TD n°04 : Les Bibliothèques de Python

3-4 heures

Rédigé par Pascal Delahaye

5 octobre 2015

Le but de ce TD est la découverte des principaux modules que nous serons amenés à utiliser durant l'année.

Il s'agit de :

1. **math** : pour l'utilisation des fonctions méthématiques usuelles (cos, sin, ln, exp, atan, ...)
2. **sympy** : pour le calcul formel (factorisation, développement, simplification, dérivée, intégrales, primitives, equations différentielles, développements limités...)
3. **matplotlib.pylab** : pour les représentations graphiques (nuages de points, fonctions, courbes paramétrées)
4. **numpy** : pour la manipulation des complexes et des tableaux
5. **numpy.linal** : pour effectuer des opérations d'algèbre linéaire (matrices...)
6. **scipy.integrate** : pour les fonctions d'analyse numérique (résolution approchée d'équations différentielles, calcul approché d'intégrales...)

1 La bibliothèque Math

Les fonctions mathématiques sont directement activées dans la console de Spyder.
Cependant, il est indispensable d'utiliser cette bibliothèque lors de la conception de programmes (dans l'éditeur) utilisant des fonctions mathématiques.

Exécutez les instructions suivantes :

```
>>> from math import *     # Importation de la bibliothèque "math"
>>> help('math')           # Vérification des fonctions disponibles dans cette bibliothèque
```

Python

```
Etablir la liste des fonctions importantes de la bibliothèque "math" :
```

2 La bibliothèque Sympy

Il s'agit de la bibliothèque de calcul formel ("sym" comme SYMBOLIC) de Python.

Grâce aux fonctions disponibles dans cette bibliothèque, il nous est possible de :

- Manipuler des expressions algébriques réelles, complexes ou matricielles
- Trouver les solutions exactes d'équations différentielles
- Résoudre de façon exactes des équations, des inéquations et des systèmes
- Calculer des limites, des dérivées, des primitives et des intégrales
- Obtenir des développements limités (DL)

Remarque 1. Toutes les informations sur les applications de la bibliothèque SYMPY se trouvent à l'adresse suivante : www.sympy.org/

2.1 Manipulation d'expressions algébriques

Exécutez les instructions suivantes :

```
>>> from sympy import var      # Fonction servant à déclarer les variables algébriques
                                 utilisées
>>> var('x:z')                 # Pour déclarer les variables x, y et z
>>> var('x,z')                 # Pour ne déclarer que les variables x et z

>>> A = (x+1)*(2*y - z)                 # Déclaration des expressions algébriques
>>> B = 1/(1-3*x**2)
>>> f = lambda x,y: x**2 + 2*x*y + y**2

>>> A + B, A*B, f(x,y)/B

>>> from sympy import expand            # Pour développer une expression algébrique
>>> expand(A)

>>> from sympy import factor            # Pour factoriser une expression algébrique
>>> factor(f(x,y))
>>> factor(f(x,1))

>>> from sympy import simplify          # Pour simplifier une expression
>>> simplify(x+x**2)/(x*(x+1))
```

`Exemple 1.` Développez puis refactoriser l'expression $A = (a+b)(a+2b)(a+3b)$.
Que fait la fonction `simplify` sur les deux expressions obtenues ?

2.2 Résolution d'équations, d'inéquations et de systèmes

Exécutez les instructions suivantes :

```
>>> var('x,y')
>>> A = x**4 - 5*x**2 + 4
>>> B = x**2 + x + 1

>>> from sympy import solve # Résolution d'équations, d'inéquations et de systèmes
>>> solve(A,x)              # Solution de l'équation : A = 0
>>> solve(B,x)              # On constate que solve() donne les solutions complexes
>>> solve(A<0,x)            # Solution de l'équation : A < 0 (écriture à décoder ;-))

>>> B1 = 2*x - 3*y +1
>>> B2 = -x + 2*y - 3
>>> solve([B1,B2],[x,y])    # Solution du système : (B1 = 0, B2 = 0)
```

`Exemple 2.` Résoudre le système : $\begin{cases} x+y-z=2 \\ -2x+y-3z=-1 \\ x-3y=1 \end{cases}$.
Ajouter un paramètre m en facteur de x dans la première équation. Que pensez-vous de la solution proposée par `solve()` ?

2.3 Calcul de limites, de dérivées, de primitives et d'intégrales

Exécutez les instructions suivantes :

```
>>> var('x')
>>> A = x**2 + x + 1              # Déclaration d'une expressions
>>> f = lambda x: -x**2 + 2*x + 3 # Déclaration d'une fonction

>>> from sympy import limit, oo   # Calcul de limites et symbole "infini"
>>> limit(A,x,oo)
>>> limit(f(x),x,-oo)

>>> from sympy import diff
>>> diff(A,x)                     # Calcul des dérivées par rapport à x
>>> diff(f(x),x)
>>> diff(f(x),x,2)                # Calcul d'une dérivée seconde

>>> from sympy import integrate   # Calcul d'une primitive ou d'une intégrale
>>> integrate(f(x),x)             # Calcul d'une primitive par rapport à x
>>> integrate(f(x),(x,0,1))       # Calcul de l'intégrale pour x variant entre 0 et 1
```

`Exemple 3.`

1. Etudier la limite de la fonction f définie par $f(x)=\dfrac{\sin(x)}{x}$ en 1 et en $+\infty$.
   Vous penserez à importer la fonction ***`sin()`*** de `sympy`.
2. Etudier le sens de variation de la fonction f définie par $f(x)=x^3-3x+2$.
3. Déterminer les primitives de la fonction logarithme népérien et de la fonction arctangente.
4. Calculer l'aire du domaine situé entre l'axe des x et la parabole d'équation $y=-x^2-4x+5$.

⚠⚠ Les fonctions de `sympy` ne reconnaissent pas les fonctions mathématiques du module `math`. ⚠⚠
Il est impératif d'importer les propres fonctions mathématiques de `sympy`.

2.4 Résolution exacte d'équations différentielles

Exécutez les instructions suivantes :

```
>>> from sympy import Function          # Pour créer des variables fonctionnelles
>>> y = Function('y')                   # y doit être déclaré comme une fonction

>>> from sympy import dsolve
>>> dsolve(diff(y(x),x,2)+y(x),y(x))    # Résolution de "y" + y = 0" sans condition initiale
```

`Exemple 4.` Résoudre l'équation différentielle $y-\dfrac{x}{2}y'=0$.

2.5 Sommes et produits

Exécutez les instructions suivantes :

```
>>> var('i')                      # Création de la variable i

>>> from sympy import summation, product
>>> summation(i**2,(i,1,10))
>>> product(i**2,(i,1,10))

>>> f = lambda x: x**2
>>> summation(f(x),(x,1,10))
>>> product(f(x),(x,1,10))

>>> product((x+i),(i,0,5))        # Construction de polynômes
```

`Exemple 5.` Contruire à l'aide de la fonction `product` la fraction rationnelle suivante :

$$F(x) = \frac{(x+1)(2x+3)(3x+5)...(10x+19)}{(10-x^2)(9-2x^2)(8-3x^2)...(1-10x^2)}$$

2.6 Obtention de valeurs approchées

Exécutez les instructions suivantes :

```
>>> from sympy import evalf

>>> a = log(2)       # Attention : a n'est pas un "objet sympy"
>>> a.evalf(50)      # ERREUR

>>> from sympy import log
>>> a = log(2)       # Cette fois, a est un "objet sympy"
>>> a.evalf(50)      # Donne une valeur approchée de a à ''10 moins 50'' près

>>> pi               # Attention : ici pi n'est pas un "objet sympy"
>>> pi.evalf(1000)   # ERREUR

>>> from sympy import pi # Cette fois, pi est bien un "objet sympy"
>>> pi.evalf(1000)   # Ca marche !
```

`Exemple 6.` Donner une approximation de e à 10^{-100} près.

2.7 Recherche de développements limités

Le développement limité d'une fonction, au voisinage d'un point à un ordre n, donne la fonction polynomiale de degré inférieur ou égal à n qui approxime le mieux la fonction au voisinage du point considéré.

Exécutez les instructions suivantes :

```
>>> from sympy import series
>>> from sympy.abc import x    # Autre façon d'importer une variable x
>>> series(1/(1-x),n=4)        # Donne le DL de 1/(1-x) au voisinage de 0 à l'ordre 4
>>> series(cos(x),x0=pi/4,n=5) # Ca ne marche pas car ici, "cos" n'est pas un "objet sympy"

>>> from sympy import cos
>>> series(cos(x),x0=pi/4,n=5) # Donne le DL de cos(x) au voisinage de pi/4 à l'ordre 5
```

`Exemple 7.` Donner le développement limité de la fonction définie par $f(x) = \ln(\cos(x))$ au voisinage de 0 à l'ordre 5.

2.8 Graphe d'une fonction

Exécutez les instructions suivantes :

```
>>> from sympy import plot           # Attention : "plot" est désormais la f° "plot" de sympy
>>> plot(x**2,(x,-2,2))              # Trace le graphe de la fonction x -> x^2 sur [-2,2]
>>> plot(x, x**2, x**3, (x,-0.5,1.5))# Trace les graphes des 3 fonctions sur [-0.5,1.5]
```

Remarque 2.

1. On peut utiliser les icônes situées en haut du graphe pour recadrer le graphe.
2. Consulter l'inspecteur d'objets pour découvrir les options disponibles de la fonction `plot()`. Intéressez-vous en particulier aux options `ylim` et `line_color`.

`Exemple 8.` Visualiser graphiquement la solution de l'équation $\tan x = x$.

3 La bibliothèque matplotlib.pylab

Pour connaître les fonctions accessibles depuis *matplotlib.pylab*, taper l'instruction suivante :

```
>>> help('matplotlib.pylab')
```

Remarque 3. On constate que le module `pylab` contient entre autre les fonctions mathématiques usuelles : sin, cos, sin, cosh, sinh, arccos, arctan, arcsin, arctan, log, sqrt, exp. Cependant, ces fonctions sont redéfinies de façon à pouvoir être utilisées par les autres fonctions de cette bibliothèque : elles diffèrent donc par leur utilisation de celles accessibles depuis la bibliothèque *sympy* ou celles définies en *built-in*

Remarque 4. Pour la conception de graphes, `matplotlib` contient aussi aussi le module `pyplot` qui présente néanmoins le désavantage de ne pas inclure les fonctions mathématiques dont nous aurons souvent besoin.

3.1 Pour tracer une séquence de points

Exécutez les instructions suivantes :

```
>>> from matplotlib.pylab import plot          # Importation de la fonction plot
>>> x = [1,2,4,6,8,9]                          # Abscisses des points
>>> y = [-2,6,4,7,2,1]                         # Ordonnées des points
>>> plot(x,y)                                  # Création du graphe
```

Exercice : 1

(∗∗) Tracer l'étoile obtenue en reliant des racines 5ème de l'unité.
Pour cela, vous pourrez utiliser les fonctions `solve`, `var`, `im` et `re` de la bibliothèque `sympy`.

Remarque 5. Tester les fonctions suivantes de matplotlib.pylab qui permettent d'ajuster, de compléter, de sauver ou d'effacer le graphe :

1. `title()`
2. `xlabel()`
3. `ylabel()`
4. `grid()`
5. `axis()`
6. `clf()`
7. `savefig()`

3.2 Pour tracer le graphe d'une fonction

Pour tracer le graphe d'une fonction, on procède de manière analogue :

1. On commence par définir la fonction f
2. Puis, on construit la liste X des abscisses des points à représenter et en tapant $f(X)$ on obtient la liste des ordonnées correspondantes
3. On peut alors utiliser la fonction `plot()` précédente

```
>>> from matplotlib.pylab import arange

>>> f = lambda x: -x**2-4*x+5
>>> x = arange(-6,2,0.01)          # On définit l'échelle sur l'axe Ox
>>> plot(x,f(x))                   # Commande pour créer le graphe de la fonction f
```

Remarque 6. Nous avons vu que la fonction `plot` de `sympy` permettait de tracer un graphe en entrant directement la fonction f à tracer. Il faut cependant veiller à bien importer les fonctions utilisées à partir de la bibliothèque `sympy`.

`Exemple 9.` $(*)$ Représenter la graphe de la fonction $x \to \dfrac{\tan x}{x}$ dans le domaine $D = [-3, 3] \times [-5, 5]$.

3.3 Tracer le graphe de plusieurs fonctions

Pas de difficulté : toutes les courbes construites à l'aide de la fonction `plot()` sont construites sur le même graphique.

Toutes les courbes sont tracées sur le même graphe :

```
>>> plot(x,cos(x))
>>> plot(x,sin(x))
>>> plot(x,tan(x))
```

Remarque 7. La commande `clf()` permet de réinitialiser la fenêtre.

Après ré-initialisation, exécutez :

```
>>> plot(x,cosh(x))
>>> plot(x,sinh(x))
>>> plot(x,tanh(x))
```

`Exemple 10.` Visualiser graphiquement la solution de l'équation $\tan x = x$ sur $[0,\ 2]$.

Exercice : 2

$(*)$ Tracer sur un même graphe les différents types de représentations graphiques obtenus sur $\mathbb{R}^{+*}$ pour les fonctions puissances.

Exercice : 3

$(**)$ La caustique du cercle d'équation polaire $\rho = 6\cos\theta$ est l'enveloppe des droites obtenues par symétrie orthogonale de la droite $(OM(\theta))$ par rapport à la tangente au cercle en $M(\theta)$. On démontre que ces droites ont pour équations cartésiennes :

$$D_\theta \ : \ (x-3) * \cos(\theta) + \sin(\theta) * y + 3 * \cos(\theta/3) = 0$$

1. Tracer sur un même graphe, les droites D_θ obtenues lorsque θ varie de 1 à 60.
2. A quoi ressemble l'enveloppe de ces droites ?

3.4 Tracer une courbe paramétrée

La courbe paramétrée définie par $\begin{cases} x = u(t) \\ y = v(t) \end{cases}$ pour $t \in I$ s'obtient tout simplement en :

1. Construisant une liste de valeurs t
2. Appliquant la fonction `plot()` aux listes $u(t)$ et $v(t)$.

`Exemple 11.`

Ainsi, pour tracer le support de la courbe $\begin{cases} x = 2\cos t \\ y = \sin t \end{cases}$ pour $t \in [-6,\ 6]$, on tapera les instructions suivantes :

```
>>> t = arange(-6,6,0.01)
>>> plot(2*cos(t),sin(t))
```

Exercice : 4

Représentez le support de la courbe paramétrée par $\begin{cases} x(t) = \cos(t)(1-\cos(t)) \\ y(t) = \sin(t)(1-\cos(t)) \end{cases}$ sur $I = [0, 7]$

4 La bibliothèque Numpy

`Numpy` est la bibliothèque de Python qui regroupe l'ensemble des fonctions mathématiques ("num" comme NUMBER) y compris celles relatives à l'algèbre linéaire. Elle permet également de construire rapidement des tableaux.

Pour connaître les fonctions accessibles depuis *numpy*, taper l'instruction suivante :

```
>>> help('numpy')
```

4.1 Manipulation des complexes

Remarque 8. Le complexe i est ici noté $1j$. Ainsi, $3-2i$ sera noté `3-2j`. On notera l'absence du symbole $*$.

```
>>> from numpy import *

>>> a = 3 -2j
>>> absolute(a)    # Donne le module de a
>>> conjugate(a)   # Donne le conjugué de a
>>> exp(a)         # Calcule l'exponentielle complexe de a
```

4.2 Création de tableaux et de matrices

Les tableaux ou les matrices (`array`) sont des listes où tous les éléments sont de même type. Ils sont adaptés au calcul numérique.
Exécutez les instructions suivantes :

```
>>> from numpy import array, matrix, ones, zeros, arange, linspace, diag, concatenate

>>> array([1,2,3])             # Transforme une liste en tableau
>>> matrix([[1,2,3],[4,5,6]])  # Construction d'une matrice
>>> ones(5)              # Crée un tableau contenant 5 uns
>>> ones((2,5))          # Crée une matrice 2x5 contenant des uns
>>> zeros(5)             # Crée un tableau contenant 5 zéros
>>> zeros((2,5))         # Crée une matrice 2x5 contenant 5 zéros
>>> eye(5)               # Crée la matrice 5x5 nulle avec une diagonale de 1 (I5)

>>> arange(-2,2,0.01)    # Crée un tableau de valeurs entre -2 et 2 avec un pas de 0.01
>>> linspace(-2,2,100)   # Crée un tableau de 100 valeurs équiréparties entre -2 et 2

>>> diag([1,2,3,4])               # Création d'une matrice diagonale contenant 1,2,3,4 sur la diag
>>> diag(arange(1,5,1))           # Création d'une matrice diagonale contenant 5 "1" sur la diag
>>> concatenate(([0,0,0,0],[1,1])) # Concaténation de deux tableaux lignes en un seul
```

`Exemple 12.` $(*)$ Construire la matrice diagonale d'éléments diagonaux : $\begin{cases} a_{ii}=1 & \text{si} \quad 1\leq i\leq 5 \\ a_{ii}=-1 & \text{si} \quad 6\leq i\leq 10 \end{cases}$

Exécutez les instructions suivantes :

Copie de tableaux :

```
>>> a = array([1,2,3])
>>> b = copy(a)             # Copie le tableau a dans le tableau b
                              Il s'agit cependant de 2 tableaux différents
>>> c = a                   # Attention, a et c représente le même tableau : la
                              modification de l'un entraîne la modification de l'autre
>>> b
>>> a[0] = 100              # On modifie la première valeur du tableau a
>>> b[0]                    # b reste inchangé
>>> c[0]                    # tandis que la première valeur de c est elle-aussi modifiée
```

4.3 Résolution numérique de systèmes linéaires

La bibliothèque *numpy* propose des fonctions de résolution numérique de systèmes linéaires. Comme il ne s'agit pas de calcul formel, les solutions sont données sous la forme de valeurs approchées.

Pour résoudre un système linéaire de la forme $AX = B$, on procède ainsi :

1. On crée les matrices A et B du système avec la commande `array()`
2. On résout le système avec la commande `solve()`

Remarque 9. L'algorithme de résolution utilisé par cette fonction `solve()` est basé sur la décomposition de la matrice A sous la forme LU, produit d'une matrice triangulaire inférieure L *(comme lower, inférieure)* et une matrice triangulaire supérieure U *(comme upper, supérieure)*.

Pour résoudre le système $\begin{cases} x+2y+3z=0 \\ -x+3y+2z=1 \\ y+5z=-1 \end{cases}$, il suffit de taper les instructions suivantes :

```
>>> from numpy import array
>>> from numpy.linalg import solve
>>> A = array([[1,2,3],[-1,3,2],[0,1,5]])        # Création des matrices du système
>>> B = array([0,1,-1])
>>> solve(A,B)                                   # Résolution du système
```

`Exemple 13.` $(*)$ Comparer les solutions du système précédent lorsqu'elles sont données par numpy avec celles données par sympy.

4.4 Opérations sur les tableaux et matrices

Exécutez les instructions suivantes :

```
>>> t = arange(1,10,1)
>>> t[0]                    # Permet de lire la 1ère valeur du tableau
>>> t[0] = 0                # Permet de modifier la 1ère valeur du tableau
>>> 2*t                     # On peut effectuer des opérations élémentaires sur les éléments
>>> t > 4                   # Un exemple d'opération booléenne sur un tableau

>>> f = lambda x: x**3
>>> f(t)                    # On peut appliquer une fonction à tous les éléments du tableau

>>> A = matrix([[1,2,3],[4,5,6],[7,8,9]])
>>> A[0,0], A[1,0]          # Pour obtenir les coefficients de la matrice
>>> A**2                    # On peut effectuer les opérations usuelles
>>> f(A)                    # On peut aussi appliquer une fonction aux coefficients
```

Remarque 10.

1. Parmi les autres fonctions disponibles, on trouve `max()`, `min()`, `prod()`, `sum()`, `mean()`
2. Il est aussi possible d'extraire des lignes et des colonnes d'un tableau

`Exemple 14.` $(*)$ Construire la matrice suivante en utilisant les fonctions précédentes :

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\ 1 & 4 & 9 & 16 & \dots & & & & \dots & 100 \\ 1 & 8 & 64 & \dots & & \dots & & & \dots & 1000 \end{pmatrix}$$

4.5 La bibliothèque numpy.linalg

Nous aurons parfois besoin de fonctions plus avancées qui se trouvent dans la sous-bibliothèque `linalg` :

1. `det()` pour calculer le déterminant d'une matrice carrée

2. `eigvals()` pour déterminer les valeurs propres d'une matrice carrée
3. `eig()` pour déterminer les valeurs propres et vecteurs propres d'une matrice carrée
4. `norm()` pour calculer la norme d'une matrice
5. `solve()` pour résoudre un système de la forme AX=B

Pour connaître les fonctions accessibles depuis *numpy.linalg*, taper l'instruction suivante :

```
>>> help('numpy.linalg')
```

Exécutez les instructions suivantes :

```
>>> from numpy.linalg import det, eigvals, norm, solve, matrix
>>> A = matrix([[1,2,3],[4,5,6],[7,8,9]])
>>> B = matrix([[1],[2],[3]])
>>> det(A)
>>> norm(A)
>>> eigvals(A)
>>> solve(A,B)
```

5 La bibliothèque Scipy.integrate

`Scipy` est la bibliothèque de Python qui rassemble les fonctions de calcul numérique ("sci" comme SCIENTIFIC). Elle contient la sous-bibliothèque `integrate` qui permet :

1. le calcul approché d'intégrales (fonctions `quad()` et `trapz()`)
2. la résolution approchée d'équations différentielles (fonction `odeint()`)

```
>>> help('scipy')
```

5.1 Calcul approché d'une intégrale

Lorsque la fonction `integrate()` vue dans la bibliothèque *sympy*, ne permet pas de calculer une intégrale, on fait appel à des méthodes numériques implémentées dans certaines fonction (`quad()`, `trapz()`) de la bibliothèque *scipy*. Exécutez les instructions suivantes :

```
*** Intégration d'une fonction ***

>>> from scipy.integrate import quad    # Importation de la fonction quad
>>> quad(lambda x: x**2,0,1)

*** Cas où la fonction est donnée par un tableau de points ***

>>> from scipy.integrate import trapz   # Importation de la fonction trapz
>>> from numpy import linspace          # Importation de la fonction linespace
>>> x = linspace(0,1,100)
>>> y = x**2
>>> trapz(y,x)                          # Integrale de x -> x^2 entre 0 et 1
```

`Exemple 15.` En utilisant les deux fonctions précédentes, donner une valeur approchée de l'intégrale : $\int_0^{10} e^{-t^2}\,\mathrm{d}t$

5.2 Résolution numérique d'une équation différentielle de la forme Y' = f(Y,t)

Nous avons vu dans la bibliothèque *sympy* la fonction `desolve()` qui permet de résoudre de façon exacte des équations différentielles. Cependant, cette fonction ne fait pas de miracle et est incapable de résoudre certaines équations différentielles. On fait alors appel à des *méthodes numériques* implémentées dans la fonction `odeint()` de *scipy*.

```
>>> from scipy.integrate import odeint  # Importation de la fonction odeint
>>> from matplotlib.pyplot import plot  # Importation de la fonction plot
>>> from numpy import linspace          # Importation de la fonction linespace

*** Cas d'une Equation d'ordre 1 ***

>>> f = lambda y,t: y                   # Equation diffdérentielle y' = y
>>> t = linspace(0,1,100)
>>> y = odeint(f,1,t)                   # Conditions Initiales : y(0)=1
>>> plot(t,y)
```

```
*** Cas d'une équation d'ordre 2 ***

>>> f = lambda y,t: (y[1], -3*y[0] - y[1])  # Equation différentielle y'' + y' + 3y = 0
                                              où Y = (y,y')
>>> t = linspace(0,10,100)                  # Intervalle de résolution : [0,10]
>>> y = odeint(f,(1,0),t)                   # Conditions Initiales : y(0)=1 et y'(0)=0
>>> plot(t,y)
```

`Exemple 16.` Représenter une approximation de la courbe intégrale de la solution de l'équation différentielle :

1. $y'' + y = \cos(x)$ vérifiant les conditions initiales $f(0) = 1$ et $f'(0) = 0$.
2. $y'' + \sin(x)y' + y = x$ vérifiant les conditions initiales $f(0) = 0$ et $f'(0) = 1$.

Exercice : 5

($*$) Résoudre numériquement le système différentiel $\begin{cases} x' = -y - x(x^2+y^2) \\ y' = x - y(x^2+y^2) \end{cases}$ sur $[0,\ 4]$.
On prendra $x(0) = 1$ et $y(0) = 0$ pour conditions initiales.

Exercice : 6

($*$) Résoudre numériquement l'équation différentielle $y' = x^3 - y^3$ sur $[-1,\ 4]$ en testant différentes conditions initiales..