Modules Python

Cette annexe propose une introduction succinte aux modules les plus employés dans les 8 chapitres de ce cours. On renvoie aussi aux documentations officielles :

- `numpy` et `scipy` (calcul scientifique) : https://numpy.org et https://scipy.org.
- `matplotlib` (dessin scientifique) : https://matplotlib.org.
- `networkx` (manipulation de graphes) : https://networkx.org.

Si ces modules ne sont pas déjà installés (ou ne sont pas à jour), `pip3 install module` dans un terminal peut remédier à cela.

1. Numpy.

Le module `numpy` introduit une structure de données essentielle pour tous nos calculs : les *arrays* (tableaux).

```
>>> L = np.array([1, 3.5, -2]) ; L

array([ 1. , 3.5, -2. ])
```

La syntaxe de manipulation des listes s'étend aux *arrays* : par exemple, pour récupérer le i-ième élément d'un *array* `L`, on utilisera la commande `L[i]`, et pour récupérer les éléments d'indices entre a et $b-1$, on utilisera la commande `L[a:b]`. Pourquoi alors utiliser les *array* à la place des listes Python ? Il y a au moins trois bonnes raisons :

(1) Les données d'un *array* sont stockées contiguement en espace mémoire, alors que les données d'une liste ne le sont pas. Ceci fait que les calculs sur un *array* sont en général beaucoup plus rapides. Par exemple, considérons 100 000 réels aléatoires stockés sous la forme d'un *array* `A` ou d'une liste `L`. Les commandes suivantes calculent le temps requis pour faire la somme de ces nombres :

```
>>> A = random.random(size=100000)
>>> L = list(A)
>>> timeit (np.sum(A))

13.6 µs ± 27.5 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)

>>> timeit (sum(L))

1.83 ms ± 880 ns per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

La même opération avec un *array* prend moins de 100 fois moins de temps !

(2) Les *arrays* peuvent être unidimensionnels (comme des listes), mais aussi multidimensionnels, et en particulier bidimensionnels (comme des matrices).

```
>>> L = np.array([[1, 3.5, -2], [-4, 0, 2.25]])
```

```
>>> L.shape

(2,3)

>>> L[:,2]

array([-2.  ,  2.25])
```

C'est par exemple utile si l'on veut considérer un échantillon $M_1, \dots, M_N$ de moyennes empiriques, chaque M_i étant la moyenne de n variables aléatoires $X_{i,1}, \dots, X_{i,n}$. Si les variables $X_{i,j}$ sont uniformes sur $[0,1]$, on peut obtenir l'échantillon des moyennes empiriques en réduisant un tableau de taille $N \times n$:

```
>>> N, n = 10, 100
>>> alea = random.random(size=(N, n))
>>> np.mean(alea, axis=1)

array([0.44985482, 0.57135158, 0.54310497, 0.46410955, 0.54345499,
       0.47592304, 0.47515065, 0.49792214, 0.45701027, 0.47436332])
```

Par ailleurs, les tableaux bidimensionnels permettent d'effectuer efficacement des opérations d'algèbre linéaire, grâce aux commandes du sous-module `numpy.linalg`. Dans ce cours, ce sera en particulier utile pour manipuler les matrices de transition des chaînes de Markov.

(3) Les *arrays* sont adaptés à l'application de fonctions à chacune de leurs entrées (vectorisation des opérations). Voyons quelques exemples :

- on peut additionner des *arrays* ensemble (terme à terme), ou additionner un nombre à chaque terme d'un *array*.

```
>>> L = np.array([[1, 3.5, -2], [-4, 0, 2.25]])
>>> L1, L2 = L[0,:], L[1:,]
>>> L1 + L2

array([[-3.  ,  3.5 ,  0.25]])

>>> L1 + 2

array([3. , 5.5, 0. ])
```

- de même, on peut multiplier des *arrays* ensemble (terme à terme), ou multiplier un *array* par un nombre.

```
>>> L1 * L2

array([[-4. ,  0. , -4.5]])

>>> 2 * L2

array([[-8. ,  0. ,  4.5]])
```

- on peut appliquer une fonction à tous les termes d'un *array*. La plupart des fonctions mathématiques usuelles ont des versions `numpy` adaptées à cet usage : `np.sqrt` pour la racine carrée, `np.exp` pour l'exponentielle, etc.

```
>>> np.exp(L1 + L2**2)

array([[2.41549528e+07, 3.31154520e+01, 2.13809428e+01]])

>>> (L1 >= 0)

array([ True,  True, False])
```

Une application importante de ces techniques est le dessin du graphe d'une fonction F. Si l'on veut dessiner sur l'intervalle $[a, b]$ le graphe $y = F(x)$, on peut : utiliser `xx = np.linspace(a, b, n)` pour créer n points régulièrement espacés entre a et b (typiquement avec $n = 500$) ; appliquer F à ce vecteur avec la commande vectorielle `yy = F(xx)` ; et finalement dessiner le graphe de la fonction avec la commande `ax.plot(xx, yy)`.

Les autres modules que nous employerons ont pour arguments ou pour résultats des *arrays* : par exemple, les variables aléatoires de `scipy.stats` produisent par la méthode `.rvs(size=N)` un N-échantillon donné sous la forme d'un *array*, et les fonctions de dessin de données de `matplotlib.pyplot` prennent comme arguments des *arrays* de données. C'est une autre bonne raison pour employer systématiquement cette structure de données.

2. Scipy.

Le module `scipy` contient de très nombreuses fonctions et algorithmes mathématiques : intégration numérique, algèbre linéaire, interpolation, etc. Nous utiliserons presque exclusivement les fonctions de probabilités et de statistiques du sous-module `scipy.stats`.

Les distributions de probabilité suivantes sont implantées dans `scipy.stats` :

- loi de Bernoulli $\mathrm{Ber}(p)$: `scs.bernoulli(p)`.
- loi binomiale $\mathrm{Bin}(n, p)$: `scs.binom(n, p)`.
- loi de Poisson $\mathrm{Poi}(\lambda)$: `scs.poisson(L)`.
- loi géométrique $\mathrm{Geom}(p)$: `scs.geom(p)`.
- loi uniforme $\mathrm{Unif}([a, b])$: `scs.uniform(loc = a, scale = b-a)`. Par défaut, $[a, b] = [0, 1]$.
- loi exponentielle $\mathrm{Exp}(\lambda)$: `scs.expon(scale = 1/L)`. Par défaut, $\lambda = 1$.
- loi normale $\mathrm{N}(m, \sigma^2)$: `scs.norm(loc = m, scale = sigma)`. Par défaut, $m = 0$ et $\sigma^2 = 1$.
- loi de Cauchy $\mathrm{Cau}(c)$: `scs.cauchy(scale = c)`. Par défaut, $c = 1$.

Cette liste est non exhaustive (il y a plus de 100 distributions déjà implantées), mais largement suffisante pour nos simulations. Étant donnée une loi `law` comme ci-dessus, différentes méthodes permettent de simuler des variables aléatoires avec cette loi, et d'obtenir des objets mathématiques reliés à la loi.

- La commande la plus utile est `law.rvs(size = N)`, qui crée un N-échantillon de variables indépendantes. On peut aussi utiliser le paramètre `size = (l, c)` pour créer une matrice de variables indépendantes, avec l lignes et c colonnes.

```
>>> scs.norm.rvs(size=5)
```

```
array([-2.77972731,  0.1897642 , -1.02439086, -0.02655945,  1.57445863])
```

- Les commandes `law.mean()` et `law.var()` donnent la moyenne et la variance (théoriques) de la loi.

```
>>> scs.norm.mean(), scs.norm.var()

(np.float64(0.0), np.float64(1.0))
```

- La commande `law.cdf` renvoie la fonction de répartition de la loi. On peut l'appliquer à un réel, ou aux termes d'un *array*.

```
>>> F = scs.norm.cdf
>>> F(2)

np.float64(0.9772498680518208)

>>> F(np.array([0,1,2,3]))

array([0.5       , 0.84134475, 0.97724987, 0.9986501 ])
```

- Dans le cas discret, `law.pmf` est la fonction sur les entiers qui donne la probabilité $\mathbb{P}[X = k]$, et dans le cas continu, `law.pdf` est la fonction sur les réels qui donne la densité de probabilité $f_X(x)$.

```
>>> M = scs.poisson(3).pmf
>>> M(np.arange(10))

array([0.04978707, 0.14936121, 0.22404181, 0.22404181, 0.16803136,
       0.10081881, 0.05040941, 0.02160403, 0.00810151, 0.0027005 ])

>>> f = scs.norm.pdf
>>> f(np.array([0,1,2,3]))

array([0.39894228, 0.24197072, 0.05399097, 0.00443185])
```

Une fois un échantillon de variables aléatoires obtenu, on peut utiliser les méthodes de `numpy` pour calculer les statistiques de cet échantillon (par exemple la moyenne et la variance empirique avec `np.mean()` et `np.var()`), et les méthodes de `matplotlib.pyplot` pour représenter cet échantillon (par exemple avec `ax.ecdf()` pour la fonction de répartition empirique, et avec `ax.hist()` pour l'histogramme empirique dans le cas discret).

3. Matplotlib.

On utilise `Matplotlib` pour tous les dessins ; c'est un module très puissant et configurable, mais sa syntaxe n'est pas la plus claire. En effet, si l'on veut juste une figure avec le dessin d'un cercle, il faut écrire :

```
import matplotlib
import matplotlib.pyplot as plt
fig, ax = plt.subplots()
ax.add_patch(matplotlib.patches.Circle((0,0), 1))
ax.set_xlim(-1, 1)
ax.set_ylim(-1, 1)
ax.set_aspect(1)
```

```
ax.set_axis_off()
plt.show()
```

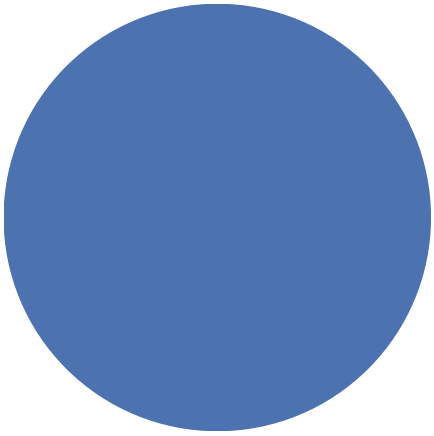

FIG. 3. Un cercle dessiné par `Matplotlib`. Après l'import des bibliothèques, il faut 7 lignes de commande...

Dans ce qui suit, on employera toujours les mêmes commandes d'amorce d'une figure : ce n'est pas la seule façon de faire, mais c'est relativement simple et clair par rapport aux autres façons.

Amorce. Dans l'exemple ci-dessus, la première commande `fig, ax = plt.subplots()` crée :

- une figure `fig`, qui sera dessinée par la dernière ligne de commande `plt.show()`. La dite figure n'apparaît plus jamais dans le programme. À la fin, on peut aussi utiliser `plt.savefig("path/to/file.pdf")` pour sauvegarder la figure à l'emplacement spécifié.
- un *Axis* `ax`, qui est une sous-figure de `fig`, sur laquelle des *Artists* `artist` viendront dessiner divers objets. On détaillera plus loin comment dessiner sur cette sous-figure : toutes les commandes seront du type `ax.artist(args)`.

Une figure peut contenir un unique *Axis*, ou plusieurs dans un *array* d'*Axes*. Si l'on veut plusieurs sous-figures l'une au-dessus de l'autre dans un tableau (respectivement, l'une à côté de l'autre), on utilisera l'amorce `fig, axs = plt.subplots(n)` (respectivement, l'amorce `fig, axs = plt.subplots(1, n)`), avec n égal au nombre de sous-figures souhaitées. On agira alors sur la k-ième figure avec `axs[k].artist(args)`. Si l'on veut carrément un tableau de $l \times c$ sous-figures, on utilisera l'amorce `fig, axs = plt.subplots(l, c)`. On agira alors sur la sous-figure d'indices (i, j) avec `axs[i,j].artist(args)`. Par exemple :

```
fig, axs = plt.subplots(1, 3)
axs[0].add_patch(matplotlib.patches.Circle((0,0), 1, color="r"))
axs[1].add_patch(matplotlib.patches.Circle((0,0), 1, color="g"))
axs[2].add_patch(matplotlib.patches.Circle((0,0), 1, color="b"))
for i in range(3):
    axs[i].set_xlim(-1, 1)
    axs[i].set_ylim(-1, 1)
    axs[i].set_aspect(1)
    axs[i].set_axis_off()
plt.show()
```

produit la figure suivante.

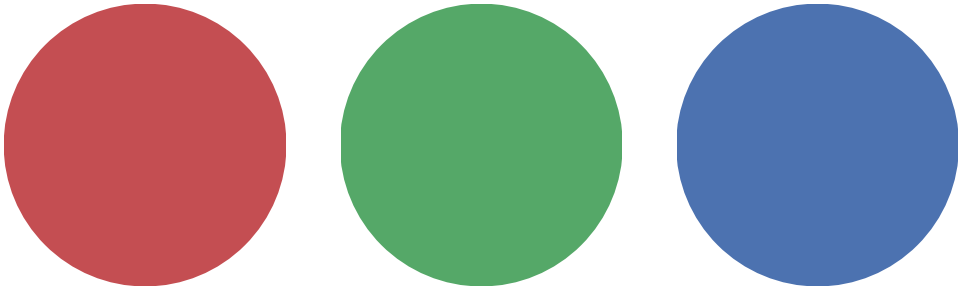

FIG. 4. Trois sous-figures Matplotlib contenant chacune un cercle.

Taille, espacement, titres. La taille de la figure peut être donnée en argument optionnel : `plt.subplots(figsize=(a, b))` force la figure à occuper un rectangle de taille $a \times b$ (par défaut, la taille d'une figure est 6.4×4.8, en pouces). Une fois la taille de la figure fixée, Matplotlib place les sous-figures avec un certain espacement. Le choix par défaut est en général assez mauvais : par exemple,

```
fig, axs = plt.subplots(2, 2) ;
plt.show()
```

donne ceci.

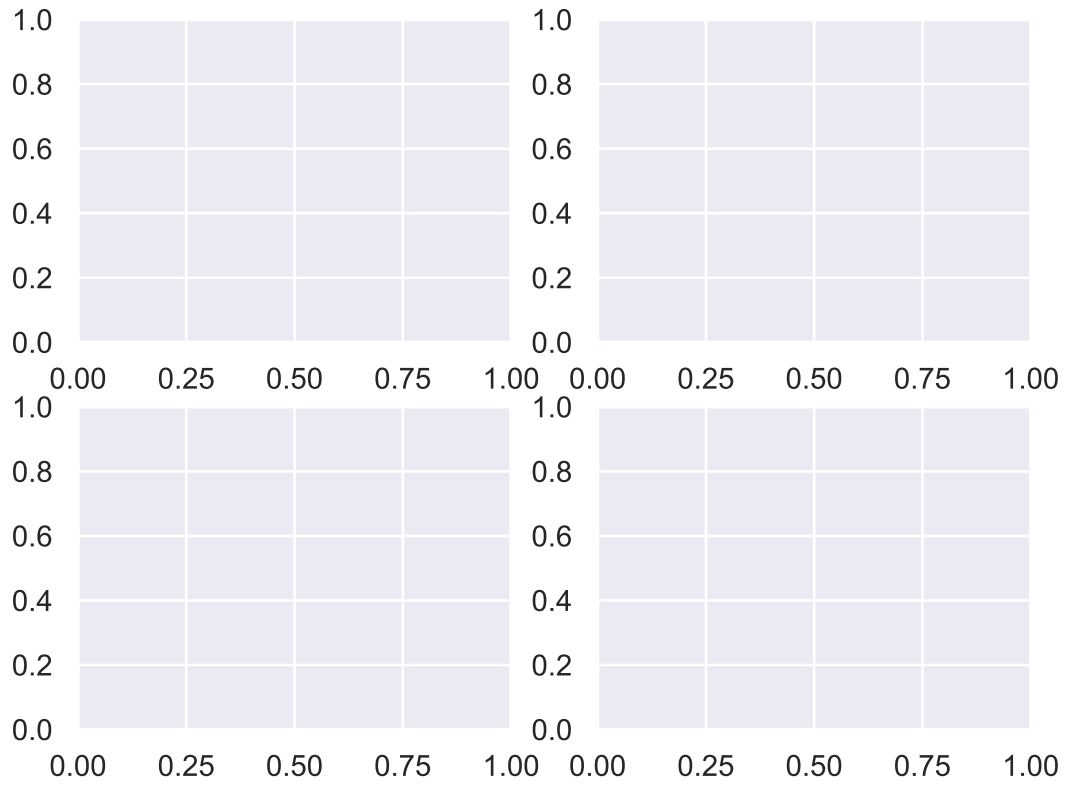

FIG. 5. Espacement par défaut d'un tableau de sous-figures.

On peut rajouter de l'espacement entre les sous-figures grâce à la commande

```
plt.subplots_adjust(wspace=x, hspace=y)
```

avec des valeurs réelles pour x et y. Par exemple, $x = y = 0.5$ donne :

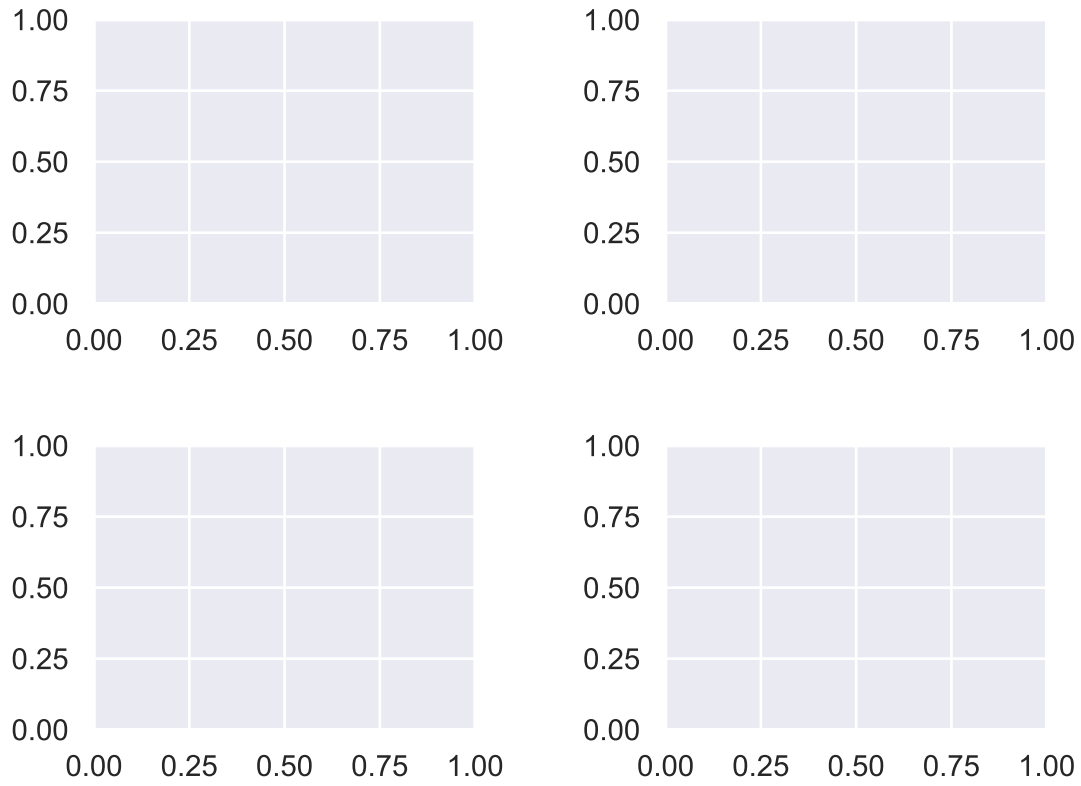

FIG. 6. Espace ajusté entre les sous-figures.

Pour chaque sous-figure `ax`, les commandes

```
ax.set_xlim(xmin, xmax)
ax.set_ylim(ymin, ymax)
```

avec des valeurs réelles pour les bornes $x_{\min}, x_{\max}, y_{\min}, y_{\max}$ déterminent la boîte $[x_{\min}, x_{\max}] \times [y_{\min}, y_{\max}]$ dans laquelle les *Artists* agissant sur `ax` ont le droit de dessiner. Sans ces commandes, `Matplotlib` essaie de deviner une boîte appropriée. Par ailleurs, par défaut, le module `Matplotlib` représente les sous-figures en essayant de remplir toute la figure qui les contient. C'est un bon aspect en général, mais par conséquent, les échelles des deux axes (abscisse et ordonnée) ne sont pas en général respectées (par exemple, une unité en abscisse peut avoir la même taille que deux unités en ordonnée). Si l'on veut fixer le ratio entre abscisse et ordonnée, on utilise `ax.set_aspect(r)`, avec $r = 1$ pour que les deux axes aient la même échelle.

Le titre de la sous-figure `ax` est spécifié par `ax.set_title("titre")`. Si l'on a plusieurs sous-figures et si l'on veut un titre global, on utilisera `plt.suptitle("titre global")`.

Axes, légendes. Voyons maintenant comment spécifier les décorations usuelles d'une sous-figure `ax`.

- Les labels des deux axes sont spécifiés par `ax.set_xlabel("label des abscisses")` et `ax.set_ylabel("label des ordonnées")`.

- La graduation sur chaque axe peut être fixée avec les commandes `ax.set_xticks(L)` et `ax.set_yticks(L)`, où `L` est une liste ou un *array*. En particulier, pour avoir une graduation de l'axe des abscisses en chaque entier de $[\![0, n]\!]$, on utilisera la commande `ax.set_xticks(np.arange(n+1))`. De même, pour obtenir une graduation avec N points entre a et b, on utilisera `ax.set_xticks(np.linspace(a, b, N))`.
- Si on veut retirer les axes, on utilisera `ax.set_axis_off()`.
- Les options des commandes `set_xlabel`, `set_ylabel` et `set_title` permettent de positionner exactement comme l'on veut les labels des axes et les titres des sous-figures.
- Si un *Artist* `artist` agit sur une sous-figure `ax` par la commande

```
ax.artist(args, label="str", color=c)
```

alors le dessin effectué par `artist` aura la couleur donnée par l'argument `color`. Les couleurs noir, rouge, vert, bleu sont données respectivement par les caractères `"k"`, `"r"`, `"g"` et `"b"`. Il y a d'autres couleurs simples données par des mots clés (par exemple, `"purple"`), et on peut sinon donner un triplet RGB $(\rho, \gamma, \beta) \in [0, 1]^3$. La commande `ax.legend()` ajoute ensuite à la sous-figure `ax` un petit encart avec les labels et couleurs des différents *Artists*.

Par exemple, voici un dessin optimal du graphe de la fonction $x \mapsto \frac{x}{2}$ sur l'intervalle $[0, 2]$:

```
fig, ax = plt.subplots()
ax.plot([0,2], [0,1], label="$y=\\frac{x}{2}$", color="b")
ax.set_xlim(0, 2)
ax.set_ylim(0, 1)
ax.set_xlabel("abscisse", labelpad=10, loc="right")
ax.set_ylabel("ordonnée", labelpad=15, loc="top")
ax.set_title("graphe d'une fonction", pad=30)
ax.set_xticks(np.linspace(0, 2, 5))
ax.set_yticks(np.linspace(0, 1, 5))
ax.legend()
ax.set_aspect(1)
plt.show()
```

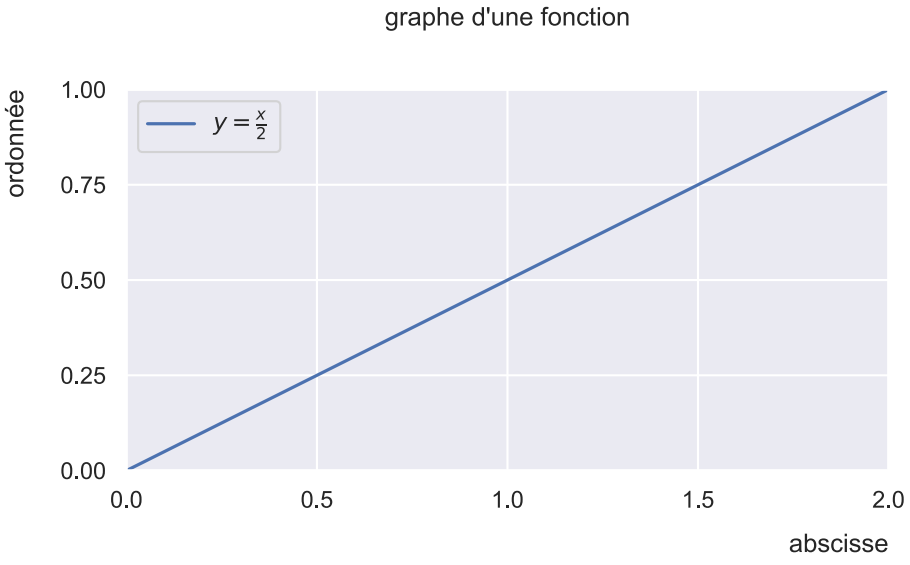

FIG. 7. Graphe de la fonction $x \mapsto \frac{x}{2}$ avec `Matplotlib`.

Lignes, graphes. On peut relier des points $(x_1, y_1), (x_2, y_2), \ldots, (x_l, y_l)$ en utilisant la commande `ax.plot(x, y)`, où `x` est la liste des abscisses $(x_1, \ldots, x_l)$ donnée sous forme de liste ou d'*array*, et `y` est la liste des ordonnées $(y_1, \ldots, y_l)$. Divers arguments optionnels permettent d'ajuster l'épaisseur, la forme, la couleur, etc. du trait. Voici par exemple le résultat de :

```
ax.plot([0, 3, 2, 0], [0, 0, 2, 1], color="orange", linestyle="--")
```

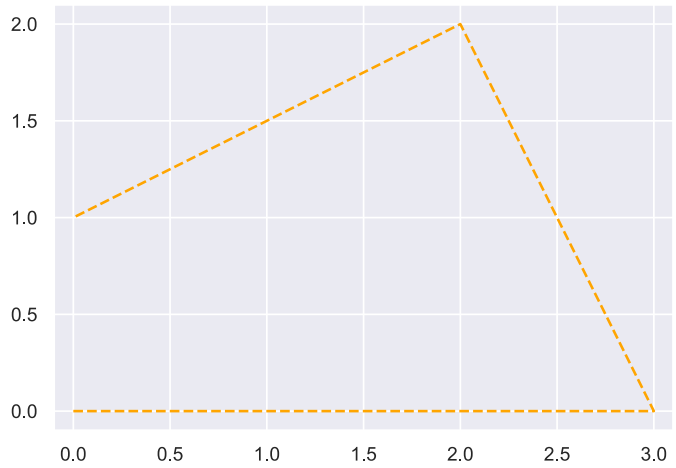

FIG. 8. Une ligne brisée dessinée avec la commande `plot`.

Si l'on veut dessiner le graphe d'une fonction $y = f(x)$, on peut utiliser une ligne reliant de nombreux points (x_i, y_i) avec $y_i = f(x_i)$. Plus précisément, la fonction :

```
def draw_function(f, a, b, samples=500):
    xx = np.linspace(a, b, samples)
    fig, ax = plt.subplots()
    ax.plot(xx, f(xx))
    plt.show()
```

dessine entre deux bornes a et b le graphe de la fonction f. Voici le résultat de cette commande avec `f = (lambda x : np.sin(x) * np.exp(-x/10))` et $[a, b] = [-5, 5]$.

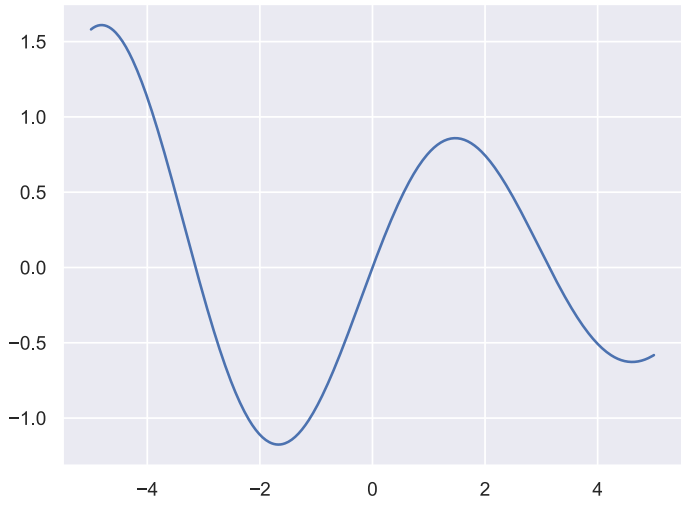

FIG. 9. Le graphe d'une fonction, approché par une ligne brisée avec de nombreux points.

Sur le même principe, on peut dessiner n'importe quelle courbe paramétrée $\gamma(t) = (x(t), y(t))$ avec $t \in [0, T]$. Il suffit de créer un échantillonnage `E = np.linspace(0, T, samples)` et de relier les points avec `ax.plot(x(E),y(E))`. Il peut y avoir des petits problèmes de lissage (la courbe dessinée a des irrégularités non souhaitées), qu'on peut généralement résoudre en augmentant le nombre `samples`.

Représentation de données. Pour conclure cette introduction à `Matplotib`, voyons comment représenter des données, qui sont par exemple obtenues à partir d'une expérience aléatoire :

- nuage de points. Si l'on a un ensemble de points du plan $((x_1, y_1), (x_2, y_2), \dots, (x_l, y_l))$, on peut le dessiner avec `ax.scatter(x, y)`. Par exemple,

```
fig, ax = plt.subplots()
x = np.linspace(0, 1, 100) + scs.norm(0, 0.1).rvs(size=100)
y = 0.5*x + 0.2 + scs.norm(0, 0.1).rvs(size=100)
ax.scatter(x, y, color=np.where(y>0.5*x+0.2, "g", "b"))
plt.show()
```

dessine un nuage de points autour de la droite d'équation $y = \frac{x}{2} + \frac{1}{5}$, avec des erreurs d'observation gaussiennes. On a mis comme paramètre de couleur une liste, qui donne la couleur verte si $y_i > \frac{x_i}{2} + \frac{1}{5}$ et la couleur bleue sinon.

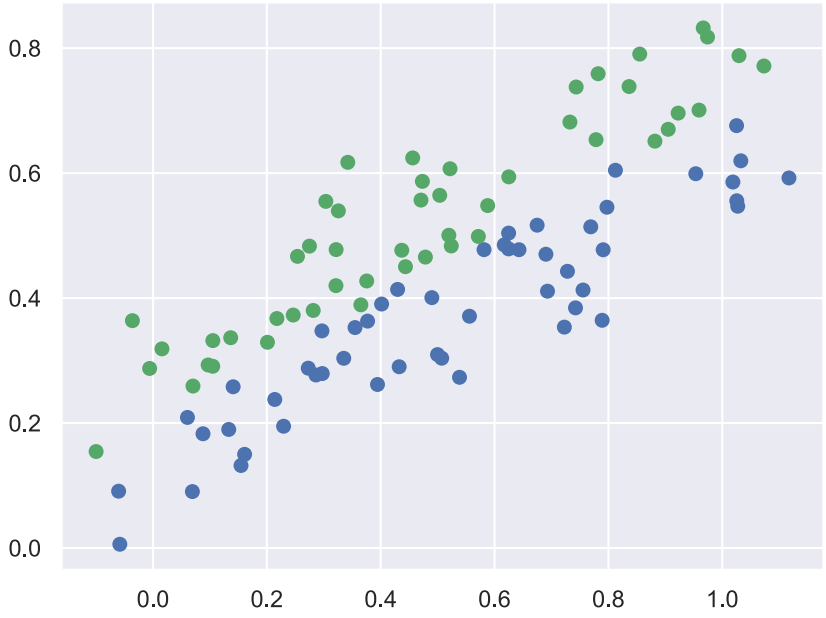

FIG. 10. Un nuage de points dessiné avec la commande `scatter`.

- fonction de répartition empirique. Si l'on a un échantillon `X` (sous forme d'*array*), on peut en représenter la fonction de répartition avec la commande `ax.ecdf(X)`. Par exemple,

```
fig, ax = plt.subplots()
X = scs.norm.rvs(size=100)
ax.ecdf(X)
ax.set_xlim(min(X), max(X))
ax.set_ylim(-0.1, 1.1)
plt.show()
```

dessine la fonction de répartition empirique d'un vecteur gaussien de taille $N = 100$.

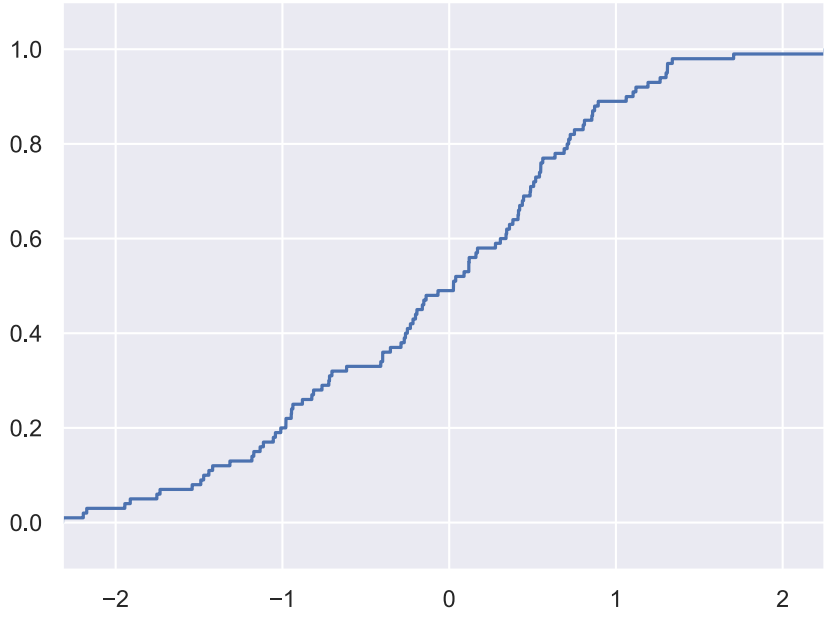

FIG. 11. Une fonction de répartition empirique obtenue avec la commande `ecdf`.

4. NetworkX.

Le module `NetworkX` permet de manipuler des graphes (paires $G = (V, E)$ formées d'un ensemble de sommets V et d'un ensemble E d'arêtes), de les dessiner et de calculer diverses statistiques sur ces objets. On ajoutera à la liste d'imports usuelle la commande :

```
import networkx as nx
```

Construction de graphes. Un graphe vide est créé avec la commande `nx.Graph()`. Ses ensembles de sommets et d'arêtes sont alors vides :

```
>>> G = nx.Graph()
>>> G.nodes(), G.edges()

(NodeView(()), EdgeView([]))
```

On peut ajouter des sommets au graphe avec la commande `G.add_nodes_from(S)`, où `S` est une liste de sommets ; et des arêtes avec la commande `G.add_edges_from(E)`, où `E` est une liste de paires de sommets. Par exemple, on crée un triangle comme suit :

```
>>> G.add_nodes_from([0, 1, 2])
>>> G.add_edges_from([(0,1), (0,2), (1,2)])
>>> G.nodes(), G.edges()

(NodeView((0, 1, 2)), EdgeView([(0, 1), (0, 2), (1, 2)]))

>>> G.number_of_nodes(), G.number_of_edges()

(3, 3)
```

On expliquera plus loin comment dessiner sur une figure `Matplotlib` une représentation de ce graphe.

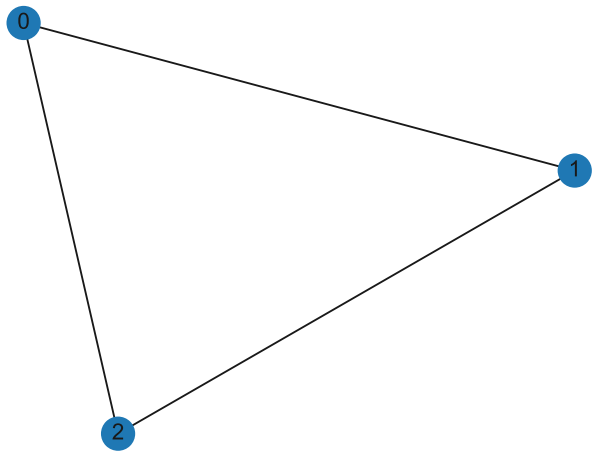

FIG. 12. Un triangle encodé avec NetworkX et dessiné avec Matplotlib.

La construction précédente est tout à fait appropriée lorsque le graphe est construit récursivement, et peut croître au fur et à mesure d'un algorithme. Alternativement, on peut définir un graphe en donnant un dictionnaire dont les clés sont les sommets, et dont les entrées sont les listes des voisins :

```
>>> H = nx.Graph({0 : [1,2], 1: [0,2], 2: [0,1]})
>>> nx.is_isomorphic(G, H)

True
```

Pour définir un graphe, on peut aussi utiliser la matrice d'adjacence. Si `G` est un graphe `NetworkX`, sa matrice d'adjacence est obtenue avec la commande `nx.to_numpy_array(G)`. Réciproquement, si `A` est un *array* carré dont les entrées sont dans $\{0, 1\}$, alors `nx.Graph(A)` renvoie le graphe dont la matrice d'adjacence est `A`.

```
>>> A = nx.to_numpy_array(G)
>>> A

array([[0., 1., 1.],
       [1., 0., 1.],
       [1., 1., 0.]])

>>> K = nx.Graph(A)
nx.is_isomorphic(G, K)

True
```

La classe `nx.Graph` est adaptée à la manipulation de graphes *simples* (pas d'arêtes multiples), éventuellement avec des *boucles* basées en les sommets, et *non orientés* (les arêtes sont des paires $\{v, w\}$ de sommets, sans prendre en compte l'ordre). Le module `NetworkX` définit également des classes permettant de manipuler des *multigraphes* (graphes avec éventuellement des arêtes multiples), ou des *graphes orientés*. En particulier, pour les graphes orientés, on utilisera la classe `nx.DiGraph`. Ainsi, la commande `nx.DiGraph({0 : [1], 1: [2], 2: [0]})` crée un triangle orienté dont les arêtes forment un cycle. On peut oublier l'orientation des arêtes d'un graphe orienté `D` avec la commande `D.to_undirected()`.

Manipulation. On a déjà expliqué comment ajouter des sommets ou des arêtes à un graphe `G`. Les méthodes `G.remove_nodes_from` et `G.remove_edges_from` correspondent aux opérations

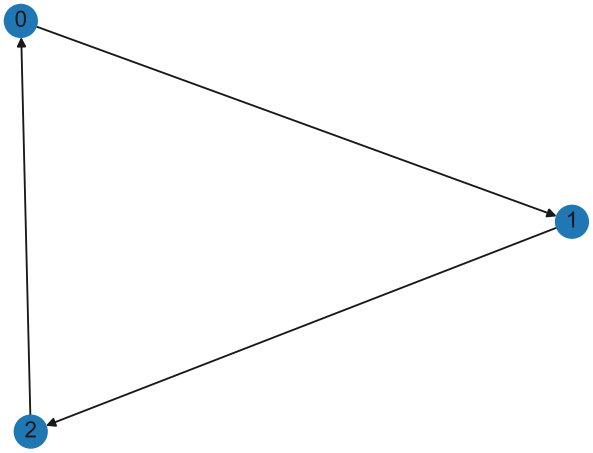

FIG. 13. Un triangle orienté avec NetworkX.

inverses de suppression d'un sommet ou d'une arête. Les commandes `G.number_of_nodes()` et `G.number_of_edges()` calculent le nombre de sommets et le nombre d'arêtes du graphe `G`. Dans tout ce qui suit, on manipulera le graphe dessiné ci-dessous :

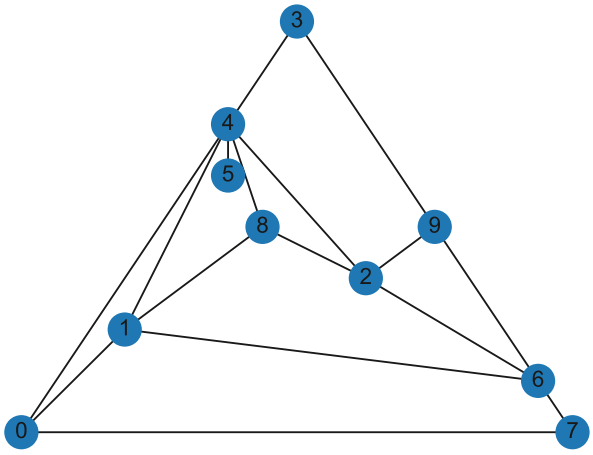

FIG. 14. Un graphe non orienté avec 10 sommets et 16 arêtes.

Chaque sommet ou arête de `G` peut être étiqueté : les étiquettes sont des paires (clé, valeur) d'un dictionnaire. Par exemple, si l'on attribue aux arêtes de `G` des poids aléatoires avec la commande :

```
for e in G.edges():
    G.edges[e]["weight"] = random.randint(0, 10)
```

alors on obtient le graphe étiqueté dessiné ci-après (où on a juste représenté les arêtes et leurs poids). Chaque sommet `v` du graphe a déjà par défaut un dictionnaire, dont les clés sont les voisins de `v`, et dont l'entrée associée à la clé `w` est le dictionnaire des étiquettes de l'arête (v, w) (par défaut, ce dictionnaire est vide).

```
>>> G[3]

AtlasView({4: {'weight': 0}, 9: {'weight': 3}})
```

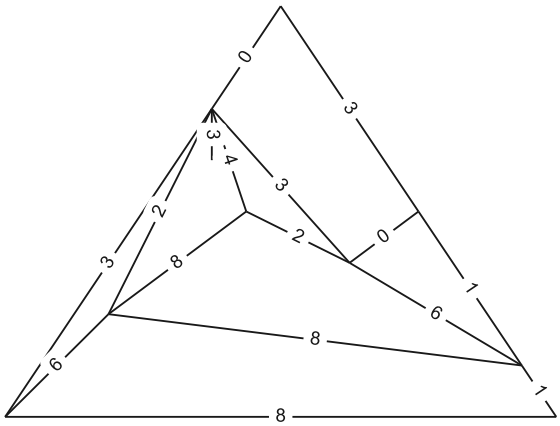

FIG. 15. Chaque arête ou sommet d'un graphe encodé avec NetworkX peut être étiqueté, avec une ou plusieurs étiquettes stockées dans un dictionnaire.

La somme de tous les poids des arêtes vis-à-vis des étiquettes de label "weight" est obtenue avec la méthode size(weight="weight"). Par exemple :

```
>>> G.size(weight="weight")

58.0
```

Algorithmes. Une longue liste d'algorithme plus ou moins complexes est implantée dans NetworkX. Donnons-en quelques uns :

- Les commandes nx.is_bipartite(G), nx.is_connected(G), nx.is_planar(G) et nx.is_tree(G) testent si G est un graphe bipartite, un graphe connexe, un graphe planaire et un arbre. Pour notre exemple :

```
>>> nx.is_bipartite(G), nx.is_connected(G), nx.is_planar(G), nx.is_tree(G)

(False, True, True, False)
```

- La commande nx.connected_components(G) renvoie un itérateur dont les objets sont les parties de l'ensemble des sommets de G qui en forment les composantes connexes. On peut aussi demander directement la composante connexe contenant un sommet v avec nx.node_connected_component(G, v).

```
>>> nx.node_connected_component(G, 0)

{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

- On peut tester l'isomorphisme entre deux graphes G et H (existence d'une bijection entre les sommets préservant l'adjacence) avec la commande nx.is_isomorphic(G, H).
- On peut facilement calculer des distances entre les sommets d'un graphe G. Par exemple, nx.shortest_path(G, v, w) trouve un chemin de longueur minimale entre les deux sommets v et w, et nx.shortest_path_length(G, v, w) calcule la distance de graphe entre ces sommets (la longueur minimale d'un chemin). Ces deux fonctions peuvent prendre en argument un poids sur les arêtes.

```
>>> nx.shortest_path_length(G, 0, 6)
```

```
2

>>> nx.shortest_path(G, 0, 6)

[0, 1, 6]

>>> nx.shortest_path_length(G, 0, 6, weight="weight")

7

>>> nx.shortest_path(G, 0, 6, weight="weight")

[0, 4, 3, 9, 6]
```

Dessin. On peut dessiner un graphe `G` encodé avec `NetworkX` en utilisant tout simplement la commande `nx.draw(G)`. Toutefois, il est préférable d'utiliser une syntaxe un peu plus compliquée, mais qui donne un bien meilleur contrôle du dessin, et qui permet de mieux comprendre ce qui se passe. On conseille ainsi :

```
fig, ax0 = plt.subplots()
pos0 = nx.spring_layout(G)
nx.draw_networkx(G, pos=pos0, ax=ax0, with_labels=True)
ax0.set_axis_off()
plt.show()
```

Expliquons en détail cette suite d'instructions :

- Le dessin du graphe `G` sera posé sur une figure `Matplotlib`, qu'on introduit comme décrit dans la section précédente. Ceci permet de combiner toute la puissance du module `NetworkX` avec les bonnes pratiques de dessin vues précédemment (sous-figures, titres, légendes, etc). Dans la commande principale `nx.draw_networkx`, la sous-figure sur laquelle on agit est précisée par l'argument `ax=ax0`.
- Pour dessiner un graphe, il faut savoir où placer ses sommets. Plusieurs choix sont possibles : aléatoirement dans le plan, en essayant d'avoir le moins d'intersections possibles entre les arêtes, sur un cercle, etc. La commande `pos0 = nx.spring_layout(G)` calcule un dictionnaire dont les clés sont les sommets, et dont les entrées sont les coordonnées où placer chaque sommet. Par exemple, les dessins précédents avaient pour *Layout* :

```
>>> nx.planar_layout(G)

{0: array([-1.        , -0.39759036]),
 1: array([-0.63855422, -0.15662651]),
 2: array([ 0.20481928, -0.03614458]),
 3: array([-0.03614458,  0.56626506]),
 4: array([-0.27710843,  0.3253012 ]),
 5: array([-0.27710843,  0.20481928]),
 6: array([ 0.80722892, -0.27710843]),
 7: array([ 0.92771084, -0.39759036]),
 8: array([-0.15662651,  0.08433735]),
 9: array([0.44578313, 0.08433735])}
```

On précise les emplacements des sommets avec l'argument `pos=pos0` dans la commande principale `nx.draw_networkx`. On peut définir soit-même à la main un dictionnaire contenant les emplacements, ou utiliser les commandes `nx.spring_layout(G)`,

`nx.planar_layout(G)` et `nx.circular_layout(G)` pour produire automatiquement des emplacements *équilibrés* (les sommets sont placés en se repoussant les uns les autres, comme s'il y avait des ressorts), *planaires* (si le graphe est planaire, les arêtes du dessin ne s'intersectent pas) et *circulaires* (les sommets sont placés sur un cercle). On renvoie à la documentation de `NetworkX` pour d'autres *Layouts* classiques.

- On peut ensuite ajouter de très nombreux arguments optionnels ; par exemple, l'argument `with_labels = True` force les numéros des sommets à apparaître. On peut préciser la taille des sommets, leur forme, leur couleur, l'épaisseur des arêtes, etc.
- Si l'on veut dessiner séparément les sommets et les arêtes, on peut utiliser les commandes `nx.draw_networkx_nodes` et `nx.draw_networkx_edges` ; c'est parfois utile pour gérer précisément les options d'affichage. Dans ce cas, il est essentiel d'avoir stocké à l'avance dans un *array* `pos0` les emplacements des sommets ; on utilise alors ce même *array* comme argument des différentes commandes de dessin.