Aide-mémoire pour Mathematica

Avertissement : quand une phrase se termine par une commande *Mathematica*, la ponctuation (.,;:?) est *omise* pour éviter toute ambiguïté avec la syntaxe du langage formel.

Cet aide-mémoire est très complet mais loin d'être exhaustif. Il faut, lors d'une utilisation en machine, profiter du Help.

A Notions de base

1 Création et utilisation d'un carnet

- Les fichiers *Mathematica* sont appelés Carnet ou en anglais Notebook.
- Pour créer un nouveau Carnet :
  → par le menu File puis New, cliquez sur Notebook ;
  → ou tapez directement Ctrl N.
- Pour enregistrer votre Carnet :
  → par le menu, cliquez sur Save ;
  → ou tapez directement Ctrl S.
- Pour ouvrir votre Carnet :
  → par le menu, cliquez sur Open ;
  → ou tapez directement Ctrl O.
- Pour fermer votre Carnet :
  → par le menu, cliquez sur Close ;
  → ou tapez directement Ctrl W.

2 Édition de texte

- *Mathematica* permet la saisie de texte.
- Chaque morceau de texte est placé dans une cellule délimitée à droite par un]
- Les styles sont modifiables par le menu Format

3 Instructions

- *Mathematica* permet l'exécution de commandes, qui ont été saisies dans le format standard Input
- Pour exécuter une cellule, il faut tapez *simultanément* Maj Entrée ou alternativement la touche Entrée qui se trouve dans le *pavé numérique*.
- Le signe % rappelle le dernier résultat calculé par *Mathematica*.
- Pour rappeler l'avant-dernier résultat (dans l'ordre *chronologique*), il faut taper %% . Et ainsi de suite.
- Plutôt que d'écrire %%%%%%%, on peut rappeler un résultat par Out[*iii*] (où *iii* est le numéro d'ordre tel qu'il apparaît sur l'écran) ou encore %*iii*
- Quand une instruction se termine par **;** (*point-virgule*), le résultat n'est pas affiché.

4 Nombres

- *Mathematica* connaît cinq types de nombres : *Integer* (entier), *Rational* (rationel non entier), *Real* (réel), *Complex* (complexe non réel) et *Boolean* (booléen).

- Les ensembles correspondants sont Integers (les entiers), Rationals (les rationels non entiers), Reals (les réels), Complexes (les complexes non réels) et Booleans (les booléens).
- 1. (avec le point) est interprété comme un réel.
- Tout entier ou rationnel dans une expression *contenant des réels* est automatiquement converti en réel, dès qu'il est évalué.
- 1+0. *i* est converti en 1.+0. *i* et interprété comme un complexe.

5 Opérations arithmétiques

- Les opérations +, –, ×, / s'écrivent comme vous en avez l'habitude. La multiplication peut alternativement s'écrire avec un espace ou le caractère × (équivalent à *).
- Les puissances s'écrivent avec ^ .
- ! est le signe factorielle ordinaire ($n! = n \times (n-1) \times \ldots \times 3 \times 2 \times 1$).
- !! est le signe double factorielle ordinaire ($n!! = n \times (n-2) \times .. \times (2 - \mathrm{Mod}(n, 2))$).
- Distribute force la distributivité d'une expression.
- Par exemple Distribute[(a+b).(c+d)]o donne a.c+b.c+a.d+b.d où le point est la multiplication matricielle, cf. § listes.
- Les parenthèses () s'utilisent comme d'ordinaire.

6 Variables

- Vous pouvez attribuer n'importe quel nom à une variable à deux restrictions près :
- Les caractères ***spéciaux*** sont interdits (par exemple que x+ ne convient pas).
- Un nom de variable ***ne peut commencer*** par un chiffre. Autrement dit, le premier caractère doit être une lettre.
- Par contre, les chiffres sont autorisés à partir du second caractère ; par exemple a1, a2, etc.

7 Affectations

- Le signe = est celui des informaticiens, il affecte la valeur qui est à sa droite dans la variable écrite à gauche. Pour cette raison, cette commande s'appelle une *affectation*.
- L'affectation est nécessaire pour mettre en mémoire le résultat de tout calcul, à de très rares exceptions près.
- Certes, par défaut, on peut toujours utiliser Out ou %, qui est une sorte d'affectation automatique.

  Toutefois, si plusieurs calculs se suivent sur une ligne, séparés par des ; (point-virgule) *seul* le dernier est mémorisé automatiquement.

a Affectation immmédiate

- Lorsque l'on écrit *symb* = *expression* , où le symbole *symb* définit une variable écrite selon les régles ci-dessus, *expression* , qui est quelconque, est immédiatement mémorisée dans la variable *symb*.
- Pour lire ce qui est mémorisé dans *symb*, il suffit d'écrire *symb* et d'exécuter la cellule.
- Par exemple, si l'on écrit f = a x^2 + b x + c, où initialement f, a, b, c et x sont vierges de toute affectation, f vaudra cette expression formelle de façon définitive.
- Toutefois, si on refait une affectation f =... en changeant l'expression de droite, elle *effacera* l'ancienne.
- Si, f restant non modifiée, on modifie les affectations de a, b ou c, et qu'on exécute f, on ne trouvera pas l'expression de départ a x^2 + b x + c mais une expression tenant compte des modifications de a, b ou c.

  Par exemple, si on fait trois affectations a = 1, b = 3 et c = – 5 puis qu'on exécute f on trouve x^2 + 3 x – 5

  Si on réaffecte maintenant f = a x^2 + b x + c, *avec* les dernières affectations a, b ou c, l'expression de f sera définitivement x^2 + 3 x – 5 et ne prendra pas en compte les changements ultérieurs d'affectation de a, b ou c.

b Affectation différée

- Lorsque l'on écrit *symb* := *expression* , cette *expression* n'est pas mémorisée dans la variable *symb*.
- L'écriture symbolique *seule* y est mémorisée, comme une chaîne de caractères, sans être interprétée.

- Par contre, à chaque fois que *symb* est exécuté, l'affectation est appliquée.
- Si, f restant non modifiée, on modifie les affectations de a, b ou c, et qu'on exécute f, on ne trouvera pas l'expression de départ a x^2 + b x + c mais une expression tenant compte des modifications de a, b ou c.

  Par exemple, si on fait trois affectations a = 1, b = 3 et c = – 5, puis qu''on exécute f on trouve x^2 + 3 x – 5

  Si on réaffecte maintenant f := a x^2 + b x + c, *avec* les dernières affectations a, b ou c, l'expression de f sera quand même a x^2 + b x + c et prendra en compte les changements ultérieurs d'affectation de a, b ou c.
- Le noyau prend en compte les valeurs de a, b et c seulement au moment où f est exécutée.
- Si on refait une affectation f:=... en changeant l'expression de droite, elle *effacera* l'ancienne.
- Toutefois, si on fait alternativement des affectations immédiates **et** différées, les deux affectations risquent d'être mémorisées ensemble. Pour résoudre le conflit, le noyau applique la plus ancienne.

8 Commandes de base

a Syntaxe générale des commandes

- Les caractères [et] sont réservés aux arguments des commandes

  Ils sont obligatoires, à l'exception des affectations = ou := , de l'effacement d'une mémoire =. (égal point) expliqué en dessous et de la syntaxe //
- S'il y a plusieurs arguments, ils sont séparés par des virgules, selon command[*arg1*,*arg2*]
- Le nombre d'arguments peut être variable.
- En particulier, les options sont des arguments facultatifs, qui sont toujours écrits en dernier.
- La plupart des options sont écrites sous forme de règles de substitution (voir après).
- *arg*//Command est équivalent Command[*arg*]. Cette écriture n'est possible que pour un *seul* argument.

b Effaçage

- Pour effacer la mémoire d'une variable a7, il suffit d'exécuter Clear[a7]
- L'instruction a7=**.** (attention au *point*) est une variante de Clear[a7]
- Pour effacer la mémoire de plusieurs variables a1, a2, ..., on écrit Clear[a1, a2, ...]

c Simplification

- *Mathematica* ne simplifie pas toujours les expressions formelles.
- Pour simplifier un résultat, on peut utiliser Simplify ou FullSimplify (plus puissant mais plus long, surtout la *validité* du résultat n'est pas garantie par *Mathematica*).
- Par exemple, si on exécute un ordre et que le résultat est mal simplifié, immédiatement après on exécute Simplify[%]
- FunctionExpand permet des simplifications plus puissantes, utilisant des fonctions de base.
- Refine permet de simplifier en tenant compte de la nature (réel, entier, positif, etc.) de l'argument.
- On peut utiliser l'option Assumptions avec Simplify et FullSimplify

  Assumptions→*condition booléenne* permet d'imposer des conditions et de simplifier éventuellement le résultat.

  Par exemple, on peut préciser que une variable est réelle, entière, positive, etc.
- Refine permet de simplifier en tenant compte de la nature supposée ou imposée (réel, entier, ...) de l'argument.

  Sa syntaxe utilise Assumptions, comme Simplify.
- Refine utilise une logique mathématique tandis que Simplify utilise une logique esthétique.

  Dans la majorité des cas, ces logiques conduisent au même résultat.
- Limit[*expression* , *var* → v_0] donne la limite de l'expression quand la variable tend vers v_0.

9 Constantes prédéfinies

- De nombreuses variables sont prédéfinies : Pi (ou π), E (ou ⅇ), I (ou ⅈ), Infinity (ou ∞) , EulerGamma, ...
- Voir aussi les constantes booléennes plus loin.

10 Information concernant une variable

- ? var (ou Definition[var]) donne l'information connu sur le symbol var
- Il existe également FullDefinition[var]) et Information[var]

B Fonctions

1 Fonctions avec une variable implicite

- Si on définit polynome $= x - 2x^2 + x^4 - 3x^7$, on dit que x est une variable implicite (ou externe).
- Pour connaître la valeur de polynome quand x vaut 3, il est nécessaire d'utiliser une règle de substitution : voir après.

2 Fonctions d'une ou plusieurs variables

a Comment définir une fonction avec une variable

- Les fonctions avec arguments sont des commandes. Leur arguments doivent être placés entre [ ]
- Pour définir une fonction fonc, on utilise une affectation immédiate fonc[x_] = *expression_avec*_x ou différée fonc[x_] := *expression_avec*_x.

  Par exemple f2[x_] = x^2
- Le caractère _ (souligné) s'utilise exclusivement quand on définit une fonction.
- Si l'affectation est immédiate, x doit être un symbole vierge de toute affectation.
- Si elle est différée, une affectation préalable de x sera sans influence.
- On *ne* peut *pas* définir une fonction fonc[x] si le symbole fonc est *déjà* défini. Il faut le nettoyer préalablement (cf. effacement).
- On peut donner *plusieurs* arguments dans la définition d'une fonction. Par exemple fonc[v1_,v2_]=...
- On peut définir *deux* fonctions ayant le *même* nom si le *nombre* d'arguments est *différents*. Il n'y a pas de risque de confusion.
- Il peut exister des paramètres implicites dans la définition d'une fonction.

  Par exemple fa[x_] = a x + 1, au lieu de définir fa[x_, a_] = a x + 1

b Comment rappeler une fonction

- Pour utiliser une fonction, il ne faut plus écrire _.

  Par exemple, on écrit fonc[t] ou fonc[3] ou fonc[2.151]
- x//fonc est équivalent fonc[x]. Syntaxe réservée à *un* argument et *pas* lors de la définition.

3 Définition avec resctrictions

a Resctriction par type de variable

- On restreint une fonction fr aux nombres réels en écrivant, lors de sa définition, fr[x_*Real*] = *expression* ou fr[x_*Real*] := *expression*
- On restreint une fonction fr aux nombres rationnels en écrivant, lors de sa définition, fr[x_*Rational*] = *expression* ou fr[x_*Rational*] := *expression*
- On restreint une fonction fr aux nombres entiers en écrivant, lors de sa définition, fr[x_*Integer*] = *expression* ou fr[x_*Integer*] := *expression*
- On peut utiliser une alternative Alternatives qui s'écrit avec |

  Par exemple, pour restreindre une fonction fr aux nombres entiers *ou* rationnels, on écrit, lors de sa définition, fr[x_*Integer*|_*Rational*] = *expression* ou fr[x_*Integer*|_*Rational*] := *expression*

- On peut restreindre une fonction aux chaînes de caractères (voir plus loin) en écrivant, lors de sa définition, fr[x_*String*] = *expression* ou fr[x_*String*] := *expression*

  Cependant, *String* n'est pas un type de nombre et il n'existe pas d'ensemble Strings

b Restriction conditionnelle

- On peut restreindre une fonction fr en imposant une condition booléenne, lors de sa définition; fr[x_/;*bool*] = *expression* ou fr[x_/;*bool*] := *expression*.

  Par exemple *bool* peut être x > 0 ou EvenQ[x]
- x_*Real* est équivalent à x_/; x∈Reals
- x_*Rational* est équivalent à x_/; x∈Rationals
- x_*Integer* est équivalent à x_/; x∈Integers
- x_*Integer*|_*Rational* est équivalent à x_/; x∈Integers || x∈Rationals

c Fonction définie par morceaux à l'aide de restrictions

- Pour définir une fonction par morceaux, on peut la restreindre par intervalle.

  Par exemple, on écrit f[x_/;x≥0]=Sqrt[x] puis f[x_/;x<0]=0
- Pour définir une fonction par morceaux, on peut aussi utiliser un test dans sa définition (cf la section sur les tests logiques).

  Par exemple, on écrit f[x_]:=If[x≥0,Sqrt[x],0] où vous remarquerez l'affectation différée ;

  certes, l'exécution immédiate est possible, mais il est préférable de différer l'exécution dès qu'une commande contient un If.

d Comment définir une fonction pour des valeurs définies

- Il est permis de définir une fonction pour des valeurs définies.

  Par exemple fonc[2]=*expression*
- L'argument peut être formel, par exemple fonc[a]=*expression*

  Ce dernier exemple doit être réservé à des usages très spécifiques, a n'étant pas variable.
- On peut définir à la fois une fonction avec une variable *et* avec des valeurs définies

  Dans ce cas, il faut se prémunir contre les conflits, avec des restrictions conditionnelles, voir §b précédent.
- L'utilisation des fonctions définies avec des valeurs définies est assez courant quand les arguments sont entiers ou rationnels.
- On peut définir une dérivée pour une valeur définie.

  Par exemple fonc'[2]=*expression*.
- On peut prolonger analytiquement une fonction.

  Par exemple, on définit sinc[x_/;x≠0]=Sin[x]/x; puis sinc[0]=1; et sinc'[0]=0; on obtient une fonction continue en 0, dont la dérivée est aussi continue en 0.

  Par comparaison, la fonction Sinc n'est pas définie en 0, ni sa dérivée Sinc'. Pour retrouver les bonnes valeurs pour x=0, on exécute Limit[Sinc[x],x→0]

4 Information concernant une fonction

- ? fonc (ou Definition[fonc]) donne l'information connue sur fonc. Il ne faut écrire *aucun* argument.

5 Effacement d'une fonction

- Clear[fonc] efface l'information connu sur la fonction fonc. Il ne faut écrire *aucun* argument.

  On peut utiliser un effacement multiple comme pour les variables en général.

  On peut également mélanger les symboles liés à des fonctions avec tout type de variable.
- ClearAll[fonc] efface l'information et les attributs (Protected, Listable, etc.) d'une fonction. Il ne faut écrire *aucun* argument.

- Pour effacer l'affectation d'une fonction définie pour une valeur définie (par exemple, 2 ou a), on *doit* écrire fonc[2]=**.** ou fonc[a]=**.**

6 Fonctions prédéfinies agissant sur les nombres réels ou complexes

- Un grand nombre de fonctions de la variable réelle sont prédéfinies par *Mathematica.*
- Il est indispensable de respecter majuscules et minuscules car les fonctions prédéfinies commencent toutes par une majuscule.
- La syntaxe de toutes ces fonctions est fonc[r]
- Un grand nombre de fonctions de la variable réelle sont prédéfinies par *Mathematica.*

a Fonction analytique définie sur ℂ

- Sqrt[r] (ou $\sqrt{r}$) est la racine carré de r.
- Abs est le module pour tous les complexes, la valeur absolue pour les réels.
- Exp est l'exponentielle et Log le logarithme népérien.
- Toutes les fonctions trigonométriques sont connues : Sin, Cos, Tan, Cot, ArcSin, ArcCos, ArcTan, ArcCot, ...
- Les fonctions hyperboliques sont également connues : Sinh, Cosh, Tanh, Coth, ArcSinh, ArcCosh, ArcTanh, ArcCoth, ...
- Sinc est le sinus cardinal, défini par sinc(x)=sin(x)/x.
- RootApproximant[r] donne le plus proche nombre algébrique d'un réel ou d'un complexe r.

  Les nombres algébriques sont les zéros des polynômes à coefficients entiers. Les nombres transcentants (comme e ou π) sont par définition les nombres non algébriques.

  Un argument optionnel la puissance maximale du polynôme dont le zéro est l'approximant de r.

  Le critère de proximité peut être modifié en option (voir le Help).
- Un cas particulier de nombre algébrique sont les nombres rationnels. RootApproximant[r,1] donne l'approximation rationnelle d'un réel ou d'un complexe r.
- L'*argument* de toutes ces fonctions peut être complexe.

b Fonction analytique définie sur ℝ

- Round donne l'entier le plus proche. Round[0.5] et Round[–0.5] donnent 0.
- Floor donne la partie entière. Floor[0.5] donne 0 et Floor[-0.5] donne –1.
- Ceiling donne la partie entière supérieure. Ceiling[0.5] donne 1 et Ceiling[-0.5] donne 0.
- Max[x_1 , x_2 , ...] donne le plus grand élément parmi x_1 , x_2 , ...
- Min[x_1 , x_2 , ...] donne le plus petit élément parmi x_1 , x_2 , ...
- BaseForm[r,n] donne la décomposition d'un nombre réel r en base n.

  Le résultat est non évaluable.n doit être un nombre compris entre 2 et 36.
- RealDigits[r,n] donne la décomposition d'un nombre réel r en base n sous format de liste.
- BaseForm et RealDigits ne fonctionnent pas sur les nombres exacts (comme π, $\sqrt{2}$, sin(1), etc.).

7 Fonctions prédéfinies agissant sur les nombres algébriques, rationnels ou entiers

a Action sur les nombres algébriques

- MinimalPolynomial[a,x] donne le plus petit polynôme[x] dont a est racine.

  x doit être un symbole vierge de toute affectation au moment de l'exécution et a un nombre algébrique.

b Action sur les nombres rationnels

- Cancel réduit une fraction exprimée avec des variables formelles.

Attention, les nombres rationnels sont automatiquement réduits, sans Cancel

Idem pour les nombres entiers *en facteur* à la fois au numérateur et dénominateur d'une fraction formelle.

Cela n'est pas le cas si le numérateur ou le dénominateur sont développés.

- Numerator donne le numérateur (réduit selon les règles précédentes) d'une fraction.
- Denominator donne le dénominateur (réduit selon les règles précédentes) d'une fraction.
- RealDigits fonctionne sur les nombres rationnels.

c Action sur les nombres entiers

- Mod[e,n] donne le reste de la division de e par n.
- Quotient[e,n] donne la partie entière (ou quotient) de e/n.
- FactorInteger[e] donne la décomposition en facteurs premiers p^i de tout nombre entier e.
  Le résultat est sous forme d'une liste de liste, dont chaque sous-liste s'écrit {*p*,*i*}
- IntegerExponant[e,b] donne la puissance de l'entier b (premier ou non) entrant dans la décomposition de l'entier e.
  Le résultat est un entier, qui peut être 0.
- IntegerExponant[e] donne la puissance de 10 entrant dans la décomposition de l'entier e.
  c'est le nombre de zéro dans la décomposition décimale de e.
- IntegerDigits[e,b] donne la liste des coefficients de l'entier e dans sa décomposition dans la base b.
- IntegerDigits[e] donne la liste des coefficients de l'entier e dans sa décomposition dans la base 10.
  Par exemple, IntegerDigits[102301] donne {1,0,2,3,0,1}
- IntegerLength[e,b] donne la longueur de e dans la base b, soit le cardinal de IntegerDigits[e,b]
- IntegerLength[e] donne la longueur de e dans la base 10, soit le cardinal de IntegerDigits[e]
- FromDigits[list,b] recompose l'entier dont la décomposition dans la base b est donné par list
- FromDigits[list] recompose l'entier dont la décomposition dans la base 10 est donné par list
  FromDigits ne contrôle pas que les entiers de list soient bien compris entre 0 et b y compris quand b est 10.
  Au lieu, il exécute $\sum_{i=0}^{n} l_i\, b^i$, où n est la longueur de la liste et l_i les entiers qui la composent.
- BaseForm et RealDigits agissent sur les nombres entiers.

8 Fonctions prédéfinies agissant sur les polynômes et les fractions polynômiales

a Action sur les polynômes

- Factor[polynome] factorise un polynôme.
  Le résultat de Factor est exact quelque soit le degré du polynôme quand les coefficients sont des nombres.
  Le résultat de Factor est moins performant quand il existe des coefficients formels.
- Expand[polynome] développe un polynôme.
- Apart[polynome] décompose un polynôme en éléments simples.
  Son action est plus complète que Expand
- Contrairement à Factor, on peut indiquer les variables dans Expand et Apart
- ExpandAll[polynome] réitère le développement dans les termes obtenues par Expand

b Action sur les fractions polynômiales

- Cancel réduit une fraction polynômiale au plus petit dénominateur.
- Numerator et Denominator agissent aussi sur les fractions polynômiales sans appliquer Cancel
- CoefficientList[polynome,x] donne la liste des coefficients $\{a_0, a_1, \ldots, a_n\}$ d'un polynôme = $a_0 + a_1\, x + a_2\, x^2 + \ldots + a_n\, x^n$, dans cet ordre.
- PolynomialRemainder[p,q,x] donne le reste de la division polynomiale de p[x] par q[x].
- PolynomialQuotient[p,q,x] donne le quotient de la division polynomiale de p[x] par q[x].

- x doit être un symbole vierge de toute affectation au moment de l'exécution, pour les trois dernières commandes.

9 Fonctions prédéfinies agissant sur les fonctions trigonométriques

- TrigFactor[trig] factorise en utilisant les formules de trigonométrie. On peut alternativement utiliser Factor avec l'option Trig→True
- TrigExpand[trig] développe en utilisant les formules de trigonométrie. On peut alternativement utiliser Expand avec l'option Trig→True

10 Dérivées

a Dérivées totales

- La dérivée d'une fonction à une variable f[x] s'écrit f'[x] ou encore D[f[x],x] ou encore ∂_xf[x] ou encore Derivative[1][f][x]

  Exemple : pour calculez la dérivée Sinc', il suffit d'écrire un ' avant les crochets , soit Sinc'[x]
- La dérivée seconde d'une fonction à une variable f[x] s'écrit f''[x] ou encore D[f[x],{x,2}] ou encore $\partial_{x,x}$f[x] ou encore Derivative[2][f][x]
- Avec D, la variable x doit être un symbole vierge de toute affectation au moment de l'exécution.

b Dérivées partielles

- La dérivée partielle d'une fonction à plusieurs variables s'écrit en généralisant ces écritures, sauf le prime qui est réservée aux dérivées totales.

  Soit une fonction f[x,y] à deux variables, la dérivée $\frac{\partial^3 f}{\partial x\, \partial^2 y}$ s'écrit D[f[x,y],x,{y,2}] ou encore $\partial_{x,y,y}$f[x,y] ou encore Derivative[1,2][f][x,y]

11 Intégrales, primitives et autres fonctionnelles

a Intégrales

- Integrate[f[x],{x, x_1,x_2}], où x_1 et x_2 sont les bornes d'intègration, est un ordre d' intégration formelle, basée sur une bibliothèque riche et sur l' algèbre formelle de *Mathematica*.

  Assumptions→*condition booléenne* permet d'imposer des conditions et de simplifier éventuellement le résultat.
- NIntegrate[f[x],{x, x_1,x_2}], où x_1 et x_2 sont des réels, est un ordre d' intégration numérique.

  NIntegrate utilise différentes méthodes numériques, qui sont choisis automatiquement ou explicitement.

b Primitives

- Integrate[f[x],x] est un ordre de primitive (sans constante d'intégration).

c Intégrales ou primitives de fonctions implicites

- On peut utiliser une fonction implicite dans Integrate, à condition d'utiliser la même variable que dans sa définition.
- On peut utiliser une fonction implicite dans NIntegrate, à condition d'utiliser la même variable que dans sa définition.
- Les variables implicites doivent être réelles dans NIntegrate (voir substitution et syntaxe avancée).

d Autres fonctionnelles

- Une fonctionnelle est une fonction agissant sur des fonctions, comme la dérivation, l'intégration, la primitivation, les transformations de Fourier, de Laplace, etc.
- Sont définies FourierTransform (et quelques variantes) et LaplaceTransform

12 Séries entières

- La série entière d'une fonction s'écrit Series[f[x],{x, x_0, n}], où l'argument x_0 est l'abscisse autour de laquelle on fait le développement et n l'ordre choisi.

- La sortie de Series comporte le reste sous forme du O de Landau.
- Pour supprimer le reste et obtenir une fonction normale, il faut faire Normal[%], ou directement Normal[Series[f...]]
- On peut choisir ∞ ou $-\infty$ pour x_0.
- Il est impossible de choisir n infini.

13 Fonctions instrinsèques

a Comment définir une fonction intrinsèque

- Les fonctions intrinsèques sont écrites sans paramètre.
- À la place, on utilise un paramètre muet # ou #1, #2, ..., pour les fonctions à plusieurs variables.
- On écrit le symbole dea fonction sans arguments à *gauche* d'une affectation, et on écrit sa définition à sa *droite*, en finissant obligatoirement par le caractère &.
- On peut inclure des paramètre ordinaires, soit implicites soit explicites, dans la définition.

b Comment utiliser une fonction intrinsèque

- Pour lui attribuer un argument t, on écrit fonc[t]
- Si la fonction est intrinsèque mais avec une variable explicite, par exemple fonc[α_]=Sin[α #]&, on écrit fonc[σ][t]

c Comment dériver une fonction intrinsèque

- La dérivée d'une fonction intrinsèque s'écrit fonc'
- Si on ajoute un argument, on écrit fonc'[t]
- Si la fonction comporte une variable explicite, on écrit, par exemple, fonc[σ]'[t]

14 Composition des fonctions

- Il n'y a pas de commande spécifique pour la composition de deux fonctions.
- La composition des fonctions à une variable f°g s'écrit f[g[x]]

a Composition répétée d'une fonction

- Pour composer une fonction f un nombre n fois avec elle-même, il faut écrire Nest[f,x,n] où x est la variable.
- f doit être une fonction intrinsèque dans cette syntaxe.

b Point fixe d'une fonction

- Le point fixe d'une fonction f s'écrit FixedPoint[f,x] où x est la variable.
  f doit être une fonction intrinsèque dans cette syntaxe.
- La série $f^{\circ n}$ doit converger de façon exacte (ou à mieux que 14 décimale si la suite est numérique).
- Pour éviter une boucle infinie, on écrit un troisième argument n, qui limite à n composition, cf. Nest

---

C Précision numérique

1 Nombre de décimales d'un réel

- *Mathematica* calcule avec une précision par défaut de 16 chiffres exprimés.
- Dans certaines circonstances, cette précision passe automatiquement à 32 ou plus.
- Pour connaître les vraies décimales connues d'un nombre x, on peut soit exécuter FullForm[x], soit copier/coller l'expression de x

2 Modification de l'affichage d'un nombre

a Syntaxe de N

- N[*nbre*, *digits*] donne la valeur numérique d'un nombre, avec *digits* décimales

  Les *digits* décimales sont comptées à partir de la première non nulle.
- N[*nbre*] donne la valeur numérique d'un nombre, avec 6 décimales par défaut.
- Plus précisément, la précision affichée (6 pour N[*nbre*]) est un champ enregistré de tout nombre, qui apparaît si on exécute FullForm

b Action de N sur les nombres réels

- N transforme les nombres exacts en approximation numérique, par exemple N[Pi]
- Attention, la précision réelle est différente du nombre de chiffres affichés.

  Par exemple, N[Pi] est affiché avec 6 chiffres, bien que, par défaut, la précision soit de 16.
- Quand on fait varier la précision affichée, la précision peut augmenter.

  Par exemple, N[Pi,13] affiche 3.141592653590 mais la précision passe de 16 à 32 chiffres.
- Sont enregistrés un très grand nombre de chiffres cachés (9=16-7 pour N[Pi]).
- On peut exécuter FullForm pour connaître la précision connue.

c Action de N sur les nombres entiers ou rationnels

- N[*nbre*] transforme un entier en réel. Par exemple N[1] affiche 1. (le *point* est dans l'affichage).
- N[*nbre*] transforme un nombre rationnel en réel.

d Action de N sur les symboles non affectés

- N[*nbre*] n'agit pas sur les objets formel.

Nombres négligeables

- Chop remplace les valeurs *numériquement* trop petites (*non pertinentes d'un point de vue numérique*) par 0.

  Par défaut, Chop élimine les nombres $< 10^{-10}$. On peut choisir la tolérance et écrire Chop[*nbre*, *tol*]

D Listes et matrices

1 Listes

- Une liste de n éléments s'écrit $\{e_0,..,\ e_n\}$, chaque élément e_i est séparé par une virgule
- La liste {} est la liste vide.

2 Matrices

- Pour créer des matrices carré ou rectangulaires, il suffit d'emboîter des listes.

  Une matrice est une liste de ligne (et non de colonne, comme cela aurait été logique).

3 Création automatique de listes

- Pour une liste simple (vecteur colonne), la syntaxe est Table[*terme_i*,{i,*idebut*,*ifin*}] où *terme_i* est une fonction de i, *idebut* et *ifin* les bornes min et max.

  On peut écrire {i,*ifin*}, où la borne min 1 est *implicite*.

  On peut écrire {i,*idebut*,*ifin*,*pas*}, où *pas* est le pas d'itération.

  On peut écrire {i,*liste_d'indices*}, où *liste_d'indices* est une liste d'entiers quelconque, pas nécessairement ordonnée.

- Si *idebut* > *ifin*, Table[*terme*_i,{i,*idebut*,*ifin*}] donne {}
- Si *ifin* – *idébut*, est de signe strictement opposé à *pas*, Table[*terme*_i,{i,*idebut*,*ifin*,*pas*}] donne {}
- Pour une liste à deux indices (matrice), la syntaxe est Table[*terme*_ij,{i,*idebut*,*ifin*},{j,*jdebut*,*jfin*}] où *terme*_ij est une fonction de i et j.

  Le premier indice correspond aux lignes, le second aux colonnes.
- On peut de même créer des listes à trois, quatre, ..., indices.

4 Listes et matrices prédéfinies

- La liste Range[n] est la liste {1,2,...,n}.

  On peut préciser le départ avec Range[n_0, n_1] qui donne {n_0,..,n_1} et le pas *di* avec Range[n_0, n_1, *di*]
- La liste ConstantArray[c,*n*] est la liste {c, c,..., c}, qui est de longueur *n*.
- La matrice ConstantArray[c,{n_1,n_2,...}] est la matrice $n_1 \times n_2 \times$... de coefficient constant c.
- La matrice IdentityMatrix[*n*] est la matrice identité $n \times n$.

5 Extraction, modification ou suppression

a Pour extraire un élément d'une liste

- Part (ou [[ ]]) permet de créer une sous-liste à partir d'une liste.
- Pour extraire le $i^{\text{ème}}$ élément d'une liste simple vec, il suffit d'écrire vec[[*i*]] ou Part[vec,*i*] ou Take[vec,{*i* }]
- Pour retirer les accolades d'une liste elem à *un* élément, on peut utiliser elem⟦1⟧ ou First[elem]
- Pour extraire le dernièr élément d'une liste vec, il suffit d'écrire vec[[-1]] ou Last[vec]
- Pour obtenir l'avant-dernier élément, il suffit d'écrire vec[[-2]] , et ainsi de suite.
- Pour extraire l'élément *ij* ($i^{\text{ème}}$ ligne, $j^{\text{ème}}$ colonne) d'une matrice m, il suffit d'écrire m[[*i*]][[*j*]] ou m[[*i* , *j*]] avec une virgule.

b Pour extraire une sous-liste

- Span (ou ;;) permet de créer une sous-liste à partir d'une liste.
- Pour obtenir la liste du $i^{\text{ème}}$ au $j^{\text{ème}}$ élément d'une liste vec, on écrit vec[[*i* ;; *j*]], ou Take[vec,{*i*, *j* }]
- Pour choisir les éléments par sauts de *di*, il faut écrire vec[[*i* ;; *j* ;; *di*]]. ou Take[vec,{*i*, *j*,*di* }]

  Par exemple, on extrait les termes impairs de Range[10] en exécutant Range[10][[1;;10;;2]]
- On peut généraliser Span en remplaçant *i* ;; *j* ou *i* ;; *j* ;; *di* par une liste {i_1, i_2, ...}
- Quand on le rencontre dans Span , All signifie 1;;-1 (=du premier au dernier).

  Par exemple, soit list={{x_1, y_1},...,{x_n, y_n}} est une liste de bipoints, list[[All,1]] est la liste des abscisses {x_1,...,x_n} et list[[All,2]] est la liste des ordonnées {y_1,...,y_n}.
- Pour extraire la $i^{\text{ème}}$ ligne d'une matrice m, il suffit d'écrire m[[*i*]]
- Pour extraire la $i^{\text{ème}}$ colonne d'une matrice m, il suffit d'écrire Transpose[m][[*i*]] ou m[[All,*i*]]
- Pour extraire les *i* premiers éléments d'une liste simple vec, il suffit d'écrire Take[vec,*i*]
- Pour extraire les *i* derniers éléments d'une liste simple vec, il suffit d'écrire Take[vec,–*i*]

c Pour supprimer un élément ou une sous-liste

- Pour supprimer le $i^{\text{ème}}$ élément d'une liste simple vec, il suffit d'écrire Drop[vec,{*i*}]
- Pour supprimer les *i* premiers éléments d'une liste simple vec, il suffit d'écrire Drop[vec,*i*]
- Pour supprimer les *i* derniers éléments d'une liste simple vec, il suffit d'écrire Drop[vec,–*i*]

d Pour ajouter des éléments

- Pour ajouter un élément x à la fin d'une liste vec, on peut exécuter Append[vec, x]

- AppendTo[vec, x] réalise la même opération et met le résultat en mémoire dans vec
  C'est un cas exceptionnel où l'affectation est automatique.
- Pour ajouter un élément x au début d'une liste vec, on peut exécuter Prepend[vec, x]
- PrependTo[vec, x] réalise la même opération et met le résultat en mémoire dans vec
  C'est un cas exceptionnel où l'affectation est automatique.
- Pour ajouter un élément x à la position n d'une liste vec, on exécute Insert[vec, x,n]
  n est la position de x dans la liste obtenue.
- Pour une insertion multiple, on utilise Insert[vec, x, {{ n_1},{ n_2},...}] où les n_i sont les positions où l'on insère x
  L'interprétation des numéros n_i est analogue au cas où on insère une seule fois x
- Pour une insertion dans une matrice list, on utilise Insert[list, x, {m,n}] où {m,n} est la position où l'on insère x.
  m est l'indice des lignes et n celui des colonnes.
- Pour une insertion multiple dans une matrice list, on utilise Insert[list, x, {{m_1, n_1},{m_2, n_2},...}] où les {m_i,n_i} sont les positions où l'on insère x.

e Pour modifier un élément ou une sous-liste

- On peut modifier une liste élément par élément.
- Si on veut modifier une liste simple vec, on peut faire une affectation du type vec[[1]] = *val*.
- Si on veut modifier une liste double mat, on fait une affectation du type mat[[1, 1]] = *val*.
- Le choix des indices est complètement libre, tant que l'élément affecté existe préalablement.

6 Opérations matricielles

a Produit matriciel

- Le point est le signe de multiplication entre matrices ou entre matrice et vecteur.
- Ce produit s'appelle également Dot
- Le produit scalaire est un cas particulier de produit matriciel.
- La norme d'un vecteur u se calcule par Norm[u] (on peut utiliser $\sqrt{\text{u.u}}$ si les composantes sont réelles).

b Opérations générales

- La commande Transpose opère la transposition d'une matrice.
- La trace d'une matrice m se calcule avec Tr[m] .
- Le déterminant d'une matrice m se calcule avec Det[m]

c Valeurs et vecteurs propres

- Les valeurs propres d'une matrice m se calculent avec Eigenvalues[m]
- Si une valeur propre est multiple, elle est dupliquée autant de fois que sa multiplicitée.
- Les vecteurs propres d'une matrice m se calculent avec Eigenvectors[m] .
- Ils sont classés dans le même ordre que les valeurs propres calculées par Eigenvalues
  Autrement dit le *premier* vectors propre correspond à la *première* valeur propre et ainsi de suite.
- Eigensysteme exécute Eigenvalues et Eigenvectors *simultanément* sous forme d'une liste.
  Le premier élément est la liste des valeurs propres et le second la liste des vecteurs propres.
  Eigensystem[m]⟦1⟧ est donc équivalent à Eigenvalues[m] et Eigensystem[m]⟦2⟧ à Eigenvectors[m]

7 Manipulation de listes

a Opérations donnant un résultat réel

- Length[list] donne la longueur d'une liste telle qu'elle est lue au premier rang.
- Max et Min opèrent sur une liste de réels.

b Ordonner des listes

- Sort[list] remet les éléments d'une liste dans l'ordre.
- On peut préciser l'ordre avec un critère Sort[list,*crit*] où *crit* est une fonction intrinsèque à deux arguments *expr*(#1,#2)&
- On peut choisir Less, qui ordonne selon <, Greater selon >, LessEqual selon ≤, GreaterEqual selon ≥.

  Less correspond à #1≤#2&.

  Par exemple, pour classer par ordre de |x| croissant et, pour deux |x|=|y| égales, du négatif *éventuel* au positif *éventuel*. La fonction intrinsèque à **deux** variables Abs[#1]≤Abs[#2]& convient. Concernant la **deuxième** prescription en cas d'égalité, si la liste y est conformément préordonnée par Sort, cet ordre est préservé par Abs[#1]≤Abs[#2]& tandis qu'il est inversé par Abs[#1]<Abs[#2]&
- Par défaut, l'ordre appliqué OrdredQ mélange numérique et alphabétique, ce qui peut le rendre obscur.
- Ordering est l'ordre d'une liste induit par Sort. Les listes Sort[list] et list[[Ordering[list]]] sont égales.

  Cette syntaxe ne fonctionne que si Sort est utilisée sans *crit* particulier.

  Sinon, il faut écrire Ordering[list,All,*crit*]

  Par exemple, si on réordonne Eigenvalues[m] avec Sort, les vecteurs propres donnés par Eigenvectors[m]⟦ Ordering[Eigenvalues[m] ⟧ sont dans l'ordre des valeurs propres données par Sort[Eigenvalues[m]]

c Sélectionner des éléments

- Pour sélectionner des éléments d'une liste, avec un critère, on écrit Select[list,*crit*] où *crit* est une fonction intrinsèque comme dans Sort
- Par exemple, pour trouver les nombres *pairs* ≤ n, on écrit Select[Range[0,n],EvenQ]
- Un ordre différent est Cases dont la syntaxe est Cases[list,*type*] où *type* est le type des éléments de la liste, au même sens que pour les restrictions de fonctions.

  type peut être un élément explicite, formel ou exprimé. Par exemple Cases[{1,2,3,4},2] donne {2}

  type peut être une variable. Par exemple, pour sélectionner les éléments entiers, on écrit Cases[list, _Integer]
- L'utilisation de Alternatives s'avère utile. Par exemple, pour sélectionner des éléments entiers *ou* rationnels, on écrit Cases[list, _Integer|_Rational]
- On peut indiquer une structure. Par exemple, pour sélectionner les listes à deux éléments, on écrit Cases[list, {_,_}]

  On peut ajouter un troisième argument, qui précise à quel niveau d'accolade Cases s'applique.

  La syntaxe de cet argument est identique à celle de Flatten
- La combinaison de ces possibilités fait de Cases une commande puissante, complémentaire de Select

  De plus, d'atures syntaxes s'applique à Cases qui sont étudiées dans la section Syntaxe avancée.

d Concaténation

- Join[list1, list2] assemble les listes dans l'ordre dans lequel on les écrit.
- Union[list1,list2] supprime tous les éléments dupliqués et les ordonne selon OrdredQ
- Union[list] , quand on l'applique sur une *seule* liste, est équivalente à Sort[DeleteDuplicates[list]]
- On peut modifier le critère de tri dans Union, avec l'option SameTest→*crit*

  Par exemple, on recherche les dégénérescences des valeurs propres d'une matrice m en comparant Eigenvalues[m] et Union[Eigenvalues[m]]

e Autres permutations

- RotateLeft[list] et RotateRight[list] font des permutations cycliques de liste.
- Reverse[list] inverse l'ordre d'une liste. Attention à ne pas confondre avec Transpose
- Flatten[list] élimine toutes les accolades des sous-listes pour ne former qu'une liste simple.
- Flatten[list,n] élimine les accolades des sous-listes du niveau 1 au niveau n
  (le niveau 0 étant celui de la liste elle-même).
- Partition[list,n] partitionne une liste en sous-listes de longueur n, fabriquant ainsi une matrice m×n où m est la partie entière de Length[list]/n.

  Les Length[list] – m n éléments restants sont supprimés.
- Partition[list,n,d] partitionne une liste en sous-listes de longueur n, chaque sous-liste ayant un recouvrement de d éléments avec la précédente.
- Partition[list, {n_1, n_2,...}] partitionne une liste en sous-matrices $n_1 \times n_2 \times$...
- Riffle[list1, list2] crée une liste où s'intercale alternativement les éléments de deux listes.
- Riffle[list1, x] intercale alternativement les éléments de la liste. avec x.
- Riffle[list1, x, d] intercale dans la liste l'élément x tous les d rangs.

8 Interpolation de listes réelles

a Interpolation polynomiale

- Interpolation[list], où list est une liste {f_1, f_2,...} de longueur n, est la fonction intrinsèque *interpolant* les points {1, f_1}, {2, f_1}, ..., {n, f_n}.
- Interpolation[list], où list est une liste de n couples {x_i, f_i}, est la fonction intrinsèque interpolant les points {x_1, f_1}, {x_2, f_2}, ..., {x_n, f_n}.
- On peut, en respectant une structuration stricte, définir des Interpolation dérivables, voire deux fois, trois fois, etc., dérivables. Consulter le Help.
- On peut utiliser Interpolation[list pour des fonctions à d variables, avec list à d + 1 éléments.

  Par exemple, quand d vaut 2, on doit utiliser une liste de {x_i, y_i, f_i}
- On peut modifier l'option InterpolationOrder→p , où p est l'ordre (entier) d'interpolation.
- Par défaut, p vaut 3.

  Dans certaines situations, cela peut aboutir à des aberrations. Il faut alors utiliser InterpolationOrder→1
- On peut modifier le résultats avec l'option Method→... , consulter le Help.
- InterpolatingPolynomial[list, x], génère un polynôme de degré n – 1 interpolant exactement list, composée de n points.
- La différence entre Interpolation et InterpolatingPolynomial est que les polynôme utilisés dans Interpolation n'interpolent qu'entre quelques points successifs.

b Interpolation par des fonctions généralisées

- Fit[list,{$fonc_1$,$fonc_2$,...},x] est une interpolation plus générale par des combinaisons linéaires de $fonc_1$, $fonc_2$, ..., qui sont des fonctions quelconques de x.

  x doit être un symbole vierge de toute affectation au moment de l'exécution.

  On peut utiliser Fit avec les mêmes types de listes que Interpolation, y compris à plusieurs dimension.
- FindFit[list,*fonc*,{a,b,...},x] est une interpolation générale par une fonction *fonc*, qui est une fonction de x quelconque dépendant des paramètres a, b, ...

  x doit être un symbole vierge de toute affectation au moment de l'exécution.

  On peut utiliser FindFit avec les mêmes types de listes que Interpolation, y compris à plusieurs dimension.

c Interpolation par des méthodes sophistiquées

- LinearModelFit[list,{*fonc*$_1$,*fonc*$_2$,...},x] a le même objectif que Fit

  x doit être un symbole vierge de toute affectation au moment de l'exécution.

  On ne doit pas préciser de fonction *fonc*$_1$ avec LinearModelFit tandis que c'est nécessaire avec Fit

- La syntaxe de LinearModelFit est très particulière. Si on exécute LinearModelFit["Properties"], on obtient une liste de variables *var*$_1$, *var*$_2$, ...

  Pour obtenir un résultat concret, il faut exécuter LinearModelFit["*var*$_i$"], en choisissant la variable *var*$_i$ à l'aide du Help.

- Il existe également une commande GeneralizedLinearModelFit

  La syntaxe de GeneralizedLinearModelFit est extrêmement compliquée, en particulier, les abscisses des données doivent être entières. Consulter le Help.

E Graphisme

1 Tracés de fonctions

a Comment tracer les courbes de fonction

- Pour tracer une fonction à une variable fonc, on écrit Plot[fonc[u], {u, x_1, x_2}], où u est un symbole quelconque (c'est une variable muette).
- Pour tracer une fonction implicite polynome, on écrit Plot[polynome, {x, x_1, x_2}], où x_1 et x_2 sont les bornes réelles de l'intervalle sur lequel on la trace et x est la même variable que dans la définition de polynome.
- Pour tracer une fonction à une variable fonc, ayant un second paramètre implicite a, on écrit Plot[fonc[u] /. $a \to a_0$, {u, x_1, x_2}], où a_0 est une valeur réelle que vous choisissez.
- Pour tracer *plusieurs* fonctions, il suffit de mettre une liste {f1,f2,...} à la place de la fonction, en adoptant les différentes syntaxes précédentes selon les fonctions. Les couleurs sont choisies par défaut.
- On peut utiliser Table pour générer les listes dans Plot
- Dans certains cas, il faut utiliser Evaluate pour que Plot s'exécute correctement.

b Comment tracer une fonction intrinsèque

- Pour tracer une fonction intrinsèque fonc, on écrit, par exemple, Plot[fonc[x] {x, −3, 3}]
- Pour tracer une fonction intrinsèque avec une variable explicite, on écrit, par exemple, Plot[fonc[π][x], {x, −3, 3}]

c Comment modifier le style

- Pour modifier le style (couleur, épaisseur, trait, ...), il faut insérer l'option PlotStyle→{*liste_de_style*}, où *liste_de_style* est une liste de couleurs ou d'épaisseurs ou etc.
- Les options *options* se placent toujours en fin de commande, {x, x_1, x_2}, *options* sans oublier une virgule.
- Les différentes couleurs prédéfinies sont Red, Green, Blue, Black, White, Gray, Cyan, Magenta, Yellow, Brown, Orange, Pink, Purple, LightRed, LightGreen, LightBlue, LightGray, LightCyan, LightMagenta, LightYellow, LightBrown, LightOrange, LightPink, LightPurple et Transparent
- Pour les couleurs, on peut utiliser Hue[n] (où n∈[0,1]) qui donne les couleurs de l'arc-en-ciel.
- Les couleurs de base sont RGBColor[n_{red},n_{green},n_{blue}] où $(n_{red}, n_{green}, n_{blue}) \in [0, 1]^3$ et CMYKColor[n_{cyan},$n_{magenta}$,n_{yellow},n_{black}] où $(n_{cyan}, n_{magenta}, n_{yellow}, n_{black}) \in [0, 1]^4$.
- On peut mélanger deux couleurs en utilisant Blend

  La syntaxe est Blend[{*col*$_1$, *col*$_2$}, t] où on prend une fraction t de *col*$_1$ et 1−t de *col*$_2$.

  D'autres syntaxes de Blend existent. Consulter le Help.

- Les ordres d'épaisseurs prédéfinis sont Thick et Thin
- Les épaisseurs de base sont Thickness[r] et AbsoluteThickness[r], qui s'utilise comme Thick et Thin

Dans les deux ordres précédents, on peut choisir comme argument Tiny, Small, Medium, Large (qui sont des grandeurs absolues).

- Les ordres de trait prédéfinis sont Dotted , Dashed et DotDashed
- Les ordres de base pour le trait sont Dashing[{r_1, r_2,...] et AbsoluteDashing[{r_1, r_2,...], dont la syntaxe ne sera pas détaillée (consulter le Help).
- Les options de PlotStyle sont attribuée dans l'ordre, *courbe à courbe*.

  Par exemple, si on écrit Plot[{f1, f2},PlotStyle→{Red,Thick}] la prermière courbe est en *rouge*, la seconde en trait ***épais***.
- Si on veut *plusieurs* options s'appliquant à une courbe donnée (d'une liste de courbes), il faut faire une sous-liste.

  Par exemple, avec PlotStyle→{{Red,Thick},...}] la première courbe est en rouge *et* trait épais.
- L'échelle $1_y/1_x$ est définie par l'option AspectRatio.

  En général, AspectRatio→Automatic donne un rapport 1.

  Par défaut, ce rapport est le GoldenRatio, qui vaut $\left(\sqrt{5}+1\right)/2$.

d Courbes de surfaces dessinées à trois dimensions

- Pour tracer la courbe tridimensionnelle d'une fonction à deux variables, on utilise Plot3D[f[x,y},{x, x_1, x_2}],{y, y_1, y_2}]
- Pour tracer les courbes de niveaux d'une fonction à deux variables, on utilise ContourPlot[f[x,y},{x, x_1, x_2}],{y, y_1, y_2}]
- Pour tracer des fonctions implicites définies par une équation , on utilise ContourPlot[*eq*[x,y]==0,{x, x_1, x_2}],{y, y_1, y_2}]
- Pour modifier le style dans ContourPlot, il faut insérer l'option ContourStyle→...

e Tracé de listes de points

- Si list est une liste simple, ListPlot[list] trace les points {1,list[[1]]}, {2,list[[2]]}, etc.
- Pour joindre les points, on peut, soit ajouter l'option Joined→True dans ListPlot, soit utiliser ListLinePlot

2 Dessins

- Pour faire des dessins, on dispose de la commande Graphics
- La syntaxe de Graphics est Graphics[*objet_graphique*] ou Graphics[*objet_graphique1*,*objet_graphique2*,...] ou Graphics[*ordre_graphique1*,*objet_graphique1*,*ordre_graphique2*,*objet_graphique2*,...]

a Objets graphiques élémentaires

- Pour générer une cercle, on écrit Circle[{x_0, y_0}, r] où {x_0, y_0} est le centre et r le rayon.
- Pour générer une ellipse, on écrit Circle[{x_0, y_0}, {r_1, r_2}] où {x_0, y_0} est le centre et {r_1, r_2} les deux demi-axes.
- Pour générer un disque, on écrit Disk[{x_0, y_0}, r] où {x_0, y_0} est le centre et r le rayon.
- Pour générer une ellipse pleine, on écrit Disk[{x_0, y_0}, {r_1, r_2}] où {x_0, y_0} est le centre et {r_1, r_2} les deux demi-axes.
- Pour générer un point, on écrit Point[{x_0, y_0}] où {x_0, y_0} sont les coordonnées du point.
- Pour générer une une collection de points, on écrit Point[{{x_1, y_1},{x_2, y_2},...}] où on écrit la liste des couples de leurs coordonnées.
- Pour générer une ligne, on écrit Line[{{x_1, y_1},{x_2, y_2},...}] où on écrit la liste des points par lesquels passe la ligne.

  L'ordre est primordial dans cette syntaxe.
- Pour générer plusieurs lignes, on utilise Line[...] en insérant une liste de collections de points.

  Chaque liste correspondant à une ligne, il y a 3 degrés d'accolades.
- Pour générer un rectangle plein, on écrit Rectangle[{x_1, y_1},{x_2, y_2}] où {x_1, y_1} et {x_2, y_2} sont les coordonnées de deux coins non adjascents.
- Pour générer un carré plein, on utilise Rectangle
- Pour générer un carré plein *unitaire*, on écrit Rectangle[{x_1, y_1}] où {x_1, y_1} est le coin en bas à gauche.

- Pour générer un polygone plein, on écrit Polygon[{x_1, y_1},{x_2, y_2},...] où {x_i, y_i} sont les coordonnées des points reliant les arrêtes.
- Pour générer un polyèdre plein, on utilise Polygon en écrivant des coordonnées tridimensionnelles {x_i, y_i, z_i}.
- Pour générer des rectangle, carré, polygones ou polyèdres *vides*, il n'existe pas d'ordre spécifique et il faut utiliser Transparent parmi les options de couleurs discutées ci-après.
- Les ordres Circle, Disk, Rectangle ne peuvent construire qu'un objet à la fois.
- Les ordres Point, Line, Polygon peuvent construire plusieurs objets à la fois. La syntaxe est celle décrite pour Point

b Ordres de couleur et d'épaisseur

- Dans Graphics : les ordres de couleurs, épaisseurs, etc., sont écrit au même niveau que les objets graphiques et agissent sur les objets qui les suivent, jusqu'au prochain ordre de couleur ou d'épaisseur qui les contredit.
- *Certaines* options cependant s'écrivent selon la syntaxe ordinaire des options, en fin de commande. C'est le cas de la couleur de fond, qu'on écrit en rajoutant Background→*une_couleur* après la ***dernière*** accolade.

  Ces options-ci sont compatibles avec Show décrit après.
- Les options de couleurs et de trait sont les mêmes que pour Plot

  Les options concernant le trait doivent être placés dans EdgeForm

  S'il n'y a qu'une option, on écrit EdgeForm[*option*]

  S'il y en a plusieurs, on les écrit en liste.

  Ces options concernent tous les ordres sauf Point
- Les options concernant la taille du point doivent être placés dans PointSize

  Les tailles de point prédéfinies sont Tiny , Small , Medium et Large

  La taille est définie basiquement dans PointSize par un réel >0.

3 Modification et juxtaposition de graphismes

a Show

- On peut rappeler une courbe ou un dessin par %
- Pour rappeler et combiner des courbes ou des dessins ensemble, on peut utiliser Show[*courb1*,*courb2*,*dessin1*, *dessin2*,...].
- Show est une commande *spécifique* pour le graphisme. On peut y inclure des options pour modifier les graphes.

  Pour retracer une courbe en modifiant certaines options, on peut utiliser Show

  Les options de couleurs et d'épaisseur de trait ne peuvent être modifiées par Show

  Toutefois, BaseStyle→ fonctionne. Son action ressemble à PlotStyle, utilisez Help.

b Overlay

- On peut aussi utiliser Overlay[*courb1*,*text1*,*dessin1*, *text2*,...]
- Overlay ne permet aucune modification des graphiques et des courbes.
- Overlay peut aussi bien juxtaposer des dessins que des Output textuels.

c Epilog

- L'option Epilog→... permet d'insérer un graphisme dans un Plot
- La syntaxe de Epilog est identique à celle de Graphics

F Affichage

1 Format d'affichage

- Certaines commandes s'appliquent uniquement à changer le mode d'affichage.

- TraditionalForm écrit les expressions sans majuscule, avec des () à la place des [], ...
- InputForm écrit les expressions comme si elles avaient été écrites *avant* leur exécution.
- OutputForm écrit les expressions tel que le résultat est écrit.

  C'est le format standard par défaut.
- FullForm écrit une expression sous une forme complète.

  Cette écriture est très lourde et n'est utilisée que pour des besoins spécifiques.
- MatrixForm écrit une liste sous forme matricielle.
- TableForm écrit une liste sous forme de tableau.

2 Mise en lignes ou colonne

- Pour les sorties multiples (de plusieurs résultats), on peut rassembler les résultats enf lignes ou colonnes.
- Row[{$elem_1$,$elem_2$,...}] ordonne les résultats en lignes.

  Toutefois, cela n'empèche pas que les résultats soient coupés à l'affichage si un passage à la ligne est nécessaire.
- Column[{$elem_1$,$elem_2$,...}] ordonne les résultats en colonnes.
- On peut emboîter Row et Column.
- Il existe un ordre cummulant lignes et colonnes : Grid[{{$elem_1$,$elem_2$,...}}] où chaque sous-liste correspond à une ligne.
- On peut régler l'alignement avec l'option Alignment→{{$align_1$,$align_2$,...}}] où sont les options d'alignement.

  Les options d'alignement sont Bottom, Center, Top, Right ou Left

  Si l'alignement est commun à toutes les lignes, on retire un seul degré d'accolades, comme pour les options de PlotStyle

3 Impression

- L'odre Print a pour syntaxe Print[{$elem_1$,$elem_2$,...}]
- L'alignement de la sortie correspond à un Row implicite.
- Print ne respecte pas les syntaxes standard. Au lieu, il provoque impérativement un affichage.
- L'affichage de Print est généralement dans le carnet courant.
- Lorsqu'il est utilisé à partir d'une autre fenêtre, l'affichage est, par défaut, dans la fenêtre de messages.
- On peut définir le carnet où se produit l'affichage.

4 Modification du style

- Le style peut être modifié à de nombreux niveaux : dans Print, il faut utiliser Style

  Style peut être utilisé dans presque toutes les commandes, y compris Graphics ou Epilog

  Style n'agit pas sur les courbes dans les commandes Plot
- Les options dans Style sont identiques aux options graphiques étudiées dans Plot et Graphics

G Logique booléenne

- *Mathematica* connaît les variables logiques de l'algèbre booléenne.

1 Constantes logiques

- True est une constante prédéfinie.
- False est l'autre constante prédéfinie.
- True et False sont les seules constantes booléennes.

2 Opérations logiques

a Égalité logique

- Le égal logique s’écrit == (ou ==). Sa syntaxe est *var1==var2*.
- Il ne faut pas le confondre avec le signe = d’affectation dans une variable.
- On peut comparer des nombres, des chaînes de caractères, des fonctions et des variables logiques.
- Le résultat de *var1==var2* est True quand les quantités sont égales, False sinon.
- Pour les nombres numériques, l’égalité est True *dès* que les 14 *premiers* chiffres *décimaux* sont égaux.
- Par exemple, 2.00000000000002==2 donne True
- Les cinq opérateurs <, >, <= (ou ≤), >= (ou ≥) et != (ou ≠) s'utilisent comme ==, avec chacun leur sens commun.

b Opérations algébriques

- || (ou ∨) est le ou logique. On écrit *bool1*||*bool2*
- L’utilisation de Alternatives donne des écritures équivalentes mais plus succintes. Par exemple (a==1||b==1) est équivalent à (a|b)==1
- && (ou ∧) est le et logique. On écrit *bool1*&&*bool2*
- Xor[*bool1*,*bool2*] est le ou exclusif.
- ! *bool* est la contraposée de *bool*. On peut également écrire Not[*bool*].
- On combine les objets logiques avec des *et logique* ou des *ou logique*, comme dans l'expression (a==1&&b==n)|||(a==n&&b==1)

3 Autres tests

a If

- La syntaxe de If est If[*condition*,*commande_si_True*,*commande_si_False*,*commande_si_Undetermined*] où *condition* est une variable booléenne ; si elle est vraie, l'exécution du If provoque celle de *commande_si_True* ; si elle est fausse, celle de *commande_si_False* ; si elle est indéterminée, celle de *commande_si_Undetermined*.
- Les commandes *commande_si_True*, *commande_si_False* ou *commande_si_Undetermined* peuvent être choisies de façon absolument quelconque et ne sont évaluées que lorsqu'elles sont exécutées.
- Les commandes *commande_si_False* et *commande_si_Undeterminate* peuvent être omises.
- La commande *commande_si_Undeterminate* peut être omise seule.

b Which

- La syntaxe de Which est Whichf[*cond1*,*commande1*, *cond2*, *commande2*,...] où *cond1*, *cond2* sont des variables booléennes ; si *cond1* est True, *commande1* est exécutée et Which est terminé ; sinon, le programme passe à *cond2*, dont la véracité conditionne pareillement l'exécution de *commande2* ; et ainsi de suite.
- Comme le couple *cond*, *commande* correspond à un *else* global, on choisit souvent True pour cette dernière *cond*.

H Boucles

1 Commandes successives

- Pour exécuter plusieurs commandes successives, on peut passer à la ligne dans une cellule.
- Dans ce cas, à chaque ligne est attribué un numéro *iii* successif d’Output.
- On peut aussi juxtaposer plusieurs ordres, dans une seule ligne, en les séparant par un **;** (point-virgule).

  Dans ce cas, seul au dernier ordre est attribué un numéro *iii* d’Output.

  De plus, seul le dernier ordre produit un affichage (sauf s’il est suivi d’un **;**, auquel cas rien n’est affiché).

2 Boucles simples

- La syntaxe d'une boucle simple est Do[*commande_i*,{i,*idebut*,*ifin*}]

  Toutes les variantes sur la boucle {i,...} décrites pour Table sont valides.

  Si *commande_i,* ne dépend pas de i, on peut écrire {n}, pour signifier que l'action est répétée n fois.
- *commande_i* est une série d'instructions quelconques, séparées par des **;** (*point-virgule*).
- Sum fait une sommation.

  Sa syntaxe est la même que celle de Do
- Product fait un produit.

  Sa syntaxe est la même que celle de Do
- Il est possible de faire une somme ou un produit à l'infini.

3 Boucles avec test

- While a pour syntaxe While[*test*,*corps*] où *test* est une variable booléenne.
- Tant que *test* vaut True la boucle continue. Quand *test* vaut False elle s'arrête.
- *corps* est une série d'instructions quelconques, séparées par des **;** (*point-virgule*).

  Généralement, une instruction d'incrémentation est incluse dans *corps* mais ce n'est pas obligatoire.
- For[*debut*,*test*,*increment*,*corps*] est une commande similaire mais plus rigide.
- Le couple Catch/Throw permet des sorties de boucles conditionnelles et est un outil très utile..

I Substitutions

1 Règles de substitution

- *Mathematica* exprime de très nombreux résultats sous forme de règles de substitution.
- Une règle s'écrit *symbol*$_1$ → *expression*

  Pour écrire la flèche, on tape – >
- On écrit **/.** placé après une expression pour signifier que l'on effectue une substitution selon la règle de substitution qui suit **/.**
- Cela revient à faire une affectation qui ne serait valable **que** *pendant ce calcul*.

  Toutefois, on constate que les substitutions agissent moins puissamment que les affectations ou les variables de fonction.
- On peut appliquer plusieurs règles simultanément (lues lit de gauche à droite) en utilisant une liste de règles {*symbol*$_1$ → *expression*$_1$, *symbol*$_2$ → *expression*$_2$, ...}.
- L'utilisation de Alternatives donne des écritures équivalentes mais plus succintes. Par exemple (a→1,b→1) est équivalent à (a|b→1)

2 Substitution différées

- On peut différer une règle, de même qu'on diffère une affectation.
- Dans ce cas la règle s'écrit *symbol*$_1$:→ expression

  Pour écrire la flèche différée, on tape : >

3 Substitutions de variables

- On peut définir une règle avec des variables *indéfinie*. Dans ce cas, on utilise un _ (souligné) dans l'expression à *gauche* de la flèche →.

  Par exemple, x^*n*_→x^(*n*-1) transforme toutes les puissances *n*≠1 de x en puissance *n*-1

  Par exemple, x_^*n*_→x^(*n*-1) transforme toutes les puissances *n*≠1 de *toute* variable en puissance *n*-1.

- Une règle peut être définie pour agir sur des objets *spécifiques*.

  Par exemple, Line[{x1_,y1_},{x2_,y_2}]→ n'agit *que* sur des *lignes* définies par *deux* points.

Substitutions répétées

- En écrivant //. à la place de /. on applique la règle de substitution un nombre infini de fois, l'exécution ne s'arrêtant que quand le résultat reste inchangé.

  Si l'action de la substitution n'atteint jamais un point fixe, il faut éteindre le noyau.

  Par exemple, il ne faut pas exécuter 1/x //. x^*n*_→x^(*n*-1)

J Equations

1 Résolution formelle d'équations ordinaires

a Syntaxe de Solve

- Pour résoudre formellement une équation *ordinaire*, on utilise Solve[*equation*,t] où *equation* s'écrit *fonction*[t]==0 et t est la variable de l'équation.

  t doit être non affecté au moment de l'exécution.

- Attention, les solutions sont des règles de substitution.

  Même quand il n'y a qu'*une* solution, elle est donnée sous forme d'une *liste* (de règles).

  La liste de solutions s'écrit avec *deux* niveaux d'*accolades* : {{x→*première_solution*},{x→*deuxième_solution*},...}

- Il est parfois utile de retirer un niveau d'accolades.

  Pour cela, on peut utiliser Solve[*equation*,t][[All,1]]

  Cette syntaxe préserve toutes les solutions.

  S'il n'y a qu'une solution, [[All,1]] est équivalent à [[1]] ou à //First

- La liste des solutions est ordonnée selon l'ordre OrderedQ (celui de Sort).

- Pour les équations à plusieurs variables, il faut écrire en dernier argument de Solve la liste des variables {x,y,z} et la syntaxe devient Solve[*equation*,{x,y,z}]

- S'il y a plusieurs équations, le premier argument de Solve devient une liste d'équations, ce qui s'écrit Solve[{*equation1*,*equation2*,*equation3*},{x,y,z}]

- NSolve[*equation*,t] est strictement équivalent à N[Solve[*equation*,t]]

b Comment mémoriser les solutions dans des variables

- Pour les *exprimer* comme des nombres ou des expressions formelles, il faut appliquer ces règles à la variable de l'équation.

  Par exemple, pour exhiber solutions=Solve[*equation*,x], on écrit regle1=solutions[[1]] puis x1=x/.regle1

  Ou directement x1=x/.solutions[[1]]

- Dans certains cas, les solutions sont exprimées à l'aide de la fonction interne Root

  On peut tenter alors de leur appliquer ToRadicals pour obtenir une expression plus standard, mais le résultat n'est pas garanti.

2 Résolution numérique d'équations ordinaires

- Pour les équations non solubles formellement, il existe *un seul* ordre de résolution numérique, FindRoot

- La syntaxe est FindRoot[*equation*,{x,*x0*}] où *x0* est une solution numérique de départ (plus précisément, un nombre réel).

- La liste des solutions s'écrit {{x→...},{x→...},...}

  Dans une très grande majorité de cas, il n'y a qu'*une* solution (la fonction implicitement définie par l'équation est *injective*).

- Attention, les solutions sont des règles de substitution., données sous forme d'une liste avec *deux* niveaux d'*accolades*.
- Il est parfois utile de retirer un niveau d'accolades. Par exemple, on peut utiliser FindRoot[*equation*,{t,*x0*}][[1]]
- Pour les équations à plusieurs variables, il faut érire en dernier argument de FindRoot une liste {{x,*x0*},{y,*y0*},{z,*z0*}} où *x0*, *y0*, *z0*, sont des nombres réels.

  La syntaxe est alors FindRoot[*equation*,{{x,*x0*},{y,*y0*},{z,*z0*}}]

3 Résolution formelle d'équations différentielles

a Syntaxe de DSolve

- Pour résoudre formellement une équation *différentielle*, on écrit DSolve[*equation*, *f*[*x*],*x*]

  Par exemple *equation* s'écrit *f* ''[*x*] + ... == 0
- Attention, les solutions sont des règles de substitution.

  Même quand il n'y a qu'*une* solution, elle est donnée sous forme d'une *liste* (de règles).

  La liste de solutions s'écrit avec *deux* niveaux d'*accolades* : {{f[x]→*première_solution*},{f[x]→*deuxième_solution*},...}
- Il est parfois utile de retirer un niveau d'accolades.

  Pour cela, on peut utiliser DSolve[*equation*,f[t],t][[All,1]]

  Cette syntaxe préserve toutes les solutions.

  S'il n'y a qu'une solution, [[All,1]] est équivalent à [[1]] ou à //First
- La liste des solutions est ordonnée selon l'ordre OrderedQ (celui de Sort).
- S'il y a *plusieurs* équations, il faut remplacer *equation* par une liste {*equation1*,*equation2*,...}
- S'il y a *plusieurs* fonctions, il faut remplacer *fonction* par une liste {*fonction1*,*fonction2*,...}

  S'il y a à la fois plusieurs équations et plusieurs fonctions, on retrouve un système d'équations couplées.
- S'il y a *plusieurs* variables, il faut remplacer *variable* par une liste **{***variable1*,*variable2*,...**}**

  Ce cas recouvre en particulier les équations différentielles partielles.
- En l'absence de conditions initiales, *Mathematica* pose des constantes arbitraires C[1], C[2], ...

  Il y a une constante par degré de liberté, donc autant que l'**ordre** de l'équation différentielle.

b Comment extraire une solution

- Pour *sélectionner* une solution donnée, parmi les solutions générales sans conditions initiales, il faut choisir des valeurs des constantes arbitraires.

  Si par exemple C[1] vaut 1 et C[2] vaut 0, il faut utiliser des règles de substitution {C[1]→1,C[2]→0} comme dans solutions/.{C[1]->1,C[2]->0}
- Pour *extraire* de la règle précédente une fonction, il faut l'appliquer à la variable fonctionnelle f[*x*] de l'équation.

  Par exemple regle1=solutions/.{C[1]->1,C[2]->0} puis f1[t_]=f[t]/.regle1

  Ou *directement* exécuter f1[t_]=f[t]/.(solutions/.{C[1]->1,C[2]->0}) (attention aux parenthèses, qui sont ici indispensables)

c Équations avec conditions initiales

- On peut préciser les conditions initiales par des équations *suppémentaires*.
- Dans DSolve, *equation* est *remplacé* par la liste {*equation*,*condition1*,*condition2*,...}
- Le **nombre** de conditions doit être inférieur ou égal à l'**ordre** de l'équation différentielle.

  Elles doivent **respecter** les règles mathématiques générales standard applicables aux équations différentielles.
- L'étape consistant à choisir C[1], etc, est supprimée dès qu'il y a un nombre de conditions initiales (exprimées sous forme d'équations) *égal* à l'ordre de l'équation.

  S'il n'y a qu'une seule solution, il suffit d'appliquer [[1]] ou //First pour retirer un niveau d'accolade.

4 Résolution numérique d'équations différentielles

a Syntaxe de NDSolve

- La syntaxe de NDSolve est NDSolve[{*equation*,*condition1*,*condition2*,...},*fonction*,{*variable*,*borne_inf*,*borne_sup*}] où on doit préciser les valeurs des bornes d'intégration *borne_inf* et *borne_sup* .

  De façon générale, NDSolve ne peut s'exécuter dès qu'il existe une variable indéterminée.
- Il s'agit obligatoirement d'une résolution avec conditions initiales.
- Le **nombre** de conditions doit être strictement égal à l'**ordre** de l'équation différentielle.
- Attention, NDSolve est un **faux-ami** car, à l'*inverse* de NSolve, il ne correspond pas à N[DSolve]

  C'est au contraire l'*analogue* de FindRoot.
- La liste des solutions s'écrit {{f[x]→InterpolatingFunction[...]},...}

  Dans une très grande majorité de cas, il n'y a qu'*une* solution.
- Attention, cette solution est une règle de substitution., donnée sous forme d'une liste (à un élément) avec *deux* niveaux d'*accolades*.
- Il est parfois utile de retirer un niveau d'accolades.

  Pour cela, on peut utiliser NDSolve[*equation*,t][[1]]

b Comment extraire une solution

- Les fonctions *première_solution* , *deuxième_solution* , ..., sont des InterpolatingFunction
- L'étape consistant à choisir C[1], ..., n'a plus lieu d'être.
- Pour extraire une *fonction*, il faut appliquer la règle f[x]→InterpolatingFunction[...] à la variable fonctionnelle f[*x*] de l'équation.

5 Les solutions des équations différentielles peuvent être des fonctions intrinsèques

a Cas pour lesquels les solutions sont nécessairement des fonctions ordinaires

- Si on écrit DSolve[*equation*, f[*x*],*x*], on obtient des listes de règles avec fonction ordinaire {{f[x]→...
- Dans ce cas, il faut *impérativement* écrire f[x]/. DSolve[*equation*, f[*x*] , *x*], pour mémoriser une solution, qui sera une fonction *ordinaire*.
- De même NDSolve[*equation*, f[*x*],*x*] produit nécessairement des fonctions ordinaires.

b Cas pour lesquels les solutions peuvent des fonctions intrinsèques

- Si on écrit DSolve[*equation*, f,*x*], on obtient des listes de règles avec fonction intrinsèque {{f→...
- Dans ce cas, si on écrit f /. DSolve[*equation*, f , *x*], pour mémoriser une solution, ce sera une fonction *intrinsèque*.
- L'utilisation d'une fonction intrinsèque nécessite ***obligatoirement*** de *supprimer* une niveau d'accolade en utilisant [[All,1]] ou d'autres moyens équivalents.

  Exemple d'erreur : on définit f1=f /. DSolve[*equation*, *f* , *x*] alors f1[x] n'existe pas.
  Exemple correct : on définit f1=f /. DSolve[*equation*, *f* , *x*][[1]] alors f1[x] fonctionne correctement.
- Par contre, si on écrit f [x]/. DSolve[*equation*, *f* , *x*], pour mémoriser une solution, ce sera une fonction *ordinaire*. et on se reportera à la section sur DSolve
- Ce qui précède s'applique également à NDSolve

K Chaînes de caractères

1 Création

- Une chaîne de caractère est écrite entre guillemets “ ”.

- Un caractère est une chaîne de caractère de longueur 1.
- Une chaîne nulle s'écrit ""
- Pour écrire, dans une chaîne de caractère, les deux caractères spéciaux " et \ il faut écrire \" et \\
- On peut également convertir toute expression *expr* en chaîne de caractère en exécutant ToString[*expr*]
- Attention, c'est l'expression évaluée qui est prise en compte.

  Par exemple, si a=2, l'exécution ToString[a] donne "2"
- Il peut s'avérer utile d'utiliser Evaluate dans ToString

2 Utilisation

- Les outils de manipulation des chaînes de caractères sont très nombreux et sont utilisés notamment en traitement du signal.
- Par ailleurs, une chaîne de caractère *string* peut être converti en une expression qu'on peut évaluer, en exécutant Evaluate[*string*]

3 Manipulation de chaînes

a Fonctions agissant sur des chaînes de caractères

- Rappel : on peut restreindre les arguments aux chaînes de caractères avec la restriction var_*String*

b Manipulations prédéfinies

- Characters[*string*] donne la liste des caractères d'une chaîne.
- StringLength[*string*] donne la longueur de la liste précédente, c'est à dire la longueur de la chaîne.
- StringJoin[*string1*,*string2*,...] concatène les chaînes dans l'ordre.

  On peut écrire de façon équivalente *string1*<>*string2*<>...
- StringInsert[*string1*,*string2*,n] insert la chaîne *string2* dans la chaîne *string1* **avant** le n-ème caractère.
- StringReplacePart[*string1*,*string2*,{n,m}] est similaire : il supprime du n-ème au m-ème caractère dans *string1* et y substitue *string2*.

  StringReplacePart ne fonctionne que si n≤m.

  StringInsert[*string1*,*string2*,n] est équivalent à StringReplacePart[*string1*,*string2*,{n,n}]
- StringExpression[*string1*,*string2*,...] est la suite de chaîne dans l'ordre.

  On peut écrire de façon équivalente *string1*~~*string2*~~...
- La différence entre ~~ et <> est subtile.

  Par exemple, *string*~~"" et *string*<>"" donnent identiquement *string*.

  Si une chaîne est lue en entrée, pour l'interpréter comme une juxtaposition de boûts de chaînes, on *doit* utiliser ~~

  Si une chaîne est créée en sortie, parfois le résultat varie selon qu'on utilise ~~ ou <>
- La syntaxe de StringReplace est très différente et s'écrit StringReplace[*string*, *règle*] où *règle* s'écrit *c*→*cc*, avec *c* et *cc* des chaînes de caractères.

  On choisit souvent comme *c* des caractères simples. C'est également le cas, moins souvent, pour *cc*.

  On peut choisir comme *c* l'expression "a"|"b" qui signifie "a" ou "b".

  On peut choisir comme *c* l'expression "ab"~~_ qui signifie les caractères "ab" suivi d'*un* caractère quelconque.

  On peut choisir comme *c* l'expression "ab"~~__ (*deux* soulignés) qui signifie les caractères "ab" suivi d'une chaîne quelconque non nulle.

  On peut choisir comme *c* l'expression "ab"~~___ (*trois* soulignés) qui signifie les caractères "ab" suivi d'une chaîne quelconque, y compris *nulle*.

  On peut choisir comme *c* l'expression "ab".. (*deux* points) qui signifie les caractères "ab" répétés un nombre quelconque mais non nul de fois, comme "abab",etc.

  On peut choisir comme *c* l'expression "ab"... (*trois* points) qui signifie les caractères "ab" répétés un nombre quelconque voire nul de fois.

- D'autres objets peuvent être utilisé dans ces syntaxes.

  Par exemple, Whitespace signifie un ou plusieurs espace blancs.

  WhitespaceCharacter signifie un espace blanc.

  WordBoundary signifie n'importe quel caractère spécial (y compris un espace blanc) qui touche un mot (composé de lettres ou de chiffres *uniquement*).

  StartOfString signifie le début d'une chaîne de caractères.

  EndOfString signifie la fin d'une chaîne de caractères.

  DigitCharacter signifie un chiffre.

  LetterCharacter signifie une lettre.

  WordCharacter signifie un chiffre ou une lettre.

- Pour préciser le genre d'une variable parmi les précédents, on ne peut utiliser _ car ce ne sont pas des types.

  À la place, on écrit par exemple *x*: WordCharacter où les : (*deux points*) généralise _ (*souligné*).

L Syntaxe avancée

1 Comment générer des nombres aléatoires

- RandomReal est la commande qui génère un nombre réel aléatoire.

  Pour générer un nombre réel aléatoire ∈[0,1], on peut utiliser RandomReal[]

  Pour générer un nombre réel aléatoire ∈[0 , x_2], on peut utiliser RandomReal[x_2]

  Pour générer un nombre réel aléatoire ∈[x_1 , x_2], on peut utiliser RandomReal[{x_1 , x_2 }]

  Pour générer *nbre* nombres réels aléatoires ∈[0 , x_2], on peut utiliser RandomReal[x_2, *nbre*]

  Pour générer *nbre* nombres réels aléatoires∈[x_1 , x_2], on peut utiliser RandomReal[{x_1 , x_2 }, *nbre*]

  Pour générer une matrice aléatoire $n_1 \times n_2$ de nombres réels ∈*range*, on peut utiliser RandomReal[*range* , {n_1, n_2 }]

- On définit de façon identique RandomInteger pour les nombres entiers.

  La seule différence est que les nombres x_1 et x_2 sont remplacés par des entiers i_1 et i_2.

- On définit également RandomPrime qui ne génère que des nombres premiers.

- On définit également RandomChoice[*liste*, *nbre*] qui génèrent *nbre* éléments de la liste *liste* de façon aléatoire.

  RandomChoice[*liste*] génèrent 1 élément de la liste *liste* de façon aléatoire.

2 En-tête d'une expression

- L'en-tête Head d'une expression est la première commande écrite quand on exécute FullForm

- On peut la modifier en appliquant une règle *head1*→*head2*

- Il est souvent préférable d'utiliser Replace plutôt que /. qui est ReplaceAll et risque de faire la substitution sur tous les Head identiques à *head1*.

  Par exemple, Sum[x^i,{i,4}]/. Plus→List donne {x, x^2,x^3,x^4} tandis que Sum[i,{i,4}]/. Plus→List donne 10, car la règle agit sur l'expression évaluée, et dans ce dernier cas, il n'y a plus de Head

- L'en-tête Sequence est une liste sans les accolades { }.

  Quand elle intervient en premier, elle est écrite explicitement.

  Mais, à l'intérieur d'une commande, elle disparaît dès que c'est possible.

  Par exemple, au lieu d'écrire Print[1," ",2," ",3," ",4," "] on peut exécuter
  Print[Table[{i," "},{i,4}]/.List→Sequence]

  List→Sequence élimite les { } sans perdre les virgules.

- Apply permet une généralisation de la procédure avec Replace mais pour n'importe quelle fonction et pas seulement une Head

  Sa syntaxe est Apply[*f*, *liste*] et donne l'équivalent de *f*[*liste*/.List→Sequence]

  Par exemple, Apply[Plus,{1,2,3,4}] est équivalent à Plus[1,2,3,4] et donne 10.

- Apply peut agir sur des expressions plus générales que les listes.

  Par exemple Apply peut agir sur une somme : par exemple Apply[*f*, a+b+c+d] donne *f*[a,b,c,d]

  La syntaxe deivent dans ce cas\.08 l'équivalent de *f*[*liste*/.Plus→Sequence]

3 Algèbre avancée

- La commande Outer permet de calculer le produit tensoriel u⊗v.

  On écrit, par exemple, Outer[Times,{x,y,z},{a,b,c}]
- Distribute agit sur les objets formels.

  Par exemple, Distribute[f[a+b,c+d]] donne f[a,c]+f[b,c]+f[a,d]+f[b,d]
- Map[*f*, *liste*] donne la liste {*f*[l_1], *f*[l_2],..., *f*[l_n]} où n est la longueur de *liste* et l_i les éléments qui la composent.
- On peut préciser, par un troisième argument, à quel niveau d'accolade Map agit, cf. Flatten

  La syntaxe pour ce troisième argument est identique à celle de Flatten mais son utilisation plus difficile.
- Map peut agir sur des expressions plus générales que les listes.

  Par exemple Map peut agir sur une somme : Map[*f*, a+b+c+d] donne {*f*[a], *f*[b], *f*[c], *f*[d]}
- Pour comprendre la syntaxe, on peut dire que Map fait la substitution de Head selon la règle Plus→List
- Cases peut être utilisé comme un Map sophistiqué. Au lieu d'écrire un type de variable, on peut écrire une règle qui transforme ces variables.

  Par exemple, si on prend le type {_,_}, on peut écrire Cases[liste,{a_,b_}→a+b]
- Cases peut agir sur des expressions plus générales que les listes, comme Apply ou Map

  Attention que dans ce cas, il agit sur l'expression évaluée après la substitution →List

4 Manipulation de listes avancée

- La commande Position donne le rang dans une liste d'un élément donné.

  Sa syntaxe est identique à celle de Cases

  On peut obtenir ainsi la position d'un élément formel, exprimé, ou d'un type d'élément.

  On peut exécuter l'ordre à un niveau d'accolade donné, si on le spécifie.
- Position peut agir sur des expressions plus générales que les listes, comme Cases

  Attention que dans ce cas, il agit sur l'expression évaluée après la substitution →List

  Toutefois, il n'est pas possible d'inclure des règles de substitution comme pour Cases
- La commande Count donne le nombre d'occurences dans une liste d'un élément donné.

  Sa syntaxe est identique à celle de Position

  Toutefois, Count ne peut pas agir sur des expressions plus générales que les listes.
- Il est possible d'inclure des règles de substitution selon la syntaxe avancée de Cases expliquée ci-dessus.
- La commande Insert peut agir sur des expressions plus générales que les listes.

  Il faut prendre garde cependant que, avec les en-tête Plus ou Times l'expression est automatiquent réordonnée.
- La commande Part peut agir sur des expressions plus générales que les listes.

  Opérant avec All cela permet des résultats très puissant.

  Par exemple, pour extraire les solutions d'un Solve on peut écrire Solve[[All,1,2]]

  Par exemple, pour extraire les solutions d'un DSolve on peut écrire DSolve[[All,1,2]]

5 Transformation d'une variable implicite

a Exemple basique

- Soit une fonction x1[u] dépendant de la variable implicite *λ*. On peut définir une fonction de traçage :

  plot1[lambda_] := Plot[x1[u]/.*λ*->lambda,{u,-*π*,*π*}]

  On a fait un transfert de la variable *λ* vers lambda, qui est une variable explicite de plot1.

b Différence entre l'effet des affectations immédiates et différées

- Quand on fait des affectations *immédiates*, cette transformation est inutile.
- Par exemple : polynome = x^2-3x^4 suivi de f[x_]=polynome est valide et f[t] donne $t^2 - 3\,t^4$
- Quand on fait des affectations *différées*, cette transformation est indispensable.

  Par exemple : polynome = x^2-3x^4 suivi de f[x_] := polynome ne fonctionne pas et f[t] donne $x^2 - 3\,x^4$

  La **bonne** syntaxe utilise une règle de substitution : f[u_] = polynome/.x→u
- Pour une série entière dont l'ordre est arbitraire, une affectation immédiate est impossible.

  La **bonne** syntaxe utilise une règle de substitution et s'écrit : fdev[n_,u_] := Normal[Series[f[x],{x,0,n}]]/.x->u
- Pour une équation différentielle résolue formellement, l'affectation immédiate est généralement possible.
- Pour une équation différentielle résolue numériquement, l'affectation immédiate n'est jamais possible.

  La **bonne** syntaxe utilise une règle de substitution et s'écrit :
  f0[n_,u_]:=f[x]/.(NDSolve[eq[n,f[x]],f[x],x][[1]]/.{C[1]->1,...})/.x->u où il est indispensable d'utiliser des parenthèses.

6 Function

- Il n'y a pas d'algèbre élémentaire avec les fonctions implicites.
- On **doit** donc utiliser des fonctions ordinaires et expliciter la variable x pour faire des combinaisons de fonctions.

  Autrement dit Sin/2[Pi] n'existe pas.
- On peut alternativement utiliser Function, où x reste une variable interne et le résultat une fonction intrinsèque.

  Par exemple, Function[x,Sin[x]/2] est intrinsèque et permet le calcul de Function[x,Sin[x]/2][Pi]

7 Répétition

a NestList

- NestList donne la liste des composées d'une fonction f, depuis l'identité jusqu'à $f^{\circ n}$, où toutes ces fonctions sont appliquées à un point $x_0$de départ. C'est la liste $\{x_0,\ f(x_0), f(\ f(x_0)),\ \ldots, f^{\circ n}(x_0)\}$

  Sa syntaxe est la même que celle de Nest étudiée à la section sur les fonctions.

b Points fixes

- NestWhile est une variante de Nest où on remplace le nombre d'itération par un test d'arrêt.

  Dans sa version simple (avec trois arguments), il faut s'assurer que NestWhile va s'arrêter.

  Dans sa version simple, le test est une fonction booléenne à *un* argument (comme EvenQ ou #>0&).
- On peut préciser par un quatrième argument m combien d'éléments sont testés dans la liste $\{x_0,\ f(x_0),\ \ldots, f^{\circ n}(x_0)\}$ déjà formée, en comptant à partir du dernier.

  Par défaut, m vaut 1. Quand m > 1, cela prolonge nécessairement la liste.

  Le test peut être une fonction booléenne à un, deux, jusqu'à m arguments.

  On peut indiquer, au lieu de m, { m_0 ,m}, où m_0 est la première itération à partir de laquelle le test est fait.

  m_0 doit être ≥ au nombre d'arguments du test. Par défaut m_0 vaut m

  Si m est indiqué (on peut le cas échéant écrire 1), un cinquième argument peut être indiqué : le nombre maximale d'itération, ce qui sécurise l'exécution.
- NestWhileList est l'équivalent deNestList pour NestWhile
- FixedPoint[f,m_0,m] est équivalent à NestWhile[f,m_0,UnsameQ,2,m]

  FixedPoint[f,m_0] est équivalent à NestWhile[f,m_0,UnsameQ,2]
- De la même façon FixedPointList est équivalent à NestWhileList
- De façon logique, le test par défaut dans FixedPoint ou FixedPointList est SameQ

  On peut ajouter dans FixedPoint ou FixedPointList une option SameTest→ qui permet de changer le test par défaut

  Ceci rend les équivalences précédentes parfaites.

M Modules

1 Modules

- Il existe trois ensembles de variables : les variables *System*, les variables *Global* et les variables locales.
- Ces dernières ont un statut *Global*, mais elles s'en distinguent parce qu'elles ont l'attribut Temporary qui les distinguent et empèche toute confusion avec des variables homonymes.
- Module[*variables*,*corps*] exécute une commande ou série de commande avec des variables locales.

  Ainsi, cela évite toute confusion sur les noms de variables.

  Cela n'évite pas la mémorisation des données.
- *variables* est obligatoirement une liste de symbole.
- On peut donner une valeur initiale à une variable locale (mais elle ne peut dépendre des autres variables locales).

  Dans ce cas, *variables* ressemble à {v1=*valeur1*,v2,...} où la liste mélange des symboles et des affectations.

  Ces affectations doivent être *immédiate*.
- *variables* peut être vide {}
- *corps* est une commande ou une série de commandes séparées par des **;** (point-virgule).
- With[*affectations*,*corps*] ressemble à Module et introduit des variables locales dans la liste *affectations*.

  La seule différence apparente est que toutes les variables locales doivent avoir une affectation initiale.

  Il se trouve que l'exécution de With est encore plus rapide que celle de Module
- La commande Block[*variables*,*corps*] ressemble à Module mais les variables sont *globales*.

  Ce sont leur *affectation* pendant l'exécution de Block qui sont locales.

  Toute modification d'une variable pendant l'exécution de Block est oubliée ensuite.

2 Manipulate

- La commande Manipulate est un module par défaut.
- La syntaxe est différente, on écrit Manipulate[*terme*_ij,{i,*idebut*,*ifin*},{j,*jdebut*,*jfin*}]
- L'exécution crée une boîte graphique, qui contrôle l'exécution de *terme*_ij avec une manette par liste {i,*idebut*,*ifin*}

  Les manettes sont gérées par Slider
- On a les mêmes variantes pour les listes qu'avec Table

  La variante {i, *ifin*} est cependant interdite.
- Il est possible de choisir une variable bidimensionnelle.

  Les manettes des variables bidimensionnelles sont gérées par Slider2D qui est un bel outil

  On peut remplacer les manettes en utilisant Locator
- Pour donner une valeur initiale, on remplace, dans {i, *idebut*, *ifin*}, i par {i, i*ini*}

  Autrement dit, on écrit {{i, i*ini*}, *idebut*, *ifin*}
- Pour utiliser Manipulate avec des variables globales, il faut écrire, à la fin, l'option LocalizeVariables→False
- Il existe de très nombreuses options et variantes, qui concernent la présentation et le fonctionnement de Manipulate

  Par exemple SaveDefinitions→True mémorise la définition dans la boîte créée par Manipulate

  Par défaut, SaveDefinitions→False

3 Dynamisme

- Les modules sont souvent utilisé avec la commande Dynamic[*symbole*]
- La valeur de Dynamic[*symbole*] est réactualisé dès que l'affectation de *symbole* est modifiée.
- La difficulté consiste à placer Dynamic aux bons endroits.

  En particulier, on ne doit pas l'utiliser de façon redondante.

- Lorsque l'on utilise avec Dynamic il est souvent nécessaire de choisir DynamicModule à la place de Module
  Les variables locales dans DynamicModule sont automatiquement dynamiques.
  La syntaxe est identique à celle de Module

4 Boutons

- On crée un bouton avec la commande Button
  La syntaxe est Button[*label*,*corps*]
  label est souvent un texte, mais un graphisme peut également être utilisé.
  corps est une commande ou une série de commandes séparées par des **;** (point-virgule).
- Parmi les options, citons Background→ qui est suivi d'un style graphique.
- Il y a de nombreuses variantes, DefaultButton CancelButton PasteButton ButtonBar
- Il y a des boutons courts sans label, RadioButton RadioButtonBar Checkbox CheckboxBar TogglerBar Toggler Setter SetterBar
- Il existe des menus déroulants ActionMenu et PopupMenu
  La syntaxe est ActionMenu[*label_par défaut*,*label* :→ *corps*] où intervient une substitution différée.

5 Fenêtres

a Création

- On crée une fenêtre avec la commande CreateDialog
  La syntaxe est CreateDialog[*texte*,*corps*]
  texte est souvent un texte, mais un graphisme peut également être utilisé.
  corps est une commande ou une série de commandes séparées par des **;** (point-virgule).
  corps peut être omis (dans ce cas, on édite juste une fenêtre)
- Il y a de nombreuses variantes, MessageDialog ChoiceDialog DialogInput CreateDialog
- Il y a des fenêtres plus spécifiques : CreateWindow ou CreateDocument pour créer un carnet et CreatePalette pour créer une palette.
- La taille de la fenêtre est gérée par l'option ImageSize

b Outils

- InputField crée des champ interactif qu'il faut remplir dans la fenêtre.
  On l'utilise souvent avec Dynamic
  On utilise tous les boutons décrits au dessus.
- Il y a des boutons courts sans label, RadioButton RadioButtonBar Checkbox CheckboxBar TogglerBar Toggler Setter SetterBar
- NotebookWrite[] écrit dans le carnet indiqué en option.
- InputNotebook[] est un outil qui permet de connaître le carnet courant où a été exécuté l'input.

c Interaction

- Input est placé à la place d'un symbole dans une expression.
- Il ouvre une fenêtre interactive et renvoie le résultat écrit dans l'expression où est placé Input

N Entrées/Sorties

- Pour ouvrir un fichier extérieur, on exécute OpenRead["nom_de_fichier"]
  "nom_de_fichier" est une chaîne de caractères, qui doit respecter la syntaxe (windows, linux, macOs) pour l'emplacement du fichier.

- Pour écrire sur le fichier, on exécute OpenWrite[“nom_de_fichier”]

  Cet ordre écrase l'ancien fichier au fur et à mesure qu'on écrit dessus.
- Pour ne pas écrase le fichier, il faut utiliser plutôt OpenAppend[“nom_de_fichier”]
- Pour fermer un fichier, on exécute Close[“nom_de_fichier”]
- Tous les ordres précédents sont généralement inclus dans les commandes qui suivent.

  Il n'est généralement pas utile de les utiliser, sauf exceptions.

1 Enregistrement sur un fichier

a Enregistrement d’une fonction

- Pour enregistrer la définition d’une fonction fonc dans un fichier def, il suffit d’exécuter Save[“def”,fonc]

  La syntaxe est *simple* mais *rigide*. Il **ne** faut **jamais** indiquer les arguments. Il est **impossible** de rebaptiser les fonctions
- À chaque exécution de Save, les définitions sont ajoutées.

b Écriture sur un fichier

- Les commandes standard pour l’écriture dans un fichier sont Put (>>) et PutAppend (>>>).

  Put écrase le fichier **s'il existe**, tandis que PutAppend ajoute en fin de fichier.
- Il n'est pas nécessaire de créer préalablement de fichier.
- La syntaxe est à l'opposée de Save. Il faut préciser le format et écrire les objets élément **par** élément.

  On écrit Put[*expr1*, *expr2*, ..., "nom_de_fichier"]

  “nom_de_fichier” est une chaîne de caractères donnant l’emplacement du fichier.

  Par exemple, on peut indiquer, en place de *expr1*, OutputForm[*expr1*]
- La syntaxe de PutAppend est identique.

c Enregistrement d’une image

- Pour exporter une figure, on exécute Export[“nom_de_fichier”,*expr*,”FORMAT”]
- “nom_de_fichier” est une chaîne de caractères donnant l’emplacement du fichier.
- *extpr* est l’objet (graphique, sonnore, texte, ..) qu’on veut enregistrer.
- ”FORMAT” est le format du fichier enregistré.

  Il peut être "EPS" "PDF" "SVG" "PICT" "WMF" "TIFF" "GIF" "JPEG" "PNG" "BMP" "PCX" "XBM" "PBM" "PPM" "PGM" "PNM" "DICOM" "AVI"

  La liste de tous les formats *accessibles* est donnée par $ExportFormats.

  Le format est détecté automatiquement et cet argument peut être généralement omis.

2 Lecture

- L’affichage du contenu d’un fichier “fich” se fait en exécutant FilePrint[“fich”]

  Les définitions sont écrites au format Input (*agréable* à lire),
- Pour lire un fichier et éventuellement mettre son contenu en mémoire, on peut utiliser Get (<<)

  La syntaxe est <<“fich”

  Elle n’admet pas d’option de format.
- Les commandes suivantes utilisent implicitement un Get :

a Lecture sous forme de liste

- Pour lire un fichier et mettre son contenu sous forme d’une liste, on peut utiliser ReadList

  La syntaxe est ReadList[“nom_de_fichier”,*type*]
- “nom_de_fichier” est une chaîne de caractères donnant l’emplacement du fichier.

- *type* est le format de lecture. Sa syntaxe est identique à celle de Cases

  Parmi les formats les plus utiles, on peut citer Number et Word

  Par exemple, s'il y a trois nombres par lignes, le format est {Number,Number,Number}

b Importation d'une image

- Pour importer une figure, on exécute Import["nom_de_fichier",*expr*,"FORMAT"]

  La syntaxe est exactement identique à celle de Export

c Exécution unix

- Pour exécuter une commande unix, on écrit <<" **!** *nom_de_commande*"

  La commande est écrite comme une chaîne de caractère.

- On doit ajouter impérativement un ! (point d'interrogation) avant la commande.

  Par exemple, pour supprimer un fichier "fich" on écrit <<"! rm def" où rm (pour remove) est la commande unix.

- L'utilisation d'une chaîne de caractères rend cette syntaxe malaisée, car certains ordres unix nécessitent aussi des guillements.

  Il faut utiliser \" mais ce n'est pas toujours suffisant.

  Surtout, la bonne syntaxe dépend du système d'exploitation.

O Performance du compilateur

1 Timing

- Timing[*commande*] donne le temps d'exécution sous forme d'une liste.

  Le premier argument est le temps et le second le résultat de l'exécution.

- *commande* est une commande ou une série de commandes quelconques.

2 Compile

- Compile est un outil qui permet de définir des fonctions purement numériques.

  La syntaxe est très différente des syntaxes *Mathematica* et ne sera pas étudiée.