02.ecosystem

March 21, 2018

1 Ecosystem

1.1 Modules

- Different way for import
- Full package in it's own namespace:

```
import package
package.function()
# or
import package as pkg
pgk.function()
```

- Only function in package:

```
from package import function
function()
# or
from package import function as func
func()
```

- Full package in the current namespace: **TO BE AVOIDED**

```
from package import *
function()
```

1.2 Standard library

- builtins : automatically imported, contains basis (`int`, `print()`, ...)
- sys et os : two common modules for read and manage code's environment
- shutil : file managment (copy, move, ...)
- math / cmath : classic mathematical functions (real / complex)
- copy : particulary deepcopy
- pathlib : dedicated to pathfile managment
  (`directory / file` better than `directory + "/" + fichier`)
- time
- ...

1.3 Your modules

- `import file` in order to use content of file.py
- if subfolders:
  - `__init__`.py mandatory in dir (may be empty)
  - `import dir.file`in order to use content of dir/file.py
- import involve execution!

  ```
  if __name__ == '__main__':
  # do stuff
  ...
  ```

- can be imported:
- each module in sys.path (feeded par PYTHONPATH)
- each submodule (with spam.egg; import spam then egg in the directory spam)

  It is useless to import a module multiple times (but it does not harm either)

```
In [1]: from sys import path
        print(path)
```

```
['', '/usr/lib/python35.zip', '/usr/lib/python3.5', '/usr/lib/python3.5/plat-x86_64-linux-gnu',
```

1.4 How to install external packages

- Windows
- conda install package
- pip install package
- Linux et Mac
- pip install --user package

1.5 Numpy

Reference
Comparison from matlab

- `dtype` : numerical precision (double by default)
- `ndarray` : an array of a unique dtype unique **much faster** than the python lists
- `shape`
- **mutable**
- same indexing and slicing as with lists, plus:
- a list of integers per dimensions
- an array of index
- an array of boolean (mask)
- creation :

- `empty` : fast allocation but without initialization
- `zeros`, `zeros_like`, `ones`, `ones_like` : allocataion and initialization with 0 or 1
- `arange(start, end, step)`
- `linspace(start, end, nb of points)`
- `meshgrid` : mesh
- matrix product : `@`
- **A lot** of packages use this data structure
- Some usefull functions
- `linalg.eigvals` : eigen values
- `linalg.det` : determinent
- `linalg.solve` : solve Ax = b
- `linalg.inv` : inverse a matrix
- `sort` : sort
- `where` : index where a mask is True
- `median`, `average`, `std`,`var` : classic statistical tools
- `cov` : covariant matrix
- `histogram` : histogram

1.6 scipy

Reference

- Many usefull constants
- **A lot** of usefull functions
- `fft`, `dct`, `dst` : fourrier transform
- `quad`, `simps`, `odr` : integration
- `solve_ivp`, `solve_bvp` : ODE solver
- `griddata`, `make_interp_spline` : interpolation
- `solve`, `inv`, `det`, `eigvals` : linalg variant
- `lu`, `svd` : more linalg
- `convolve`, `correlate`, `gaussian_filter`,`spline_filter` : filters
- `binary_closing`, `binary_dilatation`, `binary_erosion` : morphology
- `minimize`, `leastsq`, `root`, `fsolve` : optimization and zeros finding
- `find_peaks_cwt`, `spectrogram` : signal processing
- `lu`, `csr` : sparse matrices
- `bicg`, `gmres`, `splu` : sparse linalg
- `shortest_path` : graph computation
- `KDTree`, `Delaunay`, spatial computation
- `gauss`, `laplace`, `uniform`, `binom` : random distributions
- `describe`, `bayes_mvs` : more statistics
- `airy`, `jv`, `erf`, `fresnel` : classical functions

1.7 Matpotlib

Gallery - Almost anything you need to vizualize something - 1d / 2d / 3d - fixed ou animated - static or interactive (but static is easier) - Very close to the matlab syntax (too close ?) - A lot of functionnalities (too much ?) - Might be slow (does not always replace paraview / visit)

1.8 Some exemples

In [2]:
```
import numpy as np

def f(t):
    """A simple function"""
    return np.exp(-t) * np.cos(2 * np.pi * t)

# Some time steps
t = np.arange(0.0, 5.0, 0.1)

# apply f() on all the values in t, and name the result y
y = f(t)

print("Numpy arrays are capable of some basic computation:")
print(("{:6} = {:5.2f}\n" * 3).format("y min", y.min(),
                                      "y mean", y.mean(),
                                      "y max", y.max()))
```

```
Numpy arrays are capable of some basic computation:
y min  = -0.61
y mean =  0.02
y max  =  1.00
```

In [3]:
```
print("Or we can use numpy's functions on those arrays (see reference):")
print(("{:6} = {:5.2f}\n" * 3).format("y min", np.min(y),
                                      "y mean", np.mean(y),
                                      "y max", np.max(y)))
```

```
Or we can use numpy's functions on those arrays (see reference):
y min  = -0.61
y mean =  0.02
y max  =  1.00
```

In [4]:
```
%matplotlib notebook

# import can be done at any time,
# but best practice tells us to put all of them at the top of the file
import matplotlib.pyplot as plt

# The simplest plot
plt.figure()
plt.plot(t, f(t))
plt.show()
```

```
# A better plot
fig= plt.figure()
ax = plt.subplot(111)
plt.plot(t, f(t), "-o", label=f.__doc__) # we can access the docstring of f
ax.set_xlabel("Time (s)")
ax.set_ylabel("Function")
plt.suptitle("Function over Time", fontsize=16)
plt.legend()
plt.show()
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

1.9 Some interactivity is possible (not easy)

```
In [5]: from Picker import Picker

        # create random data
        data = np.random.rand(5,2,100)

        #For jupyter, keep this object in memory
        picker = Picker()

        # let the user pick one point per image (the first dim of data)
        result = picker.pick(data)

        ??Picker
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

```
In [6]: print(result)
```

```
[[0.2884752  0.65146457]
 [       nan        nan]
 [       nan        nan]
 [0.73088556 0.79146041]
 [       nan        nan]]
```

1.10 First exercise : statistics

```
In [7]: # Load more data
        data = np.loadtxt('data/populations.txt').astype(np.int)
        print(data)
```

```
[[ 1900 30000  4000 48300]
 [ 1901 47200  6100 48200]
 [ 1902 70200  9800 41500]
 [ 1903 77400 35200 38200]
 [ 1904 36300 59400 40600]
 [ 1905 20600 41700 39800]
 [ 1906 18100 19000 38600]
 [ 1907 21400 13000 42300]
 [ 1908 22000  8300 44500]
 [ 1909 25400  9100 42100]
 [ 1910 27100  7400 46000]
 [ 1911 40300  8000 46800]
 [ 1912 57000 12300 43800]
 [ 1913 76600 19500 40900]
 [ 1914 52300 45700 39400]
 [ 1915 19500 51100 39000]
 [ 1916 11200 29700 36700]
 [ 1917  7600 15800 41800]
 [ 1918 14600  9700 43300]
 [ 1919 16200 10100 41300]
 [ 1920 24700  8600 47300]]
```

```
In [8]: import numpy as np

        # Extract each collumn into a specific vector

        # Copy the data array without the year collumn

        # Compute for each species
        # - the mean population
        # - the standard deviation
        # - The year when this species had it's maximum of population
        # - The two years when this species had it's minimum of population

        # Print all that in an table
```

1.10.1 Solution

```
import numpy as np

# Extract each collumn into a specific vector
years, hares, lynxes, carrots = data.T                          # unpacking
```

```
# Copy the data array without the year collumn
populations = data[:,1:]                                        # slicing

# Compute for each species
# - the mean population
means           = populations.mean(axis=0)

# - the standard deviation
stds            = populations.std(axis=0)

# - The year when this species had it's maximum of population
max_populations = np.argmax(populations, axis=0)                # use of axis
max_years       = years[max_populations]                        # slicing with a vector

# - The two years when this species had it's minimum of population
lowest_2_pop    = np.argsort(populations, axis=0)[:2]
lowest_2_year   = years[lowest_2_pop]                           # slicing with a matrix

# Print all that in an table

format_title = "{:<30} {:^10} | {:^10} | {:^10}"
format_array = "{:<30} {:^10.3e} | {:^10.3e} | {:^10.3e}"
print(format_title.format("", "Hares", "Lynxes", "Carrots"))
print(format_array.format("Mean:", *means))                     # unpacking (rare)
print(format_array.format("Std:", *stds))
print(format_array.format("Max. year:",*max_years))
print(format_array.format("lowest populations year 1:", *lowest_2_year[0]))
print(format_array.format("lowest populations year 2:", *lowest_2_year[1]))
```

```
In [9]: # All years when at least one specie was above 50000
```

1.10.2 Solution

```
above_50000 = np.any(populations > 50000, axis=1)
print("Any above 50000:", years[above_50000])             # slicing with a mask
```

```
In [10]: # Dominants species throught the years
```

1.10.3 Solution

```
max_species = np.argmax(populations, axis=1)
species = np.array(['Hare', 'Lynx', 'Carrot'])

max_species=np.stack((years,species[max_species]), axis=1)
print("Max species:")
print(max_species)
```

```
In [11]: # Compute the correlation coeficient between the variation of hares and lynxes
```

1.10.4 Solution

```
hare_grad = np.gradient(hares, 1.0)

plt.figure()
plt.plot(years, hare_grad)
plt.plot(years, -lynxes)
plt.show()

# Compute the correlation coeficient between the variation of hares and lynxes
print("diff(Hares) vs. Lynxes correlation = ", np.corrcoef(hare_grad, -lynxes)[0, 1])
```

1.11 Alternative : Panda

- Excel-like processing

```
In [12]: import pandas as pd
         # Load data
         data = pd.read_csv('data/populations.txt', sep="\t", dtype=np.int, index_col=0)
         # fix the year title
         data.index.names = ['year']
         data.style

Out[12]: <pandas.io.formats.style.Styler at 0x7f1745161e80>

In [13]: data.describe().style.format("{:.0f}")

Out[13]: <pandas.io.formats.style.Styler at 0x7f174be71898>

In [14]: # Compute for each species
         # - the mean population
         means = data.mean()
         means.name = "Mean"

         # - the standard deviation
         stds           = data.std()
         stds.name = "Std"

         # - The year when this species had it's maximum of population
         max_populations = data.idxmax()
         max_populations.name = "Max. year"

         # - The two years when this species had it's minimum of population
         lowest_2_pop    = data.values.argsort(axis=0)[:2]
         lowest_2_year   = data.index[lowest_2_pop].values

         lowest_2_year = pd.DataFrame(lowest_2_year,
                                      columns=data.columns,
                                      index=["Lowest populations year 1", "Lowest populations ye
```

```
# Print all that in an table
result_table = pd.concat([means, stds, max_populations, lowest_2_year.T], axis=1).T
result_table.style
```

Out[14]: <pandas.io.formats.style.Styler at 0x7f173c12bda0>

In [15]:
```
# All years when at least one specie was above 50000
above_50000 = (data > 50000).any(axis=1)
print("Any above 50000:", data.index[above_50000].values)

# Dominants species throught the years

print("Max species:")
print(data.idxmax(axis=1))

data["hare_grad"] = np.gradient(data["hare"], data.index.to_series())

plt.figure()
plt.plot(data["hare_grad"])
plt.plot(-data["lynx"])
plt.legend()
plt.show()

# Compute the correlation coeficient between the variation of hares and lynxes
print("Correlation")
print(data.corr())
```

```
Any above 50000: [1902 1903 1904 1912 1913 1914 1915]
Max species:
year
1900    carrot
1901    carrot
1902      hare
1903      hare
1904      lynx
1905      lynx
1906    carrot
1907    carrot
1908    carrot
1909    carrot
1910    carrot
1911    carrot
1912      hare
1913      hare
1914      hare
1915      lynx
1916    carrot
```

```
1917    carrot
1918    carrot
1919    carrot
1920    carrot
dtype: object
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

```
Correlation
               hare      lynx    carrot  hare_grad
hare       1.000000  0.071892 -0.016604  -0.006390
lynx       0.071892  1.000000 -0.680577  -0.917925
carrot    -0.016604 -0.680577  1.000000   0.718199
hare_grad -0.006390 -0.917925  0.718199   1.000000
```

2 Image processing

2.0.1 pillow

For reading and writing images in a lot of format (tif, jpeg, ...)

2.0.2 Opencv

For image processing but usually scikit-image is better (but opencv is more complete)

2.0.3 scikits

'Plugins' for scipy http://scikits.appspot.com/scikits e.g. scikit-image

2.1 Exemple

In [16]:
```
%matplotlib notebook

from PIL import Image
waldo = Image.open("data/waldo.jpg")
waldo = np.array(waldo)

print(waldo.shape, waldo.dtype)

plt.figure()
plt.imshow(waldo)
plt.show()
```

```
(917, 1240, 3) uint8
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

```
In [17]: %matplotlib notebook

         # Make a copy of the image
         solution = waldo.copy()

         # An image is a 2d Array (3d with for colored image)
         # I need indices of pixels (x and y)
         sy, sx, sc = waldo.shape
         y, x, c = np.mgrid[0:sy, 0:sx, 0:sc]
         # Approximate position of waldo
         centerx, centery = (810, 870)
         size = 50

         # Darken everything except a circle around the waldo
         mask = ((y - centery) ** 2 + (x - centerx) ** 2) > size ** 2
         solution[mask] //= 2

         # Plot the solution
         plt.figure()
         plt.imshow(solution)
         plt.show()
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

2.2 Exercice

```
In [18]: %matplotlib notebook
         from scipy import ndimage as ndi
         from skimage import data
         from skimage.color import label2rgb

         # get data
         coins = data.coins()

         def plot_result(segmentation, title=""):
             # Select a different color for each coin
```

```python
    labeled_coins, _ = ndi.label(segmentation)
    image_label_overlay = label2rgb(labeled_coins, image=coins)

    fig = plt.figure()
    # The coins
    plt.imshow(coins, cmap=plt.cm.gray)
    # The edge of the segmentation
    plt.contour(segmentation, [0.5], linewidths=1.2, colors='y')
    # Each detected coin in a different color
    plt.imshow(image_label_overlay)

    if title:
        plt.suptitle(title, fontsize=16)
    plt.show()


def segmentation_with_threshold(datain, threshold):
    """ simple segmentation based on a threshold """
    coins = datain > threshold
    # binary_fill_holes is used to remove holes in the coins
    return ndi.binary_fill_holes(coins)


# Two plot side by side
# plot coins on the left one
fig, axes = plt.subplots(1,2)
axes[0].imshow(coins, cmap='gray')
# plot the histogram on the right
axes[1].hist(coins.flatten(), bins=np.arange(0, 256))
plt.show()

# 3D plot

from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm

sx,sy = coins.shape
x, y = np.mgrid[0:sx, 0:sy]

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

ax.plot_surface(x, y, coins, cmap=cm.coolwarm)
plt.show()
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

```
In [19]: # simple extraction based on 3 different threshold
         for threshold in [100,140,110]:
             segmentation = segmentation_with_threshold(coins, threshold)
             plot_result(segmentation, "Threshold method with {}".format(threshold))

         # We keep the 110
         segmentation_thre = segmentation_with_threshold(coins, 110)
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

```
In [20]: %matplotlib notebook

         from skimage.feature import canny

         def segmentation_with_edges(datain):
             """
             Segmentation based on edges
             1 - Apply canny edge filter
             2 - binary_fill_holes
             """
             dataout = datain > 110 # TODO replace this line
             return dataout


         segmentation_edges = segmentation_with_edges(coins)
         plot_result(segmentation_edges, "Edges method")
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

2.2.1 Solution

```python
def segmentation_with_edges(datain):
    """
    Segmentation based on edges
    1 - Apply canny edge filter
    2 - binary_fill_holes
    """
    return ndi.binary_fill_holes(canny(datain))
```

```python
In [21]: %matplotlib notebook

         from skimage.filters import sobel
         from skimage.morphology import watershed

         def segmentation_with_region(datain):
             """
             Segmentation based on region
             1 - use sobel to compute an elevation map
             2 - mark pixels
                 1 for background for sure
                 2 for coins for sure
                 0 for we don't know
             3 - apply watershed
             4 - binary_fill_holes
             """
             dataout = datain > 110 # TODO replace this line
             return dataout

         segmentation_region = segmentation_with_region(coins)
         plot_result(segmentation_region, "Region method")
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

2.2.2 Solution

```python
def segmentation_with_region(datain):
    """
    Segmentation based on region
```

```
    1 - use sobel to compute an elevation map
    2 - mak pixels
        1 for background for sure
        2 for coins for sure
        0 for we don't know
    3 - apply watershed
    4 - binary_fill_holes
    """

    # use sobel in order to compute elevation map
    elevation_map = sobel(datain)

    # mark pixels
    # 1 for background for sure
    # 2 for coins for sure
    # 0 for we don't know
    markers = np.zeros_like(datain)
    markers[datain < 30] = 1
    markers[datain > 150] = 2

    # apply watershed algo
    segmentation = watershed(elevation_map, markers) - 1
    return ndi.binary_fill_holes(segmentation)
```

In [22]:
```
%matplotlib notebook

def filter_small_objects(datain, threshold, biggest=False):
    """
    Segmentation based on edges
    1 - use label to labelize each zone
    2 - use bincount to count the number of pixel of each zone
    3 - select which zone to keep after filter
          - based on threshold
          - except background
          - eventually except biggest
    4 - apply filter
    """
    dataout = datain # TODO replace this line
    return dataout

segmentation_thre_filt = filter_small_objects(segmentation_thre,50, biggest=True)
plot_result(segmentation_thre_filt, "Threshold method at 110 filtered")
segmentation_edges_filt = filter_small_objects(segmentation_edges, 20)
plot_result(segmentation_edges_filt, "Edges method filtered")
segmentation_region_filt = filter_small_objects(segmentation_region, 20)
plot_result(segmentation_region_filt, "Region method filtered")
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

2.2.3 Solution

```
def filter_small_objects(datain, threshold, biggest=False):
    """
    Segmentation based on edges
    1 - use label to labelize each zone
    2 - use bincount to count the number of pixel of each zone
    3 - select which zone to keep after filter
         - based on threshold
         - except background
         - eventually except biggest
    4 - apply filter
    """
    # label each zone
    label_objects, nb_labels = ndi.label(datain)
    # size of each zone in pixel
    sizes = np.bincount(label_objects.flatten())
    # Only keep all zone bigger than threshold
    mask_sizes = sizes > threshold

    # remove the background
    mask_sizes[0] = False

    if biggest:
        # remove the biggest zone (except background)
        mask_sizes[np.argmax(sizes) + 1] = False

    #apply filter
    return mask_sizes[label_objects]
```

```
In [23]: def filled(datain):
             """
```

```
        Try harder to fill holes
        1 - dilation
        2 - binary_fill_holes
        3 - 2*erosion
        4 - dilation
        """
        dataout = datain # TODO replace this line
        return dataout

    segmentation_edges_fill = filled(segmentation_edges)
    plot_result(segmentation_edges_fill, "Edges method filled")
    segmentation_edges_fill_filt = filter_small_objects(segmentation_edges_fill, 20)
    plot_result(segmentation_edges_fill_filt, "Edges method filled and filtered")
```

```
<IPython.core.display.Javascript object>


<IPython.core.display.HTML object>


<IPython.core.display.Javascript object>


<IPython.core.display.HTML object>
```

2.2.4 Solution

```
def filled(datain):
    """
    Try harder to fill holes
    1 - dilation
    2 - binary_fill_holes
    3 - 2*erosion
    4 - dilation
    """
    segmentation_dil = ndi.morphology.binary_dilation(datain, iterations=1)
    segmentation_filled = ndi.binary_fill_holes(segmentation_dil)
    segmentation_ero = ndi.morphology.binary_erosion(segmentation_filled, iterations=2)
    segmentation_final = ndi.morphology.binary_dilation(segmentation_ero, iterations=1)
    return segmentation_final
```

3 signal processing

3.1 Exercice

```
In [24]: %matplotlib notebook
         import matplotlib.pyplot as plt
         import peakutils
```

```python
import numpy as np

from scipy.signal import wiener, argrelmin, argrelmax
from scipy.optimize import minimize

def filter(wavelength, signal):
    """
    Filter the signal in order to ease extraction of maxima

    scipy.signal.wiener is a first step
    """

    filtered_signal = signal
    return filtered_signal

def find_peaks(wavelength, signal):
    """
    Find some maxima of the signal

    Finding the maximum of the signal is also a first step
    scipy.signal.argrelmax is also a good idea

    indexes_max must be a list (eventually containing ony one element)
    """
    indexes_max = [256]
    return indexes_max



def get_windows(wavelength, signal, indexes_max):
    """
    Find out window of interest around peaks

    The window of interest is the time range between two local minima around the peak

    In order to find it:
    - use scipy.signal.argrelmin to extract all local minimas
    - in a loop for all peak:
      - use numpy.argmax to find the next local minima after the current peak
      - compute the edges of the window, beware of the boundaries
    """

    windows = {}
    for peak in indexes_max:
        windows[peak] = [200, 300]

    return windows
```

```
def filter_peaks(windows):
    """
    Filter out unusable peaks
    A peak is unusable when the window is too short (less thant 5 values)

    Create a new dictionnary containing only usable peaks
    """
    filtered_windows = {}
    for peak, (wmin, wmax) in windows.items():
        if wmax- wmin >= 5:
            filtered_windows[peak] = [wmin, wmax]
    return filtered_windows

def get_minmaxpeaks(windows):
    """
    Extract a list of all extrema that we will use
    """
    limits = np.asarray(list(windows.values())).flatten()
    maxpeaks = list(windows.keys())
    return limits, maxpeaks


# get data
wavelength, signal = np.loadtxt("data/Tungsten spectrum.dat").T

# filter signal
filtered_signal = filter(wavelength, signal)

# Get peaks
maximas = find_peaks(wavelength, filtered_signal)
windows = get_windows(wavelength, filtered_signal, maximas)
peaks = filter_peaks(windows)

selected_limits, selected_maxpeaks = get_minmaxpeaks(peaks)

# start plot
fig= plt.figure()
ax = plt.subplot(111)

# plot the signal
plt.plot(wavelength, signal)
#plt.plot(wavelength, filtered_signal)

# add a circle on all the peaks
if True:
    plt.plot(wavelength[selected_maxpeaks],
             filtered_signal[selected_maxpeaks],
             'o', ms=10, alpha=0.7, label="peaks")
```

```
        # add a circle on the limit of the windows
        if True:
            plt.plot(wavelength[selected_limits],
                     filtered_signal[selected_limits],
                     'o', ms=10, alpha=0.7, label="windows edges")

        # Put a legend to the right of the current axis
        box = ax.get_position()
        ax.set_position([box.x0, box.y0, box.width * 0.8, box.height])
        ax.legend(loc='center left', bbox_to_anchor=(1, 0.5))

        # finish plotting
        plt.show()
```

```
<IPython.core.display.Javascript object>


<IPython.core.display.HTML object>
```

3.1.1 Solution filter

```
def filter(wavelength, signal):
    """
    Filter the signal in order to ease extraction of maxima

    scipy.signal.wiener is a first step
    """

    filtered_signal = wiener(signal)
    return filtered_signal
```

3.1.2 Solution find_peaks

```
def find_peaks(wavelength, signal):
    """
    Find some maxima of the signal

    Finding the maximum of the signal is also a first step
    scipy.signal.argrelmax is also a good idea

    indexes_max must be a list (eventually containing ony one element)
    """

    #indexes_max = [np.argmax(signal)]
    indexes_max = argrelmax(signal)[0]
    return indexes_max
```

3.1.3 Solution get_windows

```python
def get_windows(wavelength, signal, indexes_max):
    """
    Find out window of interest around peaks

    The window of interest is the time range between two local minima around the peak

    In order to find it:
    - use scipy.signal.argrelmin to extract all local minimas
    - in a loop for all peak:
      - use numpy.argmax to find the next local minima after the current peak
      - compute the edges of the window, beware of the boundaries
    """

    windows = {}

    # get index of the minimas
    minimas = argrelmin(signal)[0]

    for peak in indexes_max:

        # get index of the next minima in minimas
        next_minima = np.argmax(minimas > peak)

        # if there is no minima after the peak
        if minimas[next_minima] < peak:
            wmax = len(signal) -1
        else:
            wmax = minimas[next_minima]

        # if there is no minima before the peak
        if next_minima == 0:
            wmin = 0
        else:
            wmin = minimas[next_minima-1]

        windows[peak] = [wmin, wmax]

    return windows
```

```
In [25]: %matplotlib notebook

         def lorentz(param, peak, x):
             """
             Lorentz function
              - param is a vector of 3 parameter : baseline, gamma and amplitude
                The optimisation will adjust them
```

```
         - peak is the position of the peak
         - x is the points of evaluation of the function
        """
        baseline, gamma, amplitude = param

        num = 2 / (np.pi * gamma)
        denom = 1 + ((x - peak) / (gamma / 2)) ** 2
        fx = num / denom

        maxlorentz = fx.max()

        return baseline + amplitude * fx / maxlorentz

    param = (100, 1, 2)
    y = lorentz(param, 656, wavelength)

    plt.figure()
    plt.plot(wavelength, y)
    plt.show()
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

```
In [26]: def lorentz_fit(wavelength, signal, peak, window):
        """
        Compute a fit of a lorentz function around the peak inside a window
        1 - define a cost function (e.g. least square)
        2 - extract the window of interest
        3 - choose appropriate intital value on the 3 lorentz parameter
        4 - apply the *minimize* procedure to optimize those parameter
            use the *args* argument in order to pass more arguments
        5 - return the optimal parameters and the resulting fit
        """

        def cost(param, peak, x, signal):
            """
            Compute the error between a lorentz fit and a signal
            1 - compute the resulting fit
            2 - compute the error
            """
            fit = lorentz(param, peak, x)
            error = np.sqrt(np.sum((fit - signal) ** 2))
            return error
```

```
    # extract data for fitting

    # initial values for baseline, gamma and amplitude
    init_param = [1., 1., 1.]

    # fitting
    opt_param = init_param

    fit = lorentz(opt_param, peak, wavelength)

    return opt_param, (wavelength, fit)


# start plot
fig= plt.figure()
ax = plt.subplot(111)

# plot the signal
plt.plot(wavelength, signal)
#plt.plot(wavelength, filtered_signal)

# add a circle on all the peaks
if False:
    plt.plot(wavelength[selected_maxpeaks],
             filtered_signal[selected_maxpeaks],
             'o', ms=10, alpha=0.7, label="peaks")

# add a circle on the limit of the windows
if False:
    plt.plot(wavelength[selected_limits],
             filtered_signal[selected_limits],
             'o', ms=10, alpha=0.7, label="base for fit")

# Apply fit on all peaks
print(("{:^8}" + " | {:^9}" * 3).format("Baseline", "Gamma", "Amplitude", "Peak"))
for key, window in peaks.items():
    peak = wavelength[key]

    param, fit = lorentz_fit(wavelength, signal, peak, window)
    print(("{:8.2f}" + " | {:9.2e}" * 3).format(*param, peak))

    # plot
    plt.plot(fit[0], fit[1], label=str(peak))

# Put a legend to the right of the current axis
box = ax.get_position()
ax.set_position([box.x0, box.y0, box.width * 0.8, box.height])
ax.legend(loc='center left', bbox_to_anchor=(1, 0.5))
```

```
        # finish plotting
        plt.show()
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

```
Baseline |   Gamma   | Amplitude |   Peak
    1.00 |  1.00e+00 |  1.00e+00 |  6.56e+02
```

3.1.4 Solution lorentz_fit

```
def lorentz_fit(wavelength, signal, peak, window):
    """
    Compute a fit of a lorentz function around the peak inside a window
    1 - define a cost function (e.g. least square)
    2 - extract the window of interest
    3 - choose appropriate intital value on the 3 lorentz parameter
    4 - apply the *minimize* procedure to optimize those parameter
        use the *args* argument in order to pass more arguments
    5 - return the optimal parameters and the resulting fit
    """

    def cost(param, peak, x, signal):
        """
        Compute the error between a lorentz fit and a signal
        1 - compute the resulting fit
        2 - compute the error
        """
        fit = lorentz(param, peak, x)
        error = np.sqrt(np.sum((fit - signal) ** 2))
        return error

    # extract data for fitting
    wmin, wmax = window
    wavelength = wavelength[wmin:wmax]
    signal = signal[wmin:wmax]

    # initial values for baseline, gamma and amplitude
    init_param = [2000., 0.1, 4000]

    # fitting
    res = minimize(cost, init_param, args=(peak, wavelength, signal))
    opt_param = res.x
```

```python
    fit = lorentz(opt_param, peak, wavelength)

    return opt_param, (wavelength, fit)
```

3.2 Exemple : EDO

In [27]:
```python
%matplotlib notebook

from scipy import integrate

def tank(t, y):
    """
    Dynamic balance for a CSTR

    C_A = y[0] = the concentration of A in the tank, [mol/L]
    T   = y[1] = the tank temperature, [K]

    Returns dy/dt = [F/V*(C_{A,in} - C_A) - k*C_A^2       ]
                    [F/V*(T_in - T) - k*C_A^2*HR/(rho*Cp) ]
    """
    F = 20.1       # L/min
    CA_in = 2.5    # mol/L
    V = 100.0      # L
    k0 = 0.15      # L/(mol.min)
    Ea = 5000      # J/mol
    R = 8.314      # J/(mol.K)
    Hr = -590      # J/mol
    T_in = 288     # K
    rho = 1.050    # kg/L

    # Assign some variables for convenience of notation
    CA = y[0]
    T  = y[1]

    # Algebraic equations
    k = k0 * np.exp(-Ea / (R * T))  # L/(mol.min)
    Cp = 4.184 - 0.002 * (T - 273)  # J/(kg.K)

    # Output from ODE function must be a COLUMN vector, with n rows
    n = len(y)      # 2: implies we have two ODEs
    dydt = np.zeros((n))
    dydt[0] = F / V * (CA_in - CA) - k * CA ** 2
    dydt[1] = F / V * (T_in - T) - (Hr * k * CA ** 2) / (rho * Cp)
    return dydt

def solve_tank():
    # Start by specifying the integrator:
    # use ``vode`` with "backward differentiation formula"
```

```
    r = integrate.ode(tank).set_integrator('vode', method='bdf')

    # Set the time range
    t_start = 0.0
    t_final = 45.0
    delta_t = 0.1
    # Number of time steps: 1 extra for initial condition
    num_steps = int((t_final - t_start) / delta_t) + 1

    # Set initial condition(s): for integrating variable and time!
    CA_t_zero = 0.5
    T_t_zero = 295.0
    r.set_initial_value([CA_t_zero, T_t_zero], t_start)

    # Create vectors to store trajectories
    t = np.zeros(num_steps)
    CA = np.zeros(num_steps)
    temp = np.zeros(num_steps)

    # Integrate the ODE(s) across each delta_t timestep
    k = 0
    while r.successful() and k < num_steps:
        # Store the results to plot later
        t[k] = r.t
        CA[k] = r.y[0]
        temp[k] = r.y[1]

        r.integrate(r.t + delta_t)
        k += 1


    # All done!  Plot the trajectories in two separate plots:
    fig = plt.figure()
    ax1 = plt.subplot(211)
    ax1.plot(t[:k], CA[:k])
    ax1.set_xlim(t_start, t_final)
    ax1.set_xlabel('Time [minutes]')
    ax1.set_ylabel('Concentration [mol/L]')
    ax1.grid('on')

    ax2 = plt.subplot(212)
    ax2.plot(t[:k], temp[:k], 'r')
    ax2.set_xlim(t_start, t_final)
    ax2.set_xlabel('Time [minutes]')
    ax2.set_ylabel('Temperaturere [K]')
    ax2.grid('on')
```

```
In [28]: solve_tank()
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

3.3 Sympy

Symbolic calculus with Python

```
In [29]: import sympy as sp
         sp.init_printing()
         x = sp.symbols('x')
         f = sp.symbols('f', cls=sp.Function)
         print(f(x))

         sp.pprint(sp.Integral(sp.sqrt(1 / x), x))
         diffeq = sp.Eq(f(x).diff(x, x) - 2 * f(x).diff(x) + f(x), sp.sin(x))
         sp.pprint(diffeq)

         sol = sp.dsolve(diffeq, f(x), dict=True)
         sp.pprint(sol)
         sol
```

```
f(x)


     ---
     1
       dx
    x


                       2
         d            d
f(x) - 2(f(x)) + (f(x)) = sin(x)
         dx            2
                     dx
                     x    cos(x)
f(x) = (C + Cx)   +
                           2
```

Out[29]:

$$f(x) = (C_1 + C_2 x)\, e^x + \frac{1}{2} \cos(x)$$

```
In [30]: C1, C2 = sp.symbols('C1 C2')
         sol.subs({x:2, C1:0, C2:3}).evalf()
```

Out[30]:

$$f(2) = 44.1262631753103$$