Introduction à Python pour le cours de modélisation numérique

Contents

1 Fondamentaux

1.1 Aperçu

Python est un langage de programmation open-source et gratuit, interprété, polyvalent et convivial. Il est largement utilisé dans le développement de logiciels, l'analyse de données, la science des données, l'apprentissage automatique, l'automatisation de tâches, et bien plus encore.

Ce document s'adresse aux étudiants du cours de modélisation numérique qui n'ont jamais utilisé Python auparavant. Le seul prérequis est une initiation élémentaire à la programmation, incluant une introduction aux concepts de variable et de boucle.

Concrètement, utiliser Python revient à écrire un script Python (un fichier texte éditable avec une extension .py, par exemple, monScript.py, qui contient une suite de commandes) et à l'exécuter dans un terminal/une console avec la commande suivante :

```
python monScript.py
```

à condition que Python et les bibliothèques appelées dans le script soient bien installés.

Dans ce cours, nous utiliserons des `Jupyter Notebooks` (par exemple, monNotebook.ipynb), qui permettent d'alterner entre des blocs de texte (écrits en `Markdown`, incluant des équations, des images, etc.) et des blocs de code Python. Cependant, il est également possible de mettre le code dans un script et de l'exécuter comme indiqué ci-dessus. Nous utiliserons l'éditeur Visual Studio Code (VS Code) pour éditer nos Jupyter Notebooks, les exécuter et visualiser les résultats facilement.

VS Code est un environnement de développement populaire (pour Python et bien d'autres langages), qui fournit une interface graphique (voir figure 1). Avant toute chose, il est important de se placer dans un répertoire dédié. Le répertoire courant de VS Code est affiché en haut à gauche. VS Code lit et écrit les données dans ce répertoire.

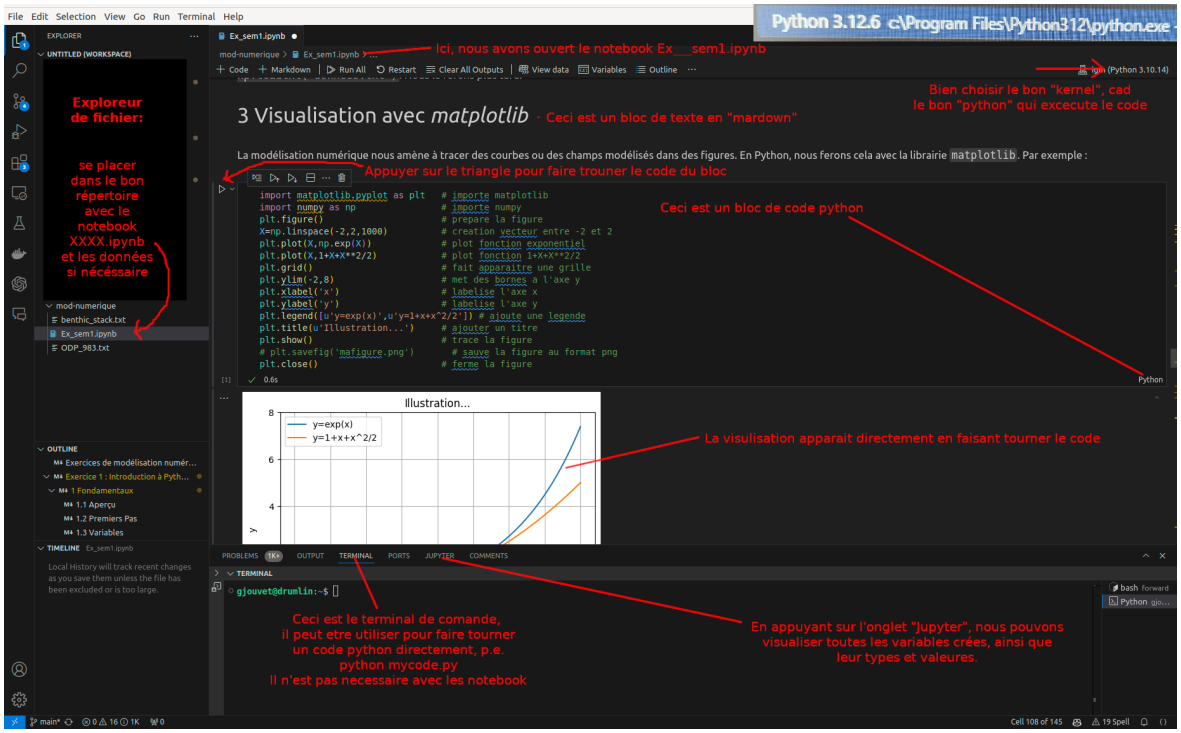

Figure 1: Vue de l'éditeur "VS Code" et de ses composantes essentielles.

Dans la figure 1, nous voyons plusieurs fenêtres :

- L'explorateur de fichiers à gauche, où apparaissent le notebook et les fichiers de données.
- Le notebook, composé de blocs de texte (Markdown) et de code (Python), chacun pouvant être exécuté en appuyant sur le triangle.
- La fenêtre en bas offre un terminal utile uniquement si l'on souhaite exécuter un script Python, mais elle permet aussi d'explorer les valeurs des variables du workspace en appuyant sur "Jupyter".

Attention : avant d'exécuter un bloc de code, il est primordial de sélectionner le "kernel". Ici, nous voulons un kernel Python spécifique (indiqué en haut à droite de la figure 1), car il est possible de sélectionner une autre version de Python (ou une version personnalisée avec certaines bibliothèques), ou même un autre langage de programmation.

1.2 Premier pas

Python peut s'utiliser comme une calculatrice ; on peut y effectuer des opérations élémentaires, que l'on peut afficher au moyen de la fonction `print` :

```
>>> print(1+3*4)     #  donnera 13
>>> print(1+10**3)   #  donnera 1001 (** est la puissance)
>>> print(10/3)      #  donnera 3.333
>>> print(10//3)     #  donnera 3 (// est la division euclidienne)
>>> print(10%3)      #  donnera 1 (% est le reste de la div.
   euclidienne)
```

Notons que "#" permet d'écrire des commentaires (ce qui suit "#" n'est pas interprété). Commenter un code est essentiel pour que d'autres puissent en comprendre le sens ! Prenez donc le temps de documenter généreusement votre code, pour vous et pour les autres.

1.3 Variables

On assigne une variable avec le signe "=". Le nom d'une variable est sensible à la casse (majuscule et minuscule), peut être aussi long qu'on le souhaite et peut contenir des chiffres, à l'exception du premier caractère.

```
>>> d  = 8                    # variable scalaire
>>> R  = 1.6524981            # variable scalaire
>>> print( d + R)             # donnera 9.6524981
```

En Python, il est commode d'utiliser l'opérateur d'incrémentation "+=" (il existe aussi "-=" , "/=" et "*=").

```
>>> n = 8
>>> n += 1                    # equivalent a : n = n + 1
>>> print(n)                  # retournera 9
```

Une variable Python a un type : "int" (nombre entier), "float" (nombre à virgule), "bool" (booléen). On peut consulter son type via la commande "type" :

```
>>> d  = 8
>>> R  = 1.6524981
>>> print(type(d))            # donnera int
>>> print(type(R))            # donnera float
```

Nous pouvons assigner une variable de type "str" (string) qui contient une liste de lettres ordonnée :

```
>>> mon_fichier = 'temperature.dat'
>>> print(mon_fichier)        # donnera  temperature.dat
>>> print(type(mon_fichier))  # donnera  str
```

Un booléen est une variable qui vaut Vrai (True) ou Faux (False). On peut lui appliquer les opérateurs `and`, `or` et `not` :

```
>>> b = True
>>> type(b)                          # donnera  bool
>>> print(not b or (b and not b)) # donner False
```

1.4 Listes

En Python, il existe un type très commode : les listes, qui peuvent contenir n'importe quel type :

```
>>> maListe = [1, 3.14, "bofs"]
>>> print(maListe)
>>> print(type(maListe))      # donnera  list
```

On accède à la longueur d'une liste avec la fonction `len`. On accède à un élément donné avec l'opérateur `[]`, en utilisant l'indice de l'élément comme argument.

Attention: En Python, les indices vont de `0` à `len(maListe)-1`.

```
>>> len(maListe)                                  # donnera 3
>>> print(maListe[0], maListe[1], maListe[2])
```

Attention : Pour utiliser des vecteurs ou des matrices (essentiels dans ce cours), nous utiliserons la bibliothèque `numpy` plutôt que les listes, leur utilisation étant similaire.

Notons que les chaînes de caractères (strings) peuvent être vues comme des listes :

```
>>> A = "cecicela"
>>> print(A[0:2])     # retournera les deux premier charactere : "ce"
```

1.5 Fonctions prédéfinies et aide

Python dispose d'un certain nombre de fonctions prédéfinies :

```
>>> max(1,2)                          # donnera 2
>>> l = [1,2,3,4,5]
>>> print(max(l), min(l), sum(l))  # donnera 5, 1, 15
>>> int(3.5)                      # donnera 3 (la partie entiere de 3.5)
```

Pour obtenir de l'aide directement ("help") dans cette fenêtre, utilisez

```
>>> help(max)
```

1.6 Clauses conditionnelles `if`, `elif`, `else`

Il est souvent nécessaire d'utiliser des clauses conditionnelles lors de la création d'un modèle. Par exemple : si la température à la surface d'un glacier dépasse 0°C, alors la glace fond. La structure est la suivante :

```
>>> if T<0:
>>>     print("La temperature est negative")
>>> else:
>>>     print("La temperature est positive")
```

En Python, on ajoute des espaces en début de ligne pour marquer de manière visible les délimitations d'un bloc d'instructions : on appelle ces espaces des **indentations**.

En Python, la syntaxe est fixée par l'indentation.

Tout nouveau bloc d'instructions nécessite une indentation supplémentaire, et ce bloc prend fin lorsque cette indentation disparaît. Le nombre d'espaces utilisés pour l'indentation n'a pas d'importance ; il faut juste toujours utiliser le même nombre au sein d'un même bloc d'instructions.

Les conditions demandées par des clauses sont des "opérateurs relationnels" :

- `<` signifie plus petit que, `<=` signifie plus petit ou égal à
- `>` signifie plus grand que, `>=` signifie plus grand ou égal à
- `==` signifie égal à, `!=` signifie pas égal à

Notez que le double `==` indique une comparaison entre deux valeurs (`a==b` : *`a` est-il égal à `b` ?*) et non pas l'attribution d'une valeur à une variable avec le simple `=` (`a=b` : *`a` se voit attribuer la valeur de `b`*). Les clauses conditionnelles peuvent aussi suivre plusieurs conditions en utilisant les termes `if` (si condition particulière), `elif` (sinon et condition particulière), et `else` (sinon). Notons que `else` peut être omis :

```
>>> if n==0:
>>>     print('n est egal a zero')
>>> elif n==1:
>>>     print('n est egal a un')
```

1.7 Boucles `for`, `while`

Un modèle numérique basé sur des équations discrétisées dans le temps utilise des boucles pour itérer d'un pas de temps au suivant. Mais de manière plus générale, une boucle est très utile pour effectuer des calculs incrémentaux. Dans le cas d'une discrétisation explicite du temps, il est nécessaire de faire appel à une boucle. Par contre, il est souvent possible d'éviter une boucle pour la discrétisation dans l'espace, au profit d'un calcul direct vectoriel / matriciel. Cette dernière option est bien plus rapide et est appelée "opération vectorisée" dans la mesure où celle-ci s'applique au vecteur / à la matrice entière.

Comme pour les clauses conditionnelles, il est indispensable d'utiliser une indentation pour déterminer le bloc d'instructions qui est itéré dans la boucle.

En python, il existe plusieurs comandes de boucles iteratives. Nous voyons ici `for` et `while`:

- `for x in E:` permet d'itérer sur tout contenu itérable de taille connue, comme par exemple une suite de nombres ou une liste. Pour une boucle sur des entiers, on utilise `range` pour construire l'ensemble des nombres voulus et itérer dessus. La convention Python est : `range(n)` contient les nombres de `0` à `n-1` et `range(m,n)` contient les nombres de `m` à `n-1`. Par example,

```
>>> for i in range(10):
>>>     print(i)      # cela va retourner 0, 1, 2, ...,9
```

tandis que faire une boucle sur une liste s'ecrit:

```
>>> maListe = [1, 3.14, 4, "eggs"]
>>> for x in maListe:
>>>     print(x) # cela va retourner les elts de maListe
```

- `while i<n:` permet d'itérer tant qu'une condition est remplie, par exemple :

```
>>> n=0
>>> while n<10:
>>>     print(i)     # cela va retourner 0, 1, 2, ...,9
>>>     n+=1
```

Attention : Oublier le symbole : ou se tromper sur l'indentation est une source classique d'erreur.

On peut imbriquer des instructions et des boucles en ajoutant un niveau d'indentation à chaque fois. Exemple de conditions imbriquées :

```
>>> for i in range(3):
>>>     for j in range(10):
>>>         print(i,j)
```

```
>>> for i in range(5):
>>>     j = 0
>>>     while j<600:
>>>         if i + j == 15:
>>>             print('la condition i + j == 15 est remplie')
>>>             break  # cela sort prematurement de la boucle
>>>         j += 1
```

Le mot clef `break` permet la sortie prematurée d'une boucle.

1.8 Fonctions

On peut imbriquer des instructions et des boucles en ajoutant un niveau d'indentation à chaque fois. Exemple de conditions imbriquées :

```
>>> def somme(i,j):
>>>     return i+j
>>> print(somme(1,2))  # retournera 3
>>> print(type(somme)) # retournera  'fction'
```

Une fonction possède des arguments et peut retourner plusieurs outputs :

```
>>> def f(n):
>>>    if n<1:
>>>       return n+m,n-m
>>>    elif n<4:
>>>       return 2*n+1,2*n-1
>>>    else
>>>       return 0,0
>>> a,b = f(3)
>>> print(a,b)    # retournera 7,5
```

Une fonction s'arrête lorsqu'elle rencontre un `return`.

1.9 Modules

Python possède un très grand nombre d'outils (appelés modules ou bibliothèques) que les utilisateurs peuvent installer[1], charger/importer (via la commande `import`), et utiliser. Par exemple,

- Le module `math` permet d'avoir à disposition le chiffre π, et bien d'autres opérations mathématiques :

```
>>> import math
>>> print(math.pi) # retournera 3.141592653589793
```

- Le module `numpy` permet de faire du calcul matriciel :

[1] Les bibliothèques classiques `numpy` et `matplotlib` sont installées par défaut sur les machines de l'UNIL. Il se peut que vous deviez les installer si vous utilisez vos machines personnelles

```
>>> import numpy as np
>>> A= np.ones((3,4))
>>> B= np.zeros((3,4))
>>> print(A+B)
```

- le module `matplotlib` permet de faire des `plots`:

```
>>> import matplotlib.pyplot as plt
>>> import numpy as np
>>> X = np.random.uniform(10,100,(10))
>>> plt.plot(A)
```

Une grande force (et la raison du succès) de Python vient du fait que Python possède un nombre considérable de bibliothèques développées par la communauté. Par exemple :

- `scipy` est une bibliothèque scientifique : algèbre linéaire, fonctions spéciales, analyse de Fourier, etc.
- `scikit`, `tensorflow`, `pytorch` permettent de faire de l'apprentissage automatique (IA),
- `geopandas`, `rasterio`, `netcdf4` sont des outils d'information géographique.
- OpenCV permet de faire de l'analyse d'image,
- `igm` (github.com/jouvetg/igm) – developpé ici à la FGSE – permet de modéliser les glaciers.

Dans ce cours, nous n'utiliserons que les bibliothèques `numpy`, `math`, `matplotlib`.

2 Calculs matriciels avec `numpy`

L'élément de base pour le cours de modélisation numérique est une matrice de dimension $n \times m$ composée de valeurs numériques (n = lignes, m = colonnes). Une matrice de dimension 1×1 représente donc un scalaire. Une matrice de dimension $n \times 1$ est un vecteur colonne de dimension n ; de même pour un vecteur ligne, qui est une matrice $1 \times m$. En Python, la gestion des matrices et des calculs que l'on peut faire dessus se fait via la bibliothèque `numpy`. N'oubliez pas de charger `numpy` via `import numpy as np` pour l'utiliser.

2.1 Creation de matrice (array) `numpy`

Il y a plusieurs façons de créer des objets `array numpy` qui décrivent des matrices. Pour des petites matrices/vecteurs, cela peut se faire avec une liste (pour un vecteur) ou une liste de listes (pour une matrice), laquelle est décrite ligne par ligne :

```
>>> import numpy as np                          # chargement de numpy
>>> A = np.array([0,1,2])                       # creation vect 1D 0,1,2
>>> B = np.array([[0,1,2],[3,4,5],[6,7,8]]) # creation matrice 2D
>>> print(B)
[[0 1 2]
 [3 4 5]
 [6 7 8]]
```

On peut aussi construire des matrices élémentaires comme ceci :

```
>>> import numpy as np
>>> A = np.zeros((3,3)) # creation matrice 2D de dim (3,3) de zeros
>>> B = np.ones((2,2))  # creation matrice 2D de dim (2,2) de uns
>>> C = np.eye((10))    # creation matrice identitee carre de dim 10
>>> D = np.arange(10)   # creation vecteur [0,1,...,9]
>>> E = np.linspace(0,1,11) # creation vecteur [0,0.1,...,1]
```

Ce formalisme est pratique notamment pour créer des matrices de grandes dimensions.

2.2 Forme (shape) d'un array numpy

La dimension (taille) d'une matrice est une donnée essentielle ; cela s'appelle la forme (ou `shape`) :

```
>>> import numpy as np
>>> A = np.array([0,1,2])
>>> print(np.shape(A))                          # retourne (3)
>>> B = np.array([[0,1,2],[3,4,5],[6,7,8]])
>>> print(np.shape(B))                          # retourne (3,3)
>>> C = np.eye((10))
>>> print(np.shape(C))                          # retourne (10,10)
```

La forme (`shape`) possède un indice pour un vecteur et deux indices pour une matrice. En prenant la transposée d'une matrice, la shape (a, b) devient (b, a) :

```
>>> B=np.transpose(A)
```

2.3 Accés indices, slicing, incrémentation

Comme pour les listes, on accède à un élément du tableau avec les crochets. Les indices vont toujours de `0` à `n-1`, où `n` est la dimension donnée par la shape. Pour un tableau à plusieurs dimensions, on place plusieurs indices dans les crochets : `Z[0]` est la première ligne de `Z`, `Z[0,0]` est le coefficient $(0, 0)$, première ligne, première colonne. Enfin, `numpy` propose toutes les options utiles de slicing avec `:`, qui permet d'extraire des sous-matrices :

```
>>> import numpy as np
>>> B = np.array([[0,1,2],[3,4,5],[6,7,8]])
>>> print(B)
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
>>> print(B[1,1]) # retourne l'elt de la ligne 1 et colonne 1, soit 4
>>> print(B[1,:]) # retourne la premiere ligne, soit np.array([3,4,5])
>>> print(B[1:,0:2])  # retourne np.array([[3,4],[6,7]])
```

On peut pointer sur l'élément d'un vecteur par un indice positif en partant du début, ou par un indice négatif en partant de la fin, comme illustré dans cet exemple :

```
            Indices ->      0  1  2  3  4  5  6  5  8  9
            Matrice =      [3, 2, 8, 1, 3, 5, 2, 7, 1, 9]
            Indices ->    -10 -9 -8 -7 -6 -5 -4 -3 -2 -1
```

Il est possible d'extraire une sous-matrice en sélectionnant les indices en partant du début et/ou de la fin. Par exemple, cette ligne tronque les 2 premiers et 3 derniers éléments :

```
>>> E = np.array([0,1,2,3,4,5,6,7,8,9])
>>> print(E[2:-3])  # retourne np.array([3,4,5,6])
```

Il est également possible de ne considérer que les lignes de chaque X colonnes avec l'opérateur `::` : Par exemple, la commande suivante ne garde que les éléments de chaque 3 colonnes :

```
>>> E = np.array([0,1,2,3,4,5,6,7,8,9])
>>> print(E[::3])  # retourne np.array([0,3,6,9])
```

Nous pouvons également modifier des éléments d'une matrice avec le même formalisme :

```
>>> import numpy as np
>>> B[1,1] = 10  # modifie un element
>>> B[2,:] = 0   # modifie une ligne
```

2.4 Operations

Les opérations algébriques usuelles ainsi que les fonctions NumPy peuvent être appliquées directement sur un tableau et sont effectuées terme à terme, et ce de manière beaucoup plus rapide qu'en faisant une boucle sur tous les éléments du tableau. Lors d'une opération matricielle, les matrices doivent être de dimensions cohérentes :

```
>>> X = np.zeros(3)
>>> Y = np.ones(3)
>>> W = X + 2*Y      # multiplication et addition elt par elt
>>> Z = X - X/Y      # division et soustraction elt par elt
```

L'opérateur * effectue la multiplication terme à terme. Le produit scalaire entre deux vecteurs, le produit matrice-vecteur et le produit matriciel se font avec l'opérateur `numpy.dot` :

```
>>> A = np.array([[0,-1,-2],[-3,-4,-5],[-6,-7,-8]])
>>> B = np.array([[0,1,2],[3,4,5],[6,7,8]])
>>> V = np.array([-4,0,4])
>>> np.dot(A,B)     # produit matriciel
>>> np.dot(A,V)     # produit matrice-vecteur
```

Le module NumPy fournit une liste de fonctions usuelles en mathématiques : `sqrt`, `exp`, `cos`, `sin`, `log`, `floor`, `ceil`, `round`, etc. :

```
>>> print(np.exp(1.0), np.sin(np.pi/2))
>>> print(np.log(np.exp(1)), np.sqrt(2))
>>> print(np.floor(1.3), np.ceil(3.4))
```

et cela s'applique également de manière matricielle (avec les matrices définies ci-dessus) :

```
>>> np.exp(A)
>>> np.cos(B)
```

2.5 Nombres aléatoires

`numpy` possède de nombreux outils pour les nombres aléatoires via `np.random`. Ce sont des fonctions très utiles pour modéliser des systèmes naturels avec des caractéristiques d'apparence aléatoire, par exemple, le déclenchement de glissements de terrain dans un paysage.

La fonction `np.random.uniform(x, y, (a, b))` génère une matrice de `a` lignes et `b` colonnes remplies de **nombres aléatoires distribués uniformément** entre `x` et `y` :

```
# vecteur de 10 nb aleatoires entre 0 et 1
x_alea1 = np.random.uniform(0,1,10)
# matrice de 5x10 nb aleatoires entre -3 et 2
x_alea2 = np.random.uniform(-3,2,(5,10))
```

La fonction `np.random.normal(m, v, (a, b))` génère une matrice de `a` lignes et `b` colonnes remplie de **nombres aléatoires distribués normalement** autour de `m` avec un écart-type de `v` :

```
# vecteur de 10 nb aleatoires distr. normal. de moy 0 et ecart-type 1
x_alea1 = np.random.normal(0,1,(10))
# matrice de 3x4 nb aleatoires distr. normal. de moy 8 et ecart-type 2
x_alea2 = np.random.normal(8,2,(3,4))
```

2.6 Importer des données à partir d'un fichier

Pour importer un fichier de données préalablement fourni et nommé `donnees.txt` (qui présente une liste de chiffres sur deux lignes), on utilise la commande `np.loadtxt('donnees.txt')` :

```
>>> X=np.loadtxt('donnees.txt')
>>> X
array([[ 1. , 2.3, 9.8, -5.9, 3.1, 4. , -8.7, 0.6],
[ 12. , -7. , -3.5, 0.1, 6. , 7.4, -3.3, 7. ]])
>>> [X,Y]=np.loadtxt('donnees.txt')
>>> print (X)
[ 1. , 2.3, 9.8, -5.9, 3.1, 4. , -8.7, 0.6]
```

Attention : assurez-vous que le fichier `donnees.txt` soit bien dans le répertoire courant.

2.7 Fonction `np.copy`

Lorsque l'on crée un objet `A` qui est un `numpy array`, `A` pointe en fait sur l'espace mémoire où est stocké l'objet `A`. De fait, si l'on définit un objet `B` avec `B=A`, `B` est une copie de l'adresse de `A`. Ainsi, si l'on modifie `B` totalement ou partiellement, `A` sera également modifié. Ce comportement propre à Python (qui gère des adresses) peut générer des problèmes inattendus. Toutefois, il est possible de créer une copie physique `B` d'un objet `A` déjà défini avec la commande `B=np.copy(A)`. Une fois cette copie `B` réalisée, toute modification de `B` ne causera aucun changement sur `A` car il ne s'agit plus des mêmes objets.

3 Visualisation avec `matplotlib`

3.1 Visualisation non-interactive

La modélisation numérique nous amène à tracer des courbes ou des champs modélisés dans des figures. En Python, nous ferons cela avec la bibliothèque `matplotlib`. Par exemple, le code suivant produira la figure 2 (gauche).

```
>>> import matplotlib.pyplot as plt   # importe matplotlib
>>> import numpy as np                # importe numpy
>>> plt.figure()                      # prepare la figure
>>> X=np.linspace(-2,2,1000)          # creation vecteur entre -2 et 2
>>> plt.plot(X,np.exp(X))             # plot fonction exponentiel
>>> plt.plot(X,1+X+X**2/2)            # plot fonction 1+X+X**2/2
```

```
>>> plt.grid()                                  # fait apparaitre une grille
>>> plt.ylim(-2,8)                              # met des bornes a l'axe y
>>> plt.xlabel('x')                             # labelise l'axe x
>>> plt.ylabel('y')                             # labelise l'axe y
>>> plt.legend([u'y=exp(x)',u'y=1+x+x^2/2']) # ajoute une legende
>>> plt.title(u'Illustration...')               # ajouter un titre
>>> plt.show()                                  # trace la figure
>>> plt.savefig('mafigure.png')                 # sauve la figure au format png
>>> plt.close()                                 # ferme la figure
```

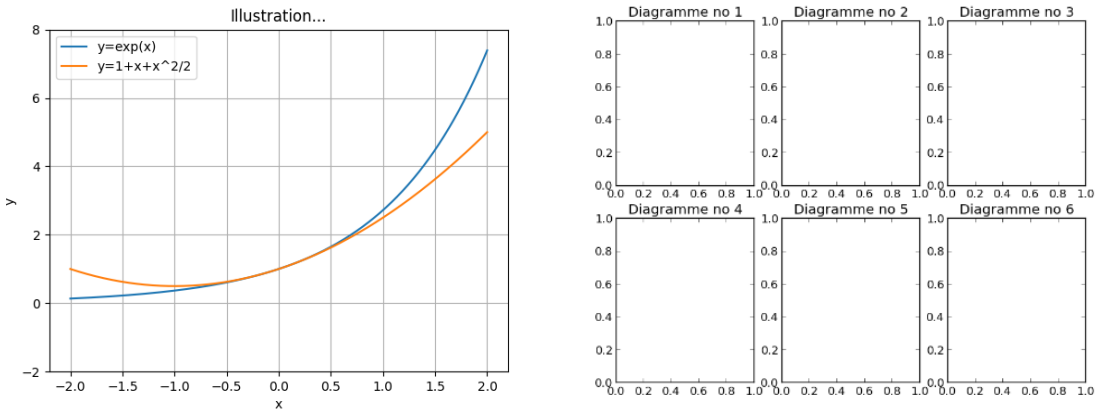

Figure 2: Illustration d'une figure produite avec matplotlib avec une simple fenetre ou multiple fenetres.

La fonction `plt.subplot(m, n, k)` découpe une même fenêtre graphique en un tableau $m \times n$ et insère les instructions de type `plt.` qui suivent dans la k-ième case. Le code suivant produira la figure 2 (droite).

```
>>> import matplotlib.pyplot as plt   # importe matplotlib
>>> plt.figure()
>>> for k in range(1,7):
>>>     plt.subplot(2,3,k)
>>>     plt.title(u'Diagramme no' + str(k))
>>> plt.show()
```

La fonction `plt.plot(X, Y)` permet de tracer une courbe à partir d'un vecteur `X` et d'un vecteur `Y`. Attention, les deux doivent avoir la même taille. La fonction `plt.plot(X, Y)` possède un bon nombre d'options (taille et type du trait, présence de marqueurs, etc.). Consultez l'aide `help(plt.plot)` pour plus d'informations.

Il existe de nombreuses autres fonctions de tracé qui seront utiles dans le cours :

- `plt.scatter`: Si `X` et `Y` sont deux listes ou tableaux de nombres réels et de même longueur, alors `plt.scatter(X, Y)` trace le nuage de points $(X[1], Y[1]), (X[2], Y[2]), \ldots, (X[n], Y[n])$. Le code suivant permet de réaliser la figure : 3 (gauche).

```
>>> import matplotlib.pyplot as plt   # importe matplotlib
>>> X=[5.6,5,3.5,7.6,2.2,4,1.9,8.8,7,5.1,3.5,4]
>>> Y=[10,5.9,7.8,6,4,3.7,10,1.3,5,8.2,9.5,2.8]
>>> plt.scatter(X,Y,color='r')
```

- `plt.imshow`: Si les matrices (numpy array) `topg` et `velsurf_mag` décrivent la topographie et les vitesses distribuées en 2D du glacier d'Aletsch, alors le code suivant permet de visualiser les champs spatiaux comme dans la figure 3 (droite).

```
>>> import matplotlib.pyplot as plt   # importe matplotlib
>>> import matplotlib
>>> fig = plt.figure(dpi=200)
>>> ax = fig.add_subplot(1, 1, 1)
>>> ax.axis("off")
>>> ax.set_aspect("equal")
>>> ax.imshow(topg,origin="lower",cmap=matplotlib.cm.terrain)
>>> im = ax.imshow(velsurf_mag, origin="lower", \
              cmap=matplotlib.cm.viridis, vmin=0, vmax=250)
>>> ax.set_title("YEAR : 1910.0" , size=15)
>>> cbar = plt.colorbar(im, label='velsurf_mag')
```

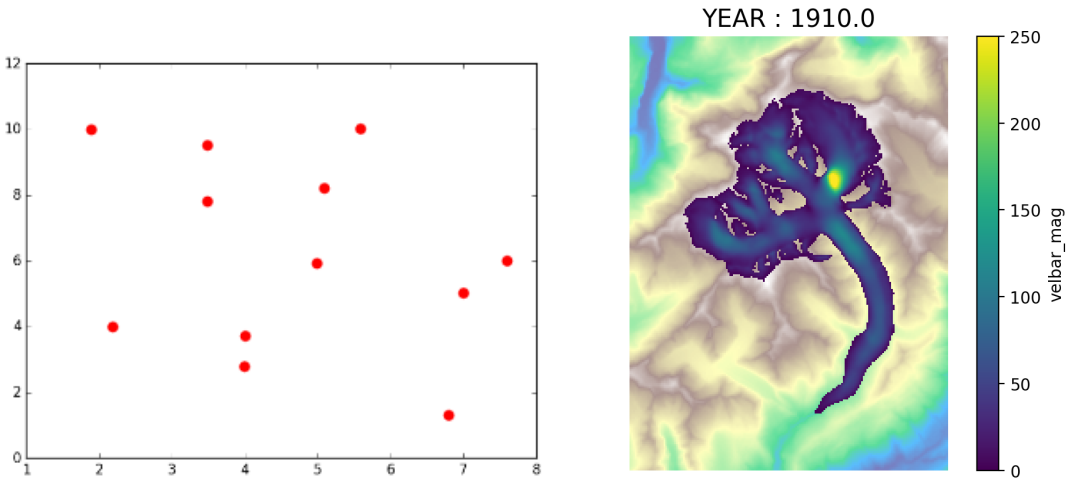

Figure 3: Illustration d'une utilisation de `plt.scatter` (gauche) et `plt.imshow` (droite).

- `plt.hist`: Si `X` est une liste ou un tableau de nombres réels, alors `plt.hist(X, bins=c, density=True)` construit un histogramme des valeurs de `X` ayant `c` classes (par défaut, `c = 10`).

3.2 Visualisation interactive

Très souvent, lorsque nous implémentons un modèle interactif, il est commode de visualiser les résultats de manière dynamique, c'est-à-dire que la solution évolue avec le temps, permettant ainsi une visualisation réaliste du phénomène étudié. Cela nécessite quelques modifications dans le code pour que VS Code puisse afficher le résultat en temps réel. Pour ce faire, il nous faut appeler la bibliothèque IPython, qui permet l'interactivité, en plus de matplotlib :

```
>>> import matplotlib.pyplot as plt
>>> from IPython.display import display, clear_output
```

Ensuite, nous devons créer une figure avec `fig, ax = plt.subplots()` avant de commencer la boucle temporelle, nous initialisons une figure (`fig`) et un ensemble d'axes (`ax`) où les données seront tracées. `fig` représente la fenêtre graphique globale, tandis que `ax` correspond à la zone où les

graphiques et les éléments visuels seront affichés, ce qui nous permet de contrôler l'apparence et le contenu du tracé.

Lors de chaque itération de notre boucle, nous utilisons `clear_output(wait=True)` pour nettoyer la sortie précédente, afin d'éviter la superposition des graphiques. La commande `ax.cla()` permet de nettoyer les axes, c'est-à-dire de supprimer toutes les données et éléments visuels précédents du tracé, en préparant ainsi un affichage propre pour les nouvelles données. Enfin, `display(fig)` affiche la figure mise à jour avec les nouvelles données et configurations de chaque étape de l'itération.

Notons qu'en définissant `fig, ax = plt.subplots()`, il faut ensuite appeler les commandes d'affichage via `ax` (et non `plt` comme c'était le cas auparavant). Bon nombre de fonctions `plt` ont un équivalent avec `ax` en ajoutant `set_` devant, par exemple :

```
>>> ax.plot(z, T, linewidth=2.5)
>>> ax.set_title('Figure')
>>> ax.set_ylabel('Concentration')
>>> ax.set_xlabel('Elevation, m')
>>> ax.set_ylim([0, 500])
```

Par ailleurs, il est également possible de définir une figure avec plusieurs sous-figures en utilisant la commande `fig, (ax1, ax2) = plt.subplots(2, 1)`. Dans ce cas, il faudra appliquer `ax1.cla()` et `ax2.cla()` (pour chaque axe) afin de les nettoyer, puis remplir les axes respectivement avec `ax1.plot(...)` et `ax2.plot(...)`.

En résumé, voici une ébauche de code interactif :

```
[...] # implementation parametres, initialisation, ...

fig, (ax1,ax2) = plt.subplots(2,1)

# Boucle temporelle
for it in range(nt):

    [...] # implementation du modele

    # plotting
    if it%nout == 0:
        clear_output(wait=True)
        ax1.cla()
        ax2.cla()

        # Create subplot 1 for concentration
        ax1.plot(x, C, linewidth=1, c='k')
        ax1.set_title("Temps = " + str(round(time)) + " jours")
        ax1.set_ylim([0, 2000])
        ax1.set_ylabel('Concentration ')

        # Create subplot 2 for flux
        ax2.plot(xv, q, linewidth=1, c='k')
        ax2.set_ylim([-0.01, 0.01])
        ax2.set_ylabel('Flux')
        ax2.set_xlabel('Distance x')

        display(fig)
        plt.pause(0.1)
```

4 References et remerciements

Ce document s'appuit largement sur les cours/tutoriaux suivants:

- *Une introduction à Python 3*, Olivier Gauthé, Laboratoire de Physique Théorique de Toulouse.
- *Liste de commandes PYTHON usuelles pour les TP*, Pascal Maillard, Institut de Mathématiques de Toulouse.

5 Ressources suplémentaires

Voila quelques ressources pour appronfondir vos connaissances en Python:

- https://github.com/jrjohansson/scientific-python-lectures
- https://www.python.org/
- https://zestedesavoir.com/tutoriels/4139/les-bases-de-numpy-et-matplotlib/
- https://github.com/emjako/pythondatascientist
- https://docs.python.org/3/tutorial/