M 2 : Utilisation de *Mathematica*

Préambule

I. Calculs formels

A. Fonctions essentielles

Mathematica est capable d'effectuer toutes sortes de calculs formels que nous allons illustrer simplement :

1. ==

La première fonction est == qui indique si les expressions placées à droite et à gauche sont identiques. C'est le cas pour les expressions ci-dessous : comme on peut le vérifier, l'évaluation renvoie "`True`".

Dans la suite, par souci de compacité et de clarté d'écriture de ce texte, nous condenserons sur la même ligne la requête initiale à gauche du signe == et le résultat à droite (`True` étant implicite).

```
Pi == π
Factorial[n] == n!
Sin[x]^2 + Cos[x]^2 == Sin[x]^2 + Cos[x]^2
```

Dans ces trois exemples, c'est un simple test d'égalité de différentes écritures d'une même expression.

Dans les exemples ci-dessous, *Mathematica* ne fait pas seulement un simple test d'égalité. En effet, associé à l'évaluateur de *Mathematica* (boucle infinie qui réécrit l'expression jusqu'à ce qu'elle soit simple, *cf.* M1), ces tests d'égalité (qui renvoient "`True`") sont des prétextes ou des exemples de réécritures sur les nombres entiers :

$$\frac{1}{\frac{1}{3}+\frac{1}{2}} == \frac{6}{5}$$

$$\frac{5}{7} - \frac{2}{7}\left(1 - \frac{3}{4}\right) == \frac{18}{28}$$

On imagine assez facilement comment fonctionne l'évaluateur de *Mathematica* il effectue systématiquement une suite de factorisations en facteurs premiers, de réarrangements et de simplifications, sans doute du type ci-dessous, de chacun des deux membres qu'il ramène à une expression unique $\left(\frac{9}{14}\right)$:

$$\frac{5}{7} - \frac{2}{7}\left(1 - \frac{3}{4}\right) == \frac{1}{7}\left(5 - 2\left(\frac{1}{4}\right)\right) == \frac{1}{7}\left(5 - \frac{1}{2}\right) == \frac{1}{7}\left(\frac{10}{2} - \frac{1}{2}\right) == \frac{1}{7}\left(\frac{9}{2}\right) == \frac{18}{28}$$

On observe et on peut vérifier que toutes ces opérations sont faites automatiquement (sans qu'il soit nécessaire de faire appel à une fonction particulière de) et qu'elles renvoient toutes "`True`".

Application : Ces deux nombres sont très particuliers : en quoi ?

$$\left(-7 - 4\sqrt{2} - 2\sqrt{2\left(10 + 7\sqrt{2}\right)}\right) \qquad \left(-7 - 4\sqrt{2} + 2\sqrt{2\left(10 + 7\sqrt{2}\right)}\right)$$

Montrer que le produit de ces 2 deux nombres vaut 1 :

C'est facile à faire numériquement, mais vous n'avez pas prouvé que ce produit vaut exactement 1.

Mais on observe que *Mathematica* n'effectue les simplifications qu'au niveau prescrit comme on peut le constater :

```
Sin[x]^2 + Cos[x]^2 == 1
```

```
Cos[x]^2 + Sin[x]^2 == 1
```

Mathematica renvoie la meilleure expression possible selon les prescriptions données. Pour que *Mathematica* reconnaisse cette expression comme vraie, il faut lui prescrire l'utilisation des règles trigonométriques.

2. Autres tests

Test :	non – égalité	supériorité	supériorité ou égalité	infériorité ou égalité
Code :	x != y	x > y	x >= y	x < y
Apparence :	x ≠ y	x > y	x ≥ y	x < y

Pour combiner des expressions logiques avec un :
"ET" logique ("&&" en *Mathematica*), ou
"OU" logique ("||" en *Mathematica*).

Par exemple : pour tester si $x < 0$ ET $y = 0$, on écrit : `x<0&&y==0`
pour tester si $x < 0$ OU $y = 0$, on écrit : `x<0||y==0`

3. Simplify

La fonction **Simplify[]** recherche à réécrire l'argument sous la forme la plus simple possible, en utilisant les transformations algébriques et trigonométriques habituelles.

```
Simplify[Sin[x]^2 + Cos[x]^2]
```

```
1
```

Dans le cas où l'argument comporte implicitement deux membres comme dans l'exemple ci-dessous : la fonction **Simplify[]** recherche à réécrire chacun des membres sous la forme la plus simple possible de la même manière :

```
Simplify[Sin[x]^2 + Cos[x]^2 == 1]
```

Une simplification n'est pas toujours (ou plutôt beaucoup plus qu'attendu!) mathématiquement possible :

```
Simplify[Abs[Cos[x]] == Cos[x]]
```

```
Abs[Cos[x]] == Cos[x]
```

Aussi faut-il rajouter les bonnes conditions : ce que nous allons voir au point suivant avec **Assumptions**.

La fonction **FullSimplify[]** procède comme **Simplify[]** utilisant une base de transformations encore plus large, et notamment en utilisant les (ou des) fonctions spéciales. L'inconvénient est qu'elle peut prendre beaucoup (trop) de temps, et donc on ne l'utilise éventuellement que si **Simplify[]** ne suffit pas.

4. Simplify avec Assumptions

```
Simplify[Abs[Cos[x]] == Cos[x], x > 0]
Simplify[Abs[Cos[x]] == Cos[x], x < 0]
Simplify[Abs[Cos[x]] == Cos[x], 0 < x < π]
```

On peut aussi donner plusieurs suppositions avec les connecteurs logiques (&& veut dire ET, || veut dire OU) :

```
Simplify[Abs[Cos[x]] == Cos[x], 0 < x < π/2]
```

$$\text{Simplify}\left[\text{Abs}[\text{Cos}[x]] == \text{Cos}[x],\ -\frac{\pi}{2} < x < \frac{\pi}{2}\right]$$

$$\text{Simplify}\left[\text{Abs}[\text{Cos}[x]] == \text{Cos}[x],\ -\frac{\pi}{2} < x < \frac{\pi}{2}\ \&\&\ \left(-\frac{\pi}{2} + 2\pi < x < \frac{\pi}{2} + 2\pi\right)\right]$$

Retenir :

```
Simplify[Sqrt[x^2]]          (* Pas de simplification possible *)
Simplify[Sqrt[x^2], x ≥ 0] == x  (* True *)
```

PowerExpand[] suppose toujours que $x \geq 0$ et donc :

```
PowerExpand[Sqrt[x^2]] == x      (* True *)
```

C'est donc une fonction très pratique quand on est sûr que x est positif.

- **5. Simplify avec Assumptions et Element**

On peut aussi spécifier le domaine d'une variable avec la fonction **Element[]**. Imposons que x soit réel :

```
Simplify[Sqrt[x^2], Element[x, Reals]] ==
 Simplify[Sqrt[x^2], x ∈ Reals] == Abs[x]     (* True *)
```

Par défaut, *Mathematica* suppose que toutes les variables symboliques sont complexes (c'est le cas le plus général), et donc *Mathematica* ne peut opérer une réelle simplification que si on lui spécifie que x est réel.

Définir une variable n entier peut être utile pour les fonctions trigonométriques :

```
Element[(n - 1) / 4, Integers]
```

$$\frac{1}{4}(-1+n) \in \text{Integers}$$

Que donnent ?

```
Simplify[Sin[x + 2 * n * Pi], Element[n, Integers]]
Simplify[Cos[n * Pi / 2 - x], Element[(n - 1) / 4, Integers]]
```

On peut aussi combiner **Element[]** avec d'autres conditions (Laquelle permet de simplifier ?) :

```
Simplify[Log[x^r]]
Simplify[Log[x^r], x > 0]
Simplify[Log[x^r], Element[r, Reals]]
Simplify[Log[x^r], x > 0 && Element[r, Reals]]
```

- **6. Application**

Comment se simplifie l'expression suivante : $5 \ln a^2 - 2 \ln a$?

$$\text{Simplify}\left[5\,\text{Log}\left[a^2\right] - 2\,\text{Log}[a] == 5\,\text{Log}\left[a^2\right] - \text{Log}\left[a^2\right] == 4\,\text{Log}\left[a^2\right] == \text{Log}\left[a^8\right],\ a > 0\right]$$

Autrement dit, cette simplification n'est vraie que si a est un réel positif.

- **7. Pour aller plus loin : les domaines ℂ, ℝ, 𝔸, ℚ, ℕ, Premiers**

B. Fonction pure

- **1. Définition**

Les fonctions anonymes, encore appelées fonctions pures dans *Mathematica,* sont des fonctions qui n'ont pas de nom.

En pratique c'est le code même de la fonction qui est utilisé comme nom ou comme moyen ou d'utiliser la fonction. Exemple : pour coder : $f[x\_] := x^3 + 4$, on écrit :

```
(#^3 + 4) &
```

Pour calculer la fonction en $x = 5$, $f[5]$, il suffit d'écrire le code suivi de l'argument entre crochets :

```
(#^3 + 4) &[5]
```

```
129
```

Le rôle de la variable muette est tenu par #, et la fin de la fonction pure est marquée par &. Toutefois, pour une lecture plus claire, c'est une bonne pratique de bien repérer le début et la fin de la fonction par des ( ), toujours suivies de &.

Les fonctions pures sont une notion importante de la programmation fonctionnelle. Par construction, pour être utilisées, le nom des fonctions doit être répertorié dans un tableau propre au système qui est consulté chaque fois que l'on recherche la définition d'une fonction portant un nom donné. Pour éviter cette recherche de nom coûteuse en temps de calcul, on peut définir des fonctions pures localement au moment de leur utilisation.

- **2. Intérêts pratiques**

C'est essentiellement un moyen pratique pour *Mathematica* de communiquer une fonction, comme l'expression de

la solution d'un système d'équations différentielles. C'est aussi un moyen de contrôler certaines fonctions très utilisées comme **Select[]**. Pour ces raisons, il est indispensable de connaître le mode de définition et d'utilisation des fonctions pures.

Toutefois, c'est un mode d'écriture qui est généralement à proscrire car très peu lisible et difficile à documenter.

3. Application

Les nombres premiers ne peuvent pas être multiples de 2, ni de 4, ils sont au plus de la forme : $4n + 1$ ou $4n - 1$. Construisons et testons une fonction pure qui ne retient que les nombres multiples de 4 augmentés de 1 (elle ne renvoie que `True` ou `False`).

II. Listes

A. Définition et Opérations immédiates

1. Définition

Par définition une liste dans *Mathematica* est une suite d'expressions : c'est donc une famille ordonnée d'expressions.

Plus concrètement une liste est un ensemble ordonné (avec répétition ou non) de n'importe quelles expressions (nombres, variables, symboles, fonctions, graphiques, ou même des listes, des listes de listes, ...) séparées par des virgules et regroupées entre accolades.

```
liste1 = {a, b, c}
```

liste1 est une liste, où a, b, c peuvent être des expressions quelconques :

```
liste2 = {{a, 3}, b, {titi, f[x], -√57}, grosminet}
```

2. Adressage

D'un point de vue informatique, en première approximation, une liste est un tableau indexé implicitement.
On peut donc adresser simplement ses éléments. Étant donné que :

les simples crochets sont déjà réservés pour définir les fonctions **fonction[]**
les accolades sont utilisées pour les **listes**
et que les () servent au regroupement d'expressions comme dans : $(2 + 3) \times 5$

l'adressage est obtenu dans *Mathematica* au moyen de double crochets, ainsi :

```
liste1[[1]]
```

```
a
```

```
liste1[[3]]
```

```
c
```

```
liste2[[3]]
```

```
{titi, f[x], -√57}
```

En seconde approximation, les éléments du "tableau" n'étant pas de même nature, une liste peut donc être un tableau de structures génériques, ou une structure.

Pour éviter l'effet visuel perturbant des doubles crochets, on utilise souvent un équivalent plus esthétique avec des raccourcis clavier ESC[[ESC et ESC]]ESC pour rendre les doubles crochets plus compacts :

```
liste2⟦4⟧
```

```
grosminet
```

3. La propriété "Listable" de fonctions *Mathematica*

De nombreuses fonctions *Mathematica* et notamment les opérations ordinaires ont la propriété d'être "Listable" : cela signifie que cette fonction f peut être appliquée directement à une liste `{a,b,c}` pour donner la liste des images de cette fonction par f : `{f[a],f[b],f[c]}` :

```
liste3 = {3, 5, 11, 13, 25, 36, 77}
```

On peut donc faire directement des opérations sur les listes.

Les quatre opérations (renvoient toutes "True") :

```
2 + liste3 == {5, 7, 13, 15, 27, 38, 79}
liste3 - 39 == {-36, -34, -28, -26, -14, -3, 38}
2 * liste3 == {6, 10, 22, 26, 50, 72, 154}
```

$$\texttt{liste3} / 3 == \left\{1, \frac{5}{3}, \frac{11}{3}, \frac{13}{3}, \frac{25}{3}, 12, \frac{77}{3}\right\}$$

Fonctions élémentaires (renvoient toutes "True") :

```
liste3^2 == {9, 25, 121, 169, 625, 1296, 5929}
liste3 * liste3 == {9, 25, 121, 169, 625, 1296, 5929}
```

$$\texttt{Sqrt[liste3]} == \left\{\sqrt{3}, \sqrt{5}, \sqrt{11}, \sqrt{13}, 5, 6, \sqrt{77}\right\}$$

```
Log[liste3] == {Log[3], Log[5], Log[11], Log[13], Log[25], Log[36], Log[77]}
```

liste3 est un vecteur qui comporte 7 entrées.

La propriété "**Listable**" est une propriété d'opérations élémentaires très pratique sur les vecteurs. Elle est effectuée directement sans boucle de programmation pour construire l'opération.

■ 4. Opérations terme à terme, et fonctions "Listable"

Soit la liste :

```
liste4 = {1, 2, 3, 4, 5, 6, 7}
```

liste3 et **liste4** sont des vecteurs de même "nature" comprenant chacun 7 entrées.
Toutes les opérations terme à terme sont alors possibles directement (renvoient toutes "True") :

```
liste3 + liste4 == {4, 7, 14, 17, 30, 42, 84}
liste3 - (2 * liste4) == {1, 1, 5, 5, 15, 24, 63}
liste3 * liste4 == {3, 10, 33, 52, 125, 216, 539}
```

$$\texttt{liste3} / \texttt{liste4} == \left\{3, \frac{5}{2}, \frac{11}{3}, \frac{13}{4}, 5, 6, 11\right\}$$

Si les listes ne sont pas de même nature, les opérations sont (évidemment) impossible comme le dit fort bien *Mathematica*.

Soient les listes :

```
liste5 = {1, 2, 3, 4, 5, 6, 7, 8};
liste6 = {1, 2, 3, {4, 4}, 5, 6, 7};

liste3 + liste5
```

```
Thread::tdlen : Objects of unequal length in {3, 5, 11, 13, 25, 36, 77} + {1, 2, 3, 4, 5, 6, 7, 8} cannot be combined. >>
```

```
{3, 5, 11, 13, 25, 36, 77} + {1, 2, 3, 4, 5, 6, 7, 8}
```

C'est écrit en rouge, certes ! Mais ce ne sont pas des insultes : *Mathematica* vous explique au mieux pourquoi il ne peut pas faire l'opération. Ce sont au contraire des indications très précieuses !

Attention toutefois à des effets moins attendus, mais tout aussi logiques :

```
liste3 + (2 * liste6)
```

```
{5, 9, 17, {21, 21}, 35, 48, 91}
```

■ B. Générations de listes

■ 1. Range[]

La fonction **Range[]** génère une liste très simple (unidimensionnelle) de nombres (pas de symboles) (*cf*. aide) avec les exemples respectifs :

```
? Range
```

Range[i_{max}] generates the list {1, 2, …, i_{max}}.
Range[i_{min}, i_{max}] generates the list {i_{min}, …, i_{max}}.
Range[i_{min}, i_{max}, di] uses step di. ≫

```
Range[10]
Range[3.1, 8.5]
Range[3.1, 8.5, 0.5]
```

```
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
```

```
{3.1, 4.1, 5.1, 6.1, 7.1, 8.1}
```

```
{3.1, 3.6, 4.1, 4.6, 5.1, 5.6, 6.1, 6.6, 7.1, 7.6, 8.1}
```

2. Table[]

La fonction **Table[]** autorise beaucoup plus de souplesse :
L'élément générique de la liste est une expression (pas seulement numérique, mais pratiquement toute expression...). De plus les indices sont explicitement utilisés, ce qui autorise des listes multidimensionnelles et des contrôles pour générer la liste (*cf*. aide) avec les exemples respectifs :

```
? Table
```

Table[*expr*, {i_{max}}] generates a list of i_{max} 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. ≫

```
Table["Coucou", {5}]
Table[i, {i, 10}]
Table[{i, i^2}, {i, 3.1, 8.5}]
liste7 = Table[{i, i^2}, {i, 3.1, 8.5, 0.5}]
```

```
{Coucou, Coucou, Coucou, Coucou, Coucou}
```

```
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
```

```
{{3.1, 9.61}, {4.1, 16.81}, {5.1, 26.01}, {6.1, 37.21}, {7.1, 50.41}, {8.1, 65.61}}
```

```
{{3.1, 9.61}, {3.6, 12.96}, {4.1, 16.81}, {4.6, 21.16}, {5.1, 26.01}, {5.6, 31.36},
 {6.1, 37.21}, {6.6, 43.56}, {7.1, 50.41}, {7.6, 57.76}, {8.1, 65.61}}
```

3. Comparaison des vitesses de calcul de Range[] et de Table[]

La fonction **Range[]** opère très rapidement sur des nombres.
La fonction **Table[]** perd en rapidité ce qu'elle gagne en puissance et souplesse : *cf*. évaluation cachée.

4. Génération fonctionnelle de listes (Voir ultérieurement)

Mentionnons seulement la génération fonctionnelle de listes avec **NestList[]**, **NestWhileList[]**, **FoldList[]**, ...
cf. Aide

5. Génération procédurale de listes (Supplément)

(Supplément : peut être sauté en première lecture ; utile pour M8)
Génération procédurale de listes avec **Do[]**, **For[]**, **While[]** ... *cf*. Aide

Nous allons illustrer un élément important de programmation procédurale avec la boucle **Do[]**.
For[], **While[]** ... sont assez similaires, et ne sont pas abordés ici (*cf*. Aide).

Une boucle "**Do**" permet d'exécuter plusieurs fois une instruction. (Le mot instruction est plutôt réservé à la programmation procédurale. **Do[]** de *Mathematica* est plus générale puisqu'elle peut traiter une expression.)

La commande :

Do[instruction, {i, idebut, ifin}]

fait varier l'indice entier "i" (appelé "itérateur") de l'entier "idebut" à l'entier "ifin" et exécute à chaque fois "instruction". Par exemple, pour afficher "Coucou" 3 fois, on utilise une boucle **Do** exécutant l'instruction **Print["Coucou"]** lorsque ***i*** varie de 1 jusqu'à 3 :

```
Do[Print["   Coucou"], {i, 1, 3}]
```

```
   Coucou

   Coucou

   Coucou
```

Les boucles **Do** sont plus intéressantes si on a besoin d'utiliser l'entier *i* dans l'instruction. Par exemple, affichons les 3 premiers entiers :

```
Do[Print["   ", i], {i, 1, 3}]
```

```
   1

   2

   3
```

Utilisons une boucle **Do** pour calculer la somme des 100 premiers entiers $s = \sum_{i=1}^{100} i$. Trois étapes sont nécessaires :

1./ On commence par initialiser la variable ***s*** à 0 :

```
s = 0;
```

2./ Puis on fait varier *i* de 1 jusqu'à 100 par pas de1, et à chaque itération on calcule ***s + i*** et on affecte le résultat obtenu à la variable ***s*** :

```
Do[s = s + i, {i, 1, 100}]
```

3./ À la fin des itérations, la variable ***s*** contient la somme recherchée :

```
s
```

```
5050
```

Il est souvent plus clair (et un peu plus sûr) de regrouper ces trois instructions dans une seule cellule en les séparant par des ";" (qui suppriment les affichages intermédiaires)

```
s = 0;
Do[s = s + i, {i, 1, 100}];
s
```

On remarque que l'on a fait une opération sur une séquence de nombres sans avoir besoin de générer une liste. À chaque étape, on ne traite que deux nombres à la fois, et rien de plus. Dans un langage compilé (C, Fortran, ...), cette méthode de calcul "dans la mémoire" de l'ordinateur est très efficace. Dans *Mathematica* cette efficacité est souvent bonne, mais n'est pas forcément toujours garantie.

De même pour générer une liste, il faut :
1./ déclarer une liste vide, que l'on va remplir : **listeResultatDo**
2./ réécrire à chaque étape cette liste avec l'élément en plus : **listeResultatDo = Append[listeResultatDo, *i*]**
3./ on vérifie la construction de la liste en l'affichant à la fin

```
listeResultatDo = {};
Do[
  listeResultatDo = Append[listeResultatDo, i],
  {i, 1, 100}
 ];
listeResultatDo
```

```
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21,
 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41,
 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61,
 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81,
 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100}
```

Les boucles **Do[]** (comme **Table[]**) permettent de faire beaucoup de constructions, et notamment celles ci-dessous :

Par défaut, l'itérateur varie par incrément de 1, mais on peut aussi utiliser n'importe quel autre incrément. Par exemple, dans la boucle **Do** suivante l'entier j varie de 3 jusqu'à 10 par incrément de 2

```
Do[Print[j], {j, 3, 10, 2}]
```

Et dans la boucle suivante l'entier j varie de 5 jusqu'à 2 par incrément de -1

```
Do[Print[j], {j, 5, 2, -1}]
```

Enfin, on peut exécuter plusieurs intructions dans une boucle **Do** en les séparant par ";". Par exemple, calculons la somme s et le produit p des 10 premiers entiers, $s = \sum_{i=1}^{100} i$ et $p = \prod_{i=1}^{10} i$, avec une seule boucle **Do** :

```
s = 0; p = 1;
Do[
  s = s + i; p = p * i,
  {i, 1, 10}
 ];
Print[s];
Print[p];
```

C. Opérations sur les listes

1. Transposition d'une liste : Transpose[]

La fonction **Transpose**[tf] permet d'inverser la liste tf = { {x1, y1}, {x2, y2}, ...} qui devient après transposition { {x1, x2, ...}, {y1, y2, ...} } (*cf.* aide) :

```
? Transpose
```

```
tf = {{a, b, c}, {x, y, z}}
Transpose[tf]
Transpose[Transpose[tf]] == tf
```

```
{{a, b, c}, {x, y, z}}
```

```
{{a, x}, {b, y}, {c, z}}
```

```
True
```

2. Traitement individuel de chaque élément d'une liste : Map[]

Une fonction quelconque f n'a pas *a priori* la propriété "Listable" :

```
f[liste3]
```

```
f[{3, 5, 11, 13, 25, 33, 78}]
```

Mais il est possible de faire l'opération équivalente avec la fonction **Map[]** :

```
Map[f, liste3]
```

```
{f[3], f[5], f[11], f[13], f[25], f[33], f[78]}
```

- **3. Traitement global d'une liste : Apply[]**

La fonction **Apply[fonction, liste]** (*cf.* aide) est une belle illustration de programmation fonctionnelle par réécriture d'expressions pour obtenir une nouvelle fonction. En effet la représentation fonctionnelle de **liste1** est :

```
liste1 == {a, b, c} == List[a, b, c]                    (* True *)
```

La fonction **Plus[]** a le même format de représentation fonctionnelle :

```
a + b + c == Plus[a, b, c]                              (* True *)
```

Les noms des fonctions **List** et **Plus** sont appelées "**Head**" ou tête de la fonction représentée en notation fonctionnelle standard.

```
Head[a + b + c]
```

```
Plus
```

Pour faire la somme des éléments de liste, il suffit donc de changer la "tête" en transformant le nom de l'un (List) en celui de l'autre (Plus). C'est le rôle de **Apply[]**.

```
Apply[Plus, liste1] == Plus[a, b, c] == a + b + c       (* True *)
```

- **4. Sélection dans une liste : Select[]**

La fonction **Select[liste, critère]** sélectionne dans une "**liste**" les éléments qui satisfont le "**critère**" (*cf.* aide).

```
? Select
```

Select[*list*, *crit*] picks out all elements e_i of *list* for which *crit*[e_i] is True.
Select[*list*, *crit*, *n*] picks out the first *n* elements for which *crit*[e_i] is True. ≫

Pour ne retenir que les nombres impairs :

```
Select[{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}, OddQ]
```

```
{1, 3, 5, 7, 9}
```

Pour ne retenir que les nombres premiers :

```
Select[{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}, PrimeQ] == {2, 3, 5, 7}    (* True *)
```

Le critère est ici une fonction pure **(# > 3) &)** pour ne retenir que les éléments plus grands que 3 :

```
Select[{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}, (# > 3) &]
```

```
{4, 5, 6, 7, 8, 9, 10}
```

Le critère est ici une fonction pure $(\#^2 > 3\,\# + 4)\ \&$ pour ne retenir que les éléments dont le carré est plus grand que 3 fois le nombre plus 4 :

```
Select[{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}, (#^2 > 3 # + 4) &]
```

```
{5, 6, 7, 8, 9, 10}
```

Exemple résolu d'application :
Utiliser la fonction pure établie plus haut pour ne retenir que les nombres de la forme : $4\,n\,+\,1$ ou $4\,n\,-\,1$.

- **5. Comparaison des méthodes de génération d'une liste**

Comparons les temps de génération pour : **Range[]**, **Table[]**, et **Do[]** : les temps sont donnés en seconde.

- **6. Comparaison des méthodes de sommation d'une liste**

Comparons les différentes méthodes vues pour calculer la somme des 10^6 premiers entiers $s = \sum_{i=1}^{1\,000\,000} i$:

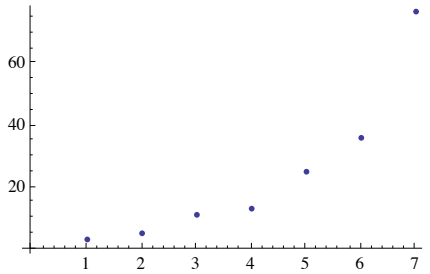
D. Mise en forme et Représentations de listes

1. Tracer une liste simple

On peut facilement tracer le graphe d'une liste. Pour une liste simple, l'abscisse qui est prise par défaut, est son numéro dans la liste.

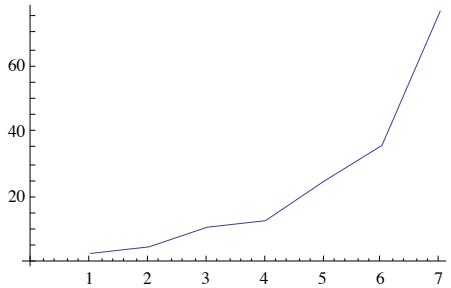
Tracer une liste en représentant les points ;

```
ListPlot[liste3]
```



Tracer une liste en représentant la ligne qui joint les points ;

```
ListPlot[liste3, Joined → True]
```



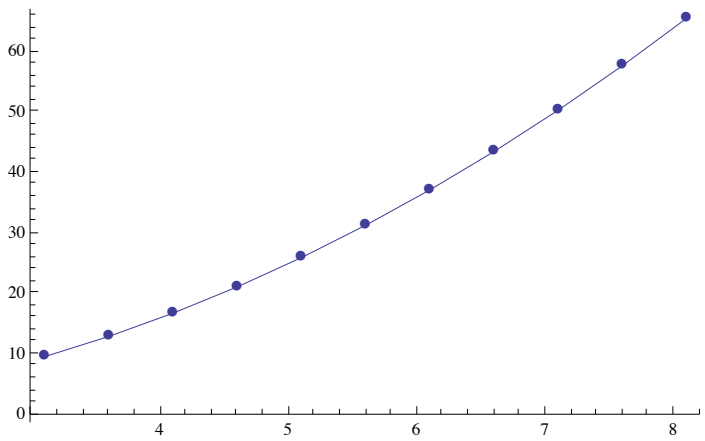
On peut le faire aussi avec :

```
ListLinePlot[liste3] == ListPlot[liste3, Joined → True]     (* True *)
```

2. Tracer une liste de deux coordonnées

Pour une liste de deux coordonnées, l'abscisse est la première des coordonnées et l'ordonnée la seconde :

```
ListPlot[
 liste7,
 Joined → True,
 PlotMarkers → Automatic
]
```



Notez que les points sont rendus visibles avec l'option : **PlotMarkers→Automatic**

■ **3. Tracer une liste à trois entrées (tracé en courbes de niveaux)**

Examinez l'exemple :

```
liste8 = Table[Cos[i] Cos[j], {i, 0, 2 π, 0.6}, {j, 0, 2 π, 0.6}]
```

```
ListContourPlot[liste8]
```

■ **4. Tracer une liste à quatre entrées (animation des tracés)**

Examinez l'exemple :

```
liste8 = Table[
   Cos[i + k (π / 30)] Cos[j],
   {i, 0, 2 π, 0.6}, {j, 0, 2 π, 0.6}, {k, 0, 30, 1}
  ];
ListAnimate[
 Map[
  ListContourPlot,
  liste8
 ]
]
```

■ **5. Mise en forme de listes avec TableForm[]**

TableForm[] est très utile pour mettre sous forme de tableaux :

```
? TableForm
```

Rappelons **liste7** :

```
liste7
```

```
{{3.1, 9.61}, {3.6, 12.96}, {4.1, 16.81}, {4.6, 21.16}, {5.1, 26.01}, {5.6, 31.36},
 {6.1, 37.21}, {6.6, 43.56}, {7.1, 50.41}, {7.6, 57.76}, {8.1, 65.61}}
```

```
liste7MiseEnForme = TableForm[
  liste7,
  TableHeadings → {None, {"i", "i²"}}
 ]
```

i	i^2
3.1	9.61
3.6	12.96
4.1	16.81
4.6	21.16
5.1	26.01
5.6	31.36
6.1	37.21
6.6	43.56
7.1	50.41
7.6	57.76
8.1	65.61

Attention à ne pas confondre **liste7** avec **liste7MiseEnForme** :
on peut faire des calculs avec **liste7** mais pas avec **liste7MiseEnForme**. (rappel : quand *Mathematica* ne peut pas traiter l'expression, il la ré-écrit à l'identique)

Exemple de mise en forme d'un tableau rectangulaire à 2 dimensions :

```
TableForm[
 Table[a_i,j, {i, 1, 3, 1}, {j, 1, 5, 1}]
]
```

$$\begin{array}{ccccc} a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} & a_{1,5} \\ a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} & a_{2,5} \\ a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} & a_{3,5} \end{array}$$

- **6. Mise en forme de matrices avec MatrixForm[]**

Exemple de mise en forme sous forme matricielle :

```
MatrixForm[
 Table[a_i,j, {i, 1, 3, 1}, {j, 1, 5, 1}]
]
```

$$\begin{pmatrix} a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} & a_{1,5} \\ a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} & a_{2,5} \\ a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} & a_{3,5} \end{pmatrix}$$

E. Caractéristiques d'une liste

Ce sont des fonctions très pratiques pour comprendre ce qu'on fait quand on manipule des listes :

- **1. Length[]**

La longueur d'une liste est donnée par la fonction **Length[]**

Rappelons pour mémoire **liste1**, **liste2**, **liste7** (**liste8** est trop volumineux pour être imprimé) :

```
liste1
liste2
liste7
```

```
{a, b, c}
```

```
{{a, 3}, b, {titi, f[x], -Sqrt[57]}, grosminet}
```

```
{{3.1, 9.61}, {3.6, 12.96}, {4.1, 16.81}, {4.6, 21.16}, {5.1, 26.01}, {5.6, 31.36},
 {6.1, 37.21}, {6.6, 43.56}, {7.1, 50.41}, {7.6, 57.76}, {8.1, 65.61}}
```

```
Length[liste1] == 3
Length[liste2] == 4
Length[liste7] == 11
Length[liste8] == 11
```

Pour gagner un peu de place, il est plus simple de faire :

```
Map[
 Length,
 {liste1, liste2, liste7, liste8}
]
```

```
{3, 4, 11, 11}
```

- **2. Dimensions[]**

Sa structure simple (rectangulaire à *n* dimensions) est donnée par **Dimensions[]**
On observe que la structure interne de **liste2** n'est pas vue par **Dimensions[]** contrairement à celles de **liste7** ou **liste8**.

```
Map[
 Dimensions,
 {liste1, liste2, liste7, liste8}
]
```

```
{{3}, {4}, {11, 2}, {11, 11}}
```

- **3. Short[]**

```
Map[
 Short,
 {liste1, liste2, liste7, liste8}
]
```

```
{{a, b, c}, {{a, 3}, b, {titi, f[x], -√57}, grosminet},
 {{3.1, 9.61}, <<9>>, {8.1, 65.61}}, {<<1>>}}
```

liste1 et **liste2** sont trop petites pour pouvoir faire un résumé.

- **4. Shallow[]**

C'est un résumé plus détaillé (non imprimé pour raison de place).

```
Map[
 Shallow,
 {liste1, liste2, liste7, liste8}
]
```

F. Calcul vectoriel

1. Combinaison linaire

Définissons deux vecteurs dans un espace de dimension 3, définissons les vecteurs $\vec{u}$ et $\vec{v}$ suivants :

```
u = {u1, u2, u3};
v = {v1, v2, v3};
```

On peut faire toutes les opérations de l'espace vectoriel $\mathbb{R}^3$, c'est à dire calculer toutes les combinaisons linéaires de ces vecteurs :

```
λ u + μ v
```

```
{u1 λ + v1 μ, u2 λ + v2 μ, u3 λ + v3 μ}
```

2. Produit scalaire de 2 vecteurs : Dot[]

Cet espace vectoriel est muni du produit scalaire ordinaire (utilisant la base canonique) :

```
u.v == u1 v1 + u2 v2 + u3 v3;
```

3. Norme d'un vecteur : Norm[]

Cet espace vectoriel est muni de la norme :

```
Norm[u] == √(Abs[u1]^2 + Abs[u2]^2 + Abs[u3]^2) ;          (* True *)
```

Attention à ne pas confondre les opérations . et * :

```
u.v == Dot[u, v] == u1 v1 + u2 v2 + u3 v3 == Apply[Plus, u * v] == Total[u * v]
```

```
u * v == Times[u, v] == {u1 v1, u2 v2, u3 v3}              (* True *)
```

4. Produit vectoriel : Cross[]

On dispose aussi de la définition du produit vectoriel :

```
Cross[u, v] == {-u3 v2 + u2 v3, u3 v1 - u1 v3, -u2 v1 + u1 v2}   (* True *)
```

- **G. Calcul matriciel (*cf.* aide)**

III. Programmation fonctionnelle et Programmation par règles

En première approximation, ce sont deux modes de programmation très proches dans l'esprit, qui diffèrent principalement par leur mode d'écriture.

- **1. ReplaceAll[] et Rule[]**

La programmation par règles consiste à utiliser des règles de remplacement.
Dans l'exemple suivant, l'élément a de la liste {a, b, c} est remplacé par b selon :

```
{a, b, c} /. a → b
```

```
{b, b, c}
```

Détaillons cette expression :

/. est la notation de type Infix de la fonction **ReplaceAll[]**.
Elle prend ici comme argument la liste **{a, b, c}** (et plus généralement une expression),
et comme second argument la règle de remplacement **a → b.**

```
({a, b, c} /. a → b) == ReplaceAll[{a, b, c}, a → b] == {b, b, c}
```

```
True
```

a → b est la notation de type Infix de la fonction **Rule[a, b]**

```
Rule[a, b]
```

```
a → b
```

- **2. Pattern[]**

L'intérêt de la programmation par règles est d'être associée aux **Pattern[],** c'est-à-dire à des motifs (et au typage).
Dans cet exemple, on remplace toutes les puissances de x^p par $f[p]$:

```
1 + x^2 + x^4 /. x^p_ -> f[p]
```

```
1 + f[2] + f[4]
```

- **3. Usages des règles de remplacement**

Attention aux petites subtilités suivantes :

Dans cette expression, les règles sont appliquées en une seule fois, et l'on obtient :

```
(x + 3 y) /. {x → y, y → z}
```

```
y + 3 z
```

Dans cette expression, on effectue les deux mêmes transformations mais successivement pour obtenir :

```
(x + 3 y) /. {x → y} /. {y → z}
```

```
4 z
```

Notez que ce mode d'écriture est ici plus clair que de faire un remplacement répété comme ci-dessous :

```
(x + 3 y) //. {x → y, y → z}
```

```
4 z
```

Application : Un brin d'ADN est constitué de nucléotides : A, T, G, C, écrire la séquence complémentaire.

IV. Traitement d'expressions symboliques

A. Transformations algébriques

1. Expand[]

Développement de l'expression $(2\,x+3)\left(x^2-6\right)$

```
Expand[(2 x + 3) (x^2 - 6)] == -18 - 12 x + 3 x^2 + 2 x^3
```

On peut spécifier la partie à développer :

```
Expand[(a + b)^2 (1 + x)^2] == a^2 + 2 a b + b^2 + 2 a^2 x + 4 a b x + 2 b^2 x + a^2 x^2 + 2 a b x^2 + b^2 x^2
```

```
Expand[(a + b)^2 (1 + x)^2, x] == (a + b)^2 + 2 (a + b)^2 x + (a + b)^2 x^2
```

```
Expand[(a + b)^2 (1 + x)^2, 1 + x] == (a + b)^2 (1 + 2 x + x^2)
```

2. Factor[] est l'opération inverse d'Expand[]

Factorisation de l'expression $-18-12\,x+3\,x^2+2\,x^3$

```
Factor[-18 - 12 x + 3 x^2 + 2 x^3] == (2 x + 3) (x^2 - 6)
```

Certaines expressions ne sont pas factorisées par **Factor** (ne fait rien) :

```
Factor[2 + 2 Sqrt[2] x + x^2]
```

Il faut ajouter une option :

```
Factor[2 + 2 Sqrt[2] x + x^2, Extension → Automatic]
```

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

En cas d'échec, il faut alors spécifier manuellement l'extension (ce mot fait référence à l'extension de Galois) :

```
Factor[x^4 - 25] == Factor[x^4 - 25, Extension → Automatic] == (-5 + x^2) (5 + x^2);
Factor[x^4 - 25, Extension → Sqrt[5]]
```

$$-\left(\sqrt{5}-x\right)\left(\sqrt{5}+x\right)\left(5+x^2\right)$$

B. Réarrangements algébriques

1. ExpandDenominator, ExpandNumerator

Cf. aide de *Mathematica*.

ExpandDenominator expands out products and powers that appear as denominators in *expr*.
ExpandNumerator expands out products and powers that appear in the numerator of *expr*.

2. Together[], Apart[], Cancel[]

cf. Aide
Together puts terms in a sum over a common denominator, and cancels factors in the result.
Apart rewrites a rational expression as a sum of terms with minimal denominators.
Cancel cancels out common factors in the numerator and denominator of *expr*.

3. Applications

→ Factoriser `x^2 - 3` avec *Mathematica*.

→ Soit $f[x]=\frac{(x+3)\,(x-1)^2}{\left(x^2+1\right)\,(x+5)^2}$. Utiliser les fonctions **Expand**, **ExpandAll**, **ExpandDenominator** et **ExpandNumerator** pour $f[x]$ et commenter les résultats.

→ Rechercher les fonctions *Mathematica* qui transforment (en une ou plusieurs étapes) $f(x)=\frac{1}{\left(x^2-16\right)}-\frac{x+4}{\left(x^2-3\,x-4\right)}$ en $g[x]=\frac{-15-7\,x-x^2}{-16-16\,x+x^2+x^3}$

C. Transformer des expressions trigonométriques

1. Introduction

Les fonctions de transformation d'expression algébrique comme **Expand** ou **Factor** laissent inchangées les expressions trigonométriques.

Pour transformer des expressions trigonométriques (et algébriques), on utilise les fonctions qui contiennent le préfixe "**Trig**", comme pour **TrigExpand** ou **TrigFactor** :

```
TrigExpand[Sin[x + y]] == Cos[y] Sin[x] + Cos[x] Sin[y]
```

```
TrigFactor[Sin[x]^2 + Tan[x]^2] == 1/2 (3 + Cos[2 x]) Tan[x]^2
```

Les trois fonctions **Simplify**, **FullSimplify** et **ComplexExpand** (voir Suppléments) sont capables de transformer à la fois des expressions algébriques et trigonométriques. Il n'existe donc pas de version "Trig" de ces trois fonctions.

Les formules d'identités trigonométriques peuvent être trouvées sur Wikipedia en français :

http://fr.wikipedia.org/wiki/Identité_trigonométrique

Mais il y a plus de détails sur la page équivalente en anglais (cliquer sur English à gauche sous Autres langues) :

http://en.wikipedia.org/wiki/List_of_trigonometric_identities

2. TrigExpand[]

TrigExpand développe une expressions en deux étapes :

1. Séparation des sommes et multiples entiers des arguments d'une fonction trigonométrique :

```
TrigExpand[Sin[2 * x]] == 2 Cos[x] Sin[x]
```

```
TrigExpand[Cos[2 * y]] == Cos[y]^2 - Sin[y]^2
```

```
TrigExpand[Sin[x + y]] == Cos[y] Sin[x] + Cos[x] Sin[y]
```

2. Ensuite développement de produits de fonctions trigonométriques en sommes de puissances :

```
TrigExpand[Sin[2 * x]] * TrigExpand[Cos[2 * y]]
```

```
TrigExpand[Sin[2 * x] * Cos[2 * y]]
```

```
2 Cos[x] Sin[x] (Cos[y]^2 - Sin[y]^2)
```

```
2 Cos[x] Cos[y]^2 Sin[x] - 2 Cos[x] Sin[x] Sin[y]^2
```

Dans cet exemple les deux calculs donnent le même résultat, car aucune identité trigonométrique n'est appliquée à la deuxième étape de TrigExpand.

Exemple où une identité trigonométrique est appliquée à la deuxième étape :

```
TrigExpand[Sin[x]^2 * Cos[y]]
```

$$\frac{\text{Cos}[y]}{2} - \frac{1}{2}\text{Cos}[x]^2\,\text{Cos}[y] + \frac{1}{2}\text{Cos}[y]\,\text{Sin}[x]^2$$

3. TrigFactor[]

TrigFactor essaie de factoriser une expression en utilisant les identités trigonométriques. Elle peut changer les arguments des fonctions trigonométriques. Exemple :

```
TrigFactor[Cos[y] * Sin[x] + Cos[x] * Sin[y]] == Sin[x + y]
```

```
TrigFactor[Sin[x] + Cos[x]] == Sqrt[2] Sin[π/4 + x]
```

4. TrigReduce[]

La fonction **TrigReduce** transforme des expressions trigonométriques en combinant les arguments ou en appliquant les règles d'identité d'angles multiples. Exemples :

```
TrigReduce[2 * Sin[x] * Cos[y]] == Sin[x - y] + Sin[x + y]  (* True *)
```

```
TrigReduce[2 * Cos[x]^2] == 1 + Cos[2 x]                    (* True *)
```

En général **TrigReduce** est la fonction inverse de **TrigExpand** :

```
Sin[2 * x] == TrigReduce[TrigExpand[Sin[2 * x]]];        (* True *)
Sin[x + y] == TrigReduce[TrigExpand[Sin[x + y]]];
```

→ **TrigFactor** peut donner dans certains cas le résultat inverse de **TrigExpand**. Tester les deux exemples. Dans lesquels des deux cas **TrigFactor** donne le résultat inverse de **TrigExpand** et pourquoi ?

→ Appliquer **TrigReduce** sur $f(x,\ y,\ z) = (\sin(x))^2 \cos(y) \sin(z)$ et **TrigFactor** sur le résultat. Commenter.

→ Comparer et commenter les trois résultats des fonctions **TrigExpand**, **TrigReduce** et **TrigFactor** appliquées à $f(x) = (\sin(x))^2 + (\tan(x))^2$

5. Simplify

Pour retrouver l'expression de départ après **TrigExpand** ou **TrigReduce** ou **TrigFactor** successifs, il peut être nécessaire d'utiliser **Simplify** :

```
ClearAll[x, f]
f[x_] := Sin[2 * x] * Cos[2 * y]
TrigExpand[f[x]] == 2 Cos[x] Cos[y]^2 Sin[x] - 2 Cos[x] Sin[x] Sin[y]^2;
TrigFactor[TrigExpand[f[x]]] == 4 Cos[x] Sin[x] Sin[π/4 - y] Sin[π/4 + y];
TrigReduce[TrigExpand[f[x]]] == 1/2 (Sin[2 x - 2 y] + Sin[2 x + 2 y]);
Simplify[TrigFactor[TrigExpand[f[x]]]] == Cos[2 y] Sin[2 x]
Simplify[TrigReduce[TrigExpand[f[x]]]] == Cos[2 y] Sin[2 x]
```

On peut facilement vérifier des identités trigonométriques avec **Simplify** et **==**, comme par exemple :

```
Simplify[Cos[a + b] == Cos[a] * Cos[b] - Sin[a] * Sin[b]]
```

Application : Vérifier les identités trigonométriques suivantes :

$\sin(a + b) = \sin(a) \cos(b) + \cos(a) \sin(b)$

$(1 - \cot(a))^2 + (1 - \tan(a))^2 = (\sec(a) - \csc(a))^2$

D. Calculs symboliques

Ces calculs ne sont qu'évoqués ici car ils seront étudiés dans la suite des séances (*cf.* Aide).

1. Résolution d'équations

Résolution de l'équation $x^2 - 3x + 2 = 0$ pour l'inconnue x

```
Solve[x^2 - 3 x + 2 == 0, x]
```

```
{{x → 1}, {x → 2}}
```

On obtient deux solutions réelles. Notez que pour définir une équation, on cherche x tel que l'équation soit égale à 0 : il faut donc utiliser le test "= =" et ne pas confondre avec l'affectation "=".

On observe que les solutions sont données sous forme d'une liste de règles. C'est très pratique et très fonctionnel.

Vérifions que les solutions sont toutes correctes :

```
x^2 - 3 x + 2 == 0 /. {{x → 1}, {x → 2}}
```

```
{True, True}
```

Pour avoir les valeurs des solutions, il suffit de faire :

```
x /. {{x → 1}, {x → 2}}
```

```
{1, 2}
```

2. Suites et sommes

```
Sum[i^2, {i, 1, 15}] == Sum_{i=1}^{15} i^2 == 1240        (* True *)
```

- **3. Limite de fonctions**

  Limite de $(x-1)\ln(x-1)$ quand x→ 1

```
Limit[(x - 1) Log[x - 1], x -> 1] == 0              (* True *)
```

- **4. Dérivée de x^n par rapport à x**

```
D[x^n, x]
```

- **5. Développement limité**

  Développement limité de cos(x) autour de $x = 0$ à l'ordre 10

```
Series[Cos[x], {x, 0, 10}]
```

- **6. Primitive ou intégrale indéfinie**

  Primitive ou intégrale indéfinie de cos(x) :

```
Integrate[Cos[x], x] == Sin[x]                    (* True *)
```

  Intégrale de $\int_0^\infty e^{-x^2}\,dx$:

```
Integrate[Exp[-x^2], {x, 0, Infinity}] == Sqrt[π]/2    (* True *)
```

- **E. Calculs numériques**

  Parfois, un calcul formel n'est pas faisable, mais on peut faire un calcul numérique approché. Quelques exemples :

  Calcul d'une valeur approchée de l'intégrale $\int_1^3 \ln(\arctan(x))\,dx$

```
NIntegrate[Log[ArcTan[x]], {x, 1, 3}]
```

```
0.135924
```

  Calcul approché des solutions de l'équation $x^5 + 3x^4 + 2x^3 - x^2 + x + 1 = 0$
  DSolve$[x^5 + 3x^4 + 2x^3 - x^2 + x + 1 == 0, x]$ ne donne rien, mais on obtient des racines numériques avec:

```
NSolve[x^5 + 3 x^4 + 2 x^3 - x^2 + x + 1 == 0, x]
```

```
{{x → -1.69305 - 0.668971 i}, {x → -1.69305 + 0.668971 i},
 {x → -0.566333}, {x → 0.476216 - 0.55321 i}, {x → 0.476216 + 0.55321 i}}
```

V. Tracer de fonctions

Pour les principaux types de tracés, on distingue

	les tracés de fonctions discrètes :	
2D	**ListPlot[]**, **ListLinePlot[]**,...	
3D	**ListPlot3D[]**, ...	
et	les tracés de fonctions continues :	
2D	**Plot[]**, **ParametricPlot[]**,...	(courbes)
3D	**Plot3D[]**, **ParametricPlot3D[]**,...	(courbes ou surfaces)
ainsi que	les tracés en courbe de niveaux	
2D	**ContourPlot[]**	
3D	**ContourPlot3D[]**	

Voici quelques exemples de tracés de fonctions continues :

Fonction	Exemple	Commande
f[x]	$\frac{\text{Sin[x]}}{\text{x}}$	Plot
{f[t], g[t]}	{Sin[3 t], Cos[3 t]}	ParametricPlot
f[x, y] == 0	$x^4 - (x^2 - y^2) == 0$	ContourPlot
f[x, y] ≥ 0	$x^4 - (x^2 - y^2) \geq 0$	RegionPlot
r = f[θ]	r = 3 Cos[5 θ]	PolarPlot

et bien d'autres encore (*cf*. guide/FunctionVisualization)

VI. Quelques méthodes de programmations

A. Définition et domaine de définition des variables

1. Motivations

Les affectations de variables sont un casse-tête, car elles sont la première cause de bogue dans un programme procédural. En principe, on devrait ne jamais les utiliser dans un cadre fonctionnel. En pratique, on risque toujours de les utiliser avec le danger permanent d'affectations à mauvais escient, ce qui constitue la bogue la plus courante. Comme c'est une source considérable de perte de temps pour vous, et pour les enseignants qui essaient de comprendre ce qui a pu se passer, et pourquoi "ça ne marche pas !", alors que "ça devrait marcher !", voici quelques méthodes très simples pour éviter les inconvénients des affectations.

2. With[]

La fonction **With[]** pourrait être traduite en français par "Soit[]". Son intérêt est de déclarer le nom d'une variable, et de réaliser une affectation qui reste strictement locale à la fonction.

Cela comporte beaucoup d'avantages. C'est à peu près équivalent à la phrase du type "soit $x = 3$". Il est souvent beaucoup plus judicieux de donner un nom plus long informatif, ce qui permet d'expliquer ce que fait la fonction, et de la documenter implicitement. Enfin, dès que la fonction **With[]** qui encapsule tout un calcul a cessé de calculer, toutes ces affectations locales cessent d'exister. Il n'y alors plus aucun risque.

Il n'est pas possible de déclarer une variable calculée à partir de celles que l'on vient de déclarer dans la même accolade d'un même **With[]**.
Les **With[]** doivent être emboîtés. On peut et on doit le faire à partir des variables déclarées dans le **With[]** qui l'emboîte.
With[] est donc contraignant, mais force le programmeur à définir les variables de base (dans le premier **With[]**), puis celles qui sont calculées à partir de celles-ci (dans le second **With[]**), et ainsi de suite.

3. Module[]

On peut aussi se passer de **With[]** en utilisant **Module**[] qui autorise la définition de variables à partir de variables définies dans **Module[]**. Il est donc plus puissant, mais moins structurant, et aussi généralement jusqu'à deux fois moins rapide que **With[]**.

B. Programmation graphique

Résumons ici quelques points qui seront présentés au cours des séances.

La programmation graphique dans *Mathematica* est tout à fait classique, et correspond dans les grandes lignes à celles qu'on retrouve pour piloter un traceur à feutre, un écran, ou pratiquement dans n'importe quel logiciel graphique.

1. Constitution d'objets graphiques

Dans un premier temps, la programmation graphique consiste à constituer des objets graphiques **Graphics[]** qui contiennent un dessin décrit par une liste (*i.e*. un ensemble ordonné) :

* des spécifications graphiques, couleur, épaisseur des traits, etc, appelés "directives graphiques" (**White, Black, Thick[]**, **Dashed[]**, ...)

* des éléments géométriques appelés "primitives graphiques" (**Point[]**, **Line[]**, **Arrow[]**, **Circle[]**, ...)

à l'aide des fonctions correspondantes,

On peut y adjoindre aussi du texte avec **Text[]**, et des conversions en chaîne de caractères avec **ToString[]**.

Ces objets graphiques peuvent être aussi des **Plot[]**, etc... .

- **2. Assemblage des objets graphiques**

Puis dans un second temps, l'assemblage de tous ces objets graphiques est effectué avec la fonction **Show[]** qui assure le cadrage et la mise à l'échelle avec les options **:**

Axes→True, PlotRange → {{xmin, xmax}, {ymin, ymax}}, AspectRatio→1, ImageSize → taille

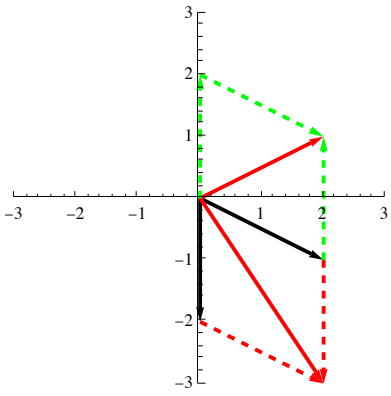
- **3. Application**

Question 19 du test : On considère 2 vecteurs dans un repère orthonormé : $\vec{u}(2,-1)$ et $\vec{v}(0,-2)$. Représenter $\vec{u}$ et $\vec{v}$ sur un schéma.

Question 20 du test : Sur un nouveau schéma, représenter la somme $\vec{u} + \vec{v}$ et la différence $\vec{u} - \vec{v}$ ($\vec{u}$ et $\vec{v}$ étant les vecteurs définis à la question 19)

Une solution simple de tracé mettant en œuvre les principes ci-dessus :

```
With[
 {O = {0, 0}, U = {2, -1}, V = {0, -2}},
 With[
  {S = U + V, D = U - V},
  Show[
   Graphics[
    {
     (* Solution question 19 en NOIR *)
     Thick, Black, Arrow[{O, U}], Arrow[{O, V}],
     (* Solution question 20 en ROUGE *)
     Red, Arrow[{O, S}], Arrow[{O, D}],
     (* Pointillés pour compléter la somme *)
     Dashed, Arrow[{U, S}], Arrow[{V, S}],
     (* Pointillés pour compléter la différence *)
     Green, Arrow[{O, -V}], Arrow[{-V, D}], Arrow[{U, D}]
    }
   ],
   (* Cadrage fixe du dessin dans Show *)
   Axes → True, PlotRange → {{-3, 3}, {-3, 3}}, AspectRatio → 1, ImageSize → 200
  ]]]
```



À partir de ces principes, il est possible de concevoir des dessins très sophistiqués.

C. Interactivité

- **1. Fonction**

Toute expression, et tout particulièrement les tracés graphiques peuvent être rendus interactifs avec M**anipulate[]** :

```
Plot[Sin[x (1 + 2 x)], {x, 0, 6}, PlotStyle → Black]
```

■ **2. Manipulate**

```
Manipulate[
 Plot[Sin[x (1 + a x)], {x, 0, 6}, PlotStyle → Black],
 {a, 0, 2}
]
```

I. Suppléments

■ A. Questions liées à l'affectation

■ 1. Règles pour les Noms de Variables et de Fonctions

Les noms de variables et de fonctions définies par l'utilisateur doivent commencer par une minuscule.

La raison est que les noms de toutes les variables et de fonctions prédéfinies dans *Mathematica* commencent tous par une majuscule. Cette règle simple permet d'une part de savoir d'emblée si une fonction est définie par l'utilisateur ou par *Mathematica*, et d'autre part, elle permet à *Mathematica* de rechercher plus rapidement les fonctions définies par l'un ou par l'autre.

Une autre règle implicite est que ces noms sont complets, non abrégés, et non cryptiques.

Enfin, ils respectent les règles usuelles, comme celle de ne pas commencer par un chiffre (ex : 2x), de ne pas contenir des caractères spéciaux (%, #, {, [), ni d'accents, ni d'espace, ni de souligné (tiret bas ou underscore) car ces symboles sont réservés comme nous l'avons vu plus haut.

■ 2. Affectations

L'affectation ne se comporte pas forcément exactement dans un système formel comme *Mathematica* que dans un contexte de programmation procédurale plus classique.

La raison est que *Mathematica* est un système. Tout ce que vous envoyez au Noyau par l'intermédiaire de la validation est mémorisé comme une règle à appliquer. Le moteur de *Mathematica* est un évaluateur (Evaluate) qui applique toutes les règles que vous lui avez données, ainsi que toutes celles dont il dispose pour obtenir un résultat jusqu'à ce qu'il ne varie plus au cours des évaluations successives, et qui vous est ensuite proposé. On imagine assez bien ce type de mécanisme de boucles infinies pour la simplification d'une expression. C'est aussi le cas pour tout calcul.

■ B. Tests d'Affectation

■ 1. Mise en place des tests

On cherche ici à tester le comportement d'une suite de commandes.
On peut écrire ces commandes, soit dans une même cellule en mettant toutes les instructions à la ligne, soit en mettant chaque élément de la liste dans une cellule séparée.
Les listes sont ordonnées. On reproduit donc plus simplement de façon synthétique sur une ligne ce que l'on peut écrire séparément d'une façon ou de l'autre.

Pour être absolument sûr que l'on repart à chaque fois d'un noyau vierge, on fait précéder l'exécution d'un "**Quit**" (qui doit évidemment être dans une cellule séparée validée séparément : une fois que le noyau est mort, il ne fait plus rien !).

Le premier, ainsi que les états intermédiaires, et le dernier élément donne l'état de **{x,y}**. On peut donc suivre les différentes étapes réellement effectuées.

■ 2. Affectation immédiate = et affectation retardée :=

```
(* Quit *) {{x, y}, x = Random[], y := Random[], {x, y}, {x, y}}
```

```
{{0, 1}, 0.430736, Null, {0.430736, 0.0381509}, {0.430736, 0.585249}}
```

x est défini une fois pour toute par une affectation issue d'un tirage aléatoire.
y n'a pas de valeur initiale, et prend une nouvelle valeur par tirage aléatoire chaque fois que *y* est invoqué.

- **3. Affectations numériques de variables, puis calcul**

```
In[1]:= (* Quit *) {{x, y}, x = 3, y = 4, x^2 + y^2, {x, y}, {x, y}}
Out[1]= {{x, y}, 3, 4, 25, {3, 4}, {3, 4}}
```

Tout est très simple.

- **4. Affectations numériques de variables, et de fonction, puis calcul**

```
In[1]:= (* Quit *) {{x, y}, x = 5, y = 4 + x, {x, y}, x = 10, {x, y}, {x, y}}
Out[1]= {{x, y}, 5, 9, {5, 9}, 10, {10, 9}, {10, 9}}
```

y est défini par une affectation qui est une fonction de ***x***, qui a été défini au préalable ($x = 5$).
y est donc affecté de la valeur 9. Cette affectation est définitive, et ne change pas même si x change.

- **5. Affectations numériques et affectation retardée, puis calcul**

```
In[1]:= (* Quit *) {{x, y}, x = 5, y := 4 + x, {x, y}, x = 10, {x, y}, x = 12, {x, y}, {x, y}}
Out[1]= {{x, y}, 5, Null, {5, 9}, 10, {10, 14}, 12, {12, 16}, {12, 16}}
```

y est défini par une affectation retardée qui est une fonction de ***x***, qui a été défini au préalable ($x = 5$).
y n'est donc pas évalué, et n'a pas de valeur (`Null`). Par contre y prend ensuite une valeur nouvelle chaque fois que x est modifié et que y est évalué.

Les comportements que l'on observe correspondent bien à ce qu'on attend.

- **6. Affectations de fonctions, affectations numériques, puis calcul**

On obtient (presque) le même comportement dans les deux cas de figures :

```
(* Quit *) {{x, y}, y = 4 + x, x = 5, {x, y}, x = 10, {x, y}, x = 12, {x, y}}
{{x, y}, 4 + x, 5, {5, 9}, 10, {10, 14}, 12, {12, 16}}
(* Quit *) {{x, y}, y := 4 + x, x = 5, {x, y}, x = 10, {x, y}, x = 12, {x, y}}
{{12, 16}, Null, 5, {5, 9}, 10, {10, 14}, 12, {12, 16}}
```

On peut aussi examiner ce que contient la mémoire du noyau :

```
? x
```

```
Global`x
```

```
x = 12
```

```
? y
```

```
Global`y
```

```
y = 4 + x
```

```
y
```

```
16
```

y est donc bien mémorisé comme une fonction de x, et si x est défini (ici 12), alors y vaut 16.

En résumé, si l'on utilise "=" pour définir des variables et des fonctions, il faut faire très attention à ce qui est vraiment mis en mémoire une valeur ou une fonction d'une variable.

- **7. Affectations mal conçues et calcul**

Exemple de comment faire une boucle infinie avec seulement deux définitions incohérentes :

```
(* Quit *) {{x, y}, y = 4 + x, x = y + 5}
```

```
$RecursionLimit::reclim : Recursion depth of 256 exceeded. >>
```

La raison fondamentale est que l'évaluateur de Mathematica est une boucle infinie qui utilise les expressions données pour atteindre un résultat qui ne varie pas. Or ce résultat qui ne varie pas est impossible à atteindre ici.

C. Calculs et Nombres complexes

1. Écriture des nombres complexes

Un nombre complexe $z = a + i\, b$ est écrit dans *Mathematica* :

```
z = a + i b == a + I b                    (* True *)
```

Certains calculs peuvent donner des nombres complexes :

```
Sqrt[-9] == 3 i                           (* True *)
```

2. Par défaut : a∈ℂ et b∈ℂ

Par défaut, *Mathematica* suppose que toute variable est complexe. Autrement dit, il suppose que $a \in \mathbb{C}$ et $b \in \mathbb{C}$ comme on peut l'observer ci-dessous :

```
z = a + i b;
Re[z] == -Im[b] + Re[a]                   (* True *)
Im[z] == Im[a] + Re[b]                    (* True *)
Abs[z] == Abs[a + i b]                    (* True *)
```

2. ComplexExpand, et par défaut : $a \in \mathbb{R}$ et $b \in \mathbb{R}$

Il est important de pouvoir supposer que $a \in \mathbb{R}$ et $b \in \mathbb{R}$. Pour cela il faut utiliser les fonctions :

```
z = a + i b;
ComplexExpand[Re[z]] == a                 (* True *)
ComplexExpand[Im[z]] == b                 (* True *)
ComplexExpand[Abs[z]] == Sqrt[a^2 + b^2]  (* True *)
```

Exercices

Cf. le fichier joint *M2_4M062UtilisationMMaEtu.nb*

I. Calculs formels

A. Fonctions essentielles

Dans les calculs ci-dessous, écrivez la suite des opérations que vous devez faire pour pour aboutir au résultat, en faisant une étape (factorisation OU réduction au même dénominateur OU simplification OU réarrangement , etc ...)

1. ==

→ Calculer au moyen d'une suite d'expressions :

$$\frac{1}{\frac{1}{3}+\frac{1}{2}} ==$$

→ Calculer au moyen d'une suite d'expressions :

$$\frac{5}{7}-\frac{2}{7}\left(1-\frac{3}{4}\right) ==$$

→ Expliquez dans les grandes lignes comment *Mathematica* parvient à ce type de résultat.

... Pour la suite des exercices *cf*. le fichier joint *M2_4M062UtilisationMMaEtu.nb* à renommer avec votre nom ...

M2 / Contents