstr. 43, 10115 Berlin
ralf.steuer@hu-berlin.de

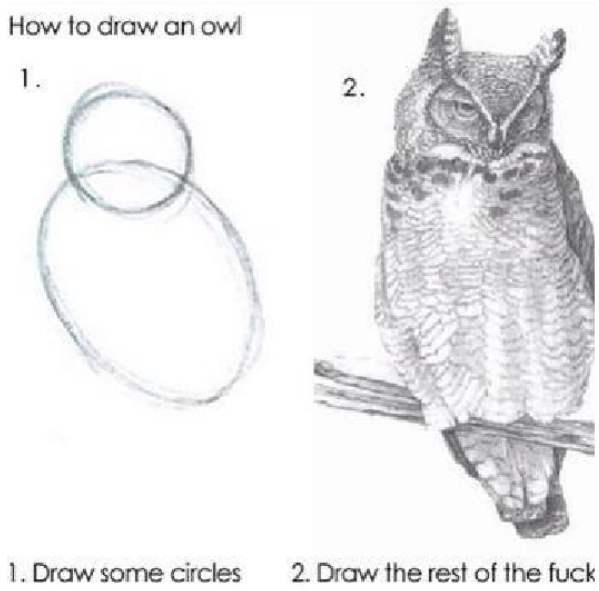
February 2, 2017

This handout gives a brief introduction into some aspects of computational modeling in biology. We will use the software 'python', which is available from python.org. This handout is to accompany the lecture and exercises, It cannot *replace* lectures, the study of original literature or textbooks. For questions please contact: ralf.steuer@hu-berlin.de

Part I
Introduction and Numerical Methods

Introduction to PYTHON

We will provide detailed step-by-step instructions for all steps.



Nonetheless, this script cannot replace a proper tutorial in Python.

The Basics

To start python in Linux type: **python**.

```
user@rechner:~$ python
Python 2.7.12 (default, Nov 19 2016, 06:48:10)
[GCC 5.4.0 20160609] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

The basic view of python is the command line interface. Here variables can be defined. Some examples:

```
>>> var = 4
>>> print var
4
>>> print var + 2
6
>>> var += 2
```

```
>>> print var
6

>>> text = "Hello World"
>>> print text
Hello World

x = 1.2
y = 3.4
print [x+y, x*y, x**y]
```

We also need to use lists

```
>>> listy = [2 , 3.4 , 7 , 0.5]
>>> print listy
[2, 3.4, 7, 0.5]
>>> listy[0]
2
>>> listy[3]
0.5
>>> listy[3] = 1.5
>>> print listy
[2, 3.4, 7, 1.5]
>>>

>>> print listy[0:3]
[2, 3.4, 7]

>>> l1 = [1 , 2]
>>> l2 = [2, 3]
>>> print l1+l2
[1, 2, 2, 3]
```

We statements that control the flow of a program (if, for while). The most import command is, as usual, `help`. Use it!

User-defined functions in `Python`

In addition to built-in functions, we can use our own functions and scripts. In general commands placed in a file can be executed as if typed directly in the command line. Define a file with the name `examplescript.py` with the content:

```
# file: examplescript.py
# everything after '#' is a comment

x = 1.2
```

```
y = 3.4
print [x+y, x*y, x**y]
```

If the script is called, the result is identical as if the content had been typed into the command line.

```
user@rechner:~$ python examplescript.py
[4.6, 4.08, 1.858729691979481]
```

Similar to scripts also *functions* can be defined. We use the file `my_func.py`. The file ending is always `.py`.

```
# The function computes the sum of two numbers
# usage: z = simple_sum(x,y)

def simple_sum(x,y):
    z = x + y
    return z
```

The script has to be imported as module so that the function can then be called by the '.'-operator.

```
>>> import my_func
>>> my_func.simple_sum(4,3)
7

>>> a = 5
>>> z = my_func.simple_sum(a,3)
>>> z
8
```

The name of the module can also be abbreviated

```
>>> import my_func as mf
>>> g = mf.simple_sum(a,-3)
>>> g
2

>>> g = mf.simple_sum(1)
```

What is the error in this command?

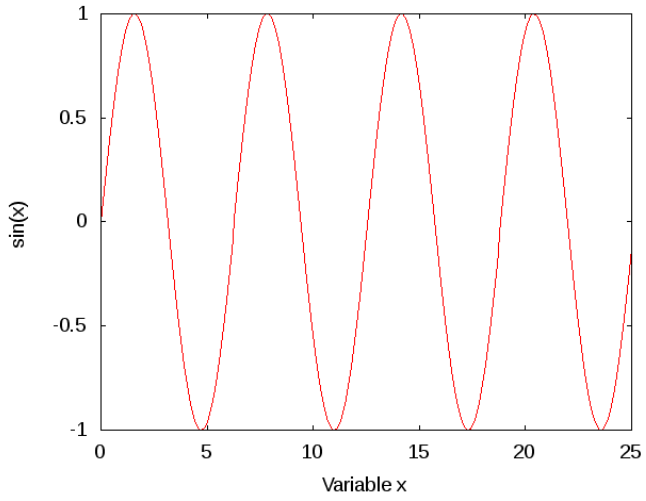
Plots in Python

Simple plots can be generated with the command `plot`. To use this built-in function, the package `matplotlib.pyplot` is required.

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt

>>> x = np.arange(0,25,0.1)
>>> plt.plot(x,np.sin(x),'r-')
[<matplotlib.lines.Line2D object at 0x7f07b758cf90>]
>>> plt.xlabel('Variable x')
```

```
<matplotlib.text.Text object at 0x7f07b761b150>
>>> plt.ylabel('sin(x)')
<matplotlib.text.Text object at 0x7f07b75b0090>
>>> plt.show()
```

For more information see `help(plt.plot)`.



- **Exercise 1:** Explore different plot functions. How can the color of lines be specified? What are the commands for three dimensional plots? Look at the commands: `hold, axis, xlabel, ylabel`.
- **Exercise 2:** Try the following commands and explain what is plotted:

```
>>> x = np.arange(0,12,0.1)
>>> y = np.arange(0,10,0.1)
>>> xx, yy = np.meshgrid(x,y)
>>> z = np.sin(yy)*np.cos(xx)
>>> cp = plt.contourf(xx,yy,z)
>>> plt.xlabel('x')
<matplotlib.text.Text object at 0x7fdc7d227d10>
>>> plt.ylabel('y')
<matplotlib.text.Text object at 0x7fdc7a28fb90>
>>> plt.colorbar(cp)
<matplotlib.colorbar.Colorbar instance at 0x7fdc79f5e8c0>
>>> plt.show()
```

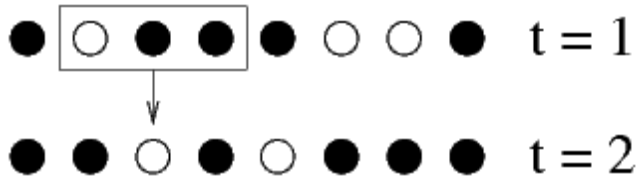
Boolean Networks and Cellular Automata

We seek to simulate a basic Boolean network.

```
# this script iterates a simple Boolean network
#
# [A, B, C] --> [A, B, C]
#

K = 10 # iterate K times

# Rules for A, B, and C
A_table = [1,0]
B_table = [1,1,1,0]
C_table = [1,0,0,0]

# Start value
X = [[1,0,1]]

for i in range(K):
    A_t = X[i][0]
    B_t = X[i][1]
    C_t = X[i][2]

    A_ix = A_t
    B_ix = 2*(A_t-1)+B_t
    C_ix = 2*(A_t-1)+B_t
    print A_ix, B_ix,C_ix
    A_new = A_table[A_ix]
    B_new = B_table[B_ix]
    C_new = C_table[C_ix]

    X.append([A_new,B_new,C_new])

    print X
```

Be careful: python uses zero-based indexing.

We now seek to implement a more complicated cellular automaton. For each element, the next state is determined by itself and its two neighbors.



All elements behave according to the same rule, i.e., we only have to specify the outcome for each of the possible 8 inputs.

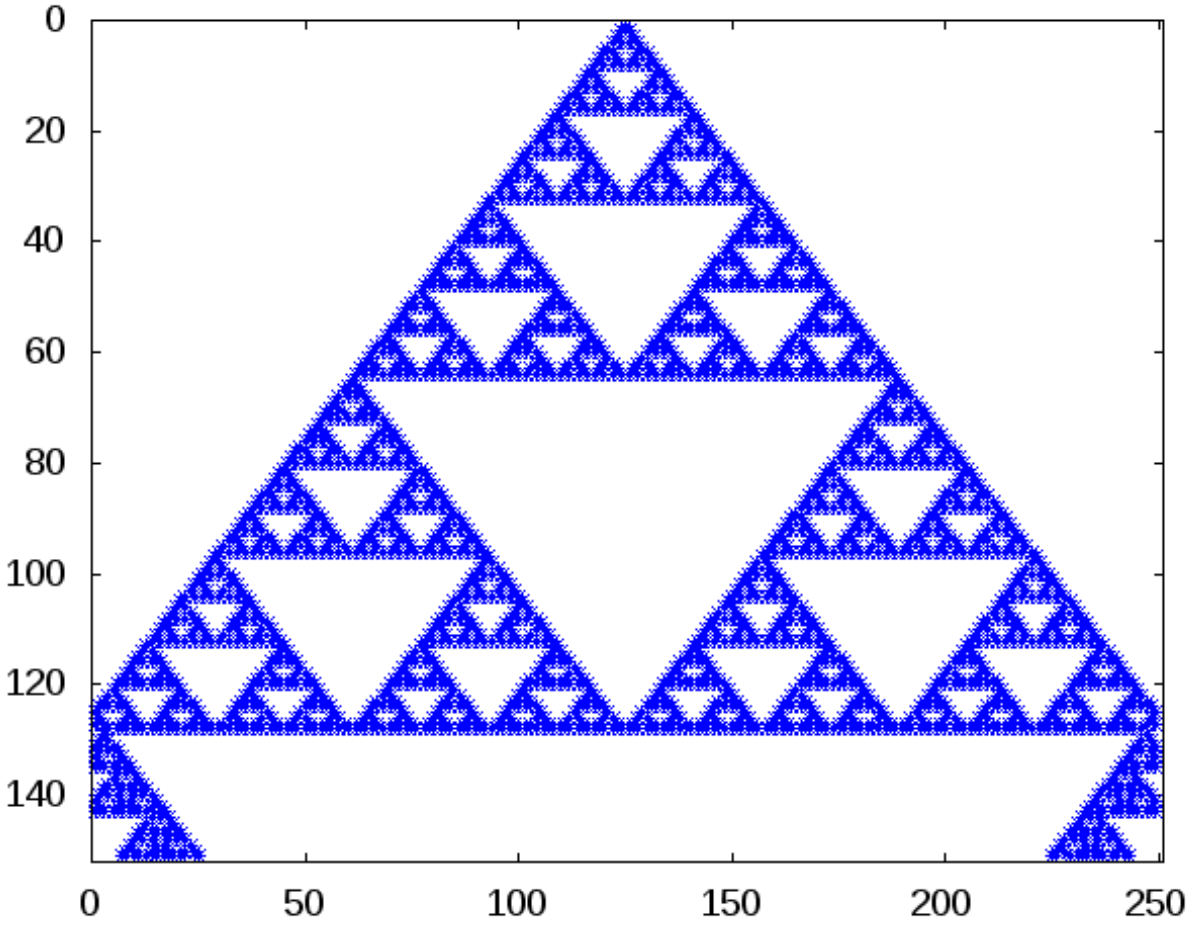
$$x_i(t+1) = F\left[x_{i-1}(t), x_i(t), x_{i+1}(t)\right] \tag{1}$$

Here F denotes a Boolean function $\in \{0, 1\}$.

```
# function: X = CellAutoSimpler(N,K)
# N: Size of System
# K: Number of iterations
# with default values N=100 and K=150

import numpy as np

def CellAutoSimpler(N=100,K=150):
# Set the rules
#                      000 001 010 011 100 101 110 111
#                     ---------------------------------
    RULES = np.array([ 0,  1,  0,  1,  1,  0,  1,  0 ])

    X = np.zeros((K,N),dtype=int)
    X[0,np.floor(N/2)] = 1

    # ITERATION
    for i in range(K-1):
        # set periodic boundaries
        X_idx = np.concatenate(([X[i,N-1]], X[i,:], [X[i,0]]))
        # calculate index
        index = X_idx[:N]*4 + 2*X_idx[1:N+1] + X_idx[2:N+2]
        # update according to RULES
        X[i+1,:] = RULES[index]

    return X
```

The result can be visualized using

```
>>> import my_func as mf
>>> X = mf.CellAutoSimpler(250,150)
>>> plt.spy(X)
<matplotlib.image.AxesImage object at 0x7f9fdab458d0>
>>> plt.show()
```



Use `help(plt.spy)` for more information.

- **Exercise:** Try different rules.

Finite-Difference Equations and Maps

We now iterate the logistic map

$$x_{n+1} = f(x_n) = rx_n(1 - x_n) \qquad r \in [0, 4] \tag{2}$$

with fix-points $x_n = f(x_n)$:

$$x^0 = \begin{cases} 0 & \text{für} \quad r \in [0, 4] \\ (r-1)/r & \text{für} \quad 1 \leq r \leq 4 \end{cases} \tag{3}$$

to test for stability, we use $f'(x) = r(1 - 2x)$

$$f'(x^0) = \begin{cases} r & : \quad \text{stable for} \;\; 0 \leq r < 1 \\ 2 - r & : \quad \text{stable for} \;\; 1 < r < 3 \end{cases} \tag{4}$$

A simple function to iterate the map is (write the function into a file):

```
# Function x = LogMap(x,R) iterates the logistic map
# R can be an array

def LogMap(x,R):
    x = R*x*(1-x)
    return x
```

The map can be iterated for different values of R. We start with setting x to a random value (`help(np.random)`).

```
>>> R = 2.5
>>> x = np.random.rand()
>>> x = mf.LogMap(x,R)
>>> x
0.15548177398954394
>>> x = mf.LogMap(x,R)
>>> x
0.3282679798665208
>>> x = mf.LogMap(x,R)
>>> x
0.5512702831521857
>>> x = mf.LogMap(x,R)
>>> x
0.6184283951637368
```

We can iterate the map as a loop using the command line

```
>>> x = [np.random.rand()]
>>> for i in range(1,100):
...     x.append(mf.LogMap(x[i-1],R))
```

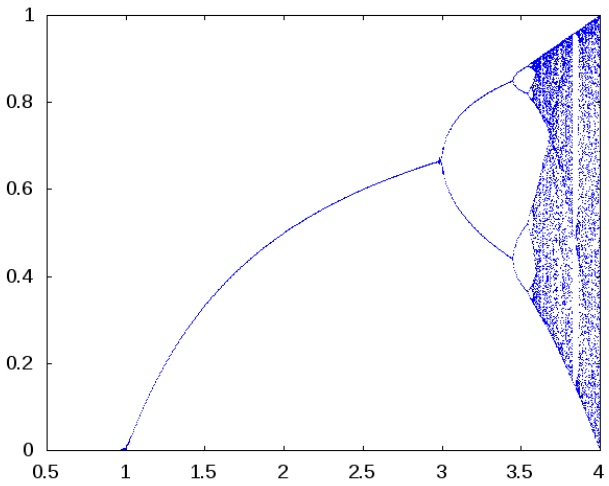
```
...
>>>
>>> plt.plot(x,'o')
[<matplotlib.lines.Line2D object at 0x7fd7b2693950>]
>>> plt.show()
```

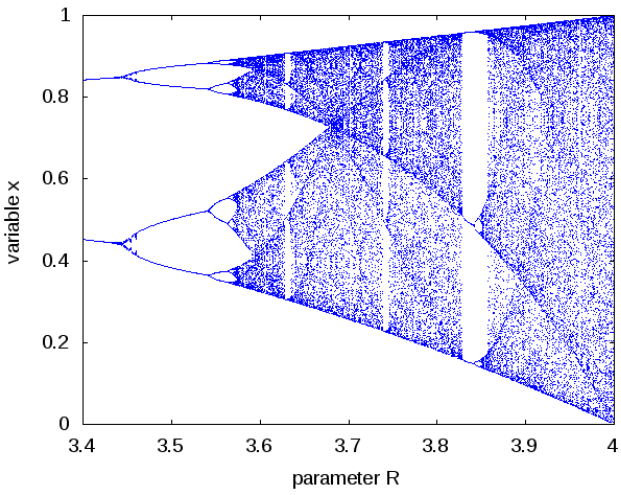
But it is far more efficient to use a script like:

```
R = np.arange(0,4,0.05)
x = [np.random.rand(np.size(R))]

# pre-iteration
for i in range(50):
    x[0] = mf.LogMap(x[0],R)

# actual iteration (values are stored)
for i in range(100):
    x.append(mf.LogMap(x[i],R))

plt.plot(R,np.transpose(x),'o')
plt.show()
```

The iteration is done for values of $0 \leq R \leq 4$.

- **Exercise 1:** Explore the region for $R > 3.4$ using a higher resolution.





Differential equations and numerical integration

We will mainly use ordinary differential equations of the form.

$$\frac{d\vec{x}}{dt} = \vec{f}(\vec{x}, \vec{p}, t) \tag{5}$$

Here $\vec{x}$ is a vector of state variables at time t. The parameters of the system are represented by the vector $\vec{p}$.
In one dimension, the system is written as

$$\frac{dx}{dt} = f(x) \quad \text{with} \quad x(t=0) := x_0 \tag{6}$$

And the time-invariant steady states are

$$\frac{dx}{dt} = 0 \quad \Leftrightarrow \quad f(x^*) = 0 \tag{7}$$

The stability of the steady state is determined by the derivative: $f'(x^*) < 0$.

The simplest way to solve equation (6) numerically is the Euler integration

$$f(x) = \frac{dx}{dt} \approx \frac{x(t+\Delta t) - x(t)}{\Delta t} \quad . \tag{8}$$

We obtain

$$x(t+\Delta t) \;=\; x(t) \;+\; \Delta t\; f(x(t)) \;+\; \mathcal{O}(\Delta t^2) \tag{9}$$

Starting from an initial value x_0 at time $t = 0$ the solution can now be determined for later time points.

It is of importance to consider the error of the method. The Euler method introduces an error of $\mathcal{O}(\Delta t^2)$ per integration step. To obtain the solution $x(t)$ at a time $t = T$, $N = T/\Delta t$ integration steps have to be performed. The total error is therefore of the order $\mathcal{O}(T\Delta t)$ and decreases with decreasing Δt. Euler integration is a first-order method. The method is rarely used in real life (too inefficient).

The Euler Method

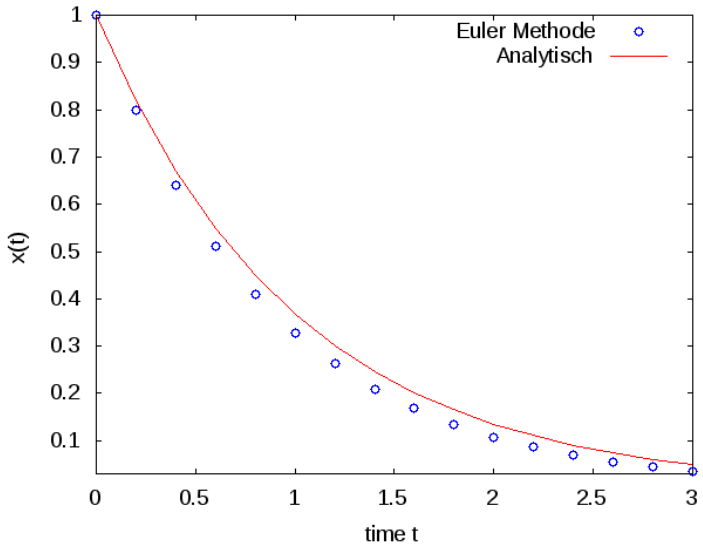
$$\frac{dx}{dt} = f(x) = -kx \quad \text{with} \quad x(0) = x_0 \tag{10}$$

For $k > 0$ the equation has a stable fix-point at $x = 0$. The analytical solution at time t is

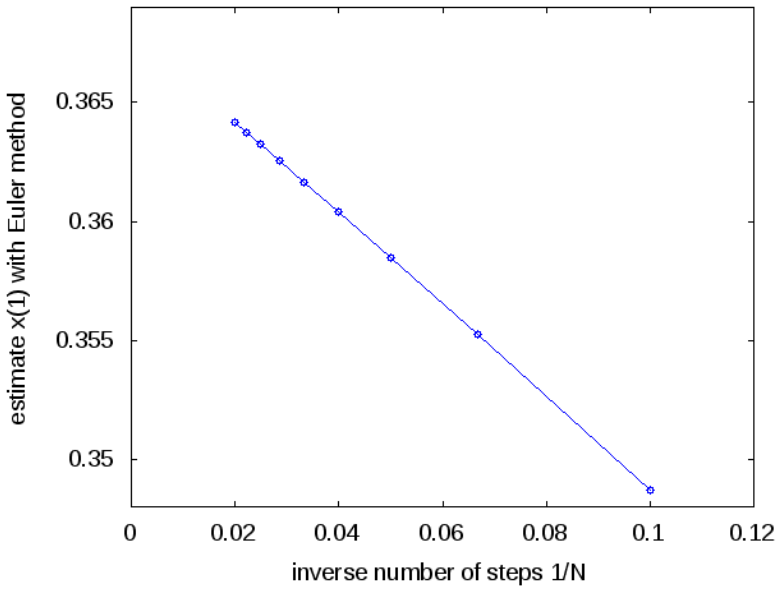
$$x(t) = x_0 \exp(-kt) \tag{11}$$

We now write a simple function that compares the numerical integration of the simple system with the (known) analytical solution.

```
# The function integrates the simple
# system dx/dt = -k x to a time T using the
# Euler method (N Steps) and initial condition x0.
#
# usage: x = SimpleEuler(x0,T,N)

import numpy as np
import matplotlib.pyplot as plt

def SimpleEuler(x0,T,N):
    k = 1 # set parameter k

    # some parameters
    dt = float(T)/N
    timespan = np.arange(0,T,dt)
    x = [float(x0)]

    # integration
    for i in range(1,N):
        x.append(x[i-1]+dt*(-k*x[i-1]))
    # plot both solutions
    plt.plot(timespan,x,'bo',markersize=8,label='Euler method')
    plt.plot(timespan,x0*np.exp(-k*timespan),'r-',label='analytical solution')
    plt.xlabel('time t')
    plt.ylabel('x(t)')
    plt.legend(loc='upper right')
    plt.show()

    # return value of function
    x = x[N-1]

    return x, dt
```



- **Exercise:** We do know the true value $x(t = 1)$ using an analytical solution. Compare the numerical estimates for different values of N and plot the error as a function of $1/N$. What does N stand for? How does the graph look like? Why?



Numerical integration in `Python`

The module `scipy.integrate` offers a variety of build-in functions for numerical integration. We will mainly use the function `odeint`.

```
>>> from scipy.integrate import odeint
>>> help(odeint)

Help on function odeint in module scipy.integrate.odepack:
```

```
odeint(func, y0, t, args=(), Dfun=None, col_deriv=0, full_output=0, ml=
None, mu=None, rtol=None, atol=None, tcrit=None, h0=0.0, hmax=0.0, hmin
=0.0, ixpr=0, mxstep=0, mxhnil=0, mxordn=12, mxords=5, printmessg=0)

    Integrate a system of ordinary differential equations.

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

    Solves the initial value problem for stiff or non-stiff systems of
    first order ode-s::

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

    where y can be a vector.

    *Note*: The first two arguments of ``func(y, t0, ...)`` are in the
    opposite order of the arguments in the system definition function
    used by the `scipy.integrate.ode` class.
```

To integrate a function $\dot{x} = f(x,t)$ we have to import the built-in function `odeint` from the module `scipy.integrate`.

A simple Example

We first consider a simple linear ODE of the form

$$\frac{dx}{dt} = c - k \cdot x \quad , \tag{12}$$

where c and k are parameters. The steady state x^0 of the system can be strightforwardly calculated

$$\frac{dx}{dt} = 0 \quad \leftrightarrow \quad x^0 = \frac{c}{k} \tag{13}$$

To solve the system numerically, we must implement the function $f(x,t) = c - k \cdot x$ into a user-defined `python` function.

```
# The function implements the simple linear
# ODE dxdt = c -k*x

def simple_ode(x,t):
    # define parameters
    c = 1.0
    k = 2.0

    dxdt = c - k*x

    return dxdt
```

To integrate the system numerically, we need to specify the initial condition $x^0 = x(t = 0)$ and a timespan. Using `odeint`:

```
from scipy.integrate import odeint
from my_func import simple_ode

# intital condition and timespan
T = np.arange(0,10,0.1)
X0 = 0

X = odeint(simple_ode,X0,T)
plt.plot(T,X,linewidth=2)
plt.show()
```

We may repeat the integration using different initial conditions.

```
X0 = 1

X = odeint(simple_ode,X0,T)
plt.plot(T,X,'r',linewidth=2)
plt.show()
```

A (not so simple) example: the Lorenz system

The Lorenz is small well-known ODE that shows some interesting behavior. From wikipedia:

> The Lorenz system is a system of ordinary differential equations (the Lorenz equations) first studied by Edward Lorenz. It is notable for having chaotic solutions for certain parameter values and initial conditions. In particular, the Lorenz attractor is a set of chaotic solutions of the Lorenz system which, when plotted, resemble a butterfly or figure eight.
>
> ...
>
> In 1963, Edward Lorenz developed a simplified mathematical model for atmospheric convection. [...]

The Lorenz system is three-dimensional, the equations are

$$\begin{aligned} \frac{dx}{dt} &= \sigma(y(t) - x(t)) \\ \frac{dy}{dt} &= -x(t)\, z(t) + rx(t) - y(t) \\ \frac{dz}{dt} &= x(t)\, y(t) - bz(t) \end{aligned} \tag{14}$$

We have to implement the corresponding function file that returns the derivative. This time the function return a vector with three components. Note that here x is understood as $x = [x, y, z]$.

```
# implements the Lorenz System
# x is a three-dimensional vector

def LorenzSys(x,t):

    # define parameters
    r = 45.92
    b = 4.0
    sig = 16.0

    dxdt =  sig*(x[1]-x[0]);
    dydt = -x[0]*x[2] + r*x[0] - x[1]
    dzdt =  x[0]*x[1] - b*x[2]

    # the function returns the vector fx
    df = [dxdt,dydt,dzdt]
    return df
```

The numerical integration is as usual:

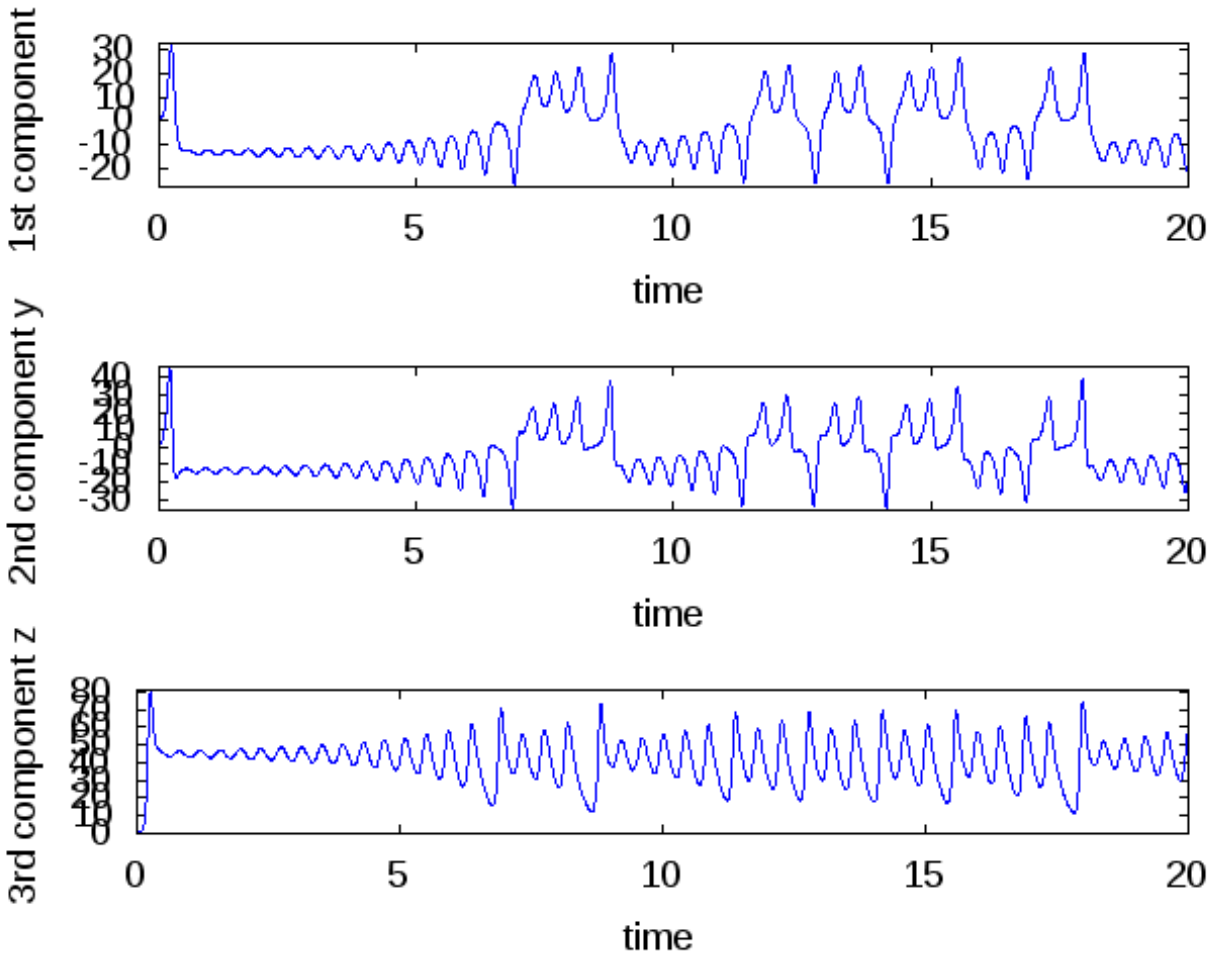
```
from scipy.integrate import odeint
from my_func import LorenzSys

T = np.arange(0,20,0.01)
X0 = [1,1,1]
```

```
X = odeint(LorenzSys,X0,T)
plt.plot(T,X)
plt.show()
```

The components can also be visualized individually

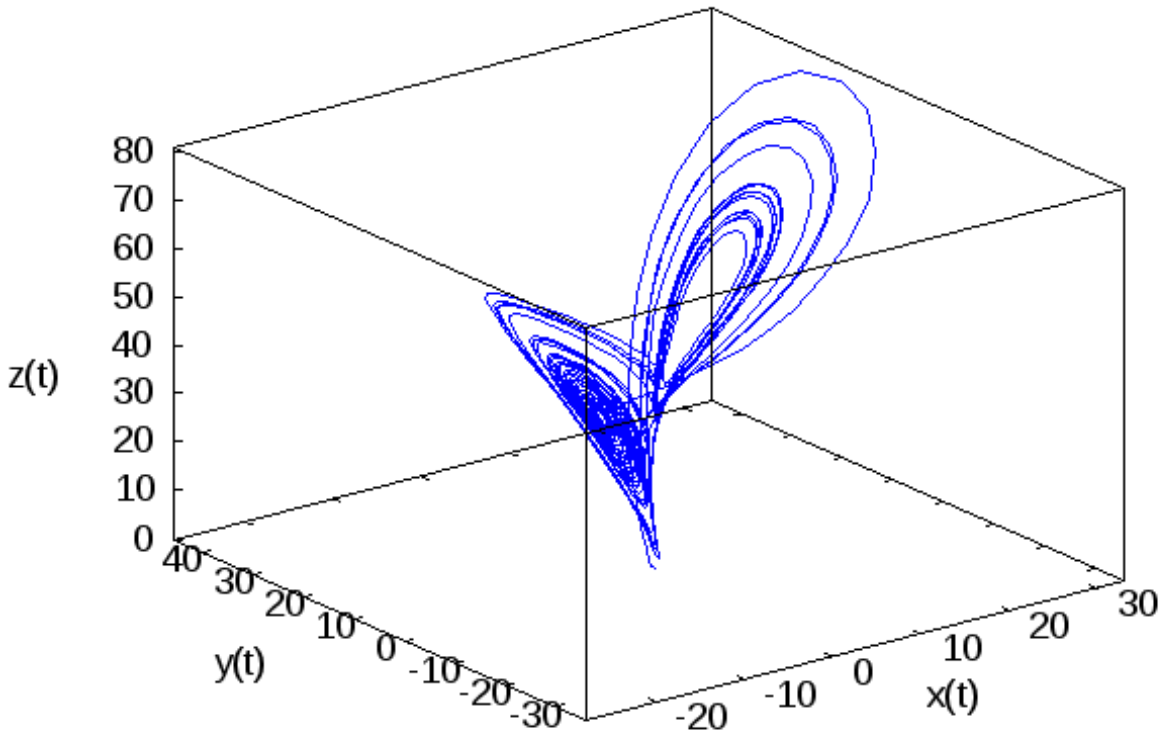
```
plt.subplot(311); plt.plot(T,X[:,0])
plt.xlabel('time'); plt.ylabel('1st component x')
plt.subplot(312); plt.plot(T,X[:,1])
plt.xlabel('time'); plt.ylabel('2nd component y')
plt.subplot(313); plt.plot(T,X[:,2])
plt.xlabel('time'); plt.ylabel('3rd component z')

plt.savefig('lorenz_plot01.png')

plt.show()
```

Note that `X` is a matrix with three columns. `X[:,0]` denotes the first column.



We can also use a three-dimensional plot.

```
from mpl_toolkits.mplot3d import Axes3D

ax = plt.gca(projection='3d')

ax.plot(X[:,0],X[:,1],X[:,2], 'b-')
ax.set_xlabel('x(t)')
ax.set_ylabel('y(t)')
ax.set_zlabel('z(t)')

plt.show()
```



The Lotka-Volterra System

We want to implement the two-dimensional Lotka-Volterra System. A suitable function is

```
# implements the Lotka-Volterra System
# x is a two-dimensional vector
# usage T = np.arange(0,50,0.1)
# x = odeint(LotkaVolterra,[1,1],T)


def LotkaVolterra(x,t):

    # define parameters
    X = x[0]
    Y = x[1]

    a = 1
    b = 2
    g = 1.5
    d = 2

    dxdt = a*X - b*X*Y
    dydt = g*X*Y - d*Y

    # the function returns the vector fx
    df = [dxdt, dydt]

    return df
```

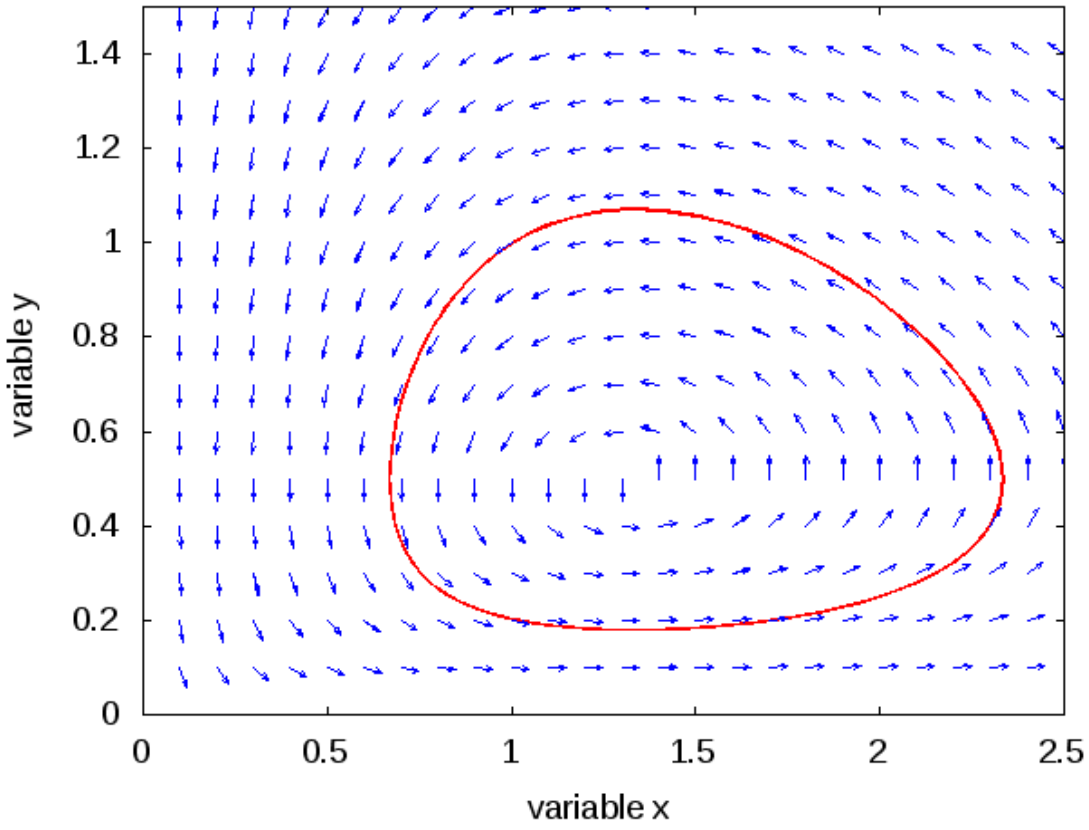
The isoclines can be plotted as follows

```
# parameters (use the same as in the function file)
a = 1
b = 2
g = 1.5
d = 2

plt.plot(x[:,0],x[:,1])
plt.plot([d/g, d/g],[0, 3*a/b],'r-')
plt.plot([0, 2*d/g],[a/b, a/b],'r-')
plt.show()
```

- **Exercise:** Compare the solutions with the isocline for different parameter values.

We can also have a more detailed look on the phase plane. A suitable method are quiver plots (`help(plt.quiver)`) .

```
x = np.arange(0.1,3,0.1); y = np.arange(0.1,3,0.1)
[xg,yg] = np.meshgrid(x,y)
n = np.size(x); m = np.size(x)
u = np.zeros([n,m])
v = np.zeros([n,m])

for i in range(n):
    for j in range(m):
        df = LotkaVolterra([xg[i,j],yg[i,j]],1)
        df = df/np.linalg.norm(df)
        u[i,j] = df[0]
        v[i,j] = df[1]

h = plt.quiver(xg,yg,u,v,0.5)
plt.show()
```



Models of Metabolism

A simple metabolic network

Our next step are simple metabolic networks. The basic ingredients for model construction are the stoichiometry, the rate equations, and the parameters. We start with a simple linear chain,

$$X \longrightarrow S_1 \longrightarrow S_2 \longrightarrow S_3 \longrightarrow Y \tag{15}$$

and irreversible Michaelis-Menten equations

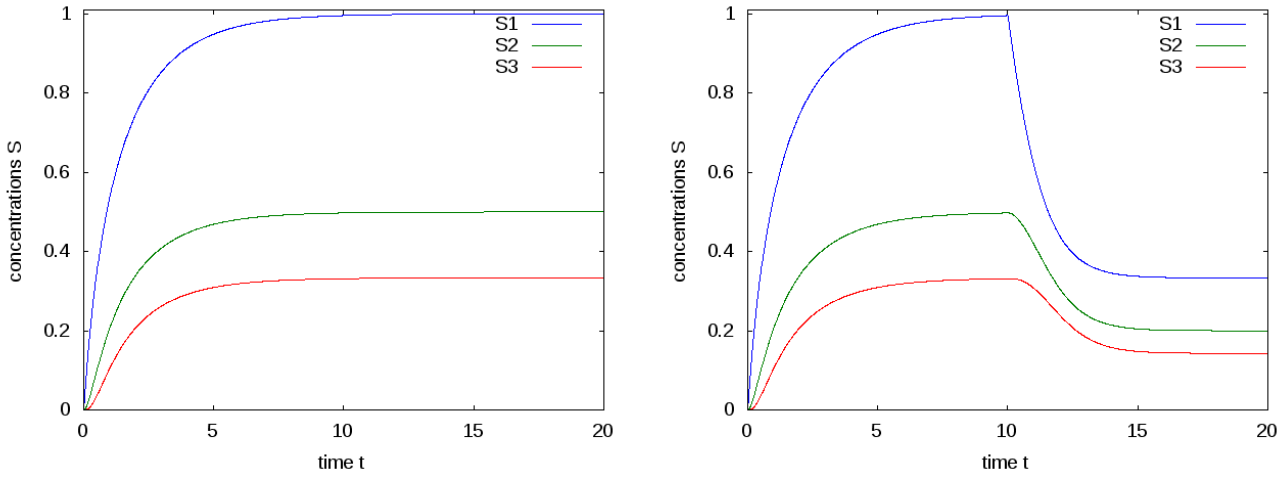
$$v_i = \frac{V_{\max} S_i}{K_M + S_i} \tag{16}$$

The external metabolites X and Y are like (constant) parameters. The concentration of X is included into the (constant) influx value.
The octave function might look like this:

```
# function: LinearChain(S,t)
# implements the simple linear chain with irreversible MM kinetics

def LinearChain(S,t):

    # define metabolites
    S1 = S[0]
    S2 = S[1]
    S3 = S[2]

    # define parameters
    influx = 1.0
    Km1 = 1.0; Vm1 = 2.0
    Km2 = 1.0; Vm2 = 3.0
    Km3 = 1.0; Vm3 = 4.0

    # define rate functions
    v0 = influx # constant
    v1 = Vm1*S1/(Km1+S1)
    v2 = Vm2*S2/(Km2+S2)
    v3 = Vm3*S3/(Km3+S3)


    # define stoichiometry
    dS1dt = +v0 - v1
    dS2dt = +v1 - v2
    dS3dt = +v2 - v3

    return [dS1dt, dS2dt, dS3dt]
```

To integrate the pathway, we use `odeint`:

```
T = np.arange(0,20,0.01)
S0 = [0, 0, 0]

S = odeint(LinearChain,S0,T)
plt.plot(T,S)
plt.legend(['S1','S2','S3'])
plt.xlabel('time t')
plt.ylabel('concentrations S')
plt.show()
```



- **Exercise:** Add a single line of code into the parameter definition of the function that changes the influx after a time T = 10.

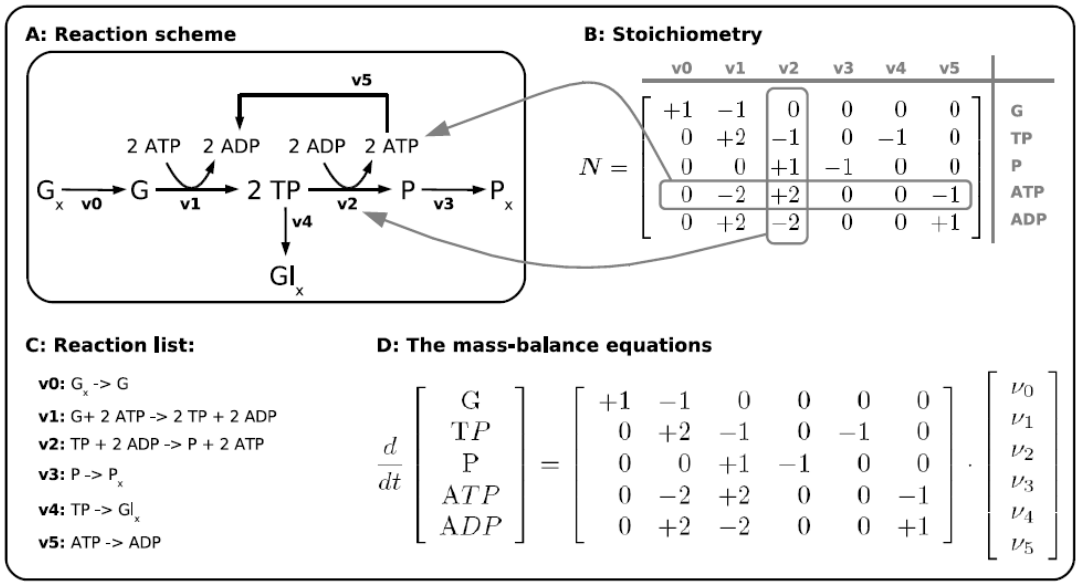
```
if (t > 10): influx = 0.5
```

The Michaelis-Menten Equation

- **Exercise:** Compare the solutions for the overall Michaelis-Menten equation with the numerical solution obtained from simulation the elementary binding reactions. Use different sets of parameters and check for each set of parameter if the requirements are fulfilled. Use also high enzyme concentrations such that $[\mathrm{E}]^T \approx [\mathrm{S}]$

A generic metabolic model: Glycolysis

We now can implement a typical example of a metabolic system.



The first step is to assemble the list of reactions

```
v0: Gx <-> G
v1: G + 2 ATP -> 2 TP + 2 ADP
v2: TP + 2 ADP -> 2 ATP + P
v3: P ->
v4: TP ->
v5: ATP -> ADP
```

We can construct the stoichiometric matrix

```
N = np.array([[+1, -1, 0, 0, 0, 0],
              [0, +2, -1, 0, -1, 0],
              [0, 0, +1, -1, 0, 0],
              [0, -2, +2, 0, 0, -1],
              [0, 2, -2, 0, 0, +1]])

r = np.rank(N)
```

- **Exercise:** What does the function `rank` calculate. What does it mean for the network?

We want to implement the pathway in python

```
# function: SimpleGlycolysis(S,t)
# defines a very minimal model of glycolysis

def SimpleGlycolysis(S,t):
```

```
    # define metabolites
    G  = S[0]
    TP  = S[1]
    P  = S[2]
    ATP = S[3]

    # define parameter
    A_total = 2.0
    ADP = A_total - ATP
    Gx = 1.0

    k0 = 5.0
    k1 = 1.0
    k2 = 1.0
    k3 = 1.0
    k4 = 1.0
    Vm5 = 1.0
    Km5 = 1.0

    # define rate equations
    v0 = k0*(Gx - G)
    v1 = k1*G*ATP
    v2 = k2*TP*ADP
    v3 = k3*P
    v4 = k4*TP
    v5 = Vm5*ATP/(Km5+ATP)

    # define equations
    dGdt = v0 - v1
    dTPdt = + 2*v1 - v2 - v4
    dPdt = + v2 - v3
    dATPdt = -2*v1 + 2*v2 - v5

    return [dGdt, dTPdt, dPdt, dATPdt]
```

The simulation is well-known by now

```
from my_func import SimpleGlycolysis

T = np.arange(0,50,0.1)
S0 = [1, 1, 1, 1]

S = odeint(SimpleGlycolysis,S0,T)
plt.plot(T,S)

plt.xlabel('time t')
plt.ylabel('concentrations S')
plt.legend(['G','TP','P','ATP'])
plt.show()
```

Maybe we can chose different initial conditions, what about

```
T = np.arange(0,50,0.1)
S0 = [1, 1, 1, 1.8]
S = odeint(SimpleGlycolysis,S0,T)
plt.plot(T,S)
plt.show()

S0 = [1, 1, 1, 1.5]
S = odeint(SimpleGlycolysis,S0,T)
plt.plot(T,S)
plt.show()
```

- **Exercise:** Replace the parameters by

  ```
  # define parameter
  A_total = 2.0
  ADP = A_total - ATP
  Gx = 1.0

  k0 = 2.0
  k1 = 2.0
  k2 = 1.5
  k3 = 3.0
  k4 = 0.5
  Vm5 = 2.0
  Km5 = 1.0
  ```

- **Exercise:** Add an increased usage of ATP after a certain time

  ```
  if (t>30): Vm5 = 2.5
  ```

Our next step is to introduce allosteric inhibition into the first reaction. We write

```
kATP = 2.0
KI = 1.0
n = 4

fATP = kATP/(1+(ATP/KI)**n)
v1 = fATP*k1*G*ATP
```

- **Exercise:** Also try the parameters `kATP = 5.0; KI = 0.1`.

Going back to the model without allosteric regulation, we can investigate the behavior of the model for different external concentrations G_x. To this end, we use a global variable that defines the concentration within the function file[1].
We can now start simulations for different values of G_x.

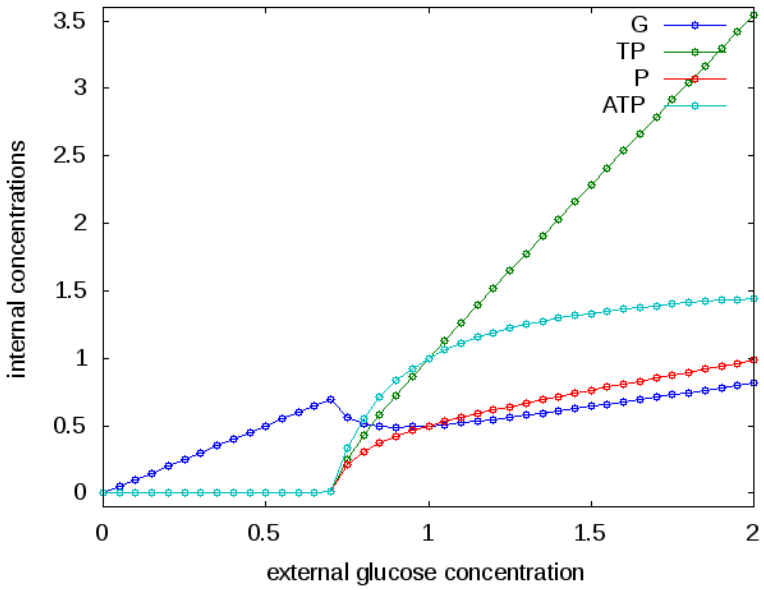
[1]To use global variables is very sloppy programming and in general not recommended. We will learn a better method in the next section

```
from my_func import SimpleGlycolysis
import my_func as mf

T = np.arange(0,50,0.1)
S0 = np.random.rand(4)

mf.Gx = 0.5

S = odeint(SimpleGlycolysis,S0,T)
plt.plot(T,S)
plt.xlabel('time t'); plt.ylabel('concentrations')
plt.show()
```

- **Exercise:** Repeat the analysis for different values of `Gx = 0.7; Gx = 0.75; Gx = 1.0`.

We can scan an entire region of G_x using a `for` loop.

```
S_final = np.zeros([len(Gx_range),4])
Gx_range = np.arange(0,2,0.05)

for i in range(len(Gx_range)):
    mf.Gx = Gx_range[i]
    S = odeint(SimpleGlycolysisEXT,S0,T)
    S_final[i,:] = S[-1,:]

plt.plot(Gx_range,S_final,'o-')
plt.xlabel('external glucose concentration')
plt.ylabel('internal concentrations')
plt.show()
```



For completeness, the entire script is reprinted below.

```
# function: SimpleGlycolysisEXT(S,t)
# defines a very minimal model of glycolysis


def SimpleGlycolysisEXT(S,t):

    # define metabolites
    G  = S[0]
    TP  = S[1]
    P  = S[2]
    ATP = S[3]

    # define parameter
    A_total = 2.0
    ADP = A_total - ATP

    global Gx

    k0 = 2.0
    k1 = 2.0
    k2 = 1.5
    k3 = 3.0
    k4 = 0.5
    Vm5 = 2.0
    Km5 = 1.0

#   if (t>30): Vm5 = 2.5

    # define rate equations
    v0 = k0*(Gx - G)
    v1 = k1*G*ATP
    v2 = k2*TP*ADP
    v3 = k3*P
    v4 = k4*TP
    v5 = Vm5*ATP/(Km5+ATP)

    # define equations
    dGdt = v0 - v1
    dTPdt = + 2*v1 - v2 - v4
    dPdt = + v2 - v3
    dATPdt = -2*v1 + 2*v2 - v5

    return [dGdt, dTPdt, dPdt, dATPdt]
```

- **Exercise:** Try the same scan for the pathway including feedback. Also look for oscillations.

Yeast Glycolysis: The Wolf Model

- **Note:** To use global variables is very sloppy programming and in general not recommended. Here is a better method using a local function. To do so, we define our ODE function to take external parameters.

```
def odesystem(x,t,paras):
    ...
    return dxdt
```

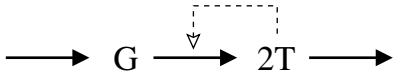
Additional parameters have to be passed to `args=(..)` within the `odeint` function in the same order as it is declared in your own function definition.

```
para = B
S = odeint(odesystem,S0,T,args=(para))
```

- **Exercise:** Implement the model of Wolf et al. (2000) and reproduce some of the figures given in the original paper.

Yeast Glycolysis: The Bier model

Yeast glycolysis is a prototypical example of an oscillatory metabolic systems. Oscillations in glycolysis are observed for more than thirty years, however their function, if any, is unknown. We look at a simple minimal model published by Bier et al. (2000). The model is highly simplified and considers only two variables, G as the internal concentration of glucose and T the concentration of ATP.



Put simple: Each unit of glucose makes two units of ATP, whereby ATP activates its own production. The resulting ODEs are

$$\dot{G} = V_{in} - k_1 G T \tag{17}$$

$$\dot{T} = 2k_1 G T - k_p \frac{T}{K_M + T} \tag{18}$$

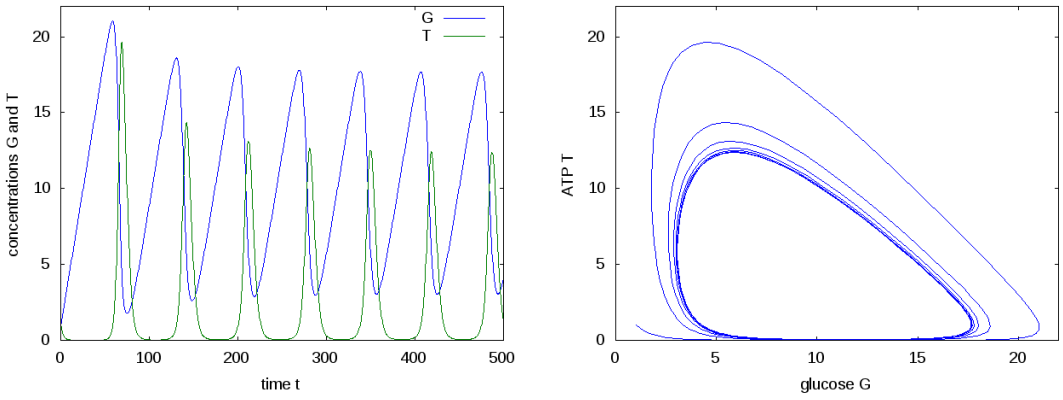
And the parameter are $V_{in} = 0.36$, $k_1 = 0.02$, $k_p = 6.0$ and $K_M = 13.0$ (after Bier et al., 2000). Again, we have to implement an appropriate function

```
# function dSdt = BierModel(S,t)
# defines the model of Bier et al. (2000)

def BierModel(S,t):

    # define metabolites
    G = S[0]
```

```
    T = S[1]

    # define parameters
    Vin = 0.36
    k1 = 0.02
    kp = 6.0
    Km = 13.0

    # define rates
    v1 = Vin
    v2 = k1*G*T
    v3 = kp*T/(Km+T)

    dGdt = v1 - v2
    dTdt = 2*v2 - v3

    return [dGdt, dTdt]

% end of function
```

The numerical integration runs as usual

```
T = np.arange(0,500,0.5)
S0 = [1, 1]

S = odeint(BierModel,S0,T)
plt.plot(T,S)
plt.legend(['G','T'])
plt.xlabel('time t'); plt.ylabel('concentrations S')
plt.show()
```



We can also make a phase plot.

```
plt.plot(S[:,0],S[:,1])
plt.xlabel('glucose G')
```

```
plt.ylabel('ATP T')
plt.show()
```

Coupled oscillations

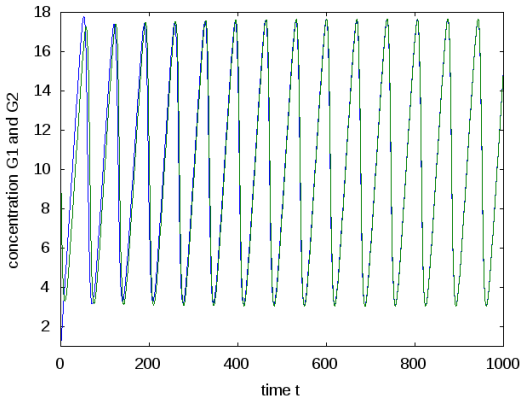
In experiments, we typically do not see single cells, but the average of a population. We therefore investigate coupling between cells.
To look at coupled cells, we implement a new python function

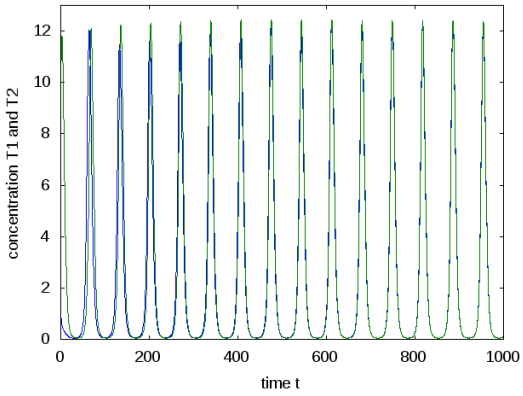
```
function [dSdt] = TwoBier(S,t);
# function: TwoBier(S,t)
# defines the mode of Bier et al. (2000)
# for two coupled cells


def TwoBier(S,t):

    # define metabolites
    G1 = S[0]
    T1 = S[1]
    G2 = S[2]
    T2 = S[3]

    # define parameter
    Vin = 0.36
    k1 = 0.02
    kp = 6.0
    Km = 13.0

    epsi = 0.01

    # define equations
    dG1dt = Vin - k1*G1*T1
    dT1dt = 2*k1*G1*T1 - kp*T1/(Km+T1) + epsi*(T2-T1)
    dG2dt = Vin - k1*G2*T2
    dT2dt = 2*k1*G2*T2 - kp*T2/(Km+T2) - epsi*(T2-T1)

    return [dG1dt, dT1dt, dG2dt, dT2dt]
```

The integration runs as usual

```
T = np.arange(0,1000,0.5)
S0 = [1, 1, 10, 10]

S = odeint(TwoBier,S0,T)
plt.plot(T,S)
plt.legend(['G','T'])
```

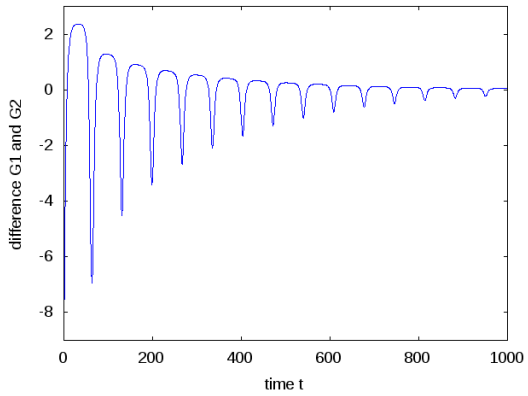
```
plt.xlabel('time t')
plt.ylabel('concentrations S')
plt.show()
```





We can also look at the difference between variables $G_1 - G_2$:

```
plt.plot(T,S[:,0]-S[:,2])
plt.show()
```



- **Exercise:** Investigate different values of the coupling constant. For example $\epsilon = 0$, $\epsilon = 0.01$ and $\epsilon = 0.05$.

Interlude: Models on the internet

Many models are provided in a ready-to-use format in databases. We may have a look at

`biomodels.org`

The model

`http://www.ebi.ac.uk/biomodels-main/BIOMD0000000254`

encodes the model from Bier et al.

A better example is the model of Hynne et al. :
Hynne F, Dano S, Sorensen PG.
Full-scale model of glycolysis in Saccharomyces cerevisiae.
Biophys. Chem. 2001 Dec; 94(1-2): 121-163

`http://www.ebi.ac.uk/biomodels-main/BIOMD0000000061`