# Python : débogueur, bibliothèques, objets

## Table des matières



# 1 Bibliothèques

## Le contexte

Il s'agit de découvrir les différents outils proposés par Python pour l'ingénierie numérique. Au casting, nous trouvons comme différents intervenants :

- La librairie `numpy` fournit ce qu'il faut pour créer et manipuler des *tableaux* (`array`), qui sont des listes homogènes (que des entiers, que des flottants... mais surtout pas de mélange). La sous-librairie `numpy.linalg` contient des fonctions spécialisées d'algèbre linéaire.
- Au dessus, on trouve `scipy`, qui est une librairie de calcul scientifique basée sur `numpy`. Sa langue naturelle est le tableau.
- La librairie `matplotlib` fournit un ensemble de fonctions permettant de représenter toutes sortes de graphiques. Nous utiliserons essentiellement la sous-librairie `matplotlib.pyplot`.
- La fonction `odeint` de la sous-librairie `scipy.integrate` réalise des résolutions numériques d'équations différentielles.

Il ne s'agit pas d'entrer dans les détails des nombreuses options (la librairie `matplotlib` est en particulier très fournie !), mais plutôt de savoir faire assez rapidement et simplement des choses assez basiques. Les détails viendront avec la pratique et à l'aide de la documentation.

## 1.1 La librairie `numpy`

Un petit aide-mémoire pour compléter l'exercice sur le pivot de Gauss :

| **Quoi** | **Comment** |
|---|---|
| Créer une matrice particulière | `zeros((n, p))` ; `random.random((n, p))`<br>`eye(n)` (identité) ; `fromfunction(f, (n, p))` |
| Manipuler des matrices | `2*a` ; `a+b` ; `a.transpose()` ou `a.T` ; `dot(a, b)` |
| Sous-bibliothèque `numpy.linalg` | `inv(a)` ; `solve(a, b)` ; `eigvals(a)` |

## 1.2 La librairie `matplotlib`

La première chose à comprendre est que par défaut, `plot` va relier des points (donnés par listes ou tableaux d'abscisses et d'ordonnées). Et une courbe, c'est par définition une ligne brisée avec très exactement *plein* de points.

| Quoi | Comment |
|---|---|
| Charger la librairie ! | `import matplotlib.pyplot as pypl` |
| Tracer une ligne brisée | `pypl.plot([x1,...,xn], [y1...,yn])` |
| Mettre des titres/noms aux axes | `pypl.title('Titre de la figure')`<br>`pypl.xlabel('Nom de l'axe des x')`<br>`pypl.ylabel('Nom de l'axe des y')` |
| Visualiser le résultat | `pypl.show()` |
| Effacer la fenêtre graphique courante | `pypl.clf()` |
| Sauver la figure<br>(sous différents formats !) | `pypl.save('jolie-figure.bmp')`<br>`pypl.savefig('jolie-figure.pdf')` |

Voici un premier exemple de figure :

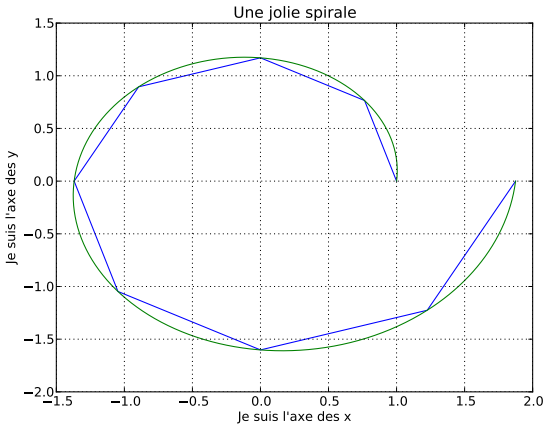


FIGURE 1 – Une spirale logarithmique

On peut l'obtenir avec les commandes suivantes :

**Programme 1 — Un premier exemple**

```
tt = np.linspace(0, math.pi*2, 9)
tt2 = np.linspace(0, math.pi*2, 1000)

def fx(t): return math.cos(t) * math.exp(t/10)
def fy(t): return math.sin(t) * math.exp(t/10)

pypl.plot(list(map(fx, tt)), list(map(fy, tt)))
pypl.plot(list(map(fx, tt2)), list(map(fy, tt2)))

pypl.title('Une␣jolie␣spirale')
pypl.xlabel('Je␣suis␣l\'axe␣des␣x')
pypl.ylabel('Je␣suis␣l\'axe␣des␣y')
pypl.grid(True)

pypl.savefig('FilesOut/spirale.pdf')
pypl.show()
```

Exercice 1 *Tracer une ellipse de rayons $a$ et $b$ dont les axes sont ceux du repère de trois façons différentes :*

- *en utilisant ses équations paramétriques* $(a\cos t, b\sin t)$
- *en utilisant son équation cartésienne* $\pm b\sqrt{1-(x/a)^2}$
- *en utilisant son équation polaire* $p/(1+e\cos\theta)$ *(e : excentricité, p : paramètre).*

Notons qu'il existe différentes solutions pour discrétiser un intervalle, plus ou moins agréables selon qu'on a fixé le pas, les bornes de l'intervalle ou encore le nombre de morceaux souhaités.

**Programme 2 — Trois façons de casser l'intervalle $[1,2]$ en dix morceaux.**

```
>>> lp0 = [1 + float(k)/10 for k in range(11)]
[1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9, 2.0]
>>> lp1 = np.arange(1, 2.1, 0.1)
array([ 1., 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9, 2. ])
>>> lp2 = np.linspace(1,2,11)
array([ 1., 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9, 2. ])
```

Pour le premier, on comprend bien ce qui se passe. Pour `arange` ça reste dans l'esprit Python (frontière droite exclue) et pour `linspace`, on compte les *poteaux* et non les *intervalles*.

## 2 Intégrer une équation différentielle avec `odeint`

Le principe d'utilisation de `odeint` (pour intégrer numériquement des équations différentielles) est le suivant : pour avoir une estimation numérique de la solution du problème de Cauchy $\begin{cases} Y'(t) = F(Y(t),t) \\ Y(t_0) = Y_0 \end{cases}$ on donne comme argument la *fonction* $F$ (qui `doit` avoir deux paramètres, même dans le cas automome, avec $t$ (suivant les notations précédentes) comme *deuxième* paramètre), la condition initiale $Y_0$, et le domaine de temps qui nous intéresse (qui commence à $t_0$). Il nous est retourné un *tableau* (même si $t$ était une liste).

**Programme 3 — Une équation pas trop compliquée**

```
from scipy.integrate import odeint

def f(y, _): return y

t = np.linspace(0, 1, 3)
y = odeint(f, 1, t)
pypl.plot(t, y)

t = np.linspace(0, 1, 100)
y = odeint(f, 1, t)
pypl.plot(t, y)

pypl.plot(t, np.exp(t))
```

Après le premier appel à `odeint`, le tableau `y` vaut :

```
array([[ 1.        ], [ 1.64872127], [ 2.71828191]])
```

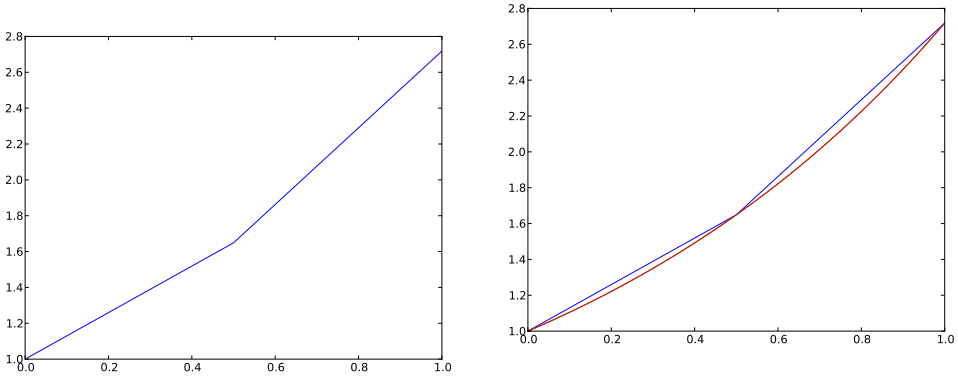

FIGURE 2 – Le résultat avec 3 points puis un peu plus

On notera que le nombre de points en lesquels les résultats sont évalués n'est pas (du moins directement) relié à la précision des calculs internes (ne pas imaginer que cela fixe le pas de la méthode, en particulier). Sur la seconde figure, la solution réelle se superpose à celle représentée en 100 points.

On peut également traiter le cas où $Y$ est un $k$-uplet. Cela permet de traiter le cas d'un système différentiel du type $\begin{cases} y_1'(t) = F_1(y_1(t), y_2(t), t) \\ y_2'(t) = F_1(y_1(t), y_2(t), t) \end{cases}$ ce qui inclut les équations différentielles d'ordre 2 : la résolution de $y''(t) = f(y(t), y'(t), t)$ passera par celle du système différentiel $\begin{cases} y_1'(t) = y_2(t) \\ y_2'(t) = f(y_1(t), y_2(t), t) \end{cases}$
Dans le résultat (qui est une liste de couple, notons-la `res`), les premières composantes (que l'on récupère à la `numpy` via `res[:, 0]`) représentent les valeurs prises par $y$, et la seconde composante représentera celles de $y'$. Cela permettra de représenter au choix le graphe de $y$ ($y(t)$ en fonction de $t$) ou bien le portrait de phase (les couples $(y(t), y'(t))$).

**Programme 4 — $x''(t) = -x(t)$**

```
x =  np.linspace(0, 7, 1000)

def g(Y, t):
    [y, yp] = Y
    return [yp, -y]

yy = odeint(g, [0, 1], x)
pl.plot(x, yy[:,0])
pl.show()
pl.clf()
pl.plot(yy[:,0], yy[:,1])
pl.show()
```

## 2.1 Exercices pour le matheux

EXERCICE 2 (UNE INSTABILITÉ NUMÉRIQUE CLASSIQUE) *On s'intéresse ici à l'équation différentielle $y'' = -y' + 6y$ avec les conditions initiales : $y(0) = 2$ et $y'(0) = -6$.*

1. *Représenter la solution obtenue par `odeint` sur l'intervalle $[0, 10]$.*
2. *Vérifier formellement que $y(t) = 2e^{-3t}$ est solution.*
3. *Prendre un air intelligent, et expliquer que tout se passe comme prévu.*
4. *Représenter la solution obtenue par `odeint` sur l'intervalle $[0, 20]$.*

5. *Réfléchir un peu, prendre un air inquiet, et expliquer que tout cela est anormal.* Indication : lire le titre !
6. *Réfléchir un peu plus (ou expérimenter), prendre un air ravi, pointer et expliquer le phénomène attendu !*
7. *Bonus : en déduire le nombre approximatif*[1] *de bits significatifs (bref : la taille de la mantisse) dans les flottants manipulés par Python.*

**Remerciements**

Un grand merci à Judicaël Courant et Stéphane Gonnord pour avoir accepté de partager leur matériel.

1. Allez, à 5 près !