Introduction

Target Audience

This workshop is intended for students with an aptitude in science and/or mathematics at upper secondary school or junior college level. It introduces students to the use of computers as a numeric calculation/simulation tool and is also meant to empower them to create simulations of physical systems on their own for self-study and research.

> Text highlighted in this manner refers to advanced or optional concepts and is intended as supplementary information which may be used in the workshop to challenge and inspire advanced students. It is however not essential to understand this material to follow the workshop.

Modelling Nature using Mathematics

What is a model?

Science aims to construct *models* of nature. A model is a real or imagined system that follows clear rules and predicts outcomes in a variety of situations that agree with observations. A good model would allow to explain or predict as many observations as possible about nature, while being as simple as possible.

Models (including so-called scientific "facts") are not the *same* as reality – at best, they *agree* with reality. Indeed, sometimes models turn out to be wrong. So why bother with models, then? Models make it possible for our limited human minds to organise, explain, predict and plan – important survival skills.

Mathematics as a model-building tool

When several simple models are used to build a more complex model, they may contribute elements predicting different or even opposite effects. For accounting how much each element contributes to the result, it is essential to have a quantitative description. It is also important to keep rigorously track of the meaning of each model using clear, unambiguous language/notation, and manipulate information in a way that avoids thinking mistakes. Normal human language is often ambiguous, imprecise, and wordy. The best tool we have to summarize and manipulate models in a concise and precise way is mathematics.

Locality and Differential Equations

Assume a model that predicts the position x of a particle as a second-order polynomial in the time t:

$$x(t)=a+b\cdot t+c\cdot t^2$$

Can this function be defined from *local* information, that is from its properties at a single time t (let's say $t=0$)?

There are three unknowns (a,b,c). The value of the function itself $x(0)$ does not provide enough information to determine several unknowns. However, the derivatives of the function are local properties:

$$x(t)=a+b\cdot t+c\cdot t^2;\quad x(0)=a$$
$$\dot{x}(t)=\frac{dx}{dt}=b+2c\cdot t;\quad \dot{x}(0)=b$$
$$\ddot{x}(t)=d\frac{\dot{x}}{dt}=\frac{d^2x}{dt^2}=2\cdot c;\quad \ddot{x}(0)=2c$$

The unknowns can therefore be found from the value of the function and its derivatives.

$$x(t)=x(0)+\dot{x}(0)\cdot t+\frac{1}{2}\ddot{x}(0)\cdot t^2$$

This illustrates that *local formulations of models take the form of equations involving functions and their derivatives (differential equations)*. Equations involving derivatives with respect to only a a single parameter (such as t) are called *ordinary differential equations* (ODEs).

Mathematical models of nature are commonly described locally, i.e. by differential equations.

> Local formulations are not the only possible way to describe models, but they are preferred since they help break down complex models into simpler ones. For example, the motion of a planet could be described by a huge set of equations relating its coordinates to the coordinates of all other objects ion the universe. This is obviously not practical. By introducing the notion of a gravitational field, the planet's motion can be described by its *local* interaction with the gravitational field. Determination of the gravitational field is split off as a separate problem.

Newton's law as an Ordinary Differential Equation (ODE)

The development of calculus in the 18th century allowed Isaac Newton to describe the motion of objects using a differential equation:

$$\vec{F}=m\cdot\vec{a}$$

- The mass m is usually considered constant (but not always – for example, a plane or rocket will shed mass by expelling fuel or its combustion products).
- The velocity $\vec{v}$ is the change of the object's position $\vec{x}$ per time unit, i.e. the derivative $\vec{v}=\dot{\vec{x}}=\frac{d}{dt}\vec{x}$.
- The acceleration $\vec{a}$ is the instantaneous change of velocity $\vec{v}$ per time unit – in other words, the derivative $\vec{a}=\dot{\vec{v}}=\frac{d}{dt}\vec{v}=\ddot{\vec{x}}=\frac{d^2x}{dt^2}$.
- The force $\vec{F}$ may commonly depend on the position of the object $\vec{x}$ (example: gravitational force), its velocity $\vec{v}$ (example: air drag/friction), and time t (example: oscillatory force): $\vec{F}=\vec{F}(\vec{x},\vec{v},t)=\vec{F}(\vec{x},\dot{\vec{x}},t)$.

With the above in mind, Newton's equation of motion can be written as

$$\ddot{\vec{x}}=\frac{1}{m}\vec{F}(\vec{x},\dot{\vec{x}},t)$$

that is, it is an ODE relating the object's position $\vec{x}(t)$ to its 1st and 2nd derivatives (it is a 2nd-order differential equation).

> ODEs usually have multiple solutions, so it is important to specify *which* solution is required. There are several ways of doing this, but they can be mostly grouped into two categories:

Boundary value problems state the desired function values at the beginning and end (boundaries) of a time period. They ask,"what happens in between?". For example:

- What path and what speed should an air plane use to leave Singapore Changi at 23:50 and arrive in London Heathrow on schedule 13 hours later?
- A violin string is clamped down on both ends. What vibrations are possible on the string?

Initial value problems state the initial situation of a problem and ask how the situation evolves in the future. They ask, "what is going to happen?". For example:

- Given today's weather data, how will the weather change over the next 3 days?
- Due to friction in the atmosphere, a satellite is falling back to earth. When and where may it hit the surface?

The technique for solving an ODE will largely depend on the type of problem. In this workshop, we cover initial value problems (IVPs) only.

Solving Ordinary Differential Equations (Initial Value Problem)

Formal Solution of Newton's Differential Equation

Newton's equation $\ddot{\vec{x}}=\frac{1}{m}\vec{F}(\vec{x},\dot{\vec{x}},t)$ can also be written as a system of two first-order differential equations by renaming the first derivate $\vec{v}=\dot{\vec{x}}$:

$$\frac{d}{dt}\begin{pmatrix}\vec{x}\\ \vec{v}\end{pmatrix}=\begin{pmatrix}\dot{\vec{x}}\\ \dot{\vec{v}}\end{pmatrix}=\begin{pmatrix}\vec{v}\\ \frac{\vec{F}(\vec{x},\vec{v},t)}{m}\end{pmatrix}$$

In other words, Newtons law in n space coordinates corresponds to a set of first-order differential equation of $2n$ variables (half of them positions, the other half velocities). The resulting 1st order initial value problem can be formally solved using integration:

$$\begin{pmatrix}\vec{x}(t_1)\\ \vec{v}(t_1)\end{pmatrix}=\int_{t_0}^{t_1}\begin{pmatrix}\dot{\vec{v}}(t)\\ \dot{\vec{x}}(t)\end{pmatrix}dt+\begin{pmatrix}\vec{x}_0\\ \vec{v}_0\end{pmatrix}=\int_{t_0}^{t_1}\begin{pmatrix}\vec{v}(t)\\ \frac{\vec{F}(\vec{x}(t),\vec{v}(t),t)}{m}\end{pmatrix}dt+\begin{pmatrix}\vec{x}_0\\ \vec{v}_0\end{pmatrix}$$

The integration constants $\vec{x}_0$ and $\vec{v}_0$ are the initial (starting) values at the time t_0. There are two practical issues:

- "Textbook problems" aside, the functions encountered may be challenging or even impossible to integrate in closed form.
- The equation is *implicit* ($\vec{x}$ and $\vec{v}$ appear on both sides of the equation), i.e. to calculate the integral, the solutions $\vec{x}$ and $\vec{v}$ must already known (chicken-and-egg problem).

For many practical problems, an exact analytical solution is not necessary. In this case, one may solve the equations approximately using numeric techniques that can handle implicit equations.

Approximate Integration of Initial Value Problems in Python

Note: For all of the following code examples, the following statements for including mathematics and plotting libraries should be included at the beginning of the program:


```
import scipy
from matplotlib import pyplot
```

For illustration, consider the one-dimensional, first-order ODE and its formal solution

$$\dot{x}(t)=f(x(t),t)$$

$$x(t_1)=x(t_0)+\int_{t_0}^{t_1} f(x(t),t)dt$$

The mathematical challenge is to approximately evaluate the integral. This is easier if the time span t_1-t_0 is relatively short (short-term predictions are easier than long-term predictions). Furthermore, one usually does not just want a single result $x(t_1)$, but an entire series $x(t_0),x(t_1),x(t_2),x(t_3),x(t_4)\ldots$ that traces the evolution of the function $x(t)$ and can for example be used to draw a graph. Both of these concerns can be addressed by dividing the integral into many integrals, each spanning a smaller time interval:

$$x(t_n)=x(t_0)+\int_{t_0}^{t_1} f(x(t),t)dt+\int_{t_1}^{t_2} f(x(t),t)dt+\int_{t_2}^{t_3} f(x(t),t)dt+\ldots+\int_{t_{n-1}}^{t_n} f(x(t),t)dt$$

Denoting $x_i=x(t_i)$, this can be calculated by repeating the same integration process:

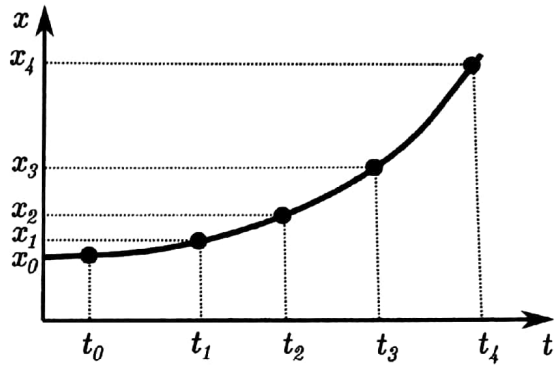
$$x_0=x(t_0)$$

$$x_1=x_0+\int_{t_0}^{t_1} f(x(t),t)dt$$

$$x_2=x_1+\int_{t_1}^{t_2} f(x(t),t)dt$$

$$\ldots$$

$$x_{i+1}=x_i+\int_{t_i}^{t_{i+1}} f(x(t),t)dt$$



The following Python function performs this general algorithm. It takes as its arguments **integrator** (a Python function to calculate the above integral), **f** (the function $f(x(t),t)$, **x0** (the initial value x_0) and a list of times, time axis.

```
def solveODE(integrator, f, x0, timeaxis):
    "Numerical integration of ODE"
    # convert the list of initial values (e.g. x, v) to a vector
    x = scipy.array(x0)
    # list holding integrated data points, starting with the initial value
    sums = [x]
    # apply integration step method and record the results
    for i in range(1, len(timeaxis)):
        t0 = timeaxis[i-1]
        t1 = timeaxis[i]
        x = x + integrator(f, x, t0, t1)
        sums.append(x)
    return scipy.array(sums)
```

What still needs to be provided is a method to approximately calculate the integral (i.e., a Python function that can be passed as the **integrator** argument).

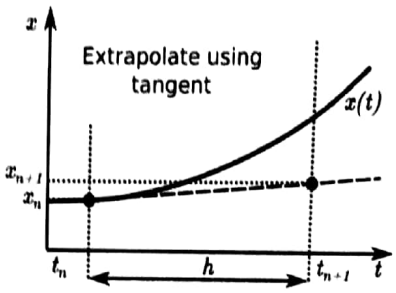
Numeric Integrators

Euler's method assumes that the function $f(x,t)$ changes only slowly from its initial value, so that any changes during the integration period are negligible:

$$k_1 = f(x_0, t_0)$$

$$\int_{t_0}^{t_1} f(x(t),t)\,dt \approx h \cdot k_1 \qquad (h = t_1 - t_0)$$

This corresponds to extrapolating the graph of the function $x(t)$ with its tangent.




```
def EulerStep(f, x0, t0, t1):
    "Single Euler integration step"
    h = t1 - t0
    k1 = f(x0, t0)
    return h * k1
```

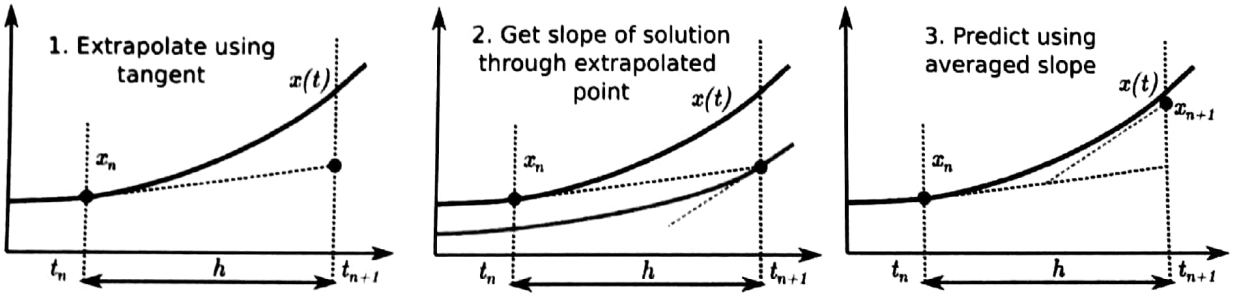
If the graph of $x(t)$ is curved, this simple extrapolation is obviously not particularly accurate. From a practical point of view, Euler's method is pretty much useless - but it shows the general idea of the following methods.

Heun's method is a refinement of Euler's method. It first *predicts* the result using Euler's method. Then it tries to *correct* for the error made by calculating the slope of the curve (i.e. the value of the function F) at the predicted point. The original slope and the slope at the predicted point are then averaged and used to estimate the final result:

$$k_1 = f(x(t_0), t_0)$$

$$k_2 = f(x(t_0) + h \cdot k_1, t_1)$$

$$\int_{t_0}^{t_1} f(x(t),t)\,dt \approx h \cdot \frac{k_1 + k_2}{2}$$




```
def HeunStep(f, x0, t0, t1):
    "Single Heun integration step"
    h = t1 - t0
    k1 = f(x0, t0)
    k2 = f(x0 + h*k1, t0+h)
    return h * (k1 + k2)/2
```

This approach of calculating and averaging intermediate predictions can be refined further, leading to an entire family of *Runge-Kutta methods* that are even more accurate. The most commonly used Runge-Kutta method is called RK4 and is popular because it is reasonably accurate, but also still reasonably easy to implement:

$$k_1 = f(x_0, t_0)$$
$$k_2 = f\left(x_0 + \frac{h}{2} \cdot k_1, t_0 + \frac{h}{2}\right)$$
$$k_3 = f\left(x_0 + \frac{h}{2} \cdot k_2, t_0 + \frac{h}{2}\right)$$
$$k_4 = f(x_0 + h \cdot k_3, t_0 + h)$$
$$\int_{t_0}^{t_1} f(x(t), t)\, dt \approx h \cdot \frac{k_1 + 2k_2 + 2k_3 + k_4}{6}$$

```
def RK4Step(f, x0, t0, t1):
    "Single 4-th order Runge-Kutta integration step"
    h = t1 - t0
    k1 = f(x0, t0)
    k2 = f(x0 + h/2*k1, t0+h/2)
    k3 = f(x0 + h/2*k2, t0+h/2)
    k4 = f(x0 + h*k3, t0+h)
    return h * (k1 + 2*k2 + 2*k3 + k4)/6
```

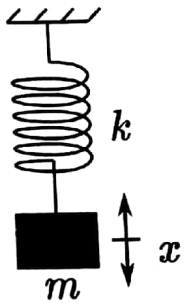
Many other, more refined methods exist. Some of them are designed to do better with "difficult" functions to integrate, others estimate the error of the approximation and adjust the time step size automatically to achieve good accuracy. Implementing such methods correctly is not always trivial, and therefore it is common to rely on proven code provided in mathematical software libraries (such as Python modules).

Comparing the Different Integrators

To demonstrate the performance of the different integrators, we can compare their results to each other, and to the exact solution for a known problem.

A good demonstration problem is a spring-mass pendulum with spring constant k and mass m. Using Hooke's law for the restoring force of the spring $F = -k \cdot x$, Newton's equation of motion in this case is:

$$\ddot{x} = -\frac{k}{m} \cdot x$$



or, as the equivalent first-order equation,

$$\begin{pmatrix} \dot{x} \\ \dot{v} \end{pmatrix} = \begin{pmatrix} v \\ -\frac{k}{m} x \end{pmatrix}$$

It is straightforward to express this model as Python code, providing concrete values for k, m, x_0 and v_0, and points in time for which to solve (start time, end time, and number of time steps):

```
def f(X, t):
    x, v = X
    a = -k/m * x
    return scipy.array([v, a])

k = 10      # spring constant in N/m
m = 1.0     # mass in kg
x0 = 1.0    # initial elongation in m
v0 = 0.5    # initial velocity in m/s
X0 = [x0, v0]   # initial values as a pair
startTime = 0 # in s
stopTime = 5 # in s
```



```
steps = 50
timeaxis = scipy.linspace(startTime, stopTime, steps)
```

Activity

Perform the actual numerical solution of the ODE (replace **integrator** with the name of the desired integrator, such as **EulerStep** etc.):

```
trajectory = solveODE(integrator, f, X0, timeaxis)
```

To display the result, use

```
pyplot.plot(timeaxis, trajectory[:, 0], color = "blue")
pyplot.show()
```

You can superimpose several graphs by repeatedly calling **pyplot.plot** (using different colours for easy distinction) before calling **pyplot.show**. Use this to compare the results obtained using the Euler, Heun, and RK4 integrators, as well as the exact solution:

$$x(t) = x_0 \cdot \cos \omega t + \frac{v_0}{\omega} \sin \omega t \qquad \left(\omega = \sqrt{\frac{k}{m}}\right)$$

- How well do the different integrators perform?
- What basic physical principle is very obviously violated by the result obtained using the Euler method?
- How does the performance change if you change the number of steps (i.e., increase or reduce the size of the time steps)? How many steps do you need for the different integrators to get a visually accurate result?

Using scipy's ODE Integrator

In the following, we will use a more advanced (and more efficiently programmed) integration routine provided by Python's **scipy** (Scientific Python) library module:

```
from scipy.integrate import odeint
```

The use of **odeint** is similar to our own home-brewed ODE solver, but we do not need to specify a step method - **odeint** automatically chooses one from a built-in set of methods. It also may subdivide the time steps further to achieve reasonable accuracy. To use the routine call it as

```
X = odeint(f, X0, timeaxis)
```

where **f** is the function representing the ODE to be integrated, **X0** is the initial value, and **timeaxis** is an (ordered) list of points in time for which to calculate the result. **odeint** also allows to specify many optional parameters to fine-tune its internal workings; however these details are beyond the scope of this workshop.

Application Problems

To allow focusing on the scientific principles rather than extensive math, textbook models are often simplified/idealised. This is a main reason why science problems often appear unrelated to the "real world".

Relegating the math to numerical integration on the computer, it is possible to explore more

realistic scenarios. The true value of numeric simulations is for problems for which an analytical solution is difficult to find – which includes most "real-world" scenarios.

> Up to the 1960s, the term "computer" typically referred to a worker doing tedious calculations (such as numerical integration), using mechanical calculation machines for hours on end, on the behalf of scientists or engineers. Computers were often female, as this "invisible" background work was one of the few STEM careers without gender barriers. For example, the trajectories of the Apollo spacecraft that brought mankind to the moon were largely determined by human computers.

The availability of affordable, electronic computers has revolutionized industry, since it allowed to accurately and rapidly predict, test and optimize the performance of engineering designs before even building the first prototype. Many designs that once took top researchers weeks can now be done in a few hours.

Ballistics in air

> Ballistic motion of a projectile is one of the earliest uses of computer simulations, since 1) it is of military significance and 2) most governments spend lavishly on the military, so the high cost of early computers was not an issue. In the 1940s, ENIAC, considered by many to be the first general-purpose electronic digital computer, was built for the US army specifically for calculating the trajectories of artillery shells.

Ballistic motion of a projectile under the sole influence of gravity is a standard textbook problem. However, a realistic simulation needs to take additional factors into account, one of them being aerodynamic effects such as air drag (friction).

When a projectile travels for a distance Δx, it has to push a volume of $V = \Delta x \cdot A$ and a mass $m = \rho \cdot \Delta x \cdot A$ of the surrounding medium out of the way (where A is the cross section of the projectile and ρ the density of the medium). If the medium gets displaced at an average velocity v_{medium}, it carries kinetic energy $E_{\text{kin}} = \frac{m v_{\text{medium}}^2}{2}$ with it that is (in the case of turbulent flow) lost.

The projectile has to provide this as work $F \cdot \Delta x = \frac{m v_{\text{medium}}^2}{2}$ and hence experiences a drag force $F_{\text{drag}} = \frac{m v_{\text{medium}}^2}{2 \Delta x} = \frac{1}{2} \rho \cdot A \cdot v_{\text{medium}}^2$. Assuming that the velocity at which the medium is displaced is proportional to the velocity of the projectile relative to the medium, one obtains

$$F_{\text{drag}} = \frac{1}{2} c_w \rho \cdot A \cdot v^2 = \frac{1}{2} C \cdot v^2 \quad (C = c_w \cdot \rho \cdot A)$$

where c_w is a unitless constant determined by the flow pattern of the medium, which in turn depends on the shape of the projectile.

In two dimensions (distance x and height y) over flat Earth (i.e. neglecting Earth's curvature), the force on a projectile of mass m is the sum of gravitational and drag forces:

$$\vec{F} = \vec{G} + \vec{F}_{\text{drag}} = m \cdot \vec{g} - \frac{C \cdot (\vec{v} - \vec{u})^2}{2} \cdot \frac{\vec{v} - \vec{u}}{|\vec{v} - \vec{u}|} = \begin{pmatrix} 0 \\ -m \cdot g \end{pmatrix} - \frac{C}{2} \cdot \sqrt{(v_x - u_x)^2 + (v_y - u_y)^2} \cdot \begin{pmatrix} v_x - u_x \\ v_y - u_y \end{pmatrix}$$

where $\vec{v}$ is the velocity of the projectile and $\vec{u}$ the velocity of the ambient wind. If the wind blows in the horizontal direction, one can eliminate the vertical wind speed component ($u_y = 0$). The overall equation of motion is then

$$\begin{pmatrix}\ddot{x}\\ \ddot{y}\end{pmatrix}=\begin{pmatrix}-\frac{C}{2m}\cdot\sqrt{(v_x-u_x)^2+v_y^2}\cdot(v_x-u_x)\\ -g-\frac{C}{2m}\cdot\sqrt{(v_x-u_x)^2+v_y^2}\cdot v_y\end{pmatrix}$$

or, formulated as first-order equation

$$\begin{pmatrix}\dot{x}\\ \dot{y}\\ \dot{v}_x\\ \dot{v}_y\end{pmatrix}=\begin{pmatrix}v_x\\ v_y\\ -\frac{C}{2m}\cdot\sqrt{(v_x-u_x)^2+v_y^2}\cdot(v_x-u_x)\\ -g-\frac{C}{2m}\cdot\sqrt{(v_x-u_x)^2+v_y^2}\cdot v_y\end{pmatrix}$$

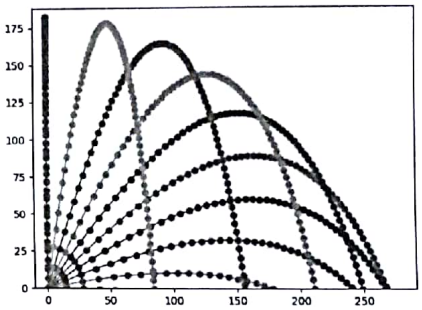
```
import scipy
from scipy.integrate import odeint
from matplotlib import pyplot

def ballistic(X, t):
    x, y, vx, vy = X
    factor = C / (2*m) * scipy.sqrt((vx-ux)**2 + vy**2)
    return [vx, vy, -factor * (vx-ux), -g - factor * vy]

m = 1.0     # mass in kg
rho = 1.225 # density of air in kg/m^3
A = 0.01    # cross section in m^2
c_w = 0.8   # drag parameter
C = c_w * rho * A
g = 9.81    # gravitational acceleration in m/s^2
ux = 0      # horizontal wind speed in m/s
x0, y0 = 0, 0 # initial position
v = 100.0   # initial speed in m/s

timeaxis = scipy.linspace(0, 15, 100)

degrees = scipy.pi/180 # for converting degrees to radians
for alpha in scipy.linspace(0*degrees, 90*degrees, 10):
    vx, vy = v * scipy.cos(alpha), v * scipy.sin(alpha)
    trajectory = odeint(ballistic, [x0, y0, vx, vy], timeaxis)
    pyplot.plot(trajectory[:,0], trajectory [:,1], "o-")
pyplot.ylim(0)  # show only positive heights
pyplot.show()
```



The above code displays the trajectories of a projectile for a range of takeoff values. Things to try:

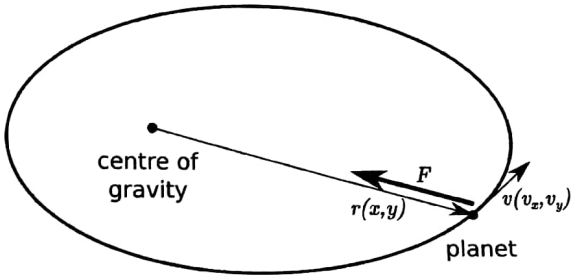
- *Change the cross section* A *or drag parameter* c_w *of the projectile (the case* $c_w=0$ *corresponds to no air drag at all). How does it affect the takeoff angle that carries the projectile the furthest?*
- *For high air drag, note that the projectile quickly approaches a height-independent terminal speed upon falling, regardless of the initial conditions. (Application: parachutes!)*
- *Introduce tail wind (***ux** > 0*) to carry the projectile further.*
- *With head wind (***ux** < 0*), the projectile may turn around and hit the ground behind the firing station (or, if you are particularly unlucky, the projectile will directly hit the station!).*

Planetary orbit

The motion of two bodies attracted to each other by Newton's law of gravitation can be solved analytically, giving rise to elliptical, parabolic, and hyperbolic orbits and Kepler's laws. While this is a standard undergraduate-level textbook problem, numeric integration allows to explore planetary motion without having to go through some tedious mathematics.

Note that the Kepler problem can be solved analytically only for two bodies (i.e. the sun and a planet). As soon as a third body comes into play, one has to resort to approximations – a few special cases aside, *closed form analytical solutions do not exist*. For planning the travel path of space probes, *numeric integration is not just a matter of convenience, but a necessity.*

For simplicity, consider the motion of Sun and Earth in a Cartesian plane, with the origin at the centre of mass. If the position of Earth in this coordinate system is denoted by the vector $\vec{r}=\begin{pmatrix} x \\ y \end{pmatrix}$, the position of the sun is $\vec{R}=-\frac{m}{M}\vec{r}$, so it is sufficient to integrate only the motion of Earth to know both. With the resulting sun-earth distance $d=\left(1+\frac{m}{M}\right)\sqrt{x^2+y^2}$, Newton's law of gravitation then results in



$$\vec{F}=-\frac{G\cdot M\cdot m}{d^2\sqrt{x^2+y^2}}\cdot\begin{pmatrix} x \\ y \end{pmatrix}=-\frac{G\cdot M\cdot m}{\left(1+\frac{m}{M}\right)^2\left(x^2+y^2\right)^{3/2}}\cdot\begin{pmatrix} x \\ y \end{pmatrix}$$

$$\vec{a}=\begin{pmatrix} \ddot{x} \\ \ddot{y} \end{pmatrix}=\frac{\vec{F}}{m}=-\frac{G\cdot M}{\left(1+\frac{m}{M}\right)^2\left(x^2+y^2\right)^{3/2}}\cdot\begin{pmatrix} x \\ y \end{pmatrix}$$

or, expressed as a first-order differential equation:

$$\begin{pmatrix} \dot{x} \\ \dot{y} \\ \dot{v}_x \\ \dot{v}_y \end{pmatrix}=\begin{pmatrix} v_x \\ v_y \\ -k\cdot x \\ -k\cdot y \end{pmatrix} \quad \text{where} \quad k=\frac{G\cdot M}{\left(1+\frac{m}{M}\right)^2\left(x^2+y^2\right)^{(3/2)}}$$

```
def earth(X, t):
    x, y, vx, vy = X
    k = G * M / ((1 + m/M)**2 * (x**2 + y**2)**(3/2))
    return [vx, vy, -k*x, -k*y]
G = 6.67408e-11      # gravitation constant in m^3/(kg*s^2)
m = 5.9722e24        # mass of Earth in kg
M = 1.98855e30       # mass of sun in kg
day = 86400          # duration of a day in seconds
year = 365.256363004 * day# duration of a year
x0, y0 = 1.52096e11, 0    # initial position (perihelion) in meters
vx0, vy0 = 0, 29776.2     # initial velocity in m/s
timeaxis = scipy.linspace(0, 2*year, 100)
trajectory = odeint(earth, [x0, y0, vx0, vy0], timeaxis)
pyplot.plot(trajectory[:,0], trajectory[:,1], "o-")
pyplot.show()
```

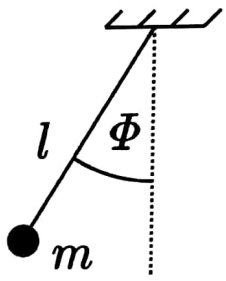
Things to try:

- Reduce the initial value v_y (i.e, slow down Earth). The orbit will become strongly elliptical, and the orbital period will be reduced.
- Increase the initial value v_y (i.e., provide more kinetic energy). You may need to extend the integration period beyond two years as the orbital period increases. Beyond the escape velocity of $v_y \approx 42\,\mathrm{km/s}$, Earth will enter a hyperbolic trajectory and be ejected from the solar system, never to return.

Mathematical Pendulum (Swing)

A mathematical pendulum consists of a point mass attached via a massless spacer of length l to a fixed pivot, in a uniform gravitational field of acceleration g. The equation of motion is

$$\ddot{\phi} = -\frac{g}{l}\sin\phi$$



where ϕ is the angular deviation from the vertical, lowest position (the stable equilibrium point). For small amplitudes, the approximation $\sin\phi \approx \phi$ is commonly made in textbooks, since then the ODE is very easy to solve.

For larger amplitudes, this approximation is unjustifiable, and the period of the oscillation differs notably even at fairly small amplitudes – for example, precise pendulum clocks need to take this into consideration or they will run too slow. However, even for such a simple system, analytical solutions cannot be expressed in closed form using standard functions. Numerical integration allows to gain an understanding of the system with minimum effort.

Simulating the mathematical pendulum is easy:

```
g = 9.81    # gravitational acceleration
l = 1.0     # length of pendulum
b = 0.1     # viscous friction coefficient in 1/s

def f(X, t):
    phi, omega = X
    alpha = -g/l*scipy.sin(phi) - b*omega
    return [omega, alpha]

X0 = [0, 7]  # initial values (angle and angular velocity)
timeaxis = scipy.linspace(0, 15, 150)
trajectory = odeint(f, x0, timeaxis)
```

Depending on the energy of the system, different regimes can be observed:

- *Non-periodic motion* (pendulum keeps rotating)
- *Large-amplitude oscillations of distinctly non-sinusoidal form and long period* T
- *Small-amplitude oscillations of sinusoidal form and a period approaching* $T \to 2\pi\sqrt{l/g}$ *(harmonic oscillator).*

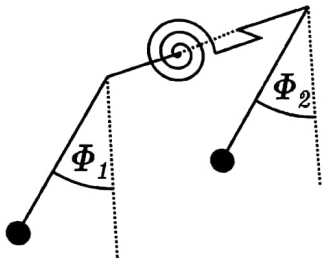
By introducing a small positive damping term b (that is, viscous friction), the system will go *through these regimes in the order listed above.*

Outlook: Nonlinear Systems and Chaos

Coupled Mathematical Pendulums

Let's investigate the dynamics of not one, but two mathematical pendulums that interact with each other. To keep things as simple as possible, both pendulums are identical and frictionless. The pendulums are connected via a torsional spring, so the motion of one pendulum will transfer a moment of force to the other that pulls it along. The system is modeled by the equation

$$\begin{pmatrix}\ddot{\phi}_1\\ \ddot{\phi}_2\end{pmatrix}=\begin{pmatrix}-\omega_0^2\cdot\sin\phi_1+k\cdot(\phi_2-\phi_1)\\ -\omega_0^2\cdot\sin\phi_2+k\cdot(\phi_1-\phi_2)\end{pmatrix}$$



where k is proportional to the torsional spring constant (which is $k\cdot l^2\cdot m$). Formulated as first-order equation, one obtains

$$\begin{pmatrix}\dot{\phi}_1\\ \dot{\phi}_2\\ \dot{\omega}_1\\ \dot{\omega}_2\end{pmatrix}=\begin{pmatrix}\omega_1\\ \omega_2\\ -\omega_0^2\cdot\sin\phi_1+k\cdot(\phi_2-\phi_1)\\ -\omega_0^2\cdot\sin\phi_2+k\cdot(\phi_1-\phi_2)\end{pmatrix}$$

where $\omega_{1,2}$ are the respective angular velocities. The code below integrates the equation and displays the oscillation angles of both pendulums as the system evolves:

```
def coupledpendulums(X, t):
    phi1, phi2, omega1, omega2 = X
    return [omega1, omega2,
       -omega0**2*scipy.sin(phi1) + k*(phi2-phi1),
       -omega0**2*scipy.sin(phi2) + k*(phi1-phi2)]
omega0 = 1.0
k = 0.001
x0 = [1, 1.01, 0, 0]
timeaxis = scipy.linspace(0, 1000, 5000)
trajectory1 = odeint(coupledpendulums, x0, timeaxis)
pyplot.plot(timeaxis, trajectory1[:,0])
pyplot.plot(timeaxis, trajectory1[:,1])
pyplot.show()
```

- Does the motion of the pendulums appear systematic/regular?
- Does the motion of the system appear to be periodic (that is, strictly repeating)?

What happens if the starting amplitude is increased? Try the following starting condition with everything else unchanged:

```
x0 = [3, 3.01, 0, 0]
```

Now, change the starting condition by a tiny amount (about 0.03%):

```
x0 = [3, 3.011, 0, 0]
```

and compare the result with the previous simulation. Does the result look similar?

Now, make a minor adjustment to the number of integration steps from 5000 to 4500 and compare the simulation result:

```
timeaxis = scipy.linspace(0, 1000, 4500)
```

The coupled pendulums show typical properties of a *chaotic system*:

- The evolution of the system follows precise rules and can be predicted short-term (the system is deterministic, not random).
- The slightest changes in initial parameters or numerical (rounding) errors in intermediate values result in drastically different situations after a while. (This is known as the "butterfly effect", after the idea that the single flap of a butterfly wing could be enough to cause a tornado on the other side of Earth later on.) That is, *even if we understand how the system works and can predict its behaviour in the short run, it is impossible to make long-term predictions.*
- The system shows episodes of somewhat regular motion patterns, but then suddenly jumps to a different pattern (strange attractor).

Chaotic Systems

One requirement for systems (using a finite number of dimensions) to become chaotic is that the equations governing the system are non-linear. Many of our models of nature are non-linear. While not every non-linear system is necessarily chaotic, *chaotic systems are very common*. Examples include:

Weather

We can create rather detailed models for how factors such as temperature, humidity and pressure affect the motion of air masses, and subsequently our weather. However, weather is a chaotic system. Usually, we can predict the weather just a few days ahead, and even then the predictions often turn out to be wrong.

Solar System

Lesser known, the solar system is also chaotic. The gravitational interaction of planets and other objects has small effects in the short-term, but can amount to drastic differences long-term. We are not able to predict the position of planets more than a few million years into the future. In fact, we do not even know if the solar system is stable. For example, it cannot be ruled out that Earth may collide with another planet, or even be ejected from the solar system in the far future.

Roulette

The roulette game was designed to increase the difficulty of cheating (by rigging the equipment) when gambling. Although we could in principle calculate how the roulette ball moves using elementary mechanics, the outcome is impossible to predict if one gives the ball enough time to move and bounce off the obstacles along the rim of the wheel.

Biology

Biology is based on and affected by a large number of non-linear chemical and physical processes. Living things exhibit chaotic characteristics. For example, the heart beat rhythm or brain activity of humans show signs of chaos, and too regular activity can be a sign of disease.

Chaos may also help resolve the centuries-old philosophical arguments between determinism and free will: even if we assumed humans to be essentially copies of the same machine who start out similar and follow identical rules, individuals still could turn out very differently from each other and act in a rather unpredictable manner.