Matplotlib

Table des matières

Matplotlib est un module de Python qui permet de dessiner des courbes en deux et trois dimensions. Il s'agit d'une bibliothèque très riche qui génère des figures que vous pouvez enregistrer sous les formats **.png** (fichier image) ou **.pdf** (portable document format).

1 Graphiques 2 D - Méthodes de base

1.1 Premiers pas

Pour utiliser **matplotlib**, il faut d'abord l'importer. Comme tous les modules de Python, la syntaxe à utiliser est **import**. Le module intéressant est en fait un sous-module de matplotlib qui se nomme **matplolib.pyplot** et que je vous conseille d'importer sous la forme :

import matplotlib.pyplot **as** plt

c'est à dire en utilisant un *alias* qui rebaptise ce module **plt** (vous pouvez bien sûr choisir un autre nom, mais plt est la convention qui, désormais, revient le plus souvent). Une fois importé sous ce nom, les différentes fonctions et constantes de ce module s'appelleront en faisant précéder leur *nom* de **plt.**, comme par exemple : **plt.figure**().

Une des fonctions les plus utilisées est **plot**. Dans sa forme la plus simple, elle prend en paramètre deux listes numériques ou encore deux tableaux unidimensionnels numpy [1], X et Y de même dimension et de la forme :

$$\mathrm{X} = [\, x_1, x_2, ..., x_n \,] \quad \text{et} \quad \mathrm{Y} = [\, y_1, y_2, ..., y_n \,]$$

La fonction **plot** considère alors que les éléments de X et Y sont les *abscisses* et les *ordonnées* de n points du plan (Oxy), de la forme : (x_1, y_1), (x_2, y_2), ..., (x_n, y_n). Elle place ces points et les relie par un trait. Voilà, c'est tout !

Bien entendu, les x_i et y_i doivent avoir le *bon format* : il ne peut s'agir que d'entiers ou de nombres réels.

En voici un premier exemple. Nous allons tracer le triangle dont les trois sommets ont pour coordonnées (0,0), (2,0) et (1,3). Comme plot relie uniquement un point (x_k, y_k) et son successeur (x_{k+1}, y_{k+1}), il va donc falloir lui passer dans l'ordre les quatre points suivants : (0,0), (2,0), (1,3) et (0,0) pour refermer le triangle.

Il faut donc les deux listes : X = [0, 2, 1, 0] et Y = [0, 0, 3, 0]. Dans *Pyzo*, vous allez donc créer un fichier programme, nommé graphique.py par exemple et qui contiendra les lignes suivantes :

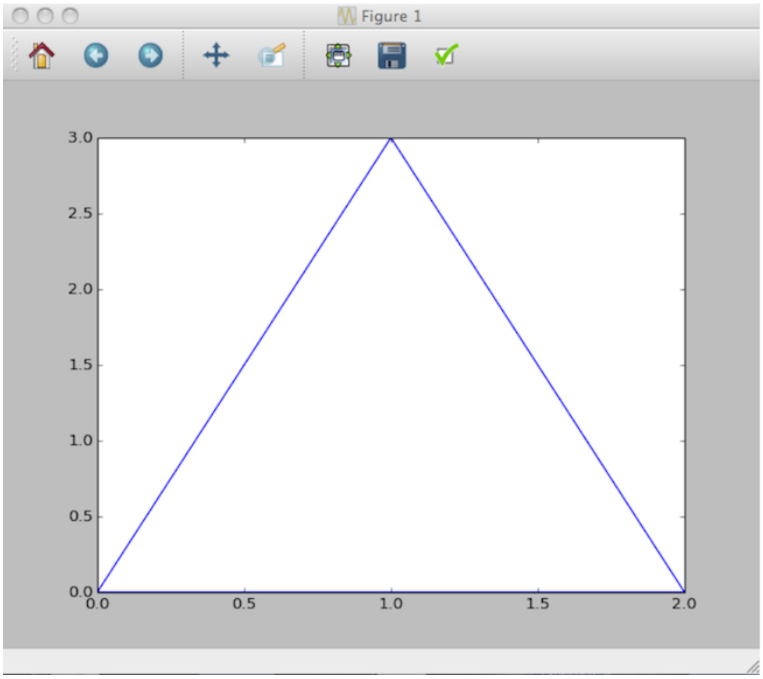
```
1  import matplotlib.pyplot as plt
2  X = [0, 2, 1, 0]
3  Y = [0, 0, 3, 0]
4  plt.figure()      # début de la figure
5  plt.plot(X, Y)   # on trace la figure
```

Dans certains environnements autres que *Pyzo*, il faut placer l'instruction **plt.show**() à la fin pour afficher la fenêtre graphique. Avec *Pyzo*, ce n'est pas la peine.

1. Vous pouvez utiliser une liste numérique ou un tableau numpy : cela n'a pas d'importance et plot se débrouille très bien avec l'un ou l'autre.

Le début de la construction de la fenêtre graphique commence avec l'instruction **plt.figure**(). Toutes les instructions de la forme **plt.***quelquechose*(...) qui suivront seront interprétées comme des instructions de construction du dessin à l'intérieur de cette fenêtre graphique [2] ...jusqu'à ce que Python arrive sur une autre instruction **plt.figure**() ou à la fin du programme.

En lançant l'exécution de votre programme, voici le résultat que vous devriez obtenir sur votre écran :



- Une fenêtre graphique portant le nom Figure 1 (tout en haut de la fenêtre) et contenant votre dessin s'affiche sur l'écran de votre ordinateur.

- **matplotlib** crée un système d'axes Ox et Oy qu'il optimize pour son tracé. Il n'en fait pas plus qu'il n'en faut pour que le dessin soit contenu dans les limites de ces axes.

- Cette fenêtre contient un menu dont une des options est : *Enregistrer*. Vous pouvez alors enregistrer votre image sous l'un des formats proposés, .png ou .pdf par exemple.

- Si vous relancez l'exécution de votre programme sans fermer cette première fenêtre, une nouvelle fenêtre va s'afficher portant le nom Figure 2 et contenant le même dessin. Et ainsi de suite...Tant que vous ne les refermez pas, les fenêtres graphiques s'empilent sur votre écran, chacune avec son numéro.

1.2 Donner un nom ou un numéro à votre figure

Vous pouvez passer un **entier** ou une **chaîne de caractères** comme paramètre à la fonction **plt.figure**(...). Cela aura pour effet d'afficher Figure *numéro* [3] ou encore la chaine de caractères en haut de la fenêtre graphique. Dans ce cas, relancer le programme sans fermer la fenêtre ne

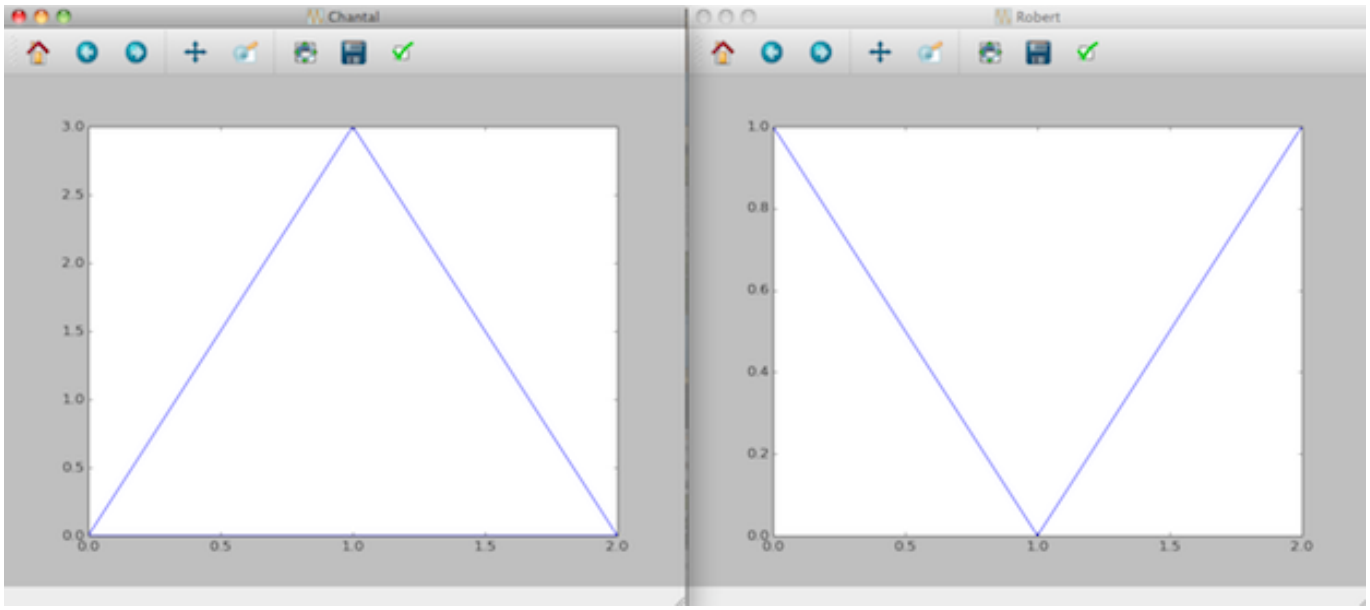
2. Vous pouvez bien sûr continuer à mettre du code Python non graphique, du genre a = 65 par exemple, après l'instruction **plt.figure**(). Seules les instructions commençant par **plt.** serviront à construire la fenêtre graphique.
3. Par exemple l'instruction **plt.figure**(1) aura pour effet d'afficher Figure 1 en haut de la fenêtre graphique.

permet pas d'en afficher plusieurs : c'est toujours la même fenêtre qui est affichée et le seul effet est que la zone graphique y soit re-dessinée, avec éventuellement une couleur différente.

Cela a un avantage quand vous souhaitez afficher plusieurs fenêtres graphiques avec des titres différents et choisis par vous-même, pour bien les identifier. Par exemple :

```
1   # On affiche deux fenêtres graphiques
2   import matplotlib.pyplot as plt
3   X = [0, 2, 1, 0]
4   Y = [0, 0, 3, 0]
5   Z = [1, 1, 0, 1]
6
7   plt.figure("Chantal")       # début de la figure "Chantal"
8   plt.plot(X, Y)    # on trace la figure
9   plt.figure("Robert")       # début de la figure "Robert"
10   plt.plot(X, Z)    # dessin de la figure
```

En exécutant ce programme, vous verrez apparaître les deux fenêtres graphiques ci-dessous, chacune bien identifiable par son titre tout en haut :



Il existe encore bien d'autres paramètres que vous pouvez passer à la fonction **plt.figure**, notamment sa taille, la couleur du contour, etc...mais je n'en dirai pas plus ici. Si cela vous intéresse, faites help(plt.figure) dans l'interpréteur ou consultez l'aide en ligne.

1.3 Tracer des courbes

a) Utiliser numpy

Venons en maintenant au tracé de courbes qui représentent des fonctions $f : x \mapsto f(x)$. Le module **numpy** est ici très efficace pour réaliser cela. Vous pouvez l'importer en tapant :

import numpy **as** np

Ce module dispose :

- d'une fonction **linspace** dont la syntaxe est :

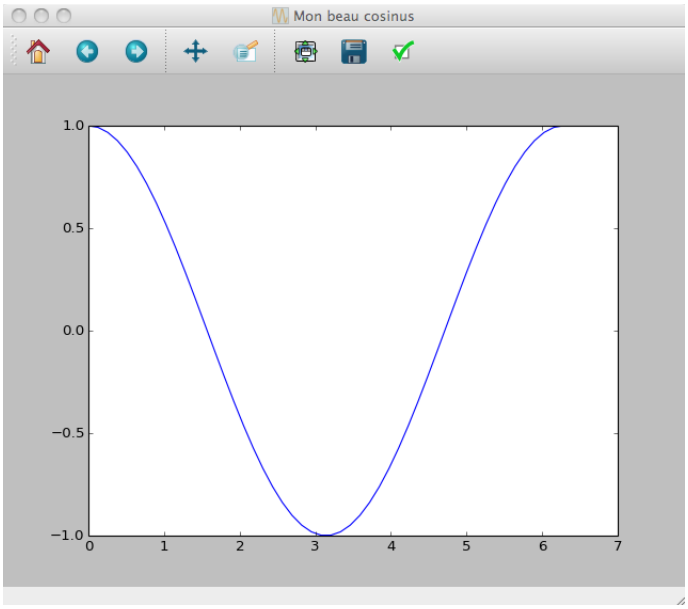
  linspace(*debut*, *fin*, *n*)

  et qui renvoie un tableau unidimensionnel commençant à *debut*, se terminant à *fin* (inclu...une fois n'est pas coutume) et contenant exactement *n* éléments. C'est très efficace pour générer les abscisses ! (Vous pouvez aussi utiliser **np.arange**(*debut*, *fin*, *pas*) si vous voulez) ;

- de fonctions spéciales qui travaillent directement sur les tableaux numpy. Par exemple, si X = np.linspace(0, 3.14, 10), alors Y = **np.cos**(X) est un tableau unidimensionnel de 10 éléments qui contient les cosinus des éléments de X.

```
1  import matplotlib.pyplot as plt
2  import math as m    # pour disposer de la constante pi
3  import numpy as np
4  X = np.linspace(0, 2*m.pi, 50)
5  Y = np.cos(X)
6  plt.figure("Mon beau cosinus")
7  plt.plot(X, Y)
```

qui donne :



Les différents points de la courbe précédente sont bien reliés par des segments de droite, mais leur grand nombre donne l'illusion d'une courbe très lisse (essayez pour voir en donnant $n = 10$ points à linspace à la place de 50).

b) Utiliser ses propres fonctions

numpy dispose d'un stock de fonctions assez étendu dont vous pouvez consulter la liste et les fonctionnalités sur internet.

Cependant, il ne peut y avoir toutes les fonctions dont vous rêvez ! Il faut bien en créer par soi-même, ce que vous pouvez faire simplement en définissant vos propres fonctions. Prenons par exemple : $f : x \mapsto \dfrac{x}{1+x^2}$. On peut toujours la définir par :

```
1  def f(x) :
2      y = x / (1 + x**2)
3      return y
```

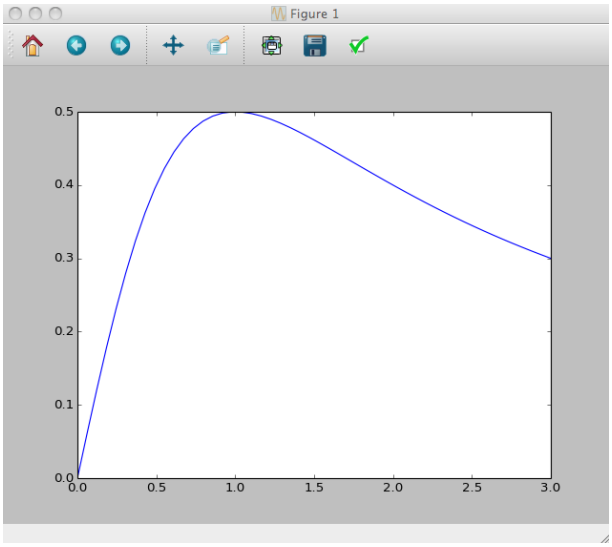
Les abscisses des points peuvent ensuite être créées très facilement à l'aide du tableau numpy X = **np.linspsace**(*debut*, *fin*, *n*).

Votre fonction f s'attend à trouver un paramètre x entier, réel ou peut être complexe, bref une valeur avec un type pour lequel les opérations / et ** sont bien définies. En y réfléchissant, vous pourrez penser que ce n'est pas le cas avec les tableaux numpy et que taper Y = f(X) dans votre programme va générer une erreur. En fait, cela dépend des versions de Python :

- En général, c'est bien le cas ! Python s'arrête tout de suite et se fâche très fort.
- Cependant, avec *Pyzo* et la version 3 de Python, çà fonctionne et tout se passe comme si la fonction savait automatiquement gérer les tableaux passés en paramètre. Une instruction de la forme Y = f(X) va créer le tableau Y contenant les valeurs f(xi) pour tous les xi dans X.

Vous pourrez donc écrire, à la suite des 3 premières lignes :

```
4  import numpy as np
5  import matplotlib.pyplot as plt
6  X = np.linspace(0, 3, 50)
7  Y = f(X)
8  plt.figure(1)
9  plt.plot(X, Y)
```



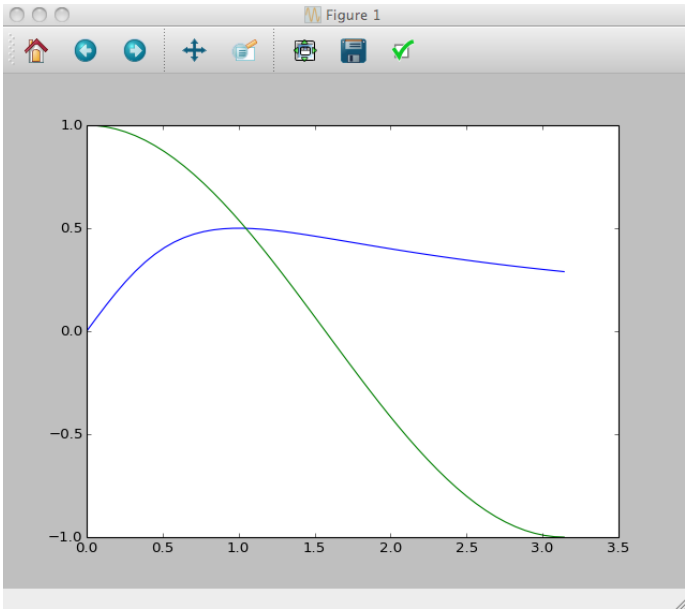
Si toutefois vous êtes amenés à travailler sur une version de Python qui n'accepte pas la syntaxe Y = f(X) directement, vous pouvez toujours définir Y grâce à la technique des *compréhensions de listes*. Cela donne :

```
...
X = np.linspace(0, 3, 50)
Y = [ f(x) for x in X ]    # Y est une liste générée par compréhension
...
```

Cette syntaxe marchera très bien.

Une dernière chose avant de passer à la suite : vous pouvez très bien superposer plusieurs courbes à l'intérieur d'une même fenêtre graphique. Il suffit pour cela d'écrire plusieurs **plt.plot**(...)

```
1   def f(x) :
2       y = x / (1 + x**2)
3       return y
4
5   import numpy as np
6   import math as m
7   import matplotlib.pyplot as plt
8   X = np.linspace(0, m.pi, 50)
9   Y = f(X)
10   Z = np.cos(X)
11   plt.figure(1)
12   plt.plot(X, Y)
13   plt.plot(X, Z)
```



1.4 Gérer le style du tracé

La fonction **plot** peut prendre un troisième paramètre qui gère le style du tracé du dessin, c'est à dire la *couleur* et la façon dont les *points sont reliés*. Il s'agit d'une chaîne de caractères, avec les valeurs suivantes :

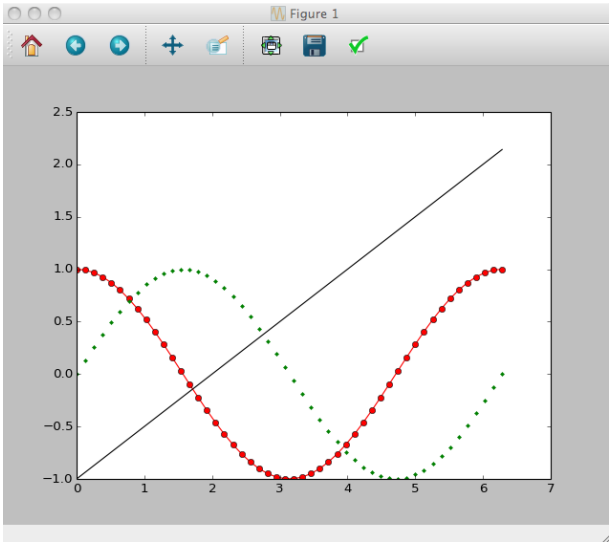
Couleur	**Chaîne**	**Style**	**Chaîne**
bleu	'b'	Points non reliés	'.'
rouge	'r'	Gros points non reliés	'o'
noir	k	Reliés par un segment	'-'
vert	'g'		
cyan	'c'		
magenta	'm'		
jaune	'y'		

Vous pouvez *combiner les différents caractères dans une même chaîne* pour préciser le tracé. Par exemple "ro-" signifie que la courbe doit être tracée en rouge, avec des gros points reliés entre eux par des segments de droite. "g" signifie que la courbe sera tracée en vert avec des points non affichés et reliés par des segments (comportement par défaut).

Le caractère de couleur doit toujours être en premier. Par défaut (si vous n'indiquez rien à plot), aucun point n'est "marqué", python utilise des segments de droite pour relier les points et choisit la couleur à votre place.

Voici ce que cela donne avec nos bonnes vieilles fonctions cosinus, sinus et une fonction affine :

```
import numpy as np
import math as m
import matplotlib.pyplot as plt

X = np.linspace(0, 2*m.pi, 50)
Y = np.cos(X)
Z = np.sin(X)
W = 0.5*X - 1

plt.figure(1)
plt.plot(X, Y, "ro-" )    # Courbe en rouge avec de gros points reliés par des segments
plt.plot(X, Z, "g.')      # Courbe en vert avec des points non reliés
plt.plot(X, W, "k')       # courbe en noir. Points non affichés mais reliés par des segments
```

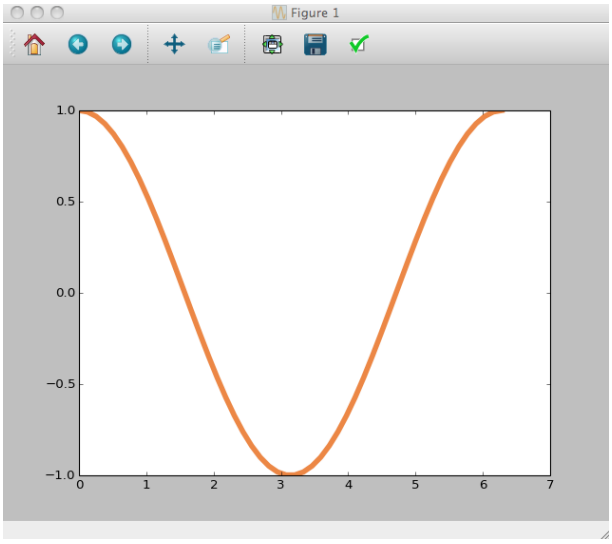


Si vous voulez paramétrer plus finement les couleurs, vous pouvez passer à **plot** un paramètre de la forme color = (r, v, b) qui est un **tuple** dont les trois éléments r, v et b sont des nombres réels appartenant à l'intervalle [0,1]. Ces trois valeurs représentent les composantes RVB d'une couleur que vous pouvez obtenir à l'aide de logiciels comme *paint* par exemple. Normalement les composantes R, V et B sont des entiers compris entre 0 et 255 et la correspondance se fait comme ci-dessous :

$$r = \frac{R}{255} \qquad v = \frac{V}{255} \qquad b = \frac{B}{255}$$

Vous pouvez aussi préciser l'épaisseur du trait grâce au paramètre **linewidth** = *numero*. Plus *numero* est grand, plus le trait est épais.

```
1  import numpy as np
2  import math as m
3  import matplotlib.pyplot as plt
4  X = np.linspace(0, 2*m.pi, 50)
5  Y = np.cos(X)
6  plt.figure(1)
7  plt.plot(X, Y, color = (0.536, 0.307, 0.157), linewidth=5)
```



1.5 Redéfinir les axes

Par défaut, Python choisit les axes à votre place. Vous pouvez cependant redéfinir leur étendue à l'aide de la fonction **axis**, ajouter une grille sur le schéma à l'aide de la fonction **grid** et ajouter un label à chaque axe grâce aux fonctions **xlabel** et **ylabel**.

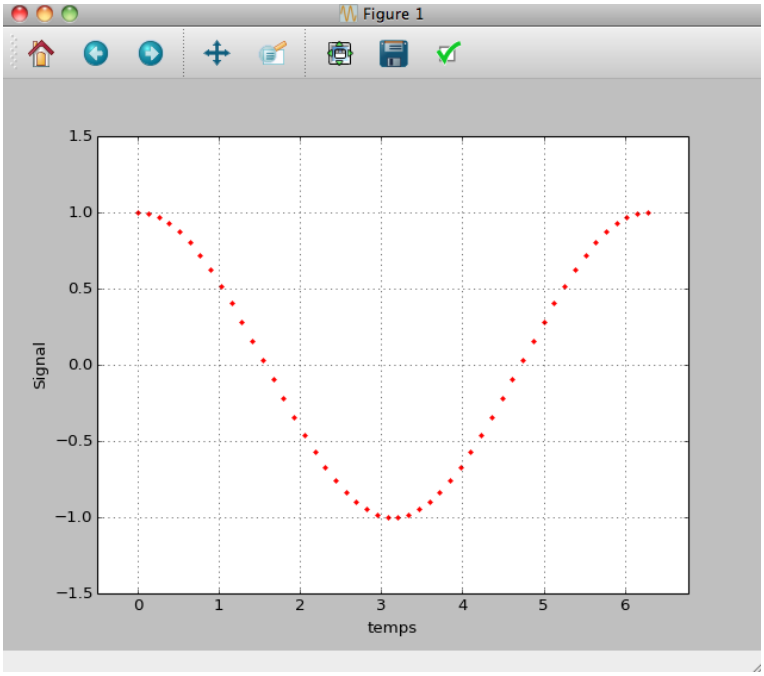
- Si vous avez importé matplotlib.pyplot sous le nom de plt, la syntaxe de axis est : **plt.axis**(L), où L est une liste de la forme L = $[x_{\min}, x_{\max}, y_{\min}, y_{\max}]$ qui contient les valeurs minimales et maximales des deux axes Ox et Oy.

- **plt.grid**() ne prend pas de paramètre.

- Les syntaxes de xlabel et de ylabel sont : **plt.xlabel**(chx) et **plt.ylabel**(chy) où **chx** et **chy** sont des chaines de caractères qui seront affichées à côtés des deux axes.

```
1  import numpy as np
2  import math as m
3  import matplotlib.pyplot as plt
4  X = np.linspace(0, 2*m.pi, 50)
5  Y = np.cos(X)
6  plt.figure(1)
7  plt.plot(X, Y, "ro-" )
8  plt.axis([-0.5, 2*m.pi+0.5, -1.5, 1.5])
9  plt.grid()
10  plt.xlabel("temps")
11  plt.ylabel("Signal")
```

ce qui donne la figure ci-dessous :



1.6 Ajouter du texte et un titre

Vous pouvez ajouter un *titre* à votre dessin (ou graphique si vous préférez) qui viendra se placer juste au dessus de la zone de dessin (zone blanche dans les exemples), à ne pas confondre avec le titre de votre fenêtre graphique. La fonction à utiliser est **title**.

Vous pouvez aussi placer du texte dans votre dessin grâce à la fonction **text**.

- Pour ajouter un titre, il suffit de taper : **plt.title**(*nom_titre*) où *nom_titre* est une chaine de caractères qui sera affichée comme le titre de votre graphique.

- Placer du texte dans le dessin se fait grâce à : **plt.text**(x, y, *text*, color = ...). *text* est une chaîne de caractères qui sera le texte affiché, x et y sont les coordonnées du premier caractère de *text*, relativement au repère (Oxy) du dessin et **color** est le paramètre de couleur.

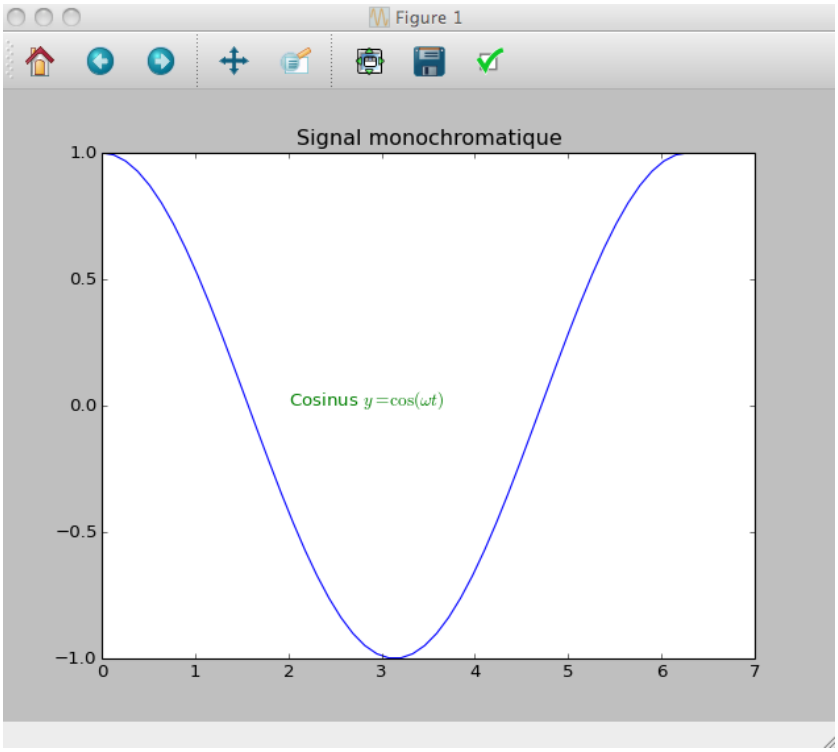
  Si vous voulez une couleur simple, du rouge par exemple, il faut taper : **color** = "r" (les caractères de couleur sont donnés dans le tableau de la page 8). Si vous voulez plus de choix de couleur, il faut taper **color** = (r, v, b) de la même façon que vous précisez la couleur à la fonction **plot** (voir pages 8 et 9).

  La chaine de caractères **text** qui sera affichée peut même contenir du code LaTeX. Par exemple, vous pouvez avoir text = "Cosinus : $y = \cos(\omega t)$".

```
1 import numpy as np
2 import math as m
3 import matplotlib.pyplot as plt
4 X = np.linspace(0, 2*m.pi, 50)
5 Y = np.cos(X)
6 plt.figure(1)
7 plt.plot(X, Y )
8 plt.title("Signal monochromatique")
```

```
9  plt.text(2, 0, "Cosinus : $y = \cos(\omega t)$", color = 'g')
```

ce qui donne :



1.7 Plusieurs graphiques dans une même fenêtre

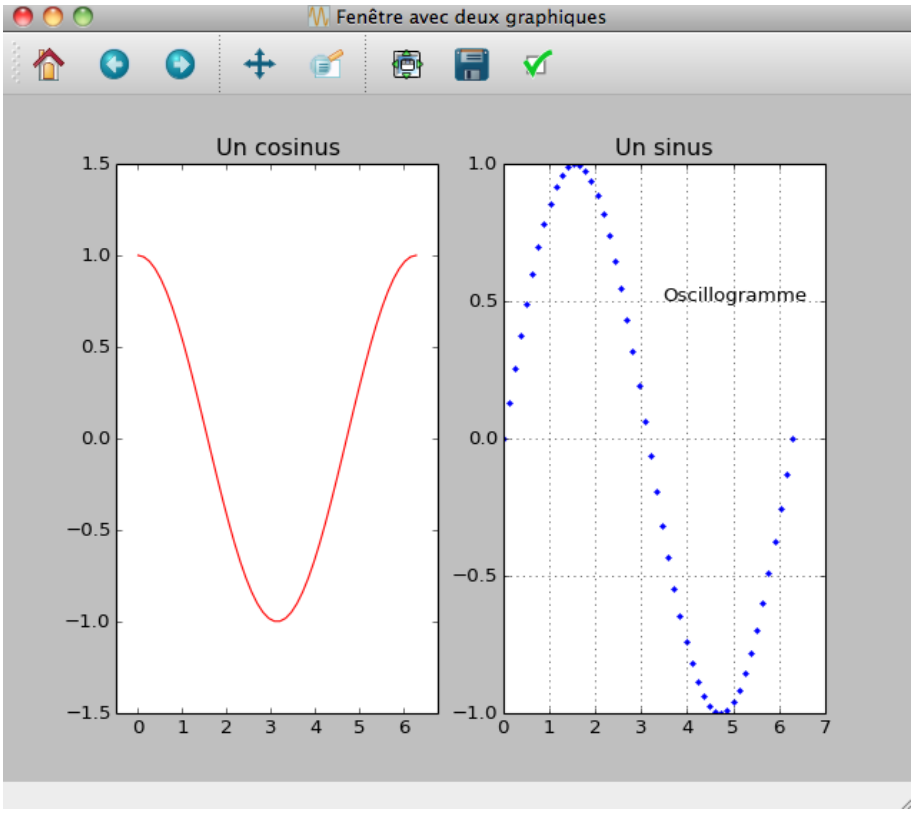
Une fenêtre graphique peut contenir plusieurs graphiques. Dans ce cas chacun d'entre eux est caractérisé par un *numero*. Il faut aussi préciser si on désire afficher les graphiques ...

- les uns à côté des autres, c'est à dire sur une seule ligne et plusieurs colonnes ;
- ou les uns en-dessous des autres, ie sur plusieurs lignes et une seule colonne ;
- ou encore sur plusieurs lignes et plusieurs colonnes.

La fonction **subplot** est là pour gérer çà. Un sous-graphique d'une fenêtre est introduit par **subplot**(*lignes*, *colonnes*, *numero*) où *lignes* et *colonnes* sont les nombres totaux de lignes et de colonnes utilisées pour afficher tous les graphiques de la fenêtre et où *numero* désigne le numéro du graphique qu'on va afficher. Restons concrets en voyant cela sur un exemple :

```
1  import numpy as np
2  import math as m
3  import matplotlib.pyplot as plt
4  X = np.linspace(0, 2*m.pi, 50)
5  Y = np.cos(X)
6  Z = np.sin(X)
7  plt.figure("Fenêtre avec deux graphiques")    # début de la fenêtre graphique
8
9  plt.subplot(1, 2, 1)     # Deux graphiques sur 1 ligne et 2 colonnes. Début du graph. n°1
10   plt.plot(X, Y, 'r')
```

```
11  plt.axis([-0.5, 2*m.pi + 0.5, -1.5, 1.5])
12  plt.title("Un cosinus")
13
14  plt.subplot(1, 2, 2)    # 2ème graph., 1 ligne et 2 colonnes. Il sera placé sur la colonne 2.
15  plt.plot(X, Z, 'b.')
16  plt.grid()
17  plt.title("Un sinus")
18  plt.text(3.5 , 0.5 , "Oscillogramme", color = 'k')
```



2 Courbes paramétrées et polaires

On donne ici des techniques permettant de tracer des courbes paramétrées et polaires dans le plan (Oxy).

2.1 Courbes paramétrées

Une *courbe paramétrée* consiste à se donner les deux coordonnées cartésiennes x et y d'un point M comme des fonctions d'une variable réelle t appelée *paramètre* de la courbe. Nous aurons donc :

$$\begin{cases} x &= f(t) \\ y &= g(t) \end{cases}$$

Ces courbes se rencontrent fréquemment en physique et t représente le temps : les fonctions $x = f(t)$ et $y = g(t)$ sont alors les lois horaires du mouvement. Dans le cas général, le paramètre t est une variable réelle quelconque, qui ne représente pas forcément le temps.

Afin de tracer ces courbes avec matplotlib, il faut d'abord créer une liste ou un tableau numpy 1D **T** dont les éléments sont les différentes valeurs de t. On construit ensuite les listes

ou tableaux numpy 1D **X** et **Y** contenant toutes les valeurs $x = f(t)$ et $y = g(t)$ associées. Pour tracer la courbe, il suffit ensuite d'utiliser la fonction **plot** en lui passant **X** et **Y** en paramètres.

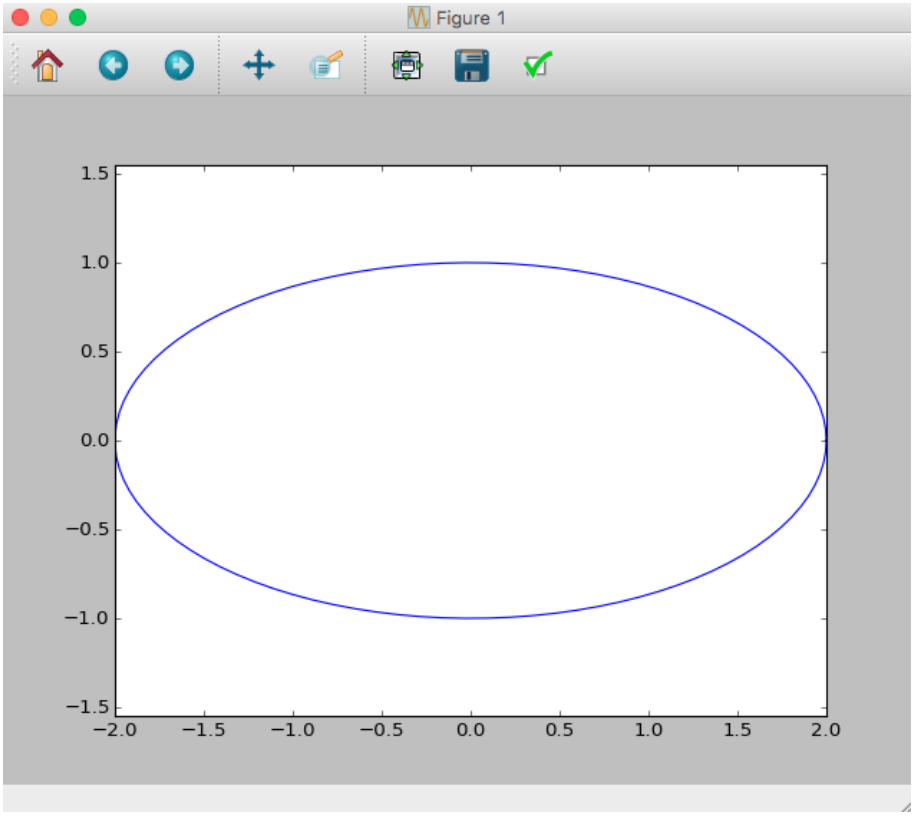
Prenons l'exemple d'une ellipse, qui peut être représentée par une courbe paramétrée de la forme :

$$\begin{cases} x &= a\cos(t) \\ y &= b\sin(t) \end{cases} \quad \Longrightarrow \quad \frac{x^2}{a^2} + \frac{y^2}{b^2} = 1$$

On peut la programmer comme suit :

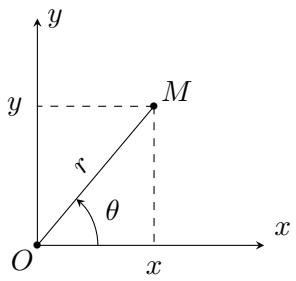
```
import numpy as np
import matplotlib.pyplot as plt

T = np.linspace(0, 2*np.pi, 100)      # liste des valeurs du paramètre
a = 2
b = 1
X = a*np.cos(T)      # tableau numpy 1D des différentes abscisses x
Y = b*np.sin(T)      # tableau numpy 1D des différentes ordonnées y

plt.figure(1)
plt.axis("equal")      # pour avoir des longueurs d'axes égaux
plt.plot(X, Y, "b" )    # Courbe paramétrée tracée en bleu
plt.show()
```

On obtient :



2.2 Courbes polaires

Dans le plan (Oxy), un point M peut être repéré soit par ses coordonnées cartésiennes (x, y), soit par ses coordonnées polaires (r, θ) comme cela est illustré sur la figure ci-dessous :



Une courbe polaire est la représentation graphique d'une fonction $f : \theta \longmapsto r = f(\theta)$. On peut utiliser deux méthodes pour obtenir cette représentation.

a) Tracé à l'aide d'une courbe paramétrée

Il suffit d'exprimer les coordonnées cartésiennes de M en fonction de l'angle polaire θ. On obtient :

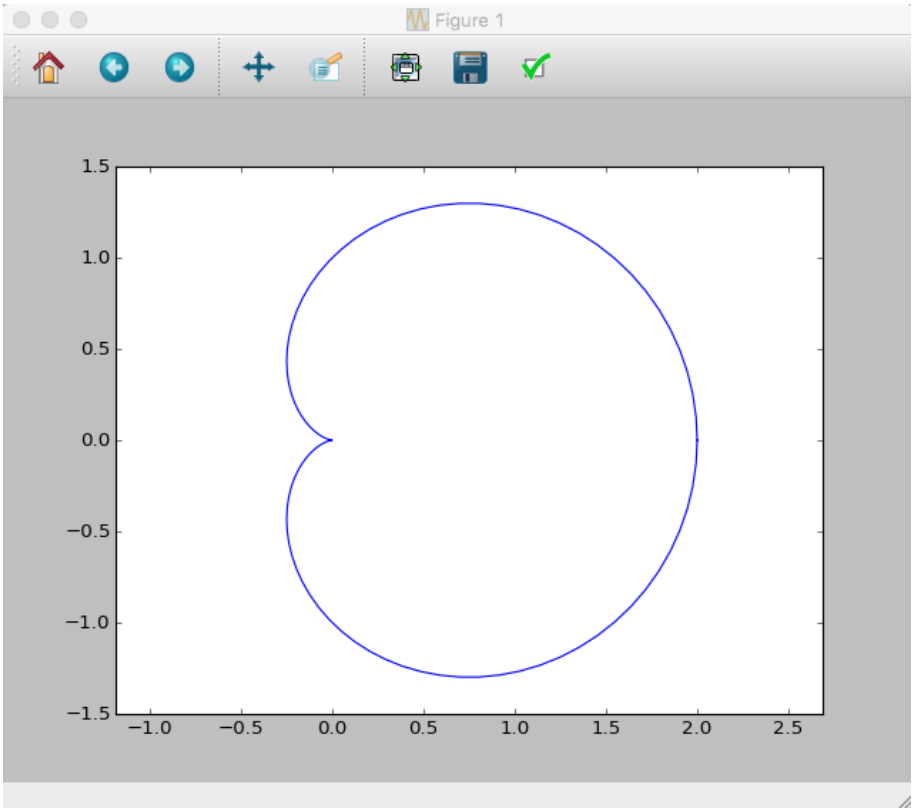
$$\left\{ \begin{array}{rcl} x & = & r\cos\theta = f(\theta)\cos(\theta) \\ y & = & r\sin\theta = f(\theta)\sin(\theta) \end{array} \right.$$

ce qui est une courbe paramétrée ayant pour paramètre θ.

Prenons l'exemple d'une cardioïde qui est la courbe polaire d'équation $f : \theta \longmapsto r = f(\theta) = a(1+\cos\theta)$) où a est un réel strictement positif. Nous allons prendre $a = 1$ pour fixer les idées.

```
import numpy as np
import matplotlib.pyplot as plt

Theta = np.linspace(0, 2*np.pi, 100)      # np.pi convient aussi pour la constante pi
a = 1
R = a*( 1 + np.cos(Theta) )      # tableau numpy 1D des différents rayons r
X = R*np.cos(Theta)      # tableau numpy 1D des différentes abscisses x
Y = R*np.sin(Theta)      # tableau numpy 1D des différentes ordonnées y

plt.figure(1)
plt.axis("equal")      # pour avoir des axes de longueurs égales
plt.plot(X, Y, "b" )    # Courbe polaire en bleu
plt.show()
```

ce qui produit le résultat :



b) Tracé direct de la courbe polaire avec polar

La fonction **polar** du module **plt** permet de tracer directement les courbes polaires. Sa syntaxe de base est :

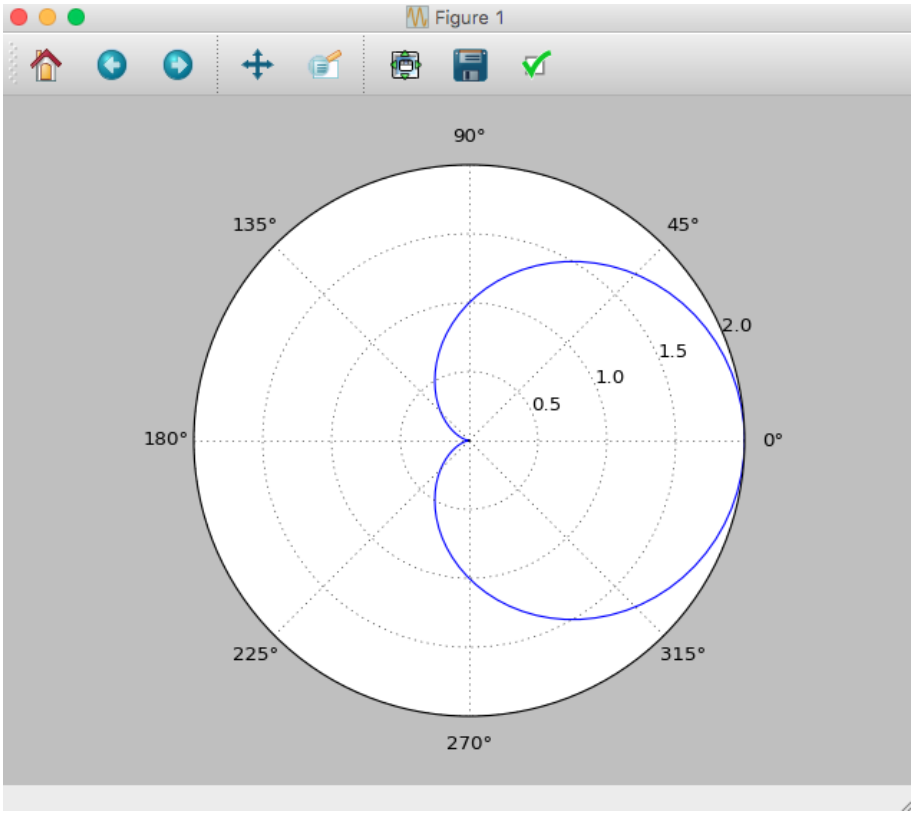
plt.polar(Theta, R)

où **Theta** est une liste ou un tableau numpy 1D des valeurs de l'angle θ et **R** est la liste (ou le tableau numpy 1D) des valeurs de $r = f(\theta)$ associées. La fonction peut prendre un troisième paramètre facultatif, identique à celui de la fonction **plot**, qui permet d'indique la couleur, le marqueur et la façon de relier les différents marqueurs, par exemple "ro-".

Avec cette technique, l'exemple précédent de la cardioïde se programme comme ci-dessous :

```
import numpy as np
import matplotlib.pyplot as plt

Theta = np.linspace(0, 2*np.pi, 100)      # Tableau numpy 1D des valeurs de theta
a = 1
R = a*( 1 + np.cos(Theta) )      # tableau numpy 1D des différents rayons r

plt.figure(1)
plt.polar(Theta, R, "b" )    # Courbe polaire en bleu
plt.show()
```

ce qui donne :



Ici, la zone de tracé est mieux adaptée aux coordonnées polaires et on peut facilement voir quels sont l'angle θ et le rayon r de chaque point de la courbe. Lorsque vous déplacez le curseur de la souris sur la figure, les coordonnées polaires du point désigné par le curseur s'affichent en bas à gauche de la figure.

3 Surfaces équipotentielles et lignes de champ

Nous traitons dans cette section de la façon d'utiliser Python pour tracer des lignes équipotentielles et des lignes de champ. Pour cela, il faudra utiliser les modules **numpy** et **matplotlib.pyplot**, que nous allons importer avec les alias **np** et **plt**.

Commençons par définir les limites de la fenêtre d'affichage :

```
import numpy as np
import matplotlib.pyplot as plt
xmin, xmax, ymin, ymax = -4, 4, -4, 4
```

3.1 Représentation d'une grandeur physique $g(x, y)$

Une grandeur physique comme le potentiel électrique $V(x, y)$ va être calculée dans le rectangle $(x_{max} - x_{min}) \times (y_{max} - y_{min})$ mais seulement en des points particuliers qui forment les nœuds d'un maillage de ce plan. Il faut pour cela définir un **pas** que l'on va noter h. À partir de là, les seuls points intéressants sont :

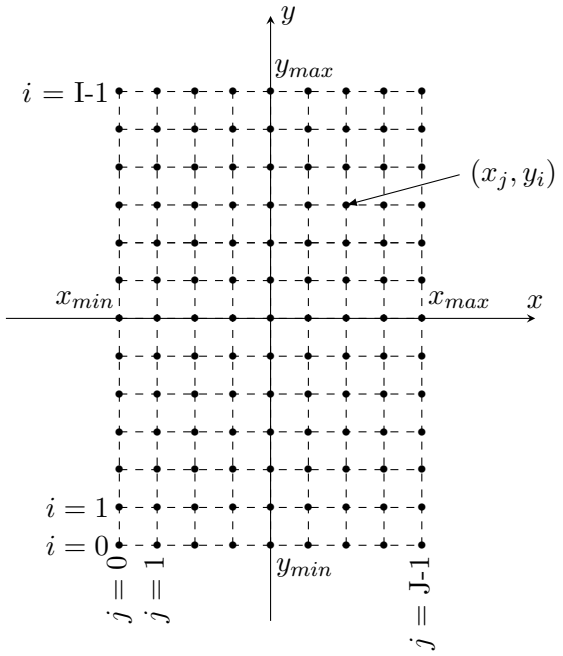
$$x_j = x_{min} + j\,h \quad \text{et} \quad y_i = y_{min} + i\,h$$

où i et j sont deux entiers naturels variant respectivement de 0 à $I-1$ et de 0 à $J-1$, de façon à ce que : $x_{J-1} = x_{max}$ et $y_{I-1} = y_{max}$.

Les grandeurs physiques ne seront donc calculées qu'aux points (x_j, y_i) : $V(x_j, y_i)$ et leurs valeurs seront rangées dans une **matrice V** de type I x J possédant I lignes et J colonnes et dont les coefficients seront notés `V[i,j]`. Nous aurons donc :

$$\boxed{\text{V[i, j]} = V(x_j, y_j)}$$

Attention donc à l'ordre des indices et à ce qu'ils représentent ! `i` est l'indice de ligne et `j` est l'indice de colonne. On peut se représenter la zone d'affichage comme dans la figure ci-dessous :



On définit ensuite le pas et les tableaux unidimensionnels des abscisses et des ordonnées [4] :

```
h = 0.01
X = np.arange(xmin, xmax, h)
Y = np.arange(ymin, ymax, h)
```

Un affichage de **X** donnera par exemple : `array([-4.,-3.99,-3.98,...,3.97,3.98,3.99])`.

Les premières grandeurs à définir ne sont pas des grandeurs physiques mais *deux matrices* **XX** et **YY** dont les coefficients sont les abscisses et les ordonnées des nœuds du maillage, c'est à dire :

$$\forall\,(i,j)\;\; \mathrm{XX}[i,j] = \mathrm{X}[j] \quad \text{et} \quad \mathrm{YY}[i,j] = \mathrm{Y}[i]$$

Cela se fait à l'aide de la fonction **meshgrid** du module numpy :

XX, **YY** = **np.meshgrid**(**X**,**Y**)

L'intérêt de ces deux matrices va apparaître plus loin, en relation avec les principes de calcul des objets numpy. Rappelons que si `A` et `B` sont deux matrices numpy, et x un flottant ou un entier, alors :

- `A` + x est une matrice dont les coefficients sont ceux de `A` auxquels on a ajouté x.
- `A`*x (ou x*`A`) est une matrice donc les coefficients sont ceux de `A` multipliés par x.
- `A`*`B` = `B`*`A` est une matrice dont les coefficients sont les produits des coefficients de `A` et `B`. Cela n'a donc rien à voir avec un produit matriciel !
- 1/`A` est la matrice dont les coefficients sont les inverses de ceux de `A`. Rien à voir avec la matrice inverse !
- `A`**3 est la matrice dont les coefficients sont ceux de `A` élevés à la puissance 3.
- Enfin, numpy dispose de fonctions classiques comme **np.cos**, **np.sin**, **np.sqrt**, etc ... Par exemple **np.sqrt**(`A`) est une matrice dont les coefficients sont les racines carrées des coefficients de `A`.

Une première application de ces règles est le calcul de la matrice `D` dont les coefficients sont les distances du point (x_j, y_i) à l'origine O du repère : `D[i,j]` $= \sqrt{x_j^2 + y_i^2}$. On écrit :

```
D = np.sqrt(XX**2 + YY**2)
```

En effet, pour tout `i` et `j` : `D[i,j]` $= \sqrt{(\mathrm{XX}[i,j])^2 + (\mathrm{YY}[i,j]^2)} = \sqrt{(\mathrm{X}[j])^2 + (\mathrm{Y}[i])^2}$, ce qui est bien le résultat recherché.

Nous allons maintenant appliquer ces connaissances au calcul du potentiel électrostatique créé par deux charges ponctuelles $q_1 = 1.10^{-7}$ C placée en $P_1(x_1 = -1, 0)$ et $q_2 = -2.10^{-7}$ C placée en $P_2(x_2 = 1, 0)$. Nous noterons K la constante de l'interaction électrique :

$$K = \frac{1}{4\pi\epsilon_0} = 9{,}0 \times 10^9 \text{ F.m}^{-1}$$

Le potentiel s'écrivant :

$$V(x_j, y_i) = K\frac{q_1}{\sqrt{(x_j - x_1)^2 + y_i^2}} + K\frac{q_2}{\sqrt{(x_j - x_2)^2 + y_i^2}}$$

4. La fonction **arange**(*debut*, *fin*, *pas*) du module numpy crée un tableau unidimensionnel (vecteur ligne) constitué de valeurs commençant à *debut* (inclu), finissant à *fin* (exclu) et espacées de *pas*. Contrairement à la fonction **range** standard qui n'admet que des entiers, *debut*, *fin* et *pas* peuvent aussi être des flottants.

en voici la programmation :

```
K = 9.0e9
q1, q2 = 1e-7, -2e-7
x1, x2 = -1, 2
D1 = np.sqrt( (XX-x1)**2 + YY**2 )
D2 = np.sqrt( (XX-x2)**2 + YY**2 )
V = K*q1/D1 + K*q2/D2
```

`V` sera donc une matrice de I lignes et J colonnes telle que : `V[i,j]` $= V(x_j, y_i)$

Il y a cependant un problème car certaines valeurs des matrices D1 et D2 peuvent être très petites, voire nulles lorsqu'on se rapproche de l'une ou l'autre des deux charges ponctuelles. Dans ce cas, certains coefficients de V peuvent être très grands, voire infini ce qui génère dans le dernier cas une erreur de calcul de la part de Python. Nous souhaitons donc exclure les valeurs trop importantes ou nulles.

Pour réaliser cela, **numpy** est formidable car il possède un embryon de structure de langage de programmation. Si `A` est une matrice, alors l'expression `A < 2` représente une matrice de même taille que `A` mais dont les coefficients sont des booléens qui valent **True** ou **False** selon que les coefficients de `A` sont < 2 ou $>= 2$. Essayons !

```
A = np.array( [ [2.68, -1.02], [1.98, 5.45] ] )
B = (A < 2)
print("A = ", A)
print("B = ", B)
```

va afficher :

$$\mathrm{A} = \begin{bmatrix} 2.68 & -1.02 \\ 1.98 & 5.45 \end{bmatrix} \qquad \mathrm{B} = \begin{bmatrix} \text{False} & \text{True} \\ \text{True} & \text{False} \end{bmatrix}$$

Les opérateurs logiques <, >, <= (inférieur ou égal), >= (supérieur ou égal), == (identique à) et != (différent de) sont autorisés pour construire des expressions booléennes.

D'autre part, si **B** est une matrice de booléens *de même taille* que **A**, alors l'instruction :

`A[B] =` x

affecte la valeur x aux coefficients de **A** correspondant aux valeurs **True** de **B**. On peut aussi écrire directement l'expression booléenne entre les deux crochets.

Ainsi, une instruction de la forme : **B** = (**A** >= 2.0) suivie que **A**[**B**] = -1.0 n'affectera la valeur -1.0 qu'aux seuls coefficients de **A** supérieurs à 2.0 On peut aussi écrire directement : **A**[**A** >= 2.0] = -1.0 cela marche aussi !

Nous allons profiter de cela pour éliminer les valeurs du potentiel non définies. La plus petite distance *non nulle* entre deux nœuds du maillage étant le pas h, on élimine les valeurs nulles (ou trop petites) de **D**1 et **D**2 en les remplaçant par h :

```
K = 9.0e9
q1, q2 = 1e-7, -2e-7
x1, x2 = -1, 2
D1 = np.sqrt( (XX-x1)**2 + YY**2 )
D1[D1 < h] = h        # Elimination des valeurs trop petites ou nulles
D2 = np.sqrt( (XX-x2)**2 + YY**2 )
D2[D2 < h] = h        # Idem pour D2
V = K*q1/D1 + K*q2/D2
```

De cette façon, **V** reste bien défini en tout point du maillage.

3.2 Courbes équipotentielles

Nous allons maintenant représenter les lignes équipotentielles dans le plan (Oxy). Pour cela, nous allons utiliser la fonction `plt.contour` de `matplotlib.pyplot` qui trace les lignes de niveaux, c'est à dire les courbes d'équation $V(x, y) = \text{Cste}$. Pour que le rendu soit intéressant, cette fonction prend 4 paramètres et s'écrit :

plt.contour(**XX**, **YY**, **V**, **K**)

où **XX** et **YY** sont les matrices déjà vues qui définissent le maillage du plan (Oxy), **V** est la matrice de la grandeur dont on veut représenter les lignes de niveau et **K** une liste ou un tableau numpy unidimensionnel qui contient toutes les constantes pour chaque ligne de niveau qu'on veut. On peut écrire **K** "à la main" ou bien la générer automatiquement en utilisant la fonction `np.linspace` par exemple [5] :

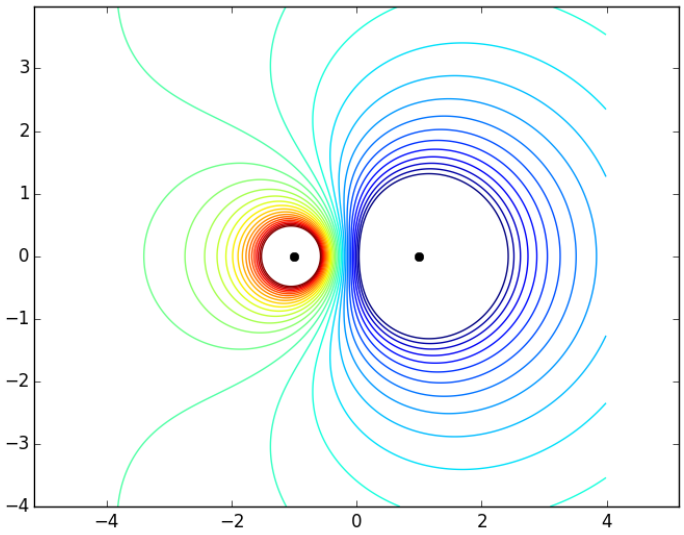
K = `np.linspace`(-1000,1000,20)

qui donnera un tableau de 20 valeurs régulièrement espacées entre -1000 et +1000.

Voici donc le programme qui représente 20 équipotentielles régulièrement espacées entre -1000 V et 1000 V et qui donne la figure ci-dessous :

```
plt.figure()
plt.axis("equal")        # Pour rendre le repère orthonormé
plt.plot([-1],[0], "ko") # Dessine un gros point noir en (-1,0) pour la charge ponctuelle
plt.plot([1],[0], "ko")  # Pareil pour la deuxième charge
plt.contour( XX, YY, V, np.linspace(-1000,1000,20) )
plt.show()
```

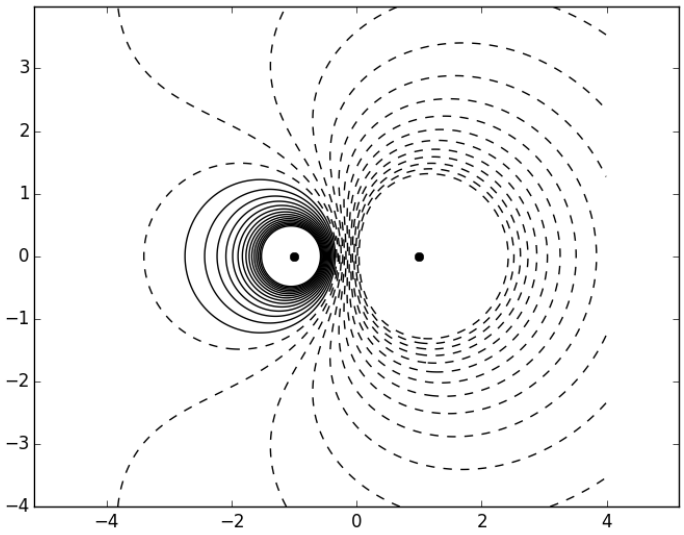
5. On peut aussi utiliser la fonction **arange**.



Chaque équipotentielle est tracée avec une couleur différente, choisie par Python. Si vous souhaitez un tracé monocolor, il faut l'indiquer à l'aide d'un cinquième paramètre **colors** (attention au s final !). Par exemple, si vous ne voulez que des équipotentielles en noir, vous écrirez :

```
plt.contour(XX, YY, V, np.linspace(-1000,1000,20), colors = "k" )
```

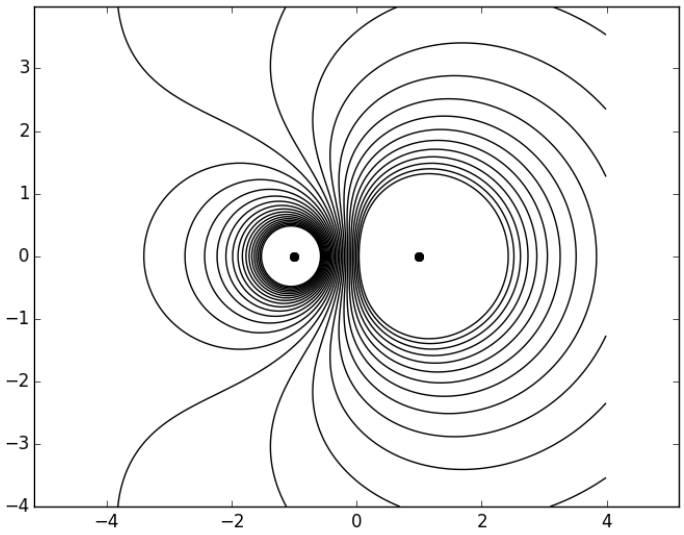
ce qui donne :



Dans ce cas, les équipotentielles négatives sont dessinées en pointillés par défaut. On peut cependant rendre ces tracés continus en écrivant (il faut importer le module matplotlib avant) :

```
import matplotlib
matplotlib.rcParams["contour.negative_linestyle"] = "solid"
plt.contour(XX, YY, V, np.linspace(-1000,1000,20), colors = "k" )
```

ce qui donne :



On peut faire beaucoup de choses avec la fonction `plt.contour` et il existe aussi un fonction `plt.contourf` qui trace les lignes de niveau et qui colorie les surfaces entre les différentes lignes. Allez voir sur internet la documentation sur ces fonctions.

3.3 Calcul d'un champ électrostatique

Nous allons maintenant calculer le champ électrostatique créé par les deux charges en utilisant deux matrices **Ex** et **Ey** de type $I \times J$ qui représentent les deux composantes cartésiennes de $\overrightarrow{E}$ en tout point du maillage et qui vérifient donc :

$$\mathbf{Ex}[i,j] = E_x(x_j, y_i) \quad \text{et} \quad \mathbf{Ey}[i,j] = E_y(x_j, y_i)$$

Notons que :

$$\overrightarrow{E}(M) = \frac{q_1}{4\pi\varepsilon_0}\frac{\overrightarrow{P_1M}}{P_1M^3} + \frac{q_2}{4\pi\varepsilon_0}\frac{\overrightarrow{P_2M}}{P_2M^3}$$

et donc :

$$E_x(x_j, y_i) = \frac{q_1}{4\pi\varepsilon_0}\frac{x_j - x_1}{\left[\sqrt{(x_j-x_1)^2 + y_i^2)}\right]^3} + \frac{q_2}{4\pi\varepsilon_0}\frac{x_j - x_2}{\left[\sqrt{(x_j-x_2)^2 + y_i^2)}\right]^3}$$

et

$$E_y(x_j, y_i) = \frac{q_1}{4\pi\varepsilon_0}\frac{y_i}{\left[\sqrt{(x_j-x_1)^2 + y_i^2)}\right]^3} + \frac{q_2}{4\pi\varepsilon_0}\frac{y_i}{\left[\sqrt{(x_j-x_2)^2 + y_i^2)}\right]^3}$$

La méthode pour créer ces matrices est la même que celle utilisée pour la matrice potentiel **V**. Nous reprenons le programme au début :

```
import numpy as np
import matplotlib.pyplot as plt

xmin, xmax, ymin, ymax = -4, 4, -4, 4        # Fenêtre de création du dessin
h = 0.01                                      # Pas du maillage du plan (Oxy)
X = np.arange(xmin, xmax, h)
Y = np.arange(ymin, ymax, h)
XX, YY = np.meshgrid(X, Y)

K = 9.0e9                                     # Constante de l'interaction
q1, q2 = 1e-7, -2e-7                          # Charges ponctuelles
x1, x2 = -1, 1
D1 = np.sqrt( (XX - x1)**2 + YY**2 )
D1[D1 < h] = h                    # On élimine les valeurs trop petites
D2 = np.sqrt( (XX - x2)**2 + YY**2 )
D2[D2 < h] = h                    # Même chose pour D2

Ex = K*q1*( XX - x1)/D1**3 + K*q2*( XX - x2)/D2**3
Ey = K*q1*YY/D1**3 + K*q2*YY/D2**3
```

Pour pouvoir tracer des lignes de champ, nous allons mettre à 0 les premières et dernières lignes et colonnes ainsi que toutes les valeurs du champ électrostatique en des points très proches des charges électriques : en ces points le champ n'est pas défini ou peut prendre des valeurs très élevées ; nous allons donc éliminer ces grandes valeurs et les mettre à 0. On verra plus loin que cela permettra de gérer l'arrêt des lignes de champ.

```
Ex[D1 < 3*h] = 0          # Mise à zéro des champ proches des charges
Ey[D1 < 3*h] = 0          # On a pris le critère 3*h mais un autre convient aussi
Ex[D2 < 3*h] = 0
Ey[D2 < 3*h] = 0

I, J = Ex.shape           # On récupère le nombre de lignes et de colonnes

for i in range(I) :
    Ex[i,0] = 0
    Ex[i,J-1] = 0
    Ey[i,0] = 0
    Ey[i,J-1] = 0

for j in range(J) :
    Ex[0,j] = 0
    Ex[I-1,j] = 0
    Ey[0,j] = 0
    Ey[I-1,j] = 0
```

3.4 Tracé d'une ligne de champ

Le tracé des lignes de champ s'appuie sur la fonction `pointSuivant(xa,ya, h)` qui prend en paramètres les coordonnées (x_a, y_a) d'un point M_a sur une ligne de champ et qui renvoie les coordonnées (x_s, y_s) du point suivant M_s sur la même ligne, en faisant le calcul avec un pas élémentaire h.

Principe de fonctionnement :

Soient $E_x(x_a, y_a)$ et $E_y(x_a, y_a)$ les deux composantes du champ électrostatique en (x_a, y_a). Le principe du calcul de x_s et y_s est illustré sur les figures ci-dessous. On considère que

$M_a(x_a, y_a)$ est le centre d'un carré d'arête $2h$ et que $M_s(x_s, y_s)$ est quelquepart sur ce carré : selon les valeurs de E_x et E_y on placera M_s comme indiqué sur les exemples ci-dessous :

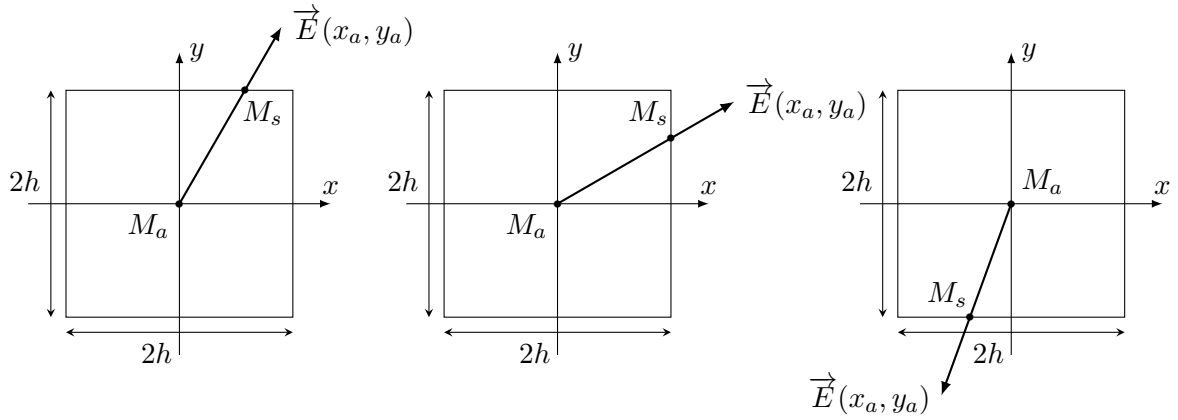


FIGURE 1 – Trois exemples de positionnement du point suivant M_s sur une ligne de champ passant par M_a, selon la valeur du champ électrostatique en M_a.

Les points $M_a(x_a, y_a)$ et $M_s(x_s, y_s)$ ne sont pas forcément des nœuds du maillage du plan (Oxy). En revanche, h est le même pas que celui qui a servi à définir le maillage. L'algorithme de calcul de x_s et y_s peut être résumé de la façon suivante :

1. Déterminer les valeurs de E_x et de E_y issues des matrices **Ex** et **Ey** les plus proches de (x_a, y_a).
2. Si $E_x == 0$ alors :
   Si $E_y == 0$ alors faire $x_s = x_a$ et $y_s = y_a$
   Si $E_y > 0$ alors faire $x_s = x_a + h$
   Sinon faire $x_s = x_a - h$
3. Sinon Si $E_x > 0$ alors :
   Calculer $qt = \dfrac{E_y}{E_x}$
   Calculer x_s et y_s selon la valeur de qt
4. Sinon $E_x < 0$ alors :
   Calculer $qt = \dfrac{E_y}{E_x}$
   Calculer x_s et y_s selon la valeur de qt

Le quotient `qt` représente la tangente de l'ange que fait $\overrightarrow{E}(x_a, y_a)$ avec l'axe $M_a x$. On voit que les cas particuliers importants pour la discussion sont `qt == -1` et `qt == 1`.

```
def pointSuivant(xa, ya, h) :
    j = round( ( xa - xmin)/h )
    i = round( ( ya - ymin)/h )
    cex, cey = Ex[i,j] , Ey[i,j]          # On récupère les deux composantes de E en Ma
    epsilon = 1e-5                        # Précision pour la comparaison à 0 des flottants

    if abs(cex) < epsilon :
        if abs(cey) < epsilon :
            xs, ys = xa, ya
        elif cey > 0 :
            xs, ys = xa, ya + h
        else :
            xs, ys = xa, ya - h

    elif cex > 0 :
        qt = cey/cex
        if qt > 1 :
            xs, ys = xa + h/qt, ya + h
        elif abs(qt) <= 1 :
            xs, ys = xa + h, ya + qt*h
        else :
            xs, ys = xa - h/qt, ya - h

    else :
        qt = cey/cex
        if qt > 1 :
            xs, ys = xa - h/qt, ya - h
        elif abs(qt) <= 1 :
            xs, ys = xa - h, ya - qt*h
        else :
            xs, ys = xa + h/qt, ya + h

     return xs, ys
```

Deux remarques :

- Pour calculer `i` et `j`, il est préférable de faire appel à la fonction `round` qui donne un résultat plus proche que `int`. En effet : `round(2.7) == 3` alors que `int(2.7) == 2` (on a aussi `round(2.3) == 2`).
- Aux lignes 5 et 6, on aurait dû écrire `if cex == 0` et `if cey == 0`. Cependant la comparaison à zéro des nombres flottants pose des problèmes car, en raison des approximations de calcul, Python peut très bien écrire 1.0001e-28 à la place de 0. Pour éviter qu'un nombre aussi petit soit considéré comme > 0, il est préférable de définir une *précision* `epsilon` en dessous de laquelle on considère que le flottant est nul.

Il reste à tracer une ligne de champ complète. Ce tracé s'appuie sur une fonction `ligneChamp(x0,y0,h)` qui prend comme paramètres les coordonnées `(x0,y0)` du premier point de la ligne de champ ainsi que le pas `h` et qui renvoie deux tableaux numpy unidimensionnels `Xl` et `Yl` qui contiennent respectivement les abscisses et les ordonnées de cette ligne de champ

```
def ligneChamp(x0, y0, h) :
    xc, yc = x0, y0                    # Coordonnées du point courant
    xs, ys = pointSuivant(xc, yc, h)   # Coordonnées du point suivant
    Xl, Yl = [xc], [yc]

    while (xc != xs or yc != ys) :
        Xl.append(xs)
        Yl.append(ys)
        xc, yc = xs, ys
        xs, ys = pointSuivant(xc, yc, h)

    Xl = np.array(Xl)                  # Transformation des listes en tableaux numpy
    Yl = np.array(Yl)

    return Xl, Yl
```

Remarques :

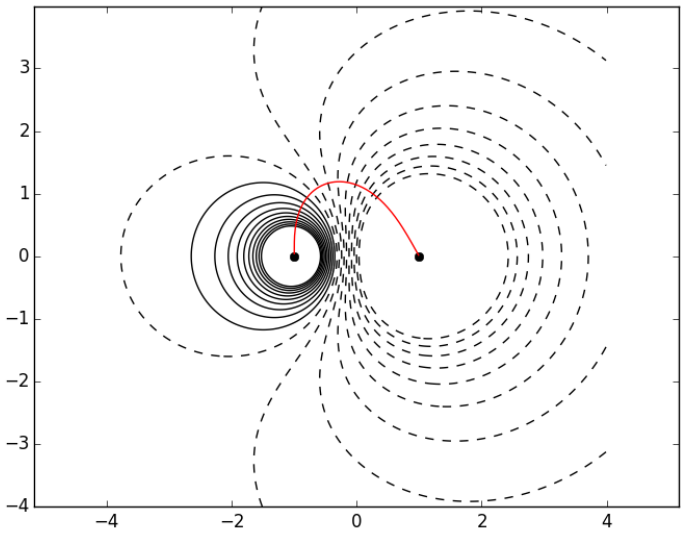
- Les points suivants sont ajoutés aux listes tant qu'ils ne sont pas confondus avec le point courant. Dès que `xc == xs and yc == ys` devient vraie[6], la boucle s'arrête. D'après la programmation de la fonction `pointSuivant`, ceci ne peut se produire que si $\overrightarrow{E} = \overrightarrow{0}$
- La ligne de champ s'arrêtera donc forcément soit à l'intérieur de la fenêtre de représentation, en des points de champ nul, soit sur les bords de cette fenêtre puisqu'on y a fait $\overrightarrow{E} = \overrightarrow{0}$.

Voici le tracé d'une ligne de champ qui commence en `(-1,4*h)` donc tout près de la charge q_1. On a aussi gardé le tracé des équipotentielles.

```
plt.figure()
plt.axis("equal")
plt.plot([-1],[0],"ko")
plt.plot([1],[0], "ko")
C = plt.contour(XX,YY, V, np.linspace(-1000,1000,20), colors="k")

Xl, Yl = ligneChamp(-1, 4*h, h)
plt.plot(Xl, Yl, "r")
plt.show()
```

ce qui donne :

6. Rappelons que la négation de l'expression logique **(xc != xs or yc != ys)** est **(xc == xs and yc == ys)**.



3.5 Tracé d'un ensemble de lignes de champs

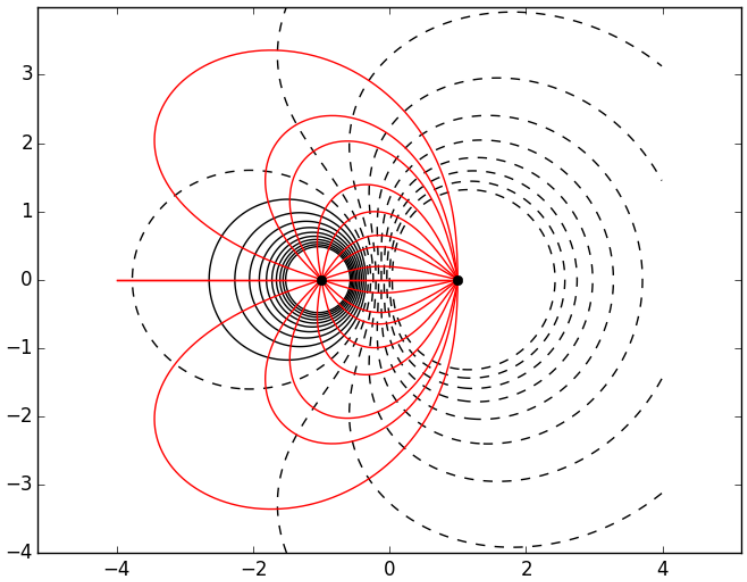
On peut tracer un ensemble de lignes de champ en créant une liste des points initiaux de chaque ligne et en traçant chacune de celles-ci à l'aide d'une boucle.

Traçons par exemples des lignes qui partent de la charge q_1 placée en $(-1, 0)$. On commence par créer deux tableaux contenant les abscisses et les ordonnées des points initiaux, situés sur un cercle de rayon $4h$ autour de la charge. On utilise aussi la symétrie des lignes de champ par rapport à l'axe des deux charges.

Le programme ci-dessous vient compléter celui écrit précédemment.

```
tableAngles = np.linspace(0, np.pi, 10)        # 10 angles compris entre 0 et pi
X0 = -1 + 4*h*np.cos(tableAngles)               # Abscisses initiales
Y0 = 4*h*np.sin(tableAngles)                    # Ordonnées initiales
N = X0.size                                     # Nombre d'éléments de X0
for n in range(N) :
    Xl, Yl = ligneChamp( X0[n], Y0[n], h)
    plt.plot( X0[n], Y0[n], "r")
    plt.plot( X0[n], -Y0[n], "r")               # Ligne symétrique par rapport à Ox
```

Remarque : `np.pi` est une constante du module numpy qui vaut π.


3
2
1
0
−1
−2
−3
−4
−4
−2
0
2
4