CONCOURS CENTRALE•SUPÉLEC

Oral

Python

MP, PC, PSI, TSI

Analyse numérique

La plupart des fonctions présentées dans cette section nécessitent l'import du module `numpy` et de sous-modules du module `scipy`. Les instructions nécessaires aux exemples suivants sont listés ci-dessous.

```
import numpy as np
import scipy.optimize as resol
import scipy.integrate as integr
import matplotlib.pyplot as plt
```

Nombres complexes

Python calcule avec les nombres complexes. Le nombre imaginaire pur i se note `1j`. Les attributs `real` et `imag` permettent d'obtenir la partie réelle et la partie imaginaire. La fonction `abs` calcule le module d'un complexe.

```
>>> a = 2 + 3j
>>> b = 5 - 3j
>>> a*b
(19+9j)
>>> a.real
2.0
>>> a.imag
3.0
>>> abs(a)
3.6055512754639896
```

Fonctions mathématiques

La constante π s'obtient grâce à la commande `pi`.

Le module `numpy` connaît les fonctions mathématiques usuelles. La fonction partie entière s'obtient par la commande `floor`. Attention la fonction logarithme népérien a pour nom de commande `log`.

```
>>> np.exp(1)
2.7182818284590451
>>> np.cos(np.pi)
-1.0
>>> np.log(np.exp(1))
1.0
>>> np.floor(3.4)
3
>>> np.floor(-3.7)
-4
```

Résolution approchée d'équations

Pour résoudre une équation du type $f(x) = 0$ où f est une fonction d'une variable réelle, on peut utiliser la fonction `fsolve` du module `scipy.optimize`. Il faut préciser la valeur initiale x_0 de l'algorithme employé par la fonction `fsolve`. Le résultat peut dépendre de cette condition initiale.

```
def f(x):
    return x**2 - 2

>>> resol.fsolve(f, -2.)
array([-1.41421356])

>>> resol.fsolve(f, 2.)
array([ 1.41421356])
```

Dans le cas d'une fonction f à valeurs vectorielles, on utilise la fonction `root`. Par exemple, pour résoudre le système non linéaire

$$\begin{cases} x^2 - y^2 = 1 \\ x + 2y - 3 = 0 \end{cases}$$

```
def f(v):
    return v[0]**2 - v[1]**2 - 1, v[0] + 2*v[1] - 3

>>> sol = resol.root(f, [0,0])
>>> sol.success
True
>>> sol.x
array([ 1.30940108,  0.84529946])

>>> sol=resol.root(f, [-5,5])
>>> sol.success
True
>>> sol.x
array([-3.30940108,  3.15470054])
```

Calcul approché d'intégrales

La fonction `quad` du module `scipy.integrate` permet de calculer des valeurs approchées d'intégrales. Elle renvoie une valeur approchée de l'intégrale ainsi qu'un majorant de l'erreur commise. Cette fonction peut aussi s'employer avec des bornes d'intégration égales à $+\infty$ ou $-\infty$.

```
def f(x):
    return np.exp(-x)

>>> integr.quad(f, 0, 1)
(0.6321205588285578, 7.017947987503856e-15)

>>> integr.quad(f, 0, np.inf)
(1.0000000000000002, 5.842607038578007e-11)
```

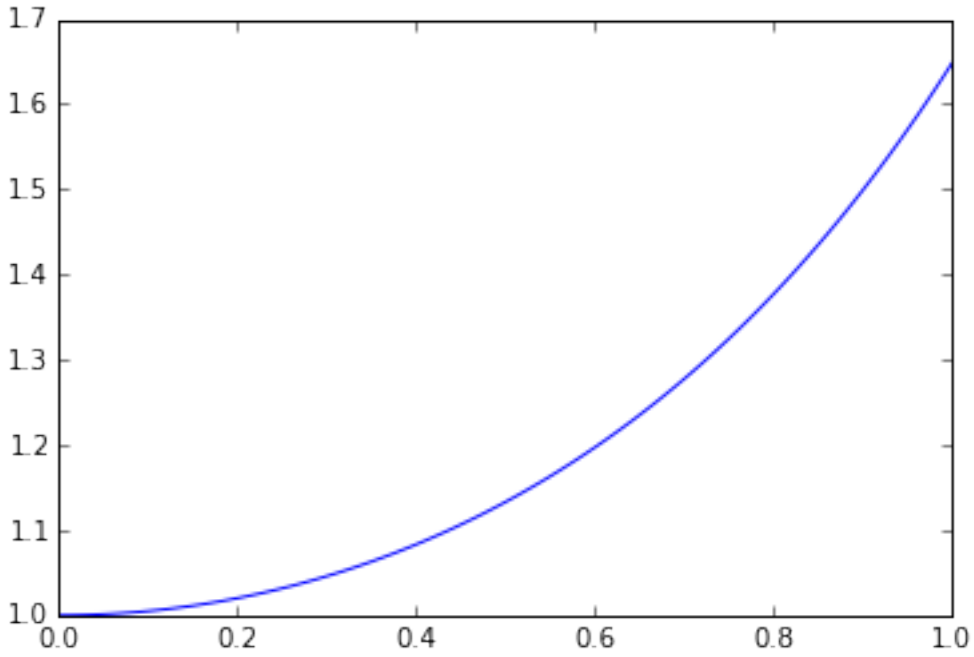
Cette fonction peut être employée pour la définition d'intégrales à paramètres. Ainsi si on veut obtenir des valeurs approchées de $\Gamma(x) = \int_0^{+\infty} e^{-t}t^{x-1}\mathrm{d}t$ pour x réel strictement positif on pourra procéder ainsi :

```
def g(x):
    def f(t):
        return np.exp(-t)*t**(x-1)
    return integr.quad(f,0,np.inf)[0]

>>> g(2)
0.9999999999999998
```

Résolution approchées d'équations différentielles

Pour résoudre une équation différentielle $x' = f(x,t)$, on peut utiliser la fonction `odeint` du module `scipy.integrate`. Cette fonction nécessite une liste de valeurs de t, commençant en t_0, et une condition initiale x_0. La fonction renvoie des valeurs approchées (aux points contenus dans la liste des valeurs de t) de la solution x de l'équation différentielle qui vérifie $x(t_0) = x_0$. Pour trouver des valeurs approchées sur $[0,1]$ de la solution $x'(t) = tx(t)$ qui vérifie $x(0) = 1$, on peut employer le code suivant.

```
def f(x, t):
    return t*x

>>> T = np.arange(0, 1.01, 0.01)
>>> X = integr.odeint(f, 1, T)
>>> X[0]
array([ 1.])
>>> X[-1]
array([ 1.64872143])
>>> plt.plot(T,X)
>>> plt.show()
```

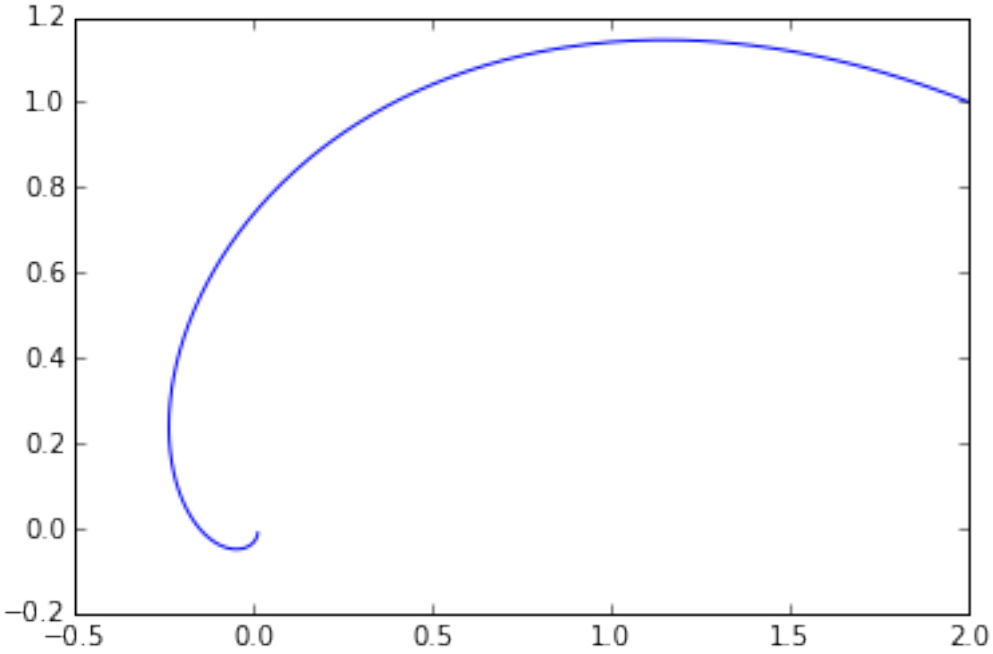


Si on veut résoudre, sur $[0,1]$, le système différentiel

$$\begin{cases} x'(t) = -x(t) - y(t) \\ y'(t) = x(t) - y(t) \end{cases}$$

avec la condition initiale $x(0) = 2, y(0) = 1$ le code devient le suivant.

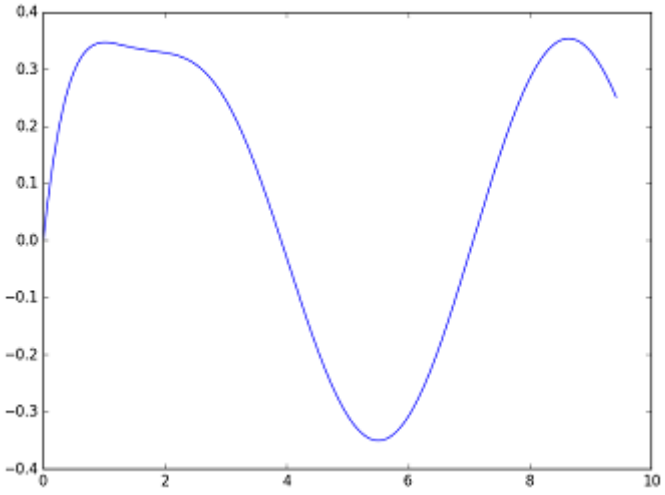
```
def f(x, t):
    return np.array([-x[0]-x[1], x[0]-x[1]])

>>> T = np.arange(0, 5.01, 0.01)
>>> X = integr.odeint(f, np.array([2.,1.]), T)
>>> X[0]
array([ 2.,  1.])
>>> plt.plot(X[:,0], X[:,1])
>>> plt.show()
```



Pour résoudre une équation différentielle scalaire d'ordre 2 de solution x, on demandera la résolution du système différentiel d'ordre 1 satisfait par $X(t) = \begin{pmatrix} x(t) \\ x'(t) \end{pmatrix}$.

Ainsi, si on considère la fonction x qui vérifie l'équation différentielle $x''(t) + 2x'(t) + 3x(t) = \sin(t)$ avec les conditions initiales $x(0) = 0$, $x'(0) = 1$ et , le vecteur X vérifiera $X'(t) = \begin{pmatrix} x'(t) \\ -2x'(t) - 3x(t) - \sin(t) \end{pmatrix}$. Pour obtenir la représentation graphique de x sur l'intervalle $[0, 3\pi]$, on pourra utiliser le code suivant :

```
def f(x,t) :
    return np.array([x[1], -2*x[1] - 3*x[0] + np.sin(t)])

T = np.arange(0, 3*np.pi + 0.01, 0.01)
X = integr.odeint(f, np.array([0,1]), T)
plt.plot(T, X[:,0])
plt.show()
```



CONCOURS CENTRALE•SUPÉLEC

Oral

Python

MP, PC, PSI, TSI

Calcul matriciel

On travaille avec les modules `numpy` et `numpy.linalg`.

```
import numpy as np
import numpy.linalg as alg
```

Création de matrices

Pour définir une matrice, on utilise la fonction `array` du module `numpy`.

```
>>> A = np.array([[1, 2, 3], [4, 5, 6]])
>>> A
array([[1, 2, 3],
       [4, 5, 6]])
```

L'attribut `shape` donne la taille d'une matrice : nombre de lignes, nombre de colonnes. On peut redimensionner une matrice, sans modifier ses termes, à l'aide de la méthode `reshape`.

```
>>> A.shape
(2, 3)
>>> A = A.reshape((3, 2))
>>> A
array([[1, 2],
       [3, 4],
       [5, 6]])
```

L'accès à un terme de la matrice `A` se fait à l'aide de l'opération d'indexage `A[i, j]` où `i` désigne la ligne et `j` la colonne. **Attention, les indices commencent à zéro !** À l'aide d'intervalles, on peut également récupérer une partie d'une matrice : ligne, colonne, sous-matrice. Rappel, `a:b` désigne l'intervalle ouvert à droite $[\![a, b[\![$, `:` désigne l'intervalle contenant tous les indices de la dimension considérée. Notez la différence entre l'indexation par un entier et par un intervalle réduit à un entier.

```
>>> A[1, 0]     # terme de la deuxième ligne, première colonne
3
>>> A[0, :]     # première ligne sous forme de tableau à 1 dimension
array([1, 2])
>>> A[0, :].shape
(2,)
>>> A[0:1, :]   # première ligne sous forme de matrice ligne
array([[1, 2]])
>>> A[0:1, :].shape
(1, 2)
>>> A[:, 1]     # deuxième colonne sous forme de tableau à 1 dimension
>>> array([2, 4, 6])
A[:, 1:2]   # deuxième colonne sous forme de matrice colonne
array([[2],
       [4],
       [6]])
>>> A[1:3, 0:2] # sous-matrice lignes 2 et 3, colonnes 1 et 2
array([[3, 4],
       [5, 6]])
```

Les fonctions `zeros` et `ones` permettent de créer des matrices remplies de 0 ou de 1. La fonction `eye` permet de créer une matrice du type I_n où n est un entier. La fonction `diag` permet de créer une matrice diagonale.

```
>>> np.zeros((2,3))
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])
>>> np.ones((3,2))
array([[ 1.,  1.],
       [ 1.,  1.],
       [ 1.,  1.]])
>>> np.eye(4)
array([[ 1.,  0.,  0.,  0.],
       [ 0.,  1.,  0.,  0.],
       [ 0.,  0.,  1.,  0.],
       [ 0.,  0.,  0.,  1.]])
>>> np.diag([1,2,3])
array([[1, 0, 0],
       [0, 2, 0],
       [0, 0, 3]])
```

Enfin la fonction `concatenate` permet de créer des matrices par blocs en superposant (`axis=0`) ou en plaçant côte à côte (`axis=1`) plusieurs matrices.

```
>>> A = np.ones((2,3))
>>> B = np.zeros((2,3))
>>> np.concatenate((A,B), axis=0)
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.],
       [ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])
>>> np.concatenate((A,B), axis=1)
array([[ 1.,  1.,  1.,  0.,  0.,  0.],
       [ 1.,  1.,  1.,  0.,  0.,  0.]])
```

Quelques méthodes ou fonctions utiles avec les tableaux Numpy

Pour copier un tableau, il est recommandé d'utiliser la méthode `copy`.

```
>>> A = np.array([[1,-2,3], [-4,5,-6]])
>>> B = A.copy()
>>> B[1, 0] = 8
>>> A
array([[ 1, -2,  3],
       [-4,  5, -6]])
>>> B
array([[ 1, -2,  3],
       [ 8,  5, -6]])
```

Les fonctions `amax`,`amin` et `mean` du module `numpy` permettent respectivement de calculer le maximum, le minimum et la moyenne des éléments d'un tableau.

```
>>> np.amax(A)
5
>>> np.amin(A)
-6
>>> np.mean(A)
-0.5
```

Enfin la commande `array_equal` permet de tester l'égalité terme à terme de deux tableaux de même taille.

```
>>> np.array_equal(A, B)
False
```

Calcul matriciel

Les opérations d'ajout et de multiplication par un scalaire se font avec les opérateurs `+` et `*`.

```
>>> A = np.array([[1,2], [3,4]])
>>> B = np.eye(2)
>>> A + 3*B
array([[ 4.,  2.],
       [ 3.,  7.]])
```

Pour effectuer un produit matriciel (lorsque que cela est possible), il faut employer la fonction `dot`.

```
>>> A = np.array([[1,2], [3,4]])
>>> B = np.array([[1,1,1], [2,2,2]])
>>> np.dot(A, B)
array([[ 5,  5,  5],
       [11, 11, 11]])
```

On peut également utiliser la méthode `dot` qui est plus pratique pour calculer un produit de plusieurs matrices. Enfin la fonction `matrix_power` du module `numpy.linalg` permet de calculer des puissances de matrices.

```
>>> A.dot(B)
array([[ 5,  5,  5],
       [11, 11, 11]])
>>> A.dot(B).dot(np.ones((3,2)))
array([[ 15.,  15.],
       [ 33.,  33.]])
>>> alg.matrix_power(A,3)
array([[ 37,  54],
       [ 81, 118]])
```

La transposée s'obtient avec la fonction `transpose`. L'expression `A.T` renvoie aussi la transposée de `A`.

```
>>> np.transpose(B)
array([[1, 2],
       [1, 2],
       [1, 2]])
>>> B.T
array([[1, 2],
       [1, 2],
       [1, 2]])
```

Le déterminant, le rang et la trace d'une matrice s'obtiennent par les fonctions `det`, `matrix_rank` du module `numpy.linalg` et `trace` du module `numpy`. Enfin la fonction `inv` du module `numpy.linalg` renvoie l'inverse de la matrice s'il existe.

```
>>> alg.det(A)
-2.0000000000000004
>>> alg.matrix_rank(A)
2
>>> np.trace(A)
5
>>> alg.inv(A)
matrix([[-2. ,  1. ],
        [ 1.5, -0.5]])
```

Pour résoudre le système linéaire $Ax = b$ lorsque la matrice A est inversible, on peut employer la fonction `solve` du module `numpy.linalg`.

```
>>> b = np.array([1,5])
>>> alg.solve(A, b)
array([ 3., -1.])
```

Éléments propres d'une matrice

La fonction `poly` du module `numpy` appliquée à une matrice carrée renvoie la liste des coefficients du polynôme caractéristique par degré décroissant.

```
>>> A = np.array([[2,-4],[1,-3]])
>>> np.poly(A)
array([ 1.,  1., -2.])
```

La fonction `eigvals` du module `numpy.linalg` renvoie les valeurs propres de la matrice.

```
>>> alg.eigvals(A)
array([ 1., -2.])
```

Pour obtenir en plus les vecteurs propres associés, il faut employer la fonction `eig`. Cette fonction renvoie un tuple constitué de la liste des valeurs propres et d'une matrice carrée. La $i^{\text{ième}}$ colonne de cette matrice est un vecteur propre associé à la $i^{\text{ième}}$ valeur de la liste des valeurs propres. Dans l'exemple ci dessous, on peut conclure que $\begin{pmatrix} 1 \\ 1 \end{pmatrix}$ est un vecteur propre de A associé à la valeur propre -2. On vérifie aussi que A est diagonalisable.

```
>>> L = alg.eig(A)
>>> L
(array([ 1., -2.]), array([[ 0.9701425,  0.70710678],
       [ 0.24253563,  0.70710678]]))
>>> L[1][:,1]
array([ 0.70710678,  0.70710678])
>>> L[1].dot(np.diag(L[0])).dot(alg.inv(L[1]))
array([[ 2., -4.],
       [ 1., -3.]])
```

Produit scalaire et produit vectoriel

La fonction `vdot` permet de calculer le produit scalaire de deux vecteurs de $\mathbb{R}^n$.

```
>>> u = np.array([1,2])
>>> v = np.array([3,4])
>>> np.vdot(u, v)
11
```

La fonction `cross` permet de calculer le produit vectoriel de deux vecteurs de $\mathbb{R}^3$.

```
>>> u = np.array([1,0,0])
>>> v = np.array([0,1,0])
>>> np.cross(u, v)
array([0, 0, 1])
```

CONCOURS CENTRALE•SUPÉLEC

Oral

Python

MP, PC, PSI, TSI

Réalisation de tracés

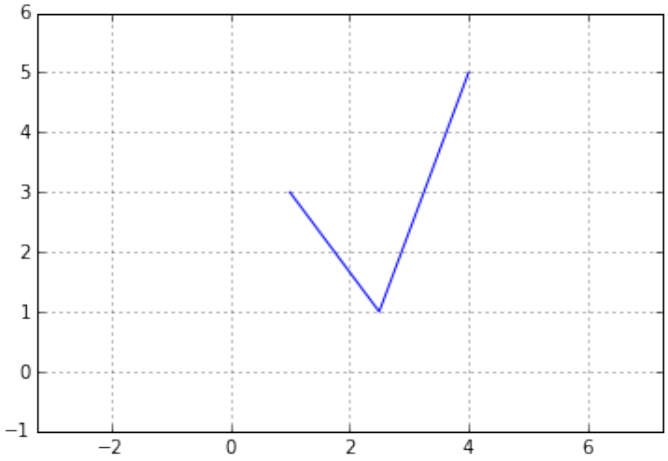
Les fonctions présentées dans ce document permettent la réalisation de tracés. Elles nécessitent l'import du module `numpy` et du module `matplotlib.pyplot`. De plus pour effectuer des tracés en dimension 3, il convient d'importer la fonction `Axes3d` du module `mpl_toolkits.mplot3d`. Les instructions nécessaires aux exemples qui suivent sont listés ci-dessous.

```
import math
import matplotlib.pyplot as plt
import numpy as np
from mpl_toolkits.mplot3d import Axes3D
```

Tracés de lignes brisées et options de tracés

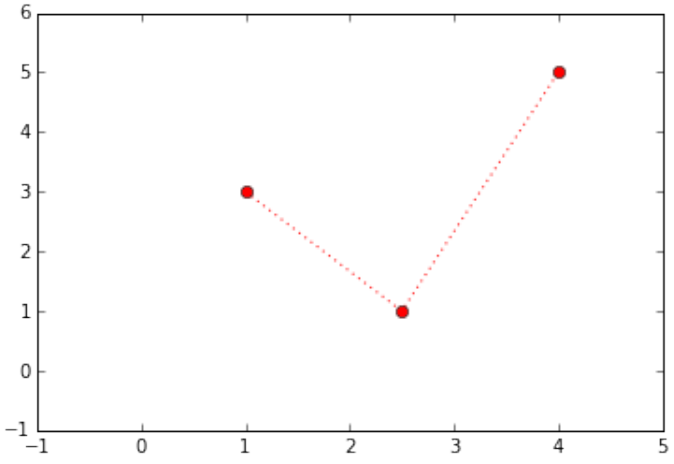
On donne la liste des abscisses et la liste des ordonnées puis on effectue le tracé. La fonction `axis` permet de définir la fenêtre dans laquelle est contenue le graphique. L'option `equal` permet d'obtenir les mêmes échelles sur les deux axes. Les tracés relatifs à divers emplois de la fonction `plot` se superposent. La fonction `plt.clf()` efface les tracés contenus dans la fenêtre graphique.

```
x = [1., 2.5, 4.]
y = [3., 1., 5.]
plt.axis('equal')
plt.plot(x, y)
plt.axis([-1., 5., -1., 6.])
plt.grid()
plt.show()
```



La fonction `plot` admet de nombreuses options de présentation. Le paramètre `color` permet de choisir la couleur (`'g'` : vert, `'r'` : rouge, `'b'` : bleu). Pour définir le style de la ligne, on utilise `linestyle` (`'-'` : ligne continue, `'--'` : ligne discontinue, `':'` : ligne pointillée). Si on veut marquer les points des listes, on utilise le paramètre `marker` (`'+'`, `'.'`, `'o'`, `'v'` donnent différents symboles).

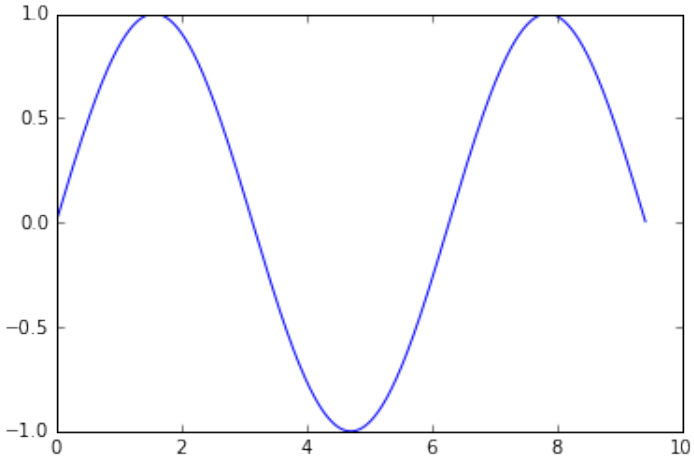
```
x = [1., 2.5, 4.]
y = [3., 1., 5.]
plt.axis([-1., 5., -1., 6.])
plt.plot(x, y, color='r', linestyle=':',
         marker='o')
plt.show()
```



Tracés de fonction

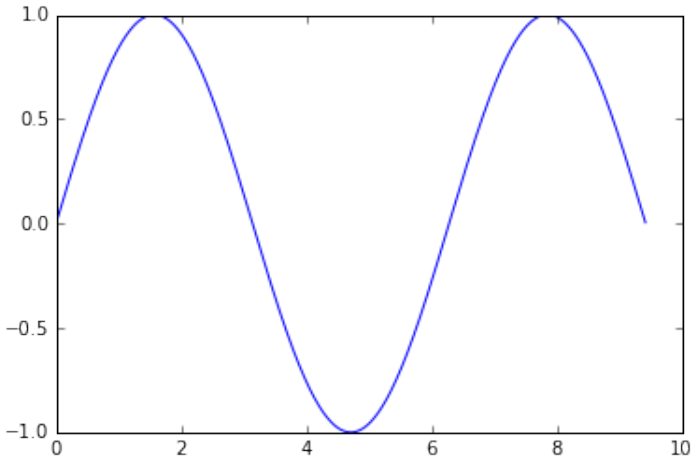
On définit une liste d'abscisses puis on construit la liste des ordonnées correspondantes. L'exemple ci-dessous trace $x \mapsto \sin x$ sur $[0, 3\pi]$.

```
def f(x):
    return math.sin(x)

X = np.arange(0, 3*np.pi, 0.01)
Y = [ f(x) for x in X ]
plt.plot(X, Y)
plt.show()
```



Il est généralement plus intéressant d'utiliser les fonctions du module `numpy`, plutôt que celles du module `math`, car elles permettent de travailler aussi bien avec des scalaires qu'avec des tableaux (on les appelle fonctions vectorisées et *universal function* ou `ufunc` dans la documentation officielle de Python).

```
def f(x):
    return np.sin(x)

X = np.arange(0, 3*np.pi, 0.01)
Y = f(X)
plt.plot(X, Y)
plt.show()
```



Une autre solution consiste à utiliser la fonction `vectorize` du module `numpy` qui permet de transformer une fonction scalaire en une fonction capable de travailler avec des tableaux. Il est cependant beaucoup plus efficace d'utiliser directement des fonctions vectorisées.

```
def f(x):
    return math.sin(x)

f = np.vectorize(f)
```

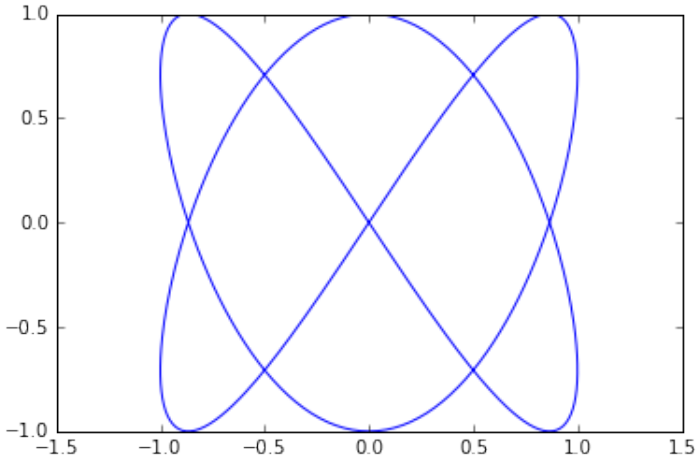
Il est à noter que les opérateurs python (`+`, `-`, `*`, etc.) peuvent s'appliquer à des tableaux, ils agissent alors terme à terme. Ainsi la fonction `f` définie ci-dessous est une fonction vectorisée, elle peut travailler aussi bien avec deux scalaires qu'avec deux tableaux et même avec un scalaire et un tableau.

```
def f(x, y):
    return np.sqrt(x**2 + y**2)

>>> f(3, 4)
5.0
>>> f(np.array([1, 2, 3]), np.array([4, 5, 6]))
array([ 4.12310563,  5.38516481,  6.70820393])
>>> f(np.array([1, 2, 3]), 4)
array([ 4.12310563,  4.47213595,  5.        ])
```

Tracés d'arcs paramétrés

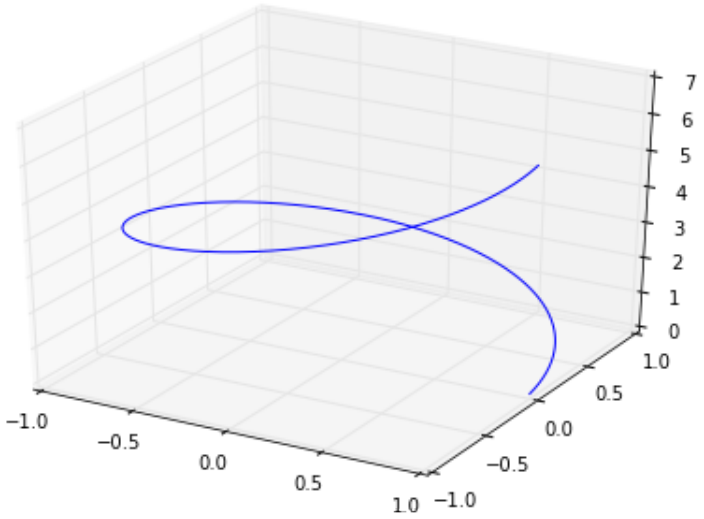
Dans le cas d'un arc paramétré plan, on définit d'abord la liste des valeurs données au paramètre puis on construit la liste des abscisses et des ordonnées correspondantes. On effectue ensuite le tracé.

```
def x(t):
    return np.sin(2*t)

def y(t):
    return np.sin(3*t)

T = np.arange(0, 2*np.pi, 0.01)
X = x(T)
Y = y(T)
plt.axis('equal')
plt.plot(X, Y)
plt.show()
```



Voici un exemple de tracé d'un arc paramétré de l'espace.

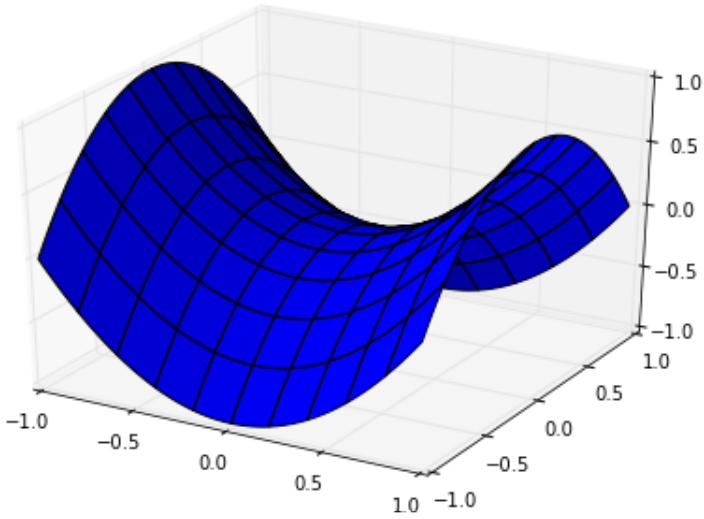
```
ax = Axes3D(plt.figure())
T = np.arange(0, 2*np.pi, 0.01)
X = np.cos(T)
Y = np.sin(T)
Z = T
ax.plot(X, Y, T)
plt.show()
```



Tracé de surfaces

Pour tracer une surface d'équation $z = f(x, y)$, on réalise d'abord une grille en (x, y) puis on calcule les valeurs de z correspondant aux points de cette grille. On fait ensuite le tracé avec la fonction `plot_surface`.

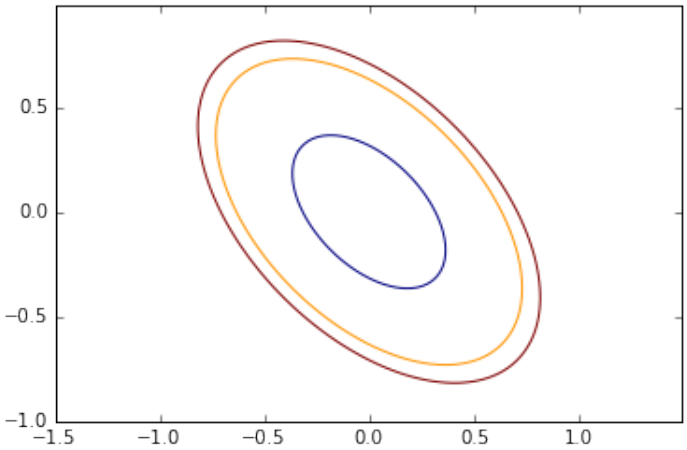
```
ax = Axes3D(plt.figure())

def f(x,y):
    return x**2 - y**2

f=np.vectorize(f)
X = np.arange(-1, 1, 0.02)
Y = np.arange(-1, 1, 0.02)
X, Y = np.meshgrid(X, Y)
Z = f(X, Y)
ax.plot_surface(X, Y, Z)
plt.show()
```



Tracé de lignes de niveau

Pour tracer des courbes d'équation $f(x,y) = k$, on fait une grille en x et en y sur laquelle on calcule les valeurs de f. On emploie ensuite la fonction `contour` en mettant dans une liste les valeurs de k pour lesquelles on veut tracer la courbe d'équation $f(x,y) = k$.

```
def f(x,y):
    return x**2 + y**2 + x*y

f=np.vectorize(f)
X = np.arange(-1, 1, 0.01)
Y = np.arange(-1, 1, 0.01)
X, Y = np.meshgrid(X, Y)
Z = f(X, Y)
plt.axis('equal')
plt.contour(X, Y, Z, [0.1,0.4,0.5])
plt.show()
```



CONCOURS CENTRALE•SUPÉLEC

Oral

Python

MP, PC, PSI, TSI

Polynômes

La classe `Polynomial` du module `numpy.polynomial.polynomial` permet de travailler avec des polynômes.

```
from numpy.polynomial import Polynomial
```

Pur créer un polynôme, il faut lister ses coefficients par ordre de degré croissant. Par exemple, pour le polynôme $X^3 + 2X - 3$,

```
p = Polynomial([-3, 2, 0, 1])
```

On peut alors utiliser cette variable comme une fonction pour calculer, en un point quelconque, la valeur de la fonction polynôme associée. Cette fonction peut agir également sur un tableau de valeurs, elle calcule alors la valeur de la fonction polynôme en chacun des points indiqués.

```
>>> p(0)
-3.0
>>> p([1, 2, 3])
array([  0.,   9.,  30.])
```

L'attribut `coef` donne accès aux coefficients ordonnés par degré croissant ; ainsi `p.coef[i]` correspond au coefficient du terme de degré `i`. La méthode `degree` renvoie le degré du polynôme alors que `roots` calcule ses racines.

```
>>> p.coef
array([-3.,  2.,  0.,  1.])
>>> p.coef[1]
2.0
>>> p.degree()
3
>>> p.roots()
array([-0.5-1.6583124j, -0.5+1.6583124j,  1.0+0.j       ])
```

La méthode `deriv` renvoie un nouveau polynôme, dérivé du polynôme initial. Cette méthode prend en argument facultatif un entier positif indiquant le nombre de dérivations à effectuer. De la même manière la méthode `integ` intègre le polynôme, elle prend un paramètre optionnel supplémentaire donnant la constante d'intégration à utiliser, ce paramètres peut être une liste en cas d'intégration multiple ; les constantes d'intégration non précisées sont prises égales à zéro.

```
>>> p.deriv().coef
array([ 2.,  0.,  3.])
>>> p.deriv(2).coef
array([ 0.,  6.])
>>> p.deriv(5).coef
array([-0.])
>>> p.integ().coef
array([ 0.  , -3.  ,  1.  ,  0.  ,  0.25])
>>> p.integ(1, 2).coef  # intégrer une fois avec la constante 2
array([ 2.  , -3.  ,  1.  ,  0.  ,  0.25])
>>> p.integ(2, [1, 2]).coef  # intégrer deux fois
array([ 2.        ,  1.        , -1.5       ,  0.33333333,  0.        ,
        0.05      ])
```

Les opérateurs `+`, `-`, `*` permettent d'additionner, soustraire et multiplier des polynômes. Ils fonctionnent également entre un polynôme et un scalaire. L'opérateur `**` permet d'élever un polynôme à une puissance entière positive. Enfin, on peut composer deux polynômes (`p(q)` remplace l'indéterminée X par le polynôme `q` dans le polynôme `p`)

```
>>> a = Polynomial([1, 2, 1])
>>> b = Polynomial([5, 3])
>>> p = 2*a * b + Polynomial([-7, 2])
>>> p.coef
array([  3.,  28.,  22.,   6.])
>>> (p**2).coef
array([   9.,  168.,  916., 1268.,  820.,  264.,   36.])
>>> a(b).coef
array([ 36., 36., 9.])
```

L'opérateur `/` permet de diviser un polynôme par un scalaire. Pour diviser deux polynômes il faut utiliser l'opérateur `//` qui renvoie le quotient ; l'opérateur `%` calcule le reste.

```
>>> (p / 2).coef
array([  1.5,  14. ,  11. ,   3. ])
>>> q = p // a
>>> r = p % a
>>> q.coef
array([ 10.,   6.])
>>> r.coef
array([-7.,  2.])
>>> (q * a + r).coef
array([  3.,  28.,  22.,   6.])
```

CONCOURS CENTRALE•SUPÉLEC

Oral

Python

MP, PC, PSI, TSI

Probabilités

Les fonctions d'échantillonnage et de génération de valeurs pseudo-aléatoires sont regroupées dans la bibliothèque `numpy.random`.

```
import numpy.random as rd
```

L'expression `randint(a, b)` permet de choisir un entier au hasard dans l'intervalle $[\![a, b[\![$. La fonction `randint` prend un troisième paramètre optionnel permettant d'effectuer plusieurs tirages et de renvoyer les résultat sous forme de tableau ou de matrice.

```
>>> rd.randint(1, 7)          # un lancer de dé
2
>>> rd.randint(1, 7, 20)      # 20 lancers de dé
array([5, 2, 2, 3, 1, 5, 5, 3, 6, 4, 2, 6, 6, 4, 3, 2, 4, 5, 1, 3])
>>> rd.randint(1, 7, (4, 5))  # 20 lancers de dé sous forme d'une matrice 4x5
array([[3, 6, 1, 6, 3],
       [5, 1, 6, 2, 2],
       [3, 1, 2, 2, 5],
       [5, 2, 6, 1, 4]])
```

La fonction `random` renvoie un réel compris dans l'intervalle $[0, 1[$. Si X désigne la variable aléatoire correspondant au résultat de la fonction `random`, alors pour tout a et b dans $[0, 1]$ avec $a \leqslant b$, on a $P(a \leqslant X < b) = b - a$. Cette fonction accepte un paramètres optionnel permettant de réaliser plusieurs tirages et de les renvoyer sous forme de tableau ou de matrice.

```
>>> rd.random()
0.9168092013708049
>>> rd.random(4)
array([ 0.98748897,  0.86589972,  0.53683001,  0.50687386])
>>> rd.random((2,4))
array([[ 0.78230688,  0.83803526,  0.62077457,  0.27432819],
       [ 0.66522387,  0.71258365,  0.25813448,  0.28833084]])
```

La fonction `binomial` permet de simuler une variable aléatoire suivant une loi binomiale de paramètres `n` et `p`. Elle permet donc également de simuler une variable aléatoire suivant une loi de Bernoulli de paramètres `p` en prenant simplement $n = 1$. Cette fonction prend un troisième paramètre optionnel qui correspond, comme pour les fonctions précédentes, au nombre de valeurs à obtenir.

```
>>> rd.binomial(10, 0.3, 7)
array([2, 2, 2, 2, 2, 4, 3])
>>> rd.binomial(1, 0.6, 20)
array([0, 1, 1, 1, 1, 0, 1, 1, 0, 0, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1])
```

Les fonctions `geometric` et `poisson` fonctionnement de la même manière pour les lois géométrique ou de Poisson.

```
>>> rd.geometric(0.5, 8)
array([1, 1, 3, 1, 3, 2, 5, 1])
>>> rd.poisson(4, 15)
array([5, 2, 3, 4, 6, 0, 5, 3, 1, 5, 1, 5, 9, 4, 6])
```