Logiciels mathématiques

Émilie Charlier

2022-2023

(Notes provisoires)

Introduction

Le cours de logiciels mathématiques est au programme du bloc 1 du bachelier en sciences mathématiques de l'Université de Liège. Sur la page web du Département de Mathématique, on peut trouver ces mots : "Le nouveau cours de logiciels mathématiques a pour but de familiariser les étudiants à l'informatique, outil omniprésent en sciences, en entreprises ou encore pour l'enseignement."

Il existe une multitude de logiciels mathématiques, numériques ou symboliques, commerciaux ou libres. Dans ce cours, nous commencerons par présenter les logiciels de calcul symbolique Mathematica et Sympy. Le premier est un logiciel commercial, pour lequel l'Université de Liège a une licence accessible à tous ses employés ainsi qu'à tous ses étudiants. Le second est une librairie Python de calcul formel, utilisée, notamment, par le logiciel libre SageMath.

La deuxième partie du cours sera consacrée à une introduction à la programmation, en utilisant Mathematica et le langage Python.

Dans la troisième et dernière partie du cours, nous présenterons deux logiciels mathématiques supplémentaires : GeoGebra et Calc. Le logiciel gratuit GeoGebra est un calculateur graphique de fonctions. Il est particulièrement utile pour l'enseignement et l'étude de la géométrie. Nous poursuivrons par une présentation des fonctionnalités de Libre Office Calc pour le calcul et analyse de données.

Cours 1 : Introduction et calculatrice avec Mathematica

Raccourcis clavier

Dans tout logiciel, il est important de connaître quelques raccourcis clavier, qui permettront une utilisation beaucoup plus fluide en évitant de nombreux “clics”. Il y a des raccourcis communs à pratiquement tous les logiciels, comme Ctrl-x, Ctrl-c, Ctrl-v, Ctrl-s, Ctrl-z, mais chaque logiciel possède aussi une liste de raccourcis spécifiques qu’il est utile de parcourir et de petit à petit apprendre à utiliser. Dans Mathematica, vous trouverez cette liste au point “tutorial/KeyboardShor - tcutListing” de l’aide.

Documentation de Mathematica

On accède à la documentation de Mathematica via l’onglet “Help”. Un moteur de recherche per - met d’utiliser des mots-clés. Lorsque l’on connaît déjà le nom Mathematica de l’objet recherché, on peut également placer le signe ? juste avant le mot recherché dans une cellule standard. L’évalua - tion de cette cellule donnera des informations brève sur cet objet. Si l’on souhaite plus d’informa - tion, on peut ensuite demander l’accès à la documentation complète en cliquant sur >>. L'aide de Mathematica est un outil à utiliser sans modération. C'est grâce à l'aide que vous apprendrez à utiliser Mathematica de façon indépendante et interactive.

Types de cellules

Un fichier de travail de Mathematica s'appelle un notebook, et est découpé en cellules, que l'on créera au fur et à mesure. Chacune de ces cellules peut être visualisée par un crochet sur la droite du notebook. Une cellule peut être soit une cellule de texte (éditable, mais qu'on ne peut pas évaluer) soit une cellule standard (éditable et évaluable). Une cellule évaluable est évaluée avec la commande clavier "shift+enter". La commande "enter" seule permet, comme dans un traitement de texte, de passer à la ligne à l'intérieur d'une cellule. Lorsqu'une cellule standard "input" est évaluée, Mathematica crée une nouvelle cellule "output" qui contient la sortie demandée. On peut passer à une cellule de texte avec la commande "alt+shift+7" et à une cellule standard avec la commande "alt+shift+9".

Opérations de base

Les opérations de base (addition, soustraction, multiplication, division et exponentiation) se font avec les opérateurs +, -, * ou “espace”, / et ^.

In[ ]:=
```
1 + 2
4 - 7
```
Out[ ]= 3

Out[ ]= -3

In[ ]:=
```
2 * 3
2 × 3
23
10 / 2
2 ^ 3
```
Out[ ]= 6

Out[ ]= 6

Out[ ]= 23

Out[ ]= 5

Out[ ]= 8

La racine carrée s'obtient en utilisant la fonction Sqrt (pour square root en anglais). Notez que la fonction Sqrt prend son argument entre crochets. C'est en fait le cas de toutes les fonctions Mathematica.

In[ ]:=
```
Sqrt[8]
```
Out[ ]= $2\sqrt{2}$

Notez déjà que Mathematica donne la valeur exacte de la racine de 8 (et non une valeur numérique approchée).

La racine n-ième se calcule en utilisant une puissance fractionnaire. Notez l'utilisation des parenthèses pour la priorité des opérations.

In[ ]:=
```
8 ^ (1 / 3)
```
Out[ ]= 2

Priorité des opérations

Mathematica utilise les parenthèses pour la priorité (usuelle) des opérations qu'il effectue.

```
In[ ]:= 2 + 3 × 4
Out[ ]= 14
```

```
In[ ]:= (2 + 3) 4
        2 + (3 × 4)
Out[ ]= 20
Out[ ]= 14
```

```
In[ ]:= 2 + 3 / 4
        (2 + 3) / 4
```

Out[ ]= $\frac{11}{4}$

Out[ ]= $\frac{5}{4}$

La division par 0 retourne "ComplexInfinity" :

```
In[ ]:= 2 / 0
```

Power : Infinite expression $\frac{1}{0}$ encountered .

```
Out[ ]= ComplexInfinity
```

Profitons-en pour aller voir ce que dit l'aide de Mathematica à propos de cette drôle de valeur. On accède à l’aide en plaçant le signe ? juste avant le mot recherché (dans la syntaxe de Mathematica).

```
In[ ]:= ? ComplexInfinity
```

Out[ ]=

Symbol

ComplexInfinity represents a quantity with
infinite magnitude , but undetermined complex phase .

Pour en savoir plus, on clique sur >>. Vous serez redirigés vers une nouvelle fenêtre interactive, avec quantité d’informations et de liens ad hoc.

L’infini

Une constante de Mathematica a une valeur particulière :

In[ ]:=
```
Infinity
```
Out[ ]=
```
∞
```

On peut réaliser des opérations avec Infinity.

In[ ]:=
```
Infinity + Infinity
3 Infinity
Infinity * Infinity
```
Out[ ]=
```
∞
```
Out[ ]=
```
∞
```
Out[ ]=
```
∞
```

Remarquons que Mathematica signale une erreur à l'évaluation de Infinity - Infinity.

In[ ]:=
```
Infinity - Infinity
```
```
Infinity : Indeterminate expression -∞ + ∞ encountered .
```
Out[ ]=
```
Indeterminate
```

Enfin, allons voir dans la documentation :

In[ ]:=
```
? Infinity
```
Out[ ]=

Symbol

Infinity or ∞ is a symbol that represents a positive infinite quantity .

Un des exemples proposés dans l'aide est le suivant. C’est le calcul d’une série. Nous y reviendrons plus tard, dans le paragraphe "Algèbre linéaire, analyse et graphiques avec Mathematica".

In[ ]:=
```
Sum[1 / n ^ 2, {n, Infinity}]
```
Out[ ]=

$$\frac{\pi^2}{6}$$

On y trouve également que

In[ ]:=
```
1 / Infinity
```
Out[ ]=
```
0
```

Division euclidienne

Le quotient et le reste de la division d'un nombre entier par un autre se calcule grâce aux fonctions Quotient et Mod :

In[ ]:=
```
48 / 5
```

Out[ ]=
$$\frac{48}{5}$$

In[ ]:=
```
Quotient[48, 5]
Mod[48, 5]
```

Out[ ]=
```
9
```

Out[ ]=
```
3
```

Vous connaissez l'algorithme d'Euclide pour calculer le pgcd de deux nombres entiers non nuls. Vous pouvez également utiliser Mathematica avec la fonction GCD. Cette fonction calcule le pgcd de n nombres entiers, où n est arbitraire.

In[ ]:=
```
GCD[3, 7]
GCD[1614, 99 732]
GCD[1614, 99 732, 18]
```

Out[ ]=
```
1
```

Out[ ]=
```
6
```

Out[ ]=
```
6
```

Factorisation d'un nombre entier

La fonction FactorInteger donne la décomposition en facteurs premiers d'un nombre entier.

In[ ]:=
```
FactorInteger[11 760]
```

Out[ ]=
```
{{2, 4}, {3, 1}, {5, 1}, {7, 2}}
```

En effet, nous avons :

In[ ]:=
```
2^4 × 3^1 × 5^1 × 7^2
```

Out[ ]=
```
11 760
```

Fonctions et constantes de base

Vous aurez peut-être déjà remarqué que Mathematica nomme ses fonctions en anglais, avec des majuscules pour commencer (mais pas que !). Voici une liste de quelques fonctions et constantes usuelles. Vous en trouverez des centaines d'autres dans l'aide.

La partie entière d'un nombre se calcule grâce à IntegerPart et la partie fractionnaire se calcule avec FractionalPart :

In[ ]:=
```
IntegerPart[7 / 3]
FractionalPart[7 / 3]
```

Out[ ]= 2

Out[ ]= $\frac{1}{3}$

Parties entières par défaut et par excès :

In[ ]:=
```
Floor[7 / 3]
Ceiling[7 / 3]
```

Out[ ]= 2

Out[ ]= 3

Les fonctions IntegerPart et Floor diffèrent sur les nombres négatifs. En fait, IntegerPart coïncide avec Floor sur [0,+∞[et avec Ceiling sur]-∞,0].

In[ ]:=
```
IntegerPart[-7 / 3]
Floor[-7 / 3]
Ceiling[-7 / 3]
```

Out[ ]= -2

Out[ ]= -3

Out[ ]= -2

Valeur absolue et norme :

In[ ]:=
```
Abs; Norm;
```

Signe d'un nombre :

In[ ]:=
```
Sign;
```

Fonctions trigonométriques :

```
In[ ]:= Cos; Sin; Tan; Cot; ArcCos; ArcSin; ArcTan; ArcCot;
```

Leurs versions hyperboliques :

```
In[ ]:= Cosh; Sinh; Tanh; Coth; ArcCosh; ArcSinh; ArcTanh; ArcCoth;
```

On remarquera qu'une commande suivie du signe ; ne voit pas sa sortie affichée par Mathematica (mais l'évaluation a bien lieu, elle).

La factorielle d'un nombre s'obtient de deux façons :

```
In[ ]:= Factorial[7]
Out[ ]= 5040
```

et

```
In[ ]:= 7!
Out[ ]= 5040
```

Fonctions logarithmes et exponentielles :

```
In[ ]:= Log[8]
        Log[2, 8]
        Exp[4]
        Exp[1]
Out[ ]= Log[8]
Out[ ]= 3
Out[ ]= e^4
Out[ ]= e
```

Trois constantes célèbres :

```
Pi
Out[ ]= π
```

```
E
Out[ ]= e
```

```
I
```

Out[ ]= i

Calcul exact et valeur numérique

Comme nous l'avons déjà remarqué, Mathematica rend des valeurs exactes. Ainsi, si on ne lui en fait pas la demande expresse, il ne donne pas d'approximation d'un nombre rationnel : 2/3 n'est pas égal à 0.66666 (car il faudrait une infinité de 6, ce qu'aucun ordinateur n'est capable de stocker). Il en va de même pour Pi, qui n'est pas égal à 3.141592.

In[ ]:=
```
2 / 3
Pi
```

Out[ ]= $\frac{2}{3}$

Out[ ]= π

C'est la fonction N qui permet de donner une approximation numérique, et ce, avec la précision souhaitée.

In[ ]:=
```
N[2 / 3]
N[Pi]
N[2 / 3, 100]
N[Pi, 100]
```

Out[ ]=
```
0.666667
```

Out[ ]=
```
3.14159
```

Out[ ]=
```
0.66666666666666666666666666666666666666666666666666666666666666666666666666⋱
 6666666666666666666666667
```

Out[ ]=
```
3.14159265358979323846264338327950288419716939937510582097494459230781640628⋱
 6208998628034825342117068
```

Mathematica travaille en valeurs approchées dès qu'on entre un nombre avec virgule (on utilise le point).

In[ ]:=
```
2. / 3
3.14 + 2 / 3
```

Out[ ]=
```
0.666667
```

Out[ ]=
```
3.80667
```

En particulier, Mathematica perd de l'information au passage. Attention aux approximations !

In[ ]:=
```
N[2. / 3]
N[2. / 3, 20]
N[2 / 3, 20]
```

Out[ ]=
```
0.666667
```

Out[ ]=
```
0.666667
```

Out[ ]=
```
0.66666666666666666667
```

Test d'égalité

Le signe == permet de tester l'égalité de deux objets. Mathematica rend True ou False, selon que le test soit positif ou négatif.

In[ ]:=
```
2 / 3 == Pi
```

Out[ ]=
```
False
```

Insistons : la prudence est de mise lorsqu'on utilise l'évaluation numérique. En effet, voici les rendus de Mathematica aux tests suivants :

In[ ]:=
```
2 / 3 == N[2 / 3]
2 / 3 == N[2. / 3]
```

Out[ ]=
```
True
```

Out[ ]=
```
True
```

In[ ]:=
```
Pi == N[Pi]
Sin[N[Pi]] == Sin[Pi]
```

Out[ ]=
```
True
```

Out[ ]=
```
False
```

Évaluons la fonction Sin en Pi et en N[pi].

In[ ]:=
```
Sin[Pi]
Sin[N[Pi]]
```

Out[ ]=
```
0
```

Out[ ]=
```
1.22465 × 10^-16
```

Mathematica évalue les constantes (qu'il connaît) à la précision souhaitée.

In[ ]:=
```
N[Pi, 50]
```

Out[ ]=
```
3.1415926535897932384626433832795028841971693993751
```

La fonction N peut aussi donner une valeur numérique des grands nombres.

In[ ]:=
```
100!
```

Out[ ]=
```
93 326 215 443 944 152 681 699 238 856 266 700 490 715 968 264 381 621 468 592 963 895 217 ⋱
  599 993 229 915 608 941 463 976 156 518 286 253 697 920 827 223 758 251 185 210 916 864 ⋱
  000 000 000 000 000 000 000 000
```

In[ ]:=
```
N[100!]
```

Out[ ]=
```
9.33262 × 10^157
```

Voici un autre exemple qui montre qu'il faut être prudent :

In[ ]:=
```
2.00006 - 2.00005 == 0.00001
```

Out[ ]=
```
False
```

Vous trouverez plus de détails à ce propos dans la documentation (cliquez sur >>).

In[ ]:=
```
? ==
```

Out[ ]=

Symbol

lhs == *rhs* returns True if *lhs* and *rhs* are identical .

Comparaisons

On peut également faire des tests de comparaison avec les signes ≤, ≥, <, >, ≠ (on tape <=, >=, <, >, !=) :

In[ ]:=
```
2 ≤ 3
```
Out[ ]=
```
True
```
In[ ]:=
```
2 ≥ 3
```
Out[ ]=
```
False
```
In[ ]:=
```
2 < 3
```
Out[ ]=
```
True
```
In[ ]:=
```
2 > 3
```
Out[ ]=
```
False
```
In[ ]:=
```
2 ≠ 3
```
Out[ ]=
```
True
```

Nombres complexes

Mathematica connaît les nombres complexes :

In[ ]:=
```
I ^ 2
Re[2 + 3 I]
Im[2 + 3 I]
Abs[2 + 3 I]
Conjugate[2 + 3 I]
```
Out[ ]=
```
-1
```
Out[ ]=
```
2
```
Out[ ]=
```
3
```
Out[ ]= $\sqrt{13}$

Out[ ]= $2 - 3 i$

On peut faire des calculs avec les nombres complexes: addition, soustraction, multiplication, division et exponentiation.

In[ ]:=
```
(2 + 3 I) + (4 + 7 I)
(2 + 3 I) (4 + 7 I)
(2 + 3 I) / (4 + 7 I)
(2 + 3 I) 3
```

Out[ ]= $6 + 10\,i$

Out[ ]= $-13 + 26\,i$

Out[ ]= $\frac{29}{65} - \frac{2\,i}{65}$

Out[ ]= $6 + 9\,i$

Mathematica ne donne pas nécessairement la forme canonique. On peut le forcer à le faire, avec ComplexExpand.

In[ ]:=
```
(a + I b) ^ 5
ComplexExpand [(a + I b) ^ 5]
```

Out[ ]= $(a + i\,b)^5$

Out[ ]= $a^5 - 10\,a^3\,b^2 + 5\,a\,b^4 + i\,(5\,a^4\,b - 10\,a^2\,b^3 + b^5)$

On peut aussi faire des approximations numériques dans C vu comme RxR.

In[ ]:=
```
N[2 / 3 + 4 / 7 I]
```

Out[ ]= $0.666667 + 0.571429\,i$

On peut aussi travailler avec la forme trigonométrique des nombres complexes. Nous avons déjà vu que Abs rend le module d’un nombre complexe.

In[ ]:=
```
Abs[2 + 3 I]
Arg[2 + 3 I]
AbsArg[2 + 3 I]
N[AbsArg[2 + 3 I]]
```

Out[ ]= $\sqrt{13}$

Out[ ]= $\text{ArcTan}\left[\frac{3}{2}\right]$

Out[ ]= $\left\{\sqrt{13}, \text{ArcTan}\left[\frac{3}{2}\right]\right\}$

Out[ ]= {3.60555 , 0.982794}

On peut aussi passer des coordonnées cartésiennes aux coordonnées polaires, et inversement.

In[ ]:=
```
ToPolarCoordinates [{x, y}]
FromPolarCoordinates [{r, θ}]
```

Out[ ]= $\left\{\sqrt{x^2 + y^2}, \text{ArcTan}[x, y]\right\}$

Out[ ]= $\{r \text{Cos}[\theta], r \text{Sin}[\theta]\}$

Remarquez que les fonctions ToPolarCoordinates et FromPolarCoordinates prennent comme argument une liste de coordonnées. Nous reviendrons plus spécifiquement aux listes un peu plus loin. Ainsi

In[ ]:=
```
ToPolarCoordinates [{2, 3}]
```

Out[ ]= $\left\{\sqrt{13}, \text{ArcTan}\left[\frac{3}{2}\right]\right\}$

In[ ]:=
```
FromPolarCoordinates [{2, Pi / 6}]
```

Out[ ]= $\left\{\sqrt{3}, 1\right\}$

Attention que I et i sont traités différemment par Mathematica :

In[ ]:=
```
i ^ 2
```

Out[ ]= i^2

In[ ]:=
```
(2 + 3 I) (4 + 7 I)
(2 + 3 i) (4 + 7 i)
```

Out[ ]=
```
-13 + 26 i
```

Out[ ]=
```
(2 + 3 i) (4 + 7 i)
```

Cours 2 : Calcul symbolique avec *Mathematica,* listes et algèbre linéaire

Définir des variables

En mathématique, on utilise des symboles pour représenter des objets appartenant à une classe d’objets. Par exemple, quand on demande de résoudre l’équation x+3=0 dans R, il est entendu que le symbole x représente un réel. Il s’agit donc de trouver pour quel réel x l’équation est vérifiée. On ne peut pas dire que x “varie” dans R. En guise d’autre exemple, lorsqu’on demande de représenter la fonction sin : R -> R, on va représenter dans le plan euclidien l’ensemble des points de coordonnées (x,sin(x)), avec x dans R. On dira parfois que x “varie” dans R. En réalité, pour chaque x dans R, on considère le point de coordonnées (x,sin(x)). En informatique, la notion de variable diffère quelque peu du sens mathématique : une variable est un symbole qui prend une valeur. Le plus souvent, on travaille avec des variables dites dynamiques, c’est-à-dire dont la valeur change au cours du temps (lors de l’exécution d’un programme par exemple).

Dans Mathematica, on ne doit pas importer les variables. Si un symbole n’est pas réservé (comme c’est le cas pour I et E par exemple), il pourra être utilisé pour définir une nouvelle variable symbolique. En fait, n’importe quelle suite de symbole non réservée et ne commençant pas par un chiffre pourra être utilisée pour définir une nouvelle variable symbolique.

Considérons par exemple les symboles x et y. Remarquons qu’ils apparaissent d’abord en bleu. La couleur bleue nous indique qu’un symbole n’a pas été défini et ne contient pas de valeur.

```
In[ ]:= x
        y
Out[ ]= x
Out[ ]= y
```

On peut réaliser des calculs avec des variables symboliques, qui sont traitées comme des indéterminées d'un polynôme.

In[ ]:=
```
x ^ 2 + 3
y + 1
(x ^ 2 + x + 1) + (4 x ^ 2 + 5)
```

Out[ ]= $3 + x^2$

Out[ ]= 1 + y

Out[ ]= $6 + x + 5 x^2$

Affectation de variable

L'affectation de la valeur 4 dans la variable symbolique x se fait en utilisant le signe =. Remarquez la différence avec le test d’égalité qui se fait avec le double signe ==.

In[ ]:=
```
x = 4
```

Out[ ]= 4

In[ ]:=
```
x == 4
y == 4
```

Out[ ]= True

Out[ ]= y == 4

Remarquons que le premier test rend True car x contient la valeur 4, et le deuxième test ne rend ni True ni False car y est une variable symbolique non affectée.

Les calculs contenant x seront affectés également, ceux contenant y continuant d'être considérés comme des polynômes.

In[ ]:=
```
x ^ 2 + 1
y + 1
```

Out[ ]= 17

Out[ ]= 1 + y

On peut changer l’affectation de x.

In[ ]:=
```
x = 3
```

Out[ ]= 3

Les calculs contenant x tiennent compte de la dernière affectation de x.

In[ ]:=
```
x^2 + 1
```
Out[ ]=
```
10
```

Clear

Pour effacer la valeur d’une variable symbolique, on utilise la fonction Clear. Notez que, ce faisant, la couleur bleue réapparaît, et ce partout dans le notebook ouvert.

In[ ]:=
```
Clear[x]
```

On vérifie que x n'a plus de valeur numérique :

In[ ]:=
```
x
```
Out[ ]=
```
x
```

Si on réévalue une ligne précédente contenant x, Mathematica refera des calculs de polynômes.

Affectation et symbole protégé

On ne peut pas affecter de valeur à un symbole protégé :

In[ ]:=
```
I = 3
Pi = 1
```

··· Set : Symbol i is Protected .

Out[ ]=
```
3
```

··· Set : Symbol π is Protected .

Out[ ]=
```
1
```

Substitution d’un symbole dans une expression

On peut substituer une variable symbolique par une autre dans une expression au moyen de “/.”. On peut aussi se servir de cette commande pour affecter une valeur à une variable symbolique dans une expression. L’affectation est locale et non globale : la variable symbolique n’a pas été affectée de cette valeur en dehors de l’expression. Cela correspond donc à évaluer une fonction de cette variable symbolique en une valeur spécifique.

In[ ]:=
```
x^2 + 3 x y + 3 y^2
x^2 + 3 x y + 3 y^2 /. x → a
x^2 + 3 x y + 3 y^2 /. x → a^2 + 1
x^2 + 3 x y + 3 y^2 /. x → 2
x
```

Out[ ]= $x^2 + 3\,x\,y + 3\,y^2$

Out[ ]= $a^2 + 3\,a\,y + 3\,y^2$

Out[ ]= $(1 + a^2)^2 + 3\,(1 + a^2)\,y + 3\,y^2$

Out[ ]= $4 + 6\,y + 3\,y^2$

Out[ ]= x

In[ ]:=
```
p = x^2 + 3 x y + 3 y^2
q = x^2 + 3 x y + 3 y^2 /. x → a
```

Out[ ]= $x^2 + 3\,x\,y + 3\,y^2$

Out[ ]= $a^2 + 3\,a\,y + 3\,y^2$

In[ ]:=
```
p /. x → a
q == p /. x → a
p
```

Out[ ]= $a^2 + 3\,a\,y + 3\,y^2$

Out[ ]= True

Out[ ]= $x^2 + 3\,x\,y + 3\,y^2$

In[ ]:=
```
q /. x → b
q /. a → b
q
```

Out[ ]= $a^2 + 3\,a\,y + 3\,y^2$

Out[ ]= $b^2 + 3\,b\,y + 3\,y^2$

Out[ ]= $a^2 + 3\,a\,y + 3\,y^2$

On peut aussi affecter à q la dernière valeur de q lorsque a est substitué par b.

In[ ]:=
```
q = q /. a → b
```

Out[ ]=
$b^2 + 3\, b\, y + 3\, y^2$

On vérifie que q a bien changé de valeur (puisqu'on l'a affecté).

In[ ]:=
```
q
```

Out[ ]=
$b^2 + 3\, b\, y + 3\, y^2$

On peut faire une substitution dans une fonction Mathematica.

In[ ]:=
```
f = Sin[x]
f /. x → a
f /. x → Pi
```

Out[ ]=
```
Sin[x]
```

Out[ ]=
```
Sin[a]
```

Out[ ]=
```
0
```

Attention qu' ici, f est un symbole qui a été affecté d'une expression symbolique contenant le symbole x. En aucun cas, f ne peut etre considéré comme une fonction de x ! Voici ce que Mathematica retourne lorsqu'on essaie d'évaluer f.

In[ ]:=
```
f[2]
f[a]
```

Out[ ]=
```
Sin[x][2]
```

Out[ ]=
```
Sin[x][a]
```

On peut également utiliser := pour substituer une variable symbolique dans une autre.

In[ ]:=
```
y := x + 2
```

Contrairement à l'affectation y=x+2, lorsque qu'on entre y:=x+2, Mathematica ne retourne rien. Mais il a enregistré que

In[ ]:=
```
y
```

Out[ ]=
```
2 + x
```

In[ ]:=
```
z = x + 2
z
y == z
```

Out[ ]=
```
2 + x
```

Out[ ]=
```
2 + x
```

Out[ ]=
```
True
```

In[ ]:=
```
x = 2
y
z
```

Out[ ]=
```
2
```

Out[ ]=
```
4
```

Out[ ]=
```
4
```

Substitution versus affectation

Au vu de ces premiers exemples, on peut, à première vue, avoir l'impression qu'on peut utiliser = et := de façon équivalente afin de substituer une variable symbolique dans une autre. Il n'en est rien. La prudence est de mise car les comportement de = (affectation) et de := (substitution) diffèrent dès qu'on affecte des valeurs aux symboles. Comparez les exemples suivants.

In[ ]:=
```
Clear[x, y]
x = 2; y = x + 2
x = 3; y
```

Out[ ]=
```
4
```

Out[ ]=
```
4
```

Dans ce premier exemple, la valeur de y est la même au début et à la fin de l'exécution. En effet, on a affecté la valeur x+2 à y lorsque x valait 2. Lorsque x change de valeur, y conserve la valeur 4.

In[ ]:=
```
Clear[x, y]
x = 2; y := x + 2;
y
x
x = 3
y
```

Out[ ]= 4

Out[ ]= 2

Out[ ]= 3

Out[ ]= 5

Dans ce deuxième exemple, y vaut 4 au début et 5 à la fin. En effet, y est un symbole qui dépend du symbole x. Si x change de valeur, y aussi.

Enfin, considérons ce troisième exemple :

In[ ]:=
```
Clear[x, y]
y = x + 2; y
x = 2; y
x = 3; y
x
```

Out[ ]= 2 + x

Out[ ]= 4

Out[ ]= 5

Out[ ]= 3

Ici on définit la variable symbolique y comme étant la variable symbolique x+2. Au moment de cette définition, aucune valeur n'a encore été affectée à x, et Mathematica considère donc x comme une variable symbolique.

In[ ]:=
```
Clear[x]
Head[x]
x = 2;
Head[x]
```

Out[ ]=
```
Symbol
```

Out[ ]=
```
Integer
```

Attention donc à ne pas confondre l' utilisation de = et de := .

Enfin, il faut faire attention à ne pas substituer un symbole par une expression contenant ce même symbole. Voyez plutôt :

In[ ]:=
```
Clear[x]
x := x + 1
x
```

$RecursionLimit : Recursion depth of 1024 exceeded during evaluation of x + 1.

Out[ ]=
```
Hold[x + 1]
```

Par contre, on peut tout à fait affecter à un symbole une expression contenant ce même symbole. Attention tout de même à ce type d'affectation, qui aura un impact sur la suite de vos opérations !

In[ ]:=
```
Clear[x]
x = 2
x = x + 2
x = x + 2
x = x + 2
```

Out[ ]=
```
2
```

Out[ ]=
```
4
```

Out[ ]=
```
6
```

Out[ ]=
```
8
```

Simplification d'une expression

Lorsqu'on réalise des calculs (lors de l'exécution d'un programme par exemple), les expressions peuvent rapidement devenir compliquées. On peut demander à Mathematica de simplifier ces expressions en utilisant les fonctions Simplify et FullSimplify.

In[ ]:=
```
Clear[x]
```
$$\frac{1}{3(1+x)}-\frac{-1+2x}{6(1-x+x^2)}+\frac{2}{3\left(1+\frac{1}{3}(-1+2x)^2\right)}$$
```
Simplify[%]
```

Out[ ]=
$$\frac{1}{3(1+x)}-\frac{-1+2x}{6(1-x+x^2)}+\frac{2}{3\left(1+\frac{1}{3}(-1+2x)^2\right)}$$

Out[ ]=
$$\frac{1}{1+x^3}$$

In[ ]:=
```
Sin[x]^2 + Cos[x]^2
Simplify[%]
```

Out[ ]= $\mathrm{Cos}[x]^2+\mathrm{Sin}[x]^2$

Out[ ]= 1

In[ ]:=
```
? FullSimplify
```

Out[ ]=

Symbol

FullSimplify [*expr*] tries a wide range of transformations on *expr* involving elementary and special functions and returns the simplest form it finds.

FullSimplify [*expr*, *assum*] does simplification using assumptions.

In[ ]:=
```
Simplify[x^3 - 6 x^2 + 11 x - 6]
FullSimplify[x^3 - 6 x^2 + 11 x - 6]
```

Out[ ]= $-6+11x-6x^2+x^3$

Out[ ]= $(-3+x)(-2+x)(-1+x)$

Accéder à une sortie précédente

Il est parfois pratique, comme dans l'exemple ci-dessus, de ne pas avoir à recopier une expression déjà entrée dans Mathematica. Ceci peut se faire grâce à la commande % : % contient la valeur de la dernière sortie, %% contient la valeur de l’avant-dernière sortie, et plus généralement, %...% (k fois) contient la valeur de la k-ième dernière sortie. Si l’on souhaite faire appel à une sortie spécifique numérotée n, on utilise les commandes %n ou Out[n].

In[ ]:=
```
Out[14]
```
Out[ ]=
$$\frac{11}{4}$$

Listes

Dans Mathematica, on travaille énormément avec des listes. On peut aussi avoir des listes de listes, et donc des listes de listes de listes, etc.

Le n-ième élément d’une liste s’obtient avec Part[liste,n] ou liste[[n]]

In[ ]:=
```
{{{1, 2}, {3}, {{{4, 5, {6, 7}}, 8}, 9}}, 10}[[1]]
Part[{{{1, 2}, {3}, {{{4, 5, {6, 7}}, 8}, 9}}, 10}, 1]
{{{1, 2}, {3}, {{{4, 5, {6, 7}}, 8}, 9}}, 10}[[2]]
Part[{{{1, 2}, {3}, {{{4, 5, {6, 7}}, 8}, 9}}, 10}, 2]
```
Out[ ]=
```
{{1, 2}, {3}, {{{4, 5, {6, 7}}, 8}, 9}}
```
Out[ ]=
```
{{1, 2}, {3}, {{{4, 5, {6, 7}}, 8}, 9}}
```
Out[ ]=
```
10
```
Out[ ]=
```
10
```

On peut imbriquer ces opérations et obtenir le m-ième élément du n-ième élément d’une liste (pour autant que celui-ci existe) avec Part[liste,n,m] ou liste[[n,m]]. Pour aller chercher des éléments plus profondément dans une liste, on peut itérer cette manoeuvre simplement en ajoutant des arguments dans Part ou [[...]].

In[ ]:=
```
{{{1, 2}, {3}, {{{4, 5, {6, 7}}, 8}, 9}}, 10}[[1, 3]]
Part[{{{1, 2}, {3}, {{{4, 5, {6, 7}}, 8}, 9}}, 10}, 1, 3]
```
Out[ ]=
```
{{{4, 5, {6, 7}}, 8}, 9}
```
Out[ ]=
```
{{{4, 5, {6, 7}}, 8}, 9}
```

Pour créer des listes, on utilise la fonction Table.

In[ ]:=
```
? Table
```

Out[ ]=

Symbol

Table [$expr$, n] generates a list of n copies of $expr$.

Table [$expr$, {i, i_{max}}] generates a list of the values of $expr$ when i runs from 1 to i_{max}.

Table [$expr$, {i, i_{min}, i_{max}}] starts with $i = i_{min}$.

Table [$expr$, {i, i_{min}, i_{max}, di}] uses steps di.

Table [$expr$, {i, {i_1, i_2, …}}] uses the successive values i_1, i_2, ….

Table [$expr$, {i, i_{min}, i_{max}}, {j, j_{min}, j_{max}}, …]
gives a nested list. The list associated with i is outermost .

In[ ]:=
```
Table[i^2, {i, 1, 10}]
```

Out[ ]=
```
{1, 4, 9, 16, 25, 36, 49, 64, 81, 100}
```

In[ ]:=
```
Table[i^2 + j, {i, 1, 10}, {j, 3, 5}]
```

Out[ ]=
```
{{4, 5, 6}, {7, 8, 9}, {12, 13, 14}, {19, 20, 21}, {28, 29, 30},
 {39, 40, 41}, {52, 53, 54}, {67, 68, 69}, {84, 85, 86}, {103, 104, 105}}
```

Algèbre linéaire avec *Mathematica*

Polynômes et fractions rationnelles

Comme vu précédemment, on peut faire du calcul formel avec des polynômes (à une ou plusieurs indéterminées). Mathematica possède une série de fonctions spécifiques : Expand, Coefficient, Exponent, Factor, Decompose, IrreduciblePolynomialQ, Apart, ...

In[•]:=
```
Clear[x, y, z]
(x + y) ^ 4
Expand[(x + y) ^ 4]
Coefficient[(x + y) ^ 4, x y ^ 3]
Exponent[1 + x ^ 2 + a x ^ 3, x]
Exponent[1 + x ^ 2 + a x ^ 3, a]
```

Out[•]= $(x + y)^4$

Out[•]= $x^4 + 4 x^3 y + 6 x^2 y^2 + 4 x y^3 + y^4$

Out[•]= 4

Out[•]= 3

Out[•]= 1

Voici quelques premières fonctions utiles pour le calcul avec des polynômes:

In[•]:=
```
Factor ; Decompose ; Apart ; Together ; Cancel ; Collect
```

Out[•]= Collect

In[•]:=
```
Factor[x ^ 10 - 1]
```

Out[•]= $(-1 + x) (1 + x) (1 - x + x^2 - x^3 + x^4) (1 + x + x^2 + x^3 + x^4)$

In[•]:=
```
Decompose[(x ^ 2 + 1) ^ 4, x]
```

Out[•]= $\{1 + 2 x + x^2, 2 x + x^2, x^2\}$

In[•]:=
```
Apart[1 / ((1 + x) (5 + x))]
```

Out[•]= $\frac{1}{4 (1 + x)} - \frac{1}{4 (5 + x)}$

In[•]:=
```
Together[x ^ 2 / (x ^ 2 - 1) + x / (x ^ 2 - 1)]
```

Out[•]= $\frac{x}{-1 + x}$

In[ ]:=
```
Cancel[x^2 + x y / x]
```
Out[ ]=
$x^2 + y$

In[ ]:=
```
Collect[(x y + y + z)^3, x]
```
Out[ ]=
$y^3 + x^3 y^3 + 3 y^2 z + 3 y z^2 + z^3 + x^2 (3 y^3 + 3 y^2 z) + x (3 y^3 + 6 y^2 z + 3 y z^2)$

In[ ]:=
```
? Decompose
```
Out[ ]=

Symbol

Decompose [*poly*, *x*] decomposes a polynomial,
if possible, into a composition of simpler polynomials.

In[ ]:=
```
2 x + x^2 /. x → x^2
```
Out[ ]=
$2 x^2 + x^4$

In[ ]:=
```
1 + 2 x + x^2 /. x → %
```
Out[ ]=
$1 + 2 (2 x^2 + x^4) + (2 x^2 + x^4)^2$

In[ ]:=
```
Simplify[%]
```
Out[ ]=
$(1 + x^2)^4$

On peut aussi tester si un polynôme est irréductible sur Q.

In[ ]:=
```
IrreduciblePolynomialQ [x^2 - 1]
IrreduciblePolynomialQ [x^2 - 2]
```
Out[ ]=
```
False
```
Out[ ]=
```
True
```

Division euclidienne de polynômes

Pour calculer le quotient et le reste d'une division euclidienne de polynômes, on utilise les fonctions PolynomialQuotient et PolynomialRemainder.

```
In[ ]:= PolynomialQuotient [4 x ^ 5 + 3 x ^ 4 + 8 x ^ 3 + x + 1, x ^ 2 + 1, x]
PolynomialRemainder [4 x ^ 5 + 3 x ^ 4 + 8 x ^ 3 + x + 1, x ^ 2 + 1, x]
```

Out[ ]= $-3 + 4 x + 3 x^2 + 4 x^3$

Out[ ]= $4 - 3 x$

In[ ]:= On vérifie :

Syntax::sntxi: Incomplete expression; more input is needed .

```
In[ ]:= Expand[%% (x ^ 2 + 1) + %]
```

Out[ ]= $1 + x + 8 x^3 + 3 x^4 + 4 x^5$

PolynomialGCD donne le pgcd de deux polynômes.

```
In[ ]:= PolynomialGCD [1 + 2 x + x^5 + 2 x^6, 1 + x^5 + x^7 + x^12]
```

Out[ ]= $1 + x^5$

On vérifie avec PolynomialQuotient :

```
In[ ]:= PolynomialRemainder [1 + 2 x + x^5 + 2 x^6, 1 + x^5, x]
PolynomialRemainder [1 + x^5 + x^7 + x^12, 1 + x^5, x]
```

Out[ ]= 0

Out[ ]= 0

On peut aussi travailler modulo m :

```
In[ ]:= PolynomialMod [4 x ^ 5 + 3 x ^ 4 + 8 x ^ 3 + x + 1, 3]
```

Out[ ]= $1 + x + 2 x^3 + x^5$

Pour la factorisation et la division euclidienne dans Z_m, m doit être premier (afin que Z_m soit un champ). Dans le cas contraire, Mathematica affichera une erreur.

In[ ]:=
```
Factor[4 x^5 + 3 x^4 + 8 x^3 + x + 1, Modulus → 3]
PolynomialQuotient [4 x^5 + 3 x^4 + 8 x^3 + x + 1, x^2 + 1, x, Modulus → 3]
PolynomialRemainder [4 x^5 + 3 x^4 + 8 x^3 + x + 1, x^2 + 1, x, Modulus → 3]
```

Out[ ]= $(1 + x)^2 (1 + 2 x + x^2 + x^3)$

Out[ ]= $x + x^3$

Out[ ]= 1

Résoudre une équation

La fonction Roots permet de résoudre des équations polynomiales. N’oubliez pas de spécifier les inconnues.

In[ ]:=
```
Roots[1 + 6 x + 9 x^2 + 3 x^3 + 4 x^4 + x^5 == 0, x]
```

Out[ ]=

$$x == -2\left(\frac{2}{3\left(-9+\sqrt{177}\right)}\right)^{1/3} + \frac{\left(\frac{1}{2}\left(-9+\sqrt{177}\right)\right)^{1/3}}{3^{2/3}} \;||$$

$$x == \left(1+i\sqrt{3}\right)\left(\frac{2}{3\left(-9+\sqrt{177}\right)}\right)^{1/3} - \frac{\left(1-i\sqrt{3}\right)\left(\frac{1}{2}\left(-9+\sqrt{177}\right)\right)^{1/3}}{2\times 3^{2/3}} \;||$$

$$x == \left(1-i\sqrt{3}\right)\left(\frac{2}{3\left(-9+\sqrt{177}\right)}\right)^{1/3} - \frac{\left(1+i\sqrt{3}\right)\left(\frac{1}{2}\left(-9+\sqrt{177}\right)\right)^{1/3}}{2\times 3^{2/3}} \;||\; x == -2-\sqrt{3} \;||\; x == -2+\sqrt{3}$$

On peut se servir de Mathematica pour se rappeler les formules étudiées en secondaire :

In[ ]:=
```
Roots[a x^2 + b x + c == 0, x]
```

Out[ ]=

$$x == \frac{-b-\sqrt{b^2-4\,a\,c}}{2\,a} \;||\; x == \frac{-b+\sqrt{b^2-4\,a\,c}}{2\,a}$$

Considérons un deuxième exemple, avec, cette fois, un polynôme irréductible sur Q.

In[ ]:=
```
IrreduciblePolynomialQ [4 x^5 + 3 x^4 + 8 x^3 + x + 1]
```

Out[ ]= True

Lorsqu' on demande, Mathematica nous rend ceci :

In[ ]:=
```
Roots[4 x^5 + 3 x^4 + 8 x^3 + x + 1 == 0, x]
```

Out[ ]=
```
x == -0.426 … || x == -0.421 … - 1.35 … i || x == -0.421 … + 1.35 … i ||
  x == 0.259 … - 0.475 … i || x == 0.259 … + 0.475 … i
```

Mathematica factorise le polynôme sur Z (au maximum de ses capacités). Mais cela ne nous donne pas toujours les zéros du polynôme. La fonction Solve de Mathematica nous donne les zéros d'un polynôme... lorsque c'est possible ! En effet, Mathematica est un logiciel de calcul symbolique, et on sait qu'il n'est pas possible d'obtenir de façon exacte les zéros d'un polynôme quelconque de degré strictement plus grand que 4. Dans notre premier exemple, Mathematica a réussi à factoriser le polynôme, et a donc pu nous donner les 5 zéros.

Néanmoins, des méthodes existent pour approcher les zéros d'un polynôme de degré arbitraire. Ceci relève de l'analyse numérique et non plus du calcul formel. Dans Mathematica, on utilise alors la fonction NSolve.

In[ ]:=
```
NRoots[4 x^5 + 3 x^4 + 8 x^3 + x + 1 == 0, x]
```

Out[ ]=
```
x == -0.425648 || x == -0.421257 - 1.35211 i || x == -0.421257 + 1.35211 i ||
  x == 0.259081 - 0.475099 i || x == 0.259081 + 0.475099 i
```

Pour des équations plus générales, on utilise les fonctions Solve et NSolve.

In[ ]:=
```
Solve[Sin[x] == Sin[Pi / 3], x]
```

Out[ ]=
$$\left\{\left\{x \to \boxed{\frac{\pi}{3} + 2\pi c_1 \text{ if } c_1 \in \mathbb{Z}}\right\}, \left\{x \to \boxed{\frac{2\pi}{3} + 2\pi c_1 \text{ if } c_1 \in \mathbb{Z}}\right\}\right\}$$

Il faut toujours spécifier les inconnues dans Solve. Effet, les équations peuvent dépendre de certains paramètres. Ainsi, les inconnues non spécifiées sont considérées comme des paramètres.

Voici un exemple d'équation paramétrique. Il s'agit d'une ellipse dépendant d'un paramètre a.

In[ ]:=
```
Solve[ x^2 + y^2 / a^2 == 4]
```

Out[ ]=
$$\left\{\left\{y \to -\sqrt{4 a^2 - a^2 x^2}\right\}, \left\{y \to \sqrt{4 a^2 - a^2 x^2}\right\}\right\}$$

Comme dit plus haut, si les inconnues ne sont pas spécifiées, Mathematica donne une inconnue (de son choix?) en fonction des autres. On peut alors forcer Mathematica à résoudre l'équation en x ou en a.

In[ ]:=
```
Solve[x^2 + y^2 / a^2 == 4, x]
Solve[x^2 + y^2 / a^2 == 4, a]
```

Out[ ]= $\left\{\left\{x \to -\frac{\sqrt{4\,a^2 - y^2}}{a}\right\}, \left\{x \to \frac{\sqrt{4\,a^2 - y^2}}{a}\right\}\right\}$

Out[ ]= $\left\{\left\{a \to -\frac{y}{\sqrt{4 - x^2}}\right\}, \left\{a \to \frac{y}{\sqrt{4 - x^2}}\right\}\right\}$

Résoudre un système d'équations

La fonction Solve nous permet de résoudre un système d'équations. Pour que des équations soit vérifiées simultanément, on les sépare par &&.

In[ ]:=
```
Solve[x + y == 2 && x - 3 y == -8, {x, y}]
```

Out[ ]= $\left\{\left\{x \to -\frac{1}{2}, y \to \frac{5}{2}\right\}\right\}$

Voici un exemple avec un système indéterminé (c'est-à-dire, de rang inférieur au nombre d'inconnues).

In[ ]:=
```
Solve[x + y == 2, {x, y}]
```

Solve : Equations may not give solutions for all "solve" variables .

Out[ ]= {{y → 2 – x}}

On peut également spécifier le domaine dans lequel on veut résoudre le système. Par exemple, on pourra spécifier Reals, Integers ou Complexes, selon que l'on souhaite résoudre le système dans R, C ou Z.

In[ ]:=
```
Solve[x + y == 2 && x ≥ 0 && y ≥ 0, {x, y}, Integers]
```

Out[ ]= {{x → 0, y → 2}, {x → 1, y → 1}, {x → 2, y → 0}}

En fait, la fonction Solve nous permet de résoudre un système d'équations quelconques (pas nécessairement linéaires), pour autant que Mathematica en soit capable.

In[ ]:=
```
Solve[x + y == 2 && x y == -8, {x, y}]
```

Out[ ]= {{x → -2, y → 4}, {x → 4, y → -2}}

In[ ]:=
```
Solve[Sin[x] == Sin[Pi / 3] && 0 ≤ x ≤ 2 Pi, x]
```

Out[ ]=

$$\left\{\left\{x \to \frac{\pi}{3}\right\}, \left\{x \to \frac{2\pi}{3}\right\}\right\}$$

Vecteurs et matrices

Dans Mathematica, les vecteurs et les matrices sont simplement représentés par des listes. Voici un vecteur-colonne de longueur 4 et une matrice 2x2.

In[ ]:=
```
v = {1, 2, 5, 0}
MatrixForm[v]
m = {{1, 3}, {0, 9}}
MatrixForm[m]
```

Out[ ]=
```
{1, 2, 5, 0}
```

Out[ ]//MatrixForm=

$$\begin{pmatrix} 1 \\ 2 \\ 5 \\ 0 \end{pmatrix}$$

Out[ ]=
```
{{1, 3}, {0, 9}}
```

Out[ ]//MatrixForm=

$$\begin{pmatrix} 1 & 3 \\ 0 & 9 \end{pmatrix}$$

La multiplication d'une matrice ou d’un vecteur par une constante est réalisée comme suit.

In[ ]:=
```
a v
3 m
```

Out[ ]=
```
{a, 2 a, 5 a, 0}
```

Out[ ]=
```
{{3, 9}, {0, 27}}
```

L'addition se fait composante à composante, quand les vecteurs (ou les matrices) sont de tailles égales.

In[ ]:=
```
v ' = {1, 3, 2, 2}
m ' = {{1, 0}, {1, 0}}
v + v '
m + m '
```

Out[ ]=
```
{1, 3, 2, 2}
```

Out[ ]=
```
{{1, 0}, {1, 0}}
```

Out[ ]=
```
{2, 5, 7, 2}
```

Out[ ]=
```
{{2, 3}, {1, 9}}
```

On peut multiplier une matrice et un vecteur, pour autant bien sûr que la multiplication soit définie. On utilise explicitement le “.”, et non juste un espace ou une étoile comme pour la multiplication par un nombre. Comparez les sorties obtenues ci-après.

In[ ]:=
```
Clear[a, b, c, d, e, f, v, m]; v = {a, b}; m = {{c, d}, {e, f}};
v.m
m.v
```

Out[ ]=
```
{a c + b e, a d + b f}
```

Out[ ]=
```
{a c + b d, a e + b f}
```

In[ ]:=
```
v m
m v
```

Out[ ]=
```
{{a c, a d}, {b e, b f}}
```

Out[ ]=
```
{{a c, a d}, {b e, b f}}
```

Dans la première, {a,b} est considéré comme un vecteur-ligne et dans la deuxième, {a,b} est considéré comme un vecteur-colonne. En particulier, on voit que ces deux multiplications sont différentes (la multiplication matricielle est effectivement non commutative). Les troisième et quatrième multiplications, on a utilisé l’espace et non le “.” et la multiplication s’est faite composante à composante. On remarque également que Mathematica considère qu’il travaille avec des symboles qui commutent pour le produit.

Quand on veut considérer un vecteur ligne ou colonne (et non indéfini comme pour v ci-dessus), on l’écrit comme une matrice.

In[ ]:=
```
ligne = {{1, 2, 5, 0}}; MatrixForm[ligne]
colonne = {{1}, {2}, {5}, {0}};
MatrixForm[colonne]
```

Out[ ]//MatrixForm=
```
(1 2 5 0)
```

Out[ ]//MatrixForm=

$$\begin{pmatrix} 1 \\ 2 \\ 5 \\ 0 \end{pmatrix}$$

Ainsi, la multiplication de m par {{a, b}} est définie à gauche mais pas à droite.

In[ ]:=
```
r = {{a, b}};
r.m
m.r
```

Out[ ]=
```
{{a c + b e, a d + b f}}
```

··· Dot : Tensors {{c, d}, {e, f}} and {{a, b}} have incompatible shapes .

Out[ ]=
```
{{c, d}, {e, f}}.{{a, b}}
```

Puisqu'il s'agit de listes, on utilise [[ ]] pour extraire les éléments d'un vecteur ou d'une matrice.

In[ ]:=
```
MatrixForm[m]
m[[1]]
m[[1, 2]]
m[[1]][[2]]
```

Out[ ]//MatrixForm=

$$\begin{pmatrix} c & d \\ e & f \end{pmatrix}$$

Out[ ]=
```
{c, d}
```

Out[ ]=
```
d
```

Out[ ]=
```
d
```

Pour extraire la première colonne d'une matrice, on peut utiliser :

In[ ]:=
```
m[[All, 1]]
m[[All, 2]]
```

Out[ ]=
```
{c, e}
```

Out[ ]=
```
{d, f}
```

Voici quelques fonctions utiles :

In[ ]:=
```
Transpose ; Dimensions ; Det; Tr; Inverse ;
```

Nous allons donner les exemples à partir de la matrice m :

In[ ]:=
```
MatrixForm [m]
```

Out[ ]//MatrixForm=

$$\begin{pmatrix} c & d \\ e & f \end{pmatrix}$$

In[ ]:=
```
Transpose [m] // MatrixForm
```

Out[ ]//MatrixForm=

$$\begin{pmatrix} c & e \\ d & f \end{pmatrix}$$

In[ ]:=
```
Dimensions [m]
```

Out[ ]=
```
{2, 2}
```

In[ ]:=
```
Det[m]
```

Out[ ]=
```
-d e + c f
```

In[ ]:=
```
Tr[m]
```

Out[ ]=
```
c + f
```

In[ ]:=
```
Inverse [m] // MatrixForm
```

Out[ ]//MatrixForm=

$$\begin{pmatrix} \frac{f}{-d\,e+c\,f} & -\frac{d}{-d\,e+c\,f} \\ -\frac{e}{-d\,e+c\,f} & \frac{c}{-d\,e+c\,f} \end{pmatrix}$$

Les fonctions IdentityMatrix et DiagonalMatrix permettent de générer des matrices facilement.

In[•]:=
```
IdentityMatrix[4] // MatrixForm
DiagonalMatrix[{1, 2, 3, 4}] // MatrixForm
```

Out[•]//MatrixForm=

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Out[•]//MatrixForm=

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 4 \end{pmatrix}$$

À partir de deux listes, la fonction Join crée une liste composée des éléments de chacune de ses listes, ceux de la première venant avant ceux de la deuxième. On peut se servir de cette fonction pour créer des matrices augmentées, soit d'une ligne, soit d'une colonne (auquel cas, on compose Join et Transpose).

In[•]:=
```
Clear[a, b, c, g, h]
Join[{a, b, c}, {1, 2, 3, 4}]
Join[m, {{g, h}}] // MatrixForm
Transpose[Join[Transpose[m], {{g, h}}]] // MatrixForm
```

Out[•]= {a, b, c, 1, 2, 3, 4}

Out[•]//MatrixForm=

$$\begin{pmatrix} c & d \\ e & f \\ g & h \end{pmatrix}$$

Out[•]//MatrixForm=

$$\begin{pmatrix} c & d & g \\ e & f & h \end{pmatrix}$$

Pour élever une matrice à une puissance donnée, on utilise la fonction MatrixPower.

In[•]:=
```
MatrixPower[m, 4]
```

Out[•]=

$$\{\{(c^2 + d\,e)^2 + (c\,d + d\,f)\,(c\,e + e\,f),\ (c^2 + d\,e)\,(c\,d + d\,f) + (c\,d + d\,f)\,(d\,e + f^2)\},$$
$$\{(c^2 + d\,e)\,(c\,e + e\,f) + (c\,e + e\,f)\,(d\,e + f^2),\ (c\,d + d\,f)\,(c\,e + e\,f) + (d\,e + f^2)^2\}\}$$

Notez la différence avec la syntaxe suivante, qui rend la matrice obtenue en élevant chaque élément à la puissance 4.

In[ ]:=
```
m^4
```

Out[ ]=
$\{\{c^4, d^4\}, \{e^4, f^4\}\}$

Valeurs propres et vecteurs propres

On peut calculer le polynôme caractéristique d'une matrice à l'aide de la fonction Characteris - ticPolynomial. Il est impératif de spécifier l'indéterminée du polynôme.

In[ ]:=
```
a = Table[i + j, {i, 1, 4}, {j, 3, 6}]
CharacteristicPolynomial [a, x]
```

Out[ ]=
{{4, 5, 6, 7}, {5, 6, 7, 8}, {6, 7, 8, 9}, {7, 8, 9, 10}}

Out[ ]=
$-20\, x^2 - 28\, x^3 + x^4$

On peut le vérifier :

In[ ]:=
```
Det[a - x IdentityMatrix [4]]
```

Out[ ]=
$-20\, x^2 - 28\, x^3 + x^4$

On calcule les valeurs propres à l'aide de la fonction Eigenvalues. La fonction Eigenvectors rend des vecteurs propres associés à chacune des valeurs propres. Remarquons qu'en fait, pour chacune des valeurs propres s, Mathematica rend d vecteurs propres linéairement indépendants si d est la multiplicité géométrique de s.

In[ ]:=
```
Eigenvalues [a]
```

Out[ ]=
$\left\{2\left(7+3\sqrt{6}\right), 2\left(7-3\sqrt{6}\right), 0, 0\right\}$

In[ ]:=
```
Eigenvectors [a]
```

Out[ ]=
$$\left\{\left\{-\frac{-461-189\sqrt{6}}{707+288\sqrt{6}}, \frac{3\left(181+74\sqrt{6}\right)}{707+288\sqrt{6}}, \frac{5\left(125+51\sqrt{6}\right)}{707+288\sqrt{6}}, 1\right\},\right.$$
$$\left.\left\{-\frac{461-189\sqrt{6}}{-707+288\sqrt{6}}, \frac{3\left(-181+74\sqrt{6}\right)}{-707+288\sqrt{6}}, \frac{5\left(-125+51\sqrt{6}\right)}{-707+288\sqrt{6}}, 1\right\}, \{2, -3, 0, 1\}, \{1, -2, 1, 0\}\right\}$$

In[ ]:=
```
Eigensystem[a]
```

Out[ ]=

$$\left\{\left\{2\left(7+3\sqrt{6}\right),-2\left(-7+3\sqrt{6}\right),0,0\right\},\left\{\left\{-\frac{-461-189\sqrt{6}}{707+288\sqrt{6}},\frac{3\left(181+74\sqrt{6}\right)}{707+288\sqrt{6}},\frac{5\left(125+51\sqrt{6}\right)}{707+288\sqrt{6}},1\right\},\left\{-\frac{461-189\sqrt{6}}{-707+288\sqrt{6}},\frac{3\left(-181+74\sqrt{6}\right)}{-707+288\sqrt{6}},\frac{5\left(-125+51\sqrt{6}\right)}{-707+288\sqrt{6}},1\right\},\{2,-3,0,1\},\{1,-2,1,0\}\right\}\right\}$$

In[ ]:=
```
N[Eigensystem[a]]
```

Out[ ]=
```
{{28.6969, -0.696938, 0., 0.}, {{0.654148, 0.769432, 0.884716, 1.},
  {-1.26284, -0.508563, 0.245719, 1.}, {2., -3., 0., 1.}, {1., -2., 1., 0.}}}
```

Diagonalisation et forme de Jordan

La fonction JordanDecomposition nous rend la forme de Jordan d'une matrice. Si la matrice est diagonalisable, la forme de Jordan correspond à la forme diagonalisée.

In[ ]:=
```
Map[MatrixForm, JordanDecomposition[a]]
```

Out[ ]=

$$\left\{\begin{pmatrix} 2 & 1 & \frac{-461+189\sqrt{6}}{-707+288\sqrt{6}} & \frac{461+189\sqrt{6}}{707+288\sqrt{6}} \\ -3 & -2 & \frac{3(-181+74\sqrt{6})}{-707+288\sqrt{6}} & \frac{3(181+74\sqrt{6})}{707+288\sqrt{6}} \\ 0 & 1 & \frac{5(-125+51\sqrt{6})}{-707+288\sqrt{6}} & \frac{5(125+51\sqrt{6})}{707+288\sqrt{6}} \\ 1 & 0 & 1 & 1 \end{pmatrix},\begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 14-6\sqrt{6} & 0 \\ 0 & 0 & 0 & 14+6\sqrt{6} \end{pmatrix}\right\}$$

Si la forme de Jordan d'une matrice A est F, alors on peut trouver une matrice S telle que A=S.F.Inverse[S]. La première matrice correspond à une telle matrice S et la seconde à la matrice F. Ici, la matrice a est diagonalisable, et F est bien une matrice diagonale. Les colonnes de S sont données par les vecteurs propres rendus précédemment.

Voici un exemple de matrice non diagonalisable. La forme de Jordan, elle, existe toujours.

In[ ]:=
```
A = {{27, 48, 81}, {-6, 0, 0}, {1, 0, 3}}; JordanDecomposition[A];
Map[MatrixForm, %]
```

Out[ ]=

$$\left\{\begin{pmatrix} 3 & 18 & 2 \\ -3 & -9 & -\frac{1}{4} \\ 1 & 2 & 0 \end{pmatrix},\begin{pmatrix} 6 & 0 & 0 \\ 0 & 12 & 1 \\ 0 & 0 & 12 \end{pmatrix}\right\}$$

Vérifions :

In[•]:=
```
S = {{3, 18, 2}, {-3, -9, -1/4}, {1, 2, 0}};
F = {{6, 0, 0}, {0, 12, 1}, {0, 0, 12}};

S.F.Inverse[S] == A
```

Out[•]=
```
True
```

Noyau d’une matrice

La fonction NullSpace rend une base du noyau d’une matrice. Dans les exemples ci-dessus, on a pu remarquer que la matrice a possédait 0 comme valeur propre double. Par contre, 0 n’est pas valeur propre de la matrice A. Nous pouvons le vérifier :

In[•]:=
```
NullSpace[a]
NullSpace[A]
```

Out[•]=
```
{{2, -3, 0, 1}, {1, -2, 1, 0}}
```

Out[•]=
```
{}
```

Cours 3 : Analyse et graphiques avec *Mathematica*

Analyse avec *Mathematica*

Limites

Le calcul d'une limite se fait à l'aide de la fonction Limit (ça ne s'invente pas).

```
In[ ]:= Limit[Sin[x] / x, x → 0]
Out[ ]= 1
```

```
In[ ]:= Limit[(1 + x / n) ^ n, n → Infinity]
Out[ ]= ⅇ^x
```

On peut calculer la limite d'une fonction définie par morceaux. C'est instructif quant à la définition de la limite utilisée par Mathematica.

```
In[ ]:= Limit[Piecewise[{{45, x == 2}}, 3 x - 1], x → 2]
Out[ ]= 5
```

```
In[ ]:= ? Piecewise
```

Out[ ]=

Symbol

Piecewise [{{val_1, $cond_1$}, {val_2, $cond_2$}, …}] represents a piecewise function with values val_i in the regions defined by the conditions $cond_i$.

Piecewise [{{val_1, $cond_1$}, …}, val] uses default value val if none of the $cond_i$ apply. The default for val is 0.

Si on souhaite calculer la limite à gauche ou à droite, on spécifie l'option Direction→"FromAbove" (limite à droite) ou Direction→"FromBelow" (limite à gauche).

```
In[ ]:= Limit[Floor[x ^ 2] x, x → 4, Direction → "FromAbove"]
        Limit[Floor[x ^ 2] x, x → 4, Direction → "FromBelow"]
Out[ ]= 64
Out[ ]= 60
```

Lorsque la limite n'existe pas, Mathematica nous le signale.

In[ ]:=
```
Limit[Floor[x^2] x, x → 4]
```
Out[ ]=
```
Indeterminate
```

Lorsqu'on entre une valeur numérique (flottant) au lieu d'une valeur exacte, Mathematica peut rendre une réponse incorrecte. Cela est dû à des erreurs d'approximations intermédiaires, qui peuvent fortement fausser le calcul d'une limite. Voici l'exemple de la documentation de Mathematica.

In[ ]:=
```
Limit[Log[1 - (Log[Exp[z] / z - 1] + Log[z]) / z] / z, z → 100.]
```
Out[ ]=
```
-∞
```

In[ ]:=
```
Limit[Log[1 - (Log[Exp[z] / z - 1] + Log[z]) / z] / z, z → 100]
```
Out[ ]=

$$\frac{1}{100}\,\mathrm{Log}\left[1+\frac{1}{100}\left(-\mathrm{Log}[100]-\mathrm{Log}\left[-1+\frac{e^{100}}{100}\right]\right)\right]$$

C'est le deuxième calcul qui donne la valeur exacte. Si on veut une valeur approchée, mieux vaut donc la demander a posteriori. Remarquons qu'a priori, on pourrait penser que la limite vaut 1.

In[ ]:=

$$N\left[\frac{1}{100}\,\mathrm{Log}\left[1+\frac{1}{100}\left(-\mathrm{Log}[100]-\mathrm{Log}\left[-1+\frac{e^{100}}{100}\right]\right)\right],\ 10\right]$$

$$N\left[\frac{1}{100}\,\mathrm{Log}\left[1+\frac{1}{100}\left(-\mathrm{Log}[100]-\mathrm{Log}\left[-1+\frac{e^{100}}{100}\right]\right)\right],\ 100\right]$$

Out[ ]=
```
-1.000000000
```
Out[ ]=
```
-0.9999999999999999999999999999999999999999999998139962011989582018520152098064408313205509706938449815 70
```

Sommes et séries

La fonction Sum permet de calculer la somme des valeurs dans une liste.

In[ ]:=
```
Sum[i^2, {i, 1, 10}]
```
Out[ ]=
```
385
```

On peut calculer la somme des termes d'indices impairs. Le 2 signifie qu'on prend un terme sur deux, donc ici, les termes d'indices 1, 3, 5, 7, 9.

In[ ]:=
```
Sum[i^2, {i, 1, 10, 2}]
```
Out[ ]=
```
165
```

On vérifie :

In[ ]:=
```
1 + 3^2 + 5^2 + 7^2 + 9^2
```
Out[ ]=
```
165
```

On peut ne garder qu'un terme sur trois, ou plus généralement, qu'un terme sur k.

In[ ]:=
```
Sum[i^2, {i, 1, 10, 3}]
1 + 4^2 + 7^2 + 10^2
```
Out[ ]=
```
166
```
Out[ ]=
```
166
```

On peut choisir les indices dans une liste d’entiers quelconque.

In[ ]:=
```
t = Table[i^2, {i, 1, 10}]
Sum[Sin[i], {i, t}]
```
Out[ ]=
```
{1, 4, 9, 16, 25, 36, 49, 64, 81, 100}
```
Out[ ]=
```
Sin[1] + Sin[4] + Sin[9] + Sin[16] + Sin[25] + Sin[36] + Sin[49] + Sin[64] + Sin[81] + Sin[100]
```

Le début et la fin de l’intervalle de variation des indices peuvent être des expressions symboliques:

In[ ]:=
```
Clear[a, b]
Sum[i^2, {i, 1, b}]
Sum[i^2, {i, a, b}]
```
Out[ ]=

$$\frac{1}{6} b (1 + b) (1 + 2 b)$$

Out[ ]=

$$-\frac{1}{6} (-1 + a - b) \left(-a + 2 a^2 + b + 2 a b + 2 b^2\right)$$

Dès lors, si par exemple la borne supérieure de l’intervalle est 3Sqrt[b]+1, Mathematica rend la sortie suivante. Quel sens donner à ceci ?

In[•]:=
```
Sum[i^2, {i, a, 3 Sqrt[b] + 1}]
```

Out[•]=
$$-\frac{1}{6}\left(-2+a-3\sqrt{b}\right)\left(3+a+2a^2+15\sqrt{b}+6a\sqrt{b}+18b\right)$$

On peut faire des sommes à multi-indices.

In[•]:=
```
Sum[(i + j)^2, {i, 1, 7}, {j, 3, 9}]
```

Out[•]=
```
5292
```

Le deuxième indice peut dépendre du premier.

In[•]:=
```
Sum[(i + j)^2, {i, 1, 4}, {j, i, 9}]
```

Out[•]=
```
2150
```

Pour les sommes infinies, aka les séries, on fait simplement varier l'indice jusqu'à l'infini :

In[•]:=
```
Sum[i^2, {i, 1, Infinity}]
```

••• Sum : Sum does not converge .

Out[•]=
$$\sum_{i=1}^{\infty} i^2$$

Si la série est divergente, Mathematica nous le signale. Sinon, et lorsqu'il le peut, Mathematica rend la valeur de la somme.

In[•]:=
```
Sum[(1 / i^2), {i, 1, Infinity}]
Sum[(1 / (i j)^2), {i, 1, Infinity}, {j, 1, Infinity}]
```

Out[•]=
$$\frac{\pi^2}{6}$$

Out[•]=
$$\frac{\pi^4}{36}$$

In[•]:=
```
Sum[1 / i, {i, 1, Infinity}]
```

••• Sum : Sum does not converge .

Out[•]=
$$\sum_{i=1}^{\infty} \frac{1}{i}$$

La fonction Series donne le développement en série de Taylor d'une fonction autour d'un point a et à l'ordre n souhaité. La fonction Normal nous permet de supprimer le terme asymptotique.

In[ ]:=
```
Series[Sin[x], {x, a, 4}]
Normal[%]
```

Out[ ]=
$$\text{Sin}[a] + \text{Cos}[a]\,(x - a) - \frac{1}{2}\,\text{Sin}[a]\,(x - a)^2 - \frac{1}{6}\,\text{Cos}[a]\,(x - a)^3 + \frac{1}{24}\,\text{Sin}[a]\,(x - a)^4 + O[x - a]^5$$

Out[ ]=
$$(-a + x)\,\text{Cos}[a] - \frac{1}{6}\,(-a + x)^3\,\text{Cos}[a] + \text{Sin}[a] - \frac{1}{2}\,(-a + x)^2\,\text{Sin}[a] + \frac{1}{24}\,(-a + x)^4\,\text{Sin}[a]$$

Par curiosité, on peut se demander comment Mathematica considère ce terme asymptotique.

In[ ]:=
```
Head[O[x - a]^5]
FullForm[O[x - a]^5]
Normal[O[x - a]^5]
```

Out[ ]=
```
SeriesData
```

Out[ ]//FullForm=
```
SeriesData[x, a, List[], 5, 5, 1]
```

Out[ ]=
```
0
```

Produits

La fonction Product est l'analogue de Sum pour les produits.

In[ ]:=
```
Product[i^2, {i, 1, 10}]
Product[Sin[i], {i, Table[i^2, {i, 1, 10}]}]
Clear[a, b]
Product[i^2, {i, 1, b}]
Product[i^2, {i, a, b}]
Product[i^2, {i, a, 3 Sqrt[b] + 1}]
```

Out[ ]=
```
13 168 189 440 000
```

Out[ ]=
```
Sin[1] Sin[4] Sin[9] Sin[16] Sin[25] Sin[36] Sin[49] Sin[64] Sin[81] Sin[100]
```

Out[ ]=
$$(b!)^2$$

Out[ ]=
$$\text{Pochhammer}[a, 1 - a + b]^2$$

Out[ ]=
$$\text{Pochhammer}\left[a, 2 - a + 3\sqrt{b}\right]^2$$

Notez que le symbole de Pochhammer est une fonction de Mathematica.

In[ ]:=
```
Pochhammer[x, 5]
```

Out[ ]=
```
x (1 + x) (2 + x) (3 + x) (4 + x)
```

On a donc bien

In[ ]:=
```
Pochhammer[x, 5] == Product[i, {i, x, x + 4}]
```

Out[ ]=
```
True
```

Définir une fonction

Pour définir une fonction dans Mathematica, on utilise la syntaxe suivante :

In[ ]:=
```
f[x_, y_, z_] := 3 x^2 y^3 + x Cos[z]
```

Remarquez que cette commande n'a pas de sortie associée. Mais Mathematica peut maintenant calculer des valeurs de cette fonction en des points choisis.

In[ ]:=
```
f[1, 2, Pi]
f[x, y, Pi]
f[a, b, c]
```

Out[ ]= 23

Out[ ]= $-x + 3 x^2 y^3$

Out[ ]= $3 a^2 b^3 + a \, \mathrm{Cos}[c]$

Remarquez que c'est la deuxième fois qu'on utilise la syntaxe :=. En effet, nous l'avions déjà utilisée pour substituer une variable symbolique dans une autre. Il s'agit donc ici d'une autre utilisation, dans un contexte différent, d'une même syntaxe. Y a-t-il pour autant ambiguïté ?

On peut dériver une telle fonction, et ce par rapport à chacun de ses arguments, et autant de fois qu'on le souhaite.

In[ ]:=
```
Derivative[1, 0, 0][f][a, b, c]
Derivative[1, 2, 0][f][a, b, c]
Derivative[1, 0, 1][f][a, b, c]
```

Out[ ]= $6 a b^3 + \mathrm{Cos}[c]$

Out[ ]= $36 a b$

Out[ ]= $-\mathrm{Sin}[c]$

Mais la dérivation mérite bien une section à part entière.

Dérivation

La dérivée d'une fonction d'une seule variable s'obtient d'une des façons suivantes :

In[◦]:=
```
Sin'[x]
```

Out[◦]=
```
Cos[x]
```

ou

In[◦]:=
```
D[Sin[x], x]
```

Out[◦]=
```
Cos[x]
```

ou

In[◦]:=
```
Derivative[1][Sin][x]
```

Out[◦]=
```
Cos[x]
```

Consultons la documentation avant de commenter les trois commandes précédentes.

In[◦]:=
```
? Derivative
```

Out[◦]=

Symbol

f' represents the derivative of a function f of one argument.

Derivative$[n_1, n_2, \ldots][f]$ is the general form, representing a function obtained from f by differentiating n_1 times with respect to the first argument, n_2 times with respect to the second argument, and so on.

In[◦]:=
```
? D
```

Out[◦]=

Symbol

D$[f, x]$ gives the partial derivative $\partial f / \partial x$.

D$[f, \{x, n\}]$ gives the multiple derivative $\partial^n f / \partial x^n$.

D$[f, x, y, \ldots]$ gives the partial derivative $\cdots (\partial / \partial y)(\partial / \partial x) f$.

D$[f, \{x, n\}, \{y, m\}, \ldots]$ gives the multiple partial derivative $\cdots (\partial^m / \partial y^m)(\partial^n / \partial x^n) f$.

D$[f, \{\{x_1, x_2, \ldots\}\}]$ for a scalar f gives the vector derivative $(\partial f / \partial x_1, \partial f / \partial x_2, \ldots)$.

D$[f, \{array\}]$ gives an array derivative.

Dans le cas de D[Sin[x],x], on dérive une expression symbolique en la variable symbolique x. Ainsi,

In[ ]:=
```
D[Sin[x^2], x]
```

Out[ ]=
```
2 x Cos[x^2]
```

alors que les essais de syntaxes suivantes ne rendent rien d’intéressant (car incorrects bien sûr) :

In[ ]:=
```
Sin[x^2]'
Sin'[x^2]
```

Out[ ]=
```
Sin[x^2]′
```

Out[ ]=
```
Cos[x^2]
```

En effet, D est utilisé pour la dérivation d’expression symbolique alors que ’ et Derivative sont utilisés pour la dérivation de fonctions. Il est à noter que f’ est juste un raccourci pour Derivative[1][f].

Pour utiliser ’ à bon escient, il faut au préalable définir la fonction x -> Sin[x^2] (voir la section précédente).

In[ ]:=
```
Clear[g]
g[x_] := Sin[x^2]
g'[x]
```

Out[ ]=
```
2 x Cos[x^2]
```

L'avantage est que g’ est maintenant une nouvelle fonction, qu’on peut évaluer en n’importe quelle expression symbolique ou valeur.

In[ ]:=
```
Clear[a]
g'[z]
g'[a^2 + b]
g'[E]
g'[0]
```

Out[ ]=
```
2 z Cos[z^2]
```

Out[ ]=
```
2 (a^2 + b) Cos[(a^2 + b)^2]
```

Out[ ]=
```
2 ⅇ Cos[ⅇ^2]
```

Out[ ]=
```
0
```

Comme noté précédemment, on a

In[ ]:=
```
g'
Derivative[1][g]
g' == Derivative[1][g]
```

Out[ ]=
```
2 Cos[#1^2] #1 &
```

Out[ ]=
```
2 Cos[#1^2] #1 &
```

Out[ ]=
```
True
```

Dans le cas de la dérivation d’expression symbolique, il faut veiller à utiliser des variables symboliques libres.

In[ ]:=
```
x = 2;
D[Sin[x], {x, 2}]
```

••• General : 2 is not a valid variable .

Out[ ]=
```
∂{2,2} Sin[2]
```

On lui rend donc sa liberté.

In[ ]:=
```
Clear[x]
```

Dérivée seconde

On peut calculer la dérivée seconde avec

In[ ]:=
```
D[Sin[x], {x, 2}]
D[g[x], {x, 2}]
```

Out[ ]=
```
-Sin[x]
```

Out[ ]=
```
2 Cos[x^2] - 4 x^2 Sin[x^2]
```

ou

In[ ]:=
```
Derivative[2][Sin][x]
Derivative[2][g][y]
```

Out[ ]=
```
-Sin[x]
```

Out[ ]=
```
2 Cos[y^2] - 4 y^2 Sin[y^2]
```

À nouveau, nous avons le raccourci ” :

In[ ]:=
```
Sin''[x]
g''[y]
```

Out[ ]=
```
-Sin[x]
```

Out[ ]=
$2\,\text{Cos}[y^2] - 4\,y^2\,\text{Sin}[y^2]$

En fait, on peut utiliser le ' autant de fois qu'on veut :

In[ ]:=
```
g'''''[y]
```

Out[ ]=
$-120\,y\,\text{Cos}[y^2] + 32\,y^5\,\text{Cos}[y^2] + 160\,y^3\,\text{Sin}[y^2]$

Dérivée partielle

Si on veut calculer une dérivée partielle, on utilise D ou Derivative selon que l'on veut effectuer une dérivation symbolique ou une dérivation de fonction. Nous prenons ici comme exemple la fonction Log.

In[ ]:=
```
Log[a, b]
```

Out[ ]=
$\frac{\text{Log}[b]}{\text{Log}[a]}$

In[ ]:=
```
D[Log[a, b], a]
```

Out[ ]=
$-\frac{\text{Log}[b]}{a\,\text{Log}[a]^2}$

In[ ]:=
```
Derivative[0, 1][Log][a, b]
Derivative[1, 0][Log][a, b]
```

Out[ ]=
$\frac{1}{b\,\text{Log}[a]}$

Out[ ]=
$-\frac{\text{Log}[b]}{a\,\text{Log}[a]^2}$

Dans la section "Définir une fonction", nous avions calculé des dérivées partielles de la fonction f, que nous avions définie au préalable :

In[ ]:=
```
f[a, b, c]
```

Out[ ]=
$3\,a^2\,b^3 + a\,\text{Cos}[c]$

On aurait pu aussi utiliser la dérivation symbolique pour obtenir le même résultat :

In[ ]:=
```
Derivative[1, 2, 0][f][a, b, c]
D[D[D[f[a, b, c], a], b], b]
```

Out[ ]= 36 a b

Out[ ]= 36 a b

Dérivée totale

Avec la fonction D, Mathematica suppose que les variables symboliques sont indépendantes les unes des autres. Si l'on souhaite au contraire calculer la dérivée par rapport à x d'une expression contenant plusieurs variables symboliques (dont x évidemment) en considérant qu'elles dépendent toutes de x, on utilise la fonction Dt. Comparez les exemples suivants.

In[ ]:=
```
D[x^2 + y^2, x]
Dt[x^2 + y^2, x]
```

Out[ ]= 2 x

Out[ ]= 2 x + 2 y Dt[y, x]

Primitivation

On calcule une primitive formelle d'une expression symbolique de la façon suivante :

In[ ]:=
```
Integrate[1 / (x^3 + 1), x]
```

Out[ ]=
$$\frac{\text{ArcTan}\left[\frac{-1+2x}{\sqrt{3}}\right]}{\sqrt{3}} + \frac{1}{3}\text{Log}[1+x] - \frac{1}{6}\text{Log}[1-x+x^2]$$

Remarquons que la primitive donnée par Mathematica est une primitive valable sur l'intervalle] -1, +∞[. Ceci est dû au caractère formel de la primitivation dans Mathematica. En effet, Integrate est l'analogue de D (et non de Derivative). Ainsi, en calculant la dérivée formelle, on obtient

In[ ]:=
```
D[Integrate[1 / (x^3 + 1), x], x]
```

Out[ ]=
$$\frac{1}{3(1+x)} - \frac{-1+2x}{6(1-x+x^2)} + \frac{2}{3\left(1+\frac{1}{3}(-1+2x)^2\right)}$$

et en simplifiant, on retrouve bien l'expression de départ :

In[ ]:=
```
FullSimplify[%]
```

Out[ ]=
$$\frac{1}{1+x^3}$$

Selon la forme de l'expression en entrée, la primitive proposée par Mathematica peut être différente (à une constante près, évidemment).

In[ ]:=
```
Integrate[1 + (x + 1)^3, x]
Integrate[Expand[1 + (x + 1)^3], x]
Simplify[%% - %]
```

Out[ ]=
$$x+\frac{1}{4}(1+x)^4$$

Out[ ]=
$$2x+\frac{3x^2}{2}+x^3+\frac{x^4}{4}$$

Out[ ]=
$$\frac{1}{4}$$

Intégration

La fonction Integrate nous permet également de calculer une intégrale. On doit alors spécifier les bornes d'intégration.

In[ ]:=
```
Integrate[1 / x^2, {x, 1, Infinity}]
```

Out[ ]=
$$1$$

Remarquons que si l'intégrale n'existe pas, Mathematica nous le signale :

In[ ]:=
```
Integrate[1 / x^2, {x, -1, 1}]
```

••• Integrate : Integral of $\frac{1}{x^2}$ does not converge on {-1, 1}.

Out[ ]=
$$\int_{-1}^{1}\frac{1}{x^2}\,dx$$

Si l'intégrale dépend d'un paramètre, on peut ajouter des hypothèses sur celui-ci. Par exemple :

In[ ]:=
```
Integrate[1/x^2, {x, n, Infinity}]
Integrate[1/x^2, {x, n, Infinity}, Assumptions → n > 0]
```

Out[ ]=
$$\frac{1}{n} \quad \text{if} \quad \mathrm{Im}[n] \neq 0 \,||\, \mathrm{Re}[n] > 0$$

Out[ ]=
$$\frac{1}{n}$$

Le calcul d'une intégrale multiple se réalise comme suit :

In[ ]:=
```
Integrate[y/x^2, {x, 1, Infinity}, {y, 1, 4}]
```

Out[ ]=
$$\frac{15}{2}$$

On peut également utiliser la version numérique de la fonction Integrate. Celle-ci est donnée par la fonction NIntegrate et donne une valeur approchée de l'intégrale demandée, ce qui est très utile lorsque l'intégrale à calculer n'a pas de forme exacte simple. Comparez les deux exemples suivants.

In[ ]:=
```
Integrate[Sin[Sin[x]], {x, 0, 2}]
NIntegrate[Sin[Sin[x]], {x, 0, 2}]
```

Out[ ]=
$$\int_0^2 \mathrm{Sin}[\mathrm{Sin}[x]]\, dx$$

Out[ ]=
```
1.24706
```

HoldForm

Si on souhaite voir l'écriture habituelle d'une somme, d'un produit, d'une dérivée ou d'une primitive, on utilise la fonction HoldForm.

In[ ]:=

```
HoldForm[Sum[1 / x ^ 2, {x, 1, n}]]
HoldForm[Product[1 / x ^ 2, {x, 1, n}]]
HoldForm[D[1 / x ^ 2, x]]
HoldForm[Integrate[1 / x ^ 2, x]]
```

Out[ ]= $\sum_{x=1}^{n} \frac{1}{x^2}$

Out[ ]= $\prod_{x=1}^{n} \frac{1}{x^2}$

Out[ ]= $\partial_x \frac{1}{x^2}$

Out[ ]= $\int \frac{1}{x^2} dx$

Équations différentielles

Une équation différentielle se résout à l'aide de la fonction DSolve. Une fonction inconnue se représente sous la forme y[x], sa dérivée première par y'[x], sa dérivée seconde par y''[x], et ainsi de suite. On a par exemple :

In[ ]:=

```
DSolve[y '[x] + y[x] == 1, y[x], x]
DSolve[y '[x] + y[x] == 1, y, x]
```

Out[ ]= {{y[x] → $1 + e^{-x} c_1$}}

Out[ ]= {{y → Function[{x}, $1 + e^{-x} c_1$]}}

Notez la différence entre les deux sorties. Dans le premier cas, nous avons demandé une solution pour y[x] et Mathematica nous a rendu une règle qui à y[x] associe une expression de la variable symbolique x. Dans le deuxième cas, nous avons demandé une solution pour y et Mathematica nous a rendu une règle qui à y associe une fonction.

En effet, il y a différents moyens de définir des fonctions dans Mathematica, et nous en rencontrons ici une deuxième (souvenons-nous que la première a été introduite dans la section "Définir une fonction"). Ainsi, on a

In[ ]:=
```
Function[{x}, 1 + e^-x C[1]][a]
Function[{x}, 1 + e^-x C[1]][2]
```

Out[ ]= $1 + e^{-a} c_1$

Out[ ]= $1 + \frac{c_1}{e^2}$

Il revient au même d'écrire

In[ ]:=
```
h[x_] := 1 + e^-x C[1]
h[a]
h[a] == Function[{x}, 1 + e^-x C[1]][a]
```

Out[ ]= $1 + e^{-a} c_1$

Out[ ]= True

Voici un deuxième exemple de résolution d'une équation différentielle, cette fois-ci d'ordre 2.

In[ ]:=
```
DSolve[y''[x] + 7 y'[x] == 1, y[x], x]
```

Out[ ]= $\left\{\left\{y[x] \to \frac{x}{7} - \frac{1}{7} e^{-7x} c_1 + c_2\right\}\right\}$

On peut aussi résoudre des équations différentielles qui ne sont pas à coefficients constants, comme par exemple

In[ ]:=
```
DSolve[y[x] × y'[x] == 1, y[x], x]
```

Out[ ]= $\left\{\left\{y[x] \to -\sqrt{2}\ \sqrt{x + c_1}\right\}, \left\{y[x] \to \sqrt{2}\ \sqrt{x + c_1}\right\}\right\}$

On peut ajouter des conditions initiales, ou d'autres types de contraintes. Cela se traduit, comme pour Solve, par un système d'équations. Remarquez cependant que la syntaxe n'est pas la même pour Solve et DSolve.

In[ ]:=
```
DSolve[{y''[x] + 7 y'[x] == 1, y[0] == 4}, y[x], x]
DSolve[{y''[x] + 7 y'[x] == 1, y[0] == 4, y'[8] == 0}, y[x], x]
DSolve[{y[x] × y'[x] == 1, y[0] == 0}, y[x], x]
```

Out[ ]= $\left\{\left\{y[x] \to \frac{1}{7} e^{-7x} \left(28 e^{7x} + e^{7x} x - c_1 + e^{7x} c_1\right)\right\}\right\}$

Out[ ]= $\left\{\left\{y[x] \to \frac{1}{49} e^{-7x} \left(e^{56} + 196 e^{7x} - e^{56+7x} + 7 e^{7x} x\right)\right\}\right\}$

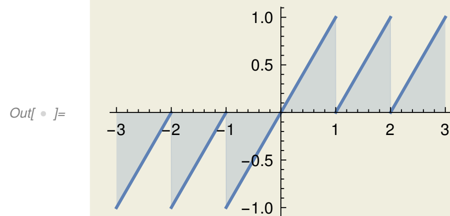
Out[ ]= $\left\{\left\{y[x] \to -\sqrt{2}\sqrt{x}\right\}, \left\{y[x] \to \sqrt{2}\sqrt{x}\right\}\right\}$

Représentations graphiques avec *Mathematica*

C'est la fonction Plot qui va nous permettre de représenter graphiquement des objets mathéma-tiques. Représentons les fonctions FractionalPart, IntegerPart, Floor et Ceiling. Notez la syntaxe : Plot prend comme premier argument la fonction à représenter, ensuite l'intervalle de variation de l'argument de la fonction, et enfin, on peut ajouter des options graphiques (couleurs, remplissages, zoom, etc.)

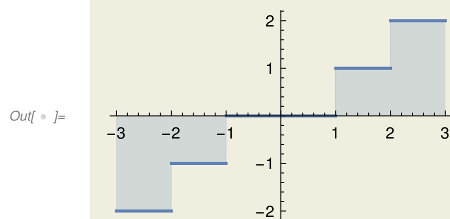
In[ ]:=
```
Plot[FractionalPart[x], {x, -3, 3}, Filling → Axis, ImageSize → Small]
```

Out[ ]=



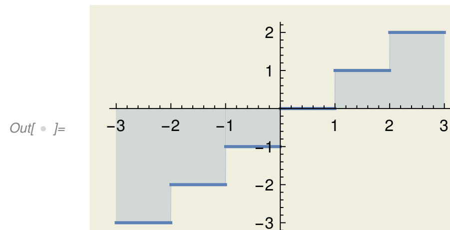
In[ ]:=
```
Plot[IntegerPart[x], {x, -3, 3}, Filling → Axis, ImageSize → Small]
```

Out[ ]=



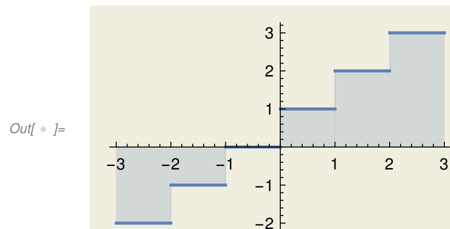
In[ ]:=
```
Plot[Floor[x], {x, -3, 3}, Filling → Axis, ImageSize → Small]
```

Out[ ]=



In[ ]:=
```
Plot[Ceiling[x], {x, -3, 3}, Filling → Axis, ImageSize → Small]
```

Out[ ]=



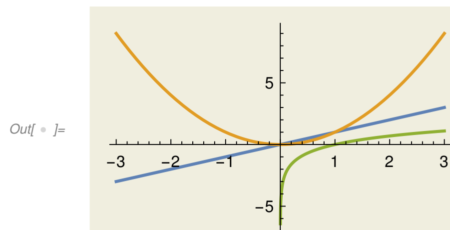
Pour chaque graphique de ce texte, j'ai ajouté l'option ImageSize→ Small, car le rendu après conversion en fichier .pdf était bien trop grand sans cette option.

Tracer plusieurs fonctions dans un graphique

On peut représenter plusieurs fonctions sur le même graphique. Elles seront automatiquement chacune de couleur différente.

In[ ]:=
```
Plot[{x, x^2, Log[x]}, {x, -3, 3}, ImageSize → Small]
```

Out[ ]=



Représenter une surface

On peut également faire des représentations en trois dimensions, et ce, grâce à la fonction Plot3D. Pour représenter une fonction de R^2 dans R, il faut spécifier les intervalles de variations de chacune des arguments de la fonction.

Voici une selle de cheval (ou plutôt un paraboloïde hyperbolique) :

In[∘]:=
```
? Plot3D
```

Out[∘]=

Symbol

Plot3D [f, {x, x_{min}, x_{max}}, {y, y_{min}, y_{max}}] generates
a three-dimensional plot of f as a function of x and y.

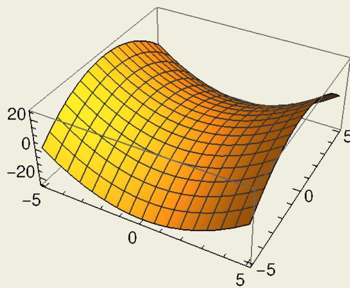
Plot3D [{f_1, f_2, …}, {x, x_{min}, x_{max}}, {y, y_{min}, y_{max}}] plots several functions .

Plot3D [{…, w[f_i], …}, …] plots f_i with features defined by the symbolic wrapper w.

Plot3D […, {x, y} ∈ reg] takes variables {x, y} to be in the geometric region reg.

In[∘]:=
```
Plot3D[x^2 - y^2, {x, -5, 5}, {y, -5, 5}, ImageSize → Small]
```

Out[∘]=

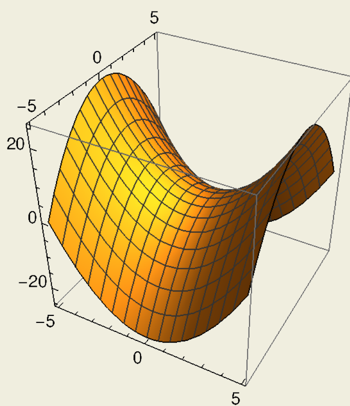


Il est possible également de faire bouger le graphique, simplement en cliquant dessus et en déplaçant la souris.

Voici le même graphique avec des unités égales sur chaque axes (grâce à l'option BoxRatios→{1,1,1}), ce qui donne une image qui correspond mieux à ce qu'on attend. En fait, grâce aux paramètres de BoxRatios, vous pouvez moduler les unités des axes à votre guise.

In[∘]:=
```
Plot3D[x^2 - y^2, {x, -5, 5}, {y, -5, 5}, BoxRatios → {1, 1, 1}, ImageSize → Small]
```

Out[∘]=



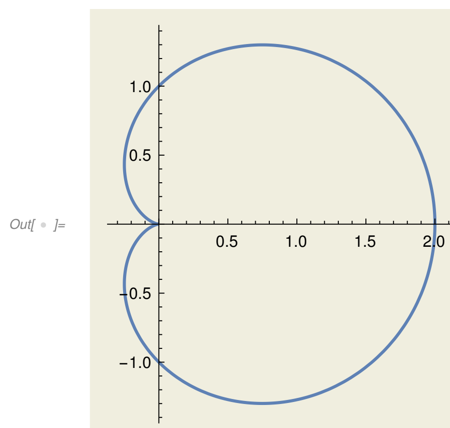
Représenter une courbe planaire

Pour représenter une fonction de R dans R^2, on utilise la fonction ParametricPlot. Voici la car-

dioïde, représentée à l'aide d'équations paramétriques :

In[ ]:=
```
ParametricPlot [{Cos[u] (1 + Cos[u]), Sin[u] (1 + Cos[u])}, {u, 0, 2 Pi}, ImageSize → Small]
```

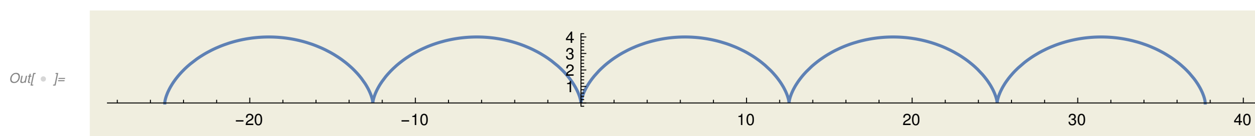
Out[ ]=



Voici la cycloïde. Avec l'option ImageSize→Full, on obtient une taille de graphique correspondant à la largeur de la page.

In[ ]:=
```
ParametricPlot [{2 (u - Sin[u]), 2 (1 - Cos[u])}, {u, -4 Pi , 6 Pi}, ImageSize → Full]
```

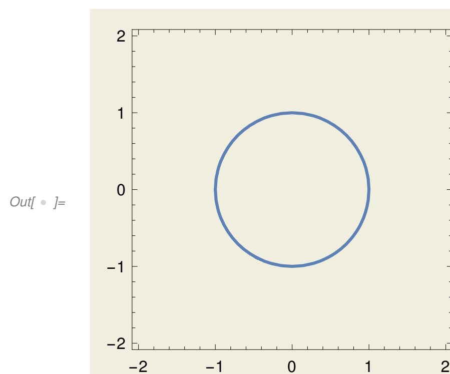
Out[ ]=



On peut également représenter une courbe planaire grâce à son équation cartésienne. Pour ce faire, on utilise ContourPlot. Cette fonction demande impérativement de spécifier les intervalles de variations des coordonnées. Il est à noter que ContourPlot est une la version actualisée de Implicit - Plot, proposée par les anciennes versions de Mathematica.

In[ ]:=
```
ContourPlot [x ^ 2 + y ^ 2 == 1, {x, -2, +2}, {y, -2, +2}, ImageSize → Small]
```

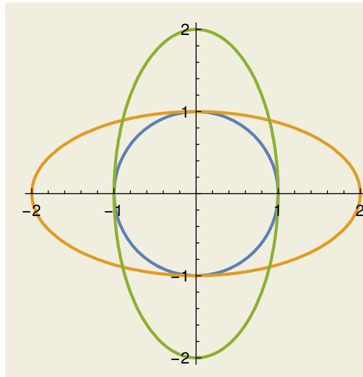
Out[ ]=



On peut demander à voir les axes et/ou enlever le cadre. On peut également représenter plusieurs courbes sur le même graphique.

In[ ]:=
```
ContourPlot[{x^2 + y^2 == 1, x^2/4 + y^2 == 1, x^2 + y^2/4 == 1},
 {x, -2, +2}, {y, -2, +2}, Axes -> True, Frame -> False, ImageSize → Small]
```

Out[ ]=

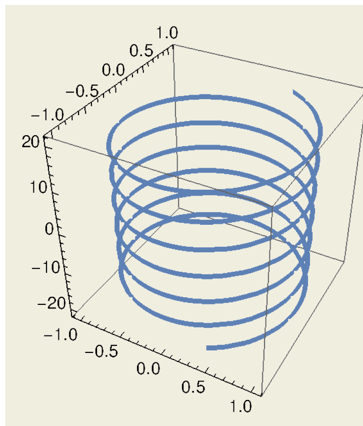


Représenter une courbe dans l'espace

Pour représenter une fonction de R dans R^3, on utilise la fonction ParametricPlot3D. Voici une hélice circulaire à pas constant :

In[ ]:=
```
ParametricPlot3D[{Cos[u], Sin[u], u},
 {u, -20, 20}, BoxRatios → {1, 1, 1}, ImageSize → Small]
```

Out[ ]=

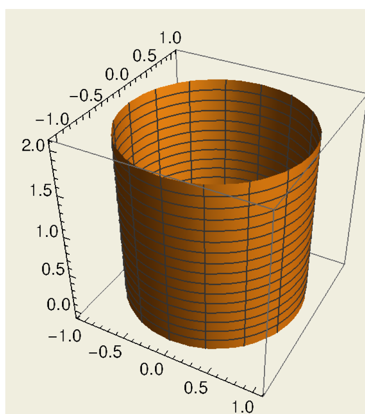


Représenter une surface donnée par un paramétrage

Voici un cylindre, représenté à l'aide de ParametricPlot3D. Il s'agit d'une fonction de R^2 dans R^3.

In[ ]:=
```
ParametricPlot3D [{Cos[u], Sin[u], v}, {u, 0, 2 Pi}, {v, 0, 2}, ImageSize → Small]
```

Out[ ]=

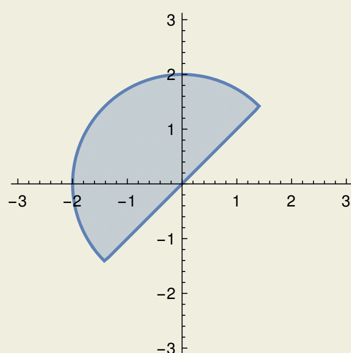


Représenter une région

Grâce à la fonction RegionPlot, on peut représenter une région du plan satisfaisant une liste d'inégalités.

In[ ]:=
```
RegionPlot [x < y && x ^ 2 + y ^ 2 < 4, {x, -3, 3},
 {y, -3, 3}, Axes → True, Frame → False, ImageSize → Small]
```

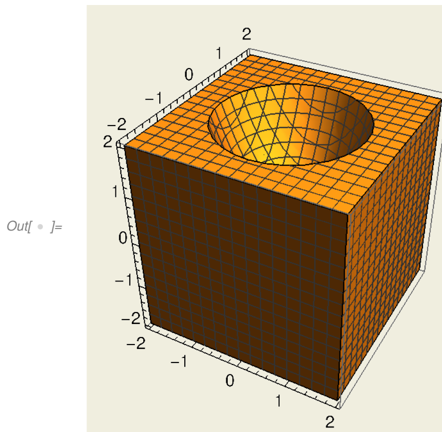
Out[ ]=



Grâce à la fonction RegionPlot3D, on peut représenter une région de l'espace satisfaisant une liste d'inégalités. On peut facilement faire un coquetier :

In[ ]:=
```
RegionPlot3D[x^2 + y^2 > z, {x, -2, 2}, {y, -2, 2}, {z, -2, 2}, ImageSize → Small]
```

Out[ ]=



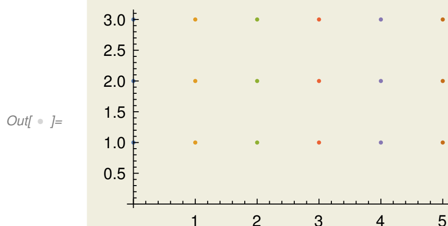
Représenter des points d'une liste

On peut également représenter des points de R^2 ou de R^3 dont les coordonnées sont le contenu d'une liste :

In[ ]:=
```
t = Table[{m, n}, {m, 0, 5}, {n, 1, 3}]
ListPlot[t, ImageSize → Small]
Flatten[t, 1]
ListPlot[Flatten[t, 1], ImageSize → Small]
```

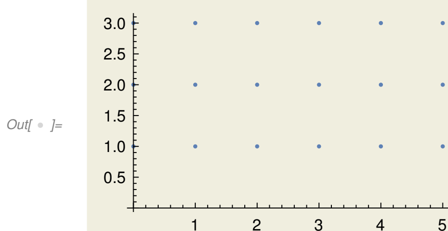
Out[ ]=
```
{{{0, 1}, {0, 2}, {0, 3}}, {{1, 1}, {1, 2}, {1, 3}}, {{2, 1}, {2, 2}, {2, 3}},
 {{3, 1}, {3, 2}, {3, 3}}, {{4, 1}, {4, 2}, {4, 3}}, {{5, 1}, {5, 2}, {5, 3}}}
```

Out[ ]=



Out[ ]=
```
{{0, 1}, {0, 2}, {0, 3}, {1, 1}, {1, 2}, {1, 3}, {2, 1}, {2, 2},
 {2, 3}, {3, 1}, {3, 2}, {3, 3}, {4, 1}, {4, 2}, {4, 3}, {5, 1}, {5, 2}, {5, 3}}
```

Out[ ]=



La liste t est une liste de listes, chacune contenant des couples de coordonnées. Pour chaque sous-liste, Mathematica utilise une couleur différente. La liste Flatten[t,1] est une liste de couples de

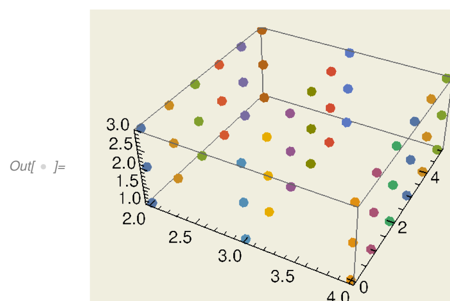
coordonnées. Mathematica n'utilise donc qu'une seule couleur pour représenter ces points.

In[∘]:=
```
t = Table[{l, m, n}, {l, 2, 4}, {m, 0, 5}, {n, 1, 3}]
Flatten[t, 1]
ListPointPlot3D [Flatten[t, 1], ImageSize → Small]
Flatten[t, 2]
ListPointPlot3D [Flatten[t, 2], ImageSize → Small]
```

Out[∘]=
```
{{{{2, 0, 1}, {2, 0, 2}, {2, 0, 3}}, {{2, 1, 1}, {2, 1, 2}, {2, 1, 3}},
   {{2, 2, 1}, {2, 2, 2}, {2, 2, 3}}, {{2, 3, 1}, {2, 3, 2}, {2, 3, 3}},
   {{2, 4, 1}, {2, 4, 2}, {2, 4, 3}}, {{2, 5, 1}, {2, 5, 2}, {2, 5, 3}}},
  {{{3, 0, 1}, {3, 0, 2}, {3, 0, 3}}, {{3, 1, 1}, {3, 1, 2}, {3, 1, 3}},
   {{3, 2, 1}, {3, 2, 2}, {3, 2, 3}}, {{3, 3, 1}, {3, 3, 2}, {3, 3, 3}},
   {{3, 4, 1}, {3, 4, 2}, {3, 4, 3}}, {{3, 5, 1}, {3, 5, 2}, {3, 5, 3}}},
  {{{4, 0, 1}, {4, 0, 2}, {4, 0, 3}}, {{4, 1, 1}, {4, 1, 2}, {4, 1, 3}},
   {{4, 2, 1}, {4, 2, 2}, {4, 2, 3}}, {{4, 3, 1}, {4, 3, 2}, {4, 3, 3}},
   {{4, 4, 1}, {4, 4, 2}, {4, 4, 3}}, {{4, 5, 1}, {4, 5, 2}, {4, 5, 3}}}}
```

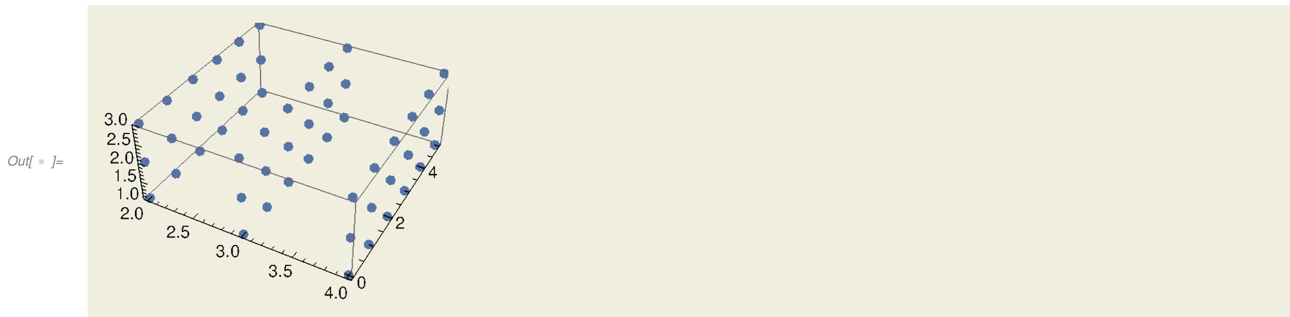
Out[∘]=
```
{{{2, 0, 1}, {2, 0, 2}, {2, 0, 3}}, {{2, 1, 1}, {2, 1, 2}, {2, 1, 3}},
  {{2, 2, 1}, {2, 2, 2}, {2, 2, 3}}, {{2, 3, 1}, {2, 3, 2}, {2, 3, 3}},
  {{2, 4, 1}, {2, 4, 2}, {2, 4, 3}}, {{2, 5, 1}, {2, 5, 2}, {2, 5, 3}},
  {{3, 0, 1}, {3, 0, 2}, {3, 0, 3}}, {{3, 1, 1}, {3, 1, 2}, {3, 1, 3}},
  {{3, 2, 1}, {3, 2, 2}, {3, 2, 3}}, {{3, 3, 1}, {3, 3, 2}, {3, 3, 3}},
  {{3, 4, 1}, {3, 4, 2}, {3, 4, 3}}, {{3, 5, 1}, {3, 5, 2}, {3, 5, 3}},
  {{4, 0, 1}, {4, 0, 2}, {4, 0, 3}}, {{4, 1, 1}, {4, 1, 2}, {4, 1, 3}},
  {{4, 2, 1}, {4, 2, 2}, {4, 2, 3}}, {{4, 3, 1}, {4, 3, 2}, {4, 3, 3}},
  {{4, 4, 1}, {4, 4, 2}, {4, 4, 3}}, {{4, 5, 1}, {4, 5, 2}, {4, 5, 3}}}
```

Out[∘]=



Out[∘]=
```
{{2, 0, 1}, {2, 0, 2}, {2, 0, 3}, {2, 1, 1}, {2, 1, 2}, {2, 1, 3}, {2, 2, 1}, {2, 2, 2},
  {2, 2, 3}, {2, 3, 1}, {2, 3, 2}, {2, 3, 3}, {2, 4, 1}, {2, 4, 2}, {2, 4, 3},
  {2, 5, 1}, {2, 5, 2}, {2, 5, 3}, {3, 0, 1}, {3, 0, 2}, {3, 0, 3}, {3, 1, 1},
  {3, 1, 2}, {3, 1, 3}, {3, 2, 1}, {3, 2, 2}, {3, 2, 3}, {3, 3, 1}, {3, 3, 2}, {3, 3, 3},
  {3, 4, 1}, {3, 4, 2}, {3, 4, 3}, {3, 5, 1}, {3, 5, 2}, {3, 5, 3}, {4, 0, 1}, {4, 0, 2},
  {4, 0, 3}, {4, 1, 1}, {4, 1, 2}, {4, 1, 3}, {4, 2, 1}, {4, 2, 2}, {4, 2, 3}, {4, 3, 1},
  {4, 3, 2}, {4, 3, 3}, {4, 4, 1}, {4, 4, 2}, {4, 4, 3}, {4, 5, 1}, {4, 5, 2}, {4, 5, 3}}
```

Out[ ]=



Ici, t est une liste de listes de listes. Pour représenter les points souhaités, on doit descendre d'un niveau (auquel cas, on aura plusieurs couleurs) ou de deux (auquel cas, on aura une seule couleur).

Enfin, mentionnons que la fonction ListPlot3D, appliquée à une liste de triplets ou à une liste de listes de triplets, rend une surface.

In[ ]:=

```
? ListPlot3D
```

Out[ ]=

Symbol

ListPlot3D [{{z_{11}, ..., z_{1n}}, ..., {z_{m1}, ..., z_{mn}}}] generates a three-dimensional plot of a surface representing an array of height values z_{ij}.

ListPlot3D [{{x_1, y_1, z_1}, ..., {x_k, y_k, z_k}}] generates a plot of the surface with heights z_i at positions {x_i, y_i}.

ListPlot3D [{$data_1$, $data_2$, ...}] plots the surfaces corresponding to each of the $data_i$.

Cours 4 - SymPy

February 28, 2022

Nous introduisons ici très brièvement le logiciel SymPy et insistons principalement sur les similarités et les différences avec Mathematica, que nous avons présenté plus longuement dans les trois premiers cours. Pour plus de détails, je vous renvoie vers la documentation de SymPy http://docs.sympy.org/dev/index.html. Vous y trouverez des tutoriels d'introduction et la partie gotchas est à consulter absolument.

Notez que j'ai utilisé ici directement le traitement de texte de Jupyter pour écrire ces notes, tout comme je l'avais fait plus haut pour l'introduction à Mathematica. J'ai ensuite converti le notebook de Jupyter en un fichier .pdf avec les liens "Download as" puis "PDF via LaTeX (.pdf)". Ce moyen de créer un pdf modifie l'aspect du notebook, contrairement aux options de création de pdf de Mathematica. Il y a probablement un moyen de garder l'aspect original du notebook Jupyter également, mais pas avec les options de base du logiciel.

1 Calculatrice avec Python

L'addition et la soustraction s'obtiennent avec + et -.

[2]:
```
2-3
```

[2]: -1

La multiplication s'obtient avec *, qu'il faut impérativement écrire, un simple espace ne fonctionnant pas.

[5]:
```
2*3
```

[5]: 6

L'exponentiation se réalise avec **.

[6]:
```
2**3
```

[6]: 8

En Python, la division de 2 par 3 ne rend pas un rationnel mais un flottant.

[7]:
```
2/3
```

[7]: 0.6666666666666666

2 Calcul formel avec SymPy

Le logiciel SymPy n'est en fait rien de plus qu'une librairie Python. Pour créer un nombre rationnel, on utilise la fonction Rational de SymPy, qu'on doit d'abord importer.

[8]:
```
from sympy import Rational
```

[9]:
```
from sympy import Rational
Rational(2,3)
```

[9]: 2/3

On peut ensuite effectuer des calculs exacts avec des rationnels.

[10]:
```
Rational(2,3)+Rational(7/4)+2
```

[10]: 53/12

[12]:
```
from sympy import Integer
Integer(2)/Integer(3)
```

[12]: 2/3

[13]:
```
Integer(2)/3
```

[13]: 2/3

[14]:
```
2/Integer(3)
```

[14]: 2/3

[15]:
```
2/3
```

[15]: 0.6666666666666666

De la même façon, pour faire des calculs avec les nombres complexes, on doit importer le symbole I de SymPy.

[16]:
```
from sympy import I
2*I+4
```

[16]: 4 + 2*I

3 Importer des fonctions et des symboles depuis des librairies

Nous venons déjà de rencontrer une différence fondamentale entre les notebooks de Jupyter et de Mathematica. Dans Mathematica, on n'a pas besoin d'importer les fonctions. Ce n'est pas le cas

dans Jupyter. Par défaut, le notebook de Jupyter ne connaît que le langage de programmation Python. Prenons un exemple éclairant.

Par défaut, la fonction sin n'est pas connue dans Jupyter.

[17]:
```
sin(4)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-17-11d024bfb023> in <module>()
----> 1 sin(4)

NameError: name 'sin' is not defined
```

Pour calculer sin(4), on doit importer la fonction sin. Selon la libraire depuis laquelle on choisit d'importer sin, le comportement sera différent.

Par exemple, si on importe sin depuis la librairie math, on obtient

[18]:
```
from math import sin
sin(4)
```

[18]: -0.7568024953079282

Observons que la fonction sin de math prend comme argument un flottant et rend un flottant.

[19]:
```
sin(3.141592)
```

[19]: 6.535897930762419e-07

Si on souhaite réaliser du calcul formel avec sin, on importe sin depuis SymPy.

[20]:
```
from sympy import sin
sin(4)
```

[20]: sin(4)

Pour obtenir une approximation numérique en SymPy, on utilise la fonction N.

[21]:
```
from sympy import N
N(sin(4))
```

[21]: -0.756802495307928

Comme dans Mathematica, si on entre un flottant comme argument, la fonction sin de SymPy retourne un flottant. La philosophie est la même (comme dans tout logiciel de calcul formel) : SymPy rend une valeur exacte, sauf si on lui dit explicitement de ne pas le faire, par exemple dans ce cas précis, en entrant un nombre à virgule.

[22]:
```
sin(3.141592)
```

[22]:
```
6.53589793076242e-7
```

Comme Mathematica, SymPy sait que sin(pi)=0. Mais, à nouveau, il faut d'abord importer le symbole pi de SymPy.

[23]:
```
pi
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-23-68f7b1e53523> in <module>()
----> 1 pi

NameError: name 'pi' is not defined
```

[24]:
```
from sympy import pi
from sympy import sin
sin(pi)
```

[24]:
```
0
```

Attention : le fait qu'on puisse utiliser dans Jupyter deux fonctions différentes portant le même nom peut être problématique. Jupyter utilisera toujours la dernière fonction importée. Si on réimporte sin de la librairie math, on n'obtient plus que sin(pi)=0, mais une valeur approchée de 0.

[25]:
```
from math import sin
sin(pi)
```

[25]:
```
1.2246467991473532e-16
```

4 Parenthèses et crochets

Tout comme dans Mathematica, les listes sont beaucoup utilisées dans SymPy. Dans SymPy, les arguments des fonctions s'écrivent entre parenthèses (comme nous l'avons vu ci-dessus pour sin(4)) et les listes s'écrivent avec des crochets. Comme dans Mathematica, les listes peuvent contenir toutes sortes d'objets, et peuvent être imbriquées les unes dans les autres.

[26]:
```
[1,2,y,7,pi]
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-26-eb2d1d3c2951> in <module>()
----> 1 [1,2,y,7,pi]

NameError: name 'y' is not defined
```

[27]:
```
from sympy.abc import y
[1,2,y,7,pi]
```

[27]: [1, 2, y, 7, pi]

[29]:
```
[1,[2,[3],y],7,pi]
```

[29]: [1, [2, [3], y], 7, pi]

[28]:
```
liste=[1,[2,[3],y],7,pi]
```

On a accès à un élément d'une liste avec les crochets []. Attention que dans Jupyter, les listes sont indicées à partir de 0, alors que dans Mathematica, les listes sont indicées à partir de 1.

[30]:
```
liste
```

[30]: [1, [2, [3], y], 7, pi]

[31]:
```
liste[0]
```

[31]: 1

[32]:
```
liste[1]
```

[32]: [2, [3], y]

[33]:
```
liste[2]
```

[33]: 7

[34]:
```
liste[3]
```

[34]: pi

Remarquons au passage que lorsqu'on entre plusieurs commandes successives au sein d'une même cellule, seule la sortie de la dernière est contenue dans le Out correspondant.

[35]:
```
liste[1]
liste[2]
```

[35]: 7

On a également la fonction flatten (à importer depuis la sous-libraire sympy.utilities.iterables), qui rend une liste simple à partir de listes imbriquées. Comme dans Mathematica, on peut ajouter en option le niveau de parenthésage (ici, de crochets) à enlever de la liste.

[37]:
```
from sympy.utilities.iterables import flatten
flatten(liste)
```

[37]: [1, 2, 3, y, 7, pi]

5 Variables et symboles

Une autre différence fondamentale entre SymPy et Mathematica est le traitement des variables. Dans Mathematica, il n'y a aucune différence entre variable et symbole. En fait, toute les variables dans Mathematica sont des variables symboliques. Dans Python, et donc dans SymPy, on distingue variable (de programmation) et variable symbolique (ou symbole). À nouveau, considérons un exemple, certes simple, mais éclairant.

Par défaut, x n'est pas défini.

[38]:
```
x
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-38-401b30e3b8b5> in <module>()
----> 1 x

NameError: name 'x' is not defined
```

Comme dans Mathematica, on peut affecter des valeurs dans des variables avec les signe =.

[41]:
```
x=2
x
```

[41]: 2

Une façon de définir le symbole x est de l'importer de la sous-librairie sympy.abc.

[42]:
```
from sympy.abc import x
```

Voyez la différence à présent.

[43]:
```
x
```

[43]: x

[44]:
```
y
```

[44]: y

[45]:
```
z
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-45-a8a78d0ff555> in <module>()
```

```
----> 1 z

NameError: name 'z' is not defined
```

On peut maintenant faire du calcul avec des polynômes.

[46]:
```
x**2-1
```

[46]: x**2 - 1

Les fonctions relatives aux polynômes vues dans Mathematica ont toutes leur analogue dans SymPy. Par exemple, pour factoriser un polynôme, on utilise la fonction factor de SymPy.

[47]:
```
from sympy import factor
factor(x**2-1)
```

[47]: (x - 1)*(x + 1)

Observons le comportement de SymPy sur le code suivant :

[48]:
```
expr=x+1
expr
```

[48]: x + 1

[50]:
```
x=100
x
```

[50]: 100

[51]:
```
expr
```

[51]: x + 1

Que s'est-il passé ? Nous avons stocké dans une variable Python, appelée expr, l'expression symbolique x+1 de SymPy. Quand on tape expr, c'est donc x+1 qui est rendu. Ensuite, nous avons affecté la valeur 100 dans une variable Python, appelée x. Ce qui est important à comprendre ici est que la variable Python appelée x et le symbole x de SymPy sont des objets différents et indépendants. Ainsi, lorsqu'on tape x, on fait appel à la variable Python x et c'est 100 qui est rendu. La variable expr, elle, stocke toujours l'expression symbolique x+1. Le comportement de ce même code dans Mathematica est différent : un effet de bord sera produit sur expr par une affectation de x.

Par contre, si on recommence une affectation de expr avec du x, c'est bien la dernière affectation de x qui sera considérée et non le symbole x. Une variable contient donc toujours la valeur de sa dernière affectation (pas d'effet de bord).

[52]:
```
expr=x+1
expr
```

[52]: 101

Il y a d'autres façons de définir des symboles. Par exemple, on peut importer la fonction Symbol de SymPy. Cette fonction transforme une suite de caractère en symbole.

[53]:
```
a
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-53-60b725f10c9c> in <module>()
----> 1 a

NameError: name 'a' is not defined
```

[54]:
```
from sympy import Symbol
a=Symbol('a')
a
```

[54]: a

Nous avons affecté la variable a du symbole a. En fait, rien ne nous oblige à donner le même nom à la variable qui stocke un symbole. La variable a peut stocker le symbole b et la variable b stocker le symbole a sans que cela ne pose de problème. Attention tout de même à ce genre de pratique, qui peut rendre un code bien difficile à lire…

[62]:
```
a=Symbol('bernard')
b=Symbol('a')
```

[63]:
```
a
```

[63]: bernard

[57]:
```
b
```

[57]: a

Comme dans Mathematica, a==b teste l'égalité des deux objets a et b. Comme ici, a stocke b et b stocke a, le test rend le booléen False.

[37]:
```
a==b
```

[37]: False

Enfin, on peut importer plusieurs symboles d'un coup depuis la sous-librairie sympy.abc. Celle-ci contient les symboles classiques, ainsi que des lettres grecques écrites in extenso.

[58]:
```
from sympy.abc import a,b,c,r,epsilon
```

[61]:
```
epsilon**5+3
```

[61]: epsilon**5 + 3

6 Accéder à la documentation d'une fonction

Pour accéder à la documentation d'une fonction de SymPy, on place un point d'interrogation avant ou après la fonction. La documentation apparaît alors en bas du notebook. On peut également ouvrir la page de documentation correspondante dans un nouvel onglet ou fenêtre de son navigateur (simplement en cliquant sur le lien en haut à droite de l'encadré).

[64]:
```
from sympy import sin
?sin
```

Comme SymPy est un logiciel libre, on peut accéder au code source de la fonction. Pour ce faire, on tape un double point d'interrogation avant ou après la fonction.

[65]:
```
??sin
```

On peut également importer toutes les fonctions de SymPy d'un coup grâce à la commande suivante. Mais attention alors aux conflits possibles de librairies !

[66]:
```
from sympy import *
```

7 Fonctions et méthodes

Python est un langage orienté-objet. Cela implique que dans SymPy, on peut utiliser deux syntaxes différentes, pour calculer la valeur d'une fonction en un argument. Par exemple, on peut écrire

[67]:
```
from sympy.abc import x
expr=x**2-1
expr.factor()
```

[67]: (x - 1)*(x + 1)

[68]:
```
expr.factor()
```

[68]: (x - 1)*(x + 1)

Ici, on a appliqué la méthode factor à l'objet expr. Dans SymPy, après un objet suivi d'un point, on peut obtenir la liste des méthodes applicables à cet objet grâce à la touche "tabulation". Cet outil est extrêmement pratique, puisqu'il vous permet de retrouver les noms des méthodes dont vous avez besoin. L'avantage des méthodes par rapport aux fonctions est qu'on n'a pas besoin de les importer pour les utiliser. Par exemple, on a :

[44]:
```
expr.diff()
```

[44]: 2*x

Les fonctions et les méthodes correspondantes portent généralement le même nom. Ce n'est pas toujours le cas. Par exemple, pour obtenir une approximation numérique dans SymPy, on a vu plus haut qu'on pouvait utiliser la fonction N. On peut également utiliser les méthodes evalf ou n. Mais la méthode N n'existe pas, pas plus d'ailleurs que les fonctions n et evalf.

[69]:
```
sin(4).evalf()
```

[69]: -0.756802495307928

[72]:
```
sin(4).n()
```

[72]: -0.756802495307928

8 Faire appel à une sortie précédente

Pour faire appel à la dernière sortie, on utilise "souligné", pour l'avant-dernière sortie, on utilise "double souligné", pour l'antépénultième sortie, on utilise "triple souligné", et ainsi de suite.

[75]:
```
__
```

[75]: -0.756802495307928

[48]:
```
__
```

[48]: -0.756802495307928

Pour faire appel à la sortie n, on tape Out[n].

[76]:
```
Out[18]
```

[76]: -0.7568024953079282

9 Substitution dans une expression

Pour substituer un symbole dans une expression symbolique, on utilise la syntaxe suivante.

[77]:
```
expr
```

[77]: x**2 - 1

[78]:
```
from sympy.abc import a
expr.subs(x,2*a)
```

[78]: 4*a**2 - 1

On peut également substituer plusieurs symboles à la fois en utilisant la syntaxe suivante.

[79]:
```
from sympy.abc import a,b,c
expr=a**2+b**2+a*b*c
expr
```

[79]:
```
a**2 + a*b*c + b**2
```

[80]:
```
expr.subs({a:1,b:x,c:pi})
```

[80]:
```
x**2 + pi*x + 1
```

10 Dictionnaires

Nous venons de rencontrer un nouveau type d'objet de python: un dictionnaire. Un dictionnaire est comme une liste, mais on a accès à ses éléments non plus par un indice de position, mais par un code (ou une clé). Ceci peut se révéler pratique dans bien des situations.

[81]:
```
dico={a:1,b:x,c:pi}
dico
```

[81]:
```
{a: 1, b: x, c: pi}
```

Ici les clés sont simplement a,b,c. On accède à l'élément de "dico" relatif à la clé "yayaya" avec "dico[yayaya]".

[82]:
```
dico[a]
```

[82]:
```
1
```

[56]:
```
dico[c]
```

[56]:
```
pi
```

[83]:
```
belgique, bruxelles, france, paris, italie, rome = symbols('belgique bruxelles␣
↪france paris italie rome')
capitale={belgique:bruxelles,france:paris,italie:rome}
```

[85]:
```
espagne
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-85-8dbc559b5ad2> in <module>()
----> 1 espagne

NameError: name 'espagne' is not defined
```

[86]:
```
capitale[france]
```

[86]: paris

11 Résoudre des équations

Le code x==3 teste l'égalité des objets x et 3. Pour définir une équation, on utilise la fonction Eq.

[87]:
```
x=Symbol('x')
x==3
```

[87]: False

[88]:
```
from sympy import Eq
Eq(x,3)
```

[88]: Eq(x, 3)

Pour résoudre une équation, on combine les fonctions solve et Eq.

[89]:
```
from sympy import solve
solve(Eq(x,3),x)
```

[89]: [3]

12 Analyse avec Sympy

[96]:
```
from sympy import *
x,y,z=symbols('x y z')
init_printing(use_unicode=True)
diff(exp(x**2),x,2)
```

[96]:

$$2\left(2x^2+1\right)e^{x^2}$$

[45]:
```
expr=exp(x**2)
```

[95]:
```
expr.
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-95-db19c4c14853> in <module>()
----> 1 expr.diff(x,2)

AttributeError: module 'sympy.core.expr' has no attribute 'diff'
```

[97]:
```
?series
```

[ ]:
```

```

Cours 5 - Introduction a la programmation avec Python

February 27, 2023

1 Exemple introductif : la conjecture de Goldbach

Une conjecture est un énoncé que la plupart des mathématiciens supposent vrai, mais qui n'a pas été démontré. Une conjecture n'est donc pas un théorème ! La conjecture de Goldbach est l'une des plus ancienne conjecture de théorie des nombres. Son énoncé est le suivant : tout entier pair strictement supérieur à 2 est la somme de deux nombres premiers. Rappelons qu'un nombre premier est un nombre naturel qui a exactement 2 diviseurs. Ainsi, les premiers nombres premiers sont 2, 3, 5, 7, 11, 13, 17, 19, 23…

Pour nous familiariser avec la conjecture de Goldbach, testons-là pour les premières entiers pairs supérieur à 2. On a 4=2+2, 6=3+3, 8=3+5, 10=3+7, 12=5+7, 14=7+7, 16=5+11, 18=7+11, 20=3+17, 22=3+19, 24=5+19… Afin de vérifier cette conjecture pour un grand nombre de valeurs, nous allons définir une fonction Goldbach qui à un entier n rend True ou False selon que n est la somme de deux nombres premiers.

Tout d'abord, nous définissons une fonction qui teste si un naturel est un nombre premier, c'est-à-dire qui à un naturel n associe true si n est un nombre premier et false sinon. Rappelons que Python calcule le quotient et le reste d'une division euclidienne avec les commandes // et %.

[1]:
```
17//3
```

[1]: 5

[2]:
```
17%3
```

[2]: 2

[3]:
```
def isPrime(n):
    if n==0 or n==1:
        return False
    elif n==2:
        return True
    else:
        for x in range(2,n//2+1):
            if n%x==0:
                return False
    return True
```

[4]:
```
[[n,isPrime(n)] for n in range(2,20)]
```

```
[4]: [[2, True],
 [3, True],
 [4, False],
 [5, True],
 [6, False],
 [7, True],
 [8, False],
 [9, False],
 [10, False],
 [11, True],
 [12, False],
 [13, True],
 [14, False],
 [15, False],
 [16, False],
 [17, True],
 [18, False],
 [19, True]]
```

Ensuite, nous définissons une fonction qui à un naturel n associe le plus petit nombre premier strictement plus grand que n.

```
[5]: def NextPrime(n):
         m=n+1
         while not isPrime(m):
             m=m+1
         return m
```

```
[6]: [NextPrime(n) for n in range(20)]
```

```
[6]: [2, 2, 3, 5, 5, 7, 7, 11, 11, 11, 11, 13, 13, 17, 17, 17, 17, 19, 19, 23]
```

```
[7]: def Goldbach(n):
         j=2
         while j <= n//2 and not isPrime(n-j):
             j=NextPrime(j)
         return isPrime(n-j)
```

```
[8]: [Goldbach(n) for n in range(0,20)]
```

```
[8]: [True,
 True,
 False,
 False,
 True,
 True,
 True,
```

```
 True,
 True,
 True,
 True,
 False,
 True,
 True,
 True,
 True,
 True,
 False,
 True,
 True]
```

Pour vérifier la conjecture de Goldbach, nous n'avons besoin d'utiliser notre fonction Goldbach que sur les entiers pairs à partir de 4.

```
[9]: [Goldbach(n) for n in range(4,20,2)]
```

```
[9]: [True, True, True, True, True, True, True, True]
```

La fonction Goldbach a été conçue pour recevoir des arguments naturels. On peut se demander comment cette fonction se comporte lorsqu'on entre un entier négatif.

```
[10]: [Goldbach(n) for n in range(-20,0)]
```

```
[10]: [True,
 True,
 True,
 True,
 True,
 True,
 True,
 True,
 True,
 True,
 True,
 True,
 True,
 True,
 True,
 True,
 True,
 True,
 True,
 True]
```

Puisque la fonction Goldbach dépend des fonctions isPrime et NextPrime, il faut se demander

comment agissent ces fonctions lorsqu'on entre des arguments négatifs. On voit que la fonction isPrime rend systématiquement True. Pourquoi? Dès lors, comment va se comporter NextPrime sur des entrées négatifs?

[11]:
```
[isPrime(n) for n in range(-20,0)]
```

[11]:
```
[True,
 True,
 True,
 True,
 True,
 True,
 True,
 True,
 True,
 True,
 True,
 True,
 True,
 True,
 True,
 True,
 True,
 True,
 True,
 True]
```

En plus de l'information qu'un nombre est la somme de deux nombres premiers, on pourrait vouloir fournir deux tels nombres. On modifie notre fonction Goldbach pour rendre non pas uniquement True ou False à partir d'une entrée n, mais également deux nombres premiers dont la somme vaut n (lorsqu'il en existe bien entendu). Ainsi, la fonction Goldbach2 rend un quadruplet (n,True,i,j) où n est l'entrée et i et j sont des nombres premiers tels que n=i+j si n peut d'écrire comme la somme de deux nombres premiers, et rend le couple (n,False) lorsque n n'est pas la somme de deux nombres premiers.

[12]:
```
def Goldbach2(n):
    j=2
    while j <= n/2 and not isPrime(n-j):
        j=NextPrime(j)
    if isPrime(n-j):
        return [n,isPrime(n-j),j,n-j]
    return [n, False]
```

[13]:
```
[Goldbach2(n) for n in range(4,30,2)]
```

[13]:
```
[[4, True, 2, 2],
 [6, True, 3, 3],
 [8, True, 3, 5],
```

```
[10, True, 3, 7],
[12, True, 5, 7],
[14, True, 3, 11],
[16, True, 3, 13],
[18, True, 5, 13],
[20, True, 3, 17],
[22, True, 3, 19],
[24, True, 5, 19],
[26, True, 3, 23],
[28, True, 5, 23]]
```

Si nous voulons vérifier la conjecture de Goldbach pour un nombre arbitraire de nombres pairs, il est impraticable de procéder comme nous l'avons fait car nous obtiendrions des listes beaucoup trop longues. Nous allons créer une nouvelle fonction Goldbach3 qui à un entier n>=4 rend True si la conjecture est vérifiée pour les nombres pairs entre 4 et n, et rend False sinon.

[14]:
```
def Goldbach3(n):
    i=4
    while i <= n and Goldbach(i):
        i=i+2
    if i <= n:
        return False
    else:
        return True
```

[15]:
```
Goldbach3(10**3)
```

[15]:
```
True
```

Dans les sections suivantes, nous allons étudier et formaliser les différentes notions de programmation utilisée dans l'exemple introductif de la conjecture de Goldbach, à savoir la définition d'une fonction ou d'une procédure, les boucles for et while, les conditions (ou branchements) if, elif, else. Plus spéciquement, l'utilisation des listes en programmation (donnant lieu à la programmation dite fonctionnelle) est prévue par Python (et aussi par Mathematica). Je vous invite également à consulter le tutoriel https://docs.python.org/3/tutorial/.

2 Manipulation de listes

Une liste est une suite d'éléments entre crochets et séparés par des virgules. Les éléments d'une liste peuvent être n'importe quel objet python (une variable, un symbole réservé, une fonction, une liste…).

[16]:
```
from sympy import *
exemple=[2,4,sin,pi]
```

On accède au n-ième élément d'une liste avec des simples crochets. Les listes sont indicées à partir de 0 (et non de 1 comme dans Mathematica).

[17]:
```
exemple[0]
```

[17]: 2

On peut également accéder à une sous-liste d'une liste l avec la commande l[i:j]. On a alors la nouvelle liste constituée des éléments l[i], l[i+1],..., l[j-1].

[18]:
```
exemple[2:4]
```

[18]: [sin, pi]

On peut concaténer deux listes avec +. Ceci est différent de ce qui se passe dans Mathematica où l'addition de listes se fait composante à composante, pour autant que les listes soient compatibles. (Dans Sympy, les matrices ne sont pas encodées comme des listes, mais il faut utiliser la commande Matrix.)

[19]:
```
exemple+[1,31]
```

[19]: [2, 4, sin, pi, 1, 31]

[20]:
```
exemple
```

[20]: [2, 4, sin, pi]

Remarquez qu'avec cette façon de faire, le contenu de la variable exemple n'a pas été modifié. Un autre façon de faire est de concaténer deux listes grâce à la méthode .extend(). Dans ce cas, une fois la méthode appliquée, le contenu de la variable exemple a été automatiquement modifié !

[21]:
```
b=[2,17]
exemple.extend(b)
```

[22]:
```
exemple
```

[22]: [2, 4, sin, pi, 2, 17]

Des fonctions et méthodes utiles à la manipulation de listes sont len, max, min, .count(), .index(), .remove(), .reverse(), .sort(), .append(). Mais pour rappel, le plus simple pour trouver la fonction adéquate est d'utiliser exemple."tabulation", ce qui donne une liste de méthode applicable à la liste exemple.

[23]:
```
exemple.append(29)
```

[24]:
```
exemple
```

[24]: [2, 4, sin, pi, 2, 17, 29]

Pour créer des listes, on utilise la définition en compréhension (là où on utilise Table dans Mathematica).

[25]:
```
[i**2 for i in range(10)]
```

[25]:
```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

On peut également ajouter des conditions. Par exemple, on peut ne demander de ne conserver que les carrés qui sont divisibles par 3.

[26]:
```
[i**2 for i in range(10) if i%3==0]
```

[26]:
```
[0, 9, 36, 81]
```

La fonction range que nous venons d'utiliser est très pratique et très utilisée. La fonction range(n,m) permet de créer la liste des entiers de n à m-1. Pour voir la sortie de range sous forme de liste dans Python 3 (pas nécessaire dans Python 2), il faut appliquer la fonction list.

[27]:
```
range(0,4)
```

[27]:
```
range(0, 4)
```

[28]:
```
list(range(4))
```

[28]:
```
[0, 1, 2, 3]
```

[29]:
```
list(range(1,10,2))
```

[29]:
```
[1, 3, 5, 7, 9]
```

[ ]:
```

```

[ ]:
```

```

[ ]:
```

```

3 for

La boucle for permet de parcourir une liste et d'effectuer une instruction pour chaque élément de cette liste.

[30]:
```
exemple
```

[30]:
```
[2, 4, sin, pi, 2, 17, 29]
```

[31]:
```
for a in exemple:
    print(a)
```

```
2
4
sin
```

```
pi
2
17
29
```

Dans Python, l'indentation joue un rôle syntaxique primordial. En comparaison à d'autres langages de programmation qui utilisent des crochets, virgules et points-virgules ou encore des mots-clés tels que "begin" et "end" pour délimiter une instruction de boucle, Python paraît très épuré. Mais il faut faire extrêmement attention à l'indentation pour savoir si une instruction fait partie d'une partie du programme (comme par exemple une boucle for) ou d'une autre. Voyez l'exemple suivant.

[32]:
```
from sympy.abc import x
for n in range(1,5):
    t=n+x
    print(t)
```

```
x + 1
x + 2
x + 3
x + 4
```

[33]:
```
for n in range(1,5):
    t=n+x
print(t)
```

```
x + 4
```

Dans le premier cas, la commande print(t) fait partie de la boucle for. Au niveau de l'indentation, elle est placée dans le même alignement que la commande t=n+x. Dans le second cas, la commande print(t) ne fait pas partie de la boucle. L'affichage de la valeur de t se fera donc cette fois uniquement après la fin de la boucle, lorsque n aura la valeur 4.

4 while

[34]:
```
t=x
i=1
while i**2<10:
    t=t**2+i
    i=i+1
    print(t)
```

```
x**2 + 1
(x**2 + 1)**2 + 2
((x**2 + 1)**2 + 2)**2 + 3
```

Attention à l'ordre des instructions dans la boucle! Regardons ce qu'il se passe si on inverse l'ordre des instructions t=t**2+i et i=i+1.

[35]:
```
t=x
i=1
while i**2<10:
    i=i+1
    t=t**2+i
    print(t)
```

```
x**2 + 2
(x**2 + 2)**2 + 3
((x**2 + 2)**2 + 3)**2 + 4
```

En guise de 2ème exemple, nous pouvons implémenter facilement l'algorithme d'Euclide pour calculer le pgcd.

[36]:
```
a,b=27,6
while b != 0:
    a,b=b,a%b
a
```

[36]:
```
3
```

Remarquer que l'assignation simultanée s'écrit simplement comme a,b=b,a (ici on échange les valeurs de a et b).

5 Procédures et fonctions

Quelle est la différence entre une procédure et une fonction ? Une procédure est simplement une suite d'instructions. Ces instructions sont exécutées successivement (à la chaîne). Une procédure ne retourne pas nécessairement une valeur. Une fonction peut alors être vue comme une procédure particulière : une procédure qui retourne une valeur.

6 Définition d'une nouvelle fonction (ou procédure)

Pour définir la fonction x->x+2 dans Python, on écrit:

[37]:
```
def f(x):
    return x + 2
```

[38]:
```
f(29)
```

[38]:
```
31
```

La commande return de Python ne peut être utilisée qu'à l'intérieur d'un environnement def. C'est elle qui permet de définir la sortie de la fonction. Il est important de noter que dès que le programme définissant la fonction rencontre la commande return, celui-ci s'arrête. Il peut donc y avoir plusieurs utilisations de la commande return (typiquement, au sein d'un branchement if-else, voir plus loin), mais la première exécution de return arrête le programme. Autrement dit, tout ce qui suit cette commande est ignoré par Python.

[39]:
```
def f(x):
    return x + 2
    print('bonjour')
```

[40]:
```
f(3)
```

[40]: 5

[41]:
```
def f(x):
    print('bonjour')
    return x + 2
```

[42]:
```
f(3)
```

```
bonjour
```

[42]: 5

[ ]:
```

```

7 if - else

[43]:
```
def abs(x):
    if x < 0:
        print('salut')
        return -x
    else:
        return x
```

[44]:
```
abs(8)
```

[44]: 8

Le else n'est pas obligatoire. Si on ne veut rien faire lorsque la condition n'est pas satisfaite, on écrit simplement:

[45]:
```
def abs2(x):
    if x < 0:
        return -x
```

[46]:
```
abs2(7)
```

8 if - elif - else

Lorsqu'il y a plusieurs conditions successives à tester, on utilise le très pratique elif. Notez que les conditions sont alors testées consécutivement et ne doivent donc pas être forcément mutuellement

exclusives.

[47]:
```
from sympy.abc import x,y,z
def f2(a):
    if a>=1:
        return x
    elif a>=0:
        return y
    else:
        return z
```

[48]:
```
print(f2(2),f2(0.5),f2(-3))
```

```
x y z
```

Notez qu'il peut y avoir plusieurs elif consécutifs.

[ ]:
```

```

9 Procédures et fonctions

Quelle est la différence entre une procédure et une fonction ? Une procédure est simplement une suite d'instructions. Ces instructions sont exécutées successivement (à la chaîne). Une procédure ne retourne pas nécessairement une valeur. Une fonction peut alors être vue comme une procédure particulière : une procédure qui retourne une valeur.

10 Définition d'une nouvelle fonction (ou procédure)

Pour définir la fonction x->x+2 dans Python, on écrit:

[49]:
```
def f(x):
    return x + 2
```

[50]:
```
f(29)
```

[50]:
```
31
```

La commande return de Python ne peut être utilisée qu'à l'intérieur d'un environnement def. C'est elle qui permet de définir la sortie de la fonction. Il est important de noter que dès que le programme définissant la fonction rencontre la commande return, celui-ci s'arrête. Il peut donc y avoir plusieurs utilisations de la commande return (typiquement, au sein d'un branchement if-else, voir plus loin), mais la première exécution de return arrête le programme. Autrement dit, tout ce qui suit cette commande est ignoré par Python.

[51]:
```
def f(x):
    return x + 2
    print('bonjour')
```

[52]:
```
f(3)
```

[52]: 5

[53]:
```
def f(x):
    print('bonjour')
    return x + 2
```

[54]:
```
f(3)
```

```
bonjour
```

[54]: 5

[ ]:
```

```

11 Variables globales et variables locales

Toute variable utilisée à l'intérieur d'un environnement def de Python est considérée comme locale. Si on veut utiliser une variable globale, il faut le demander explicitement avec la commande global. Ceci est la convention inverse de celle de Mathematica. Il faut donc toujours avoir en tête comment se comporte un langage de programmation par rapport aux variables locales et globales.

Dans un programme, il est conseillé de séparer les suites d'instructions qui sont indépendantes, et d'utiliser des variables locales pour celles-ci. Ceci se fait en définissant des sous-procédures, auxquelles on fera appel ensuite. Il est donc important de leur donner un nom qui fait sens ! Ceci se réalise avec de la même façon que pour la définition de fonction (on n'est pas obligé de produire une sortie).

Par exemple, pour exécuter la procédure suivante (déjà vue plus haut), on peut lui donner un nom et ensuite y faire appel.

[55]:
```
def proc():
    for n in range(1,5):
        t=n+x
        print(t)
```

[56]:
```
proc()
```

```
x + 1
x + 2
x + 3
x + 4
```

Vérifions que la variable t utilisée dans cette définition est bien locale. On commence par lui assigner la valeur 2. Ensuite on exécute la procédure proc() où t est utilisée comme variable. Et enfin, on vérifie que t vaut toujours bien 2 après.

[57]:
```python
t=2
```

[58]:
```python
def proc():
    for n in range(1,5):
        t=n+x
        print(t)
```

[59]:
```python
proc()
```

```
x + 1
x + 2
x + 3
x + 4
```

[60]:
```python
t
```

[60]: 2

Si malgré tout, on veut faire appel à une variable globale, on peut le faire grâce à la commande global de Python. Mais attention aux effets de bord!

[61]:
```python
def proc2():
    global t
    for n in range(1,5):
        t=n+x
        print(t)
```

[62]:
```python
proc2()
```

```
x + 1
x + 2
x + 3
x + 4
```

[63]:
```python
t
```

[63]: $x + 4$

12 Objets modifiables/non modifiables

Voici une série d'exemples permettant d'appréhender comment Python traite les variables locales et globales, ainsi que les objets modifiables (variables, listes et dictionnaires) et les objets immuables (tuples).

Voici une fonction de deux arguments, nommés a et b. La fonction possède trois variables locales, nommées a, b et c. L'affectation de la variable locale a nous fait immédiatement perdre l'information du premier argument de la fonction. C'est

pourquoi, lors de l'affectation de la variable locale c, celle-ci prend la valeur `new-value' et non celle de l'argument a de la fonction.

[64]:
```
def fonction1(a, b):
    a = 'new-value'
    b = b + 1
    c = a
    return a, b, c

x, y = 'old-value', 99
fonction1(x, y)
```

[64]: ('new-value', 100, 'new-value')

Afin de conserver les valeurs des arguments de la fonction, il est recommandé d'utiliser des noms de variables locales différents de ceux des arguments.

[65]:
```
def fonction2(a, b):
    c = 'new-value'
    d = b + 1
    e = a
    return c, d, e

x, y = 'old-value', 99
fonction2(x, y)
```

[65]: ('new-value', 100, 'old-value')

On peut aussi utiliser d'autres objets modifiables en arguments, comme une liste ou un dictionnaire. La fonction suivante donne la même sortie que la précédente. Mais attention: elle modifie la liste entrée en arguments!

[66]:
```
def fonction3(a):
    a[0] = 'new-value'
    a[1] = a[1] + 1
    return a

args = ['old-value', 99]
fonction3(args)
```

[66]: ['new-value', 100]

La liste args a été modifiée par l'exécution de fonction3(args).

[67]:
```
args
```

[67]: ['new-value', 100]

Regardons à présent le comportement de Python sur le code suivant.

[68]:
```python
def fonction4(a):
    b = 'new-value'
    c = a[1] + 1
    return a,[b,c]

args = ['old-value', 99]
fonction4(args)
```

[68]:
```
(['old-value', 99], ['new-value', 100])
```

Ici, la liste args n'est pas modifiées par l'exécution de fonction4(args).

[69]:
```python
args
```

[69]:
```
['old-value', 99]
```

Le tuple est un objet python qui ressemble à une liste, mais qui a un comportement différent dans ce type de situation. Un tuple est une liste d'objets python quelquonque séparée par des virgules.

[70]:
```python
tuple=1,2,3
tuple
```

[70]:
```
(1, 2, 3)
```

[71]:
```python
liste=[1,2,3]
liste
```

[71]:
```
[1, 2, 3]
```

On accède aux éléments d'une liste et d'un tuple de la même façon, en utilisant des crochets avec l'indice de l'élément (sachant qu'on compte toujours à partir de 0 dans Python).

[72]:
```python
tuple[0]
```

[72]:
```
1
```

[73]:
```python
liste[1]
```

[73]:
```
2
```

Mais contrairement à la liste qui est un objet modifiale, le tuple est un objet qu'on ne peut plus modifier après sa création. Voyons un exemple pour comprendre. Lorsqu'on entre un tuple comme argument de fonction4 au lieu d'une liste, on en remarque aucune différence.

```
[74]: args = ('old-value', 99)
      fonction4(args)
```

```
[74]: (('old-value', 99), ['new-value', 100])
```

```
[75]: args
```

```
[75]: ('old-value', 99)
```

La différence se marque lorsqu'on modifie les éléments d'une liste à l'intérieur d'une fonction, comme on l'a fait dans fonction3. Ceci n'est pas possible avec un tuple. Pour le voir, essayons d'entrer un tuple comme argument de fonction3.

```
[76]: args = 'old-value', 99
      fonction3(args)
```

```
        ␣
 ↪---------------------------------------------------------------------------

        TypeError                                 Traceback (most recent call␣
 ↪last)

        <ipython-input-76-8996bde84d6c> in <module>
          1 args = 'old-value', 99
    ----> 2 fonction3(args)


        <ipython-input-66-8d3783d3cb23> in fonction3(a)
          1 def fonction3(a):
    ----> 2     a[0] = 'new-value'
          3     a[1] = a[1] + 1
          4     return a
          5


        TypeError: 'tuple' object does not support item assignment
```

Par contre, il n'y a pas de problème à entrer un tuple comme argument de fonction4 car son exécution ne demande pas d'affectation des éléments du tuple en entrée.

```
[77]: args = 'old-value', 99
      fonction4(args)
```

```
[77]: (('old-value', 99), ['new-value', 100])
```

[ ]:

```

```

Cours 6 : Introduction à la programmation avec *Mathematica*

Nous abordons ici très brièvement les bases de la programmation avec Mathematica et insistons principalement sur les similarités et les différences avec Python.

En particulier, nous reprenons le même exemple introductif (sur lequel nous passerons rapidement en classe, mais dont tous les détails se trouvent ci-dessous). Le but est de comprendre comment passer d'un langage de programmation à un autre en implémentant un même algorithme.

Conjecture de Goldbach

Rappelons que la conjecture de Goldbach s'énonce comme suit : tout entier pair strictement supérieur à 2 est la somme de deux nombres premiers.

Afin de vérifier cette conjecture pour un grand nombre de valeurs, nous allons définir une fonction Goldbach qui à un entier n rend True ou False selon que n est la somme de deux nombres premiers. Nous allons avoir recours à la fonction NextPrime de *Mathematica* qui à un entier n associe le nombre premier p strictement plus grand que n tel qu'aucun entier compris strictement entre n et p n'est premier, ainsi qu'à la fonction PrimeQ qui retourne True si l'argument entré est un nombre premier et False sinon. Remarquons que dans Python, nous avions implémenté nous-même des fonctions appelées isPrime et NextPrime. Ici, c'est plus confortable puisque de telles fonctions existent déjà.

```
In[1]:= Table[{n, PrimeQ[n]}, {n, 2, 20}]

Out[1]= {{2, True}, {3, True}, {4, False}, {5, True}, {6, False}, {7, True}, {8, False},
 {9, False}, {10, False}, {11, True}, {12, False}, {13, True}, {14, False},
 {15, False}, {16, False}, {17, True}, {18, False}, {19, True}, {20, False}}

In[2]:= Table[NextPrime[n], {n, 1, 20}]

Out[2]= {2, 3, 5, 5, 7, 7, 11, 11, 11, 11, 13, 13, 17, 17, 17, 17, 19, 19, 23, 23}

In[3]:= Goldbach[n_] :=
 Module[{j = 2},
  While[j ≤ n / 2 && ! PrimeQ[n - j],
   j = NextPrime[j]
  ];
  Return[PrimeQ[n - j]]
 ]

In[4]:= Table[Goldbach[n], {n, 0, 20}]

Out[4]= {True, False, False, False, True, True, True, True, True, True,
 True, False, True, True, True, True, True, False, True, True, True}
```

Pour tester la conjecture de Goldbach, seuls les nombres pairs plus grand ou égaux à 4 nous

intéressent :

```
In[5]:= Table[Goldbach[n], {n, 4, 20, 2}]
Out[5]= {True, True, True, True, True, True, True, True, True}
```

La fonction Goldbach a été conçue pour recevoir des arguments naturels. On peut se demander comment cette fonction se comporte lorsqu'on entre un entier négatif.

```
In[6]:= Table[Goldbach[n], {n, -20, 0}]
Out[6]= {False, False, False, True, False, True, False, False, False, True,
  False, True, False, False, False, True, False, True, False, True, True}
```

Il faut en fait se demander comment agit la fonction PrimeQ lorsqu'on entre des arguments négatifs.

```
In[7]:= Table[PrimeQ[n], {n, -20, 0}]
Out[7]= {False, True, False, True, False, False, False, True, False, True,
  False, False, False, True, False, True, False, True, True, False, False}
```

En plus de l'information qu'un nombre est la somme de deux nombres premiers, on pourrait vouloir fournir deux tels nombres. On modifie notre fonction Goldbach pour rendre non pas uniquement True ou False à partir d'une entrée n, deux nombres premiers dont la somme vaut n (lorsqu'il en existe bien entendu). Ainsi, la fonction Goldbach2 rend un quadruplet (n,True,i,j) où n est l'entrée et i et j sont des nombres premiers tels que n=i+j si n peut d'écrire comme la somme de deux nombres premiers, et rend le couple (n,False) lorsque n n'est pas la somme de deux nombres premiers.

```
In[8]:= Goldbach2[n_] :=
 Module[{j = 2},
  While[j ≤ n / 2 && ! PrimeQ[n - j],
   j = NextPrime[j]
  ];
  If[PrimeQ[n - j], Return[{n, PrimeQ[n - j], j, n - j}], Return[{n, False}]]
 ]
```

```
In[9]:= Table[Goldbach2[n], {n, 4, 30, 2}]
Out[9]= {{4, True, 2, 2}, {6, True, 3, 3}, {8, True, 3, 5}, {10, True, 3, 7}, {12, True, 5, 7},
 {14, True, 3, 11}, {16, True, 3, 13}, {18, True, 5, 13}, {20, True, 3, 17},
 {22, True, 3, 19}, {24, True, 5, 19}, {26, True, 3, 23}, {28, True, 5, 23}, {30, True, 7, 23}}
```

Si nous voulons vérifier la conjecture de Goldbach pour un nombre arbitraire de nombres pairs, il est impraticable de procéder comme nous l'avons fait car nous obtiendrions des listes beaucoup trop longues. Nous allons créer une nouvelle fonction Goldbach3 qui à un entier n≥4 rend True si la conjecture est vérifiée pour les nombres pairs entre 4 et n, et rend False sinon.

In[10]:=
```
Goldbach3[n_] := Module[{i = 4},
  While[i ≤ n && Goldbach[i], i = i + 2];
  If[i ≤ n, Return[False], Return[True]]
 ]
```

In[11]:=
```
Goldbach3[10 ^ 6]
```

Out[11]=
```
True
```

Si on compare les temps de calculs par Python ou Mathematica sur une même machine, on peut remarquer que Mathematica est plus rapide. Cela est probablement dû au calcul optimisé de PrimeQ et NextPrime dans Mathematica (boîte noire).

La fonction AbsoluteTiming nous permet de voir le temps mis par Mathematica pour effecter un calcul.

In[12]:=
```
AbsoluteTiming[Goldbach3[10 ^ 6]]
```

Out[12]=
```
{19.7245, True}
```

Dans les sections suivantes, nous allons étudier et formaliser les différentes notions de programmation utilisée dans l'exemple introductif de la conjecture de Goldbach, à savoir la définition d'une fonction ou d'une procédure, les boucles Do, For et While, les conditions (ou branchements) If. Plus spécique - ment, l'utilisation des listes en programmation (donnant lieu à la programmation dite fonctionnelle) est prévue par Mathematica et par Python. Je vous invite également à consulter la documentation via les liens guide/ProceduralProgramming et guide/FunctionalProgramming de Mathematica , ainsi que le tutoriel https://reference.wolfram.com/language/tutorial/LoopsAndControlStructures.html.

Listes, expressions, Map et Apply

Vous avez déjà eu l'occasion de manipuler des listes lors des TD. Nous présentons ici quelques complé - ments sur les expressions et les listes dans Mathematica.

Expressions, FullForm et Head

Mathematica représente (presque) toutes les expressions sous la forme f[a,b,...]. On peut toujours demander à Mathematica de nous montrer la forme complète d'une expression, c'est-à-dire l'expres - sion telle qu'il se la représente. Par exemple :

In[13]:=
```
FullForm[a + b + c]
```

Out[13]//FullForm=
```
Plus[a, b, c]
```

La tête de l'expression correspond alors à la fonction f. Dans notre exemple, c'est la fonction Plus.

In[14]:=
```
Head[a + b + c]
```

Out[14]=
```
Plus
```

Considérons maintenant la liste {a,b,c}. C'est aussi, pour Mathematica une expression de la forme f[a,b,...]. Ici, f est la fonction List.

In[15]:=
```
Head[{a, b, c}]
```
Out[15]=
```
List
```

Listes et Map

Si on souhaite appliquer une fonction à tous les éléments d'une liste, on utilise la fonction Map. Voyez la différence entre

In[16]:=
```
f[{a, b, c}]
Head[f[{a, b, c}]]
```
Out[16]=
```
f[{a, b, c}]
```
Out[17]=
```
f
```

et

In[18]:=
```
Map[f, {a, b, c}]
Head[Map[f, {a, b, c}]]
```
Out[18]=
```
{f[a], f[b], f[c]}
```
Out[19]=
```
List
```

In[20]:=
```
? Map
```
Out[20]=

Symbol

Map[*f*, *expr*] or *f* /@ *expr* applies *f* to each element on the first level in *expr*.

Map[*f*, *expr*, *levelspec*] applies *f* to parts of *expr* specified by *levelspec*.

Map[*f*] represents an operator form of Map that can be applied to an expression.

Remarquons que beaucoup de fonctions Mathematica s'appliquent par défaut directement à une liste. Dans ce cas, il est inutile d'utiliser Map. C'est le cas par exemple de Cos.

In[21]:=
```
Cos[{a, b, c}]
Map[Cos, {a, b, c}]
```
Out[21]=
```
{Cos[a], Cos[b], Cos[c]}
```
Out[22]=
```
{Cos[a], Cos[b], Cos[c]}
```

C' est aussi le cas pour la fonctions PrimeQ utilisée plus haut. Ainsi les deux commandes suivantes donnent la même sortie. Mais restez vigilents, car ce n'est pas toujours le cas.

In[23]:=
```
PrimeQ[{x, Pi, 1, 2, 3, 6, 7}]
Map[PrimeQ, {x, Pi, 1, 2, 3, 6, 7}]
```
Out[23]=
```
{False, False, False, True, True, False, True}
```
Out[24]=
```
{False, False, False, True, True, False, True}
```

Listes et NestList

Une autre fonction pratique est NestList. Voyez la différence avec Map :

In[25]:=
```
NestList[f, a, 4]
```

Out[25]= {a, f[a], f[f[a]], f[f[f[a]]], f[f[f[f[a]]]]}

In[26]:=
```
? NestList
```

Out[26]=

Symbol

NestList [f, $expr$, n] gives a list of the results of applying f to $expr$ 0 through n times.

Listes et Apply

La fonction Apply permet de prendre les éléments d'une liste comme arguments d'une fonction.

In[27]:=
```
f[{a, b, c}]
Apply[f, {a, b, c}]
Plus[{a, b, c}]
Apply[Plus, {a, b, c}]
```

Out[27]= f[{a, b, c}]

Out[28]= f[a, b, c]

Out[29]= {a, b, c}

Out[30]= a + b + c

Ici, Plus a un comportement particulier puisqu'elle peut s'appliquer à deux listes. Elle procède alors à l'addition composante à composante, ce qui correspond à la somme naturelle de deux vecteurs.

In[31]:=
```
Plus[{a, b, c}, {d, e}]
Plus[{a, b, c}, {d, e, f}]
```

••• Thread : Objects of unequal length in {a, b, c} + {d, e} cannot be combined .

Out[31]= {d, e} + {a, b, c}

Out[32]= {a + d, b + e, c + f}

Il faudra, lorsqu'on spécifie une fonction f, veiller à ce qu'elle ait le bon nombre d'arguments. Ici, Plus peut prendre un nombre arbitraire d'arguments. Il n'en va pas de même pour Cos, par exemple.

In[33]:=
```
Apply[Cos, {a}]
Apply[Cos, {a, b}]
```

Out[33]= Cos[a]

Out[34]= Cos[a, b]

En fait, la commande Apply[f, expression] remplace simplement le Head de l'expression par f.

In[35]:=
```
? Apply
```

Out[35]=

Symbol

f @@ *expr* or Apply[*f*, *expr*] replaces the head of *expr* by *f*.

f @@@ *expr* or Apply[*f*, *expr*, {1}] replaces heads at level 1 of *expr* by *f*.

Apply[*f*, *expr*, *levelspec*] replaces heads in parts of *expr* specified by *levelspec*.

Apply[*f*] represents an operator form of Apply that can be applied to an expression.

On peut par exemple utiliser ce principe pour remplacer un produit par une liste de ses facteurs, ou bien par la somme de ses facteurs.

In[36]:=
```
Factor[(Cos[x])^5 + (Sin[x])^5]
Head[%]
Apply[List, Factor[(Cos[x])^5 + (Sin[x])^5]]
Apply[Plus, Factor[(Cos[x])^5 + (Sin[x])^5]]
```

Out[36]= (Cos[x] + Sin[x]) (Cos[x]^4 – Cos[x]^3 Sin[x] + Cos[x]^2 Sin[x]^2 – Cos[x] Sin[x]^3 + Sin[x]^4)

Out[37]= Times

Out[38]= {Cos[x] + Sin[x], Cos[x]^4 – Cos[x]^3 Sin[x] + Cos[x]^2 Sin[x]^2 – Cos[x] Sin[x]^3 + Sin[x]^4}

Out[39]= Cos[x] + Cos[x]^4 + Sin[x] – Cos[x]^3 Sin[x] + Cos[x]^2 Sin[x]^2 – Cos[x] Sin[x]^3 + Sin[x]^4

Manipulations de listes

Lorsqu'on programme avec des listes, on se retrouve très rapidement, si l'on n'y prend garde, avec de nombreuses listes imbriquées les unes dans les autres. Je rappelle donc ici que la fonction Flatten, extrêmement utile, nous permet soit de revenir à une liste simples, ou bien de descendre de n niveaux de listes.

```
In[40]:= Flatten[{{{1, 2}, {3}, {{{4, 5, {6, 7}}, 8}, 9}}, 10}]
         Flatten[{{{1, 2}, {3}, {{{4, 5, {6, 7}}, 8}, 9}}, 10}, 1]
         Flatten[{{{1, 2}, {3}, {{{4, 5, {6, 7}}, 8}, 9}}, 10}, 2]
         Flatten[{{{1, 2}, {3}, {{{4, 5, {6, 7}}, 8}, 9}}, 10}, 3]
         Flatten[{{{1, 2}, {3}, {{{4, 5, {6, 7}}, 8}, 9}}, 10}, 4]
         Flatten[{{{1, 2}, {3}, {{{4, 5, {6, 7}}, 8}, 9}}, 10}, 5]
Out[40]= {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
Out[41]= {{1, 2}, {3}, {{{4, 5, {6, 7}}, 8}, 9}, 10}
Out[42]= {1, 2, 3, {{4, 5, {6, 7}}, 8}, 9, 10}
Out[43]= {1, 2, 3, {4, 5, {6, 7}}, 8, 9, 10}
Out[44]= {1, 2, 3, 4, 5, {6, 7}, 8, 9, 10}
Out[45]= {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
```

On accède au n-ième élément d'une liste avec [[n]].

```
In[46]:= l = {1, 2, 3, 4, 5};
         l[[4]]
Out[47]= 4
```

Si l est une liste de longueur n, on peut accéder à l'élément de l en position n-i+1 avec la commande l[[-i]].

```
In[48]:= l[[-1]]
Out[48]= 5
```

On génère la liste des n premiers entiers avec Range ou avec Table.

```
In[49]:= Range[7]
         Table[n, {n, 1, 7}]
Out[49]= {1, 2, 3, 4, 5, 6, 7}
Out[50]= {1, 2, 3, 4, 5, 6, 7}
```

```
In[51]:= Range[3, 10]
         Table[n, {n, 3, 10}]
Out[51]= {3, 4, 5, 6, 7, 8, 9, 10}
Out[52]= {3, 4, 5, 6, 7, 8, 9, 10}
```

Si on n'en veut qu'un sur p, on écrit (ici avec p=4) :

```
In[53]:= Range[3, 20, 4]
         Table[n, {n, 3, 20, 4}]
Out[53]= {3, 7, 11, 15, 19}
Out[54]= {3, 7, 11, 15, 19}
```

La syntaxe de Range est plus brève, mais n’est utile que pour créer des listes en progressions arithmé - tiques. En revanche, la fonction Table permet plus de flexibilité.

In[55]:=
```
Table[(2 n + 1) ^ 3, {n, 0, 10}]
```
Out[55]=
```
{1, 27, 125, 343, 729, 1331, 2197, 3375, 4913, 6859, 9261}
```

Voici quelques fonctions utiles pour la manipulation de listes.

In[56]:=
```
Union;
Complement;
Intersection;
Append;
Length;
Select;
Take;
Position;
Count;
Delete;
Insert;
Sort;
Reverse;
```

Chaînes de caractères

Les chaînes de caractères s’écrivent entre guillemets.

In[57]:=
```
"123"
```
Out[57]=
```
123
```

In[58]:=
```
Head[%]
```
Out[58]=
```
String
```

On peut avoir des caractères spéciaux : espace, tiret, apostrophe, etc. Pour avoir des guillemets, on utilise la commande \”. Pour avoir un backslash, on écrit \\.

In[59]:=
```
"Comment t'appelles-tu?"
"Je cite Cécile : \"Hello tout le monde\""
"Voici un backslash : \\"
```
Out[59]=
```
Comment t'appelles-tu?
```
Out[60]=
```
Je cite Cécile : "Hello tout le monde"
```
Out[61]=
```
Voici un backslash : \
```

In[62]:=
```
StringJoin["123", "435"]
```
Out[62]=
```
123435
```

In[63]:=
```
ToString[143 xy]
```
Out[63]=
```
143 xy
```

In[64]:=
```
ToExpression["123"]
ToExpression["123"] + 2
```
Out[64]=
```
123
```
Out[65]=
```
125
```

Voici quelques fonctions utiles à la manipulation de chaînes de caractères :

In[66]:=
```
RotateLeft ;
RotateRight ;
StringLength ;
StringReverse ;
StringTake ;
StringDrop ;
StringDelete ;
Sort;
```

Voivi le lien vers le tutoriel sur les chaînes de caractères : https : // reference.wolfram.com/language/tu - torial/StringsAndCharactersOverview.html.

Boucle Do

On utilise une boucle Do lorsque le nombre fois qu'on veut effectuer une instruction est connu à l'avance.

In[67]:=
```
Do[Print[n ^ 2], {n, 1, 4}]
```
```
1
4
9
16
```

On peut aussi utiliser la syntaxe suivante :

In[68]:=
```
n = 1; Do[Print[n ^ 2]; n++, 4]
```
```
1
4
9
16
```

n++ est équivalent à n=n+1.

In[69]:=
```
n = 1; Do[Print[n ^ 2]; n = n + 1, 4]
```

1

4

9

16

On peut aussi effectuer l'instruction Print[n^2] pour n allant de i jusque j en avançant avec un pas k.

In[70]:=
```
Do[Print[n^2], {n, 1, 9, 2}]
```

1

9

25

49

81

Cela revient à écrire

In[71]:=
```
n = 1; Do[Print[n^2]; n = n + 2, 5]
```

1

9

25

49

81

Une virgule délimite les différentes parties du Do, alors qu'un point-virgule sépare les instructions d'une procédure. Par exemple, nous avons

In[72]:=
```
Do[t = n + x; Print[t], {n, 1, 4}]
```

1 + x

2 + x

3 + x

4 + x

Remarquez que la commande suivante ne rend aucune sortie. A-t-elle tout de même une utilité ? En réalité, ni cette commande ni la précédente n'a de "sortie". En effet, Il n'y a pas de "Out" produit par leur évalutation. La première contient une commande Print, et donc Mathematica "affiche" de l'information. Afficher et produire une sortie sont deux actions différentes ! Il s'agit donc ici de procédures et non de fonctions.

In[73]:=
```
Do[t = n + x, {n, 1, 4}]
```

Procédures et fonctions

Nous venons d'utiliser plusieurs fois le mot "procédure". Nous avons dit que les instructions d'une procédure devaient être séparés par un point-virgule dans Mathematica. Mais qu'est elle est la différence entre une procédure et une fonction ? Une procédure est simplement une suite d'instructions.

Ces instructions sont exécutées successivement (à la chaîne). Une procédure ne retourne pas nécessairement une valeur. Une fonction peut alors être vue comme une procédure particulière : une procédure qui retourne une valeur.

Variables globales et variables locales

Attention que les deux procédures précédentes ont un effet de bord. En effet, lorsqu'on évalue t, on obtient

```
In[74]:= t
Out[74]= 4 + x
```

La variable t est une variable globale et la procédure a modifié la valeur de cette variable. En programmation, l'utilisation de variable globale est très souvent problématique. Il est prudent d'utiliser systématiquement des variables locales. Dans Mathematica, ceci se fait en utilisant Module. Voici la même procédure, où cette fois u est une variable locale.

```
In[75]:= Clear[t]

In[76]:= Module[{t}, Do[t = n + x; Print[t], {n, 1, 4}]]
         1 + x
         2 + x
         3 + x
         4 + x

In[77]:= t
Out[77]= t
```

Définition d'une nouvelle fonction

Nous avons déjà vu à plusieurs reprises comment définir une nouvelle fonction dans Mathematica.
Pour rappel, la syntaxe est la suivante.

```
In[78]:= f[x_] := x + 2

In[79]:= f[4]
Out[79]= 6

In[80]:= f[x_, y_] := x ^ 2 + y

In[81]:= f[3, 6]
Out[81]= 15
```

Lorsqu'on définit des fonctions plus compliquées, nécessitant l'introduction de variables intermédiaires, on veillera à utiliser Module.

In[82]:=
```
Goldbach[n_] :=
 Module[{j = 2},
  While[j ≤ n / 2 && PrimeQ[n - j] == False ,
   j = NextPrime[j]
  ];
  Return[PrimeQ[n - j]]
 ]
```

Dans un programme, il est conseillé de séparer les suites d'instructions qui sont indépendantes, et d'utiliser des variables locales pour celles-ci. Ceci se fait en définissant des sous-procédures, auxquelles on fera appel ensuite. Il est donc important de leur donner un nom qui ait du sens ! Ceci se réalise de la même façon que pour la définition de fonction (on n'est pas obligé de produire une sortie). Par exemple, pour exécuter la procédure

In[83]:=
```
Module[{t}, Do[t = n + x; Print[t], {n, 1, 4}]]
```

```
1 + x
2 + x
3 + x
4 + x
```

déjà vue plus haut, on peut exécuter Proc[4] après avoir défini la procédure Proc pour n'importe quelle entrée e :

In[84]:=
```
Proc[e_] := Module[{t}, Do[t = n + x; Print[t], {n, 1, e}]]
```

In[85]:=
```
Proc[4]
```

```
1 + x
2 + x
3 + x
4 + x
```

Il est d'autant plus utile de définir des sous-procédures qu'il est très courant qu'au cours d'un programme, on fasse appel plusieurs fois à une même sous-procédure. Ca permet de diminuer (souvent significativement) la longueur du code.

Pour communiquer son code, mais également pour se souvenir soi-même ce que l'on a codé, il est important d'insérer des commentaires. Par exemple, lors de la création d'une sous-procédure, on veillera à écrire quelques mots sur ce que fait celle-ci, voire de son fonctionnement (comment elle y parvient). Les commentaires dans Mathematica s'écrivent entre (* ... *) :

In[86]:=
```
(* La fonction Goldbach appliquée à un entier strictement positif n rend True
 lorsque n est la somme de deux nombres premiers et rend False sinon. *)
```

Boucle For

On utilise une boucle For lorsque le nombre fois qu'on veut effectuer une instruction n'est pas connu à

l’avance. La syntaxe est la suivante : For[initialisation,test,incrémentation,procedure].

In[87]:=
```
For[n = 1, n ≤ 4, n++, t = n + x; Print[t]]
```

1 + x

2 + x

3 + x

4 + x

Le principe du For est d’exécuter la procédure placée en premier argument (initialisation), et d’ensuite répéter la procédure placée en dernier argument jusqu’à ce que la condition placée en deuxième argument devienne fausse. À chaque étape, la variable d’incrémentation est modifiée.

In[88]:=
```
? For
```

Out[88]=

Symbol

For[*start*, *test*, *incr*, *body*] executes *start*, then repeatedly evaluates *body* and *incr* until *test* fails to give True.

Voici un deuxième exemple, provenant de la documentation de *Mathematica*.

In[89]:=
```
For[i = 1; t = x, i^2 < 10, i++, t = t^2 + i; Print[t]]
```

$1 + x^2$

$2 + (1 + x^2)^2$

$3 + \left(2 + (1 + x^2)^2\right)^2$

Inversement, on obtient le même rendu avec en utilisant un Do.

In[90]:=
```
t = x; Do[t = t^2 + i; Print[t], {i, 1, 3}]
```

$1 + x^2$

$2 + (1 + x^2)^2$

$3 + \left(2 + (1 + x^2)^2\right)^2$

Remarquez que, dans ce cas, l'initialisation a lieu en dehors de la boucle Do. Avec le Do, on doit savoir que i va prendre les valeurs entre 1 et 3. Il bien souvent plus pratique d'itérer une procédure jusqu'à ce qu'une condition devienne fausse, ce qui est permis par le For, mais pas par le Do.

À nouveau, ici t est une variable globale. On veillera à utiliser ceci au sein d’une procédure ou t a été déclarée comme variable locale.

In[91]:=
```
t
```

Out[91]= $3 + \left(2 + (1 + x^2)^2\right)^2$

In[92]:=
```
Clear[t]
Module[{t},
 For[i = 1; t = x,
  i ^ 2 < 10,
  i ++,
  t = t ^ 2 + i; Print[t]
 ]
]
```

$1 + x^2$

$2 + (1 + x^2)^2$

$3 + (2 + (1 + x^2)^2)^2$

In[94]:=
```
t
```

Out[94]= t

Remarquez la différence avec le code suivant. La commande t=x a été déplacée de la première partie du For à la quatrième partie du For. Dans le premier cas, t est initialisé à x au départ, puis est mis à jour à partir de la dernière valeur de t. Dans le deuxième cas, chaque itération de la boucle For réinitialise t avec la valeur x. Quand on effectue ensuite l'affectation t=t^2+i, on obtient donc t=x^2+i.

In[95]:=
```
Clear[t]
Module[{t},
 For[i = 1,
  i ^ 2 < 10,
  i ++,
  t = x; t = t ^ 2 + i; Print[t]
 ]
]
```

$1 + x^2$

$2 + x^2$

$3 + x^2$

Pour la deuxième procédure, on aurait donc pu écrire

In[97]:=
```
Clear[t]
Module[{t},
 For[i = 1,
  i ^ 2 < 10,
  i ++,
  t = x ^ 2 + i; Print[t]
 ]
]
```

$1 + x^2$

$2 + x^2$

$3 + x^2$

ou même

In[99]:=
```
For[i = 1,
 i^2 < 10,
 i++,
 Print[x^2 + i]
]
```

$1 + x^2$

$2 + x^2$

$3 + x^2$

puisque la variable t n'a plus d'utilité.

Boucle While

Comme pour la boucle For, la boucle While est utilisée quand on ne peut prédire le nombre exact d'itérations nécessaires d'une procédure pour arriver au résultat escompté. Ici, la syntaxe est la suivante : While[test,procedure]. La procédure est exécutée tant que le test rend True.

In[100]:=
```
Clear[i]
```

In[101]:=
```
i = 1; t = x;
While[i^2 < 10,
 t = t^2 + i;
 i++;
 Print[t]
]
```

$1 + x^2$

$2 + (1 + x^2)^2$

$3 + \left(2 + (1 + x^2)^2\right)^2$

La différence avec le For est qu'on n'a pas nécessairement une incrémentation, c'est à dire une variable qui augmente d'une constante (généralement 1) à chaque étape.

On pourrait en fait toujours utiliser le While à la place de Do et For. Par exemple,

In[103]:=
```
n = 1; While[n < 4, Print[n]; n++]
```

1

2

3

a le même rendu que

In[104]:=
```
Do[Print[n], {n, 1, 3}]
```

```
1
2
3
```

ou encore, que

In[105]:=
```
For[n = 1, n < 4, n++, Print[n]]
```

```
1
2
3
```

Un exemple classique : l'algorithme d'Euclide

L'algorithme d'Euclide se code facilement aver un While. Pour rappel (voir votre cours d'algèbre), l'algorithme d'Euclide calcule le PGCD de deux entiers non nuls en effectuant une suite de divisions euclidiennes.

In[106]:=
```
{a, b} = {27, 6};
While[b ≠ 0, {a, b} = {b, Mod[a, b]}];
a
```

Out[108]= 3

Notez la syntaxe {a,b}={b,Mod[a,b]}, qui permet de modifier simultanément les variables a et b. Voyez la différence avec le code suivant :

In[109]:=
```
{a, b} = {27, 6};
While[b ≠ 0, a = b; b = Mod[a, b]];
a
```

Out[111]= 6

Ici, on a modifié successivement les variables a et b, de telle sorte qu'au moment où la variable b est modifiée, elle tient compte de la dernière valeur de la variable a (qui est donc b, valeur qu'on vient de lui assigner) et non de la valeur de a à la fin de la dernière itération de la boucle. Dans ce cas, ce n'est pas ce qu'on souhaitait attribuer comme valeur à b, d'où le mauvais rendu. Si on veut utiliser des assignations de variables successives, il faudra alors introduire une nouvelle variable, souvent qualifiée de temporaire.

In[112]:=
```
{a, b} = {27, 6};
While[b ≠ 0, c = a; a = b; b = Mod[c, b]];
a
```

Out[114]= 3

Remarquons également que lorsqu'on utilise des assignations successives, l'ordre d'assignation des variables a toute son importance !

In[115]:=
```
{a, b} = {27, 6};
While[b ≠ 0, b = Mod[a, b]; a = b];
a
```
Out[117]= 0

Remarquons qu'avec cette implémentation de l'algorithme d'Euclide, on suppose que l'utilisateur rentre un couple (a,b) avec a≥b. Si on veut créer une fonction PGCD symétrique, on peut par exemple écrire :

In[118]:=
```
PGCD[a_, b_] := Module[{x = Max[a, b], y = Min[a, b]},
   While[y ≠ 0, {x, y} = {y, Mod[x, y]}];
   Return[x]
  ]
```

On a bien

In[119]:=
```
PGCD[27, 6]
PGCD[6, 27]
```
Out[119]= 3

Out[120]= 3

Remarque importante

Lorsque, comme dans PGCD, une fonction est définie par une procédure, il est interdit de modifier les entrées au cours de la procédure. Autrement dit, les arguments de la fonctions ne sont pas des variables de la procédure définissant la fonction. Au cours de la procédure, les arguments sont considérés comme des constantes. Ainsi, nous n'aurions pas pu écrire :

D'ailleurs, Mathematica nous signale immédiatement l'erreur : les "mauvaises" variables locales apparaissent en rouge.

In[121]:=
```
PGCD[a_, b_] := Module[{a = Max[a, b], b = Min[a, b]},
   While[b ≠ 0, {a, b} = {b, Mod[a, b]}];
   Return[a]
  ]
```

Si on n'utilise pas de variables locales, mais qu'on essaie de modifier les arguments d'une procédure, cela posera un problème également.

In[122]:=
```
PGCD[a_, b_] :=
   While[b ≠ 0, {a, b} = {b, Mod[a, b]}];
```

In[123]:=
```
PGCD[27, 6]
```

Set : Cannot assign to raw object 27.

Set : Cannot assign to raw object 6.

Set : Cannot assign to raw object 27.

General : Further output of Set::setraw will be suppressed during this calculation .

Out[123]=
```
$Aborted
```

Ceci crée en fait un problème de dépassement, et l'on est obligé de forcer l'arrêt de l'évaluation.

Condition If

Le If nous permet d'exécuter une procédure lorsqu'un test est évalué à True et une procédure alternative lorsque ce même test évalué à False. La syntaxe est If[test,procédure,procédure alternative]. Par exemple, on définit la fonction valeur absolue comme ceci :

In[124]:=
```
abs[x_] := If[x < 0, -x, x]
```

In[125]:=
```
abs[4]
abs[-7]
```

Out[125]= 4

Out[126]= 7

Ceci nous permet également de ne permettre des arguments que d'un certain type. Par exemple, si x est un nombre complexe, la fonction valeur absolue en x ne rend pas le module.

In[127]:=
```
abs[I]
```

Less : Invalid comparison with ⅈ attempted .

Out[127]=
```
If[ⅈ < 0, -ⅈ, ⅈ]
```

On peut, par exemple, signaler à l'utilisateur que l'entrée n'est pas valide :

In[128]:=
```
abs2[x_] := If[NumberQ[x] && Im[x] == 0, abs[x], Print[x, " n'est pas un argument valide"]]
```

In[129]:=
```
abs2[I]
abs2[z]
```

```
ⅈ n'est pas un argument valide
z n'est pas un argument valide
```

In[131]:=
```
NumberQ[I]
Im[I]
NumberQ[z]
```

Out[131]= True

Out[132]= 1

Out[133]= False

Rien ne nous empêche d'imbriquer des If les uns dans les autres, pour tester plusieurs conditions. Les tests sont alors appliqués successivement et ne doivent donc pas nécessairement être mutuellement exclusifs. En effet, le test 2 ne sera appliqué que si le test 1 rend False, et ainsi de suite.

```
In[134]:= Clear[x, y, z, f1]
f1[a_] := If[a ≥ 1, Return[x], If[a ≥ 0, Return[y], Return[z]]]
Map[f1, {2, 1 / 2, -1 / 2}]
Out[136]= {x, y, z}
```

Condition Which

Plutôt que d'imbriquer des If, on peut utiliser la commande Which, qui va effectuer la première commande après un True. La syntaxe est Which[test 1,procédure 1,...,test n,procédure n], où n est arbitraire.

```
In[137]:= f2[a_] := Which[a ≥ 1, Return[x], a ≥ 0, Return[y], True, Return[z]]
Map[f2, {2, 1 / 2, -1 / 2}]
Out[138]= {x, y, z}
```

```
In[139]:= Clear[x, b]
a = 2; Which[a == 1, x, a == 2, b]
Out[140]= b
```

Si aucune des conditions successives de Which ne rend True, Which ne rend rien.

```
In[141]:= a = 2; Which[a == 1, x, a == 3, b]
```

Si on souhaite que dans ce dernier cas, une action précise soit effectuée, il suffit de placer True comme dernier test. C'est ce que nous avons déjà fait dans l'exemple ci-dessus. Voici l'exemple de la documentation de *Mathematica*.

```
In[142]:= sign[x_] := Which[
  x < 0, -1,
  x > 0, 1,
  True, Indeterminate]
```

```
In[143]:= sign[0]
Out[143]= Indeterminate
```

Définition récursive d'une fonction

On peut aussi définir des fonctions récursivement. Par exemple, la fonction f suivante rend le nième nombre f[n] de Fibonacci.

```
In[144]:= Clear[f]
f[0] = 1; f[1] = 2;
f[n_] := f[n - 1] + f[n - 2];
```

```
In[147]:= Table[f[n], {n, 0, 10}]
Out[147]= {1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144}
```

```
In[148]:= Clear[f]
f[0] = 1; f[1] = 2;
f[n_] := f[n - 1] + f[n - 2];
```

Un autre exemple est celui de la factorielle.

```
In[151]:= Clear[g]
g[1] = 1;
g[n_] := n g[n - 1];
```

```
In[154]:= Table[g[n], {n, 1, 10}]
Out[154]= {1, 2, 6, 24, 120, 720, 5040, 40 320, 362 880, 3 628 800}
```

Quelques explications s'imposent.

```
In[155]:= Clear[x, y, a, b]
```

Lorsqu'on exécute la commande suivante, seul le symbole a devient noir. Comme attendu, la sortie est u (la valeur qui vient d'affecter l'expression a[x], qui est à gauche du signe =).

```
In[156]:= a[x] = u
Out[156]= u
```

Maintenant, si l'on exécute la commande suivante, c'est-à-dire si on "demande à Mathematica qui est a", on obtient simplement a comme sortie.

```
In[157]:= a
Out[157]= a
```

Regardons maintenant la sortie de la commande suivante.

```
In[158]:= a[x] + a[y]
Out[158]= u + a[y]
```

Cela peut paraître étrange a priori. En fait, le symbole a est devenu un objet hybride. On peut le voir comme une fonction définie uniquement pour l'argument spécifique x. Si on veut, on peut lui ajouter d'autres valeurs, mais la fonction n'est pas définie pour n'importe quel argument comme avec la syntaxe a[x_].

```
In[159]:= a[3] = 8
a[h] = t
Out[159]= 8
```

Out[160]= $3 + \left(2 + \left(1 + x^2\right)^2\right)^2$

On a donc à présent

In[161]:=
```
a[h]
a[j]
```

Out[161]= $3+\left(2+\left(1+x^2\right)^2\right)^2$

Out[162]= a[j]

Reprenons Fibonacci :

In[163]:=
```
Clear[f]
f[0] = 1; f[1] = 2;
f[n_] := f[n - 1] + f[n - 2];
```

Pour calculer f[n] pour un n donné, Mathematica va utiliser la récurrence autant de fois que nécessaire, et ce, tant qu'il n'a pas obtenu une valeur de i pour laquelle il connaît f[i]. Ce principe fonctionne pour la raison suivante : Mathematica donne toujours priorité à la définition de f donnée par le signe =. Les expressions f[0] et f[1] sont connues de lui, et lorsqu'il les obtient dans un calcul, il faire purement et simplement la substitution de leurs valeurs dans le calcul qu'il est en train d'effectuer.

Attention donc, lors d'une définition récursive, à bien spécifier les conditions initiales, en nombre suffisant ! La définition suivante n'est donc pas suffisante.

In[166]:=
```
Clear[f]
f[0] = 1;
f[n_] := f[n - 1] + f[n - 2];
```

In[169]:=
```
f[3]
```

$RecursionLimit : Recursion depth of 1024 exceeded during evaluation of f[-1020 - 1].

Out[169]= Hold[f[3 - 1] + f[3 - 2]]

Il faut spécifier les valeurs prises pas f en 0 ET en 1.

In[170]:=
```
Clear[f]
f[0] = 1; f[1] = 2;
f[n_] := f[n - 1] + f[n - 2];
```

In[173]:=
```
f[3]
```

Out[173]= 5

Enfin, si l'on souhaite de calculer f[n] pour un grand n, Mathematica va mettre beaucoup de temps avec cette définition. Par exemple, on met les temps suivantes pour les valeurs 30, 31, 32 et 33. Cela est typique d'un algorithme exponentiel : le temps de l'exécution est pratiquement doublé à chaque fois. En supposant que l'on continue à ce rythme, pour calculer f[38], cela prendra donc environ 3 minutes et cela mettra plus de 200 heures pour calculer f[50] !

In[174]:=
```
AbsoluteTiming[f[30]]
AbsoluteTiming[f[31]]
AbsoluteTiming[f[32]]
AbsoluteTiming[f[33]]
AbsoluteTiming[f[34]]
```

Out[174]= {1.27281 , 2 178 309}

Out[175]= {2.05608 , 3 524 578}

Out[176]= {3.33259 , 5 702 887}

Out[177]= {5.82778 , 9 227 465}

Out[178]= {10.1496 , 14 930 352}

Pour remédier à cela, il convient d'actualiser f[n] à chaque itération. En effet, Mathematica stocke ainsi les valeurs de tout les f[n] déjà calculés, et il ne calcule un nouveau f[n] que si celui si n’est pas encore disponible dans sa banque de valeurs stockées. L'écriture correspondante est la suivante :

In[179]:=
```
Clear[f]
f[0] = 1; f[1] = 2;
f[n_] := f[n] = f[n - 1] + f[n - 2];
```

Voyez la différence :

In[182]:=
```
AbsoluteTiming[f[50]]
```

Out[182]= {0.000177 , 32 951 280 099}

Remarquez tout de même que même cette manière de faire peut atteindre ses limites. Si on demande d’emblée à Mathematica de calculer f[2000], il ne le fera pas. Mais si on lui demande de le faire graduellement, il y arrivera.

In[183]:=
```
AbsoluteTiming[f[2000]]
```

$RecursionLimit : Recursion depth of 1024 exceeded during evaluation of f[979 - 1] + f[979 - 2].

Out[183]= {0.003666 , Hold[f[2000] = f[2000 - 1] + f[2000 - 2]]}

```
In[184]:= AbsoluteTiming[f[1000]]
          AbsoluteTiming[f[2000]]

Out[184]= {0.002718,
           113 796 925 398 360 272 257 523 782 552 224 175 572 745 930 353 730 513 145 086 634 176 691 092 ⋱
           536 145 985 470 146 129 334 641 866 902 783 673 042 322 088 625 863 396 052 888 690 096 969 577 ⋱
           173 696 370 562 180 400 527 049 497 109 023 054 114 771 394 568 040 040 412 172 632 376}

Out[185]= {0.002671,
           11 060 398 592 968 111 525 752 122 151 512 062 889 635 260 869 616 205 663 417 833 505 112 391 ⋱
           038 778 184 722 178 657 918 592 255 313 815 049 814 308 433 051 085 878 386 904 358 122 707 211 ⋱
           926 650 904 838 731 358 970 622 312 655 676 817 043 129 750 358 765 817 784 900 185 247 579 396 ⋱
           906 521 619 480 982 447 822 746 241 147 103 233 784 674 509 700 828 415 972 370 518 173 238 786 ⋱
           616 285 000 193 968 035 458 167 783 632 091 600 439 101 395 042 217 802 789 913 558 621 814 093 ⋱
           174 481 297 371 917 655 853 533 113 087 912 842 725 598 622 533 579 109 639 751}
```

Comment cela se fait-il ? En fait, comme il nous l'indique, Mathematica n'accepte pas de faire plus de 1024 itérations dans un calcul. C'est une limite artificielle qui lui est imposée. Soit ! Il n'est donc pas capable de calculer f[2000] s'il doit revenir aux conditions initiales f[0] et f[1]. Mais lorsqu'on lui demande de calculer f[1000], il peut non seulement le faire, mais en plus, puisque nous avons pris soin de stocker les valeures intermédiaires de f[n], il connait maintenant toutes les valeurs de f[n] pour n allant de 0 à 1000. Lorsqu'ensuite, on demande le calcul de f[2000], Mathematica va arrêter d'utiliser la définition récursive dès qu'il va atteindre des valeurs de n pour lesquelles il connaît f[n]. Il ne doit donc plus effectuer plus de 1024 étapes (mais seulement 1000) pour calculer f[2000] et est donc maintenant capable de le faire.

Par contre, on a toujours un problème pour calculer f[10000]. Au vu des explications précédentes, comment feriez-vous pour tout de même obtenir la bonne réponse?

```
In[186]:= f[10 000]

          ••• $RecursionLimit : Recursion depth of 1024 exceeded during evaluation of f[8978 - 1] + f[8978 - 2].

Out[186]= Hold[f[10 000] = f[10 000 - 1] + f[10 000 - 2]]
```

Cours 7 : Projet

Séance hybride cours/TP sur le projet en salle d'informatique. Démarrage du projet.

Cours 8 : GeoGebra

Je vous renvoie vers le livre "Introduction à GeoGebra" http://static.geogebra.org/book/intro-fr.pdf de Judith & Markus Hohenwarter (traduit en français par Noël Lambert), et bien sûr sur le site www.geoge - bra.org. De plus, vous pourrez consulter les tutoriels en français disponibles à l'adresse https://wiki. - geogebra.org/fr/Tutoriels. Enfin, de nombreux tutoriels en vidéo sont disponibles sur youtube.

Cours 9 et 10 – Introduction aux tableurs avec Calc

La suite bureautique de OpenOffice.org comprend les logiciels Writer, Calc, Impress, Draw, Math et Base. Les notes de ce dernier cours sont rédigées avec Writer. Nous y présentons le logiciel Calc, un tableur qui représente une alternative libre à Excel de Microsoft.

> **Avertissement** : Ces notes constituent un support, qui ne se substitue en rien à la présentation orale, avec le logiciel ouvert et projeté. Vous ne trouverez donc pas ici d'images provenant de Calc, mais plutôt une série de commentaires introductifs à son utilisation. Le mieux est bien sûr de lire ceci avec Calc ouvert devant les yeux.

Premier contact

Un fichier de Calc est appelé un classeur. Celui-ci est constitué de tableaux de lignes et de colonnes, appelés feuilles. Chaque ligne est référencée par un numéro et chaque colonne est référencée par une suite de lettres. Une cellule est référencée par sa colonne suivie de sa ligne. Par exemple, D2 est la cellule située sur la 2e ligne et la 4e colonne. Un groupe de cellules rectangulaires peut-être référencé en indiquant la référence de la cellule en haut à gauche, suivi de la référence de la cellule en bas à droite. Ainsi, E4:G7 est l'ensemble de cellules E4, E5, E6, E7, F4, F5, F6, F7, G4, G5, G6, G7.f

Cellule active

Parmi les cellules du tableau, seule une cellule est active. Elle se distingue par un contour marqué en gras. Si on tape une entrée au clavier, c'est dans cette cellule que l'entrée ira se placer. On peut changer la cellule active avec les flèches ou la souris. Pour entrer du contenu dans la cellule active, il suffit de taper au clavier ou de cliquer sur les options souhaitées. Pour valider le contenu de la cellule active (par exemple lors de l'écriture d'une fonction, voir ci-dessous), il faut presser la touche « Enter ».

Types de contenus

Les types de données que peut contenir une cellule sont notamment les suivants : du texte, une valeur numérique, un booléen, une date, mais aussi une fonction. Calc est généralement utilisé pour manipuler des données, et il est donc important d'être sûr du type de contenu de chaque cellule. En effet, comment par exemple additionner du texte ? Selon le type de contenu d'une cellule, Calc choisit un alignement à gauche ou à droite. On peut modifier cet alignement dans les options.

Copier, couper, coller et tirer des cellules

On peut copier, coller et couper des (ou un groupe de) cellules facilement, comme nous en avons l'habitude dans d'autres logiciels. Dans Calc, on peut également « tirer » des cellules. En plaçant la souris dans le coin inférieur droit de la cellule active, une croix apparaît. En cliquant sur la croix et

en déplaçant la souris (haut, bas, droite et/ou gauche), on modifie le contenu des cellules touchées. Calc essaie alors de deviner comment on souhaite modifier ces cellules connexes. Par exemple, si on tire une cellule qui contient 1, les cellules voisines deviendront 2, 3, 4, etc. Si on tire une cellule qui contient la lettre a, les cellules voisines deviendront a, a, a, etc. Il existe quantité d'autres possibilités. Essayez par exemple de tirer une cellule qui contient lundi, ou janvier, ou encore 2 cellules consécutives contenant 1 et 4.

Fonctions dans Calc

Comme dit ci-dessus, une cellule peut contenir une fonction. Dans ce cas, elle n'affichera pas la fonction elle-même, mais la valeur de cette fonction en des arguments qui peuvent dépendre du contenu d'autres cellules. Si ce contenu est modifié, la valeur de la fonction affichée sera adaptée. Par exemple, si la cellule A1 contient la fonction =SOMME(K1:K4), la valeur affichée à tout moment dans la cellule A1 sera la somme des valeurs des contenus des cellules K1, K2, K3 et K4 (pour autant que celles-ci contiennent des valeurs sommables). Notez que c'est le fait de commencer un contenu par le signe = qui indique à Calc qu'il s'agit d'une fonction. Une liste des fonctions de Calc se trouve dans l' « assistant fonction » (d'icône f_x). Pour chacune de ces fonctions, une documentation est fournie. Il existe également un raccourci spécifique pour la fonction SOMME. Outre SOMME, les fonctions les plus utilisées sont PRODUIT, MIN, MAX, MOYENNE, ARRONDI. On utilise aussi beaucoup les fonctions conditionnelles SI, NB.SI, SOMME.SI, MOYENNE.SI, etc. Essayez aussi la fonction AUJOURDHUI.

Référencements relatif, absolu et mixte

Dans la définition d'une fonction, on fait généralement référence à d'autres cellules. Il existe trois types de référencement à ces cellules, qu'il est important de distinguer.

Le premier est un référencement relatif. Cela signifie que le référencement se fait par rapport à la cellule qui contient la fonction. Dans notre exemple ci-dessus, la fonction =SOMME(K1:K4) dans la cellule A1 est interprétée par Calc en fonction des positions relatives de A1 et des cellules en argument K1:K4. En effet, pour Calc, cela signifie la somme des cellules du groupe rectangulaire dont la cellule en haut à gauche se trouve « 10 colonnes à droite de A1 » et « sur la même ligne que A1 » et la cellule en bas à droite se trouve « 10 colonnes à droite de A1 » et « 3 lignes plus bas que A1 ». Un des effets de ce référencement relatif est que si maintenant la cellule A1 est copiée et collée en B7, Calc adaptera automatiquement le contenu de B7 en =SOMME(L7:L10). Le copier-coller peut également se faire d'une feuille à l'autre et lorsque le référencement est relatif. Dans ce cas également, Calc adaptera les arguments de la fonction en fonction de la nouvelle cellule qui contiendra cette fonction.

Ceci est extrêmement pratique dans de nombreuses situations, mais il faut rester vigilent car cette action automatique de Calc n'est pas toujours souhaitable. C'est pourquoi il existe un deuxième type de référencement, appelé référencement absolu. Pour référencer de manière absolue une cellule, c'est-à-dire non plus en fonction de la cellule qui contient la fonction mais bien par rapport à la grille elle-même, on ajoute un symbole $ avant les références de lignes et de colonnes. Donnons un exemple d'une telle fonction. Si la cellule A17 contient la fonction =SOMME($K$1:$K$4), cela

signifie que les références des cellules du bloc rectangulaire K1:K4 ne pourront plus être modifiées lors d'un copier-coller.

Enfin, il est également possible de placer un $ devant la référence de la ligne uniquement ou de la colonne uniquement. Dans ce cas, on parle de référencement mixte.

Par défaut, la cellule référencée est celle de la feuille courante (celle dans laquelle on est en train de travailler). Si c'est la feuille 1 qui est la feuille courante, Calc adapte d'ailleurs automatiquement la référence C3 à Feuille1.C3. Vous l'aurez donc compris, on peut également référencer, dans une fonction, une cellule d'une autre feuille. Pour ce faire, il suffit de préciser Feuille4.C3, si on souhaite référence la cellule C3 de la feuille 4 plutôt que de la feuille 1. Il est courant de renommer les feuilles (il est souvent préférable de leur donner des noms qui font sens). Par exemple, si la feuille 4 a été renommée en « Bloc_1 », on fera référence à la cellule C3 de cette feuille en écrivant Bloc_1.C3.

Barre de formule

Au dessus de la grille elle même se trouve plusieurs « barres » contenant des informations sur la cellule active. La référence de cette cellule est dans la barre le plus à gauche. Dans le cas où on a sélectionné une colonne, une ligne, ou un groupe rectangulaire de cellules, cette barre contient les références de ces groupes de cellules, comme expliqué plus haut. Une deuxième barre, dite de formule, contient l'information sur le contenu de la cellule active. Lorsqu'il ne s'agit pas d'une fonction, la barre indique simplement le contenu de la cellule comme affiché par la cellule. Lorsqu'il s'agit d'une fonction, la cellule affiche la valeur de cette fonction au moment présent, mais la barre de formule affiche la fonction elle-même.

Masquer et afficher des lignes ou des colonnes

Il est parfois pratique de n'afficher que les lignes ou colonnes qui sont pertinentes à notre propos. C'est une façon d'extraire l'information utile sans pour autant la supprimer. Pour masquer des lignes, ou un groupe de lignes successives, on sélectionne ces lignes et on fait un « clic droit » sur la colonne des indices de lignes. On choisit ensuite l'option « masquer les lignes ». Ces lignes ne seront alors plus visibles. On peut tout de même voir que des lignes ont été masquées grâce à la numérotation de celles-ci, qui passe par exemple de 10 à 12 si la ligne 11 a été masquée. On voit également qu'un trait plus foncé sépare les lignes visibles, ici 10 et 12. Pour faire réapparaître ces lignes, on sélectionne la ligne qui précède le trait et celle qui le suit. Avec un nouveau « clic droit » effectué sur la colonne des indices de lignes, on choisit l'option « afficher les lignes ».

Fixer des lignes ou des colonnes

On peut aussi vouloir qu'une certaine ligne restent affichées à l'écran tout en déroulant les autres rangées. Pour ce faire, on sélectionne la ligne juste en-dessous de la ligne qu'on souhaite fixer et on fait un « clic droit » sur la colonne des indices de lignes. On choisit ensuite l'option « fixer lignes et colonnes ». La ligne du dessus restera alors visible même si on sélectionne des lignes très en-dessous d'elle. Ceci se remarque grâce à la numérotation des lignes, qui passe par exemple de 3 à n si la ligne 3 a été fixée et que les lignes affichées à l'écran sont indicées à partir de n. On voit également qu'un trait plus foncé sépare la ligne 3 et la ligne n. On peut annuler cette action en

exécutant un « clic droit » dans la colonne des indices de lignes et en désélectionnant l’option « fixer lignes et colonnes ». La même chose peut se faire au niveau des colonnes. Par contre, il n’est pas possible de fixer simultanément des lignes et des colonnes.

Options de style et formatage

Calc offre de nombreuses possibilités d'options de style et de formatage via les barres de raccourci. Plus des options plus avancées, vous pouvez aller consulter l'onglet « Format, Cellules », « Lignes » ou « Colonnes ». Un petit truc: pour passer à la ligne au sein d'une cellule, on appuie simultanément sur les touches « Control » et « Enter ».

Recherche dans un tableur

Un fichier de données Calc pouvant être très gros (souvent des centaines de lignes et colonnes), il est utile de ne pas devoir parcourir ce fichier visuellement pour en extraire des informations. Outre l'habituel « Control + f », il existe des fonctions de recherchées plus avancées : RECHERCHE, RECHERCHEV et RECHERCHEH (« V » pour vertical et « H » pour horizontal). Ici, en plus d'une démonstration au cours, je vous renvoie vers les pages de la documentation officielle :

https://wiki.openoffice.org/wiki/FR/Documentation/Calc:_fonction_RECHERCHE

https://wiki.openoffice.org/wiki/FR/Documentation/Calc:_fonction_RECHERCHEV

https://wiki.openoffice.org/wiki/FR/Documentation/Calc:_fonction_RECHERCHEH

Filtres

Il est souvent utile de pouvoir extraire les informations utiles d'un ensemble de données et de n'afficher que celles-ci. Ceci se réalise en utilisant des filtres. Calc propose plusieurs types de filtres. Du plus simple ou plus avancé, il y a l'auto-filtre, le filtre standard et le filtre spécial. Pour accéder aux filtres, on définit tout d'abord un ensemble de données. Pour ce faire, on peut soit sélectionner l'ensemble (généralement rectangulaire) de cellules qui constituent ce tableau. On peut aussi définir cet ensemble en lui donnant un nom. Ensuite, via l'onglet « Données », on sélectionne « Auto-filtre » ou « Plus de filtres ». Une fois le tableau filtré, seules les lignes répondant aux critères que l'on aura sélectionnés seront affichées. Il est parfois (voire souvent) plus intéressant de copier l’ensemble de cellules filtré dans une autre zone de la feuille, voire dans une autre feuille, afin de conserver un affichage de l’ensemble original. Pour ce faire, il faut cocher la case « Copier le résultat vers » et indiquer la zone où l'on souhaite copier l’ensemble filtré.

Diagrammes

Calc offre de nombreuses possibilités de créer des diagrammes: colonnes, barres, secteurs, zones, etc. On accède à l'assistant via l'onglet-raccourci « Diagramme ». Là, on peut choisir le type de diagramme souhaité, la plage de données concernée, ainsi que les séries de données et les éléments du diagramme souhaités.

Zone d'impression

Au moment de convertir son fichier .ods en un fichier .pdf, plus transportable, ou lorsqu'on souhaite imprimer son classeur, il est conseillé de définir une zone d'impression. Ainsi, rien ne vous oblige à imprimer tout ce qui se trouve sur votre feuille, mais seulement la partie que vous souhaitez voir imprimée. C'est très pratique car cela permet notamment de commenter ses données dans le fichier .ods sans voir ses commentaires imprimés dans le fichier .pdf. Pour définir une zone d'impression, on sélectionne le groupe de cellules à imprimer, et on clique ensuite sur le raccourci « Définir la zone d'impression ». On peut également trouver cette option via l'onglet « Format - Zones d'impression »