Computational Methods and Modelling

Antonio Attili

antonio.attili@ed.ac.uk

School of Engineering
University of Edinburgh
United Kingdom

Lecture 7
Systems of ODEs and Stiff Equations

THE UNIVERSITY of EDINBURGH

Systems of Equations

- A single ordinary differential equation (ODE) is usually written as:

$$\frac{dy}{dx} = f(x, y) \quad \text{with} \quad y_0 = y(x_0)$$

- However, most engineering applications require the solution of many coupled equations (from a handful to billions).
- The general form of a **system of coupled ODEs** is:

$$\frac{dy_1}{dx} = f_1(x, y_1, y_2, ..., y_n)$$
$$\frac{dy_2}{dx} = f_2(x, y_1, y_2, ..., y_n)$$
$$.$$
$$.$$
$$\frac{dy_n}{dx} = f_n(x, y_1, y_2, ..., y_n)$$

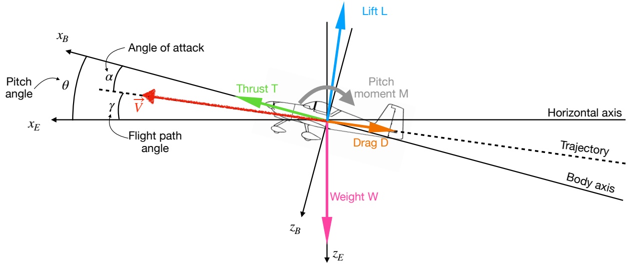
with the set of n initial conditions:

$$y_1^0 = y_1(x_0); \quad y_2^0 = y_2(x_0); \quad ... \quad ; y_n^0 = y_n(x_0)$$

- Or, in a more compact form:

$$\frac{dy_j}{dx} = f_j(x, y_j) \quad \text{with} \quad y_j^0 = y_j(x_0) \quad \text{and} \quad j = 1, ..., n$$

Example: Equations of Motion for an Airplane



Velocity equations

$$\frac{\mathrm{d}u_B}{\mathrm{d}t} = \frac{L}{m} sin\alpha - \frac{D}{m} cos\alpha - qw_B - \frac{W}{m} sin\theta + \frac{T}{m}$$

$$\frac{\mathrm{d}w_B}{\mathrm{d}t} = -\frac{L}{m} cos\alpha - \frac{D}{m} sin\alpha + qu_B + \frac{W}{m} cos\theta$$

Angular velocity and pitch angle equations

$$\frac{\mathrm{d}q}{\mathrm{d}t} = \frac{M}{I_{yy}}$$

$$\frac{\mathrm{d}\theta}{\mathrm{d}t} = q$$

Navigation equations

$$\frac{\mathrm{d}x_E}{\mathrm{d}t} = u_B cos\theta + w_B sin\theta$$

$$\frac{\mathrm{d}z_E}{\mathrm{d}t} = -u_B sin\theta + w_B cos\theta$$

Euler Method for a Systems of Equations

- For the system of ODEs:

$$\frac{dy_j}{dx} = f_j(x, y_j) \quad \text{with} \quad y_j^0 = y_j(x_0) \quad \text{and} \quad j = 1, \ldots, n$$

- We apply the Euler method to each of the equations:

$$y_j^{i+1} = y_j^i + \phi_j h$$

  where the n values ϕ_j are estimates of appropriate slopes for each of the functions y_j over the step h.

- For the Euler method the slopes ϕ_j are estimated as f_j, evaluated at the point i: $f_j(x^i, y_j^i)$.
- If we consider a system of 2 equations, the first iterations of the Euler method are:

$$\begin{aligned}
\text{First Euler step:} &\quad y_1^1 = y_1^0 + f_1(x^0, y_1^0, y_2^0)h \quad \text{and} \quad y_2^1 = y_2^0 + f_2(x^0, y_1^0, y_2^0)h \\
\text{Second Euler step:} &\quad y_1^2 = y_1^1 + f_1(x^1, y_1^1, y_2^1)h \quad \text{and} \quad y_2^2 = y_2^1 + f_2(x^1, y_1^1, y_2^1)h \\
\text{Third Euler step:} &\quad y_1^3 = y_1^2 + f_1(x^2, y_1^2, y_2^2)h \quad \text{and} \quad y_2^3 = y_2^2 + f_2(x^2, y_1^2, y_2^2)h
\end{aligned}$$

Euler Method for a System of Equation: python code

- Let's consider the following system of equations:

$$\frac{dy_1}{dx} = -0.5y_1 \quad \text{and} \quad \frac{dy_2}{dx} = 4 - 0.3y_2 - 0.1y_1 \quad \text{with} \quad y_1(0) = 4,\ y_2(0) = 6$$

Euler method for a system of equations: euler_system.py

```python
# importing modules
import numpy as np
import matplotlib.pyplot as plt
import math

# -----------------------------------------------------
# functions that returns dy/dx
# i.e. the equation we want to solve: dy_j/dx = f_j(x,y_j)
# (j=[1,2] in this case)
def model(x,y_1,y_2):
    f_1 = -0.5 * y_1
    f_2 = 4.0 - 0.3 * y_2 - 0.1 * y_1
    return [f_1 , f_2]
# -----------------------------------------------------

# -----------------------------------------------------
# initial conditions
x0 = 0
y0_1 = 4
y0_2 = 6
# total solution interval
x_final = 2
# step size
h = 0.5
# -----------------------------------------------------

# -----------------------------------------------------
# Euler method

# number of steps
n_step = math.ceil(x_final/h)

# Definition of arrays to store the solution
y_1_eul = np.zeros(n_step+1)
y_2_eul = np.zeros(n_step+1)
x_eul = np.zeros(n_step+1)

# Initialize first element of solution arrays
# with initial condition
y_1_eul[0] = y0_1
y_2_eul[0] = y0_2
x_eul[0]   = x0

# Populate the x array
for i in range(n_step):
    x_eul[i+1]  = x_eul[i]  + h

# Apply Euler method n_step times
for i in range(n_step):
    # compute the slope using the differential equation
    [slope_1 , slope_2] = model(x_eul[i],y_1_eul[i],y_2_eul[i])
    # use the Euler method
    y_1_eul[i+1] = y_1_eul[i] + h * slope_1
    y_2_eul[i+1] = y_2_eul[i] + h * slope_2
    print(y_1_eul[i],y_2_eul[i])
# -----------------------------------------------------
```


THE UNIVERSITY of EDINBURGH

Using python libraries to solve ODEs

- Python is used by a number of different communities, ranging from from fundamental physics and numerical analysis to business, machine learning, education, statistics.
- Each community has developed tools that are often available open source.

- Notable examples are

numpy	package for scientific computing
matplotlib	plotting library
math	mathematical functions (sin, cos, ...)
PyTorch	a deep learning framework
astropy	packages designed for use in astronomy
biopython	computational biology and bioinformatics
scipy	library for scientific computing (numpy and matplotlib are actually part of scipy)
scipy.integrate	library for numerical integration and solution ODEs
scipy.optimize	library for optimization and root finding

Solution of a system of ODEs with solve_ivp from scipy

scipy.integrate.solve_ivp for a system of equations: solve_ivp_system.py

```python
# importing modules
import numpy as np
import matplotlib.pyplot as plt
import math
from scipy.integrate import solve_ivp

# ----------------------------------------------------
# functions that returns dy/dx
# i.e. the equation we want to solve:
# dy_j/dx = f_j(x,y_j) (j=[1,2] in this case)
def model(x,y):
    y_1 = y[0]
    y_2 = y[1]
    f_1 = -0.5 * y_1
    f_2 = 4.0 - 0.3 * y_2 - 0.1 * y_1
    return [f_1 , f_2]
# ----------------------------------------------------

# ----------------------------------------------------
# initial conditions
x0 = 0
y0_1 = 4
y0_2 = 6
# total solution interval
x_final = 2
# step size
# not needed here. The solver solve_ivp
# will take care of finding the appropriate step
# ----------------------------------------------------
```

```python
# ----------------------------------------------------
# Apply solve_ivp method
y = solve_ivp(model, [0 , x_final] ,[y0_1 , y0_2])
# ----------------------------------------------------

# ----------------------------------------------------
# plot results
plt.plot(y.t,y.y[0,:] , 'b.-',y.t,y.y[1,:] , 'r-')
plt.xlabel('x')
plt.ylabel('y_1(x), y_2(x)')
plt.show()
# ----------------------------------------------------

# ----------------------------------------------------
# print results in a text file (for later use if needed)
file_name= 'output.dat'
f_io = open(file_name,'w')
n_step = len(y.t)
for i in range(n_step):
    s1 = str(i)
    s2 = str(y.t[i])
    s3 = str(y.y[0,i])
    s4 = str(y.y[1,i])
    s_tot = s1 + ' ' + s2 + ' ' + s3  + ' ' + s4
    f_io.write(s_tot + '\n')
f_io.close()
# ----------------------------------------------------
```

- The structured data y contains the solution: $y.t$ is the x coordinate, $y.y[0,:]$ and $y.y[1,:]$ are y_1 and y_2.

Reference guide for solve_ivp from scipy

docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.solve_ivp.html

SciPy.org

SciPy.org | Docs | SciPy v1.5.2 Reference Guide | Integration and ODEs (scipy.integrate) | index | modules | next | previous

scipy.integrate.solve_ivp

scipy.integrate.solve_ivp(*fun*, *t_span*, *y0*, *method='RK45'*, *t_eval=None*, *dense_output=False*, *events=None*, *vectorized=False*, *args=None*, ***options*) [source]

Solve an initial value problem for a system of ODEs.

This function numerically integrates a system of ordinary differential equations given an initial value:

```
dy / dt = f(t, y)
y(t0) = y0
```

Here t is a 1-D independent variable (time), y(t) is an N-D vector-valued function (state), and an N-D vector-valued function f(t, y) determines the differential equations. The goal is to find y(t) approximately satisfying the differential equations, given an initial value y(t0)=y0.

Some of the solvers support integration in the complex domain, but note that for stiff ODE solvers, the right-hand side must be complex-differentiable (satisfy Cauchy-Riemann equations [11]). To solve a problem in the complex domain, pass y0 with a complex data type. Another option always available is to rewrite your problem for real and imaginary parts separately.

Parameters: **fun** : ***callable***

Right-hand side of the system. The calling signature is `fun(t, y)`. Here *t* is a scalar, and there are two options for the ndarray *y*: It can either have shape (n,); then *fun* must return array_like with shape (n,). Alternatively, it can have shape (n, k); then *fun* must return an array_like with shape (n, k), i.e., each column corresponds to a single column in *y*. The choice between the two options is determined by *vectorized* argument (see below). The vectorized implementation allows a faster approximation of the Jacobian by finite differences (required for stiff solvers).

t_span : ***2-tuple of floats***

Interval of integration (t0, tf). The solver starts with t=t0 and integrates until it reaches t=tf.

y0 : ***array_like, shape (n,)***

Initial state. For problems in the complex domain, pass *y0* with a complex data type (even if the initial value is purely real).

method : ***string or OdeSolver, optional***

Integration method to use:

- 'RK45' (default): Explicit Runge-Kutta method of order 5(4) [1]. The error is controlled assuming accuracy of the fourth-order method, but steps are taken using the fifth-order accurate formula (local extrapolation is done). A quartic interpolation polynomial is used for the dense output [2]. Can be applied in the complex domain.
- 'RK23': Explicit Runge-Kutta method of order 3(2) [3]. The error is controlled assuming accuracy of the second-order method, but steps are taken using the third-order accurate formula (local extrapolation is done). A cubic Hermite polynomial is used for the dense output. Can be applied in the complex domain.
- 'DOP853': Explicit Runge-Kutta method of order 8 [13]. Python implementation of the "DOP853" al-

Previous topic

scipy.integrate.romb

Next topic

scipy.integrate.RK23

Quick search

search

Reference guide for solve_ivp from scipy

`docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.solve_ivp.html`

The minimum allowed step size for 'LSODA' method. By default *min_step* is zero.

Returns: **Bunch object with the following fields defined:**

t : *ndarray, shape (n_points,)*
Time points.

y : *ndarray, shape (n, n_points)*
Values of the solution at *t*.

sol : *OdeSolution or None*
Found solution as **OdeSolution** instance; None if *dense_output* was set to False.

t_events : *list of ndarray or None*
Contains for each event type a list of arrays at which an event of that type event was detected. None if *events* was None.

y_events : *list of ndarray or None*
For each value of *t_events*, the corresponding value of the solution. None if *events* was None.

nfev : *int*
Number of evaluations of the right-hand side.

njev : *int*
Number of evaluations of the Jacobian.

nlu : *int*
Number of LU decompositions.

status : *int*
Reason for algorithm termination:

- -1: Integration step failed.
- 0: The solver successfully reached the end of *tspan*.
- 1: A termination event occurred.

message : *string*
Human-readable description of the termination reason.

success : *bool*
True if the solver reached the interval end or a termination event occurred (`status >= 0`).

Stability of the ODE Solution Methods

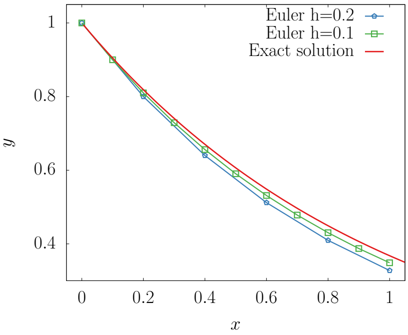
- Let's consider again the ODE (already discussed in one of the previous lectures):

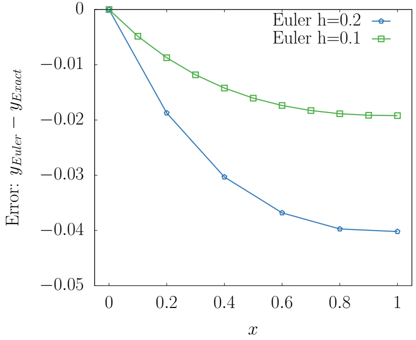
$$\frac{dy}{dx} = -y \quad \text{with} \quad y(x=0) = 1$$

- which has the analytical (exact) solution

$$y(x) = e^{-x}$$

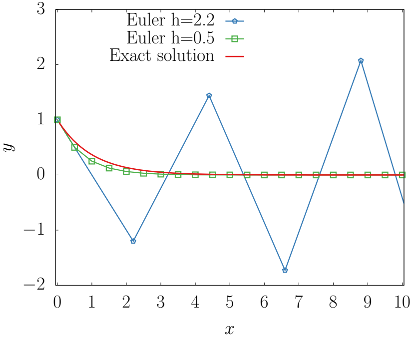
- As already shown in the previous lecture, plotting the two solutions in the interval $0 \le x \le 1$ with different steps $h = 0.2$ and $h = 0.1$ we obtain:

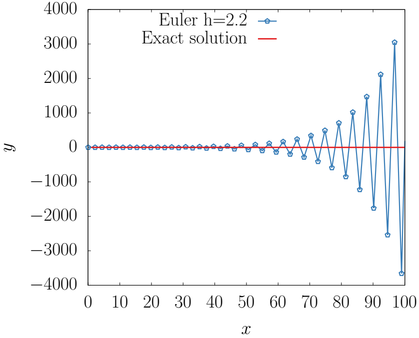




Stability of the ODE Solution Methods

- Now, we try to solve the same equation with a very large step.
- In addition, we solve the equation in a large x interval to highlight the undesirable effect of a large step.



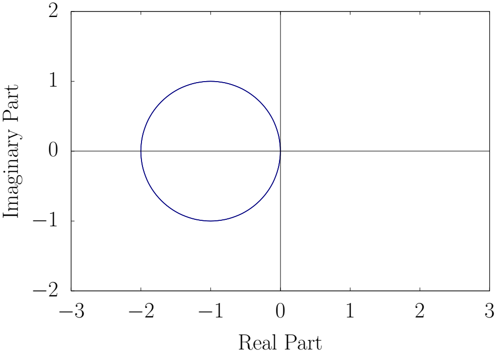


- For large steps ($h = 0.5$), the solution first become rather inaccurate, but still qualitative correct (the trend is captured)
- For even larger steps ($h = 2.2$), the numerical solution is completely wrong and **diverges** for large intervals (large values of x). In this case, we say that the numerical method is **unstable**.

Stability of the ODE Solution Methods

- For the simple **linear** equation $dy/dx = -ky$ (where k is a positive real number), it can be shown that the Euler method is **stable** if the product $-kh$ is inside the region in the complex plane:

$$|z + 1| \leq 1$$



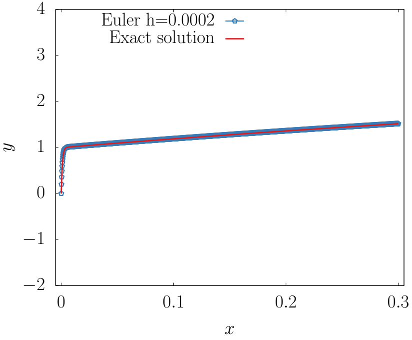
- In our previous example, we had a real equation (no imaginary part) so the condition becomes simply $-2 \leq -kh \leq 0$.
- Therefore, since $k = 1$ the case with $h = 2.2$ was unstable.

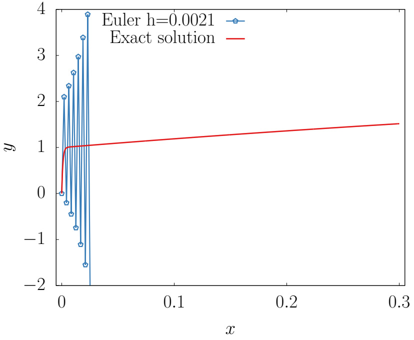
Stiff ODEs

- ODE or a system of ODEs where fast and slow components exist.
  - Slow component: we need to solve the equation over a large interval.
  - Fast component: we usually need a small step h to capture the fast component.
  - Long interval with small steps means a **large number of steps**.
- An example of a stiff equation is:

$$\frac{dy}{dx} = -1000y + 3000 - 2000e^{-x} \quad \text{with i.c.} \quad y(x=0) = 0$$

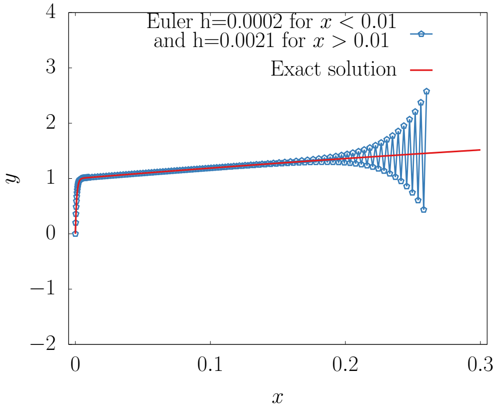
which has the analytical solution $y(x) = 3 - 0.998e^{-1000x} - 2.002e^{-x}$





Stiff ODEs

- Often, the fast component is localized in a small interval and we are not interested in it. However, we still need to capture it to avoid numerical instability.
- A tempting (and naive) strategy would be to use a small step h during the initial fast transient and then use a larger step later.



- However, as shown in the figure, the fast component still causes instability even if the solution is not actually changing as fast as in the initial transient.
- Conclusion: we need a small step for the entire solution interval.

Unconditionally stable methods: Implicit Euler

- Let's consider the usual equation:

$$\frac{dy}{dx} = f(x, y) \quad \text{with} \quad y_0 = y(x_0)$$

As shown in the previous lectures, the explicit Euler method is:

$$y_{i+1} = y_i + f(x_i, y_i)h$$

where the slope is approximated with the derivative in the original point x_i, where the solution and its derivatives are know.

- Implicit methods employ information at locations that have not been computed yet.
- For the implicit Euler method, or backward Euler, we use the derivative in the point x_{i+1} to estimate the slope

$$y_{i+1} = y_i + f(x_{i+1}, y_{i+1})h$$

- This is called implicit, because the unknown y_{i+1} appears on both sides of the formula.

- To compute y_{i+1} we need to invert this formula:
  - Analytically or with a root finding method, if the function $f(x, y)$ is linear.
  - Necessarily with a root finding method, if the function $f(x, y)$ is non-linear.
- In other words, we have to find the root of the function $F(y_{i+1})$:

$$F(y_{i+1}) = y_i + f(x_{i+1}, y_{i+1})h - y_{i+1} = 0$$

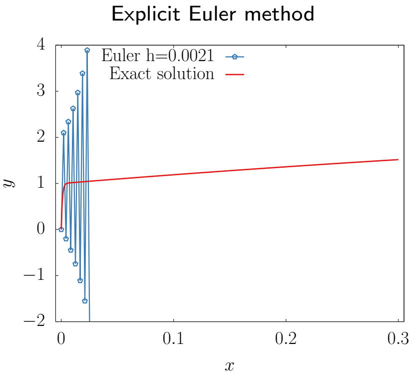
(the value of y_{i+1} where $F(y_{i+1}) = 0$).

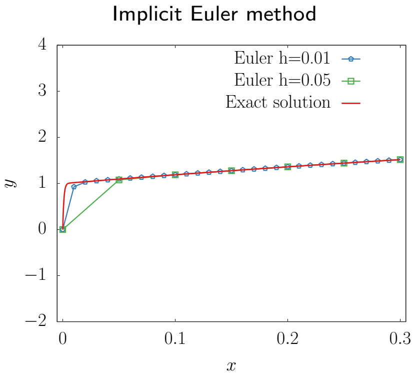
Solution of a stiff ODE with the Implicit Euler Method

- Let's consider again the stiff ODE:

$$\frac{dy}{dx} = -1000y + 3000 - 2000e^{-x} \quad \text{with i.c.} \quad y(x=0) = 0$$

and solve it with the explicit and implicit Euler methods.





- As we have seen before, the explicit method is unstable (it diverges) already for $h = 0.0021$.
- The implicit method is stable (it does not diverge), even with the very large step $h = 0.05$

Python code, Implicit Euler method

Implicit Euler method: euler_implicit.py

```python
# importing modules
import numpy as np
import matplotlib.pyplot as plt
import math

# ------------------------------------------------------
# inputs
# functions that returns dy/dx
# i.e. the equation we want to solve: dy/dx = - y
def model(y,x):
    dydx = -1000.0*y + 3000.0 - 2000.0*math.exp(-x)
    return dydx

# initial conditions
x0 = 0
y0 = 0
# total solution interval
x_final = 0.3
# step size
h = 0.05

# ------------------------------------------------------
# Secant method (a very compact version)
def secant_2(f, a, b, iterations):
    for i in range(iterations):
        c = a - f(a)*(b - a)/(f(b) - f(a))
        if abs(f(c)) < 1e-13:
            return c
        a = b
        b = c
    return c

# ------------------------------------------------------
# Euler implicit method


# number of steps
n_step = math.ceil(x_final/h)

# Definition of arrays to store the solution
y_eul = np.zeros(n_step+1)
x_eul = np.zeros(n_step+1)

# Initialize first element of solution arrays
# with initial condition
y_eul[0] = y0
x_eul[0] = x0

# Populate the x array
for i in range(n_step):
    x_eul[i+1]  = x_eul[i]  + h

# Apply implicit Euler method n_step times
for i in range(n_step):
    F = lambda y_i_plus_1: y_eul[i] + \
            model(y_i_plus_1,x_eul[i+1])*h - y_i_plus_1
    y_eul[i+1] = secant_2(F, \
            y_eul[i],1.1*y_eul[i]+10**-3,10)


# ------------------------------------------------------
```

Python code, Explicit vs Implicit Euler

- For the equation:

$$\frac{dy}{dx} = f(x, y) \quad \text{with} \quad y_0 = y(x_0)$$

  The implicit Euler method is:

$$y_{i+1} = y_i + f(x_{i+1}, y_{i+1})h$$

- To find the solution y_{i+1} in x_{i+1} we hve to find the root of the function $F(y_{i+1})$:

$$F(y_{i+1}) = y_i + f(x_{i+1}, y_{i+1})h - y_{i+1} = 0$$

Implicit Euler: euler_implicit.py

```
# ------------------------------------------------------
# functions that returns dy/dx
# i.e. the equation we want to solve: dy/dx = - y
def model(y,x):
    dydx = -1000.0*y + 3000.0 - 2000.0*math.exp(-x)
    return dydx

# <MISSING CODE HERE>

# ------------------------------------------------------
# Secant method (a very compact version)
def secant_2(f, a, b, iterations):
    for i in range(iterations):
        c = a - f(a)*(b - a)/(f(b) - f(a))
        if abs(f(c)) < 1e-13:
            return c
        a = b
        b = c
    return c

# ------------------------------------------------------
# Euler implicit method

# <MISSING CODE HERE>

# Apply implicit Euler method n_step times
for i in range(n_step):
    F = lambda y_i_plus_1: y_eul[i] + \
            model(y_i_plus_1,x_eul[i+1])*h - y_i_plus_1
    y_eul[i+1] = secant_2(F, \
            y_eul[i],1.1*y_eul[i]+10**-3,10)
# ------------------------------------------------------
```

Python code, Explicit vs Implicit Euler

Explicit Euler: euler.py

```
# --------------------------------------------------------
# Euler method

# Apply Euler method n_step times
for i in range(n_step):
    # compute the slope using the differential equation
    slope = model(y_eul[i],x_eul[i])
    # use the Euler method
    y_eul[i+1] = y_eul[i] + h * slope
# --------------------------------------------------------
```

Implicit Euler: euler_implicit.py

```
# --------------------------------------------------------
# Secant method (a very compact version)
def secant_2(f, a, b, iterations):
    for i in range(iterations):
        c = a - f(a)*(b - a)/(f(b) - f(a))
        if abs(f(c)) < 1e-13:
            return c
        a = b
        b = c
    return c

# --------------------------------------------------------
# Euler implicit method

# Apply implicit Euler method n_step times
for i in range(n_step):
    F = lambda y_i_plus_1: y_eul[i] + \
            model(y_i_plus_1,x_eul[i+1])*h - y_i_plus_1
    y_eul[i+1] = secant_2(F, \
            y_eul[i],1.1*y_eul[i]+10**-3,10)


# --------------------------------------------------------
```

- In comparison with the explicit method, the implicit Euler requires the solution of the (in general non-linear) equation $F(y_{i+1}) = 0$
- This requires a root finding method, for example the secant method in this case.

Python code, Implicit Euler method with function import

- Often, it is useful to reuse some functions in multiple codes.
- In this case it is conveneient to save the function in a separate file and import this file in our main code.
- For example we can store the `def secant_2` function in the file secant_function.py, importing it with `import secant_function` and use it with `secant_function.secant_2`.

Secant method: secant_function.py

```python
# importing modules
import numpy as np
import matplotlib.pyplot as plt
import math

# ------------------------------------------------------
# Secant method (a very compact version)
def secant_2(f, a, b, iterations):
    for i in range(iterations):
        c = a - f(a)*(b - a)/(f(b) - f(a))
        if abs(f(c)) < 1e-13:
            return c
        a = b
        b = c
    return c
# ------------------------------------------------------
```

Implicit Euler method: euler_implicit_import_fun.py

```python
# importing modules
import numpy as np
import matplotlib.pyplot as plt
import math

# importing our own module
import secant_function


# <MISSING CODE HERE>

# ------------------------------------------------------
# Apply implicit Euler method n_step times
for i in range(n_step):
    F = lambda y_i_plus_1: y_eul[i] + \
            model(y_i_plus_1,x_eul[i+1])*h - y_i_plus_1
    y_eul[i+1] = secant_function.secant_2(F, \
            y_eul[i],1.1*y_eul[i]+10**-3,10)
# ------------------------------------------------------
```