INS1

Introduction à Numpy et Scipy

La base à importer (pas nécessairement tous ensembles, mais tous vont servir dans ce chapitre).

```
1 import numpy as np                # Ces deux là amènent aussi un certain
2 import scipy as sp                # nombre de fonctions mathématiques.
3 import scipy.integrate            # Intégration de fonctions ou d'équadiffs.
4 import scipy.optimize             # Zéros et ajustements de fonction.
5 import matplotlib.pyplot as plt # Pour la génération de graphiques.
```

Partie I

Overview

1 Numpy

NumPy is the fundamental package for scientific computing with Python. It contains among other things:

- a powerful N-dimensional array object
- sophisticated (broadcasting) functions
- tools for integrating C/C++ and Fortran code
- useful linear algebra, Fourier transform, and random number capabilities

Besides its obvious scientific uses, NumPy can also be used as an efficient multi-dimensional container of generic data. Arbitrary data-types can be defined. This allows NumPy to seamlessly and speedily integrate with a wide variety of databases.

Numpy is licensed under the BSD license, enabling reuse with few restrictions.

2 Scipy

SciPy and friends can be used for a variety of tasks:

- NumPy's array type augments the Python language with an efficient data structure useful for numerical work, e.g., manipulating matrices. NumPy also provides basic numerical routines, such as tools for finding eigenvectors.
- SciPy contains additional routines needed in scientific work: for example, routines for computing integrals numerically, solving differential equations, optimization, and sparse matrices.
- The matplotlib module produces high quality plots. With it you can turn your data or your models into figures for presentations or articles. No need to do the numerical work in one program, save the data, and plot it with another program.

– Using IPython makes interactive work easy. Data processing, exploration of numerical models, trying out operations on-the-fly allows to go quickly from an idea to a result. See the IPython site for many examples.
– There is a sizeable collection of both generic and application-specific numerical and scientific code, written using Python, Numpy and Scipy. Don't reinvent the wheel, there may already be a pre-made solution for your problem. See Topical Software for a partial list.
– As Python is a popular general-purpose programming language, it has many advanced modules for building for example interactive applications (see e.g. wxPython and Traits) or web sites (see e.g. Django). Using SciPy with these is a quick way to build a fully-fledged scientific application.

3 Trouver de l'aide

Plusieurs méthodes existent pour trouver de la documentation sur les fonctions cherchées:

Sur internet: `http://numpy.org` et `http://scipy.org` peuvent être de bons points de départ. Voir aussi `http://docs.scipy.org`

Via `help()`: Comme d'habitude, toutes les aides internes aux fonctions et objets manipulés sont disponibles via la commande `help()`, par exemple `help(numpy)` ou `help(numpy.trapz)`.

Via `np.info()`: a le même comportement que `help()` mais ne rentre pas dans un éditeur: il affiche directement l'aide sur l'écran (ou l'imprime dans un fichier, pour plus de détails, voir `np.info(np.info)`).

Via `np.lookfor()`: Permet de regarder dans les fichiers d'aide les occurences de la chaîne de caractère donnée en argument.

```
>>> import numpy as np
>>> np.lookfor('determinant')
Search results for 'determinant'
--------------------------------
numpy.linalg.det
    Compute the determinant of an array.
numpy.linalg.slogdet
    Compute the sign and (natural) logarithm of the determinant of an array.
numpy.vander
    Generate a Vandermonde matrix.
numpy.linalg.qr
    Compute the qr factorization of a matrix.
numpy.ma.vander
    Generate a Vandermonde matrix.
numpy.linalg._umath_linalg.slogdet
    slogdet on the last two dimensions and broadcast on the rest.
```

Via `np.source()`: Permet de regarder le code source d'une fonction donnée. Particulièrement utile pour comparer et comprendre l'implémentation des différents algorithmes.

```
>>> import numpy as np
>>> np.source(np.identity)
In file: /opt/local/Library/Frameworks/Python.framework/Versions/3.3/lib/python3.3/

def identity(n, dtype=None):
    """
    Return the identity array.

```

```
9       The identity array is a square array with ones on
10      the main diagonal.
11
12      Parameters
13      ----------
14      n : int
15          Number of rows (and columns) in `n` x `n` output.
16      dtype : data-type, optional
17          Data-type of the output.  Defaults to ``float``.
18
19      Returns
20      -------
21      out : ndarray
22          `n` x `n` array with its main diagonal set to one,
23          and all other elements 0.
24
25      Examples
26      --------
27      >>> np.identity(3)
28      array([[ 1.,  0.,  0.],
29             [ 0.,  1.,  0.],
30             [ 0.,  0.,  1.]])
31
32      """
33      from numpy import eye
34      return eye(n, dtype=dtype)
```

Partie II

Fonctions ou objets utiles de Numpy

1 np.array, np.arange, np.reshape

Pour créer des tableaux de nombres sur lesquels les calculs peuvent être particulièrement efficaces et rapides (car codés en C). Permettra aussi tout le calcul matriciel par la suite si besoin.

```
>>> import numpy as np
>>> np.array([1, 2, 3])                 # Un tableau d'entiers
array([1, 2, 3])
>>> np.array([1, 2, 3.0])               # On impose implicitement les flottants
array([ 1.,  2.,  3.])
>>> np.array([[1, 2], [3, 4]])          # Plus d'une dimension
array([[1, 2],
       [3, 4]])
>>> np.array([1, 2, 3], dtype=complex) # On impose explicitement les complexes
array([ 1.+0.j,  2.+0.j,  3.+0.j])
>>> np.array(np.mat('1 2; 3 4'))        # À partir d'une matrice (notation scilab)
array([[1, 2],
       [3, 4]])
>>> np.arange(1,6,2) # Même comportement que range, mais renvoie un array
```

```
    array([1, 3, 5])
    >>> a = np.arange(25)# Tableau 1D à 25 éléments
    >>> a.reshape((5,5)) # changé en tableau 2D à 5*5 éléments
    array([[ 0,  1,  2,  3,  4],
           [ 5,  6,  7,  8,  9],
           [10, 11, 12, 13, 14],
           [15, 16, 17, 18, 19],
           [20, 21, 22, 23, 24]])
```

Attention: il faut se méfier de l'opérateur `+=` avec des `np.array` car `a+=0.1` n'aura pas le même effet si l'array `a` ne contient que des entiers ou s'il contient des entiers sous forme de flottants. En revanche, `a=a+0.1` donnera toujours le résultat voulu.

2 np.linspace, np.trapz

`np.linspace(a,b,N)`: permet de distribuer N points régulièrement espacés[1] sur un intervalle $[a\,;b]$ (b inclus pour une fois ! Peut-être exclu si besoin par l'argument optionnel `endpoint=False`). Si l'argument optionnel `retstep=True` est donné, alors renvoie la largeur des intervalles en plus du tableau de points.

`np.trapz(fx,x)`: permet de faire l'intégration par méthode des trapèzes qu'on a implémenté dans le chapitre précédent à partir d'une série `fx` des valeurs d'une fonction prise en des points `x`. Le tableau `x` est optionnel, on peut aussi définir la valeur `dx=h` si les points sont tous espacés d'une largeur `h` fixée.

```
1  def integration_trapz_n(f,a,b,n=100):
2      '''Retour sur l'intégration par méthode des trapèzes en délégant tout le
3      travail à np.trapz(). Rappel: n est le nombre d'intervalles et non le
4      nombre de points (cf pb clôtures/piquets)'''
5      # On génère d'abord les n+1 points x régulièrement espacés sur [a,b]
6      x_arr = np.linspace(a,b,n+1)
7      # Puis les valeurs de la fonction à intégrer sur ces points x
8      fx_arr= [f(x) for x in x_arr] # trapz() fera la traduction "au vol" en array
9      # enfin l'intégration proprement dite, déléguée à trapz()
10     # Le premier argument est le tableau des valeurs de la fonction à intégrer,
11     # le 2e argument étant les positions où ont été prises ces valeurs
12     return np.trapz(fx_arr,x_arr)
```

3 Utilisation de np.matrix

La classe des matrices, permet de faire du calcul matriciel

```
1  >>> import numpy as np
2  >>> A = np.matrix(np.arange(9).reshape((3,3)))
3  >>> A                          # La matrice A,
4  matrix([[0, 1, 2],
5          [3, 4, 5],
6          [6, 7, 8]])
7  >>> A**2                       # son carré matriciel,
8  matrix([[ 15,  18,  21],
```

[1] Il a un petit copain `np.logspace()` qui fait la même chose mais sur une échelle logarithmique.

```
         [ 42,  54,  66],
         [ 69,  90, 111]])
>>> A.transpose()            # sa transposée,
matrix([[0, 3, 6],
         [1, 4, 7],
         [2, 5, 8]])
>>> A*A.transpose()          # le produit matriciel avec la transposée
matrix([[  5,  14,  23],
         [ 14,  50,  86],
         [ 23,  86, 149]])
>>> (A*A.transpose()).trace()# La trace donne alors la somme des carrés des coeffs
matrix([[204]])
>>> np.array([i**2 for i in range(9)]).sum()  # Vérification
204
```

Partie III

Fonctions ou objets utiles de Scipy

1 Sous-modules

Chacun des sous-modules (`sous_module` pour l'exemple) s'importe selon la syntaxe

```
import scipy.sous_module
```

Voici la liste (par ordre alphabétique) de ceux qui pourront (a priori) vous être utiles.

```
constants   | Constantes physiques ou mathématiques
fftpack     | Routines de FFT (Transformation de Fourier Rapide)
integrate   | Integration et résolution d'équations différentielles ordinaires
interpolate | Routines d'interpolation et de lissage (smoothing splines)
linalg      | Algèbre linéaire
ndimage     | Traitement d'images à N dimensions
optimize    | Routines d'optimisation et de recherche de racines
signal      | Traitement du signal
stats       | Fonctions et distributions statistiques
```

Celui qu'on utilisera le plus dans ce cours est bien sûr `scipy.integrate`.

2 Intégration numérique

a) Calculs d'intégrales

Plusieurs méthodes d'intégration différentes existent selon les usages attendus.

```
Methods for Integrating Functions given function object.

  quad          -- General purpose integration.
  dblquad       -- General purpose double integration.
  tplquad       -- General purpose triple integration.
  fixed_quad    -- Integrate func(x) using Gaussian quadrature of order n.
  quadrature    -- Integrate with given tolerance using Gaussian quadrature.
  romberg       -- Integrate func using Romberg integration.
```

```
    Methods for Integrating Functions given fixed samples.

      trapz         -- Use trapezoidal rule to compute integral from samples.
      cumtrapz      -- Use trapezoidal rule to cumulatively compute integral.
      simps         -- Use Simpson's rule to compute integral from samples.
      romb          -- Use Romberg Integration to compute integral from
                       (2**k + 1) evenly-spaced samples.

      See the special module's orthogonal polynomials (special) for Gaussian
         quadrature roots and weights for other weighting factors and regions.

    Interface to numerical integrators of ODE systems.

      odeint        -- General integration of ordinary differential equations.
      ode           -- Integrate ODE using VODE and ZVODE routines.
```

Si l'on dispose d'un échantillonnage de la fonction:

`np.trapz(fx,x)`: on l'a déjà vu dans Numpy (et on peut aussi l'appeler selon `sp.trapz(fx,x)` ou `sp.integrate.trapz(fx,x)`). Connaissant un échantillonnage `fx` de la fonction à intégrer sur un ensemble `x` de points, elle renvoie la valeur de l'intégrale via la méthode des trapèzes.

`sp.integrate.cumtrapz(fx,x)`: fait la même chose que `np.trapz(fx,x)` sauf qu'il renvoie un tableau contenant les valeurs cumulées de l'intégrale après chaque point. L'intégrale totale se retrouve donc dans le dernier élément du tableau. Peut être utile en statistiques pour les calculs d'effectifs cumulés. Attention: par défaut, le tableau de sortie a un élément de moins que le tableau d'entrée, mais on peut donner la première valeur (pour avoir correspondance des indices des deux tableaux) via l'argument optionnel `initial=0`.

`sp.integrate.simps(fx,x)`: Connaissant un échantillonnage `fx` de la fonction à intégrer sur un ensemble `x` de points, elle renvoie la valeur de l'intégrale via la méthode de Simpson qui revient à approximer la courbe par une parabole sur trois points consécutifs.

`sp.integrate.romb(fx,dx)`: Intégration par méthode de Romberg (méthode des trapèzes « améliorée » à l'aide de DL, cf page sur wikipédia). Ne marche que s'il y a un nombre de points d'échantillonnages égal à 2^n+1 (donc 2^n intervalles élémentaires d'intégration). Attention, contrairement aux précédents, il faut fournir la largeur `dx` des intervalles (car elle doit rester fixe pour que la méthode marche)

```
>>> import scipy as sp
>>> import scipy.integrate
>>> # Points d'échantillonnage.
>>> # 'retstep=True' permet d'avoir la largeur entre deux points
>>> x,dx = sp.linspace(0,2,11,retstep=True)
>>> fx   = [xi**3 for xi in x]  # On va regarder la fonction x^3
>>> sp.trapz(fx,x)              # Méthode des trapèzes
4.04
>>> sp.trapz(fx,dx=dx)          # Méthode des trapèzes en ne précisant que 'dx'
4.0400000000000009
>>> sp.integrate.cumtrapz(fx,x) # Méthode des trapèzes avec accumulation
array([  8.00000000e-04,   8.00000000e-03,   3.60000000e-02,
         1.08800000e-01,   2.60000000e-01,   5.32800000e-01,
         9.80000000e-01,   1.66400000e+00,   2.65680000e+00,
         4.04000000e+00])
```

```
>>> sp.integrate.simps(fx,x)     # Méthode de Simpson (paraboles)
4.0
>>> # Pour la méthode de Romberg, on prend 2^3+1=9 points
>>> xromb,dxromb = sp.linspace(0,2,9,retstep=True)
>>> fxromb= [xi**3 for xi in xromb]
>>> # Usage de 'dx' obligatoire car les points (en plus d'être en nombre 2^n+1)
>>> # doivent être régulièrement espacés pour que la méthode fonctionne.
>>> sp.integrate.romb(fxromb,dx=dxromb)
4.0
```

Si l'on veut intégrer directement une fonction sans avoir à passer par la phase d'échantillonnage:

`sp.integrate.quad(f,a,b)`: intègre la fonction `f(x)` sur l'intervalle $[a\,;b]$. Renvoie un doublet contenant la valeur de l'intégrale et une estimation de l'erreur maximale commise lors de l'intégration. Si la fonction `f` prend plus de une variable, les valeurs des autres peuvent être spécifiées via l'argument optionnel `args`. Par exemple, si l'on veut intégrer `f(x,y,z)` avec `y=1` et `z=2`, on notera `sp.integrate.quad(f,a,b,args=(1,2))`. Attention, la variable d'intégration est toujours le premier argument de la fonction.

```
>>> from math import exp
>>> import scipy as sp
>>> import scipy.integrate
>>> def f(x,y,z): return exp(-x)*y*z**2
...
>>> g = lambda x: f(x,1,2)
>>> # Les bornes d'intégration peuvent être infinies
>>> sp.integrate.quad(g,0,float('inf'))
(4.000000000000001, 2.3370426971624016e-10)
>>> # Utilisation de l'argument optionnel 'args'
>>> sp.integrate.quad(f,0,float('inf'),args=(1,2))
(4.000000000000001, 2.3370426971624016e-10)
```

`sp.integrate.dblquad(f,a,b,g,h)`: intègre la fonction `f(x,y)` en x sur l'intervalle $[a\,;b]$ et en y sur l'intervalle $[g(x)\,;h(x)]$. Renvoie un doublet contenant la valeur de l'intégrale et une estimation de l'erreur maximale commise lors de l'intégration. Attention, la première variable `x` d'intégration est toujours le premier argument de la fonction et la deuxième variable `y` toujours la deuxième. Attention (bis), `g` et `h` doivent être « callable », c'est-à-dire accepter un argument (en l'occurence `x`). Si les bornes d'intégration en `y` sont fixes, on pourra utiliser des « fonctions anonymes » comme dans l'exemple ci-dessous.

```
>>> import scipy as sp
>>> import scipy.integrate
>>> def f(x,y): return x**2 * y
...
>>> sp.integrate.dblquad(f,0,1,lambda x: 0,lambda x: 1)
(0.16666666666666669, 3.6927075527489535e-15)
```

`sp.integrate.tplquad(f,a,b,g,h,r,q)`: intègre la fonction `f(x,y,z)` en x sur l'intervalle $[a\,;b]$, en y sur l'intervalle $[g(x)\,;h(x)]$ et en z sur l'intervalle $[r(x,y)\,;q(x,y)]$. Renvoie un doublet contenant la valeur de l'intégrale et une estimation de l'erreur maximale commise lors de l'intégration. Attention, la première variable `x` d'intégration est toujours le premier argument de la fonction, la deuxième variable `y` toujours la deuxième et la troisième `z` toujours la troisième. Attention (bis), `g` et `h` doivent être « callable » à un argument (`x`) et `r` et `q` doivent être « callable » à deux argument (`x,y`).

```
>>> from math import sqrt,pi
>>> import scipy as sp
>>> import scipy.integrate
>>> # On va calculer le volume d'un quart de sphère de rayon R = 1
>>> R = 1
>>> f = lambda x,y,z: 1
>>> g = lambda x: 0
>>> h = lambda x: sqrt(R**2 - x**2)
>>> r = lambda x,y: 0
>>> q = lambda x,y: sqrt(R**2 - x**2 - y**2)
>>> sp.integrate.tplquad(f,-1,1,g,h,r,q)
(1.0471975511965979, 8.833911380179416e-11)
>>> (4/3*pi*R**3) * 0.25  # Le quart du volume de la sphère de rayon R
1.0471975511965976
```

b) Résolution d'équation différentielle

C'est la procédure `odeint(f,y0,t)` qu'il faut utiliser pour intégrer numériquement une équation différentielle du type

$$\frac{\mathrm{d}y}{\mathrm{d}t} = f(y,t)$$

où la « fonction inconnue » y peut être un tableau de fonctions inconnues (ce qui permet de résoudre soit un système d'équations couplées, soit une équation d'ordre supérieur vue comme un système d'équations du premier ordre couplées). `y0` est la liste des conditions initiales (pour chaque élément du tableau `y`) à l'instant `t[0]`, et `t` représente l'ensemble des temps auquels ont veut que l'équation différentielle soit résolue. Voici les exemples qui vont être traités par la suite (les deux derniers serviront pour la présentation de Matplotlib):

$$\frac{\mathrm{d}x}{\mathrm{d}t} = -\frac{x}{\tau} \tag{1}$$

$$\begin{cases} \dfrac{\mathrm{d}x_1}{\mathrm{d}t} = -k\,x_2 \\ \dfrac{\mathrm{d}x_2}{\mathrm{d}t} = -k\,x_1 \end{cases} \tag{2}$$

$$\ddot{x} + 2\lambda\,\dot{x} + {\omega_0}^2\,x = 0 \qquad \text{qui s'écrit} \qquad \begin{cases} \dfrac{\mathrm{d}x}{\mathrm{d}t} = \dot{x} \\ \dfrac{\mathrm{d}\dot{x}}{\mathrm{d}t} = -2\lambda\,\dot{x} - {\omega_0}^2\,x \end{cases} \tag{3}$$

$$\ddot{x} + 2\lambda\,\dot{x} + {\omega_0}^2\,x = \mathrm{A}\,\cos\omega t \qquad \text{qui s'écrit} \qquad \begin{cases} \dfrac{\mathrm{d}x}{\mathrm{d}t} = \dot{x} \\ \dfrac{\mathrm{d}\dot{x}}{\mathrm{d}t} = -2\lambda\,\dot{x} - {\omega_0}^2\,x + \mathrm{A}\,\cos\omega t \end{cases} \tag{4}$$

$$\dddot{x} + \alpha\,\ddot{x} + 2\lambda\,\dot{x} + {\omega_0}^2\,x = 0 \qquad \text{qui s'écrit} \qquad \begin{cases} \dfrac{\mathrm{d}x}{\mathrm{d}t} = \dot{x} \\ \dfrac{\mathrm{d}\dot{x}}{\mathrm{d}t} = \ddot{x} \\ \dfrac{\mathrm{d}\ddot{x}}{\mathrm{d}t} = -\alpha\,\ddot{x} - 2\lambda\,\dot{x} - {\omega_0}^2\,x \end{cases} \tag{5}$$

```
>>> from math import *  # Pour exp, cosh, sinh, cos, sin, pi, etc.
>>> import scipy as sp
>>> import scipy.integrate

>>> # La solution du premier système devrait donner exp(-t/tau)
>>> def f(y,t):
...     tau = 1.0
...     return - y / tau
...
>>> y0 = 1         # Condition initiale
>>> t  = range(4) # Points temporels pour les calculs
>>> sp.integrate.odeint(f,y0,t) # Intégration proprement dite
array([[ 1.        ],
       [ 0.36787947],
       [ 0.13533528],
       [ 0.04978706]])
>>> [exp(-ti) for ti in t]      # Vérification
[1.0, 0.36787944117144233, 0.1353352832366127, 0.049787068367863944]
```

```
>>> from math import *  # Pour exp, cosh, sinh, cos, sin, pi, etc.
>>> import scipy as sp
>>> import scipy.integrate

>>> # Le second système à résoudre: 'y' est maintenant le tableau [x1,x2]
>>> # Les solutions devraient être ch(t) et -sh(t)
>>> def f(y,t):
...     k = 1.0
...     x1,x2 = y
...     return [-k*x2,-k*x1]
...
>>> y0 = [1.0,0.0]   # x1(0)=1 et x2(0)=0
>>> t  = range(4)    # Points temporels pour les calculs
>>> sp.integrate.odeint(f,y0,t) # Intégration proprement dite
array([[  1.        ,   0.        ],
       [  1.54308068,  -1.17520125],
       [  3.76219585,  -3.62686056],
       [ 10.06766261, -10.01787554]])
>>> [cosh(ti) for ti in t]      # Vérification
[1.0, 1.5430806348152437, 3.7621956910836314, 10.067661995777765]
```

```
>>> from math import *  # Pour exp, cosh, sinh, cos, sin, pi, etc.
>>> import scipy as sp
>>> import scipy.integrate

>>> # Le 3e système: équivalent à une équation différentielle du second degré.
>>> # 'y' est le tableau [x(t),dx/dt]
>>> def f(y,t):
...     lamb,w0 = 0.01,1.0  # NB: lambda est un mot "réservé" par le langage
...     x,xpoint = y
...     return [xpoint,-lamb*xpoint -w0**2*x]
...
>>> y0 = [1.0,0.0]   # x(0)=1 et dx/dt(0)=0
>>> t  = range(4)    # Points temporels pour les calculs
>>> sp.integrate.odeint(f,y0,t) # Intégration proprement dite
array([[ 1.        ,  0.        ],
       [ 0.54180441, -0.83727785],
       [-0.40748215, -0.90027134],
       [-0.97455291, -0.13905741]])
>>> # Le cours de physique dit que la solution est approximativement
>>> # exp(-lamb*t)*cos(w0*t) si w0>>lamb comme c'est le cas ici
>>> [exp(-ti*0.01)*cos(ti) for ti in t]  # Vérification
[1.0, 0.5349262080990439, -0.4079065770843121, -0.9607337965724388]
```

c) Analyse de Fourier

En physique, vous avez déjà croisé l'analyse de Fourier dans LatisPro qui peut être particulièrement utile pour étudier un signal numérisé et déterminer les fréquences des harmoniques qui le composent. Python permet lui-aussi de faire l'analyse de Fourier d'un signal temporel donné. Dans l'exemple suivant, on produit un signal

$$s(t) = 10\,\sin(2\pi\,f_1 t) + 5\,\sin(2\pi\,f_2 t) \qquad \text{avec} \qquad f_1 = 2\text{ Hz} \quad \text{et} \quad f_2 = 8\text{ Hz}$$

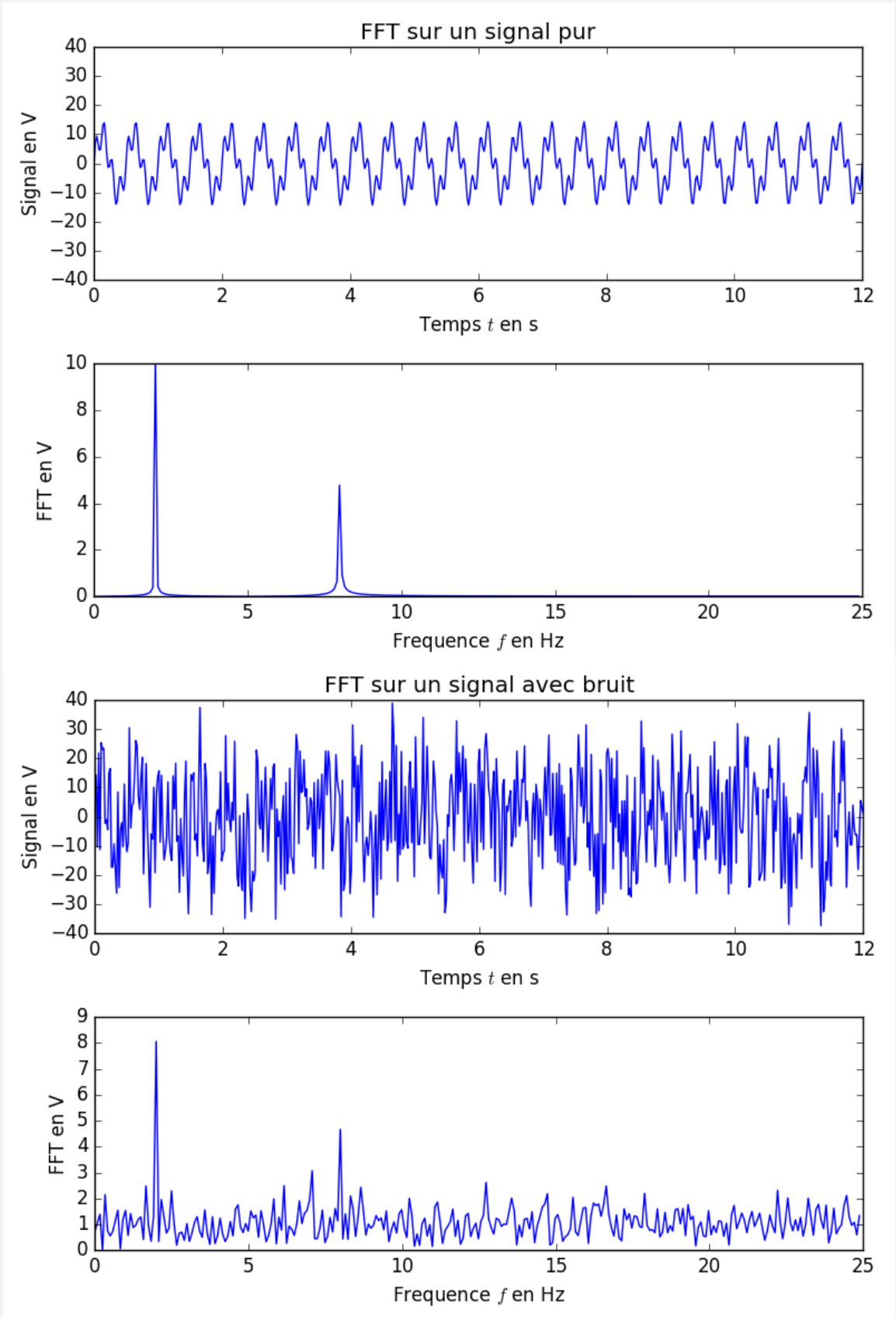
On l'échantillonne sur un certain nombre de points entre $t = 0$ et $t = 12$ s et on peut décider d'y ajouter un bruit aléatoire dont l'amplitude des variations possibles est 5 fois plus élevée que celle du signal de plus haute fréquence (les nombres aléatoires sont tirés entre 0 et 1 et comme on soustrait 0,5, les nombres avant multiplications ont des valeurs entre $-0{,}5$ et $+0{,}5$, soit un intervalle de valeurs réparties sur ± 25 autour de zéro au final). On applique alors l'algorithme de transformée de Fourier « rapide » (FFT pour *Fast Fourier Transform*) via la fonction `sp.fft()` qui renvoie des valeurs complexes dont on ne s'intéresse qu'au module (d'où l'usage de `abs()`) pour permettre l'affichage sur un graphique. Il faut aussi récupérer les fréquences d'échantillonnage via la fonction `sp.fftpack.fftfreq()`. À noter que la deuxième moitié de ces fréquences correspond à des fréquences négatives qui ne nous intéressent pas (la fonction étant réelle, le module de la FFT est une fonction paire de la fréquence), c'est pourquoi on ne prend que la moitié des valeurs pour l'affichage.

```
import scipy as sp                # La base
import scipy.fftpack              # Pour récupérer les fréquences de la FFT
import matplotlib.pyplot as plt   # Pour faire les schémas

# Définition des temps pour l'échantillonnage
t = sp.linspace(0,12,600)

```

```
8  # Le signal pur
9  signal = 10*sp.sin(2*sp.pi*2.0*t) + 5*sp.sin(2*sp.pi*8.0*t)
10 # On y rajoute du bruit, potentiellement près de 5 fois plus important que le
11 # signal à haute fréquence (sp.random.random(n) renvoie un Array de taille n
12 # dont les éléments sont des flottants aléatoire entre 0 et 1)
13 signal_avec_bruit = signal + 50*(sp.random.random(len(t))-0.5)
14
15 # On fait les FFT et on récupère les fréquences.
16 # NB: pour avoir directement les amplitudes correspondant au signal, il est
17 # nécessaire de renormaliser le résultat en divisant par la moitié du nombre
18 # de points disponibles (la normalisation est un paramètre "libre" de la
19 # transformée de Fourier discrète et le choix par défaut n'est pas celui qui
20 # correspond le mieux aux physiciens)
21 FFT = abs(sp.fft(signal)) / (signal.size/2)
22 FFT_bruit = abs(sp.fft(signal_avec_bruit)) / (signal_avec_bruit.size/2)
23 freqs = sp.fftpack.fftfreq(signal.size, t[1]-t[0])
24
25 # La FFT renvoie un complexe (d'où l'usage de "abs" précédemment pour
26 # récupérer le module) et peut s'appliquer sur des fonctions à valeurs
27 # complexes. Lorsque, comme en physique, on reste sur des fonctions à valeurs
28 # réelles, la 2e partie (qui correspond à des fréquences négatives) est
29 # redondante, on s'en débarrasse donc sans autre forme de procès.
30 FFT = FFT[0:len(FFT)//2]
31 FFT_bruit = FFT_bruit[0:len(FFT_bruit)//2]
32 freqs = freqs[0:len(freqs)//2]
33
34 # Reste à faire les graphiques (voir plus bas pour l'utilisation de matplotlib)
35 def fais_graphique(signal,FFT,freqs,titre,fichier):
36     plt.subplot(211)
37     plt.title(titre)
38     plt.ylim(-40,40)          # On impose les bornes verticales pour comparer
39     plt.plot(t, signal)       # Le signal en fonction du temps
40     plt.ylabel('Signal en V') # Légendes en x et y
41     plt.xlabel('Temps $t$ en s')
42     plt.subplot(212)
43     plt.plot(freqs,FFT)       # La FFT en fonction de la fréquence
44     plt.ylabel('FFT en V')    # Légendes en x et y
45     plt.xlabel('Frequence $f$ en Hz')
46     plt.tight_layout()        # Pour mettre un peu d'air entre les sous-figures
47     plt.savefig(fichier)      # Sauvegarde
48     plt.clf()                 # Nettoyage
49
50 fais_graphique(signal,FFT,freqs,"FFT sur un signal pur","signal_pur.png")
51 fais_graphique(signal_avec_bruit,FFT_bruit,freqs,
52                "FFT sur un signal avec bruit","signal_avec_bruit.png")
```

Les résultats sont affichés ci-après. On remarque que le pic à 2 Hz est très clairement retrouvé malgré l'ajout du bruit alors qu'à l'œil, ce n'est pas franchement évident à détecter. C'est encore pire pour le 8 Hz, absolument indétectable à l'œil, mais bien récupéré par l'analyse de Fourier la plupart du temps[2].

[2]Comme les programmes de ce fichier sont réexécutés à chaque mise à jour, il est possible que, si on n'a pas de chance avec



le tirage, le pic à 8 Hz disparaisse entièrement ou soit un peu décalé. Ce sont les aléas du métier...

3 Recherche des racines d'une fonction

Le module `scipy.optimize` regorge de procédures (cf `help(scipy.optimize)` pour le détail) permettant de trouver des minima, des zéros ou des points fixes. On va décrire ici deux algorithmes de recherche de racines:

`sp.optimize.newton(f,x0,fprime=None)`: Étant donné une fonction `f` et le voisinage `x0` d'une de ses racines, cette méthode essaie de calculer la position du zéro, soit par la méthode de la sécante (cas où `fprime` n'est pas donné), soit par la méthode de Newton (cas où `fprime`, la dérivée de `f`, est donnée). Avantage: il n'est pas nécessaire d'encadrer la racine d'un intervalle $[a\,;b]$ tel que $f(a)f(b) < 0$. Inconvénient: la convergence n'est pas assurée (mais si ça converge, ça converge très vite !).

`sp.optimize.brentq(f,a,b)`: Trouve une racine de `f` sur l'intervalle $[a\,;b]$ à condition que $f(a)f(b) < 0$ au départ. Utilise l'algorithme de Brent (1973) qui semble (d'après la doc de Scipy) être la meilleure (en terme de « rapport qualité/prix ») des méthodes permettant de détecter une racine disponibles dans le module.

```
>>> import scipy as sp
>>> import scipy.optimize
>>> f = lambda x: x**2 - 2
>>> fp= lambda x: 2*x
>>> sp.optimize.newton(f,1)       # Méthode des sécantes en partant de 1
1.4142135623730947
>>> sp.optimize.newton(f,1,fp)    # Méthode de Newton en partant de 1
1.4142135623730951
>>> sp.optimize.brentq(f,0,2)     # Méthode de Brent entre 0 et 2
1.4142135623731364
>>> sp.sqrt(2)                    # La bonne réponse
1.4142135623730951
>>> sp.optimize.newton(f,100)     # Méthode des sécantes en partant de 100
1.4142135623730951
>>> sp.optimize.newton(f,100,fp) # Méthode de Newton en partant de 100
1.4142135623730951
```

Une autre procédure qui peut s'avérer très utile est celle qui permet de « fitter »[3] un modèle sur des données expérimentales:

`sp.optimize.curve_fit(f,x,data)`: Essaie de faire coller la fonction `f` de variable principale `x` (mais ayant pour variables secondaires les paramètres d'ajustement désirés) sur les données mesurées `data`. Deux arguments optionnels sont possibles: `p0`, qui représente le jeu de paramètre initial servant de point de départ à la procédure d'optimisation (par défaut un ensemble de 1) et `sigma` qui doit être un tableau de même taille que `data` et qui sera utilisé comme poids statistique dans la régression (cas où certaines mesures sont moins précises que d'autres). La procédure renvoie un doublet constitué d'un tableau 1D des valeurs optimales des paramètres d'ajustement par la méthode des moindres carrés ainsi qu'un tableau 2D (une matrice de covariances) dont les éléments sur la diagonale donnent la variance du paramètre d'ajustement correspondant.

[3] On dit « ajuster » en français...

```
>>> import numpy as np
>>> import scipy as sp
>>> import scipy.optimize
>>> def fonction_lineaire(x,a,b): # Définition de notre ax+b
...     return a*x+b
...
>>> x = np.linspace(0,10,100)      # Distribution des points en x
>>> y = fonction_lineaire(x,3,18) # Les mesures "parfaites"
>>> data = y + 0.2*np.random.normal(size=len(x)) # On rajoute du "bruit"
>>> # L'ajustement et l'affichage des paramètres ajustés et des covariances
>>> p,pcov = sp.optimize.curve_fit(fonction_lineaire,x,data)
>>> p
array([  2.99587613,  18.01391521])
>>> pcov
array([[  6.05751189e-05,  -3.02875594e-04],
       [ -3.02875594e-04,   2.02936846e-03]])
>>> np.sqrt(pcov[0,0]) # incertitude sur a
0.0077830019233741362
>>> np.sqrt(pcov[1,1]) # incertitude sur b
0.045048512251469233
```

4 Pivot de Gauss

L'algorithme du pivot de Gauss revient à résoudre un système linéaire du type

$$\mathrm{AX} = \mathrm{B} \qquad \text{avec} \qquad \mathrm{A} \in \mathscr{M}_n(\mathbb{R}) \quad \text{et} \quad \mathrm{B} \in \mathscr{M}_1(\mathbb{R})$$

Cela revient à trouver l'inverse de la matrice A pour pouvoir écrire

$$\mathrm{X} = \mathrm{A}^{-1}\mathrm{B}$$

Nous aurons loisir de l'implémenter nous-même dans les prochains chapitres, mais voici déjà comment faire simplement avec les outils prédéfinis.

```
>>> # Inversion à l'aide de Numpy
>>> import numpy as np
>>> A = np.matrix([[1,3,2],[2,1,4],[-1,2,3]])
>>> B = np.matrix([7,8,1]).transpose()
>>> A**(-1) * B
matrix([[ 2.6],
        [ 1.2],
        [ 0.4]])
>>> # Inversion à l'aide de Scipy
>>> import scipy as sp
>>> import scipy.linalg
>>> sp.linalg.solve(A,B) # Fonctionne aussi si A et B sont des np.array
array([[ 2.6],
       [ 1.2],
       [ 0.4]])
```

Partie IV

Matplotlib

1 Overview

Matplotlib is a python 2D plotting library which produces publication quality figures in a variety of hardcopy formats and interactive environments across platforms. matplotlib can be used in python scripts, the python and ipython shell (ala MATLAB or Mathematica), web application servers, and six graphical user interface toolkits.

Matplotlib tries to make easy things easy and hard things possible. You can generate plots, histograms, power spectra, bar charts, errorcharts, scatterplots, etc, with just a few lines of code.

2 Commandes utiles de `Matplotlib`

a) `plot`, `show`, `savefig` et `clf`

Les commandes de base sont:

`plt.plot(x,y)`: permet de représenter des fonctions en reliant par une lignes les couples de points `(x[i],y[i])` successifs contenus dans les deux tableaux `x` et `y` (qui doivent donc être de même taille). La commande admet une multitude d'options, mais la plupart sont modifiables « à côté »: on s'en remettra aux divers exemples pour en découvrir quelques unes.

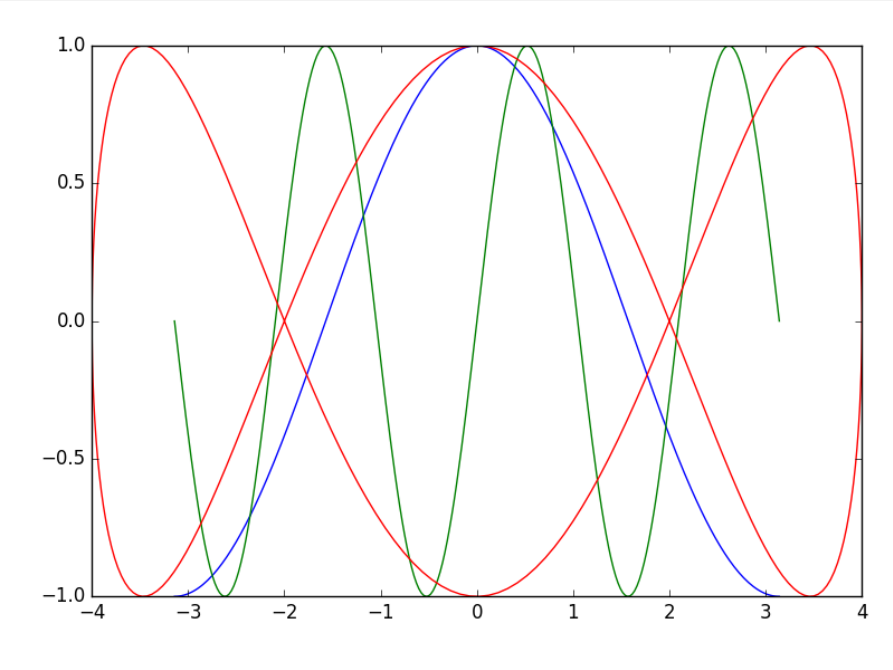
`plt.show()`: déclenche l'affichage à l'écran. Attention, cela stoppe l'exécution du script en cours jusqu'à ce que vous ayez fermé la fenêtre d'affichage. NB: ladite fenêtre d'affichage permet tout de même de faire une multitude de choses en mode interactif, notamment de zoomer sur une partie de la figure.

`plt.savefig(fichier)`: permet de sauvegarder la figure en cours dans le fichier `fichier`. Utile pour circonvenir à la limitation précédente d'arrêt du script lors de l'affichage.

`plt.clf()`: efface la figure en cours (`clf` pour « CLear Figure ») car `plt` continue d'ajouter des éléments à la même figure tant qu'on ne lui a pas dit d'arrêter. Or, il peut arriver que l'on veuille représenter plusieurs graphiques différents issus des résultats d'un même calcul. Il convient donc de penser à réinitialiser les choses avant de commencer à dessiner le prochain graphique.

```python
import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt
# Un dessin très simple commence toujours par un échantillonnage de l'axe des x
X = np.linspace(-np.pi,np.pi,256)
# Les fonctions de numpy peuvent s'appliquer séquentiellement à tous les
# éléments d'un tableau numpy:
C = np.cos(X)
S = np.sin(3*X)
# Maintenant, il faut demander l'affichage des courbes (superposées par défaut)
plt.plot(X,C)
plt.plot(X,S)
plt.plot(4*C,S)  # une belle figure de Lissajoux
# Si l'on décommentait la ligne suivante, on aurait directement l'affichage
# du graphe lors de l'exécution du script python
# plt.show()
# Mais pour les besoins de la cause, on va plutôt toujours sauvegarder les
# graphes dans des fichiers que l'on affiche par après dans le présent
# document pdf.
```

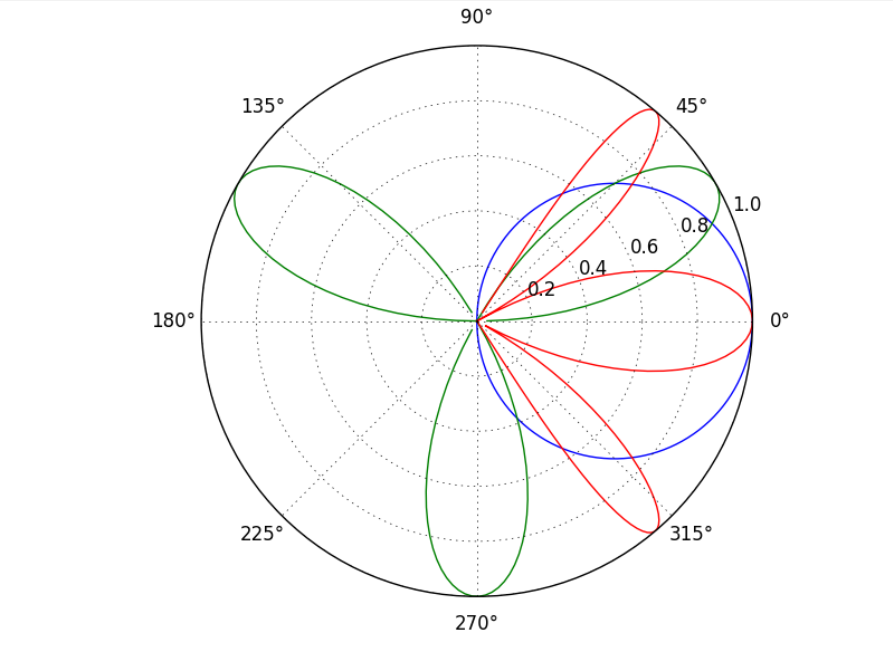
```
20 plt.savefig('cos_sin.png')
21 plt.clf()  # On nettoie la figure si jamais on veut en faire d'autres par après
```



b) `axes` et `hist`

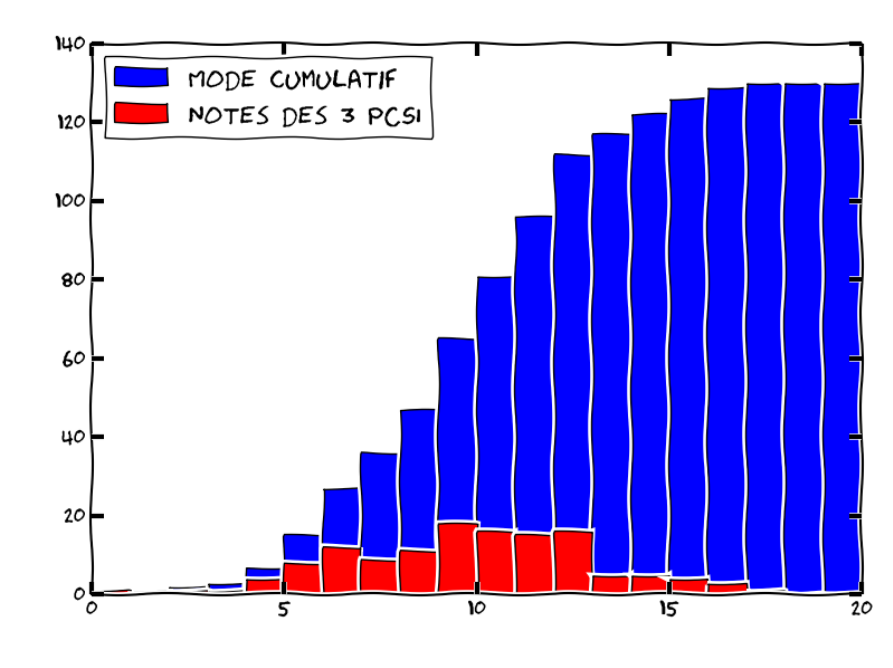
`plt.axes(polar=True)`: permet de se placer en coordonnées polaires et interprète les `plt.plot(x,y)` comme étant `plt.plot(theta,r)`.

```
1 # On continue le code précédent
2 plt.axes(polar=True)  # On se met en coordonnées polaires
3 plt.plot(X,C)         # Les trois courbes précédentes
4 plt.plot(X,S)         # du coup en polaires
5 plt.plot(C,S)
6 plt.savefig('cos_sin_polar.png')  # Sauvegarde
7 plt.clf()                         # Nettoyage
```

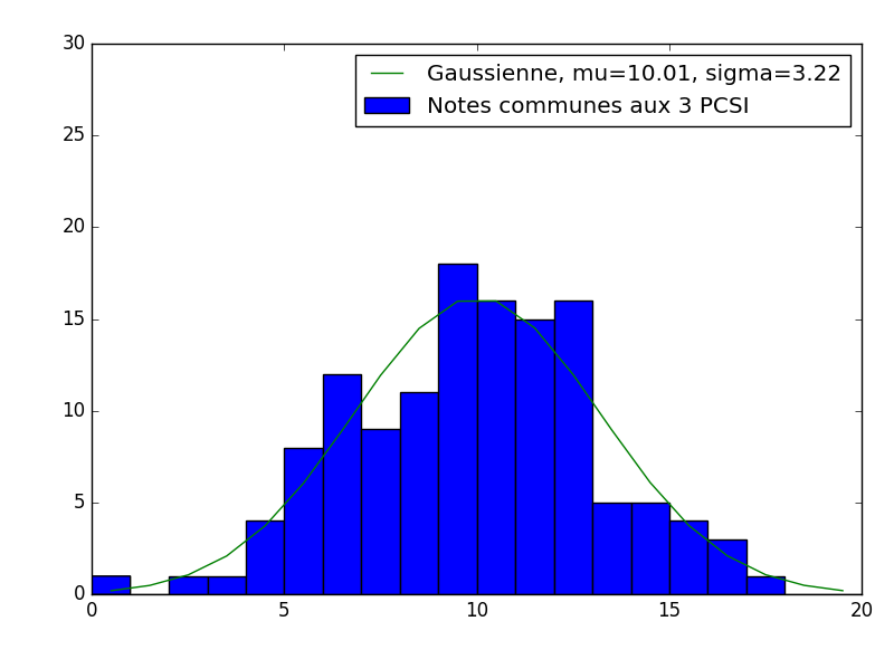


`plt.hist(data)`: représente automatiquement un histogramme où les données sont comptabilisées selon `bins` intervalles (par défaut `bins=10`) répartis sur l'intervalle total `range` (donné comme un doublet, par défaut `range=(data.min(),data.max())`). On peut aussi demander un histogramme cumulatif (option `cumulative=True`) ou une échelle verticale logarithmique (option `log=True`). À noter que `plt.hist()` est un sorte de raccourci entre l'appel à `np.histogram()` pour la fabrication de l'histogramme et `plt.plot()` pour l'affichage proprement dit. C'est d'ailleurs plutôt `np.histogram()` qu'on a intérêt à utiliser si l'on veut essayer d'ajuster un modèle théorique sur les données expérimentales, comme on peut le voir dans le 2e exemple qui essaie de retrouver la gaussienne à l'aide de `sp.optimize.curve_fit()`.

```
1  import random  # Pour tirer des notes au hasard
2  # Gaussienne de moyenne 10 et écart-type 3 pour toutes les PCSI
3  notes = [random.gauss(10,3) for k in range(130)]
4  # Dessin de l'histogramme avec 20 bins
5  # (on suppose de ce fait que personne n'aura pile 20/20... [ou plus !])
6  with plt.xkcd():              # XkcD sketch style, for the fun of it !!
7      plt.hist(notes,bins=20,range=(0,20),cumulative=True,label="Mode cumulatif")
8      plt.hist(notes,bins=20,range=(0,20),label="Notes des 3 PCSI")
9      plt.legend(loc='upper left') # Les légendes
10     plt.savefig('hist.png')  # Sauvegarde
11     plt.clf()                # Nettoyage
```




```python
import scipy.optimize
# On veut retrouver la courbe de distribution des notes
# On affiche l'histogramme "de base"
plt.hist(notes,bins=20,range=(0,20),label='Notes communes aux 3 PCSI')

# On demande à numpy de faire l'histogramme "à la main"
hist_data,bin_edges = np.histogram(notes,bins=20,range=(0,20))
# Notez qu'il y a toujours un piquet de plus que de clôtures...
# N'ayant pas plus d'info, on va prendre le point milieu pour la valeur des notes
bin_centers = (bin_edges[:-1] + bin_edges[1:])/2

# On définit le modèle à "fitter" aux données, à savoir la gaussienne
def gauss(x, A, mu, sigma):
    return A*np.exp(-(x-mu)**2/(2.*sigma**2))

depart = [30,8,6]    # On définit un point de départ "plausible" pour [A,mu,sigma]
                     # Et on fait l'ajustement
coeff,var_matrix = sp.optimize.curve_fit(gauss,bin_centers,hist_data,p0=depart)
A,mu,sigma = coeff  # Récupération des paramètres
hist_fit = gauss(bin_centers,A,mu,sigma) # La courbes résultantes

# Son affichage par-dessus l'histogramme
rmu,rsigma = round(mu,2),round(sigma,2)
plt.plot(bin_centers,hist_fit,label='Gaussienne, mu={}, sigma={}'.format(rmu,rsigma))
plt.legend()            # Choix automatique du placement de la légende
plt.ylim(0,30)          # Pour laisser de la place pour la légende
plt.savefig('fit_result.png')
plt.clf()
```



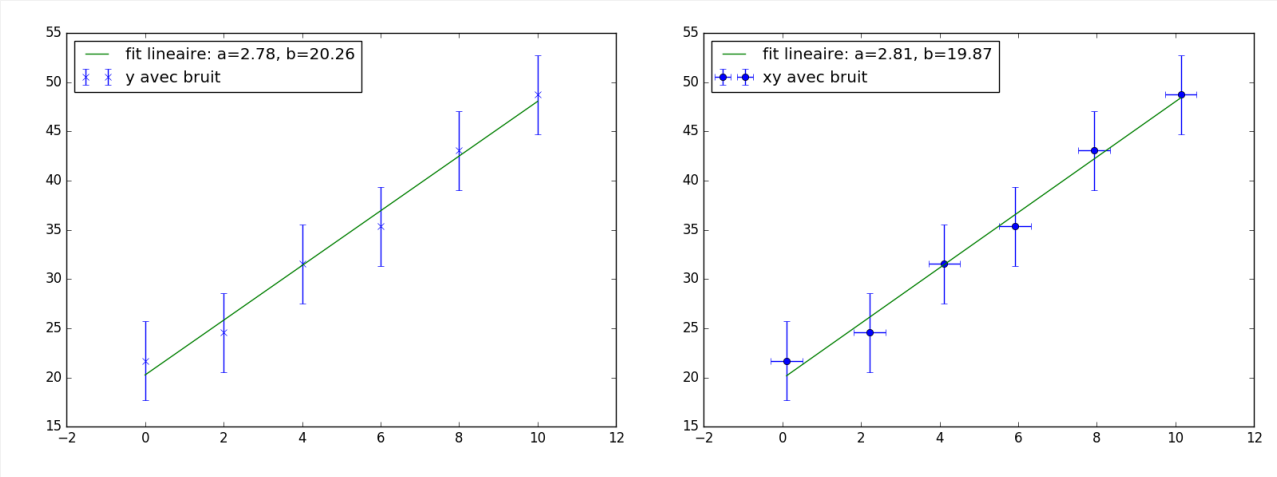
On voit que l'on retrouve bien les valeurs injectées initialement mais qu'avec ces « petits » nombres de notes, l'écart à la gaussienne ajustée peut être important.

c) Barres d'erreurs

En TIPE, il peut être utile de présenter des résultats signalant les barres d'erreur associées aux mesures expérimentales. Pour ce faire, on dispose de la commande `plt.errorbar(x,y)`. Bien sûr, sans argument optionnel, elle se comporte comme `plt.plot(x,y)` et se contente de tracer `y` en fonction de `x`. Néanmoins, elle dispose des arguments optionnels `yerr=` et `xerr=` pour signaler les barres d'erreur à rajouter sur le graphe qui peuvent être soit un simple nombre (donc tout le monde aura la même barre d'erreur), soit un tableau de taille `len(x)` (barres d'erreur individuelle), soit un tableau 2D de taille $2\times$`len(x)` (barres d'erreur non symétrique gauche/droite ou bas/haut). Voyons ce qu'il advient sur un exemple

```
# On reprend l'exemple linéaire
f = lambda x,a,b: a*x+b  # Notre fonction
x = np.linspace(0,10,6)  # Distribution des points en x
y = f(x,3,18)            # Les mesures "parfaites"
datay = y + 2*np.random.normal(size=len(x))   # On rajoute du "bruit" en y
datax = x + 0.2*np.random.normal(size=len(x)) # On rajoute du "bruit" en x

# L'ajustement, d'abord avec des mesures parfaites en x
p1,p1cov = sp.optimize.curve_fit(f,x,datay)
fit1 = f(x,*p1)              # On échantillonne la fonction "fittée"
# Puis avec des erreurs sur x aussi
p2,p2cov = sp.optimize.curve_fit(f,datax,datay)
fit2 = f(datax,*p2)          # On échantillonne la fonction "fittée"



# Reste à dessiner le tout, barres d'erreur à 2sigma
```

```
a1,b1,a2,b2 = [round(a,2) for a in (p1[0],p1[1],p2[0],p2[1])] # Un peu de formatage
plt.errorbar(x,datay,yerr=4,label='y avec bruit',fmt='x')
plt.plot(x,fit1,label='fit lineaire: a={}, b={}'.format(a1,b1))
plt.xlim((-2,12))              # On change les limites pour voir les bords
plt.legend(loc='upper left') # Légende en haut à gauche
plt.savefig('yerr.png')      # Sauvegarde
plt.clf()                    # Nettoyage avant 2e graphique
plt.errorbar(datax,datay,xerr=0.4,yerr=4,label='xy avec bruit',fmt='o')
plt.plot(datax,fit2,label='fit lineaire: a={}, b={}'.format(a2,b2))
plt.legend(loc='upper left') # Toujours la légende
plt.savefig('xyerr.png')     # Sauvegarde
plt.clf()                    # Nettoyage
```

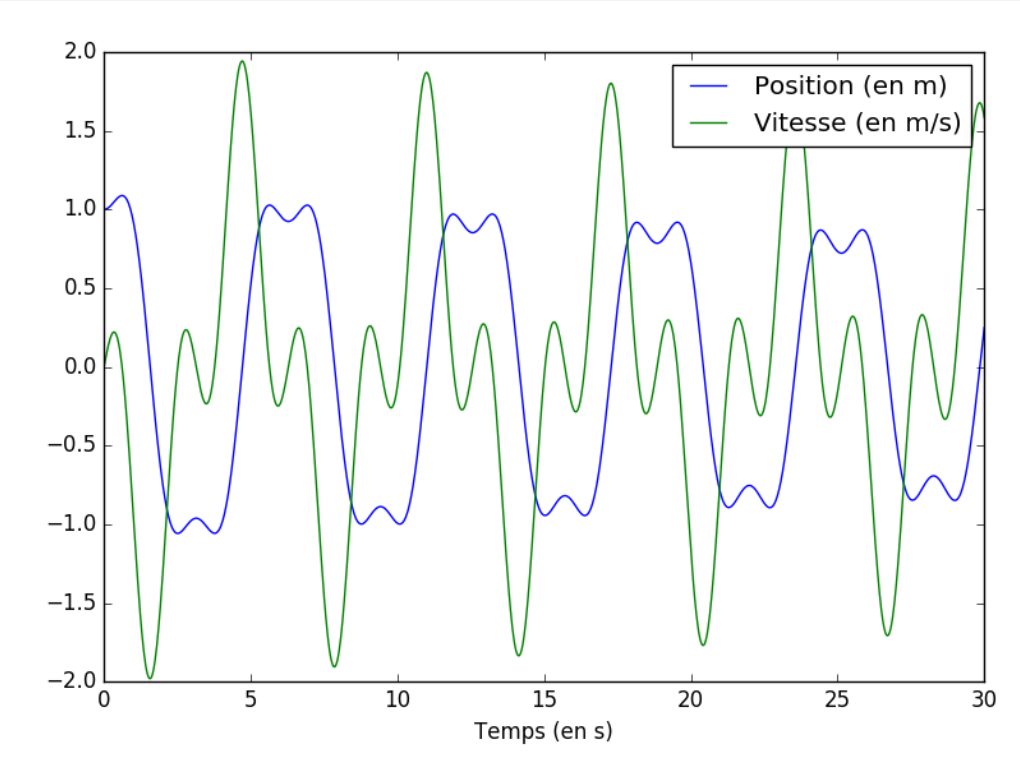


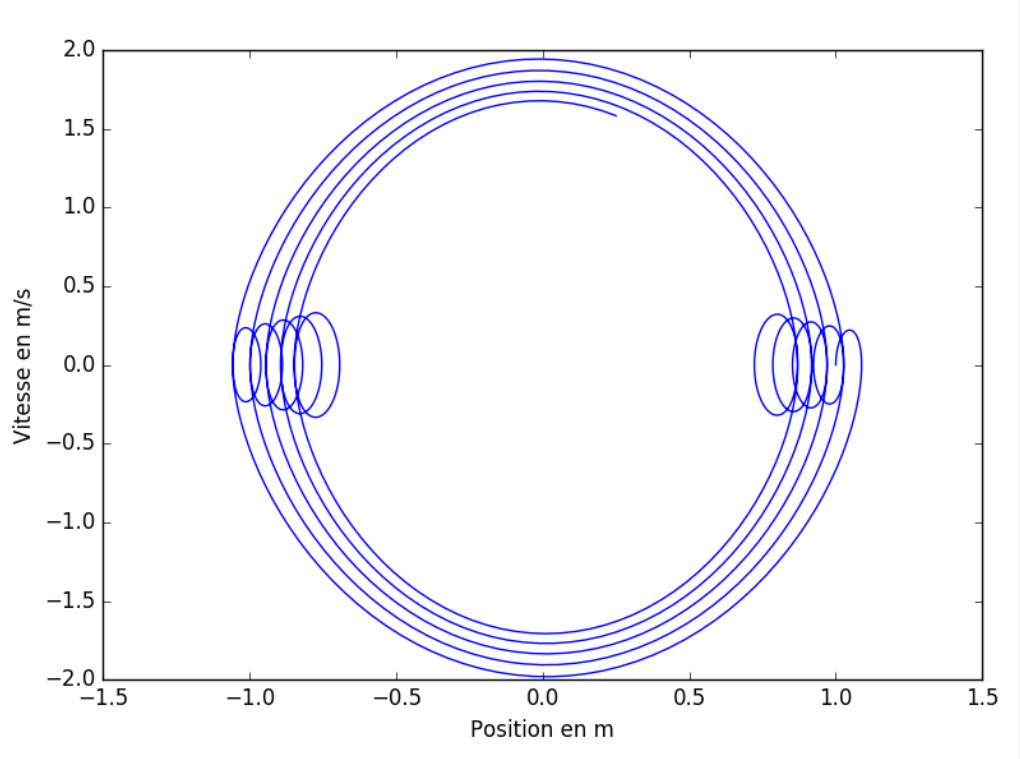
d) Exemples de résolution d'équations différentielles

Terminons par la résolution et les divers affichages des systèmes (**4**) et (**5**) précédemment définis.

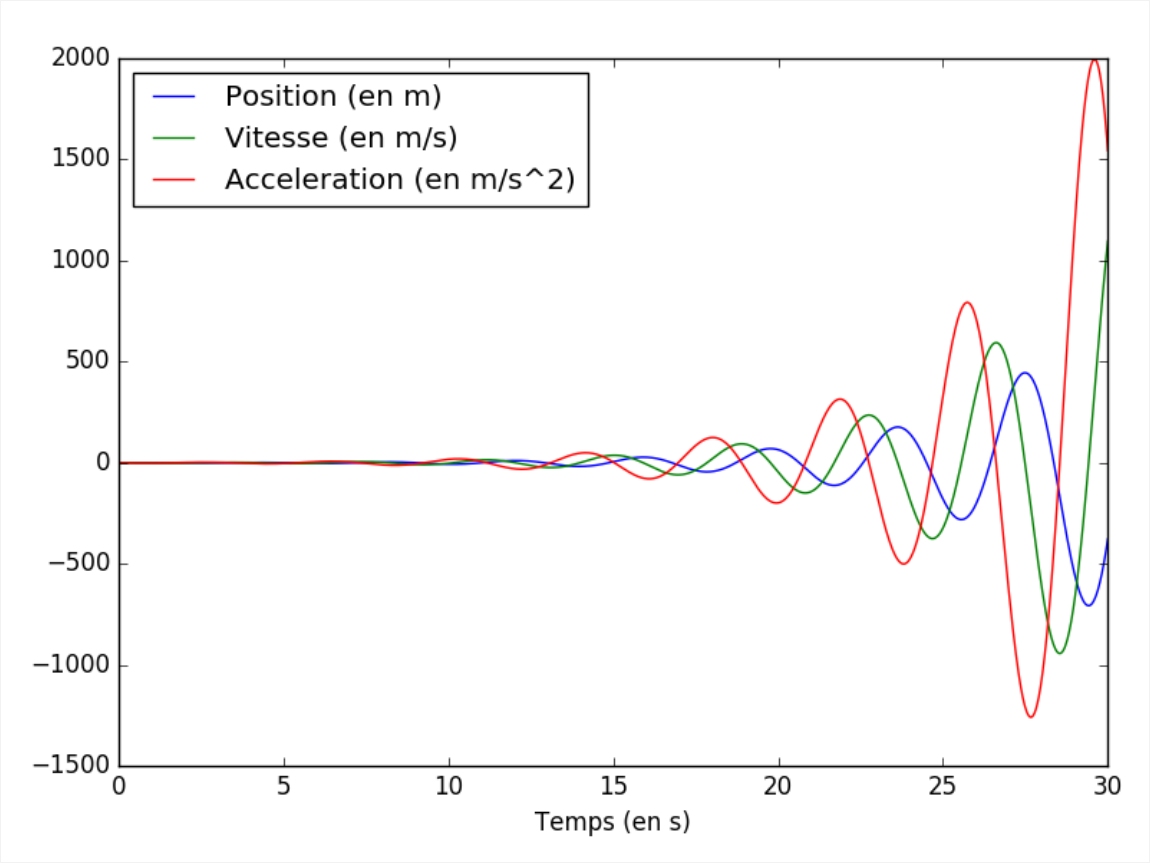
```
# Résolution numérique du système 4 et affichage
def f(y,t):
    x,xpoint = y
    lamb,w0,A,w = 0.01,1,2,3
    return [xpoint,-2*lamb*xpoint-w0**2*x + A*np.cos(w*t)]

t  = sp.linspace(0,30,1001)       # 1001 points de mesure
y0 = [1,0]                        # x(0) = 1, dx/dt(0) = 0
sol = sp.integrate.odeint(f,y0,t) # Résolution proprement dite
position = sol[:,0]               # Récupération de la position
vitesse  = sol[:,1]               # et de la vitesse (noter le double indiçage)
plt.plot(t,position,label="Position (en m)")  # La position
plt.plot(t,vitesse, label="Vitesse (en m/s)") # et la vitesse
plt.legend(loc='upper right')                 # On rajoute la légende,
plt.xlabel('Temps (en s)')                    # le label horizontal
plt.savefig('systeme_4.png')                  # et on sauvegarde le tout
plt.clf()                                     # avant nettoyage.
plt.plot(position,vitesse)        # Portrait de phase
plt.xlabel('Position en m')       # Légende sur l'axe x
```

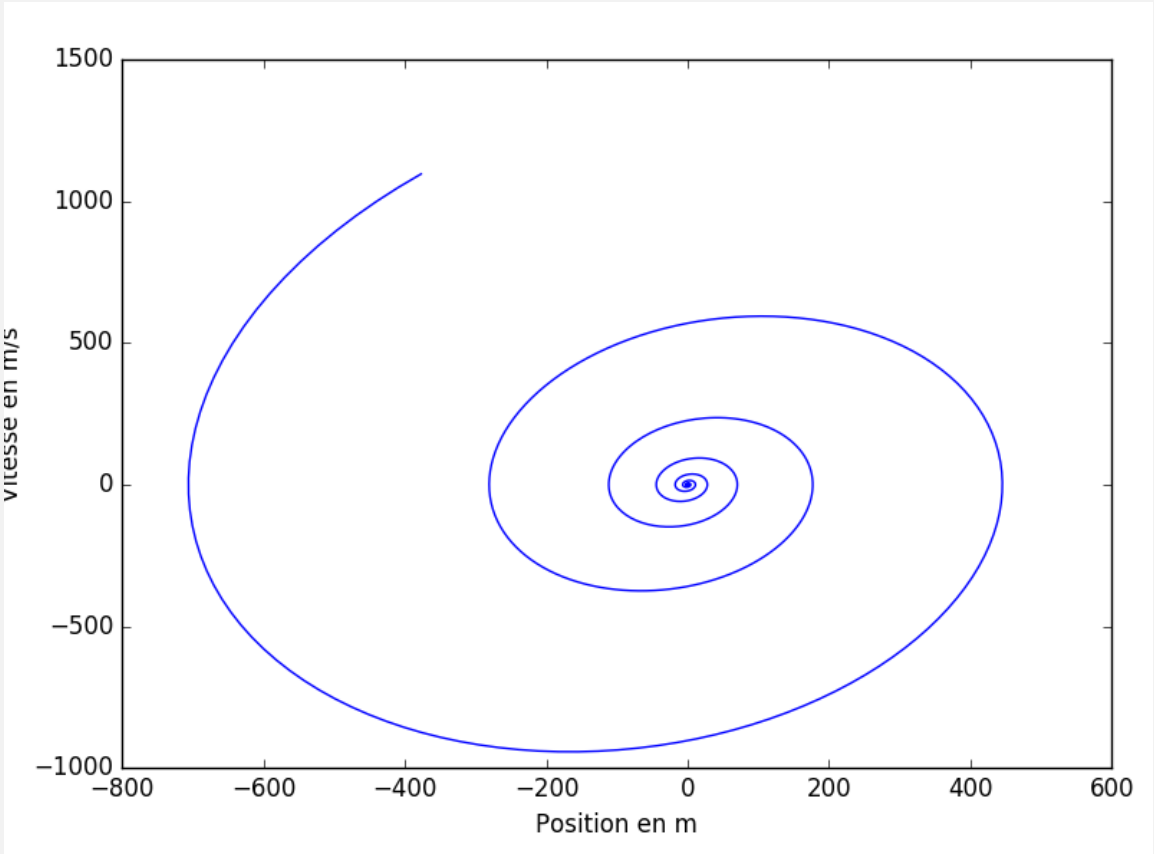
```
20 plt.ylabel('Vitesse en m/s')       # Légende sur l'axe y
21 plt.savefig('systeme_4_phase.png')# et on sauvegarde le tout
22 plt.clf()                         # Nettoyage
```

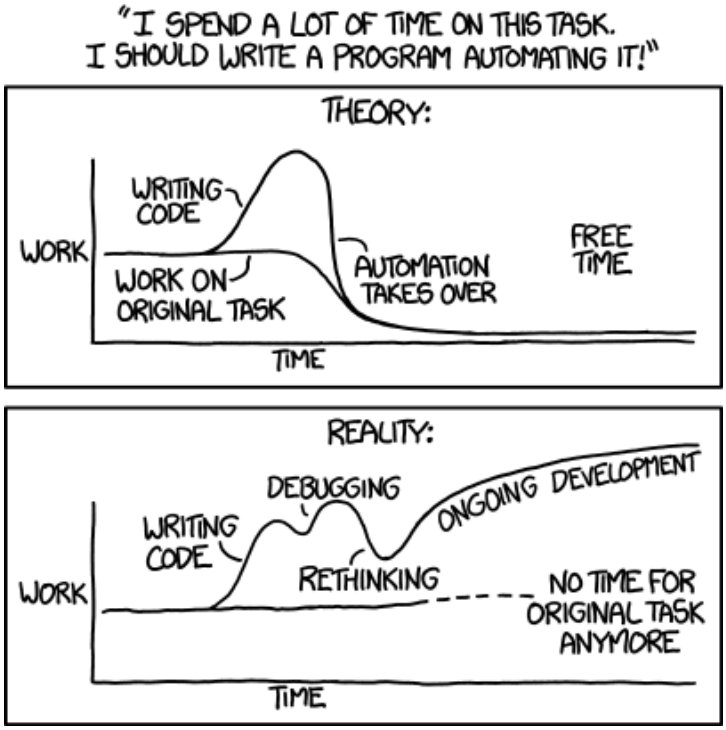




```
##########################################################################
##########################################################################
# Résolution numérique du système 5 et affichage. Cette fois-ci, on va
# jusqu'à la dérivée seconde dans y car l'équation est d'ordre 3.
##########################################################################
##########################################################################

############################
# Préparation de l'équadiff
############################
def f(y,t):
    x,xp,xpp = y
    alpha,lamb,w0 = 1,1,2
    return [xp,xpp,-alpha*xpp-2*lamb*xp-w0**2*x]

##############################
# Intégration proprement dite
##############################
t  = sp.linspace(0,30,1001)       # 1001 points de mesure
y0 = [1,0,0]                      # x(0) = 1, dx/dt(0) = 0, d2x/dt2(0)=0
sol = sp.integrate.odeint(f,y0,t) # Résolution proprement dite
position     = sol[:,0]           # Récupération de la position
vitesse      = sol[:,1]           # de la vitesse et
acceleration= sol[:,2]            # de l'accélération (noter le double indiçage)

#####################################
# Représentation graphique: temporelle
#####################################
plt.plot(t,position,label="Position (en m)")             # La position,
plt.plot(t,vitesse, label="Vitesse (en m/s)")            # la vitesse
plt.plot(t,acceleration,label="Acceleration (en m/s^2)") # et l'accélération
plt.legend(loc='upper left')      # On rajoute la légende,
plt.xlabel('Temps (en s)')        # le label horizontal
plt.savefig('systeme_5.png')      # et on sauvegarde le tout
plt.clf()                         # Nettoyage


############################################
# Représentation graphique: portrait de phase
############################################
plt.plot(position,vitesse)        # Portrait de phase
plt.xlabel('Position en m')       # Légende sur l'axe x
plt.ylabel('Vitesse en m/s')      # Légende sur l'axe y
plt.savefig('systeme_5_phase.png')# et on sauvegarde le tout
plt.clf()                         # Nettoyage
```

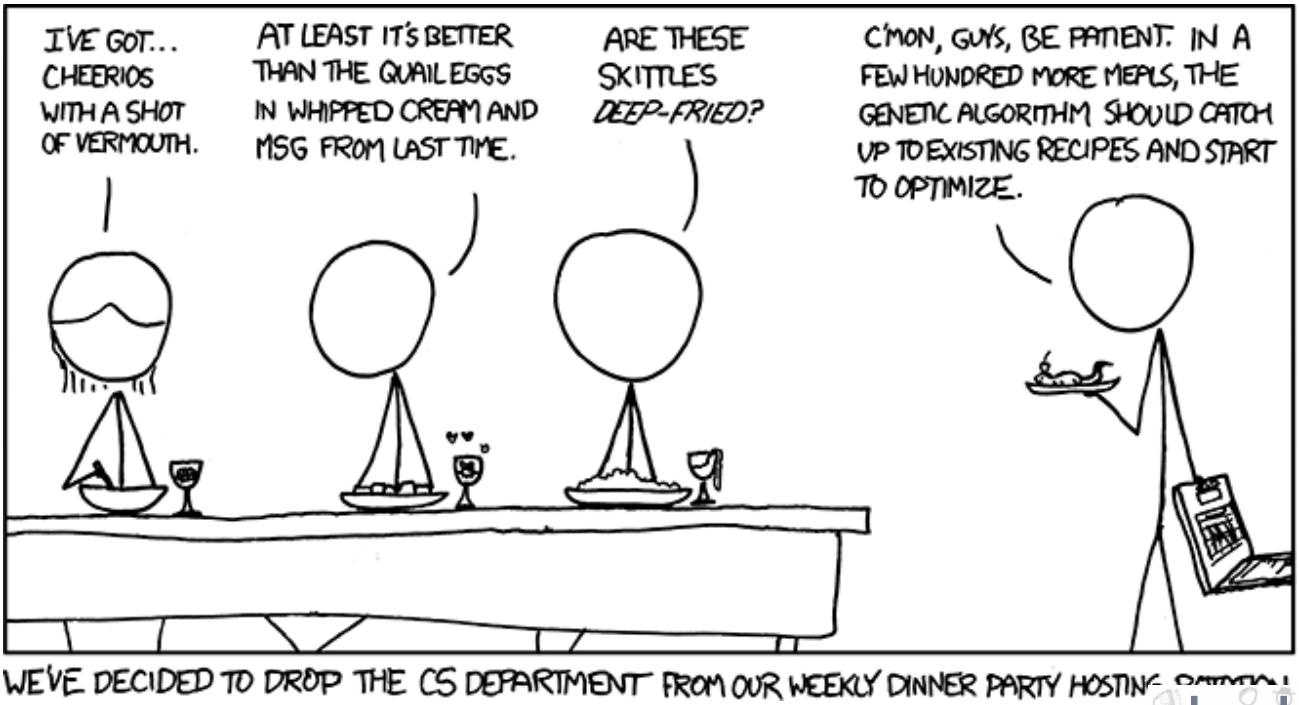

Position (en m)
Vitesse (en m/s)
Acceleration (en m/s^2)
2000
1500
1000
500
0
−500
−1000
−1500
0
5
10
15
20
25
30
Temps (en s)


1500
1000
500
0
−500
−1000
−800
−600
−400
−200
0
200
400
600
Vitesse en m/s
Position en m



xkcd.com

"Automating" comes from the roots "auto-" meaning "self-",
and "mating", meaning "screwing".



xkcd.com

To be fair, the braised and confused newt
on a bed of crushed Doritos turned out to be delicious.