Lec0102a

August 10, 2023

1 Simulation of a simple system

In this notebook, we will simulate a simple system. See https://parsys.lri.fr/teaching/NatAlg/ps02/ps02_python.html for another example.

Our system will be

$$\begin{aligned}\dot{X} &= U - aEX^{0.5} \\ \dot{Y} &= aEX^{0.5} - bY^{0.5} \\ \dot{Z} &= bY^{0.5} - cZ^{0.5}\end{aligned}$$

This is of the form

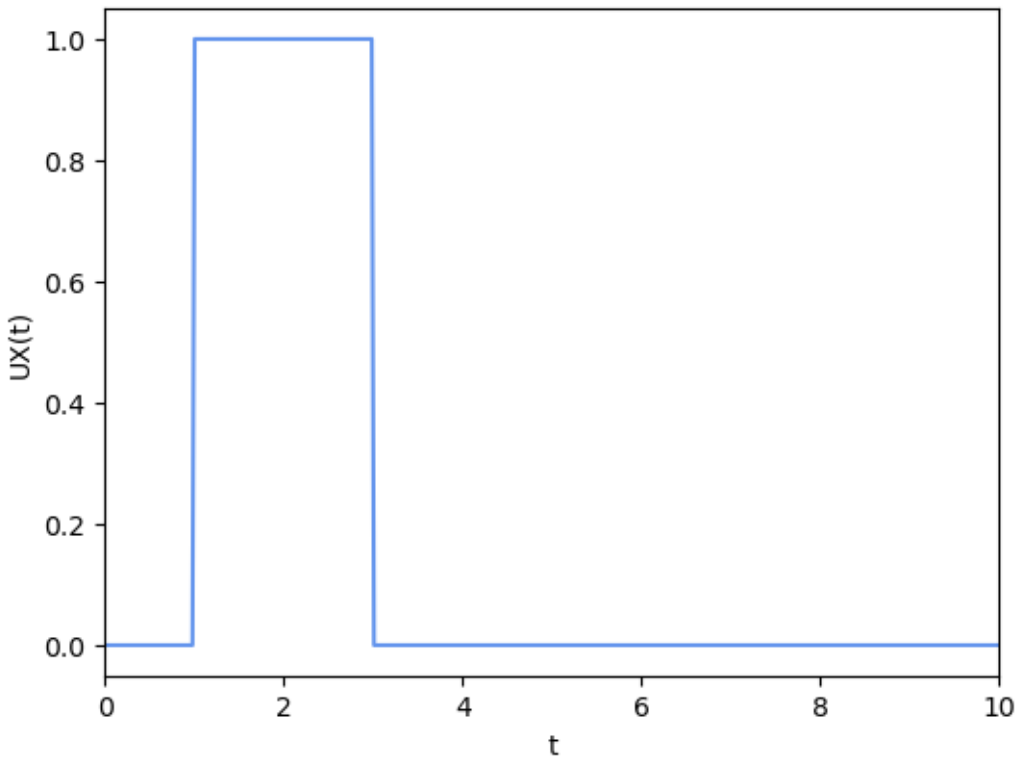
$\mathbf{S}' = \mathbf{F}(t, \mathbf{S}(t)) + \mathbf{U}(t)$

[1]:
```
import numpy as np
from scipy.integrate import odeint
import matplotlib.pyplot as plt
%matplotlib inline

figsize=(6, 4.5)
```

[2]:
```
# Number of time points we want for the solutions
n = 400
t0 = 0
tF = 10

# Time points we want for the solution
t = np.linspace(t0, tF, n)
```

[3]:
```
def U(t, t_0, tau, u_0):
    """
    Returns x value for a pulse beginning at t = 0
    and ending at t = t_0 + tau.
    """
    u=np.logical_and(t >= t_0, t <= (t_0 + tau)) * u_0
    return np.array([u,0,0])

U_args = (1.0, 2.0, 1.0)
```

```
plt.figure(figsize=figsize)
plt.xlabel("t")
plt.ylabel("UX(t)")
plt.xlim([0, 10])

UX = np.zeros(t.shape)
for i, ti in enumerate(t):
    UX[i]=U(ti, *U_args)[0]

plt.plot(t, UX, color="cornflowerblue")

plt.show()
```



[4]:
```
a=0.5
b=0.5
c=0.5
E=0.5
```

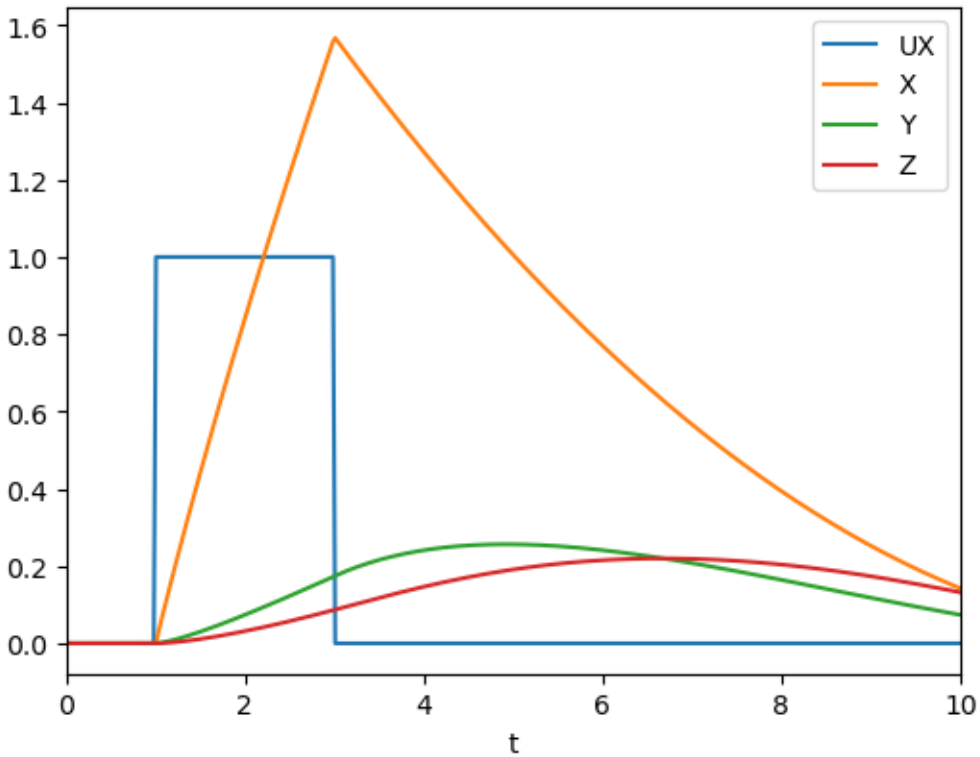
[5]:
```
def FU(S, t, a, b, c, E, U_fun, U_args):
    """
    U_fun is a function of the form U_fun(t, *U_args), so U_args is a tuple
    containing the arguments to pass to U_fun.
```

```
    """
    # Compute U
    U = U_fun(t, *U_args)

    # Compute F
    # Unpack S
    X,Y,Z = S
    FX = -a * E * np.power(X, 0.5)
    FY = a * E * np.power(X, 0.5) - b * np.power(Y, 0.5)
    FZ = b * np.power(Y, 0.5) - c*np.power(Z,0.5)

    F = np.array([FX, FY, FZ])
    return F+U
```

[6]:
```
# Initial state
S0=np.array([0,0,0])
```

[7]:
```
# Integrate ODEs
S = odeint(FU, S0, t, args=(a, b, c, E, U, U_args))
```

[8]:
```
plt.figure(figsize=figsize)
plt.xlabel("t")
plt.xlim([0, 10])

plt.plot(t, UX);
plt.plot(t, S[:,0]);
plt.plot(t, S[:,1]);
plt.plot(t, S[:,2]);
plt.legend(['UX','X','Y','Z']);
```


1.6
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
0
2
4
6
8
10
t
UX
X
Y
Z

Lec0102b

August 10, 2023

1 Difference between an explanatory and a correlative model

Let us study how a ball falls from a given height $y(0)$. A **correlative** model is of the form

$y = a - bt^2$

with $a = y(0)$ and $b = 4.905$

An **explanatory** model is of the form

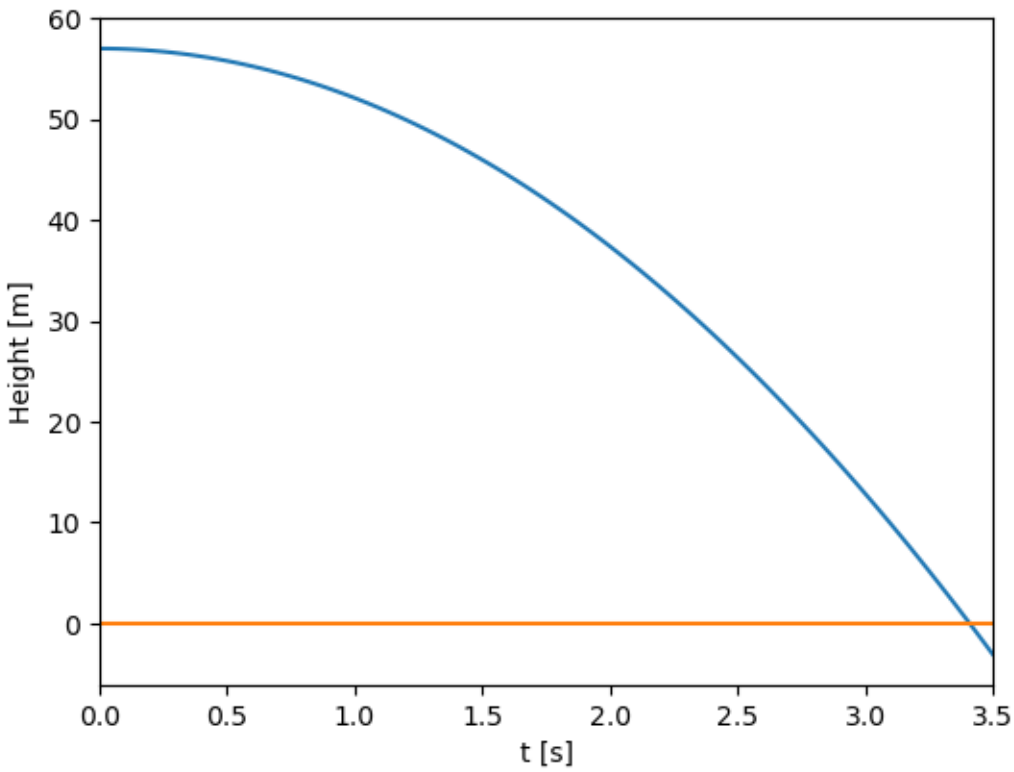
$$\begin{aligned} \dot{Y} &= V \\ \dot{V} &= -g \end{aligned}$$

1.1 Correlative model

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

figsize=(6, 4.5)
```

[2]:
```
a=57      # [m] Height of Pisa tower
b=4.905 # [m/s^2] =0.5*g=0.5*9.81
t=np.linspace(0,3.5,100)
y=a-b*np.power(t,2)

plt.figure(figsize=figsize)
plt.plot(t,y)
plt.plot(t,0*y)
plt.xlabel("t [s]")
plt.ylabel("Height [m]")
plt.xlim([0, 3.5])
```

[2]: (0.0, 3.5)

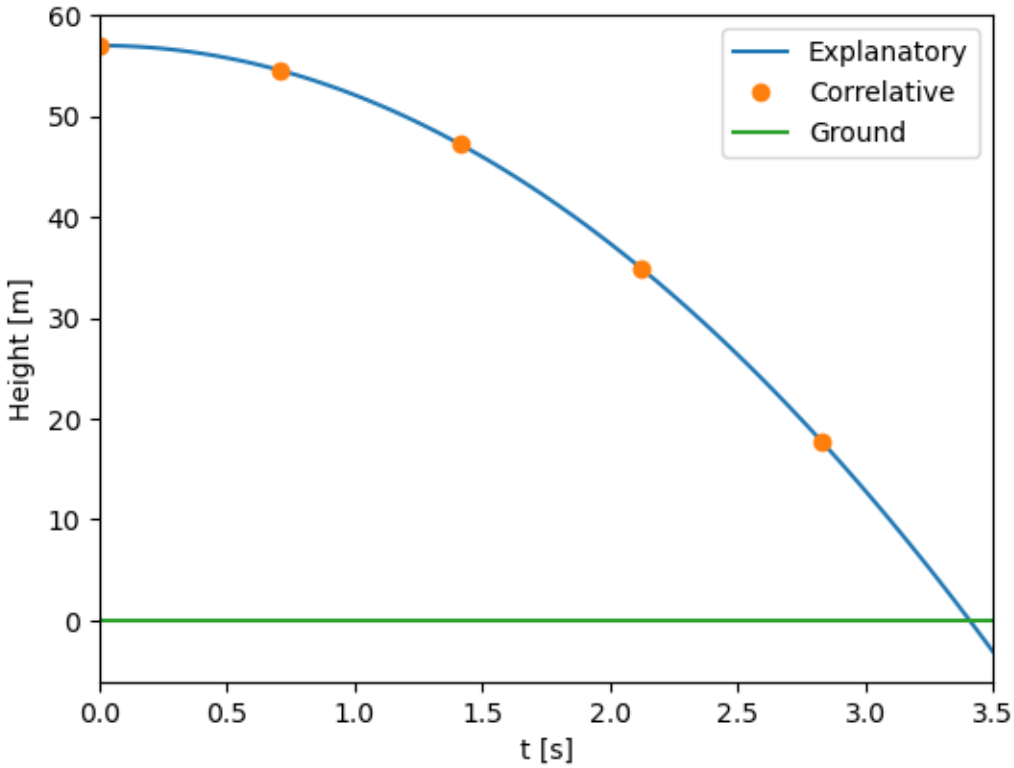


1.2 Explanatory model

[3]:
```
from scipy.integrate import odeint

def F(S, t):
    # Gravity constant in Earth
    g = 9.81

    # Compute F
    # Unpack S
    Y,V = S
    FY = V
    FV = -g
    F = np.array([FY,FV])
    return F

S = odeint(F, np.array([57, 0]), t)
y2=S[:,0]
v2=S[:,1]
```

[4]:
```
plt.figure(figsize=figsize)
plt.plot(t,y2)
plt.plot(t[0::20],y[0::20],'o')
plt.plot(t,0*y2)
plt.xlabel("t [s]")
plt.ylabel("Height [m]")
plt.xlim([0, 3.5])
plt.legend(['Explanatory','Correlative','Ground'])
```

[4]: <matplotlib.legend.Legend at 0x1d4737dec10>



Lec0102c

August 10, 2023

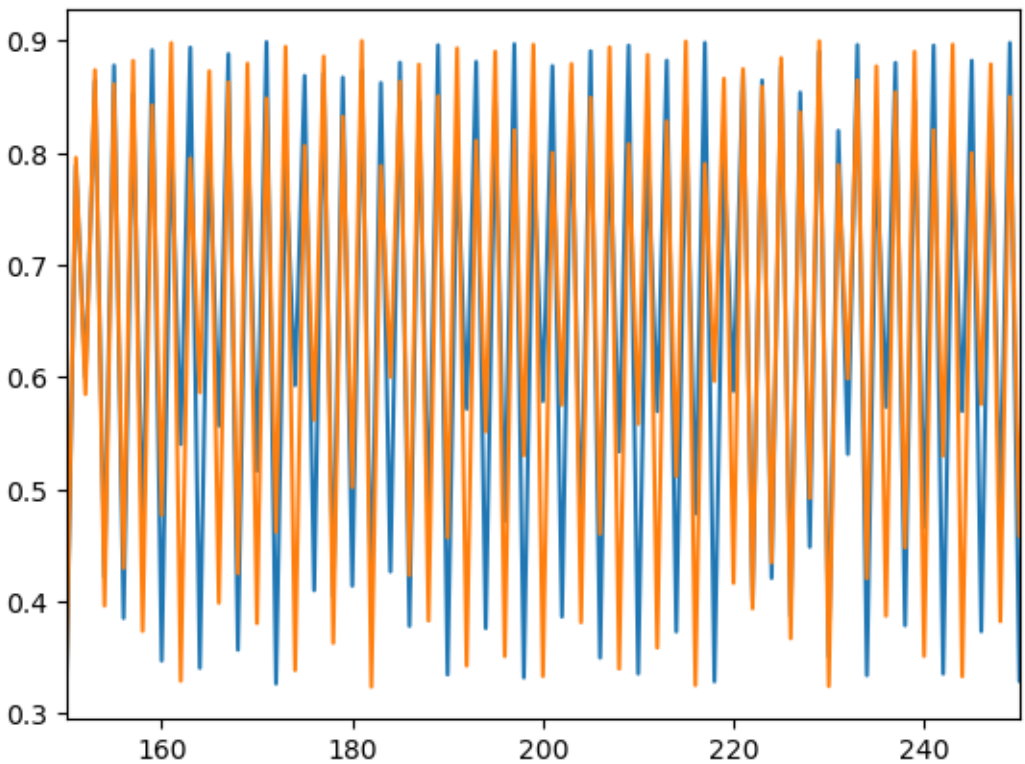
1 Logistic map

The logistic map constructs the following sequence:

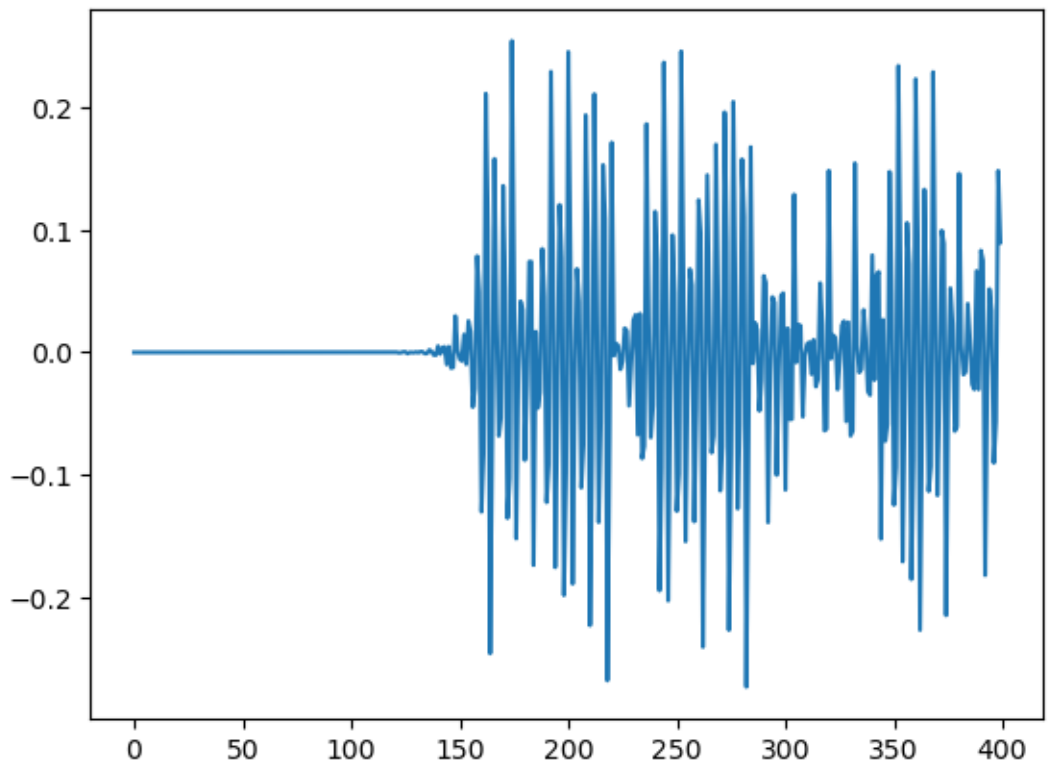
$x_n = rx_{n-1}(1 - x_{n-1})$

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
```

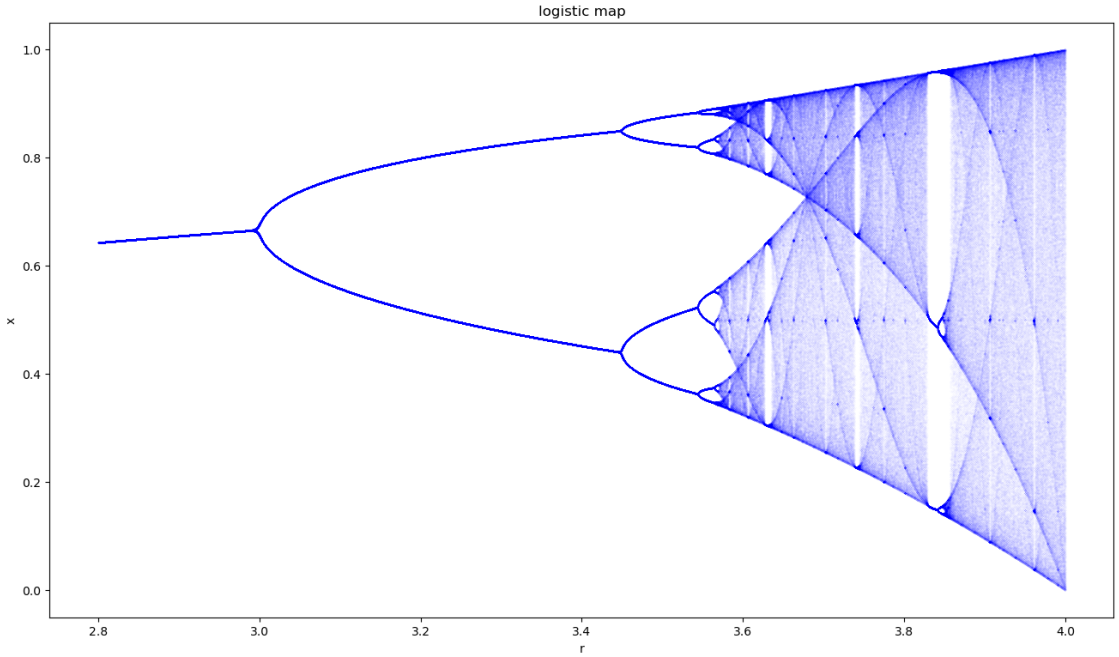
[2]:
```
def logistic(x0,r,t):
    x=np.zeros(t.shape)
    x[0]=x0
    for n in range(1,t.size):
        x[n]=r*x[n-1]*(1-x[n-1])
    return x

t=np.arange(0,400,1)
r=3.6
x1=logistic(0.500000,r,t)
x2=logistic(0.5+1e-8,r,t)

plt.plot(t,x1)
plt.plot(t,x2)
plt.xlim([150,250]);
```



[3]:
```
plt.plot(t,x1-x2);
```



[4]:
```
# The following plot shows the limit values for different r's

interval = (2.8, 4)  # start, end
accuracy = 0.0001
reps = 600  # number of repetitions
numtoplot = 200
lims = np.zeros(reps)

fig, biax = plt.subplots()
fig.set_size_inches(16, 9)

lims[0] = np.random.rand()
for r in np.arange(interval[0], interval[1], accuracy):
    for i in range(reps - 1):
        lims[i + 1] = r * lims[i] * (1 - lims[i])

    biax.plot([r] * numtoplot, lims[reps - numtoplot :], "b.", markersize=0.02)

biax.set(xlabel="r", ylabel="x", title="logistic map")
plt.show()
```


logistic map
1.0
0.8
0.6
0.4
0.2
0.0
x
2.8
3.0
3.2
3.4
3.6
3.8
4.0
r

Lec0102d

August 11, 2023

1 Deterministic vs stochastic models

Deterministic models based on differential equations are approximations in the limit of many physical processes. For instance, a chemical reaction of association/dissociation of a dimer is usually modelled with a differential equation.

$$H + H \rightarrow H^2 \quad (k_+)$$
$$H + H \leftarrow H^2 \quad (k_-)$$

$\frac{dH}{dt} = 2k_- H_2 - 2k_+ H^2$

$\frac{dH_2}{dt} = -k_- H_2 + k_+ H^2$

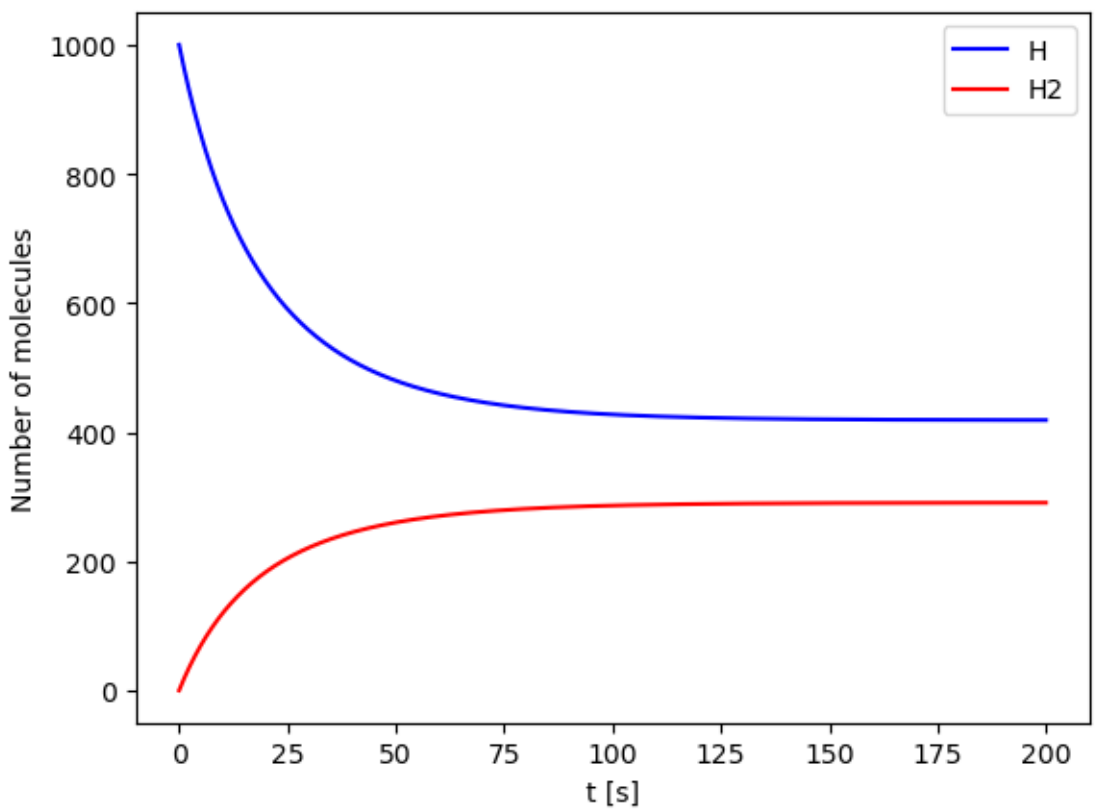
Based on https://www.biosym.uzh.ch/modules/models/Modeling_Basics/Molecular_Simulation/molsim.xhtml

1.1 Deterministic simulation

[1]:
```
from scipy.integrate import odeint
import numpy as np

# Input parameters
V = 1e-18          # Reaction volume [cm^3]=100A x 100A x 100A
H0 = 1000          # Initial number of H molecules
kminus = 0.01;     # Dissociation constant [s^-1]
kplus = 10;        # Association constant [cm^3 mol^-1 s^-1]
tF = 200           # End time

# Calculation
NA  = 6.0221415e23  # Avogadro's number
h0  = H0/(NA * V)   # Initial H concentration [mol cm^-3]
h20 = 0             # Initial H2 concentration [mol cm^-3]

def F(S, t, kminus, kplus):
    # Compute F
    # Unpack S
    H,H2 = S
    FH  = 2*kminus*H2-2*kplus*H*H
    FH2 = -kminus*H2+kplus*H*H
```

```
    F = np.array([FH,FH2])
    return F

t=np.linspace(0,tF,int(tF/10e-3)) # A sample every 10 ms
S = odeint(F, np.array([h0, h20]), t, args=(kminus, kplus))

# The differential equations are valid for the concentrations. But, we have to␣
 ↪translate into number of molecules
NH  = S[:,0]*NA*V
NH2 = S[:,1]*NA*V
```

[2]:
```
import matplotlib.pyplot as plt
%matplotlib inline

plt.plot(t,NH,'b');
plt.plot(t,NH2,'r');
plt.xlabel("t [s]");
plt.ylabel("Number of molecules");
plt.legend(['H','H2']);
```



1.2 Stochastic simulation

Stochastic simulations are based on Petri nets https://en.wikipedia.org/wiki/Petri_net and Gillespie algorithm (https://www.biosym.uzh.ch/modules/models/Modeling_Basics/Molecular_Simulation/gillespie_

1. Determine when the next reaction takes place. The **time interval** from now till the next reaction is an exponentially distributed random number. The distribution depends on the transition probabilities (=stochastic rate constants).
2. Determine **which process** happens next. A process is randomly choosen with a probability which is equal to the corresponding transition probability.
3. Repeat steps 1) and 2).

The transistion probabilities are

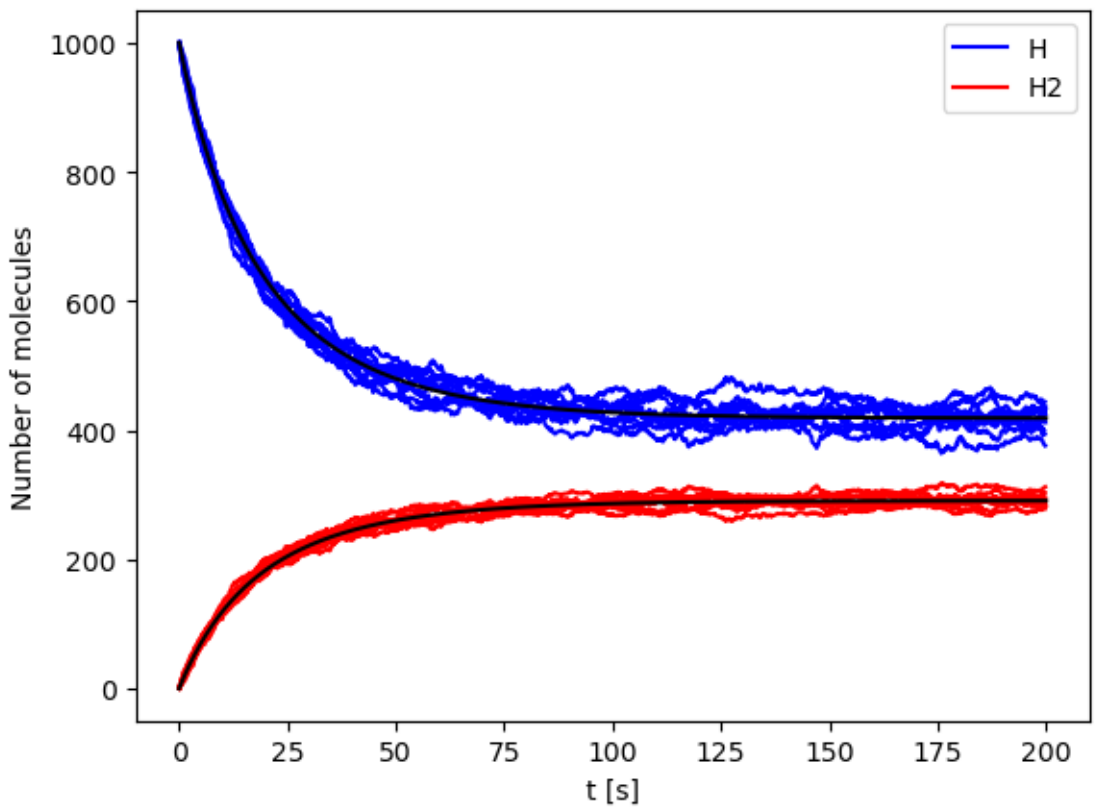
$w_- = k_- H_2$

$w_+ = k_+ H \frac{(H-1)}{N_A V}$

The time between events is distributed as $Exp\left(\frac{1}{w_- + w_+}\right)$

[3]:
```
def petrinet(H0, H20, kminus, kplus, NA, V, tF):
    m_monomer = H0;
    m_dimer = H20;
    time = 0;
    ts = [0];
    Hs = [m_monomer]
    H2s = [m_dimer];
    while True:
        # calculate transition probabilities
        wminus = kminus * m_dimer;
        wplus  = kplus  * m_monomer * (m_monomer-1)/(NA * V);

        # when does the next reaction take place?
        tau = np.random.exponential(1.0/(wplus+wminus));
        time = time + tau;
        if time > tF:
            break

        # which process happens?
        u = np.random.uniform() * (wplus + wminus);
        if u < wplus:
            # 2R --> R2
            m_monomer = m_monomer - 2;
            m_dimer = m_dimer + 1;
        else:
            # 2R <-- R2
            m_dimer = m_dimer - 1;
            m_monomer = m_monomer + 2;

        # store results
```

```
        ts.append(time)
        Hs.append(m_monomer);
        H2s.append(m_dimer);
    return (ts, Hs, H2s)

N_Petri = 10; # Number of stochastic simulations
plt.figure()
for i in range(N_Petri):
    ts, hs, h2s = petrinet(H0, 0, kminus, kplus, NA, V, tF);
    plt.plot(ts, hs, 'b')
    plt.plot(ts, h2s, 'r')
plt.plot(t,NH,'k');
plt.plot(t,NH2,'k');
plt.xlabel("t [s]");
plt.ylabel("Number of molecules");
plt.legend(['H','H2']);
```



Lec030405a

August 12, 2023

1 Simulation of the E.Coli's DNA damage detection

Our system will be

$$\begin{aligned}
\dot{X}_1 &= 10X_2^{-0.4}X_4^{0.2} - X_1^{0.5} \\
\dot{X}_2 &= 20X_1^{-0.4}X_5^{0.2} - X_2^{0.5}X_3^{0.2} \\
\dot{X}_3 &= 3X_6^{0.4} - X_3^{0.5}
\end{aligned}$$

The initial conditions are $X_1(0) = 7$, $X_2(0) = 88$, $X_3(0) = 9$, $X_4 = 10$, $X_5 = 10$.

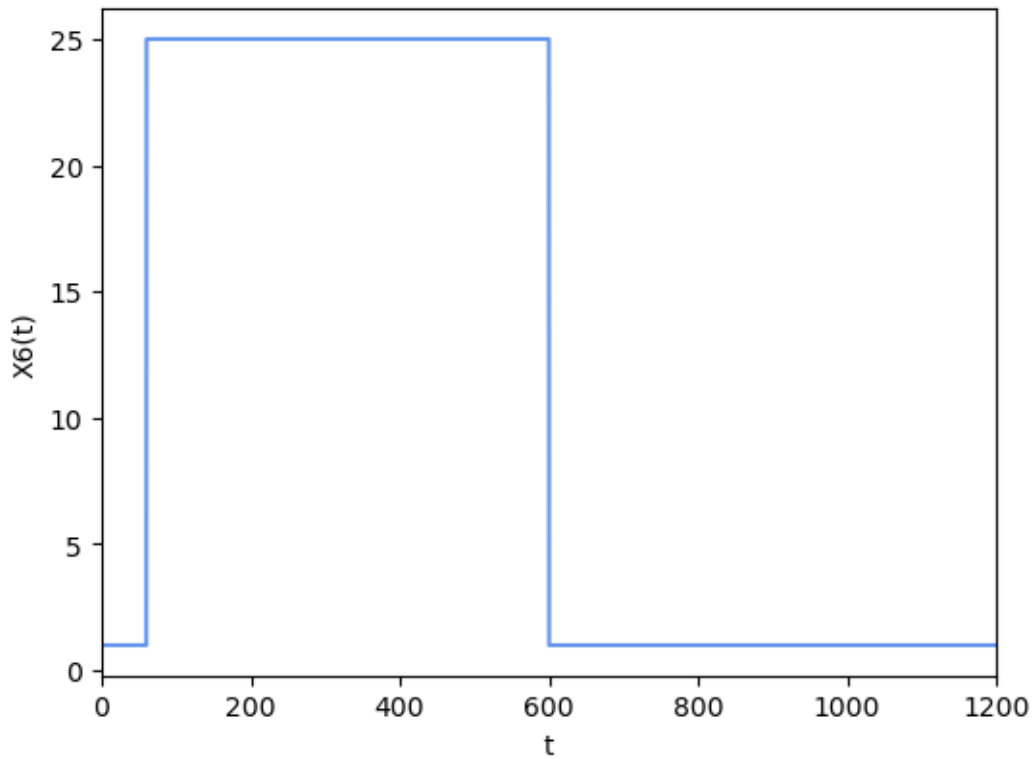
$X_6 = 1$ for the first hour showing no DNA damage. At 1h, there will be DNA damage till t=10h.

[1]:
```
import numpy as np
from scipy.integrate import odeint
import matplotlib.pyplot as plt
%matplotlib inline

figsize=(6, 4.5)
```

[2]:
```
# Number of time points we want for the solutions
n = 10000
t0 = 0
tF = 1200

# Time points we want for the solution
t = np.linspace(t0, tF, n)
```

[3]:
```
def X6Func(t, t_0, tau):
    """
    Returns x value for a pulse beginning at t = t_0
    and ending at t = t_0 + tau.
    """
    x6=1+np.logical_and(t >= t_0, t <= (t_0 + tau)) * 24
    return x6

plt.figure(figsize=figsize)
plt.xlabel("t")
plt.ylabel("X6(t)")
plt.xlim([0, 1200])
```

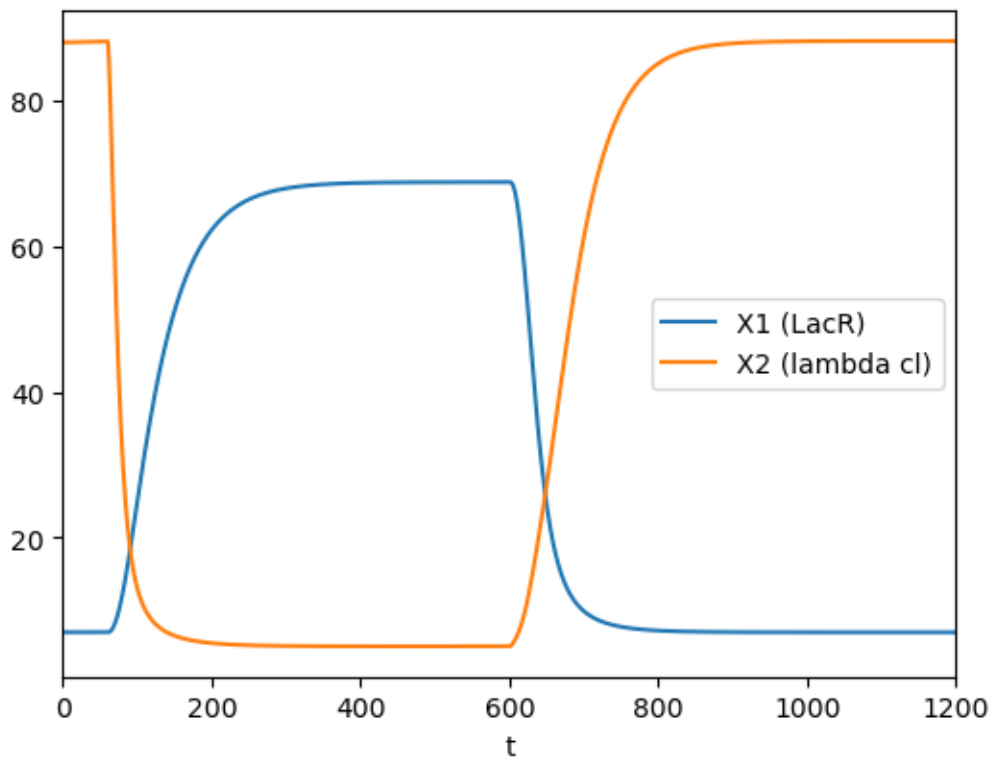
```
plt.plot(t, X6Func(t, 60.0, 540.0), color="cornflowerblue")

plt.show()
```



[4]:
```
def F(S, t):
    X4=10
    X5=10
    X6=X6Func(t, 60.0, 540.0)

    # Compute F
    # Unpack S
    X1,X2,X3 = S
    FX1 = 10*np.power(X2,-0.4)*np.power(X4,0.2)-np.power( 1,0.5)
    FX2 = 20*np.power(X1,-0.4)*np.power(X5,0.2)-np.power( 2,0.5)*np.power(X3,0.
 ↪2)
    FX3 = 3*np.power(X6,0.4)-np.power( 3,0.5)

    F = np.array([FX1, FX2, FX3])
    return F
```

[5]:
```
S = odeint(F, np.array([7,88,9]), t)
```

[6]:
```
plt.figure(figsize=figsize)
plt.xlabel("t")
plt.xlim([0, 1200])

plt.plot(t, S[:,0]);
plt.plot(t, S[:,1]);
plt.legend(['X1 (LacR)','X2 (lambda cl)']);
```



Lec030405b

August 16, 2023

1 Exploring the Lorenz System of Differential Equations

From https://github.com/jupyter-widgets/ipywidgets/blob/main/docs/source/examples/Lorenz%20Differential%

In this Notebook we explore the Lorenz system of differential equations:

$$\dot{x} = \sigma(y - x)$$
$$\dot{y} = \rho x - y - xz$$
$$\dot{z} = -\beta z + xy$$

This is one of the classic systems in non-linear differential equations. It exhibits a range of different behaviors as the parameters (\(\), \(\), \(\)) are varied.

1.1 Imports

First, we import the needed things from IPython, NumPy, Matplotlib and SciPy.

[1]:
```
# Imports for JupyterLite
%pip install -q ipywidgets matplotlib numpy scipy
```

```
Note: you may need to restart the kernel to use updated packages.
```

[2]:
```
%matplotlib inline
```

[3]:
```
from ipywidgets import interact, interactive
from IPython.display import clear_output, display, HTML
```

[4]:
```
import numpy as np
from scipy import integrate

from matplotlib import pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from matplotlib.colors import cnames
from matplotlib import animation
```

1.2 Computing the trajectories and plotting the result

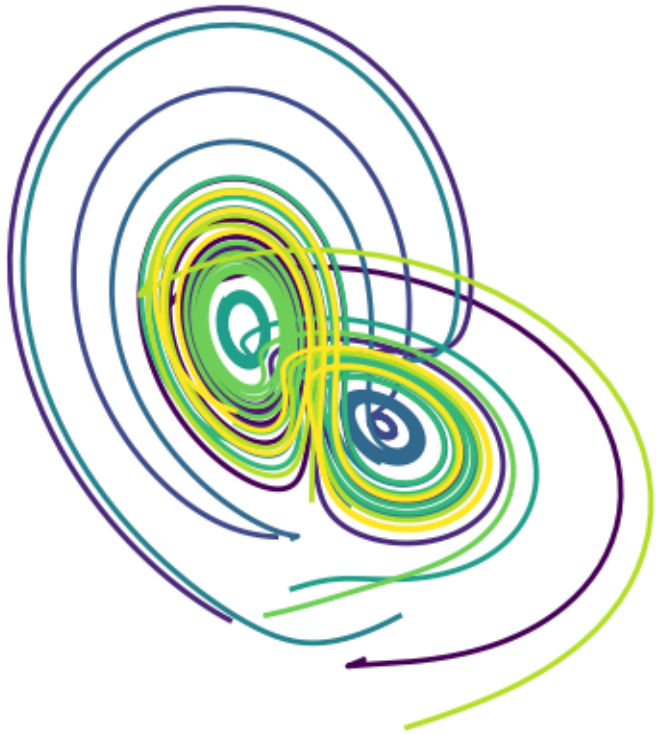
We define a function that can integrate the differential equations numerically and then plot the solutions. This function has arguments that control the parameters of the differential equation

(\(\), \(\), \(\)), the numerical integration (`N`, `max_time`) and the visualization (`angle`).

[5]:
```
def solve_lorenz(N=10, angle=0.0, max_time=4.0, sigma=10.0, beta=8./3, rho=28.
↪0):

    fig = plt.figure()
    ax = fig.add_axes([0, 0, 1, 1], projection='3d')
    ax.axis('off')

    # prepare the axes limits
    ax.set_xlim((-25, 25))
    ax.set_ylim((-35, 35))
    ax.set_zlim((5, 55))

    def lorenz_deriv(x_y_z, t0, sigma=sigma, beta=beta, rho=rho):
        """Compute the time-derivative of a Lorenz system."""
        x, y, z = x_y_z
        return [sigma * (y - x), x * (rho - z) - y, x * y - beta * z]

    # Choose random starting points, uniformly distributed from -15 to 15
    np.random.seed(1)
    x0 = -15 + 30 * np.random.random((N, 3))

    # Solve for the trajectories
    t = np.linspace(0, max_time, int(250*max_time))
    x_t = np.asarray([integrate.odeint(lorenz_deriv, x0i, t)
                      for x0i in x0])

    # choose a different color for each trajectory
    colors = plt.cm.viridis(np.linspace(0, 1, N))

    for i in range(N):
        x, y, z = x_t[i,:,:].T
        lines = ax.plot(x, y, z, '-', c=colors[i])
        plt.setp(lines, linewidth=2)

    ax.view_init(30, angle)
    plt.show()

    return t, x_t
```

Let's call the function once to view the solutions. For this set of parameters, we see the trajectories swirling around two points, called attractors.

[6]:
```
t, x_t = solve_lorenz(angle=0, N=10)
```



Using IPython's `interactive` function, we can explore how the trajectories behave as we change the various parameters.

[14]:
```
w = interactive(solve_lorenz, angle=(0.,360.), max_time=(0.1, 4.0),
                N=(0,50), sigma=(0.0,50.0), rho=(0.0,50.0))
display(w)
```

```
interactive(children=(IntSlider(value=10, description='N', max=50),␣
↪FloatSlider(value=0.0, description='angle'…
```

The object returned by `interactive` is a `Widget` object and it has attributes that contain the current result and arguments:

[15]:
```
t, x_t = w.result
```

[16]:
```
w.kwargs
```

[16]:
```
{'N': 10,
 'angle': 0.0,
 'max_time': 4.0,
 'sigma': 10.0,
 'beta': 2.6666666666666665,
 'rho': 28.0}
```

After interacting with the system, we can take the result and perform further computations. In this case, we compute the average positions in \(x\), \(y\) and \(z\).

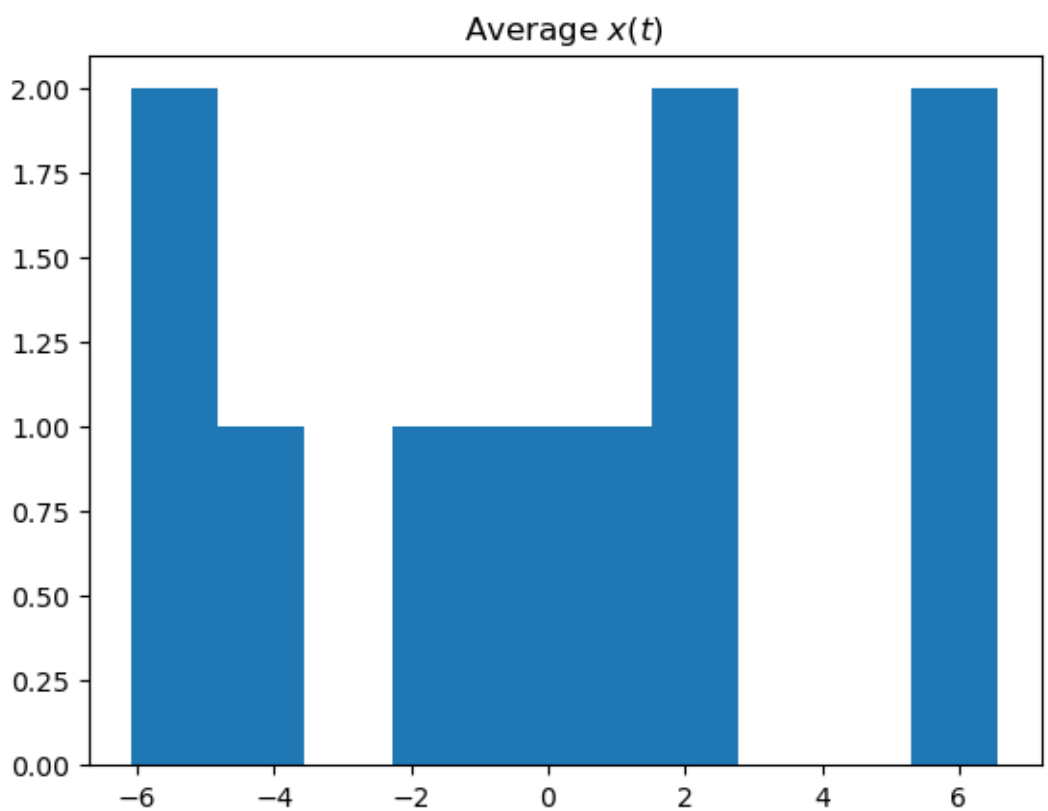
[17]:
```
xyz_avg = x_t.mean(axis=1)
```

[18]:
```
xyz_avg.shape
```

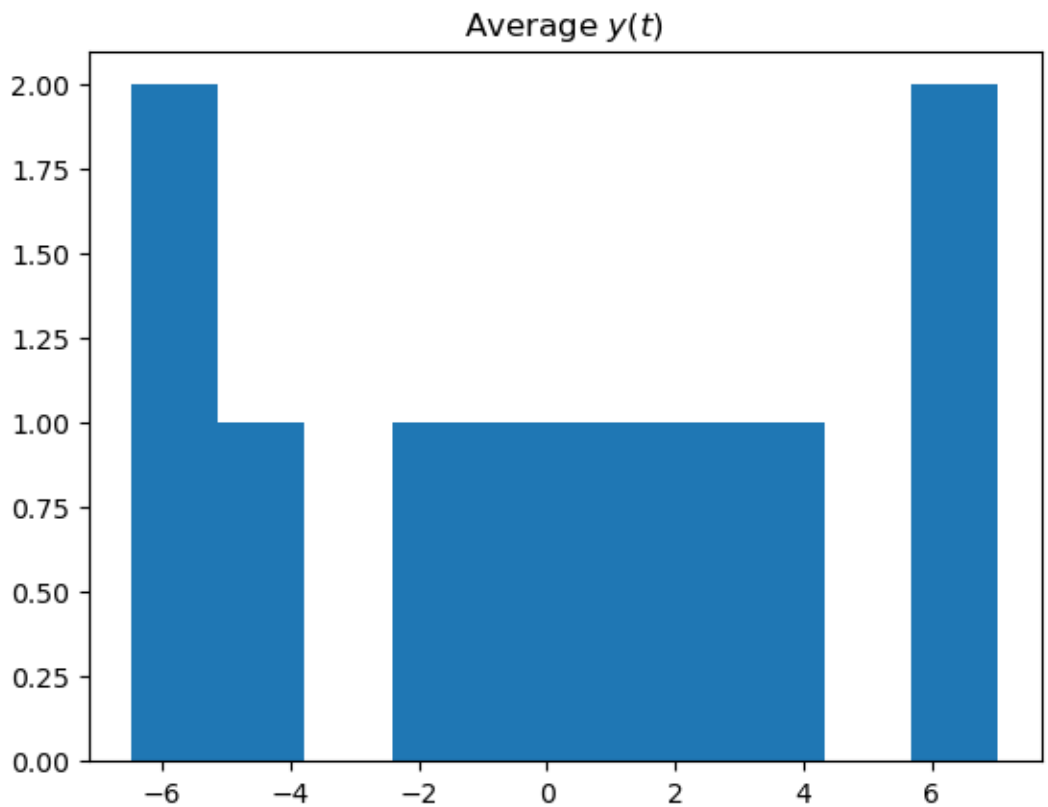
[18]:
```
(10, 3)
```

Creating histograms of the average positions (across different trajectories) show that on average the trajectories swirl about the attractors.

[19]:
```
plt.hist(xyz_avg[:,0])
plt.title('Average $x(t)$');
```



[20]:
```
plt.hist(xyz_avg[:,1])
plt.title('Average $y(t)$');
```



[ ]:
```

```

Lec030405c

August 16, 2023

1 Example of linear regression

Let us assume that bacterial growth follows the following linear model:

$\log(N(t)/N_0) = (\beta_0 + \beta_1 T)t$

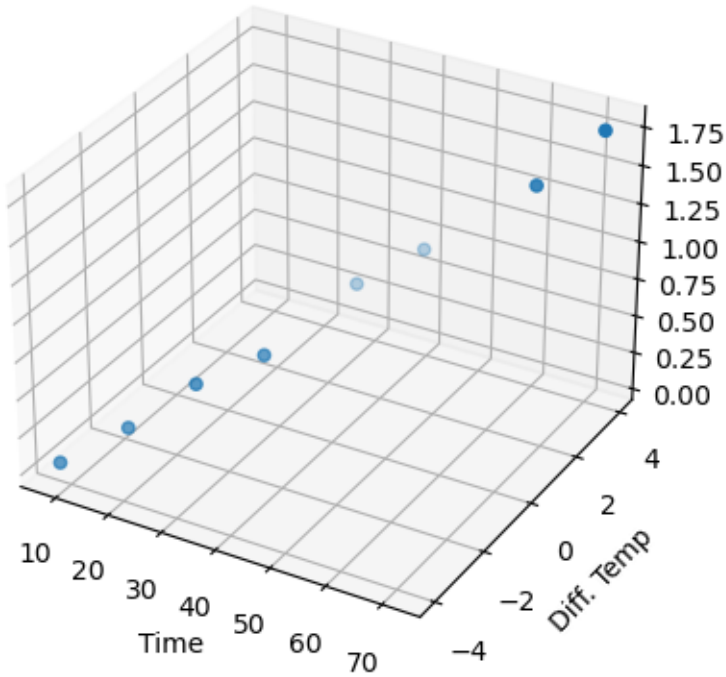
We will simulate the data with $\beta_0 = 0.02$ and $\beta_1 = 0.001$.

[1]:
```
# Data simulation
import numpy as np
t=np.array([ 8, 16, 24, 32, 40, 48, 64, 72])
T=np.array([-4, -3, -2, -1,  1,  2,  3,  4])
beta0=0.02
beta1=0.001

y=np.multiply(beta0+beta1*T,t)
print(y)
```

```
[0.128 0.272 0.432 0.608 0.84  1.056 1.472 1.728]
```

[2]:
```
# Observed data
yobs = np.array([0.05, 0.22, 0.45, 0.58, 0.85, 1.02, 1.46, 1.76])
```

[3]:
```
import matplotlib.pyplot as plt
# for creating a responsive plot
#%matplotlib widget

fig = plt.figure()
ax = fig.add_subplot(projection='3d')
ax.scatter(t, T, yobs, marker='o')

ax.set_xlabel('Time')
ax.set_ylabel('Diff. Temp')
ax.set_zlabel('')

plt.show()
```

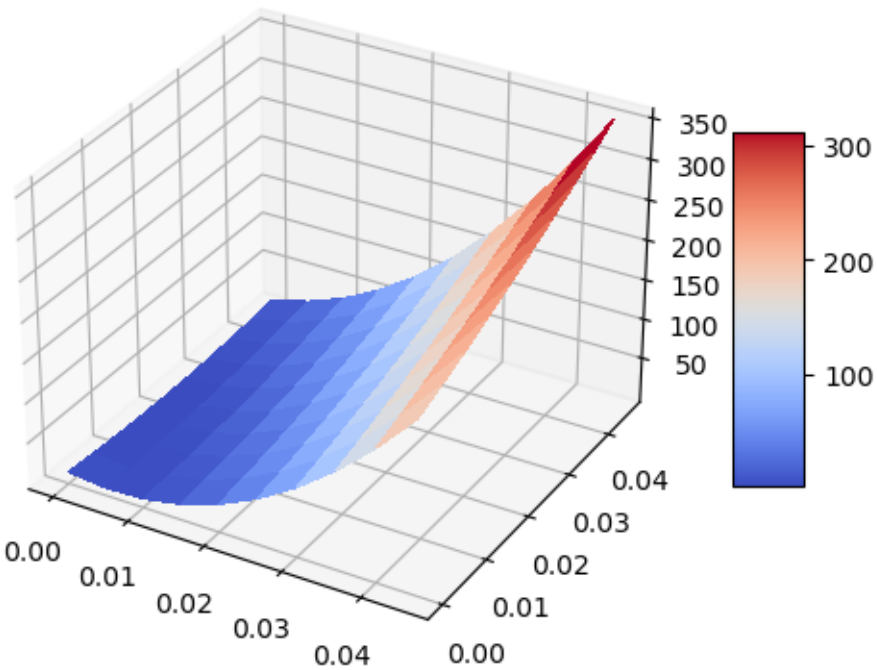


1.1 Exhaustive search

We will perform an exhaustive search in a grid

[4]:
```
b0grid=np.arange(0,0.05,0.005)
b1grid=np.arange(0,0.05,0.005)

def f(b0,b1,t,T):
    return np.multiply(b0+b1*T,t)

MSE=np.zeros((b0grid.size, b1grid.size))
for i in range(b0grid.size):
    for j in range(b1grid.size):
        ypredict = f(b0grid[i], b1grid[j], t, T)
        E = y-ypredict
        MSE[i,j] = np.sum(np.power(E,2))

from matplotlib import cm
fig, ax = plt.subplots(subplot_kw={"projection": "3d"})
X, Y = np.meshgrid(b0grid, b1grid)
surf = ax.plot_surface(X, Y, MSE, cmap=cm.coolwarm,
                       linewidth=0, antialiased=False)
fig.colorbar(surf, shrink=0.5, aspect=5)
```

```
plt.show()
```



[5]:
```
imin, jmin=np.where(MSE==MSE.min())
beta0min=b0grid[imin[0]]
beta1min=b1grid[jmin[0]]
print("Beta0min=%f"%beta0min)
print("Beta1min=%f"%beta1min)
```

```
Beta0min=0.020000
Beta1min=0.000000
```

1.2 Gradient descent

The MSE with respect to the model is

$MSE = \sum_i (y_i - (t_i\beta_0 + \Delta T_i t_i \beta_1))^2$

The derivative of the MSE is

$\frac{\partial MSE}{\partial \beta_0} = \sum_i \frac{\partial E}{\partial e_i}\frac{\partial e_i}{\partial \beta_0} = \sum_i 2e_i(-t_i)$

$\frac{\partial MSE}{\partial \beta_1} = \sum_i \frac{\partial E}{\partial e_i}\frac{\partial e_i}{\partial \beta_1} = \sum_i 2e_i(-\Delta T_i t_i)$

We will start from the solution above (0.02, 0.0)

[6]:
```
beta = np.array([0.02, 0.0])
Nits = 100
mu=1e-6

for n in range(Nits):
    beta0=beta[0]
    beta1=beta[1]

    ypredict = f(beta0, beta1, t, T)
    diff = y-ypredict
    MSE = np.sum(np.power(diff,2))
    print("Iter %d: beta0=%f beta1=%f MSE=%f"%(n, beta0, beta1, MSE))

    gradB0 = 0
    gradB1 = 1
    for i in range(ypredict.size):
        ei = y[i]-ypredict[i]
        gradB0 += -2*ei*t[i]
        gradB1 += -2*ei*t[i]*T[i]

    beta-=mu*np.array([gradB0, gradB1])
```

```
Iter 0: beta0=0.020000 beta1=0.000000 MSE=0.137280
Iter 1: beta0=0.020072 beta1=0.000274 MSE=0.068751
Iter 2: beta0=0.020122 beta1=0.000467 MSE=0.034555
Iter 3: beta0=0.020157 beta1=0.000603 MSE=0.017479
Iter 4: beta0=0.020181 beta1=0.000700 MSE=0.008942
Iter 5: beta0=0.020197 beta1=0.000768 MSE=0.004666
Iter 6: beta0=0.020208 beta1=0.000817 MSE=0.002519
Iter 7: beta0=0.020215 beta1=0.000851 MSE=0.001437
Iter 8: beta0=0.020219 beta1=0.000876 MSE=0.000888
Iter 9: beta0=0.020221 beta1=0.000893 MSE=0.000606
Iter 10: beta0=0.020222 beta1=0.000905 MSE=0.000460
Iter 11: beta0=0.020222 beta1=0.000914 MSE=0.000382
Iter 12: beta0=0.020222 beta1=0.000921 MSE=0.000338
Iter 13: beta0=0.020221 beta1=0.000926 MSE=0.000313
Iter 14: beta0=0.020220 beta1=0.000929 MSE=0.000296
Iter 15: beta0=0.020218 beta1=0.000932 MSE=0.000285
Iter 16: beta0=0.020216 beta1=0.000934 MSE=0.000277
Iter 17: beta0=0.020215 beta1=0.000935 MSE=0.000270
Iter 18: beta0=0.020213 beta1=0.000937 MSE=0.000263
Iter 19: beta0=0.020211 beta1=0.000938 MSE=0.000258
Iter 20: beta0=0.020209 beta1=0.000939 MSE=0.000253
Iter 21: beta0=0.020207 beta1=0.000939 MSE=0.000248
Iter 22: beta0=0.020205 beta1=0.000940 MSE=0.000243
Iter 23: beta0=0.020203 beta1=0.000941 MSE=0.000238
Iter 24: beta0=0.020201 beta1=0.000941 MSE=0.000234
```

```
Iter 25: beta0=0.020200 beta1=0.000942 MSE=0.000229
Iter 26: beta0=0.020198 beta1=0.000943 MSE=0.000225
Iter 27: beta0=0.020196 beta1=0.000943 MSE=0.000221
Iter 28: beta0=0.020194 beta1=0.000944 MSE=0.000217
Iter 29: beta0=0.020192 beta1=0.000944 MSE=0.000213
Iter 30: beta0=0.020191 beta1=0.000945 MSE=0.000209
Iter 31: beta0=0.020189 beta1=0.000945 MSE=0.000205
Iter 32: beta0=0.020187 beta1=0.000946 MSE=0.000202
Iter 33: beta0=0.020185 beta1=0.000946 MSE=0.000198
Iter 34: beta0=0.020184 beta1=0.000946 MSE=0.000194
Iter 35: beta0=0.020182 beta1=0.000947 MSE=0.000191
Iter 36: beta0=0.020180 beta1=0.000947 MSE=0.000187
Iter 37: beta0=0.020179 beta1=0.000948 MSE=0.000184
Iter 38: beta0=0.020177 beta1=0.000948 MSE=0.000181
Iter 39: beta0=0.020175 beta1=0.000949 MSE=0.000177
Iter 40: beta0=0.020174 beta1=0.000949 MSE=0.000174
Iter 41: beta0=0.020172 beta1=0.000950 MSE=0.000171
Iter 42: beta0=0.020171 beta1=0.000950 MSE=0.000168
Iter 43: beta0=0.020169 beta1=0.000950 MSE=0.000165
Iter 44: beta0=0.020167 beta1=0.000951 MSE=0.000162
Iter 45: beta0=0.020166 beta1=0.000951 MSE=0.000159
Iter 46: beta0=0.020164 beta1=0.000952 MSE=0.000156
Iter 47: beta0=0.020163 beta1=0.000952 MSE=0.000154
Iter 48: beta0=0.020162 beta1=0.000953 MSE=0.000151
Iter 49: beta0=0.020160 beta1=0.000953 MSE=0.000148
Iter 50: beta0=0.020159 beta1=0.000953 MSE=0.000146
Iter 51: beta0=0.020157 beta1=0.000954 MSE=0.000143
Iter 52: beta0=0.020156 beta1=0.000954 MSE=0.000140
Iter 53: beta0=0.020154 beta1=0.000954 MSE=0.000138
Iter 54: beta0=0.020153 beta1=0.000955 MSE=0.000136
Iter 55: beta0=0.020152 beta1=0.000955 MSE=0.000133
Iter 56: beta0=0.020150 beta1=0.000956 MSE=0.000131
Iter 57: beta0=0.020149 beta1=0.000956 MSE=0.000129
Iter 58: beta0=0.020148 beta1=0.000956 MSE=0.000126
Iter 59: beta0=0.020146 beta1=0.000957 MSE=0.000124
Iter 60: beta0=0.020145 beta1=0.000957 MSE=0.000122
Iter 61: beta0=0.020144 beta1=0.000957 MSE=0.000120
Iter 62: beta0=0.020142 beta1=0.000958 MSE=0.000118
Iter 63: beta0=0.020141 beta1=0.000958 MSE=0.000116
Iter 64: beta0=0.020140 beta1=0.000958 MSE=0.000114
Iter 65: beta0=0.020139 beta1=0.000959 MSE=0.000112
Iter 66: beta0=0.020138 beta1=0.000959 MSE=0.000110
Iter 67: beta0=0.020136 beta1=0.000959 MSE=0.000108
Iter 68: beta0=0.020135 beta1=0.000960 MSE=0.000106
Iter 69: beta0=0.020134 beta1=0.000960 MSE=0.000104
Iter 70: beta0=0.020133 beta1=0.000960 MSE=0.000103
Iter 71: beta0=0.020132 beta1=0.000961 MSE=0.000101
Iter 72: beta0=0.020130 beta1=0.000961 MSE=0.000099
```

```
Iter 73: beta0=0.020129 beta1=0.000961 MSE=0.000098
Iter 74: beta0=0.020128 beta1=0.000962 MSE=0.000096
Iter 75: beta0=0.020127 beta1=0.000962 MSE=0.000094
Iter 76: beta0=0.020126 beta1=0.000962 MSE=0.000093
Iter 77: beta0=0.020125 beta1=0.000963 MSE=0.000091
Iter 78: beta0=0.020124 beta1=0.000963 MSE=0.000090
Iter 79: beta0=0.020123 beta1=0.000963 MSE=0.000088
Iter 80: beta0=0.020122 beta1=0.000963 MSE=0.000087
Iter 81: beta0=0.020121 beta1=0.000964 MSE=0.000085
Iter 82: beta0=0.020120 beta1=0.000964 MSE=0.000084
Iter 83: beta0=0.020119 beta1=0.000964 MSE=0.000082
Iter 84: beta0=0.020118 beta1=0.000964 MSE=0.000081
Iter 85: beta0=0.020117 beta1=0.000965 MSE=0.000080
Iter 86: beta0=0.020116 beta1=0.000965 MSE=0.000078
Iter 87: beta0=0.020115 beta1=0.000965 MSE=0.000077
Iter 88: beta0=0.020114 beta1=0.000966 MSE=0.000076
Iter 89: beta0=0.020113 beta1=0.000966 MSE=0.000075
Iter 90: beta0=0.020112 beta1=0.000966 MSE=0.000074
Iter 91: beta0=0.020111 beta1=0.000966 MSE=0.000072
Iter 92: beta0=0.020110 beta1=0.000967 MSE=0.000071
Iter 93: beta0=0.020109 beta1=0.000967 MSE=0.000070
Iter 94: beta0=0.020108 beta1=0.000967 MSE=0.000069
Iter 95: beta0=0.020107 beta1=0.000967 MSE=0.000068
Iter 96: beta0=0.020106 beta1=0.000968 MSE=0.000067
Iter 97: beta0=0.020106 beta1=0.000968 MSE=0.000066
Iter 98: beta0=0.020105 beta1=0.000968 MSE=0.000065
Iter 99: beta0=0.020104 beta1=0.000968 MSE=0.000064
```

1.3 Analytical solution

For linear systems of the form $y = A\beta$, the least squares solution is $\beta^* = (A^T A)^{-1} A^T y$.

[7]:
```
A=np.row_stack((t, np.multiply(t,T))).transpose()
print(A)
```

```
[[  8 -32]
 [ 16 -48]
 [ 24 -48]
 [ 32 -32]
 [ 40  40]
 [ 48  96]
 [ 64 192]
 [ 72 288]]
```

[8]:
```
betaLS=np.matmul(np.linalg.inv(np.matmul(A.transpose(),A)),(np.matmul(A.
 ↪transpose(),y)))
print(betaLS)
```

```
[0.02  0.001]
```

Lec030405d

August 17, 2023

1 Optimization differential equations' parameters

First-order release of a drug responds to the differential equation

$\frac{dA}{dt} = -KA$

where A is the amount still available for release, $A(0)$ =A_0$. Because, it is a very simple equation, its analytical solution is of the form

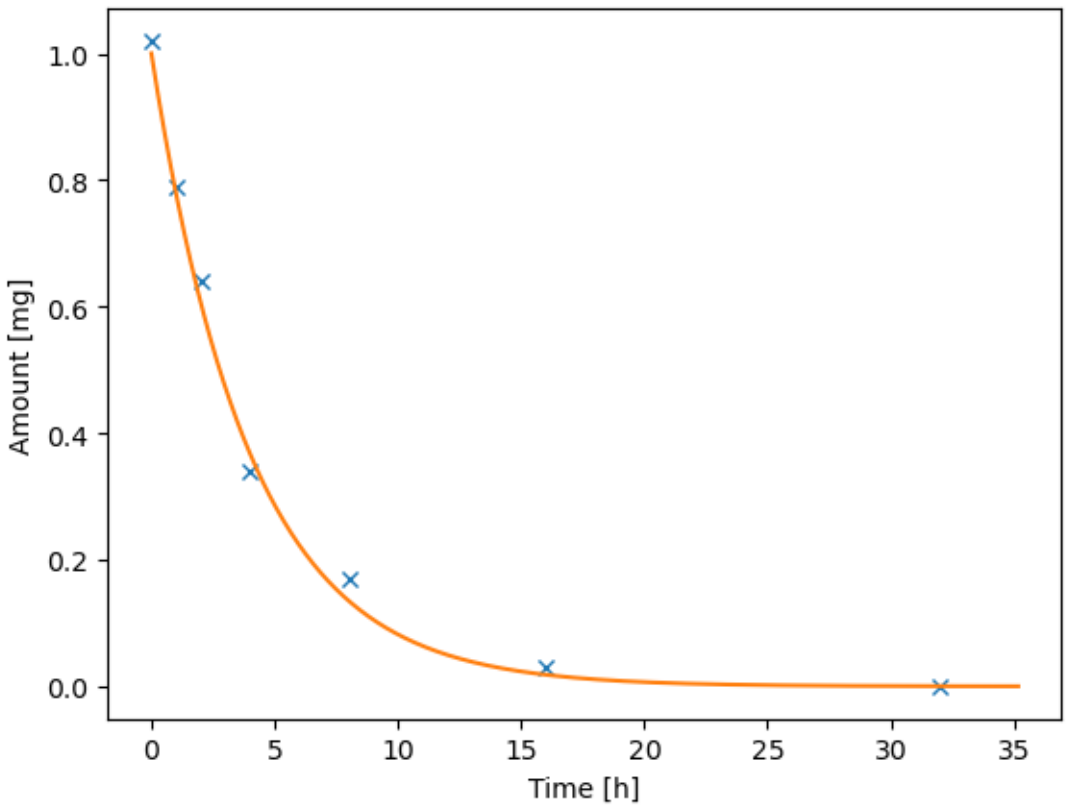
$A(t) = A_0 \exp(-Kt)$

Still, we will find K for some specific data using the differential equation.

[1]:
```
import numpy as np

# Simulate data
K=1.0/4.0;                          # True constant [h^-1]
t=np.array([0, 1, 2, 4, 8, 16, 32]) # Sampling times [h]
A0=1;                               # Initial amount [mg]

y=A0*np.exp(-K*t)                   # True underlying amount
print(y)
```

```
[1.00000000e+00 7.78800783e-01 6.06530660e-01 3.67879441e-01
 1.35335283e-01 1.83156389e-02 3.35462628e-04]
```

[2]:
```
yobs=np.array([1.02, 0.79, 0.64, 0.34, 0.17, 0.03, 0.0])
import matplotlib.pyplot as plt
%matplotlib inline

plt.plot(t,yobs,'x')
tF = np.max(t)*1.1
ts=np.arange(0,tF,1/30.0)           # Simulation time
ys=A0*np.exp(-K*ts)
plt.plot(ts,ys)
plt.xlabel('Time [h]');
plt.ylabel('Amount [mg]');
```



[3]:
```
# Definition of the ODE and the objective function
from scipy.integrate import odeint
from scipy import interpolate

def F(A,t,K):
    return -K*A

def MSE(K, t, yobs, tF):
    ts=np.arange(0,tF,1/30.0)
    A = np.resize(odeint(F, np.array([A0]), ts, args=(K,)),ts.shape)

    Afunc = interpolate.interp1d(ts,A)
    ypred = Afunc(t)
    e=yobs-ypred
    MSE = np.sum(np.power(e,2.0))
    print("K=%f MSE=%f"%(K, MSE))
    return MSE
```

[4]:
```
# Optimization
from scipy.optimize import differential_evolution
Kbounds = [(0,1)]
```

```
result = differential_evolution(MSE, Kbounds, args=(t, yobs, tF))
```

```
K=0.988019 MSE=0.558824
K=0.923908 MSE=0.516527
K=0.668136 MSE=0.321176
K=0.570860 MSE=0.236920
K=0.863721 MSE=0.474389
K=0.211769 MSE=0.009050
K=0.768958 MSE=0.403203
K=0.085229 MSE=0.363829
K=0.635994 MSE=0.293803
K=0.342104 MSE=0.044166
K=0.276766 MSE=0.009752
K=0.033805 MSE=1.174668
K=0.441372 MSE=0.122379
K=0.527300 MSE=0.198205
K=0.175118 MSE=0.038232
K=0.488371 MSE=0.163559
K=0.257335 MSE=0.004756
K=0.013026 MSE=2.102336
K=0.212384 MSE=0.008796
K=0.211769 MSE=0.009050
K=0.386172 MSE=0.076591
K=0.257009 MSE=0.004701
K=0.464544 MSE=0.142533
K=0.233641 MSE=0.003649
K=0.209651 MSE=0.009975
K=0.376798 MSE=0.069296
K=0.495078 MSE=0.169511
K=0.253077 MSE=0.004113
K=0.246362 MSE=0.003473
K=0.397718 MSE=0.085808
K=0.472995 MSE=0.149964
K=0.285432 MSE=0.012935
K=0.248110 MSE=0.003594
K=0.480087 MSE=0.156224
K=0.481834 MSE=0.157769
K=0.178804 MSE=0.033831
K=0.037827 MSE=1.061360
K=0.244615 MSE=0.003386
K=0.278186 MSE=0.010237
K=0.239076 MSE=0.003338
K=0.269313 MSE=0.007462
K=0.204998 MSE=0.012289
K=0.230293 MSE=0.004029
K=0.221584 MSE=0.005740
K=0.191604 MSE=0.021340
```

```
K=0.220182 MSE=0.006119
K=0.255931 MSE=0.004525
K=0.221262 MSE=0.005824
K=0.207981 MSE=0.010760
K=0.259736 MSE=0.005195
K=0.266982 MSE=0.006838
K=0.260885 MSE=0.005425
K=0.228745 MSE=0.004255
K=0.215028 MSE=0.007777
K=0.222221 MSE=0.005578
K=0.240808 MSE=0.003315
K=0.271128 MSE=0.007980
K=0.223953 MSE=0.005167
K=0.295176 MSE=0.017116
K=0.251714 MSE=0.003945
K=0.245839 MSE=0.003443
K=0.237522 MSE=0.003390
K=0.226591 MSE=0.004625
K=0.240501 MSE=0.003316
K=0.221421 MSE=0.005783
K=0.245839 MSE=0.003443
K=0.235297 MSE=0.003515
K=0.246015 MSE=0.003453
K=0.235284 MSE=0.003516
K=0.254151 MSE=0.004258
K=0.242352 MSE=0.003323
K=0.232617 MSE=0.003750
K=0.227464 MSE=0.004467
K=0.253707 MSE=0.004197
K=0.241958 MSE=0.003319
K=0.238932 MSE=0.003342
K=0.251146 MSE=0.003881
K=0.231394 MSE=0.003888
K=0.245010 MSE=0.003402
K=0.245254 MSE=0.003414
K=0.236162 MSE=0.003459
K=0.246531 MSE=0.003483
K=0.243627 MSE=0.003351
K=0.238156 MSE=0.003365
K=0.230611 MSE=0.003987
K=0.246019 MSE=0.003453
K=0.236748 MSE=0.003427
K=0.246019 MSE=0.003453
K=0.246535 MSE=0.003483
K=0.241944 MSE=0.003318
K=0.241940 MSE=0.003318
K=0.239623 MSE=0.003327
K=0.236939 MSE=0.003417
```

```
K=0.240631 MSE=0.003315
K=0.236926 MSE=0.003417
K=0.244563 MSE=0.003384
K=0.241992 MSE=0.003319
K=0.236694 MSE=0.003429
K=0.246105 MSE=0.003458
K=0.242632 MSE=0.003328
K=0.247860 MSE=0.003575
K=0.239021 MSE=0.003339
K=0.245377 MSE=0.003420
K=0.233755 MSE=0.003639
K=0.243939 MSE=0.003361
K=0.242452 MSE=0.003325
K=0.241384 MSE=0.003315
K=0.244086 MSE=0.003366
K=0.241528 MSE=0.003315
K=0.239318 MSE=0.003333
K=0.245369 MSE=0.003419
K=0.238675 MSE=0.003349
K=0.241154 MSE=0.003314
K=0.235855 MSE=0.003478
K=0.242442 MSE=0.003325
K=0.244285 MSE=0.003373
K=0.244964 MSE=0.003400
K=0.240907 MSE=0.003314
K=0.241346 MSE=0.003315
K=0.240186 MSE=0.003319
K=0.241154 MSE=0.003314
K=0.241154 MSE=0.003314
K=0.240419 MSE=0.003317
K=0.240419 MSE=0.003317
K=0.241092 MSE=0.003314
K=0.241092 MSE=0.003314
```

[5]:
```
print("Optimum K=%f"%result.x)
print("MSE=%f"%result.fun)
```

```
Optimum K=0.241092
MSE=0.003314
```

[6]:
```
Kopt=result.x
ys2=A0*np.exp(-Kopt*ts)
plt.plot(t,y,'x')
plt.plot(ts,ys)
plt.plot(ts,ys2)
plt.xlabel('Time [h]');
plt.ylabel('Amount [mg]');
```

```
plt.legend(['Observed','Ground truth','Optimum']);
```