



Observatoire aquitain

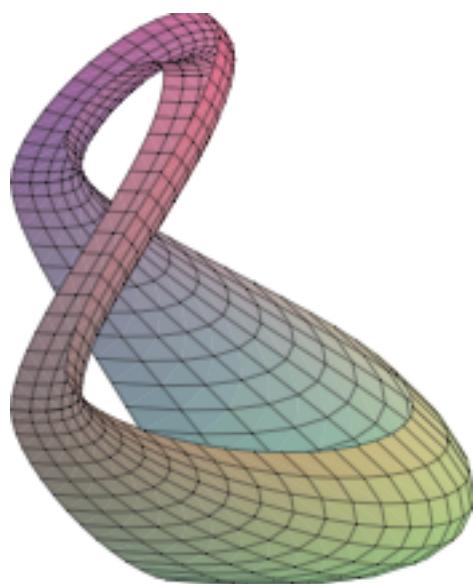
des sciences de l'univers

Manuel
MAPLE 9.5

Jean Brillet

Observatoire de Bordeaux

www.obs.u-bordeaux1.fr



Avant-propos

MAPLE est un logiciel développé par l'Université de Waterloo au Canada (www.maplesoft.com). Il permet de manipuler des expressions mathématiques de façon symbolique ou numérique. Établir un dialogue avec une machine pour parler de mathématique suppose un langage de communication particulier. En plus de la notion plus ou moins élémentaire de nombre, celui-ci doit être capable de "comprendre" des concepts plus abstraits comme ceux d'infini, de fonctions, de dérivée et d'intégrale, d'intervalles, de limites, de continuité, etc. MAPLE n'est qu'un programme informatique et doit de ce fait, pour pouvoir résoudre les problèmes posés, manipuler ces "objets" à travers des algorithmes. Ceci a deux conséquences fondamentales :

1) Outre les problèmes posés par l'entrée des commandes avec un clavier et qui nécessite l'abandon des notations traditionnelles, il est nécessaire de bien comprendre comment MAPLE appréhende et manipule ces concepts.

2) MAPLE n'est pas un mathématicien : c'est un manipulateur puissant et rapide d'expressions mathématiques. Aussi ne résoudra-t-il pas un problème si l'utilisateur n'a pas les idées claires sur la façon de le formuler et de conduire les calculs (cela dit, c'est aussi un outil pédagogique puissant pour se familiariser avec de nouveaux concepts).

L'objet de ce cours est d'introduire le point 1) mais en aucun cas le point 2). Les possibilités de MAPLE sont si vastes qu'il est impossible d'en donner un aperçu complet à travers un manuel comme celui-ci. Il est donc plus que recommandé de compléter cette information en explorant les aides disponibles, soit en ligne sur l'ordinateur soit dans des livres.

On conçoit qu'un logiciel aussi sophistiqué puisse comporter ici ou là quelques (rares) anomalies de fonctionnement et la prudence est toujours de mise. Ce manuel a été développé pour MAPLE 9.5 (version pour MacOS X). MAPLE fonctionne sur de nombreuses plates-formes et il n'est pas exclu que d'une machine à l'autre on puisse observer quelques différences mineures de comportement.

ATTENTION : Même si elles sont peu nombreuses, certaines remarques rencontrées dans ce manuel ne correspondent qu'à la version 9.5 et ne sont pas valables pour les autres versions. MAPLE est en constante évolution et les nouvelles versions, non seulement ajoutent des fonctionnalités ou corrigent les erreurs rencontrées, mais modifient aussi le comportement de certains algorithmes.

Table des matières

Figure de la page de garde : coupe d'une bouteille de Klein (voir la fin du chapitre 17)

Divers *Cette première rubrique regroupe des sujets divers qui ne sont pas nécessairement faciles à localiser avec les titres de chapitres.*

• Aide MAPLE	3
• Commentaires sur le fonctionnement de MAPLE	15
• Entrée des commandes	1, 15
• Syntaxe des noms, alphabet grec, noms indicés	15
• Noms indicés	15, 38, 117, 351
• « Fausses » erreurs de syntaxes	15
• État de MAPLE, noyau	16
• Ordre restart	17
• Initialisation de MAPLE	17
• Gestion des noyaux	18
• Accès aux bibliothèques, fonction with	18, 94, 371
• Ouvertures et sauvegardes des feuilles de calculs	18
• Organisation des feuilles de calcul	19
• Impression des feuilles de calculs	20
• Affichage des résultats, étiquetage	20
• Dito (%)	1, 17, 403
• Symboles et chaînes de caractères, manipulations	37, 350
• Fonction map	68, 156
• Fonction zip	70, 157
• Fonction de tri : sort	66, 128
• Distribution de nombres pseudo-aléatoires	147
• Interpolation et lissage (CurveFitting)	159, 410
• Fonction alias	134, 136, 142
• Opérateurs logiques (>, <, =, etc.) or , and	346
• Expressions définies par morceaux, fonction piecewise	158
• Transformations en Z	231
• Transformations de Fourier discrètes (FFT)	209
• Mot clé NULL	56
• Point décimal et opérateur « . »	7, 91
• Génération de code C, Fortran, ...	165
• Unités physiques, Constantes physiques	255, 257
• Fonctions spéciales	252
• Distributions	212

Chapitre 1 Les nombres dans Z, Q, R et C	1
• Nombres entiers	1
• Élision	2
• Nombres rationnels	4
• Nombres irrationnels	4
• Nombres remarquables	5
• Nombres décimaux	6
• Évaluations décimales implicites	7
• Évaluations décimales forcées	9
• Fonction evalf , variable Digits	9
• Fonction evalhf	10
• Nombres complexes	11
• Identification des nombres décimaux	13
• Divers	14
• Commentaires sur l'utilisation de MAPLE	15
Chapitre 2 Assignations, Evaluations et Substitutions, Types, Eléments sur la structure des expressions.	21
• Assignations et évaluations	21
• Assignations multiples	21
• Désassignation	22
• Évaluation, assignation et déassignation	22
• Évaluations explicites, fonction eval	23
• Fonction inerte Eval et fonction d'activation value	23
• Évaluations à valeurs multiples	24
• Modification d'une expression par eval	24, 31
• Non évaluation d'un nom ou d'une expression	26
• Test d'assignation, fonctions assigned , anames	27
• Protection d'un nom : fonctions protect et unprotect	27
• Éléments sur la structure des expressions : opérandes	28
• Soustractions et divisions	30
• Indéterminées d'une expression : fonction indets	31
• Retour sur les évaluations et substitutions	31
• Fonction subs	33
• Fonctions subsop et algsubs	34
• Type : fonctions whattype et type , profils de types	35
• Fonction whattype	35

• Fonction type	37
• Profils de types	41
• Noms des types et profils de type créés	42
• Fonction subtype	44
• Bibliothèque TypeTools	44
• Types attribués, fonctions setattribute et attributes	46
Chapitre 3 Intervalles, Suites, Listes, Ensembles	49
• Intervalles	49
• Suites	50
• Comptage, changement des éléments	52
• Génération automatique, fonction seq (voir aussi 62) et \$	52
• Ensembles	53
• Évaluations, substitutions	55
• Changements et suppressions d'un élément	55
• Égalité entre ensembles, appartenance d'un élément	56
• Opérations globales, union , minus , select , remove	59
• Listes	61
• Deuxième forme de la fonction seq	62
• Transformation en suites et réciproque	62, 63
• Suppression des éléments d'une liste	65
• Modification, ajout et insertion	65
• Bibliothèque ListTools	66
• Tri d'une liste	66
• Transformation en ensembles	66
• Addition et soustraction de listes	67
• Addition ou multiplication des éléments	67
• Fonction map	68
• Fonction zip	70
Chapitre 4 Vecteurs, Matrices et Algèbre Linéaire	71
• Construction des vecteurs et des matrices	71
• Constructions élémentaires	71
• Extraction, assignation, changement des éléments	74
• Affichage des grands vecteurs et des grandes matrices	76
• Génération automatique des composantes	77
• Vecteurs de type ligne	78
• Types et opérandes des vecteurs et des matrices	78

• Constructions avec options	80
• Option shape	80
• Option scan	83
• Option fill	83
• Combinaisons des options shape , scan et fill	84
• Option datatype	85
• Option storage	87
• Option readonly	87
• Assignations et fonction copy	88
• Algèbre linéaire élémentaire	89
• Opérateur produit matriciel	91
• La bibliothèque LinearAlgebra	92
• Option inplace	93
• Accès aux fonctions de la bibliothèque LinearAlgebra	94
• Quelques exemples de fonctions de LinearAlgebra	95
• Résolution d'un système d'équations linéaires	96
• Tests d'égalité, fonction Equal , verify	97
• Valeurs propres d'une matrice, fonction Eigenvalues	98
• Vecteurs propres d'une matrice, fonction Eigenvectors	101
• Application d'un opérateur sur les éléments d'une matrice	102
• Fonctions Map et Zip	103
• Calculs numériques	104
• Arithmétique câblée	104
• Erreurs numériques	107
• Forme du datatype d'un résultat numérique	108
• Arithmétique émulée	108
Chapitre 5 Tables et Tableaux	111
• Les Tables	111
• Contenus des tables	113
• Les options d'indexation	113
• Indexation <i>symmetric</i>	113
• Indexation <i>antisymmetric</i>	114
• Indexation <i>diagonal</i>	115
• Indexation <i>identity</i>	115
• Indexation <i>sparse</i>	116
• Création de fonctions d'indexation	116
• Fonction copy	116

• Noms indicés	117
• Opérandes et types des objets <i>table</i>	118
• Les Tableaux (Array)	120
• Construction et utilisation des tableaux	120
• Types des tableaux	122
• Fonction d'indexation	123
• Fonctions d'interrogation des tableaux	123
• Tableaux indépendants et duplication, fonction copy	124
• Bibliothèque ArrayTools	125
• Tableaux hfarray	125
Chapitre 6 Polynômes et Fractions rationnelles	127
• Polynômes	127
• Fonctions sort , coeff , degree , expand , collect	127
• Familles de polynômes orthogonaux	129
• Bibliothèque orthopoly	131
• Factorisation et recherche des racines d'un polynôme	131
• Factorisation exacte, fonction factor	131
• Fonction irreduc	131
• Extensions algébriques	132
• Alias RootOf , fonction allvalues	133
• Bibliothèque PolynomialTools	134
• Fonction alias	134
• Fonction roots	134
• Factorisation approchée	134
• Étude d'un exemple	135
• Variable globale _EnvExplicit	137
• Approximations décimales des racines	138
• Fractions rationnelles	139
• Type	139
• Fonctions numer et denom	139
• Division euclidienne, fonction quo (rem)	140
• Réduction au même dénominateur, fonction normal	140
• Décomposition de seconde espèce	141
• Décomposition de première espèce	141
• Radicaux au dénominateur, fonction rationalize	142
• Décomposition en fractions continues	143

Chapitre 7 Fonctions	145
• Définition et création, opérateur -> (arrow)	145
• Contrôle du type des arguments	150
• Règles générales d'évaluation des arguments et des fonctions	151
• Fonctions de plusieurs variables	152
• Opérateur unapply	153
• Manipulation des fonctions par leurs noms	154
• Composition des fonctions (opérateurs @ et @@)	155
• Fonction map	156
• Fonction zip	157
• Fonctions définies par intervalles, fonction piecewise	158
• Fonctions d'interpolation et de lissage. Approximation	159
• Interpolation polynomiale	159
• Fonctions splines	160
• Lissage au sens des moindres carrés	161
• Approximation d'une fonction, génération de code C/Fortran	162
• Fonctions et procédures	165
• Opérandes d'une fonction ; tables Cache et remember	166
• Fonctions prédéfinies de MAPLE	166
• Fonctions définies par l'utilisateur	170
• Fonction forget	172
• Types associés à une fonction	173
Chapitre 8 Dérivation	175
• Fonctions diff , Diff et value	175
• Dérivation d'ordre 0	176
• Opérateur D	178
• Dérivation des fonctions définies implicitement	180
Chapitre 9 Limites, Continuité	185
• Fonctions limit , Limit et value	185
• Options left et right	186
• Continuité	188
• Fonction iscont	188
• Fonction discont	190
• Détermination numérique des discontinuités, fonction fdiscont	191

Chapitre 10	Intégration	195
• Fonctions int , Int et value		195
• Calcul numérique d'une intégrale		197
• Divers autres aspects de l'intégration		198
• Manipulations		198
• Fonctions spéciales		199
• Intégrales de fonctions définies par morceaux		199
• Intégrales de fonctions définies par quelques points		200
• Intégration et contraintes		200
• Traitement des singularités		201
• Valeur principale au sens de Cauchy		201
• Intégrales en un point, opérateurs intat , Intat et value		202
• Intégration dans le plan complexe		203
• Illustration du théorème des résidus, fonction residue		204
• Changements de variables, bibliothèque Student		207
• Transformations intégrales et transformées de Fourier rapides		207
• Bibliothèque inttrans		207
• Transformées de Fourier rapides, FFT		209
• Distributions, Dirac		212
• Fonction map et opérateurs d'intégration		214
Chapitre 11	Développements en séries	215
• Fonction series , variable Order		215
• Transformation en expression ou fonction		217
• Fonctions taylor et mtaylor		219
• Bibliothèque powseries		221
• Opérandes et types des développements en séries		222
• Développements asymptotiques		223
• Développements en fractions continues		224
Chapitre 12	Sommes et Produits	225
• Sommes, fonctions sum , Sum et value		225
• Fonction add		230
• Transformations en Z		231
• Produits, fonctions product , Product et value		232
• Fonction mul		233

Chapitre 13 Simplifications, Manipulations.	
Fonctions Spéciales.	
Calculs avec unités et Constantes physiques	235
Simplifications, manipulations	
• Introduction, position du problème	235
• Fonctions expand , combine et convert	236
• Simplifications et valeurs complexes	239
• Options de la fonction simplify	240
• Fonctions is et coulditbe	241
• La fonction is	241
• La fonction coulditbe	242
• Fonctions assume et additionally	242
• Fonction assume	242
• Fonction additionally	244
• Fonctions about et getassumptions	246
• Fonctions assume et protect	247
• Évaluations des fonctions et variables contraintes	247
• Environnement assuming	248
• Relations de simplification	249
• Les fonctions factor et normal	250
• Autres exemples de transformation	251
Fonctions spéciales	252
• Les fonctions expand et convert	252
• FunctionAdvisor	253
• convert(..., StandardFunction)	254
Utilisations d'unités physiques	255
• Conversions d'unités, conversions d'échelles	255
• Environnement Units[Natural]	256
• Environnement Units[Standard]	257
• Librairie ScientificConstants	257
Chapitre 14 Equations ordinaires	261
• Résolutions exactes	261
• Équation, définition et type. Fonction solve	261
• Solutions, absence de solution,	
Non résolutions, Résolutions partielles	263
• Variable globale _SolutionsMayBeLost	263
• Fonctions solve et variables contraintes	264

• Fonction de Lambert	264
• Ensemble infini de solutions	265
• Résolutions de systèmes d'équations	266
• Assignation des solutions, fonction assign	267
• Résolutions des inéquations et systèmes d'inéquations	268
• Variable globale _EnvExplicit	269
• Résolutions numériques, fonction fsolve	270
• Équations non polynomiales	270
• Équations polynomiales	274
• Module RootFinding	276
• Résolution numérique d'une équation dans le plan complexe	277
• Système de polynômes bivariés	278
• Solution par homotopie	279
• Résolutions des équations de récurrence, fonction rsolve	279
• Résolutions de Systèmes Linéaires	280
Chapitre 15 Equations différentielles	283
• Résolutions exactes, fonctions dsolve et odetest	283
• Conditions initiales	284
• Transformation des solutions en fonctions	284, 285
• Opérateurs diff et D , conversion	285
• Divers aspects de l'intégration des équations différentielles	286
• Résolution par transformations intégrales	286
• Fonctions discontinues définissant les équations différentielles	287
• Solutions explicites et implicites	288
• Classes des équations différentielles, fonction odeadvisor	289
• Développement en série de la solution	291
• Systèmes d'équations différentielles	292
• Résolutions numériques, conditions initiales, aux limites	293
• Tracés, fonction odeplot de plots	294
• Création d'une procédure (fonction) solution	295
• Systèmes d'équations	296
Chapitre 16 Représentations graphiques 2D	299
• Représentations cartésiennes , fonctions plot	299
• Modifications interactives du dessin	299
• Tracés de fonctions définies par une procédure ou un opérateur	300
• Exemples des options de plot	303

• Couleurs	303
• Intervalle des ordonnées	303
• Discontinuités	303
• Système d'axes orthonormés	304
• Axes infinis	305
• Précision du tracé	305
• Tracés de données discrètes	305
• Modification du mode de tracé (lignes, points, etc)	306
• Tracés multiples	307
• Ajout d'un titre, modification des polices, etc.	307
• Modification du style des axes	308
• Courbes paramétriques	309
• Courbes en coordonnées polaires , option coords	310
• Bibliothèque plots	312
• Options permanentes	312
• Fonctions définies de façon implicite, fonction implicitplot	312
• Tracés logarithmiques, fonctions loglogplot	313
• Tracés dans le plan complexe, fonction complexplot	314
• Transformations conformes, fonction conformal	315
• Représentations animées, fonction animate	316
• Structures INTERFACE_PLOT, fonction display de plots	317
• Animations avec la fonction display , option insequence	320
• Création de fichiers dessins	320
• Représentations graphiques des solutions d'équations différentielles	321
• Bibliothèques plottools et geometry	324
• Création d'une fonction graphique	325
Chapitre 17 Représentations graphiques 3D	327
• Surface définie par $z = f(x,y)$	327
• Tracés des axes	328
• Limitation des ordonnées du graphique	328
• Modification du rendu des surfaces	328
• Modification de l'orientation du système d'axes	329
• Tracés multiples	330
• Tracés multiples de mêmes caractéristiques	330
• Tracés multiples composés, fonction display de plots	330
• Limitation du domaine du tracé, définition du tracé des surfaces	331
• Tracés de données contenues dans un fichier, fonction surfdata	332

• Surfaces en coordonnées paramétriques	333
• Surfaces en coordonnées sphériques, cylindriques , option <i>coords</i>	334
• Coordonnées sphériques	334
• Coordonnées cylindriques	336
• Bibliothèque plots	338
• Options permanentes	338
• Surfaces en coordonnées tubulaires	338
• Courbes paramétriques	338
• Surfaces définies implicitement	339
• Visualisation d'une matrice	340
• Surfaces animées, animate3d	340
• Fonction display de plots , option <i>insequence</i>	340
Chapitre 18 Eléments du langage de programmation	
Symboles et chaînes de caractères	343
• Les boucles	343
• Les boucles for	343
• Règles de syntaxe	344
• Boucles sans variable de comptage	346
• Les boucles for-while	346
• Boucle commandée par des opérandes	347
• Contrôle du déroulement d'une boucle	348
• Boucle do-break	348
• Branchements conditionnels : bloc if	349
• Symboles et chaînes de caractères	350
• Définitions	350
• Fonctions de recherche	352
• Bibliothèque StringTools	353
• Conversions	354
• Fonction parse	354
• Concaténation	355
• Fonction cat	355
• Opérateur 	356
Chapitre 19 Procédures et modules. Librairies	359
• Procédures	359
• Définition, proc ... end proc	359
• Ponctuation et style d'écriture	360

• Contenus des procédures	360
• Principe de fonctionnement et résultats rendus par les procédures	361
• Arguments des procédures. Variables locales et globales	363
• Variables locales et globales	366
• Déclaration local	366
• Transfert de données sans utiliser les arguments	368
• Déclaration global	369
• Variables globales d'environnement	369
• Arguments optionnels, variables args et nargs	370
• Appels des fonctions de MAPLE dans une procédure	371
• Gestion des erreurs	371
• La fonction error	372
• Le bloc try/catch	374
• Retour anticipé, instruction return , mot clé procname	377
• Options cache , remember et trace	379
• Option cache	379
• Option remember	380
• Option trace	380
• Contrôle simplifié des arguments	381
• Génération d'une procédure par une procédure	383
• Opérandes d'une procédure	385
• Modules	386
• Définition, exportation	386
• Noms exportés particuliers	388
• Nom ModuleApply	388
• Noms ModuleLoad et ModuleUnload	389
• Variables structurées, fonction Record	389
• Définition, exportation	389
• Types	391
• Bloc use...in/end use	392
• Héritage	392
• Surcharge des opérateurs	394
• Enregistrement et lecture des librairies	396
• Création, sauvegarde et modification	396
• Création de pages d'aide	399
• Utilisation	400
• Sauvegarde simple, lecture de procédures, de fonctions, etc.	401

Chapitre 20	Lecture et écriture de fichiers	403
•	Lecture interactive de données au clavier	403
•	Fonction readstat	403
•	Fonction readline	404
•	Lecture de données dans un fichier	404
•	Désignation du fichier	405
•	Fichier de données numériques seules.	405
•	Fonction readdata	405
•	Manipulation des données	406
•	Fichier de données numériques mélangées à du texte	410
•	Fonction readline	410
•	Fonction sscanf	413
•	Ecriture dans un fichier	414
•	Mode d'écriture <i>Maple Notation</i>	414
•	La fonction writedata	414
•	La fonction fprintf	417

1 - Les nombres dans Z, Q, R et C

Avertissement

Certains aspects de ce chapitre sont élémentaires et peuvent donner l'impression qu'une lecture rapide est suffisante. En réalité, on a profité de la simplicité des exemples pour présenter des points très importants (et pas toujours évidents) du fonctionnement de MAPLE. Une lecture attentive, y compris de l'annexe *Commentaires sur l'utilisation de MAPLE* en fin de ce chapitre, est vivement conseillée pour aborder la suite.

Nombres entiers

Entrées des commandes : Pour qu'une ligne d'opérations soit effectuée et affichée elle doit être écrite après un prompt (ici ">") et doit se terminer par un ";" puis par *Return*. Si la ligne se termine par ":" puis par *Return*, elle est exécutée mais aucun résultat n'est affiché. Nous aurons l'occasion de revenir sur l'utilité de cette écriture.

Les nombres entiers et les opérations sont représentés de façon très simple : la multiplication est désignée par une astérisque *obligatoire* (avis aux utilisateurs de *Mathematica*) et les parenthèses définissent la priorité des opérations. *Notez le ; terminant la commande*

```
> (-3+1)*(4-1)-3;
```

-9

Pour une commande longue MAPLE coupe l'expression automatiquement et passe à la ligne suivante (césure). On peut cependant souhaiter, pour une meilleure présentation, effectuer soi-même ces césures. Ainsi dans l'exemple suivant la première ligne se termine par *Majuscule-Return*. MAPLE attend alors la suite de l'expression qui sera donnée à la ligne suivante. Celle-ci se termine par un ";" puis *Return*. Des espaces peuvent permettre d'améliorer la lisibilité mais ils ne peuvent pas être insérés dans l'écriture d'un nombre (par exemple 1000 et non 1 000)

```
> ((3+7)*(5-3)+2) - (-32*2/(64-4*8)) + (6-4)/2  
+ (121-81)/2 - 3*(5/2+3/2) - 720/5/6/4/2;
```

30

```
> 3*1 000;
```

Error, unexpected number

Important

On peut toujours, à l'aide de la souris, positionner le pointeur (petite barre verticale clignotante) sur une commande déjà exécutée, la modifier (ou non) et l'exécuter à nouveau. Il faut alors noter les deux points suivants:

- L'état de MAPLE n'est pas nécessairement celui qui est affiché par la feuille de calcul... ! Deux exemples d'illustration sont donnés dans les *Commentaires sur l'utilisation de MAPLE*, § IV en fin de chapitre.

- Pour exécuter une commande, le pointeur peut se trouver n'importe où sur la ligne de commande quand on tape *Return*. Après la correction d'une commande il est inutile de perdre son temps à déplacer ce pointeur en fin de ligne avant de taper *Return* comme on voit le faire souvent chez les débutants.

Symboles %, %% et %%% : Le symbole % (dito; souvent avec deux t en anglais) désigne le résultat de la dernière instruction exécutée *dans l'ordre chronologique* et peut être utilisé dans une expression. Le double symbole %% et le triple symbole %%% désignent les résultats précédents.

```

> 2*%;
3*% ;
3*(%%%/2+1);

```

60
90
48

Attention : ces symboles sont d'un usage commode mais doivent être manipulés avec précautions (voir à la fin de ce chapitre *Commentaires sur le fonctionnement de MAPLE*, § IV).

Autre exemple permettant une présentation sur plusieurs lignes de plusieurs commandes: la première ligne se termine par un *Majuscule-Return*, la deuxième par *Return*. Chaque expression se termine par un ; ce qui entraînera l'évaluation de chacune et l'affichage sur des lignes séparées. On peut écrire un commentaire à la fin de la ligne (après le : ou le ;) en le faisant précédé du caractère #. L'élévation à une puissance s'écrit à l'aide du symbole ^

```

> -6*(-4)    ;   -10/(-2); # Les parenthèses sont ici nécessaires
(-3-1)^2    ;   (1/2)^(-2);

```

24
5
16
4

Si on termine chaque expression par une virgule et la dernière par point-virgule on obtiendra aussi les résultats. Ceux-ci s'afficheront sur la même ligne mais "l'objet" MAPLE créé est une *suite* que nous examinerons au chapitre 3, *Intervalles, Listes, Suites, Ensembles*.

```

> -6*(-4)    ,   -10/(-2),
(-3-1)^2    ,   (1/2)^(-2);

```

24, 5, 16, 4

MAPLE connaît beaucoup de fonctions définies pour les entiers comme par exemple la fonction factorielle qu'on utilise avec la notation standard

```

> 40!-6^36;

```

815915283247897734335296844797625358725828050944

Le nombre maximal de chiffres autorisés, bien que très grand, est limité et sa valeur dépend de l'ordinateur utilisé. On peut l'obtenir avec la commande (le résultat indique que les entiers peuvent contenir plus de 268 millions de chiffres !)

```

> kernelopts(maxdigits);

```

268435448

On peut *affecter*, on dit aussi *assigner*, un nombre à un nom en utilisant la notation :=. **Attention:** le signe = seul a une autre signification sur laquelle nous reviendrons (définition d'une équation)

```

> n:=2^77-1;

```

n := 151115727451828646838271

Elision

Il se peut que les nombres sur lesquels on travaille soient très grands et que leur affichage devienne encombrant. On peut, à l'aide des commandes suivantes, limiter par *élision* le nombre de chiffres affichés (en

syntaxe française, "l(a)'éision est une éision"). La première commande ("threshold" signifiant "seuil" et "digits", "chiffres") fixe le nombre de chiffres à partir duquel le mécanisme s'applique. Les deux suivantes fixent le nombre de chiffres qui doivent précéder et suivre

```
> interface(elisionthreshold=10):  
interface(elisiondigitsbefore=2):  
interface(elisiondigitsafter=3):  
n;
```

15[...19 digits...]271

Bien entendu, ceci ne concerne que l'affichage et l'intégrité de n est préservée ! On annulera l'éision en posant un seuil élevé

```
> interface(elisionthreshold=10000):
```

Fonctions relatives aux entiers

On peut se poser la question " n est-il premier ? " en utilisant la fonction **isprime** (lire : is prime ?). La réponse *false* signifiant "faux", la réponse est "non"

```
> isprime(n);  
false
```

On obtient alors sa décomposition en facteurs premiers avec la fonction **ifactor** (integer factorization)

```
> ifactor(n);  
(23) (89) (127) (581283643249112959)
```

Autre exemple de fonction sur les nombres entiers: les coefficients du binôme $C(n, p) = \frac{n!}{p! (n-p)!}$ sont donnés

par la fonction **binomial(n, p)**. On peut par exemple vérifier la relation entre les coefficients du binôme $C(n+1, p+1) = C(n, p) + C(n, p+1)$

```
> binomial(5,3), binomial(4,2)+binomial(4,3);  
10, 10
```

Aide MAPLE

On ne détaillera pas dans ce manuel toutes les fonctions de MAPLE relatives aux calculs sur les nombres entiers pas plus que bien d'autres fonctions. Celles-ci peuvent être découvertes dans l'aide (help) en ligne (c'est-à-dire directement à l'aide du logiciel). Il faut aussi savoir que toutes les fonctions ne sont pas utilisables au moment du lancement de MAPLE. Elles ne le seront qu'après les avoir rendues accessibles dans des bibliothèques avec la fonction **with**. C'est un point sur lequel nous reviendrons en temps utile. L'aide en ligne, très développée, peut être obtenue de trois façons:

- Soit en donnant le nom de la fonction et en la faisant précéder d'un point d'interrogation
`> ?binomial`

Une fenêtre s'ouvre alors et donne la définition, des explications et des exemples. On peut aussi utiliser la fonction

```
> example(binomial);
```

- *La principale source d'informations* est dans le menu **Help** qui offre plusieurs options dont
 - L'option **Using Help** qui ouvre une fenêtre contenant un classement hiérarchisé des fonctionnalités de MAPLE et un guide d'utilisation de l'aide en ligne.
 - L'option **Help on context**. Lorsque le pointeur d'insertion (petite barre verticale clignotante indiquant

où l'on se trouve dans la fenêtre) est sur un mot, par exemple *binomial*, l'item *Help on "binomial"* du menu Help permet d'ouvrir la fenêtre correspondante de l'aide.

- L'option **Topics Index** ouvre une fenêtre dans laquelle on indique un mot pour lequel on recherche une information. Dès l'entrée de la première lettre une liste de tous les mots clés de MAPLE commençant par cette lettre s'affichent. Au fur et à mesure que l'on précise le mot, la liste se réduit. Un double "clic" sur la ligne du mot souhaité ouvre la fenêtre d'aide.

- L'option **Search**. Après avoir donné le mot complet, la liste des points d'entrée de l'aide en ligne associée à ce mot sont donnés sous la forme d'une liste. Un double "clic" sur la ligne souhaitée ouvre la fenêtre d'aide correspondante.

- L'item **Math Dictionary** donne environ 5000 références: définitions mathématiques, mathématiciens, etc.

- Les pages d'aide contiennent des **hyperliens** (en vert souligné) qui renvoient sur des pages associées qui complètent les informations.

- On trouvera aussi des fonctions spécialisées comme **FunctionAdvisor** (des exemples seront donnés au cours des développements des chapitres et plus particulièrement à la fin du chapitre 13, *Simplifications, Manipulations*) ou **odeadvisor** (voir chapitre 15, *Equations différentielles*).

On recommande vivement de compléter ces informations de base par une exploration des possibilités de l'aide en ligne que l'on trouvera avec l'option "Using Help" du menu Help.

Nombres Rationnels

Les nombres rationnels sont considérés par MAPLE comme tels s'ils ne sont pas simplifiables, c'est-à-dire comme des rapports de deux entiers et non comme leurs approximations décimales. L'évaluation d'un résultat non entier reste sous sa forme rationnelle réduite au plus petit dénominateur

```
> 1/4-1/3;  
243/72+45/2-456/132;
```

$$\begin{array}{r} -1 \\ \hline 12 \\ \hline 1973 \\ \hline 88 \end{array}$$

et par conséquent peut être renvoyé sous la forme d'un entier

```
> 4*(15/32+2*(-3/192+1/32));  
2
```

Nombres Irrationnels

Comme pour les nombres rationnels, les nombres irrationnels sont conservés en tant que tels (sous une forme symbolique) s'ils ne sont pas simplifiables. Par exemple, pour une racine carrée on dispose de deux notations équivalentes : ${}^{\wedge}(1/2)$ ou **sqrt** (square root) :

```
> a:=3^(1/2);  
a:=sqrt(3);  
sqrt(9);
```

$$\begin{aligned} a &:= \sqrt{3} \\ a &:= \sqrt{3} \\ 3 & \end{aligned}$$

Le symbole a est alors associé, non à une approximation numérique, mais à un **nombre symbolique** positif dont le carré vaut 3 exactement, etc.

```
> a^2;
```

3

MAPLE effectue parfois automatiquement des transformations pour présenter un nombre

```
> b:=2^(-1/3);  
b^(-3);
```

$$b := \frac{1}{2} 2^{(2/3)}$$

2

Il n'effectue pas nécessairement des calculs qui peuvent paraître évidents à l'utilisateur. Ici on doit le forcer à effectuer le travail à l'aide de la fonction **simplify** sur laquelle nous reviendrons souvent et notamment au chapitre 13, *Simplifications, Manipulations*

```
> sqrt(6)/sqrt(2);  
simplify(%);
```

$$\frac{1}{2} \sqrt{6} \sqrt{2}$$

$\sqrt{3}$

Nombres Remarquables

MAPLE connaît, *au sens de leurs propriétés* et pas seulement par leurs approximations numériques, quelques nombres remarquables comme par exemple **Pi** = π . **Attention**: notez le *P* pour lequel la **majuscule est obligatoire**. Sinon la lettre π s'affichera aussi, mais désignera un nom, pas le nombre... De même **PI** désigne seulement la lettre majuscule Π . Avec cette définition symbolique du nombre, MAPLE peut exprimer certains résultats classiques de fonctions qui sont, si possible, évaluées automatiquement (GAMMA est la fonction eulérienne Γ)

```
> Pi;  
cos(Pi/4);  
GAMMA(1/2);
```

π

$$\frac{1}{2} \sqrt{2}$$

$\sqrt{\pi}$

Pour illustrer la remarque sur la syntaxe de π

```
> cos(Pi);  
cos(pi), cos(PI);
```

-1

$\cos(\pi), \cos(\Pi)$

Certaines valeurs que MAPLE sait calculer (ce n'est pas toujours le cas) ne le sont cependant pas et il faut forcer l'évaluation (voir **convert**, Chapitre 13, *Simplifications, Manipulations...*)

```
> cos(Pi/5);
```

```
convert(cos(Pi/5), radical);
```

$$\cos\left(\frac{1}{5}\pi\right)$$
$$\frac{1}{4} + \frac{1}{4}\sqrt{5}$$

La base **e** des logarithmes népériens n'est pas une constante prédéfinie. Elle l'est seulement lors de l'*affichage* de **exp(1)**

```
> exp(1);
```

e

Attention: Le symbole **e** affiché ci-dessus n'a aucun rapport avec le nom *e* d'une variable, sauf bien sûr si on lui assigne cette valeur. Ici, *e* n'est encore qu'un nom indéfini.

```
> ln(e);
```

ln(e)

Après assignation

```
> e:=exp(1);  
e;  
ln(e);
```

e := **e**

e

1

MAPLE ne connaît que deux autres nombres réels remarquables prédéfinis et que nous renconterons (la *constante d'Euler* γ et la *constante de Catalan*).

```
> constants;
```

false, γ , ∞ , true, Catalan, FAIL, π

Exercice: Avec l'aide en ligne, trouver une définition de la constante de Catalan.

MAPLE est capable de reconnaître (quelquefois après simplification) l'équivalence de deux expressions comme par exemple $\exp(1/2)$ et \sqrt{e} .

```
> exp(1/2)-sqrt(e);
```

0

Nombres Décimaux

Ils sont notés de façon standard (avec un point) et ne doivent pas, comme pour les entiers, contenir d'espaces (les espaces entourant le + sont par contre autorisés).

```
> -1.2 + 0.035;
```

-1.165

Il existe trois façons de définir des nombres avec des puissances de 10, la plus simple utilisant la notation scientifique (indifféremment *e* ou *E*). Pour le dernier exemple, les arguments de la fonction **Float** doit être des entiers (risques d'erreurs)

```
> -13.345E-27, 45e543, 1.542*10^(-30), Float(1542,-33);
```

$-1.3345 \cdot 10^{-26}, 4.5 \cdot 10^{544}, 1.542000000 \cdot 10^{-30}, 1.542 \cdot 10^{-30}$

Point décimal et opérateur "."

Attention : on ne pourra pas laisser le point décimal seul devant e ou E

> **45.e6;**

Error, missing operator or `;

et il faudra écrire (sans caractère d'espacement)

> **45.0e6;**

$4.50 \cdot 10^7$

La raison vient de ce que "." peut être compris par MAPLE, contrairement à "*", comme un opérateur de multiplication **non commutatif**

> **y*x-x*y;**

y.x-x.y;

0

$(y \cdot x) - (x \cdot y)$

Lorsqu'il peut exister une ambiguïté (ce qui n'était pas le cas dans l'exemple précédent) le "." doit être **obligatoirement** précédé d'un espace pour être compris comme opérateur de multiplication

> **4.5 , 4 . 5 , 4 . 5;**

4.5, 20, 20

Ici MAPLE comprend "45 x 0.". On se doute qu'une telle faute de frappe peut avoir des conséquences redoutables dans une expression plus compliquée !

> **45 .0e6;**

0.

Maintenant MAPLE détecte une erreur

> **45. 0e6;**

Error, unexpected real number

On ne saurait trop recommander de réservier l'usage de cet opérateur aux seules opérations non commutatives comme par exemple les produits de matrices (voir chapitre 4, *Vecteurs, Matrices et Algèbre linéaire*).

Evaluation décimale implicite

Le mélange dans une expression de nombres exprimés en notation décimale avec des nombres entiers ou rationnels entraîne l'évaluation en valeur décimale.

> **0.5 + 1/3 + 1;**

1.833333333

alors que

> **1/2 + 1/3 + 1;**

$\frac{11}{6}$

Notez la différence entre ces deux résultats:

```
> 1./7*10^(-10);  
1 /7*10^(-10);
```

$$\frac{1}{7000000000} \cdot 10^{-11}$$

Lorsqu'une fonction est invoquée avec un argument décimal ou un mélange conduisant à un calcul décimal sans composante symbolique de l'argument, MAPLE calcule le résultat sous une forme décimale

```
> exp(1.5 - 3/4);  
2.117000017
```

Par contre, *priorité est toujours donnée aux calculs symboliques* et MAPLE ne prendra pas l'initiative de remplacer π ou $\sqrt{2}$ par leurs valeurs décimales approchées.

```
> 1. + sqrt(2);  
Pi + 5.;
```

$$\frac{1. + \sqrt{2}}{\pi + 5.}$$

De même

```
> exp(1. + Pi);# Voir le paragraphe suivant sur la fonction evalf  
exp(1. - sqrt(2));
```

$$e^{(1. + \pi)}$$
$$e^{(1. - \sqrt{2})}$$

Exercice : Comparer ces deux exemples et expliquez la différence

```
> 1. + sqrt(2);  
1 + sqrt(2.);
```

$$1. + \sqrt{2}$$
$$2.414213562$$

La représentation des nombres décimaux est limitée en magnitude

```
> exp(-2.0e10), exp(1.0e30), exp(-1.0e29)*exp(1.0e30);
```

$$-0., \text{Float}(\infty), \text{Float(undefined)}$$

Le plus grand et le plus petit nombre décimal positif que peut représenter MAPLE sur l'ordinateur utilisé sont donnés par

```
> Maple_floats(MAX_FLOAT), Maple_floats(MIN_FLOAT);
```

$$1 \cdot 10^{2147483646}, 1 \cdot 10^{-2147483646}$$

Il est possible de convertir un nombre décimal en nombre fractionnaire. MAPLE renvoie alors une fraction réduite au plus petit dénominateur

```
> r_Pi:=convert(3.1415927,rational);  
ifactor(%);
```

$$r_Pi := \frac{126003}{40108}$$

$$\frac{(3)(97)(433)}{(2)^2(37)(271)}$$

On obtient la partie entière d'un nombre avec la fonction **trunc** ou l'entier le plus proche avec **round**.

```
> a:=1.648;
trunc(-a);
round(-a);#-2 est l'entier le plus proche de -1.648...
a := 1.648
-1
-2
```

On remarquera qu'il n'est pas nécessaire de forcer au préalable l'évaluation décimale de l'argument.

```
> trunc(-Pi), round(sqrt(5));
-3, 2
```

Evaluations décimales des nombres

La fonction evalf

MAPLE est avant tout un calculateur symbolique mais on peut toujours obtenir une approximation décimale d'une expression numérique avec la fonction **evalf** (contraction de "Floating-Point Evaluation")

```
> evalf(5/6);
evalf(100!);
0.8333333333
9.332621544 10157
```

MAPLE ne dit rien sur l'évaluation de $\exp(1/2)$. Une évaluation décimale passe par la fonction **evalf**

```
> a:=exp(1/2);
evalf(a);
a := e(1/2)
1.648721271
```

Le nombre de chiffres significatifs rendus par MAPLE est fixé par défaut à 10 par le symbole global prédéfini **Digits** (le sens du mot "global" sera défini au chapitre 19, *Procédures*).

```
> Digits;
10
```

Cependant le nombre n de chiffres significatifs de x peut être fixé en écrivant **evalf[n](x)**. Ceci n'affecte que le calcul demandé et ne modifie pas la valeur de la variable **Digits**

```
> evalf[100](Pi);
```

Notez le "\ " en fin de première ligne qui indique que l'affichage du nombre se poursuit.

3.141592653589793238462643383279502884197169399375105820974944592307816406286208998628\

Il existe aussi une autre forme, **evalf(x,n)**, utilisée dans les précédentes versions et encore valable. La forme **evalf[n](x)** est maintenant recommandée.

On peut changer la valeur de **Digits** pour une série de calculs puis, si nécessaire, revenir à la valeur standard. On notera que certaines des commandes suivantes se terminent par deux points. En effet l'affichage de leurs résultats n'est pas d'un grand intérêt ou peut, pour des affichages longs, encombrer inutilement la lecture

```
> Nb_digits:=Digits: # Pour mémoriser la valeur par défaut
Digits:=50:
evalf(a);   evalf(Pi)/5;
1.6487212707001281468486507878141635716537761007101
0.62831853071795864769252867665590057683943387987502
```

```
> Digits:=Nb_digits: # Pour restituer la valeur par défaut
evalf(a);   evalf(Pi)/5;
1.648721271
0.6283185308
```

La fonction evalhf

Les langages de programmation comme FORTRAN, C ou C++ travaillent avec une précision fixe (6 à 7 chiffres en simple précision et 15 à 16 chiffres en double précision pour un ordinateur 32 bits) et la comparaison avec les possibilités offertes par MAPLE semble difficile à soutenir à première vue. En fait MAPLE génère ses calculs numériques à partir d'algorithmes dont la précision peut être étendue (arithmétique émulée) alors que les autres langages utilisent (sauf emploi de bibliothèques spécialisées) l'arithmétique câblée du processeur dont la précision ne peut évidemment pas être changée mais dont la vitesse d'exécution est beaucoup plus grande. La possibilité de fixer la précision va donc être au prix d'une baisse de la rapidité des calculs. On peut néanmoins indiquer à MAPLE d'utiliser l'arithmétique binaire câblée du processeur avec la fonction **evalhf** (Hardware Floating-Point Evaluation). Bien sûr on ne pourra plus fixer la précision qui dépendra de la machine sur laquelle on travaille et qui sera celle de sa double précision (généralement 15 à 16 chiffres).

Malgré l'utilisation d'evalhf, il n'en demeure pas moins que l'on travaille avec MAPLE dans un environnement de calcul symbolique qui reste très gourmand en mémoire et en temps de calcul. MAPLE ne saurait donc remplacer les langages de calcul numériques.

```
> a;
evalhf(a);

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

1.64872127070012819
```

Avec **evalhf** l'on ne pourra pas se fier au dernier ou deux derniers chiffres affichés (en raison des conversions "représentation binaire" <=> "représentation décimale").

```
> evalhf(7/3);
evalf[18](7/3);
2.33333333333333348
2.3333333333333333
```

ou encore

```

> evalhf(7/5);
evalf[18](7/5);
                                1.3999999999999990
                                1.4000000000000000

```

Attention: certaines fonctions de MAPLE ne sont pas calculables avec **evalhf** (voir aussi le paragraphe suivant relatif aux nombres complexes)

```

> evalf(GAMMA(2.3));
evalf(GAMMA(1.2,2.3)); # Fonction eulérienne incomplète
evalf(BesselJ(1,2.7));
                                1.166711905
                                0.1265378441
                                0.4416013791

```

```

> evalhf(GAMMA(2.3));
evalhf(GAMMA(1.2,2.3));
evalhf(BesselJ(1,2.7));
                                1.16671190519816025

```

```
Error, remember tables are not supported in evalhf
Error, remember tables are not supported in evalhf
```

On trouvera la liste des fonctions calculables avec

```
> ?evalhf/fcnlist
```

Attention : on notera que **Digits** est une variable très particulière relativement à la fonction **evalhf** et que le résultat, bien que cohérent, peut surprendre !!! Même si le nombre de chiffres affichés est de 18, MAPLE considère que le nombre effectif de chiffres significatifs est de 15, et évidemment quelque soit la valeur de **Digits**.

```

> Digits;
evalhf(Digits); # 10 devient 15. !!!
trunc(evalhf(Digits));
                                10
                                15.
                                15

```

Exercice : pouvez-vous expliquer la raison de la différence entre les résultats suivants :

```

> evalhf(2*Pi/3);
evalhf(2*Pi)/3;
                                2.09439510239319526
                                2.094395102

```

Nombres Complexes

Les nombres complexes, mis sous la forme algébrique, s'écrivent $a + I * b$. Le symbole prédéfini I désigne $\sqrt{-1}$. **Noter** la majuscule **obligatoire** pour I .

```
> sqrt(-1);
z:=3+2*I;
```

I

$$z := 3 + 2I$$

Ce symbole est protégé contre toute assignation

```
> I:=3;
```

```
Error, illegal use of an object as a name
```

Remarque: la notation *I* n'est pas figée. On peut la changer en utilisant la commande suivante et en choisissant une autre lettre (ici *i*) ou un symbole quelconque valide comme par exemple *Im*.

```
> interface(imaginaryunit=i);
```

Mais *i* deviendrait alors un nom protégé et on ne pourrait plus lui assigner une valeur (i.e. *i:=2* ; deviendrait interdit). De plus on ne pourrait pas, par exemple, l'utiliser pour indexer une boucle. Inversement *I* deviendrait un nom ordinaire. Ce changement ne vaudrait que pour la session en cours ou serait annulé après un ordre **restart** (voir cependant la partie 2 de ce chapitre, *Commentaires sur l'utilisation de MAPLE*, § IV-5).

Les fonctions numériques de MAPLE sont à arguments et valeurs complexes. On peut demander d'évaluer une expression complexe sous sa forme algébrique $a + I * b$ avec la fonction **evalc**

```
> c:=exp(z);  
evalc(c);
```

$$c := e^{(3 + 2I)}$$

$$e^3 \cos(2) + I e^3 \sin(2)$$

On peut extraire les parties réelles et imaginaires avec

```
> Re(c), Im(exp(-z^2));
```

$$e^3 \cos(2), -e^{(-5)} \sin(12)$$

On peut aussi en donner une représentation polaire

```
> polar(c);
```

$$\text{polar}(e^3, 2)$$

dans laquelle on reconnaît la valeur absolue et l'argument

```
> abs(c);  
argument(c);
```

$$\begin{matrix} e^3 \\ 2 \end{matrix}$$

Pour obtenir une expression conjuguée on écrira

```
> conjugate(c);
```

$$e^{(3 - 2I)}$$

Pour les nombres symboliques, les expressions conjuguées sont notées avec une barre de surlignement.

```
> conjugate(x-y)*c;
```

$$\overline{e^{(3 - 2I)}} \frac{x - y}{x - y}$$

On peut également demander une approximation décimale

```
> evalf(c);
```

-8.358532651 + 18.26372704 I

On retrouve, comme pour les réels, les mêmes règles générales d'évaluation des expressions quand les décimaux sont mélangés aux entiers et aux rationnels.

```
> 1 + 2*I - sin(3/2 + 2.0*I);  
-2.752771340 + 1.743446044  $I$ 
```

```
> exp(-1.0*I*Pi);  
evalf(%);  
 $e^{(-1.0 I \pi)}$   
-1. + 4.102067615  $10^{-10} I$ 
```

Attention : la fonction **evalhf** fonctionne aussi avec les complexes, mais présente certaines subtilités difficiles à expliquer ici sans avoir abordé la notion de type...

```
> evalhf(7/5+1/3*I);  
1.3999999999999990 + 0.3333333333333314  $I$ 
```

```
> x:=7/5+1/3*I;  
 $x := \frac{7}{5} + \frac{1}{3} I$ 
```

```
> evalhf(x);  
Error, unable to evaluate expression to hardware floats
```

Bien sûr, une représentation algébrique symbolique n'est pas toujours possible, mais une représentation décimale peut être calculée avec le nombre de chiffres significatifs souhaités.

```
> evalc(GAMMA(z));  
evalf[20](GAMMA(z));  
 $\Gamma(3 + 2 I)$   
-0.42263728631120216673 + 0.87181425569650686075  $I$ 
```

Identification de nombres décimaux

Supposons qu'après de laborieux calculs *numériques* d'intégrales, de résolutions d'équations ou autres, nous ayons obtenu la valeur suivante

```
> x := 3.878241035;  
 $x := 3.878241035$ 
```

On suspecte néanmoins que tous nos calculs pourraient avoir pour résultat une valeur symbolique exacte qu'on a pas su calculer. La fonction **identify** peut nous donner une indication sur la validité de notre intuition. Evidemment cette identification n'a de sens que dans la limite de la précision du nombre décimal proposé.

```
> identify(x);  
 $\frac{1}{3} + 2 \sqrt{\pi}$ 
```

Attention à un mythe très répandu chez certains étudiants: tout problème mathématique formulé exactement n'a pas **nécessairement**, si elle existe, sa (ou ses) solution exprimable exactement en termes de combinaisons de symboles connus (entiers, rationnels, radicaux, constantes spéciales connues, valeurs de fonctions élémentaires ou spéciales, etc. comme ci-dessus).

Le résultat donné par **identify** conforte cependant notre intuition et tend à montrer que nos calculs pourraient peut-être avoir un résultat symbolique exact, ce qui reste à démontrer... Encore deux autres exemples

```
> identify(1.769230769), identify(4.269867112);
 $\frac{23}{13}, \frac{1}{2}\pi e$ 
```

Une réponse négative n'est pas absolue

```
> identify(2.599079725);
2.599079725
```

Il y a en effet une infinité de combinaisons possibles et il peut être nécessaire d'aider **identify** en lui communiquant notre intuition (voir l'aide en ligne).

```
> identify(2.599079725, BasisSumConst=[1,Pi,sqrt(Pi),ln(2),exp(1),ln(Pi)]);
-e + 3  $\sqrt{\pi}$ 
```

Divers

MAPLE révèle encore de nombreuses ressources que l'on ne pourra pas toutes présenter...!

```
> convert(MDXV,arabic);
1515
```

```
> convert(3.265,binary), convert(207,octal);
11.01000011, 317
```

Corps padiques (ici décomposition canonique de Hensel de -1 en diadique)

```
> Digitsp:=10: padic[evalp](-1,2);
1 + (2) + (2)2 + (2)3 + (2)4 + (2)5 + (2)6 + (2)7 + (2)8 + (2)9 + O((2)10)
```

Nombres de Bernoulli, d'Euler, de Mersenne, de Bell, aléatoires... (voir des exemples pour ces derniers au chapitre 7, *Fonctions*)...

```
> bernoulli(2^5), euler(10), numtheory[mersenne]([7]),
combinat[bell](5), rand();

$$\frac{-7709321041217}{510}, -50521, 524287, 52, 474256143563$$

```

Pour plus d'informations sur les calculs avec unités physiques, voir la fin du chapitre 13 (M_e désigne ici la masse de l'électron).

```
> with(Units[Natural]): sqrt(3.0*mile^2+4.5*ft^2-2*m^2);
M[e]:=evalf(ScientificConstants:-Constant('m[e]', system=CGS, units));
2787.465291 [m]
```

$$M_e := 9.109381882 \cdot 10^{-28} [g]$$

```
> cat(`En l'an de grâce `, convert(2004,roman),
", le logiciel MAPLE 9.5...`);
En l'an de grâce MMIV, le logiciel MAPLE 9.5...
```

Commentaires sur l'utilisation de MAPLE

I - A propos de l'entrée des commandes

Il est fréquent, surtout au début, d'oublier le ; ou le : devant terminer une commande. MAPLE renvoie un message d'avertissement, mais exécute la commande. Pour effacer ce message, rajouter le ; ou le : à la fin de la commande avant de la relancer.

On répète aussi ce qui a déjà été dit : *inutile de déplacer le pointeur en fin de ligne pour exécuter une commande que l'on vient de corriger.*

Pour quitter MAPLE on utilisera la commande *Exit* du menu *File*.

II - A propos des noms dans MAPLE

- Les noms dans MAPLE (noms des variables, des expressions, des fonctions, etc.) doivent **commencer** par une lettre, mais peuvent contenir des chiffres ainsi que le caractère de soulignement _.

- MAPLE fait la distinction entre majuscules et minuscules. Ainsi le nom *a_1* est différent de *A_1*.

- On peut aussi créer des noms **commençant** par un _, mais ceci est **très vivement déconseillé** car MAPLE génère lui-même de tels noms et l'utilisateur risque de subir les conséquences d'interférences difficiles, voire très difficiles à détecter et pouvant conduire à des calculs faux.

- Les caractères d'espacement sont autorisés pour séparer les éléments d'une commande afin d'améliorer la présentation, mais ils ne sont pas admis dans l'écriture d'un nombre ou d'un nom (voir cependant chapitre 18, § *Symboles et chaînes de caractères*).

- S'ils sont correctement orthographiés, les noms de variables grecs tels alpha, beta, theta, phi, rho, etc. seront affichés avec l'alphabet grec. Si seule la **première** lettre est une majuscule la lettre grecque sera majuscule (à l'exception de Pi qui s'écrira π et désignera le nombre). On pourra utiliser la palette de symboles disponible dans le menu *View/Palettes/Symbol*. Noter que cette (ces) palette(s) qui s'affichent à gauche de la fenêtre peuvent être masquées pour agrandir l'espace de travail disponible.

- Bien que représentant une opération particulière, on peut définir des variables indiquées à l'aide de crochets (voir les chapitres 2 et 5). Cette écriture réclame cependant quelques précautions.

```
> Lambda[0]=rho*exp(theta)*cos(phi[i+1]);  
Lambda[0] = rho e^theta cos(phi[i+1])
```

III - A propos des « fausses » erreurs de syntaxe

L'analyseur syntaxique de MAPLE détecte les erreurs comme, par exemple, l'oubli de parenthèses ouvrantes ou fermantes. Mais il existe des erreurs assez fréquentes qui ne peuvent pas être détectées car ce sont des erreurs pour l'utilisateur, mais pas pour MAPLE qui est un calculateur symbolique.

- On a déjà rencontré l'opérateur d'assignation := . Si l'utilisateur tape par exemple
`> a = x+1 ;`

il aura peut-être voulu assigner l'expression *x+1* au nom *a*. Il a écrit en réalité une **équation** dont la syntaxe est parfaitement correcte pour MAPLE qui n'a aucune raison de détecter une erreur. De plus, aucune assignation n'a été effectuée.

- On peut aussi faire une erreur dans l'orthographe d'un nom. Par exemple

```
> x_1 := 1 :  
> a := x1+1 ;  
a := x1+1      (au lieu de 2)
```

Il n'y a pas d'erreur détectée car **x1** est compris par MAPLE comme un nom symbolique indéfini.

- De même, lorsque le nom d'une fonction est mal orthographié, MAPLE se contente de le renvoyer à l'identique. Il considère cette fonction comme symbolique et non définie et par conséquent ne signale aucune erreur, ce qui est un comportement normal pour un calculateur symbolique. Par exemple on obtiendra (*cosinus* au lieu de **cos**) :

```
> 3!*cosinus(Pi)/3+2;
2 cosinus(π) + 2
```

alors que l'on attend 0. Cette erreur est assez fréquente pour des fonctions moins habituelles. Elle est source de confusions, l'utilisateur non averti cherchant l'erreur ailleurs ou pensant qu'il ne sait pas se servir de la fonction. Un exemple fréquent est celui des calculs de limites avec la fonction de MAPLE qui s'écrit **limit** et non *lim*.

Ceci est également vrai pour une fonction dont la syntaxe est correcte mais qui n'est pas automatiquement accessible par MAPLE au moment de son lancement (voir § IV-7).

Pensez toujours à vérifier la syntaxe de vos commandes et assurez-vous que les fonctions utilisées sont bien connues de MAPLE avant de vous poser des questions inutiles !

IV - États de MAPLE

Au lancement, le programme MAPLE se charge dans la mémoire de la machine : c'est ce qui va réaliser les opérations de calcul. En même temps une zone mémoire, extensible au fur et à mesure des besoins, va être créée pour stocker l'environnement propre à l'utilisateur (noms des variables, expressions, valeurs numériques etc.) pendant la période d'utilisation (appelée session) : c'est ce qu'on appelle le *noyau* (kernel).

Attention : L'aspect de la feuille de calcul et l'état interne du noyau de MAPLE sont deux choses distinctes comme on va le voir sur deux exemples très simples. **C'est un point capital à comprendre !**

1) On peut « cliquer » sur la zone *input* d'une commande déjà calculée, la modifier et presser sur *Return* pour l'exécuter à nouveau. Il faut alors faire très attention au fait que le texte de la feuille de calcul ne reflète plus nécessairement l'état interne du noyau de MAPLE. Certains résultats peuvent apparaître faux car ceux des autres commandes ne sont pas modifiés en conséquence !

```
> ....
> a:=1;
      a:=1
> ...
> a+2;
      3
> ...
```

Maintenant on clique sur la commande **a:=1;** et on la modifie en écrivant **a:=2;** puis on l'exécute. La feuille de calcul aura alors l'aspect suivant

```
> ....
> a:=2;
      a:=2
> ...
> a+2;
      3
> ...
```

Tant que la commande **a+2;** n'aura pas été ré-exécutée le résultat affiché ne sera pas modifié. Si vous sauvez la feuille de calcul telle quelle ou si vous l'imprimez, un doute risque de germer dans votre esprit : “MAPLE sait-il réellement faire des calculs ?”. L'exemple est trivial, mais il n'en est pas toujours ainsi.

2) L'utilisation du dito (%) peut conduire aux mêmes types d'ambiguïtés car ce symbole désigne *le résultat de la dernière opération effectuée chronologiquement* et non nécessairement le résultat de la commande située sur la ligne précédente, ce que l'on a généralement tendance à penser en lisant les feuilles de calculs.

3) Lorsque l'on charge une feuille de calculs enregistrée, celle-ci s'affiche à l'écran, mais le noyau de MAPLE est vide. *Pour retrouver l'intégralité du noyau il faut ré-exécuter la feuille de calcul*. Ainsi, supposons que la feuille de calcul que vous venez de charger se résume à la seule ligne

```
> a:=2;
          a:=2
>
```

En entrant directement la commande a+1 ; sur le prompt final libre vous obtiendrez

```
> a:=2;
          a:=2
> a+1;
          a+1
```

car 2 n'aura pas été automatiquement assigné à la variable *a* au chargement de la feuille de calculs. Pour obtenir 3 il faudra d'abord ré-exécuter : > a:=2 ;

Pour exécuter la totalité d'une feuille de calcul on pourra utiliser *Edit/Execute/Worksheet*

4) L'ordre restart

Entrez systématiquement, bien que ce ne soit pas une nécessité, l'ordre

```
> restart;
```

comme première commande de votre feuille de calculs. Cette commande "vide" le contenu de la mémoire associée au noyau. Compte tenu de ce qui a été dit précédemment on peut comprendre que cette commande puisse être placée et exécutée n'importe où dans la feuille et les commandes ré-exécutées à partir du début. Mais vous risquez alors de repasser sur l'ordre **restart** par inadvertance et de vider malencontreusement le noyau (ceci est particulièrement vrai si on exécute la feuille par le menu *Edit/Execute/Worksheet*).

On peut par contre, introduire autant d'ordres restart que l'on souhaite dans une feuille de calcul pour séparer des calculs indépendants.

5) Initialisation de MAPLE

On peut prédefinir l'état de MAPLE au moment de son lancement *ou* après un ordre **restart**. Il suffit de placer dans le répertoire d'entrée de l'utilisateur un fichier contenant les ordres d'initialisation. **Attention:** le fichier doit être en ASCII sans formatage du genre .doc ou .rtf ; on utilisera pour le créer un éditeur en mode « caractères » comme ceux utilisés en programmation ou sous TeX (emacs, nedit, alpha, vi, pico, etc.). Le nom du fichier est dépendant du système d'exploitation et, pour plus de détails, on se reporterà à l'aide en ligne

```
> ?init
```

Un tel fichier contiendra simplement des ordres MAPLE valides qui seront exécutés de façon automatique. Suivant la ponctuation finale des ordres (; ou :) ils apparaîtront ou non. On montre ici un exemple de contenu de ce type de fichier dont l'exécution sera transparente (terminaison par : de tous les ordres).

```
Digits:=6: e:=exp(1): protect('e'):
with(LinearAlgebra,LinearSolve): # voir §7 et le Chapitre 4
interface(imaginaryunit=j):
```

On demande donc de travailler avec 6 chiffres significatifs par défaut, d'assigner la base des logarithmes népériens au nom *e* et de protéger ce nom contre toute autre assignation (voir

chapitre 2), d'accéder au solveur de systèmes linéaires et de fixer la syntaxe des nombres complexes comme $a+j*b$ et non $a+I*b$.

6) Gestion des noyaux

On peut ouvrir plusieurs feuilles de calcul simultanément (menu **File/New**) et gérer la dépendance entre les noyaux qui leur sont associés. On utilisera pour cela l'item **Preferences...** du menu **MAPLE 9[.5]**. L'onglet **General** propose l'item **Kernel mode** avec 3 options

- **Shared** (partagé) : les noyaux partagent les données. Si on tape `> a:=2;` dans une feuille de calcul, on obtiendra `3` en tapant `> a+1;` dans une autre feuille.

- **Parallel** : les noyaux sont indépendants. Si on tape `> a:=2;` dans une feuille de calcul, on obtiendra `a+1` en tapant `> a+1;` dans une autre feuille.

- **Mixed** : à chaque ouverture d'une nouvelle feuille, une fenêtre s'ouvre et donne le choix entre les deux options précédentes sous la forme

- **New** : le nouveau noyau sera indépendant des autres.

- **KernelConnection: Serveurs n** : où **n** désigne un noyau déjà ouvert. Le nouveau noyau partagera les données avec ce dernier. Une telle construction peut très vite devenir confuse et doit être réservée à des projets complexes. Ces liens doivent être re-créés à chaque ouverture des feuilles de calcul.

Pour une utilisation standard, choisir le mode **Parallel**.

L'option choisie (Shared, Parallel ou Mixed) peut être permanente, i.e. la même à chaque lancement de MAPLE, avec le bouton **Preferences/General/Apply Globally** ou seulement valable pour la session avec **Preferences/General/Apply to Session**.

7) Accès aux fonctions des bibliothèques

Certaines fonctions appartiennent à des bibliothèques auxquelles on n'a pas automatiquement accès au lancement ou après un ordre **restart** (sauf utilisation d'un fichier d'initialisation, voir § IV-5). Pour pouvoir accéder à ces fonctions, on utilisera :

soit

- a) la syntaxe dite « longue » qui permet *d'utiliser ponctuellement une fonction dans une instruction sans avoir un accès permanent* aux fonctions de la bibliothèque. Elle peut prendre deux formes

- **Bibliothèque:-Fonction(arguments)**. Cette écriture correspond aux bibliothèques récentes de MAPLE (bibliothèques de modules, voir chapitre 19).

- **Bibliothèque:[Fonction](arguments)**. Identique à la précédente et pour toutes les bibliothèques (les anciennes et le nouvelles).

soit

- b) l'opérateur **with**

- qui donne un accès permanent à **toute** la bibliothèque.

- `> with(Bibliothèque);`

- qui *donne un accès permanent seulement* à une ou plusieurs fonctions

- `> with(Bibliothèque,Fonction_1,Fonction_2,...);`

On trouvera un exemple d'utilisation au chapitre 4 *Vecteurs, Matrices,...*, § IV *Algèbre Linéaire* : bibliothèque *LinearAlgebra*, §§ *Accès aux fonctions de LinearAlgebra*.

V – Ouvertures et sauvegardes des feuilles de calculs

Une feuille de calculs peut, et le plus souvent doit être sauvegardée dans un fichier pour pouvoir être réutilisée ultérieurement. Deux situations sont alors possibles :

- **La feuille de calcul est nouvelle** : elle est automatiquement ouverte (vierge) au lancement de MAPLE et a pour nom « *untitled* » (sans titre). L'option **Save** du menu **File** permet de sauvegarder la feuille et MAPLE demande, à l'aide d'une fenêtre de dialogue, de lui donner un nom et de fixer le répertoire d'écriture. Cette opération peut aussi être réalisée

en « appuyant » sur le bouton marqué par le dessin d'une disquette. On peut également ouvrir une nouvelle feuille vierge avec le menu **File/New** ou avec le premier bouton de gauche, que l'on ait ou non fermé les autres feuilles de travail (voir cependant la fin du paragraphe précédent).

• **La feuille existe déjà :** après avoir lancé MAPLE on ouvre la feuille avec l'option **Open** du menu **File**. Cette opération peut être réalisée en « appuyant » sur le bouton marqué par le dessin d'un dossier. On peut également sauver la feuille de calcul sous un autre nom et/ou une autre forme à l'aide du menu **File/Save as...**

On peut demander aussi à MAPLE de faire une *sauvegarde automatique* toutes les *n* minutes à l'aide de **Preferences...** dans le menu **Maple 9[.5]**, onglet **General**, option **Auto save every n minutes**. Les fichiers de sauvegarde ont le même nom, mais les extensions sont remplacées par **MAS.bak**.

Conseils d'utilisation : lorsqu'on ouvre une nouvelle feuille, la sauvegarder immédiatement, même vierge, en lui donnant un nom. Effectuer ensuite fréquemment des sauvegardes en « appuyant » simplement sur le bouton marqué de l'icône « disquette ».

Les paragraphes suivants ne contiennent que des informations de base qui pourront être complétées avec l'aide en ligne obtenue soit avec la menu Help, soit avec les commandes suivantes

- > **?worksheet,managing**
- > **?worksheet,documenting**

VI – Organisation des feuilles de calcul

Feuilles de calcul

Le contenu d'une fenêtre MAPLE porte le nom de *feuille de travail* ou *feuille de calculs* ou *worksheet*. On remarquera que les ensembles « commande plus résultats associés » avec éventuellement des commentaires, sont démarqués sur la gauche par des sortes de crochets ouverts de longueurs variables. Ils délimitent des *groupes d'exécution* (*Group Ranges*). Un groupe d'exécution peut contenir cinq types de zones :

- les zones *input* pour entrer les commandes. Elles se reconnaissent par le prompt >. Dans les feuilles « help » l'exécution ou la modification des commandes données en exemple est interdite mais elles peuvent être facilement transportées par « copier-coller » dans les feuilles de calcul utilisateur.
 - les zones *output* où s'affichent les résultats.
 - les zones *paragraph* où l'on peut écrire du texte.
 - les zones *graphisme* où s'inscrivent les graphiques.
 - les zones *tableur* dont on ne dira rien dans ce manuel.

La plupart des commandes suivantes possèdent un raccourci clavier (voir dans les menus correspondants)

- **Pour supprimer une zone dans un groupe d'exécution**, cliquer n'importe où dans la zone et activer la commande *Delete Element* dans le menu *Edit*.
- **Pour insérer un groupe d'exécution** on utilisera l'une des options, *Before* (avant) ou *After* (après) du menu *Insert/Execution Group*, « avant » ou « après » se référant au groupe d'exécution pointé par le curseur, quelque soit la zone pointée.
- **Pour insérer un paragraphe de texte dans un groupe d'exécution**, on utilisera l'une des options, *Before* ou *After* du menu *Insert/Paragraph*, « avant » ou « après » se référant à la zone pointée par le curseur dans le groupe d'exécution.
- **Pour supprimer un groupe d'exécution**, le vider.
- **Pour couper un groupe d'exécution en deux**, placer le pointeur à l'endroit où l'on veut couper puis activer la commande *Edit/Split or Join/Split Execution group*.
- **Pour rassembler deux groupes d'exécution**, cliquer quelque part sur le premier groupe puis activer la commande *Edit/Split or Join/Join Execution group*.

- Pour supprimer l'affichage des crochets délimiteurs utiliser la commande *View>Show Group Ranges*.
- Certaines de ces opérations sont aussi possibles après sélection grâce aux « boutons » de la barre des menus de MAPLE.

On complètera ces informations avec la commande

>?worksheet,documenting

Sections, sous sections, hyperliens

Les feuilles de travail peuvent être organisées à l'aide de sections et sous-sections qui permettent de masquer ou non certaines parties de la feuille de calcul. Pour créer de telles sections dans les feuilles de travail, on utilisera le menu *Insert/Section* ou *Insert/Subsection*. Pour obtenir des informations supplémentaires exécuter la commande

>?worksheet,documenting,structuring2

On trouvera aussi dans les feuilles d'aide des « **hyperliens** » (*hyperlink*) visualisés par des mots soulignés de couleur différente qui renvoient par un simple « clic » sur le sujet mentionné. La construction d'hyperliens dans une feuille de calcul utilisateur vers des pages de l'aide en ligne ou vers une autre feuille de calcul est d'une grande simplicité grâce au menu *Edit/Hyperlinks...*

>?worksheet,documenting,hyperlinks

Il existe de nombreuses possibilités d'édition (modèles typographiques, insertions d'expressions mathématiques dans les textes, etc.) que l'on ne détaillera pas ici.

>?worksheet,documenting,styles

VII - Impression des feuilles de calculs

On utilisera *File/Print...* ou le « bouton imprimante ». Ne pas oublier de formater la mise en page avec *Edit/Print Setup...* On peut pré-visualiser le résultat avec *Edit/Print Preview...*

VIII - Affichage des résultats, étiquetage

MAPLE dispose de plusieurs options d'affichage que l'on peut sélectionner à l'aide du menu *Préférences* et l'onglet *Display*.

- *Le mode « MAPLE notation »* : les expressions sont affichées avec les mêmes notations que celles utilisées pour entrer les commandes (sous forme de caractères en ligne). Ces expressions sont éditables et on peut sélectionner et copier tout ou partie du texte.

- *Le mode « Character notation »* : les expressions sont affichées avec des caractères dessinant au mieux les notations mathématiques standard. Ces expressions sont éditables.

- *Le mode « Typeset notation »* : les expressions sont dessinées avec les notations mathématiques standard mais ne sont pas éditables.

- *Le mode « Standard Math Notation »* : les expressions sont dessinées avec les notations mathématiques standard mais sont éditables. C'est le mode d'affichage par défaut de MAPLE et c'est le plus agréable d'utilisation. Lorsqu'une sélection d'une zone Output est collée dans une zone Input, elle est convertie en mode caractères. *Cependant ce mode est aussi le plus gourmand en mémoire.*

- Pour l'item *Assumed variables*, voir le chapitre 13 et pour *Plot display*, voir les chapitres 16 et 17.

Avec les modes « *Character notation* » et « *Typeset notation* » (qui ne sont pas les modes par défaut) et lorsque les expressions affichées sont longues, MAPLE tente de repérer des sous-expressions qui se répètent et les remplace par des symboles %1, %2, etc. qu'il affiche ensuite. Cette méthode, dite d'**étiquetage**, peut être modifiée (voir l'aide en ligne **?interface** rubrique *labelling* et *labelwidth*).

2 - Assignations, Evaluations et Substitutions, Types, Eléments sur la structure des expressions

Assignations (affectations) et évaluations

MAPLE permet d'associer un nom à un " objet " quelconque (valide). Dans l'exemple suivant on assigne la constante entière 7 au nom E après évaluation de l'expression. **Notez** que la feuille de calcul commence par l'ordre **restart** comme conseillé au chapitre précédent.

```
> restart;  
E:=cos(0)+3!;
```

$$E := 7$$

Du point de vue informatique, le nom E est maintenant un identificateur qui pointe sur la constante 7. Pour savoir ce qui est assigné au nom E il suffit d'écrire

```
> E;
```

$$7$$

et MAPLE affiche l'objet pointé par E . A tout nom valide non protégé (voir plus loin cette fonctionnalité) on peut assigner n'importe quel objet de MAPLE comme, par exemple, une expression. **On notera que l'assignation précédente à E est automatiquement détruite et remplacée par la nouvelle.**

```
> E:=a*x+b*x^2+2*y-sin(Pi*a);  
E;
```

$$\begin{aligned}E &:= a x + b x^2 + 2 y - \sin(\pi a) \\&a x + b x^2 + 2 y - \sin(\pi a)\end{aligned}$$

Il est important de comprendre qu'à chaque fois que E sera invoquée, seule ou dans un calcul, MAPLE évaluera l'expression associée en fonction du contexte. Ici, en assignant à a une valeur ou une expression, l'expression de E est ré-évaluée en tenant compte de a

```
> a:=3*x+1;  
E;  
a := 3 x + 1  
(3 x + 1) x + b x^2 + 2 y - \sin(\pi (3 x + 1))
```

Toute expression demandée est évaluée en tenant compte de la définition de ses éléments.

```
> E+sin(Pi*a)+2*a;  
(3 x + 1) x + b x^2 + 2 y + 6 x + 2
```

Assignations multiples

On peut assigner une suite d'objet de MAPLE à une suite de noms. On entend par "suite" des éléments séparés par des virgules. Cette écriture économise la frappe au clavier ainsi que l'espace d'affichage et permet de regrouper des assignations qui ont, *pour l'utilisateur et pas nécessairement pour MAPLE*, un lien logique entre elles. Cette écriture améliore la concision de la feuille de calcul sans trop nuire à sa clarté.

```
> a1,b2,c3:=1,x-1,x+sin(x);  
b2;  
a1, b2, c3 := 1, x - 1, x + \sin(x)  
x - 1
```

Désassignation

On peut libérer l'assignation à a en écrivant

```
> a:='a';  
a := a
```

Attention: Il existe deux caractères distincts, ' et ` , qui ont des significations différentes (voir plus loin dans ce chapitre)

Maintenant l'identificateur a existe mais il est désassigné (il pointe sur lui-même) et l'évaluation de E renvoie son expression d'origine

```
> a;  
E;  
a  
a x + b x2 + 2 y - sin(π a)
```

Evaluation, assignation et désassignation

MAPLE évaluant toujours les expressions (sauf demande explicite ; voir plus loin), ce que l'on assigne à un nom peut être différent de ce que l'on écrit

```
> a:=x+1;  
Ex:=a*x+3;  
a := x + 1  
Ex := (x + 1) x + 3
```

Mais maintenant, le fait de désassigner a **après** avoir défini Ex ne changera pas son expression car on a assigné à Ex **l'expression évaluée** de $a x + 3$ (avec $a = x + 1$) et non $a x + 3$.

```
> a:='a';  
Ex;  
a := a  
(x + 1) x + 3
```

Voici un autre exemple d'assignations successives. Avant l'affectation à s , MAPLE évalue r qui vaut 2, etc.

```
> r:=2; s:=r; t:=s;  
r := 2  
s := 2  
t := 2
```

Si on 'libère' (désassigne) un des éléments on ne supprime pas les autres assignations

```
> s:='s';  
r, s, t;  
s := s  
2, s, 2
```

Attention: lors d'assignations multiples à l'aide de suites, le mécanisme n'est pas récursif

```
> u,v,w := 2,u,v;  
u, v, w := 2, u, v
```

Exercice : expliquez ce résultat

```
> u:=3;
  u,v,w := 1,u,v;
  u,v,w;
          u := 3
          1, 3, 3
```

Evaluation explicite

Remarque : le type d'opération que nous allons examiner reste d'un usage relativement limité mais doit être assimilé pour comprendre les principes de fonctionnement de MAPLE. Pour un usage systématique de cette d'opération *on préférera la notion de fonction* (voir le chapitre 7, *Fonctions*).

On peut demander l'évaluation d'une expression pour une valeur particulière d'un paramètre avec la fonction **eval**. Les arguments doivent être successivement l'expression à évaluer puis une égalité fixant la valeur du paramètre.

Notez l'utilisation du caractère = seul et non de := (ce ne sont pas des assignations mais des équivalences ; nous verrons que cette notation est aussi utilisée pour écrire des équations).

```
> P:=(3*x+1)*(x-2);
  eval(P,x=-2); # Evaluer P pour x=-2
          P := (3 x + 1) (x - 2)
          20
```

L'évaluation ne change en rien l'expression de P et aucune assignation à x n'a été effectuée

```
> P;
  x;
          (3 x + 1) (x - 2)
          x
```

Fonction inerte Eval et fonction d'activation value

MAPLE dispose d'une fonction inerte **Eval** (E majuscule) qui enregistre une opération d'évaluation sans l'exécuter, affichant seulement une écriture symbolique.

```
> Eval(P,x=-2);
          ((3 x + 1) (x - 2)) |_
          x = -2
```

D'autre part, une fonction **value** rend effective l'opération. Le symbole % vaut ici $\text{Eval}(P, x = -2)$ et la fonction **value** exécute l'opération.

```
> value(%);
          20
```

Ceci permet d'exprimer plus clairement le résultat d'un calcul en l'écrivant sous la forme d'une égalité (= et non :=). On utilise généralement pour cela la syntaxe suivante. **Notez** le ":" terminant la première commande et qui évite un affichage sans intérêt).

```
> Eval(P,x=-2): % =value(%);
          ((3 x + 1) (x - 2)) |_
          x = -2 = 20
```

Cette écriture est donc équivalente à

$$> \text{Eval}(P, x=-2) = \text{eval}(P, x=-2); \\ ((3x+1)(x-2)) \Big|_{x=-2} = 20$$

On peut également assigner l'opération et le résultat à des noms différents. D'autres opérateurs de MAPLE offrent aussi cette possibilité et nous aurons l'occasion d'en montrer l'utilité

$$> Q := \text{Eval}(P, x=-2); \\ q := \text{value}(Q); \\ Q = q; \\ ((3x+1)(x-2)) \Big|_{x=-2} = 20$$

Evaluations à valeurs multiples

La fonction **eval** admet plusieurs spécifications à condition de les écrire entre deux accolades (voir cette écriture avec la notion d'ensembles au chapitre 3, *Ensembles, Listes ...*). On remarquera également qu'un élément d'une expression peut être remplacé par une autre expression (encore que... ce ne soit pas si simple; voir le paragraphe suivant). On rappelle que la deuxième opération n'est qu'une évaluation de F , mais ne change pas F

```
> F := (x-y)/(3*x^2+2*y+1); \\ eval(F, {x=1, y=exp(-u)-1}); \\ F;
```

$$F := \frac{x - y}{3x^2 + 2y + 1} \\ \frac{2 - e^{(-1)}}{2 + 2e^{(-1)}} \\ \frac{x - y}{3x^2 + 2y + 1}$$

Exercice : calculer la valeur exacte de F pour $x=4$ et $y=2$ puis donner une expression décimale du résultat avec 6 chiffres significatifs

Modification d'une expression par la fonction eval

La fonction **eval** permet également de changer la valeur d'une constante dans une expression. Notez ici qu'au dénominateur le coefficient de y a changé et que x^2 est devenu x^3 . Comme l'opération **eval** ne change en rien F , elle n'a une utilité que si on ré-affecte le résultat soit à un autre nom (Q), soit à F lui-même

$$> F; \\ Q := \text{eval}(F, 2=3); \\ \frac{x - y}{3x^2 + 2y + 1} \\ Q := \frac{x - y}{3x^3 + 3y + 1}$$

Les deux exemples suivants montrent comment changer soit le coefficient de x^2 , soit x^2 en x^3 . Ici on change F en ré-affectant le résultat à F lui-même

```
> F:=eval(F,3*x^2=2*x^2);
F:=eval(F,x^2=x^3);
```

$$F := \frac{x - y}{2x^2 + 2y + 1}$$

$$F := \frac{x - y}{2x^3 + 2y + 1}$$

On peut aussi changer le nom d'une fonction.

```
> Esin:=x+2*y-sin(Pi*x);
Ecos:=eval(Esin,{x=y,sin=cos});
Esin := x + 2y - sin(πx)
Ecos := 3y - cos(πy)
```

Ces évaluations ne sont pas toujours aussi simples qu'elles en ont l'air comme le montrent les trois exemples qui suivent. Pour les comprendre nous devrons examiner dans un prochain paragraphe la façon dont MAPLE construit ses expressions.

Exemple 1 : Il est possible de donner une valeur à une "partie" d'une expression. On remarque que seul $x-2$ est effectivement remplacé dans P mais que l'écriture $x-2 = -2$ n'entraîne pas l'évaluation de l'expression pour $x=0$.

```
> P;
eval(P,x-2=-2);
```

$$(3x + 1)(x - 2)$$

$$-6x - 2$$

Exemple 2 : On veut remplacer $2x^3 + 2y$ dans F par le symbole t mais la commande est sans effet

```
> F;
eval(F,2*x^3+2*y=t);
```

$$\frac{x - y}{2x^3 + 2y + 1}$$

$$\frac{x - y}{2x^3 + 2y + 1}$$

Exemple 3 : Le remplacement de la constante 3 par 2 ne s'effectue pas et ne produit pas la simplification attendue

```
> E:=2*x/3;
eval(E,3=2);
```

$$E := \frac{2}{3}x$$

$$\frac{2}{3}x$$

Nous reviendrons sur ces particularités après avoir étudié les structures de ces expressions et la notion d'opérande.

Non évaluation d'un nom ou d'une expression

De façon générale, un nom ou une expression écrit entre deux caractères ' est interprété par MAPLE comme un "objet" ne devant pas être évalué, ou plus exactement dont l'évaluation doit être retardée. **Attention:** utiliser le caractère ' et non `

```
> a:=x^2+1;  
'a'+1;  
  
a :=  $x^2 + 1$   
a + 1
```

La commande ci-dessous montre que l'évaluation n'est pas annulée, mais seulement retardée

```
> %;  
  
 $x^2 + 2$ 
```

On peut également écrire (**evaluation as a name**)

```
> evaln(a)+1;  
  
a + 1
```

L'usage de '...' n'est cependant pas valable pour les nombres exprimés *strictement* de façon numérique (non symbolique) et pour lesquels MAPLE effectue toujours les opérations.

```
> a:='6+3';  
a:='6+3' + 'sqrt(2)^2 + 1';  
  
a := 9  
  
a :=  $10 + \sqrt{2}^2$ 
```

Nous reviendrons avec d'autres exemples sur l'utilité de cette non évaluation.

Exercices : Pouvez-vous expliquer cette suite de résultats ?

```
> a1:=2: a2:='a1'; a3:=a2; a1:=3; a2;  
a2 := a1  
  
a3 := 2  
  
a1 := 3  
  
3
```

Ou encore

```
> a:=3: b:=1:  
'a^2-b';  
%;  
%;  
  
'a^2 - b'  
 $a^2 - b$   
8
```

Test d'assignation

On peut tester (utile en programmation) si un nom est assigné avec la fonction **assigned**. Le résultat est un état logique : *true* (vrai) ou *false* (faux)

```
> assigned(a);  
assigned(Exy);  
true  
false
```

La commande suivante (**assined names**) donne la suite des noms qui ont été assignés depuis le début du travail ou depuis le dernier ordre **restart**

```
> anames(user);  
Ex, a2, a3, a1, c3, b2, Q, P, E, F, w, u, v, r, q, t, Ecos, b, a, Esin
```

Protection d'un nom

Il peut être utile de protéger un nom pour éviter toute assignation ultérieure malencontreuse. Cette opération se fait avec la fonction **protect**.

Attention aux graves interférences possibles entre la fonction **protect** et l'utilisation sans précaution de la fonction **setattribute** (voir à la fin du chapitre).

Sachant que MAPLE cherche toujours à évaluer les arguments d'une fonction avant de les lui transmettre, il est nécessaire de mettre le nom entre deux ' pour éviter l'évaluation. En effet, si dans l'exemple suivant on avait écrit **protect(a)**, MAPLE aurait cherché à exécuter **protect(3)**, ce qui n'a aucun sens. La fonction **protect** n'affiche rien.

```
> restart;  
a:=3;  
protect('a');  
a;  
a := 3  
3
```

Toute tentative d'assignation à un nom protégé est ensuite interdite et conduit à un message d'erreur, y compris la désassignation (qui n'est qu'une assignation particulière). Il est possible aussi de protéger un nom non assigné qui restera alors un nom symbolique.

```
> a:=4;  
a:='a';  
Error, attempting to assign to `a` which is protected  
Error, attempting to assign to `a` which is protected
```

L'opération inverse se fait avec la fonction **unprotect('nom')**

```
> unprotect('a');  
a:=4;  
a := 4
```

Certains noms **prédéfinis** de MAPLE, tels **Pi**, **I**, **O**, **gamma** ou **D**, ainsi que ceux des fonctions sont protégés (on peut les déprotéger, mais c'est une opération vraiment peu recommandée...!) :

```
> Pi:=22/7;  
sin:=0:
```

```
Error, attempting to assign to `Pi` which is protected
Error, attempting to assign to `sin` which is protected
```

Attention: si les noms des fonctions sont protégés, leurs images ne le sont pas ! La raison en sera donnée au chapitre 7, *Fonctions, § opérandes; table Cache et remember*. On évitera donc soigneusement d'écrire une bêtise comme :

```
> sin(0):=1;
sin(0) := 1
```

dont les conséquences pour les calculs seront désastreuses...

```
> x:=2: y:=1:
sin(x-2*y)^2 + cos(x-2*y)^2; # ?????
2
```

et on s'empresse de réparer le problème... (voir chapitre 7, *Fonctions, § tables Cache et remember*)

```
> forget(sin):
sin(x-2*y)^2 + cos(x-2*y)^2;
1
```

Eléments sur la structure des expressions : opérandes

Considérons l'expression E suivante

```
> restart:
E:=a*x+b*x^2+2*y-sin(a);
E := a x + b x2 + 2 y - sin(a)
```

Pour MAPLE cet objet (au sens informatique) est constitué avant tout par une somme de termes appelés *opérandes*, positives ou négatives, que l'on peut lister à l'aide de la fonction **op**

```
> op(E);
a x, b x2, 2 y, -sin(a)
```

E est bien la somme des éléments de cette suite d'objets. Le nombre d'opérandes est donné par la fonction **nops**

```
> nops(E);
4
```

La fonction **op** permet d'extraire une opérande particulière en donnant son numéro d'ordre en premier argument

```
> op(3,E);
2 y
```

Chaque opérande peut être un objet pouvant avoir lui-même plusieurs opérandes qui peut être aussi décomposé avec la fonction **op**. Ceci permet d'atteindre plus profondément la structure d'une expression. Ainsi la deuxième opérande de E est composée de deux opérandes liées par un opération de multiplication

```
> op2:=op(2,E);
op(op2);
op2 := b x2
b, x2
```

ou encore en imbriquant les fonctions (décomposition de x^2)

```
> op(op(2,op(2,E)));  
x, 2
```

On notera que la quatrième opérande de E , $-\sin(a)$, est en fait le produit de -1 par $\sin(a)$

```
> op(op(4,E));  
-1, sin(a)
```

Si on demande maintenant quelles sont les opérandes de $\sin(a)$ dans E , MAPLE ne renvoie que l'argument de la fonction sinus

```
> op_sin:=op(2,op(4,E)); # op(2,...) parce que op(1,...) est associé à -1  
op_sin; op_sin := sin(a)  
a
```

La fonction **sin** est associée à une opérande d'ordre 0 qu'il faut explicitement désigner si on veut la faire apparaître

```
> op(0,op_sin);  
sin
```

De façon générale, l'opérande 0 est la fonction qui s'applique sur les opérandes 1, 2, etc. de même niveau. L'expression E peut ainsi être considérée comme le résultat d'une fonction de sommation

```
> op(0,E);  
+
```

ainsi d'ailleurs que l'opérande 0 des quatre opérandes de E comme étant la fonction produit

```
> op(1,E),op(2,E),op(3,E),op(4,E);  
op(0,op(1,E)),op(0,op(2,E)),op(0,op(3,E)),op(0,op(4,E));  
a x, b x2, 2 y, -sin(a)  
*, *, *, *
```

On voit que les expressions sont construites à l'aide d'opérations imbriquées dont l'ordre est fixé par la priorité des opérations. On peut aussi, si l'on préfère, se faire une représentation en arbre avec différents niveaux, ceux-ci étant définis par la priorité des opérations. Choisissons un autre exemple

```
> E:=a*x+x^3-3*x^(3*a);  
E := a x + x3 - 3 x(3 a)
```

La représentation en arbre serait la suivante, les opérandes étant symbolisées par des crochets

Niveau 4	[3] * [a]		
Niveau 3	[x] ^ [3*a]		
Niveau 2	[a] * [x] [x] ^ [3] [-3] * [x^(3*a)]		
Niveau 1 (E):	[a*x] + [x^3] + [-3*x^(3*a)]		

Au niveau 1, les opérandes de cette expression sont liées par une fonction de sommation désignée par l'opérande 0 de E . De même

```
> op(0,op(2,E)); # Niveau 2 : [x]^3
^
```

Pour atteindre une opérande dans une expression ayant une grande hiérarchie de niveaux la syntaxe utilisant des fonctions **op** imbriquées peut être notablement simplifiée par l'utilisation d'une liste (voir cette notion au chapitre suivant). La liste est faite d'une suite d'entiers séparés par des virgules et entourée de crochets. Ces entiers désignent successivement le numéro d'ordre de l'opérande de premier niveau, puis de second niveau, etc.

```
> Exyzt:=x*(t-y-5)+4*x-(1-z)*y^(2*(1-t));
op(3,op(2,op(1,Exyzt)));
op([1,2,3],Exyzt); # opérande 3 de 1'opérande 2 de 1'opérande 1
```

$$\begin{aligned} \text{Exyzt} := & x(t - y - 5) + 4x - (1 - z)y^{(2 - 2t)} \\ & -5 \\ & -5 \end{aligned}$$

Ici encore, il est assez rare que l'on ait besoin d'effectuer ce type d'opérations. Pourtant il est *indispensable de bien connaître ces structures et les mécanismes qui leur sont associés pour maîtriser MAPLE*.

Exercice : pourquoi la liste doit-elle être [3,3,2,2] et non [3,2,2,2] pour faire apparaître ce terme ?

```
> op([3,3,2,2],Exyzt);
-2 t
```

Soustractions et divisions

On a vu que $A-B$ est compris comme $A+(-1*B)$. Pas plus que la soustraction, MAPLE ne connaît la division: ici $3/x$ est le **produit** de 3 par $1/x$:

```
> E:=3/x;
op(E);
op(0,E);
```

$$\begin{aligned} E := & \frac{3}{x} \\ & 3, \frac{1}{x} \\ & * \end{aligned}$$

et $1/x$ est en réalité compris par MAPLE comme $x^{(-1)}$

```
> op(op(2,E));
op(0,op(2,E));
x, -1
^
```

Après avoir étudié la notion de *types* nous reviendrons sur d'autres exemples sous la forme d'exercices

Indéterminées d'une expression

La fonction **indets** permet de savoir quelles sont les indéterminées d'une expression, c'est-à-dire les noms non assignés et les fonctions dépendant de tels noms (le sens des accolades dans l'écriture du résultat sera donné au chapitre 3)

```
> restart:  
z:=3;  
expr:=exp(z)+x-sin(z*x)+u^2+cos(Pi/7);  
indets(expr);
```

$$z := 3$$

$$\begin{aligned} \text{expr} &:= e^3 + x - \sin(3x) + u^2 + \cos\left(\frac{1}{7}\pi\right) \\ &\quad \{\sin(3x), x, u\} \end{aligned}$$

Avec l'argument supplémentaire **name** ou **symbol**, cette fonction ne renvoie que les noms non assignés qui, comme on peut le voir, ne sont pas nécessairement des variables. On constate en particulier que π (Pi) est une constante symbolique protégée à laquelle rien n'est assigné

```
> indets(expr,name);  
{\pi, x, u}
```

Retour sur les Evaluations et les Substitutions

Lorsque MAPLE effectue une évaluation avec la fonction **eval**, il substitue l'opérande désignée à gauche du signe égal par la valeur ou l'expression située à droite. La substitution s'effectue à tous les niveaux de la structure de l'expression.

```
> restart:  
E:=a*x+x^3-3*x^(3*a);  
eval(E,a=z+2);
```

$$\begin{aligned} E &:= ax + x^3 - 3x^{(3a)} \\ &= (z+2)x + x^3 - 3x^{(3z+6)} \end{aligned}$$

On remarquera en examinant l'arbre de l'expression E que 3 et -3 sont des opérandes distinctes, ce qui explique les résultats suivants

```
> Eval(E, 3=2): %value(%);  
Eval(E, -3=2): %value(%);
```

$$\begin{aligned} (ax + x^3 - 3x^{(3a)}) &\Bigg|_{3=2} = ax + x^2 - 3x^{(2a)} \\ (ax + x^3 - 3x^{(3a)}) &\Bigg|_{-3=2} = ax + x^3 + 2x^{(3a)} \end{aligned}$$

Exercice : examiner et expliquer les résultats:

```
> Ep:=a*x+1/x^3-3*x^(3*a);  
Eval(Ep, 3=2): %value(%);  
Eval(Ep, -3=2): %value(%);
```

$$Ep := ax + \frac{1}{x^3} - 3x^{(3a)}$$

$$\left(\begin{array}{l} ax + \frac{1}{x^3} - 3x^{(3a)} \\ \hline 3=2 \end{array} \right) = ax + \frac{1}{x^3} - 3x^{(2a)}$$

$$\left(\begin{array}{l} ax + \frac{1}{x^3} - 3x^{(3a)} \\ \hline -3=2 \end{array} \right) = ax + x^2 + 2x^{(3a)}$$

Pour que la fonction **eval** puisse effectuer les opérations il est **impératif** que l'expression située à gauche du signe = soit une opérande de l'expression, quel que soit le niveau de la structure. Ainsi $3a$ est une opérande de niveau 3

```
> E;
eval(E, 3*a=u-1);
eval(E, op([3,2,2],E)=t);
```

$$\begin{aligned} & ax + x^3 - 3x^{(3a)} \\ & ax + x^3 - 3x^{(u-1)} \\ & ax + x^3 - 3x^t \end{aligned}$$

Par contre $ax + x^3$ n'est opérande d'aucun niveau (c'est une combinaison de deux opérandes de niveau 1) et l'évaluation ne fonctionne pas

```
> eval(E, a*x+x^3=u);
a x + x^3 - 3 x^{(3a)}
```

De même on ne peut pas évaluer l'expression en posant $x^2=t$ qui remplacerait x^3 par xt . On pourra seulement poser $x^3=x*t$

```
> eval(E, x^2=t);
eval(E, x^3=x*t);
a x + x^3 - 3 x^{(3a)}
a x + x t - 3 x^{(3a)}
```

Il en est de même pour une évaluation automatique

```
> t:=x^2;
E;
t := x^2
a x + x^3 - 3 x^{(3a)}
```

Attention: même si un nombre rationnel possède deux opérandes, *elles ne sont pas substituables*. On parle ici de rationnels numériques et non de rationnels symboliques (qui ne peuvent pas encore être définis dans ce chapitre).

```
> q:=3/2;
op(q);
eval(q, 3=5);
q :=  $\frac{3}{2}$ 
```

3, 2

$\frac{3}{2}$

Aussi, la tentative d'évaluation suivante ne simplifie pas l'expression car $2*x/3$ est transformée par l'analyseur syntaxique de MAPLE en $(2/3)*x$, $(2/3)$ étant un nombre rationnel dont les opérandes ne sont pas substituables

```
> q:=2*x/3;  
op(q);  
eval(q,3=2);
```

$$q := \frac{2}{3} x$$

$$\frac{2}{3}, x$$

$$\frac{2}{3} x$$

Fonction subs

MAPLE dispose également d'une fonction **subs**, assez similaire à **eval**, mais qui n'effectue que les substitutions d'opérandes sans nécessairement faire les évaluations. Les commentaires donnent un moyen de se souvenir de l'ordre des arguments qui est différent pour les deux fonctions

```
> Es:=x+x^2+2*y-sin(Pi*a);  
E1:=subs(a=2,Es);# Substituer a par 2 dans Es  
E2:=eval(Es,a=2);# Evaluer Es pour a valant 2  
Es := x + x2 + 2 y - sin(π a)  
E1 := x + x2 + 2 y - sin(2 π)  
E2 := x + x2 + 2 y
```

On notera que $E1$ contient un terme $\sin(2 \pi)$ que **eval** a calculé pour $E2$. La fonction **eval** avec pour seul argument une expression permet de forcer (si possible) ces évaluations résiduelles

```
> eval(E1);  
x + x2 + 2 y
```

Comme **eval**, la fonction **subs** permet d'effectuer plusieurs substitutions à l'aide de la même instruction

```
> subs({a=1,x^2=y^3,sin=cos},Es);  
x + y3 + 2 y - cos(π)
```

Attention: Jusqu'ici la fonction **subs** n'avait que deux arguments, les termes entre accolades constituant un seul argument. La fonction **subs** admet une syntaxe avec plus de deux arguments (suppression des accolades; la fonction **eval** n'admet pas cette syntaxe). Les substitutions se font alors dans l'ordre d'apparition (de gauche à droite) des arguments de **subs** et chaque substitution prend en compte la précédente. Dans le premier exemple l'opérande a prend la valeur 2 puis toutes les opérandes 2 de la nouvelle expression prennent la valeur 3. Pour cette syntaxe l'ordre des arguments a donc une importance comme le montre le deuxième exemple. *Sauf raisons particulières on préférera la syntaxe avec accolades.*

```
> Es;  
subs(a=2,2=3,Es);
```

```

subs(2=3,a=2,Es);

$$x + x^2 + 2y - \sin(\pi a)$$


$$x + x^3 + 3y - \sin(3\pi)$$


$$x + x^3 + 3y - \sin(2\pi)$$


```

Exercices : interprétez les résultats suivants en se rappelant que MAPLE évalue les arguments d'une fonction avant de l'évaluer (**eval** et **subs** sont des fonctions).

```

> a:='a': Ex:=4*x+a^2;
a:=3:
eval(Ex,a=4);
eval(Ex,4=a);

```

$$\begin{aligned}Ex &:= 4x + a^2 \\&4x + 9 \\&3x + 9\end{aligned}$$

```

> a:='a':
eval(Ex,a=4);
eval(Ex,4=a);

$$\begin{aligned}&4x + 16 \\&ax + a^2\end{aligned}$$


```

Fonctions **subsop** et **algsubs**

On peut effectuer une substitution à l'aide de la fonction **subsop**, le premier argument s'écrivant $n = \dots$ où n désigne le numéro d'ordre d'une opérande de premier niveau

```

> Es;
subsop(2=4,Es);# Remplace la 2ème opérande par 4

$$\begin{aligned}&x + x^2 + 2y - \sin(\pi a) \\&x + 4 + 2y - \sin(\pi a)\end{aligned}$$


```

que l'on comparera à

```

> subs(2=4,Es);

$$x + x^4 + 4y - \sin(\pi a)$$


```

Les fonction **subs** ou **eval** ne permettent pas d'agir sur un groupe d'opérandes associées par une opération algébrique comme $x + x^2$. Seules les opérations conformes seront exécutées (compléter ces information en consultant l'aide en ligne concernant la levé des ambiguïtés)

```

> eval(Es+exp(-x^2-x^3),{x+x^2=t,a=1,y=0});

$$x + x^2 + e^{(-x^2 - x^3)}$$


```

On pourra, pour réaliser de telles opérations, utiliser la fonction **algsubs** (**algebraic substitution**). On remarque que **algsubs** ne se contente pas d'une simple substitution d'opérandes car l'argument $-x^2 - x^3$ de l'exponentielle a été factorisé en $-x(x + x^2)$ avant d'être substitué. Toutefois **algsubs** n'admet pas d'effectuer plusieurs substitutions simultannées. On pourra pour cela imbriquer les fonctions pour réaliser toutes les

opérations souhaitées en une seule commande.

```
> eval(algsubs(x+x^2=u,Es+exp(-x^2-x^3)),{a=1,y=0});  
u + e(-u x)
```

Fonctions `whattype` et `type`, profils de types

Fonction `whattype`

Tous les objets de MAPLE possèdent au moins une caractéristique appelée aussi *type*. En fait, ils en possèdent souvent plusieurs, le type de base et les sous-types. Le type de base d'un objet peut être obtenu avec la fonction `whattype`. Ici MAPLE répond + indiquant que *Es* est du type "somme" (somme des opérandes de niveau 1). Cette fonction renvoie *souvent* l'équivalent de l'opérande d'ordre 0 de niveau 1

```
> Es;  
op(0,Es);  
whattype(Es);
```

$$\begin{aligned} &x + x^2 + 2 y - \sin(\pi a) \\ &+ \\ &+ \end{aligned}$$

Cependant ce n'est pas systématique car le type et la fonction ne se recouvrent pas nécessairement...

```
> op(0,cos(x));  
whattype(cos(x));
```

cos
function

```
> s:=series(sin(x),x):# voir chapitre 11  
op(0,s);  
whattype(s);
```

x
series

Voici d'autres exemples dont les réponses sont simples à comprendre.

```
> op(0,-3);  
whattype(-3);  
op(0,3/2);  
whattype(3/2);
```

Integer
integer
Fraction
fraction

Ici le mot *Float* désigne la fonction qui, à partir des arguments entiers *n* et *m*, construit le décimal $n \cdot 10^m$ (voir chapitre 1, § *Nombres décimaux*)

```
> op(0,3.5e-3);  
op(3.5e-3);  
Float(%);
```

```

          Float
          35, -4
          0.0035

```

```

> whattype(3.5e-3);
whattype(exp(-3.5e-3));

          float
          float

```

Les nombres complexes dont les parties réelles et imaginaires sont définies avec des valeurs numériques non symboliques ont leur type propre

```

> z:=1+3*I; whattype(z);
I;           whattype(I);

          z := 1 + 3 I
          complex(extended_numeric)
          I
          complex(extended_numeric)

```

Mais ceci n'est pas vrai pour des nombres complexes définis à l'aide de nombres symboliques. Le type est alors celui de l'opération qui les définit

```

> Z:=sqrt(2)+3*I;    whattype(Z);
T:=Pi/I;            whattype(T);
G:=cos(2+3*I);     whattype(G);

          Z :=  $\sqrt{2} + 3 I$ 
          +
          T := -I \pi
          *
          G := cos(2 + 3 I)
          function

```

Attention: la fonction **whattype** ne donne *pas* la nature mathématique d'un nombre mais caractérise la nature informatique de l'objet, définie de façon interne par MAPLE (voir le paragraphe suivant sur la fonction **type**).

Les réponses peuvent donc révéler quelques surprises pour qui manque de pratique (nous aurons l'occasion d'y revenir tout au long du cours). Le lecteur attentif peut déjà comprendre les deux derniers exemples.

```

> whattype(cos);      # voir chapitre 7
whattype(eval(cos)); # idem
whattype(cos(x));
whattype(cos(Pi/4));

          symbol
          procedure
          function
          *

```

Symboles et Chaînes de caractères : introduisons, pour les besoins des exemples donnés dans les paragraphes suivants, les notions de symbole et de chaîne de caractères (elles seront étudiées plus en détail au chapitre 18, *Eléments du langage de Programmation*).

- Des caractères entourés par deux ` définit un objet de type **symbol**.
- Des caractères entourés par deux caractères " définit un objet de type chaîne de caractères (character **string**)

Attention, " est un seul caractère et non la répétition de '

```
> whattype(`Valeur de x`);  
symbol
```

```
> whattype("Chaine de caracteres");  
string
```

Un objet de type **symbol** peut être utilisé comme un nom valide à condition toutefois que les caractères soient entourés des deux caractères `. Une chaîne de caractères n'a pas cette propriété et ne peut pas être utilisée comme symbole.

```
> `Valeur de x`:=1;  
exp(`Valeur de x`+1);  
Valeur de x := 1  
e^2
```

Si cette possibilité permet une grande liberté d'écriture, elle doit être manipulée avec prudence car il est facile de rendre confuse une feuille de calcul comme le montre cet exemple caricatural

```
> `1+1`:=4: ' `1+1` -2 ` = `1+1` -2;  
I+I - 2 = 2
```

En fait, tout nom valide créé avec la syntaxe habituelle (commençant par une lettre ou un " _ ", ne contenant pas d'espace et constitué de lettres et de chiffres) est considéré par MAPLE comme un symbole même en l'absence des ` . Dans l'exemple suivant la réponse signifie que x est un nom qui ne pointe sur rien (sinon sur lui-même). Le type de x est alors celui d'un symbole.

```
> whattype(x);  
symbol
```

Exercice : interprétez ces résultats

```
> Z:=x^2+1;  
whattype('Z');  
whattype( Z );  
Z := x^2 + 1  
symbol  
+
```

Fonction type

On peut vérifier qu'un objet est bien du type souhaitée avec la fonction **type** (l'intérêt des fonctions **type** et **whattype** apparaît clairement en programmation). Si on demande par exemple avec **type** si la structure de *Es* est du type "somme" en désignant l'opérateur par `+`, MAPLE répond *true* c'est-à-dire "vrai".

```
> Es;
```

```
type(Es, `+`);
```

$$x + x^2 + 2y - \sin(\pi a)$$

true

On remarque que `+` dans la commande précédente est entouré de deux accents graves. Ici ``+`` représente le *symbole* qui est le nom MAPLE de la fonction "sommation". Ecrire simplement `+` n'aurait aucun sens puisque les opérandes ne sont pas présentes et conduirait à une erreur de syntaxe.

L'identificateur `x` ne pointe sur aucun objet et il possède seulement le type *symbol* et le type *name*

```
> x;  
type(x, symbol);  
type(x, name);
```

x

true

true

On pourra sauter les remarques suivantes (entre les deux traits) pour une première lecture.

Remarques : Les deux types *symbol* et *name* ne sont cependant pas complètement identiques lorsqu'ils s'appliquent à un *nom indexé* non assigné

```
> X[0];  
whattype(X[0]);  
type(X[0], name);  
type(X[0], symbol);
```

X₀

indexed

true

false

On rencontre ici certaines subtilités syntaxiques de MAPLE sur lesquelles nous ne nous attarderons pas. Si, non assigné, $X[0]$ était un *nom indexé*, assigné il devient l'élément d'une table (voir le chapitre 5, *Tables et Tableaux*)

```
> X[0]:=1;  
whattype(eval(X));
```

X₀ := 1

table

Avec les noms prédéfinis de MAPLE comme D , I , γ , \sin , etc., on peut créer des variables indexées mais elles ne pourront pas être assignées car la transformation en table est interdite, ces noms étant propégés

```
> I[0];
```

I
0

```
> I[0]:=1;
```

Error, illegal use of an object as a name

Tous les noms prédéfinis (ici une exponentielle intégrale) n'étant pas nécessairement connus de l'utilisateur, le message peut surprendre

```
> Ei[0];
Ei[0]:=1;
Ei0
Error, attempting to assign to array/table `Ei` which is protected
```

Attention: La fonction **type** (comme **whattype**) analyse la propriété informatique (ou une combinaison de propriétés), relativement aux définitions internes de MAPLE, des objets spécifiés. Elle n'analyse pas la nature mathématique et c'est la fonction **is** qui réalise cette opération (fonction sur laquelle nous reviendrons au chapitre 13, *Simplifications, Manipulations*). Il peut y avoir quelquefois recouvrement entre les réponses. Par exemple, les entiers positifs et négatifs ont des types internes propres qui coïncident avec leurs propriétés mathématiques

```
> type(-4,integer), type(4,positive),
  type(4,posint), # posint = entier positif
  type(4,negint); # negint = entier négatif
true, true, true, false
```

La fonction **is** analyse la nature mathématique et les réponses sont les mêmes

```
> is(-4,integer), is(4,positive), is(+4,posint), is(+4,negint);
true, true, true, false
```

Mais un nombre comme $\sqrt{3}$, bien qu'il soit positif, ne possède pas le type *positive* et les réponses sont différentes.

```
> type(sqrt(3),positive), is(sqrt(3),positive);
false, true
```

De même

```
> type(Pi,positive), is(Pi,positive);
false, true
```

Le nombre de types prédéfinis par MAPLE est de l'ordre de 200. Nous aurons l'occasion d'en rencontrer d'autres dans ce cours. Par exemple le type **algebraic** est associé à un nombre, un symbole ou une expression

```
> type(-3,algebraic), type(x,algebraic),
  type(exp(-x^2)+sqrt(y),algebraic);
true, true, true
```

mais une chaîne de caractères ne possède pas ce type

```
> type("abc",algebraic);
false
```

On aura pu remarquer qu'un objet peut avoir plusieurs types. Ainsi -3 possède le type *integer*, *negint*, *algebraic* ou *constant*. Voici un autre exemple pour $\sqrt{3}$

```
> type(sqrt(3),`^`), type(sqrt(3),constant),
  type(sqrt(3),algebraic);
```

true, true, true

Signalons aussi la distinction faite par MAPLE entre un nombre du type *numeric* et un nombre du type *constant*. Pour MAPLE, le nombre -3 est une *constante numérique* alors que π n'est qu'une *constante*.

```
> type(-3,constant), type(-3,numeric);  
type(Pi,constant), type(Pi,numeric);  
true, true  
true, false
```

Il en est de même pour $\sqrt{3}$

```
> type(sqrt(3),constant), type(sqrt(3),numeric);  
true, false
```

Contrairement à un type comme `+` qui ne relève que de la structure informatique d'une expression, le type *constant* ne peut être déterminé que par une analyse complète. Ainsi les deux exemples suivants montrent que MAPLE "sait" que *gamma* est une constante (constante d'Euler) alors qu'il ne sait rien de *kappa* qu'il considère comme un simple nom.

```
> sqrt(2)+exp(1-kappa)+cos(Pi/7);  
type(% ,constant);  
sqrt(2)+exp(1-gamma)+cos(Pi/7);  
type(% ,constant);  

$$\sqrt{2} + e^{(1 - \kappa)} + \cos\left(\frac{1}{7}\pi\right)$$
  
false  

$$\sqrt{2} + e^{(1 - \gamma)} + \cos\left(\frac{1}{7}\pi\right)$$
  
true
```

MAPLE reconnaît par défaut 7 constantes symboliques qui sont contenues dans la suite (*sequence*, voir chapitre 3) **constants**

```
> constants;  
false,  $\gamma$ ,  $\infty$ , true, Catalan, FAIL,  $\pi$ 
```

Les types sont hiérarchisés mais ne sont pas exclusifs. Si -2 est un entier, c'est aussi un polynôme (certes particulier)

```
> type(-2,integer), type(-2,polynom);  
true, true
```

Alors que

```
> type(x^2+1,integer), type(x^2+1,polynom);  
false, true
```

Enfin, les types ont le type... *type*

```
> type(integer,type);  
true
```

Profils de type

En programmation (chapitre 19) il est souvent utile de savoir si une objet correspond à un profil défini. On compose un profil en reproduisant l'expression souhaitée avec les types des opérandes

```
> type(x^n,name^name), type(x^n,name^constant),
  type(exp(-x+1),function(algebraic));
          true, false, true
```

On complètera l'information de ce paragraphe avec l'aide en ligne

```
> ?type,structure
```

L'expression suivante n'a pas le type *polynom* en raison de la puissance fractionnaire

```
> type(x^(2/3)+x+2,polynom);
                           false
```

Avant de poursuivre il faut faire une parenthèse. L'opération de sommation peut s'écrire comme une fonction qui a pour nom le symbole `~+` (et non simplement `+` qui produirait une écriture syntaxiquement incorrecte)

```
> `~+(4,1,-2);
               3
```

```
> `~+(1,x,exp(-x));
               x + e^(-x) + 1
```

Il est possible de "neutraliser" l'opérateur en le faisant précédé du caractère `&`. L'opérateur n'agit plus sur les opérandes (il perd aussi sa commutativité et sa distributivité).

```
> `&+`(4,1,-2); 4 &+ (x - 1);
               &+(4, 1, -2)
               4 &+ (x - 1)
```

Reprendons notre exemple. L'expression $x^{\frac{2}{3}} + x + 2$ ne correspond pas au type *polynom*, mais correspond maintenant au profil proposé. La neutralisation de l'opérateur est nécessaire car on ne veut pas effectuer l'opération mais seulement proposer un profil

```
> type(x^(2/3)+x+2,`&+`(name^rational,name,constant));
               true
```

Les entiers possèdent aussi le type *rational* et MAPLE accepte pour $x^{(-2)} + y$ le profil de type suivant

```
> type(-2,rational),
  type(x^(-2)+y,`&+`(name^rational,name));
               true, true
```

Ici $x^2 + 1$ n'est pas un nom et la fonction **type** répond que l'expression n'a pas le bon profil

```
> type((x^2+1)^(1/3),name^rational);
                           false
```

mais

```
> type((x^2+1)^(1/3),algebraic^rational);
```

true

On donne encore un exemple de profil de type pour lequel les expressions doivent être la somme d'une fonction à un argument et d'une constante. De plus l'argument de la fonction doit être un nom de variable à une puissance qui doit être *soit* un entier, *soit* un nom. L'alternative "soit" est définie par les accolades {} (voir le chapitre 3, *Ensembles*).

```
> Q:=cos(x^n)+1;
  type(Q, `&+` (function(name^{integer,name}),constant));
```

$$Q := \cos(x^n) + 1$$

true

```
> Q:=tan(x^(-2))-exp(1-I*Pi);
  type(Q, `&+` (function(name^{integer,name}),constant));
```

$$Q := \tan\left(\frac{1}{x^2}\right) - e^{(1 - I\pi)}$$

true

```
> Q:=ln(x^sqrt(2))-1;
  type(Q, `&+` (function(name^{integer,name}),constant));
```

$$Q := \ln(x^{\sqrt{2}}) - 1$$

false

Exercice : ce profil ne reconnaît pas certaines situations particulières. Pourquoi ? Modifiez le profil pour obtenir la réponse *true* pour les situations suivantes :

```
> c:=0: type(cos(x^2)+c, `&+` (function(name^{integer,name}),constant));
  k:=1: type(cos(x^k)+2, `&+` (function(name^{integer,name}),constant));
  k:=0: type(cos(x^k)+2, `&+` (function(name^{integer,name}),constant));
```

false

false

false

Exercice : on a déjà vu que l'expression suivante répondait correctement au profil proposé

```
> type(x^(2/3)+x+2, `&+` (name^rational,name,constant));
  true
```

mais, pourquoi cette réponse ? (analysez les opérandes de premier niveau). Modifiez le profil pour obtenir *true*

```
> type(x^(2/3)-x+2, `&+` (name^rational,name,constant));
  false
```

On pourra également utiliser les fonctions **match**, **patmatch** et **typematch** (voir l'aide en ligne).

Noms des types et profils de types créés

On peut donner des noms aux types ou profils de types que l'on construit en utilisant la syntaxe suivante. On veut par exemple créer un type ayant pour nom **monome** et qui reconnaît les objets de la forme "monôme sans coefficient multiplicateur". Il suffira d'écrire (**posint**: type correspondant à un entier strictement positif)

```
> restart;
```

```
`type/monome` := name^posint;
type/monome := nameposint
```

On peut utiliser une assignation directe à un nom (**monome:=name^{posint}**), mais ceci engendre des difficultés.

Rappel: On pourra générer ces types dans le fichier d'initialisation de MAPLE si on veut les garder permanents (voir chapitre 1, *Commentaires sur l'utilisation de MAPLE*, § IV-5).

```
> type(x^3, monome);
true
```

Avec la syntaxe `type/monome` le nom **monome** est bien un type défini, mais ce n'est pas un nom assigné et il reste libre d'utilisation.

```
> type(monome, type);
monome;
assigned(monome);
true
monome
false
```

En cas d'assignation à ce nom, ce qui est possible mais assez maladroit du point de vue de la clarté, il faudra transmettre le nom à la fonction **type** en prévenant l'évaluation.

```
> monome:=3;
type(x^3, monome), type(y^3, 'monome');
monome := 3
false, true
```

Il est conseillé, pour lever toute ambiguïté, de protéger les noms homonymes non assignés des types construits.

```
> monome:='monome':
protect(monome):
```

Les noms associés aux types prédéfinis de MAPLE sont eux aussi protégés.

```
> integer:=3;
Error, attempting to assign to `integer` which is protected
```

Le type **monome** ne reconnaît pas les monômes de la forme $x^0 = 1$ ou x .

```
> type(1, monome), type(z, monome);
false, false
```

Il suffit de redéfinir et de compléter (**noter** que **protect** protège le nom, pas le type)

```
> `type/monome` := {1, name, name^posint};
type/monome := {1, name, nameposint}
```

En effet, ça fonctionne...

```
> type(1, monome), type(t, monome), type(s^5, monome);
true, true, true
```

On voudrait maintenant étendre ce type aux objets de la forme x^r où r est un rationnel

```
> type(x^(-1/3), monome);  
false
```

On peut étendre un type construit par "héritage" de types déjà construits

```
> `type/monome_etendu` := {monome, name^rational};  
type/monome_etendu := {namerational, monome}  
  
> type(x^2, monome_etendu), type(x^(-1/3), monome_etendu);  
true, true
```

Pour des situations plus complexes on peut associer une procédure à un type (pour les informations sur les procédures, se reporter au chapitre 19). Elle devra avoir pour seule réponse *true* ou *false*. On montre un exemple pour un type qui est une restriction du précédent aux exposants rationnels positifs

```
> `type/monome_etendu_pos` := proc(x)  
  if type(x, {1, name, name^posint}) then  
    return true;  
  elif type(x, name^rational) and is(op(2, x)>0) then  
    return true;  
  end if;  
  false;  
end proc;  
  
> type(x^2+1, monome_etendu_pos);  
false  
  
> type(1, monome_etendu_pos), type(z^2, monome_etendu_pos);  
type(x^(-1/3), monome_etendu_pos), type(y^(1/3), monome_etendu_pos);  
true, true  
false, true
```

Fonction subtype

Comme son nom l'indique, elle permet de déterminer si un type est un sous type d'un autre dans la structuration des types:

> subtype(a, b) => *true* si *a* est un sous type de *b*. Un entier (*integer*) n'est pas un sous-type d'un entier positif (*posint*), mais l'inverse est vrai.

```
> subtype(monome, monome_etendu);  
subtype(integer, posint), subtype(posint, integer);  
subtype(And(posint, negint), rational);  
true  
false, true  
true
```

Bibliothèque TypeTools

Il existe aussi une autre possibilité d'attribuer des types avec la bibliothèque **TypeTools**. Elle contient cinq fonctions dont **GetTypes** (avec un s) qui liste les types créés avec cette bibliothèque. On peut constater que les types précédents ne sont pas reconnus par cette fonction.

```
> TypeTools[GetTypes]();
```

[]

Réinitialisons pour lever toute ambiguïté en détruisant les types précédents. On crée de nouveaux types avec la fonction **AddType**. On notera les appels qui s'écrivent indifféremment **TypeTools[fonction](...)** ou **TypeTools:-fonction(...)**. On pourrait aussi utiliser **with(TypeTools)** (voir chapitre 1, seconde partie: *Commentaires sur l'utilisation de MAPLE*, § IV-7).

```
> restart;
TypeTools[AddType](monome, 'name^posint');
TypeTools:-AddType(monome_etendu, {monome, name^rational});
```

Comme précédemment on peut associer au type une procédure de détermination. *Attention:* on notera qu'ici **end proc** ne se termine pas par ; ou :

```
> TypeTools:-AddType(monome_etendu_pos,
proc(x)
  if type(x,{1,name,name^posint}) then
    return true;
  elif type(x,name^rational) and is(op(2,x)>0) then
    return true;
  end if;
  false;
end proc);
```

La fonction **Type** de cette bibliothèque teste un type. *Elle est identique à la fonction type*

```
> TypeTools:-Type(s^5,monome);
type(s^5,monome);
type(x^(-3/2),monome_etendu_pos);
                           true
                           true
                           false

> TypeTools:-GetTypes();
[monome_etendu, monome_etendu_pos, monome]
```

La fonction **GetType** (sans s) récupère la définition d'un type.

```
> TypeTools:-GetType(monome_etendu);
                           {monome, name^rational}
```

La fonction **RemoveType** permet d'éliminer un type

```
> TypeTools:-RemoveType(monome);
type(x^3,monome);
Error, type `monome` does not exist
```

Le type dérivé *monome_etendu* construit à l'aide du type *monome* existe toujours mais n'est plus utilisable

```
> TypeTools:-GetTypes();
type(x^(1/3),monome_etendu);
                           [monome_etendu, monome_etendu_pos]
Error, type `monome` does not exist
```

Par contre le type *monome_etendu_pos* n'étant pas construit par héritage de la définition de *monome*, il existe toujours

```
> type(x^(1/3),monome_etendu_pos);  
true
```

Types attribués, fonctions *setattribute* et *attributes*

Il existe encore une autre possibilité d'attribuer un ou plusieurs types choisis par l'utilisateur aux objets de MAPLE avec la fonction **setattribute**. Voici quelques exemples où "Variable", "Constante", etc. sont des noms imaginés par l'utilisateur. On rappelle que si un nom est assigné (*c*), il ne faudra pas oublier de le rendre non évalué pour le transmettre aux commandes qui le nécessitent comme **protect** ou **setattribute**

```
> restart;  
setattribute(x,Variable):  
c:=2.99792:  
protect('c');  
setattribute('c',Constante,Physique,Protegee):
```

[*Attention* avec la fonction **protect**. L'exemple ci-dessus ne fait pas ce que l'on croit !!! (voir plus loin)

On peut interroger MAPLE avec la fonction **attributes** pour savoir quels sont les types attribués à un nom

```
> attributes(x);  
attributes('c');  
Variable  
Constante, Physique, Protegee
```

Les termes comme "Constante", "Variable" définis par l'utilisateur n'ont pas la caractéristique d'un type propre de MAPLE, mais celle d'un "type attribué". Un type attribué n'est pas non plus un nom et on peut se servir de la même orthographe pour un nom assignable, même si, pour des raisons de clarté, ceci est peu recommandé.

```
> type('c',Constante);  
Error, type `Constante` does not exist
```

Pour vérifier si un nom possède un ou plusieurs types attribués donnés on écrira

```
> type('c',attributed(Constante));  
type('c',attributed(Constante,Protegee));  
type('c',attributed(Variable));  
true  
true  
false
```

Certaines objets de MAPLE ont aussi des types attribués

```
> attributes(cos);  
type(cos,protected);  
type(cos,attributed(protected));  
protected, _syslib  
true  
true
```

Même si la fonction **type** répond correctement au type *protected*, il s'agit en réalité d'un type attribué comme le montre l'exemple précédent. Or, quand on attribue un ou des types à un nom, celui-ci perd ceux qu'il avait précédemment. Si on examine la façon dont on a défini la constante *c*, on constatera qu'elle a en fait perdu sa protection par l'application de **setattr** après **protect** et l'attribut "Protegee" donné par l'utilisateur est illusoire !!!.

```
> type('c',protected);
  attributes('c');
                                false
  Constante, Physique, Protegee
```

La commande **protect('a')** fonctionne comme un alias de

```
> setattr('a',protected,attributes('a'));
```

Il faut donc, soit rajouter **protected** dans la liste des attributs, soit appliquer la fonction **protect** *après* **setattr**.

```
> setattr('c',Constante,Physique, Protegee):
  protect('c');

> type('c',protected);
  attributes('c');
                                true
  protected, Constante, Physique, Protegee
```

Hélas, ces remarques sont aussi vraies pour les objets prédéfinis de MAPLE et on doit faire preuve d'une grande prudence. On donne un exemple, certes caricatural car il ne viendrait à l'idée de personne de changer la nature de la fonction **cos** ! Mais combien d'utilisateurs peuvent prétendre connaître l'orthographe des noms des 87 940 librairies et fonctions de MAPLE version 9 ?

```
> setattr(cos,Nouvelles_Mathematiques):# cos n'est plus protégée
  cos:=x->sin(x):# On redéfinit la fonction cos (voir chapitre 7) et ...
  'cos(x)^2+sin(x)^2'=simplify(cos(x)^2+sin(x)^2);# c'est la catastrophe !
  cos(x)^2 + sin(x)^2 = 2 sin(x)^2
```

2+1 = 3, êtes-vous sûr ? ;-)

```
> setattr(`+`):
  `+`:=(x,y)->x*y:
  `+(2,1)`=`+(2,1);
  +(2,1)=2
```

Il n'y a guère qu'une réinitialisation radicale qui permette de remettre de l'ordre...

```
> restart:
  'cos(x)^2+sin(x)^2'=simplify(cos(x)^2+sin(x)^2); # Ouf !
  `+(2,1)`=`+(2,1);
  cos(x)^2 + sin(x)^2 = 1
  +(2,1)=3
```

Vérifiez le statut d'un nom avant de lui appliquer la fonction `setattribute` !

Il ne devrait pas être difficile aux concepteurs de MAPLE de modifier ce comportement pour le rendre plus sécurisé...

On notera également que la désassignation conserve les types attribués

```
> setattribute(a, Constante): a:=3; attributes('a');
a:='a'; attributes(a);
a := 3
Constante
a := a
Constante
```

Exercice : Pourquoi ces réponses ?

```
> unprotect('c'):
setattribute(c, Constante):
c:=2.99792:
type(c, attributed(Constante));
Attributs_c:=attributes(c);
false
Attributs_c :=
```

Exercices de rappel : expliquez les résultats suivants.

```
> restart: Es:=x+x^2+2*y-sin(Pi*a); eval(Es,op(nops(Es),Es)=1);
Es := x + x2 + 2 y - sin(π a)
x + x2 + 2 y + 1

> ex:=x/2+1/sin(x); type(op(1,ex),`*`); whattype(op(2,ex));
true
^

> ex2:=x/2+2/sin(x); type(op(2,ex2),`*`); whattype(op(2,ex2));
true
*
```

Que se passe-t-il ?

```
> ex:=5*x-2; Eval(ex,2=3):%value(%);
ex := 5 x - 2
(5 x - 2) | = 5 x - 2
           2 = 3
```

3 - Intervalles, Suites, Listes, Ensembles

Le débutant est souvent déconcerté par les réponses de MAPLE qui écrit a, b, c ou $[a, b, c]$ ou encore $\{a, b, c\}$. Il peut penser qu'il s'agit d'une simple question de présentation, mais il n'en est rien et ces écritures ont un sens bien particulier qu'il est indispensable de connaître.

Ce chapitre et les deux suivants, sont principalement destinés à introduire les structures de données de MAPLE, c'est-à-dire des objets de nature informatique contenant des collections de nombres ou d'autres objets manipulables par MAPLE. La plupart du temps, ils auront une forme qui rappelle celles de concepts mathématiques tels que des ensembles, des suites, des matrices, etc. Mais ils pourront aussi se présenter sous une forme sans équivalent traditionnel, comme par exemple des listes, des tables ou des tableaux.

Intervalles

MAPLE permet de définir des intervalles sous la forme de deux termes séparés par *au moins* deux points.

```
> restart:  
a..b;  
-2/3.....exp(z)+1;  
0..infinity;  
-1.5..3.6;  
a .. b  
-2 .. ez + 1  
0 .. infinity  
-1.5 .. 3.6
```

Tout objet valide de MAPLE peut avoir un nom y compris un intervalle (ne pas utiliser les noms **int**, **Int** ou **I** qui sont des noms protégés de MAPLE qui renverrait un message d'erreur).

```
> Interv:= binomial(4,1)..5!;  
Interv := 4 .. 120
```

L'interprétation d'un intervalle dépend, comme nous le verrons, du contexte de son utilisation. Par exemple l'intervalle $0 .. \infty$ représente-t-il l'ensemble des entiers ou des réels positifs ou nuls ? La réponse dépend du calcul que l'on effectue, par exemple une intégration ou une sommation discrète. Les intervalles ont le type *range*

```
> whattype(Interv);  
type(Interv,range);  
type(Interv,`..`);  
..  
true  
true
```

Un intervalle possède deux opérandes qui sont simplement ses bornes.

```
> op(Interv);
```

```
op(2,Interv);
```

4, 120

120

Pour les changements de valeurs, voir les fonctions **subs** et **subsops** au chapitre 2, *Affectations...*

```
> Interv:=subs(4=-3,Interv);  
Interv:=subsop(2=infinity,Interv);  
Interv := -3 .. 120  
Interv := -3 ..  $\infty$ 
```

Il existe une autre façon de décrire un intervalle. La façon précédente est plutôt destinée aux calculs (voir par exemple le paragraphe suivant et la fonction **seq**, les chapitres 10, *Intégration* ou 12, *Sommes et produits*) et, comme nous l'avons déjà dit, ne précise pas la nature des nombres contenus dans l'intervalle. La façon suivante est plutôt liée à une notion d'appartenance et décrit la nature précise des nombres. On définit ici l'ensemble des entiers strictement plus grand que 2: $]2, \infty]$. La fonction **AndProp** associe deux propriétés (voir chapitre 13). On notera la transformation opérée: comme il s'agit de réels entiers, $2 < n$ entraîne $3 \leq n$

```
> N[>2]:=AndProp(RealRange(Open(2),infinity),integer);  
N[>2] := AndProp(integer, RealRange(3,  $\infty$ ))
```

et on vérifie par exemple que $n = 3$ appartient à cet intervalle, mais pas π

```
> n:=3:  
is(n in N[>2]), is(Pi in N[>2]);  
true, false
```

On peut aussi écrire (**posint** est un type: positive integer)

```
> is(n in posint and n > 2);  
true
```

Elle sert aussi à émettre des hypothèses de calcul (voir chapitre 13) ou définir des solutions d'inéquations (voir chapitre 14)

```
> solve(x^3-3*x^2+2 > 0,x);  
RealRange(Open(1 -  $\sqrt{3}$ ), Open(1)), RealRange(Open(1 +  $\sqrt{3}$ ),  $\infty$ )
```

Attention: ces objets ont le type **function**, mais pas le type **range**

```
> whattype(N[>2]), type(N[>2],range);  
function, false
```

Suites (sequence)

Une suite ou **sequence** est une suite d'éléments de MAPLE, même disparates, séparés par des virgules. Nous en avons déjà rencontré par exemple sous la forme de résultats de la fonction **op**. Leur utilisation est, comme nous le verrons dans ce chapitre, fréquente pour la construction automatique d'objets. On peut les assigner à des noms et l'expression suivante définit une suite *s*

```
> restart;  
s:=a,b,1,1,3,a*x+1,b,exp(x)-y;  
s := a, b, 1, 1, 3,  $a x + 1$ , b,  $e^x - y$ 
```

On peut multiplier une suite par une constante

```
> -2/3*s;

$$-\frac{2}{3} a, -\frac{2}{3} b, -\frac{2}{3}, -2, -\frac{2}{3} a x - \frac{2}{3}, -\frac{2}{3} b, -\frac{2}{3} e^x + \frac{2}{3} y$$

```

On pourra aisément inclure une suite dans une autre

```
> Y:=A,B,C;
1,Y,2,3;
Y := A, B, C
1, A, B, C, 2, 3
```

Quelquefois, pour des raisons de syntaxe, on pourra être amené à définir une suite en entourant les éléments par deux parenthèses. On trouvera un exemple au chapitre 7, *Fonctions*, § *Fonctions de plusieurs variables*.

Les suites, comme le montre **whattype**, ont un type connu qui est *exprseq*. Cependant le type ne peut pas être testé par la fonction **type**.

```
> whattype(s);
type(s,exprseq);
exprseq
Error, invalid input: type expects 2 arguments, but received 9
```

Le message d'erreur nous dit que la fonction **type** attend 2 arguments, mais en a reçu 9. En effet, MAPLE évaluant *s* avant d'exécuter la fonction **type**, ce que nous avons écrit est équivalent à l'instruction

```
> type(a,b,1,1,3,a*x+1,b,exp(x)-y,exprseq);
```

La fonction **type** a donc reçu un nombre incorrect d'arguments. Voici un autre exemple qui montre comment *S* est évalué pour servir d'arguments à la fonction **max**

```
> S:=-4,3,-1,0,2;
max(-4,3,-1,0,2);
max(S);
S := -4, 3, -1, 0, 2
3
3
```

On peut extraire le *n*-ième élément d'une suite *s* avec la notation *s[n]*

```
> s;
s[2];
a, b, 1, 1, 3, a x + 1, b, e^x - y
b
```

ou utiliser un intervalle pour extraire une sous-suite d'éléments consécutifs

```
> s[2..5],u,v,s[8];
b, 1, 1, 3, u, v, e^x - y
```

Un indice négatif indique que la position de l'élément doit être comptée en partant de la fin

```
> s[-1];
s[-4..-2];
```

```
s[5..-2];
 $e^x - y$ 
3, a x + 1, b
3, a x + 1, b
```

Comptage, changement des éléments d'une suite

On ne peut pas utiliser **nops** (fonction qui compte le nombre d'opérandes d'un objet; voir chapitre 2) directement pour compter les éléments pour la raison donnée plus haut

```
> nops(s);
Error, invalid input: nops expects 1 argument, but received 8
```

ou assigner un élément d'une suite

```
> s[3]:=cos(x);
Error, invalid assignment (a, b, 1, 1, 3, a*x+1, b, exp(x)-y)[3] := cos(x); cannot
assign to an expression sequence
```

Mais voici des "recettes" qui seront expliquées au paragraphe *Listes* de ce chapitre (voir chapitre 2 pour **subsop**)

```
> nops([s]);
s:=op(subsop(3=cos(x), -2=x^2, [s]));
8
s := a, b, cos(x), 1, 3, a x + 1, x^2, e^x - y
```

Génération automatique de suites, fonction seq et opérateur \$

La fonction **seq** permet la création automatique de suites avec une boucle indexée par un entier. L'expression suivante définit une suite pour un entier i variant de 0 à 10 par pas de 1 et dont les éléments sont définis par $i^2 - i$. Remarquons que la variable i est muette (i.e. la fonction **seq** ne modifie pas i). Nous verrons au paragraphe *Listes* une autre syntaxe *très utile* de la fonction **seq**.

```
> i:=3; # Pour montrer que la fonction seq ne modifie pas i
s:=seq(i^2-i,i=0..10);
i;
i := 3
s := 0, 0, 2, 6, 12, 20, 30, 42, 56, 72, 90
3
```

un autre exemple

```
> n:=5:
seq(evalf[3](cos(k*Pi/n)),k=0..n);
1., 0.809, 0.306, -0.306, -0.809, -1.
```

Cette fonction ouvre de nombreuses possibilités de générations automatiques d'objets

```
> seq(x||i, i=0..n); # voir chapitre 18
x0, x1, x2, x3, x4, x5
```

```
> P:='+'(seq(a[i]*x^i, i=0..n));
P := a0 + a1 x + a2 x^2 + a3 x^3 + a4 x^4 + a5 x^5
```

La génération automatique de suites peut aussi être obtenue à l'aide de l'opérateur \$. Contrairement à la fonction seq, *la variable de boucle doit être non assignée*. On utilisera sinon la syntaxe suivante

```
> 'i^2-i' $ 'i'=0..10;
0, 0, 2, 6, 12, 20, 30, 42, 56, 72, 90
```

ou bien une désassignation préalable (ou un autre nom non assigné)

```
> i:='i':
i^2-i $ i=0..10;
0, 0, 2, 6, 12, 20, 30, 42, 56, 72, 90
```

Quand les éléments de la liste ne dépendent pas de la variable de boucle on peut simplifier la syntaxe et obtenir ainsi un *opérateur de répétition* (voir une utilisation pratique au chapitre 8, *Dérivation*)

```
> x $ 4;
cos(x)-1 $ 3;
x, x, x, x
cos(x) - 1, cos(x) - 1, cos(x) - 1
```

Ensembles (set)

Un ensemble est une structure de données de MAPLE définie syntaxiquement par une suite encadrée par les accolades: { . , . , . }. *Conformément à la définition des ensembles, l'ordre d'entrée des éléments peut être changé par MAPLE puisqu'il est sans importance et les éléments répétés sont considérés comme un élément unique.*

```
> restart;
e:={z,y,x};
f:={x,x,y,x,z};
e := {z, y, x}
f := {z, y, x}
```

La nature des éléments d'un même ensemble peut être quelconque, y compris des ensembles.

```
> g:={sqrt(2)/2,{x,a},6,{a,x},x^2+3*x-1};
g := {  $\frac{\sqrt{2}}{2}$ , {x, a}, 6, {a, x},  $x^2 + 3x - 1$  }
```

Les ensembles possèdent le type set :

```
> whattype(e);   type(e, set);
set
true
```

Les éléments de l'ensemble e sont exclusivement des symboles, ce qui n'est pas le cas de g

```
> type(e, set(symbol));  type(g, set(symbol));
true
false
```

L'ensemble suivant est exclusivement constitué de termes de types symbol ou numeric

```
> type({3,x,z,0,y},set({symbol,numeric}));  
true
```

Avec la fonction **SetOf** on peut construire des ensembles de façon abstraite sans décrire les éléments un à un, mais simplement par une propriété globale *définie par un type*. Z est ici l'ensemble des entiers relatifs et $Z2$ l'ensemble des nombres pairs.

```
> Z:=SetOf(integer):  
Z2:=SetOf(even):
```

Attention: de tels objets n'ont pas le type *set*.

```
> type(Z2,set),type(Z2,function);  
false,true
```

Pour *transformer un ensemble en une suite* de ses éléments il suffit de lister les opérandes.

```
> op(e);  
z,y,x
```

L'expression **op(n,e)** permet d'extraire la n -ième opérande de l'ensemble e .

Attention : L'*ordre des éléments d'un ensemble est imprévisible* et cette instruction doit donc être utilisée avec prudence (voir plus loin la fonction **member**).

```
> op(2,e);  
y
```

Elle peut être remplacée par une notation plus concise (avec les mêmes précautions)

```
> e[2];  
y
```

Attention: la syntaxe suivante ne construit pas une suite de un élément, mais un sous-ensemble

```
> e[2..2];  
{y}
```

De même

```
> g[2..-2];  
 $\left\{ x^2 + 3x - 1, \frac{1}{2}\sqrt{2} \right\}$ 
```

La fonction **nops** donne le nombre d'opérandes d'un ensemble qui correspond donc à son cardinal.

```
> Card_e:=nops(e);  
Card_e := 3
```

Cependant ceci ne fonctionne pas pour des ensembles définis formellement

```
> Z;  
nops(Z);  
SetOf(integer)
```

1

Heureusement ;-), vous ne risquez pas ainsi de lister les nombres premiers en créant une catastrophe informatique sur votre ordinateur...

```
> PI:=SetOf(prime);
op(PI);
 $\Pi := SetOf(prime)$ 
prime
```

Pour *transformer une suite*

```
> p:=seq(2*i,i=0..5);
 $p := 0, 2, 4, 6, 8, 10$ 
```

en un ensemble, il suffit d'écrire

```
> P:={p};
 $P := \{0, 2, 4, 6, 8, 10\}$ 
```

ou directement

```
> P:={seq(2*i,i=0..5)};
 $P := \{0, 2, 4, 6, 8, 10\}$ 
```

Evaluations, Substitutions d'éléments dans un ensemble

Comme pour les expressions, elles se font avec les fonctions **eval**, **subs** ou **subsop** puisqu'un ensemble est constitué d'opérandes:

```
> G:= {x,2,3,-2*sqrt(2),0,ln(2),c,1};
 $G := \{0, 1, 2, 3, x, c, \ln(2), -2 \sqrt{2}\}$ 
```

```
> eval(G,x=5);
 $\{0, 1, 2, 3, 5, c, \ln(2), -2 \sqrt{2}\}$ 
```

On notera ici les changements opérés sur les éléments 2, $\sqrt{2}$ et $\ln(2)$. De plus 3 devient un élément double et l'ensemble résultat compte un élément de moins que G

```
> nops(G);
subs(2=3,G);
nops(%);
 $8$ 
 $\{0, 1, 3, x, c, \ln(3), -2 \sqrt{3}\}$ 
 $7$ 
```

Changement et suppression d'un élément dans un ensemble

La notation **G[n]:=** est réservée aux listes (voir paragraphe suivant), aux tables, tableaux, matrices et vecteurs (voir les chapitres 4, *Matrices, Vecteurs, Algèbre Linéaire* et 5, *Tables et Tableaux*) et ne peut pas être utilisée pour changer un élément dans un ensemble. On veut changer le 3-ième élément de G et le remplacer par z en écrivant

```
> G[3]:=z;
Error, cannot reassign the entries in a set
```

Il faudra substituer z à la troisième opérande de G (i.e. 2) et réassigner le résultat à G

```
> G:=subsop(3=z,G); # Attention: l'ordre des éléments est modifié
 $G := \{0, 1, 3, z, x, c, \ln(2), -2 \sqrt{2}\}$ 
```

Pour **supprimer un ou plusieurs éléments** d'un ensemble on peut écrire

```
> G:= G minus {1,0};
```

$$G := \{3, \ln(2), -2\sqrt{2}, z, x, c\}$$

On peut aussi remplacer un élément par le mot clé **NULL**.

```
> G:=subsop(3=NULL,G);
```

$$G := \{3, \ln(2), z, x, c\}$$

Le mot clé **NULL**

Ce mot clé est un concept informatique plutôt que mathématique. Il ne signifie pas qu'un objet est nul (égal à 0), mais qu'il n'existe pas. Pour illustrer ce concept, assignons **NULL** à x

```
> x:=NULL;
```

$$x :=$$

Si on cherche à afficher x , MAPLE ne répond rien. En fait MAPLE renvoie **NULL**

```
> x;
```

Mais x est bien un *symbole assigné* (sinon MAPLE aurait répondu x). La première réponse montre d'ailleurs que **NULL** ne signifie pas 0

```
> x+1;
assigned(x);
```

$$(0 + 1$$

true

Assigner **NULL** à un nom est différent d'une désassignation

```
> x:='x';
x+1;
assigned(x);
```

$$x := x$$

$$x + 1$$

false

Noter que **NULL** possède quand même un type qui est le même que celui d'une suite.

```
> whattype(NULL);
```

exprseq

Egalité entre ensembles, appartenance d'un élément à un ensemble

L'expression suivante est une **égalité** supposée entre deux ensembles. C'est le signe = seul qui sépare les deux membres et non := (ce n'est pas une assignation). La fonction **evalb** permet de tester cette égalité (*true* = vraie, *false* = faux). On remarque ici encore que toute expression est, dans la mesure du possible évaluée, ici factorielle 3 et $\sin(\pi/4)$ (dans cet exemple l'évaluation est retardée par l'utilisation des '...')

```
> g:={sqrt(2)/2,{x,a},6,{a,x},x^2+3};
'{{a,x},x^2+3,sin(Pi/4),3!}' = g;
evalb(%);
```

$$g := \left\{ 6, \frac{1}{2}\sqrt{2}, x^2 + 3, \{x, a\} \right\}$$

$$\left\{ x^2 + 3, 3!, \sin\left(\frac{1}{4}\pi\right), \{x, a\} \right\} = \left\{ 6, \frac{1}{2}\sqrt{2}, x^2 + 3, \{x, a\} \right\}$$

true

On peut également utiliser la fonction **verify** qui vérifie si A est égal à B en tant qu'ensemble

```
> A:={1,2,x*(x-1)*(x+1)-(x^2-1)*x};  
B:={0,1,2};  
evalb(A=B);  
verify(A,B,'set');
```

$$A := \{1, 2, x(x - 1)(x + 1) - (x^2 - 1)x\}$$

$$B := \{0, 1, 2\}$$

false
false

Exercice : cependant... expliquez

```
> verify(A,B,'set(expand)');
```

true

L'opérateur logique **subset** teste si un ensemble est un sous-ensemble d'un ensemble.

```
> {6,sqrt(2)/2} subset g;
```

true

On constate que *tout ensemble contient l'ensemble vide*

```
> {} subset g;
```

true

L'opérateur **in** permet d'écrire des relations d'appartenance et **evalb** ou **is** permettent de les vérifier

```
> sqrt(2)/2 in g;  
evalb(%);  
is(x+6 in g);
```

$$\frac{1}{2}\sqrt{2} \in \left\{ 6, \frac{1}{2}\sqrt{2}, x^2 + 3, \{x, a\} \right\}$$

true
false

On appelle Q l'ensemble des rationnels. On vérifie que $22/7$ appartient à cet ensemble

```
> Q:=SetOf(rational);  
22/7 in 'Q';  
is(%);
```

$Q := \text{SetOf}(rational)$

$$\frac{22}{7} \in Q$$

true

mais pas π ...! (Johann Heinrich Lambert, 1761)

```
> is(Pi in Q);  
false
```

Mais il y a encore des progrès à faire...

```
> evalb(Pi in SetOf(realcons) minus Q); # juste  
is(Pi in SetOf(realcons) minus Q); # ? (FAIL=Echec)  
true  
FAIL
```

Exercice: définir l'ensemble Π des nombres premiers (utiliser l'aide en ligne de **type**) et vérifier que $2^7 - 1$ est un élément de cet ensemble. En utilisant une fonction appropriée, vérifier d'une autre manière cette propriété.

```
> PI:=...;
```

Pour savoir si un élément appartient à un ensemble on pourra aussi utiliser la fonction **member**

```
> g;  
member(x,g);  
member({x,a},g);  

$$\left\{ 6, \frac{1}{2}\sqrt{2}, x^2 + 3, \{x, a\} \right\}$$
  
false  
true
```

La fonction **member** accepte un troisième argument facultatif qui permet d'obtenir la position de l'élément

```
> member({x,a},g,i);  
i;  
g[i];  
true  
4  
{x, a}
```

Attention, maintenant 4 est assigné à i et on conçoit qu'il ne soit pas possible d'exécuter la commande **member(6,g,i)** car MAPLE reçoit, en raison de l'évaluation préalable des arguments, la commande **member(6,g,4)** ce qui n'a aucun sens

```
> member(6,g,i);  
Error, invalid input: member expects its 3rd argument, p, to be of type name, but  
received 4
```

Pour exécuter à nouveau cette fonction, il faut transmettre le nom i non évalué (ou désassigner i ou utiliser un autre nom non assigné)

```
> member(6,g,'i');  
i; g[i];  
true  
1  
6
```

Attention, **member** ne reconnaît pas les ensembles définis formellement

```
> member(3, SetOf(integer));
false
```

Par contre...

```
> is(3 in SetOf(integer));
true
```

Opérations globales sur les ensembles

MAPLE peut effectuer des opérations usuelles sur les ensembles, comme une *intersection* ou une *union*. Par exemple on vérifie que l'*intersection* des ensembles *f* et *g* est vide

```
> f,g;
'f intersect g';
evalb(% = {});
```

$$\{z, y, x\}, \left\{ 6, \frac{1}{2}\sqrt{2}, x^2 + 3, \{x, a\} \right\}$$

$$f \cap g$$

$$true$$

ou encore

```
> Phi:='(f union e) intersect g';
Phi:=(f union e) intersect g;
Φ := (f ∪ e) ∩ g
Φ := {}
```

La "soustraction" (\setminus) de deux ensembles *A* et *B* se fait à l'aide de l'opérateur **minus**. L'opération *A minus* (\setminus) *B* peut se définir comme " éliminer de *A* tous les éléments communs à *A* et à *B*". Contrairement aux deux autres, cet opérateur n'est évidemment pas commutatif

```
> 'g minus {{x,a},7,sqrt(3),6}'=g minus {{x,a},7,sqrt(3),6};
'{{x,a},7,sqrt(3),6} minus g'={{x,a},7,sqrt(3),6} minus g;
g \ {6, 7, \sqrt{3}, \{x, a\}} = \left\{ \frac{1}{2}\sqrt{2}, x^2 + 3 \right\}
{6, 7, \sqrt{3}, \{x, a\}} \ g = \{7, \sqrt{3}\}
```

Ces fonctions s'appliquent aussi aux ensembles définis de façon formelle. Ici *R1* est l'ensemble des nombres réels non pairs (on "enlève" les nombres pairs à l'ensemble des réels)

```
> R1:=SetOf(realcons) minus SetOf(even);
evalb(-6 in R1);
evalb(Pi/2 in R1);
R1 := SetOf(realcons) \ SetOf(even)
false
true
```

Exercice: interprétez et testez...

```
> R1_0:=SetOf(realcons) minus SetOf(even) union {0};
R1_0 := (SetOf(realcons) \ SetOf(even)) ∪ {0}
```

Ces opérateurs sont binaires en ce sens qu'ils ne peuvent, en raison de la syntaxe utilisée, agir que sur deux ensembles. Si besoin, on peut aussi écrire ces opérateurs avec la syntaxe d'une fonction (de la même façon que $a + b$ peut s'écrire $+(a,b)$, voir chapitre 2)

```
> 'g union f union e union {{u,b}}' = `union` (g,f,e,{{u,b}});
((g ∪ f) ∪ e) ∪ {{b,u}} = { 6,  $\frac{1}{2}\sqrt{2}$ , z, y, {b,u}, x,  $x^2 + 3$ , {x,a} }
```

Les fonctions **minus**, **intersect** ou **subset** ne peuvent avoir, bien sûr, que deux arguments dont l'ordre, contrairement à **union**, n'est pas commutatif

```
> `minus` (f,g);
`minus` (g,f);
{z,y,x}
{ 6,  $\frac{1}{2}\sqrt{2}$ ,  $x^2 + 3$ , {x,a} }
```

La fonction **select** permet d'extraire d'un ensemble, l'ensemble des éléments ayant une propriété donnée. Le premier argument est un opérateur dont la réponse doit être un état logique (*true* ou *false*), le deuxième désigne l'ensemble, le troisième étant le deuxième argument de l'opérateur si nécessaire. Par exemple, si on souhaite savoir si x est du type **numeric** on écrit

```
> type(x, numeric);
```

Par conséquent, sachant que la réponse de **type** est toujours *true* ou *false*, pour sélectionner dans un ensemble S les éléments de type **numeric** on écrira

```
> select(type, S, numeric);
```

Rappel : si $\sqrt{3}$ est du type **constant**, c'est une écriture symbolique qui n'est pas du type **numeric** même si elle exprime une valeur numérique (voir chapitre 2, § *Fonctions whattype et type*).

```
> E:= {x,2,-2*sqrt(3),0,ln(2),c,1};
select(type,E,numeric);
select(type,E,constant);
E := {0, 1, 2, x, c, ln(2), -2  $\sqrt{3}$ }
{0, 1, 2}
{0, 1, 2, ln(2), -2  $\sqrt{3}$ }
```

Certaines fonctions comme **isprime** ou **issqr**, dont la réponse est un état logique *true* ou *false*, ne nécessitent qu'un argument. La fonction **select** ne reçoit alors que deux arguments. Ici la fonction **issqr** construit le sous-ensemble des carrés extraits de N_n

```
> Nn:={1,-2,2,4,0,6,7,9};
select(issqr,Nn);
Nn := {-2, 0, 1, 2, 4, 6, 7, 9}
{0, 1, 4, 9}
```

Avec la fonction **remove** on enlève d'un ensemble les éléments d'un type donné

```
> remove(type,E,symbol);
remove(type,Nn,even); # "even"=pair, "odd"=impair
{0, 1, 2, ln(2), -2  $\sqrt{3}$ }
```

{1, 7, 9}

On construit le sous-ensemble de Nn constitués par des nombres premiers

```
> remove(not isprime, Nn); # ou bien...
  select(isprime,Nn);
```

{2, 7}

{2, 7}

Les opérateurs de sélection peuvent être construits par l'utilisateur. Cet exemple construit un opérateur qui répond *true* si un élément est égal à x^2 (voir chapitre 7, *Fonctions*). On enlève ici de Lx le ou les éléments x^2

```
> Lx:={1,x,x^2,x^3,x^2,y^2}:
  remove(y->evalb(y=x^2),Lx);
```

{1, x^3 , y^2 , x }

Encore quelques exemples utilisant la fonction **has** qui détermine si un élément contient une expression donnée

```
> E:={sqrt(3*x),2,x+y,xy,x*y+2};
```

$E := \{2, x + y, \sqrt{3} \sqrt{x}, xy\}$

On crée l'ensemble des éléments de E ne contenant pas le nom y (un nom seul peut être considéré comme une expression) (**attention**: le dernier terme, xy , est un nom simple et n'est pas défini par $x*y$; ce n'est donc pas une expression contenant y)

```
> remove(has,E,y);
```

{2, $\sqrt{3} \sqrt{x}$, xy }

On crée l'ensemble des éléments de E contenant l'expression $x + y$ ou l'expression $x*y$. Le terme $x*y+2$ contient $x*y$

```
> select(has,E,{x+y,x*y});
```

{ $xy + 2$, $x + y$ }

Avec la fonction **hastype** on détermine si un élément contient une opérande du type précisé. Seuls $x + y$ et le nom xy ne contiennent pas d'opérande de type **constant**

```
> remove(hastype,E,constant);
```

{ $x + y$, xy }

Listes (list)

Une liste est une structure de données de MAPLE représentée syntaxiquement par une suite d'éléments encadrée par des crochets: [. . .]. Contrairement aux ensembles, *l'ordre des éléments est important et les répétitions sont préservées*. Ces objets jouent, comme les ensembles, un rôle essentiel dans MAPLE.

```
> restart:
L1:=[x,y,z,a,x,y,x];
L2:=[y,x,z,a,x,y,x];
```

$L1 := [x, y, z, a, x, y, x]$

$L2 := [y, x, z, a, x, y, x]$

Les listes ont le type **list**:

```
> whattype(L1), type(L2, list(symbol)), type(L2, list(integer));
list, true, false
```

On retrouve dans les deux listes précédentes un même nombre d'éléments identiques. Mais l'ordre étant différent, elles seront considérées comme différentes

```
> evalb(L1=L2);
false
```

Les éléments d'une liste peuvent être quelconques et hétéroclites. On rencontrera fréquemment des *listes de listes*. Elles sont notamment utiles pour trier, grouper, associer ou classer des objets.

```
> LL1:=[[a,b,c],[x,y,[u,v]],[1,{x,y},2,[3,z]]];
LL2:=[[a,b,c],[x,y,[u,v]],[1,{x,y},[3,z]]];
LL1 := [[a, b, c], [x, y, [u, v]], [1, {x, y}, 2, [3, z]]]
LL2 := [[a, b, c], [x, y, [u, v]], [1, {x, y}, [3, z]]]
```

Ci-dessus, $[u, v]$ est une opérande et une seule de $[x, y, [u, v]]$. *Si et seulement si* tous les éléments d'une liste sont des listes *de même nombre d'opérandes*, alors elle a le type *listlist*

```
> type(LL1,listlist);# voir la troisième liste
type(LL2,listlist);
false
true
```

```
> type(LL2,listlist(integer));
false
```

Et même si les listes sont vides... Une liste vide est non seulement valide, mais utile ! (difficile de donner un exemple ici, voir chapitre 19, *Procédures*)

```
> type([],[],listlist);
true
```

Pour transformer une suite en une liste il suffit d'écrire

```
> s:=a,b,2,z,cos(z);
Ls:=[s];
s := a, b, 2, z, cos(z)
Ls := [a, b, 2, z, cos(z)]
```

De même avec la fonction **seq**

```
> L:=[seq(i^2,i=1..5)];
L := [1, 4, 9, 16, 25]
```

IMPORTANT

Il existe une *deuxième forme* très utile de la fonction **seq** qui permet d'utiliser les opérandes d'une liste ou d'un ensemble au lieu d'un intervalle. Par exemple la variable k prend ici les valeurs successives des opérandes de la liste L et crée une liste dont les éléments seront triples de ceux de L

```
> [seq(3*k,k=L)];
[3, 12, 27, 48, 75]
```

On peut aussi écrire de façon plus naturelle

```
> [seq(3*k,k in L)];  
[3, 12, 27, 48, 75]
```

avec toutes formes de variantes possibles (avec les deux écritures)

```
> [seq(3*k,k in L[2..-2])];  
[12, 27, 48]
```

```
> E:={a,b,c,d};  
seq(cos(x^2),x in E);  
Lc:=[%];
```

$$\begin{aligned}E &:= \{b, c, a, d\} \\ \cos(b^2), \cos(c^2), \cos(a^2), \cos(d^2) \\ Lc &:= [\cos(b^2), \cos(c^2), \cos(a^2), \cos(d^2)]\end{aligned}$$

En réalité cette forme de la fonction **seq** est valable pour tout objet sur lequel on peut appliquer la fonction **op**, *i* prenant successivement les valeurs des opérandes (l'utilisation qui suit est plutôt marginale)

```
> ex:=x^2/2+ln(x)-3;  
op(ex);  
seq(i^2,i in ex);
```

$$\begin{aligned}ex &:= \frac{1}{2}x^2 + \ln(x) - 3 \\ &\frac{1}{2}x^2, \ln(x), -3 \\ &\frac{1}{4}x^4, \ln(x)^2, 9\end{aligned}$$

Réciproquement pour transformer une liste en une suite il suffit de lister ses opérandes

```
> Ls;  
op(Ls);
```

$$\begin{aligned}&[a, b, 2, z, \cos(z)] \\ &a, b, 2, z, \cos(z)\end{aligned}$$

Exercice: on dispose de deux listes $A := [1, 2, 3]$ et $B := [a, b, c]$. Construire C telle que $[1, 2, 3, a, b, c]$. L'assignation multiple $A, B := \dots$ s'applique aussi aux listes, aux ensembles, etc. (voir chapitre 2, *Assignations...*)

```
> A,B:=[1,2,3],[a,b,c];  
A, B := [1, 2, 3], [a, b, c]
```

Exercice: on dispose d'une **suite** *sp* de valeurs dérivées de π . Calculer une nouvelle **suite** constituée des cosinus des éléments de *s*.

```
> sp:=Pi,Pi/3,-Pi/4,3*Pi/2;
```

On retrouve beaucoup d'opérations communes aux listes et aux ensembles

```
> L1;
n:=nops(L1);
[x, y, z, a, x, y, x]
n := 7
```

Exercice: on a donné, au paragraphe *Suites*, une "recette" pour compter le nombre d'éléments d'une *suite*. Expliquez la recette.

```
> s:=x,[a,3],{y,x},-2,exp(x)-1,0;
nops(s);
nops([s]);
s := x, [a, 3], {x, y}, -2, e^x - 1, 0
```

```
Error, invalid input: nops expects 1 argument, but received 6
6
```

```
> L2:=eval(L1,z=cos(x));
L2 := [x, y, cos(x), a, x, y, x]
```

De façon similaire aux suites ou aux ensembles, pour extraire les éléments d'une liste on écrira

```
> L2[3];
L2[3..3];
L2[3..-2];
cos(x)
[cos(x)]
[cos(x), a, x, y]
```

Exercice: ce qui est très différent de... Analysez ces syntaxes.

```
> op(L2)[3..-2];
op(L2[3..-2]);
cos(x), a, x, y
cos(x), a, x, y
```

On trouvera cependant des différences et évidemment certaines opérations n'ont pas de sens avec les listes

```
> L1 union L2;
Error, invalid input: union expects its 1st argument, to be of type set, but received
[x, y, z, a, x, y, x]
```

Les fonctions comme **remove** ou **select** fonctionnent aussi sur des listes (voir § *Ensembles*). **Attention**, on crée ici une nouvelle liste anonyme, mais on ne modifie pas *L2* (réassigner à *L2* si c'est ce que l'on veut faire)

```
> remove(y->evalb(y=cos(x)),L2);
[x, y, a, x, y, x]
```

Les éléments d'une liste pouvant être répétés, la fonction **member** ne donnera que la première occurrence de l'élément demandé

```
> Lx:=[y,x,b,a,x,x,0];
member(x,Lx,'i');
i,Lx[i];
```

$Lx := [y, x, b, a, x, x, 0]$

true

$2, x$

Suppression d'éléments d'une liste

On peut utiliser **remove**

```
> Lx:=[x,x^2,x^3,x^3+1,x^4,x^5+2];
remove(has,Lx,x^3);# enlève x^3 et x^3+1
remove(y->evalb(y=x^3),Lx);
```

$Lx := [x, x^2, x^3, x^3 + 1, x^4, x^5 + 2]$

$[x, x^2, x^4, x^5 + 2]$

$[x, x^2, x^3 + 1, x^4, x^5 + 2]$

Comme pour les ensembles, on peut remplacer les opérandes concernées par **NULL** (voir paragraphe précédent). **Mais on ne peut pas le faire par une assignation**

```
> Lx[4]:=NULL;
```

Error, expression sequences cannot be assigned to lists

On peut procéder de plusieurs façons. **On notera** que l'on a créé précédemment des nouvelles listes sans modifier la liste initiale. Il faut procéder à une ré-assignation du résultat à Lx si on veut la modifier

```
> Lx:=subsop(4=NULL,Lx);
```

$Lx := [x, x^2, x^3, x^4, x^5 + 2]$

Ou bien

```
> eval(Lx,{x^3=NULL,x^5+2=NULL});
```

$[x, x^2, x^4]$

ou encore

```
> subsop(3=NULL,-1=NULL,Lx);
```

$[x, x^2, x^4]$

Modification des éléments d'une liste

Si l'on souhaite modifier un terme d'une liste on peut, **contrairement aux ensembles**, utiliser des assignations

```
> Lx[3]:=sin(x);
Lx;
```

$Lx_3 := \sin(x)$

$[x, x^2, \sin(x), x^4, x^5 + 2]$

Pour **ajouter un élément** à la fin (ou au début) d'une liste, il suffit de reconstruire la liste avec ses opérandes et ce que l'on souhaite rajouter au bon endroit

```
> Lx:=[op(Lx),0];
```

$Lx := [x, x^2, \sin(x), x^4, x^5 + 2, 0]$

Pour **insérer un élément** on procède de même en coupant la liste en deux parties (on veut insérer ici Y entre

x^2 et $\sin(x)$, c'est à dire entre la 2-ième et la 3-ième opérande). L'utilisation de **nops(Lx)** évite d'avoir à compter le nombre d'opérandes.

```
> L:=[op(Lx[1..2]),Y,op(Lx[3..nops(Lx)])];
L := [x, x2, Y, sin(x), x4, x5 + 2, 0]
```

Rappel: on peut remplacer **nops(Lx)** par -1

```
> L:=[op(Lx[1..2]),Y,op(Lx[3..-1])];
L := [x, x2, Y, sin(x), x4, x5 + 2, 0]
```

Pour **supprimer un élément** on peut aussi écrire

```
> L:=[op(L[1..2]),op(L[4..-1])];
L := [x, x2, sin(x), x4, x5 + 2, 0]
```

On ne manquera pas d'explorer également la bibliothèque ListTools qui contient 21 fonctions de manipulations des listes. Par exemple

```
> ListTools[Reverse](L);
ListTools[FlattenOnce]([1,[1,2],[a,b,c]]);
[0, x5 + 2, x4, sin(x), x2, x]
[1, 1, 2, a, b, c]
```

Tri d'une liste: on trie les éléments d'une partie de liste par ordre croissant puis décroissant avec la fonction **sort** (voir l'aide en ligne de *sort*; voir également les chapitres 7, *Fonctions* et 13, *Simplifications, Manipulations*)

```
> L:=[-3,4,0,2,-1,0,-3];
sort(L[2..-2]);
L := [-3, 4, 0, 2, -1, 0, -3]
[-1, 0, 0, 2, 4]
```

```
> sort(L[2..-2],(x,y)->is(x>y));
sort(L[2..-2],`>`);
[4, 2, 0, 0, -1]
[4, 2, 0, 0, -1]
```

ou encore

```
> ListTools[Reverse](sort(L[2..-2]));
[4, 2, 0, 0, -1]
```

Transformation d'une liste en ensemble

La fonction **convert** permet (parmi d'autres utilisations sur lesquelles nous reviendrons) de transformer une liste en un ensemble et réciproquement. Le deuxième argument *set* indique ici que la liste doit être convertie en un ensemble.

```
> L;
e:=convert(L,set);
[-3, 4, 0, 2, -1, 0, -3]
e := {-3, -1, 0, 2, 4}
```

Exercice : Réaliser la même opération en utilisant la fonction **op**

Maintenant on peut convertir cet ensemble en une liste

```
> convert(e,list);  
[-3, -1, 0, 2, 4]
```

Exercice : expliquez les syntaxes suivantes

```
> S:=[1,[x,y,z],{a,b,c,d*x}];  
S[2][-3];  
op([3,1],S);  
S := [1, [x, y, z], {b, c, a, d x}]  
x  
b
```

Addition et soustraction de listes

On peut ajouter et soustraire des listes, les opérations s'effectuant par éléments correspondants et à condition que les listes aient le même nombres de termes. On peut également multiplier ou diviser globalement une liste par des constantes de type **numeric** (à condition que ces opération est un sens). Voir aussi les fonctions **map** et **zip** au chapitre 7 et à la fin de ce chapitre pour des opérations plus complexes.

```
> L:=[a,b,c,d];  
2*[1,2,-3,4] + 3*L/2 - evalf(cos(Pi/5))*[1,0,-4,1];  
L := [a, b, c, d]  
[ 1.190983006 +  $\frac{3}{2}$  a, 4. +  $\frac{3}{2}$  b, -2.763932023 +  $\frac{3}{2}$  c, 7.190983006 +  $\frac{3}{2}$  d ]
```

Addition ou multiplication des éléments d'une liste

Avec la fonction **add** on peut très simplement ajouter les éléments, *ou une fonction des éléments*, d'une liste ou d'un ensemble. De façon similaire à la fonction **seq** la variable *i* prend ici les valeurs successives de la liste *L* (ou de l'ensemble). Cette variable est muette, c'est-à-dire qu'elle ne prend aucune valeur particulière après l'opération ou garde la valeur qu'elle avait avant celle-ci.

```
> i:=1;# Seulement pour vérifier que i reste muet  
add(i,i=L);  
i; # i conserve la valeur 1.  
i := 1  
a + b + c + d  
1
```

On peut, bien entendu, construire une fonction des éléments

```
> add(ln((x+1)^2),x in L);  
ln((a + 1)2) + ln((b + 1)2) + ln((c + 1)2) + ln((d + 1)2)
```

Encore un exemple

```
> S:=[sin(x),cos(x),2*sin(z)*cos(z),cos(2*z)];  
add(s^2,s in S);  
simplify(combine(%));# Voir Chapitre 13, Simplifications...  
S := [sin(x), cos(x), 2 sin(z) cos(z), cos(2 z)]
```

$$\frac{\sin(x)^2 + \cos(x)^2 + 4 \sin(z)^2 \cos(z)^2 + \cos(2z)^2}{2}$$

Avec la fonction **mul** on effectue ici le produit des inverses des carrés des éléments de la liste

```
> L:=[x,y,z-1];
mul(i^(-2),i in L);
L := [x, y, z - 1]

```

$$\frac{1}{x^2 y^2 (z - 1)^2}$$

Exercice : Créer une liste $L:=[a,b,c,d,e,f]$ et une suite $s:=1,2,3$. Insérer s dans L pour créer la liste $[a,b,1,2,3,c,d,e,f]$

La fonction map

Cette fonction joue un rôle important dans la manipulation des structures de données. Nous la considérerons ici dans le contexte des structures étudiées dans ce chapitre mais nous y reviendrons chaque fois que cela sera nécessaire. Elle permet d'appliquer un opérateur de quelque nature que ce soit aux opérandes d'un objet, mais naturellement sous réserve de compatibilité. Le **résultat est un objet de même nature**, une liste si on l'applique à une liste, un ensemble si on l'applique à un ensemble, une expression si on l'applique à une expression, etc. (voir chapitre 7, *Fonctions*)

Ici on crée une liste dont les éléments (les opérandes) vont servir d'arguments à la fonction *sinus*. La fonction **map** va appliquer cette fonction sur chacun des éléments de la liste pour fournir une nouvelle liste de résultats

```
> restart;
G:=[seq(Pi/i,i=1..8)];
G_sin:=map(sin,G);
G := [ π, 1/2 π, 1/3 π, 1/4 π, 1/5 π, 1/6 π, 1/7 π, 1/8 π ]
G_sin := [ 0, 1, 1/2 √3, 1/2 √2, sin(1/5 π), 1/2, sin(1/7 π), sin(1/8 π) ]
```

Si par exemple on veut transformer cette liste en approximations décimales à 3 chiffres, on pourra appliquer l'opérateur **evalf[3]** sur chacun des éléments de la liste en écrivant (voir aussi plus loin)

```
> map(evalf[3],G_sin);
[0., 1., 0.865, 0.705, 0.588, 0.500, 0.434, 0.382]
```

La fonction **map** s'applique aussi aux ensembles, aux intervalles et en fait à tous les objets de MAPLE ayant des opérandes et pour lesquels cette opération a un sens. Le résultat est un objet de même nature

```
> Intv:=-infinity..0;
map(exp,Intv);
map(evalc@ln,{ -1+3*I, 1+I, 1}); # pour l'opérateur @ voir Chap. 7,
Fonctions
```

$$Intv := -\infty .. 0$$

0 .. 1

$$\left\{ 0, \frac{1}{2} \ln(10) + I(-\arctan(3) + \pi), \frac{1}{2} \ln(2) + \frac{1}{4} I \pi \right\}$$

Voici un autre exemple : après avoir créé une liste de sinus d'arguments multiples, on souhaite la transformer en termes de lignes trigonométriques d'arguments simples. On appliquera pour cela l'opérateur **expand** (voir chapitre 13, *Simplifications, Manipulations*).

```
> H:=[seq(sin(i*x),i=1..4)];  
map(expand,H);
```

$$H := [\sin(x), \sin(2x), \sin(3x), \sin(4x)]$$

$$[\sin(x), 2 \sin(x) \cos(x), 4 \sin(x) \cos(x)^2 - \sin(x), 8 \sin(x) \cos(x)^3 - 4 \sin(x) \cos(x)]$$

Cependant, quand on applique certains opérateurs, notamment les opérateurs de transformations comme **subs**, **evalf**, **eval**, **expand**, etc., à une liste, un ensemble..., ils s'appliquent à tous les éléments sans qu'il soit nécessaire d'utiliser explicitement la fonction **map**.

```
> expand(H);  
[sin(x), 2 sin(x) cos(x), 4 sin(x) cos(x)^2 - sin(x), 8 sin(x) cos(x)^3 - 4 sin(x) cos(x)]
```

```
> evalf[3](G_sin[2..5]);  
[1., 0.865, 0.705, 0.588]
```

Mais ceci n'est pas valable pour toutes les fonctions, notamment les fonctions numériques

```
> sin(G);  
Error, invalid input: sin expects its 1st argument, x, to be of type algebraic, but  
received [Pi, 1/2*Pi, 1/3*Pi, 1/4*Pi, 1/5*Pi, 1/6*Pi, 1/7*Pi, 1/8*Pi]
```

Lorsque l'opérateur nécessite plus d'un argument, les arguments supplémentaires sont placés à la suite de l'objet sur lequel s'applique **map**. Par exemple **type** peut s'écrire **type(x, numeric)** d'où la syntaxe :

```
> G_sin;  
map(type,G_sin,numeric);  
[0, 1,  $\frac{1}{2}\sqrt{3}$ ,  $\frac{1}{2}\sqrt{2}$ ,  $\sin\left(\frac{1}{5}\pi\right)$ ,  $\frac{1}{2}$ ,  $\sin\left(\frac{1}{7}\pi\right)$ ,  $\sin\left(\frac{1}{8}\pi\right)$ ]  
[true, true, false, false, false, true, false, false]
```

Exercice : Expliquer, comparer

```
> type(G_sin,list(constant));  
true  
  
> op(convert(map(type,G_sin,constant),set));  
true  
  
> op({op(map(type,G_sin,constant))});  
true
```

La fonction map permet aussi d'appliquer des opérateurs définis par l'utilisateur (voir chapitre 7, Fonctions).

La fonction zip

Cette fonction permet de construire une liste en prenant pour arguments d'un **opérateur à deux variables**, les éléments de deux listes (elle s'applique aussi aux matrices, tableaux et vecteurs). Construisons deux listes de nombres entiers positifs de longueurs différentes

```
> N:=[seq(3*i+1,i=1..6)];  
P:=[seq(i,i=2..4)];  
N := [4, 7, 10, 13, 16, 19]  
P := [2, 3, 4]
```

La fonction binomial est une fonction à deux variables et **zip** va permettre de construire la liste des coefficients du binôme $C_{N_i}^{P_i} = \frac{N_i!}{P_i!(N_i - P_i)!}$ pour i variant de 1 au nombre d'éléments le plus petit des deux listes : C_4^2, C_7^3, C_{10}^4 .

```
> zip(binomial,N,P);  
[6, 35, 210]
```

La fonction zip permet aussi (de même que map) d'appliquer des opérateurs définis par l'utilisateur (voir chapitre 7, Fonctions). Les variables x et y prennent successivement les valeurs des opérandes de la première et deuxième liste.

```
> zip((x,y)->exp(-x/y),N,P);  
[ e^(-2), e^(-7/3), e^(-5/2) ]
```

On montre comment "entrelacer" deux listes pour obtenir une liste de listes (on rencontrera souvent cette application).

```
> A:=[2,-3,0,-5,4,1];  
B:=[x,y,z,t,u];  
zip((x,y)->[x,y],A,B);  
A := [2, -3, 0, -5, 4, 1]  
B := [x, y, z, t, u]  
[[2, x], [-3, y], [0, z], [-5, t], [4, u]]
```

4 - Vecteurs, Matrices et Algèbre Linéaire

Avertissement aux utilisateurs des versions MAPLE 5 et antérieures : les concepteurs de MAPLE ont considérablement modifié, dès la version 6, les éléments attachés à ce chapitre : nouvelles constructions et structures des vecteurs et matrices, notation de l'algèbre, nouvelle bibliothèque d'algèbre linéaire, calculs numériques... Les anciennes structures et notations ont été conservées pour des raisons de compatibilité mais nous ne décrirons ici que les nouvelles.

I - Construction des vecteurs et des matrices

1) Constructions élémentaires

MAPLE possède deux formes de constructions de vecteurs et de matrices. La première consiste en une écriture symbolique de la suite (sequence) des composantes du vecteur entourée des caractères < et >

```
> restart:
```

```
v:=<1,-1,3>;
```

$$v := \begin{bmatrix} 1 \\ -1 \\ 3 \end{bmatrix}$$

Les matrices sont définies *colonne par colonne* en juxtaposant les "vecteurs colonnes" avec le caractère | et entourant le tout par < et >

```
> M:=<<1,-1,3>|<1,0,-3>|<-1,2,1>>;
```

$$M := \begin{bmatrix} 1 & 1 & -1 \\ -1 & 0 & 2 \\ 3 & -3 & 1 \end{bmatrix}$$

Cette syntaxe autorise de façon simple l'*augmentation* en colonne(s) des matrices

```
> M1:=<M|<0,-2,-1>>;
```

$$M1 := \begin{bmatrix} 1 & 1 & -1 & 0 \\ -1 & 0 & 2 & -2 \\ 3 & -3 & 1 & -1 \end{bmatrix}$$

La deuxième construction utilise les fonctions **Vector** et **Matrix**. Nous verrons que ces fonctions permettent des constructions beaucoup plus élaborées.

Attention, Attention

On sera très attentif à utiliser un **V** et un **M** majuscule pour les mots **Vector** et **Matrix**. En effet, avec un **v** ou un **m** minuscule, MAPLE construit aussi un vecteur ou une matrice mais avec une autre structure (celle utilisée dans les versions antérieures du logiciel et conservée dans cette version pour des raisons de compatibilité; voir plus loin). **MAPLE ne signalera donc aucune erreur, mais l'objet ainsi construit n'aura pas les propriétés décrites dans ce chapitre.**

Pour les vecteurs on donne la *liste* des composantes

```
> v:=Vector([1,-1,3]);
```

$$v := \begin{bmatrix} 1 \\ -1 \\ 3 \end{bmatrix}$$

Pour les matrices on donne en argument une *liste de listes* décrivant la matrice *ligne par ligne* (voir cependant l'option *scan*, § 4 dans ce chapitre)

```
> M:=Matrix([[1,-1,0],[-1,2,-3]]);
```

$$M := \begin{bmatrix} 1 & -1 & 0 \\ -1 & 2 & -3 \end{bmatrix}$$

Evidemment les composantes peuvent être symboliques. **On notera** qu'il est possible d'assigner une suite de vecteurs ou de matrices à une suite de noms. On réalise ainsi des *assignations multiples* (voir chapitre 2) tout en économisant l'espace d'affichage. Ainsi:

```
> v,v1,M:=Vector([1,x,x^2]),<1,y,y^2>,
    Matrix([[1,x,x^2],[1,y,y^2],[0,1,2]]);
```

$$v, vI, M := \begin{bmatrix} 1 \\ x \\ x^2 \end{bmatrix}, \begin{bmatrix} 1 \\ y \\ y^2 \end{bmatrix}, \begin{bmatrix} 1 & x & x^2 \\ 1 & y & y^2 \\ 0 & 1 & 2 \end{bmatrix}$$

```
> v,M;
```

$$\begin{bmatrix} 1 \\ x \\ x^2 \end{bmatrix}, \begin{bmatrix} 1 & x & x^2 \\ 1 & y & y^2 \\ 0 & 1 & 2 \end{bmatrix}$$

Si le vecteur que l'on veut construire a des composantes toutes nulles ou toutes égales on pourra utiliser les formes suivantes

```
> v,v1:=Vector(3),Vector(3,-4);
```

$$v, vI := \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} -4 \\ -4 \\ -4 \end{bmatrix}$$

On retrouve une syntaxe analogue pour les matrices en donnant *dans l'ordre* le nombre de lignes puis le nombre de colonnes

```
> M,M1:=Matrix(2,3),Matrix(2,3,5);
```

$$M, M1 := \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \begin{bmatrix} 5 & 5 & 5 \\ 5 & 5 & 5 \end{bmatrix}$$

MAPLE comprend certaines syntaxes elliptiques dont la logique repose sur une information implicite. Pour la matrice M , la valeur -4 ne pouvant représenter une dimension, tous les éléments prennent cette valeur. Par défaut les matrices sont alors carrées. *On évitera néanmoins d'utiliser cette possibilité d'écriture dont la lecture est peu claire.*

```
> M,M1:=Matrix(2,-4),Matrix(2);
M, M1 :=  $\begin{bmatrix} -4 & -4 \\ -4 & -4 \end{bmatrix}, \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$ 
```

Il est donc très simple de créer des vecteurs à partir de *listes* et des matrices à partir de *listes de listes*. Si les dimensions ne sont pas spécifiées en argument, elles sont déterminées par la *taille maximum* des listes pour les colonnes et par leur *nombre* pour les lignes. Les informations manquantes sont alors remplacées par des 0 (par défaut; voir l'option *fill*, plus loin dans ce chapitre).

```
> Lv,LM:=[1,2],[[1,2,3],[4]];
Vector(Lv),Matrix(LM);

Lv, LM := [1, 2], [[1, 2, 3], [4]]
 $\begin{bmatrix} 1 \\ 2 \end{bmatrix}, \begin{bmatrix} 1 & 2 & 3 \\ 4 & 0 & 0 \end{bmatrix}$ 
```

Maintenant on utilise la même liste, mais on spécifie les dimensions

```
> v,M:=Vector(3,Lv),Matrix(3,3,LM);
v, M :=  $\begin{bmatrix} 1 \\ 2 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 & 2 & 3 \\ 4 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$ 
```

Réiproquement tout vecteur peut être converti en liste. Les matrices seront converties avec l'option *listlist*

```
> convert(v,list);
convert(M,listlist);
[1, 2, 0]
[[1, 2, 3], [4, 0, 0], [0, 0, 0]]
```

Il n'est pas prévu, avec la fonction **Matrix**, de construire directement une matrice à partir d'une liste simple. On doit donc reconstruire au préalable la *liste de listes* appropriée

```
> L:=[1,2,3,4,5,6];
L := [1, 2, 3, 4, 5, 6]

> Matrix(3,2,L);
Error, (in Matrix) initializer defines more columns (6) than column
dimension parameter specifies (2)

> Matrix([seq([seq(i,i in L[(j-1)*3+1..j*3])],j in 1..2)]);
 $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$ 
```

On pourra aussi utiliser une "astuce" avec l'ancienne fonction **matrix** qui permet une construction à partir d'une liste simple. Il reste ensuite à donner avec **convert** la structure correcte à la matrice que l'on veut créer:
Matrix, rtable

```
> convert(matrix(2,3,L),Matrix);
```

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

2) Extraction, assignation, changement des éléments d'un vecteur ou d'une matrice

La sélection des composantes d'un vecteur est identique à celle des listes. On peut utiliser des intervalles ou des valeurs négatives (voir au chapitre 3 le paragraphe relatif aux Listes). **Attention:** $v[2]$ donne x qui est la deuxième **composante** du vecteur v , alors que $v[2..2]$ donne $[x]$ qui est un **vecteur à une composante** (de même pour $[3]$).

```
> v:=[1,x,3,y];
v[2], v[2..3], v[2..2], v[-1], v[2..-2], v[-2..-2];
```

$$v := \begin{bmatrix} 1 \\ x \\ 3 \\ y \end{bmatrix}$$

$$x, \begin{bmatrix} x \\ 3 \end{bmatrix}, [x], y, \begin{bmatrix} x \\ 3 \end{bmatrix}, [3]$$

Attention: même remarque que précédemment: $M[1,-1]$ donne 3 qui est l'élément correspondant de la matrice alors que $M[1..1,-1..-1]$ forme [3] qui est **une matrice à un élément**

```
> M, M[1..2,2..-1], M[1,-1], M[1..1,-1..-1];
```

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \begin{bmatrix} 2 & 3 \\ 0 & 0 \end{bmatrix}, 3, [3]$$

Attention: de même que précédemment, le premier des objets suivants n'est pas un vecteur colonne mais une **matrice à une colonne** (de même pour une matrice ligne) alors que le second est bien un **vecteur** (voir § 4 , §§ Types et opérandes)

```
> M[1..2,1..1], M[1..2,1];
```

$$\begin{bmatrix} 1 \\ 4 \end{bmatrix}, \begin{bmatrix} 1 \\ 4 \end{bmatrix}$$

Pour les **assignments** on écrira

```
> v[2]:=2; # Ecrire v[2]:= ou v[2..2]:= est équivalent
v2:=2
```

On peut également remplacer un suite de composantes, *dont les indices sont consécutifs*, par un **Vecteur** de même nombres de composantes

```
> v[2..3]:=<4,5>; v;
```

$$\begin{bmatrix} 1 \\ 4 \\ 5 \\ y \end{bmatrix}$$

Si toutes les composantes à remplacer doivent prendre la même valeur, la syntaxe peut se simplifier

```
> v[2..-1]:=0; v;
```

$$\begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

Si les composantes à modifier sont contenues dans une liste, il faut utiliser une syntaxe cohérente. Voici d'abord ce qu'il ne faut pas faire

```
> L:=[-1,-2];
v[2..3]:=L;
v;
```

$$L := [-1, -2]$$

$$\begin{bmatrix} 1 \\ [-1, -2] \\ [-1, -2] \\ 0 \end{bmatrix}$$

Et ce que l'on doit faire

```
> v[2..3]:=<op(L)>;
v;
```

$$\begin{bmatrix} 1 \\ -1 \\ -2 \\ 0 \end{bmatrix}$$

ou encore

```
> v[2..3]:=Vector(L);
v;
```

$$\begin{bmatrix} 1 \\ -1 \\ -2 \\ 0 \end{bmatrix}$$

Pour les matrices on retrouve la même logique

```
> M:=Matrix([[1,x,x^2],[1,y,y^2],[1,2,3]]):      M;
M[2..3,1..2]:=<<a,b>|<c,d>>;
M[2..3,1..2]:=0:                                    M;
M;
```

$$\begin{bmatrix} 1 & x & x^2 \\ 1 & y & y^2 \\ 1 & 2 & 3 \end{bmatrix}$$

$$\begin{bmatrix} 1 & x & x^2 \\ a & c & y^2 \\ b & d & 3 \end{bmatrix}$$

$$\begin{bmatrix} 1 & x & x^2 \\ 0 & 0 & y^2 \\ 0 & 0 & 3 \end{bmatrix}$$

3) Affichage des grands vecteurs et des grandes matrices.

Lorsque les tailles des vecteurs ou des matrices sont grandes, MAPLE n'affiche qu'une information générale sur la structure de l'objet. Mais il est possible d'examiner son contenu réel en utilisant un afficheur spécial (browser) intégré dans MAPLE. Pour le faire apparaître, "double-cliquer" sur la partie affichée ou bien cliquer avec le **bouton de droite** de la souris puis sélectionner **browse...** On notera, avec cette deuxième méthode, les autres items du menu comme la lecture des matrices en mode "Image" et en niveaux de gris ou en couleurs (**attention** : tous les éléments de la matrice doivent avoir pour cela le type **numeric**). Pour cette fonctionnalité, l'exemple suivant n'est pas très parlant puisque tous les éléments sont identiques; voir plus loin l'exercice au paragraphe 4, *Génération automatique des composantes*. On peut également afficher une matrice avec un dessin perspective de type "surface" ou "histogramme" avec la fonction **matrixplot** de la bibliothèque **plots** (voir le chapitre 17 pour les représentations 3D).

```
> M_grande:=Matrix(20,20,4);
```

M_grande := $\begin{bmatrix} 20 \times 20 \text{ Matrix} \\ \text{Data Type: anything} \\ \text{Storage: rectangular} \\ \text{Order: Fortran_order} \end{bmatrix}$

On trouvera des informations sur le fonctionnement de l'afficheur avec la commande

```
> ?structuredview
```

Pour modifier la taille maximale par défaut que MAPLE est autorisée à afficher on utilisera la commande

suivante (la réponse affichée correspond à la valeur précédente)

```
> interface(rtsize=3);
```

10

Maintenant les objets dont *au moins* une des dimensions est supérieure à 3, ne seront plus affichés

```
> M_g2_3,M_g4_2:=Matrix(2,3,0),Matrix(4,2,0);
```

$$M_g2_3, M_g4_2 := \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \begin{bmatrix} 4 \times 2 \text{ Matrix} \\ \text{Data Type: anything} \\ \text{Storage: rectangular} \\ \text{Order: Fortran_order} \end{bmatrix}$$

On revient maintenant à une dimension maximum d'affichage de 10

```
> interface(rtsize=10);
```

3

4) Génération automatique des composantes

Les fonctions **Vector** et **Matrix** permettent la génération automatique de composantes avec l'introduction d'un opérateur (voir le chapitre 7, *Fonctions*). Dans le premier exemple, *i* va prendre successivement les valeurs 1, 2 et 3 et l'opérateur leur fera correspondre les valeurs $(-1)^i i^2$ (on peut naturellement choisir un autre nom que *i* pour construire l'opérateur)

```
> v_1,v_2:=Vector(3,i->(-1)^i*i^2),  
vector(3,k->x^(k-1)/k!);
```

$$v_1, v_2 := \begin{bmatrix} -1 \\ 4 \\ -9 \end{bmatrix}, \begin{bmatrix} 1 \\ \frac{1}{2}x \\ \frac{1}{6}x^2 \end{bmatrix}$$

mais d'autres formes d'écriture peuvent être utilisées

```
> L:=[seq(x^i/(i+1)!,i=0..2)];  
<seq((-1)^i*i^2,i=1..3)>;  
<op(L)>;
```

$$L := \left[1, \frac{1}{2}x, \frac{1}{6}x^2 \right]$$

$$\begin{bmatrix} -1 \\ 4 \\ -9 \end{bmatrix}, \begin{bmatrix} 1 \\ \frac{1}{2}x \\ \frac{1}{6}x^2 \end{bmatrix}$$

Pour les matrices on utilisera un opérateur à deux variables (voir le chapitre 7, *Fonctions*) qui représenteront respectivement l'indice de ligne puis l'indice de colonne (on *prendra soin* de mettre entre parenthèses les deux variables devant la flèche)

```
> M:=Matrix(2,3,(i,j)->(-1)^(i+j)*(i+j));
```

$$M := \begin{bmatrix} 2 & -3 & 4 \\ -3 & 4 & -5 \end{bmatrix}$$

Exercice : Construire une matrice 20x20 avec la même fonction d'indexation. L'examiner avec l'afficheur (voir paragraphe 3 de ce chapitre) en mode numérique et mode "Image".

5) Vecteurs de type ligne

La fonction **Vector** admet une option, **orientation**, qui définit la caractéristique **column** ou **row** c'est-à-dire **colonne** ou **ligne**. L'orientation par défaut est **column** (voir aussi la fonction **Transpose** de la bibliothèque **LinearAlgebra** examinée plus loin dans ce chapitre).

```
> l_1,l_2:=Vector([1,-2,3],orientation=row),
    Vector(3,i->x^(i-1)/i!,orientation=row);
l_1,l_2:=[1, -2, 3],  $\left[ 1, \frac{1}{2}x, \frac{1}{6}x^2 \right]$ 
```

MAPLE admet aussi la syntaxe

```
> l_1,l_2:=Vector[row]([1,-2,3]),
    Vector[row](3,i->x^(i-1)/i!);
l_1,l_2:=[1, -2, 3],  $\left[ 1, \frac{1}{2}x, \frac{1}{6}x^2 \right]$ 
```

ainsi que

```
> <1|2|x|x^2/2>;
 $\left[ 1, 2, x, \frac{1}{2}x^2 \right]$ 
```

6) Types et opérandes des vecteurs et des matrices

Les vecteurs construits précédemment ont le type **Vector** auquel est associé la caractéristique **column** ou **row**.

```
> v_1;
whattype(v_1);
```

$$\begin{bmatrix} -1 \\ 4 \\ -9 \end{bmatrix}$$

Vector_{column}

```
> l_1;
whattype(l_1);
[1, -2, 3]
Vectorrow
```

Avec la fonction **type** on écrira (voir chapitre 2)

```
> type(v_1,Vector);
```

```

type(v_1,Vector[column]);
type(v_1,Vector[row]);
                                true
                                true
                                false

> type(M,Matrix);
                                true

```

Attention aux Majuscules. Maple ne signale pas d'erreur de syntaxe car les types **vector** et **matrix** existent aussi

```

> type(v_1,vector);
type(v_1,Vector[column]);
type(M,matrix);
                                false
                                false
                                false

```

Les objets construits avec **Vector** ou **Matrix** ont leur structure propre, **rtable** (voir l'aide en ligne), que l'on peut découvrir avec les opérandes. Pour les vecteurs, la première opérande est la dimension du vecteur, la seconde l'ensemble des composantes. La troisième donne d'autres caractéristiques sur lesquelles nous reviendrons.

```

> op(v_1);
      3, {(1) = -1, (2) = 4, (3) = -9}, datatype = anything, storage = rectangular, order = Fortran_order,
      shape = []

```

Attention car contrairement aux apparences le nombre d'opérandes est seulement de 3 et la troisième opérande qui donne les caractéristiques est une suite (*datatype*=...*shape*=[])!

```

> nops(v_1);
whattype(op(3,v_1));
op(2,op(3,v_1)[1]);
                                3
                                exprseq
                                anything

```

Il en est de même pour les matrices. La première opérande (qui est ici la suite 2,3) donne le nombre de lignes puis le nombre de colonnes.

```

> nops(M);
op(M);
op(1,M);
                                3
      2, 3, {(1, 1) = 2, (1, 2) = -3, (1, 3) = 4, (2, 1) = -3, (2, 2) = 4, (2, 3) = -5}, datatype = anything,
      storage = rectangular, order = Fortran_order, shape = []
                                2, 3

```

L'opérande 0 indique le type du vecteur ou de la matrice. C'est le résultat renvoyé par **whattype**

```
> op(0,v_1),op(0,M);
```

Pour extraire les caractéristiques d'un vecteur ou d'une matrice (sur lesquelles nous reviendrons dans les prochains paragraphes), MAPLE dispose de deux fonctions **VectorOptions** et **MatrixOptions**.

```
> VectorOptions(v_1, orientation);
MatrixOptions(M, datatype, shape);
column
anything, []
```

Comme il a déjà été indiqué, les objets construits avec la fonction **vector** (**v** minuscule) ou **matrix** (**m** minuscule) ont une structure différente (structure **array**). Il est préférable d'utiliser la nouvelle forme, sauf pour raison de compatibilité.

```
> v_ancien:=vector([1,-1,3]);
v_ancien := [1, -1, 3]

> whattype(eval(v_ancien));
type(v_ancien, vector);
op(eval(v_ancien));
array
true
1 .. 3, [1 = 1, 2 = -1, 3 = 3]
```

Attention: il faut se méfier des apparences !

Exercice : Quels sont les types des objets suivants ?

```
> A:=<<1>|<2>|<x>>;
B:=<<1,2,x>>;
A := [1      2      x]
B := [ 1
      2
      x ]
```

7) Constructions avec options

L'option *shape*

L'option **shape** permet de caractériser le "profil" d'une matrice. L'option **identity** crée une *vraie* matrice identité, **non nécessairement carrée** (on rappelle que le nom **I** est, par défaut, réservé par MAPLE pour désigner $\sqrt{-1}$)

```
> Id:=Matrix(3,4,shape=identity);
Id := [ 1   0   0   0
        0   1   0   0
        0   0   1   0 ]
```

On entend par "vraie" le fait que l'on ne peut modifier aucun des éléments, diagonal ou non, de cette matrice qui perdrait sinon son caractère de matrice identité

```

> Id[2,2]:=3;
Id[2,3]:=1;
MatrixOptions(Id,shape);
Error, invalid assignment to identity diagonal
Error, invalid assignment of non-zero to identity off-diagonal
[identity]

```

L'option **shape=scalar[n]** définit une matrice diagonale, *non nécessairement carrée*, dont les éléments diagonaux sont égaux à n

```

> S:=Matrix(3,4,shape=scalar[-x]);
MatrixOptions(S,shape);
S := 
$$\begin{bmatrix} -x & 0 & 0 & 0 \\ 0 & -x & 0 & 0 \\ 0 & 0 & -x & 0 \end{bmatrix}$$

[scalar-x]

```

Ici aussi, toute tentative pour changer un des éléments, même diagonal, ferait perdre à la matrice sa caractéristique

```

> S[3,2]:=y;
S[2,2]:=4;
Error, scalar matrices must have zeros off the diagonal
Error, scalar matrices cannot be changed

```

Attention: si le produit d'une matrice identité par un scalaire (ici $-x$) peut donner une matrice diagonale identique à la précédente, elle n'aura pas pour autant la caractéristique $scalar[-x]$ et les modifications des composantes seront possibles.

```

> Sp:=-x*Id;
MatrixOptions(Sp,shape);
Sp := 
$$\begin{bmatrix} -x & 0 & 0 & 0 \\ 0 & -x & 0 & 0 \\ 0 & 0 & -x & 0 \end{bmatrix}$$

[]
```

L'option **shape=diagonal** construit des matrices diagonales. Ici encore on notera que de telles matrices ne sont pas nécessairement carrées (on rappelle que **D** est un nom prédéfini et protégé de MAPLE, voir chapitre 8, *Dérivation*)

```

> Di:=Matrix(3,4,-2,shape=diagonal);
Di := 
$$\begin{bmatrix} -2 & 0 & 0 & 0 \\ 0 & -2 & 0 & 0 \\ 0 & 0 & -2 & 0 \end{bmatrix}$$


```

Seuls les éléments diagonaux de ces matrices peuvent être changés

```
> Di[3,2]:=1;
Error, attempt to assign non-zero to off-diagonal entry of a diagonal
Matrix
```

```
> Di[3,3]:=5;
Di;
```

$$Di_{3,3} := 5$$

$$\begin{bmatrix} -2 & 0 & 0 & 0 \\ 0 & -2 & 0 & 0 \\ 0 & 0 & 5 & 0 \end{bmatrix}$$

Lorsque les éléments de la diagonale sont différents, ils doivent être spécifiés à l'aide d'un *Vecteur* (une simple liste ne convient pas)

```
> v:=Vector([-2,-1,1,2,3]):
Matrix(3,4,v[2..4],shape=diagonal), # ou bien directement
Matrix(3,4,Vector([-1,1,2]),shape=diagonal);
```

$$\begin{bmatrix} -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 2 & 0 \end{bmatrix}, \begin{bmatrix} -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 2 & 0 \end{bmatrix}$$

L'option *shape=symmetric* permet la construction de matrices symétriques. On notera les différences de fonctionnement de cette option suivant que la matrice est construite à partir d'une liste de listes ou d'une matrice

1) *La matrice symétrique est construite à partir d'une liste de listes.* On remarque que ce sont les colonnes (en partant de la première) qui définissent les éléments symétriques de la matrice

```
> L:=[[3,-2,1],[1,0,2],[0,-3,-2]]:
M,Ms:= Matrix(L), Matrix(L,shape=symmetric,scan=columns);
M, Ms :=
```

$$\begin{bmatrix} 3 & -2 & 1 \\ 1 & 0 & 2 \\ 0 & -3 & -2 \end{bmatrix}, \begin{bmatrix} 3 & 1 & 0 \\ 1 & 0 & -3 \\ 0 & -3 & -2 \end{bmatrix}$$

2) *La matrice symétrique est construite à partir d'une matrice.* Ce sont alors les lignes qui définissent les éléments symétriques. La deuxième construction est directe et ne nécessite pas la construction d'une matrice intermédiaire. Pour la troisième forme, se reporter au paragraphe suivant relatif à l'option *scan*.

```
> Matrix(M,shape=symmetric),
Matrix(Matrix(L),shape=symmetric),
Matrix(Matrix(L,scan=columns),shape=symmetric);
```

$$\begin{bmatrix} 3 & -2 & 1 \\ -2 & 0 & 2 \\ 1 & 2 & -2 \end{bmatrix}, \begin{bmatrix} 3 & -2 & 1 \\ -2 & 0 & 2 \\ 1 & 2 & -2 \end{bmatrix}, \begin{bmatrix} 3 & 1 & 0 \\ 1 & 0 & -3 \\ 0 & -3 & -2 \end{bmatrix}$$

Naturellement ces matrices sont de "vraies" matrices symétriques et *tout changement d'un de ces éléments entraîne celui de son symétrique*

```
> Ms[1,2]:=4:
```

```
Ms;
```

$$\begin{bmatrix} 3 & 4 & 0 \\ 4 & 0 & -3 \\ 0 & -3 & -2 \end{bmatrix}$$

Un exemple de l'option **shape=band** est donné au paragraphe *Combinaisons des options shape, scan et fill*.

L'option scan

L'option **scan** permet d'indiquer comment les éléments de la *liste de listes* donnée en argument de la fonction **Matrix** doivent être interprétés. Sans cette option, chaque liste est interprétée par défaut comme une ligne de la matrice. On peut modifier cette interprétation avec l'option **scan=columns**

```
> Matrix([[1,2,3],[0,-1,2],[-1,-3,-2]],scan=columns);
```

$$\begin{bmatrix} 1 & 0 & -1 \\ 2 & -1 & -3 \\ 3 & 2 & -2 \end{bmatrix}$$

Ici **scan=diagonal** indique que les éléments de la liste simple sont les éléments diagonaux de la matrice. **Attention:** en l'absence de l'option **shape=diagonal** la matrice n'aura pas ce profil et tous les éléments pourront être modifiés (on peut rajouter cette option). Aucune indication de taille n'étant donnée, c'est le nombre d'éléments de la liste qui fixe la taille de la matrice.

```
> Matrix([1,2,3],scan=diagonal);
```

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 3 \end{bmatrix}$$

L'option fill

Cette option permet de compléter les termes manquants dans la liste de listes par un terme fixé. Les dimensions sont ici fixées, par défaut, par les listes des éléments (on rappelle que \$ est l'opérateur de répétition qui est donné ici à titre d'exemple)

```
> Matrix([[2$3],[1,2,3,4],[y^2]],fill=x);
```

$$\begin{bmatrix} 2 & 2 & 2 & X \\ 1 & 2 & 3 & 4 \\ y^2 & X & X & X \end{bmatrix}$$

Mais naturellement les dimensions peuvent être fixées par deux entiers donnés en premiers arguments

```
> Matrix(3,4,[[1,2,3],[y]],[fill=-z);
```

$$\begin{bmatrix} 1 & 2 & 3 & -Z \\ y & -Z & -Z & -Z \\ -Z & -Z & -Z & -Z \end{bmatrix}$$

Combinaisons des options *shape*, *scan* et *fill*

La fonction **Matrix** autorise des combinaisons complexes de constructions grâce à ses fonctions d'indexation comme le montre ces exemples. Ici, les listes d'entrée sont interprétées en colonnes, mais la matrice doit être triangulaire supérieure. L'option *shape* va donc agir comme un filtre posant les éléments sous la diagonale égaux à 0

```
> Matrix([[1,2,3],[1,-1,2],[-1,-3,-2]],
      scan=columns,shape=triangular[upper]);
      ⎡ 1   1   -1 ⎤
      ⎢ 0   -1  -3 ⎥
      ⎣ 0    0  -2 ⎦
```

Ici le profil *shape=band[0,1]* crée une matrice "bande" avec sa diagonale normale, aucune (0) diagonale inférieure et une (1) diagonale supérieure. Le profil (*shape*) prévaut sur les données de la liste et les éléments de celle-ci sont remplacés par des 0. Ainsi l'élément (1,3) est nul ainsi que les éléments sous la diagonale normale

```
> Matrix([[1,2,3],[1,-1,2],[-1,-3,-2]],
      shape=band[0,1],scan=columns);
      ⎡ 1   1   0 ⎤
      ⎢ 0   -1  -3 ⎥
      ⎣ 0    0  -2 ⎦
```

La matrice est maintenant triangulaire supérieure et les listes d'entrée sont interprétées par *scan=band[0,3]* comme la diagonale et les 3 bandes supérieures successives. On notera que les dimensions de la matrice (qui n'est pas carrée bien que triangulaire) sont définies par la taille maximale des bandes données en entrée et que les éléments manquants sont remplacés par la valeur donnée par *fill*.

```
> Matrix([[1$3],[2$2],[3$4],[4]],
      shape=triangular[upper],scan=band[0,3],fill=z);
      ⎡ 1   2   3   4   z   z ⎤
      ⎢ 0   1   2   3   z   z ⎥
      ⎢ 0   0   1   z   3   z ⎥
      ⎣ 0   0   0   z   z   3 ⎦
```

Et toutes les possibilités de combinaisons compatibles sont autorisées. Le profil "triangulaire inférieure" prévaut ici et fait disparaître la bande supérieure définie par la liste [3,-2,1]:

```
> Matrix(3,5,[x$2],[y$3],[3,-2,1]),
      shape=triangular[lower],scan=band[1,1],fill=z);
      ⎡ y   0   0   0   0 ⎤
      ⎢ x   y   0   0   0 ⎥
      ⎣ z   x   y   0   0 ⎦
```

L'option datatype

Une autre option importante est **datatype** qui définit le type des composantes d'un vecteur ou d'une matrice. Par défaut le type est **anything** qui indique que les composantes peuvent être d'un type quelconque.

```
> VectorOptions(v,datatype);  
anything
```

En effet on peut construire, par défaut, un vecteur ou une matrice dont les composantes sont des objets quelconques de MAPLE (il est difficile de dire ce que l'on pourrait faire de l'exemple qui suit!)

```
> v_exotique:=Vector([1,-1..3,exp(-x),{1,2},3*x^2-1=0]);
```

$$v_{\text{exotique}} := \begin{bmatrix} 1 \\ -1 .. 3 \\ e^{-x} \\ \{1, 2\} \\ 3x^2 - 1 = 0 \end{bmatrix}$$

Par contre le vecteur suivant ne pourra avoir que des entiers pour composantes

```
> v:=Vector(3,datatype=integer);
```

$$V := \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

Et on ne peut assigner que des entiers aux composantes de ce vecteur

```
> v[2]:=3/2;  
v[2]:=x;  
  
Error, unable to store '3/2' when datatype=integer  
  
Error, unable to store 'x' when datatype=integer
```

Une autre option importante de **datatype** est **float** qui mérite une attention particulière. Construisons d'abord un vecteur ordinaire dont les composantes sont des décimaux

```
> vf_1:=Vector([0.5,1.3,2.657]);  
  
Vf_1 := 
$$\begin{bmatrix} 0.5 \\ 1.3 \\ 2.657 \end{bmatrix}$$

```

Comme le **datatype** de Vf_1 est **anything**, une assignation d'un nombre fractionnaire (ou tout autre objet) est autorisé et est conservé tel quel

```
> vf_1[2]:=-1/2;  
vf_1;
```

$$\begin{bmatrix} 0.5 \\ -\frac{1}{2} \\ 2.657 \end{bmatrix}$$

Maintenant utilisons l'option **float** pour *datatype*. Comme on peut le voir tous les nombres entiers sont **mis automatiquement** sous forme décimale (1 et -2 avec point décimal)

```
> vf_2:=Vector([1,2.,-2],datatype=float);
```

$$Vf_2 := \begin{bmatrix} 1. \\ 2. \\ -2. \end{bmatrix}$$

Attention

Contrairement à l'exemple de *Vf_1*, le nombre fractionnaire $-1/2$ est maintenant mis dans *Vf_2* sous forme décimale avant d'être affecté à la deuxième composante. On notera cependant que le nombre compte 18 chiffres et non 10 (la valeur par défaut de **Digits**). Le nombre est en effet évalué avec la fonction **evalhf** (et non **evalf**) et mis en mémoire sous une représentation binaire standard sur 8 octets

```
> vf_2[2]:=-1/2;
vf_2;
```

$$Vf_2_2 := -\frac{1}{2}$$

$$\begin{bmatrix} 1. \\ -0.5000000000000000 \\ -2. \end{bmatrix}$$

```
> evalf(-1/2);
evalhf(-1/2);
-0.5000000000
-0.5000000000000000
```

Les conséquences sont importantes: l'utilisation de **evalf** avec une précision insuffisante pour un vecteur ou une matrice construite avec l'option **datatype=float** conduit à une perte de précision

```
> vf_2[2]:=evalf(1/3);
vf_2[3]:=evalf(Pi);
vf_2;
```

$$Vf_2_2 := 0.333333333$$

$$Vf_2_3 := 3.141592654$$

$$\begin{bmatrix} 1. \\ 0.33333333299999978 \\ 3.14159265400000010 \end{bmatrix}$$

Il faudra utiliser soit la fonction **evalf** avec une précision suffisante soit la fonction **evalhf**. Les erreurs que

l'on peut constater sur les deux derniers chiffres sont dues à la conversion de la représentation décimale vers la représentation binaire et réciproquement.

```
> vf_2[2]:=evalf[20](1/3);
vf_2[3]:=evalhf(Pi);
vf_2;
```

$$Vf_2 := 0.33333333333333333333$$

$$Vf_2 := 3.14159265358979312$$

$$\begin{bmatrix} & & & 1. \\ & & 0.33333333333333314 \\ & 3.14159265358979312 \end{bmatrix}$$

Naturellement ces remarques concernent aussi les matrices.

Conclusion: l'option **datatype=float** construit des matrices et des vecteurs dont les éléments sont rangés en mémoire avec 8 octets et la précision ne peut être changée. Cette option est utilisée automatiquement, par défaut, par la bibliothèque **LinearAlgebra** lorsque les vecteurs et les matrices sont traités par des méthodes numériques (voir plus loin dans ce chapitre).

L'option storage

L'option **storage** sera utilisée exclusivement pour des matrices de grandes tailles.

```
> T:=Matrix(3,4,[[-1$3],[1$3]],storage=triangular[upper],
            scan=band[0,1],fill=x);
```

$$T := \begin{bmatrix} -1 & 1 & X & X \\ - & -1 & 1 & X \\ - & - & -1 & 1 \end{bmatrix}$$

Elle permet d'économiser l'espace mémoire en ne mémorisant que les éléments significatifs. La contrepartie est que certains éléments ne sont pas accessibles tout simplement parce qu'ils n'existent pas!

```
> T[2,3];
T[3,2];
1
```

```
Error, unable to lookup index (3, 2) -- not in storage
```

L'option readonly

L'option **readonly** ("lire seulement") est un état logique qui, par défaut, vaut *false* et indique que les éléments peuvent être modifiés (ceux autorisés par l'option **shape**). En posant **readonly=true**, les éléments du vecteur ou de la matrice ne pourront être que lus.

```
> M:=Matrix([[1,2],[-1,3]],readonly=true);
M[1,2];
M[1,2]:=3;
```

$$M := \begin{bmatrix} 1 & 2 \\ -1 & 3 \end{bmatrix}$$

Error, cannot assign to a read-only Matrix

On pourrait ainsi multiplier les exemples et ceux donnés ici sont loin de représenter un échantillon exhaustif.
On devra se reporter à l'aide en ligne pour les compléments d'information.

II - Assignations et fonction copy

Les matrices et les vecteurs ont un comportement particulier relativement à l'assignation. Rappelons tout d'abord le comportement habituel: lors de l'assignation de a à b , a est évalué

```
> restart:  
a:=[x,y,z]:  
b:=a:  
a,b;  
[x, y, z], [x, y, z]
```

Si l'on modifie l'un des éléments de l'une des listes, l'autre reste inchangée.

```
> a[1]:=0:  
a,b;  
[0, y, z], [x, y, z]
```

Construisons un schéma d'assignation identique avec des matrices (il en serait de même pour les vecteurs)

```
> A:=Matrix([[1,2],[0,-1]]):  
B:=A:  
A,B;  

$$\begin{bmatrix} 1 & 2 \\ 0 & -1 \end{bmatrix}, \begin{bmatrix} 1 & 2 \\ 0 & -1 \end{bmatrix}$$

```

Modifions un des éléments de la matrice A . On peut constater que *les deux matrices ont été modifiées*. En effet, lorsque l'on écrit $B:=A$, MAPLE demande aux deux identificateurs A et B de pointer sur le même tableau de données (une même structure *rtable*). Autrement dit, on crée deux symboles, A et B , pour une même matrice.

```
> A[2,1]:=-3:  
A,B;  

$$\begin{bmatrix} 1 & 2 \\ -3 & -1 \end{bmatrix}, \begin{bmatrix} 1 & 2 \\ -3 & -1 \end{bmatrix}$$

```

Pour obtenir deux matrices indépendantes il faut utiliser la fonction **copy**. La raison de cette différence de traitement vient d'un souci d'économiser la mémoire de l'ordinateur car souvent les matrices et les vecteurs sont des objets encombrants. La fonction **copy** oblige l'utilisateur à être attentif à ce problème.

```
> A:=Matrix(2,3,<1,2>,shape=diagonal);  
B:=copy(A);  
A[1,1]:=-1: # On modifie un élément de A  
A,B; # Les deux matrices sont distinctes  
A := 
$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \end{bmatrix}$$

```

$$B := \begin{bmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ -1 & 0 & 0 \\ 0 & 2 & 0 \end{bmatrix}, \begin{bmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \end{bmatrix}$$

On notera que la fonction **copy** conserve les options de la matrice copiée.

```
> MatrixOptions(A, shape);
MatrixOptions(B, shape);
B[1,2]:=1;
[diagonal]
[diagonal]
```

Error, attempt to assign non-zero to off-diagonal entry of a diagonal Matrix

III - Algèbre linéaire élémentaire

L'algèbre des vecteurs et des matrices peut être symbolisée d'une façon très voisine de celle des quantités scalaires *à l'exception du produit standard qui n'est pas une opération commutative* (et qui peut aussi avoir plusieurs significations). Construisons des matrices et des vecteurs pour les exemples qui vont suivre

```
> restart;
v1,v2:=Vector([1,2]),Vector([x,y]);
M1,M2:=Matrix([[1,2],[-1,0]]),
Matrix([[1,3],[a,b]]);
```

$$v1, v2 := \begin{bmatrix} 1 \\ 2 \end{bmatrix}, \begin{bmatrix} x \\ y \end{bmatrix}$$

$$M1, M2 := \begin{bmatrix} 1 & 2 \\ -1 & 0 \end{bmatrix}, \begin{bmatrix} 1 & 3 \\ a & b \end{bmatrix}$$

L'addition (ou la soustraction) se traite comme pour les scalaires. On notera que le produit d'une matrice ou d'un vecteur par un scalaire est autorisé et fournit une matrice ou un vecteur dont tous les éléments sont multipliés par ce scalaire.

```
> v1+2*v2;
M2-3*M1;
```

$$\begin{bmatrix} 1 + 2x \\ 2 + 2y \end{bmatrix}$$

$$\begin{bmatrix} -2 & -3 \\ a + 3 & b \end{bmatrix}$$

De même pour les puissances. L'expression $M2^{-2}$ calcule le carré de l'inverse de $M2$.

```
> M1^2,M2^(-2);
```

$$\begin{bmatrix} -1 & 2 \\ -1 & -2 \end{bmatrix}, \begin{bmatrix} \frac{3a+b^2}{b^2+9a^2-6ba} & -\frac{3(1+b)}{b^2+9a^2-6ba} \\ -\frac{a(1+b)}{b^2+9a^2-6ba} & \frac{1+3a}{b^2+9a^2-6ba} \end{bmatrix}$$

Ceci permet de construire des expressions matricielles plus complexes

```
> M1^2-2*M1+Matrix(2,shape=identity);
[ -2 -2 ]
[ 1 -1 ]
```

L'analyseur syntaxique de MAPLE interdit toute opération illicite et refuse en particulier d'effectuer le produit $M1*M2$.

```
> v1+<1,2,3>;#Les dimensions sont incompatibles
Error, (in rtable/Sum) invalid arguments
> v1+M1;
Error, (in rtable/Sum) invalid arguments
> M1*M2;
Error, (in rtable/Product) invalid arguments
> M1/M2;
Error, (in rtable/Product) invalid arguments
```

Cependant la logique n'est pas toujours évidente et doit inciter à la prudence. Ainsi MAPLE autorise l'addition d'un scalaire à une matrice, mais l'opération consiste à ajouter à la matrice le produit de ce scalaire avec l'identité (l'opération suivante calcule $M1-3 = M1-3Id$)

```
> M1, M1-3, M1-3*Matrix(2,shape=identity);
[ 1 2 ] [ -2 2 ] [ -2 2 ]
[ -1 0 ] [ -1 -3 ] [ -1 -3 ]
```

Il sera cependant facile d'écrire pour ajouter -3 à chacun des éléments de $M1$

```
> M1+Matrix(2,2,-3);
[ -2 -1 ]
[ -4 -3 ]
```

De même l'opération suivante qui voudrait ajouter 3 à chaque composante est interdite

```
> v1+3;
Error, (in rtable/Sum) invalid arguments
```

et il faut écrire

```
> v1+Vector(2,3);
```

$$\begin{bmatrix} 4 \\ 5 \end{bmatrix}$$

L'opérateur produit matriciel ".."

La raison pour laquelle les produits standard sont refusés vient de ce que "*" est toujours considéré par l'analyseur syntaxique de MAPLE comme commutatif ce qui est incorrect pour l'algèbre matricielle (sauf si les matrices commutent - par définition!). Aussi MAPLE introduit l'**opérateur produit non commutatif ". "**. Il est automatiquement interprété en fonction des objets sur lesquels il doit agir et effectue les opérations appropriées.

Cependant, hors du contexte de l'algèbre matricielle (ou d'un contexte nécessitant une multiplication non commutative), son utilisation est déconseillée en raison d'une règle syntaxique pas toujours aisée à manipuler. En particulier cette règle doit lever l'ambiguïté d'interprétation avec le point décimal (voir l'aide en ligne et le chapitre 1, *Nombres dans N, Q, R et C*). Voici quelques exemples à titre de rappel

```
> 3 . 4;   3.4;
                                         12
                                         3.4

> 3.e2; # On doit écrire 3.0e2;
Error, missing operator or `;`

> 3. 4;
Error, unexpected number

> x . 3 .(a*x+b);
x. 3. (a*x+b);      #!!!
                                         3 (x . (a x + b))
                                         3.x
```

Par contre, pour les objets de l'algèbre linéaire, son utilisation ne présente aucune difficulté

```
> M1 . v1 , M1.M2;
                                         5
                                         [-1], [ 1 + 2 a      3 + 2 b]
                                         -1                  -3
```

Ici aussi la cohérence des opérations est contrôlée. On ne peut pas multiplier à gauche une matrice carrée par un vecteur colonne

```
> v1.M1;
Error, (in LinearAlgebra:-VectorMatrixMultiply) invalid input:
`LinearAlgebra:-VectorMatrixMultiply` expects its 1st argument, v, to be
of type Vector[row] but received Vector[Column]( 1..2,[ 1, 2] , datatype =
anything, storage = rectangular, order = C_order )
```

Par contre l'opération suivante est parfaitement correcte

```
> V,M3:=Vector[row]([x,y]),<M1|<-1,2>>;
V.M3;
whattype(%);
```

$$V, M3 := [x, y], \begin{bmatrix} 1 & 2 & -1 \\ -1 & 0 & 2 \end{bmatrix}$$

$$[x - y, 2x, -x + 2y]$$

Vector_{row}

IV - Algèbre linéaire : la bibliothèque LinearAlgebra

Les calculs de l'algèbre linéaire font appel à de nombreuses opérations qui ne se résument pas à l'algèbre symbolique présentée ci-dessus. Aussi MAPLE dispose d'un ensemble de procédures (fonctions) contenues dans la bibliothèque **LinearAlgebra**.

Attention: ces fonctions ne seront disponibles qu'après avoir fait appel à cette bibliothèque avec la fonction **with** (ou l'opérateur "`:-`", voir le §§ *Accès aux fonctions de LinearAlgebra* de ce paragraphe). On pourra *supprimer l'affichage* de la liste des fonctions en terminant la commande par "`:`".

> **restart;**

with(LinearAlgebra);

```
[&x, Add, Adjoint, BackwardSubstitute, BandMatrix, Basis, BezoutMatrix, BidiagonalForm, BilinearForm,
CharacteristicMatrix, CharacteristicPolynomial, Column, ColumnDimension, ColumnOperation,
ColumnSpace, CompanionMatrix, ConditionNumber, ConstantMatrix, ConstantVector, Copy,
CreatePermutation, CrossProduct, DeleteColumn, DeleteRow, Determinant, Diagonal,
DiagonalMatrix, Dimension, Dimensions, DotProduct, EigenConditionNumbers, Eigenvalues,
Eigenvectors, Equal, ForwardSubstitute, FrobeniusForm, GaussianElimination, GenerateEquations,
GenerateMatrix, GetResultDataType, GetResultShape, GivensRotationMatrix, GramSchmidt,
HankelMatrix, HermiteForm, HermitianTranspose, HessenbergForm, HilbertMatrix,
HouseholderMatrix, IdentityMatrix, IntersectionBasis, IsDefinite, IsOrthogonal, IsSimilar, IsUnitary,
JordanBlockMatrix, JordanForm, LA_Main, LUDecomposition, LeastSquares, LinearSolve, Map,
Map2, MatrixAdd, MatrixExponential, MatrixFunction, MatrixInverse, MatrixMatrixMultiply,
MatrixNorm, MatrixPower, MatrixScalarMultiply, MatrixVectorMultiply, MinimalPolynomial, Minor,
Modular, Multiply, NoUserValue, Norm, Normalize, NullSpace, OuterProductMatrix, Permanent,
Pivot, PopovForm, QRDecomposition, RandomMatrix, RandomVector, Rank,
RationalCanonicalForm, ReducedRowEchelonForm, Row, RowDimension, RowOperation, RowSpace,
ScalarMatrix, ScalarMultiply, ScalarVector, SchurForm, SingularValues, SmithForm, SubMatrix,
SubVector, SumBasis, SylvesterMatrix, ToeplitzMatrix, Trace, Transpose, TridiagonalForm,
UnitVector, VandermondeMatrix, VectorAdd, VectorAngle, VectorMatrixMultiply, VectorNorm,
VectorScalarMultiply, ZeroMatrix, ZeroVector, Zip]
```

Il ne peut être question de donner des exemples pour toutes ces fonctions et on se contentera d'en illustrer quelques unes. Un certain nombre de ces fonctions comme **Add**, **MatrixMatrixMultiply**, etc. réalisent des opérations d'algèbre matricielle (vectorielle) qui sont automatiquement invoquées par l'analyseur syntaxique comme indiqué au paragraphe précédent. Il sera donc inutile de les appeler explicitement *sauf si l'on souhaite pouvoir utiliser les options* de ces fonctions (on en donnera quelques exemple et on complétera l'information avec l'aide en ligne)

```
> M1, M2:=Matrix([[1,2],[3,4]]),Matrix([[-1,3],[-2,1]]);
Add(M1,M2) , M1+M2; # 2 écritures identiques
```

$$M1, M2 := \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}, \begin{bmatrix} -1 & 3 \\ -2 & 1 \end{bmatrix}$$

$$\begin{bmatrix} 0 & 5 \\ 1 & 5 \end{bmatrix}, \begin{bmatrix} 0 & 5 \\ 1 & 5 \end{bmatrix}$$

```
> MatrixMatrixMultiply(M1,M2) , M1.M2; # 2 écritures identiques
```

$$\begin{bmatrix} -5 & 5 \\ -11 & 13 \end{bmatrix}, \begin{bmatrix} -5 & 5 \\ -11 & 13 \end{bmatrix}$$

L'option *inplace*

Certaines fonctions de **LinearAlgebra** autorisent l'option *inplace*. Elle indique que la fonction va remplacer par le résultat le premier argument compatible. Son intérêt est évidemment lié à un souci d'économie de l'espace mémoire de l'ordinateur quand on travaille avec de grosses matrices. Ici le résultat de la somme est une matrice 2x2 et le premier argument compatible est la matrice *M1* initiale qui sera remplacée par la somme

```
> M1,M2:=Matrix([[1,2],[2,1]]),Matrix([-1,0],[2,1]);
Add(M1,M2,inplace):
M1,M2;
```

$$M1, M2 := \begin{bmatrix} 1 & 2 \\ 2 & 1 \end{bmatrix}, \begin{bmatrix} -1 & 0 \\ 2 & 1 \end{bmatrix}$$

$$\begin{bmatrix} 0 & 2 \\ 4 & 2 \end{bmatrix}, \begin{bmatrix} -1 & 0 \\ 2 & 1 \end{bmatrix}$$

On aurait pu écrire **M1 :=M1+M2**. Pourtant ce n'est pas toujours équivalent. On peut constater dans cet exemple que la réassigmentation de la somme à *M1* fait perdre à cette matrice son profil symétrique

```
> M1,M2:=Matrix([[1,2],[2,1]],shape=symmetric),
      Matrix([-1,0],[2,1]);
M1:=M1+M2;
M1,M2;
MatrixOptions(M1,shape);
```

$$M1, M2 := \begin{bmatrix} 1 & 2 \\ 2 & 1 \end{bmatrix}, \begin{bmatrix} -1 & 0 \\ 2 & 1 \end{bmatrix}$$

$$\begin{bmatrix} 0 & 2 \\ 4 & 2 \end{bmatrix}, \begin{bmatrix} -1 & 0 \\ 2 & 1 \end{bmatrix}$$

[]

Réalisée avec la fonction **Add** et l'option *inplace* la matrice *M1* conserve son profil. Cependant le résultat de l'addition contenu dans la nouvelle matrice *M1* est différent puisqu'il est incompatible avec le profil. Le calcul tient compte de la symétrie et la somme est calculée suivant la première colonne suivit de la définition de la première ligne par symétrie, puis de la deuxième colonne, etc. et par conséquent

```
> M1,M2:=Matrix([[1,2],[2,1]],shape=symmetric),
      Matrix([-1,0],[2,1]);
Add(M1,M2,inplace):
M1,M2;
MatrixOptions(M1,shape);
```

$$M1, M2 := \begin{bmatrix} 1 & 2 \\ 2 & 1 \end{bmatrix}, \begin{bmatrix} -1 & 0 \\ 2 & 1 \end{bmatrix}$$

$$\begin{bmatrix} 0 & 4 \\ 4 & 2 \end{bmatrix}, \begin{bmatrix} -1 & 0 \\ 2 & 1 \end{bmatrix}$$

[symmetric]

On échange maintenant l'ordre des matrices par rapport à l'exemple précédent. L'opération conserve la propriété de "matrice symétrique" à *M1* et la matrice *M2* contient alors la somme réelle des matrices.

```
> M1,M2:=Matrix([[1,2],[2,1]]),shape=symmetric),
      Matrix([[-1,0],[2,1]]);
```

Add(M2,M1,inplace):

M1,M2;

MatrixOptions(M1,shape);

$$M1, M2 := \begin{bmatrix} 1 & 2 \\ 2 & 1 \end{bmatrix}, \begin{bmatrix} -1 & 0 \\ 2 & 1 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 2 \\ 2 & 1 \end{bmatrix}, \begin{bmatrix} 0 & 2 \\ 4 & 2 \end{bmatrix}$$

[symmetric]

L'opération d'addition avec **Add(...,inplace)** n'est donc pas toujours commutative et la prudence doit être de mise avec la conjugaison des options *shape* et *inplace*. L'opération est même impossible avec le produit de matrices (*MatrixMatrixMultiply*) puisqu'on ne peut plus réaliser la commutation.

Accès aux fonctions de LinearAlgebra

Pour l'instant toutes les fonctions de **LinearAlgebra** ont été rendues accessibles avec **with** et sont toutes disponibles. On peut, par exemple, réaliser le transposé d'un vecteur (ou d'une matrice) ainsi que son transposé conjugué (la barre de surlignement indique le complexe conjugué des composantes)

```
> V:=Vector([a,b,c]);
Transpose(V);
HermitianTranspose(V);
```

$$V := \begin{bmatrix} a \\ b \\ c \end{bmatrix}$$

$[a, b, c]$

$[\bar{a}, \bar{b}, \bar{c}]$

Après un ordre **restart** ces fonctions ne seront plus disponibles et **Transpose** devient le nom d'une fonction non définie. On rappelle qu'en tant que calculateur symbolique, *il est normal que MAPLE accepte cette situation et ne signale aucune erreur*.

```
> restart;
V:=Vector([a,b,c],orientation=row);
Transpose(V);
V := [a, b, c]
Transpose([a, b, c])
```

La fonction **with** permet d'avoir accès à la ou les fonctions de la bibliothèque qui sont utiles pour le travail en cours.

```
> with(LinearAlgebra,Transpose,HermitianTranspose);
[Transpose, HermitianTranspose]
```

Seules les deux fonctions désignées deviennent disponibles. Ces exemples montrent comment, avec l'option **inplace**, le vecteur V devient son propre transposé puis son transposé conjugué

```
> Transpose(V,inplace);
HermitianTranspose(V,inplace);
```

$$\begin{bmatrix} a \\ b \\ c \end{bmatrix}$$

$$[\bar{a}, \bar{b}, \bar{c}]$$

Il existe également une deuxième forme qui permet l'utilisation d'une fonction sans utiliser l'ordre **with**. Elle est utilisée, soit lorsqu'une fonction n'est invoquée que peu de fois, soit lors de la création d'opérateurs ou de procédures. La forme générale est **Bibliothèque[fonction](Argument_1,...,Argument_n)**. *La fonction n'est pas accessible de façon permanente avec cette forme d'appel.*

```
> restart;
V:=Vector[row]([a,b,c]);
LinearAlgebra[HermitianTranspose](V,inplace):= V;
V:=[a, b, c]
```

$$\begin{bmatrix} \bar{a} \\ \bar{b} \\ \bar{c} \end{bmatrix}$$

Une troisième forme identique à la précédente est utilisable, mais seulement avec les nouvelles bibliothèques telle que **LinearAlgebra** (bibliothèques de modules). Elle utilise l'opérateur ":-" (voir chapitre 19, *Procédures et Modules*).

```
> restart;
V:=Vector([a,b,c],orientation=row);
LinearAlgebra:-HermitianTranspose(V,inplace):=V;
V:=[a, b, c]
```

$$\begin{bmatrix} \bar{a} \\ \bar{b} \\ \bar{c} \end{bmatrix}$$

Quelques exemples de fonctions de la bibliothèque LinearAlgebra

Attention: les exemples donnés ne montrent que quelques possibilités offertes. Il est impératif de compléter cette information avec l'aide en ligne.

```
> restart;
with(LinearAlgebra):
```

1) Résolution d'un système d'équations linéaires

Soit à résoudre $A \cdot x = b$ avec A et b définis par

```
> A,b:=Matrix([[1,2,-1],[0,1,2],[-2,3,1]]),Vector([-3,4,4]);
```

$$A, b := \begin{bmatrix} 1 & 2 & -1 \\ 0 & 1 & 2 \\ -2 & 3 & 1 \end{bmatrix}, \begin{bmatrix} -3 \\ 4 \\ 4 \end{bmatrix}$$

La solution peut être obtenue en écrivant $\mathbf{A}^{\wedge}(-1) \cdot \mathbf{b}$ mais cette opération est *déconseillée car trop peu efficace*. On utilisera toujours la fonction **LinearSolve** qui utilise des algorithmes appropriés dont le résultat est un *vecteur anonyme* solution du système. On rappelle (voir ci-dessus) que l'on a rendu accessible avec **with**, toutes les fonctions de **LinearAlgebra**).

```
> LinearSolve(A,b);
```

$$\begin{bmatrix} -1 \\ 0 \\ 2 \end{bmatrix}$$

Si on veut donner un nom à cette solution, il faudra effectuer une assignation.

```
> x:=LinearSolve(A,b);
```

$$x := \begin{bmatrix} -1 \\ 0 \\ 2 \end{bmatrix}$$

Sans assignation et avec l'option *inplace*, la solution est assignée au vecteur b . C'est un exemple de l'utilité de cette option qui permet d'économiser ainsi l'espace mémoire pour les systèmes de grandes tailles.

```
> LinearSolve(A,b,inplace), b;
```

$$\begin{bmatrix} -1 \\ 0 \\ 2 \end{bmatrix}, \begin{bmatrix} -1 \\ 0 \\ 2 \end{bmatrix}$$

La fonction **LinearSolve** contient de nombreuses options permettant par exemple de choisir la méthode de résolution (voir l'aide en ligne). Elle permet également de résoudre des systèmes matriciels multiples du type $A \cdot X = B$, où B est une matrice ($X_{i,j}$ est la composante i de la colonne X_j solution de $A \cdot X = B$) où B_j désigne la j -ième colonne de B)

```
> A,B:=Matrix([[1,2,-1],[0,1,2],[-2,3,1]]),
      Matrix([[-2,0],[-2,1],[2,0]]);
X:=LinearSolve(A,B);
```

$$A, B := \begin{bmatrix} 1 & 2 & -1 \\ 0 & 1 & 2 \\ -2 & 3 & 1 \end{bmatrix}, \begin{bmatrix} -2 & 0 \\ -2 & 1 \\ 2 & 0 \end{bmatrix}$$

$$X := \begin{bmatrix} -2 & \frac{1}{3} \\ \frac{-2}{5} & \frac{1}{15} \\ \frac{-4}{5} & \frac{7}{15} \end{bmatrix}$$

On trouvera un complément d'information sur ce sujet à la fin du chapitre 14, *Équations ordinaires*. Pour les résolutions de systèmes en mode décimal on lira attentivement les remarques concernant l'option datatype et la fin de ce chapitre.

2) Tests d'égalité entre deux vecteurs ou deux matrices.

La fonction **evalb**, qui teste si une équation est vraie, ne fonctionne pas pour les matrices et les vecteurs. On utilisera la fonction **Equal**. On teste ici la solution du système linéaire précédent

```
> Equal(A.X, B);
true
```

```
> Column(X, 1); # Voir l'aide de Column
[ -2
  -2
  -4 ]
  5
  5
```

La fonction **verify**, fonction générale non spécifique à **LinearAlgebra**, permet aussi cette comparaison

```
> verify(A.X, B, 'Matrix');
true
```

Exercice : que font ces instructions ?

```
> Equal(A.X[1..-1,1], B[1..-1,1]);
Equal(A.Column(X, 1), Column(B, 1));
verify(A.X[1..-1,2], B[1..-1,2], 'Vector');
true
true
true
```

Les deux matrices suivantes ont les mêmes dimensions, les mêmes éléments mais pas le même profil

```
> A, B:=Matrix([[1,2],[2,-1]]),
    Matrix([[1,2],[2,-1]], shape=symmetric);
A, B := [ 1  2 ], [ 1  2 ]
          [ 2  -1 ], [ 2  -1 ]
```

La fonction **Equal** teste, par défaut, les types (**Vector** ou **Matrix**), les dimensions et les éléments (**entries**). La fonction possède plusieurs options de comparaison associées au mot clé **compare** (voir l'aide en ligne). En

général, il sera plus prudent de mettre entre ' ' le mot clé de l'option au cas où celui-ci serait assigné à une valeur. Les deux instructions suivantes sont équivalentes

```
> Equal(A,B);
Equal(A,B,compare='entries');
true
true
```

Maintenant la comparaison porte sur toutes les caractéristiques des deux matrices et comme A et B n'ont pas le même profil

```
> Equal(A,B,compare='all');
false
```

3) Recherche des valeurs propres d'une matrice.

Le résultat de la fonction **Eigenvalues** est, par défaut, un *vecteur* dont les composantes sont les valeurs propres.

```
> A:=Matrix([[-1,1,-2],[1,-2,-1],[-2,-1,1]]);
lambda:=Eigenvalues(A);
```

$$A := \begin{bmatrix} -1 & 1 & -2 \\ 1 & -2 & -1 \\ -2 & -1 & 1 \end{bmatrix}$$

$$\lambda := \begin{bmatrix} -2 \\ \sqrt{7} \\ -\sqrt{7} \end{bmatrix}$$

Attention: l'ordre des valeurs propres et des vecteurs propres n'ayant pas de signification particulière, MAPLE ne donnera pas toujours pour une même matrice, les valeurs propres dans le même ordre (la ré-exécution de la commande précédente pourrait ne pas donner -2 comme première valeur propre).

Avec l'option **output** on peut modifier la forme du résultat. La fonction **Eigenvalues** possède aussi d'autres options. On demande ici de renvoyer les valeurs propres sous la forme d'une liste et on vérifie que le déterminant de la matrice est égal au produit des valeurs propres puis que la somme des valeurs propres est égale à la trace

```
> mul(i,i=Eigenvalues(A,output='list'))=Determinant(A);
add(i,i=Eigenvalues(A,output='list'))=Trace(A);
14 = 14
-2 = -2
```

Noter que les fonctions **add** et **mul** fonctionnent non seulement avec des listes mais aussi avec les composantes des vecteurs

```
> mul(i,i=Eigenvalues(A))=Determinant(A);
add(i,i=Eigenvalues(A))=Trace(A);
14 = 14
-2 = -2
```

Attention: les exercices généralement proposés aux étudiants sont choisis pour avoir des résultats simples et relativement faciles à calculer avec une feuille et un crayon. Ceci ne doit pas faire oublier que dans la majorité des cas, même pour des matrices 3x3, les valeurs propres ont généralement des expressions compliquées

```
> A:=Matrix([[-2,1,-2],[1,-2,-1],[-2,-1,-1]]);
```

$$A := \begin{bmatrix} -2 & 1 & -2 \\ 1 & -2 & -1 \\ -2 & -1 & -1 \end{bmatrix}$$

Cette matrice est *symétrique et à coefficients réels*. Les valeurs propres doivent donc être réelles. Pourtant ceci ne semble pas être le cas (voir le chapitre 6, *Polynôme et Fractions rationnelles*; on a limité l'affichage un peu long à la seconde valeur propre)

```
> lambda:=Eigenvalues(A):
```

```
lambda[2];
```

$$\begin{aligned} & -\frac{1}{12}(548 + 36I\sqrt{107})^{(1/3)} - \frac{19}{3(548 + 36I\sqrt{107})^{(1/3)}} - \frac{5}{3} \\ & + \frac{1}{2}I\sqrt{3} \left(\frac{1}{6}(548 + 36I\sqrt{107})^{(1/3)} - \frac{38}{3(548 + 36I\sqrt{107})^{(1/3)}} \right) \end{aligned}$$

Après une manipulation on obtient bien des expressions réelles (dont la simplicité n'est pas la première caractéristique!)

```
> simplify(evalc(lambda[1]));
simplify(evalc(lambda[2]));
simplify(evalc(lambda[3]));
```

$$\begin{aligned} & \frac{2}{3}\sqrt{19} \cos\left(\frac{1}{3}\arctan\left(\frac{9}{137}\sqrt{107}\right)\right) - \frac{5}{3} \\ & - \frac{1}{3}\sqrt{19} \cos\left(\frac{1}{3}\arctan\left(\frac{9}{137}\sqrt{107}\right)\right) - \frac{5}{3} - \frac{1}{3}\sqrt{3}\sqrt{19} \sin\left(\frac{1}{3}\arctan\left(\frac{9}{137}\sqrt{107}\right)\right) \\ & - \frac{1}{3}\sqrt{19} \cos\left(\frac{1}{3}\arctan\left(\frac{9}{137}\sqrt{107}\right)\right) - \frac{5}{3} + \frac{1}{3}\sqrt{3}\sqrt{19} \sin\left(\frac{1}{3}\arctan\left(\frac{9}{137}\sqrt{107}\right)\right) \end{aligned}$$

Pour les matrices de tailles supérieures à 4x4 il est exceptionnel que MAPLE puisse donner une expression exacte des valeurs propres pour la simple raison qu'**il n'existe pas, dans le cas général, d'expressions symboliques exactes exprimables sous forme de radicaux** (voir le chapitre 6, *Polynômes et Fractions rationnelles*). On construit une matrice dont les éléments sont des entiers générés aléatoirement et compris dans l'intervalle [-3,+3] (voir l'aide en ligne)

```
> A:=RandomMatrix(5,5,generator=-3..3);
```

$$A := \begin{bmatrix} 2 & 0 & -1 & -1 & -1 \\ 3 & -3 & 1 & 3 & 3 \\ 3 & -3 & -1 & 0 & -2 \\ 0 & 2 & -1 & -2 & -3 \\ 1 & -3 & -2 & -1 & 0 \end{bmatrix}$$

Les valeurs propres seront en général données sous la forme symbolique "*Racines du polynôme caractéristique*" ("RootOf" => "Racines de"; l'index=*n* permettant de les distinguer de façon formelle).

> **Eigenvalues(A);**

$$\left[\begin{array}{l} \text{RootOf}(_Z^5 + 4_Z^4 + 2_Z^3 - 76_Z^2 + 125_Z - 20, \text{index} = 1) \\ \text{RootOf}(_Z^5 + 4_Z^4 + 2_Z^3 - 76_Z^2 + 125_Z - 20, \text{index} = 2) \\ \text{RootOf}(_Z^5 + 4_Z^4 + 2_Z^3 - 76_Z^2 + 125_Z - 20, \text{index} = 3) \\ \text{RootOf}(_Z^5 + 4_Z^4 + 2_Z^3 - 76_Z^2 + 125_Z - 20, \text{index} = 4) \\ \text{RootOf}(_Z^5 + 4_Z^4 + 2_Z^3 - 76_Z^2 + 125_Z - 20, \text{index} = 5) \end{array} \right]$$

Pour obtenir les valeurs numériques des valeurs propres il suffira de remplacer ***un*** des éléments de la matrice par sa valeur décimale. On utilise la fonction **simplify** pour transformer des valeurs réelles qui apparaissent sous la forme *a + 0. I* en *a*. Pour ce qui concerne les **calculs numériques**, on se reportera à la fin de ce chapitre.

> **A[1,1]:=evalhf(A[1,1]);**
simplify(Eigenvalues(A));

$$\begin{aligned} A_{1,1} &:= 2. \\ &\left[\begin{array}{l} -3.960391649 + 3.575593657 I \\ -3.960391649 - 3.575593657 I \\ 1.870665457 + 0.6444554648 I \\ 1.870665457 - 0.6444554648 I \\ 0.1794523827 \end{array} \right] \end{aligned}$$

Il existe aussi une fonction, **CharacteristicPolynomial**, permettant de calculer le polynôme caractéristique d'une matrice (attention à l'orthographe). Il suffit de donner en arguments, le nom de la matrice suivi du nom choisi pour la variable (on désassigne au préalable λ qui a été assignée précédemment). On applique la fonction **sort** pour présenter le polynôme sous la forme d'une somme de monômes de degrés décroissants.

> **lambda:='lambda':**
CharacteristicPolynomial(A,lambda);
sort(%);

$$\begin{aligned} &-20. + 125.\lambda - 76.\lambda^2 + 2.\lambda^3 + 4.\lambda^4 + \lambda^5 \\ &\lambda^5 + 4.\lambda^4 + 2.\lambda^3 - 76.\lambda^2 + 125.\lambda - 20. \end{aligned}$$

Si l'on souhaite connaître seulement le **nombre de valeurs propres réelles** on utilisera les suites de Sturm (voir un exemple au paragraphe *Etude d'un exemple* au chapitre 6, *Polynômes et Fractions Rationnelles*).

4) Vecteurs propres d'une matrice.

Les vecteurs propres calculés avec la fonction **Eigenvectors** sont toujours associés aux valeurs propres correspondantes sous la forme d'une suite (sequence) formée du vecteur des valeurs propres et de la matrice des vecteurs propres (les vecteurs propres sont rangés en colonnes).

```
> A:=Matrix([[-1,1,-2],[1,-2,-1],[-2,-1,-1]]);  
lambda,Vp:=Eigenvectors(A);
```

$$A := \begin{bmatrix} -1 & 1 & -2 \\ 1 & -2 & -1 \\ -2 & -1 & -1 \end{bmatrix}$$

$$\lambda, Vp := \begin{bmatrix} -3 \\ -\frac{1}{2} + \frac{1}{2}\sqrt{17} \\ -\frac{1}{2} - \frac{1}{2}\sqrt{17} \end{bmatrix},$$

$$\begin{bmatrix} 1 & -\frac{4(4+\sqrt{17})}{\left(\frac{13}{2} + \frac{3}{2}\sqrt{17}\right)\left(\frac{1}{2} + \frac{1}{2}\sqrt{17}\right)} & -\frac{4(4-\sqrt{17})}{\left(\frac{13}{2} - \frac{3}{2}\sqrt{17}\right)\left(\frac{1}{2} - \frac{1}{2}\sqrt{17}\right)} \\ 0 & -\frac{2\left(\frac{3}{2} + \frac{1}{2}\sqrt{17}\right)}{\frac{13}{2} + \frac{3}{2}\sqrt{17}} & -\frac{2\left(\frac{3}{2} - \frac{1}{2}\sqrt{17}\right)}{\frac{13}{2} - \frac{3}{2}\sqrt{17}} \\ 1 & 1 & 1 \end{bmatrix}$$

Des simplifications peuvent s'avérer nécessaires (voir chapitre 13, *Simplifications, Manipulations*)

```
> Vp:=map(expand,map(rationalize,Vp));
```

$$Vp := \begin{bmatrix} 1 & -1 & -1 \\ 0 & \frac{3}{2} - \frac{1}{2}\sqrt{17} & \frac{3}{2} + \frac{1}{2}\sqrt{17} \\ 1 & 1 & 1 \end{bmatrix}$$

Attention: l'ordre des valeurs propres et des vecteurs propres étant indifférent, MAPLE ne donnera pas toujours pour une même matrice, les valeurs propres dans le même ordre et la ré-exécution de la commande précédente ne donnerait pas nécessairement -3 comme première valeur propre. Mais évidemment le i -ème vecteur propre (la i -ème colonne de la matrice) correspondra toujours à la i -ème composante du vecteur des valeurs propres. On peut vérifier l'égalité $A \cdot (Vp_1) = \lambda_1 Vp_1$.

```
> Equal(A.Column(Vp,1),lambda[1]*Column(Vp,1));
```

true

Les vecteurs propres sont, par définition, définis à une constante multiplicative près. Ils ne sont pas, contrairement aux calculs effectués numériquement (voir plus loin), normalisés à l'unité (norme euclidienne; voir l'aide en ligne)

```
> Norm(Column(Vp,1),2); # 2 demande le calcul avec la norme euclidienne  
√2
```

5) Application d'un opérateur sur les éléments d'un vecteur ou d'une matrice.

Cherchons maintenant à vérifier $A \cdot (Vp_2) = \lambda_2 Vp_2$

```
> Equal(A.Column(Vp,2),lambda[2]*Column(Vp,2));  
false
```

????? En examinant les résultats des 2 opérations on comprend l'origine du problème

```
> A.Column(Vp,2),lambda[2]*Column(Vp,2);
```

$$\begin{bmatrix} \frac{1}{2} - \frac{1}{2}\sqrt{17} \\ -5 + \sqrt{17} \\ -\frac{1}{2} + \frac{1}{2}\sqrt{17} \end{bmatrix}, \begin{bmatrix} \frac{1}{2} - \frac{1}{2}\sqrt{17} \\ \left(-\frac{1}{2} + \frac{1}{2}\sqrt{17}\right)\left(\frac{3}{2} - \frac{1}{2}\sqrt{17}\right) \\ -\frac{1}{2} + \frac{1}{2}\sqrt{17} \end{bmatrix}$$

Il suffit donc d'appliquer avec **map** l'opérateur de développement **expand** sur les composantes du deuxième vecteur

```
> Equal(A.Column(Vp,2),map(expand,lambda[2]*Column(Vp,2)));  
true
```

On peut aussi écrire

```
> verify(A.Column(Vp,2),lambda[2]*Column(Vp,2),'Vector(expand)');  
true
```

Un autre exemple : cherchons à diagonaliser la matrice **A**. Pour éviter un affichage trop encombrant on n'a présenté que la seconde colonne de la matrice. Il ne semble pas évident que la troisième composante soit nulle comme il se doit

```
> Column(Vp^(-1).A.Vp,2);
```

$$\begin{bmatrix} 0 \\ \frac{1}{34}(3 + \sqrt{17})\sqrt{17} + \frac{2}{17}\sqrt{17} + \left(-\frac{1}{34}(3 + \sqrt{17})\sqrt{17} + \frac{2}{17}\sqrt{17}\right)\left(\frac{3}{2} - \frac{1}{2}\sqrt{17}\right) \\ \frac{1}{34}(-3 + \sqrt{17})\sqrt{17} - \frac{2}{17}\sqrt{17} + \left(-\frac{1}{34}(-3 + \sqrt{17})\sqrt{17} - \frac{2}{17}\sqrt{17}\right)\left(\frac{3}{2} - \frac{1}{2}\sqrt{17}\right) \end{bmatrix}$$

Ici aussi, pour avoir le résultat attendu, il faut développer les produits en appliquant l'opérateur **expand**

```
> map(expand,Vp^(-1).A.Vp);
```

$$\begin{bmatrix} -3 & 0 & 0 \\ 0 & -\frac{1}{2} + \frac{1}{2}\sqrt{17} & 0 \\ 0 & 0 & -\frac{1}{2} - \frac{1}{2}\sqrt{17} \end{bmatrix}$$

Exercice : que vérifie-t-on ?

```
> Equal(% , Matrix(lambda, shape=diagonal));
true
```

6) Les fonctions Map et Zip

La bibliothèque possède aussi une fonction **Map** qui est formellement équivalente à une **instruction qui n'est pas acceptée** par MAPLE:

```
> map(opérateur, Matrice ou Vecteur, inplace);
```

A chaque application de **Map** c'est la matrice elle-même qui est remplacée par le résultat de l'action de l'opérateur sur chaque terme de la matrice

```
> M:=Matrix([[1,2],[2,5]]);
Map(sqrt,M):
M;
```

$$M := \begin{bmatrix} 1 & 2 \\ 2 & 5 \end{bmatrix}$$

$$\begin{bmatrix} 1 & \sqrt{2} \\ \sqrt{2} & \sqrt{5} \end{bmatrix}$$

L'opérateur peut être défini par l'utilisateur (voir le chapitre 7, *Fonctions*).

```
> Map(x->x^3+z,M):
M;
```

$$\begin{bmatrix} 1+z & 2\sqrt{2}+z \\ 2\sqrt{2}+z & 5\sqrt{5}+z \end{bmatrix}$$

Ici chaque élément de M sera évalué numériquement avec 4 chiffres significatifs. L'écriture $x \rightarrow \text{evalf}[4] \dots$ signifie que à x on fait correspondre l'évaluation décimale de x avec 4 chiffres significatifs (voir le chapitre 7, *Fonctions*)

```
> M:=Map(sqrt,(Matrix([[1,2],[2,5]])));
Map(x->evalf[4](x),M):
M;
```

$$M := \begin{bmatrix} 1 & \sqrt{2} \\ \sqrt{2} & \sqrt{5} \end{bmatrix}$$

$$\begin{bmatrix} 1. & 1.414 \\ 1.414 & 2.236 \end{bmatrix}$$

L'opérateur **Zip** permet d'appliquer un opérateur à deux variables sur deux matrices. Ici, x prendra successivement la valeur de chaque élément de M , y la valeur de l'élément correspondant de N et l'opérateur calculera l'opération définie et le résultat sera placé dans une nouvelle matrice à la place correspondante. Cette fonction possède aussi l'option *inplace* (avec cet option le résultat aurait été rangé dans M).

```
> M,N:=Matrix([[1,2],[2,4]]),Matrix([-1,0],[-2,3]);
Zip((x,y)->x*exp(y),M,N);
```

$$M, N := \begin{bmatrix} 1 & 2 \\ 2 & 4 \end{bmatrix}, \begin{bmatrix} -1 & 0 \\ -2 & 3 \end{bmatrix}$$

$$\begin{bmatrix} e^{(-1)} & 2 \\ 2e^{(-2)} & 4e^3 \end{bmatrix}$$

Autre exemple : on calcule le *produit de Hadamard* de M et de N

```
> Zip((x,y)->x*y,M,N);
```

$$\begin{bmatrix} -1 & 0 \\ -4 & 12 \end{bmatrix}$$

V - Calculs numériques

Bien des opérations de l'algèbre linéaire ne peuvent être réalisées sous forme symbolique et doivent être effectuées à l'aide de calculs numériques (résolution de grands systèmes linéaires, calculs de valeurs propres et vecteurs propres pour des matrices de tailles supérieures à 4×4 , etc...). MAPLE, depuis la version 6, sait réaliser de telles opérations avec efficacité grâce à l'intégration de la bibliothèque numérique NAG dans la librairie **LinearAlgebra** (*Numerical Algorithm Group* (<http://www.nag.co.uk>) avec qui Waterloo-Maplesoft a créé une collaboration). *La règle est*: si les éléments sont tous de *type numeric* (-1/3 est du type *numeric* mais $\sqrt{3}$ ne l'est pas; voir chapitre 2), il suffit qu'un seul d'entre eux (ici 1.0) soit exprimé sous forme décimale pour que tous les calculs soient effectués numériquement.

```
> restart;
with(LinearAlgebra);
A:=Matrix([[1.0,2],[-1/3,0]]);
```

$$A := \begin{bmatrix} 1.0 & 2 \\ -\frac{1}{3} & 0 \end{bmatrix}$$

Attention: le datatype de A défini par défaut est tel que tout type d'élément est accepté et ceci n'est pas sans conséquence (voir ci-dessous)

```
> MatrixOptions(A,datatype);
anything
```

1) Arithmétique câblée

Toute opération impliquant A et ne faisant intervenir que des éléments de *type numeric dont un au moins est décimal* sera alors effectuée numériquement. *Noter* comment est récupérée ici la matrice des vecteurs propres Vp (deuxième élément de la suite des résultats rendus par **Eigenvectors** => [2]) en ignorant le vecteur des valeurs propres

```

> A_1:=A^(-1);
V:=A.<3,-1/2>;
Vp:=LinearAlgebra[Eigenvectors](A)[2];
A_1 := 
$$\begin{bmatrix} 0. & -3. \\ 0.5000000000000000 & 1.500000000000000 \end{bmatrix}$$

V := 
$$\begin{bmatrix} 2. \\ -1. \end{bmatrix}$$

Vp := [[0.925820099772551419 + 0. I, 0.925820099772551419 + 0. I], [
-0.231455024943137882 + 0.298807152333598390 I,
-0.231455024943137882 - 0.298807152333598390 I]]

```

Les résultats sont affichés avec 18 chiffres indiquant que les éléments sont stockés en mémoire sur 8 octets et que les calculs ont été effectués à l'aide de l'*arithmétique câblée du processeur*. Il en est de même pour le vecteur V malgré un affichage simplifié dû à des parties décimales nulles. La matrice des vecteurs propres Vp contenant des résultats complexes, *tous* les résultats ont le *type complex* avec des parties imaginaires nulles pour les réels

```

> MatrixOptions(A_1,datatype);
VectorOptions(V,datatype);
MatrixOptions(Vp,datatype);
float8
float8
complex8

```

ou encore (on notera la syntaxe qui place entre deux caractères ' les mots clés *Vector* et *Matrix*, évitant ainsi à l'analyseur syntaxique de les interpréter comme des appels aux fonctions correspondantes)

```

> type(V,'Vector'(float[8]));
type(Vp,'Matrix'(complex[8]));
true
true

```

L'application de **simplify** fait apparaître une écriture standard avec le nombre de chiffres significatifs fixé par la variable **Digits** (ici le défaut, i.e. 10). Cependant le *datatype* redevient *anything*.

```

> simplify(Vp);
MatrixOptions(% ,datatype);

$$\begin{bmatrix} 0.9258200998 & 0.9258200998 \\ -0.2314550249 + 0.2988071523 I & -0.2314550249 - 0.2988071523 I \end{bmatrix}$$

anything

```

Contrairement au calculs effectués exactement (voir plus haut), les vecteurs propres sont normalisés à l'unité

```

> Norm(Vp[1..-1,1],2);
Norm(Vp[1..-1,2],2);
1.
1.

```

Attention: rappelons que **type constant** n'implique pas nécessairement **type numeric** alors que la réciproque est toujours vraie

```
> type(sqrt(3), numeric);
type(sqrt(3), constant);
```

false
true

Aussi, la présence du symbole $\sqrt{3}$ interdit les opérations numériques approchées.

```
> A:=Matrix([[1.0,2.5],[sqrt(3),2.]]);
A^(-1);
```

$$A := \begin{bmatrix} 1.0 & 2.5 \\ \sqrt{3} & 2. \end{bmatrix}$$

$$A^{-1} := \begin{bmatrix} -\frac{2.}{-2.0 + 2.5\sqrt{3}} & \frac{2.5}{-2.0 + 2.5\sqrt{3}} \\ \frac{\sqrt{3}}{-2.0 + 2.5\sqrt{3}} & -\frac{1.0}{-2.0 + 2.5\sqrt{3}} \end{bmatrix}$$

Il faut donc utiliser une approximation décimale de $\sqrt{3}$ pour effectuer un calcul purement numérique.

Attention à la précision utilisée pour définir les termes d'une matrice: calculons d'abord l'expression exacte de l'inverse de A en remplaçant les valeurs décimales des éléments par leurs valeurs exactes

```
> As:=Matrix([[1,5/2],[sqrt(3),2]]);
As_1:=As^(-1);
```

$$As := \begin{bmatrix} 1 & \frac{5}{2} \\ \sqrt{3} & 2 \end{bmatrix}$$

$$As^{-1} := \begin{bmatrix} -\frac{4}{-4 + 5\sqrt{3}} & \frac{5}{-4 + 5\sqrt{3}} \\ \frac{2\sqrt{3}}{-4 + 5\sqrt{3}} & -\frac{2}{-4 + 5\sqrt{3}} \end{bmatrix}$$

Convertissons les éléments exacts de A^{-1} en valeurs décimales avec 18 chiffres significatifs

```
> As_1n:=evalf[18](As_1);
As_In :=
```

-0.858322307650466884	1.07290288456308360
0.743328923060186752	-0.429161153825233442

Maintenant effectuons un calcul numérique direct en remplaçant $\sqrt{3}$ par une valeur décimale

```
> An:=Matrix([[1,5/2],[sqrt(3.),2]]);
A_1n:=An^(-1);
```

$$An := \begin{bmatrix} & 1 & \frac{5}{2} \\ & 1.732050808 & 2 \end{bmatrix}$$

$$A_In := \begin{bmatrix} -0.858322307253447713 & 1.07290288406680955 \\ 0.743328922901379086 & -0.429161153626723856 \end{bmatrix}$$

La différence entre les deux calculs montre des écarts relatifs trop grands relativement au nombre de chiffres significatifs

> **A_1n - As_1n;**

$$\begin{bmatrix} 3.97019195297332317 \cdot 10^{-10} & -4.96274132899543474 \cdot 10^{-10} \\ -1.58807633710011941 \cdot 10^{-10} & 1.98509597648666158 \cdot 10^{-10} \end{bmatrix}$$

Ceci est évidemment dû à la précision trop faible donnée à l'évaluation décimale de $\sqrt{3}$ qui a été définie avec 10 chiffres significatifs (la valeur par défaut de **Digits**). ***On sera donc attentif à utiliser une précision suffisante*** soit avec **evalhf**, soit avec, disons, **evalf[18](...)**.

> **An:=Matrix([[1,5/2],[evalhf(sqrt(3)),2]]);**
A_1n:=An^(-1);

$$An := \begin{bmatrix} & 1 & \frac{5}{2} \\ & 1.73205080756887720 & 2 \end{bmatrix}$$

$$A_In := \begin{bmatrix} -0.858322307650467130 & 1.07290288456308391 \\ 0.743328923060186830 & -0.429161153825233564 \end{bmatrix}$$

On obtient alors des résidus conformes à la précision des éléments de la matrice initiale.

> **A_1n - As_1n;**

$$\begin{bmatrix} -2.22044604925031308 \cdot 10^{-16} & 2.22044604925031308 \cdot 10^{-16} \\ 1.11022302462515654 \cdot 10^{-16} & -1.11022302462515654 \cdot 10^{-16} \end{bmatrix}$$

Erreurs numériques

Tout calcul numérique conduit à des erreurs de ce type et dans certaines circonstances elles peuvent devenir catastrophiques. Un cas pathologique bien connu est, par exemple, celui de la résolution de systèmes linéaires dont la matrice a la forme d'une matrice de Hilbert (voir l'aide en ligne de **LinearAlgebra[HilbertMatrix]**). Résolvons exactement un tel système de dimensions 20x20

> **x_exact:=LinearSolve(HilbertMatrix(20),Vector(20,1));**

Résolvons le même système numériquement en donnant des valeurs décimales au second membre

> **x_num:=LinearSolve(HilbertMatrix(20),Vector(20,evalhf(1)));**

Cherchons maintenant l'erreur absolue relative maximale entre les composantes exactes et celles calculées numériquement (exprimée en %).

```
> max(op(convert(zip((x,y)->abs((x-y)/x),x_exact,x_num),list)))*100;
100.0017465
```

A un tel niveau, ce n'est plus une erreur mais un calcul faux ! La difficulté vient du conditionnement très grand (gigantesque) de la matrice (*Condition Number*; voir l'aide en ligne)

```
> evalf(ConditionNumber(HilbertMatrix(20)));
6.283579684 1028
```

A comparer avec une valeur typique pour une matrice de même taille générée aléatoirement

```
> evalf(ConditionNumber(RandomMatrix(20,20,generator=-100..100)));
2611.697384
```

Heureusement les problèmes ne sont que rarement aussi délicats à traiter, mais c'est une éventualité qu'il faut toujours avoir présente à l'esprit.

Forme du datatype d'un résultat numérique

MAPLE sait adapter le stockage des résultats sous une forme appropriée à condition toutefois de disposer de l'information *a priori*. Construisons par exemple une matrice hermitienne (hermitique) dont les valeurs propres doivent être réelles

```
> Ah:=Matrix([[3.0,-4+I],[-4-I,2]],datatype=complex[8]);
Ah := 
$$\begin{bmatrix} 3. + 0. I & -4. + 1. I \\ -4. - 1. I & 2. + 0. I \end{bmatrix}$$

```

Pourtant les résultats des valeurs propres sont stockés sous forme complexe avec des valeurs imaginaires non nulles mais très petites, dues au calcul approché

```
> Eigenvalues(Ah);
VectorOptions(% ,datatype);

$$\begin{bmatrix} 6.65331193145903743 - 1.04079876148095112 \cdot 10^{-16} I \\ -1.65331193145903699 + 1.18029130358854576 \cdot 10^{-16} I \end{bmatrix}$$

complex8
```

Mais si la matrice est convertie en matrice avec profil hermitien

```
> Ah:=Matrix(Ah,shape=hermitian):
```

Eigenvalues sachant *a priori* que la matrice est hermitienne utilisera un algorithme approprié et rangera les valeurs propres dans un vecteur avec un datatype adapté à des réels

```
> Eigenvalues(Ah);
VectorOptions(% ,datatype);

$$\begin{bmatrix} -1.65331193145903654 \\ 6.65331193145903832 \end{bmatrix}$$

float8
```

2) Arithmétique émulée

Il existe une possibilité supplémentaire pour les calculs numériques lié au mot clé à valeur logique

UseHardwareFloat. Par défaut il vaut *deduced* indiquant que MAPLE choisira d'effectuer les calculs en fonction de la nature du *datatype* des objets à traiter.

Attention: ces indications ne s'appliquent (pour cette version de Maple et les précédentes) qu'aux calculs sur des structures de type *rtable* (c'est-à-dire *Matrix*, *Vector* ou *Array*).

```
> UseHardwareFloats;
                                         deduced

> Ah;
MatrixOptions(Ah, shape, datatype);
lambda:=Eigenvalues(Ah);
VectorOptions(lambda, datatype);

$$\begin{bmatrix} 3. + 0. I & -4. + 1. I \\ -4. - 1. I & 2. + 0. I \end{bmatrix}$$

[hermitian], complex8


$$\lambda := \begin{bmatrix} -1.65331193145903654 \\ 6.65331193145903832 \end{bmatrix}$$

float8
```

Si par contre on donne au mot clé **UseHardwareFloats** la valeur logique *false* les calculs numériques seront effectués avec une arithmétique émulée (calculs arithmétiques réalisés par programme) comme ils le sont, par défaut, pour les calculs décimaux réalisés avec *evalf*. Ceci permet en particulier de traiter les calculs avec une nombre de chiffres significatifs fixé avec le mot clé **Digits**. Evidemment il ne sagit ici que d'un exemple et il faut avoir de bonnes raisons pour choisir une telle précision. On conçoit que les temps de calcul peuvent devenir rapidement prohibitifs pour des matrices de grandes tailles

```
> UseHardwareFloats:=false;
Digits:=50;
lambda:=Eigenvalues(Ah);
                                         UseHardwareFloats := false
                                         Digits := 50


$$\lambda := \begin{bmatrix} -1.6533119314590374262921313724537460051161071244777 \\ 6.6533119314590374262921313724537460051161071244780 \end{bmatrix}$$

```

Le type de stockage est alors *sfloat* (pour software floating number)

```
> VectorOptions(lambda, datatype);
                                         sfloat
```

Juste à titre de rappel (voir la fin du chapitre 1)

```
> identify(lambda);

$$\begin{bmatrix} \frac{5}{2} - \frac{1}{2}\sqrt{69} \\ \frac{5}{2} + \frac{1}{2}\sqrt{69} \end{bmatrix}$$

```


5 - Tables et Tableaux

Les tables

Une table est une structure de données formée par une liste indexée. C'est la forme d'organisation la plus générale disponible pour l'utilisateur (et dont MAPLE fait un usage intensif pour son fonctionnement de façon invisible).

Attention : une table est une structure de données de type **table** différente de celle des vecteurs, des matrices et des tableaux qui sont de type **rtable**.

Dans l'exemple suivant on construit, à l'aide de la fonction **table**, une table associant trois longueurs d'onde à trois couleurs. Les index de la table sont ici les noms *rouge*, *vert* et *bleu*.

```
> restart;
L_onde:=table([rouge=0.8,vert=0.65,bleu=0.4]);
L_onde := table([bleu = 0.4, rouge = 0.8, vert = 0.65])
```

On accède aux valeurs de la table avec la syntaxe

```
> L_onde[rouge], (L_onde[vert]+L_onde[bleu])/2;
0.8, 0.5250000000
```

Il n'est en réalité pas nécessaire de passer par la fonction **table** pour créer une table. En effet une simple assignation avec la syntaxe *Nom_table*[*index*]:=... crée automatiquement la table *Nom_table*. Si elle existe déjà, on ajoute ou on modifie ainsi un élément. Cependant cette méthode ne permet pas de définir une fonction d'indexation, notion que nous définirons plus tard. La commande suivante crée une nouvelle table à un élément indexé par le nombre 0.

```
> Tab[0]:=1.4356; # Crée une table contenant un élément
Tab0 := 1.4356
```

Pour *ajouter des éléments* à une table (*ou les modifier*) il suffira d'écrire

```
> L_onde[jaune]:=0.55;
Tab[10]:=1.3472e-1;

L_ondejaune := 0.55
Tab10 := 0.13472
```

Les index et les valeurs d'une table ne sont pas nécessairement des noms ou des valeurs numériques et peuvent être n'importe quels objets valides de MAPLE

```
> Tab[interv]:=0..sin(Pi/4);
Tab[a*x+b*y+c*z^2]:=[x,y,z];

Tabinterv := 0 ..  $\frac{1}{2}\sqrt{2}$ 
Taba x + b y + c z2 := [x, y, z]
```

Pour *enlever un élément* on utilisera la syntaxe de désassignation

```
> L_onde[jaune]:='L_onde[jaune]':
```

Attention: assigner le mot clé **NULL** (voir chapitre 3, § *Ensembles*) à un élément d'une table ne le supprime pas de la table.

```
> L_onde[rouge]:=NULL;
```

$L_{\text{onde}}_{\text{rouge}} :=$

L'élément existe toujours, vaut **NULL** et MAPLE n'affiche rien si on demande la valeur de cet élément. Si l'élément n'avait pas existé, MAPLE aurait répondu par le nom symbolique $L_{\text{onde}}_{\text{rouge}}$, mais pas en n'affichant rien.

```
> L_onde[rouge];
```

Rétablissementsons la valeur initiale

```
> L_onde[rouge]:=0.81;
```

Exercice : cet exemple illustre de degré de généralité des structures **table**. Analysez la syntaxe des commandes (un peu confuses!) et les résultats

```
> L_onde[`Un exemple`]:=Tab;
  exp(-L_onde[`Un exemple`][10]);
L_onde[`Un exemple`]:='L_onde[`Un exemple`]':
 $L_{\text{onde}}_{\text{Un exemple}} := \text{Tab}$ 
0.8739605864
```

Exercice: analysez (très attentivement !) ces instructions et expliquez les opérations effectuées et les résultats. Remarquez aussi comment un **index de type "suite"** doit être écrit entre parenthèses: (r,s)

```
> x:=2:
T:=table([3=x,x^2-1=6,8=x,{s,t}=3,(r,s)=-1]):
```

```
> x:='x':
T[x^2-1];
T[3];
```

$$\begin{aligned} & T \\ & x^2 - 1 \\ & 6 \end{aligned}$$

Exercice: analysez cette instruction et expliquez le résultat.

```
> T[T[{s,t}]^2+T[r,s]];

$$2$$

```

Si un **index** est un ensemble, MAPLE testera l'équivalence des ensembles (celui demandé et ceux des index de la table) pour choisir l'index lors d'un appel sans tenir compte de l'ordre des éléments

Exercice : examiner ces commandes

```
> T:=table([x^2-1=3,2=x,{t,u}=4,(r,s)=-1]):
T[{u,t}];
```

$$4$$

```
> E:={v,x-1}:
T[{t,u,v,x-1,u,x-1} minus E];
```

Ces divers exemples montrent que l'utilisation des tables est très souple et d'un usage très général auquel il faut penser quand on veut traiter un problème manipulant des objets de natures différentes. Les tables présentent l'avantage d'avoir des index de nature quelconque (penser aussi aux variables structurées créées avec **Record**, voir chapitre 19, § *Modules*). De plus la taille d'une table n'est pas définie a priori et il est possible d'y ajouter indéfiniment des éléments. Sa création et son "remplissage" peut s'effectuer sans ordre particulier et au fur et à mesure des besoins.

Contenus des tables

On peut afficher le contenu de toute la table soit avec la fonction **eval**

```
> eval(Tab);
```

$$\text{table}\left(\left[0 = 1.4356, 10 = 0.13472, a x + b y + c z^2 = [x, y, z], interv = 0 .. \frac{1}{2} \sqrt{2} \right] \right)$$

soit avec la fonction **print**

```
> print(T);
```

$$\text{table}([2 = x, x^2 - 1 = 3, (r, s) = -1, \{t, u\} = 4])$$

La fonction **indices** permet d'obtenir les **index** d'une table sous la forme d'une **suite de listes** (même si ces listes ne contiennent qu'un élément)

```
> indices(T);
indices(L_onde);
```

$$\begin{aligned} & [2], [x^2 - 1], [r, s], [\{t, u\}] \\ & [bleu], [rouge], [vert] \end{aligned}$$

De même la fonction **entries** permet d'obtenir les **valeurs** d'une table sous la forme d'une **suite de listes**.

```
> entries(Tab);
```

$$[1.4356], [0.13472], [[x, y, z]], \left[0 .. \frac{1}{2} \sqrt{2} \right]$$

On peut toujours convertir les valeurs d'une table en une liste ou un ensemble

```
> convert(T,list);
convert(Tab,set);
```

$$[x, 3, -1, 4]$$

$$\left\{ [x, y, z], 1.4356, 0.13472, 0 .. \frac{1}{2} \sqrt{2} \right\}$$

Les options d'indexation

Les tables peuvent avoir des options d'indexation prédéfinies que l'on construit avec le premier argument **facultatif** de la fonction **tab**.

L'option d'indexation **symmetric** (attention, avec deux m !)

On définit ici une table avec cette option

```
> L2_onde:=table(symmetric,[ (1,rouge)=0.79, (2,rouge)=0.81]);
```

```
L2_onde := table(symmetric, [(2, rouge) = 0.81, (1, rouge) = 0.79])
```

Examinons les deux termes

```
> L2_onde[1,rouge], L2_onde[rouge,1];  
0.79, 0.79
```

L'option *symmetric* a permis de définir *implicitement* le terme $L2_{onde,rouge}, 1$ qui n'était pas dans les arguments donnés à la fonction **table** (de même pour $L2_{onde,rouge}, 2$). La notion de symétrie est en fait plus générale et s'étend à des tables ayant des index de plus de deux entrées (suites de plus de deux éléments). Si l'on peut, par permutations des entrées, obtenir un index identique à un index défini, ils sont considérés comme égaux.

```
> Sym:=table(symmetric,[ (1,2,3)=x,(1,2..3,sin(t),u,v)=y]):  
Sym[2,3,1];  
Sym[u,2..3,v,1,sin(t)];  
Sym[1,3];  
x  
y  
Sym1, 3
```

Et plus généralement encore cette identification se fait formellement pour des éléments non définis

```
> Sym[i,j,k];  
evalb(Sym[i,j,k]=Sym[i,k,j]);  
evalb(Sym[i,j,k]=Sym[i,j,j]);  
Symi, j, k  
true  
false
```

Avec ce mode d'indexation les termes diagonaux peuvent être définis

```
> Sym[x,x]:=-3;  
Symx, x := -3
```

L'option d'indexation antisymmetric

Elle définit une table pour laquelle $T_{i,j} = -T_{j,i}$ et plus généralement

$T_{i,j,k,\dots} = (-1)^n T_{i',j',k',\dots}$ où n est le nombre de permutations nécessaires pour passer de la suite (i',j',k',\dots) à la suite (i,j,k,\dots) .

```
> Anti_Sym:=table(antisymmetric,[ (1,2,3)=x^2-1,(1,2,t,u,v)=y]):  
Anti_Sym[2,3,1];  
Anti_Sym[3,2,1];  
x2 - 1  
-x2 + 1
```

La définition vaut aussi pour les éléments non définis

```
> evalb(Anti_Sym[a,b,c]=-Anti_Sym[b,a,c]);  
true
```

Les éléments "diagonaux" ne peuvent pas être définis

```
> Anti_Sym[x,x]:= -3;  
Error, cannot assign non-zero to diagonal entry of antisymmetric
```

et sont implicitement égaux à 0.

```
> Anti_Sym[x,x];  
0
```

Il va de soi que le signe n'a de sens que si l'objet associé est susceptible d'être multiplié par -1. C'est le cas par exemple de cette liste

```
> Anti_Sym[3,2,4]:=[1,2,-3,a,x,y];  
Anti_Sym[3,4,2];  
Anti_Sym3,2,4 := [1, 2, -3, a, x, y]  
[-1, -2, 3, -a, -x, -y]
```

Rappel: une liste peut en effet être multipliée par un scalaire (voir chapitre 3)

```
> -2*[1,2,-3,a,x,y];  
[-2, -4, 6, -2 a, -2 x, -2 y]
```

Mais évidemment...

```
> Anti_Sym[0,1]:="abc";  
Anti_Sym[1,0];  
Anti_Sym0,1 := "abc"
```

```
Error, invalid terms in product
```

L'option d'indexation diagonal

Elle définit une table dont les éléments associés à des index ayant des entrées identiques prennent des valeurs spécifiées. Les éléments "non diagonaux" sont tous définis implicitement comme valant 0. Un élément diagonal non spécifié est indéterminé et s'affiche sous la forme d'un nom indexé.

```
> Diag:=table(diagonal,[ (3,3,3)=2,(y,y)=x^2-1,(z,z)=0]):  
Diag[y,y];  
Diag[2,3];  
Diag[1,2,3];  
Diag[u,u,u,u];  
x2 - 1  
0  
0  
Diagu, u, u, u
```

A la différence d'une même table construite sans cette option d'indexation, on ne peut pas changer un élément non diagonal

```
> Diag[4,3,x]:=6;  
Error, cannot assign to the off-diagonal elements of a diagonal array
```

L'option d'indexation identity

Elle ressemble à la précédente sinon que les éléments "diagonaux" valent tous 1 par définition. Il est donc

inutile de spécifier les éléments. On ne peut rien assigner aux éléments d'une telle table.

```
> Id:=table(identity):  
Id[2,2],Id[2,3];  
1, 0
```

L'option d'indexation sparse (clairsemée)

Elle permet de définir une table qui renvoie 0 quand on l'invoque pour des valeurs d'indices non définis. On crée ainsi, en quelque sorte, une table virtuellement infinie dont tous les éléments sont nuls sauf ceux qui sont spécifiés.

```
> Sp:=table(sparse,[3=2,y=x^2-1,z=Pi,(c,s)=cos+sin]):  
Sp[3],Sp[z];  
Sp[t-1];  
  
2, π  
0
```

Exercice: expliquez ce résultat

```
> Sp[c,s](Sp[z]/4);  
√2
```

Création de fonctions d'indexation

Il existe aussi la possibilité de construire des fonctions d'indexation spéciales grâce à des procédures (des fonctions programmées par l'utilisateur). Nous n'étudierons pas ces méthodes dont on trouvera un exemple dans l'aide en ligne si nécessaire.

```
> ?indexfcn
```

Fonction copy

Rappelons par un exemple ce que nous savons déjà à propos de l'évaluation: on assigne à b l'évaluation de a

```
> a:=1;  
b:=a;  
a := 1  
b := 1
```

Si on modifie a on ne change pas b

```
> a:=2;  
b;  
a := 2  
1
```

Le comportement des tables et des tableaux est différent (il est par contre identique à celui des matrices, des vecteurs et des tableaux). Appliquons à une table le même schéma d'assiguation

```
> Tab_1:=table();# On crée une table vide  
Tab_2:=Tab_1; # On assigne Tab_1 à Tab_2  
Tab_1 := table([])
```

```

Tab_2 := Tab_1

> Tab_1[2]:=3;    # On modifie Tab_1 en créant un élément
                  Tab_12 := 3

```

Qu'en est-il des contenus "des deux" tables ? (on peut utiliser indifféremment **eval** ou **print**)

```

> print(Tab_1);
                  table([2 = 3])

> eval(Tab_2);
                  table([2 = 3])

```

Le fait d'avoir apporté une modification a affecté *Tab_1 et Tab_2*. La raison vient de ce que quand on écrit **Tab_2:=Tab_1**, on demande seulement aux deux identificateurs de pointer sur *la même table*. Pour obtenir deux tables indépendantes il faut dupliquer le contenu de la table avec la fonction **copy** et écrire

```

> Tab_1:=table():
  Tab_2:=copy(Tab_1);
                  Tab_2 := table([])

> Tab_1[x]:=3;
                  Tab_1x := 3

> Tab_1=eval(Tab_1);
                  Tab_1 = table([x = 3])

> Tab_2=eval(Tab_2);
                  Tab_2 = table([])

```

Tab_1 et Tab_2 sont maintenant indépendantes et leurs contenus sont différents. On peut comprendre que ce changement de comportement introduit par les concepteurs de MAPLE ait été motivé par un soucis d'économiser la mémoire, les tables étant souvent des structures contenant beaucoup de données. L'usage explicite de **copy** incite l'utilisateur à la prudence.

Noms indicés

On a pu remarquer qu'à l'affichage des résultats, MAPLE écrit les éléments des tables en mettant en indice le nom de l'index. Ceci permet d'utiliser une notation familière de nom indicé.

Rappel: un nom indicé **non assigné** n'est pas un élément d'une table mais possède simplement le type **indexed** ("nom indexé") sans posséder le type **symbol**.

```

> mu[0];
  whattype(mu[0]);
  type(mu[0],name);
  type(mu[0],symbol);
                  μ₀
                  indexed
                  true
                  false

```

Dès qu'il reçoit une assignation il devient un élément d'une table

```
> mu[0]:=1;
```

```

whattype(eval(mu));
 $\mu_0 := 1$ 
table

```

Si elle est pratique, cette notation indicée présente des inconvénients. Par exemple, la définition d'une variable v à partir de v_0 provoque une récursion car v_0 se définit à partir de v lui-même.

```

> nu:=nu[0]+y;
Error, recursive assignment

```

Ici, v_0 étant assigné,

```

> nu[0]:=1;
whattype(eval(nu));

 $v_0 := 1$ 
table

```

son évaluation dans l'expression assignée à v évite la récursion, mais provoque la disparition de la table v (et donc de v_0) au profit du symbole v

```

> nu:=nu[0]+z;
nu,nu[0];
whattype(eval(nu));

 $v := 1 + z$ 
 $1 + z, (1 + z)_0$ 
+

```

Opérandes et type des objets *table*

Opérandes des tables

Au **nom** d'une table n'est associée qu'une seule opérande, c'est à dire la structure **table**

```

> nops(L_onde);
op(L_onde);

1





```

Par contre la structure de la table a deux opérandes. La première opérande est la fonction d'indexation

```

> nops(eval(L2_onde)), nops(op(L2_onde)); # Ecritures identiques
op(1,op(L2_onde));
2, 2
symmetric

```

Attention: lorsque la fonction d'indexation n'a pas été définie, elle possède l'état particulier **NULL** et MAPLE n'affiche rien (voir chapitre 3, § *Ensembles*, §§ *Le mot clé NULL*). C'est le cas de L_{onde}

```
> op(1,op(L_onde));
```

Pourtant cette structure **table** a bien deux opérandes

```
> nops(op(L_onde));  
2
```

L'opérande 2 est la liste des éléments de la table

```
> op(2,op(L_onde));  
op(2,op(L2_onde));  
[bleu = 0.4, rouge = 0.81, vert = 0.65]  
[(2, rouge) = 0.81, (1, rouge) = 0.79]
```

Type des tables

Comme nous l'avons déjà précisé au chapitre 2, *Eléments sur la structure des expressions*, il existe aussi une opérande d'ordre 0 qui définit la nature de l'objet par sa fonction et qui est aussi renvoyée par la fonction **whattype**

```
> op(0,eval(L_onde));  
watttype(op(L_onde));  
table  
table
```

Attention: l'exemple suivant applique **whattype** sur le **nom**, d'où le résultat

```
> whattype(L_onde);  
symbol
```

On note que **type** ne demande pas d'évaluer obligatoirement le nom

```
> type(op(L_onde),table), type(L_onde,table);  
true, true
```

Rappel: une matrice ou un vecteur (ou un tableau, voir paragraphe suivant) n'ont pas le type **table**, mais le type **rtable**

```
> M:=Matrix([[a,b],[c,d]]): v:=Vector([a,b,c]):  
type(M,table), type(M,rtable);  
type(v,table), type(v,rtable);  
false, true  
false, true
```

...mais pour les anciennes formes (avec **matrix** et **vector** commençant par des minuscules et que l'on peut considérer comme obsolètes), c'est l'inverse !

```
> m:=matrix([[a,b],[c,d]]): v:=vector([a,b,c]):  
type(m,table), type(m,rtable);  
type(v,table), type(v,rtable);  
true, false  
true, false
```

Par contre, ils ont tous le type **tabular** (ce n'est pas le cas des listes ou des ensembles)

```
> type(L_onde,tabular);  
type(M,tabular), type(V,tabular);  
type(m,tabular), type(v,tabular);
```

```

true
true, true
true, true

```

Exercices : Analysez les opérations suivantes. **Attention**: l'ordre des éléments de la liste d'une table est imprévisible et les réponses peuvent être différentes si on reproduit ces exemples. La syntaxe d'utilisation normale des tables rend ce fait sans importance.

```

> op(2,op(2,eval(L_onde)));
whattype(%);
type(%%,`=`);
rouge = 0.81
=
true

> op(1,op(2,op(2,eval(L_onde)))); 
op(indices(L_onde)[2]);
rouge
rouge

> op(1,eval(L_onde));
> print(L_onde);
table([bleu = 0.4, rouge = 0.81, vert = 0.65])

```

Exercice : transformer, sans utiliser la fonction **convert** et en une seule instruction, les valeurs de la table *L_onde* en une liste.

Les Tableaux (Array)

Construction et utilisation des tableaux

Les tableaux sont des structures de données de type **rtable** pour lesquelles l'indexation des éléments est réalisée seulement par des nombres entiers successifs. L'exemple suivant montre comment créer un tableau en utilisant la fonction **Array**.

Attention, Attention

On sera très attentif à utiliser un *A* majuscule pour le mot **Array**. En effet, avec un *a* minuscule, MAPLE construit aussi un tableau mais avec une autre structure (celle utilisée dans les versions antérieures du logiciel et conservée dans cette version pour des raisons de compatibilité). MAPLE ne signalera donc aucune erreur, mais l'objet ainsi construit n'aura pas les propriétés décrites dans ce chapitre.

```

> restart;
A:=Array(-2..-1,-3..0, [[a,b,c,d],[1,2,3,4]]);
A := RTABLE(295660, Array(-2 .. -1, -3 .. 0,
{(-2, -3) = a, (-2, -2) = b, (-2, -1) = c, (-2, 0) = d, (-1, -3) = 1, (-1, -2) = 2, (-1, -1) = 3, (-1, 0) = 4},
datatype = anything, storage = rectangular, order = Fortran_order), Array)

```

Etant basé sur une structure **rtable** les différences majeures (mais ce ne sont pas les seules) entre **Array** et **Matrix** ou **Vector** portent sur le mode d'affichage et sur les indices dont le nombre peut être supérieur à 2 et

qui peuvent prendre des valeurs négatives ou nulles. On examinera avec soin, dans l'exemple précédent, comment **Array** associe les index aux éléments de la liste de listes.

Comme pour les matrices, on accède à un élément avec la syntaxe suivante (**attention**: -1 ne signifie plus, comme pour listes, les matrices ou les vecteurs, la dernière valeur de l'index)

```
> A[-1,-1], A[-2,0]/A[-2,-3];
3,  $\frac{d}{a}$ 
```

On modifie les éléments d'un tableau par assignation. Le **datatype** de *A* étant **anything** par défaut (voir ci-dessus) on peut assigner tout objet valide (voir chapitre 4)

```
> A[-1,-1]:=x^2+1; A[-1,0]:="Abcd efg";
A_{-1, -1} := x^2 + 1
A_{-1, 0} := "Abcd efg"
```

Comme les tableaux sont des structures **rtable**, on retrouvera certaines options de construction des matrices (voir chapitre 4). Ici on fixe le **datatype** et *N* ne peut contenir que des nombres en représentation flottante sur 8 octets

```
> N:=Array(0..1,0..2,datatype=float[8]);
N := RTABLE(298396,
    Array(0 .. 1, 0 .. 2, {}, datatype = float8, storage = rectangular, order = Fortran_order), Array)
```

```
> N[0,0]:=3/2; N[0,0];
1.5000000000000000
```

```
> N[0,1]:=x;
Error, unable to store 'x' when datatype=float[8]
```

Si la liste n'est pas conforme à l'indexation on peut obtenir un remplissage partiel du tableau, le reste étant rempli avec la valeur par défaut 0. Avec l'option **fill** on peut changer cette valeur

```
> B:=Array(-2..-1,-1..2,[ [a,b,c,d,1,2,3,4]],fill=W);
B := RTABLE(303188, Array(-2 .. -1, -1 .. 2,
    {(-2, -1) = a, (-2, 0) = b, (-2, 1) = c, (-2, 2) = d, (-1, -1) = W, (-1, 0) = W, (-1, 1) = W, (-1, 2) = W},
    datatype = anything, storage = rectangular, order = Fortran_order), Array)
```

```
> B[-2,-1],B[-1,-1],B[-1,0];
a, W, W
```

Un tableau peut d'ailleurs être créé sans définir aucun de ces éléments

```
> X:=Array(-1..2,0..1);
X := RTABLE(344900,
    Array(-1 .. 2, 0 .. 1, {}, datatype = anything, storage = rectangular, order = Fortran_order), Array)
```

Ils prennent tous la valeur 0 sauf si on utilise **fill**

```
> X[0,1];
0
```

Si l'indexation ne porte que sur deux intervalles et que les deux index débutent à 1, **Array** affiche alors les

tableaux sous une forme qui *ressemble* à une matrice. Le deuxième exemple montre que cette indexation est implicite si elle n'est pas donnée en arguments.

```
> B:=Array(1..2,1..4,[[a,b,c,d],[1,2,3,4]]);  
B:=Array([[a,b,c,d],[1,2,3,4]]);
```

$$B := \begin{bmatrix} a & b & c & d \\ 1 & 2 & 3 & 4 \end{bmatrix}$$

$$B := \begin{bmatrix} a & b & c & d \\ 1 & 2 & 3 & 4 \end{bmatrix}$$

Type des tableaux

Un tableau a le type *Array* et sa fonction intrinsèque porte le même nom

```
> whattype(A), op(0,A);  
Array, Array
```

Attention à la Majuscule. Le type *array* existe aussi mais correspond à une forme obsolète des constructions avec la fonction *array*

```
> type(A,Array), type(A,array);  
true, false
```

On peut être plus précis (*attention*: écrire **Array** sous la forme d'une fonction en écrivant 'Array'(...))

```
> type(N,'Array'(datatype=float[8],storage=rectangular));  
true
```

Attention, le tableau **B**, malgré les apparences, n'est pas une matrice

```
> type(B,Matrix), type(B,Array);  
false, true
```

Aussi, une opération matricielle sera refusée

```
> B.<1,2,3,4>;  
Error, (in LinearAlgebra:-Multiply) expects its 1st argument, MV1, to be of type  
{Matrix,Vector,scalar}, but received Array( 1..2,1..4,[ a, 1, b, 2, c, 3, d, 4 ] ,  
datatype = anything, storage = rectangular, order = C_order )
```

On peut néanmoins opérer une conversion

```
> M:=convert(B,Matrix);  
type(M,Matrix);  
M.<1,-2,3,-4>;
```

$$M := \begin{bmatrix} a & b & c & d \\ 1 & 2 & 3 & 4 \end{bmatrix}$$

true

$$\begin{bmatrix} a - 2b + 3c - 4d \\ -10 \end{bmatrix}$$

Tous les tableaux ont aussi le type *tabular* (de même que les matrices et les vecteurs, mais pas les listes ou les

ensembles)

```
> type(A,tabular), type(B,tabular),
  type(N,tabular), type(M,tabular);
                                         true, true, true
```

Fonctions d'indexation

Le premier argument de **Array** peut être une fonction d'indexation prédéfinie

```
> C:=Array(triangular[upper], [[1,2,3],[4,5],[[]],[],[]], fill='x');
```

$$C := \begin{bmatrix} 1 & 2 & 3 \\ 0 & 5 & x \\ 0 & 0 & x \\ 0 & 0 & 0 \end{bmatrix}$$

On trouvera toutes les informations sur les modes d'indexation dans l'aide en ligne.

```
> ?rtable_indexfcn
```

Attention: l'option **storage** n'a aucun rapport avec la fonction d'indexation. Ici **storage** précise que la partie "inférieure" du tableau n'est pas stockée en mémoire, aussi...

```
> T:=Array(1..3,1..4,fill=x,storage=triangular[upper]);
  T[2,3];
  T[3,2];
```

$$T := \begin{bmatrix} x & x & x & x \\ - & x & x & x \\ - & - & x & x \\ & & x \end{bmatrix}$$

```
Error, unable to lookup index (3, 2) -- not in storage
```

Fonctions d'interrogation des tableaux

Il existe comme pour les matrices, des fonctions d'interrogation dont voici quelques exemples

```
> ArrayOptions(A);
  ArrayOptions(A,datatype);
  ArrayOptions(N,datatype);
                                         datatype = anything, storage = rectangular, order = Fortran_order
                                         anything
                                         float8
```

```
> ArrayNumDims(C); # Nombres d'index
  ArrayNumElems(C);# Nombres d'éléments
  ArrayElems(C);
                                         2
                                         6
                                         {(1, 1) = 1, (1, 2) = 2, (1, 3) = 3, (2, 2) = 5, (2, 3) = x, (3, 3) = x}
```

Comme souvent avec MAPLE, il existe plusieurs façons d'obtenir un résultat

```
> rtable_indfns(C); # ou bien  
ArrayOptions(C,storage); # ou bien  
op(1,C);  
triangularupper  
triangularupper  
triangularupper
```

Encore que l'on puisse rencontrer certaines subtilités liées ici au fait que *storage* et *fonction d'indexation* sont des concepts différents

```
> ArrayOptions(A,storage);  
rectangular
```

En réalité le tableau *A* n'a pas de fonction d'indexation et MAPLE répond ici **NULL** et n'affiche rien

```
> rtable_indfns(A);  
op(1,A);
```

On notera qu'un tableau ayant soit plus de deux indices, soit au plus deux indices mais plus de 10 éléments (sauf modification par **interface**; voir chapitre 4, § I-3) sera affiché sous une forme résumée. Un examen du tableau peut se faire alors à l'aide d'un afficheur (voir chapitre 4, § I-3).

Attention: pour les tableaux à plus de deux indices, fixer la "tranche en 2 dimensions" à examiner du tableau avec l'onglet **Options** de l'afficheur).

```
> E:=Array(0..2,-1..1,1..2,[[[1,2,-3,5],[-1,0,3,2]]],fill=x);  
E := 
$$\begin{bmatrix} 2 \times 1 \times 2 \text{ 3-D Array} \\ \text{Data Type: anything} \\ \text{Storage: rectangular} \\ \text{Order: Fortran_order} \end{bmatrix}$$

```

Tableaux indépendants et duplication, fonction copy

Rappelons que, comme pour les Matrices, les Vecteurs, les tables ou les tableaux, la fonction **copy** est indispensable pour rendre indépendantes deux structures de données. MAPLE génère un entier *nnn* qui lui permet d'identifier chaque structure **rtable** créée (*RTABLE*(*nnn*,...))

```
> X;  
RTABLE(344900,  
      Array(-1 .. 2, 0 .. 1, {}, datatype = anything, storage = rectangular, order = Fortran_order), Array)
```

En réalisant l'assignation ci-dessous on voit que *Y* est associé à la même structure **rtable** (même identificateur *nnn*). En écrivant **Y:=X;** on demande seulement aux deux identificateurs *X* et *Y* de pointer sur la même structure **rtable**

```
> Y:=X;  
Y := RTABLE(344900,  
            Array(-1 .. 2, 0 .. 1, {}, datatype = anything, storage = rectangular, order = Fortran_order), Array)
```

et tout changement sur un des éléments de *X* (resp. *Y*) change l'élément correspondant de *Y* (resp. *X*) (il s'agit donc en réalité du même élément).

```
> X[-1,0]:=a+2*x;  
Y[-1,0];
```

$X_{-1, 0} := a + 2 x$

$a + 2 x$

Avec **copy** on voit que la structure associée à Y est différente de celle de X (ce n'est pas le même entier nnn)

```
> Y:=copy(X);

$$Y := \text{RTABLE}(353060, \text{Array}(-1 .. 2, 0 .. 1, \{(-1, 0) = a + 2 x\}, \text{datatype} = \text{anything}, \text{storage} = \text{rectangular}, \text{order} = \text{Fortran\_order}), \text{Array})$$

```

```
> X[-1,0], Y[-1,0];

$$X_{-1, 0} := b - 3 y$$


$$X[-1,0], Y[-1,0];$$


$$a + 2 x, a + 2 x$$


$$b - 3 y, a + 2 x$$

```

Une copie de **rtable** hérite aussi de la fonction d'indexation et des options

```
> Z:=copy(C): Np:=copy(N):

$$\text{op}(1, Z);$$


$$\text{ArrayOptions}(Np, \text{datatype});$$


$$triangular_{upper}$$


$$float_8$$

```

Bibliothèque ArrayTools

Il existe aussi cinq fonctions d'interrogation et de manipulation des tableaux (et matrices) dans une librairie, **ArrayTools**, pour laquelle on renvoie le lecteur vers l'aide en ligne

```
> ?ArrayTools
```

Tableaux hfarray

Ce sont des tableaux de structure **rtable** dont le **datatype** est **float[8]** par défaut. La raison de l'existence d'un nom particulier pour ces tableaux est historique et liée à des raisons de compatibilité

```
> A:=hfarray(-1..0,1..2): ArrayOptions(A,datatype);

$$A[-1,1] := \pi;$$


$$A[-1,1];$$


$$float_8$$


$$A_{-1, 1} := \pi$$


$$3.14159265358979312$$

```

Mais...

```
> A[-1,1]:=Pi/2;
Error, unable to store '1/2*Pi' when datatype=float[8]
```

```
> A[-1,1]:=evalhf(Pi/2):

$$A[-1,1] := \text{evalf}[18](\pi/2);$$


$$A[-1,1];$$


$$1.57079632679489655$$

```


6 - Polynômes et Fractions Rationnelles

Polynômes

Les polynômes sont pour MAPLE des expressions comme les autres si ce n'est qu'elles possèdent aussi le type **polynom**. Le type de base est cependant + puisque l'expression d'un polynôme est avant tout une somme. Définissons deux expressions polynomiales

```
> restart;
p1:=x^3+5*x^2+8*x+4;
p1 :=  $x^3 + 5x^2 + 8x + 4$ 

> p2:=x^4-9*x^n+23*x^2-3*x-36;
p2 :=  $x^4 - 9x^n + 23x^2 - 3x - 36$ 
```

Ces deux expressions sont bien du type "somme" (^+^)

```
> whattype(p1), whattype(p2);
+ , +
```

Vérifions que se sont bien des polynômes

```
> type(p1,polynom), type(p2,polynom);
true, false
```

??? En effet on a écrit x^n au lieu de x^3 . Les exposants doivent être des *entiers numériques positifs ou nuls* pour que MAPLE puisse affirmer qu'une expression est un polynôme. Corrigeons par substitution

```
> p2:=subs(x^n=x^3,p2);
type(p2,polynom);
p2 :=  $x^4 - 9x^3 + 23x^2 - 3x - 36$ 
true
```

On effectue maintenant le produit des polynômes que l'on peut développer avec **expand**

```
> p1*p2;
Px:=expand(p1*p2);
(x^3 + 5x^2 + 8x + 4)(x^4 - 9x^3 + 23x^2 - 3x - 36)
Px :=  $x^7 - 4x^6 - 14x^5 + 44x^4 + 97x^3 - 112x^2 - 300x - 144$ 
```

Vérifions que $x = -1$ est racine de Px :

```
> Eval(Px,x=-1):%>value(%);
(x^7 - 4x^6 - 14x^5 + 44x^4 + 97x^3 - 112x^2 - 300x - 144) |  
x = -1 = 0
```

Il existe de nombreuses opérations possibles sur les expressions polynomiales comme par exemple la mise

sous une forme ordonnée par degré décroissant d'un polynôme avec la fonction **sort**

```
> px:=sort(2*x^2-4*a*x^3+x^4-1-3*x);  
px :=  $x^4 - 4x^3a + 2x^2 - 3x - 1$ 
```

ou la recherche avec la fonction **coeff** du coefficient d'un terme de degré donné.

```
> coeff(px,x,3);  
coeff(px,x,0) ;  
-4 a  
-1
```

Les fonctions **degree** et **ldegree** permettent d'obtenir le degré d'un polynôme et son degré inférieur

```
> degree(4*x^4-x^2-x^6), ldegree(4*x^4-x^2-x^6);  
6, 2
```

Attention: le polynôme doit être complètement développé sinon on risque d'obtenir un résultat faux, MAPLE ne reconnaissant pas ici que les termes de plus haut degré s'éliminent

```
> px:=(x^2-1)*(x-1)-x^3;  
degree(px);  
degree(expand(px));  
px :=  $(x^2 - 1)(x - 1) - x^3$   
3  
2
```

Pour un polynôme nul MAPLE considère son degré comme étant $-\infty$ et 0 pour un polynôme constant

```
> p0:=3*x*(x^2-1)-1/2*x*(6*x^2-6):  
degree(p0);  
degree(expand(p0));  
degree(2);  
3  
 $-\infty$   
0
```

Pour les polynômes à plusieurs variables la fonction **degree** donne le degré total. En donnant un nom de variable en argument supplémentaire elle donne le *degré relatif à cette variable*.

```
> pxy:=expand((x-a*y)*(x-a*y)^3+b*x^2-x*(a*x*y+b*x)-a*(y-1));  
degree(pxy);  
degree(pxy,y);  
pxy :=  $x^4 - 4ax^3y + 6x^2a^2y^2 - 4xa^3y^3 + a^4y^4 - ax^2y - ay + a$   
8  
4
```

Exercice: commentez ces résultats (voir pour rappel le chapitre 2)

```
> degree(subs(a*x=z,pxy),z);  
degree(algsubs(a*x=z,pxy),z);  
0
```

La fonction **collect** permet de regrouper les coefficients suivant les puissances d'une variable choisie. Remarquez que ces fonctions considèrent tous les noms indéterminés comme des variables et non comme d'éventuels coefficients paramétrés. Seule l'habitude prise par convention pour utiliser par exemple a pour désigner un paramètre plutôt qu'une variable peut laisser croire que le terme $-4a$ est le coefficient de x^3y

```
> collect(pxy,x);
coeff(%,x,3);
coeff(%,x,0);

$$x^4 - 4ax^3y + (-ay + 6a^2y^2)x^2 - 4xa^3y^3 + a^4y^4 - ay + a$$


$$-4ay$$


$$a^4y^4 - ay + a$$

```

```
> collect(pxy,a);
coeff(%,a,1);

$$a^4y^4 - 4xa^3y^3 + 6x^2a^2y^2 + (-4x^3y - y + 1 - x^2y)a + x^4$$


$$-4x^3y - y + 1 - x^2y$$

```

Ces fonctions ne s'appliquent pas qu'aux polynômes mais au "caractère polynomial" des expressions

Exercice : Interprétez les réponses suivantes (vérifier le rôle de **expand**)

```
> ex:=cos(x)^2*exp(t)+cos(3*x)*exp(2*t);
type(ex,polynom);

$$ex := \cos(x)^2 e^t + \cos(3x) e^{(2t)}$$

false
```

```
> degree(ex,exp(t));
degree(expand(ex),exp(t));
1
2
```

```
> degree(ex,cos(x));
degree(expand(ex),cos(x));
2
3
```

Familles de polynômes orthogonaux

MAPLE connaît la plupart des polynômes spéciaux (Bernoulli, Bernstein...) ainsi que les familles de **polynômes orthogonaux**, Legendre, Laguerre, Hermite, ChebyshevT, ChebyshevU, Gegenbauer et Jacobi. On rappelle que les polynômes p_n à coefficients réels et de degré n sont orthogonaux sur l'intervalle $[a, b]$ relativement à une fonction de poids ρ si

$$\int_a^b p_n(x) p_m(x) \rho(x) dx = C \delta_{n,m}$$

avec C constante réelle et $\delta(n, m) = 1$ si $n = m$ et 0 sinon. Les n racines des p_n sont toujours réelles simples et

contenues dans $[a, b]$.

Par exemple, pour les **polynômes de Legendre** $\rho(x) = 1$ et $[a, b] = [-1, 1]$ (pour l'intégration, se reporter au chapitre 10)

> **restart:**

```
LegendreP(3,x)=expand(LegendreP(3,x));
Int(LegendreP(3,x)*LegendreP(4,x),x=-1..1):%value(%);
```

$$\text{LegendreP}(3, x) = \frac{5}{2}x^3 - \frac{3}{2}x$$

$$\int_{-1}^1 \text{LegendreP}(3, x) \text{LegendreP}(4, x) dx = 0$$

Pour **Laguerre**, $\rho(x) = e^{(-x)}$ et $[a, b] = [0, \infty]$. Pour **Hermite**, $\rho(x) = e^{(-x^2)}$ et $[a, b] = [-\infty, \infty]$. Les polynômes de **Jacobi**, JacobiP(n, m, p, x), sont définis avec $[a, b] = [-1, 1]$, mais $\rho(x) = (1 - x)^m (1 + x)^p$

> **JacobiP(1,2,3,x)=expand(JacobiP(1,2,3,x));**

```
Int(JacobiP(1,2,3,x)*JacobiP(2,2,3,x)*
(1-x)^2*(1+x)^3,x=-1..1):%value(%);
```

$$\text{JacobiP}(1, 2, 3, x) = -\frac{1}{2} + \frac{7}{2}x$$

$$\int_{-1}^1 \text{JacobiP}(1, 2, 3, x) \text{JacobiP}(2, 2, 3, x) (1 - x)^2 (x + 1)^3 dx = 0$$

Pour obtenir l'expression du polynôme, on peut utiliser soit **expand**, soit **simplify**

> **ChebyshevT(3,x)=simplify(ChebyshevT(3,x));**

```
Int(ChebyshevT(3,x)*ChebyshevT(3,x)/sqrt(1-x^2),x=-1..1):%value(%);
```

$$\text{ChebyshevT}(3, x) = 4x^3 - 3x$$

$$\int_{-1}^1 \frac{\text{ChebyshevT}(3, x)^2}{\sqrt{1 - x^2}} dx = \frac{1}{2}\pi$$

En réalité ce sont des fonctions qui ne s'expriment sous la forme de polynômes que quand le ou les premiers arguments sont des entiers positifs ou nuls. Elles peuvent être définies comme solutions d'équations différentielles du second ordre et comme telles d'autres fonctions leurs sont associées qui ne sont pas des polynômes même pour des valeurs entières des arguments : ici *LegendreP* (première espèce) associée à *LegendreQ* (seconde espèce) (voir l'aide en ligne de **LegendreQ**)

> **LegendreQ(2,x)=simplify(LegendreQ(2,x));**

$$\text{LegendreQ}(2, x) = -\frac{1}{4} \ln(x + 1) + \frac{1}{4} \ln(x - 1) + \frac{3}{4}x^2 \ln(x + 1) - \frac{3}{4}x^2 \ln(x - 1) - \frac{3}{2}x$$

Ces fonctions sont aussi reliées à d'autres fonctions dites "spéciales" (voir aussi la fonction **FunctionAdvisor** au chapitre 13, *Simplifications, Manipulations*)

```
> LegendreP(1/2,x)=convert(convert(LegendreP(1/2,x),hypergeom),
StandardFunctions);
```

$$\text{LegendreP}\left(\frac{1}{2}, x\right) = - \frac{2 \text{EllipticK}\left(\frac{\sqrt{-\frac{1}{2} + \frac{1}{2}x}}{\sqrt{\frac{1}{2} + \frac{1}{2}x}}\right)}{\sqrt{\frac{1}{2} + \frac{1}{2}x} \pi} + \frac{4 \sqrt{\frac{1}{2} + \frac{1}{2}x} \text{EllipticE}\left(\frac{\sqrt{-\frac{1}{2} + \frac{1}{2}x}}{\sqrt{\frac{1}{2} + \frac{1}{2}x}}\right)}{\pi}$$

Bibliothèque orthopoly

MAPLE peut donner aussi ces polynômes orthogonaux à l'aide de la bibliothèque **orthopoly** qui contient les fonctions polynômes orthogonaux G, H, L, P, T et U correspondant respectivement aux polynômes de Gegenbauer, de Hermite, de Laguerre, de Legendre, de Tchebychev-T et de Tchebychev-U.

Attention: cette bibliothèque est déclarée obsolète par *Maplesoft* et ne sera plus maintenue.

```
> with(orthopoly);
[G, H, L, P, T, U]
```

```
> P(4,x);
H(2,x);
```

$$\begin{aligned} & \frac{3}{8} + \frac{35}{8}x^4 - \frac{15}{4}x^2 \\ & -2 + 4x^2 \end{aligned}$$

Factorisation et recherche des racines d'un polynôme

Factorisation exacte

On cherche avec la fonction **factor** la mise en facteurs exacte du premier degré de *Px*

```
> Px := x^7-4*x^6-14*x^5+44*x^4+97*x^3-112*x^2-300*x-144;
factor(Px);
```

$$\begin{aligned} Px := & x^7 - 4x^6 - 14x^5 + 44x^4 + 97x^3 - 112x^2 - 300x - 144 \\ & (x - 4)(x - 3)^2(x + 2)^2(x + 1)^2 \end{aligned}$$

Attention: sans informations supplémentaires la mise en facteurs exacte (partielle ou complète) d'expressions polynomiales par MAPLE ne peut être effective que sous la forme de produit de polynômes dont les coefficients appartiennent au corps engendré par les coefficients du polynôme à factoriser (les entiers ne constituent bien sûr pas un corps mais appartiennent à celui des rationnels). Les coefficients du polynôme *px1* sont des entiers mais il ne possède aucune racine entière ou rationnelle et MAPLE n'effectue aucune mise en facteurs

```
> px1:=x^3-x^2-2*x+5:
px1=factor(px1);
```

$$x^3 - x^2 - 2x + 5 = x^3 - x^2 - 2x + 5$$

On dit que le polynôme est **irréductible** dans le corps des rationnels

```
> irreduc(px1);
```

true

Attention: irreduc(p) => false ne signifie *pas* que le polynôme p soit réductible à un produit de polynômes du premier degré uniquement. Le polynôme est ici **partiellement réductible** dans le corps des rationnels (un des facteurs est de degré supérieur à 1)

```
> px2:=x^3-(3/4)*x-(4/3)*x^2+1:  
irreduc(px2);  
px2=factor(px2);
```

false

$$x^3 - \frac{3}{4}x - \frac{4}{3}x^2 + 1 = \frac{1}{12}(3x - 4)(4x^2 - 3)$$

Maintenant le corps engendré par les coefficients est l'extension algébrique $Q \cup \{\sqrt{2}, \sqrt{3}\}$. Les racines appartiennent à ce corps et MAPLE effectue la factorisation

```
> px3:=x^3-(sqrt(3)+sqrt(2)+1)*x^2  
+ (sqrt(2)+sqrt(2)*sqrt(3)+sqrt(3))*x  
- sqrt(3)*sqrt(2);  
factor(px3);
```

$$\begin{aligned} px3 := & x^3 - (\sqrt{3} + \sqrt{2} + 1)x^2 + (\sqrt{2} + \sqrt{2}\sqrt{3} + \sqrt{3})x - \sqrt{2}\sqrt{3} \\ & (x - 1)(x - \sqrt{3})(x - \sqrt{2}) \end{aligned}$$

Ici l'extension algébrique est $Q \cup \{i, \sqrt{2}\}$. Les racines appartenant à ce corps et MAPLE effectue encore la factorisation

```
> px4:=x^3-(2*sqrt(2)+I*sqrt(2)-1)*x^2+(4*I+2-2*sqrt(2))*x+2-2*I*sqrt(2);  
factor(px4);
```

$$\begin{aligned} px4 := & x^3 - (2\sqrt{2} + I\sqrt{2} - 1)x^2 + ((2 + 4I) - 2\sqrt{2})x + 2 - 2I\sqrt{2} \\ & (x + 1 - I\sqrt{2})(x - \sqrt{2})^2 \end{aligned}$$

L'utilisation de un ou plusieurs *coefficients exprimés sous forme décimale* engendre un résultat entièrement exprimé en notation décimale

```
> pxd:=x^2/2+2.5*x+1;  
factor(pxd);
```

$$\begin{aligned} pxd := & \frac{1}{2}x^2 + 2.5x + 1 \\ & 0.5000000000(x + 4.561552813)(x + 0.4384471872) \end{aligned}$$

Extensions algébriques : on peut préciser, si on les connaît, les éléments d'une extension algébrique du corps engendré par les coefficients. Ils sont donnés sous la forme d'un ensemble ou d'une liste ou même directement si il n'y en a qu'un.

```
> px:=x^3+x^2+x+1;  
factor(px,I);  
irreduc(px,I);
```

$$\begin{aligned} px := & x^3 + x^2 + x + 1 \\ & (x + I)(x - I)(x + 1) \\ & \text{false} \end{aligned}$$

Maintenant, la première commande ne donne pas de factorisation car les racines n'appartiennent pas au corps des rationnels. Avec le nombre $\sqrt{2}$, **factor** trouve la mise en facteurs du premier degré

```
> x^2-2*x+7/9: %=factor(%);
x^2-2*x+7/9: %=factor(%,sqrt(2));

$$x^2 - 2x + \frac{7}{9} = x^2 - 2x + \frac{7}{9}$$


$$x^2 - 2x + \frac{7}{9} = \frac{1}{9}(3x - 3 - \sqrt{2})(3x - 3 + \sqrt{2})$$

```

Il faut donner l'extension algébrique complète pour obtenir une factorisation complète (en facteurs du premier degré)

```
> px:=x^4-2*x^3-7*x^2/9+4*x-22/9:
irreduc(px,sqrt(2));
px=factor(px,sqrt(2));
false

$$x^4 - 2x^3 - \frac{7}{9}x^2 + 4x - \frac{22}{9} = \frac{1}{9}(9x^2 - 18x + 11)(x - \sqrt{2})(x + \sqrt{2})$$

```

```
> irreduc(px,{I,sqrt(2)}); 
px=factor(px,{sqrt(2),I});
false

$$x^4 - 2x^3 - \frac{7}{9}x^2 + 4x - \frac{22}{9} = -\frac{1}{9}(3x - 3 + I\sqrt{2})(-3x + 3 + I\sqrt{2})(x - \sqrt{2})(x + \sqrt{2})$$

```

Encore faut-il, pour donner une extension algébrique, avoir une idée des expressions des racines. Avec un peu de chance (voir plus loin) on peut les obtenir avec la fonction **allvalues** de la façon suivante. **RootOf(q)** n'est pas une fonction qui calcule les racines de q mais une expression qui représente formellement celles-ci

```
> q:=x^3-2*x+1;
factor(q);
allvalues(RootOf(q));

$$q := x^3 - 2x + 1$$


$$(x - 1)(x^2 + x - 1)$$


$$1, -\frac{1}{2} + \frac{1}{2}\sqrt{5}, -\frac{1}{2} - \frac{1}{2}\sqrt{5}$$

```

On sait maintenant que $\sqrt{5}$ est l'élément à considérer pour l'extension algébrique, et

```
> factor(q,sqrt(5));

$$\frac{1}{4}(2x + 1 - \sqrt{5})(2x + 1 + \sqrt{5})(x - 1)$$

```

La factorisation peut aussi s'appliquer à des polynômes à plusieurs variables

```
> p_xy:=3*x^2+3*y^2:
p_xy=factor(p_xy,I);

$$3x^2 + 3y^2 = 3(y + Ix)(y - Ix)$$

```

Attention: ces opérations de factorisation peuvent être très couteuses en temps de calcul et en espace mémoire

pour des polynômes dont le degré est élevé.

Il peut arriver que l'on ait besoin d'une factorisation formelle complète du polynôme sans pour autant avoir besoin de l'expression des racines. On peut l'obtenir avec la fonction **Split** du module **PolynomialTools** (`_Z` est une variable interne créée par MAPLE)

```
> PolynomialTools[Split](q,x);
(x + 1 + RootOf(_Z2 + _Z - 1)) (x - RootOf(_Z2 + _Z - 1)) (x - 1)
```

Si on le souhaite, la factorisation peut aussi être obtenue symboliquement en fonction des racines d'un polynôme approprié de degré plus faible. L'usage de la fonction **alias** est indispensable. Ici λ représente donc les racines de $x^2 + x - 1$ (comparer avec le résultat précédent)

```
> alias(lambda=RootOf(x^2+x-1));
factor(q,lambda);
(x + 1 + λ) (x - λ) (x - 1)
```

La fonction **roots** permet de donner les racines d'un polynôme réductible sous forme d'une liste de listes de 2 opérandes : la racine et sa multiplicité. Ici +1 est racine de multiplicité 2 et -2 est racine de multiplicité 1

```
> factor(x^3-3*x+2);
roots(x^3-3*x+2);
(x + 2) (x - 1)2
[[1, 2], [-2, 1]]
```

On peut préciser si nécessaire et si elles sont connues, comme pour **factor**, une ou plusieurs éléments des extensions algébriques du corps

```
> roots(q,sqrt(5));
[[[-½ + ½ √5, 1], [-½ - ½ √5, 1], [1, 1]]]
```

On peut également déterminer les racines en utilisant la fonction **solve** (sur laquelle nous reviendrons au chapitre 14, *Equations ordinaires*). Cette fonction résout, quand cela est possible, une équation donnée en premier argument, le deuxième précisant le nom de la variable par rapport à laquelle on cherche la solution. Il faut préciser cette variable dans le cas de polynômes à plusieurs variables, sinon cette donnée est facultative. La fonction **solve** renvoie une *suite (sequence)* et la deuxième écriture permet d'obtenir le résultat sous la forme d'une liste

```
> solve(q=0,x);# Le résultat est une suite (sequence)
[solve(q=0,x)];# Pour obtenir le résultat sous la forme d'une liste
1, -½ + ½ √5, -½ - ½ √5
[[1, -½ + ½ √5, -½ - ½ √5]]
```

Factorisation approchée

Si on ne peut obtenir une factorisation exacte on peut demander une factorisation approchée en précisant que le corps de recherche est l'ensemble des réels. Dire que l'on cherche les racines sur le corps des réels est un peu présomptueux car il ne s'agit que d'approximations décimales ! On rappelle qu'il est toujours possible grâce à la variable globale **Digits** d'augmenter la précision des calculs.

```

> p:=x^5-3*x^4+x^2-1;
factor(p,real);
sort(expand(%));

```

$$p := x^5 - 3x^4 + x^2 - 1$$

$$(x - 2.894914039) (x^2 + 1.168854159 x + 0.5180851873) (x^2 - 1.273940120 x + 0.6667501962)$$

$$x^5 - 3.000000000 x^4 + 0.999999999 x^2 - 0.999999999$$

On peut également chercher la factorisation dans le corps des complexes

```

> factor(p,complex);

```

$$(x + (0.5844270795 + 0.4201549429 I)) (x + (0.5844270795 - 0.4201549429 I)) (x$$

$$+ (-0.6369700601 + 0.5109005174 I)) (x + (-0.6369700601 - 0.5109005174 I)) (x - 2.894914039)$$

Etude d'un exemple

Puisque MAPLE sait calculer avec **allvalues** ou **solve** les racines exactes utilisant des radicaux de $q = x^3 - 2x + 1$, pourquoi n'effectue-t-il pas une mise en facteur sans avoir besoin de préciser les extensions algébriques ? Les exercices proposés dans certains manuels de travaux dirigés sont souvent académiques (cas particuliers, possibiliter d'abaisser le degré, propriétés connues a priori des racines, etc.) afin de permettre une résolution (plus ou moins !) aisée avec une feuille et un crayon. Ceci peut laisser penser à un étudiant non averti qu'avec beaucoup d'expérience un bon calculateur devrait pouvoir résoudre n'importe quelle équation polynomiale. Il n'en est rien, bien au contraire. De plus, l'apparente simplicité d'un polynôme n'est en aucun cas le gage de la "simplicité" (notion très subjective) de ses racines.

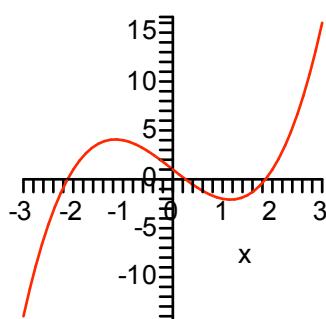
Il est bon de rappeler que l'on ne peut pas résoudre par combinaison de radicaux, *sauf cas particuliers*, les équations polynomiales de degré plus grand que 4 (comme la forme canonique pour le second degré ou les méthodes de Cardan et Ferrari pour le troisième et quatrième degré). Ce résultat a été définitivement démontré par Evariste Galois en 1830. On sait néanmoins résoudre exactement les équations de cinquième et sixième degré, mais à l'aide de fonctions spéciales (fonctions θ de Jacobi et fonctions de Kampé de Fériet). Le polynôme p suivant n'a pas l'air trop compliqué. Il est du troisième degré et on sait que les expressions exactes des racines peuvent être obtenues. De plus une représentation graphique ou une factorisation approchée montrent à l'évidence qu'elles sont réelles distinctes.

```

> restart;
p:=x^3-4*x+1;
factor(p,complex);

```

$$p := x^3 - 4x + 1$$

$$(x + 2.114907541) (x - 0.2541016884) (x - 1.860805853)$$


En fait MAPLE possède tous les outils pour déterminer de manière théorique le nombre des racines réelles

(suites de Sturm).

```
> sturm(sturmseq(p,x),x,-infinity,infinity);  
3
```

Ce qui prouve que toutes les racines sont réelles

```
> evalb(%=degree(p,x));  
true
```

Malheureusement la factorisation ne donne rien car le polynôme est irréductible dans le corps des rationnels

```
> factor(p);  
irreduc(p);  
 $x^3 - 4x + 1$   
true
```

La factorisation par **Split** donne un résultat dont l'intérêt est très relatif (du moins pour notre propos) puisqu'elle fait apparaître les racines du polynôme p sous une forme symbolique...

```
> alias(lambda=RootOf(p)):  
PolynomialTools[Split](p,x);  
 $(x - \text{RootOf}(\underline{Z}^2 + \lambda \underline{Z} - 4 + \lambda^2)) (x + \lambda + \text{RootOf}(\underline{Z}^2 + \lambda \underline{Z} - 4 + \lambda^2)) (x - \lambda)$ 
```

La fonction **roots** n'est pas plus efficace car la liste revient vide

```
> roots(p);  
[]
```

Pourtant MAPLE sait donner les expressions exactes des racines (puisque le polynôme est de degré inférieur à 5). On peut les obtenir avec **solve** qui les donne sous la forme d'une suite à laquelle on donne ici un nom (on a supprimé l'affichage un peu long)

```
> Rac:=solve(p=0,x):
```

Affichons la première. On constate que son expression fait intervenir des quantités complexes alors que nous savons que les racines sont réelles

```
> Rac[1];  

$$\frac{1}{6} (-108 + 12 I \sqrt{687})^{(1/3)} + \frac{8}{(-108 + 12 I \sqrt{687})^{(1/3)}}$$

```

Evaluons cette expression en la mettant sous la forme $A+iB$ avec la fonction **evalc**, simplifions et donnons lui le nom $x1$: la racine est bien réelle. On reconnaît ci-dessus la forme typique des racines d'une équation du troisième degré obtenues avec la méthode de Cardan qui fait intervenir des quantités complexes dont les parties imaginaires s'annulent lorsque les racines sont réelles (c'est ainsi que commence, au XVI-ième siècle, la longue histoire des nombres "impossibles" puis "imaginaires" puis "complexes"... Cette terminologie est assez symptomatique de la réticence des mathématiciens à accepter ce concept qui n'était considéré au départ que comme un artifice de calcul).

```
> x1:=simplify(evalc(Rac[1]));  

$$x1 := \frac{4}{3} \sqrt{3} \cos\left(-\frac{1}{3} \arctan\left(\frac{1}{9} \sqrt{3} \sqrt{229}\right) + \frac{1}{3} \pi\right)$$

```

de même :

```

> x2:=simplify(evalc(Rac[2]));
x3:=simplify(evalc(Rac[3]));
x2 := - $\frac{2}{3}\sqrt{3}\cos\left(-\frac{1}{3}\arctan\left(\frac{1}{9}\sqrt{3}\sqrt{229}\right) + \frac{1}{3}\pi\right)$  -  $2\sin\left(-\frac{1}{3}\arctan\left(\frac{1}{9}\sqrt{3}\sqrt{229}\right) + \frac{1}{3}\pi\right)$ 
x3 := - $\frac{2}{3}\sqrt{3}\cos\left(-\frac{1}{3}\arctan\left(\frac{1}{9}\sqrt{3}\sqrt{229}\right) + \frac{1}{3}\pi\right)$  +  $2\sin\left(-\frac{1}{3}\arctan\left(\frac{1}{9}\sqrt{3}\sqrt{229}\right) + \frac{1}{3}\pi\right)$ 

```

Vérifions

```

> simplify(expand((x-x1)*(x-x2)*(x-x3)));
x^3 +  $\frac{64}{9}\cos\left(\frac{1}{3}\arctan\left(\frac{1}{9}\sqrt{3}\sqrt{229}\right)\right)^3\sqrt{3} - 4x - \frac{16}{3}\cos\left(\frac{1}{3}\arctan\left(\frac{1}{9}\sqrt{3}\sqrt{229}\right)\right)\sqrt{3}$ 

```

??? Pas de panique ! Peut-être qu'en combinant les expressions trigonométriques ? (voir chapitre 13, *Simplifications et Manipulations*)

```

> combine(%);
x^3 - 4x + 1

```

mais malheureusement

```

> x1:=combine(x1);
x2:=combine(x2);
x3:=combine(x3);
x1 :=  $\frac{4}{3}\sqrt{3}\cos\left(-\frac{1}{3}\arctan\left(\frac{1}{9}\sqrt{687}\right) + \frac{1}{3}\pi\right)$ 
x2 := - $\frac{2}{3}\sqrt{3}\cos\left(-\frac{1}{3}\arctan\left(\frac{1}{9}\sqrt{687}\right) + \frac{1}{3}\pi\right)$  -  $2\sin\left(-\frac{1}{3}\arctan\left(\frac{1}{9}\sqrt{687}\right) + \frac{1}{3}\pi\right)$ 
x3 := - $\frac{2}{3}\sqrt{3}\cos\left(-\frac{1}{3}\arctan\left(\frac{1}{9}\sqrt{687}\right) + \frac{1}{3}\pi\right)$  +  $2\sin\left(-\frac{1}{3}\arctan\left(\frac{1}{9}\sqrt{687}\right) + \frac{1}{3}\pi\right)$ 

```

Cet exemple illustre ce qui a été dit en introduction : la simplicité du polynôme n'implique pas la simplicité de l'expression des racines. Et encore, l'exemple a été choisi pour avoir des racines assez simples ! On pourra facilement se convaincre de la difficulté du problème en cherchant avec **solve** les racines de $x^4 + x^3 + 1$. Même MAPLE rechigne à afficher le résultat sous une forme explicite et il faut lui forcer la main avec la commande

```
> _EnvExplicit:=true;
```

Si de telles mises en facteurs sont potentiellement possibles, elle ne sont généralement pas d'une grande utilité et on va s'empresser de calculer les racines sous forme d'approximations décimales...

```

> 'x1'=evalf(x1);
'x2'=evalf(x2);
'x3'=evalf[30](x3);# à titre de rappel
x1 = 1.860805853
x2 = -2.114907542
x3 = 0.254101688365052412129777879830

```

Recherche des approximations décimales des racines d'un polynôme

On sait que l'on ne peut pas résoudre par radicaux, dans le cas général, des équations polynomiales de degré plus grand que 4. Le polynôme suivant n'a pas de racine exprimables par radicaux et la réponse est symbolique (la valeur de *index* permet de distinguer symboliquement les racines)

```
> p:=x^6-3*x^2+4*x-1;
  allvalues(RootOf(p));
```

$$p := x^6 - 3x^2 + 4x - 1$$

```
RootOf(_Z^6 - 3_Z^2 + 4_Z - 1, index = 1), RootOf(_Z^6 - 3_Z^2 + 4_Z - 1, index = 2),
  RootOf(_Z^6 - 3_Z^2 + 4_Z - 1, index = 3), RootOf(_Z^6 - 3_Z^2 + 4_Z - 1, index = 4),
  RootOf(_Z^6 - 3_Z^2 + 4_Z - 1, index = 5), RootOf(_Z^6 - 3_Z^2 + 4_Z - 1, index = 6)
```

On rappelle que l'on obtient une factorisation approchée avec

```
> factor(p, real);
(x + 1.563024176) (x - 0.3326564636) (x^2 + 0.5413949538 x + 2.058821799) (x^2 - 1.771762666 x
  + 0.9341563714)

> factor(p, complex);
(x + 1.563024176) (x + (0.2706974769 + 1.409093565 I)) (x + (0.2706974769 - 1.409093565 I)) (x
  - 0.3326564636) (x + (-0.8858813332 + 0.3864849737 I)) (x + (-0.8858813332 - 0.3864849737 I))
```

La fonction **fsolve** (floating solve) sur laquelle nous reviendrons au chapitre 14, *Équations ordinaires*, permet de résoudre numériquement une équation en précisant un encadrement de la racine cherchée. Cependant, lorsqu'elle est appliquée à une équation polynomiale, elle permet d'obtenir toutes les racines réelles sans fournir cet encadrement

```
> fsolve(p=0, x);
-1.563024176, 0.3326564636
```

Exercice: que cherche-t-on ?

```
> identify(%);
[-1.563024176, 0.3326564636]
```

Les 4 autres racines sont complexes et toutes les racines peuvent être déterminées en précisant

```
> fsolve(p=0, x, complex);
-1.563024176, -0.2706974769 - 1.409093565 I, -0.2706974769 + 1.409093565 I, 0.3326564636,
  0.8858813332 - 0.3864849737 I, 0.8858813332 + 0.3864849737 I
```

Si on ne s'intéresse qu'aux racines réelles situées dans un intervalle particulier on peut préciser celui-ci avec la variable en deuxième argument. Ici l'intervalle de recherche est [0,1] et on précise d'effectuer temporairement les calculs avec 30 chiffres significatifs.

```
> k:=Digits: Digits:=30:
  fsolve(p, x=0..1);
  Digits:=k:
0.332656463615427095348448999995
```

Fractions rationnelles

Les fractions rationnelles sont définies comme le rapport de deux polynômes à une ou plusieurs variables ou toute expression équivalente. Ces objets ont pour type de base `*` mais possèdent aussi le type **ratpoly**.

```
> restart:  
F:=(x^2+4*x+4)/(x^3-4*x^2-3*x+18);  
whattype(F);# Pourquoi le type * ?  
type(F, ratpoly);
```

$$F := \frac{x^2 + 4x + 4}{x^3 - 4x^2 - 3x + 18}$$

*

true

```
> ff:=x+x*y^2/(x-1);  
type(ff, ratpoly);
```

$$ff := x + \frac{xy^2}{x - 1}$$

true

```
> S_x:=(cos(x)^2+1)/(cos(x)^2-1);  
type(S_x, ratpoly);
```

$$S_x := \frac{\cos(x)^2 + 1}{\cos(x)^2 - 1}$$

false

```
> S_u:=subs(cos(x)=u,S_x);  
type(S_u, ratpoly);
```

$$S_u := \frac{u^2 + 1}{u^2 - 1}$$

true

```
> f_x:=(1+x^x)/(x^2-3*x+1);  
type(f_x, ratpoly);
```

$$f_x := \frac{1 + x^x}{x^2 - 3x + 1}$$

false

Exercice : Expliquez cette réponse (juste!)

```
> px:=x^2-4*x+2;  
type(px, ratpoly);
```

$$px := x^2 - 4x + 2$$

true

Les fonctions **numer** et **denom** permettent d'isoler le numérateur et le dénominateur d'une fraction

```

> F;
Eval(numer(F),x=3):%:=value(%);
Eval(denom(F),x=3):%:=value(%);# 3 est donc un pôle de F

$$\frac{x^2 + 4x + 4}{x^3 - 4x^2 - 3x + 18}$$


$$(x^2 + 4x + 4) \Big|_{x=3} = 25$$


$$(x^3 - 4x^2 - 3x + 18) \Big|_{x=3} = 0$$


```

Voir l'exercice précédent

```

> denom(x^2-1);
1

```

On peut réaliser diverses opérations algébriques comme la division euclidienne avec la fonction **quo** (quotient)

```

> F:=(x^4-4*x^2-3*x+18)/(x^2+4*x+4);
F :=  $\frac{x^4 - 4x^2 - 3x + 18}{x^2 + 4x + 4}$ 

```

```

> quo(numer(F),denom(F),x);
x^2 - 4x + 8

```

Un troisième argument (non assigné) permet de récupérer le reste de la division (on peut aussi utiliser la fonction **rem**)

```

> Q:=quo(numer(F),denom(F),x,'R');
R;
Q := x^2 - 4x + 8
      -14 - 19x

```

La fonction **normal** réduit au même dénominateur

```

> Q+R/denom(F);
normal(%);
evalb(expand(Q*denom(F))+R = numer(F));
x^2 - 4x + 8 +  $\frac{-14 - 19x}{x^2 + 4x + 4}$ 

$$\frac{x^4 - 4x^2 - 3x + 18}{x^2 + 4x + 4}$$

true

```

On trouve dans MAPLE des fonctions spécifiques aux fractions rationnelles telle que la décomposition de seconde espèce (dans R) en somme de fractions simples par la fonction **convert** et l'option **parfrac**. La donnée de la variable (ici x) est optionnelle et seulement obligatoire quand la fraction dépend de plusieurs indéterminées.

```
> Fx:=(x^3+3*x^2-x+3)/(x^4+2*x^3-x^2+4*x+12);
convert(Fx,parfrac,x);
```

$$Fx := \frac{x^3 + 3x^2 - x + 3}{x^4 + 2x^3 - x^2 + 4x + 12}$$

$$\frac{78x - 48}{121(x^2 - 2x + 3)} + \frac{9}{11(x+2)^2} + \frac{43}{121(x+2)}$$

Le dénominateur de la première fraction est un polynôme non factorisé d'un degré supérieur à 1 et peut être réduit de façon formelle avec l'option **fullparfrac** en utilisant les racines symboliques du polynôme correspondant

```
> convert(Fx,fullparfrac,x);
```

$$\left(\sum_{\alpha = \text{RootOf}(Z^2 - 2Z + 3)} \frac{-\frac{15}{242} - \alpha + \frac{93}{242}}{x - \alpha} \right) + \frac{9}{11(x+2)^2} + \frac{43}{121(x+2)}$$

Pour tenter d'obtenir la décomposition explicite de première espèce (dans C) on pourra donner, si on les connaît, les éléments d'une extension algébrique

```
> convert(Fx,parfrac,x,[I,sqrt(2)]);
```

$$-\frac{\frac{3}{2}I(1+I\sqrt{2})\sqrt{2}}{(x-1+I\sqrt{2})(-3+I\sqrt{2})^2} - \frac{9}{(x+2)^2(-3+I\sqrt{2})(3+I\sqrt{2})} + \frac{\frac{3}{2}I(-1+I\sqrt{2})\sqrt{2}}{(-x+1+I\sqrt{2})(3+I\sqrt{2})^2}$$

$$+ \frac{\frac{43}{2}}{(x+2)(-3+I\sqrt{2})^2(3+I\sqrt{2})^2}$$

Les résultats pourront être donnés sous une forme simplifiée avec la fonction **rgf_pfrac** de la bibliothèque **genfunc**.

```
> C_Fx:=genfunc[rgf_pfrac](Fx,x);
```

$$C_Fx := -\frac{\frac{3}{242}I(-5+13I\sqrt{2})\sqrt{2}}{x-1+I\sqrt{2}} - \frac{\frac{3}{242}I(5+13I\sqrt{2})\sqrt{2}}{x-1-I\sqrt{2}} + \frac{9}{11(x+2)^2} + \frac{43}{121(x+2)}$$

On remarque que la factorisation est complète sans avoir eu besoin de donner les éléments de l'extension algébrique $\mathbb{Q} \cup \{I, \sqrt{2}\}$. Pas de miracle..., ceci n'est pas en contradiction avec ce que nous avons dit jusqu'ici. Simplement **rgf_pfrac** fait un peu plus que **factor** en résolvant les facteurs du second degré qui peuvent apparaître. On est simplement tombé sur un cas favorable pour lequel le dénominateur de Fx est un produit de facteurs du second et premier degré

```
> factor(denom(Fx));
```

$$(x^2 - 2x + 3)(x + 2)^2$$

En effet, dans une situation moins favorable on obtient seulement un résultat symbolique

```
> F:=1/(x^3-4*x+1):
F:=convert(F,fullparfrac,x);
F:=genfunc[rgf_pfrac](F,x);
```

$$\frac{1}{x^3 - 4x + 1} = \sum_{\alpha = \text{RootOf}(Z^3 - 4Z + 1)} \frac{-\frac{64}{229} + \frac{24}{229}\alpha^2 + \frac{9}{229}\alpha}{x - \alpha}$$

$$\frac{1}{x^3 - 4x + 1} = \sum_{R = \text{RootOf}(Z^3 - 4Z + 1)} \frac{\lim_{x \rightarrow -R} \left(\frac{x - -R}{x^3 - 4x + 1} \right)}{x - -R}$$

La fonction **rationalize** permet de mettre une expression sous une forme qui élimine les radicaux du ou des dénominateurs d'une expression.

Exercice : comparez ces deux résultats :

```
> rationalize(C_Fx);
- -363 - 121 x^3 - 363 x^2 + 121 x
  121 (x^2 + 4 x + 4) (x^2 - 2 x + 3)
```



```
> map(rationalize,C_Fx);
3 I (-5 + 13 I sqrt(2)) sqrt(2) (-x + 1 + I sqrt(2)) - 3 I (5 + 13 I sqrt(2)) sqrt(2) (x - 1 + I sqrt(2)) + 9
  242 x^2 - 484 x + 726
+ 43
  121 x + 242
```

La fonction **factor** cherche, si possible, la factorisation du numérateur et du dénominateur.

```
> fx:=(x^2-4)/(x^2-4*x+3);
factor(fx);
```

$$fx := \frac{x^2 - 4}{x^2 - 4x + 3}$$

$$\frac{(x - 2)(x + 2)}{(x - 1)(x - 3)}$$

Exercice: on veut simplifier l'écriture. Expliquez

```
> e:=rho=RootOf(denom(F)): e; alias(e):
F=subs(_alpha=alpha,factor(convert(F,fullparfrac,x)));
rho = RootOf(Z^3 - 4 Z + 1)


$$\frac{1}{x^3 - 4x + 1} = \sum_{\alpha = \rho} \left( -\frac{-64 + 24\alpha^2 + 9\alpha}{229(-x + \alpha)} \right)$$

```

On peut avec **convert**, obtenir aussi une décomposition de première espèce en donnant éventuellement les éléments d'une extension algébrique.

```
> Fx;
factor(Fx,{I,sqrt(2)});
convert(%,parfrac,x);
```

$$\frac{x^3 + 3x^2 - x + 3}{x^4 + 2x^3 - x^2 + 4x + 12}$$

$$\begin{aligned}
& - \frac{x^3 + 3x^2 - x + 3}{(x - 1 + I\sqrt{2})(-x + 1 + I\sqrt{2})(x + 2)^2} \\
& - \frac{\frac{3}{2}I(1 + I\sqrt{2})\sqrt{2}}{(x - 1 + I\sqrt{2})(-3 + I\sqrt{2})^2} - \frac{9}{(x + 2)^2(-3 + I\sqrt{2})(3 + I\sqrt{2})} - \frac{\frac{3}{2}I(-1 + I\sqrt{2})\sqrt{2}}{(x - 1 - I\sqrt{2})(3 + I\sqrt{2})^2} \\
& + \frac{\frac{43}{43}}{(x + 2)(-3 + I\sqrt{2})^2(3 + I\sqrt{2})^2}
\end{aligned}$$

On pourra aussi obtenir la décomposition d'une fraction rationnelle en fraction continue

```
> Q:=x^2*F;
fc_Q:=convert(Q,confrac,x);
```

$$\begin{aligned}
Q &:= \frac{x^2}{x^3 - 4x + 1} \\
fc_Q &:= \cfrac{1}{x - \cfrac{4}{x + \cfrac{1}{4} + \cfrac{1}{16\left(x - \cfrac{1}{4}\right)}}}
\end{aligned}$$

7 - Fonctions

La syntaxe adoptée pour définir une fonction mathématique à une variable est

$f : x \rightarrow \text{expression dépendant de } x$

Ce qui se lit : " f est la fonction qui à x fait correspondre ...". MAPLE permet de définir très simplement des fonctions en utilisant la même syntaxe (la flèche est construite avec les deux caractères - et $>$)

> restart:

f := x -> 3*x^2 - sin(Pi*x);

$$f := x \rightarrow 3x^2 - \sin(\pi x)$$

que l'on traduira en langage MAPLE par "on assigne au nom f un opérateur qui à l'objet x fait correspondre l'expression $3x^2 - \sin(\pi x)$ ". La syntaxe d'utilisation de la fonction est alors habituelle

> f(a);

x^2*f(x-1)+1;

expand(x^2*f(x-1)+1);

$$3a^2 - \sin(\pi a)$$

$$x^2(3(x-1)^2 - \sin(\pi(x-1))) + 1$$

$$3x^4 - 6x^3 + 3x^2 + x^2 \sin(\pi x) + 1$$

Attention

On ne confondra pas les écritures f et $f(x)$:

Respectant strictement la notation mathématique, la première écriture, f , est le **nom** de l'opérateur et la seconde, $f(x)$, l'**expression** de f quand l'argument vaut x . MAPLE fait totalement cette distinction. Pour afficher la définition d'une fonction on utilisera la fonction **eval**

> f; # Est le nom de l'opérateur (de la fonction)
f(x); # Est l'expression de f quand la variable vaut x
eval(f); # Est la définition de f

f

$$3x^2 - \sin(\pi x)$$

$$x \rightarrow 3x^2 - \sin(\pi x)$$

C'est à dessein que les termes "opérateur" et "objet" ont été employés plus haut. En effet les fonctions ne sont pas toujours définies par de simples combinaisons de fonctions numériques comme précédemment. Elles peuvent par exemple comporter des opérateurs d'évaluation. Voici deux écritures d'une même fonction dont la première est classique

> z:=x->x+sin(x);
z(2+3*I);

$$z := x \rightarrow x + \sin(x)$$

$$(2 + 3I) + \sin(2 + 3I)$$

La nouvelle fonction z_alg est identique à la précédente, mais on force le résultat à être exprimé sous sa forme algébrique, c'est-à-dire $a + I b$, avec la fonction **evalc**.

```
> z_alg:=x->evalc(x+sin(x));
z_alg(2+3*I);
z_alg := x → evalc(x + sin(x))
2 + sin(2) cosh(3) + I (3 + cos(2) sinh(3))
```

Les fonctions peuvent aussi être construites avec des opérateurs fonctionnels. L'opérateur **diff** dérive l'expression associée au nom x par rapport à la variable t (voir chapitre 8, *Dérivation*)

```
> d:=x->sin(t)*diff(x,t);
d(sin(t^2)-1/t); # sin(t^2)-1/t est mis pour x
d(sin(z^2)-1/t); # sin(z^2)-1/t est mis pour x
d := x → sin(t)  $\left( \frac{\partial}{\partial t} x \right)$ 
sin(t)  $\left( 2 \cos(t^2) t + \frac{1}{t^2} \right)$ 
 $\frac{\sin(t)}{t^2}$ 
```

Avec **expand** on force le développement des produits de facteurs (voir chapitre 13, *Simplifications, Manipulations*)

```
> d:=x->expand(sin(t)*diff(x,t));
d(sin(t^2)-1/t);
d := x → expand  $\left( \sin(t) \left( \frac{\partial}{\partial t} x \right) \right)$ 
2 sin(t) cos(t^2) t +  $\frac{\sin(t)}{t^2}$ 
```

Les arguments comme les images peuvent être des objets quelconques valides. Ici on suppose que l'argument de la fonction sera un nombre entier et son image sera une liste

```
> g:=n->[seq(i^n,i=0..n)];
g(3);
g := n → [seq( $i^n$ ,  $i = 0 .. n$ )]
[0, 1, 8, 27]
```

Maintenant on suppose que l'argument est une liste. Comme on peut le voir, la construction d'une fonction peut faire intervenir des opérateurs internes anonymes tel que $t \rightarrow 2 t \sin(t)$ qui va être appliqué avec **map** sur chaque élément de la liste pris pour argument pour former une nouvelle liste

```
> p:=x->map(t->2*t*sin(t),x);
L:=[0,Pi/2,3*Pi/2,Pi/4];
p(L);
p := x → map( $t \rightarrow 2 t \sin(t)$ , x)
```

$$L := \left[0, \frac{1}{2}\pi, \frac{3}{2}\pi, \frac{1}{4}\pi \right]$$

$$\left[0, \pi, -3\pi, \frac{1}{4}\pi\sqrt{2} \right]$$

L'image de la fonction précédente étant une liste on a parfaitement le droit d'écrire la combinaison

$$> p(L)[3]; \\ -3\pi$$

Voici encore deux exemples qui montrent que l'image d'une fonction peut être un objet quelconque. Ici l'image est une **équation ordinaire** (on pourrait tout aussi bien construire une fonction qui génère des équations différentielles...)

$$> eq:=n->(x-n)^n=sort(add(i,i in [seq(i*x^(i-1),i=1..n)])); \\ eq := n \rightarrow (x - n)^n = sort(add(i, i \in [seq(i x^{(i - 1)}, i = 1 .. n)]))$$

$$> eq(3); \text{ # C'est une équation, pas une identité !} \\ (x - 3)^3 = 3x^2 + 2x + 1$$

Maintenant, à chaque entier n positif la fonction M fera correspondre une matrice

$$> M:=n->Matrix([seq([seq((-x)^(i-1)/(i-1)!,i=(j-1)*n+1..j*n)],j=1..n)]); \\ M := n \rightarrow Matrix \left(\left[\begin{array}{c} seq \left(\left[\begin{array}{c} seq \left(\frac{(-x)^{(i - 1)}}{(i - 1)!}, i = (j - 1) n + 1 .. j n \right) \right], j = 1 .. n \right) \end{array} \right] \right)$$

$$> M(2);$$

$$\begin{bmatrix} 1 & -x \\ \frac{1}{2}x^2 & -\frac{1}{6}x^3 \end{bmatrix}$$

A la fin du chapitre 16, *Représentations graphiques 2D*, on trouvera l'exemple d'une fonction dont l'image est une **représentation graphique**. On en donne aussi un exemple ci-dessous (voir l'histogramme).

Certaines fonctions peuvent ne pas avoir d'argument. MAPLE possède divers générateurs de nombres **pseudo-aléatoires** à partir desquels nous construirons des exemples. MAPLE contient en particulier une bibliothèque, **RandomTools**, permettant de travailler avec des objets ayant un caractère aléatoire. La fonction **Generate** de cette bibliothèque permet la génération de divers types d'objets dont évidemment des nombres avec des caractéristiques fixées (ou définies par l'utilisateur, voir **AddFlavor**).

Remarque : on parle de nombres **pseudo-aléatoires** car des nombres générés par un algorithme, processus par nature déterministe, ne peuvent pas être réellement aléatoires; ils en ont simplement l'essentiel des propriétés statistiques.

On définit ici une fonction sans argument qui génère des nombres décimaux aléatoires à 3 chiffres significatifs et uniformément répartis sur l'intervalle [0,10].

$$> r:=()->RandomTools[Generate](\\ float(range=0..10,digits=3,method=uniform)); \\ r := () \rightarrow RandomTools[Generate](float(range = 0 .. 10, digits = 3, method = uniform))$$

Pour les bibliothèques de modules (voir chapitre 19) comme **RandomTools** on peut aussi utiliser la syntaxe (":-")

```
> r:=()->RandomTools:-Generate(  
    float(range=0..10,digits=3,method=uniform));  
r := () → (RandomTools:-Generate)(float(range = 0 .. 10, digits = 3, method = uniform))
```

Remarque : on aura noté la façon dont ces constructions de *r* extraient la fonction **Generate** de la bibliothèque **RandomTools**. Ces écritures rendent l'utilisation de la fonction indépendante d'un accès préalable de la bibliothèque par **with** (voir chapitre 1, *Commentaires sur l'utilisation de MAPLE*, § IV-7 ainsi qu'un exemple au chapitre 4, *Vecteurs, Matrices et Algèbre linéaire*, § IV *Algèbre Linéaire; accès aux fonctions de LinearAlgebra*).

Voici une liste de valeurs générées par cette fonction

```
> [seq(r(),i=1..10)];  
[7.88, 5.98, 0.969, 7.11, 1.90, 7.96, 5.62, 1.51, 4.59, 4.24]
```

La répétition de cette instruction donnera une liste différente

```
> [seq(r(),i=1..10)];  
[6.53, 9.91, 7.52, 4.75, 0.844, 4.32, 8.17, 9.32, 9.20, 9.62]
```

Autre exemple : on veut simuler des tirages à pile (0 ou 1 au choix) ou face (resp. 1 ou 0) (tirages de Bernoulli). On écrira

```
> pile_face:=()>RandomTools[Generate](integer(range=0..1));  
pile_face := () → RandomToolsGenerate(integer(range = 0 .. 1))
```

Voici une suite de 20 tirages

```
> seq(pile_face(),i=1..20);  
0, 0, 0, 1, 1, 1, 0, 0, 1, 0, 1, 1, 0, 1, 0, 1, 0, 0, 1, 1
```

Exercice : expliquez l'écriture de cette fonction

```
> pile_face:=()>[ 'P' , 'F' ] [RandomTools:-Generate(integer(range=1..2))];  
seq(pile_face(),i=1..20);  
pile_face := () → ['P', 'F']  
                                         (RandomTools:-Generate)(integer(range = 1 .. 2))  
F, P, F, P, P, P, F, F, F, F, F, P, F, F, F, F, F, F, F, F, F, F
```

ainsi que ces résultats

```
> F,P:=0,1;  
seq(pile_face(),i=1..20);  
%;  
F, P := 0, 1  
P, P, F, F, F, F, P, F, P, P, F, P, F, F, P, F, F, P, F, P, F, P  
1, 1, 0, 0, 0, 0, 1, 0, 1, 1, 0, 1, 0, 0, 1, 0, 0, 1, 0, 1, 0, 1
```

Noter aussi l'existence de la fonction **rand** à laquelle fait appel **Generate** et qui renvoie un nombre entier positif de 12 chiffres compris entre 0 et 999999999999. Ces nombres sont statistiquement répartis de façon uniforme sur cet intervalle. La fonction **rand** est une fonction de base de MAPLE et n'appartient pas à **RandomTools**.

```
> seq(rand(), i=1..3);
681585809504, 709739738485, 231104248045
```

MAPLE contient aussi une bibliothèque, **stats**, avec des générateurs pour diverses distributions statistiques. On donne un exemple d'une fonction générant une liste de n nombres suivant une distribution gaussienne de moyenne μ et d'écart type σ . Si on regarde bien, cette définition n'est rien d'autre qu'une ré-écriture d'une fonction existante de MAPLE avec une syntaxe simplifiée

```
> Normale:=(mu,sigma,n)->[stats[random, normald[mu,sigma]](n)];
Normale := (mu, sigma, n) → [statsrandom, normald $\mu, \sigma$ (n)]
```

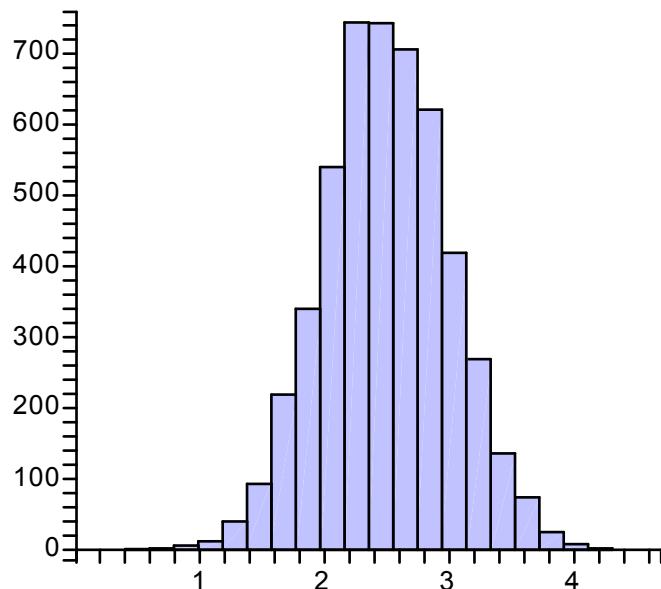
```
> Normale(1.5,0.3,5);
[1.737484094, 1.596968867, 1.802078631, 2.350595074, 1.705446673]
```

La bibliothèque **stats** contient aussi des possibilités de tracés. La fonction qui suit a pour image (au sens mathématique!) une **représentation graphique**. Elle renvoie l'histogramme d'un échantillon statistique généré de n nombres qui suivent une loi normale de moyenne μ et d'écart type σ . Elle combine une fonction de **stats** et la fonction **Normale** définie ci-dessus. Avec *nb* on fixe le nombre de barres de l'histogramme. Par défaut la fonction **histogram** dessine des barres de surfaces identiques de largeurs variables dont la valeur peut être fixée avec *area=valeur*. On impose ici, avec *area='count'*, qu'elles soient de largeurs fixes et de surfaces proportionnelles au dénombrement des valeurs contenues dans chaque intervalle

```
> Histogramme:=(n,mu,sigma,nb)->
    stats[statplots,histogram](Normale(mu,sigma,n),
    area='count',numbars=nb);
Histogramme := (n, mu, sigma, nb) → statsstatplots, histogram(Normale( $\mu, \sigma, n$ ), area = 'count',
    numbars = nb)
```

Cette fonction a pour image une représentation graphique

```
> Histogramme(5000,2.5,0.5,20);
```



Contrôle du type des arguments

On voit à travers ces divers exemples que MAPLE offre beaucoup de possibilités pour créer des fonctions. Cependant rien n'interdit d'écrire des bêtises...

```
> restart;  
p:=x->2*x*sin(x);  
L:=[0,Pi/2,3*Pi/2];
```

$$p := x \rightarrow 2 x \sin(x)$$

$$L := \left[0, \frac{1}{2}\pi, \frac{3}{2}\pi \right]$$

L'expression suivante n'a pas de sens car MAPLE ne fait que remplacer la variable de construction (x) par l'argument actuel (L) au moment de l'exécution.

```
> p(L);  
Error, (in p) invalid input: sin expects its 1st argument, x, to be of type algebraic,  
but received [0, 1/2*Pi, 3/2*Pi]
```

Par contre l'expression ci-dessous a un sens. On applique avec **map** la fonction p sur chaque élément de la liste et on constitue une nouvelle liste

```
> map(p,L);  
[0, π, -3 π]
```

On voit que l'utilisation du domaine d'application d'un opérateur ainsi créé est laissé à l'initiative de l'utilisateur. On peut néanmoins contraindre une fonction de n'accepter que certains types d'arguments. La fonction suivante opère sans erreur sur un objet de type "expression". Il applique, avec **map**, l'opérateur sur chaque opérande de x (voir le paragraphe sur la fonction **map** dans ce chapitre). Peu importe si le lecteur ne comprend pas le sens de ce premier résultat (d'ailleurs juste), ce qui importe pour notre propos immédiat est que MAPLE ne détecte aucune erreur.

```
> p:=x->map(t->t*sin(t/2),x);  
p(x^2-1);
```

$$\begin{aligned} p &:= x \rightarrow \text{map}\left(t \rightarrow t \sin\left(\frac{1}{2}t\right), x \right) \\ &x^2 \sin\left(\frac{1}{2}x^2\right) + \sin\left(\frac{1}{2}\right) \end{aligned}$$

Cette même fonction s'applique aussi sans erreur sur une liste

```
> p([0,Pi/2,Pi/3]);  
[0, 1/4 π √2, 1/6 π]
```

La première utilisation étant plutôt "exotique" on peut souhaiter que cette fonction n'opère que sur des objets de **type list**. On écrira alors

```
> p:=(x::list)->map(t->t*sin(t/2),x);  
p := x::list → \text{map}\left( t \rightarrow t \sin\left(\frac{1}{2}t\right), x \right)
```

Si on applique maintenant cette fonction sur tout autre objet qu'une liste elle renverra un message d'erreur.

```
> p(x^2-1);  
Error, invalid input: p expects its 1st argument, x, to be of type list, but received  
x^2-1
```

On peut également contraindre l'argument à n'être que des objets du type *list ou set* (ensemble) ne contenant que des constantes, et ainsi de suite (voir chapitre 2, § *Profils de types*)

```
> p:=(x::{list(constant),set(constant)})->map(t->t*sin(t/2),x);  

$$p := x::\{list(constant), set(constant)\} \rightarrow map\left( t \rightarrow t \sin\left(\frac{1}{2}t\right), x \right)$$

```

Ici l'argument est bien un ensemble mais *x* n'est pas une constante

```
> p({0,x,3});  
Error, invalid input: p expects its 1st argument, x, to be of type {list(constant),  
set(constant)}, but received {0, 3, x}
```

Maintenant les arguments sont acceptés

```
> p({0,Pi,3});  

$$\left\{ 0, 3 \sin\left(\frac{3}{2}\right), \pi \right\}$$

```

Règles générales d'évaluation des arguments et des fonctions

1) MAPLE *cherche toujours à évaluer les arguments avant de les transmettre à une fonction*. La non évaluation d'un argument (obtenue en le mettant entre 2 caractères ') au moment de l'appel peut donc être indispensable comme par exemple pour la fonction **protect** (voir chapitre 2) ou **member** (chapitre 3). *Cette règle d'évaluation préalable est très importante et doit toujours être présente à l'esprit* comme l'illustre l'exercice suivant

Exercice : Comparez et interprétez ces résultats :

```
> evalf[30](cos((1.+10^(-10))*Pi));  
evalf[30](cos((1 +10^(-10))*Pi));  
-1.  
-0.9999999999999999999950651977995
```

2) L'argument placé à gauche de " \rightarrow " au moment de la création de la fonction est muet, i.e. il ne tient pas compte d'éventuelles assignations à une "vraie" variable qui porterait le même nom. Dans l'exemple qui suit, le fait que *t* soit préalablement assigné est sans importance. On notera également que le terme *p* n'est pas évalué au moment de la construction de la fonction, ni même la constante $e^{(I\pi)}$

```
> p:= 2 ; t:=3;  
g:=t->exp(I*Pi)*p*t^2-1;  

$$p := 2$$
  

$$t := 3$$
  

$$g := t \rightarrow e^{(I\pi)} p t^2 - 1$$

```

C'est seulement au moment de l'appel de la fonction que ces quantités seront évaluées.

```
> g(z); # (-1) * (p=2) * z^2 - 1
g(t); # (-1) * (p=2) * (t=3)^2 - 1
                                         -2 z^2 - 1
                                         -19
```

Ceci permettra par exemple de changer la valeur de p qui jouera le rôle d'un paramètre (voir aussi au chapitre 13 le comportement des variables dites "contraintes").

```
> p:=-1;
g(t);
                                         p := -1
                                         8
```

Exercice : pourquoi a-t-on le droit d'utiliser le même nom x pour les deux opérateurs ? On évitera néanmoins cette écriture qui est peu lisible en choisissant des noms différents.

```
> p:=x->map(x->2*x*sin(x),x);
p(L);
                                         p := x → map(x → 2 x sin(x), x)
                                         [0, π, -3 π]
```

Fonctions de plusieurs variables

Pour définir des fonctions de plusieurs variables on écrira les arguments entre parenthèses.

```
> F:=(x,y,t)->a*sin(x/y)+b*cos(x/y)+t^2;
                                         F := (x, y, t) → a sin( x y ) + b cos( x y ) + t^2
```

Exercice : Pourquoi les parenthèses entourant les noms des variables sont-elles *nécessaires* ? Sinon qu'aurait-on assigné à F ?

Il va de soi que l'ordre des arguments lors de l'appel d'une fonction doit correspondre à l'ordre utilisé lors de la création

```
> F(0,2,u^2);
                                         b + u^4
> F(2,0,u^2);
Error, (in F) numeric exception: division by zero
```

Exercice : expliquez ce résultat. Décrivez l'ensemble des images de cette fonction ?

```
> g:=(x::numeric,y::numeric,z::numeric)->nops({x*y,-y*exp(I*Pi*x),z^2});
g(1,4,2);
                                         g := (x::numeric, y::numeric, z::numeric) → nops({-y e^(I π x), x y, z^2})
```

Transformations des expressions en fonctions: l'opérateur unapply

L'opérateur **unapply** permet de transformer une expression en une fonction. Considérons l'expression

```
> ex:=a*x^2-1;
```

$$ex := a x^2 - 1$$

Pour transformer *ex* en une fonction *f* de la variable *x* on écrira

```
> f:=unapply(ex,x);
```

$$f := x \rightarrow a x^2 - 1$$

Alors que pour construire la fonction *g* de la variable *a* on écrira

```
> g:=unapply(ex,a);
```

$$g := a \rightarrow a x^2 - 1$$

On peut construire aussi la fonction *F* des deux variables *x* et *a* en écrivant

```
> F:=unapply(ex,x,a);
```

$$F := (x, a) \rightarrow a x^2 - 1$$

qui est différente de la fonction *G* en raison de l'ordre des variables

```
> G:=unapply(ex,a,x);
```

$$G := (a, x) \rightarrow a x^2 - 1$$

```
> F(1,2);
```

```
G(1,2);
```

1

3

Attention: les fonctions construites avec **unapply** et celles construites avec \rightarrow peuvent présenter dans certaines circonstances des différences subtiles mais essentielles en raison des évaluations. Prudence !

Exercice : examinez **avec soin** cet exemple et expliquer les résultats. Précisons que MAPLE ne fait aucune erreur !

```
> k:='k':  
E:=diff(k(x),x)*sin(x);  
k:=x->x^2:
```

$$E := \left(\frac{d}{dx} k(x) \right) \sin(x)$$

```
> f:=x->E;  
g:=unapply(E,x);
```

$$f := x \rightarrow E$$

$$g := x \rightarrow 2 x \sin(x)$$

Les deux fonctions *f* et *g* ainsi construites semblent bien être identiques

```
> f(x);  
g(x);
```

$$2 x \sin(x)$$

$$2 x \sin(x)$$

Mais

```
> k:=x->cos(x):  
f(x);  
g(x);
```

$$\begin{aligned} & -\sin(x)^2 \\ & 2 x \sin(x) \end{aligned}$$

Ou encore

```
> t:='t':  
f(t);# L'expression dépend de x et non de t !  
g(t);
```

$$\begin{aligned} & -\sin(x)^2 \\ & 2 t \sin(t) \end{aligned}$$

ou

```
> f:=s->eval(E,x=s);  
f(t);  
g(t);
```

$$\begin{aligned} & f := s \rightarrow E \Big|_{x=s} \\ & -\sin(t)^2 \\ & 2 t \sin(t) \end{aligned}$$

Manipulation des fonctions par leurs noms

Les fonctions se manipulent par leurs noms comme leurs expressions associées. Soient les deux fonctions

```
> restart:  
g:=x->x-sin(2*x);  
f:=x->x^2-1;
```

$$g := x \rightarrow x - \sin(2 x)$$

$$f := x \rightarrow x^2 - 1$$

On peut définir de nouvelles fonctions en écrivant par exemple

```
> h:=g+f/2-3; q:=g/f+exp;
```

$$h := g + \frac{1}{2}f - 3$$

$$q := \frac{g}{f} + \exp$$

```
> h(s); q(s);
```

$$s - \sin(2s) + \frac{1}{2}s^2 - \frac{7}{2}$$

$$\frac{s - \sin(2s)}{s^2 - 1} + e^s$$

> (2*h-q)(s);

$$2s - 2\sin(2s) + s^2 - 7 - \frac{s - \sin(2s)}{s^2 - 1} - e^s$$

Compositions de fonctions

L'opérateur @ permet de composer les fonctions f et g en écrivant (la notation habituelle est $f \circ g$)

```
> eval(f),eval(g);
fg:=f@g;
fg(u); # c'est-à-dire f(g(u))
```

$$x \rightarrow x^2 - 1, x \rightarrow x - \sin(2x)$$

$$fg := f @ g$$

$$(u - \sin(2u))^2 - 1$$

Evidemment l'opérateur @ n'est pas commutatif (fog différent de gof)

```
> gf:=g@f;
gf(u);
(g@f)(u); # 2 écritures équivalentes
gf := g @ f
u^2 - 1 - \sin(2u^2 - 2)
u^2 - 1 - \sin(2u^2 - 2)
```

> (g@f-f@g)(u);

$$u^2 - \sin(2u^2 - 2) - (u - \sin(2u))^2$$

L'opérateur @@n permet la composition répétée n fois d'une fonction (n positif ou nul). **Attention** à la confusion de notation: celle du résultat affiché pour $f@@n$ est souvent utilisée dans les livres pour désigner la fonction dérivée n -ième de f

```
> eval(f);
f@@2;
x \rightarrow x^2 - 1
f^(2)
```

```
> (f@@2)(u); # c'est-à-dire f(f(u))
(sin@@2)(u);# c'est-à-dire sin(sin(u))
(sin@@2)(Pi/4);
```

$$(u^2 - 1)^2 - 1$$

$$\sin^{(2)}(u)$$

$$\sin\left(\frac{1}{2}\sqrt{2}\right)$$

Le résultat renvoyé pour $n = 0$ est la fonction identité : la fonction créée renvoie sous forme de suite (sequence) le ou les arguments donnés pour son évaluation. Ils sont désignés ici par le mot prédéfini **args** (pour arguments ; voir chapitre 19)

```
> Ident:=sin@@0;
Ident(x,y,z);
```

$$\begin{aligned} \text{Ident} &:= () \rightarrow \text{args} \\ &x, y, z \end{aligned}$$

Le paramètre n admet des valeurs négatives et MAPLE cherche alors la fonction réciproque qu'il compose n fois. En fait il ne faut pas se faire d'illusions car il ne fait que chercher des fonctions connues dans deux tables **invfunc** et **Invfunc** ! (voir l'aide en ligne).

```
> sin@@(-1);
(ln@@(-2))(u); # le résultat signifie exp(exp(u))
arcsin
exp^(2)(u)
```

S'il ne peut pas répondre il se contente d'une écriture symbolique (**erf** est une fonction connue de MAPLE).

```
> (erf@@(-1))(u);
erf^(-1)(u)
```

La fonction map

Nous avons déjà utilisé cette fonction sur des listes, des ensembles ou des matrices et c'est généralement l'usage que l'on en fait. Elle permet en fait d'appliquer un opérateur quelconque sur tout objet ayant des opérandes, même celles d'une expression, *son image étant un objet de même nature*. Afin de mieux comprendre sa signification générale nous l'appliquons ici dans un contexte particulier et à vrai dire rarement utilisé. Considérons l'expression

```
> restart;
ex:=ln(u)+3;
ex := ln(u) + 3
```

ainsi que la fonction

```
> f:=x->x^2+1;
f := x → x2 + 1
```

Appliquons l'opérateur f sur l'expression ex avec la fonction **map**

```
> map(f,ex);
ln(u)2 + 11
```

Ce résultat doit-être compris de la façon suivante : f s'applique d'abord sur $\ln(u)$, première opérande de l'expression, pour donner $\ln(u)^2 + 1$ puis sur 3 pour donner 9+1. L'expression ex étant une somme, **map** calcule finalement la somme de ces deux termes.

```

> op(ex);
f(op(1,ex));
f(op(2,ex));
%+%%;

```

$\ln(u), 3$
 $\ln(u)^2 + 1$
 10
 $\ln(u)^2 + 11$

On aurait pu aussi écrire directement

```

> map(x->x^2+1, ln(t)+3);

```

$\ln(t)^2 + 11$

L'opérateur **map** s'applique sur les opérandes de premier niveau d'un objet pour donner *un objet de même nature*. Ici on applique l'opérateur sur un intervalle, en fait sur ses deux opérandes, et on obtient un intervalle.

```

> map(x->exp(-x^2), infinity..0);

```

$0 .. 1$

Pour un opérateur dépendant de plusieurs variables, les opérandes remplacent le premier argument, les autres arguments étant *obligatoirement* fixés par des arguments supplémentaires de la fonction **map** (ici 1 est mis pour y et u pour z)

```

> map((x,y,z)->x^2+y+1/z, [1,2,3], 1, u);

```

$$\left[2 + \frac{1}{u}, 5 + \frac{1}{u}, 10 + \frac{1}{u} \right]$$

La fonction **zip**

Cette fonction permet d'appliquer un opérateur dépendant de *deux variables à deux listes* (pour les matrices ou les vecteurs voir la fonction **Zip** de **LinearAlgebra**, chapitre 4) en prenant successivement les opérandes de la première pour le premier argument et les opérandes de la seconde pour le deuxième. Le résultat est une liste dont *le nombre d'opérandes est égal à celui de la plus petite des deux listes*.

```

> restart:
l1:=[a,b,c,d,e,f,g,h];    l2:=[2,3,5];
zip((x,y)->x/y,l1,l2),   zip((x,y)->x/y,l2,l1);
l1:=[a, b, c, d, e, f, g, h]

```

$l2 := [2, 3, 5]$

$$\left[\frac{1}{2}a, \frac{1}{3}b, \frac{1}{5}c \right], \left[\frac{2}{a}, \frac{3}{b}, \frac{5}{c} \right]$$

Le résultat de l'opérateur que l'on introduit n'est pas nécessairement une opération numérique. On compose ici une liste de liste à partir de deux listes simples

```

> zip((x,y)->[y,x], l1, l2);

```

$[[2, a], [3, b], [5, c]]$

ou encore

```
> zip((x,y)->[x,x-y,y/2],l1[3..-1],l2);

$$\left[ [c, c - 2, 1], \left[ d, d - 3, \frac{3}{2} \right], \left[ e, e - 5, \frac{5}{2} \right] \right]$$

```

Exercice : Analysez les différentes opérations de cette instruction et expliquez le résultat ? Que vaudra le résultat si on change la valeur de n ?

```
> n:=5;
nops(convert(zip((x,y)->y+x,[seq(i,i=1..n)], [seq(n-i,i=1..n)]),set));
1
```

Expressions et fonctions définies par intervalles: la fonction piecewise

MAPLE permet, avec **piecewise** de définir des *expressions* composées par intervalles. Il suffit de donner dans l'ordre des arguments, un intervalle défini par une inégalité suivie de la définition de la fonction dans cet intervalle, et ce, autant de fois que nécessaire. Le dernier argument définit la fonction partout ailleurs (*otherwise*).

```
> restart;
f:=piecewise(x>=2,cos(2),x>-2,cos(x),exp(-abs(x+1)));
```

$$f := \begin{cases} \cos(2) & 2 \leq x \\ \cos(x) & -2 < x \\ e^{(-|x+1|)} & \text{otherwise} \end{cases}$$

Attention: comme on vient de le dire, **piecewise** ne définit pas une fonction mais seulement une *expression* composée par intervalles. Pour obtenir une fonction on doit utiliser soit **unapply**

```
> f:=unapply(f,x);
f(0), f(Pi);
f := x \rightarrow \begin{cases} \cos(2), & 2 \leq x \\ \cos(x), & -2 < x \\ e^{(-|x+1|)}, & \text{otherwise} \end{cases}
```

soit la syntaxe habituelle

```
> f:=x->piecewise(x>=2,cos(2),x>-2,cos(x),exp(-abs(x+1)));
f := x \rightarrow \begin{cases} \cos(2), & 2 \leq x \\ \cos(x), & -2 < x \\ e^{(-|x+1|)}, & \text{otherwise} \end{cases}
```

Attention: l'ordre dans lequel on donne les conditions a une importance. En effet la définition se lit de gauche à droite et de la façon suivante : *Si* la condition 1 est vraie alors expression 1, *si non si* la condition 2 est vraie alors expression 2, ... *si non* expression finale. Aussi, en inversant l'ordre des deux conditions précédentes on obtient un résultat différent car pour tout $x > -2$, la première condition sera satisfaite même si x est supérieur ou égal à 2 et la deuxième condition ne sera jamais prise en considération

```
> g:=x->piecewise(x>-2,cos(x),x>=2,cos(2),exp(-abs(x+1)));
g(0), g(Pi);
g := x \rightarrow \begin{cases} \cos(x), & -2 < x \\ \cos(2), & 2 \leq x \\ e^{(-|x+1|)}, & \text{otherwise} \end{cases}
```

1, -1

Les conditions peuvent aussi s'exprimer par des expressions. Ces fonctions ne seront toutefois manipulables (dérivation, simplification etc.) qu'à la condition quand même que MAPLE puisse résoudre ces inégalités ! (voir chapitre 14, *Equations ordinaires*)

```
> h:=x->piecewise(x^2-1>0,exp(x),exp(-x));
          h := x → piecewise(0 < x2 - 1, ex, e(-x))
```

Ceci permet aussi de construire des fonctions (ou des expressions) à plusieurs variables

```
> g:=(x,y)->piecewise(x^2+y<2,x+y/2,x-3*sin(y));
          g := (x, y) → piecewise(x2 + y < 2, x + 1/2 y, x - 3 sin(y))
```

Fonctions d'interpolation et de lissage. Approximations de fonctions

MAPLE dispose d'une bibliothèque **CurveFitting** contenant diverses fonctions d'interpolation (polynôme, fraction rationnelle, spline, etc.) ainsi qu'une fonction de lissage par un modèle linéaire au sens des moindres carrés.

Pour l'*approximation des fonctions* on se reportera à la bibliothèque **numapprox** (développements de Tchebycheff, approximation de Padé, etc.; voir un exemple ci-après).

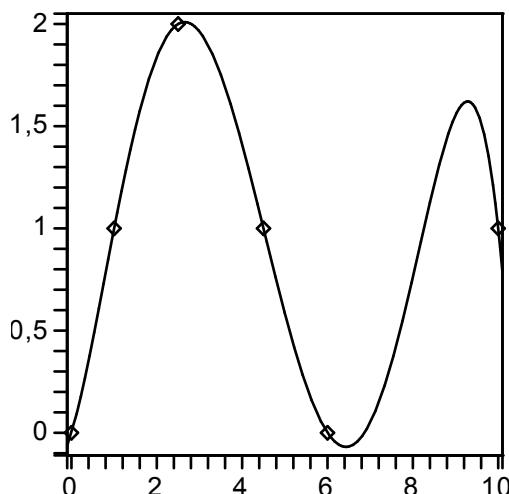
Polynôme d'interpolation

La fonction **PolynomialInterpolation** calcule l'*expression* du polynôme d'interpolation d'un ensemble de coordonnées (avec **RationalInterpolation** on obtiendra une fraction rationnelle). Deux formes d'appels sont possibles suivant la façon dont on dispose des abscisses et des ordonnées

1) Les arguments sont successivement *la liste des abscisses*, *la liste des ordonnées* et le nom (non assigné) choisi pour la variable.

```
> restart:
X:=[0,1,5/2,9/2,6,10]: Y:=[0,1,2,1,0,1]:
p:=CurveFitting[PolynomialInterpolation](X,Y,x);
          p := - 19/9450 x5 + 71/1575 x4 - 263/840 x3 + 21697/37800 x2 + 877/1260 x
```

On notera, pour comparer avec les splines du prochain paragraphe, "l'oscillation" du polynôme entre les abscisses $x=6$ et $x=10$



Pour transformer p en fonction il suffit d'écrire

```
> p:=unapply(p,x);

$$p := x \rightarrow -\frac{19}{9450}x^5 + \frac{71}{1575}x^4 - \frac{263}{840}x^3 + \frac{21697}{37800}x^2 + \frac{877}{1260}x$$

```

Exercice : quelles sont les vérifications effectuées par ces commandes ?

```
> evalb(degree(p(x))=nops(x)-1);
evalb({seq(Y[i]-p(X[i]), i=1..nops(X))}={0});
true
true
```

Il suffit qu'**une seule** valeur des abscisses ou des ordonnées soit décimale pour que les coefficients du polynôme soient décimaux.

```
> X:=evalf(X); # On rend décimales toutes les abscisses
p:=CurveFitting[PolynomialInterpolation](X,Y,x);

$$p := -0.002010582006x^5 + 0.04507936501x^4 + 0.6960317466x - 0.3130952376x^3 + 0.5739947080x^2$$

```

2) Les arguments sont successivement une *liste de listes donnant les couples (abscisse,ordonnée)* et le nom (non assigné) choisi pour la variable. On profite de cet exemple pour rappeler comment, à partir des deux listes X et Y précédentes, on peut construire une telle liste de listes avec la fonction **zip**

```
> Lxy:=zip((x,y)->[x,y],X,Y);
p:=CurveFitting[PolynomialInterpolation](Lxy,x);
Lxy := [[0., 0], [1., 1], [2.5, 0], [4.5, 0], [6., 0], [10., 1]]

$$p := -0.002010582006x^5 + 0.04507936501x^4 + 0.6960317466x - 0.3130952376x^3 + 0.5739947080x^2$$

```

Fonctions splines

La fonction **Spline** calcule avec les mêmes syntaxes que **PolynomialInterpolation** une fonction spline de degré 3 par défaut (on peut augmenter ou diminuer le degré n par un quatrième argument $degree=n$).

Rappel : une fonction spline de degré n est une fonction d'interpolation définie entre les points du support d'interpolation par des polynômes de degré au plus égal à n . Cette fonction est, par construction, $n - 1$ fois continuement dérivable partout sur l'intervalle d'interpolation. Une spline de degré 3 possède donc des dérivées première et seconde continues sur l'intervalle d'interpolation et notamment aux points de raccordement des polynômes.

Comme pour **PolynomialInterpolation**, si une ou plusieurs des abscisses ou ordonnées sont décimales les coefficients des polynômes seront tous décimaux. On aurait pu utiliser aussi la liste de listes Lxy pour donner les abscisses et ordonnées. La fonction **simplify** est uniquement destinée à améliorer la présentation du résultat.

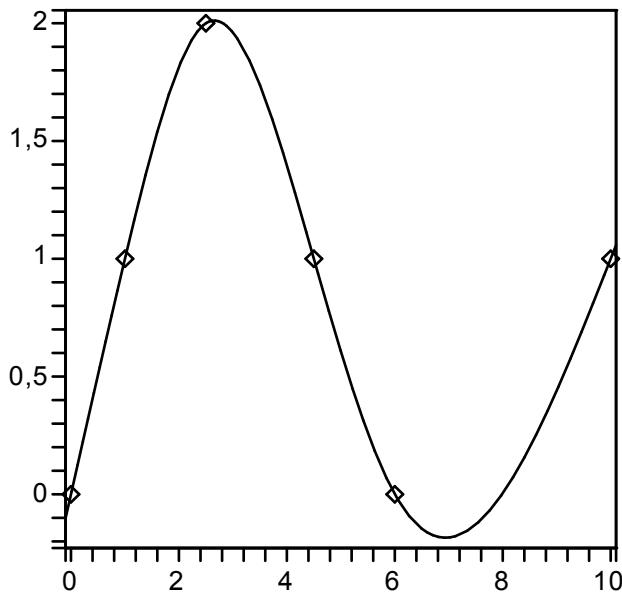
```
> S:=simplify(CurveFitting[Spline](X,Y,x));

$$S := \begin{cases} 1.017307638x - 0.01730763836x^3 & x < 1. \\ 0.08084066558 + 0.7747856416x + 0.2425219968x^2 - 0.09814830397x^3 & x < 2.5 \\ -2.781107990 + 4.209124029x - 1.131213358x^2 + 0.08501641000x^3 & x < 4.5 \\ 0.2833880342 + 2.166126679x - 0.6772139471x^2 + 0.05138682400x^3 & x < 6. \\ 15.84242195 - 5.613390278x + 0.6193722125x^2 - 0.02064574042x^3 & 6. \leq x \end{cases}$$

```

Par comparaison avec le polynôme d'interpolation obtenu au paragraphe précédent, on obtient un résultat qui

semble plus "naturel" entre $x = 6$ et $x = 10$. Un autre exemple de comparaison entre splines et polynôme d'interpolation est donné au chapitre 16, *Représentations graphiques 2D*, § *Ajout d'un titre*, ...



Lissage au sens des moindres carrés

La fonction **LeastSquares** calcule une fonction de *lissage linéaire* par rapport aux paramètres à déterminer.

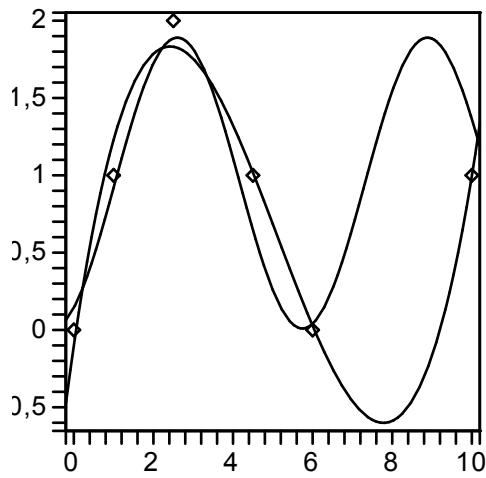
Rappel : "linéaire" ne signifie pas nécessairement lissage par une droite. Le polynôme défini ci-dessous par l'option **curve** est bien linéaire par rapport aux paramètres a , b , c et d . De même $curve = a \cos(x) + b \sin(x) + c$ est aussi linéaire par rapport à a , b et c . Si la fonction n'est pas précisée **curve** sera affine en x (c'est-à-dire $curve = a x + b$).

Si au moins un des éléments de X ou Y est décimal, le résultat sera exprimé avec des coefficients décimaux. On peut utiliser aussi une liste de listes pour les abscisses et ordonnées.

```
> Sq1:=CurveFitting[LeastSquares]
      (X,Y,x,curve=a*x^3+b*x^2+c*x+d);
Sq1 := -0.1018325524 + 1.78433500847590998 x - 0.483370430499256542 x2
      + 0.0315905726140406768 x3

> Sq2:=CurveFitting[LeastSquares]
      (X,Y,x,curve=a*cos(x)+b*sin(x)+c);
Sq2 := 0.9497203132 - 0.806617047602295911 cos(x) + 0.483223376172294494 sin(x)
```

Dire qu'un des lissages est meilleur que l'autre n'a, en soi, pas de signification si on n'est pas capable de donner une justification claire au choix que l'on a fait pour la fonction (par exemple une loi résultant d'une théorie physique pour un lissage de mesures).



Il existe également un option de **CurveFitting, Interactive**, offrant une applet Java permettant ces traitements de façon interactive.

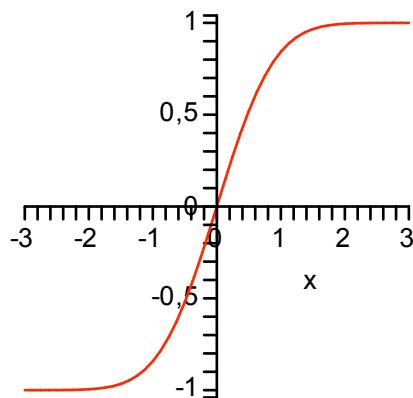
Approximation d'une fonction

MAPLE possède toutes les approximations nécessaires pour calculer les approximations numériques de ses propres fonctions avec **evalf**. Mais c'est aussi un outil d'une grande efficacité pour calculer des approximations de fonctions que l'on veut implémenter dans un code numérique écrit en C, Fortran, Java, etc.

On construit ici un exemple avec la fonction **erf** définie par une intégrale et utilisée en physique ou en théorie des probabilités. On veut pouvoir calculer cette fonction dans un programme écrit en C et pour laquelle on ne dispose pas de bibliothèque toute prête.

```
> restart;
FunctionAdvisor(definition,erf)[1]; # voir ch. 13
plot(erf(x),x=-3..3); # voir ch. 16
```

$$\text{erf}(z) = \frac{2}{\sqrt{\pi}} \int_0^z \frac{1}{e^{(\underline{k}l)^2}} d_{kl}$$



La fonction **erf** étant impaire, on ne considérera que les x positifs ou nuls. Un développement asymptotique

montre qu'elle converge très rapidement vers 1 comme $1 - e^{(-x^2)/x\sqrt{\pi}}$

asympt(erf(x), x); # voir ch. 11

$$1 + \frac{-\frac{1}{\sqrt{\pi}x} + \frac{1}{2\sqrt{\pi}x^3} - \frac{3}{4\sqrt{\pi}x^5} + O\left(\frac{1}{x^7}\right)}{e^{(x^2)}}$$

Nous allons donc considérer la fonction $e^{(x^2)}(1 - erf(x))$. Un développement en série autour de l'origine montre que cette nouvelle fonction peut être approchée par un polynôme

> series(exp(x^2)*(1-erf(x)), x); # voir ch. 11

$$1 - \frac{2}{\sqrt{\pi}}x + x^2 - \frac{4}{3\sqrt{\pi}}x^3 + \frac{1}{2}x^4 - \frac{8}{15\sqrt{\pi}}x^5 + O(x^6)$$

Il est bien connu en analyse numérique qu'il est plus efficace d'approcher une fonction, pour une précision numérique fixée, par un développement sur une base de polynômes de Chebyshev (ou Tchebycheff, ou autres orthographes) que d'utiliser un développement limité. C'est le propos de la fonction **chebyshev** de la bibliothèque **numapprox** (cette bibliothèque contient d'autres méthodes comme les *approximants de Padé*). On se fixe pour précision absolue la valeur 10^{-6} . On va utiliser pour domaine d'approximation l'intervalle $[0,3]$, considérant que pour $3 < x$ le comportement de la fonction est dominée (pour l'erreur considérée) par son comportement exponentiel.

> q:=numapprox[chebyshev](exp(x^2)*(1-erf(x)), x=0..3, 1.0e-6);

$$\begin{aligned} q := & 0.4413272100 T\left(0, \frac{2}{3}x - 1\right) - 0.3611037989 T\left(1, \frac{2}{3}x - 1\right) + 0.1327716736 T\left(2, \frac{2}{3}x - 1\right) \\ & - 0.04486270820 T\left(3, \frac{2}{3}x - 1\right) + 0.01413370748 T\left(4, \frac{2}{3}x - 1\right) - 0.004193978073 T\left(5, \frac{2}{3}x - 1\right) \\ & + 0.001181031233 T\left(6, \frac{2}{3}x - 1\right) - 0.0003174531705 T\left(7, \frac{2}{3}x - 1\right) \\ & + 0.00008182442894 T\left(8, \frac{2}{3}x - 1\right) - 0.00002030029272 T\left(9, \frac{2}{3}x - 1\right) \\ & + 0.000004862871502 T\left(10, \frac{2}{3}x - 1\right) - 0.000001127713989 T\left(11, \frac{2}{3}x - 1\right) \\ & + 2.537467914 \cdot 10^{-7} T\left(12, \frac{2}{3}x - 1\right) \end{aligned}$$

Le développement fait intervenir les polynômes de Chebyshev sous la forme symbolique $T(n, x)$ dont le nom correspond à l'ancienne bibliothèque **orthopoly** déclarée obsolète (**numapprox** n'a pas suivi cette évolution et devra être mis à jour par Maplesoft). Si on veut un polynôme classique il faut faire un développement en remplaçant T par le nouveau nom

> Q:=expand(subs(T=ChebyshevT, q));

$$\begin{aligned} Q := & -0.2878029766 x^5 - 0.7509833719 x^3 + 0.4947709638 x^4 + 0.02004984468 x^8 - 1.128370890 x \\ & - 0.00008544588298 x^{11} + 0.0008312105112 x^{10} + 0.1449432893 x^6 + 0.000004005303250 x^{12} \\ & - 0.004920744202 x^9 + 0.9998374004 x^2 - 0.06068958711 x^7 + 0.9999999297 \end{aligned}$$

ou encore en écrivant

```
> with(orthopoly,T):  
Q:=expand(q);  

$$Q := -0.2878029766 x^5 - 0.7509833718 x^3 + 0.9999999296 + 0.4947709636 x^4 + 0.02004984468 x^8  
- 1.128370890 x - 0.00008544588300 x^{11} + 0.0008312105113 x^{10} + 0.1449432892 x^6  
+ 0.000004005303250 x^{12} - 0.004920744203 x^9 + 0.9998374004 x^2 - 0.06068958712 x^7$$

```

Il reste à transformer éventuellement cette expression en fonction avec **unapply**.

Rappel : l'évaluation numérique d'un polynôme sous la forme d'une somme de monômes est numériquement inefficace et on utilisera toujours la **forme de Horner**. Pour un polynôme $a x^3 + b x^2 + c x + d$ on écrira $((a x + b) x + c) x + d$ qui remplacent des exponentiations coûteuses (et au final peu précises) par un nombre minimal de multiplications. La fonction **convert(...,horner)** sait faire ce travail (voir aussi **numapprox[hornerform]**)

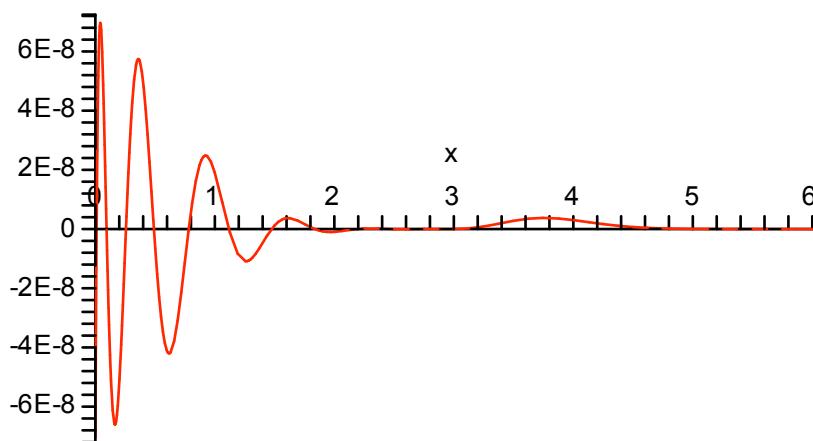
```
> erf_a:=unapply(1-exp(-x^2)*convert(Q,horner),x);  
erf_a := x → 1 - e(-x2) (0.9999999296 + (-1.128370890 + (0.9998374004 + (-0.7509833718 +  
(0.4947709636 + (-0.2878029766 + (0.1449432892 + (-0.06068958712 + (0.02004984468  
+ (-0.004920744203 + (0.0008312105113 + (-0.00008544588300 + 0.000004005303250 x) x) x) x) x) x)  
x) x) x) x) x) x)
```

Comparons des valeurs données par MAPLE et par notre approximation

```
> evalf(erf(1/2)), evalf(erf_a(1/2));  
evalf(erf(3.5)), evalf(erf_a(3.5));  
0.5204998778, 0.5204998836  
0.9999992569, 0.9999992541
```

Les outils graphiques de MAPLE permettent d'un coup d'oeil de vérifier la qualité de l'approximation en tracant la fonction d'erreur. On note que l'erreur absolue est toujours inférieure ou au plus de l'ordre de 6.5×10^{-8} . **Attention**, on fait ici implicitement l'hypothèse que l'approximation numérique de **erf** donnée par MAPLE est meilleure que celle donnée par la fonction **erf_a** que l'on vient de construire ! C'est sensé être le cas puisque **Digits** = 10.

```
> plot(erf(x)-erf_a(x),x=0..6);
```



On invoque ensuite le générateur de code C de la bibliothèque **CodeGeneration**. Un copié-collé du résultat dans le code source C, une petite modification pour prendre en compte les x négatifs et le tour est joué ! Le code généré par l'option **Fortran** est un peu "vieillot" (mais juste) en regard des normes F90/95/2003. On n'oubliera pas ensuite, pour vérification, de comparer quelques résultats obtenus par le code avec ceux donnés par MAPLE.

```
> CodeGeneration['C'](erf_a);
Warning, procedure/module options ignored
#include <math.h>

double erf_a (double x)
{
    return(0.1e1 - exp(-x * x) * (0.9999999296e0 + (-0.1128370890e1 +
(0.9998374004e0 + (-0.7509833718e0 + (0.4947709636e0 + (-0.2878029766e0 +
(0.1449432892e0 + (-0.6068958712e-1 + (0.2004984468e-1 + (-0.4920744203e-2 +
(0.8312105113e-3 + (-0.8544588300e-4 + 0.4005303250e-5 * x) * x) * x) * x) *
x) * x) * x) * x) * x));
}
```

Fonctions et procédures

Les fonctions définies jusqu'ici sont en fait des **procédures** écrites avec une notation simplifiée. Comparons les deux écritures

```
> restart;
f:=(x,y)->sin(x/y);
```

$$f := (x, y) \rightarrow \sin\left(\frac{x}{y}\right)$$

et

```
> g:=proc(x,y) sin(x/y) end proc;
```

$$g := \text{proc}(x, y) \sin((x)/(y)) \text{end proc};$$

Cette forme doit se lire : "*g est le nom d'une procédure ayant 2 arguments x et y et dont la seule instruction est sin(x/y). Fin de la procédure*" (voir chapitre 19, *Procédures*). On l'active en donnant son nom et les arguments associés, exactement comme pour les constructions précédentes. Comparons *f* et *g* :

```
> f(Pi,4), g(Pi,4);
```

$$\frac{1}{2} \sqrt{2}, \frac{1}{2} \sqrt{2}$$

Il suffit de rajouter les options **operator** et **arrow** (flèche) pour avoir une construction strictement identique à *f*

```
> g:=proc(x,y) option operator, arrow; sin(x/y) end proc;
```

$$g := (x, y) \rightarrow \sin\left(\frac{x}{y}\right)$$

Sans rien savoir de la programmation on comprend aisément l'écriture suivante (sachant que **if** signifie *si*, **then** *alors*, **else** *sinon* et **end if** "fin du bloc if")

```
> h:=proc(x) option operator, arrow; if x = 0 then 1 else sin(x)/x end if
end proc;
```

```

h(0),h(Pi/4);

$$h := x \rightarrow \text{if } x = 0 \text{ then } 1 \text{ else } \frac{\sin(x)}{x} \text{ end if;}$$


$$1, \frac{2\sqrt{2}}{\pi}$$


```

Opérandes d'une fonction ; tables *Cache* et *remember*

Ce paragraphe s'adresse aux lecteurs intéressés par une introduction aux mécanismes d'évaluation utilisés par les fonctions. Les fonctions internes de MAPLE telles **sin** ou **ln** possèdent 8 opérandes

```

> restart;
nops(eval(ln));
8

```

Il en est de même des fonctions créées par l'utilisateur

```

> F:=(a,x)->a*x^2-1;
nops(eval(F));

$$F := (a, x) \rightarrow a x^2 - 1$$

8

```

Nous ne nous intéresserons ici qu'à deux d'entre elles, i.e. 1 et 4, les autres étant d'un intérêt secondaire dans ce chapitre. Elles seront sommairement décrites au chapitre 19, *Procédures et modules*.

L'opérande 1 est simplement la liste des arguments avec éventuellement leurs types autorisés. Le type **algebraic** rencontré ici est un type assez général qui accepte les noms, les expressions, les nombres, etc. Il exclut par exemple les listes, les ensembles ou les chaînes de caractères. On remarquera que la fonction **eval** appliquée à une fonction de MAPLE ne donne pas le contenu de la procédure, mais on peut l'obtenir dans certains cas (voir chapitre 19, *Procédures*).

```

> eval(ln);
op(1,eval(ln));
op(1,eval(F));
proc(x::algebraic) ... end proc;
x::algebraic
a, x

```

On notera que c'est **eval(ln)** ou **eval(F)** qui possède 8 opérandes, **ln** ou **F** ne sont que des noms qui font référence à une seule opérande qui est la procédure elle-même.

```

> nops(ln);
op(ln);
1
proc(x::algebraic) ... end proc;

```

Les fonctions prédefinies de MAPLE

L'opérande 4 des fonctions prédefinies est beaucoup plus intéressante car elle nous entraîne au cœur du fonctionnement de MAPLE. C'est une table dite **Cache**. Ce type de table est une évolution des anciennes tables **remember** ("se souvenir") qui restent encore en service avec certaines fonctions. Demandons à

MAPLE de calculer

```
> cos(Pi/4);
```

$$\frac{1}{2}\sqrt{2}$$

L'expression est évaluée parce que MAPLE connaît le résultat après l'avoir lu dans la table **Cache** de la fonction **cos** ! Le petit "miracle" du résultat précédent ainsi dévoilé ressemble à un banal tour de passe-passe déguisé en calcul intelligent. Bien sûr ce n'est pas si simple. Commençons par examiner la table **Cache** de cette fonction

```
> op(4,eval(cos));
```

```
Cache( 512, 'permanent' = [ 0 = 1,  $\frac{1}{2}\pi = 0$ ,  $\pi = -1$ ,  $-\infty = \text{undefined}$ ,  $\frac{1}{4}\pi = \frac{1}{2}\sqrt{2}$ ,  $\frac{1}{3}\pi = \frac{1}{2}$ ,  $\frac{1}{6}\pi = \frac{1}{2}\sqrt{3}$ ,  
I =  $\cosh(1)$ ,  $\infty = \text{undefined}$  ] )
```

La première valeur entière fixe une taille limite à la table (c'est toujours une puissance de 2). Le mot clé '**permanent**' s'oppose à '**temporary**' (voir plus loin) et identifie une liste qui définit un ensemble d'arguments tels $0, \pi, \infty$, etc., à gauche des signes = et pour lesquels la fonction possède des résultats remarquables et connus et placés à droite des signes =. Ces tables **Cache** sont de type **table** avec le type supplémentaire **cache** et peuvent être créées et manipulées avec la bibliothèque **Cache** indépendamment des fonctions (voir **?Cache**).

```
> type(op(4,eval(cos)),table);
```

```
type(op(4,eval(cos)),cache);
```

true

true

On comprend dès lors facilement ce "calcul"

```
> cos(I);
```

cosh(1)

Mais demandons maintenant

```
> cos(3*Pi/4);
```

$$-\frac{1}{2}\sqrt{2}$$

L'argument n'existe pas dans les index de la table mais MAPLE renvoie quand même le résultat. La procédure **cos** a "reconnu" dans un premier temps, grâce à ses algorithmes, que

$$\cos\left(\frac{3\pi}{4}\right) = \cos\left(\pi - \frac{\pi}{4}\right) = -\cos\left(\frac{\pi}{4}\right)$$

Elle s'appelle ensuite elle-même (appel récursif) pour finalement donner le résultat: -1*"résultat contenu dans la table Cache". Les procédures associées aux fonctions de MAPLE travaillent ainsi sur les propriétés mathématiques de ces fonctions. On aura compris que "reconnaître" avec un algorithme est plus difficile que lire une table et, bien entendu, de telles procédures ne s'écrivent pas en une ligne comme celles que nous avons définies précédemment. Nous en verrons quelques éléments d'écriture aux chapitres 18, *Eléments de programmation* et 19, *Procédures et Modules*.

Observons maintenant la nouvelle table **Cache** de **cos**. On remarque l'ajout d'une nouvelle liste dite '**temporary**'. Elle enregistre les résultats que la fonction calcule au fur et à mesure (dans les limites de la taille

de la table, sinon des valeurs sont éliminées). *Avant d'effectuer un calcul la fonction explore sa table Cache et affiche directement le résultat si elle le trouve.*

```
> op(4,eval(cos));
```

$$\text{Cache}\left(512, \text{'temporary'} = \left[\frac{3}{4}\pi = -\frac{1}{2}\sqrt{2} \right], \text{'permanent'} = \left[0 = 1, \infty = \text{undefined}, \pi = -1, I = \cosh(1), \frac{1}{6}\pi = \frac{1}{2}\sqrt{3}, \frac{1}{2}\pi = 0, \frac{1}{4}\pi = \frac{1}{2}\sqrt{2}, -\infty = \text{undefined}, \frac{1}{3}\pi = \frac{1}{2} \right] \right)$$

Comme il a déjà été dit, MAPLE contient une bibliothèque, appelée aussi **Cache**, qui exporte des modules permettant la gestion des tables de type *Cache*. On utilisera les deux fonctions de lecture des "arguments" et des "résultats" de la table temporaire pour limiter l'affichage

```
> Cache:-TemporaryIndices(op(4,eval(cos)));
```

```
Cache:-TemporaryEntries(op(4,eval(cos)));
```

$$\left[\frac{3}{4}\pi \right]$$

$$\left[-\frac{1}{2}\sqrt{2} \right]$$

Exercice : on peut faire mieux. Les amateurs peuvent chercher l'explication...

```
> Temporaire:=f->op(zip((x,y)->op(x)=op(y),
  [Cache:-TemporaryIndices(op(4,eval(f)))],
  [Cache:-TemporaryEntries(op(4,eval(f))))]));
Permanent:=f->op(zip((x,y)->op(x)=op(y),
  [Cache:-PermanentIndices(op(4,eval(f)))],
  [Cache:-PermanentEntries(op(4,eval(f))))]):
```

```
> Temporaire(cos);
```

$$\frac{3}{4}\pi = -\frac{1}{2}\sqrt{2}$$

```
> Permanent(cos);
```

$$0 = 1, \infty = \text{undefined}, \pi = -1, I = \cosh(1), \frac{1}{6}\pi = \frac{1}{2}\sqrt{3}, \frac{1}{2}\pi = 0, \frac{1}{4}\pi = \frac{1}{2}\sqrt{2}, -\infty = \text{undefined}, \frac{1}{3}\pi = \frac{1}{2}$$

Une fonction répond souvent par un résultat non évalué. Avant d'afficher une réponse elle cherche à donner un résultat évalué et utilise du temps de calcul

```
> cos(Pi/15);
```

$$\cos\left(\frac{1}{15}\pi\right)$$

Aussi, même ce type de résultat est enregistré dans la table *Cache* et la fonction donnera directement le résultat sans calcul si on l'appelle à nouveau avec cet argument

```
> Temporaire(cos);
```

$$\frac{3}{4}\pi = -\frac{1}{2}\sqrt{2}, \frac{1}{15}\pi = \cos\left(\frac{1}{15}\pi\right)$$

Ceci ne concerne pas les résultats décimaux (voir à la fin de ce sous-paragraphe)

```
> cos(2.3176875);
Temporaire(cos);
-0.6793607758
```

$$\frac{3}{4}\pi = -\frac{1}{2}\sqrt{2}, \frac{1}{15}\pi = \cos\left(\frac{1}{15}\pi\right)$$

On peut forcer **cos** à donner un résultat sous forme de combinaison de radicaux (s'il est calculable et si **convert** sait le faire!)

```
> convert(cos(Pi/15), radical);

$$\frac{1}{8}\sqrt{2}\sqrt{3}\sqrt{5+\sqrt{5}} + \frac{1}{8}\sqrt{5} - \frac{1}{8}$$

```

Ce calcul n'a pas modifié la liste temporaire (ni permanente) de la table *Cache*

```
> Temporaire(cos);

$$\frac{3}{4}\pi = -\frac{1}{2}\sqrt{2}, \frac{1}{15}\pi = \cos\left(\frac{1}{15}\pi\right)$$

```

mais on peut ajouter ce résultat à la liste temporaire par une simple assignation

```
> cos(Pi/15):=convert(cos(Pi/15), radical);

$$\cos\left(\frac{1}{15}\pi\right) := \frac{1}{8}\sqrt{2}\sqrt{3}\sqrt{5+\sqrt{5}} + \frac{1}{8}\sqrt{5} - \frac{1}{8}$$

```

```
> Temporaire(cos);

$$\frac{3}{4}\pi = -\frac{1}{2}\sqrt{2}, \frac{1}{15}\pi = \frac{1}{8}\sqrt{2}\sqrt{3}\sqrt{5+\sqrt{5}} + \frac{1}{8}\sqrt{5} - \frac{1}{8}$$

```

Alors la fonction **cos** donnera directement la réponse

```
> cos(Pi/15);

$$\frac{1}{8}\sqrt{2}\sqrt{3}\sqrt{5+\sqrt{5}} + \frac{1}{8}\sqrt{5} - \frac{1}{8}$$

```

Attention : ce mécanisme n'interdit pas les bêtises...

```
> cos(0):=2;      # !!!!
Temporaire(cos);
cos(0):=2
0 = 2,  $\frac{3}{4}\pi = -\frac{1}{2}\sqrt{2}, \frac{1}{15}\pi = \frac{1}{8}\sqrt{2}\sqrt{3}\sqrt{5+\sqrt{5}} + \frac{1}{8}\sqrt{5} - \frac{1}{8}$ 
```

On observe que les deux listes '*temporary*' et '*permanent*' de la table *Cache* contiennent chacune un résultat associé à l'argument 0. Or c'est le résultat de la table temporaire qui est utilisé en priorité et... c'est la catastrophe !

```
> cos(0)^2+sin(0)^2;
```

4

Et on s'empresse de rétablir une situation normale

```
> cos(0):=1;
Temporaire(cos);
```

```

cos(0)^2+sin(0)^2; # Ouf !
cos(0) := 1
0 = 1,  $\frac{3}{4}\pi = -\frac{1}{2}\sqrt{2}$ ,  $\frac{1}{15}\pi = \frac{1}{8}\sqrt{2}\sqrt{3}\sqrt{5+\sqrt{5}} + \frac{1}{8}\sqrt{5} - \frac{1}{8}$ 
1

```

Remarque: toutes les fonctions de MAPLE ne fonctionnent pas encore en mode **Cache**. Certaines possèdent seulement une table **remember** ordinaire mémorisant par défaut les valeurs remarquables (aucun résultat nouveau n'est enregistré *automatiquement*)

```

> op(4,eval(Ei)); # Ei : fonction exponentielle intégrale
Ei(3);
op(4,eval(Ei));
table([(1, infinity) = 0, infinity = infinity, (1, -infinity) = -infinity - I*pi, -infinity = 0])
Ei(3)
table([(1, infinity) = 0, infinity = infinity, (1, -infinity) = -infinity - I*pi, -infinity = 0])

```

D'autres fonctionnent en mode *option remember*, c'est à dire mettent en mémoire les résultats calculés, mais dans une table ordinaire. En y regardant de près, on notera que pour **evalf** seules sont conservées les valeurs qui demandent des calculs élaborés....

```

> op(4,eval(evalf)); # voir plus haut
evalf[20](Pi/3);
evalf[3](3/5+6/7);
op(4,eval(evalf));
table([cos(2.3176875) = (-0.6793607758, 10)])
1.0471975511965977462
1.46
table([cos(2.3176875) = (-0.6793607758, 10), pi = (3.1415926535897932385, 20)])

```

Fonction définies par l'utilisateur

Les fonctions définies par l'utilisateur possèdent aussi 8 opérandes et l'opérande 4 est une table, dite **remember**, qui est vide a priori. Considérons par exemple la fonction suivante qui n'est pas définie pour $x=1$

```

> f:=x->exp(-1/abs(x-1));
f(1);
f := x → e
$$\left( -\frac{1}{|x-1|} \right)$$


```

```
Error, (in f) numeric exception: division by zero
```

Pour l'instant, l'opérande 4 de f existe mais vaut **NULL**, ce qui explique que **op** n'affiche rien

```
> op(4,eval(f));
```

Pour remplir la table **remember** il suffit d'écrire $f(valeur\ particuli\`ere):=r\`esultat$; autant de fois que nécessaire et de la même façon que nous l'avions fait pour **cos**. Par exemple, les limites à droite et à gauche de $f(x)$ pour $x = 1$ valent 0. On peut rendre continue sur R cette fonction en imposant que $f(1) = 0$ et en écrivant

```
> f(1):=0;
f(1);
```

```
f(2);
f(1) := 0
0
e(-1)
```

La table **remember** de *f* est maintenant

```
> op(4, eval(f));
table([1 = 0])
```

La table **remember** a le type **table** mais pas le type **cache** (c'est une table ordinaire). **Attention**, la présence de la **table remember** ne signifie pas un fonctionnement en mode **option remember** (voir plus bas). En effet, la table ne retient ici que les résultats imposés, pas les résultats calculés (le résultat de *f*(2) n'est pas dans la table)

```
> type(op(4, eval(f)), table);
type(op(4, eval(f)), cache);
true
false
```

On peut imposer à une fonction de travailler en enregistrant les résultats calculés avec une table **Cache** (comme les fonctions prédéfinies de MAPLE). On écrira pour cela

```
> F:=proc(x) option operator, arrow, cache; exp(-1/abs(x-1)) end proc;
F(1):=0;
F(2);

F := x → e $\left( -\frac{1}{|x - 1|} \right)$ 
F(1) := 0
e(-1)

> op(4, eval(F));
Temporaire(F);

Cache(512, 'temporary' = [1 = 0, 2 = e(-1)])
1 = 0, 2 = e(-1)
```

On peut remplacer **cache** par **remember** et le fonctionnement utilisera une table ordinaire. **Attention** car alors, contrairement à la table **Cache** décrite plus haut, tous les résultats seront maintenant mémorisés dans cette table. Ce mode de fonctionnement doit être évité pour les fonctions facilement calculables car la table n'a pas, contrairement à une table **Cache**, de taille limite et la fonction utilisera, après un usage intensif, beaucoup de mémoire et passera plus de temps à scruter la table qu'à faire les calculs !

```
> Frm:=proc(x) option operator, arrow, remember; exp(-1/abs(x-1)) end
proc;
Frm(1):=0;
Frm(2);

Frm := x → e $\left( -\frac{1}{|x - 1|} \right)$ 
```

Frm(1) := 0

e⁽⁻¹⁾

Remarque : on peut imaginer ici de définir la fonction *F* par morceaux

```
> F_m:=x->piecewise(x=1,0,exp(-1/abs(x-1)));
F_m(1);
F_m(2);
```

$$F_m := x \rightarrow \begin{cases} e^{-\frac{1}{|x-1|}} & x = 1, 0 \\ 0 & \end{cases}$$

e⁽⁻¹⁾

L'utilisation des tables ***Cache*** ou ***remember*** permet également de faire apparaître, pour simplifier l'écriture des résultats, des valeurs particulières des fonctions sous une forme symbolique. Supposons que les arguments $\sqrt{2}/2$ et $-\sqrt{2}/2$ soient particuliers pour l'utilisateur de la fonction *F* dans le cadre de son travail. Il pourra alors écrire

```
> F(sqrt(2)/2) :=alpha;
F(-sqrt(2)/2) :=beta;
Temporaire(F);
```

$$1 = 0, 2 = e^{(-1)}, -\frac{1}{2}\sqrt{2} = \beta, \frac{1}{2}\sqrt{2} = \alpha$$

et les résultats des calculs pourront apparaître sous une forme symbolique simplifiée

```
> 2*F(sin(Pi/4))-F(cos(3*Pi/4))/3;
```

$$2\alpha - \frac{1}{3}\beta$$

La fonction forget

La fonction **forget** rend **NULL** la table ***Cache*** ou ***remember*** d'une fonction créée par l'utilisateur

```
> op(4,eval(F));
```

$$Cache\left(512, 'temporary' = \left[1 = 0, 2 = e^{(-1)}, -\frac{1}{2}\sqrt{2} = \beta, \frac{1}{2}\sqrt{2} = \alpha \right] \right)$$

```
> forget(F);
op(4,eval(F)); # L'opérande vaut NULL et rien n'est affiché
```

On est donc ramené au stade initial de la création de la fonction

```
> 2*F(sin(Pi/4))-F(cos(3*Pi/4))/3;
```

$$2e^{\left(-\frac{1}{2}\sqrt{2} + 1\right)} - \frac{1}{3}e^{\left(-\frac{1}{2}\sqrt{2} + 1\right)}$$

```
> F(1);
```

```
Error, (in F) numeric exception: division by zero
```

Pour les fonctions prédéfinies de MAPLE fonctionnant en mode *Cache*

```
> op(4,eval(cos));
```

```
Cache( 512, 'temporary' = [ 0 = 1,  $\frac{3}{4}\pi = -\frac{1}{2}\sqrt{2}$ ,  $\frac{1}{15}\pi = \frac{1}{8}\sqrt{2}\sqrt{3}\sqrt{5+\sqrt{5}}$  +  $\frac{1}{8}\sqrt{5}-\frac{1}{8}$  ], 'permanent'  
[ 0 = 1, I = cosh(1),  $\frac{1}{6}\pi = \frac{1}{2}\sqrt{3}$ ,  $\infty = undefined$ ,  $\frac{1}{3}\pi = \frac{1}{2}$ ,  $\frac{1}{2}\pi = 0$ ,  $-\infty = undefined$ ,  $\frac{1}{4}\pi = \frac{1}{2}\sqrt{2}$ ,  
 $\pi = -1$  ] )
```

seule la table '**temporary**' est effacée (heureusement !)

```
> forget(cos);  
op(4,eval(cos));
```

```
Cache( 512, 'permanent' = [ 0 = 1, I = cosh(1),  $\frac{1}{6}\pi = \frac{1}{2}\sqrt{3}$ ,  $\infty = undefined$ ,  $\frac{1}{3}\pi = \frac{1}{2}$ ,  $\frac{1}{2}\pi = 0$ ,  
 $-\infty = undefined$ ,  $\frac{1}{4}\pi = \frac{1}{2}\sqrt{2}$ ,  $\pi = -1$  ] )
```

Types associés à une fonction

Il existe une opérande 0 qui contient le type de base d'une fonction qui est avant tout une procédure. C'est ce type que renvoie la fonction **whattype**

```
> eval(f);  
op(0,eval(f));  
whattype(eval(f));
```

$$x \rightarrow e^{\left(-\frac{1}{|x-1|}\right)}$$

procedure
procedure

Exercice : que signifie ce résultat ?

```
> whattype(f);  
symbol
```

Les fonctions construites avec la syntaxe \rightarrow possèdent en plus un type **operator** et un sous type **arrow** (flèche)

```
> type(eval(f),operator);  
type(eval(f),operator(arrow));
```

true

true

La fonction définie ci-dessous possède bien sûr le type *procedure* mais pas les autres sous-types

```
> g:=proc(x,y) sin(x/y) end proc;  
type(eval(g),procedure);  
type(eval(g),operator);  
g := proc(x, y) sin((x)/(y)) end proc;  
true  
false
```

Attention: le type *function* est retourné pour l'*expression* d'une fonction interne de MAPLE non évaluée comme $\ln(x)$, $\sin(x)$, etc. ou une fonction non évaluée créée par l'utilisateur avec la syntaxe $->$. Le choix du terme *function* pour désigner un tel type n'est pas très approprié.

```
> whattype(f(x-1)),whattype(ln(x^2));  
function, function
```

L'*expression* $f(x)$ prise par une fonction f ne possède naturellement plus le type *operator*. En tant qu'*expression* elle portent aussi le type *algebraic*.

```
> type(f(x),operator), type(f(x),algebraic);  
type(ln(x^2),operator), type(ln(x^2),algebraic);  
false, true  
false, true
```

Distributions

Voir le chapitre 10, *Intégration*.

8 - Déivation

La fonction diff

La fonction **diff** calcule *l'expression dérivée d'une expression* dépendant d'une ou de plusieurs variables. Le premier argument donne l'expression à dériver, le second la variable par rapport à laquelle on dérive. *Le résultat de diff est une expression, pas une fonction.*

```
> restart;
diff(sin(x^2+cos(x)),x);
cos(x^2 + cos(x)) (2 x - sin(x))
```

MAPLE évaluant toujours les noms des arguments, l'expression à dériver peut être désignée par un nom.

```
> e:=sin(x^2+cos(x));
diff(e,x);
e := sin(x^2 + cos(x))
cos(x^2 + cos(x)) (2 x - sin(x))
```

La fonction **diff** ne s'applique pas à une *fonction* (voir l'opérateur **D** dans ce chapitre), mais dérive toute *expression* créée par une fonction (voir aussi la remarque à la fin de ce paragraphe)

```
> f:=(x,y)->exp(-x)*x*y^2;
diff(f(z/2,t+1),t);
f:=(x,y) → e^(-x) x y^2
e^(-z/2) z (t+1)
```

Afin d'améliorer la clarté des feuilles de calculs et faciliter leur lecture on aura intérêt à utiliser, comme pour la fonction **eval**, l'opérateur inerte **Diff**. Il permet d'enregistrer l'opération de dérivation mais sans faire les calculs. On pourra alors écrire sous forme d'équation avec l'opérateur **value**

```
> Diff(e,x):=%=value(%);
d
--- sin(x^2 + cos(x)) = cos(x^2 + cos(x)) (2 x - sin(x))
dx
```

La dérivation multiple s'obtient sans difficulté avec le même opérateur. Il suffit de rajouter autant de fois que nécessaire la variable de dérivation comme arguments supplémentaires.

```
> E:=x^x:
Diff(E,x,x,x):=%=value(%);
d^3
--- x^x = x^x (ln(x) + 1)^3 + 3 x^x (ln(x) + 1) - x^x
dx^3
```

On a mentionné l'opérateur de répétition **\$n** au chapitre 3, *Ensembles, Listes,...* qui permet de créer une suite

de n objets identiques. On comprend dès lors la notation utilisée dans la commande suivante

```
> x$3;
Diff(E,x$3):%value(%);
```

x, x, x

$$\frac{d^3}{dx^3} x^x = x^x (\ln(x) + 1)^3 + \frac{3x^x (\ln(x) + 1)}{x} - \frac{x^x}{x^2}$$

On pourra constater que MAPLE n'a pas "d'états d'âme" pour dériver très rapidement des expressions un peu compliquées

```
> Diff(x^arcsin(exp(sqrt(1/x))),x): % = value(%);
```

$$\frac{d}{dx} x^{\arcsin\left(e^{\sqrt{\frac{1}{x}}}\right)} = x^{\arcsin\left(e^{\sqrt{\frac{1}{x}}}\right)} \left[-\frac{e^{\sqrt{\frac{1}{x}}} \ln(x)}{2 \sqrt{\frac{1}{x}} x^2 \sqrt{1 - \left(e^{\sqrt{\frac{1}{x}}}\right)^2}} + \frac{\arcsin\left(e^{\sqrt{\frac{1}{x}}}\right)}{x} \right]$$

MAPLE sait analyser les expressions pour déterminer si elles dépendent de une ou de plusieurs variables (voir chapitre 2). Il utilise alors pour écrire l'opérateur inerte **Diff**, conformément à la notation standard, un d "droit" ou un d "rond". On notera que dans l'expression suivante, a est pour MAPLE une variable au même titre que x . Seules nos habitudes et conventions d'écriture pourraient nous laisser penser que a est un paramètre ne jouant pas un rôle de variable.

```
> e:=exp(-a*x^2):
Diff(e,x):%value(%);
```

$$\frac{\partial}{\partial x} e^{(-a x^2)} = -2 a x e^{(-a x^2)}$$

On peut dériver successivement par rapport aux variables souhaitées.

```
> Diff(e,x$2,a):%value(%);
```

$$\frac{\partial^3}{\partial a \partial x^2} e^{(-a x^2)} = -2 e^{(-a x^2)} + 10 a x^2 e^{(-a x^2)} - 4 a^2 x^4 e^{(-a x^2)}$$

L'opérateur **diff** sait dériver formellement des *expressions de fonctions non définies* (qui ont le type *function*). Ici α n'est pas définie, mais MAPLE sait, en raison de la syntaxe utilisée, qu'il s'agit d'une expression dépendant de la variable x

```
> Diff(exp(-alpha(x)*x),x):%value(%);
```

$$\frac{d}{dx} e^{(-\alpha(x)x)} = \left(-\left(\frac{d}{dx} \alpha(x) \right) x - \alpha(x) \right) e^{(-\alpha(x)x)}$$

```
> whattype(alpha(x));
```

function

Il existe également *une autre syntaxe* avec laquelle les dérivations successives sont introduites par une liste

```
> Diff(e,[x,x,a]):=%value(%);

$$\frac{\partial^3}{\partial a \partial x^2} e^{(-ax^2)} = -2e^{(-ax^2)} + 10ax^2e^{(-ax^2)} - 4a^2x^4e^{(-ax^2)}$$

```

Cette syntaxe est la seule qui autorise la dérivation d'ordre 0, c'est-à-dire l'identité. Ceci peut s'avérer utile pour construire des expressions générales dans lesquelles l'ordre de dérivation est une variable entière positive mais qui peut prendre la valeur 0 (voir un exemple d'utilisation au chapitre 10, § *Intégration et fonctions complexes; définition des résidus*).

```
> Diff(e,[]):=%value(%);
[seq(diff(e,[x$n]),n=0..2)];

$$Diff\left(e^{(-ax^2)}, []\right) = e^{(-ax^2)}$$


$$\left[ e^{(-ax^2)}, -2ax e^{(-ax^2)}, -2a e^{(-ax^2)} + 4a^2 x^2 e^{(-ax^2)} \right]$$

```

Attention: l'opérateur **diff** s'applique, rappelons le, **aux expressions et non aux fonctions** (voir plus loin l'opérateur **D**). Autrement dit, l'opération $\frac{d}{dx} f(x)$ permet de trouver l'expression de la *valeur dérivée* en x d'une fonction f et non la fonction dérivée de f . Dans l'exemple ci-dessous MAPLE cherche à dériver une expression qui ne représente que le nom de la fonction mais qu'il considère comme une expression constituée du simple nom d'une variable f , d'où le résultat

```
> f:=(x,y)->x^2*sin(y);
Diff(f,x,y):      %value(%);

$$f := (x, y) \rightarrow x^2 \sin(y)$$


$$\frac{\partial^2}{\partial y \partial x} f = 0$$

```

Exercice : Expliquez les résultats suivants

```
> y:=x->x^2;
diff(y(x),y);

$$y := x \rightarrow x^2$$


$$0$$


> diff(y(y),y);

$$2y$$


> diff(y,y);

$$1$$


> diff(eval(y),x);
Error, non-algebraic expressions cannot be differentiated
```

Exercice : Construire à l'aide de **diff** et de **unapply** la fonction dérivée f' de la fonction f suivante

```
> f:=x->sin(3*x^2);

$$f := x \rightarrow \sin(3x^2)$$

```

L'opérateur D

Pour calculer *la fonction dérivée d'une fonction* on utilisera l'opérateur de différentiation **D** qui ne s'applique qu'à une fonction pour donner une nouvelle fonction.

```
> f:=x->sin(x)*exp(-x);  
fp:=D(f);
```

$$f := x \rightarrow \sin(x) e^{(-x)}$$
$$fp := x \rightarrow \cos(x) e^{(-x)} - \sin(x) e^{(-x)}$$

Il est donc équivalent d'écrire pour calculer la valeur de la fonction dérivée pour la valeur $z + 1$ de la variable

```
> fp(z+1) = D(f)(z+1);  
cos(z+1) e^{(-z-1)} - sin(z+1) e^{(-z-1)} = cos(z+1) e^{(-z-1)} - sin(z+1) e^{(-z-1)}
```

L'opérateur **D** sait appliquer la règle de dérivation des fonctions composées, même formellement (on rappelle que $r@s$ signifie $r \circ s$)

```
> 'D(r@s)'=D(r@s); # r et s sont des fonctions non définies  
D(r @ s) = D(r) @ s D(s)
```

Dans l'exemple suivant on applique 2 fois l'opérateur de différentiation à l'aide de l'opérateur de composition **@@** (voir chapitre 7, *Fonctions*, § *Manipulation des fonctions par leurs noms*)

```
> (D@@2)(f);  
D(D(f)); # Deux écritures équivalentes  
x → -2 cos(x) e^{(-x)}  
x → -2 cos(x) e^{(-x)}
```

Attention: il ne faut pas confondre l'opérateur de composition **@@n** avec l'opérateur de répétition **\$n** qui donne une séquence formée de n fois la fonction dérivée première (d'une utilité ici, pour le moins marginale)

```
> (D$2)(f);  
D(f)$2; # Deux écritures équivalentes  
x → cos(x) e^{(-x)} - sin(x) e^{(-x)}, x → cos(x) e^{(-x)} - sin(x) e^{(-x)}  
x → cos(x) e^{(-x)} - sin(x) e^{(-x)}, x → cos(x) e^{(-x)} - sin(x) e^{(-x)}
```

L'opérateur **D** agissant sur les fonctions et non sur leurs expressions, on ne peut plus désigner par leurs noms, les variables par rapport auxquelles on dérive. Considérons la fonction h de deux variables

```
> h:=unapply(t^2*exp(-x),x,t);  
h := (x, t) → t^2 e^{(-x)}
```

Les deux exemples suivants montrent une syntaxe permettant la dérivation multiple. Pour le calcul de la fonction F on dérive une fois par rapport à la première variable et une fois par rapport à la seconde. Pour le calcul de G on dérive une fois par rapport à la première et deux fois par rapport à la seconde. Cette subtilité de notation peut apparaître un peu formelle mais s'avère intéressante dans certaines circonstances (développements de Taylor de fonctions de plusieurs variables ou créations d'opérateurs par exemple).

```
> F:=D[1,2](h);  
G:=D[1,2$2](h);
```

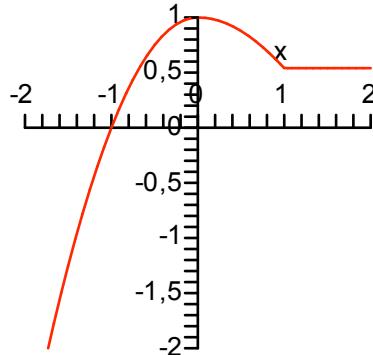
$$F := (x, t) \rightarrow -2 t e^{(-x)}$$

$$G := (x, t) \rightarrow -2 e^{(-x)}$$

Les opérateurs **diff** et **D** s'appliquent aussi sur des expressions ou des fonctions définies par morceaux

> **f:=x->piecewise(x<=0,-x^2+1,x<=1,cos(x),cos(1));**

$$f := x \rightarrow \text{piecewise}(x \leq 0, -x^2 + 1, x \leq 1, \cos(x), \cos(1))$$



> **diff(f(x),x), D(f)(x);**

$$\begin{cases} -2x & x \leq 0 \\ -\sin(x) & x < 1 \\ \text{undefined} & x = 1 \\ 0 & 1 < x \end{cases} \quad \begin{cases} -2x & x \leq 0 \\ -\sin(x) & x < 1 \\ \text{undefined} & x = 1 \\ 0 & 1 < x \end{cases}$$

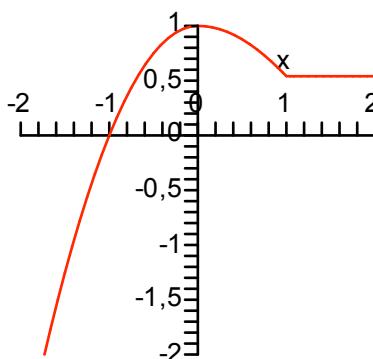
Exercice : expliquez la différence avec le résultat précédent

> **f:=x->piecewise(x<0,-x^2+1,x<=1 and x>0,cos(x),cos(1));**

D(f)(x);

$$f := x \rightarrow \text{piecewise}(x < 0, -x^2 + 1, x \leq 1 \text{ and } 0 < x, \cos(x), \cos(1))$$

$$\begin{cases} -2x & x < 0 \\ \text{undefined} & x = 0 \\ -\sin(x) & x < 1 \\ \text{undefined} & x = 1 \\ 0 & 1 < x \end{cases}$$



Dérivation des fonctions définies implicitement

Une fonction est souvent définie à l'aide d'une expression : $y = \text{expression}(x) \Rightarrow y := x \rightarrow \text{expression}(x)$. Mais, plus généralement, une fonction peut être définie par une équation $f(x, y) = g(x, y)$ qui peut toujours s'écrire $F(y, x) = 0$. Ces équations définissent implicitement soit $y(x)$ soit $x(y)$ sans qu'il soit nécessairement possible de les mettre sous la forme explicite $y = \text{expression}(x)$ ou $x = \text{expression}(y)$. Supposons que l'on décide que F définit $y(x)$. La dérivée de $F(y(x), x)$ par rapport à x nous donne

> restart;

Diff(F(y(x), x)=0, x):%:=value(%);

$$\frac{d}{dx} (F(y(x), x) = 0) = \left(D_1(F)(y(x), x) \left(\frac{d}{dx} y(x) \right) + D_2(F)(y(x), x) \right) = 0$$

où $D_1(F)$ et $D_2(F)$ désignent respectivement les dérivées de F par rapport à y et à x . De l'équation de droite on tire

$$\frac{d}{dx} y(x) = - \frac{D_2(F)(y(x), x)}{D_1(F)(y(x), x)}$$

Le propos de la fonction **implicitdiff** est d'exprimer des dérivées des fonctions ainsi définies, c'est-à-dire implicitement. Considérons un exemple qui, à l'évidence, ne nécessite pas l'utilisation de cette fonction, mais qui est simple à comprendre et à vérifier. Soit E l'équation définissant soit y fonction de x , soit x fonction de y

> E:=y+x^2=0;

$$E := y + x^2 = 0$$

Si l'on veut calculer l'expression de la dérivée de $y(x)$ par rapport à x on écrira

> implicitdiff(E, y, x);

$$-2x$$

On aura aucune difficulté à vérifier que ce résultat est juste. Si, au contraire, on veut calculer l'expression de la dérivée de x par rapport à y , on écrira

> implicitdiff(E, x, y);

$$-\frac{1}{2x}$$

Attention: x désigne ici l'expression $x(y)$ de la fonction x et non la variable x . L'équation E sous-entend deux fonctions distinctes $x(y)$ en raison du carré et que l'on peut définir par

> x[1]:=y->sqrt(-y);
x[2]:=y->-sqrt(-y);

$$x_1 := y \rightarrow \sqrt{-y}$$

$$x_2 := y \rightarrow -\sqrt{-y}$$

Les résultats de **diff** sur les expressions de ces fonctions sont bien conformes à celui de **implicitdiff** dans lequel on remplacerait respectivement x par x_1 ou x_2 (en fait $x_1(y)$ ou $x_2(y)$)

> diff(x[1](y), y);

```

diff(x[2](y),y);

$$-\frac{1}{2 \sqrt{-y}}$$


$$\frac{1}{2 \sqrt{-y}}$$


```

Notez que l'équation peut être entrée directement et pas nécessairement sous la forme *expression = 0*.

```

> implicitdiff(x^2=-y,y,x);

$$-2x$$


```

Si on utilise une simple expression elle sera supposée égale à 0. Tout nom non spécifiée comme fonction ou variable sera considéré comme une constante relativement à la variable

```

> implicitdiff(x^2+y+z,y,x);

$$-2x$$


```

Mais une fonction peut être définie de façon formelle

```

> implicitdiff(x^2+y+z(x),y,x);

$$-2x - D(z)(x)$$


```

```

> implicitdiff(x^2+y+z(x,y),y,x);

$$-\frac{2x + D_1(z)(x, y)}{1 + D_2(z)(x, y)}$$


```

Voici une autre façon d'interpréter les résultats de **implicitdiff**. Soit l'équation de définition

```

> x:='x': # en raison des définitions précédentes
F:=y^2-exp(-x^2)=0;

$$F := y^2 - e^{(-x^2)} = 0$$


```

Ici on a encore de la chance car cette équation est séparable en x et y . Différentions chaque membre par rapport à x en précisant que y est fonction de x

```

> diff(y(x)^2,x)=diff(exp(-x^2),x);

$$2y(x) \left( \frac{d}{dx}y(x) \right) = -2x e^{(-x^2)}$$


```

Cherchons l'expression de la dérivée de $y(x)$ par rapport à x en utilisant **implicitdiff**. On notera immédiatement l'identité des deux résultats

```

> Diff(y(x),x)=implicitdiff(F,y(x),x);

$$\frac{d}{dx}y(x) = -\frac{x e^{(-x^2)}}{y}$$


```

On peut encore faire les remarques suivantes. L'équation précédente est une équation différentielle du premier ordre que MAPLE peut résoudre avec sa condition initiale. En regardant l'équation de définition on note que pour $x = 0$ on doit avoir soit $y = 1$ soit $y = -1$. Considérons la condition initiale $y(0) = 1$ et résolvons l'équation différentielle (voir chapitre 15, *Equations différentielles*)

```

> dsolve({Diff(y(x),x)=-x*exp(-x^2)/y(x),y(0)=1},y(x));

```

$$y(x) = e^{\left(-\frac{1}{2}x^2\right)}$$

qui correspond à l'expression de l'une des fonctions que l'on tirerait de l'équation de définition. Pour l'autre détermination $y(0) = -1$ on a bien sûr

```
> dsolve({diff(y(x),x)=-x*exp(-x^2)/y(x),y(0)=-1},y(x));
```

$$y(x) = -e^{\left(-\frac{1}{2}x^2\right)}$$

Pour $x(y)$ on écrit

```
> Diff(x(y),y)=implicitdiff(F,x,y);
```

$$\frac{d}{dy} x(y) = - \frac{y}{x e^{(-x^2)}}$$

Comme condition initiale définie par l'équation de définition on n'a qu'une seule possibilité qui est $x(1) = 0$, mais la solution de l'équation différentielle donne aussi deux déterminations

```
> dsolve({diff(x(y),y)=-y/x(y)/exp(-x(y)^2),x(1)=0},x(y));
```

$$x(y) = \sqrt{-2 \ln(y)}, x(y) = -\sqrt{-2 \ln(y)}$$

Evidemment l'intérêt de **implicitdiff** vient de ce que les équations de définition ne sont pas toujours séparables et qu'il n'est pas toujours possible de réaliser ce type de calculs "inverses". Les résultats ne sont pas toujours aussi intuitifs !

```
> E:=sin(y*exp(x))=x*ln(x*cos(y)):E;
implicitdiff(E,y,x);
```

$$\begin{aligned} \sin(y e^x) &= x \ln(x \cos(y)) \\ - \frac{\cos(y e^x) e^x \cos(y) y - \ln(x \cos(y)) \cos(y) - \cos(y)}{\cos(y e^x) e^x \cos(y) + x \sin(y)} \end{aligned}$$

On aura également noté que de façon générale (en effet $Z(t, u) = 0$ n'est ici qu'une définition symbolique non précisée)

```
> Z(t,u)=0;
Diff(u(t),t)*Diff(t(u),u)=implicitdiff(Z(t,u)=0,u,t)*
implicitdiff(Z(t,u)=0,t,u);
Z(t,u)=0
```

$$\left(\frac{d}{dt} u(t) \right) \left(\frac{d}{du} t(u) \right) = 1$$

Les **fonctions de plusieurs variables** peuvent être définies par un système (un ensemble) d'équations. On précise ces équations dans **implicitdiff** par un ensemble comme premier argument. Il faut préciser ensuite quels sont les noms qui représentent les fonctions, ici u et z . Les noms t et y seront considérés comme constantes **ou** variables. On précise ensuite la fonction dont on cherche la dérivée (ici u) et finalement le nom de la variable de dérivation (ici y).

```
> implicitdiff({u^2-z^2-y^2=0,u^2-t+y+sin(z)=0},{u,z},u,y);
```

$$\frac{-z + \cos(z)y}{u(2z + \cos(z))}$$

On peut obtenir plusieurs dérivées simultanément en utilisant un ensemble

$$> \text{implicitdiff}(\{u^2 - z^2 - y^2 = 0, u^2 - t + y + \sin(z) = 0\}, \{u, z\}, \{u, z\}, t);$$

$$\left\{ D_2(z) = \frac{1}{2z + \cos(z)}, D_2(u) = \frac{z}{u(2z + \cos(z))} \right\}$$

Enfin **implicitdiff** peut ne pas savoir répondre (*FAIL* = Echec). Ici le système d'équations est surdéterminé

$$> \text{implicitdiff}(\{u^2 - z^2 - y^2 = 0, u^2 - t + y + \sin(z) = 0, u^2 * y = t\}, \{u, z\}, u, t);$$

$$FAIL$$

Dérivation des distributions

Voir chapitre 10, *Intégration*.

9 - Limites, Continuité

Limite

La fonction **limit** et son opérateur inerte associé **Limit** permettent de calculer la limite d'une *expression* pour une valeur donnée de la ou d'une des variables. Ici on demande que la limite soit calculée en $x = 0$

```
> restart;
```

```
Limit(x*ln(x),x=0)=limit(x*ln(x),x=0);  
limx → 0 (x ln(x)) = 0
```

On peut aussi utiliser une syntaxe similaire à celle de **eval** ou **diff** avec l'opérateur **value** qui active l'opérateur inerte

```
> Limit(exp(1-x/a),x=a):%value(%);
```

$$\lim_{x \rightarrow a} e^{\left(1 - \frac{x}{a}\right)} = 1$$

```
> Limit(-1/abs(x),x=0):%value(%);
```

$$\lim_{x \rightarrow 0} \left(-\frac{1}{|x|}\right) = -\infty$$

```
> Limit(x!/x^x/exp(-x)/sqrt(x),x=infinity):%value(%);
```

$$\lim_{x \rightarrow \infty} \left(\frac{x!}{x^x e^{(-x)} \sqrt{x}}\right) = \sqrt{2} \sqrt{\pi}$$

Avec l'expression suivante **limit** écrit prudemment qu'il ne sait pas calculer la limite : en effet celle-ci dépend du signe de a

```
> Limit(exp(-a*x),x=infinity):%value(%);
```

$$\lim_{x \rightarrow \infty} e^{(-a x)} = \lim_{x \rightarrow \infty} e^{(-a x)}$$

A l'aide de la fonction **assume** ou l'environnement **assuming** que nous détaillerons au chapitre 13, on peut imposer à **limit** de considérer a comme une variable réelle strictement positive

```
> Limit(exp(-a*x),x=infinity):%value(%)  
assuming a>0;
```

$$\lim_{x \rightarrow \infty} e^{(-a x)} = 0$$

Répétons que **limit** calcule la limite d'une *expression*. Pour calculer la limite d'une *fonction* f on devra donner une expression associée et pas simplement son nom

```
> f:=(x,a)->sqrt(a*x)*ln(x);
```

```
Limit(f(z,u),z=0):%value(%);
```

$$f := (x, a) \rightarrow \sqrt{a x} \ln(x)$$

$$\lim_{z \rightarrow 0} (\sqrt{u z} \ln(z)) = 0$$

Une limite peut être indéterminée, mais comprise dans un intervalle. MAPLE affiche alors cet intervalle avec la syntaxe qui lui est propre

```
> Limit(sin(x),x=infinity):%value(%);
Limit((1/x)*sin(x)*sin(1/x),x=0):%value(%);

$$\lim_{x \rightarrow \infty} \sin(x) = -1 .. 1$$


$$\lim_{x \rightarrow 0} \left( \frac{\sin(x) \sin\left(\frac{1}{x}\right)}{x} \right) = -1 .. 1$$

```

Une limite peut-être indéterminée si les limites à gauche et à droite du point considéré sont différentes. La réponse sera alors donnée sous la forme du mot clé ***undefined***

```
> Limit(1/x,x=0):%value(%);
g:=x->exp(-1/x):
Limit(g(x),x=0):%value(%);

$$\lim_{x \rightarrow 0} \left( \frac{1}{x} \right) = \text{undefined}$$


$$\lim_{x \rightarrow 0} e^{\left( -\frac{1}{x} \right)} = \text{undefined}$$

```

L'indétermination peut-être levée en cherchant la limite à droite ($x + \varepsilon, \varepsilon > 0$) avec un troisième argument ***right*** ou à gauche avec ***left*** ($x - \varepsilon, \varepsilon > 0$)

```
> Limit(g(x),x=0,right):%value(%);
Limit(g(x),x=0,left):%value(%);

$$\lim_{x \rightarrow 0^+} e^{\left( -\frac{1}{x} \right)} = 0$$


$$\lim_{x \rightarrow 0^-} e^{\left( -\frac{1}{x} \right)} = \infty$$

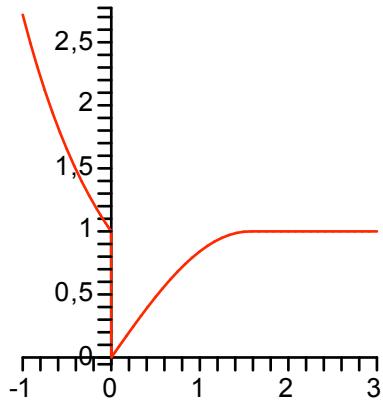
```

Les limites aux points de discontinuité d'une expression définie par morceaux sont traitées correctement

```
> p:=x->piecewise(x<0,exp(-x),x<Pi/2,sin(x),1);

$$p := x \rightarrow \text{piecewise}\left( x < 0, e^{(-x)}, x < \frac{1}{2}\pi, \sin(x), 1 \right)$$

```



On utilise ici une "double non-évaluation" (évaluation deux fois retardée: ' $p(x)$ ') pour éviter un affichage encombrant

```
> Limit( ' 'p(x)' ',x=0):%value(%);
Limit( ' 'p(x)' ',x=0,right):%value(%);
Limit( ' 'p(x)' ',x=0,left):%value(%);

$$\lim_{x \rightarrow 0} p(x) = \text{undefined}$$


$$\lim_{x \rightarrow 0^+} p(x) = 0$$


$$\lim_{x \rightarrow 0^-} p(x) = 1$$

```

En $\pi/2$ la fonction change de définition mais les limites à droite et à gauche sont identiques. **Attention**, comme le montre un exemple au paragraphe suivant, on ne doit pas se fier à une simple représentation graphique pour déterminer la continuité d'une expression...

```
> Limit( ' 'p(x)' ',x=Pi/2):%value(%);

$$\lim_{x \rightarrow \frac{\pi}{2}} p(x) = 1$$

```

On peut également calculer la limite d'expressions de plusieurs variables en donnant les valeurs sous la forme d'un ensemble

```
> Limit((x+y)^2,{x=0,y=0}):%value(%);
Limit((x + y)2, {x = 0, y = 0}) = 0
```

Attention: les limites sont calculées en faisant tendre *indépendamment* les variables vers leurs valeurs prescrites.

```
> limit(1/(x^2+y^2),{x=0,y=0});

$$\text{undefined}$$

```

À comparer avec

```
> limit(1/r^2,r=0);

$$\infty$$

```

Continuité

MAPLE dispose de deux fonctions relatives à la continuité des expressions.

1) La fonction *iscont*

Cette fonction (comprendre "is continue ?") permet de tester la continuité d'une expression **sur un intervalle réel donné** considéré *ouvert par défaut*. Il y a trois réponses possibles : **true** (la fonction est continue), **false** (elle ne l'est pas), **FAIL** (*iscont* ne peut pas répondre).

```
> restart:  
iscont(1+x/(x-1),x=1..infinity); # Attention : 1 est exclu  
iscont(1+x/(x-1),x=-infinity..infinity);  
                                true  
                                false  
  
> f:=(x,a)->exp(1/abs(x-a));  
iscont(f(x,a),x=-1..2); # La réponse dépend de la valeur de a  
a:=0;  
iscont(f(x,a),x=-1..2);  
  

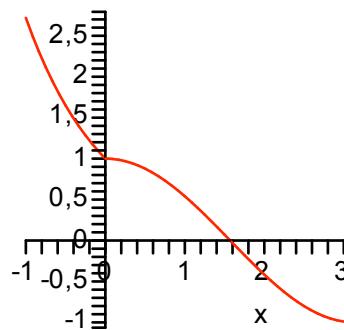
$$f := (x, a) \rightarrow e^{\left(\frac{1}{|x - a|}\right)}$$
  
                                FAIL  
                                a := 0  
                                false
```

On peut, à l'aide d'un troisième argument, ***closed***, exprimer que l'intervalle doit être considéré fermé

```
> iscont(1+x/(x-1),x=1..infinity,closed);# Attention +1 est maintenant inclus  
iscont(f(x,a),x=0..2); # On rappelle que a=0  
iscont(f(x,a),x=0..2,closed);# 0 est inclus  
a:='a':  
                                false  
                                true  
                                false
```

Cette fonction peut traiter correctement les fonctions définies par morceaux

```
> p:=x->piecewise(x>=0,cos(x),exp(-x));  
iscont(p(x),x=-infinity..infinity);  
                                p := x → piecewise(0 ≤ x, cos(x), e(-x))  
                                true
```

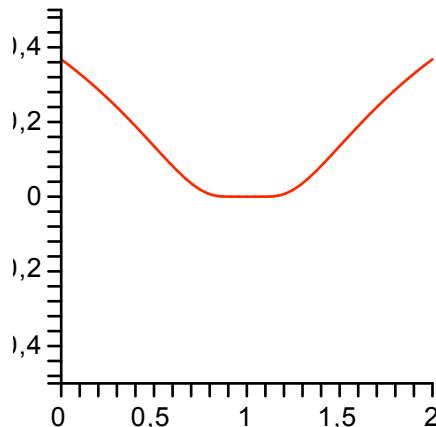


Attention: on ne doit pas se baser sur une simple représentation graphique pour déterminer la continuité d'une expression... prudence !

```
> e:=exp(-1/abs(x-1));
iscont(e,x=0..2);
```

$$e := e \left(-\frac{1}{|x - 1|} \right)$$

false



En effet, malgré l'apparence du graphique, l'expression n'est pas définie pour $x = 1$

```
> eval(e,x=1);
Error, numeric exception: division by zero
```

même si

```
> Limit(e,x=1):%:=value(%);
```

$$\lim_{x \rightarrow 1} e \left(-\frac{1}{|x - 1|} \right) = 0$$

Attention: les résultats donnés par **iscont** (et **discont**) ne sont pas toujours fiables. Ils dépendent en particulier de la capacité de MAPLE à résoudre de façon exacte les équations (voir chapitre 14, *Equations ordinaires*). L'équation $\cos(\sqrt{x}) - x^2 = 0$ possède en effet une racine dans $[0, +1]$ (voir plus bas), mais...

```
> iscont(1/(cos(sqrt(x))-x^2),x=0..infinity);
```

true

Attention: La fonction **iscont** ne prend pas en compte la *table Cache* ou *remember* d'une fonction définie par l'utilisateur. Dans l'exemple suivant, selon la règle des évaluations préalables des arguments, ce qui est transmis à la fonction **iscont** c'est l'évaluation de $f(x)$ c'est-à-dire l'expression associée $\exp(1/|x-1|)$. La fonction **iscont** ne peut donc pas "savoir" que cette expression vient de la définition de f et donc prendre en compte la table *Cache* ou *remember* de f (en fait seulement celle de **exp**).

```
> f:=x->exp(-1/abs(x-1));
f(1):=0; # Chargement de la table "remember" de f
iscont(f(x),x=0..2);
```

$$f := x \rightarrow e^{\left(-\frac{1}{|x - 1|}\right)}$$

$$f(1) := 0$$

$$false$$

Pourtant, après ce chargement de la table *remember*, on a bien

```
> evalb(f(1)=limit(f(x),x=1)); # à droite et à gauche
true
```

Exercice : l'écriture suivante donne une réponse qui semble correcte, mais qui est en réalité totalement illusoire. La réponse est néanmoins juste relativement à la question que comprend **iscont** ! Pourquoi ?

```
> iscont(f,x=0..infinity);
true
```

2) La fonction *discont*

Cette fonction essaie de déterminer l'ensemble des points de discontinuité sur tout l'*ensemble* des réels pour une des variables de l'expression

```
> e:=y+1/((x-z)*(x+2));
discont(e,x); # par rapport à x
discont(e,z); # par rapport à z
```

$$e := y + \frac{1}{(x - z)(x + 2)}$$

$$\{-2, z\}$$

$$\{x\}$$

Cette fonction peut renvoyer un ensemble de valeurs définies par une expression à l'aide de noms spéciaux symboliques. On peut rencontrer plusieurs types de noms : *_Zn* désigne ici les entiers relatifs, *_NNn* désigne les entiers Non Négatifs (0 fait partie de cet ensemble ; *n* n'a qu'un caractère distinctif si plusieurs ensembles identiques doivent apparaître)

```
> discont(1/cos(x),x);
discont(GAMMA(x+1)+1/(x+2),x);

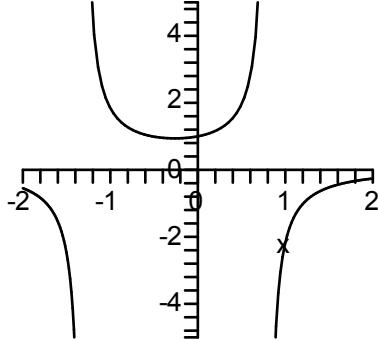
```

$$\left\{ \frac{1}{2} \pi + \pi _ZI\sim \right\}$$

$$\{-2, -1 - _NNI\sim\}$$

Attention: comme **iscont**, la fonction **discont** peut donner des résultats incomplets ou franchement inexacts. Pour l'exemple suivant il y a deux discontinuités sur $[-\infty, \infty]$. Pour $x > 0$, c'est la solution de $\cos(\sqrt{x}) - x^2 = 0$ ($x = 0.79308\dots$) et pour $x < 0$, la solution de $\cosh(\sqrt{-x}) - x^2 = 0$ ($x = -1.3167\dots$)

```
> discont(1/(cos(sqrt(x))-x^2),x);
{0}
```



La réponse donnée par **discont** est fausse, non seulement parce que les vraies discontinuités ne sont pas trouvées, mais aussi parce que la valeur de l'expression en 0 est définie et les limites à droite et à gauche sont égale à cette valeur.

```
> Eval(1/(cos(sqrt(x))-x^2),x=0):%value(%);
Limit(1/(cos(sqrt(x))-x^2),x=0):%value(%);


$$\left( \frac{1}{\cos(\sqrt{x}) - x^2} \right) \Big|_{x=0} = 1$$


$$\lim_{x \rightarrow 0} \left( \frac{1}{\cos(\sqrt{x}) - x^2} \right) = 1$$

```

Détermination numérique des discontinuités

La fonction **fdiscont** tente de déterminer numériquement dans un intervalle donné une liste d'encadrements des discontinuités d'une expression et de sa dérivée première **sur les réels**. Evidemment, il ne peut y avoir ici qu'une indéterminée (ici x) dans l'expression. On cherche les singularités de E dans $[1,5]$ et **fdiscont** va déterminer, sous forme de liste d'intervalles, des encadrements autour de 2 et π

```
> E:=1/abs((x-2))/sin(x);
fdiscont(E,x=1..5);


$$E := \frac{1}{|x - 2| \sin(x)}$$


[1.99971127949675464 .. 2.00070551408184016, 3.14118612558853050 .. 3.14220145597209122]

> evalf[8](%);

[1.9997113 .. 2.0007055, 3.1411861 .. 3.1422015]
```

Ici, c'est seulement la **dérivée** de F qui présente une discontinuité en $x = 2$

```

> F:=abs(x-2)+sin(x);
  evalf[8](fdiscont(F,x=1..5));
      F := |x - 2| + sin(x)
      [1.9995653 .. 2.0005275]

```

Mais la priorité est donnée aux discontinuités de l'expression elle-même

```

> F:=abs(x-2)+1/sin(x);
  evalf[8](fdiscont(F,x=1..5));
      F := |x - 2| + 1/sin(x)
      [3.1411264 .. 3.1422562]

```

Attention: la fonction **fdiscont** présente des restrictions d'utilisation et ses réponses doivent être considérées avec *beaucoup de prudence*.

1) La fonction **fdiscont** ne détecte pas tous les types de discontinuités (voir paragraphe précédent)

```

> G:=x->exp(-1/abs(x-1));
  G(1);
  fdiscont(G(x),x=0..2);
      G := x → e(- 1/|x - 1|)

```

Error, (in G) numeric exception: division by zero

La nouvelle expression $G(x)$ ne présente plus de singularité dans $[0,2]$, mais la singularité de la dérivée pour $x = 1$ n'est pas détectée

```

> G:=x->abs(x-1)*exp(-1/abs(x+3));
  fdiscont(G(x),x=0..2);
      G := x → |x - 1| e(- 1/|x + 3|)

```

2) La fonction **fdiscont** ne fonctionne que *sur les réels*. On cherche les singularités de H dans $[1/2,2]$ avec un précision de l'ordre de $10^{(-5)}$ (se reporter à sa représentation graphique au paragraphe précédent)

```

> H:=1/(cos(sqrt(x))-x^2);
  dis_num_1:=evalf[8](fdiscont(H,x=1/2..2,1.0e-5));
      H := 1/(cos(sqrt(x)) - x2)
      dis_num_1 := [0.79307994 .. 0.79308974]

```

Malgré les indications de la représentation graphique **fdiscont** ne trouve rien dans l'intervalle $[-2,-1]$. Bien que l'expression soit à valeurs réelles, ceci vient de ce que \sqrt{x} est imaginaire et **fdiscont** ne fonctionne plus

```

> dis_num_2:=fdiscont(H,x=-2..-1,1.0e-5);
      dis_num_2 := []

```

Si on donne une forme équivalente appropriée tout rentre dans l'ordre ($\cos(\sqrt{x})$ pour $x < 0$ devient $\cosh(\sqrt{-x})$).

```
> H_R:=simplify(H) assuming x<0; # voir Chapitre 13
  dis_num_2:=evalf[8](fdiscont(H_R,x=-2..-1,1.0e-5));

$$H_R := -\frac{1}{-\cosh(\sqrt{-x}) + x^2}$$

dis_num_2 := [-1.3167719 .. -1.3167610]
```

Il existe d'autres possibilités que l'on pourra trouver dans l'aide en ligne.

10 - Intégration

L'opérateur int

MAPLE permet de calculer des intégrales indéfinies d'expressions à l'aide de l'opérateur **int**, le premier argument donnant l'*expression* à intégrer, le deuxième la variable d'intégration. Comme pour **diff** on peut utiliser l'opérateur inerte **Int** pour mémoriser l'intégrale sans effectuer le calcul et **value** pour l'exécuter. On remarque que ces calculs ne font pas apparaître de constantes additives.

```
> restart:  
e:=1-a/sqrt(1-x^2):  
Int(e,x) = int(e,x);  
Int(e,a): % = value(%);
```

$$\int \frac{1 - \frac{a}{\sqrt{1 - x^2}}}{dx} dx = x - a \arcsin(x)$$
$$\int \frac{1 - \frac{a}{\sqrt{1 - x^2}}}{da} da = a - \frac{a^2}{2 \sqrt{1 - x^2}}$$

Cette écriture permet également le calcul d'intégrales multiples. On vérifie ici que l'échange de l'ordre des intégrations suivant a et x donne le même résultat (théorème de *Fubini*)

```
> Int(Int(e,x),a):%value(%);  
Int(Int(e,a),x):%value(%);
```

$$\int \int \frac{1 - \frac{a}{\sqrt{1 - x^2}}}{dx da} dx da = a x - \frac{1}{2} a^2 \arcsin(x)$$
$$\int \int \frac{1 - \frac{a}{\sqrt{1 - x^2}}}{da dx} da dx = a x - \frac{1}{2} a^2 \arcsin(x)$$

On peut bien sûr intégrer des fonctions, ou *plus exactement l'expression d'une fonction*

```
> f:=x->exp(-x)*sin(x)^2;  
Int(f(x),x): % = value(%);
```

$$f := x \rightarrow e^{(-x)} \sin(x)^2$$

$$\int e^{(-x)} \sin(x)^2 dx = \frac{(-\sin(x) - 2 \cos(x)) \sin(x)}{5 e^x} - \frac{2}{5 e^x}$$

avec les mêmes remarques que pour la dérivation à propos du nom de la fonction

```
> Int(f, x):%:=value(%);
```

$$\int f dx = fx$$

Le calcul des *intégrales définies* s'effectue de la même façon en précisant par le signe = après la variable d'intégration, l'intervalle d'intégration

```
> Int(f(x), x=0..2*Pi): %:=value(%);
```

$$\int_0^{2\pi} e^{(-x)} \sin(x)^2 dx = \frac{2}{5} (-1 + e^{(2\pi)}) e^{(-2\pi)}$$

L'intégrale suivante est dite *impropre* car l'intégrant $F(x)$ n'est pas défini pour $x = 0$.

```
> F:=x->1/sqrt(x*(1-x));
```

```
Int(F(x), x=0..1/2) = int(F(x), x=0..1/2);
```

$$\int_0^{\frac{1}{2}} \frac{1}{\sqrt{x(1-x)}} dx = \frac{1}{2}\pi$$

Conformément au calcul de ce type d'intégrale, MAPLE calcule sa limite quand la borne inférieure tend vers $0+$

```
> Int(F(x), x=z..1/2) = int(F(x), x=z..1/2);
```

```
Limit(-arcsin(2*z-1), z=0):%:=value(%);
```

$$\int_z^{\frac{1}{2}} \frac{1}{\sqrt{x(1-x)}} dx = -\arcsin(-1 + 2z)$$

$$\lim_{z \rightarrow 0} (-\arcsin(-1 + 2z)) = \frac{1}{2}\pi$$

Le concept d'*infini* doit être ici compris dans le même sens, c'est-à-dire comme la limite de l'intégrale quand la borne supérieure tend vers l'infini

```
> Int(f(x), x=0..infinity) = int(f(x), x=0..infinity);
```

$$\int_0^{\infty} e^{(-x)} \sin(x)^2 dx = \frac{2}{5}$$

Certaines intégrales, même d'apparence simple (et pour des fonctions intégrables), s'avèrent incalculables de façon symbolique. Dans cette situation MAPLE se contente de ré-écrire l'intégrale demandée.

$$> \text{Int}(\exp(-x) * \sin(\sqrt{x}), x=0..1) : \% = \text{value}(\%);$$

$$\int_0^1 e^{(-x)} \sin(\sqrt{x}) dx = \int_0^1 e^{(-x)} \sin(\sqrt{x}) dx$$

Calcul numérique d'une intégrale

Il est cependant possible d'appliquer l'opérateur **evalf** sur **int** ou **Int** pour évaluer numériquement une intégrale en fixant éventuellement le nombre de chiffres significatifs (15 dans l'exemple suivant)... mais attention au temps de calcul !

Important : on appliquera la fonction **evalf** sur l'opérateur inerte **Int** plutôt que **int**: **evalf(Int(...))**. En procédant ainsi MAPLE déclenche immédiatement la procédure de calcul numérique. Sinon, suivant la règle d'évaluation préalable des arguments d'une fonction, il cherche d'abord l'expression exacte de l'intégrale et, s'il peut la trouver, lui applique **evalf**. Ceci peut allonger sérieusement les temps de calcul.

$$> \text{Int}(\exp(-x) * \sin(\sqrt{x}), x=0..1) : \% = \text{evalf}[15](\%);$$

$$\int_0^1 e^{(-x)} \sin(\sqrt{x}) dx = 0.346614487078394$$

MAPLE sait quelquefois gérer les singularités qui peuvent apparaître dans l'intégrant. On sait que l'intégrale suivante n'est pas convergente et MAPLE le signale même si on lui demande une intégration numérique

$$> \text{Int}(1/(x-1)^2, x=0..2) : \% = \text{evalf}(\%);$$

$$\int_0^2 \frac{1}{(x-1)^2} dx = \text{Float}(\infty)$$

Bien que l'intégrale suivante soit impropre (mais convergente), l'évaluation numérique donne un résultat correct

$$> \text{Int}(\exp(-1/x^2)/x^2, x=-1..1) : \% = \text{evalf}(\%);$$

$$\int_{-1}^1 \frac{e^{-\frac{1}{x^2}}}{x^2} dx = 0.2788055853$$

En fait cette intégrale est calculable grâce à la fonction d'erreur *erf* (elle-même définie par un intégrale; voir plus loin). Ceci nous permet de vérifier le calcul numérique précédent

$$> \text{Int}(\exp(-1/x^2)/x^2, x=-1..1) : \% = \text{value}(\%);$$

$$\text{sqrt}(\text{Pi}) * (1 - \text{erf}(1)) : \% = \text{evalf}(\%);$$

$$\int_{-1}^1 \frac{e^{-\frac{1}{x^2}}}{x^2} dx = -\text{erf}(1) \sqrt{\pi} + \sqrt{\pi}$$

$$\sqrt{\pi} (1 - \text{erf}(1)) = 0.2788055854$$

Par défaut MAPLE choisit un algorithme en fonction du nombre de chiffres significatifs demandé, mais on pourra fixer la méthode parmi celles qui sont disponibles (voir l'aide en ligne). Ici on choisit l'algorithme adaptatif de **Newton-Cotes** (notez le `_` devant **NRule**).

```
> Int(exp(-x)*sin(sqrt(x)), x=0..1)=
    evalf(Int(exp(-x)*sin(sqrt(x)), x=0..1, 'method'=_NRule));
    
$$\int_0^1 e^{(-x)} \sin(\sqrt{x}) dx = 0.3466144871$$

```

Une intégration numérique approchée permet de créer une approximation numérique d'une fonction définie par une intégrale non calculable symboliquement (le nombre de chiffres significatifs est ici fixé à 4 par **evalf**). Si cette méthode permet une mise en œuvre rapide, les calculs de la fonction peuvent être longs et on préférera, pour des calculs intensifs, appliquer la théorie de l'approximation des fonctions (polynômes d'interpolation, splines, approximants de Padé, etc.; voir aussi la bibliothèque **numapprox** et le chapitre 7, § *Approximation des fonctions*)

```
> Fn:=t->evalf[4](Int(exp(-x)*sin(sqrt(x)), x=0..t));
'Fn(t)' = Fn(t);
```

$$Fn := t \rightarrow evalf_4 \left(\int_0^t e^{(-x)} \sin(\sqrt{x}) dx \right)$$

$$Fn(t) = \int_0^t e^{(-1. x)} \sin(\sqrt{x}) dx$$

```
> 'Fn(0)' = Fn(0);
'Fn(1)' = Fn(1);
'Fn(Pi)' = Fn(Pi);
```

$$Fn(0) = 0.$$

$$Fn(1) = 0.3466$$

$$Fn(\pi) = 0.6522$$

Divers autres aspects de l'intégration

1) **Manipulations** : on peut appliquer des opérateurs de manipulations sur des intégrales (se reporter au chapitre 13 pour la définition et les autres utilisations des opérateurs **combine** et **simplify**)

```
> restart;
a:=Int(exp(-x),x=0..alpha):b:=Int(exp(-x),x=alpha..infinity):
a+b=combine(a+b);

$$\int_0^{\alpha} e^{-x} dx + \int_{\alpha}^{\infty} e^{-x} dx = \int_0^{\infty} e^{-x} dx$$

```

ou encore (*attention*, cette simplification n'est pas toujours valide et n'est, comme son nom l'indique, que symbolique, voir chapitre 13)

```
> Int(sqrt(x-1)/sqrt(x^2-1),x):%:=simplify(% ,symbolic);

$$\int \frac{\sqrt{x-1}}{\sqrt{x^2-1}} dx = \int \frac{1}{\sqrt{x+1}} dx$$

```

2) *Fonctions spéciales* : MAPLE connaît un grand nombre de fonctions dites spéciales qui interviennent dans les réponses aux calculs de certaines intégrales

```
> Int(sqrt(x)*sin(x),x=0..1): % = value(%);

$$\int_0^1 \sqrt{x} \sin(x) dx = -\cos(1) + \frac{1}{2} \sqrt{2} \sqrt{\pi} \text{FresnelC}\left(\frac{\sqrt{2}}{\sqrt{\pi}}\right)$$

```

Ces fonctions spéciales sont en fait souvent elles mêmes définies par des intégrales. Les fonctions **erf** et **FresnelC** sont définies par

```
> FunctionAdvisor(definition,erf);

$$\text{erf}(z) = \frac{2}{\sqrt{\pi}} \int_0^z \frac{1}{e^{(\_kl)^2}} d\_kl, \text{with no restrictions on (z)}$$

```

Remarque: la variable d'intégration est ici "*_kl*". Elle est générée par MAPLE qui fait débuter les noms de ce type de variables "internes" par un caractère de soulignement. On évitera donc de créer des noms avec cette syntaxe, même si ce n'est pas interdit. La mention "*with no restriction on (z)*" signifie que la variable peut parcourir tout le plan complexe sans restriction

```
> FunctionAdvisor(definition,FresnelC);

$$\text{FresnelC}(z) = \int_0^z \cos\left(\frac{1}{2} \pi \_kl^2\right) d\_kl, \text{with no restrictions on (z)}$$

```

3) *Intégrales de fonctions définies par morceaux* : MAPLE sait traiter des fonctions (distributions) comme l'échelon de Heaviside

```
> A:=Int(Heaviside(-x+1)*R(x),x=-2..2): A = value(A);
```

$$\text{subs}(R(x)=\exp(-x), A) := \text{value}(\%);$$

$$\int_{-2}^2 \text{Heaviside}(-x+1) R(x) dx = \int_{-2}^1 R(x) dx$$

$$\int_{-2}^2 \text{Heaviside}(-x+1) e^{(-x)} dx = e^2 - e^{-1}$$

ainsi que les expressions définies par morceaux. **Notez** dans cet exemple la double non-évaluation de l'intégrant dans l'opérateur **Int** qui évite un affichage encombrant de l'expression de $f(x)$

$$> f := x \rightarrow \text{piecewise}(x \leq 0, \cos(x), e^{(-x^2)});$$

$$\text{Int}('f(x)', x = -\pi/4 .. \text{infinity}): \% = \text{value}(\%)$$

$$f := x \rightarrow \text{piecewise}\left(x \leq 0, \cos(x), e^{(-x^2)}\right)$$

$$\int_{-\frac{1}{4}\pi}^{\infty} f(x) dx = \frac{1}{2}\sqrt{2} + \frac{1}{2}\sqrt{\pi}$$

4) **Intégrales de fonctions connues seulement par points** : l'intégration de fonctions définies par morceaux permet en particulier de calculer des intégrales de fonctions définies par points à l'aide, par exemple, d'une *fonction spline* (voir chapitre 7, *Fonctions*)

$$> X := [0, 1, 1.5, 2, 3, 4.2];$$

$$Y := [0, 1, 1.2, 1, 0.5, 0.1];$$

$$\text{int}(\text{CurveFitting}[Spline](X, Y, x), x = 0.1 .. 3.2);$$

$$X := [0, 1, 1.5, 2, 3, 4.2]$$

$$Y := [0, 1, 1.2, 1, 0.5, 0.1]$$

$$2.485028765$$

On peut aussi calculer cette intégrale en cherchant le *polynôme d'interpolation*. On ne peut évidemment pas s'attendre à ce que le résultat soit exactement le même

$$> \text{int}(\text{CurveFitting}[PolynomialInterpolation](X, Y, x), x = 0.1 .. 3.2);$$

$$2.367072179$$

Attention: aucune de ces deux méthodes n'est satisfaisante pour un échantillon contenant beaucoup de points. La première consommera du temps de calcul et de la mémoire. La seconde risque de donner des résultats fantaisistes... Pour un échantillon de points permettant une représentation raisonnable de la fonction par des segments de droites, on trouvera un exemple de procédure de calcul au chapitre 19, § *Modules* (méthode des trapèzes pour abscisses non équidistantes).

5) **Intégration et contraintes** : MAPLE est prudent : le paramètre p étant indéterminé, il ne présume pas qu'il est réel et positif et donne une réponse en conséquence. La convergence de l'intégrale dépend du signe de p

$$> a := \text{Int}(x * \exp(-p * x), x = 0 .. \text{infinity});$$

$$a = \text{value}(a);$$

$$\int_0^\infty x e^{(-p)x} dx = \lim_{x \rightarrow \infty} \left(-\frac{e^{(-p)x} + xp e^{(-p)x} - 1}{p^2} \right)$$

La fonction d'environnement **assuming** (voir chapitre 13) indique ici à MAPLE qu'il doit considérer la variable p comme réelle strictement positive.

```
> a := value(a) assuming p>0;
```

$$\int_0^\infty x e^{(-p)x} dx = \frac{1}{p^2}$$

6) **Traitement des singularités** : La convergence de l'intégrale suivante est fonction du signe de ζ . Aussi **int** ne donne aucune réponse

```
> Int(1/(x-xi)^2, x=-xi..infinity) := value(%);
```

$$\int_{-\zeta}^\infty \frac{1}{(x - \zeta)^2} dx = \int_{-\zeta}^\infty \frac{1}{(x - \zeta)^2} dx$$

On peut demander à **int** de calculer néanmoins l'intégrale en faisant abstraction de la discontinuité

```
> Int(1/(x-xi)^2, x=-xi..infinity, continuous) := value(%);
```

$$\int_{-\zeta}^\infty \frac{1}{(x - \zeta)^2} dx = -\frac{1}{2\zeta}$$

ou de donner toutes les possibilités

```
> Int(1/(x-xi)^2, x=-xi..infinity, AllSolutions) := value(%);
```

$$\int_{-\zeta}^\infty \frac{1}{(x - \zeta)^2} dx = \begin{cases} -\frac{1}{2\zeta} & \zeta < 0 \\ \infty & \zeta = 0 \\ \infty & 0 < \zeta \end{cases}$$

7) **Valeur principale au sens de Cauchy** : L'intégrale impropre suivante n'est pas convergente

```
> Int(1/x^3, x=-1..2):
% := value(%);
```

$$\int_{-1}^2 \frac{1}{x^3} dx = \text{undefined}$$

On peut alors demander de calculer la *valeur principale au sens de Cauchy*

```
> Int(1/x^3, x=-1..2, CauchyPrincipalValue):
% := value(%);
```

$$\int_{-1}^2 \frac{1}{x^3} dx = \frac{3}{8}$$

ou encore

```
> Int(exp(-x)/(x-1), x=0..2):
%value(%);
```

$$\int_0^2 \frac{e^{(-x)}}{x-1} dx = \text{undefined}$$

```
> Int(exp(-x)/(x-1), x=0..2, CauchyPrincipalValue):
%value(%);
```

$$\int_0^2 \frac{e^{(-x)}}{x-1} dx = -\text{Ei}(1) e^{(-1)} + \text{Ei}(-1) e^{(-1)}$$

$\text{Ei}(x)$ ou $\text{Ei}(n, x)$ sont des fonctions spéciales dites "exponentielles intégrales" définies aussi par des intégrales

```
> FunctionAdvisor(definition, Ei);
```

$$\left[\text{Ei}(z) = \int_{-\infty}^z \frac{e^{-kl}}{kl} d_{kl}, \text{And}(z::\text{real}) \right], \left[\text{Ei}(a, z) = \int_1^{\infty} \frac{1}{e^{(kl)z} k l^a} d_{kl}, \text{And}(0 < \Re(z)) \right]$$

Sous certaines conditions, ces fonctions sont reliées aux fonctions eulériennes incomplètes

```
> Int(exp(-x*t)/t^n, t=1..infinity):
%value(%)) assuming n>1, x>0;
```

$$\int_1^{\infty} \frac{e^{(-x)t}}{t^n} dt = x^{(n-1)} \Gamma(-n+1, x)$$

8) **Intégrale en un point** : l'opérateur **intat** (integral at) et son opérateur inerte associé **Intat** permettent d'exprimer la valeur en un point d'une fonction définie par une intégrale. Cette écriture est notamment utilisée pour la résolution des équations différentielles. Le premier résultat exprime la primitive de F (sans constante additive) pour $x = z$. La deuxième expression donne cette intégrale quand $F(x)$ vaut $x e^{-x}$.

```
> Intat(F(x), x=z);
Eval(% , F(x)=x*exp(-x)): %value(%);
```

$$\int^z F(x) dx$$

$$\int_{F(x)=x e^{(-x)}}^z F(x) dx = -(z+1) e^{(-z)}$$

On peut naturellement utiliser l'opérateur **intat** directement

$$> \text{Intat}(x * \cos(x^2), x=t^2) : \% = \text{value}(\%);$$

$$\int_{t^2}^2 x \cos(x^2) dx = \frac{1}{2} \sin(t^4)$$

L'opérateur **diff** sait traiter correctement ce type d'intégrales

$$> f := 'f'; g := 'g';$$

$$\text{Intat}((f@g)(x, t), x=r(t));$$

$$\text{Diff}(\%, t); \% = \text{value}(\%);$$

$$\frac{\partial}{\partial t} \int_{r(t)}^{r(t)} f(g(x, t)) dx = \int_{r(t)}^{r(t)} D(f)(g(x, t)) \left(\frac{\partial}{\partial t} g(x, t) \right) dx + f(g(r(t), t)) \left(\frac{d}{dt} r(t) \right)$$

Par exemple (on demande ici à Maple de prendre en compte $k > 0$ et de combiner si possible les expressions trigonométriques)

$$> \text{Diff}(\text{Intat}(t * x + \exp(-\sqrt{k^2 * x^2}), x=\cos(t)), t);$$

$$\% = \text{combine}(\text{value}(\%), \text{trig}) \text{ assuming } k > 0;$$

$$\frac{\partial}{\partial t} \int_{\cos(t)}^{\cos(t)} x t + e^{-\sqrt{k^2 x^2}} dx = \frac{1}{4} \cos(2t) + \frac{1}{4} - \frac{1}{2} t \sin(2t) - \sin(t) e^{-\frac{1}{2} k \sqrt{2 \cos(2t) + 2}}$$

9) **Intégration dans le plan complexe** : les intégrants comme les bornes peuvent être écrits sous une forme explicitement complexe (les fonctions numériques comme **cos**, **exp**, **GAMMA** etc. sont implicitement considérées comme complexes). Ici, la variable x est supposée implicitement complexe ainsi que a . Par contre b est supposé (imposé) réel.

$$> \text{restart};$$

$$f := z \rightarrow \exp(z) - I * z^2;$$

$$\text{Int}(f(x), x=a+I*\pi..I*b);$$

$$\% = \text{simplify}(\text{value}(\%)) \text{ assuming } b::\text{real};$$

$$f := z \rightarrow e^z - I z^2$$

$$\int_{a+I\pi}^{Ib} e^x - I x^2 dx = e^a - I a \pi^2 + e^{(Ib)} - \frac{1}{3} b^3 + \frac{1}{3} I a^3 - a^2 \pi + \frac{1}{3} \pi^3$$

On peut vérifier que l'intégrant précédent est analytique en vérifiant les conditions de **Cauchy-Riemann**. On pose $z = u + i v$ avec u et v réelles

$$> z := u + I * v;$$

```

Diff(Re(f(z)),v)+Diff(Im(f(z)),u):
%=value(%) assuming u::real,v::real;
Diff(Re(f(z)),u)-Diff(Im(f(z)),v):
%=value(%) assuming u::real,v::real;

$$\left( \frac{\partial}{\partial v} (\mathrm{e}^u \cos(v) + 2uv) \right) + \left( \frac{\partial}{\partial u} (\mathrm{e}^u \sin(v) - u^2 + v^2) \right) = 0$$


$$\left( \frac{\partial}{\partial u} (\mathrm{e}^u \cos(v) + 2uv) \right) - \left( \frac{\partial}{\partial v} (\mathrm{e}^u \sin(v) - u^2 + v^2) \right) = 0$$


```

Par contre, la fonction g suivante ne vérifie pas les conditions de **Cauchy-Riemann** et n'est donc pas analytique

```

> g:=z->exp(z)-I*z*conjugate(z);
Diff(Re(g(z)),v)+Diff(Im(g(z)),u):
%=value(%) assuming u::real,v::real;
Diff(Re(g(z)),u)-Diff(Im(g(z)),v):
%=value(%) assuming u::real,v::real;

```

$$g := z \rightarrow \mathrm{e}^z - I z \bar{z}$$

$$\left(\frac{\partial}{\partial v} (\mathrm{e}^u \cos(v)) \right) + \left(\frac{\partial}{\partial u} (\mathrm{e}^u \sin(v) - u^2 - v^2) \right) = -2u$$

$$\left(\frac{\partial}{\partial u} (\mathrm{e}^u \cos(v)) \right) - \left(\frac{\partial}{\partial v} (\mathrm{e}^u \sin(v) - u^2 - v^2) \right) = 2v$$

On peut donc interpréter l'intégrale ci-dessus comme la différence des primitives de f prises aux bornes de l'intégrale indépendamment du chemin suivi

```

> Intat(f(x),x=I*b)-Intat(f(x),x=a+I*Pi):
%=simplify(value(%) assuming b::real);

$$\int_a^{ib} \mathrm{e}^x - Ix^2 dx - \int_{a+I\pi}^{ib} \mathrm{e}^x - Ix^2 dx = \mathrm{e}^a - Ia\pi^2 + \mathrm{e}^{ib} - \frac{1}{3}b^3 + \frac{1}{3}Ia^3 - a^2\pi + \frac{1}{3}\pi^3$$


```

On va maintenant illustrer le **théorème des résidus** à travers un exemple. Considérons la fonction méromorphe f de la variable complexe z

```

> restart:
f:=z->(z+1)/((3*z+I)*(z-1));

$$f := z \rightarrow \frac{z+1}{(3z+I)(z-1)}$$


```

Cette fonction a des pôles d'ordre 1 dont on peut former la liste (voir chapitre 14, *Equations ordinaires*)

```

> p:=[solve(denom(f(z)))];

$$p := \left[ 1, -\frac{1}{3}I \right]$$


```

On va intégrer la fonction en suivant un contour défini par un cercle de centre O et de rayon R parcouru dans

le sens trigonométrique.

$$> \mathbf{C:=(O,R,theta) \rightarrow O + R*exp(I*theta);}$$

$$C := (O, R, \theta) \rightarrow O + R e^{(I \theta)}$$

Considérons le cercle C de centre défini par l'affixe $1 + i$ et de rayon $\sqrt{5}$ et intégrons

Exercice: on est tenté d'écrire O plutôt que o dans ce calcul. Essayez et expliquez... (voir chapitre 11)

$$> o:=1+I;$$

$$R:=\sqrt{5};$$

$$\text{Int}(f(z), z=\text{`C'}..)=$$

$$\text{Int}((f@C)(o,R,\theta)*\text{diff}(C(o,R,\theta), \theta), \theta=0..2*\pi);$$

$$o := 1 + I$$

$$R := \sqrt{5}$$

$$\int_C \frac{z+1}{(3z+I)(z-1)} dz = \int_0^{2\pi} \frac{\frac{I((2+I)+\sqrt{5} e^{(I\theta)})}{((3+4I)+3\sqrt{5} e^{(I\theta)})} \sqrt{5} e^{(I\theta)}}{(I+\sqrt{5} e^{(I\theta)})} d\theta$$

Les fonctions **lhs** et **rhs** récupèrent respectivement les membres de gauche et de droite d'une équation

$$> \text{lhs}(\%)=\text{value(rhs(\%));}$$

$$\int_C \frac{z+1}{(3z+I)(z-1)} dz = \frac{2}{3} I \pi$$

Le **théorème des résidus** nous dit que ce résultat peut être calculé comme $2i\pi$ fois la somme des résidus associés aux pôles contenus dans le cercle. Calculons d'abord la liste des résidus associés à chaque pôle à l'aide de la fonction **residue**

$$> r:=[\text{seq}(\text{residue}(f(z), z=q), q \in p)];$$

$$r := \left[\frac{3}{5} - \frac{1}{5}I, -\frac{4}{15} + \frac{1}{5}I \right]$$

On rappelle que l'on peut aussi utiliser les **séries de Laurent** (voir chapitre 11, *Développements en séries*)

$$> r:=\text{map}(q \rightarrow \text{coeff}(\text{convert}(\text{series}(f(z), z=q), \text{polynom}), z-q, -1), p);$$

$$r := \left[\frac{3}{5} - \frac{1}{5}I, -\frac{4}{15} + \frac{1}{5}I \right]$$

ou une **définition des résidus**. Remarquer la notation particulière pour **diff** qui utilise une liste et autorise ainsi la dérivation d'ordre 0, c'est-à-dire l'identité. Elle est nécessaire ici si on veut avoir une expression générale valable pour tout n positif (l'ordre du pôle), y compris pour $n = 1$ (voir chapitre 8, *Dérivation* ou l'aide en ligne de **diff**, pourquoi **simplify**?).

$$> \text{Residu}:=(\text{pole}, n::\text{posint}) \rightarrow \text{eval}(\text{diff}(\text{simplify}($$

$$(z-\text{pole})^n * f(z)), [z$(n-1)]) / (n-1)!, z=\text{pole});$$

$$Residu := (pole, n::posint) \rightarrow \left(\frac{\frac{\partial^{n-1}}{\partial z^{n-1}} \text{simplify}((z - pole)^n f(z))}{(n - 1)!} \right) \Bigg|_{z = pole}$$

L'utilisation de **map** est possible ici parce que les deux pôles ont le même ordre. Il faudrait sinon faire des calculs séparés pour chaque pôle.

```
> r:=map(Residu,p,1); # 1 est l'ordre du pôle et 0!=1
r := \left[ \frac{3}{5} - \frac{1}{5} I, - \frac{4}{15} + \frac{1}{5} I \right]
```

Le cercle C contient les deux pôles et on peut vérifier le théorème

```
> 2*I*Pi*Sum(''r[i]'', i=1..nops(r)):%=value(%);
2 I \pi \left( \sum_{i=1}^2 r_i \right) = \frac{2}{3} I \pi
```

ou encore

```
> 2*I*Pi*add(i, i=r);
\frac{2}{3} I \pi
```

Le cercle C' suivant ne contient que le pôle $z = 1$

```
> o:=1+I:
R:=sqrt(2):
Int(f(z), z=``C'``..``)=
int((f@C)(o,R,theta)*diff(C(o,R,theta), theta), theta=0..2*Pi);
\int_{C'} \frac{z + 1}{(3 z + I) (z - 1)} dz = \frac{6}{5} I \pi + \frac{2}{5} \pi
```

et le théorème des résidus donne

```
> '2*I*Pi*r[1]'=expand(2*I*Pi*r[1]);
2 I \pi r_1 = \frac{6}{5} I \pi + \frac{2}{5} \pi
```

Le cercle C'' ne contient que le pôle $z = -1/3 * i$

```
> o:=0:
R:=1/2:
Int(f(z), z=``C''``..``)=
int((f@C)(o,R,theta)*diff(C(o,R,theta), theta), theta=0..2*Pi);
'2*I*Pi*r[2]'=expand(2*I*Pi*r[2]);
```

$$\int_{C''} \frac{z+1}{(3z+I)(z-1)} dz = -\frac{8}{15} I\pi - \frac{2}{5}\pi$$

$$2I\pi r_2 = -\frac{8}{15} I\pi - \frac{2}{5}\pi$$

Enfin le cercle C''' ne contient aucun pôle et

```
> o:=2+I;
R:=1;
Int(f(z),z=``C''..``)=
int((f@C)(o,R,theta)*diff(C(o,R,theta),theta),theta=0..2*Pi);

```

$$\int_{C'''} \frac{z+1}{(3z+I)(z-1)} dz = 0$$

10) **Changements de variables** : signalons enfin la possibilité d'effectuer des changement de variables dans une intégration avec la fonction **student[changevar]**.

Transformations intégrales et Transformées de Fourier Rapides

MAPLE possède une bibliothèque, **inttrans**, permettant de calculer des transformations intégrales et transformation inverses. La fonction **addtable** permet d'ajouter des transformations symboliques (seulement pour la durée d'une session). La fonction **savetable** permet de sauvegarder de telles tables (voir l'aide en ligne).

```
> restart;
with(inttrans);
[addtable,fourier,fouriercos,fouriersin,hankel,hilbert,invfourier,invhilbert,invlaplace,invmellin,
laplace,mellin,savetable]
```

L'identification des transformations ne pose aucun problème et leurs définitions sont données dans l'aide en ligne. Par exemple

```
> ?inttrans,hankel
```

L'exemple suivant montre la construction d'une fonction f et de sa transformée de Fourier F , les deux dépendant d'un paramètre p que l'on suppose positif. Le premier argument est l'expression de la fonction f exprimée par $f(x, p)$. Le second argument indique la variable d'intégration x (qui aurait pu être p) et le troisième la variable de transformation ω . La fonction **fourier**, comme **int**, s'applique sur une expression et renvoie une expression.

```
> f:=(x,p)->p*x/(x^2+p^2);
Fe:=fourier(f(x,p),x,omega) assuming p>0;
```

$$f := (x, p) \rightarrow \frac{p x}{x^2 + p^2}$$

$$Fe := I p \pi (-e^{(-p \omega)} \text{Heaviside}(\omega) + e^{(p \omega)} \text{Heaviside}(-\omega))$$

On transforme maintenant l'expression Fe en une fonction F avec **unapply**

```
> F:=unapply(Fe,omega,p);
F(1,1);

$$F := (\omega, p) \rightarrow I p \pi (e^{(p \omega)} \text{Heaviside}(-\omega) - e^{(-p \omega)} \text{Heaviside}(\omega))$$


$$-I \pi e^{(-1)}$$

```

Notez la différence de résultat avec la fonction **fouriersin**

```
> Fs:=fouriersin(f(x,p),x,omega) assuming p>0;
eval(Fs,{p=1,omega=1});
```

$$Fs := \frac{1}{2} p \sqrt{2} \sqrt{\pi} e^{(-p \omega)}$$

$$\frac{1}{2} \sqrt{2} \sqrt{\pi} e^{(-1)}$$

qui est auto-inverse

```
> fouriersin(Fs,omega,x);
```

$$\frac{p x}{p^2 + x^2}$$

On peut également calculer une transformation de Fourier inverse avec la fonction **invfourier**

```
> Q:=exp(-x^2);
fourier(% ,x,t);
simplify(invfourier(% ,t,x));
```

$$Q := e^{(-x^2)}$$

$$e^{\left(-\frac{1}{4} t^2\right)} \sqrt{\pi}$$

$$e^{(-x^2)}$$

Vérifions que MAPLE "connaît" le théorème relatif à la transformée de Fourier d'un produit de convolution : *la TF d'un produit de convolution est égale au produit des TF.*

```
> a:=Int(h(x)*g(t-x),x=-infinity..infinity):
TF(a)=fourier(value(a),t,x);
```

$$TF\left(\int_{-\infty}^{\infty} h(x) g(t - x) dx\right) = \text{fourier}(h(t), t, x) \text{fourier}(g(t), t, x)$$

De même on peut calculer des transformations de Laplace et Laplace inverses grâce aux fonctions **laplace** et **invlaplace**.

```
> laplace(exp(-x),x,t);
invlaplace(1/(1+t),t,x);
```

$$\frac{1}{1 + t}$$

Transformées de Fourier Rapide (FFT)

MAPLE sait également calculer des FFT ("Fast Fourier Transform") avec les fonctions **DiscreteTransforms[FourierTransform]** et **DiscreteTransforms[InverseFourierTransform]** (on notera que ces fonctions remplacent les anciennes fonctions **FFT** et **iFFT**).

Voici un petit exemple pratique qui ne dispense pas de consulter l'aide en ligne qui contient d'autres options non décrites ici. On dispose d'un fichier de données contenant les valeurs d'un signal *réel* associées à un échantillonage temporel régulier. On sait également qu'il correspond à une durée d'enregistrement total $T_{\max} = 123.0$ "unités de temps". Voici comment se présente le début du fichier:

```
6.00000
 3.06996
-0.488329
 0.418124
 1.70956
...

```

(on peut aussi imaginer un signal complexe sur deux colonnes). La première opération va donc consister à transférer par lecture vers MAPLE les données du fichier qu'il faut d'abord localiser sur l'ordinateur. On associe ici au nom **Fichier** le nom du fichier avec son chemin d'accès. La syntaxe de cette commande *dépend du système d'exploitation utilisé*. Ici on "concatène" (on juxtapose) avec la fonction **cat** deux chaînes de caractères. La première, résultat de la commande "**kernelopts(homedir)**", correspond au répertoire d'entrée de l'utilisateur, la seconde complète le(s) "répertoire(s) fils" et finalement le nom du fichier (voir le chapitre 20, *Lecture et écriture de fichiers* pour plus de détails):

```
> restart;
Fichier:=cat(kernelopts(homedir), "/Desktop/Fourier.dat");
```

A l'aide de la fonction **readdata** (voir aussi le chapitre 20) on lit le contenu du fichier que l'on transfère dans la *liste X*.

```
> X:=readdata(Fichier,[float]):
```

On donne également la longueur temporelle totale (unité de temps arbitraire) associée aux données du fichier

```
> T[max]:=123.0;
Tmax := 123.0
```

On récupère très simplement le nombre de données de la liste (du fichier)

```
> N:=nops(X);
N := 1021
```

Attention: les fonctions de Transformations de Fourier Rapides de MAPLE demandent que les données, qu'elles soient *réelle ou complexes*, soient transmises par arguments sous forme de **vecteur** (Vector) ou de **tableau** (Array). Il faut donc transformer la liste *X* en un tableau ou un vecteur. Par ailleurs les textes théoriques et pratiques sur ces méthodes de transformation utilisent généralement une indexation des éléments de 0 à $N - 1$. Ce n'est *pas une obligation*, mais si l'on veut utiliser cette indexation on écrira

```
> X:=Array(0..N-1,X);
```

Sinon, on peut aussi écrire simplement

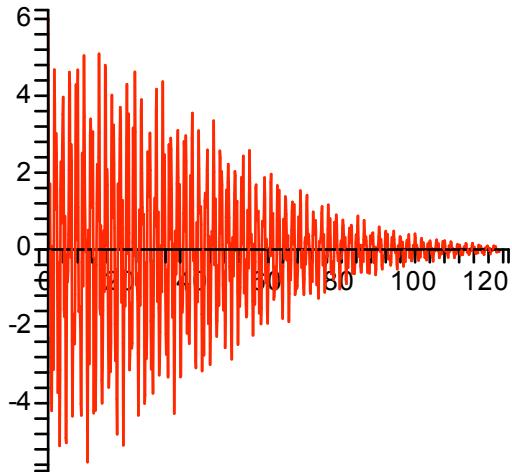
```
> X:=Array(X);
```

ou (un vecteur ne peut pas être indexé de 0 à $N - 1$)

```
> X:=Vector(X);
```

Voici la représentation graphique de ce signal (voir chapitre 16, *Graphisme 2D* pour plus de détails)

```
> plot([seq([i*T[max]/(N-1),X[i]],i=0..N-1)]);
```



La FFT se calcule alors simplement. On notera que le résultat est un tableau (un vecteur si X avait été un vecteur) indexé de 1 à N malgré l'indexation du tableau X de 0 à $N - 1$.

```
> Q:=DiscreteTransforms[FourierTransform](X);
```

$$Q := \begin{bmatrix} 1 .. 1021 \text{ 1-D Array} \\ \text{Data Type: anything} \\ \text{Storage: rectangular} \\ \text{Order: Fortran_order} \end{bmatrix}$$

Avec les Transformées de Fourier Rapides, on sait que le nombre d'éléments N de l'échantillon est un point important. La FFT est d'autant plus efficace, en termes de temps de calcul, que N est factorisable par des puissances les plus élevées possibles des entiers les plus petits possibles: $N = 2^k 3^p \dots$, la meilleure des situations étant $N = 2^m$. Manque de chance, notre échantillon correspond à la pire des situations car N est premier ! La FFT se comportera alors, du point de vue de l'efficacité du temps de calcul, comme une simple DFT (Transformée de Fourier Discrète).

```
> ifactor(N);
isprime(N);
(1021)
true
```

La technique, dite du "**padding**", consiste à rallonger artificiellement la longueur de l'échantillon avec des 0. On indique ici à Maple de réaliser cette opération avec un *maximum* de 5 zéros. Maple fait, dans cette limite, "au mieux" pour se rapprocher des critères donnés précédemment. Cette technique est aussi utilisée pour améliorer (artificiellement, cela va de soi) la résolution de la FFT. Si l'on veut **imposer** un nombre M de termes on écrira par exemple

```
> X:=Array([seq(X[i],i=1..N),seq(0,i=N+1..M)]);
```

La situation la plus favorable, dans la limite imposée ici par la valeur 5 du padding, correspond à $1024 = 2^{10}$, ce que donne **FourierTransform** (pour un seul calcul et pour un échantillon aussi petit, la différence entre FFT et DFT ne sera pas sensible). On notera que l'utilisation de l'option "padding" modifie la réponse de la fonction qui devient un *suite* (que l'on peut assigner de façon "multiple"), et qui donne comme premier élément le nombre de termes choisi par la fonction FFT

```
> M,Q:=DiscreteTransforms[FourierTransform](x,padding=5);
```

$$M, Q := [1024], \begin{bmatrix} I..1024 \text{ } 1\text{-D Array} \\ \text{Data Type: anything} \\ \text{Storage: rectangular} \\ \text{Order: Fortran_order} \end{bmatrix}$$

On pourra trouver curieux que le nouveau nombre d'éléments soit donné dans une liste à un élément (M est une liste), mais les fonctions FFT de MAPLE peuvent traiter des données multidimensionnelles (par exemple une image 2D) et une liste aurait alors toute sa justification en donnant la taille de chacune des dimensions.

```
> M:=op(M);
```

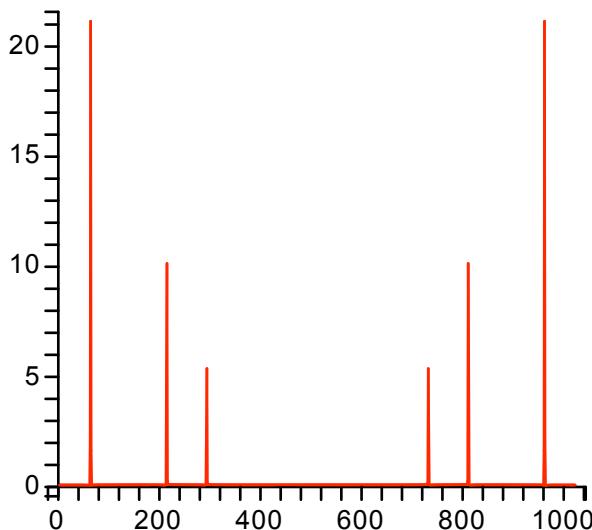
$$M := 1024$$

Si on le souhaite, on peut ré-indexer le tableau de la TF de 0 à $M - 1$.

```
> Q:=Array(0..M-1,Q):
```

Le tracé de la **partie réelle** de la transformée de Fourier montre la symétrie propres aux signaux réels et fait clairement apparaître 3 fréquences caractéristiques dans le signal numérisé du fichier.

```
> plot([seq([i,Re(Q[i])],i=0..M-1)]);
```



On pourrait de même tracer la partie imaginaire de la TF qui montrerait le caractère hermitien de la symétrie. La résolution fréquentielle de la TF est donnée, avec les données d'origines, par $1/T_{\max}$. Mais il faut tenir compte du "padding"

```
> delta[t]:=N/M/T[max];
```

$$\delta_t := 0.008106262703$$

On veut maintenant déterminer les trois fréquences.

Exercice : la méthode proposée est laissée à la sagacité du lecteur...

```

> Ir:=0..100:
member(max(seq(Re(Q[i]),i=Ir)),[seq(Re(Q[i]),i=Ir)],'k[1]'):
k[1]:=op(1,Ir)+k[1]-1:
nu[1]:=k[1]*delta[t];
v1 := 0.5188008130

> Ir:=100..500:
member(max(seq(Re(Q[i]),i=Ir)),[seq(Re(Q[i]),i=Ir)],'k[2]'):k[2]:
k[2]:=op(1,Ir)+k[2]-1:
nu[2]:=k[2]*delta[t];
v2 := 1.742846481

> Ir:=220..512:
member(max(seq(Re(Q[i]),i=Ir)),[seq(Re(Q[i]),i=Ir)],'k[3]'):k[3]:
k[3]:=op(1,Ir)+k[3]-1:
nu[3]:=k[3]*delta[t];
v3 := 2.383241235

```

Distributions

MAPLE connaît la fonction de Heaviside, qui est aussi une distribution, mais également la distribution de Dirac qui n'est pas définie comme une fonction

```

> restart:
Dirac(0);
Dirac(1);
Dirac(0)
0

```

Notez que les ordres **restart** ont fait perdre les définitions des fonctions de la bibliothèque **inttrans**. L'exemple suivant rappelle comment on peut utiliser une de ces fonctions sans la rendre accessible de façon permanente. La distribution de Dirac peut être définie formellement par une transformation de Fourier

```

> c:=1/(2*Pi);
inttrans[fourier](c,x,t);
c :=  $\frac{1}{2\pi}$ 
Dirac(t)

```

```

> Int(Dirac(x-p)*g(x),x=-infinity..infinity):
% = value(%);

```

$$\int_{-\infty}^{\infty} \text{Dirac}(x - p) g(x) dx = g(p)$$

La distribution dérivée de la distribution de Heaviside est la distribution de Dirac

```
> Diff(Heaviside(x-p),x): % = value(%);
```

$$\frac{\partial}{\partial x} \text{Heaviside}(x - p) = \text{Dirac}(x - p)$$

Les *distributions dérivées* de Dirac sont notées **Dirac(k,,)** où k est l'ordre de dérivation.

```
> (D@@2)(Dirac);
z → Dirac(2, z)
```

```
> Dirac(p*x);
Diff(Dirac(p*x), x$3):%:=value(%);
```

$$\frac{\partial^3}{\partial x^3} \text{Dirac}(p x) = \text{Dirac}(3, p x) p^3$$

Ces calculs sont bien conformes à la théorie des distributions.

Rappel : une fonction, sous certaines conditions, peut définir une distribution mais celle de Dirac, δ , n'est pas une fonction, c'est une mesure (les dérivées de δ ne sont pas des mesures). Mais on la manipule souvent de façon incorrecte (comme le fait ici MAPLE) en la considérant comme une densité sommable de masse 1... mais ne rentrons pas trop dans les détails et considérons l'écriture suivante comme symbolique

```
> Int(Dirac(x-t), x=-infinity..infinity): %:=value(%);
```

$$\int_{-\infty}^{\infty} \text{Dirac}(x - t) dx = 1$$

Afin de distinguer la dérivation d'une distribution de celle d'une fonction nous noterons ∂ la première et **D** la seconde, **D** étant l'opérateur standard de MAPLE. En fait MAPLE ne fait pas cette distinction. Conformément à la théorie, l'application de cette distribution sur une fonction test φ (indéfiniment dérivable à support compact) s'écrit

$$\langle \partial^n \delta, \varphi \rangle = (-1)^n \langle \delta, D^n \varphi \rangle = (-1)^n (D^n \varphi)(0)$$

C'est bien ce que nous répond MAPLE, par exemple pour $n=3$

```
> Int(Diff(Dirac(t), t$3)*phi(t), t=-infinity..infinity): % = value(%);

```

$$\int_{-\infty}^{\infty} \left(\frac{d^3}{dt^3} \text{Dirac}(t) \right) \varphi(t) dt = -D^{(3)}(\varphi)(0)$$

Exercice : Expliquez cette réponse (juste, mais qui semble fausse). Corrigez.

```
> f:=x->exp(-x^2);
F:=unapply(inttrans[fourier](f,x,t),t);
f:=x → e(-x2)
F := t → 2fπ Dirac(t)
```

Fonction map et opérateurs d'intégration

Au même titre que n'importe quel opérateur, **int**, **Int**, **intat**, **fourier** etc., peuvent être appliqués avec la fonction **map**

```
> M:=Matrix(3,3,[seq([seq((i+j+1)*x^(i+j),j=0..2)],i=0..2));
T:=map(intat,M,x=t);
```

$$M := \begin{bmatrix} 1 & 2x & 3x^2 \\ 2x & 3x^2 & 4x^3 \\ 3x^2 & 4x^3 & 5x^4 \end{bmatrix}$$

$$T := \begin{bmatrix} t & t^2 & t^3 \\ t^2 & t^3 & t^4 \\ t^3 & t^4 & t^5 \end{bmatrix}$$

ou encore

```
> M:=Matrix([seq([seq(cos((i+j-1)*x)/Pi,j=1..2)],i=1..2));
F:=map(inttrans[fourier],M,x,omega);
```

$$M := \begin{bmatrix} \frac{\cos(x)}{\pi} & \frac{\cos(2x)}{\pi} \\ \frac{\cos(2x)}{\pi} & \frac{\cos(3x)}{\pi} \end{bmatrix}$$

$$F := \begin{bmatrix} \text{Dirac}(\omega + 1) + \text{Dirac}(\omega - 1) & \text{Dirac}(\omega + 2) + \text{Dirac}(\omega - 2) \\ \text{Dirac}(\omega + 2) + \text{Dirac}(\omega - 2) & \text{Dirac}(\omega - 3) + \text{Dirac}(\omega + 3) \end{bmatrix}$$

[>

11 - Développements en séries

L'opérateur series

On peut obtenir le développement en série (Taylor, Laurent ou autres) d'une *expression* à l'aide de la fonction **series**. Le premier argument est l'expression à développer et le second la variable par rapport à laquelle on développe

```
> restart:  
series(exp(-x/a),x);  
1 -  $\frac{1}{a}x + \frac{1}{2a^2}x^2 - \frac{1}{6a^3}x^3 + \frac{1}{24a^4}x^4 - \frac{1}{120a^5}x^5 + O(x^6)$ 
```

Le dernier terme $O(x^6)$ indique que le développement est limité à l'ordre 6. Il a été effectué *par défaut* autour de $x = 0$. MAPLE ne pourra donc pas calculer avec cette syntaxe le développement suivant, l'expression présentant une singularité pour $a = 0$

```
> series(exp(-x/a),a);  
Error, (in series/exp) unable to compute series
```

Pour obtenir un développement autour d'une valeur régulière, par exemple $a = 1$, on écrira

```
> series(exp(-x/a),a=1);  
 $e^{(-x)} + e^{(-x)}x(a-1) + e^{(-x)}\left(-x + \frac{1}{2}x^2\right)(a-1)^2 + e^{(-x)}\left(x - x^2 + \frac{1}{6}x^3\right)(a-1)^3$   
 $+ e^{(-x)}\left(-x + \frac{3}{2}x^2 - \frac{1}{2}x^3 + \frac{1}{24}x^4\right)(a-1)^4 + e^{(-x)}\left(x - 2x^2 + x^3 - \frac{1}{6}x^4 + \frac{1}{120}x^5\right)(a-1)^5 +$   
 $O((a-1)^6)$ 
```

Un développement limité à l'ordre 6 peut être insuffisant. Notez bien que la fonction **series** a pour argument *l'expression de la fonction*, $(f - g)(x)$, et pas seulement la composition $f - g$

```
> f:=x->sinh(x)-sin(x):  
g:=x->x^3/3:  
series((f-g)(x),x);  
 $O(x^6)$ 
```

L'ordre des développements est fixé à 6 par défaut par la variable globale **Order**. Noter que **O** est un nom protégé de MAPLE.

```
> Order;  
O:=3;
```

```
Error, attempting to assign to `O` which is protected
```

On peut modifier l'ordre du développement limité de deux façons. La première méthode consiste à affecter l'ordre n du développement souhaité à la variable **Order**

> **Order := n:**

Tous les développements se feront alors (jusqu'à nouvelle modification) à l'ordre n . La seconde méthode consiste à donner la valeur souhaitée en troisième argument de **series**. La valeur de l'ordre du développement est alors limitée à la seule commande:

> **series((f-g)(x),x,18);**

$$\frac{1}{2520}x^7 + \frac{1}{19958400}x^{11} + \frac{1}{653837184000}x^{15} + O(x^{18})$$

Dans l'exemple suivant a n'est pas une variable libre de f , mais plutôt un paramètre. Cependant, comme MAPLE travaille sur l'*expression de la fonction*, cette distinction disparaît totalement au moment du calcul du développement. Rapelons que MAPLE considère toujours par défaut des fonctions de variables complexes à valeurs complexes.

> **f:=x->sin(a*x);**
series(f(x),a=ln(-1),3);

$$f := x \rightarrow \sin(ax)$$

$$I \sinh(\pi x) + \cosh(\pi x) x (a - I\pi) - \frac{1}{2} I \sinh(\pi x) x^2 (a - I\pi)^2 + O((a - I\pi)^3)$$

Exercice: que signifie ce résultat ?

> **g:=x->exp(sin(x^2));**
series(g+cos(x),x);

$$g := x \rightarrow e^{\sin(x^2)}$$

$$(g + 1) - \frac{1}{2}x^2 + \frac{1}{24}x^4 + O(x^6)$$

La fonction **series** calcule également les **séries de Laurent** (comment appelle-t-on le terme r_{-1} ?)

> **ez:=1/(z*(z-1)*(z-2));**
s1:=series(ez,z=1);
r_1:=coeff(%,(z-1),-1);

$$ez := \frac{1}{z(z-1)(z-2)}$$

$$s1 := -(z-1)^{(-1)} - (z-1) - (z-1)^3 + O((z-1)^5)$$

$$r_{-1} := -1$$

Les séries peuvent aussi se présenter sous une forme plus générale en termes de puissances non entières de la variable (séries de Puiseux) ou de coefficients qui sont dépendants de celle-ci

> **s2:=series(sqrt(sinh(x))/x^4,x);**
s3:=series(sin(x^x),x,3);

$$s2 := \frac{1}{x^{(7/2)}} + \frac{1}{12x^{(3/2)}} + \frac{1}{1440}\sqrt{x} + O(x^{(5/2)})$$

$$s3 := \sin(1) + \cos(1) \ln(x) x + \left(-\frac{1}{2} \sin(1) \ln(x)^2 + \frac{1}{2} \cos(1) \ln(x)^2 \right) x^2 + O(x^3)$$

On peut également développer en série une intégrale

> **f:=x->exp(-x^2);**

```
Int(f(x),x):
%=series(%,x,10);
```

$$f := x \rightarrow e^{(-x^2)}$$

$$\int e^{(-x^2)} dx = \left(x - \frac{1}{3}x^3 + \frac{1}{10}x^5 - \frac{1}{42}x^7 + \frac{1}{216}x^9 + O(x^{11}) \right)$$

ou intégrer un développement en série

```
> A:=Int(''series(cos(x),x)'',x):%value(%);
'series(sin(x),x)'=series(sin(x),x);
```

$$\int \text{series}(\cos(x), x) dx = \left(x - \frac{1}{6}x^3 + \frac{1}{120}x^5 + O(x^7) \right)$$

$$\text{series}(\sin(x), x) = \left(x - \frac{1}{6}x^3 + \frac{1}{120}x^5 + O(x^7) \right)$$

Transformation d'un développement en expression ou fonction

Il est fréquent que l'on veuille utiliser l'expression d'un développement en série comme une expression ou comme une fonction (approximation d'une fonction par exemple). Cependant ces développements ne se manipulent pas comme des expressions ordinaires et ne sont pas, en tout cas, des polynômes

```
> restart;
s:=series(sin(x^2-1),x=1);
type(s,polynom);
```

$$s := 2(x - 1) + (x - 1)^2 - \frac{4}{3}(x - 1)^3 - 2(x - 1)^4 - \frac{11}{15}(x - 1)^5 + O((x - 1)^6)$$

false

En conséquence un développement en série ne peut pas être transformé *directement* en une fonction numérique.

```
> f:=unapply(s,x);
f(2);
f:=x → 2(x - 1) + (x - 1)^2 -  $\frac{4}{3}(x - 1)^3 - 2(x - 1)^4 - \frac{11}{15}(x - 1)^5 + O((x - 1)^6)$ 
```

Error, (in f) invalid substitution in series

Mais la fonction **convert** avec l'argument **polynom** permet de transformer une structure de type **series** en une expression polynomiale

```
> p:=convert(s,polynom);
type(p,polynom);
```

$$p := 2x - 2 + (x - 1)^2 - \frac{4}{3}(x - 1)^3 - 2(x - 1)^4 - \frac{11}{15}(x - 1)^5$$

true

ou une fraction rationnelle

```
> convert(s, ratpoly);

$$\frac{-\frac{19}{30}(x-1)^3 - \frac{4}{5}(x-1)^2 + 2x - 2}{\frac{19}{10} - \frac{9}{10}x + \frac{4}{5}(x-1)^2}$$

```

Il reste ensuite à transformer cette expression en fonction avec l'opérateur **unapply**. Généralement on regroupe les deux opérations en une seule commande

```
> f:=unapply(convert(s, polynom), x);
f(2);

$$f := x \rightarrow 2x - 2 + (x-1)^2 - \frac{4}{3}(x-1)^3 - 2(x-1)^4 - \frac{11}{15}(x-1)^5 - \frac{16}{15}$$

```

Le terme **polynom** de cette transformation doit-être compris au sens large puisque le résultat n'est pas nécessairement un polynôme !

```
> s1:=series(1/(z*(z-1)*(z-2)), z=1);
convert(s1, polynom);
type(% , polynom), type(% , ratpoly);
s1 := -(z-1)^{(-1)} - (z-1) - (z-1)^3 + O((z-1)^5)
-  $\frac{1}{z-1} - z + 1 - (z-1)^3$ 
false, true
```

```
> s2:=series(sqrt(sinh(x))/x^4, x);
convert(s2, polynom);
type(% , polynom), type(% , ratpoly);
s2 :=  $\frac{1}{x^{(7/2)}} + \frac{1}{12x^{(3/2)}} + \frac{1}{1440}\sqrt{x} + O(x^{(5/2)})$ 
 $\frac{1}{x^{(7/2)}} + \frac{1}{12x^{(3/2)}} + \frac{1}{1440}\sqrt{x}$ 
false, false
```

```
> s3:=series(sin(x^x), x, 3);
convert(s3, polynom);
type(% , polynom), type(% , ratpoly);
s3 := sin(1) + cos(1) ln(x)x +  $\left( -\frac{1}{2}\sin(1)\ln(x)^2 + \frac{1}{2}\cos(1)\ln(x)^2 \right)x^2 + O(x^3)$ 
sin(1) + cos(1) ln(x)x +  $\left( -\frac{1}{2}\sin(1)\ln(x)^2 + \frac{1}{2}\cos(1)\ln(x)^2 \right)x^2$ 
false, false
```

Comme précédemment ces développements peuvent être transformés en fonctions. Par exemple

```
> s:=unapply(evalf(convert(s3,polynom)),x);
s := x → 0.8414709848 + 0.5403023059 ln(x) x - 0.1505843394 ln(x)² x²
```

On compare ici la fonction avec son approximation. Bien entendu, cette dernière se dégrade quand on s'éloigne du point de développement (ici $x = 0$; voir une autre exemple d'approximation plus efficace au chapitre 7, *Fonctions*)

Exercice: Quelle est l'utilité de la fonction **evalf**? Enlevez-la et essayez.

```
> sin(0.01^0.01),s(0.01);
sin(0.5^0.5),s(0.5);
0.8163094580, 0.8162697910
0.6496369391, 0.6361292999
```

Les fonctions **taylor** et **mtaylor**

La fonction **taylor** est simplement une restriction de la fonction **series** aux seuls développements en série de Taylor.

```
> restart;
taylor(exp(x),x);
1 + x + 1/2 x² + 1/6 x³ + 1/24 x⁴ + 1/120 x⁵ + O(x⁶)

> type(%,series);
true
```

Comme l'indique les messages renvoyés, certaines expressions n'ont pas ce type de développement

```
> taylor(1/(z*(z-1)*(z-2)),z);# pas de développement autour du pole z=0
Error, does not have a taylor expansion, try series()
```

alors que **series** donnera la série de Laurent

```
> series(1/(z*(z-1)*(z-2)),z);
1/2 z(-1) + 3/4 + 7/8 z + 15/16 z² + 31/32 z³ + 63/64 z⁴ + 127/128 z⁵ + O(z⁶)
```

ou encore

```
> taylor(x^x,x); # pas de développement de Taylor autour de x=0
Error, does not have a taylor expansion, try series()

> series(x^x,x);
1 + ln(x) x + 1/2 ln(x)2 x² + 1/6 ln(x)3 x³ + 1/24 ln(x)4 x⁴ + 1/120 ln(x)5 x⁵ + O(x⁶)
```

La fonction **mtaylor** permet d'obtenir des développements en série de Taylor de fonctions de plusieurs variables. Le premier argument est l'expression à développer, le deuxième la liste des variables par rapport auxquelles on souhaite le développement et le troisième (facultatif), l'ordre *total* du développement. On utilise ici une fonction non définie *f* pour montrer comment MAPLE ordonne ses développements (on rappelle que les indices de l'opérateur **D** désigne les numéros d'ordre de la variable par rapport à laquelle on dérive)

```
> f:='f':
T:=mtaylor(f(x,y,z),[x,y,z],3);
```

$$\begin{aligned}
T := & f(0, 0, 0) + D_1(f)(0, 0, 0)x + D_2(f)(0, 0, 0)y + D_3(f)(0, 0, 0)z + \frac{1}{2}D_{1,1}(f)(0, 0, 0)x^2 \\
& + xD_{1,2}(f)(0, 0, 0)y + xD_{1,3}(f)(0, 0, 0)z + \frac{1}{2}D_{2,2}(f)(0, 0, 0)y^2 + yD_{2,3}(f)(0, 0, 0)z \\
& + \frac{1}{2}D_{3,3}(f)(0, 0, 0)z^2
\end{aligned}$$

Attention: le développement n'est plus du type *series*, mais *polynom*

```
> type(M,series),type(M,polynom);
false, true
```

Exercice: comparez avec le résultat précédent

```
> series(f(x,0,0),x);
f(0, 0, 0) + D_1(f)(0, 0, 0)x + \frac{1}{2}D_{1,1}(f)(0, 0, 0)x^2 + \frac{1}{6}D_{1,1,1}(f)(0, 0, 0)x^3 + \frac{1}{24}D_{1,1,1,1}(f)(0, 0, 0)x^4
+ \frac{1}{120}D_{1,1,1,1,1}(f)(0, 0, 0)x^5 + O(x^6)
```

Ici on développe autour de $x = 0$ et $z = 1$ mais pas autour de y :

```
> mtaylor(f(x,y,z),[x,z=1],3);
f(0, y, 1) + D_1(f)(0, y, 1)x + D_3(f)(0, y, 1)(z - 1) + \frac{1}{2}D_{1,1}(f)(0, y, 1)x^2 + xD_{1,3}(f)(0, y, 1)(z - 1)
+ \frac{1}{2}D_{3,3}(f)(0, y, 1)(z - 1)^2
```

Exercice: comparer avec l'exercice précédent

```
> mtaylor(f(x,0,0),x);
f(0, 0, 0) + D_1(f)(0, 0, 0)x + \frac{1}{2}D_{1,1}(f)(0, 0, 0)x^2 + \frac{1}{6}D_{1,1,1}(f)(0, 0, 0)x^3 + \frac{1}{24}D_{1,1,1,1}(f)(0, 0, 0)x^4
+ \frac{1}{120}D_{1,1,1,1,1}(f)(0, 0, 0)x^5
```

Ces exemples plus concrets montrent, grâce au rôle symétrique joué par les variables, comment s'organisent les calculs.

```
> f:=(x,y,z)->exp(-(x^2+y^2+z^2));
mtaylor(f(x,y,z),[x,y],5);
mtaylor(f(x,y,z),[y,z],5);
```

$$\begin{aligned}
f := (x, y, z) \rightarrow e^{(-x^2 - y^2 - z^2)} \\
e^{(-z^2)} - e^{(-z^2)}x^2 - e^{(-z^2)}y^2 + \frac{1}{2}e^{(-z^2)}x^4 + e^{(-z^2)}y^2x^2 + \frac{1}{2}e^{(-z^2)}y^4 \\
e^{(-x^2)} - e^{(-x^2)}y^2 - e^{(-x^2)}z^2 + \frac{1}{2}e^{(-x^2)}y^4 + e^{(-x^2)}z^2y^2 + \frac{1}{2}e^{(-x^2)}z^4
\end{aligned}$$

Le premier des développements suivants montre que l'ordre maximal est 4 sans terme impair (parce que la fonction est totalement symétrique par rapport à l'origine). On peut également donner un poids à chacune des

variables à l'aide d'une liste d'entiers positifs afin d'influencer l'ordre des termes du développement. L'affectation d'un poids 2 au développement par rapport à x diminue d'un facteur 2 l'ordre du développement relativement à cette variable (on passe d'un terme d'ordre maximal 4 à un terme d'ordre maximal 2 pour x). Le troisième exemple montre un effet identique pour la variable y .

```
> mtaylor(f(x,y,z),[x,y,z],5);
mtaylor(f(x,y,z),[x,y,z],5,[2,1,1]);
mtaylor(f(x,y,z),[x,y,z],5,[1,2,1]);

$$1 - x^2 - y^2 - z^2 + \frac{1}{2}x^4 + z^2x^2 + y^2x^2 + \frac{1}{2}y^4 + z^2y^2 + \frac{1}{2}z^4$$


$$1 - y^2 - z^2 - x^2 + \frac{1}{2}y^4 + z^2y^2 + \frac{1}{2}z^4$$


$$1 - x^2 - z^2 - y^2 + \frac{1}{2}x^4 + z^2x^2 + \frac{1}{2}z^4$$

```

La bibliothèque powseries

MAPLE possède une bibliothèque contenant des fonctions permettant de créer et de manipuler des développements en série.

```
> restart;
with(powseries);
[compose, evalpow, inverse, multconst, multiply, negative, powadd, powcos, powcreate, powdiff, powexp,
powint, powlog, powpoly, powsin, powsolve, powsqrt, quotient, reversion, subtract, template, tpsform]
```

Nous ne donnerons que quelques exemples. La fonction **powcreate** permet de créer une procédure (ou une fonction si on préfère) qui génère les termes de la série. Dans cet exemple $q(n)$ désignera le coefficient du terme de puissance n et vaudra $(-1)^n 2/(n+1)!$ On peut aussi modifier certains coefficients explicitement. Par exemple $q(5)$ vaudra ici -4 et non $-2/(5+1)!$ *Attention*, par précaution on désassignera toujours le nom de la procédure avant d'utiliser **powcreate**

```
> q:='q':
powcreate(q(n)=2*(-1)^n/(n+1)!,q(5)=-4,q(7)=0);
```

La fonction **powcreate** a générée une procédure qui a pour nom q

```
> eval(q);
q(0),q(1),q(2),q(3),q(5),q(6),q(7),q(8);
proc(powparm) ... end proc;
```

$$2, -1, \frac{1}{3}, \frac{-1}{12}, -4, \frac{1}{2520}, 0, \frac{1}{181440}$$

On peut également construire les coefficients par récurrence. Ici le ou les premiers termes devront être donnés pour initialiser la récurrence. Cet exemple montre aussi que les coefficients sont modifiés pendant la génération et non après. Ainsi $r(11) = r(9)$ parce que l'on a imposé $r(10) = 0$.

```
> r:='r':
powcreate(r(n)=r(n-2)+r(n-1),r(0)=1,r(1)=2,r(10)=0);
r(2),r(3),r(9),r(10),r(11);
3, 5, 89, 0, 89
```

Avec la fonction **tpsform** on crée la série en donnant le nom de la procédure générant les termes, le nom de la

variable et éventuellement, l'ordre

```
> S:=tpsform(q,x,9);# le terme en x^7 est nul  
S := 2 - x +  $\frac{1}{3}x^2 - \frac{1}{12}x^3 + \frac{1}{60}x^4 - 4x^5 + \frac{1}{2520}x^6 + \frac{1}{181440}x^8 + O(x^9)$ 
```

Parmi les fonctions de **powseries** on trouve une fonction de dérivation **powdiff**.

Attention : cette fonction, comme les autres de **powseries**, s'applique sur la procédure de génération des coefficients et non sur la série elle même :

```
> qd:=powdiff(q);  
Sd:=tpsform(qd,x);  
qd := proc(powparm) ... end proc;  
Sd := -1 +  $\frac{2}{3}x - \frac{1}{4}x^2 + \frac{1}{15}x^3 - 20x^4 + \frac{1}{420}x^5 + O(x^6)$ 
```

Opérandes et Types des développements en séries

Les développements obtenus avec **series** ont souvent une structure de données du type **series**.

```
> restart:  
s0:=series(exp(x-2),x=2,3);  
whattype(s0);  
type(s0,series);  
s0 := 1 + (x - 2) +  $\frac{1}{2}(x - 2)^2 + O((x - 2)^3)$   
series  
true
```

Contrairement à ce qu'on pourrait croire, ils ne sont pas toujours du type `+'.

```
> type(s0,`+`);  
false
```

Cette structure peut-être visualisée en affichant les opérandes et en sachant qu'il existe une opérande 0 qui donne l'expression $x - a$ par rapport à laquelle on développe. Le résultat de **op** est une suite donnant la succession des paires "coefficient-puissance" de l'expression :

```
> op(0,s0);  
op(s0);  
x - 2  
1, 0, 1, 1,  $\frac{1}{2}$ , 2, O(1), 3
```

On interprétera ces résultats comme $1(x - 2)^0 + 1(x - 2) + \frac{1}{2}(x - 2)^2 + \dots$

Les développements ont aussi des sous-types hiérarchisés. Par exemple, **s0** a le type **series** et **taylor** et donc **laurent**.

```
> type(s0,series) , type(s0,taylor) , type(s0,laurent);  
true, true, true
```

La série *s1* a le type **series** et **laurent** mais n'est pas un développement de Taylor.

```
> s1:=series(1/(z*(z-1)*(z-2)),z=1);
type(s1,series) , type(s1,laurent) , type(s1,taylor);
```

$$s1 := -(z - 1)^{(-1)} - (z - 1) - (z - 1)^3 + O((z - 1)^5)$$

true, true, false

Les coefficients peuvent dépendre de la variable par rapport à laquelle on effectue le développement mais la série conserve le type **series** car les puissances *entières* de *x* successives apparaissent quand même

```
> s2:=series(sin(x^x),x,3);
type(s2,series) , type(s2,taylor);
s2 := \sin(1) + \cos(1) \ln(x) x + \left( -\frac{1}{2} \sin(1) \ln(x)^2 + \frac{1}{2} \cos(1) \ln(x)^2 \right) x^2 + O(x^3)
```

true, false

Si le développement ne peut pas s'écrire sous la forme d'une série en puissances entières (positives ou négatives) de *x* - *a*, MAPLE ne peut plus utiliser cette structure qu'il remplace simplement par celle d'une somme. Le type n'est alors plus **series** mais `+`

```
> s3:=series(sqrt(sinh(x))/x^4,x);
type(s3,series) , type(s3,`+`);
s3 := \frac{1}{x^{(7/2)}} + \frac{1}{12 x^{(3/2)}} + \frac{1}{1440} \sqrt{x} + O(x^{(5/2)})
```

false, true

On rappelle que, contrairement aux développements obtenus avec **series** ou **taylor**, ceux obtenus avec **mtaylor** n'ont pas le type **series** mais seulement le type **somme** (+).

```
> mT:=mtaylor(exp(x*y),x);
type(mT,series) , type(mT,`+`);
mT := 1 + x y + \frac{1}{2} y^2 x^2 + \frac{1}{6} y^3 x^3 + \frac{1}{24} y^4 x^4 + \frac{1}{120} y^5 x^5
```

false, true

Développements asymptotiques

MAPLE permet également de calculer, avec la fonction **asympt**, des développements asymptotiques. On reconnaîtra ici un développement (dont on ne garde que le premier terme) très utilisé en mécanique statistique

```
> restart;
stat:=asympt(ln(x!),x,3);
expand(op(1,%));
stat := (\ln(x) - 1) x + \ln(\sqrt{2} \sqrt{\pi}) + \frac{1}{2} \ln(x) + O\left(\frac{1}{x}\right)
```

x ln(x) - x

La structure du résultat de **asympt** n'est pas du type **series** mais `+`.

```
> type(stat,series) , type(stat,`+`);  
false, true
```

Le développement suivant donne l'*expression* de l'approximation de Stirling (ordre 1) de la fonction factorielle

```
> simplify(convert(asympt(x!,x,1),polynom),power);  

$$\sqrt{2} \sqrt{\pi} \left( \frac{1}{x} \right)^{\left( -\frac{1}{2} - x \right)} e^{(-x)}$$

```

On construit maintenant une *fonction* d'approximation (pour x grand) d'une des fonctions de Bessel, $J_0(x)$, à l'ordre 1. On peut constater que cette fonction se comporte à l'infini comme un sinus avec un déphasage $\pi/4$ et "amorti" par une fonction lentement décroissante ($\sqrt{\frac{1}{x}}$).

```
> Bs0_asymp:=unapply(convert(asympt(BesselJ(0,x),x,1),polynom),x);  

$$Bs0\_asymp := x \rightarrow \frac{\sqrt{2} \sin\left(x + \frac{1}{4}\pi\right)}{\sqrt{\pi}} \sqrt{\frac{1}{x}}$$

```

Développements en fractions continues

Nous avons déjà mentionné que des fractions rationnelles pouvaient être développées en fractions continues. Cette possibilité s'étend à tout type d'expressions. Evidemment il s'agira alors de développements limités mais il est possible de fixer l'ordre de ceux-ci avec un argument entier supplémentaire.

```
> convert(sin(x^2)*exp(-x),confrac,x,8);  

$$\frac{x^2}{1 + \cfrac{x}{1 + \cfrac{x}{-2 + \cfrac{x}{-3 + \cfrac{x}{-\frac{2}{11} + \frac{241}{605}x}}}}}$$

```

Un tel développement possède le type *ratpoly*.

```
> type(%,ratpoly);  
true
```

Exercice : MAPLE est un calculateur puissant. Calculez le développement en fraction continue à l'ordre 5 de la fonction eulérienne $\Gamma(x)$ (écrire GAMMA(x)) ; γ est la constante d'Euler et ζ la fonction de Riemann.

12 - Sommes et Produits

Sommes

La fonction sum

La fonction **sum** permet de calculer des sommes d'un nombre fini, indéfini ou infini de termes donnés par une expression dépendant (ou non) d'un indice de sommation. On remarque dans l'écriture suivante l'utilisation de l'opérateur inert **Sum** dont l'usage est le même que pour **Eval**, **Diff**, **Int** ou **Limit**. Par exemple pour calculer la somme des 10 ou des n premiers carrés entiers on écrira

```
> restart;
Sum(i^2,i=1..10):=%value(%);
Sum(i^2,i=1..n):=%factor(value(%));
```

$$\sum_{i=1}^{10} i^2 = 385$$

$$\sum_{i=1}^n i^2 = \frac{1}{6} n (n + 1) (2 n + 1)$$

La fonction **sum** n'assigne pas de valeur à l'indice de sommation

```
> i;
```

i

_____...mais, attention_____

On devra être particulièrement attentif à une erreur fréquente qui consiste à utiliser comme nom d'indice de sommation, un nom ayant une valeur préalablement assignée. Ainsi dans l'exemple qui suit, le message est clair: MAPLE évaluant toujours les arguments avant de les transmettre à la fonction et la valeur 1 ayant été assignée à *i*, l'expression $i = 1..10$ est transmise à **sum** sous la forme $1 = 1..10$, ce qui n'a pas de sens

```
> i:=1;
sum(1/i,i=1..10);
```

i := 1

```
Error, (in sum) summation variable previously assigned, second argument evaluates to 1
= 1 .. 10
```

Avec l'exemple suivant, l'indice de sommation est non évalué en le plaçant entre '...' et le message d'erreur a disparu, mais c'est pire, le résultat est faux! (relativement à ce que l'on cherche). Ce que l'on calcule est "somme de 1 à 10 de 1", *i* étant remplacé par 1 dans l'expression à sommer:

```
> sum(1/i,'i'=1..10);
```

10

La syntaxe correcte est :

```
> sum('1/i','i'=1..10);
```

$\frac{7381}{2520}$

Mais, elle s'avère inefficace pour la conjugaison de **Sum** et **value**

```
> Sum('1/i','i'=1..10):%<math>=value(%);  
Error, (in sum) summation variable previously assigned, second argument evaluates to 1  
= 1 .. 10
```

On peut retarder encore les évaluations en écrivant...

```
> 'Sum('1/i','i'=1..10)':%<math>=value(%);  

$$\sum_{i=1}^{10} \frac{1}{i} = \frac{7381}{2520}$$

```

mais ça devient un peu pénible et on préférera la désassiguation préalable de l'indice ou l'utilisation d'un nom non assigné

```
> i:='i':  
Sum(1/i,i=1..10):%<math>=value(%);  

$$\sum_{i=1}^{10} \frac{1}{i} = \frac{7381}{2520}$$

```

Dans ce chapitre on a pris soin de n'utiliser que des indices non assignés pour alléger l'écriture.

Naturellement MAPLE ne peut pas toujours trouver une forme explicite, soit parce qu'il ne sait pas la trouver, soit parce que la série n'est pas convergente. Il se contente alors d'afficher le résultat sous la forme non évaluée comme pour le calcul intégral

```
> Sum(sin(k*x)/x,k=1..infinity):%<math>=simplify(value(%));  

$$\sum_{k=1}^{\infty} \frac{\sin(kx)}{x} = \sum_{k=1}^{\infty} \frac{\sin(kx)}{x}$$

```

La fonction **sum** est souvent capable de calculer la limite de l'expression quand la somme contient une infinité de termes. Ce peut-être également l'inverse comme le montre la comparaison avec l'exemple précédent

```
> Sum(sin(k*x)/x,k=1..n):%<math>=simplify(value(%));  

$$\sum_{k=1}^n \frac{\sin(kx)}{x} = \frac{\sin((n+1)x)\cos(x) - \sin((n+1)x)\sin(x)\cos((n+1)x) + \sin(x)}{2x(\cos(x)-1)}$$

```

Exercice : calculer la limite de cette somme pour pour $x = \pi$ ou $x = \pi/2$ et $n \rightarrow \infty$ et expliquer pourquoi **sum** ne répond pas au calcul de la somme infinie.

Exercice : interpréter ces résultats

```
> Int(Sum((-x)^k/(k!),k=0..infinity),x): %<math>=value(%);
```

$$\int \sum_{k=0}^{\infty} \frac{(-x)^k}{k!} dx = -\frac{1}{e^x}$$

```
> Sum(Int((-x)^k/(k!),x),k=0..infinity): %<math>=expand(value(%));
```

$$\sum_{k=0}^{\infty} \frac{(-x)^k}{k!} dx = -\frac{1}{e^x} + 1$$

MAPLE connaît un grand nombre de fonctions dites "spéciales" qui lui permettent de présenter les résultats sous une forme explicite. On voit apparaître ici la fonction eulérienne incomplète $\Gamma(n, m)$. On notera qu'il peut être commode pour manipuler les expressions et faciliter l'affichage, d'assigner à des noms distincts l'expression de la somme et les résultats.

```
> S:=Sum((x^2)^k/(k-1)!, k=0..n); # On mémorise l'opération à effectuer
s:=value(S); # On mémorise le résultat séparément
```

$$\sum_{k=0}^n \frac{(x^2)^k}{(k-1)!}$$

$$s := \frac{x^2 e^{(x^2)} (n! - \Gamma(n+1) + n \Gamma(n, x^2))}{n!}$$

```
> s:=simplify(s,GAMMA); # On peut manipuler séparément
# l'opération et son résultat.
```

```
s:=convert(s,factorial); # (voir ci dessous)
```

$$s := \frac{x^2 e^{(x^2)} \Gamma(n, x^2)}{\Gamma(n)}$$

$$s := \frac{x^2 e^{(x^2)} \Gamma(n, x^2)}{(n-1)!}$$

```
> S=s; # et afficher le résultat final.
```

$$\sum_{k=0}^n \frac{(x^2)^k}{(k-1)!} = \frac{x^2 e^{(x^2)} \Gamma(n, x^2)}{(n-1)!}$$

Les résultats de **sum** font souvent apparaître les fonctions **digamma** (**Psi(x)** en entrée et $\Psi(x)$ à l'affichage), **polygamma** (**Psi(n,x)** en entrée et $\Psi(n, x)$ à l'affichage)

```
> Sum(1/(k+1)^2, k=0..n):=%=value(%);
```

$$\sum_{k=0}^n \frac{1}{(k+1)^2} = -\Psi(1, n+2) + \frac{1}{6} \pi^2$$

Ces fonctions ont pour définitions

```
> Diff(GAMMA(x),x)/GAMMA(x):=%=value(%);
Diff(ln(GAMMA(x)),x):=%=value(%);
Diff(Psi(x),x$3):=%=value(%);
'Psi(0,x)'=Psi(0,x);
```

$$\frac{\frac{d}{dx} \Gamma(x)}{\Gamma(x)} = \Psi(x)$$

$$\frac{d}{dx} \ln(\Gamma(x)) = \Psi(x)$$

$$\frac{d^3}{dx^3} \Psi(x) = \Psi(3, x)$$

$$\Psi(0, x) = \Psi(x)$$

Ou encore (*nonnegint*: entier non négatif, c'est-à-dire entier positif ou nul)

```
> FunctionAdvisor(Psi(n,z),definition);
```

$$\left[\Psi(n, z) = \frac{d^n}{dz^n} \Psi(z), \text{And}(n::\text{nonnegint}) \right]$$

On rencontre aussi la constante d'Euler γ (voir l'exercice en fin de paragraphe)

```
> Sum(1/k,k=1..n):%:=value(%);
```

$$\sum_{k=1}^n \frac{1}{k} = \Psi(n+1) + \gamma$$

La fonction **convert(...factorial)** (**convert** possède 108 options) permet d'exprimer un résultat sous forme de factorielles

```
> S:=Sum(1/(k*binomial(k+n,k)),k=1..n):s:=value(%):S=s;
```

$$\sum_{k=1}^n \frac{1}{k \text{ binomial}(k+n, k)} = -\frac{2 n+1}{n (n+1) \text{ binomial}(2 n+1, n+1)} + \frac{1}{n}$$

Après quelques manipulations, que nous étudierons au chapitre 13, on obtient

```
> expand(simplify(convert(s,factorial)));  
S=%;
```

$$-\frac{n!^2}{n (2 n)!} + \frac{1}{n}$$

$$\sum_{k=1}^n \frac{1}{k \text{ binomial}(k+n, k)} = -\frac{n!^2}{n (2 n)!} + \frac{1}{n}$$

Apparaît souvent aussi la *fonction Φ de Lerch*

```
> S:=Sum(z^k/(k-1)^2,k=n..infinity):s:=value(S):  
S=s;
```

$$\sum_{k=n}^{\infty} \frac{z^k}{(k-1)^2} = \frac{z^n (1 + (-2 n + 1 + n^2) z \text{ LerchPhi}(z, 2, n))}{(n-1)^2}$$

Ceci tient à la définition de cette fonction dont on trouvera ici deux illustrations (voir aussi **FunctionAdvisor**)

```
> Sum(z^k/(k+alpha)^3,k=0..infinity):%:=value(%);
```

$$\sum_{k=0}^{\infty} \frac{z^k}{(k+\alpha)^3} = \text{LerchPhi}(z, 3, \alpha)$$

```
> Sum(z^k/(k+alpha)^3, k=n..infinity):=%=simplify(value(%));

$$\sum_{k=n}^{\infty} \frac{z^k}{(k+\alpha)^3} = z^n \operatorname{LerchPhi}(z, 3, n+\alpha)$$

```

On pourra aussi voir apparaître la *fonction polylogarithme* liée aux fonctions **ln** et **LerchPhi** par des relations fonctionnelles simples.

```
> Sum(z^k/k^n, k=1..infinity):=%=value(%);
'-polylog(1,1-z)':=%;
'z*LerchPhi(z,n,1)':=%;

$$\sum_{k=1}^{\infty} \frac{z^k}{k^n} = \operatorname{polylog}(n, z)$$


$$\ln(z) = -\operatorname{polylog}(1, -z + 1)$$


$$\operatorname{polylog}(n, z) = z \operatorname{LerchPhi}(z, n, 1)$$

```

MAPLE sait dériver ces fonctions et les intégrer pour des valeurs particulières des arguments

```
> S:=Sum(exp(-k*z)/k, k=1..n):
s:=expand(simplify(value(S))):
S=s;
```

$$\sum_{k=1}^n \frac{e^{-kz}}{k} = -\ln(1 - e^{-z}) - \frac{\operatorname{LerchPhi}(e^{-z}, 1, n)}{e^{(z)n}} + \frac{1}{e^{(z)n} n}$$

De tels résultats permettent de créer des fonctions définies par des séries en leur donnant une écriture explicite. On pourra alors obtenir des dérivées et quelquefois des intégrales

```
> F:=unapply(s,n,z);
Diff('F(2,z)',z)=expand(simplify(diff(F(2,z),z)));
Diff(Sum(exp(-k*z)/k,k=1..2),z):=%=value(%);

$$F := (n, z) \rightarrow -\ln(1 - e^{-z}) - \frac{\operatorname{LerchPhi}(e^{-z}, 1, n)}{e^{(n)z}} + \frac{1}{e^{(n)z} n}$$


$$\frac{d}{dz} F(2, z) = -\frac{1}{(e^{-z})^2} - \frac{1}{e^{-z}}$$


$$\frac{\partial}{\partial z} \left( \sum_{k=1}^2 \frac{e^{-kz}}{k} \right) = -e^{-z} - e^{-2z}$$

```

Dans l'exemple suivant on rencontre la fonction ζ de Riemann (**Zeta** en entrée) définie par une série lorsque $\operatorname{Re}(z) > 1$ (pour l'environnement **assuming** ou la fonction **assume**, voir le chapitre 13 ; écrire $z > 1$ implique z réel). La fonction de Riemann est un cas particulier de la fonction de Φ de Lerch

```
> Sum(1/i^z, i=1..infinity): % = value(%) assuming z>1;
'LerchPhi(1,z,1)'=LerchPhi(1,z,1);
```

$$\sum_{i=1}^{\infty} \frac{1}{i^z} = \zeta(z)$$

$$\text{LerchPhi}(1, z, 1) = \zeta(z)$$

```
> 'Zeta(2)'=Zeta(2), 'Zeta(6)'=Zeta(6);
Sum(1/i^3, i=1..infinity):%:=value(%); # Constante d'Apéry
Zeta(3)=evalf(Zeta(3)); # et son approximation décimale
```

$$\zeta(2) = \frac{1}{6}\pi^2, \zeta(6) = \frac{1}{945}\pi^6$$

$$\sum_{i=1}^{\infty} \frac{1}{i^3} = \zeta(3)$$

$$\zeta(3) = 1.202056903$$

Exercices : Calculez le développement en série de la fonction Γ en 0. Le terme γ (en entrée on écrira **gamma**) qui apparaît dans cette série est la constante d'Euler prédéfinie par MAPLE. Affichez sa valeur numérique avec 30 chiffres significatifs. Calculez la quantité

$$\lim_{n \rightarrow \infty} \left(\left(\sum_{k=1}^n \frac{1}{k} \right) - \ln(n) \right)$$

Pour les séries "récalcitrantes" on pourra essayer d'utiliser les fonctions de la bibliothèque **sumtools**.

Les fonctions **sum** (et **product** ; voir ci-après) admettent aussi **RootOf** comme argument (voir chapitre *Polynômes et Fractions Rationnelles*). On calcule ici la somme des racines d'un polynôme (cette opération, comme celle du produit, ne nécessite pas la recherche explicite les racines. Pourquoi ?)

```
> P:=3*x^5-2*x^4-4;
Sum(alpha, alpha=RootOf(P)):%:=value(%);
P := 3 x^5 - 2 x^4 - 4
```

$$\sum_{\alpha = \text{RootOf}(3 _Z^5 - 2 _Z^4 - 4)} \alpha = \frac{2}{3}$$

La fonction add

La fonction **add** permet dans son utilisation la plus simple de sommer des termes d'une suite (sequence) et de ce point de vue est d'un même usage que **sum**

```
> restart;
add(i^2, i=1..3);
```

14

Mais cette fonction permet en particulier de sommer très simplement les termes, ou une fonction des termes, d'une liste ou d'un ensemble (comparer avec l'écriture de **sum**)

```
> L:=[1,-3,-2,x,-1,-2];
add(i^2, i = L); # i prend successivement les valeurs des opérandes de L
add(i^2, i in L); # une variante de cette écriture (9.5)
```

```

sum(L[i]^2,i=1..nops(L));

$$L := [1, -3, -2, x, -1, -2]$$


$$19 + x^2$$


$$19 + x^2$$


$$19 + x^2$$


```

On notera *deux différences importantes* entre les deux fonctions **sum** at **add**. L'assignation du nom de l'indice de sommation (*i*) est sans conséquence sur le fonctionnement de **add**. On sait que ce n'est pas le cas pour la fonction **sum**

```

> i:=3;
add(i^2,i = L); # L'assignation de 3 à i est sans conséquence
i; # i reste non assigné (ne prend pas la valeur -2)
i := 3

$$19 + x^2$$


$$3$$


```

Par contre **add** ne fait pas de calcul formel, contrairement à **sum**

```

> add(x^(n-1)/n!,n=0..infinity);
Error, unable to execute add
> sum(x^(n-1)/n!,n=0..infinity);

$$\frac{e^x}{x}$$


```

Transformations en Z

MAPLE sait réaliser la *transformation en Z* (unilatérale) utilisée en traitement des signaux échantillonnés $\{f_n\}$, $n \in N$. Cette transformation donne une fonction appelée *transformée en Z* ou *fonction génératrice* définie par

$$Z(\{f_n\}, z) = \sum_{n=0}^{\infty} \frac{f_n}{z^n}$$

La fonction **ztrans** réalise cette transformation. On donne successivement comme arguments à cette fonction, l'expression du signal, la variable de sommation (*n* dans la définition et dans le premier exemple, φ dans le deuxième) et la variable de transformation (*z* dans la définition, θ dans les deux exemples)

```

> ztrans(phi*cos(n*phi),n,theta);
ztrans(phi*cos(n*phi),phi,theta);

$$\frac{\varphi (\theta - \cos(\varphi)) \theta}{\theta^2 - 2 \theta \cos(\varphi) + 1}$$


$$\frac{\theta (\theta^2 \cos(n) - 2 \theta + \cos(n))}{(\theta^2 - 2 \theta \cos(n) + 1)^2}$$


```

Cette fonction admet le traitement de la fonction **charfcn[...](p)** qui vaut 1 en *p* si *p* appartient à la liste définie

par l'utilisateur et 0 ailleurs, mais pas son homologue Dirac qui est une distribution.

```
> C:=x->charfcn[1,Pi,-1,3](x);  
C(1),C(Pi),C(-1),C(2),C(0);  
C := x → charfcn1, π, -1, 3(x)  
1, 1, 1, 0, 0
```

```
> ztrans(charfcn[2](n)*sin(n),n,t);  
sin(2)  
-----  
t2
```

C'est-à-dire

```
> Sum(charfcn[2](n)*sin(n)/t^n,n=0..infinity):%:=value(%);  
∞  
Σ  $\frac{\text{charfcn}_2(n) \sin(n)}{t^n} = \frac{\sin(2)}{t^2}$ 
```

Mais **ztrans** ne traite pas correctement la distribution de Dirac (qui n'est pas une fonction)

```
> ztrans(Dirac(n-2)*sin(n),n,t);  
2 Dirac(0) cos(1) sin(1)  
-----  
t2
```

Elle sait également utiliser la *fonction de Heaviside*

```
> T[Z](Heaviside(n-1)*sin(n),n,t) = ztrans(Heaviside(n-1)*sin(n),n,t);  
T_Z(Heaviside(n - 1) sin(n), n, t) =  $\frac{t \sin(1)}{t^2 - 2 t \cos(1) + 1}$ 
```

On peut également calculer la transformée inverse avec **invztrans**. Elle redonne ici l'expression des termes f_n de la suite dont la transformée en Z est $z/(z-x+1)$

```
> simplify(ztrans((x-1)^n,n,z));  
invztrans(% ,z,n);  

$$\frac{z}{z - x + 1}$$
  

$$(x - 1)^n$$

```

Produits

La fonction product

Les mêmes remarques que pour **sum** concernant la syntaxe des expressions s'appliquent aux produits qui se calculent avec les fonctions **product** ou **Product/value**

```
> restart:  
Product(k,k=1..6):%:=value(%); # = 6!  

$$\prod_{k=1}^6 k = 720$$

```

```

> A:=Product(k/(k+2),k=2..n): a:=value(%): A=a;

$$\prod_{k=2}^n \frac{k}{k+2} = \frac{6 \Gamma(n+1)}{\Gamma(n+3)}$$


> a:=simplify(a):
A=a;


$$\prod_{k=2}^n \frac{k}{k+2} = \frac{6}{(n+1)(n+2)}$$


> P:=Product(1/(2*k+1)^2,k=1..n):p:=value(P):
P=simplify(p);


$$\prod_{k=1}^n \frac{1}{(2k+1)^2} = \frac{4^{(-n-1)} \pi}{\Gamma\left(n+\frac{3}{2}\right)^2}$$


```

La fonction **product** (comme **sum**) admet aussi **RootOf** comme argument (voir chapitre *Polynômes et Fractions Rationnelles*). On calcule ici le produit des racines du polynôme (cette opération ne nécessite pas la recherche explicite les racines. Pourquoi ?)

```

> P:=3*x^5-2*x^4-3*x^2+x-4;
Product(x,x=RootOf(P)):%=value(%);
P := 3 x5 - 2 x4 - 3 x2 + x - 4

```

$$\prod_{x=\text{RootOf}(3 \text{ } \underline{Z}^5 - 2 \text{ } \underline{Z}^4 - 3 \text{ } \underline{Z}^2 + \underline{Z} - 4)} x = \frac{4}{3}$$

La fonction mul

Voir la fonction **add** dans ce chapitre.

```

> L:=[1,-3,2,x,8,4];
mul(sqrt(i),i in L);
L := [1, -3, 2, x, 8, 4]
8 I  $\sqrt{3}$   $\sqrt{x}$ 

```


13 - Simplifications, Manipulations

Fonctions spéciales

Calculs avec unités et Constantes physiques

Nous étudirons dans la première partie de ce chapitre quelques possibilités offertes par MAPLE pour manipuler, transformer ou simplifier des expressions mathématiques. A titre d'introduction commençons par quelques exemples simples qui vont permettre de se faire une idée de la façon dont le problème se pose. Après une série de calculs supposons que nous ayons obtenu une expression E que nous allons chercher à simplifier

> **restart:**

E:=-(cos(2*x)-cos(x)^2)/tan(x)^2;

$$E := -\frac{\cos(2x) - \cos(x)^2}{\tan(x)^2}$$

La fonction **simplify** permet de réaliser cette opération en cherchant une forme équivalente plus compacte

> **Es:=simplify(E);**

$$Es := \cos(x)^2$$

La simplification a utilisé l'identité $\cos(2x) = \cos(x)^2 - \sin(x)^2$. On remarque cependant que MAPLE n'a pas pris en compte l'indétermination de E pour $x = 0$ (forme 0/0 de E) et les deux expressions ne sont pas strictement équivalentes. Moyennant quelques précautions la fonction **simplify** semble très séduisante. Il n'existe cependant pas de recette miracle pour la simplification. Cette fonction, sur laquelle nous reviendrons plus en détails dans ce chapitre, n'est pas universelle. Elle est parfois source d'agacement pour l'utilisateur novice car elle s'entête à ne pas comprendre ses souhaits. Il y a deux raisons à cela:

1) Si les algorithmes utilisés par **simplify** sont puissants, ce ne sont que des algorithmes qui appliquent des règles de simplifications qui ne sont pas nécessairement les plus judicieuses en fonction du contexte. De plus, elles sont volontairement limitées.

2) La notion de "simple" est très relative... Dans les deux exemples qui suivent on comprend le sous-entendu que l'utilisateur a voulu exprimer. Dans le premier cas, il pense que l'expression en fonction de l'arc simple serait la mieux adaptée et inversement dans le second cas. Mais la pensée de l'utilisateur dépend d'un contexte qui échappe totalement à MAPLE dont les algorithmes de simplification ne "voient aucune raison" de changer ces expressions "qu'ils trouvent suffisamment simples" !

> **simplify(cos(2*x));**

$$\cos(2x)$$

> **simplify(2*cos(x)^2-1);**

$$2\cos(x)^2 - 1$$

Il existe cependant de nombreux opérateurs de transformations qui permettent à l'utilisateur de conduire les calculs suivant sa propre intuition. Voici quelques exemples qui ne représentent qu'un petit échantillon des possibilités de MAPLE.

Fonctions expand, combine et convert

1) La fonction **expand** permet de répondre à la première question

```
> expand(cos(2*x));
```

$$2 \cos(x)^2 - 1$$

De même

```
> simplify(tan(a+b));  
expand(tan(a+b));
```

$$\frac{\tan(a) + \tan(b)}{1 - \tan(a) \tan(b)}$$

Ici la fonction **simplify** "pense" que la somme de deux logarithmes est plus "simple" que le logarithme du produit des arguments

```
> simplify(ln(2*x));  
expand(ln(2*x));
```

$$\begin{aligned} &\ln(2) + \ln(x) \\ &\ln(2) + \ln(x) \end{aligned}$$

Pourtant MAPLE ne fait pas toujours ce que l'on souhaiterait *et à juste titre*. Nous verrons pourquoi.

```
> simplify(ln(x*y));  
expand(ln(x*y));
```

$$\begin{aligned} &\ln(xy) \\ &\ln(xy) \end{aligned}$$

Le rôle d'**expand** est d'appliquer des propriétés fonctionnelles permettant de réduire les arguments. Elle permet également de développer les produits de facteurs. Encore un dernier exemple :

```
> ex:=(1-exp(-x))*exp(x*(1+2*ln(x)));  
expand(ex);  
factor(%);
```

$$\begin{aligned} ex := &(1 - e^{-x}) e^{(x(1 + 2 \ln(x)))} \\ &e^x (x^x)^2 - (x^x)^2 \\ &(x^x)^2 (e^x - 1) \end{aligned}$$

2) La fonction **combine** réalise l'opération inverse de celle effectuée par **expand**. Elle est utilisée ici pour répondre au deuxième souhait exprimé en exemple dans l'introduction.

```
> combine(2*cos(x)^2-1);  
cos(2 x)
```

ou encore

```
> combine(2*sin(x)*cos(y));  
sin(x + y) + sin(x - y)
```

Afin de mieux contrôler les opérations effectuées dans une expression on peut donner à cette fonction un deuxième argument prédéfini indiquant le champ d'application souhaité. Les *principales* options (voir l'aide en

ligne) de cette fonction sont : *trig*, *ln*, *exp*, *radical*, *abs*, *power*, *icombine*. Voici trois façons de traiter une même expression. La première force **combine** à n'effectuer que les combinaisons dans le champ de la fonction logarithme avec l'argument *ln*

```
> expr:=ln(cos(x)^2)+ln(2);
combine(expr,ln);
```

$$expr := \ln(\cos(x)^2) + \ln(2)$$

$$\ln(2 \cos(x)^2)$$

puis trigonométrique avec *trig*,

```
> combine(expr,trig);
```

$$\ln\left(\frac{1}{2} \cos(2x) + \frac{1}{2}\right) + \ln(2)$$

puis les deux.

```
> combine(expr);
```

$$\ln(\cos(2x) + 1)$$

Autre exemple :

```
> ex:=x^2*x^(1/2)-x^z/x^(z-1)+z^x*z^(x-1)
      -exp(x-1)*exp(3*x-2)/(ln(x)+ln(2));
ex := x^{(5/2)} - \frac{x^z}{x^{(z-1)}} + z^x z^{(x-1)} - \frac{e^{(x-1)} e^{(3x-2)}}{\ln(2) + \ln(x)}
```

On remarque que le premier terme a été automatiquement réduit (exposants numériques) par MAPLE alors que les suivants sont restés identiques. On obtiendra la simplification relative aux puissances en posant

```
> combine(ex,power);
```

$$x^{(5/2)} - x + z^{(2x-1)} - \frac{e^{(4x-3)}}{\ln(2) + \ln(x)}$$

On remarque que le produit des deux exponentielles a aussi été simplifié. Si on ne souhaite réduire que ce terme on écrira

```
> combine(ex,exp);
```

$$x^{(5/2)} - \frac{x^z}{x^{(z-1)}} + z^x z^{(x-1)} - \frac{e^{(4x-3)}}{\ln(2) + \ln(x)}$$

Pour réduire les logarithmes

```
> combine(ex,ln);
```

$$x^{(5/2)} - \frac{x^z}{x^{(z-1)}} + z^x z^{(x-1)} - \frac{e^{(x-1)} e^{(3x-2)}}{\ln(2x)}$$

On peut aussi appliquer un ensemble d'options. Ici on exclut par cette méthode les recombinaisons des puissances

```
> combine(ex,{exp,ln});
```

$$x^{(5/2)} - \frac{x^z}{x^{(z-1)}} + z^x z^{(x-1)} - \frac{e^{(4x-3)}}{\ln(2x)}$$

Exercice : Exprimer ex en ne simplifiant que le troisième terme (utiliser la notion d'opérandes).

3) La fonction **convert** (qui possède 108 options) permet également de réaliser des transformations complexes reliant les fonctions trigonométriques et hyperboliques aux fonctions logarithme et exponentielle. L'exemple suivant montre comment on peut passer d'une expression trigonométrique à une écriture sous une forme complexe

```
> convert(sin(x+y)+sin(x-y), exp);
```

$$\frac{1}{2} I \left(e^{(I(x+y))} - \frac{1}{e^{(I(x+y))}} \right) - \frac{1}{2} I \left(e^{(I(x-y))} - \frac{1}{e^{(I(x-y))}} \right)$$

que l'on peut écrire sous la forme $A + iB$

```
> evalc(%);
```

$$\begin{aligned} & \frac{1}{2} \sin(x+y) + \frac{\sin(x+y)}{2(\cos(x+y)^2 + \sin(x+y)^2)} + \frac{1}{2} \sin(x-y) + \frac{\sin(x-y)}{2(\cos(x-y)^2 + \sin(x-y)^2)} \\ & + I \left(-\frac{1}{2} \cos(x+y) + \frac{\cos(x+y)}{2(\cos(x+y)^2 + \sin(x+y)^2)} - \frac{1}{2} \cos(x-y) + \frac{\cos(x-y)}{2(\cos(x-y)^2 + \sin(x-y)^2)} \right) \end{aligned}$$

La fonction **simplify** va "détecter" les identités $\cos(a)^2 + \sin(a)^2 = 1$ et permettre de revenir à la notation trigonométrique classique

```
> simplify(%);
```

$$\sin(x+y) + \sin(x-y)$$

Les options de **convert** qui nous concernent ici sont : *exp*, *ln*, *expln*, *exp sincos*, *sincos*, *tan* et *trig*. En voici quelques exemples

```
> arctanh(x+2*y+1):=%=convert(% ,ln);
```

$$\operatorname{arctanh}(x+2y+1) = \frac{1}{2} \ln(x+2y+2) - \frac{1}{2} \ln(-x-2y)$$

```
> ex:=1/(tan(x)^2+1);
```

```
simplify(convert(ex,sincos));
```

$$\begin{aligned} ex := & \frac{1}{\tan(x)^2 + 1} \\ & \cos(x)^2 \end{aligned}$$

Avec l'argument *tan*, cette fonction convertit les fonctions trigonométriques en fonction de la tangente de l'arc moitié

```
> ex:=sin(2*x):
```

```
%=convert(% ,tan);
```

$$\sin(2x) = \frac{2 \tan(x)}{\tan(x)^2 + 1}$$

On peut voir que les résultats peuvent revenir non simplifiés

```
> ex:=sin(2*x)+cot(x)*cos(2*x); ex=convert(ex,tan);
ex=simplify(convert(ex,tan));
```

$$\sin(2x) + \cot(x) \cos(2x) = \frac{2 \tan(x)}{1 + \tan(x)^2} + \frac{1 - \tan(x)^2}{\tan(x)(1 + \tan(x)^2)}$$

$$\sin(2x) + \cot(x) \cos(2x) = \frac{1}{\tan(x)}$$

Il existe bien d'autres options que l'on pourra explorer avec l'aide en ligne (GAMMA désigne la fonction eulérienne Γ) :

```
> A:=GAMMA(k+n)/binomial(k+n,n);
A=simplify(convert(A,factorial));
```

$$\frac{\Gamma(k+n)}{\text{binomial}(k+n, n)} = \frac{n! k!}{k+n}$$

```
> L:=convert(7183457,base,16);
add(L[k]*16^(k-1),k=1..nops(L));
```

$$L := [1, 6, 12, 9, 13, 6]$$

$$7183457$$

```
> convert(7183457,hexadecimal);
```

$$6D9C61$$

Simplification et valeurs complexes

On va donner au cours des paragraphes suivants trois règles importantes que l'on doit toujours avoir à l'esprit quand on fait des manipulations avec MAPLE

Règle 1 : MAPLE travaille toujours en considérant par défaut des expressions et des fonctions à valeurs complexes.

MAPLE est un logiciel prudent et si certaines opérations ne sont pas effectuées, c'est pour de bonnes raisons comme le montre l'exemple suivant

```
> simplify(ln(x*y));
```

$$\ln(xy)$$

En effet la somme de deux logarithmes n'est pas toujours égale au logarithme du produit et ne sachant rien sur la nature de x et de y , aucune opération n'est effectuée.

```
> ln(6)=simplify(ln(6));
combine(ln(-3)+ln(-2))-2*I*Pi='ln(-2)+ln(-3)'-2*I*Pi;
ln(6) = ln(2) + ln(3)
```

$$\ln(6) = \ln(-3) + \ln(-2) - 2I\pi$$

On rappelle les définitions des principales fonctions d'une variable complexe

- $\exp(z) = \exp(\operatorname{Re}[z]) \{ \cos(\operatorname{Im}[z]) + i \sin(\operatorname{Im}[z]) \}$
- $\ln(z) = \ln(|z|) + i \arg[z]$
- $\cos(z) = \cos(\operatorname{Re}[z]) \cosh(\operatorname{Im}[z]) - i \sin(\operatorname{Re}[z]) \sinh(\operatorname{Im}[z])$
- $\sin(z) = \sin(\operatorname{Re}[z]) \cosh(\operatorname{Im}[z]) + i \cos(\operatorname{Re}[z]) \sinh(\operatorname{Im}[z])$
- $x^z = \exp(z \ln(x)) \quad x \in C, z \in C$

Pour la fonction **sqrt** voici la définition qu'en donne MAPLE où apparaît la fonction **signum** qui est définie par $z/\text{abs}(z)$ pour z différent de 0 et $\text{signum}(0)=0$ par défaut (voir l'aide en ligne).

```
> evalc(sqrt(z));
```

$$\frac{1}{2} \sqrt{|z|} (1 + \text{signum}(z)) + \frac{1}{2} I \sqrt{|z|} (1 - \text{signum}(z))$$

Attention: les valeurs de **signum** sont bien sûr -1, 0 ou 1 pour les valeurs réelles de z mais pas pour les valeurs imaginaires ou complexes.

```
> signum(3), signum(-3), signum(0);
signum(I), signum(-3*I), signum(1-I);
1, -1, 0
I, -I,  $\left( \frac{1}{2} - \frac{1}{2} I \right) \sqrt{2}$ 
```

Options de la fonction **simplify**

On peut imposer à la fonction **simplify** de n'appliquer que certaines règles de simplifications en utilisant un deuxième argument tel que *ln*, *trig*, *exp*, *power*, *radical*, *GAMMA* (pour les fonctions eulériennes), etc... (voir `>?simplify`)

Dans l'exemple qui suit apparaît la fonction **csgn** (complex sign) qui détermine le "signe" d'un complexe de la façon suivante

$$\begin{aligned} \text{csgn}(x) = & \begin{cases} +1 & \text{si } \text{Re}(x) > 0 \text{ ou si } \text{Re}(x) = 0 \text{ et } \text{Im}(x) > 0 \\ -1 & \text{si } \text{Re}(x) < 0 \text{ ou si } \text{Re}(x) = 0 \text{ et } \text{Im}(x) < 0 \end{cases} \end{aligned}$$

et $\text{csgn}(0)=0$ par défaut (voir l'aide en ligne)

```
> ex:=1-sin(x)^2+sqrt(x^2+2*x+1);
```

$$ex := 1 - \sin(x)^2 + \sqrt{(1+x)^2}$$

Pour MAPLE, x est par défaut une variable complexe.

```
> simplify(ex);
```

$$\cos(x)^2 + \text{csgn}(1+x) + \text{csgn}(1+x)x$$

```
> simplify(ex,trig);
simplify(ex,radical);
```

$$\cos(x)^2 + \sqrt{(1+x)^2}$$

$$1 - \sin(x)^2 + \text{csgn}(1+x) + \text{csgn}(1+x)x$$

L'option **symbolic** permet de réaliser des simplifications formelles.

```
> xx:=sqrt((1+x)^2);
xx_symb:=simplify(xx,radical,Symbolic);
```

$$\begin{aligned} xx &:= \sqrt{(1+x)^2} \\ xx_symb &:= 1+x \end{aligned}$$

Attention, car elles ne sont bien sûr pas valables pour tout x .

```
> Eval(xx,x=-2+I):%>evalc(value(%));
Eval(xx_symb,x=-2+I):%>evalc(value(%));

$$\left( \sqrt{(1+x)^2} \right) \Big|_{x = -2 + I} = 1 - I$$


$$(1+x) \Big|_{x = -2 + I} = -1 + I$$

```

Les fonctions **is** et **coulditbe**

1) La fonction **is**

Cette fonction permet d'interroger une variable ou une expression sur sa nature mathématique. Elle pose la question "est-elle ?". Les exemples suivants montrent que la logique de MAPLE est à trois états : vrai (true), faux (false) et échec (FAIL). Les réponses de la fonction **is** doivent être comprises de la façon suivante

- **true** : la fonction **is** sait répondre et la proposition est **toujours vraie**.

• **false** : la fonction **is** sait répondre et la proposition **n'est pas nécessairement vraie, mais ne signifie pas qu'elle est toujours fausse**.

• **FAIL** : la fonction **is** ne peut pas donner de réponse soit par manque d'information, soit parce qu'elle ne sait pas résoudre la condition, soit elle fonctionne mal...

La réponse est ici évidente

```
> restart;
is(Pi+ln(2),real);
true
```

S'il ne fait pas de doute que la réponse à la question suivante est toujours fausse, il faut se rappeler qu'en règle générale la réponse doit être comprise dans un **sens relatif**.

```
> is(Pi-4 > 0);
false
```

Ecrire " ≥ 0 " signifie implicitement "*réel* positif ou nul". Or, sans précision, x peut être réel négatif ou complexe et la condition ne sera pas remplie. Mais si x est positif ou nul, la réponse est vraie. La réponse **false** est donc relative.

```
> is(sqrt(x) >= 0);
false
```

Exercice: la fonction **is** est ici prise en défaut et J.H. Lambert se retourne dans sa tombe... Interprétez les questions et réponses

```
> is(Pi in SetOf(realcons) minus SetOf(rational));
FAIL
```

Pourtant Maple connaît le résultat

```
> is(Pi in SetOf(rational));
false
```

et **evalb** répond correctement

```
> evalb(Pi in SetOf(realcons) minus SetOf(rational));
true
```

2) La fonction coulditbe

Elle complète la fonction **is** en posant la question "peut-elle être ?". Il est vrai en effet que l'expression \sqrt{x} peut être réelle positive ou nulle si x est réel positif ou nul et

```
> coulditbe(sqrt(x) >= 0);  
true  
  
> coulditbe(Pi-4 > 0);  
false
```

Dans l'immédiat il est assez difficile de trouver des exemples d'illustration non triviaux, mais nous reviendrons sur l'utilisation de ces fonctions.

Les fonctions assume et additionally

1) La fonction assume

Elle permet d'imposer une contrainte à une variable directement ou à travers une expression. Une variable ainsi contrainte apparaît à l'affichage suivie d'un \sim (tilde, notation un peu malencontreuse qui ne devra pas être confondue avec un signe $-$). Il est possible de changer ce mode d'affichage ("Trailing Tildes") à l'aide du menu *MAPLExx/Preferences...*, onglet *Display*. Deux autres modes sont possibles ; le mode "*Phrase*" ou les \sim sont remplacés par une phrase sous le résultat et le mode "*No Annotation*". Ces options peuvent aussi être introduites avec la commande

```
> interface(showassumed=n);
```

où $n = 0$ vaut pour "*No Annotation*", $n = 1$ pour "*Trailing Tildes*" et $n = 2$ pour "*Phrase*". Cet ordre peut aussi être introduit dans le fichier d'initialisation (voir le chapitre 1, *Commentaires sur l'utilisation de MAPLE, & IV-5*). Quand une feuille de calcul est affichée, le mode dépend de l'état choisi avec *MAPLExx/Preferences...*, onglet *Display*.

On fait ici l'hypothèse que z est réel strictement positif et la proposition $\sqrt{z} > 0$ devient toujours vraie. On rappelle que lorsqu'on écrit $z > 0$ on fait l'hypothèse implicite que z est réel et la simplification $\sqrt{z^2} \rightarrow z$ devient licite.

```
> assume(z > 0); # Cette fonction ne renvoie aucun message  
is(sqrt(z) > 0);  
sqrt(z^2);  
true  
z~
```

Maintenant, la condition imposée ci dessous fait que la proposition n'est pas toujours vérifiée. Le résultat de la simplification est bien conforme à la nouvelle condition. On remarque que:

Règle 2 : Toute nouvelle contrainte faite avec assume sur un objet annule la précédente

```
> assume(z+1>0);  
is(sqrt(z)>0);  
false
```

Comme nous l'avons déjà dit, la réponse *true* de **is** est absolue alors que *false* est relative. Ici $\sqrt{z} > 0$ n'est pas toujours vraie, mais peut l'être. Aussi

```
> coulditbe(sqrt(z)>0);  
true
```

Les options de **assume** sont nombreuses et ne peuvent pas toutes être données ici (voir l'aide en ligne à la rubrique *property*). On en rencontrera d'autres exemples dans la suite.

```
> assume(n,integer);
cos(n*Pi);
(-1)n~
```

Attention

La fonction **assume** n'agit pas sur le *type* d'un objet (voir le chapitre 2)

```
> assume(K>0):
  is(K,positive); # is(K,positive) identique à is(K>0)
type(K,positive),type(K,name);
                           true
                           false, true
```

Pour *libérer une variable de ses hypothèses*, il suffit de la désassigner

```
> n:='n':
cos(n*Pi);
cos(n π)
```

Règle 3 : Toute assignation à une variable lui fait perdre ses contraintes. Suivant les règles habituelles d'évaluation, elle prend les propriétés de l'objet assigné.

MAPLE ne possède pas de mécanisme permettant d'interdire cette opération. Au contraire, ceci permet de rendre la variable libre en la faisant pointer sur son nom (voir ci-dessus).

```
> restart:
assume(z>0);
simplify(sqrt(z^2)); # z réel positif
z~

> assume(x,real);
z:=x+1;
simplify(sqrt(z^2)); # z = x+1 avec x réel => z réel
z := x~ + 1
| x~ + 1 |
```

On peut faire plusieurs hypothèses *avec le même ordre assume*. On prendra alors soin de *répéter le nom de la variable pour chaque hypothèse*.

```
> restart:
rad:=sqrt(1-x^2);
is(rad>0);
rad := √1 - x2
false

> assume(-1<x,x<1); # assume(-1<x<1) serait incorrect
is(rad>0);
```

true

On pourra aussi écrire (Open(-1) signifiant -1 exclu)

```
> assume(x,RealRange(Open(-1),Open(1)));
  is(rad>0);
                                         true

> assume(x,RealRange(-1,Open(1)));
  is(rad>0); # Ce n'est pas toujours vrai
                                         false

> is(rad>=0); # C'est toujours vrai
                                         true
```

On peut également imposer dans un même ordre **assume** des hypothèses sur plusieurs variables. Noter dans l'exemple suivant que l'on peut écrire "m, integer" ou "m::integer". On prendra soin d'écrire \geq et non $=\geq$ (de même \leq et non $=\leq$) sinon MAPLE détecte une erreur de syntaxe. Répétons que si plusieurs propriétés doivent être attachées à un même nom, la syntaxe **assume(nom,prop1,prop2)** est incorrecte et doit être écrite en répétant le nom: **assume(nom,prop1,nom,prop2)**.

```
> cos(m*n*Pi);
  assume(n,integer,m::integer);
  cos(m*n*Pi);
                                         cos(m n π)
                                         (-1)(m~ n~)
```

2) La fonction **additionally**

Cette fonction permet d'*ajouter des contraintes sans détruire celles qui sont déjà définies*.

```
> restart;
  r:=sqrt(1-x^2);
  is(r>0);
                                         r := √1 - x2
                                         false

> assume(x<1);
  is(r>0);

                                         false
```

La condition précédente était insuffisante et on rajoute $-1 < x$

```
> additionally(x>-1); # Maintenant -1<x<1
  is(r>0);

                                         true
```

Attention: répétons-le, les fonctions **assume** et **additionally** n'agissent pas sur le *type* d'un objet (voir le chapitre 2)

```
> assume(K::integer):additionally(K>0):
  type(K,positive),type(K,name);
```

```

is(K,positive); # is(K,positive) identique à is(K>0)
false, true
true

```

Voici encore quelques exemples montrant comment **simplify** prend en compte les contraintes imposées par **assume**. On notera aussi que **assume** ne concerne pas que **simplify** ou **is** et on trouvera au paragraphe "Environnement assuming" quelques exemples d'illustration. On cherche à mettre en facteur les exposants de $x - 1$

```

> x:='x': y:='y': z:='z':
ex:=((x-1)^y)^(z-2);
ex :=  $((x - 1)^y)^{(z - 2)}$ 

```

Sans contrainte rien, ou presque, de ce qu'on souhaite ne se produit
simplify(ex);

$$((x - 1)^y)^z (x - 1)^{(-2)y}$$

L'option *symbolic* permet de faire l'opération mais nous savons qu'elle peut ne pas être valable pour certaines valeurs des variables

```

> simplify(ex,symbolic);
(x - 1)^ $(y(z - 2))$ 

```

En effet cette transformation sera valide pour $x - 1$ positif et y et z réels. Aussi

```

> assume(x-1>0,y,real,z,real);
simplify(ex);
(x~ - 1)^ $(y~(z~ - 2))$ 

```

De même

```

> x:='x':y:='y':z:='z':
simplify(ln(x^y));
ln( $x^y$ )

```

```

> assume(x>0):
'ln(x^y)'=simplify(ln(x^y));
'ln(x*y*z)'=expand(ln(x*y*z));
ln( $x^y$ ) =  $y \ln(x)$ 
ln( $x y z$ ) =  $\ln(x) + \ln(y z)$ 

```

Exercice : quel est le sens de ces réponses ?

```

> is(exp(-t*sin(t)),RealRange(Open(0),infinity));
assume(t,real):
is(exp(-t*sin(t)),RealRange(Open(0),infinity));
false
true

```

Exercice : Pourquoi MAPLE ne dit-il pas que a et b sont respectivement les parties réelle et imaginaire de z ? Que doit-on faire pour cela ?

```

> z:=a+I*b;
  Re(z), Im(z);
  abs(z);

$$z := a + I b$$


$$\Re(a + I b), \Im(a + I b)$$


$$|a + I b|$$


```

Exercice : on définit ici un nom symbolique k qui désigne un entier supérieur ou égal à 2 et strictement inférieur à 6. Les trois ordres **assume** appliquent les mêmes contraintes et sont donnés à titre d'exemple. La fonction **AndProp**($prop_1, prop_2, \dots$) permet d'associer des propriétés qui doivent être satisfaites simultanément (et non contradictoires). Il existe également une fonction **OrProp**. On rappelle que si tous les réels ne sont pas des entiers, tous les entiers sont des réels. Expliquez les réponses de la fonction **is**.

```

> assume(k,integer,k>=2,k<6):
  assume(k::integer,k,RealRange(2,5)): # Autre écriture
  assume(k,AndProp(RealRange(2,Open(6)),integer)); # Autre écriture
  is(k-2,posint);
  is(k-2,nonnegative);

$$FAIL$$


$$true$$


```

Les fonctions **about** et **getassumptions**

La fonction **about** permet d'interroger une variable pour savoir quelles sont les hypothèses qui lui sont attachées.

```

> assume(x>=0,y<0);
  about(x,y);
Originally x, renamed x~:
  is assumed to be: RealRange(0,infinity)

Originally y, renamed y~:
  is assumed to be: RealRange(-infinity,Open(0))

```

```

> assume(x::posint);# entier positif non nul
  about(x);
Originally x, renamed x~:
  is assumed to be: AndProp(integer,RealRange(1,infinity))

```

```

> assume(x::nonnegint);# entier positif ou nul (non négatif)
  about(x+y);
x+y:
  nothing known about this object

```

La fonction **getassumptions** donne l'ensemble des hypothèses attachées aux variables d'une expression. Les variables sans contraintes (ici w) ne sont pas listées.

```

> getassumptions(sin(x)+y^2+w);
{y~::(RealRange(-∞, Open(0))), x~::(AndProp(integer, RealRange(0, ∞)))}

```

Fonctions assume et protect

Le rôle de la fonction **protect**, vue au chapitre 2, est d'interdire toute assignation ou contrainte à une variable. On ne peut pas émettre d'hypothèse sur une variable non contrainte protégée

```
> restart:  
protect(z):  
assume(z>0);  
Error, (in t1) attempting to assign to `z` which is protected
```

Mais attention: on peut changer les hypothèses d'une variable contrainte protégée. La raison est technique et liée à la gestion des variables contraintes (les variables non contrainte et contraintes portent le même nom mais sont en réalité différentes). Ceci peut conduire à des confusions et devrait être changé !

```
> unprotect(z):  
assume(z>0):  
protect(z):
```

La variable z est protégée et strictement positive

```
> about(z);  
Originally z, renamed z~:  
is assumed to be: RealRange(Open(0),infinity)
```

Mais malgré la protection une nouvelle contrainte peut être appliquée !

```
> assume(z<0);  
about(z);  
Originally z, renamed z~:  
is assumed to be: RealRange(-infinity,Open(0))
```

Evaluation des fonctions et variables contraintes

On rappelle que les noms utilisées pour la construction des fonctions (ici x et p) sont muets et ne prennent pas en compte d'éventuelles assignations à des variables qui porteraient le même nom. Les contraintes ne sont donc pas non plus prises en compte: le nom p n'est pas suivi d'un \sim dans l'expression de la définition de f malgré la contrainte appliquée avant.

```
> x:='x';  
assume(p>0):  
f:=x->sqrt(x^2+p);  
x := x  
f := x →  $\sqrt{x^2 + p}$ 
```

C'est seulement au moment de l'appel de f que la contrainte sera prise en compte. Ceci est bien en accord avec la façon dont sont définies des fonctions (voir chapitre 7).

```
> f(x);  
 $\sqrt{x^2 + p}$   
> assume(x,real):  
f(x);  
is(f(x)>0); # p>0
```

$$\sqrt{x^2 + p}$$

true

Par conséquent, la contrainte sur p peut-être changée

```
> assume(p<0):
  f(x);
  is(f(x)>0); # p<0
```

$$\sqrt{x^2 + p}$$

false

Environnement assuming

Cette possibilité permet d'appliquer des contraintes aux variables pour effectuer une opération sans que ces contraintes soient permanentes. Si l'on peut faire l'hypothèse que toutes les variables d'une expression possèdent une même propriété, on peut la formuler en faisant suivre l'expression par **assuming propriété**. *L'hypothèse ne vaut que pour l'opération demandée et n'impose aucune contrainte sur les variables après l'exécution de la commande.* Ici aucune contrainte particulière ne s'applique aux variables et aucune simplification ne peut s'effectuer.

```
> restart:
  e:=sqrt(x^2*y^2)/x/y;
```

$$e := \frac{\sqrt{x^2 y^2}}{x y}$$

```
> simplify(e);
```

$$\frac{\sqrt{x^2 y^2}}{x y}$$

Maintenant on suppose que x et y sont réelles strictement positives.

```
> simplify(e) assuming positive;
```

1

Après l'opération de simplification, aucune contrainte ne reste appliquée aux variables.

```
> about(x,y);
x:
  nothing known about this object

y:
  nothing known about this object
```

Les hypothèses peuvent aussi porter sur tout ou partie des variables. Ici aucune contrainte n'est imposée à y

```
> simplify(ln(k*x/y));
  simplify(ln(k*x/y)) assuming k>0, x>0;
```

$$\ln\left(\frac{k x}{y}\right)$$

$$\ln(k) + \ln(x) + \ln\left(\frac{1}{y}\right)$$

Ou encore (les deux syntaxes sont équivalentes)

```
> simplify(ln(k*x/y)) assuming k::positive, x>0;

$$\ln(k) + \ln(x) + \ln\left(\frac{1}{y}\right)$$

```

Alors que

```
> simplify(ln(k*x/y)) assuming positive;

$$\ln(k) + \ln(x) - \ln(y)$$

```

L'environnement **assuming** (comme la fonction **assume**) ne s'applique pas qu'à la fonction **simplify**

```
> is(sqrt(1/(x^2-1)), positive) assuming x::real, abs(x)>=1;

$$true$$

```

```
> evalc(ln(-z));

$$\ln(|z|) + I\left(\frac{1}{2} + \frac{1}{2}\operatorname{signum}(z)\right)\pi$$

```

```
> evalc(ln(-z)) assuming z>0;

$$\ln(z) + I\pi$$

```

```
> cos(Pi*x) assuming x::integer;

$$(-1)^x$$

```

```
> Limit(exp(-a*x), x=infinity): %value(%);

$$\lim_{x \rightarrow \infty} e^{(-a)x} = \lim_{x \rightarrow \infty} e^{(-a)x}$$

```

```
> Limit(exp(-a*x), x=infinity): %value(%) assuming a::positive;

$$\lim_{x \rightarrow \infty} e^{(-a)x} = 0$$

```

```
> Int(exp(k*x), x=0..infinity): %value(%);

$$\int_0^\infty e^{(kx)} dx = \lim_{x \rightarrow \infty} \left( \frac{e^{(kx)} - 1}{k} \right)$$

```

```
> Int(exp(k*x), x=0..infinity): %value(%) assuming k < 0;

$$\int_0^\infty e^{(kx)} dx = -\frac{1}{k}$$

```

Relations de simplification

La fonction **simplify** ne reconnaît pas toujours "toutes" les "bonnes" règles de simplification.

```
> restart:
```

```

ex:=sin(x)*cos(n*x)^2/(cos(x)^2-1)-sin(x)/(cos(x)^2-1);
simplify(ex);

ex := 
$$\frac{\sin(x) \cos(n x)^2}{\cos(x)^2 - 1} - \frac{\sin(x)}{\cos(x)^2 - 1}$$


$$- \frac{\cos(n x)^2 - 1}{\sin(x)}$$


```

On peut cependant lui donner une ou plusieurs spécifications sous forme d'égalités qu'elle va appliquer. Ces identités sont placées dans une **liste** ou un **ensemble** et ne doivent pas comporter de termes avec dénominateurs.

```

> simplify(ex, {cos(n*x)^2=1-sin(n*x)^2});

$$- \frac{\sin(x) \sin(n x)^2}{\cos(x)^2 - 1}$$


```

Notez que les sous-expressions appartenant à l'expression devront figurer à gauche du signe égal et celles que l'on veut faire apparaître, à droite. Ici MAPLE ne fait rien, sinon une mise en facteur

```

> simplify(ex, [sin(x)^2=1-cos(x)^2]);

$$\frac{\sin(x) (\cos(n x)^2 - 1)}{\cos(x)^2 - 1}$$


```

Attention: **simplify** ne vérifie pas les identités qui lui sont spécifiées... ce qui lui permet d'appliquer consciencieusement une grosse ânerie !

```

> cos(x/3)-cos(x)/3:
% = simplify(%, [3*cos(x/3)=cos(x)]);

$$\cos\left(\frac{1}{3}x\right) - \frac{1}{3}\cos(x) = 0$$


```

Les fonctions factor et normal

Ces fonctions ont déjà été étudiées plus en détail au chapitre 6, *Polynômes et Fractions rationnelles*. On en donne cependant quelques exemples pour montrer qu'elles ne s'appliquent pas seulement aux objets de type **polynom**. Elles s'appliquent plus généralement à des expressions qui ont un **caractère polynomial**. Pour préciser ce que l'on entend par là, considérons l'expression

```

> ex:=2*cos(x)^2*sin(x)^2-4*cos(x)^2*sin(2*x)
    +8*cos(x)^4-sin(x)^2+2*sin(2*x)-4*cos(x)^2;
ex := 
$$2 \cos(x)^2 \sin(x)^2 - 4 \cos(x)^2 \sin(2 x) + 8 \cos(x)^4 - \sin(x)^2 + 2 \sin(2 x) - 4 \cos(x)^2$$


```

L'expression possède un caractère polynomial, bien sûr pas en x , mais en $\sin(x)$, $\cos(x)$ et $\sin(2x)$. Exprimons la, par un changement de variable, sous une forme plus familière en un polynôme en s , t et r

```

> s:='s': t:='t': r:='r':
subs(sin(x)=s,cos(x)=t,sin(2*x)=r,ex);
factor(%);


$$2 t^2 s^2 - 4 t^2 r + 8 t^4 - s^2 + 2 r - 4 t^2$$


$$-(2 t^2 - 1) (-4 t^2 + 2 r - s^2)$$


```

Des expressions comme ex pourront être mises éventuellement en facteurs par la fonction **factor**. On a bien dit éventuellement car le résultat dépend de conditions que nous avons mentionnées au chapitre 6, *Polynômes et Fractions rationnelles*.

```
> exf:=factor(ex);
```

$$exf := (2 \cos(x)^2 - 1)(4 \cos(x)^2 - 2 \sin(2x) + \sin(x)^2)$$

La fonction **normal** a pour but principal de réduire au même dénominateur une somme de fractions

```
> ex:=(x+exp(z))/(x-z) - exp(-z)/(x+z) - x/(x-z);
expr:=normal(ex);
```

$$\begin{aligned} ex &:= \frac{x + e^z}{x - z} - \frac{e^{(-z)}}{x + z} - \frac{x}{x - z} \\ expr &:= \frac{e^z x + e^z z - e^{(-z)} x + e^{(-z)} z}{(x - z)(x + z)} \end{aligned}$$

Exercice : on veut développer le produit de facteurs du dénominateur que l'on sait remarquable. Pourquoi l'opération suivante n'est-elle pas correcte ? Que faudrait-il écrire ?

```
> op(1,expr)/expand(op(2,expr));
```

$$(e^z x + e^z z - e^{(-z)} x + e^{(-z)} z)(x - z)$$

On rappelle que pour répondre simplement à une question du type de celle de l'exercice précédent on utilisera les deux fonctions, **numer** et **denom**, qui renvoient respectivement le numérateur et le dénominateur d'une fraction.

```
> numer(expr)/expand(denom(expr));
```

$$\frac{e^z x + e^z z - e^{(-z)} x + e^{(-z)} z}{x^2 - z^2}$$

Autres exemples de transformations

On donne encore quelques exemples divers qui montrent que les possibilités offertes par MAPLE sont très vastes. Les fonctions de manipulation peuvent s'appliquer à beaucoup de types d'objets. Ici on manipule l'intégrale sans chercher à la calculer

```
> restart;
Int(exp(-x)*sinh(x),x=0..sqrt(z^2));
assume(z,real);
% = expand(simplify(convert(%,exp)));
```

$$\int_0^{\sqrt{z^2}} e^{(-x)} \sinh(x) dx = \frac{1}{2} \int_0^{|z|} 1 - \frac{1}{(e^x)^2} dx$$

On convertit maintenant une expression définie par morceaux à l'aide de la fonction de **Heaviside**

```
> e:=piecewise(x<=-Pi,0,x<=Pi,cos(x),0);
convert(e,Heaviside);
```

$$e := \begin{cases} 0 & x \leq -\pi \\ \cos(x) & x \leq \pi \\ 0 & \text{otherwise} \end{cases}$$

$$\cos(x) \operatorname{Heaviside}(\pi + x) - \cos(x) \operatorname{Heaviside}(-\pi + x)$$

On simplifie l'écriture de l'addition de deux sommes

```
> Sum(x^(-n)/n!, n=1..m)+Sum(x^(-n)/n!, n=m+1..infinity):
% = combine(%);
```

$$\left(\sum_{n=1}^m \frac{x^{(-n)}}{n!} \right) + \left(\sum_{n=m+1}^{\infty} \frac{x^{(-n)}}{n!} \right) = \sum_{n=1}^{\infty} \frac{x^{(-n)}}{n!}$$

L'application sur des objets comme les tableaux peut nécessiter l'utilisation explicite de la fonction **map**

```
> A:=convert(matrix(2,2,[seq((exp(i*I*x)+exp(-i*I*x))/2,i=1..4)]),Matrix);
A_trig:=convert(A,trig);
expand(A_trig), map(expand,A_trig);
```

$$A := \begin{bmatrix} \frac{1}{2} e^{(Ix)} + \frac{1}{2} e^{(-Ix)} & \frac{1}{2} e^{(2Ix)} + \frac{1}{2} e^{(-2Ix)} \\ \frac{1}{2} e^{(3Ix)} + \frac{1}{2} e^{(-3Ix)} & \frac{1}{2} e^{(4Ix)} + \frac{1}{2} e^{(-4Ix)} \end{bmatrix}$$

$$A_{\text{trig}} := \begin{bmatrix} \cos(x) & \cos(2x) \\ \cos(3x) & \cos(4x) \end{bmatrix}$$

$$\begin{bmatrix} \cos(x) & \cos(2x) \\ \cos(3x) & \cos(4x) \end{bmatrix}, \begin{bmatrix} \cos(x) & 2 \cos(x)^2 - 1 \\ 4 \cos(x)^3 - 3 \cos(x) & 8 \cos(x)^4 - 8 \cos(x)^2 + 1 \end{bmatrix}$$

Maple et les fonctions spéciales

MAPLE connaît beaucoup de fonctions dites "spéciales" et sait les manipuler. Ces fonctions n'ont rien de "spécial". Elles sont appelées ainsi parce qu'elles sont simplement d'un usage moins fréquent que **sin** ou **exp**... MAPLE peut exprimer des récurrences pour des valeurs particulières ou donner les relations (si elles existent et s'il les connaît) entre différents types de fonctions

```
> expand(BesselJ(2,x));
expand(KelvinBei(2,x));
convert(KelvinBei(2,x),BesselJ);
```

$$\begin{aligned} & \frac{2 \operatorname{BesselJ}(1, x)}{x} - \operatorname{BesselJ}(0, x) \\ & - \frac{\sqrt{2} \operatorname{KelvinBei}(1, x)}{x} - \frac{\sqrt{2} \operatorname{KelvinBer}(1, x)}{x} - \operatorname{KelvinBei}(0, x) \\ & \frac{1}{2} I \left(\operatorname{BesselJ}\left(2, \left(-\frac{1}{2} - \frac{1}{2} I\right) x \sqrt{2}\right) - \operatorname{BesselJ}\left(2, \left(-\frac{1}{2} + \frac{1}{2} I\right) x \sqrt{2}\right) \right) \end{aligned}$$

La fonction **FunctionAdvisor** est une sorte d'encyclopédie recensant les propriétés de 131 fonctions avec pour chacune 24 mots clés tels que *definition*, *differentiation_rule*, *integral_form*, etc. (voir l'aide en ligne)

> **FunctionAdvisor(BesselJ(n,x),definition);**

Warning: when the "hypergeom" form of a function is requested, only the function name (e.g., BesselJ(n,x)) - is expected. Extra arguments are being ignored.

$$\left[\text{BesselJ}(n, x) = \frac{x^n \text{hypergeom}\left(\left[\right], [1+n], -\frac{1}{4}x^2\right)}{\Gamma(1+n) 2^n}, \text{with no restrictions on } (n, x) \right]$$

La mention "with no restrictions on (n,x)" signifie que la définition est valable pour des arguments complexes quelconques comme le montre un exemple numérique

> **BesselJ(1+2*I,3+2*I):**

```
%=evalf(%);
(3+2*I)^(1+2*I)*hypergeom([], [2+2*I], -(3+2*I)^2/4)/GAMMA(2+2*I)/2^(1+2*I)
%:
```

$$\text{BesselJ}(1 + 2I, 3 + 2I) = 0.4189701155 + 0.01854549359I$$

$$\frac{(3 + 2I)^{(1 + 2I)} \text{hypergeom}\left(\left[\right], [2 + 2I], -\frac{5}{4} - 3I\right)}{\Gamma(2 + 2I) 2^{(1 + 2I)}} = 0.4189701157 + 0.01854549349I$$

Les règles de dérivation sont données pour un argument sous forme de fonction (dérivation d'une fonction composée)

> **FunctionAdvisor(BesselJ,differentiation_rule);**

Warning: when differentiation rule information is required, only one argument - the function name - is expected. Extra arguments are being ignored.

$$\frac{\partial}{\partial z} \text{BesselJ}(a, f(z)) = \left(-\text{BesselJ}(a + 1, f(z)) + \frac{a \text{BesselJ}(a, f(z))}{f'(z)} \right) \left(\frac{d}{dz} f(z) \right)$$

Exercice : Que cherche-t-on à faire ? A quoi sert "[1][1]"

> **FunctionAdvisor(BesselJ(n,x),integral_form)[1][1];**

Warning: when the "integral" form of a function is requested, only the function name (e.g., BesselJ(n,x)) - is expected. Extra arguments are being ignored.

$$\text{BesselJ}(n, x) = \int_{-\pi}^{\pi} \frac{e^{(-In_kl + Ix \sin(kl))}}{2\pi} d_kl$$

L'option **specialize** (placée en premier argument) permet d'obtenir, quand c'est possible, des relations entre les fonctions spécifiées

> **FunctionAdvisor(specialize,sin,BesselJ);**

$$\left[\sin(z) = \frac{1}{2} \sqrt{z} \sqrt{\pi} \sqrt{2} \text{BesselJ}\left(\frac{1}{2}, z\right), \text{with no restrictions on } (z) \right]$$

Evidemment de telles relations n'existent pas toujours

```
> FunctionAdvisor(specialize,BesselJ,GAMMA);
```

GAMMA form for BesselJ is unknown to the system

On retrouve ces résultats dans l'utilisation de la fonction **convert**

```
> sin(x)=convert(sin(x),BesselJ);
```

$$\sin(x) = \frac{1}{2} \sqrt{x} \sqrt{\pi} \sqrt{2} \text{BesselJ}\left(\frac{1}{2}, x\right)$$

```
> cos(x)=convert(cos(x),hypergeom,include='all');
```

$$\cos(x) = \text{hypergeom}\left(\left[\right], \left[\frac{1}{2}\right], -\frac{1}{4}x^2\right)$$

ou des fonctions **simplify**, **combine**, etc.

```
> polylog(2,x)+polylog(2,1/x):
```

```
%=combine(% ,polylog) assuming x>-1,x<1;
```

$$\text{polylog}(2, x) + \text{polylog}\left(2, \frac{1}{x}\right) = -\frac{1}{6}\pi^2 - \frac{1}{2}\ln\left(-\frac{1}{x}\right)^2$$

```
> sqrt(Pi)*pochhammer(1/2,x)/pochhammer(x,1/2):
```

```
%=simplify(%);
```

$$\frac{\sqrt{\pi} \text{pochhammer}\left(\frac{1}{2}, x\right)}{\text{pochhammer}\left(x, \frac{1}{2}\right)} = \Gamma(x)$$

```
> hypergeom([-1,2],[1/n],x):=%=convert(% ,StandardFunctions);
```

```
hypergeom([-1,2],[1/2],x^2):=%=convert(% ,StandardFunctions);
```

```
-rhs(%)=sort(expand(-rhs(%)));
```

$$\text{hypergeom}\left([-1, 2], \left[\frac{1}{n}\right], x\right) = n \text{JacobiP}\left(1, \frac{1}{n} - 1, -\frac{1}{n} + 1, 1 - 2x\right)$$

$$\text{hypergeom}\left([-1, 2], \left[\frac{1}{2}\right], x^2\right) = -\text{ChebyshevU}(2, \sqrt{x^2})$$

$$\text{ChebyshevU}(2, \sqrt{x^2}) = 4x^2 - 1$$

De même avec la *fonction de Meijer*

```
> MeijerG([[1],[1/2]],[[0],[2]],x^2):=%=convert(% ,StandardFunctions);
```

$$\text{MeijerG}\left(\left[[1], \left[\frac{1}{2} \right] \right], [[2], [0]], x^2\right) = \frac{1}{\sqrt{\pi}} + \frac{(12x^2 - 8)\sqrt{1-x^2}}{8\sqrt{\pi}(x^2 - 1)^2}$$

La fonction **convert** permet encore de nombreuses autres possibilités comme ici de formuler une relation de récurrence entre des polynomes de Legendre dans le sens d'un abaissement du premier argument n (désigné en général par "a")

```
> n*LegendreP(n,x):=%=convert(% ,LegendreP,"lower a");
n LegendreP(n,x)=(-x+2xn)LegendreP(n-1,x)+LegendreP(n-2,x)(-n+1)
```

On ne saurait trop conseiller au lecteur d'explorer l'aide en ligne de la fonction **convert** et notamment

```
> ?convert,to_special_function
```

Utilisation d'unités physiques

MAPLE permet de travailler avec des unités physiques. On peut par exemple pratiquer des conversions d'unités (on trouvera aussi un convertisseur d'unités plus convivial dans le menu *Tools/Assistants/Unit Converter...*). Que vaut 1 °C en degrés Fahrenheit?

```
> restart;
convert(1.0,units,degC,degF);
1.800000000
```

De même un "pied" fait en centimètres

Attention: les noms des unités (tels que ft ou cm) ne doivent pas être assignés (ou alors être écrits 'ft' et 'cm')

```
> convert(1.0,units,ft,cm);
30.48000000
```

En remplaçant le mot clé **units** par **temperature** on obtient une conversion d'échelle (10 °C font 50 °F)

```
> convert(10.,temperature,degC,degF);# = 32 °C + 1.8*10 °C
50.0000000
```

Inversement 451 °F font en °C

```
> convert(451.,temperature,degF,degC);
232.7777778
```

Le 0 absolu sur l'échelle Celsius vaut

```
> convert(0.,temperature,kelvin,degC);
-273.1500000
```

Combien de mètres font 3 pieds, 6.3 pouces et 2.1 cm ou combien d'années lumière font 3.2 parsecs

```
> convert(3*ft+6.3*inches+2.1*cm,metric);
convert(3.2,units,parsecs,ly); # ly = "light year"
1.095420000 m
10.43700408
```

Impossible de donner un aperçu complet de la librairie **Units** et on se reportera à l'aide en ligne

```
> ?Units/Index
```

Il y en a pour tout le monde... même pour les photométristes: combien font 1 candela/cm² en apostilb ???
 $(=10^4 * \pi)$

```
> convert(1.,units,cd/cm^2,apostilb);  
31415.92654
```

On notera deux possibilités pour travailler dans un environnement de calculs avec unités.

1) Avec Units[Natural]

```
> restart;  
with(Units[Natural]):  
Warning, the assigned names factor and polar now have a global binding  
Warning, these protected names have been redefined and unprotected: `*`, `+`, `-`, `/`, `<`, `<=`, `>`, `=`, Im, Re, `^`, abs, add, arccos, arccosh, arccot, arccoth, arccsc, arccsch, arcsec, arcsech, arcsin, arcsinh, arctan, arctanh, argument, ceil, collect, combine, conjugate, convert, cos, cosh, cot, coth, csc, csch, csgn, diff, eval, evalc, evalr, exp, expand, floor, frac, int, ln, log, log10, max, min, mul, normal, root, round, sec, sech, seq, shake, signum, simplify, sin, sinh, sqrt, surd, tan, tanh, trunc, type, verify
```

toute expression est manipulée, calculée et convertie dans le système d'unité par défaut. Quelle est la surface en m² quand on ajoute 3 surfaces, une de 3 pieds², etc. ?

Attention: les noms des unités (tels que ft, inches ou mm) ne doivent pas être assignés (ou alors être écrits 'ft', 'inches' et 'mm')

```
> 3*ft^2+6.3*inches^2+1078.2*mm^2;  
0.2838518280 [m2]
```

On veut maintenant calculer le côté du carré de même surface. On comprend pourquoi l'appel à **Units** redéfinit les fonctions standards pour pouvoir accepter et traiter les arguments avec unités

```
> sqrt(%);  
0.5327774657 [m]
```

Soit en pieds...

```
> convert(% ,units,ft);  
1.747957565 [ft]
```

Quelle est la force (en Newton) exercée sur une masse de 3.5 tonnes et produisant une accélération de 2.7 hectomètres par minute² ?

```
> 3.5*t*2.7*hm/min^2;  
262.5000000 [N]
```

On dit, en première approximation, que la pression atmosphérique diminue exponentiellement avec l'altitude. Si la pression au sol est de 1.02 atm ("atmosphère") on devrait avoir à 2 km d'altitude

```
> P:=1.02*atm*exp(-2*km);  
Error, (in Units:-Standard:-exp) the function ``Units:-Standard:-exp`` cannot handle  
the unit km with dimension length
```

Evidemment, ça ne peut pas fonctionner car l'argument d'une exponentielle (et de toute fonction transcendante) ne peut contenir des unités (le résultat dépendrait entre autre de l'unité choisie: 2 km ou 2000 m ?!). Il nous

manque l'échelle de hauteur, H_0 , de l'atmosphère terrestre (c'est la hauteur moyenne de l'atmosphère si tout le gaz était à la pression normale du sol). On exprime ici la pression calculée en *millimètres de mercure*...

```
> H[0]:=8.5*km;
P:=convert(1.02*atm*exp(-2.0*km/H[0]),units,millimeter_mercury);
H0 := 8.5 [km]
P := 612.6702990 [mmHg]
```

Le système d'unité par défaut est SI, mais on peut en choisir d'autres

```
> Units[UseSystem](CGS);
3.5*t^2.7*hm/min^2;
1.0*g*1.0*cm/s^2;
2.625000000 107 [dyn]
1.00 [dyn]
```

2) Avec Units[Standard]

```
> restart;
with(Units[Standard]);
Warning, the assigned names factor and polar now have a global binding
Warning, these protected names have been redefined and unprotected: `*`, `+`, `-`,
`/`, `<`, `<=`, `>`, `=`, Im, Re, `^`, abs, add, arccos, arccosh, arccot, arccoth,
arccsc, arccsch, arcsec, arcsech, arcsin, arcsinh, arctan, arctanh, argument, ceil,
collect, combine, conjugate, convert, cos, cosh, cot, coth, csc, csch, csgn, diff,
eval, evalc, evalr, exp, expand, floor, frac, int, ln, log, log10, max, min, mul,
normal, root, round, sec, sech, seq, shake, signum, simplify, sin, sinh, sqrt, surd,
tan, tanh, trunc, type, verify
```

on doit utiliser une écriture plus lourde. Par contre les calculs sont plus rapides

Attention: les noms des unités (tels que ft, inches ou mm) ne doivent pas être assignés (ou alors être écrits 'ft', 'inches' et 'mm')

```
> 3*Unit(ft^2)+6.3*Unit(inches^2)+1.2*Unit(mm^2);
0.2827748280 [m2]

> sqrt(%);
0.5317657642 [m]

> convert(% ,units,ft);
1.744638334 [ft]

> 3.5*Unit(t)*2.7*Unit(hm/minute^2);
262.5000000 [N]

> convert(% ,units,dyn);
2.625000000 107 [dyn]
```

La librairie ScientificConstants

Maple contient un librairie très importante de constantes physiques dont nous ne donnerons que très peu d'exemples. On se reportera à l'aide en ligne

```
> restart;
with(Units[Standard]):
```

Warning, the assigned names factor and polar now have a global binding

Warning, these protected names have been redefined and unprotected: `*`, `+`, `-`, `/`, `<`, `<=`, `<>`, `=`, Im, Re, `^`, abs, add, arccos, arccosh, arccot, arccoth, arccsc, arccsch, arcsec, arcsech, arcsin, arcsinh, arctan, arctanh, argument, ceil, collect, combine, conjugate, convert, cos, cosh, cot, coth, csc, csch, csgn, diff, eval, evalc, evalr, exp, expand, floor, frac, int, ln, log, log10, max, min, mul, normal, root, round, sec, sech, seq, shake, signum, simplify, sin, sinh, sqrt, surd, tan, tanh, trunc, type, verify

> ?ScientificConstants

Elle contient essentiellement deux parties. L'une concerne les constantes physiques

> ?ScientificConstants,PhysicalConstants

l'autre les constantes relatives aux éléments chimiques et à la table de Mendeleiev (poids atomiques, densités, points d'ébullition, isotopes, stabilités isotopiques, etc.)

> ?ScientificConstants,elements

> ?ScientificConstants,Periodic

On trouvera également une feuille de calcul d'exemples d'applications

> ?examples/SCApps;

On charge le module ScientificConstants

> with(ScientificConstants);

[AddConstant, AddElement, AddProperty, Constant, Element, GetConstant, GetConstants, GetElement, GetElements, GetError, GetIsotopes, GetProperties, GetProperty, GetUnit, GetValue, HasConstant, HasElement, HasProperty, ModifyConstant, ModifyElement]

Nous sommes actuellement dans le *Système International*

> Units:-UsingSystem();

SI

La fonction GetConstant donne les différentes informations associées à un nom de constante physique

> GetConstant('h');

Planck_constant, symbol = h, value = 6.62606876 10⁻³⁴, uncertainty = 5.2 10⁻⁴¹, units = Js

Encore faut-il que le nom soit connu; on consultera l'aide en ligne

> ?ScientificConstants,PhysicalConstants

> HasConstant('h'), HasConstant('Planck_constant');

true, true

La constante $\frac{h}{2\pi}$ est aussi connue

> HasConstant('hbar');

true

mais elle n'a pas de valeur numérique ni d'unité associées car c'est une constante dérivée de *h*

```
> GetConstant('hbar');
```

$$\text{Planck_constant_over_2pi, symbol} = hbar, \text{derive} = \frac{h}{2\pi}$$

La vitesse de la lumière dans ce système s'obtient avec **Constant** (**units** indique qu'il faut préciser les unités correspondantes dans le système courant)

```
> c:=Constant('c',units);
```

$$c := \text{Constant}_{SI}(c) \left[\frac{m}{s} \right]$$

En pratique, ce qui intéresse, c'est la valeur numérique que l'on obtient avec **evalf**

```
> c:=evalf(Constant('c',units));
```

$$c := 2.99792458 \cdot 10^8 \left[\frac{m}{s} \right]$$

On veut maintenant la masse de l'électron dans le système CGS

```
> m[e]:=evalf(Constant('m[e]',system=CGS,units));
```

$$m_e := 9.109381882 \cdot 10^{-28} [g]$$

L'énergie au repos d'un électron sera calculée dans le système d'unité courant malgré la disparité des unités des constantes

Attention: la conversion des unités en Joules ne se fait que parce que nous sommes dans un environnement de calcul avec unités (**with(Units[...])**) effectué plus haut).

```
> E[e]:=m[e]*c^2;
```

$$E_e := 8.187104141 \cdot 10^{-14} [J]$$

C'est-à-dire en "Cheval vapeur * minute" ... ;-)

```
> convert(E[e],units,hp*min);
```

$$1.829847917 \cdot 10^{-18} [HP \text{ min}]$$

Quelle est la masse d'une molécule de Méthane dans le système CGS ?

```
> evalf[4](convert(Element(C,atomicweight,units)+
```

```
4*Element(H,atomicweight,units),system,CGS));
```

$$2.665 \cdot 10^{-23} [g]$$

Quelles sont les températures de fusion et d'ébullition du Mercure en degrés Celsius ?

```
> evalf[6](convert(Element(Hg,meltingpoint,units),
```

```
temperature,Celsius)),
```

```
evalf[6](convert(Element(Hg,boilingpoint,units),
```

```
temperature,Celsius));
```

$$-38.830 [\text{degC}], 356.730 [\text{degC}]$$

Quel est la température de sublimation du Carbone ?

```
> evalf(Element(C,boilingpoint(sp),units));
```

$$3915. [K]$$

Quelle est la demi-vie radioactive du Tritium ? La librairie de constantes nous donne à ce sujet, les informations suivantes

```
> GetProperties('isotopic');
abundance, atomicmass, betadecayenergy, bindingenergy, electronaffinityisotopic, halflife, massexcess
```

Les isotopes connus de l'hydrogène sont

```
> GetIsotopes(element=H);
H1, H2, H3, H4, H5, H6
```

La durée de demi-vie du Tritium exprimée en années est donc

```
> evalf[4](convert(Element[H[3]],halflife,units,year));
12.32 [yr]
```

On peut ajouter à la base de donnée des valeurs, comme ici, les masses du Soleil et de la Terre

```
> AddConstant(Masse_du_Soleil,symbol=M[Soleil],value=1.9891e30,units=kg);
AddConstant(Masse_de_la_Terre,symbol=M[Terre],value=5.9742e24,units=kg);
```

On vérifie que G est bien, dans la base de données de MAPLE, le symbole associé à la constante de gravitation

```
> GetConstant('M[Soleil]');
GetConstant('G');
Masse_du_Soleil, symbol = MSoleil, value = 1.9891 1030, uncertainty = undefined, units = kg
```

```
Newtonian_constant_of_gravitation, symbol = G, value = 6.673 10-11, uncertainty = 1.0 10-13, units =  $\frac{m^3}{kg \cdot s^2}$ 
```

On récupère les valeurs numériques et on leur donne un nom

```
> G:=evalf(Constant('G',units)):
M[S]:=evalf(Constant('M[Soleil]',units)):
M[T]:=evalf(Constant('M[Terre]',units)):
```

La force d'attraction entre la Terre et le Soleil à une unité astronomique de distance est

```
> F:=evalf[4](G*M[S]*M[T]/(1*Unit(UA))^2);
F := 3.542 1022 [N]
```

Conclusions

On voit par ces divers exemples que, malgré la puissance de calcul offerte, il est nécessaire pour obtenir des résultats, de posséder une bonne intuition des calculs alliée à une certaine technique qui ne peut s'acquérir qu'avec l'expérience. MAPLE est un logiciel de manipulations symboliques, pas un mathématicien ! Les manipulations sont des opérations délicates qui ne peuvent être confiées aux seuls algorithmes de MAPLE. Elles doivent être effectuées avec prudence et utiliser ce logiciel comme une machine presse-bouton peut conduire à des anomalies. Si le calcul symbolique n'est pas né avec MAPLE, il est en perpétuelle évolution et fait l'objet de recherches très actives dont MAPLE bénéficie à chaque nouvelle version.

14 - Equations ordinaires

MAPLE permet de résoudre divers types d'équations ordinaires de façon exacte ou numérique (pour les équations différentielles, voir chapitre 15). Le cas des *systèmes d'équations linéaires* sera brièvement présenté à la fin de ce chapitre.

Résolutions exactes

Profitons tout d'abord d'un premier exemple pour rappeler qu'une équation, comme tout objet de MAPLE, peut être assignée à un nom.

```
> restart:  
eq:= 2*x+Pi = 1-x:  
eq;  
2 x + π = 1 - x
```

Le type d'une équation est tout simplement *equation* ou `=».

```
> whattype(eq),      op(0,eq);  
type(eq,equation),  type(eq,`=`);  
=, =  
true, true
```

Le membre de gauche et le membre de droite peuvent être extraits respectivement avec les fonctions **lhs** et **rhs** (left and right hand side). Une équation possède deux opérandes de premier niveau qui sont les images des deux fonctions **lhs** et **rhs**

```
> lhs(eq) , op(1,eq);  
rhs(eq) , op(2,eq);  
2 x + π, 2 x + π  
1 - x, 1 - x
```

Exercice : que cherche-t-on avec cette commande ? Expliquez

```
> op(indets(eq,name) minus indets(eq,constant));
```

ou encore (notez le N majuscule de **Not**; voir l'aide en ligne de **type**)

```
> op(indets(eq,name(Not(constant))));
```

L'expression suivante montre comment résoudre cette équation, certes triviale (!), à l'aide de la fonction **solve** que nous avons déjà rencontrée à propos de la recherche des expressions des racines des polynômes. Ici on n'a donné pour seul argument que l'équation. On notera que **solve** trouve tout seul l'unique indéterminée *x* **mais ne lui assigne pas la valeur de la solution**

```
> solve(eq);  
x;          # Reste non assignée  
- 1/3 π + 1/3  
x
```

On peut vérifier que la solution est correcte en substituant à *x* le résultat de **solve(eq)** dans l'équation

```
> evalb(subs(x=%%,eq));
true
```

Lorsque l'équation présente *plusieurs indéterminées*, il sera nécessaire de préciser par un deuxième argument la variable par rapport à laquelle on veut résoudre

```
> eq:=a*x+2=2*x; eq;
solve(eq,x); # x est l'inconnue
```

$$ax + 2 = 2x$$

$$-\frac{2}{a - 2}$$

```
> solve(eq,a); # a est l'inconnue
```

$$\frac{2(-1+x)}{x}$$

On obtient sinon une résolution implicite et MAPLE choisit l'inconnue

```
> solve(eq);
\left\{ x = x, a = \frac{2(-1+x)}{x} \right\}
```

Comme précédemment les solutions n'ont été assignées ni à x ni à a

```
> x , a;
x, a
```

On pourra donner un nom à une solution en écrivant simplement (voir aussi la fonction **assign** dans ce chapitre au § *Assignation des solutions d'un système d'équations*)

```
> x_sol:=solve(eq,x);
x_sol := -\frac{2}{a - 2}
```

Lorsque MAPLE trouve plusieurs solutions, il les écrit sous la forme d'une suite (sequence)

```
> S:=solve(x^3-2*x^2-x+2=0,x);
S := -1, 1, 2
```

On pourra toujours extraire l'une des solutions en écrivant

```
> S[2];
1
```

Lorsque l'équation s'écrit "*expression* = 0" on peut se dispenser d'écrire " = 0" et MAPLE le suppose implicitement. Par ailleurs il est très simple de transformer le résultat en une liste ou un ensemble. Ainsi par exemple, pour que les solutions soient présentées sous forme d'une liste ou d'un ensemble, on écrira, avec ou non une affectation à un nom

```
> S:=[solve(4*x^3+3*x^2+x^4-4*x-4)];
{solve(4*x^3+3*x^2+x^4-4*x-4)};
S := [-1, 1, -2, -2]
{-2, -1, 1}
```

Attention: avec un ensemble une racine multiple apparaîtra, conformément à la définition des ensembles,

comme une racine unique.

Exercice: expliquez le sens de ces réponses

```
> eq:=x^2+a^2=(x+a)^2-2*a*x; eq;
solve(eq,x);
```

$$x^2 + a^2 = (x + a)^2 - 2 a x$$

```
> solve(eq,a);
```

a

Solutions, Absence de solution, Non résolution, Résolutions partielles

Il faut tout d'abord rappeler que les outils standards de l'analyse mathématique sont impuissants à résoudre symboliquement la très grande majorité des équations ordinaires que l'on peut rencontrer (ceci est rarement mentionné dans les livres de cours et ceux de travaux dirigés ne rendent pas compte de cette situation car les exemples sont toujours bien choisis pour pouvoir être résolus). Pour être plus précis, les solutions de beaucoup d'équations ne sont généralement pas exprimables en termes de combinaisons de constantes et de valeurs de fonctions élémentaires ou spéciales (telles que $1 - \sqrt{2}$, $\cos(e)$ e^π , $1 - \cos(\ln(\sqrt{\pi}))$ ou plus compliquées). On ne s'étonnera donc pas de voir que **solve** ne peut pas résoudre exactement bon nombre d'équations, même d'aspect "débonnaire", et dont on sait, par des procédés analytiques, graphiques et/ou numériques, qu'elles ont une ou plusieurs solutions. Pour prendre un exemple simple, on ne sait pas (on ne peut pas) exprimer les solutions de l'équation $\cos(x) = x$ en terme de symboles ou de fonctions connus. Elles ne peuvent être obtenues que numériquement.

La réponse de **solve** peut être symbolique et d'un intérêt assez limité (du moins pour la recherche d'une solution explicite). Une **représentation graphique** montre que l'équation suivante possède au moins deux solutions réelles que **solve** n'exprime que de façon symbolique. On rappelle que **RootOf** signifie "Racine(s) De". Cette réponse nous dit malgré tout que les solutions peuvent s'exprimer comme des exponentielles de nombres (réels ou complexes).

```
> eq:=ln(x)=exp(x)-3; eq;
S:=solve(eq,x);
```

$$\begin{aligned} \ln(x) &= e^x - 3 \\ S := e^{\text{RootOf}}\left(-Z - e^{(e^{-Z})+3} \right) \end{aligned}$$

Quand MAPLE ne peut rien répondre, il renvoie tout simplement **NULL**.

```
> cos(sqrt(x))-x^2=0;
S:=solve(% ,x); # 2 solutions dans [-2,+1] (cf. fin du chap. 9)
cos(\sqrt{x}) - x^2 = 0
S :=
```

Cependant, afin de prévenir l'utilisateur, la fonction **solve** rend la variable globale **_SolutionsMayBeLost** vraie ("Des solutions peuvent être perdues"). **Pour être informé il est indispensable d'afficher cette variable.** Nous verrons plus loin comment trouver numériquement les solutions des équations.

```
> _SolutionsMayBeLost;
true
```

Attention: **solve** renvoie aussi **NULL** quand elle pourrait résoudre certains types d'équations, mais "sait"

qu'elles n'ont pas de solution. La variable `_SolutionsMayBeLost` reste alors non assignée. Sans l'affichage de cette variable les réponses de `solve` dans les deux cas sont donc identiques.

```
> 2*x=2*x+1;  S:=solve(%,x);
   _SolutionsMayBeLost;
```

$$2x = 2x + 1$$

$$S :=$$

`_SolutionsMayBeLost`

Attention: comme la réponse aux solutions de ces équations est **NULL**, aucun affichage n'apparaît si on n'assigne pas les solutions à des noms. Ceci peut être la source d'un malentendu.

```
> solve(cos(sqrt(x))-x^2=0,x);
   solve(2*x=2*x+1,x);
```

Fonction `solve` et variables contraintes

La fonction `solve` *ne tient pas compte* des contraintes que l'on peut appliquer à la variable pour restreindre les solutions dans un domaine particulier (voir le paragraphe *Inéquations* dans ce chapitre pour traiter ce problème)

```
> assume(x>0):
   eq:=x^2-1=0: eq;
   s:=solve(eq,x);
```

$$x \sim \sqrt{2} - 1 = 0$$

$$s := 1, -1$$

De plus, on rappelle que toute assignation à une variable lui donne les propriétés de l'objet attribué (voir chapitre 13).

```
> x:=s[2];
   about(x); # En fait, about(-1) !
               x := -1
-1:
   All numeric values are properties as well as objects.
   Their location in the property lattice is obvious,
   in this case integer.
```

Fonction de Lambert

La fonction `solve` donne les solutions de certaines équations, notamment celles faisant intervenir des exponentielles et des logarithmes en utilisant la fonction W (ou ω) de Lambert, `LambertW`, définie par la propriété fonctionnelle $W(x) e^{W(x)} = x$. *On notera que l'inéterminée* (ici $W(x)$) *peut avoir la forme d'une fonction* (le type `function`).

```
> x:='x':
   Lbt:=W(x)*exp(W(x))=x: Lbt;
   solve(Lbt,W(x));
```

$$W(x) e^{W(x)} = x$$

$$\text{LambertW}(x)$$

```

> type(W(x),function);
true

> eq:=exp(-x+1)=2*x: eq;
S:=solve(eq,x);
evalf(S);

$$e^{(-x + 1)} = 2x$$


$$S := \text{LambertW}\left(\frac{1}{2} e\right)$$

0.6850769421

> eq:=x^2-ln(x)=2: eq;
solve(eq);
evalf(%);

$$x^2 - \ln(x) = 2$$


$$e^{\left(-\frac{1}{2} \text{LambertW}(-2 e^{(-4)}) - 2\right)}, e^{\left(-\frac{1}{2} \text{LambertW}(-1, -2 e^{(-4)}) - 2\right)}$$

0.1379348256, 1.564462258

```

MAPLE considérant toujours par défaut des fonctions à valeurs complexes, **solve** trouvera une solution complexe à l'équation suivante qui, bien sûr, n'a pas de solution réelle (elle a en réalité une infinité de solutions). La fonction **LambertW** possède une infinité de branches dont une seule est analytique en 0 et qui est notée avec un seul argument (voir l'aide en ligne; exécuter la commande > **branches(LambertW);**)

```

> eq:=exp(x)=x: eq;
S:=solve(eq,x);
evalf(S);

$$e^x = x$$


$$S := -\text{LambertW}(-1)$$

0.3181315052 - 1.337235701 I

```

Ensemble infini de solutions

On peut également demander à MAPLE de donner (si possible) l'ensemble des solutions d'une équation possédant une infinité de solutions en rendant vraie la variable logique globale **_EnvAllSolutions** (voir aussi dans l'aide en ligne la variable **_MaxSols**). Le symbole **_Z1~** désigne l'ensemble des entiers relatifs et **_B1~** l'ensemble "binaire" {0,1}. Avec les notations **_Zn~** ou **_Bm~**, les *n* et *m* sont simplement distinctifs si des ensembles de même nature doivent apparaître plusieurs fois.

```

> _EnvAllSolutions:=true;
eq:=cos(x/2)-sqrt(2)/2=0: %;
solve(eq,x);
_EnvAllSolutions:=false;


$$\text{_EnvAllSolutions} := \text{true}$$


$$\cos\left(\frac{1}{2}x\right) - \frac{1}{2}\sqrt{2} = 0$$


```

$$\frac{1}{2}\pi - \pi_{BL} + 4\pi_{ZL}$$

Résolutions de systèmes d'équations

Quand on résout un *système d'équations*, on résout un *ensemble d'équations* (voir chapitre 3) par rapport à un *ensemble de variables*. En effet, l'ordre dans lequel on donne les équations est indifférent et deux équations identiques sont une seule et même équation: c'est bien un ensemble d'équations. Le résultat est aussi une suite d'*ensemble de solutions* (ici les racines sont doubles)

```
> solve({(x-y-1)^2=0, x+y-2=0}, {x, y});

$$\left\{ y = \frac{1}{2}, x = \frac{3}{2} \right\}, \left\{ y = \frac{1}{2}, x = \frac{3}{2} \right\}$$

```

Il est toujours possible de donner un nom aux équations et/ou aux ensembles des équations ou des variables. On pourra par exemple écrire (utile seulement pour des cas compliqués)

```
> eq1:=(x-y-1)^2=0;
eq2:=x+y-2=0;
eq:={eq1,eq2};
var:={x,y};
es:=solve(eq,var);

$$eq1 := (x - y - 1)^2 = 0$$


$$eq2 := x + y - 2 = 0$$


$$eq := \{(x - y - 1)^2 = 0, x + y - 2 = 0\}$$


$$var := \{x, y\}$$


$$es := \left\{ y = \frac{1}{2}, x = \frac{3}{2} \right\}, \left\{ y = \frac{1}{2}, x = \frac{3}{2} \right\}$$

```

Il va de soi que MAPLE ne pourra pas toujours exprimer les solutions exactes de certains systèmes. Par exemple celui-ci possède quatre solutions réelles (calculables numériquement).

```
> sys:={2*exp(-x^2-y^2)=ln(1+x*y), x^2+y^2=1+x*y}: sys;
S:=solve(sys, {x,y});
_SolutionsMayBeLost;

$$\left\{ 2 e^{(-x^2 - y^2)} = \ln(1 + xy), x^2 + y^2 = 1 + xy \right\}$$

S :=
true
```

MAPLE peut résoudre exactement certains systèmes d'équations polynomiales (algébriques). Le résultat montre qu'il existe plusieurs solutions (un système d'équations polynomiales peut ne *pas* avoir de solution). Cependant **solve** ne donnera généralement les solutions exactes que partiellement ou de façon symbolique. Le mot clé **label** permet de distinguer les solutions exprimées formellement par **RootOf**.

```
> sys:={2*x^3-y^2-x-1=0, y^2+x-1=0}, {x,y}: sys;
S:=solve(%);

$$\{2x^3 - y^2 - x - 1 = 0, y^2 + x - 1 = 0\}, \{x, y\}$$

```

```

S := {x = 1, y = 0}, {x = 1, y = 0}, {y = RootOf(-RootOf(_Z^2 - 3 _Z + 3, label = _L4) + _Z^2, label = _L5),
      x = 1 - RootOf(_Z^2 - 3 _Z + 3, label = _L4)}

```

Il faut forcer **solve** à calculer les solutions explicites **quand cela est possible** (voir le chapitre 6)

```
> _EnvExplicit:=true:
```

```
S:=solve(sys);
```

$$S := \{x = 1, y = 0\}, \{x = 1, y = 0\}, \left\{ x = -\frac{1}{2} - \frac{1}{2}I\sqrt{3}, y = -\frac{1}{2}\sqrt{6 + 2I\sqrt{3}} \right\}, \\ \left\{ x = -\frac{1}{2} - \frac{1}{2}I\sqrt{3}, y = \frac{1}{2}\sqrt{6 + 2I\sqrt{3}} \right\}, \left\{ x = -\frac{1}{2} + \frac{1}{2}I\sqrt{3}, y = -\frac{1}{2}\sqrt{6 - 2I\sqrt{3}} \right\}, \\ \left\{ x = -\frac{1}{2} + \frac{1}{2}I\sqrt{3}, y = \frac{1}{2}\sqrt{6 - 2I\sqrt{3}} \right\}$$

Exercice : que signifie cette écriture ?

```
> evalc(rhs(op(2,S[3])));
```

$$-\frac{1}{2}\sqrt{2\sqrt{3} + 3} - \frac{1}{2}I\sqrt{2\sqrt{3} - 3}$$

Attribution des solutions d'un système d'équations et fonction assign

Si l'on veut que les variables prennent les valeurs d'**une des** solutions d'un système d'équations on peut utiliser la fonction **assign** (cette fonction agit aussi avec les solutions d'une équation simple).

```
> restart:
```

```
_EnvExplicit:=true:
sys:={3*s-1=0,t^2-s=2}: %;
sol:=solve(sys,{s,t});
```

$$\{3s - 1 = 0, t^2 - s = 2\} \\ sol := \left\{ s = \frac{1}{3}, t = \frac{1}{3}\sqrt{21} \right\}, \left\{ s = \frac{1}{3}, t = -\frac{1}{3}\sqrt{21} \right\}$$

La réponse de **assign** étant toujours **NULL**, aucun d'affichage n'apparaît

```
> assign(sol);
```

Les variables *s* et *t* sont maintenant assignées à la **dernière des solutions obtenue** et les équations du système sont maintenant évaluées comme étant des identités

```
> s; t; sys;
```

$$\frac{1}{3} \\ -\frac{1}{3}\sqrt{21} \\ \{2 = 2, 0 = 0\}$$

La fonction **assign** ne génère pas automatiquement de nouveaux noms pour ranger toutes les solutions. Seule la **dernière** solution obtenue par **solve** est pris en compte.

Maintenant on s'aperçoit que l'on s'est trompé dans l'écriture du système (=2 doit par exemple être changé en =3). Après correction on recommence

```
> sys:={3*s-1=0,t^2-s=3}:
sol:=solve(sys,{s,t});
Error, (in solve) a constant is invalid as a variable, 1/3, -1/3*21^(1/2)
```

On était trop pressé d'effectuer les assignations : les variables étant assignées aux solutions, ce ne sont plus des indéterminées. La fonction **assign** est pratique mais présente cet inconvénient. Il vaut mieux, le plus souvent, donner aux solutions des noms différents de ceux des variables. On peut néanmoins corriger facilement la situation en désassignant les indéterminées

```
> s:='s':t:='t':
sys:={3*s-1=0,t^2-s=3}:
sol:=solve(sys,{s,t});
sol :=  $\left\{ s = \frac{1}{3}, t = \frac{1}{3}\sqrt{30} \right\}, \left\{ s = \frac{1}{3}, t = -\frac{1}{3}\sqrt{30} \right\}$ 
```

On donne maintenant un nom différent pour chaque solution en opérant par substitution (voir chapitre 2)

```
> s_1,t_1 := subs(sol[1],s), subs(sol[1],t);
s_2,t_2 := subs(sol[2],s), subs(sol[2],t);
s_1, t_1 :=  $\frac{1}{3}, \frac{1}{3}\sqrt{30}$ 
s_2, t_2 :=  $\frac{1}{3}, -\frac{1}{3}\sqrt{30}$ 
```

On peut vérifier les identités qui résultent des assignations précédentes. On crée ici une fonction de vérification pour raccourcir l'écriture.

Exercice : expliquer la construction de **verif**

```
> verif:=(x,y)->op(map(evalb,subs({s=x,t=y},sys)));
verif(s_1,t_1), verif(s_2,t_2), verif(t_2,t_1);
verif := (x, y) → op(map(evalb, subs({s = x, t = y}, sys)))
true, true, false
```

Résolutions des inéquations et systèmes d'inéquations

On peut résoudre également des inéquations avec **solve**. Notez que les signes utilisables sont $>$, $<$, \geq et \leq mais pas $=>$ ou $=<$ qui provoquent une erreur de syntaxe. La fonction **solve** donne une suite (sequence) d'ensembles de conditions indépendantes qui doivent être satisfaites pour que l'inégalité le soit. La réponse donnée ici signifie que P est positif ou nul si $x \geq 2$ ou si $(1-\sqrt{5})/2 \leq x \leq (1+\sqrt{5})/2$ (*RealRange(a, b)* signifiant "Intervalle Réel fermé $[a,b]$ "; voir le chapitre 13).

```
> restart:
P:=x^3-3*x^2+x+2:P;
solve(P >= 0,x);
x^3 - 3 x^2 + x + 2
RealRange $\left(\frac{1}{2} - \frac{1}{2}\sqrt{5}, \frac{1}{2} + \frac{1}{2}\sqrt{5}\right), RealRange(2, \infty)$ 
```

RealRange(Open(a), Open(b)) signifie "Intervalle Réel $]a,b[$ ", *Open(a)* signifiant a exclu :

```
> solve(P > 0,x);
```

$$\text{RealRange}\left(\text{Open}\left(\frac{1}{2} - \frac{1}{2}\sqrt{5}\right), \text{Open}\left(\frac{1}{2} + \frac{1}{2}\sqrt{5}\right)\right), \text{RealRange}(\text{Open}(2), \infty)$$

Si l'inéquation est donnée comme l'élément d'un ensemble, la présentation des résultats change (il n'y a pas de sens précis à ce changement de présentation)

```
> solve({P > 0}, x);
```

$$\left\{ \frac{1}{2} - \frac{1}{2}\sqrt{5} < x, x < \frac{1}{2} + \frac{1}{2}\sqrt{5} \right\}, \{2 < x\}$$

La résolution des *systèmes d'inéquations* ou des *systèmes mixtes d'équations-inéquations* renvoie une réponse présentée de façon similaire. Le sens de **RootOf** est le suivant: x doit être inférieur à une des deux racines de l'équation $x^2 - x - 1 = 0$ et plus précisément celle dont la valeur numérique est 1.618033989... Le résultat suivant signifie donc que, soit x doit être positif (contrainte imposée) et inférieur à 1.618..., soit x doit être supérieur à 2.

```
> solve({P > 0, x > 0}, x);
```

$$\{0 < x, x < \text{RootOf}(\underline{Z}^2 - \underline{Z} - 1, 1.618033989)\}, \{2 < x\}$$

On cherche maintenant les solutions strictement positives de $P = 0$. Ceci répond au problème que nous n'avions pas pu résoudre avec la fonction **assume**

```
> solve({P = 0, x > 0}, x);
```

$$\{x = 2\}, \{x = \text{RootOf}(\underline{Z}^2 - \underline{Z} - 1, 1.618033989)\}$$

Si l'on veut avoir (si possible) la forme symbolique associée à la valeur numérique on pourra résoudre $P = 0$ ou utiliser la magie de **identify** (voir la fin du chapitre 1)

```
> [solve(P)]; evalf(%);
identify(1.618033989);
```

$$\left[2, \frac{1}{2} + \frac{1}{2}\sqrt{5}, \frac{1}{2} - \frac{1}{2}\sqrt{5} \right]$$

$$[2., 1.618033988, -0.6180339880]$$

$$\frac{1}{2} + \frac{1}{2}\sqrt{5}$$

Mais on peut aussi assigner **true** à la variable **_EnvExplicit** pour obtenir (quand c'est possible) des solutions explicites.

```
> _EnvExplicit:=true:
```

```
solve({P > 0, x > 0}, x);
solve({P = 0, x > 0}, x);
```

$$\left\{ 0 < x, x < \frac{1}{2} + \frac{1}{2}\sqrt{5} \right\}, \{2 < x\}$$

$$\{x = 2\}, \left\{ x = \frac{1}{2} + \frac{1}{2}\sqrt{5} \right\}$$

Autre exemple dans lequel on ne donne pas le nom de l'indéterminée x que **solve** trouve seule. Ici aussi on obtient des solutions explicites parce que **_EnvExplicit=true**

```
> solve({x^3-3*x^2+x+2>0, x^2-2>0});
```

$$\left\{ x < \frac{1}{2} + \frac{1}{2}\sqrt{5}, \sqrt{2} < x \right\}, \{2 < x\}$$

Bien sûr MAPLE ne trouvera pas, comme pour les systèmes d'équations, de réponses pour tous les systèmes

```
> _EnvExplicit:=false;
Ex1:=x^3-3*x^2-exp((x-2)^2)+5;
Ex2:=exp(x^2)-2;
S:=solve({Ex1 > 0, Ex2 > 0},x);
_SolutionsMayBeLost;
```

*S :=
true*

Ce système d'inéquations a pourtant des solutions puisque, par exemple,

```
> is(eval(Ex1,x=3)>0) and is(eval(Ex2,x=3)>0);
true
```

Résolutions numériques : fonction fsolve

La résolution exacte d'une équation ou d'un système d'équations n'étant pas toujours possible, MAPLE permet de trouver les valeurs numériques des solutions avec la fonction **fsolve** (floating solve). Deux situations peuvent se présenter : les équations polynomiales ou non polynomiales.

1) *Les équations non polynomiales*: reprenons un exemple déjà rencontré pour lequel **solve** ne donne aucune réponse. Elle dit pourtant qu'il y a peut-être des solutions.

```
> restart;
p:=x->cos(sqrt(x))-x^2;
p(x)=0;
S:=solve(% ,x);
_SolutionsMayBeLost;
```

$\cos(\sqrt{x}) - x^2 = 0$
*S :=
true*

La première démarche est d'essayer de déterminer, si possible par analyse, le nombre de solutions que peut avoir l'équation (si elle en possède !) et de les localiser graphiquement (voir chapitre 16). On ne considérera ici que $x \geq 0$ (pour $x < 0$, $\cos(\sqrt{x})$ devient $\cosh(\sqrt{|x|})$). Comme $|\cos(\sqrt{x})| \leq 1$ on aura $\cos(\sqrt{x}) - x^2$ toujours < 0 pour $x \geq 1$. S'il y a des racines, elles seront donc dans $[0, +1]$. La fonction p étant continue, un calcul simple montre qu'il y a au moins un nombre impair de racines positives

```
> is(p(0)*p(1)<0);
true
```

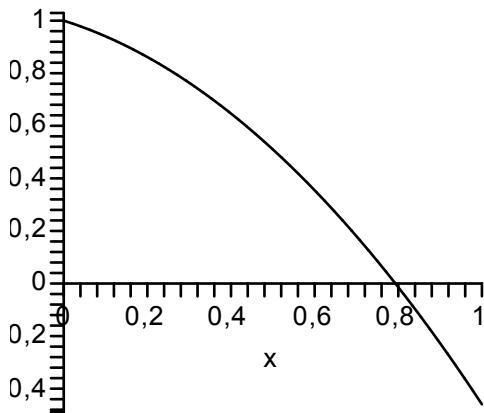
Calculons la dérivée

```
> dp:=diff(p(x),x);
dp := - \frac{\sin(\sqrt{x})}{2\sqrt{x}} - 2x
```

On vérifie que pour x compris entre 0 et +1, p est strictement décroissante.

```
> is(dp < 0) assuming x>=0, x<=1;
                                         true
```

Il ne peut donc y avoir qu'une racine positive. Une représentation graphique (voir chapitre 16) montre en effet qu'elle est localisée autour de 0.8



La fonction **fsolve** peut s'utiliser exactement comme **solve** mais nous verrons qu'il existe des variantes

```
> fsolve(p(x)=0);
                                         0.7930841288
```

Exercice: on peut toujours espérer ! Que cherche-t-on ici ? (voir la fin du chapitre 1)

```
> identify(%);
                                         0.7930841288
```

Si besoin (mais il faut avoir une bonne justification), on peut augmenter la précision de la solution

```
> Digits:=50:
  fsolve(p(x)=0,x);
                                         0.79308412879489318208322803361124038501113179658387
```

Pour certaines équations dont on peut penser qu'elles pourraient engendrer des difficultés de calcul numérique (équations mal conditionnées), on peu contraindre **fsolve** à effectuer ses calculs en conservant en permanence, la totalité de la précision demandée. Manifestement ici, ce n'est pas le cas

```
> fsolve(p(x),x,fulldigits);   Digits:=10:
                                         0.79308412879489318208322803361124038501113179658387
```

Considérons maintenant l'équation suivante dépendant du paramètre k (on rappelle que l'on peut construire une fonction, ici **eq**, dont l'image est une équation; voir chapitre 7). On voit que **solve** ne trouve pas de solution explicite, que l'on fixe ou non la valeur de k

```
> eq:=(k,z)->sin(z)=k*z;   eq(k,x);
  solve(% ,x);
                                         eq := (k, z) → sin(z) = k z
                                         sin(x) = k x
                                         RootOf(_Z k - sin(_Z))
```

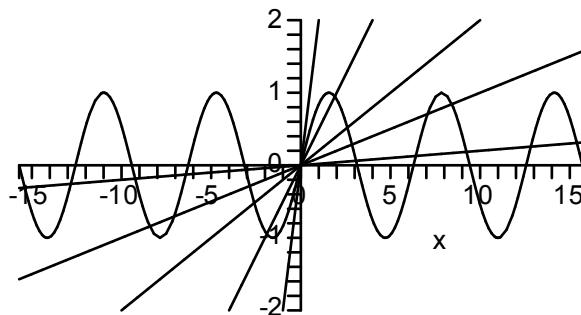
```
> eq(1/4,x);
solve(% ,x);
```

$$\sin(x) = \frac{1}{4}x$$

`RootOf(-_Z + 4 sin(_Z))`

Il va de soi que la recherche *numérique* d'une solution n'a de sens que si l'équation ne possède qu'une indéterminée (ici x , une valeur numérique étant assignée à k). Les représentations graphiques ci-dessous sont celles de $\sin(x)$ et de kx pour $k = 2, 1/2, 1/5, 1/10$ et $1/50$. Elles montrent à l'évidence que le nombre N de solutions (intersections des droites et de la sinusoïde) dépend de la valeur de k par un relation qui n'est pas simple. En prenant en compte la solution triviale $x = 0$, on passe ainsi de $N = 1$ pour $k > 1$, à $N = 3$ pour $k = 1/2$ et $k = 1/5$, à $N = 7$ pour $k = 1/10$ et à $11 < N$ solutions pour $k \leq 1/50$.

A l'exception des équations polynomiales, il n'existe pas, en général, de règles simples permettant de déterminer le nombre de solutions des équations.

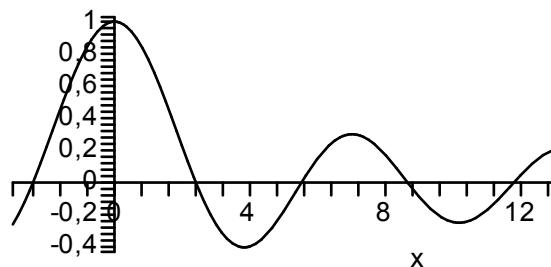


Cherchons maintenant les zéros de la fonction de Bessel J_0 (voir la représentation graphique ci-dessous). Il y en a une infinité et **solve** (ni personne !) ne pouvant exprimer exactement les racines, renvoie une solution symbolique:

```
> eq:=BesselJ(0,x)=0: eq;
solve(eq,x);
```

`BesselJ(0, x) = 0`

`RootOf(BesselJ(0, _Z))`



La fonction **fsolve** donne une solution puis s'arrête de chercher. Mais ce n'est pas nécessairement celle qui est souhaitée.

```
> fsolve(eq,x);
```

`-2.404825558`

On peut alors fixer l'intervalle contenant la racine cherchée (voir l'aide en ligne de **BesselJZeros(n,m)** qui représente le m -ième zéro réel pour x positif de $J_n(x)$)

```
> x_1:=fsolve(eq,x=2..3);
  BesselJZeros(0.,1); # Pour vérification
          x_1 := 2.404825558
          2.404825558

> x_3:=fsolve(eq,x=8..10);
          x_3 := 8.653727913
```

Il est *inutile d'augmenter l'intervalle de recherche* pour essayer d'obtenir plusieurs solutions car **fsolve** s'arrête dès qu'une solution est trouvée.

```
> x_i:=fsolve(eq,x=2..12);
          x_i := 5.520078110
```

Si l'intervalle ne contient aucune solution, **fsolve** renvoie son expression non évaluée.

```
> x_f:=fsolve(eq,x=1..2);
          x_f := fsolve(BesselJ(0, x) = 0, x, 1 .. 2)
```

En programmation (voir chapitres 18 et 19) on pourra distinguer ces deux types de résultats avec

```
> type(x_i,function), type(x_f,function);
          false, true
```

ou inversement

```
> type(x_i,numeric), type(x_f,numeric);
          true, false
```

Grâce à l'option **avoid** (éviter) on peut demander à **fsolve** de chercher une solution en excluant une ou plusieurs solutions connues ou déjà trouvées dans l'intervalle spécifié. On remarquera que ces solutions n'ont pas à être données dans un ordre particulier (c'est un ensemble).

```
> x_2:=fsolve(eq,x=4..8):
  x_4:=fsolve(eq,x=2..12,avoid={x=x_1,x=x_3,x=x_2});
          x_4 := 11.79153444
```

Lorsque **fsolve** ne trouve plus de solution il renvoie la commande non évaluée

```
> fsolve(eq,x=2..12,avoid={x=x_1,x=x_3,x=x_2,x=x_4});
  type(% , numeric);
          fsolve(BesselJ(0, x) = 0, x, 2 .. 12,
          avoid = {x = 5.520078110, x = 11.79153444, x = 2.404825558, x = 8.653727913})
          false
```

Une autre possibilité est de demander à **fsolve** de calculer une solution en donnant simplement une valeur au *voisinage* de la solution cherchée.

```
> fsolve(eq,x=2);
  fsolve(eq,x=6);
          2.404825558
          5.520078110
```

Attention car avec cette syntaxe, **fsolve** n'est pas contraint de donner une solution appartenant à un intervalle spécifié et peut renvoyer une valeur qui peut être très éloignée de celle cherchée (voir dans un livre la notion de *bassin d'attraction*)

Exercice : on a forcé ici la malchance. Pouvez-vous dire qu'elle est probablement la méthode numérique utilisée par **fsolve** et expliquer sa réponse ?

```
> x_voisin:=fsolve(diff(BesselJ(0,x),x)=0,x=6..8);
  fsolve(eq,x=x_voisin);
x_voisin := 7.015586670
5.322270826 109
```

Pour les *systèmes d'équations* on procèdera de même. Le système suivant possède au moins 4 solutions réelles (que le lecteur se rassure, on ne peut pas affirmer cela simplement en regardant le système !) et **fsolve** n'en donnera qu'une

```
> sys:={2*exp(-x^2-y^2)=ln(2+x*y),x^2+y^2=1+x*y}: sys;
  fsolve(sys,{x,y});
{ 2 e(-x2 - y2) = ln(2 + x y), x2 + y2 = 1 + x y}
{y = -1.016750525, x = -0.03438815532}
```

On peut alors chercher une autre solution en précisant un ou plusieurs intervalles. Le rôle symétrique joué par les variables de notre exemple nous fournit ici un guide mais, en général, la donnée de ces intervalles est difficile sans une représentation graphique toujours possible si le système est à deux variables (c'est ainsi que l'on a pu affirmer que le système précédent avait quatre solutions réelles). Pour plus de deux variables, sauf cas particulier pour lequel une analyse mathématique est possible, on ne peut que chercher "à l'aveuglette" ou tenter de savoir si le système possède une solution dans un domaine donné. *Il faut noter que ces remarques ne constituent pas une restriction due à MAPLE mais bien une problème mathématique intrinsèque dont on ne connaît pas de solution dans le cas général.*

```
> xy_1:=fsolve(sys,{x,y},x=-2..2,y=-2..2);
  xy_2:=fsolve(sys,{x,y},x=-2..2,y=-2..2,avoid=xy_1);
  xy_3:=fsolve(sys,{x,y},x=-2..2,y=-2..2,avoid={xy_1,xy_2});
  xy_4:=fsolve(sys,{x,y},x=-2..2,y=-2..2,avoid={xy_1,xy_2,xy_3});
  type(fsolve(sys,{x,y},x=-2..2,y=-2..2,
    avoid={xy_1,xy_2,xy_3,xy_4}),numeric);
xy_1 := {y = -1.016750525, x = -0.03438815532}
xy_2 := {y = 0.03438815532, x = 1.016750525}
xy_3 := {x = 0.03438815532, y = 1.016750525}
xy_4 := {x = -1.016750525, y = -0.03438815532}
false
```

2) *Les équations polynomiales*: nous avons déjà évoqué le problème de la recherche des racines au chapitre 6, *Polynomes et Fractions rationnelles*. MAPLE "sait" que le nombre des solutions est égal au degré du polynôme. Comparons les résultats dans un cas simple que MAPLE peut toujours résoudre par **solve** (équation à coefficients rationnels de degré inférieur à 5). On rappelle néanmoins que même pour des équations de degré inférieur à 5, **solve** peut répondre par la solution symbolique **RootOf** si les racines ont une forme trop compliquée. Pour obtenir les solutions explicites il faut ici assigner la valeur **true** à la variable globale **_EnvExplicit**. La fonction **solve** n'a aucune difficulté à détecter toute seule qu'il s'agit d'une équation

polynomiale (type *polynom* ou *ratpoly* pour les deux membres et coefficients rationnels).

```
> restart;

$$\text{eqp} := x^4 - x^3 - 4 = -6x + 2x^2; \quad \text{type(rhs(%), ratpoly);}$$


$$\text{_EnvExplicit:=true;}$$


$$\text{solve(eqp, x);}$$

```

$$x^4 - x^3 - 4 = -6x + 2x^2$$

true

$$1, -2, 1 + I, 1 - I$$

L'utilisation de **fsolve** est alors semblable à celle de **solve** mais elle ne donne par défaut que les *racines réelles* (pour des coefficients réels; voir ci-dessous)

```
> fsolve(eqp);
-2.000000000, 1.000000000
```

S'il n'y a pas de solution *réelle*, la réponse est **NULL**

```
> S:=fsolve(x^2+1);
S :=
```

Naturellement **fsolve** peut résoudre des équations polynomiales à coefficients quelconques (numériquement définis). Elle peut aussi trouver seule l'indéterminée (on n'a pas précisé *x*)

```
> eqf:=sqrt(2)*x^3+Pi/2*x^2-0.451e-1*x+3/5=0;
fsolve(eqf);

$$eqf := \sqrt{2} x^3 + \frac{1}{2} \pi x^2 - 0.0451 x + \frac{3}{5} = 0$$

-1.362623575
```

La fonction **realroot** permet de donner des intervalles contenant les racines réelles d'un polynôme à *coefficients rationnels*. On doit transformer au préalable les deux membres de l'équation en polynôme unique.

```
> realroot(lhs(eqp)-rhs(eqp));
realroot(lhs(eqf)-rhs(eqf));
[[1, 1], [-4, 0]]
```

Error, (in realroot) requires exact, non-floating polynomial coefficients

Ceci permet de préciser l'intervalle de recherche d'une racine réelle particulière.

Exercice : expliquer cette écriture (un peu compliquée pour pas grand chose !)

```
> fsolve(eqp, x=%[2][1]..%[2][2]);
-2.000000000
```

On peut aussi spécifier par un troisième argument, **complex**, la recherche de toutes les solutions (les solutions réelles étant des solutions complexes particulières). On notera que la variable doit être précisée car **complex** doit être le *troisième argument*.

```
> fsolve(eqp, x, complex);
-2.000000000, 1. - 1. I, 1.000000000, 1. + 1. I
```

```
> fsolve(eqf, x, complex);
-1.362623575, 0.1259514202 - 0.5435940331 I, 0.1259514202 + 0.5435940331 I
```

Remarque: si l'un au moins des coefficients est complexe, il n'y a besoin de rien préciser

```
> fsolve(3*x^2+(Pi-2*I)*x+1=0);
-0.8538253834 + 0.8618580528 I, -0.1933721679 - 0.1951913861 I
```

Ici **solve** trouve 4 racines exactement et donne les 5 autres formellement avec **RootOf**. On rappelle que si l'égalité est absente, **solve** comme **fsolve** suppose $P = 0$. L'écriture $Index=n$ distingue les racines

```
> P := x^9+2*x^8-10*x^7-25*x^6+11*x^5+
      57*x^4+15*x^3-41*x^2-15*x+9;
solve(P);
P := x^9 + 2 x^8 - 10 x^7 - 25 x^6 + 11 x^5 + 57 x^4 + 15 x^3 - 41 x^2 - 15 x + 9
-1, 3, - $\frac{1}{2}$  +  $\frac{1}{2}\sqrt{13}$ , - $\frac{1}{2}$  -  $\frac{1}{2}\sqrt{13}$ , RootOf(_Z^5 + 3 _Z^4 + _Z^3 - 3 _Z^2 - 2 _Z + 1, index = 1),
RootOf(_Z^5 + 3 _Z^4 + _Z^3 - 3 _Z^2 - 2 _Z + 1, index = 2),
RootOf(_Z^5 + 3 _Z^4 + _Z^3 - 3 _Z^2 - 2 _Z + 1, index = 3),
RootOf(_Z^5 + 3 _Z^4 + _Z^3 - 3 _Z^2 - 2 _Z + 1, index = 4),
RootOf(_Z^5 + 3 _Z^4 + _Z^3 - 3 _Z^2 - 2 _Z + 1, index = 5)

> fsolve(P);
-2.302775638, -2.120399841, -1.000000000, 0.3581754456, 0.8915986834, 1.302775638, 3.000000000

> fsolve(P,x,complex);
-2.302775638, -2.120399841, -1.064687144 - 0.5858544088 I, -1.064687144 + 0.5858544088 I,
-1.000000000, 0.3581754456, 0.8915986834, 1.302775638, 3.000000000
```

Exercice : Que cherche-t-on à faire ? Expliquer les détails des instructions utilisées

1)

```
> Nrr:=sturm(sturmseq(P,x),x=-infinity..infinity);
op(select(not is,[fsolve(P,x,complex)],real));
evalb(degree(P)-Nrr = nops(%));
Nrr := 7
-1.064687144 - 0.5858544088 I, -1.064687144 + 0.5858544088 I
true
```

2)

```
> R_exa:=op(select(not is,[solve(P)],RootOf));
R_num:=fsolve(quo(P,mul((x-xi),xi in [R_exa]),x),x,complex);
evalb(nops([R_exa])+nops([R_num])=degree(P));
R_exa := -1, 3, - $\frac{1}{2}$  +  $\frac{1}{2}\sqrt{13}$ , - $\frac{1}{2}$  -  $\frac{1}{2}\sqrt{13}$ 
R_num := -2.120399841, -1.064687144 - 0.5858544088 I, -1.064687144 + 0.5858544088 I, 0.3581754456,
0.8915986834
true
```

Module RootFinding

Ce module fournit des méthodes de résolution d'équations ou de système d'équations qui étendent et

complètent celles que nous venons de décrire.

Résolution numérique d'une équation dans le plan complexe

Reprendons une équation que nous avons déjà résolue symboliquement avec la fonction W de Lambert

```
> restart: eq:=exp(x)=x: eq;
```

$$e^x = x$$

```
> s:=solve(eq,x);
```

$$s := -\text{LambertW}(-1)$$

Cette solution est bien sûr complexe

```
> evalf(s);
```

$$0.3181315052 - 1.337235701 I$$

Cette équation possède au moins deux solutions, l'autre étant le complexe conjuguée de s . Malheureusement Maple ne sachant pas exprimer $\overline{W(-1)}$, on ne peut pas tester symboliquement ce résultat. On peut néanmoins le vérifier numériquement

```
> simplify(subs(x=evalf(conjugate(s)),eq));
```

$$0.3181315058 + 1.337235701 I = 0.3181315052 + 1.337235701 I$$

Contrairement à **fsolve** la fonction **Analytic** de **RootFinding** nous permet, par la méthode de **Newton_Raphson**, d'explorer un "rectangle" du plan complexe en donnant les affixes du "coin inférieur gauche" et du "coin supérieur droit" (le rectangle ne doit pas se réduire à une droite). Elle nous donne bien les deux solutions attendues.

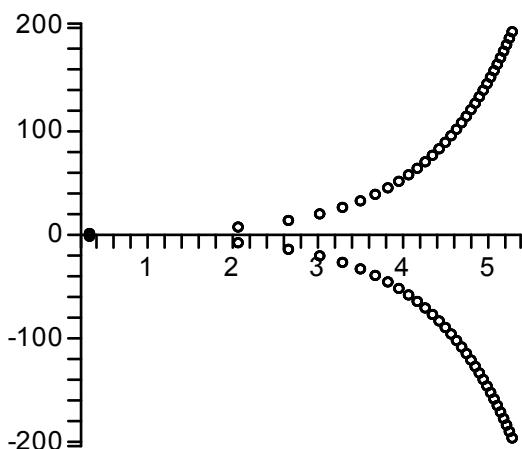
```
> S:=RootFinding:-Analytic(eq,x=-1-2*I..1+2*I);
```

$$S := 0.31813150520476 - 1.3372357014307 I, 0.31813150520476 + 1.3372357014307 I$$

La fonction présente d'autres possibilités comme celle de récupérer, avec **AnalyticZerosFound**, les zéros calculés, même si le calcul est interrompu (voir l'aide en ligne). En fait, l'équation a une infinité de solutions (conjuguées) qui apparaissent en élargissant le champ de recherche. On utilise une autre écriture possible pour **Analytic**

```
> RootFinding:-Analytic(eq,x,im=-200..200,re=0..200):
```

```
> plots[complexplot]([%],style=point,  
symbol=circle,color=black); # Voir Ch. 16
```



Système de polynômes bivariés

La fonction **BivariatePolynomial** permet de résoudre un système de 2 (ou plus) polynômes dépendants de 2 (et pas plus) variables. **Attention:** la méthode de fonctionne que si le nombre de solutions est fini.

```
> sys:=[x*y-3*x-2,-3*x^3*y+1]: sys;
[x y - 3 x - 2, -3 x3 y + 1]
```

Les solutions sont présentées sous forme d'une *suite de listes*

```
> S:=RootFinding:-BivariatePolynomial(sys,[x,y]);
S := [-0.5000000000 + 0.2886751346 I, 2.8 10-14 - 1.732050808 I],
[-0.5000000000 - 0.2886751346 I, 2.7 10-14 + 1.732050808 I],
[0.3333333333 + 9.639895416 10-16 I, 9.000000000 + 1.171806279 10-14 I]
```

Exercice: On peut vérifier une solution aux erreurs numériques près. Expliquez ces écritures

```
> subs([x=S[1,1],y=S[1,2]],sys);
[0. + 2 10-10 I, 0. + 2.325281353 10-11 I]
```

Exercice: Que cherche-t-on à faire ? Expliquez les commandes

```
> identify(0.2886751346);
 $\frac{1}{6}\sqrt{3}$ 

> X[1], Y[1]:= -1/2+I*sqrt(3)/6 , -I*sqrt(3):
X[2], Y[2]:= conjugate(X[1]) , conjugate(Y[1]):
X[3], Y[3]:= 1/3 , 9:

> simplify(subs([x=X[1],y=Y[1]],sys)),
simplify(subs([x=X[2],y=Y[2]],sys)),
simplify(subs([x=X[3],y=Y[3]],sys));
[0, 0], [0, 0], [0, 0]
```

Exercice: expliquez les commandes

```
> _EnvExplicit:=true:
solve({op(sys)},{x,y});
 $\left\{ x = \frac{1}{3}, y = 9 \right\}, \left\{ x = -\frac{1}{2} + \frac{1}{6}I\sqrt{3}, y = -I\sqrt{3} \right\}, \left\{ x = -\frac{1}{2} - \frac{1}{6}I\sqrt{3}, y = I\sqrt{3} \right\}$ 
```

Exercice: expliquez encore. Ce qui montre que MAPLE est plein de ressources...

```
> Sys:=Groebner[gbasis](sys,lexdeg([y],[x]));
Sys := [9 x3 + 6 x2 - 1, -3 + y - 18 x2 - 12 x]
```

```
> X:=solve(%[1],x): S:=[];
for i to degree(Sys[1]) do
  S:=[op(S),[x=X[i],y=solve(subs(x=X[i],Sys[2]),y)]];
end do:
op(S);
_EnvExplicit:=false:
```

$$\left[x = \frac{1}{3}, y = 9 \right], \left[x = -\frac{1}{2} + \frac{1}{6}I\sqrt{3}, y = -I\sqrt{3} \right], \left[x = -\frac{1}{2} - \frac{1}{6}I\sqrt{3}, y = I\sqrt{3} \right]$$

Solution d'un système d'équations polynomiales par homotopie

La fonction **Homotopy** permet d'obtenir les solutions d'un système de polynômes multivariés par intégration d'un système différentiel sur une courbe d'homotopie. La méthode peut demander un temps de calcul important. On peut cependant contrôler le pas d'intégration ou la précision (voir l'aide en ligne).

```
> sys:=[5*x*y-3*x*z-1,-3*x^3*y*z-2*x+z^2+1,4*x+2*y^2-z^2]: sys;
[5 x y - 3 x z - 1, -3 x3 y z - 2 x + z2 + 1, 4 x + 2 y2 - z2]
```

```
> S:=RootFinding:-Homotopy(sys):
```

Les solutions sont présentées sous la forme d'une *liste de listes* (un peu longue pour être imprimée). On affiche ici la septième qui correspond à une solution réelle écrite sous la forme $a + 0. I$. On peut simplifier cette (ces) écriture(s) en appliquant **simplify** à la liste des solutions

```
> S[7];
S:=simplify(S):
S[7];
[x = 0.8389156704 + 0. I, y = -1.347739378 - 0. I, z = -2.643570605 - 0. I]
[x = 0.8389156704, y = -1.347739378, z = -2.643570605]
```

On trouve 12 solutions que l'on peut vérifier numériquement aux erreurs près.

```
> nops(S);
subs(S[7],sys), subs(S[1],sys);
12
[-2 10-9, 1 10-9, 0.], [0. + 2.3 10-9 I, 0. + 0. I, 0. - 1 10-8 I]
```

Résolutions des relations de récurrence

La fonction **rsolve** permet de résoudre certaines relations de récurrence. Nous nous contenterons ici d'un exemple avec la suite de Fibonacci (Léonard de Pise, dit "fils de Bonacci", 1175-1240). Soit la relation $u_{n+1} = u_n + u_{n-1}$ avec par exemple $\{u_0 = 1, u_1 = 2\}$. On cherche l'expression générale qui permet de calculer directement u_n (Jacques Binet, 1786-1856):

```
> restart:
u_n:=rsolve({u(n+1)=u(n-1)+u(n),u(0)=1,u(1)=2},u);
u_n := \left( \frac{3}{10}\sqrt{5} + \frac{1}{2} \right) \left( \frac{1}{2} + \frac{1}{2}\sqrt{5} \right)^n + \left( -\frac{3}{10}\sqrt{5} + \frac{1}{2} \right) \left( \frac{1}{2} - \frac{1}{2}\sqrt{5} \right)^n
```

Exercice: transformer u_n en une fonction U telle que $U(n) = u_n$. Montrez en utilisant la solution de l'équation de récurrence que $u_5 = 13$.

Jeux et exercice : demandez à quelqu'un de choisir deux nombres a et b . Faites lui calculer la suite $a, b, a+b, a+2b, 2a+3b$, etc... jusqu'à ce qu'elle contienne 10 termes (a et b compris), chaque terme étant la somme des deux précédents. Demandez lui ensuite de calculer la somme des 10 termes obtenus (si a et b ne sont pas choisis petits, ce peut être assez long). Vous pouvez donner la réponse quasi instantanément ...

- a) Calculez l'expression générale des termes de la série.
 b) Calculez l'expression de la somme des 10 premiers termes d'une suite de Fibonacci en fonction de a et b .
 c) Calculez l'expression du septième terme.
 d) En déduire la solution de ce tour.

Résolutions de systèmes linéaires

Les fonctions **solve** et **fsolve** pourraient bien entendu être employées pour résoudre des systèmes linéaires mais on préférera utiliser la fonction **LinearSolve** de la bibliothèque **LinearAlgebra** pour simplifier la syntaxe des commandes et *surtout utiliser des algorithmes appropriés* (on n'utilisera surtout pas une matrice inverse, méthode trop peu efficace). Il suffit de définir la matrice du système et son second membre avec les fonctions **Matrix** et **Vector** (voir chapitre 4). On notera que

- Le résultat de **LinearSolve** est un vecteur anonyme et on devra effectuer une assignation si on veut donner un nom au vecteur solution.

- La forme du résultat dépendra de celle des données de la matrice et/ou du second membre.

- Si tous les termes de la matrice ou du second membre sont donnés sous forme rationnelle ou symbolique, la solution sera sous la même forme. **Attention** dans ce cas au temps de calcul et au "gigantisme" des expressions de la solution quand la taille du système augmente.

*- Si tous les termes de la matrice et du second membre ont le type **numeric**, il suffit qu'un seul d'entre eux soit sous forme décimale* pour que l'ensemble du vecteur solution soit calculé sous forme décimale. Le solveur de **LinearAlgebra** accède alors, pour plus d'efficacité, aux algorithmes NAG de calculs numériques (voir chapitre 4). On rappelle (voir chapitre 2) que si $\sqrt{3}$ ou π sont des constantes numériques, elles ont pour MAPLE le type **constant** mais pas le type **numeric** et l'application de **evalf** ou **evalhf** est nécessaire (pour avoir le type **numeric** un nombre ne doit contenir aucun symbole sinon le e ou E de la notation scientifique de la puissance de 10).

```
> restart;
type(Pi,constant), type(Pi,numeric);
true,false

> f_Pi:=evalhf(Pi);
type(f_Pi,constant), type(f_Pi,numeric), type(3.45e-3,numeric);
f_Pi := 3.14159265358979312
true, true, true

> with(LinearAlgebra):
M,b_e:=Matrix(3,3,[[1,-4,3],[-2,1,0],[-1,-2,6]]),
  Vector([-3,2,1]);
x_e:=Transpose(LinearSolve(M,b_e));
M, b_e := 
$$\begin{bmatrix} 1 & -4 & 3 \\ -2 & 1 & 0 \\ -1 & -2 & 6 \end{bmatrix}, \begin{bmatrix} -3 \\ 2 \\ 1 \end{bmatrix}$$

x_e := 
$$\begin{bmatrix} -\frac{5}{9}, \frac{8}{9}, \frac{10}{27} \end{bmatrix}$$

```

Il suffit qu'**un seul** des coefficients de la matrice ou du second membre soit exprimé sous forme décimale

pour que la totalité de la solution soit exprimée sous cette forme (ici on remplace 1 par 1.0 dans le second membre).

Attention: il est fortement conseillé de lire attentivement les paragraphes relatifs aux calculs numériques avec les matrices et la bibliothèque **LinearAlgebra** (chapitre 4).

```
> b_d:=Vector([-3,2,1.0]):  
x_d:=LinearSolve(M,b_d);
```

$$x_d := \begin{bmatrix} -0.55555555555555580 \\ 0.88888888888888729 \\ 0.370370370370370294 \end{bmatrix}$$

Vérifions les deux solutions

```
> M . x_e - b_e, M . x_d - b_d;
```

$$\begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ -1.11022302462515654 \cdot 10^{-16} \end{bmatrix}$$

LinearSolve permet aussi, parmi d'autres options, de fixer la méthode de résolution. La matrice S est ici *symétrique définie positive* (voir plus bas).

```
> S:=Matrix([[13,5,9],[5,14,-3],[9,-3,10]]);
```

$$S := \begin{bmatrix} 13 & 5 & 9 \\ 5 & 14 & -3 \\ 9 & -3 & 10 \end{bmatrix}$$

On sait que la **méthode de Cholesky** est alors applicable et efficace (l'effet sera sensible pour les systèmes de grandes tailles). De plus **LinearSolve** permet de résoudre simultanément plusieurs systèmes linéaires ayant la même matrice. Les seconds membres sont donnés sous forme de matrice, colonne par colonne (voir chapitre 4). Les solutions sont données dans une matrice solution et rangées par colonnes correspondantes.

```
> b:=-<-2,0,1>|<2,3,-1>;  
x:=LinearSolve(S,b,method='Cholesky'): b,x;
```

$$\begin{bmatrix} -2 & 2 \\ 0 & 3 \\ 1 & -1 \end{bmatrix}, \begin{bmatrix} \frac{-403}{49} & \frac{172}{49} \\ \frac{34}{7} & \frac{-13}{7} \\ \frac{439}{49} & \frac{-187}{49} \end{bmatrix}$$

On peut également utiliser des **méthodes itératives** (pour de grandes matrices creuses) avec un grand nombre de paramètres de contrôle que l'on trouvera dans l'aide en ligne.

```
> ?LinearAlgebra,LinearSolveItr
```

Ces méthodes ne s'appliquent qu'à des matrices avec éléments décimaux et n'accepte pas des seconds

membres multiples

```
> LinearSolve(convert(S,float,18),b[1..-1,1],  
method='SparseIterative');  
[ -8.22448979591835538  
  4.85714285714284965  
  8.95918367346937394 ]
```

Soit

```
> %-evalf[18](x[1..-1,1]);  
[ 1.24344978758017532 10-14  
 -7.10542735760100186 10-15  
 -1.42108547152020037 10-14 ]
```

Il est vivement recommandé de compléter ces informations avec le chapitre 4 et l'aide en ligne.

Exercice : Dire que S est symétrique, c'est évident. Vérifions le

```
> Equal(Transpose(S)-S,Matrix(Dimension(S),0));  
true
```

Mais qu'elle soit *définie positive* ne saute pas aux yeux. Généralement, on connaît a priori la propriété, par exemple si $M = (A^T) \cdot A$, où les coefficients de A sont réels. Ou encore

```
> H:=Matrix(2,2,[[1,-1-I],[-1,2+I]]);  
S_H:=HermitianTranspose(H).H;  
H := [ 1 -1 - I  
      -1 2 + I ]  
S_H := [ 2 -3 - 2 I  
      -3 + 2 I 7 ]
```

Pour qu'une matrice soit *définie positive* il faut et il suffit que ses valeurs propres soient réelles strictement positives. On vérifie ci-dessous que S_H est définie positive. Expliquez ces écritures avec l'aide en ligne

```
> map(is,Eigenvalues(S_H),positive);  
[ true  
  true ]  
  
> IsDefinite(S_H);  
true  
  
> IsDefinite(S,query='negative_semidefinite');  
false
```

15 - Equations différentielles

Ce chapitre ne donne qu'un bref aperçu des possibilités de MAPLE dans ce domaine mais il sera peut être suffisant pour une première approche.

Fonction dsolve, équations différentielles et conditions initiales

MAPLE permet de résoudre beaucoup d'équations différentielles à l'aide de l'opérateur **dsolve**. On définit les termes différentiels des équations à l'aide de l'opérateur **diff**. La fonction **dsolve** génère avec la solution les noms des constantes d'intégration sous la forme $_C1, _C2\dots$ On peut donner un nom à l'équation différentielle ainsi qu'à la solution.

On rappelle que s'il est possible à l'utilisateur de créer des noms de variables commençant par un caractère de soulignement ($_$) ceci est très fortement déconseillé et doit être laissé à l'initiative de MAPLE. Néanmoins **dsolve** n'utilisera pas un nom assigné pour désigner une constante d'intégration.

```
> restart;
eqd:=diff(y(x),x)-k*x*y(x) = x: eqd;
sol:=dsolve(eqd);
```

$$\left(\frac{d}{dx} y(x) \right) - k x y(x) = x$$
$$sol := y(x) = -\frac{1}{k} + e^{\left(\frac{1}{2} k x^2 \right)} _{CI}$$

On peut vérifier la réponse à l'aide de la fonction **odetest** qui doit répondre 0 si la solution est juste. Quelquefois il peut être nécessaire d'effectuer quelques simplifications sur la réponse pour obtenir ce résultat.

```
> odetest(sol,eqd);
0
```

La donnée du nom de la fonction inconnue comme argument peut être nécessaire car **dsolve** est capable de résoudre formellement une équation pour des termes contenant des fonctions non explicitées. Les deux réponses suivantes sont différentes car on ne résout pas la même équation, les fonctions solutions étant différentes. On remarque également que l'on peut employer indifféremment **Diff** ou **diff**.

```
> eqd:=Diff(h(x)+y(x),x)-k*x*y(x) = G(x): eqd;
\left( \frac{d}{dx} (h(x) + y(x)) \right) - k x y(x) = G(x)
```

```
> dsolve(eqd,y(x));
y(x) = \int \left( \left( -\left( \frac{d}{dx} h(x) \right) + G(x) \right) e^{\left( -\frac{1}{2} k x^2 \right)} dx + _{CI} \right) e^{\left( \frac{1}{2} k x^2 \right)}
```

```
> dsolve(eqd,h(x));
```

$$h(x) = \int -\left(\frac{d}{dx} y(x) \right) + k x y(x) + G(x) dx + _C1$$

S'il n'y a pas d'ambiguïté, MAPLE choisit pour fonction solution celle sur laquelle s'applique l'opérateur **diff**.

```
> eqd:=diff(h(x),x)-2*x*y(x)=x: eqd;
dsolve(eqd);
```

$$\begin{aligned} & \left(\frac{d}{dx} h(x) \right) - 2 x y(x) = x \\ h(x) &= \int x (2 y(x) + 1) dx + _C1 \end{aligned}$$

Lorsque l'on fixe une ou plusieurs *conditions initiales*, elles forment avec l'équation différentielle un **ensemble d'équations** à résoudre (d'où la mise entre {} de ces équations). MAPLE calcule (si elle existe) la (les) valeur(s) des constante(s) d'intégration

```
> eqd:=diff(y(x),x)-2*x*y(x) = x: eqd;
```

```
c_i:=y(0)=1: c_i;
```

```
sol:=dsolve({eqd,c_i});
```

$$\begin{aligned} & \left(\frac{d}{dx} y(x) \right) - 2 x y(x) = x \\ y(0) &= 1 \end{aligned}$$

$$sol := y(x) = -\frac{1}{2} + \frac{3}{2} e^{(x^2)}$$

```
> odetest(sol,eqd);
```

0

Attention

La fonction y n'existe pas, sinon de façon symbolique. On rappelle qu'une écriture comme $y(x)$ possède le type **function** en raison de sa syntaxe. La réponse de **dsolve** est seulement une **équation**

```
> whattype(y(x)), whattype(sol);
```

function, =

et y n'est pas associée à une fonction explicitée

```
> eval(y);
y(0);
```

y
y(0)

On peut assigner la solution à $y(x)$. Il faut alors comprendre que $y(x)$ se comporte simplement comme un **nom** qui pointe sur l'expression de la solution.

```
> assign(sol);# Cette commande ne renvoie aucun message
y(x);
```

$$-\frac{1}{2} + \frac{3}{2} e^{(x^2)}$$

Mais y n'est toujours pas une fonction...

```
> y(t),y(3);
type(eval(y),operator);
y(t),y(3)
false
```

Maintenant on peut transformer la solution en une fonction en écrivant

```
> y:=unapply(y(x),x);
y := x → - 1/2 + 3/2 e^(x^2)
```

Résumons:

1) on résout l'équation différentielle (après avoir désassigné y) et on teste la validité de la solution

Les étapes suivantes sont facultatives et permettent de transformer la solution en fonction

2) on assigne à $y(x)$ la solution

3) on transforme $y(x)$ en une fonction y

```
> y:='y':
eqd;
sol:=dsolve({eqd,y(0)=1});
odetest(sol,eqd);
assign(sol);
y:=unapply(y(x),x);
```

$$\left(\frac{d}{dx} y(x) \right) - 2x y(x) = x$$

$$sol := y(x) = -\frac{1}{2} + \frac{3}{2} e^{(x^2)}$$

0

$$y := x \rightarrow -\frac{1}{2} + \frac{3}{2} e^{(x^2)}$$

Pour définir les dérivées d'ordres supérieurs on peut utiliser l'opérateur **diff** conformément aux indications données au chapitre 8, *Dérivation*.

```
> y:='y':
eqd:=diff(y(x),x$2)-3*diff(y(x),x)+2*y(x) = 0: eqd;
sol:=dsolve(eqd,y(x));

```

$$\left(\frac{d^2}{dx^2} y(x) \right) - 3 \left(\frac{d}{dx} y(x) \right) + 2 y(x) = 0$$
$$sol := y(x) = _C1 e^{(2x)} + _C2 e^x$$

L'opérateur **D** permet une écriture équivalente

```

> eqD:=(D@@2)(y)(x)+2*D(y)(x)+y(x) = 0: eqD;
sol:=dsolve(eqD,y(x));

```

$$D^{(2)}(y)(x) + 2 D(y)(x) + y(x) = 0$$

$$sol := y(x) = _C1 e^{(-x)} + _C2 e^{(-x)} x$$

On peut d'ailleurs convertir un type d'écriture vers l'autre

```

> convert(eqD,D);
convert(eqD,diff);

```

$$D^{(2)}(y)(x) - 3 D(y)(x) + 2 y(x) = 0$$

$$\left(\frac{d^2}{dx^2} y(x) \right) + 2 \left(\frac{d}{dx} y(x) \right) + y(x) = 0$$

L'opérateur **D** peut être utilisé en même temps que **diff** ou **Diff**. Mais il permet surtout d'introduire des conditions initiales sur les dérivées comme ici $y'(0) = 0$. On notera aussi la possibilité (non nécessaire) d'utiliser l'opérateur **union** pour rassembler l'équation et les conditions initiales.

```

> eqd:=Diff(y(x),x$2)-3*D(y)(x)+2*y(x)=0: eqd;
c_i:={y(0)=1,D(y)(0)=0 }: c_i;
sol:=dsolve({eqd} union c_i,y(x));

```

$$\left(\frac{d^2}{dx^2} y(x) \right) - 3 D(y)(x) + 2 y(x) = 0$$

$$\{y(0) = 1, D(y)(0) = 0\}$$

$$sol := y(x) = -e^{(2 x)} + 2 e^x$$

```

> convert({eqd} union c_i,D);

```

$$\{y(0) = 1, D^{(2)}(y)(x) - 3 D(y)(x) + 2 y(x) = 0, D(y)(0) = 0\}$$

Divers aspects de l'intégration des équations différentielles

Résolution par transformations intégrales

On peut demander à **dsolve** d'appliquer une méthode de résolution par transformations intégrales de Laplace ou de Fourier si l'on sait que l'une de ces méthodes est efficace pour résoudre une équation. Ici l'équation différentielle est à coefficients constants et l'utilisation des transformées de Laplace est bien adaptée à sa résolution.

```

> restart:
eqd_ci:={ diff(y(x),x$2)+4*diff(y(x),x)+8*y(x) = sin(x) ,
           y(0)=1 , D(y)(0)=0 }: eqd_ci;

```

$$\left\{ \left(\frac{d^2}{dx^2} y(x) \right) + 4 \left(\frac{d}{dx} y(x) \right) + 8 y(x) = \sin(x), y(0) = 1, D(y)(0) = 0 \right\}$$

On peut imposer à MAPLE de chercher la solution par cette méthode en utilisant comme *troisième* argument le mot clé **method**. Cette troisième position impose donc de spécifier la fonction inconnue $y(x)$

```
> sol:=dsolve(eqd_ci,y(x),method=laplace);
```

$$sol := y(x) = -\frac{4}{65} \cos(x) + \frac{7}{65} \sin(x) + \frac{1}{130} e^{(-2x)} (138 \cos(2x) + 131 \sin(2x))$$

Fonctions discontinues définissant les équations différentielles

La fonction **dsolve** sait traiter la fonction de Heaviside.

```
> eqd:=diff(y(x),x$2)-2*y(x)=Heaviside(x)*x: eqd;
c_i:={y(0)=0,D(y)(0)=0}: c_i;
sol:=dsolve({eqd} union c_i);
```

$$\left(\frac{d^2}{dx^2} y(x) \right) - 2 y(x) = \text{Heaviside}(x) x$$

$$\{D(y)(0) = 0, y(0) = 0\}$$

$$sol := y(x) = -\frac{1}{8} \text{Heaviside}(x) (4x - \sqrt{2} e^{(\sqrt{2}x)} + \sqrt{2} e^{(-\sqrt{2}x)})$$

Quelques manipulations peuvent être utiles pour éventuellement simplifier l'expression de la solution

```
> sol:=factor(convert(sol,trig));
odetest(sol,eqd);
```

$$sol := y(x) = -\frac{1}{4} \text{Heaviside}(x) (2x - \sinh(\sqrt{2}x) \sqrt{2})$$

0

On montre ici comment on peut se dispenser de la fonction **assign** pour créer la *fonction solution* (**rhs** = right hand side)

```
> y:=unapply(rhs(sol),x);
y := x → -\frac{1}{4} \text{Heaviside}(x) (2x - \sinh(\sqrt{2}x) \sqrt{2})
```

Les fonctions définies par morceaux avec **piecewise** peuvent aussi entrer dans l'écriture d'une équation différentielle.

```
> y:='y':
eqd:=diff(y(x),x)+y(x) =
piecewise(x<=0,1,exp(-x))+k*cos(x): eqd;
```

$$\left(\frac{d}{dx} y(x) \right) + y(x) = \begin{cases} 1 & x \leq 0 \\ e^{(-x)} & otherwise \end{cases} + k \cos(x)$$

```
> sol:=dsolve({eqd,y(0)=1});
```

$$sol := y(x) = \begin{cases} \frac{1}{2} k \cos(x) + \frac{1}{2} k \sin(x) + 1 - \frac{1}{2} e^{(-x)} k & x < 0 \\ \frac{1}{2} k \cos(x) + \frac{1}{2} k \sin(x) - \frac{1}{2} e^{(-x)} k + e^{(-x)} + e^{(-x)} x & 0 \leq x \end{cases}$$

Cette solution peut être exprimée à l'aide de la fonction de Heaviside

```
> sol:=convert(sol,Heaviside);
sol :=  $y(x) = \frac{1}{2} k \cos(x) + \frac{1}{2} k \sin(x) + 1 - \text{Heaviside}(x) - \frac{1}{2} e^{(-x)} k + \text{Heaviside}(x) e^{(-x)} + e^{(-x)} \text{Heaviside}(x) x$ 
```

De façon générale, et si c'est possible, on préférera convertir l'équation elle-même avant de chercher la solution (souvent, c'est plus efficace).

```
> eqd:=convert(eqd,Heaviside);
```

$$eqd := \left(\frac{d}{dx} y(x) \right) + y(x) = k \cos(x) + 1 - \text{Heaviside}(x) + \text{Heaviside}(x) e^{(-x)}$$

```
> sol:=dsolve({eqd,y(0)=1});
```

$$sol := $y(x) = \frac{1}{2} k \cos(x) + \frac{1}{2} k \sin(x) + 1 - \text{Heaviside}(x) - \frac{1}{2} e^{(-x)} k + \text{Heaviside}(x) e^{(-x)} + e^{(-x)} \text{Heaviside}(x) x$$$

Solutions explicites et implicites

Quelquefois la méthode de résolution utilisée par **dsolve** conduit à une forme implicite de la solution, c'est-à-dire une solution sous la forme d'une **équation ordinaire**. L'opérateur **dsolve** tente alors de résoudre cette équation et présente la ou les solutions explicites. Si cette résolution n'est pas possible, seule la forme implicite est renvoyée. On remarquera ici que la solution est sous cette forme: $F(x, y(x), Cte) = 0$. De plus, la solution implicite contient un terme exprimé à l'aide de l'opérateur **Intat** (voir chapitre 10, *Intégration*)

```
> y:='y':
eqd:= diff(y(x),x)=sin(y(x)^2):eqd;
dsolve(eqd);
```

$$\begin{aligned} \frac{d}{dx} y(x) &= \sin(y(x)^2) \\ x - \int^y(x) \frac{1}{\sin(a^2)} da + _C1 &= 0 \end{aligned}$$

La fonction **odetest** fonctionne malgré tout

```
> odetest(%,eqd);
0
```

Pour l'équation suivante, MAPLE renvoie normalement deux solutions

```
> eqd:= diff(y(x),x)=-y(x)+y(x)^3: eqd;
dsolve(eqd); # 2 solutions explicites
```

$$\begin{aligned} \frac{d}{dx} y(x) &= -y(x) + y(x)^3 \\ y(x) &= \frac{1}{\sqrt{1 + e^{(2 x)} _C1}}, y(x) = -\frac{1}{\sqrt{1 + e^{(2 x)} _C1}} \end{aligned}$$

On notera que les solutions étant alors renvoyées sous forme de suite (sequence) on devra former l'**ensemble** des solutions pour **odetest** et on obtiendra un **ensemble** de 2 zéros (qui ne font qu'un; voir chapitre 3, § *Ensembles*)

```
> odetest({%},eqd);
{0}
```

On peut demander à **solve** de donner la solution sous une forme *implicite* avec le mot clé **implicit**.

```
> dsolve(eqd,implicit);
```

$$\frac{1}{y(x)^2} - 1 - e^{(2x)} - CI = 0$$

Les solutions précédentes sont bien les racines de la solution-équation implicite.

```
> solve(% , y(x));
```

$$\frac{1}{\sqrt{1 + e^{(2x)} - CI}}, - \frac{1}{\sqrt{1 + e^{(2x)} - CI}}$$

On ne sera donc pas surpris si les solutions explicites font souvent apparaître la fonction *W* de Lambert rencontrée au chapitre 14, *Equations Ordinaires*

```
> eqd:=(1+x*y(x)+x^2)*diff(y(x),x)+1+x*y(x)+y(x)^2=0: eqd;
dsolve(eqd,implicit);
dsolve(eqd);
```

$$(1 + x y(x) + x^2) \left(\frac{d}{dx} y(x) \right) + 1 + x y(x) + y(x)^2 = 0$$

$$\frac{1}{3} + \frac{1}{3} x y(x) + \frac{1}{3} \ln \left(\frac{9 x (y(x) + x)}{2 (1 + x y(x) + x^2)} \right) - \frac{1}{3} \ln \left(- \frac{9}{1 + x y(x) + x^2} \right) - \frac{1}{3} \ln(x) - CI = 0$$

$$y(x) = \frac{-x^2 + \text{LambertW} \left(-2 x e^{(-1)} e^{(x^2)} - CI \right)}{x}$$

Dans l'exemple suivant la solution explicite est d'un intérêt relatif. La solution implicite montre que la difficulté rencontrée vient de ce que MAPLE ne "connaît" pas de définition pour la fonction réciproque de la fonction de Fresnel-sinus (voir chapitre 10, *Intégration*)

```
> eqd:= diff(y(x),x)=sin(y(x)^2)^(-1): eqd;
dsolve(eqd);
dsolve(eqd,implicit);
```

$$\frac{d}{dx} y(x) = \frac{1}{\sin(y(x)^2)}$$

$$y(x) = \frac{1}{2} \sqrt{2} \sqrt{\pi} \text{RootOf}(-\text{FresnelS}(Z) \sqrt{\pi} + \sqrt{2} x + \sqrt{2} - CI)$$

$$x - \frac{1}{2} \sqrt{2} \sqrt{\pi} \text{FresnelS} \left(\frac{\sqrt{2} y(x)}{\sqrt{\pi}} \right) + CI = 0$$

Classes des équations différentielles

La fonction **odeadvisor** de la bibliothèque **DEtools** peut indiquer à quelle classe appartient une équation différentielle

```
> eqd:= diff(y(x),x)=-x*y(x)+y(x)^3: eqd;
DEtools[odeadvisor](eqd);
```

$$\frac{d}{dx} y(x) = -x y(x) + y(x)^3$$

[*Bernoulli*]

Ici on rencontre une équation du type Riccati mais avec une propriété supplémentaire (voir l'aide en ligne à la rubrique **odeadvisor**).

```
> eqd:=Diff(y(x),x)+(2*x+1)*y(x)+(1+x+x^2)+y(x)^2=0: eqd;
DEtools[odeadvisor](eqd);
```

$$\left(\frac{dy}{dx} \right) + (2x + 1)y + 1 + x + x^2 + y^2 = 0$$

[[_homogeneous, class C], _Riccati]]

Il n'y a pas de méthode générale pour résoudre les équations de Riccati mais **dsolve** trouve la solution de cette équation grâce à la propriété d'homogénéité.

```
> dsolve(eqd);
```

$$y(x) = -\frac{-1 + x e^x - C1 - x}{e^x - C1 - 1}$$

MAPLE "connaît" la plupart des fonctions dites spéciales, ce qui lui permet de résoudre certains types d'équations

```
> eqd:=x^2*diff(y(x),x$2)+x*diff(y(x),x)+(x^2-n^2)*y(x)=0: eqd;
DEtools[odeadvisor](eqd);
dsolve(eqd);
```

$$x^2 \left(\frac{d^2}{dx^2} y(x) \right) + x \left(\frac{dy}{dx} \right) + (x^2 - n^2) y(x) = 0$$

[_Bessel]]

$$y(x) = _C1 \operatorname{BesselJ}(n, x) + _C2 \operatorname{BesselY}(n, x)$$

```
> eqd:=Diff(y(x),x$2)=3*cos(2*x)*y(x)-y(x): eqd;
c_i:={y(0)=1,D(y)(0)=0}: c_i;
DEtools[odeadvisor](eqd);
dsolve({eqd,op(c_i)},y(x));
```

$$\frac{d^2}{dx^2} y(x) = 3 \cos(2x) y(x) - y(x)$$

$$\{y(0) = 1, D(y)(0) = 0\}$$

[_ellipsoidal]]

$$y(x) = \operatorname{MathieuC}\left(1, \frac{3}{2}, x \right)$$

Enfin, **dsolve** ne sera pas toujours capable de donner une solution (souvent parce qu'elle ne s'exprime pas de façon symbolique et exacte). La réponse est alors soit **NULL**, soit une ré-écriture de l'équation avec la fonction **DESol** (cette fonction est aux équations différentielles ce que **RootOf** est aux équations ordinaires). L'apparente simplicité d'une équation n'est pas un gage sur la possibilité d'expliciter la solution... (ici une équation voisine du *premier transcendant de Painlevé*; remplacer $y(x)^2$ par $6 y(x)^2$)

```
> eqd:=Diff(y(x),x$2)=y(x)^2+x: eqd;
DEtools[odeadvisor](eqd);
```

```

sol:=dsolve(eqd);

$$\frac{d^2}{dx^2}y(x) = y(x)^2 + x$$


```

[NONE]

sol :=

Développement en série de la solution

Lorsque MAPLE ne trouve pas de solution ou si la solution est trop compliquée, on peut lui demander un développement en série autour des conditions initiales avec le mot clé **series**. L'ordre du développement dépend comme pour la fonction **series** de la variable globale **Order** (6 par défaut et que l'on a changé ici en 5). Le mot clé **series** doit être en **troisième position** ce qui nécessite d'expliciter la fonction inconnue.

```
> eqd:=D(y)(x)+(2*x+1)*y(x)+(1+x+x^2)*y(x)^3=0: eqd;
```

```
C_init:=y(1)=1: C_init;
```

```
Order:=5;
```

```
sol:=dsolve({eqd,C_init},y(x),series);
```

$$D(y)(x) + (2x + 1)y(x) + (1 + x + x^2)y(x)^3 = 0$$

$$y(1) = 1$$

$$Order := 5$$

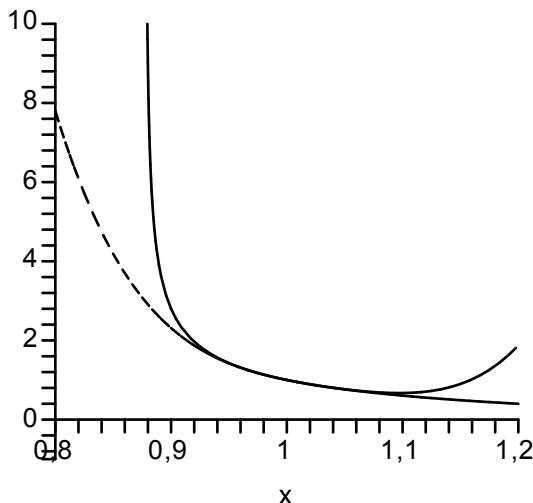
$$sol := y(x) = \left(1 - 6(-1 + x) + \frac{67}{2}(-1 + x)^2 - \frac{661}{3}(-1 + x)^3 + \frac{12471}{8}(-1 + x)^4 + O((-1 + x)^5) \right)$$

Voici comment transformer ce type de solution en une fonction, solution approchée au voisinage de la condition initiale

```
> y_ap:=unapply(convert(rhs(sol),polynom),x);
```

$$y_{ap} := x \rightarrow 7 - 6x + \frac{67}{2}(x - 1)^2 - \frac{661}{3}(x - 1)^3 + \frac{12471}{8}(x - 1)^4$$

L'équation précédente peut être résolue exactement par **dsolve**. Ceci nous permet de montrer un tracé de la solution exacte (courbe pleine) et de son approximation par une série autour de $x = 1$ (courbe en pointillés).



Équations linéaires homogènes à coefficients polynomiaux

La solution explicite de ce type d'équation, comme le lecteur pourra le tester avec cet exemple, peut être

compliquée et difficilement exploitable...

```
> eqd:=(x^3+x)*diff(y(x),x$3)+(x^2-1)*diff(y(x),x$2)-y(x)=0:
eqd;
```

$$(x^3 + x) \left(\frac{d^3}{dx^3} y(x) \right) + (x^2 - 1) \left(\frac{d^2}{dx^2} y(x) \right) - y(x) = 0$$

Avec l'option **formal_solution** on peut obtenir l'expression de la solution sous forme de série entière

```
> dsolve({eqd,y(0)=0},y(x),formal_solution);
```

$$y(x) = _C1 \left(\sum_{n=1}^{\infty} (-1)^{(n+1)} 4^{(1-n)} \left(\prod_{l=1}^{n-1} \frac{-1 + 4l^2 + 8l^3}{l(3 + 5l + 2l^2)} \right) x^{(1+2n)} \right)$$

Système d'équations différentielles

MAPLE peut aussi résoudre des systèmes d'équations différentielles.

```
> eqd:={diff(x(t),t)-x(t)+2*y(t)=0,diff(y(t),t)+x(t)-2*y(t)=0}: eqd;
c_init:={x(0)=0,y(1)=1}: c_init;
fct:={x(t),y(t)}: fct;
```

$$\left\{ \left(\frac{d}{dt} y(t) \right) + x(t) - 2y(t) = 0, \left(\frac{d}{dt} x(t) \right) - x(t) + 2y(t) = 0 \right\}$$

$$\{y(1) = 1, x(0) = 0\}$$

$$\{x(t), y(t)\}$$

```
> sol:=dsolve(eqd union c_init,fct);
```

$$sol := \left\{ y(t) = \frac{2e^{(3t)}}{2e^3 + 1} + \frac{1}{2e^3 + 1}, x(t) = \frac{2}{2e^3 + 1} - \frac{2e^{(3t)}}{2e^3 + 1} \right\}$$

```
> odetest(sol,eqd);
```

$$\{0\}$$

On peut maintenant construire les *fonctions solutions*

```
> xf:=unapply(subs(sol,x(t)),t);
yf:=unapply(subs(sol,y(t)),t);
```

$$xf := t \rightarrow \frac{2}{2e^3 + 1} - \frac{2e^{(3t)}}{2e^3 + 1}$$

$$yf := t \rightarrow \frac{2e^{(3t)}}{2e^3 + 1} + \frac{1}{2e^3 + 1}$$

Exercice : Calculez la solution de ce système d'équations différentielles

$$\begin{cases} \left(\frac{d}{dt} x(t) \right) - t x(t) + 2 y(t) = 0 \\ \left(\frac{d}{dt} y(t) \right) + x(t) - 2 t y(t) = 0 \end{cases}$$

avec les conditions initiales $x(0) = 1$ et $y(1) = 0$. Vérifier ensuite la solution.

Résolutions numériques

Lorsque MAPLE ne trouve pas de solutions exactes, il est possible de trouver une solution numérique en utilisant le mot clé **numeric**. MAPLE détectera la nature du problème posé, c'est-à-dire avec **conditions initiales** ou **conditions aux limites** afin de choisir un algorithme adapté. Pour de telles résolutions les équations ne doivent pas dépendre de paramètres non définis et toutes les conditions (initiales ou aux limites) qui fixent la solution doivent être données.

> **restart:**

On construit une équation différentielle avec **conditions initiales** au point $x = 0$ (IVP pour *initial value problem*)

```
> eqd:=diff(f(x),x$2)=(x-f(x))/(1+x^2+f(x)^2): eqd;
c_i:=f(0)=1,D(f)(0)=-1: c_i;
```

$$\frac{d^2}{dx^2} f(x) = \frac{x - f(x)}{1 + x^2 + f(x)^2}$$

$$f(0) = 1, D(f)(0) = -1$$

dont l'intégrateur **dsolve** ne trouve pas de solution exacte

```
> sol:=dsolve({eqd,c_i},f(x));
sol :=
```

Avec le mot clé **numeric**, **dsolve** crée une procédure d'intégration numérique en utilisant, *par défaut*, une méthode du type *Runge-Kutta-Fehlberg* d'ordre 4 (rkf).

```
> sol:=dsolve({eqd,c_i},f(x),numeric);
sol := proc(x_rkf45) ... end proc;
```

On construit maintenant, avec la même équation, un problème aux **conditions aux limites** en $x = 0$ et $x = 1$ (dit encore "aux bornes"; BVP pour *boundary value problem*)

```
> c_1:=f(0)=1,f(1)=1: c_1;
f(0) = 1, f(1) = 1
```

dsolve détecte la nature du problème posé, choisit un algorithme adapté (bvp) et construit de même une procédure

```
> sol_1:=dsolve({eqd,c_1},f(x),numeric);
sol_1 := proc(x_bvp) ... end proc;
```

Attention

Il existe un très grand nombre de possibilités et d'options, tant pour les problèmes aux conditions initiales

que pour les problèmes aux limites, que nous ne présenterons pas mais que l'on pourra explorer avec l'aide en ligne

> **?dsolve, numeric**

On donne comme exemple un problème aux conditions initiales avec une équation déclarée "raide" (stiff) avec l'option **stiff=true**. Ces équations (ou systèmes d'équations comme par exemple ceux des équilibres chimiques) sont caractérisées par des coefficients ayant des ordres de grandeur relatifs importants. On laisse **dsolve** choisir la méthode (Rosenbrock), mais on pourrait l'imposer, etc.

```
> eqd_stiff:=diff(f(t),t$2)+1.0e3*diff(f(t),t)+1.0e-2*f(t)^2=0;
sol_stiff:=dsolve({eqd_stiff,f(0)=1,D(f)(0)=0},f(t),
numeric,stiff=true,range=0..10,relerr=1.0e-6);
eqd_stiff:=
$$\left(\frac{d^2}{dt^2}f(t)\right) + 1000\left(\frac{d}{dt}f(t)\right) + 0.010f(t)^2 = 0$$

sol_stiff:=proc(x_rosenbrock) ... end proc;
```

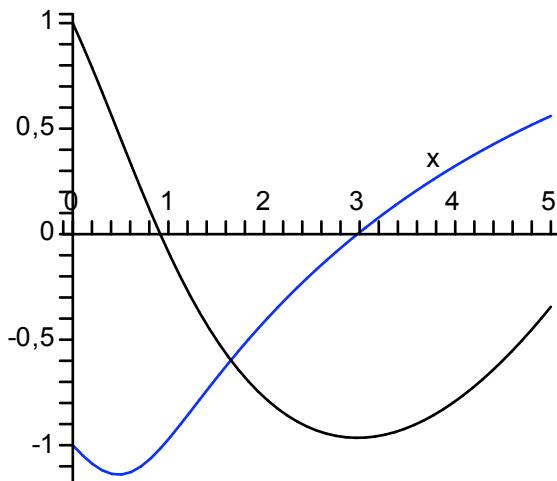
On peut utiliser la procédure comme une fonction qui donne comme résultat un liste d'équations. On reprend la solution du premier problème du paragraphe

> **sol(1.5);**

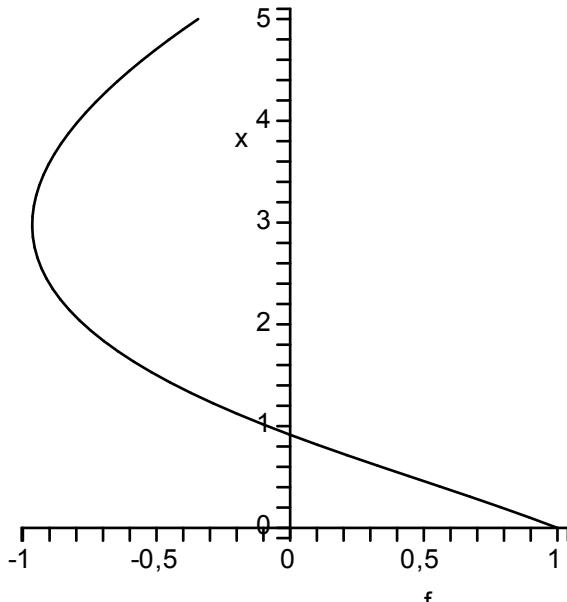
$$\left[x = 1.5, f(x) = -0.498964712481904493, \frac{d}{dx}f(x) = -0.683454657349700567 \right]$$

La solution pourra être visualisée graphiquement avec la fonction **odeplot** (pour *ordinary differential equation plot*) de la bibliothèque graphique **plots**. Le premier argument est le nom de la procédure, le deuxième une liste qui donne la ou les représentations choisies ($f(x)$ en fonction de x ou x en fonction de $f(x)$). Le troisième argument donne l'intervalle des valeurs choisies pour x (pour plus de détails voir les chapitres 16 et 17, *Graphisme 2D et 3D*).

```
> plots[odeplot](sol,[[x,f(x),color=black],
[x,diff(f(x),x),color=blue]],0..5);
```



```
> plots[odeplot](sol,[f(x),x],0..5,color=black);
```



Création d'une procédure (fonction) solution

La fonction **dsolve(...,numeric)** possède l'option **output=listprocedure** qui renvoie une liste de procédures

```
> Ls:=dsolve({eqd,c_i},f(x),numeric,output=listprocedure);
```

$$Ls := \left[x = \text{proc}(x) \dots \text{end proc}; f(x) = \text{proc}(x) \dots \text{end proc}; \frac{d}{dx} f(x) = \text{proc}(x) \dots \text{end proc} \right]$$

On peut associer à des noms ces procédures d'intégration numérique (la fonction **subs** admet des ensembles de substitutions mais aussi des listes).

```
> f_n :=subs(Ls,f(x));
  fp_n:=subs(Ls,diff(f(x),x));
```

Ces noms s'utilisent alors comme des fonctions

```
> f_n(2),fp_n(2);
-0.772248842748144315, -0.416113275412975158
```

Par exemple, si on cherche une approximation de l'abscisse du minimum de la solution (voir chapitre 14, *Equations ordinaires*)...

```
> fsolve(fp_n(x)=0,x=2..4);
2.976436817
```

On peut aussi écrire pour construire les fonctions solutions

```
> f_n :=rhs(Ls[2]):
  fp_n:=rhs(Ls[3]):
```

Exercice : qu'elle est l'origine de cette erreur ? (le message est trompeur)

```
> z:=dsolve(diff(z(x),x)+z(x)^2=sin(x),z(x),
    numeric,output=listprocedure);
```

```
Error, (in dsolve/numeric/process_input) missing differential equations in the first
```

```
argument: (diff(z(x), x))+z(x)^2 = sin(x)
```

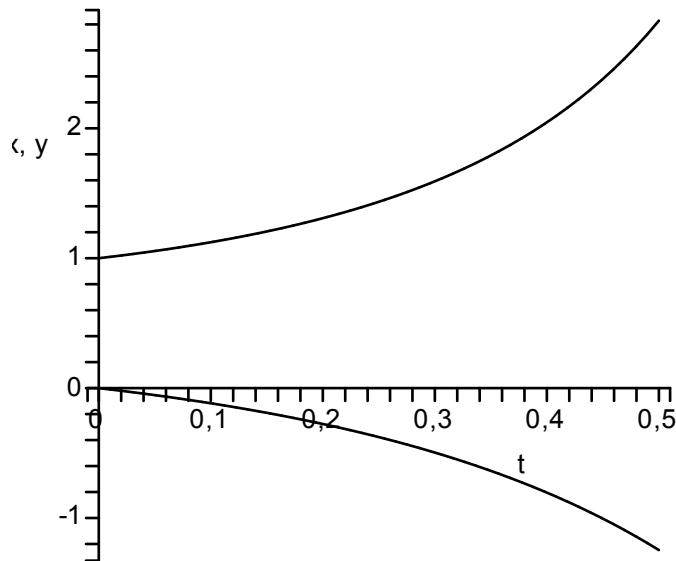
Systèmes d'équations

La fonction **dsolve** permet de traiter de la même façon les *systèmes d'équations* (avec conditions initiales ou aux bornes).

```
> eqnl:={D(x)(t)-x(t)^2+2*y(t)=0,D(y)(t)+x(t)-2*y(t)=0,  
x(0)=1,y(0)=0};  
sol:=dsolve(eqnl,{x(t),y(t)},numeric);  
eqnl := {y(0) = 0, D(x)(t) - x(t)2 + 2 y(t) = 0, D(y)(t) + x(t) - 2 y(t) = 0, x(0) = 1}  
sol := proc(x_rkf45) ... end proc;  
  
> sol(0);sol(0.1);sol(0.5);  
[t = 0., x(t) = 1., y(t) = 0.]  
[t = 0.1, x(t) = 1.12302576462700942, y(t) = -0.116830127033904107]  
[t = 0.5, x(t) = 2.82877723007528603, y(t) = -1.24673998693012034]
```

Ayant cette solution on obtiendra une représentation des fonctions $x(t)$ et $y(t)$ avec **odeplot**

```
> plots[odeplot](sol,[[t,x(t)],[t,y(t)]],0..0.5,color=black);
```



On peut, comme pour une équation simple, obtenir les approximations numériques des solutions sous forme de fonctions

```
> sol:=dsolve(eqnl,{x(t),y(t)},numeric,output=listprocedure):  
xf:=subs(sol,x(t));  
yf:=subs(sol,y(t));  
xf(0.5), yf(0.5);  
  
xf := proc(t) ... end proc;  
yf := proc(t) ... end proc;  
1.09036467197321628, 0.396141928490851779
```

Conclusion

Répétons-le, ce court chapitre ne donne qu'un très bref aperçu des possibilités de MAPLE dans ce domaine (voir aussi chapitre 16, § *Représentations graphiques des solutions d'équations différentielles*). Pour une étude plus approfondie on devra explorer l'aide en ligne et notamment celui de la bibliothèque **DEtools**. Elle contient de nombreuses fonctions de manipulations, changement de variables, portraits de phases, coupes de Poincaré, études Hamiltoniennes, etc.

16 - Représentations graphiques 2D

MAPLE permet de tracer les représentations graphiques cartésiennes, polaires, paramétriques, etc. de fonctions ou d'ensemble de fonctions dans un plan et que l'on a coutume d'appeler "graphisme 2D".

Les possibilités graphiques de MAPLE sont très nombreuses et extrêmement utiles mais ce logiciel ne peut pas se comparer avec d'autres, spécialisés dans ce type de traitements, quant à la qualité des graphiques. Les possibilités offrent avant tout un support intégré de visualisation des objets mathématiques.

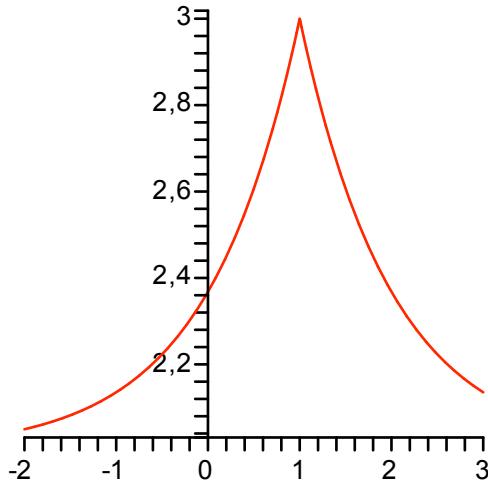
Les dessins s'affichent par défaut sur la feuille de calcul, mais on peut les afficher sur des fenêtres séparées avec le menu *MAPLE xx/Preferences...* et l'onglet *Display* option *Plot display*.

On pourra aussi, comme nous le verrons, créer des fichiers de dessins, par exemple au format *postscript*.

Représentations cartésiennes

Pour tracer la représentation cartésienne d'une *expression* on utilise la fonction **plot** dont le premier argument est cette expression qui ne doit naturellement *dépendre que d'une variable non assignée et d'une seule*. Le deuxième argument est le nom de cette variable considérée comme l'abscisse avec son intervalle de variation souhaité. On peut remarquer que l'axe des ordonnées s'adapte à la représentation graphique et n'est pas nécessairement orthonormé avec celui des abscisses. Le croisement des axes n'est pas non plus nécessairement situé sur le point de coordonnées (0,0).

```
> restart: y:=2:  
plot(exp(-abs(x-1))+y,x=-2..3);
```



En raison de l'évaluation des arguments, l'expression peut aussi être assignée à un nom et la commande précédente deviendrait par exemple

```
> e:=exp(-abs(x-1))+y;  
> plot(e,x=-2..3);
```

Modifications interactives du dessin

Quand on "clique" avec le bouton gauche de la souris sur le graphique celui-ci est entouré par un cadre avec des petits carrés. Ce sont des "poignées" virtuelles qui peuvent être manipulées avec la souris en maintenant le bouton enfoncé et en opérant un glissement. Ceci permet de *modifier la taille et la proportion*

du cadre dans lequel **plot** replace automatiquement le dessin. La ré-exécution de la commande du tracé, même modifiée, conserve la proportion et la taille du cadre qui ont été définies.

Cliquer sur le dessin change aussi la barre des menus et on pourra y trouver différentes opérations: tracés ou non des axes, orthonormés ou non, représentation des courbes sous forme des points calculés par **plot** ou sous forme de lignes continues, etc.). Ces opérations peuvent, comme nous le verrons, être spécifiées directement dans les commandes. Le bouton carré à droite de ce menu permet trois fonctions **avec le mouvement de la souris**

- **Point Probe** est le mode par défaut qui affiche (en haut à gauche) les coordonnées du pointeur dans la représentation graphique.

- **Scale** permet de changer l'échelle du dessin dans le cadre défini.

- **Pan** : permet de changer la position du dessin dans le cadre.

Tracés de fonctions définies par un opérateur

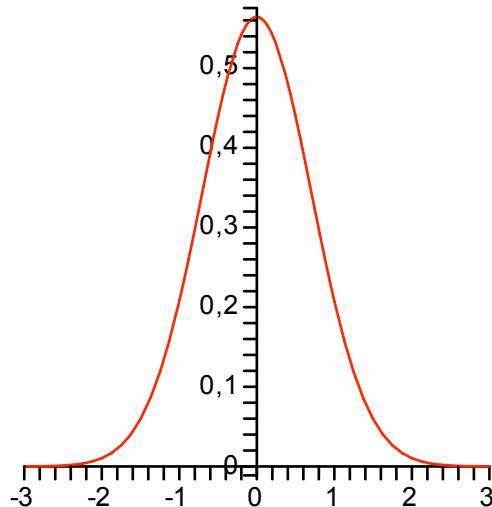
Pour tracer une **fonction définie par un opérateur**, par exemple f , on peut procéder de deux façons

1) En donnant l'expression de cette fonction sous la forme $f(x)$, ce qui nous ramène au cas précédent.

```
> f:=x->exp(-x^2)/sqrt(Pi):  
plot(f(x),x=-3..3); # ou bien
```

2) En donnant le nom de la fonction et en ne précisant pas de nom pour l'abscisse mais seulement son intervalle de variation

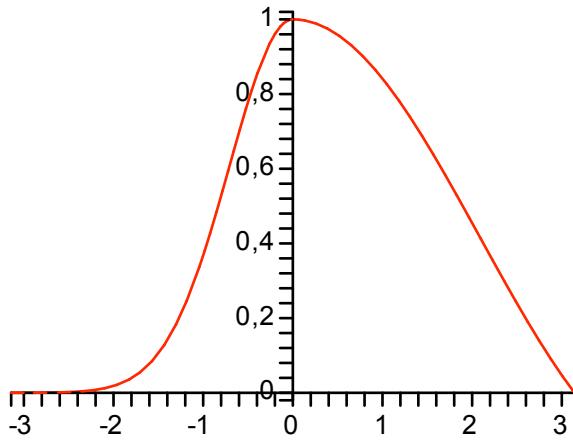
```
> plot(f,-3..3);
```



Pour tracer une **fonction définie par une procédure** on peut utiliser deux syntaxes dont une qui n'utilise pas de nom de variable. **Attention**: pour l'autre syntaxe, ne pas oublier de retarder l'évaluation en utilisant '...'.

Pour les **procédure à plusieurs variables**, voir plus loin dans ce paragraphe

```
> f:=proc(x) if x<=0 then exp(-x^2) else sin(x)/x fi end:  
# plot('f(x)',x=-Pi..Pi);  
plot(f,-Pi..Pi);
```



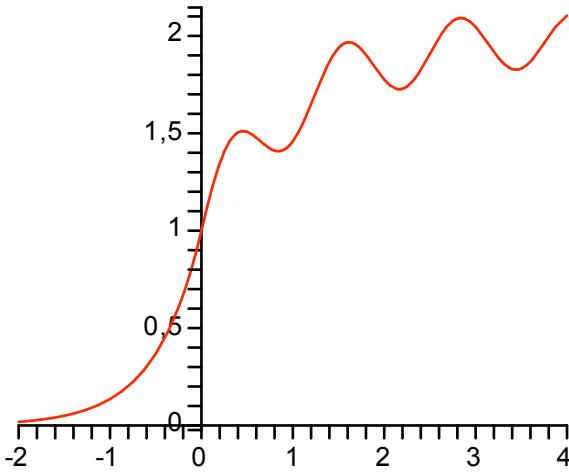
Attention: ces syntaxes ne sont pas correctes, mais comparer avec l'exemple suivant

```
> plot(f(x), -1..1);
Error, (in f) cannot determine if this expression is true or false: x <= 0
```

```
> plot(f, x=-1..1):
Error, (in plot) invalid plotting of procedures, perhaps you mean plot(f, -1 .. 1)
```

On peut également tracer une *expression ou une fonction définie par intervalles*. On notera que f est maintenant le nom d'une *expression*, pas d'une fonction ou d'une procédure, d'où la syntaxe correcte de la fonction **plot**.

```
> f:=piecewise(x<=0, exp(2*x),
                 1+(1-exp(-x))+0.2*sin(5*x)*exp(-0.1*x)):
plot(f, x=-2..4);
```



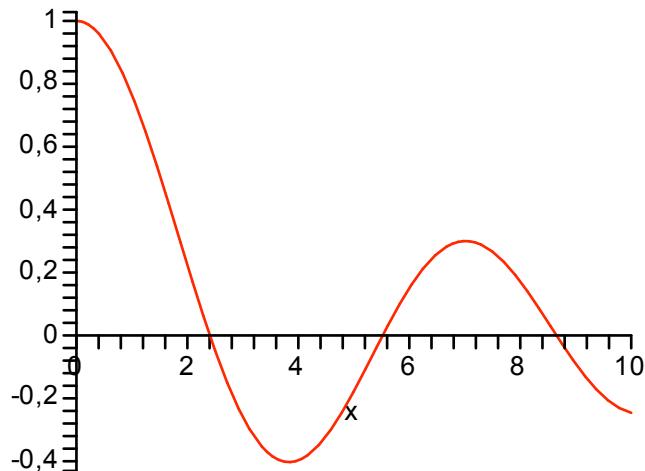
Si la fonction dépend de plusieurs variables et pour des raisons logiques évidentes, seule la forme utilisant l'expression de la fonction peut être utilisée en laissant une seule variable indéterminée et en donnant des valeurs numériques aux autres. La variable n n'étant pas ici assignée à une valeur numérique, **plot** ne peut évidemment pas représenter la fonction

```
> plot(BesselJ(n,x), x=0..10);
Warning, unable to evaluate the function to numeric values in the region; see the
plotting command's help page to ensure the calling sequence is correct
```

```
Error, empty plot
```

Maintenant tout est correct :

```
> plot(BesselJ(0,x),x=0..10);
```



Si on doit tracer une *procédure à plusieurs variables*, disons $f(x, y, a)$ on utilisera la syntaxe suivante

```
> plot('f(1/3,z,0.5)',z=-Pi..Pi);
```

Attention aux variables préalablement assignées. *La variable décrivant l'abscisse doit être libre*

```
> e:=exp(-x^2)-1;
```

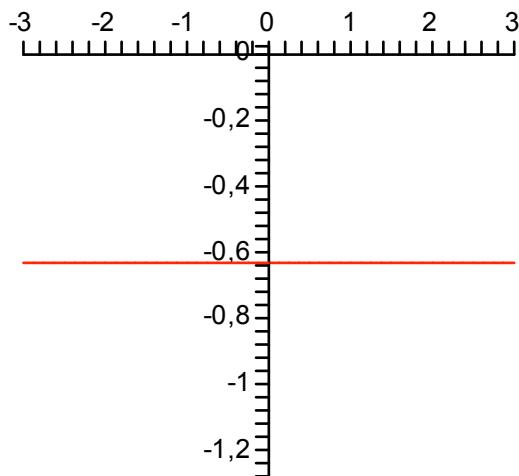
$$e := e^{(-x^2)} - 1$$

```
> x:=1:  
plot(e,x=-3..3);
```

```
Error, (in plot) invalid arguments
```

Exercice : expliquez ce résultat et corrigez

```
> plot(e,'x'=-3..3);
```



Exemples des options de plot

On n'a pas voulu donner ici un catalogue exhaustif des options. Seules les principales sont classées par paragraphe, certaines autres apparaîtront au fur et à mesure des exemples. On pourra toujours les explorer avec l'aide en ligne.

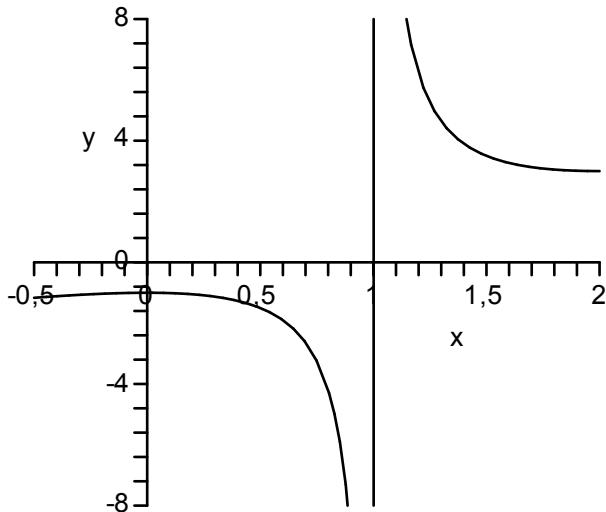
La couleur

On peut imposer une couleur à la courbe avec l'option **color=...** (voir l'exemple suivant; voir également le paragraphe relatif aux tracés multiples). Les couleurs de base sont au nombre de 25 et on les trouvera dans l'aide en ligne **?plot,color** (ex: **black, green, red**, etc.). On peut également utiliser d'autres modes plus sophistiqués avec des fonctions de couleurs.

Intervalle des ordonnées

On peut souhaiter fixer l'intervalle des ordonnées plutôt que de laisser faire **plot**. Par exemple, autour d'une singularité le graphique devient illisible car **plot** adapte automatiquement l'échelle des ordonnées à ces valeurs. On fixe alors l'intervalle des variations autorisées par un **troisième** argument. Si cette option est donnée sous la forme **nom=intervalle** le nom sera affiché pour les ordonnées. Les autres options doivent être placées après .

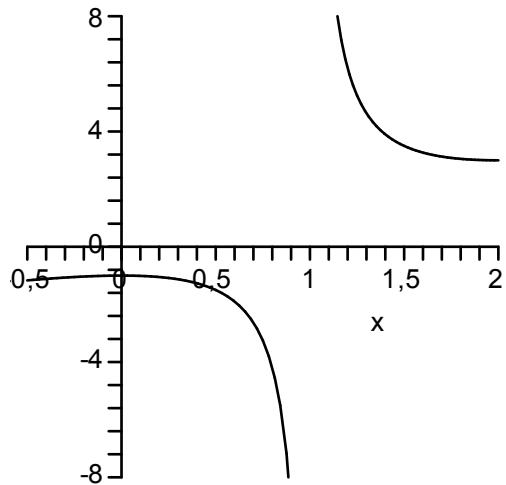
```
> restart:  
plot(x+1/(x-1),x=-0.5..2,y=-8..8,color=black);
```



Discontinuités

La fonction **plot** aurait-elle tracé elle-même l'asymptote ? Pour tracer une représentation graphique elle calcule un ensemble de points qu'elle interpole par des segments de droites. On peut visualiser ces points en cliquant sur le graphique et en "appuyant" sur le bouton marqué par une sinusoidale en pointillés qui apparaît dans le nouveau menu. Le trait "vertical" précédent n'est que celui qui joint les deux points extrêmes calculés pour les deux branches asymptotiques de la fonction. On peut demander à **plot** (d'essayer) de traiter les discontinuités d'une fonction avec l'option **discont=true**.

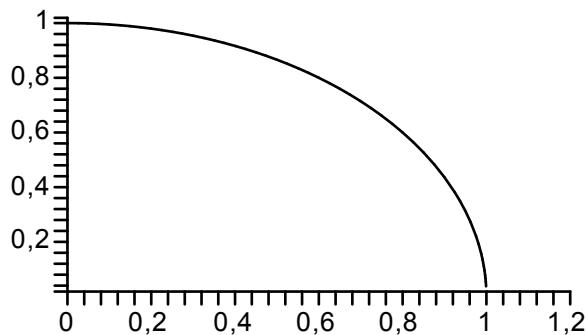
```
> plot(x+1/(x-1),x=-0.5..2,-8..8,discont=true,color=black);
```



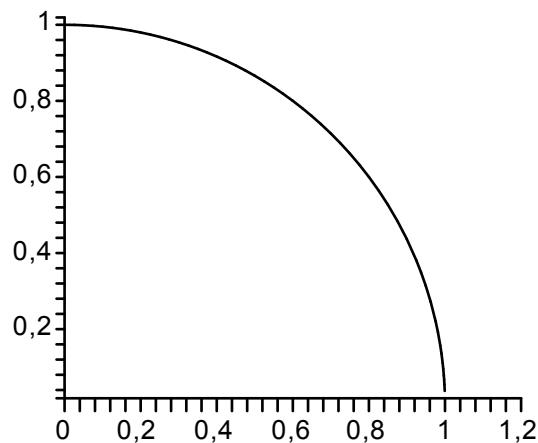
Système d'axes orthonormés

La fonction **plot** adapte l'échelle du dessin en fonction de l'amplitude des valeurs prises par les ordonnées pour donner une représentation graphique apparaissant dans un rectangle aux dimensions standard. Les axes sont alors seulement orthogonaux. On peut imposer que les axes soient aussi orthonormés avec l'option **scaling=constrained**. Cette option peut aussi être réalisée en cliquant sur le graphique et en appuyant sur le bouton marqué **1:1** dans le menu qui apparaît. Notez dans l'exemple suivant que x parcourt un intervalle où les valeurs réelles de la fonction ne sont pas toujours définies ($x > 1$) mais **plot** ne signale aucun message et rejette simplement les points.

```
> plot(sqrt(1-x^2),x=0..1.2,color=black);#On espérait un quart de cercle
```



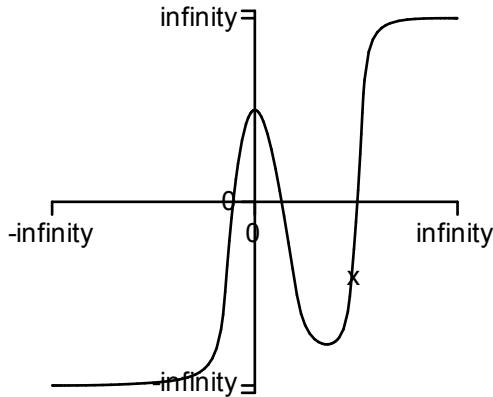
```
> plot(sqrt(1-x^2),x=0..1.2,scaling=constrained,color=black);
```



Axes infinis

Si on précise un intervalle de tracé infini (de la forme $a \dots \infty$ ou $-\infty \dots b$ ou $-\infty \dots \infty$), on obtient un tracé pour lequel les abscisses et les ordonnées reçoivent une transformation anamorphique par un **arctan**. Ceci permet de se faire une idée du comportement général de la fonction.

```
> plot(x^3-4*x^2+4*sin(x)/x, x=-infinity..infinity, color=black);
```



Précision du tracé

La fonction **plot** adapte le nombre de points calculés pour définir le tracé de façon à représenter au mieux la courbe (option **adaptive=true** ; option par défaut). On peut aussi fixer le nombre n minimal de points utilisés (50 par défaut) avec l'option **numpoints=n**.

Tracés de données discrètes

On peut obtenir le tracé d'une fonction à partir d'une liste de valeurs des coordonnées. Celles-ci seront données en écrivant les couples $(x_1, y_1), (x_2, y_2), \dots$ sous la forme d'une liste de listes $[[x_1, y_1], [x_2, y_2], \dots]$. On montre ici comment obtenir, à partir de deux listes contenant séparément les abscisses et les ordonnées, une liste de listes adaptée pour un tracé (voir la fonction **zip** à la fin du chapitre 7, *Fonctions*).

```
> x:=[0.6,0.8,1,1.5,2,2.5]:  
y:=[3,2,2,2.6,3,3]:  
Lxy:=zip((x,y)->[x,y],x,y);
```

$$Lxy := [[0.6, 3], [0.8, 2], [1, 2], [1.5, 2.6], [2, 3], [2.5, 3]]$$

On peut bien sûr extraire ces valeurs de **fichiers de données**. On dispose par exemple d'un fichier *XY.dat* contenant les mêmes données que ci-dessus et organisé en colonnes

0.6	3.0
0.8	2
...	
2.5	3.0

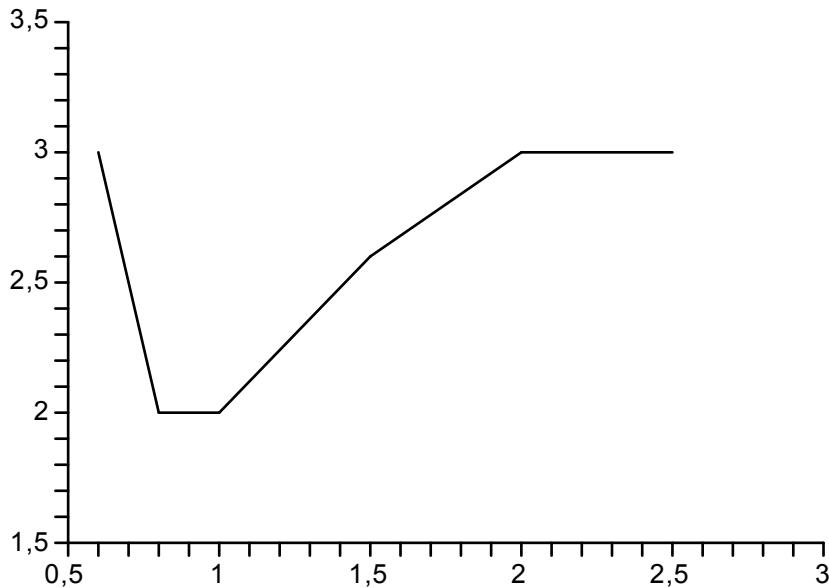
On écrira alors (voir explications et détails au chapitre 20, *Lecture, écriture et fichiers*).

```
> Fic:=cat(kernelopts(homedir), "/Desktop/XY.dat"):  
Lxy:=readdata(Fic,[float$2]);  
Lxy := [[0.6, 3.], [0.8, 2.], [1., 2.], [1.5, 2.6], [2., 3.], [2.5, 3.]]
```

On notera dans la commande suivante, la donnée (facultative) des intervalles qui obligent **plot** à ne pas fixer

implicitement le tracé des axes sur les intervalles des abscisses et ordonnées de la liste. Le tracé continu (par défaut; voir exemple suivant) est obtenu par une interpolation avec des segments de droites.

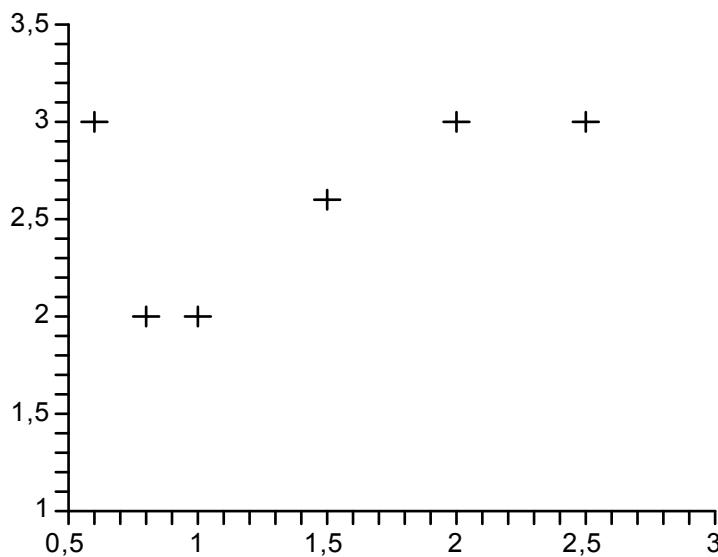
```
> plot(Lxy,0.5..3,1.5..3.5,color=black);
```



Modification du mode de tracé

On peut souhaiter tracer les données précédentes avec un marqueur pour chaque coordonnée. Ceci s'obtient en demandant à **plot** d'utiliser le style **point**. On peut changer la forme des marqueurs avec l'option **symbol**, les options possibles étant **box**, **cross**, **circle**, **point** (un pixel de l'écran) et **diamond** (le défaut). La taille des symboles peut être fixée avec l'option **symbolsize**. Notez que cette taille est définie en unité "points", qui vaut généralement 1/72 d'inch à l'écran et 1/300 d'inch (ou autres) sur une imprimante. Ceci génère des tailles de symboles différentes à l'écran et à l'impression

```
> plot(Lxy,0.5..3,1..3.5,
       style=point,symbol=cross,symbolsize=20,
       color=black);
```



Tracés multiples

On propose maintenant un exemple dans lequel on réalise une interpolation des données précédentes par deux méthodes, une spline et un polynôme (voir le chapitre 7, *Fonctions*, § *Fonctions d'interpolation*). Notez que ces interpolations permettent des extrapolations (plus ou moins heureuses) en dehors de l'intervalle spécifié par les données.

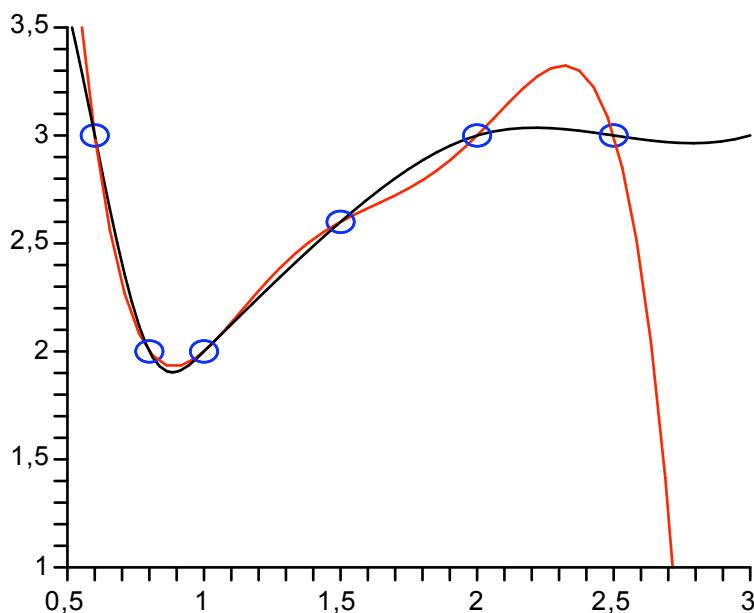
On obtient plusieurs tracés simultanés en demandant à **plot** de tracer la *liste* (ou l'*ensemble*) des expressions. On a donné ici la liste [Lxy (liste de listes contenant les coordonnées discrètes), l'expression de la spline, l'expression du polynôme d'interpolation].

Par défaut, une couleur serait associée à chaque courbe par MAPLE. Mais on peut imposer que chaque courbe prenne une couleur donnée, un style, une épaisseur, etc. en associant une liste pour chaque option.

De façon générale, quand une option s'applique à plusieurs tracés, on lui associe une liste et chaque élément de la liste s'appliquera dans l'ordre à chaque tracé. Ainsi, à Lxy seront associés des petits cercles bleus (point, circle, blue), une courbe (line) noire pour la spline et rouge pour le polynôme. Par contre, l'écriture **option="valeur de l'option"** applique la valeur à tous les tracés (ici, bien sûr, **symbol=circle** et **symbolsize=20** ne concernent que l'option **point**).

```
> plot([ Lxy,
        CurveFitting[Spline](Lxy,x),
        CurveFitting[PolynomialInterpolation](Lxy,x) ],
        x=0.5..3,1..3.5,
        style=[point,line,line],symbol=circle,symbolsize=20,
        color=[blue,black,red]);
```

Attention: nous étudierons dans les prochains paragraphes les tracés de fonctions paramétriques, polaires, etc. On ne peut pas afficher avec cette méthode sur un même graphique, des représentations de natures différentes. On utilisera pour cela la fonction **display** de la bibliothèque **plots** (voir paragraphe *Fonction display de plots*).



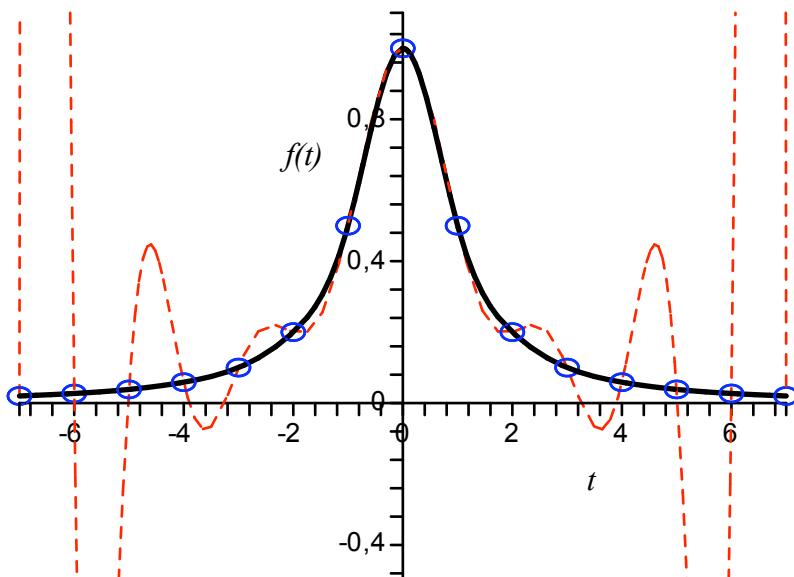
Ajout d'un titre, modification des polices, etc.

L'exemple suivant montre quelques possibilités offertes par les options de **plot** (pour obtenir la liste des options possibles voir l'aide en ligne avec **>?plot,options**). Il illustre le phénomène dit de *Runge* en montrant que l'interpolation polynomiale peut avoir, contrairement à une spline, un comportement "oscillant" pour

certains types de fonctions, ici $f(t) = \frac{1}{1+t^2}$. Notez les chaînes de caractères qui définissent les titres et légendes

```
> Lxy:=[seq([n,1/(1+n^2)],n=-7..7)]:
> plot([Lxy,
       CurveFitting[Spline](Lxy,x),
       CurveFitting[PolynomialInterpolation](Lxy,x)],
       x=-7..7,-0.5..1.1,
       color=[blue,black,red],
       style=[point,line,line],symbol=circle,symbolsize=15,
       linestyle=[1,1,3],thickness=[0,2,0],
       titlefont=[TIMES,BOLDITALIC,16],title="Interpolation",
       labelfont=[TIMES,ITALIC,12],labels=["t","f(t)"]);
```

Interpolation

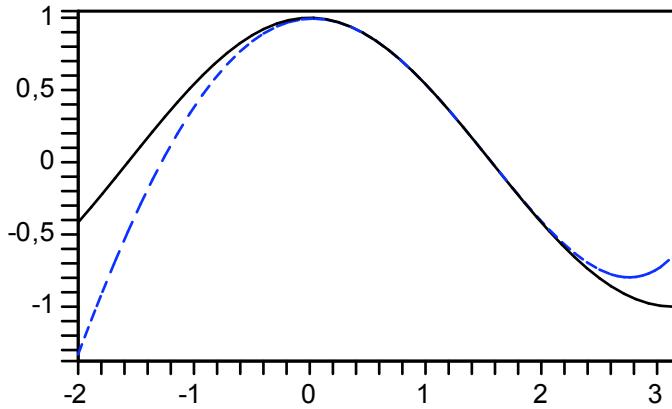


Modification du style des axes

A l'aide de l'option **axes=...** on peut modifier le style des axes. Les options sont **normal** (le défaut), **frame** (les axes sont reportés à gauche et en bas du graphique), **boxed** (les axes forment un cadre) et **none** pour ne pas tracer d'axes. Ces options peuvent être activées par les boutons du menu qui apparaît lorsque l'on clique sur le graphique.

Exercice : Préciser ce que l'on a représenté ici ?

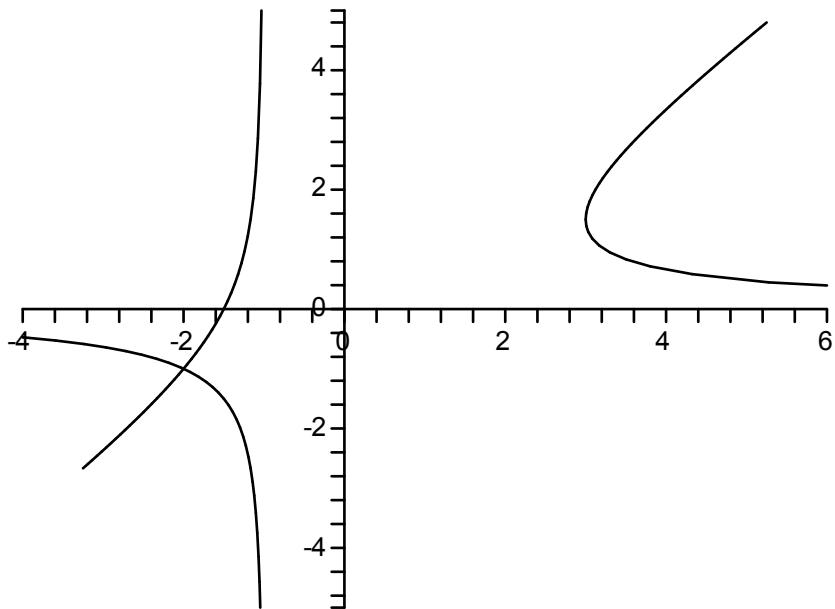
```
> plot([cos(x),convert(series(cos(x),x=1,5),polynom)],
      x=-2..Pi,
      color=[black,blue],linestyle=[1,3],
      axes=boxed);
```



Courbes paramétriques

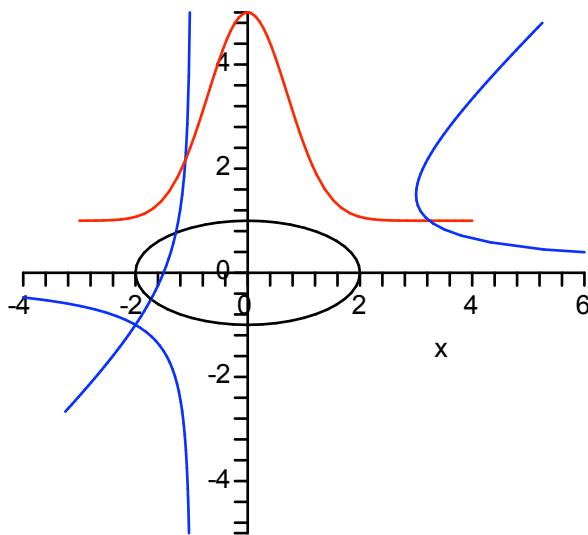
MAPLE permet de tracer simplement des courbes en *coordonnées paramétriques* toujours à l'aide de la fonction **plot**. Il suffit de remplacer l'expression et les variations de la variable par une liste de la forme $[x(t), y(t), t = a .. b]$. Ici t est le paramètre et $a .. b$ son intervalle de variation, $x(t)$ l'expression décrivant les variations de l'abscisse en fonction du paramètre et $y(t)$ celles de l'ordonnée. On peut également imposer les intervalles de variation en abscisse et ordonnée avec l'option **view=[x_min..x_max,y_min..y_max]**. Ces valeurs définiront les limites de la représentation graphique.

```
> plot([(t^3+1)/(t^2-1),t-1/t,t=-3..5],
       view=[-4..6,-5..5],discont=true,color=black);
```



On peut tracer plusieurs tracés paramétriques en créant une liste de ces représentations. Les tracés paramétriques **ne sont pas incompatibles** avec les tracés de la forme $y = f(x)$ (ici $1+4\exp(-x^2)$). Notez que pour tracer cette dernière, la donnée de l'intervalle en abscisse devient nécessaire ($x = -3 .. 4$).

```
> plot([ 1+4*exp(-x^2),
        [(t^3+1)/(t^2-1),t-1/t,t=-3..5],
        [2*cos(t),sin(t),t=0..2*Pi] ],x=-3..4,
       view=[-4..6,-5..5],discont=true,color=[red,blue,black]);
```



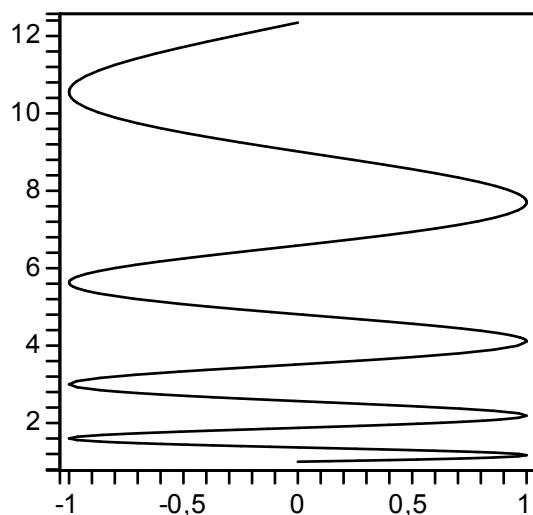
Courbes en coordonnées polaires, option *coords*

Les courbes en coordonnées polaires sont introduites dans **plot** comme des courbes paramétriques mais pour que **plot** interprète les deux premiers termes de la liste, non comme une abscisse et une ordonnée, mais comme un rayon $\rho(t)$, et un angle $\theta(t)$ (exprimés tous deux en fonction d'un paramètre t) on ajoute l'option **coords = polar**. Avec **polar** il existe 13 autres possibilités de représentations comme **bipolar**, **elliptic**, **hyperbolic**, etc. (voir l'aide en ligne). Définissons deux fonctions

```
> rho:=t->sin(t): theta:=t->exp(t/10):
```

Voici la courbe obtenue en *coordonnées paramétriques* (pas d'option *coords* ; ρ donne l'abscisse et θ l'ordonnée).

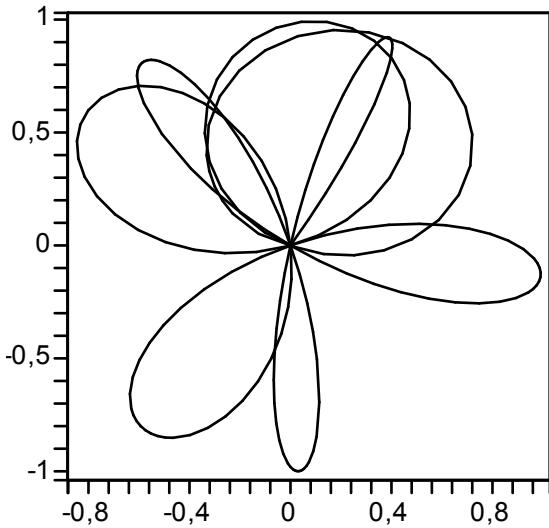
```
> plot([rho(t),theta(t),t=0..8*Pi],  
      axes=boxed,color=black);
```



En *coordonnées polaires*, la première donnée de la liste va représenter le rayon et la seconde l'angle (on peut

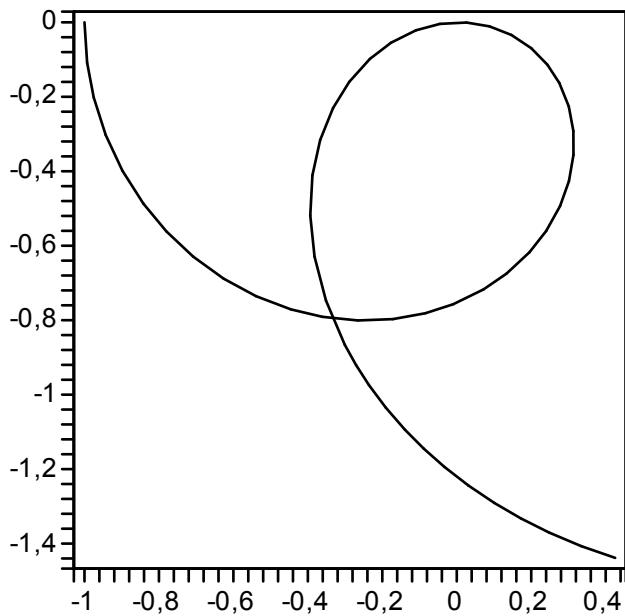
obtenir le même type de tracés avec la fonction **polarplot** de la bibliothèque **plots** ; voir plus loin)

```
> plot([rho(t),theta(t),t=0..8*Pi],coords=polar,  
axes=boxed,color=black);
```



Si $\theta(t) = t$ la syntaxe peut se simplifier de la façon suivante (ici $\rho(t) = \frac{t^2}{10} - 1$).

```
> plot(t^2/10-1,t=0..5,coords=polar,  
axes=boxed,color=black,scaling=constrained);
```



On comprend donc, qu'en raison de la syntaxe des commandes, les courbes en polaires sont incompatibles avec des représentations paramétriques et/ou de la forme $y = f(x)$ (voir plus loin la fonction **display** de **plots** pour résoudre ce problème).

La bibliothèque plots

Les fonctions contenues dans cette bibliothèque complète la fonction **plot**. On en rencontera quelques exemples dans ce paragraphe et dans le suivant (attention au **s** de **plots**). On rappelle que l'on peut appeler les fonctions de **plots** (comme celles de toute bibliothèque) de plusieurs façons:

1) > **with(plots);** et toutes les fonctions sont accessibles de façon permanente.

2) > **with(plots,f,g);** et seules les fonctions f et g sont accessibles de façon permanente.

3) > **plots[f](...);** et on utilise la fonction f directement, mais n'est pas accessible de façon permanente.

Options permanentes

Lorsque l'on doit faire un nombre important de dessins avec les mêmes options (*scaling*, *color* ou autres) il peut être intéressant de définir ces options de façon permanente (pour la durée de la session) avec la fonction **setoption**. Par exemple

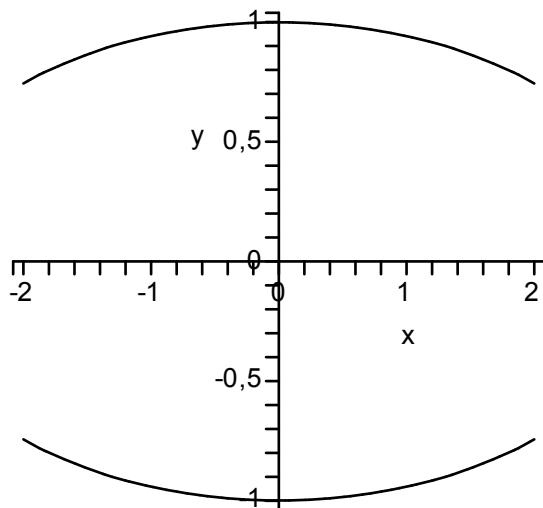
> **plots[setoptions](scaling=constrained,color=black);**

Une option spécifiée dans une commande remplace une option permanente juste pour cette commande.

Fonctions définies de façon implicite

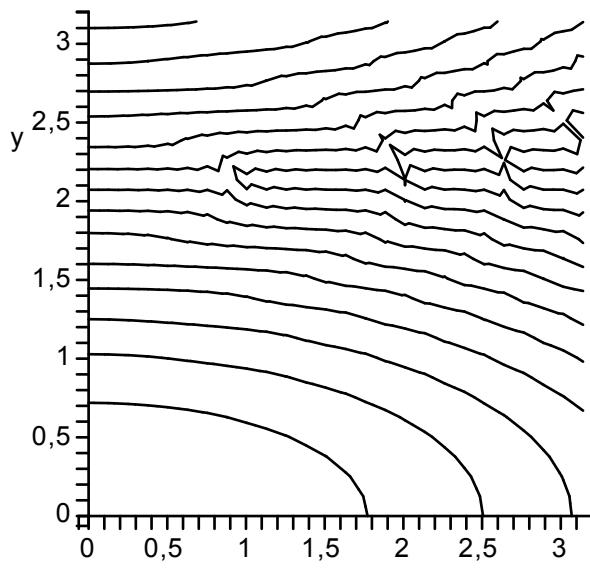
A l'aide de la fonction **implicitplot** de **plots** on peut tracer des courbes de fonctions définies de façon implicite, c'est-à-dire par une équation de la forme $f(x, y) = g(x, y)$ qui servira de premier argument. Les deuxième et troisième arguments, ici obligatoires, donnent les intervalles de variations autorisés pour les deux variables. En fait MAPLE résout $f(x, y) - g(x, y) = 0$ par interpolation sur une grille 25x25.

```
> plots[implicitplot]((x/3)^2+y^2=1,x=-2..2,y=-1.5..1.5,  
color=black);
```



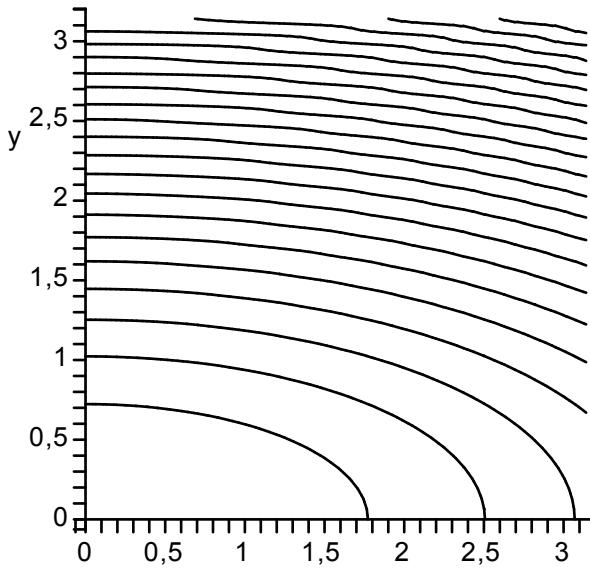
On peut modifier si nécessaire ce paramètre de tracé en précisant le nombres de points sur la grille avec l'option **grid = [n,m]** (n points en x et m points en y). Par exemple on suspecte que le tracé suivant n'est pas correct:

```
> plots[implicitplot](sin(x^2+6*y^2)=0,  
x=0..Pi,y=0..Pi,scaling=constrained,color=black);
```



On augmente la résolution de la grille de calcul et tout rentre dans l'ordre. **Attention** cependant au temps de calcul.

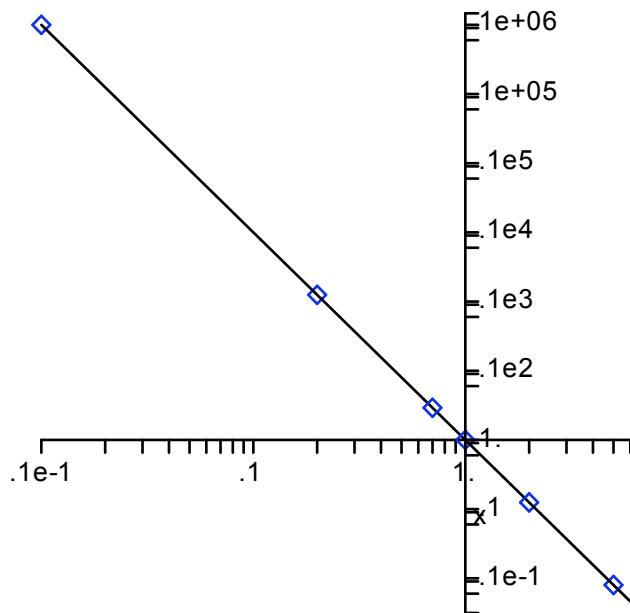
```
> plots[implicitplot](sin(x^2+6*y^2)=0,x=0..Pi,
                      grid=[50,50],scaling=constrained,color=black);
```



Tracés logarithmiques

Voici un exemple de tracé avec la fonction **loglogplot** (voir aussi **logplot**).

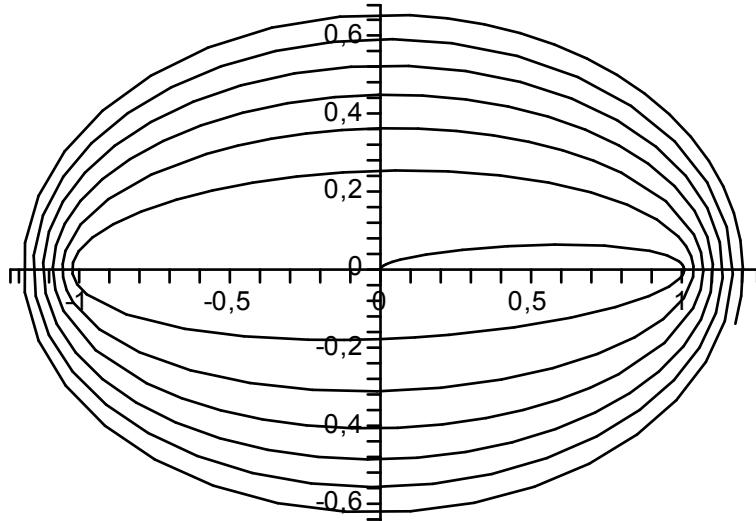
```
> f:=x->x^(-3):
> X:=[0.01,0.2,0.7,1,2,5]:
Lxy:=[seq([x,f(x)],x=X)]:
plots[loglogplot]([Lxy,f(x)],x=0.01..6,
                  style=[point,line],
                  symbol=diamond,symbolsize=20,
                  color=[blue,black]);
```



Tracés dans le plan complexe

La fonction **complexplot** de **plots** permet d'obtenir la représentation dans le plan complexe des valeurs d'une fonction à valeurs complexes.

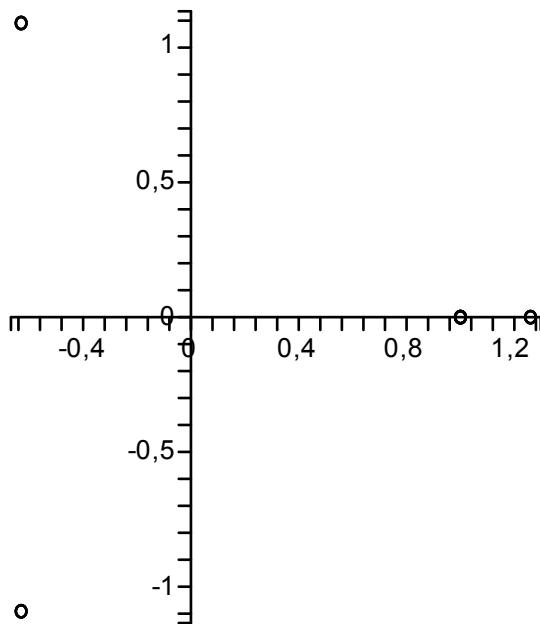
```
> plots[complexplot](sin(t^2+I*t/10),t=0..2*Pi,color=black);
```



Ici on trace les racines d'un polynôme (1 est racine double).

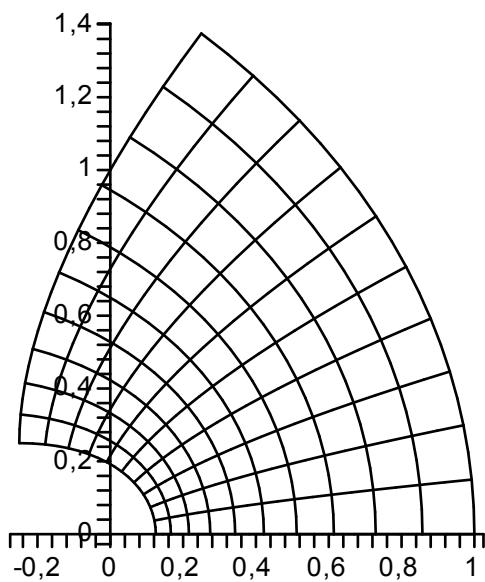
```
> Liste_racines:=[solve(x^5-2*x^4+x^3-2*x^2+4*x-2)];
plots[complexplot](Liste_racines,style=point,
symbol=circle,symbolsize=10,
scaling=constrained,color=black);
```

$$Liste_racines := \left[1, 1, 2^{(1/3)}, -\frac{1}{2} 2^{(1/3)} + \frac{1}{2} I \sqrt{3} 2^{(1/3)}, -\frac{1}{2} 2^{(1/3)} - \frac{1}{2} I \sqrt{3} 2^{(1/3)} \right]$$



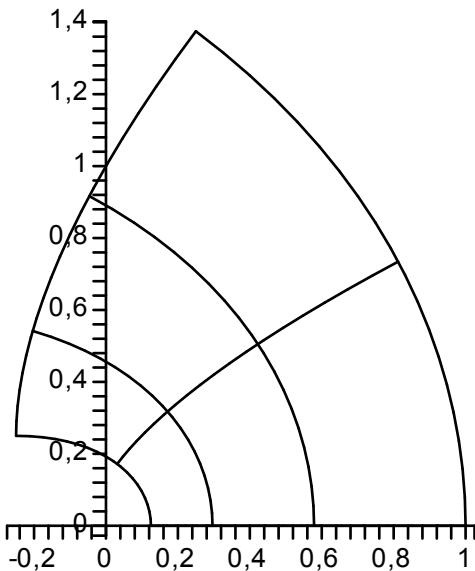
La fonction **conformal** permet de tracer les *représentations conformes* de fonctions à valeurs complexes. La fonction est donnée en premier argument. On doit ensuite spécifier le domaine exploré dans le plan de la variable sous la forme d'un intervalle complexe $[Zi, Zf]$. La fonction **conformal** calcule alors la fonction pour les valeurs de la variable prises, par défaut, sur 11 droites parallèles à parties réelles constantes, l'intervalle $[\Re(Zi), \Re(Zf)]$ étant divisé en 10 parties égales. Il fait de même pour des droites parallèles à partie imaginaire constante en découplant l'intervalle $[\Im(Zi), \Im(Zf)]$. La fonction **conformal** trace alors les courbes dans le plan complexe des valeurs de la fonction quand la variable suit l'une de ces droites. Ces droites étant orthogonales, les intersections des courbes images seront aussi orthogonales. Cependant cette propriété ne sera effective que si on utilise l'option *scaling = constrained*.

```
> plots[conformal](z^3, z=1/2..1+I/2,
                     scaling=constrained, color=black);
```



L'option **grid** permet de fixer les nombres d'intervalle pour le découpage. L'option **numxy** permet également de modifier le nombre de points calculés pour tracer les courbes.

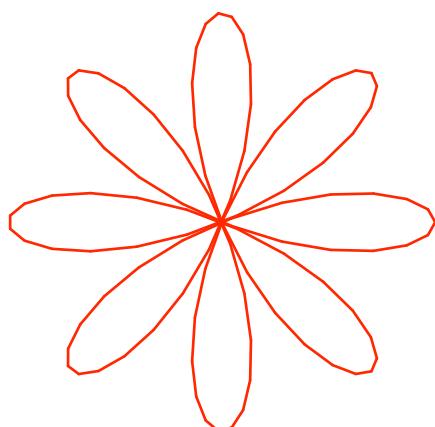
```
> plots[conformal](z^3,z=1/2..1+I/2,grid=[4,3],numxy=[20,20],
scaling=constrained,color=black);
```



Représentations animées

La fonction **animate** permet de construire un graphique animé d'une représentation (simples, polaires ou autres) dépendant d'un paramètre. Ici **animate** va mémoriser les représentations graphiques d'une rosace pour θ appartenant à $[0, 2\pi]$. Le *paramètre d'animation* est le déphasage φ variant dans $[0, 2\pi]$ pour 50 valeurs fixées par l'option **frames** (16 par défaut). On impose avec **numpoints** que chacune des 50 représentations soit définie avec 100 points. Il suffit ensuite de cliquer sur le graphique et lancez l'animation à l'aide du nouveau menu qui apparaît. MAPLE va afficher successivement les 50 représentations graphiques donnant l'illusion du mouvement. Ce menu associé présente différentes options dont *FPS* (Frame Per Second) permettant de régler la vitesse du défilement.

```
> plots[animate]([\cos(4*(theta+phi)),theta,theta=0..2*Pi],
phi=0..2*Pi,numpoints=100,
frames=50,coords=polar,axes=none);
```



Une autre possibilité pour obtenir des graphiques animés est l'option ***insequence*** de la fonction **display** de **plots** (voir la fin du paragraphe suivant).

Exercice: animez les courbes définies paramétriquement par $[x(t) = \cos(t), y(t) = \sin(\alpha t + \varphi)]$ où le déphasage φ est le paramètre d'animation variant de 0 à 2π . On prendra $\alpha = 3$, t dans $[0, 2\pi]$, 80 points pour définir chaque courbe et 100 courbes pour l'animation. Comment appelle-t-on ce type de courbes ?

Il existe encore de nombreuses possibilités offertes par MAPLE comme les tracés de champs de vecteurs **gradplot**, **fieldplot** de **plots**, les tracés de polygones, etc.

Fonction **display** de **plots**

On ne peut pas afficher **avec un même ordre plot** des courbes issues de représentations différentes, soit parce que les options de **plot** sont incompatibles, soit parce qu'elles sont issues de fonctions graphiques différentes comme **implicitplot**. Par exemple, les représentations cartésiennes ordinaires $y(x)$ et paramétriques $[x(t), y(t)]$ sont, pour **plot**, compatibles entre elles mais pas avec les représentations polaires. Nous allons voir comment remédier à cette difficulté. Lorsqu'on lance une commande de tracé, on déclenche deux étapes :

1) La construction d'une structure de données **INTERFACE_PLOT**. Elle n'est pas la description d'un graphique mais contient simplement les données nécessaires à sa construction.

2) La construction et l'affichage du graphique.

On peut cependant éviter la deuxième étape et conserver la structure créée en 1) en l'assignant à un nom. En général **on prendra soin** de terminer la commande par : et non ; pour éviter l'affichage du contenu de la structure qui est généralement long et de peu d'intérêt. Voici deux exemples de structures **INTERFACE_PLOT** (ou **PLOT**) générée par **plot**. On a choisi des options pour que la structure soit courte et ne contienne que 3 points $[x_i, y_i]$.

```
> a:=plot(sin(x)+1,x=0..Pi/6,adaptive=false,numpoints=3);
a := INTERFACE_PLOT(CURVES([[0., 1.], [0.273911117048781349, 1.27049880779170188],
[0.523598774551000012, 1.49999999909301262]], COLOUR(RGB, 1.0, 0., 0.)),
AXESLABELS("x", ""), VIEW(0.. 0.5235987758, DEFAULT))
```

Il existe quatre types de structures: CURVES, POINTS, POLYGONS et TEXT.

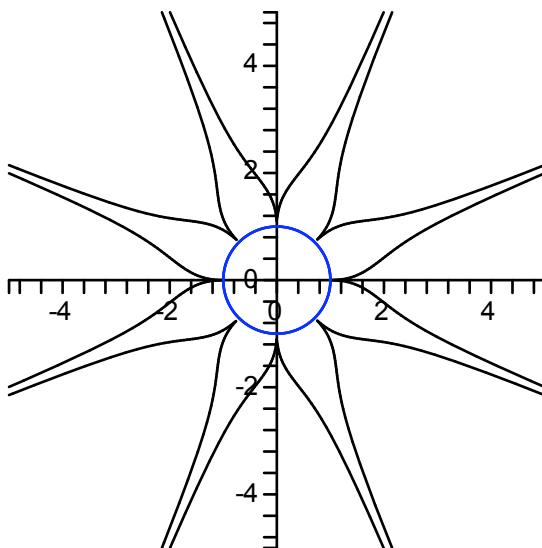
```
> b:=plots[textplot]([0.1,1.4,`f(x)=1+sin(x)`],align={ABOVE,RIGHT},
font=[TIMES,20]);
b := INTERFACE_PLOT(TEXT([0.1, 1.4], "f(x)=1+sin(x)", ALIGNABOVE, ALIGNRIGHT),
FONT(TIMES, 20))
```

En terminant la commande par : et en assignant le résultat de **plot** à un nom, on bloque les affichages de la structure et du graphique.

```
> A:=plot([[1,x,x=0..2*Pi],[1+sqrt(abs(tan(4*x))),x,x=0..2*Pi]],
x=-5..5,y=-5..5,color=[blue,black],
coords=polar,numpoints=500,scaling=constrained):
```

Pour afficher le graphique correspondant on écrira simplement (on passe maintenant à l'étape 2)

```
> A;
```

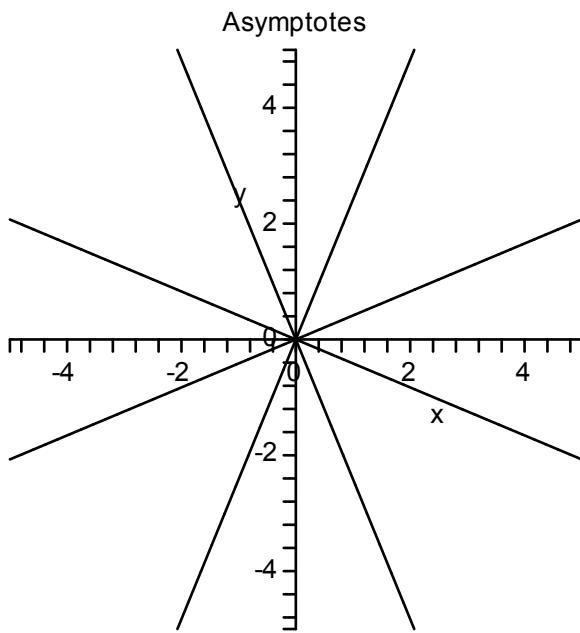


Dans les exemples des paragraphes précédents, les structures n'étaient pas assignées à des noms et n'avaient d'existence que pendant la durée du tracé.

Si l'on veut maintenant placer les asymptotes de la figure précédente, il sera plus simple de les définir en coordonnées cartésiennes

Exercice : Déchiffrez cette écriture un peu cryptographique. Trouvez une autre écriture.

```
> B:=plot([tan((1+2*k)*Pi/8)*x $ 'k'=0..3],  
x=-5..5,y=-5..5,color=black,title="Asymptotes"):  
B;
```



Ces deux graphiques étant issus de deux représentations mathématiques de natures différentes (polaire et cartésienne), on ne peut pas les afficher sur un même dessin à l'aide d'une seule fonction **plot**. Avec la fonction **display** de la bibliothèque **plots** on peut tracer sur un même graphique ces deux structures *A* et *B* que l'on a pris soin de mémoriser. On remarquera les points suivants :

- L'option *scaling=constrained* s'applique au deux représentations bien qu'elle n'est été donnée que

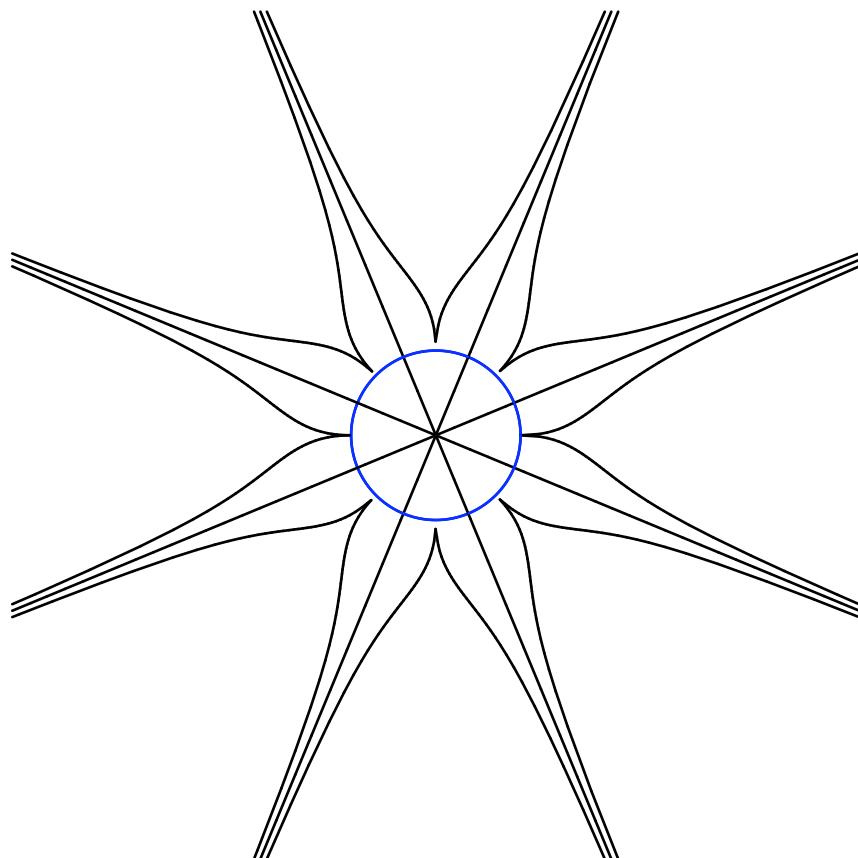
pour la première avec **plot**.

- Le titre "Etoile" remplace le titre "Asymptotes" donné pour B .
- L'option $axes=none$ a fait disparaître les axes pourtant présents dans les deux représentations.

On peut ainsi fixer dans **display** des spécifications d'affichage à condition qu'elles ne modifient pas la nature des représentations.

```
> plots[display](A,B,titlefont=[TIMES,BOLD,14],  
title="Etoile",axes=none);
```

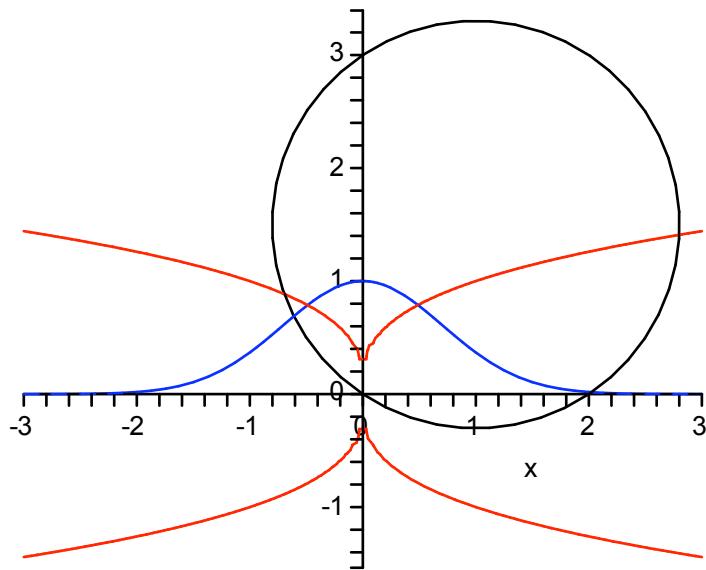
Etoile



On n'est pas obligé de créer et d'assigner les structures INTERFACE_PLOT à des noms. On peut aussi donner directement les commandes graphiques en arguments de **display**. On notera que les 3 courbes suivantes ont des modes de construction différents et incompatibles pour être tracées par un même ordre **plot**.

```
> plots[display](  
plot(exp(-x^2),x=-3..3,color=blue),  
plot([3*sin(t)+2*cos(t),t,t=0..Pi],coords=polar,color=black),  
plots[implicitplot]((x/y)^2-abs(x*y)=0,x=-3..3,y=-3..3,grid=[100,100]),  
title="Mélange",titlefont=[TIMES,20],scaling=constrained);
```

Mélange



Animations avec la fonction display

L'option **insequence=true** de **display** permet d'afficher **successivement** une série de graphiques pouvant (ou non) provenir de méthodes de constructions diverses (avec la fonction **animate** par contre, la méthode de construction dépend continuellement d'un paramètre et ne peut être que la même pour tous les dessins). Par exemple, on dispose de deux suites S_1 et S_2 de 50 graphiques chacune (50 cercles de rayons croissants et 50 cercles avec une "ondulation" superposée). L'une des suites est faite avec des constructions en coordonnées paramétriques et l'autre en polaires, ce qui n'affectera en rien le fonctionnement de **display**.

```
> S1:=seq(plot([r/50*cos(theta),r/50*sin(theta),theta=0..2*Pi]),r=1..50):
S2:=seq(plot([r/50*(1+sin(10*t))^2/20],t,t=0..2*Pi),
coords=polar),r=1..50):
```

On alterne les deux suites en une suite simple: $S_2_1, S_1_1, S_2_2, S_1_2, \dots$

Exercice: expliquer cette écriture

```
> S:=op(zip((x,y)->(x,y),[S2],[S1])):
```

L'instruction suivante va afficher les 100 graphiques en un seul dessin fixe

Exercice: S peut être une liste ou un ensemble au lieu d'une suite. Modifier l'expression suivante ou précédente et essayer

```
> plots[display](S,scaling=constrained);
```

alors qu'avec l'option **insequence=true**, les dessins vont être affichés successivement donnant l'illusion d'une animation.

```
> plots[display]([S],scaling=constrained,insequence=true);
```

Création de fichiers dessins

On peut, à l'aide de la fonction **plotsetup**, modifier la façon dont MAPLE va renvoyer son graphique. En particulier il est possible de lui demander de ne pas l'afficher sur l'écran mais de le **stocker dans un fichier** qui sera imprimé ultérieurement ou inclu dans un autre fichier.

On doit préciser

1) Le nom du "pilote", ici ***jpeg***.

2) Le nom du fichier avec l'option ***plotoutput="nom_du_fichier"***. Le *nom_du_fichier* est une chaîne de caractères pouvant contenir le chemin d'accès complet au répertoire (voir les explications au chapitre 20, *Lecture, écriture et fichiers*).

```
> restart:  
plotsetup(jpeg,plotoutput=cat(kernelopts(homedir),  
"/Desktop/Gauss.jpg")):  
plot(exp(-x^2)/sqrt(Pi),x=-3..3);
```

Traçons maintenant cette fonction dans un fichier de type ***postscript (ps)***. Avec ***plotoptions*** on a fixé

- la taille du dessin avec ***width*** (largeur) et ***height*** (hauteur). Préciser l'unité (pt, cm, in; pt par défaut où pt vaut 1/72 d'in).

- l'orientation de la feuille de papier en mode ***portrait***. L'option par défaut est ***landscape*** (paysage)
- la suppression du cadre avec l'option ***noborder***
- la marge à gauche avec ***leftmargin*** et en bas avec ***bottommargin*** (préciser ici aussi les unités)

Il existe d'autres options.

```
> plotsetup(ps,plotoutput=cat(kernelopts(homedir),"/Desktop/Gauss.ps"),  
plotoptions=`width=10cm,height=10cm,portrait,noborder,  
leftmargin=8cm,bottommargin=17cm`):  
plot(exp(-x^2)/sqrt(Pi),x=-3..3);
```

On revient à un affichage à l'écran dans la feuille de calcul

```
> plotsetup(default);
```

Attention

Si l'ordre de tracé qui écrit dans un fichier est ***dans le corps d'une procédure*** (chapitre 19) on écrira

```
...  
print(plot(...));  
...
```

Représentations graphiques des solutions d'équations différentielles

Nous avons déjà vu au chapitre 15, *Equations différentielles*, comment tracer avec ***odeplot*** de la bibliothèque ***plots***, la solution numérique d'une équation différentielle obtenue avec ***dsolve(...,numeric)***. La bibliothèque ***DEtools*** contient aussi des fonctions pour tracer directement les représentations graphiques des solutions d'équations différentielles sans passer par ***dsolve***. On ne donnera ici que deux exemples relatifs à la fonction ***DEplot***. On rappelle que la fonction choisie doit être lue dans la bibliothèque.

La fonction ***DEplot*** permet de tracer les solutions d'une équation (ou d'un système autonome d'équations) associées à un ensemble de conditions initiales données sous la forme d'un ensemble de listes $\{y(t_i) = y_i, \dots\}$. L'intervalle de variation de la variable d'intégration est ici $[-1,5]$. On impose également la limite en ordonnée du graphique (argument optionnel), c'est-à-dire $[-3,3]$. On peut également fixer le pas d'intégration par l'option ***stepsize*** ou indiquer à MAPLE d'arrêter l'intégration, avec ***obsrange=true***, dès que l'ordonnée calculée (avec des abscisses croissantes) sort des limites maximales fixées pour les ordonnées.

Cette fonction permet également de tracer (par défaut) le champ de directions associé à l'équation différentielle pour les équations de la forme $y' = f(x, y)$. Ce champ est visualisé par des vecteurs dont la 'pente' est donnée par la valeur prise par f sur une grille de points de coordonnées $\{x, y\}$. L'orientation correspond à un accroissement positif de la variable d'intégration. Si les conditions initiales sont absentes, seul le champ de

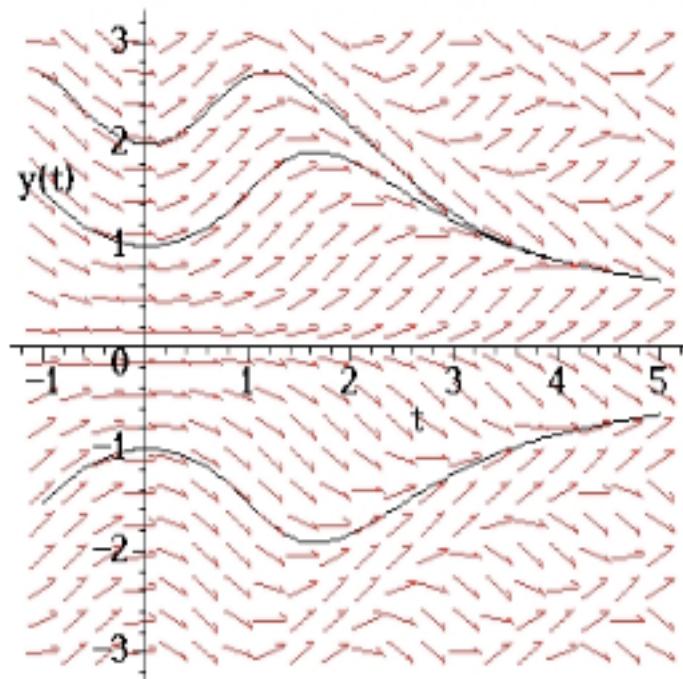
directions sera donné. Cette représentation est très utile pour se faire une idée du comportement général des solutions. On peut supprimer l'affichage du champ de directions avec l'option **arrows=none**. L'équation différentielle suivante est réputée pour n'avoir pas d'expression connue de sa solution

```
> restart;
eqd:=diff(y(t),t)=sin(t*y(t)); eqd;
sol:=dsolve({eqd,y(0)=1},y(t));

$$\frac{dy}{dt} = \sin(t y(t))$$

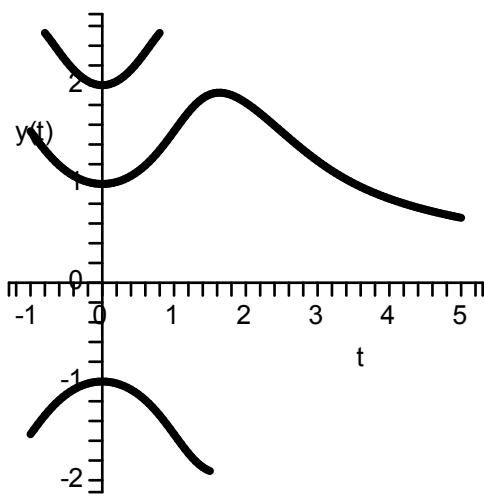
sol :=

> DEtools[DEplot](eqd,y(t),t=-1..5,
{[y(0)=1],[y(0)=2],[y(0)=-1]},y=-3..3,
stepsize=0.05,obsrange=true,linecolor=black);
```



Attention avec l'option **obsrange=true** car une partie de la représentation graphique peut disparaître pour des ordonnées qui reviennent dans l'intervalle imposé pour les abscisses supérieures ou inférieures (puisque le calcul s'arrête).

```
> DEtools[DEplot](eqd,y(t),t=-1..5,
{[y(0)=1],[y(0)=2],[y(0)=-1]},y=-1.9..2.5,
stepsize=0.05,obsrange=true,linecolor=black,
arrows=none);
```



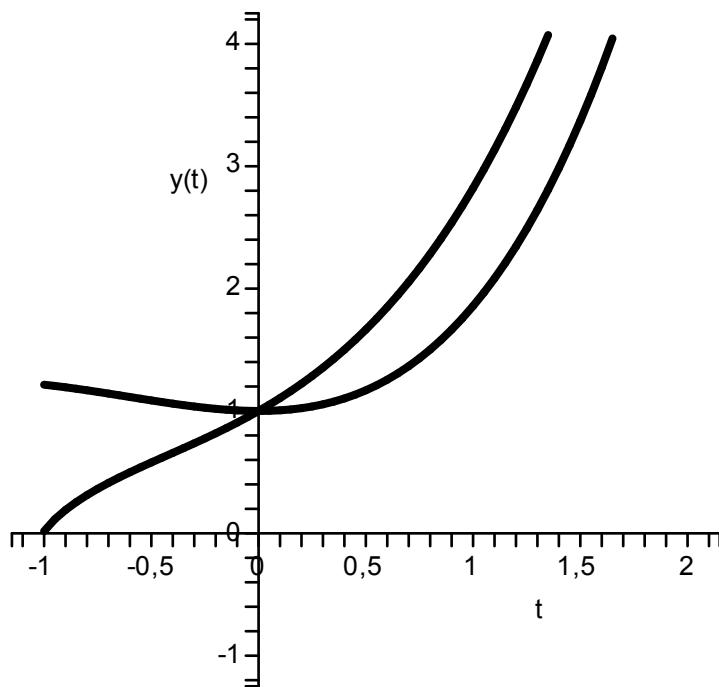
Pour une équation du second ordre **DEplot** ne peut pas tracer le champ de directions.

```
> eqd:=diff(y(t),t$2)=exp(t)+t/y(t): eqd;
sol:=dsolve({eqd,D(y)(0)=0,y(0)=1},y(t));

$$\frac{d^2}{dt^2}y(t) = e^t + \frac{t}{y(t)}$$

sol :=
```

```
> DEtools[DEplot](eqd,y(t),t=-1..2,
{[D(y)(0)=0,y(0)=1],[D(y)(0)=1,y(0)=1]},
y=-1..4,stepsize=0.05,obsrange=true,linecolor=black);
```



On trouvera de même la fonction **DEplot3d** de **DEtools** permettant de tracer des courbes en représentation dans l'espace pour un système de deux équations différentielles.

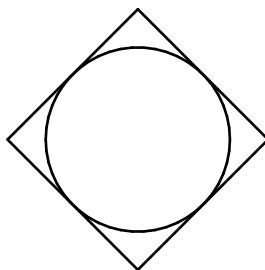
Bibliothèques plottools et geometry

Voici deux courts exemples pour illustration

La bibliothèque plottools

Elle permet de créer et manipuler des objets géométriques (cercles, ellipses, etc.).

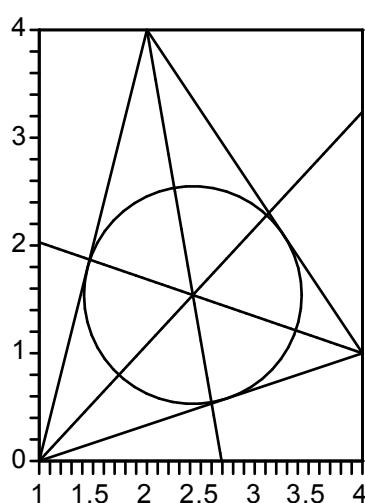
```
> Carre:=plottools[rectangle]([-1,-1],[1,1]):  
plots[display]([plottools[circle]([0,0],1,color=black),  
plottools[rotate](Carre,Pi/4)],  
scaling=constrained,axes=none);
```



La bibliothèque geometry

Elle permet de traiter la géométrie euclidienne dans le plan.

```
> with(geometry):  
triangle(T, [point(A,1,0),point(B,4,1),point(C,2,4)]):  
incircle(C_ins,T):  
bisector(bA,A,T):  
bisector(bB,B,T):  
bisector(bC,C,T):  
  
> draw([C_ins,T,bA,bB,bC],color=black);
```



```
> AreConcurrent(bA,bB,bC);  
true
```

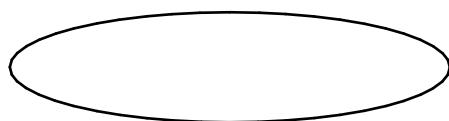
Création d'une fonction graphique

Au chapitre 7, *Fonctions*, nous avons vu que les images d'une fonction de MAPLE créée par l'utilisateur pouvait être de nature quelconque. En particulier, une image (au sens mathématique) peut être une représentation graphique. On veut tracer par exemple des ellipses de demi grands axes a et b ayant pour "centre" un point de coordonnées (x,y) . Une représentation paramétrique sera bien adaptée. De plus on ne veut pas tracer le système d'axes. On peut définir pour cela une fonction E

```
> E:=(a,b,x,y)->  
    plot([x+a*cos(theta),y+b*sin(theta),theta=0..2*Pi],  
         scaling=constrained,axes=none,color=black):
```

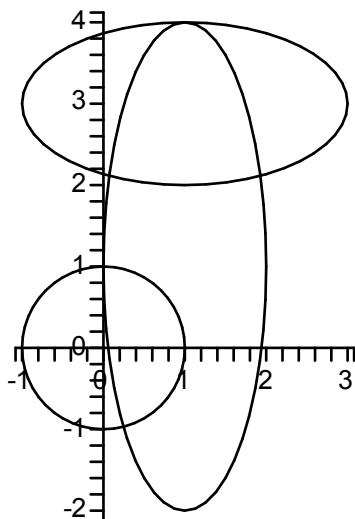
On écrira alors (on comprend comment on peut construire des librairies comme **plottools** ou **geometry**; voir chapitre 19, *Procédures*)

```
> E(2,0.5,1,3);
```



Ou encore

```
> plots[display](E(2,1,1,3),E(1,1,0,0),E(1,3,1,1),axes=normal);
```



Exercice: créer une fonction qui dessine un carré dont le côté a une longueur a et dont les diagonales se coupent en (x, y) .

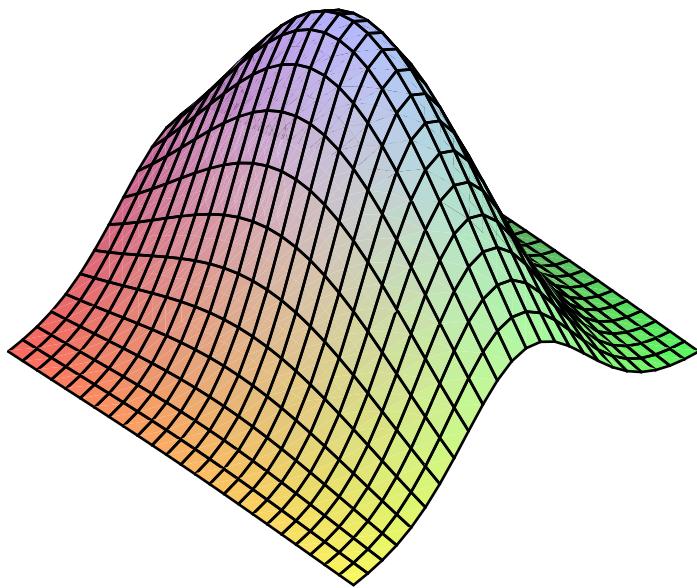
17 - Représentations graphiques 3D

On appelle tracés 3D des dessins de surfaces ou de courbes dans l'espace dans une représentation perspective. Il existe de nombreux points communs entre la syntaxe des commandes des constructions 2D et 3D. Aussi nous mettrons surtout en avant dans ce chapitre les différences entre ces deux types de représentations. Ces dessins peuvent être enregistrés dans des fichiers de type *postscript* ou autre (voir le chapitre 16, *Représentations graphiques 2D*).

Surface définie par $z = f(x,y)$

Ce type de surface est engendrée par une fonction donnant son ordonnée z en fonction d'un couple de variables x et y balayant un domaine rectangulaire du plan xOy . On utilisera pour créer cette représentation la fonction **plot3d** et on donnera d'abord l'*expression* $f(x, y)$ suivie des deux intervalles de variations de x et y . La fonction **plot3d** supporte de nombreuses options dont beaucoup sont partagées avec **plot**. D'autres sont spécifiques et nous en donnerons seulement quelques exemples. On ne dira rien en particulier sur les divers modes du coloriage des surfaces ni sur les options "d'éclairage" (voir l'aide en ligne `>?plot3d,options`).

```
> restart:  
plot3d(exp(-(x^2+y^2)), x=-2..2, y=-1..1);
```



Quand $f(x, y)$ est donnée par une *fonction* ou une *procédure* on retrouve les mêmes règles de syntaxe que pour les représentations 2D

```
> f:=(x,y)->exp(-(x^2+y^2));  
plot3d(f(x,y), x=-2..2, y=-1..1);  
plot3d(f,-2..2,-1..1);
```

On peut redimensionner le graphique comme pour le graphisme 2D (voir chapitre précédent) et modifier

l'angle de vue (voir plus loin).

Tracés des axes

Par défaut **plot3d** ne trace pas d'axes. On peut lui demander un tel tracé avec l'option **axes = ...**. Les différentes possibilités sont

- **boxed** : les axes forment une boîte autour de la surface.
- **frame** : ~~boxed mais~~ seuls les axes gradués sont dessinés.
- **normal**
- **none** : les axes ne sont pas tracés (défaut).

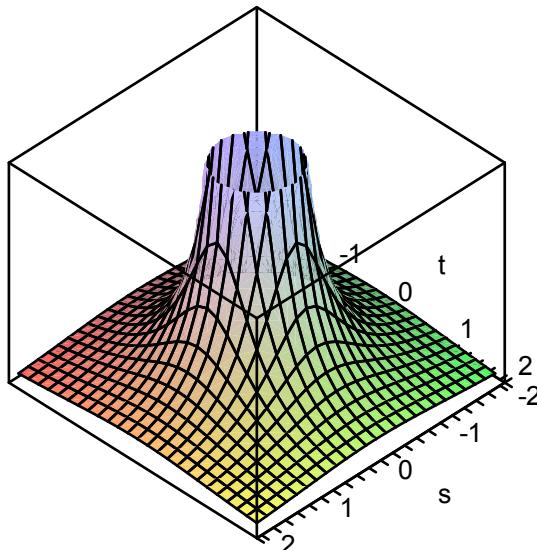
Lorsque l'on clique avec le bouton gauche de la souris sur le dessin des menus apparaissent qui permettent diverses opérations. Tous ces types de systèmes d'axes sont accessibles directement par ces menus.

Pour dessiner un système d'*axes orthonormés* on utilisera (voir 2D) l'option **scaling=constrained**.

Limitation des ordonnées du graphique

Pour des raisons identiques à celles donnée pour **plot**, on peut vouloir réduire l'intervalle de variation de l'ordonnée. On utilisera pour cela l'option **view = z_{min} .. z_{max}**.

```
> plot3d(1/(s^2+t^2), s=-2..2, t=-2..2, view=0..3, axes=boxed);
```



Modification du rendu des surfaces

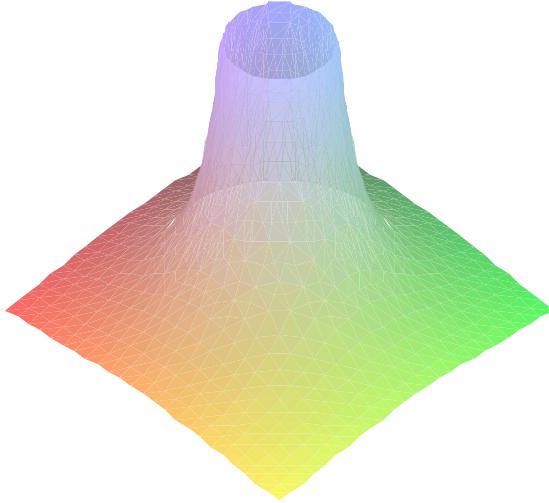
L'effet de perspective et le rendu de la surface sont obtenus grâce à un maillage de celle-ci. Chaque élément prend une couleur (ou un niveau de gris) et le dégradé de l'ensemble renforce le rendu. On peut obtenir différents rendus avec l'option **style=...** dont les possibilités sont:

- **style=patch** : c'est le mode par défaut décrit précédemment.
- **style=hidden** : (hidden : caché) identique au précédent mais les mailles ne sont pas coloriées. Le mot "caché" se comprend car certaines parties de la surface peuvent en cacher d'autres.
- **style=wireframe** : (wireframe : charpente de fils, treillis) c'est le mode précédent mais toutes les parties de la surface sont transparentes.
 - **style=patchnogrid** : c'est le mode **patch** mais sans le tracé de la maille.
 - **style=point** : seuls les points d'intersection de la maille sont représentés (peut s'utiliser avec l'option **symbol**).
 - **style=patchcontour** : identique à **patchnogrid** mais avec des "lignes de niveaux" régulièrement espacées.
 - **style=contour** : seules les lignes de contour sont tracées.

Tous ces styles sont accessibles par menu en cliquant sur le dessin avec le bouton gauche de la souris.

Il existe aussi une option **transparency=x** rendant une surface plus ou moins transparente avec x variant de 0 à 1. Avec $x = 0$ la surface est opaque et $x = 1$ la rend... invisible (voir deux exemples plus loin à *tracés multiples* et *coordonnées sphériques*).

```
> plot3d(1/(s^2+t^2),s=-2..2,t=-2..2,view=0..3,style=patchnogrid);
```



Modification de l'orientation du système d'axes (de l'angle de vue)

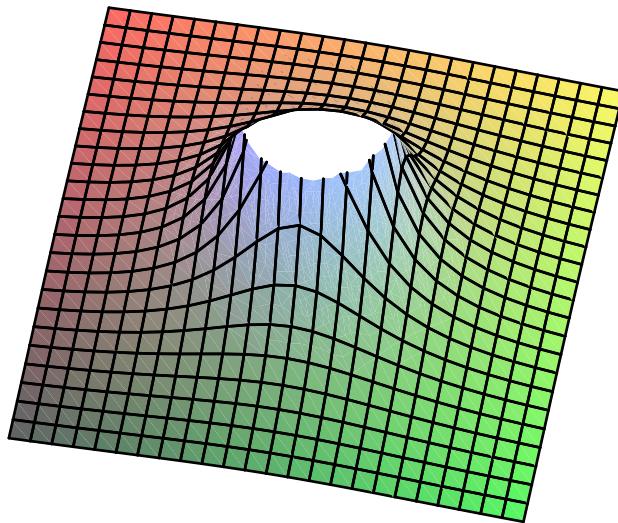
Le dessin apparaît en perspective avec une orientation par défaut du système d'axes. Il est possible de changer cette orientation de trois façons:

- En cliquant sur le graphique avec le bouton de gauche de la souris et *en le maintenant enfoncé*. Le déplacement de la souris entraîne alors la rotation du dessin dans l'espace. En même temps, en haut à gauche de la fenêtre, deux angles θ et φ donnent l'orientation du trièdre. **Notez** que c'est le mode par défaut d'utilisation de la souris. Dans le menu qui apparaît quand on clique sur le dessin, le bouton à droite (dessin d'une flèche de rotation) ouvre un menu. Les options **Scale** ou **Pan** permettent de changer le mouvement de la souris en *changement d'échelle* ou *déplacement* du dessin.

- En cliquant sur le graphique avec le bouton de gauche de la souris et en faisant défiler les deux angles θ et φ à l'aide des flèches.

- En spécifiant cette orientation dans la commande **plot3d** avec l'option **orientation=[θ , φ]** (angles en degrés). Cette option est très utile quand, après avoir défini une orientation de façon interactive, on a besoin de retracer souvent des dessins sous un même angle de vue.

```
> plot3d(1/(s^2+t^2),s=-2..2,t=-2..2,view=0..3,  
          orientation=[15,155]);
```

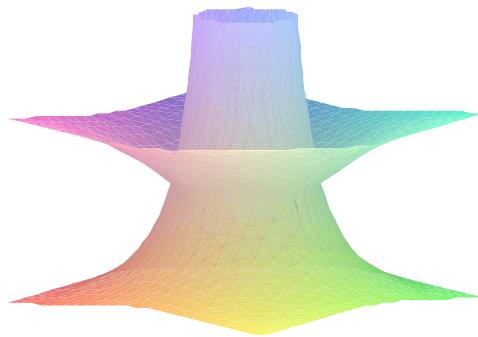


Tracés multiples

1) Tracés multiples de mêmes caractéristiques

On peut tracer sur un même dessin des surfaces **partageant les mêmes options** en donnant comme premier argument, la liste ou l'ensemble des expressions définissant les surfaces.

```
> plot3d([2-exp(-(x^2+y^2)/2), 1/(x^2+y^2)],
      x=-2..2,y=-2..2,view=0..3,
      style=patchnogrid,transparency=[0.8,0.5],
      orientation=[40,80]);
```



2) Tracés multiples composés

On peut, avec la fonction **display** de **plots**, tracer, sur une même représentation, plusieurs dessins 3D issus de représentations de natures différentes (courbes dans l'espace, surfaces en coordonnées cartésiennes, sphériques, etc) ou d'options non identiques. Quand on lance une commande **plot3d** on déclenche, comme pour la fonction **plot**, deux opérations distinctes :

1) **plot3d** construit une structure particulière du type **INTERFACE_PLOT3D** (l'analogie de **INTERFACE_PLOT** en 2D) contenant les informations nécessaires à la construction du graphique.

2) Ces données sont ensuite utilisées pour construire le graphique qui s'affichera sur l'écran ou s'enregistrera dans un fichier. Les structures **INTERFACE_PLOT3D** sont ensuite détruites.

De façon identique aux tracés 2D, la deuxième étape peut être annulée en assignant la structure à un nom.

On retrouvera alors la possibilité, avec la fonction **display** de **plots**, de tracer simultanément plusieurs surfaces définies avec des options ou des représentations différentes.

Notez, avec le prochain exemple, les points suivants :

- Les deux domaines du plan xOy ne sont pas les mêmes.

• Il est préférable de terminer les commandes suivantes par ":" pour éviter un affichage encombrant et sans grand intérêt.

```
> S1:=plot3d(2-exp(-(x^2+y^2)/2),x=-1..1,y=-2..2,  
             style=patchnogrid):  
S2:=plot3d(1/(x^2+y^2),x=-2..2,y=-2..2,  
             style=wireframe,view=0..3,color=black):
```

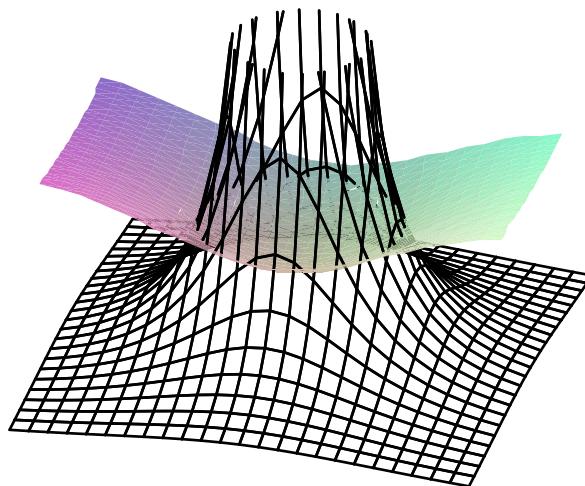
Avec **display** on va pouvoir afficher, bien qu'elles n'aient pas les mêmes options, ces deux surfaces sur le même dessin. **Notez** que :

- Certaines options ne sont pas propres à la définition des surfaces et peuvent être redéfinies par **display**. Les structures INTERFACE_PLOT3D contiennent les coordonnées des points de la maille représentant la surface dans le trièdre $Oxyz$. C'est le programme de tracé qui va calculer les coordonnées dans le plan du dessin de ces points en fonction de la perspective choisie. Comme **display** ne fait qu'activer le programme de tracé, on comprend que l'on puisse choisir une orientation avec cette fonction.

- Les tracés obtenus avec **plot** et **plot3d** sont naturellement incompatibles comme ne manquerait pas de le rappeler MAPLE par un message d'erreur.

```
> plots[display]({S1,S2},title="Figure composée",  
             orientation=[15,55]);
```

Figure composée



Limitation du domaine de tracé, définition du tracé des surfaces

La limitation du tracé des surfaces dans un volume défini par des intervalles sur les axes Ox , Oy et Oz peut aussi être obtenue avec l'option **view** et une liste sous la forme

view=[$x_{\min} \dots x_{\max}$, $y_{\min} \dots y_{\max}$, $z_{\min} \dots z_{\max}$].

Le nombre de points de la maille est par défaut de 25x25. Ceci peut s'avérer insuffisant pour une surface présentant par exemple de nombreuses "oscillations". On peut modifier le nombre de points ou la nature de la grille de calcul avec les options :

- **grid** = $[m, n]$: la grille aura m points suivant la première coordonnée et n points suivant la seconde.
- **numpoints** = n : MAPLE va calculer l'entier m le plus proche de \sqrt{n} et utiliser une grille $m \times m$.
- **gridstyle** = *triangular* : la maille de la grille est triangulaire au lieu de rectangulaire par défaut.

Tracés à partir de données contenues dans un fichier, fonction **surfdata**

On donne un exemple à partir d'un fichier contenant un série de $n = 20$ points d'abscisses en x variants de -2 à 1 et de $m = 15$ points d'abscisses en y variants de -3 à 4, ainsi que des valeurs associées d'une fonction. Ces abscisses ne sont pas équidistantes et forme donc une grille irrégulière. La première colonne contient d'abord les 20 abscisses *croissantes* variant de -2 à 1 pour la valeur $y_1 = -3$ en deuxième colonne et la valeur de la fonction en troisième colonne. Puis viennent à nouveau les 20 points d'abscisses en x pour $y_2 = -2.853284$ et ainsi de suite jusqu'à $y_m = 4$.

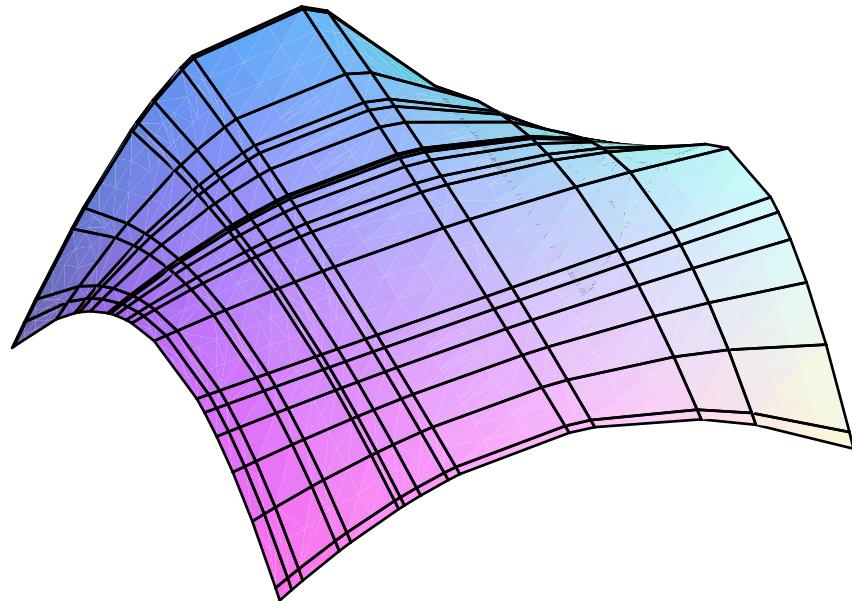
-2	-3	0.3623577545
-1.9828336	-3	0.3719382177
-1.505599	-3	0.6189749572
...		
0.69294	-3	0.9148080163
0.95432	-3	0.8404994223
1	-3	0.8253356149
-2	-2.853284	0.4164005631
-1.9828336	-2.853284	0.4252868925
-1.505599	-2.853284	0.6530584469
...		
0.69294	-2.853284	0.922830656
0.95432	-2.853284	0.8553401861
1	-2.853284	0.8415463633
...		
0.69294	4	0.8502417211
0.95432	4	0.7224507743
1	4	0.6967067093

On lit ce fichier (pour les explications et les détails, se reporter au chapitre 20, *Lecture, écriture et fichiers*)

```
> fic:=cat(kernelopts(homedir), "/Desktop/Surfdata.dat"):
L:=readdata(fic,[float$3]):
```

La structure L doit être ré-organisée pour être conforme à une de celle qui est spécifiée dans l'aide en ligne de la fonction **surfdata**

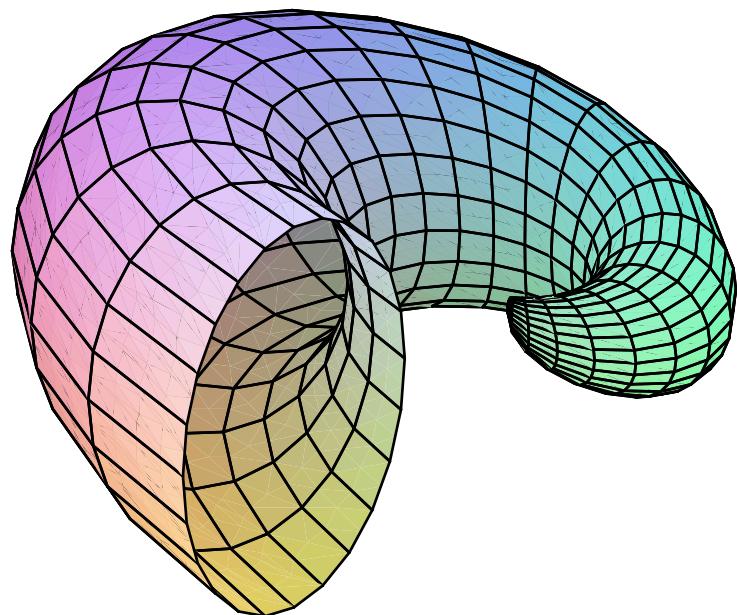
```
> n:=20: m:=15:
P:=[seq([seq(L[(k*n)+i], i=1..n)], k=0..m-1)]:
> plots[surfdata](P, orientation=[-25, 40]);
```



Surfaces en coordonnées paramétriques

Certaines surfaces peuvent être tracées grâce à une représentation paramétrique donnée par 3 fonctions dépendant de 2 paramètres, chacune représentant une coordonnée: $x(u, v)$, $y(u, v)$ et $z(u, v)$, u et v étant les deux paramètres. Comme pour les tracés 2D, on constitue une liste de ces trois fonctions et on donne les intervalles de variation des deux paramètres

```
> plot3d([(2+cos(r))*3*s*cos(s),(2+cos(r))*3*s*sin(s),3*s*sin(r)],
r=0..2*Pi,s=0..2*Pi);
```



Surfaces en coordonnées sphériques, cylindriques, etc. Option *coords*

L'option *coords* de **plot3d** permet de donner une interprétation différente de la fonction $f(x, y)$ donnée en premier argument. Pour décrire une surface dans un espace à 3 dimensions on a besoin de faire varier deux variables. Par exemple, considérons le trièdre direct Oxyz de référence où le plan Oxy est, pour fixer les idées, le plan horizontal et z l'axe vertical. Quand x et y sont donnés, un point de la surface est défini par l'ordonnée z liée à x et y par $z = f(x, y)$. On peut cependant utiliser d'autres modes de représentation mais on aura toujours besoin de deux variables. L'option *coords* offre 29 modes de représentations possibles mais on ne donnera que les deux plus habituels. Pour connaître les autres, exécuter la commande

```
> ?plot3d,coords
```

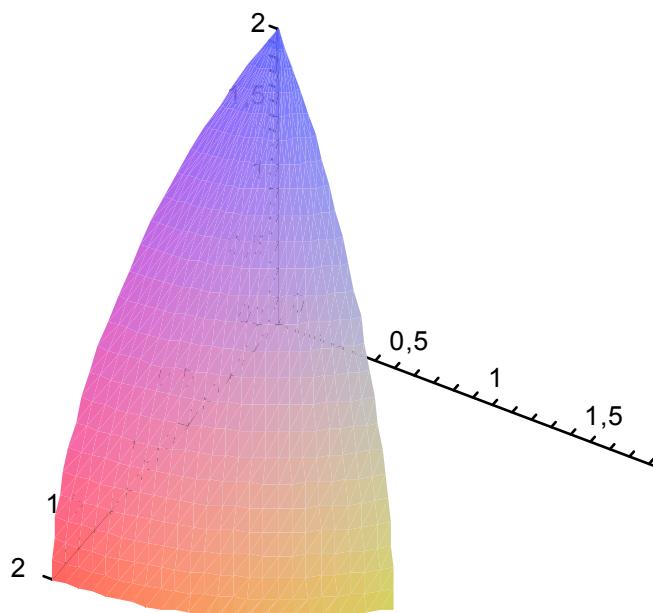
Coordonnées sphériques, option *coords=spherical*

Dans cette représentation on considère deux angles. L'un, φ , balaye tout ou partie du plan Oxy dans le sens trigonométrique autour de Oz et avec pour origine l'axe Ox. L'autre, θ , a pour origine l'axe $z > 0$. Pour balayer tout l'espace, l'angle φ varie de 0 à 2π dans le sens trigonométrique et l'angle θ de 0 à π . Le point de la surface décrite est fixé par le rayon $\rho = f(\theta, \varphi)$ suivant la direction définie par les deux angles.

Les arguments de **plot3d** doivent être successivement *et dans cet ordre*, l'expression $f(\theta, \varphi)$, l'angle φ puis l'angle θ avec leurs variations. Viennent ensuite les options dont *coords=spherical*.

L'exemple le plus simple est celui de la sphère de rayon R pour laquelle f est la constante R . On trace ici un sixième de sphère où on reconnaîtra l'angle φ et l'angle θ par les valeurs balayées.

```
> R:=2:  
plot3d(R,phi=0..Pi/4,theta=0..Pi/2,  
       style=patchnogrid,axes=normal,coords=spherical,  
       view=[0..2,0..2,0..2],orientation=[30,45],  
       transparency=0.8);
```

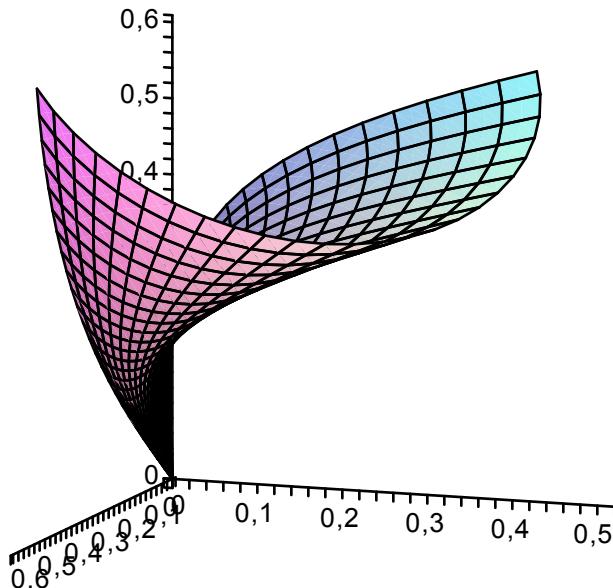


On pourra utiliser aussi la syntaxe suivante dans laquelle la liste en premier argument contient trois expressions, dépendant de deux paramètres u et v et décrivant respectivement le rayon, l'angle φ et l'angle θ . L'exemple présenté ci-dessous donne le même résultat que le précédent.

```
> plot3d([R,u,v],u=0..Pi/4,v=0..Pi/2,coords=spherical,
         style=patchnogrid,axes=normal,
         view=[0..2,0..2,0..2],orientation=[30,45]):
```

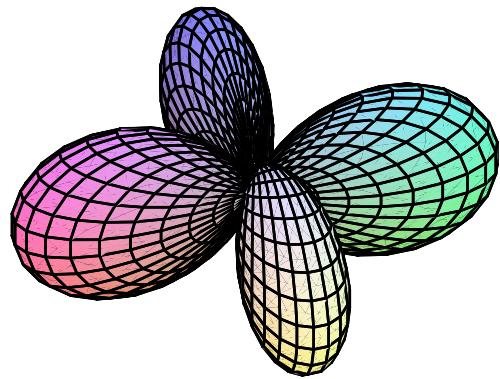
Cette syntaxe autorise des constructions plus complexes. Afin de donner un exemple plus parlant on crée préalablement trois fonctions ρ , φ et θ , mais cette étape n'est absolument pas nécessaire et les expressions peuvent, bien sûr, être données directement dans la liste.

```
> rho:=(u,v)->u;
phi:=(u,v)->u*v;
theta:=(u,v)->cos(u-v);
plot3d([rho(u,v),phi(u,v),theta(u,v)],u=0..Pi/4,v=0..Pi/2,
       coords=spherical,axes=normal,orientation=[20,80]);
 $\rho := (u, v) \rightarrow u$ 
 $\varphi := (u, v) \rightarrow u v$ 
 $\theta := (u, v) \rightarrow \cos(u - v)$ 
```



L'option **grid** peut s'avérer nécessaire pour améliorer la précision du dessin. Ici 100 s'applique au découpage de u (l'angle φ) et 30 à v (l'angle θ).

```
> plot3d(sin(v)^2*sin(2*u)^2,u=0..2*Pi,v=0..Pi,
         coords=spherical,grid=[100,30],
         orientation=[30,35]);
```

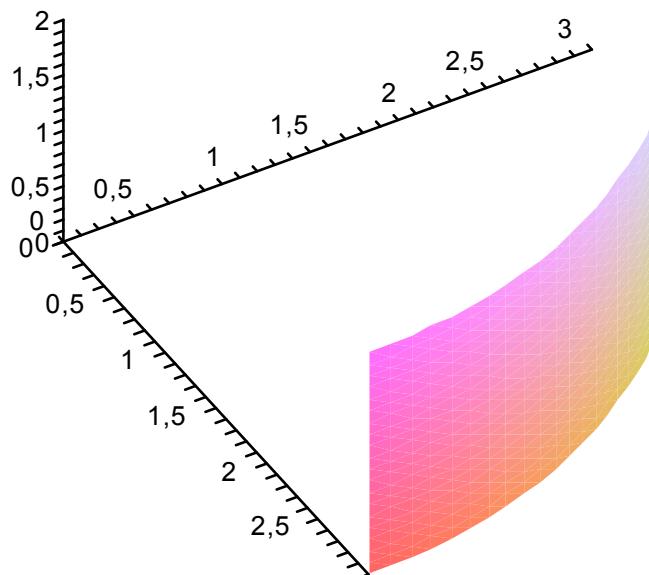


Coordonnées cylindriques, option coords=cylindrical

Dans cette représentation on considère un angle, φ , qui peut balayer tout ou partie du plan Oxy dans le sens trigonométrique avec pour origine l'axe Ox. L'autre variable est simplement l'ordonnée suivant z . La surface est alors définie par l'extrémité du rayon $\rho(\varphi, z)$ de coordonnée origine $(0,0,z)$, parallèle au plan Oxy dans la direction définie par φ .

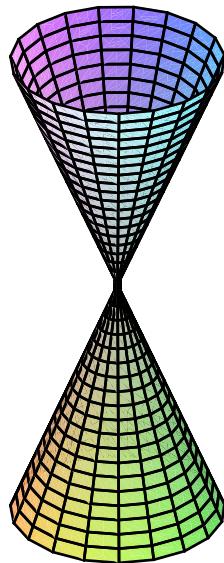
La représentation la plus simple est un cylindre d'axe Oz et de rayon R . On ne trace ici qu'un huitième de cylindre

```
> R:=3:
plot3d(R,phi=0..Pi/4,z=0..2,coords=cylindrical,grid=[20,20],
       axes=normal,orientation=[-30,30],
       style=patchnogrid,view=[0..R,0..R,0..2]);
```



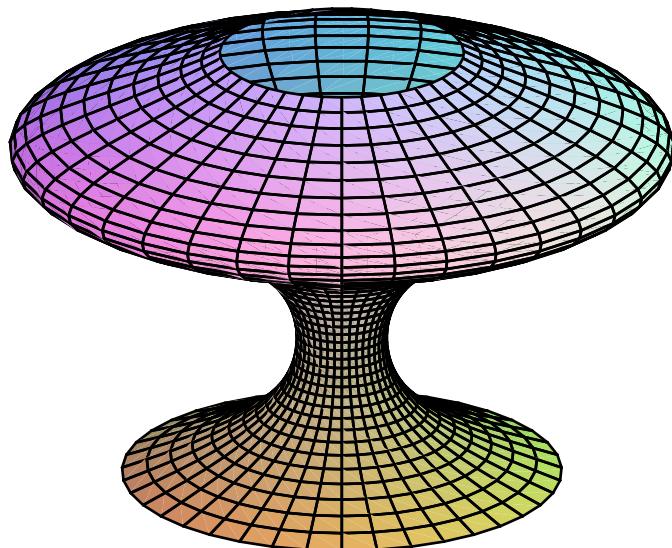
On représente maintenant un cône dont le demi angle au sommet vaut $\pi/8$.

```
> plot3d(abs(z)*tan(Pi/8),phi=0..2*Pi,z=-1..1,
       coords=cylindrical,grid=[20,40],
       orientation=[75,60],scaling=constrained);
```



Exercice : Très "design" le vase..., mais il n'a pas de fond ! Faire tourner la figure pour s'en convaincre. Pouvez-vous lui en donner un de forme conique aplatie (la pointe vers le haut évidemment ! ;-)) et de hauteur 0.1 ? (solution en fin de chapitre).

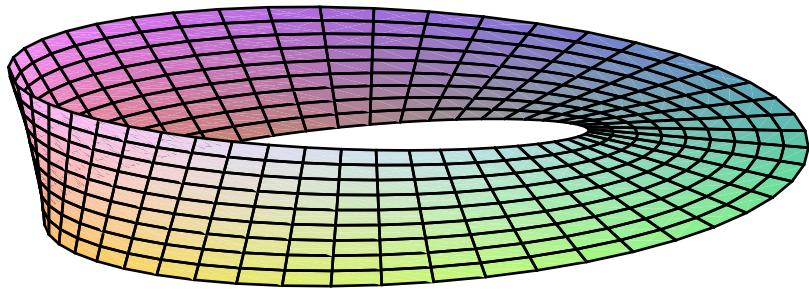
```
> plot3d(exp(-sin(Pi*z)),phi=0..2*Pi,z=-2.2..0,
          coords=cylindrical,grid=[40,50],orientation=[0,60]);
```



Suivant une syntaxe identique à celle des représentations en coordonnées sphériques (voir plus haut), on peut utiliser une liste qui autorise des constructions plus complexes. La première opérande de la liste est l'expression du rayon $\rho(u, v)$ fonction de deux paramètres u et v , la deuxième l'expression de l'angle $\varphi(u, v)$ et la troisième, l'ordonnée $z(u, v)$.

L'exemple ci dessous dessine une surface célèbre, dite ***ruban de Möbius*** (August Ferdinand Möbius, 1790-1868). On trouvera en fin de chapitre un exercice et des remarques évoquant les propriétés très particulières de cette surface.

```
> plot3d([3-u*sin(v/2),v,u*cos(v/2)],u=-1..1,v=0..2*Pi,
          coords=cylindrical,orientation=[70,35],grid=[10,50]);
```



Bibliothèque plots

La bibliothèque **plots** contient aussi des fonctions de représentation 3D dont on ne donnera que quelques exemples

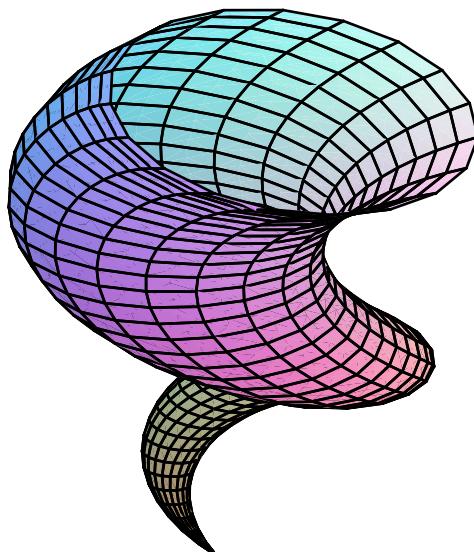
Options permanentes

Si l'on a beaucoup de dessins à faire il peut être souhaitable de rendre permanentes certaines options (pour la durée de la session). On peut réaliser cette opération avec la fonction **setoptions3d** de la bibliothèque **plots** (voir son équivalent 2D, **setoptions**, au chapitre 16). Lorsqu'une option est introduite explicitement dans une commande de **plot3d**, elle remplace, pour cette commande, celle indiquée dans **setoptions3d**.

Surfaces en coordonnées tubulaires

La fonction **tubeplot** de **plots** permet la représentation de surfaces équivalentes (topologiquement) à un "tube". A l'aide d'une liste de la forme $[x(t), y(t), z(t), radius = r(t)]$ on donne les coordonnées paramétriques $x(t), y(t), z(t)$ de la courbe de l'espace définissant "l'axe" du tube, la quatrième composante donnant, avec le mot clé **radius**, la loi de variation du rayon du tube. Il existe d'autres formes d'écriture permettant diverses variantes pour la construction de telles surfaces (voir le dernier exemple de ce chapitre et l'aide en ligne).

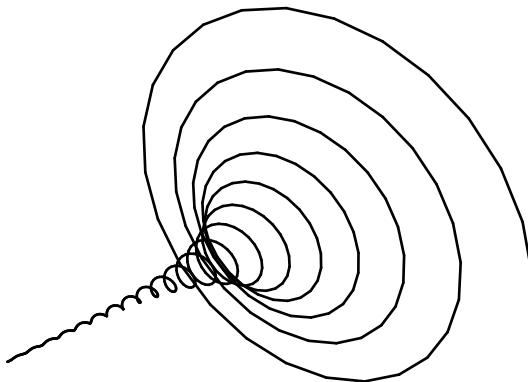
```
> plots[tubeplot]([sin(t), cos(t), 2*t, radius=0.25*(t-Pi),
                   t=Pi..4*Pi], tubepoints=20, orientation=[-60, 60]);
```



Courbes paramétriques

La fonction **spacecurve** de **plots** permet la représentation de courbes définies par une liste donnant les coordonnées paramétriques $[x(t), y(t), z(t)]$, t désignant un paramètre:

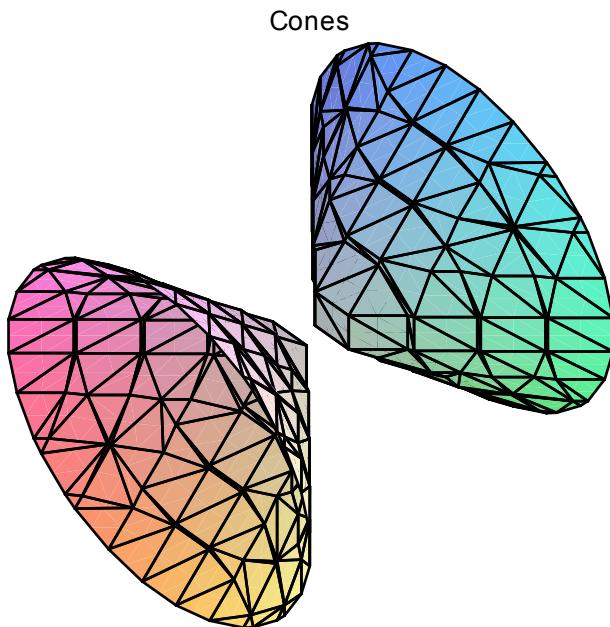
```
> plots[spacecurve]([t,exp(-t)*cos(20*t),exp(-t)*sin(20*t)],
    t=-Pi..Pi,color=black,
    numpoints=500,orientation=[40,60]);
```



Surfaces définies implicitement

La surface à tracer peut être définie implicitement par une équation $f(x, y, z) = g(x, y, z)$. On utilisera la fonction **implicitplot3d** de **plots**. Elle calcule $(f - g)(x, y, z)$ sur un maillage de l'espace 3D et cherche ensuite la surface en interpolant par des petites cellules planes les points proches de $(f - g)(x, y, z) = 0$. La finesse de la grille peut être modifiée par $grid = [nx, ny, nz]$, le défaut utilisant 1000 points : [10,10,10] (mais attention au temps de calcul !).

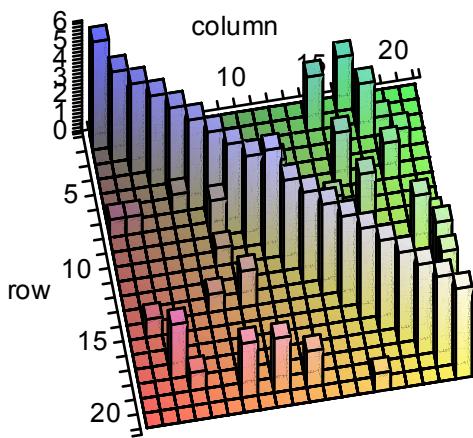
```
> plots[implicitplot3d](x^2-y^2=z^2,x=-2..2,y=-2..2,z=-2..2,
    scaling=constrained,title="Cones");
```



Visualisation d'une matrice

En plus des moyens décrits au chapitre 4, *Vecteurs, Matrices et Algèbre Linéaire*, pour visualiser les grandes matrices, on peut aussi utiliser une représentation sous forme d'histogramme qui permet une appréciation rapide de la forme de la matrice.

```
> M:=LinearAlgebra[RandomMatrix](20,20,density=0.1,generator=0..3) +  
    Matrix(20,20,shape=diagonal,fill=5);  
  
> plots[matrixplot](M, heights=histogram, orientation=[-10,20], axes=normal);
```



Surfaces animées

Il est possible, avec la fonction **animate3d** de la bibliothèque **plots**, d'animer des surfaces ou des courbes. On se reportera à l'aide en ligne relative à cette fonction ainsi qu'à la fonction similaire **animate** pour les animations 2D. Ici t joue le rôle de la variable d'animation.

```
> omega:=0.5;  
k:=0.3;  
plots[animate3d]([rho,phi,exp(-k*rho)*cos(2*Pi*(omega*rho + t))],  
    rho=0..10,phi=0..2*Pi,t=0..1,coords=cylindrical,  
    style=patchnogrid,frames=100);
```

On pourra aussi utiliser la fonction **display** de **plots** avec l'option **insequence=true** (voir chapitre 16).

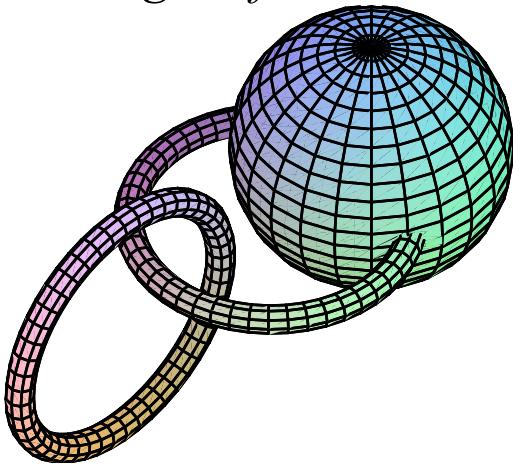
Dernier exemple sur l'utilisation de display de plots

Toutes les structures du type INTERFACE_PLOT3D peuvent être combinées sur un même graphe avec la fonction **display**. On remarquera l'écriture pour la fonction **tubeplot** : on définit ici un ensemble d'axes avec la même fonction **radius**.

Notez également les modifications de la police de caractères du titre, de son style et de sa taille.

```
> T:=plots[tubeplot]({[cos(t)+1,sin(t),0],[cos(t)+5/2,0,sin(t)]},  
    radius=0.1,t=0..2*Pi); # Tracé des  
anneaux  
S:=plot3d(1,phi=0..2*Pi,theta=0..Pi,coords=spherical); # Tracé de la  
sphère  
plots[display](S,T,scaling=constrained,  
    titlefont=[TIMES,ITALIC,20],title="Figure finale");
```

Figure finale



On est vraiment loin d'avoir épuisé avec ce chapitre les options et possibilités de MAPLE pour le graphisme. Aussi est-il conseillé d'explorer l'aide en ligne...

Exercice: Ruban de Möbius et bouteille de Klein.

A la fin du paragraphe relatif aux représentations en coordonnées cylindriques, on a représenté le **ruban de Möbius**. Cette surface est topologiquement équivalente à celle présentée sur la page de garde. Cette dernière peut être représentée par la réunion de quatre surfaces paramétriques dépendant de deux paramètres u et v

$$s_1 \Rightarrow x(u, v) = [5/2 + 3/2 \cos(v)]\cos(u), y(u, v) = [5/2 + 3/2 \cos(v)]\sin(u), z(u, v) = -5/2 * \sin(v)$$

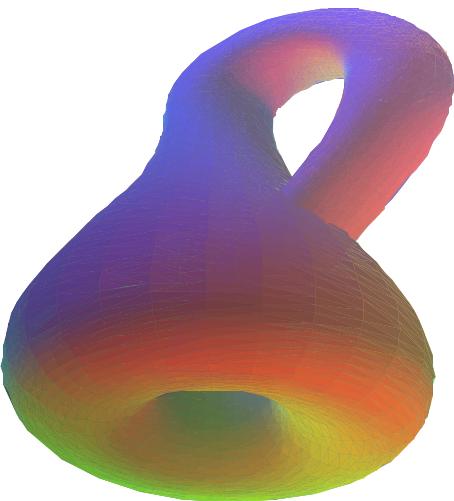
$$s_2 \Rightarrow x(u, v) = [5/2 + 3/2 \cos(v)]\cos(u), y(u, v) = [5/2 + 3/2 \cos(v)]\sin(u), z(u, v) = 3v$$

$$s_3 \Rightarrow x(u, v) = 2 - 2\cos(v) - \cos(u), y(u, v) = \sin(u), z(u, v) = 3v$$

$$s_4 \Rightarrow x(u, v) = 2 + [2 + \cos(u)]\cos(v), y(u, v) = \sin(u), z(u, v) = 3\pi + [2 + \cos(u)]\sin(v)$$

Les paramètres u et v varient respectivement dans les intervalles $[\pi/4, 3\pi/4]$ et $[0, \pi]$. Reproduire cette figure.

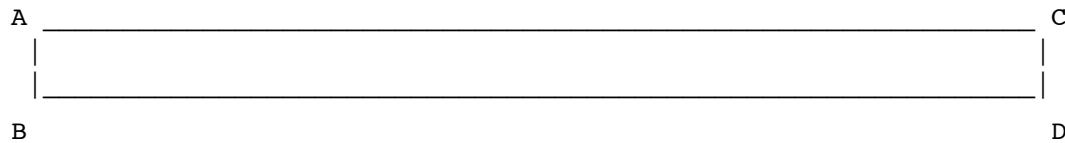
On prendra une grille de calcul de 20×20 et une orientation de $-35^\circ, 125^\circ$. Lorsque le paramètre u varie dans l'intervalle $[0, 2\pi]$ on obtient un volume appelé **bouteille de Klein**. Tracez ce volume et observez-le. Jouez avec les variations des paramètres pour observer "l'intérieur".



Récréation à propos du ruban de Möbius:

Pouvez-vous faire une remarque fondamentale sur cette surface, déjà tracée dans ce chapitre au paragraphe

Coordonnées cylindriques ? Non ? Alors prenez une feuille de papier et découpez un ruban semblable à celui-ci



Si on colle les deux extrémités du ruban en mettant C sur A et D sur B on obtient un banal ruban cylindrique dont on peut dire qu'il possède deux surfaces, l'intérieur et l'extérieur que l'on peut colorier avec deux couleurs distinctes. Maintenant collez le ruban en mettant C sur B et D sur A par un torsion d'un demi tour. La surface que vous obtenez est un ruban de Möbius. Et maintenant coloriez... Quelle conclusion en tirez-vous ? Observez attentivement la figure dessinée avec MAPLE et retrouvez cette propriété. Construisez avec MAPLE une bouteille de Klein et observez-là. Quelle propriété possède-t-elle ?

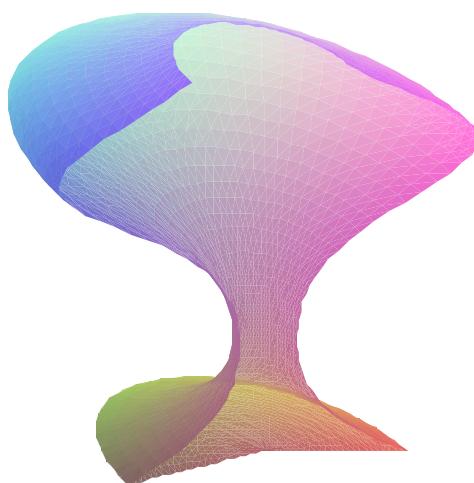
Si vous coupez suivant une circonférence un cylindre en deux vous obtenez évidemment deux cylindres moins hauts. Que devient le ruban de Möbius quand on lui applique cette opération et en prenant soin de couper longitudinalement en suivant une courbe toujours située au milieu de la largeur ? Observez bien le résultat.

Refaite un ruban de Möbius et coupez le longitudinalement par le tiers de sa largeur. Le résultat est plutôt intéressant...

Solution pour le "fond du vase"

On a "coupé" le vase en deux pour une meilleure vue.

```
> Vase:=plot3d(exp(-sin(Pi*z)),phi=0..Pi,z=-2.2..0,  
 coords=cylindrical,grid=[40,50]):  
 Fond:=plot3d((exp(-sin(-2.2*Pi))/0.1)*abs(z+2.1),phi=0..Pi,  
 z=-2.2..-2.1,coords=cylindrical):  
 plots[display](Vase,Fond,orientation=[-110,70],style=patchnogrid);
```



18 - Eléments du langage de programmation.

Symboles et chaînes de caractères.

Avertissement

Les exemples donnés dans ce chapitre sont volontairement très simples et n'ont qu'une valeur d'illustration. Certaines boucles pourraient être écrites plus simplement ou remplacées par des instructions uniques utilisant des fonctions comme **add**, **sum**, **product**, **seq**, **map**, etc. (on en donnera d'ailleurs quelques exemples).

Les boucles

1) Les boucles for

Une boucle permet la répétition du calcul d'une expression ou d'une suite d'expressions. Elle commence par le mot clé **for** suivi du nom de la variable de comptage (ici *i*). Celle-ci débute la boucle avec la valeur qui suit le mot clé **from**, puis est incrémentée de la valeur suivant le mot clé **by** et ceci tant qu'elle reste inférieure ou égale à la valeur suivant le mot clé **to**. L'expression ou les expressions sont placées entre les mots clés **do** et **end do** (ou **od**) et séparées par la ponctuation ; ou : . Ici il n'y a trois expressions. La première est une liste de deux nombres, la deuxième une mise en facteur premiers et la troisième l'affichage d'un symbole non assigné.

Remarque: en effet, `-----` est un symbole créé qui pourrait être utilisé (si l'on peut dire !), dans des calculs au même titre que *x* ou *y* (voir *Symboles et Chaînes de caractères* dans ce chapitre). Son usage est ici de pure commodité pour séparer les affichages de chaque tour de boucle.

```
> restart:  
for i from 0 by 2 to 7 do [i,i!];ifactor(i!);`-----`; end do;  
[0, 1]  
1  
-----  
[2, 2]  
(2)  
-----  
[4, 24]  
(2)3 (3)  
-----  
[6, 720]  
(2)4 (3)2 (5)  
-----
```

Attention: la variable de comptage n'est pas muette et vaut après exécution *la dernière valeur calculée avant le test qui provoque l'arrêt de la boucle* (ici 8 et non 6)

```
> i;
```

Attention: les valeurs définissant les bornes et le pas de la variable de comptage doivent être du type **numeric** (ou **character**, mais c'est d'une utilisation plus rare). Rappelons que des constantes symboliques comme π ou $\sqrt{5}$ sont du type **constant** mais pas du type **numeric**. La variable de comptage ne prend pas nécessairement des valeurs entières. Ainsi, dans l'exemple suivant, les valeurs prises par x sont fractionnaires et correctes mais l'utilisation du symbole π est incorrecte et il est nécessaire d'utiliser la fonction **evalf**

```
> type(Pi,constant), type(Pi,numeric);
                                         true, false

> for x from 0 by 3/4 to Pi/2 do cos(Pi*x) end do;
Error, final value in for loop must be numeric or character

> for x from 0 by 3/4 to evalf(Pi/2) do cos(Pi*x) end do;
                                         1
                                         -  $\frac{1}{2}\sqrt{2}$ 
                                         0
```

Les mots clés **from** et **by** sont optionnels. S'ils sont absents les valeurs associées prennent la valeur 1 par défaut.

```
> for i to 2 do i^3 end do;
                                         1
                                         8
```

Une boucle contient la plupart du temps plusieurs instructions mais, par souci de concision, les exemples que nous donnerons n'en contiendrons généralement qu'une.

Règles de syntaxe

Les boucles peuvent être écrites sur une même ligne comme précédemment, mais il sera de loin préférable d'utiliser une forme "aérée" mettant en évidence la structure de la boucle. Les instructions dans le corps de la boucle doivent être obligatoirement séparées indifféremment par " ; " ou " : ". Cependant **do** étant un mot clé, il n'est pas nécessaire de le faire suivre de " ; " ou " : ". De même, *l'instruction qui précède end do* peut ne pas être terminée par " ; " ou " : ".

La ponctuation qui suit **end do** est ici " ; " et quel que soit la ponctuation des fins d'instructions (: ou ;) que l'on utilisera, tous les résultats associés aux commandes du corps de boucle seront affiché.

L'écriture suivante utilise la frappe **Shift-Return** (ou **Shift-Enter**) à chaque fin de ligne pour passer à la suivante.

```
> L:=[1,-4,5,-7];
S_3:=0:
for i to nops(L) do
  S_3:=S_3+L[i]^3:
end do;
`Somme des cubes `= S_3;
```

$L := [1, -4, 5, -7]$
 $S_3 := 1$
 $S_3 := -63$

```

S_3 := 62
S_3 := -281
Somme des cubes = -281

```

Avec **Return** seul on obtient l'écriture suivante pour laquelle chaque ligne débute par un prompt. Le message d'avertissement (Warning) disparaîtra quand on donnera le mot clé **end do;** ou **end do:** ou encore **od;** ou **od:**. Il n'y a aucune différence sur le résultat entre les deux écritures (on peut d'ailleurs mélanger les deux formes).

```

> S_3:=0:
> for i to nops(L) do
> S_3:=S_3+L[i]^3:
>
Warning, premature end of input, use <Shift> + <Enter> to avoid this
message.

```

Attention: les mots **from**, **by**, **do**, etc. sont réservés et non assignables.

```

> 1+by;
Error, reserved word `by` unexpected

```

C'est la ponctuation qui suit les mots clés end do qui déterminera si les résultats intermédiaires de la boucle seront ou non affichés. Ici "end do :" au lieu de "end do ;" supprime les affichages des calculs dans la boucle.

```

> S_3:=0:
for i to nops(L) do
  S_3:=S_3+L[i]^3
end do:
`Somme des cubes `= S_3;
Somme des cubes = -281

```

On peut néanmoins souhaiter afficher certains résultats intermédiaires, lors de tests par exemple. On utilisera pour cela la fonction **print**

```

> S_3:=0:
for i to nops(L) do
  print(i,L[i],L[i]^3):
  S_3:=S_3+L[i]^3
end do:
`Somme des cubes `= S_3;
1, 1, 1
2, -4, -64
3, 5, 125
4, -7, -343
Somme des cubes = -281

```

On rappelle que ce ne sont que des exemples à caractère pédagogique. On peut faire beaucoup plus court... !

```

> `Somme des cubes `= add(i^3,i in L);
Somme des cubes = -281

```

2) Les boucles sans variable de comptage

Quand les instructions d'une boucle ne dépendent pas de la variable de comptage, celle-ci peut être omise ainsi que le mot clé **for**. Comme le comptage peut toujours commencer à 1, le mot clé **from** peut aussi être omis. Voici un exemple pour la solution numérique de l'équation $\cos(x) = x$ par la méthode de *Newton-Raphson* et pour laquelle on suppose que 4 itérations suffisent.

```
> x:=1.0:  
  to 4 do  
    x:=x+(cos(x)-x)/(sin(x)+1):  
  end do;  
                                              x := 0.7503638679  
                                              x := 0.7391128909  
                                              x := 0.7390851334  
                                              x := 0.7390851332
```

Exercice: vérifiez ce résultat avec une fonction de MAPLE

>

3) Les boucles for-while

On peut également construire des boucles **for** avec le mot clé **while** suivi d'une condition logique (expression booléenne). Le mot anglais **while** signifiant "tant que", la boucle sera effectuée tant que la condition demandée est vraie. Les opérateurs logiques (booléens) sont

- $a = b$ \rightarrow true si a égal à b
- $a < b$ \rightarrow true si a plus petit que b
- $a \leq b$ \rightarrow true si a plus petit ou égal à b
- $a \neq b$ \rightarrow true si a différent de b

Les opérateurs $=>$ et $=<$ ne sont pas admis. On peut aussi utiliser les opérateurs $>$ et \geq bien que MAPLE ne les utilise pas. Remarquez la façon dont est ré-écrite l'expression de comparaison suivante. On rappelle également que la fonction **is** donne une des réponses **true**, **false** ou **FAIL** en fonction de la condition exprimée en argument (voir chapitre 13)

```
> 3 > 2;  
                                              2 < 3
```

```
> is(Pi>2);  
                                              true
```

Les autres opérateurs sont

- $a \text{ and } b$ \rightarrow true si a et b sont des expressions logiques vraies
- $a \text{ or } b$ \rightarrow true si a ou b ou a et b sont des expressions logiques vraies
- $\text{not } a$ \rightarrow true si a est une expression logique fausse et réciproquement

Dans l'exemple qui suit on somme les termes de la liste jusqu'à ce que l'on rencontre un élément dont le module est supérieur ou égal à 6

```
> Somme:=0:  
  for i while i<=nops(L) and abs(L[i])<6 do  
    Somme:=Somme+L[i]  
  end do:  
  L; Somme;
```

[1, -4, 5, -7]

2

Exercice : Expliquez l'erreur suivante

```
> Somme:=0:  
  for i from 1 while L[i]>-8 and L[i]<10 do  
    Somme:=Somme+L[i]  
  end do:  
  Somme;  
  
Error, invalid subscript selector
```

-5

On peut également construire une boucle avec seulement le mot clé **while** mais l'utilisateur doit alors gérer lui-même le comptage de la boucle (interprétez le résultat).

```
> x:=5:  Somme:=0:  
  while x <> 0 do  
    Somme:=Somme+ln(x):  
    x:=x-1  
  end do:  
  Somme;
```

$\ln(5) + 3 \ln(2) + \ln(3)$

4) Les boucles commandées par des opérandes

Une boucle **for ... end do** : peut aussi être commandée par les **opérandes** d'une liste ou de tout autre objet avec le mot clé **in**. Ici le nom *k* prend successivement les valeurs de la liste (que font ces deux boucles ? Trouvez plus simple).

```
> L:=[3,2,5,7,11]:  
  Imp:=true:  
  for k in L do  
    Imp:=Imp and is(k,odd)  
  end do:  
  Imp;
```

false

```
> Isp:=true:  
  for k in L do  
    Isp:=Isp and isprime(k)  
  end do:  
  Isp;
```

true

Exercice : interprétez le résultat suivant:

```
> x:='x':  Poly:=3*x^2+2*x+1;  
  P:=NULL:  
  for e in Poly do  
    P:=P,int(e,x)  
  end do:  
  [P];
```

$Poly := 3x^2 + 2x + 1$

$$[x^3, x^2, x]$$

5) Contrôle du déroulement d'une boucle

Si l'on souhaite arrêter une boucle sur une condition on peut utiliser le mot clé **break** (casser). Ainsi la somme suivante est arrêtée dès qu'un élément de la liste est négatif ou nul. On introduit dans cette boucle la notion de bloc conditionnel **if...end if** qui sera détaillée plus loin dans ce chapitre.

```
> Somme:=0:  
  for i in [3,2,5,-1,6,-2] do  
    if i <= 0 then  
      break  
    end if;  
    Somme:=Somme+ln(i)  
  end do:  
Somme;
```

$$\ln(3) + \ln(2) + \ln(5)$$

On peut aussi sauter des instructions d'une boucle sur une condition : on utilisera le mot clé **next** (suivant). La boucle reprend alors avec la première instruction de la boucle avec la valeur suivante du compteur. La somme ne porte ici que sur les logarithmes des valeurs positives de la liste

```
> Somme:=0:  
  for i in [3,2,5,-1,6,-2] do  
    if i <= 0 then  
      next  
    end if;  
    Somme:=Somme+ln(i)  
  end do:  
Somme;
```

$$\ln(3) + \ln(2) + \ln(5) + \ln(6)$$

Exercice: trouver une écriture un peu plus simple (qui n'utilise pas **next**) pour l'exemple précédent.

>

Exercice: analysez cette commande:

```
> add(ln(x), x in select(is,[3,2,5,-1,6,-2],positive));  
                                         ln(3) + ln(2) + ln(5) + ln(6)
```

6) Les boucles do-break

Une boucle peut se construire avec seulement les mots clés **do** et **end do** mais l'utilisateur doit alors gérer lui-même le comptage de la boucle et l'utilisation du mot clé **break** est impérative sous peine de voir la boucle s'exécuter indéfiniment...

```
> Somme:=0:  
  i:=1:  
  do  
    if i > 7 then  
      break  
    end if:  
    Somme:=Somme+i;  
    i:=i+1:  
  end do:
```

Somme;

28

C'est-à-dire, tout simplement...

> **Sum(n, n=1..7) :% =value(%);**

$$\sum_{n=1}^7 n = 28$$

Exercice: ...mais attention ! Expliquez pourquoi

> **sum(i, 'i'=1..7);**

56

Branchements conditionnels : bloc if

Les blocs conditionnels **if** ont une structure semblable à tous ceux des autres langages de programmation. Il faut seulement savoir que les mots anglais **if**, **then** et **else** signifient respectivement "si", "alors" et "sinon". Le mot clé **elif** est une contraction de "else if" qui signifie "sinon si". Comme **if**, **elif** est suivi d'une condition logique puis de **then**. Un bloc **if** se ferme avec **end if** (ou **fi**) et tout bloc ouvert doit être fermé. Deux blocs **if** peuvent être imbriqués (inclus) l'un dans l'autre mais ne doivent pas se "chevaucher". Seuls les éléments **if...then...end if;** sont indispensables. Les autres éléments de structure **else** et **elif** ne seront utilisés qu'en fonction des besoins. Comparer en les analysant les résultats suivants

```
> x:=-2:  
  if x > 1 then  
    y:=x-1;  
  elif x < -1 then  
    y:=x+1;  
  else  
    y:=0;  
  end if;
```

y := -1

```
> x:=-1/2:  
  if x > 1 then  
    y:=x-1;  
  elif x < -1 then  
    y:=x+1;  
  else  
    y:=0;  
  end if;
```

y := 0

Attention: Comme pour les boucles, les opérateurs de comparaison standard n'agissent qu'entre des valeurs de type **numeric** (ou **character**).

Le message "cannot determine..." signifie que MAPLE n'a pas pu effectuer une opération de type logique (ici la comparaison $\sqrt{3} > 1$).

```

> x:=sqrt(3):
  if x > 1 then
    y:=x-1;
  elif x < -1 then
    y:=x+1;
  else
    y:=0;
  end if;
Error, cannot determine if this expression is true or false: 1 < 3^(1/2)

```

On utilisera alors la fonction **is**

```

> if is(x > 1) then
  y:=x-1;
elif is(x < -1) then
  y:=x+1;
else
  y:=0;
end if;

```

$$y := \sqrt{3} - 1$$

ou la fonction **evalf**

```

> if evalf(x) > 1 then
  y:=x-1;
elif evalf(x) < -1 then
  y:=x+1;
else
  y:=0;
end if;

```

$$y := \sqrt{3} - 1$$

On notera également que **if** peut être vu comme une fonction, ce qui conduit à une écriture simplifiée pour une condition *if-else*.

```

> x:=3:
z:='if`(x>2,x+Pi,x-Pi);
x:=1:
z:='if`(x>2,x+Pi,x-Pi);

```

$$z := 3 + \pi$$

$$z := 1 - \pi$$

Symboles et chaînes de caractères

1) Définitions

Au chapitre 2 nous avons déjà rencontré la notion de symbole et de chaîne de caractères :

- Une chaîne de caractères entourée par deux caractères ` définit un objet de type **symbol**.
- Une chaîne de caractères entourée par deux caractères " définit un objet de type **string** (string = chaîne). Le caractère " est un caractère unique et non la répétition de '

```
> restart:
```

```

whattype(`Valeur de x`);
type(`Valeur de x`,symbol);
is(`Valeur de x`,symbol);
symbol
true
true

> Ch:="Chaine de caracteres";
whattype(Ch);
type(Ch,string);
Ch := "Chaine de caracteres"
string
true

```

Pour introduire le caractère " dans une chaîne ou un symbole on l'écrira soit \"", soit, par doublement, """". En raison du rôle de "continuation" joué dans MAPLE par le caractère \ on l'écrira \\ pour le faire apparaître et lui supprimer son rôle. Les chaînes ou symboles *isolés* obtenues par doublements "" ou `` constituent par contre des chaînes et des symboles vides.

```

> C:="Ceci est une \"Chaine de caractères""...\\..."; 
C := "Ceci est une "Chaine de caractères"...\\..."

> S:=`Symbole \"\\Spécial\\``;
S := Symbole "Spécial\"

```

Attention: d'un système d'exploitation d'ordinateur à un autre, les caractères accentués peuvent provoquer des difficultés.

Un objet de type *symbol* peut être utilisé comme un nom valide à condition toutefois que les caractères soient entourés des deux caractères `. Une chaîne de caractères n'a pas cette propriété et ne peut pas être utilisée comme symbole.

```

> type(`2-ième valeur de x`,name);
type(`2-ième valeur de x`,symbol);
`2-ième valeur de x`:=1;
exp(`2-ième valeur de x` + 1);
true
true
2-ième valeur de x := 1
 $e^2$ 

```

Si la suite des caractères qui définit un symbole ne contient ni espace ni caractères spéciaux (sauf "_") et ne commence pas par un chiffre, il n'y aura aucune ambiguïté de syntaxe et les caractères ` deviennent facultatifs: on se trouve alors dans la situation habituelle. Un nom ordinaire possède les types *name* et *symbol*, mais un *nom indexé* n'est pas un symbole.

```

> z_0 , type(z_0,name), type(z_0,name(indexed)) , type(z_0,symbol);
z[0], type(z[0],name), type(z[0],name(indexed)), type(z[0],symbol);
z_0, true, false, true
Z0, true, true, false

```

Rébus et exercice: si tous les objets de type **symbol** peuvent servir de noms de variables, il n'est pas souhaitable d'abuser de cette possibilité ! Pouvez-vous décrypter ce calcul, au demeurant parfaitement exact et réalisé sans trucage? (non, ce n'est pas un jeu extrait du Journal de Mickey pour étudiants du supérieur...)

```
> assume(`^o`>0): Int(`#@`^`^o`, `#@`=`/`..`+`):% = value(%);
```

$$\int_{\text{+}}^{\text{+}} \# @ \circ \sim d \# @ = - \frac{(\circ \sim + 1)}{\circ \sim + 1}$$

2) Fonctions de recherche

MAPLE contient des fonctions permettant de manipuler les chaînes de caractères ou les symboles. D'abord la fonction **length** (longueur) qui renvoie le nombre de caractères d'une chaîne ou d'un symbole, espaces compris.

```
> c := "x = Y+3";
length(c);
length(`valeur de x`);
```

c := "x = Y+3"

On peut adresser les caractères d'une chaîne comme pour une liste ou une suite

```
> c[3];
c[3..6];
c[-3..-1];
          "≡"
          "= Y+"
          "Y+3"
```

Exercice: mais ça ne fonctionne pas avec les symboles. Pourquoi ?

```
> X:=`Ab*c De`; is(%,name),is(%,symbol);
X[2..4];           is(%,name),is(%,symbol);
X:=Ab*c De
true, true
Ab*c De2 .. 4
true, false
```

La fonction **substring** extrait une sous chaîne de caractères d'une chaîne ou d'un symbole en donnant sous la forme d'un intervalle la position du premier et du dernier caractère. Etant donné la possibilité d'adresser directement les éléments d'une chaîne, cette fonction est surtout utile pour les symboles.

```
> substring(c,3..6);
substring(X,2..3);
substring(Xyztu,2..-2);
                           " = Y+
                                         b*
                                         vzt
```

La fonction **searchtext** détermine la position du premier caractère d'une sous chaîne dans une chaîne ou un symbole. Cette fonction ne distingue pas les majuscules des minuscules. On remarque également que cette fonction ne fait pas de distinction entre chaînes de caractères et symboles et seul l'aspect traitement des caractères est retenu.

```
> c;
  searchtext("x ",c);
  searchtext(`y+`,c);
  searchtext(yz,Xyztu);
                                         "x = Y+3"
                                         1
                                         5
                                         2
```

Si la sous chaîne n'est pas présente *dans sa totalité*, la fonction renvoie la valeur 0.

```
> searchtext(`Y+4`,c);
                                         0
```

La fonction **SearchText** est la même que la précédente mais *fait la distinction* entre majuscules des minuscules

```
> SearchText(`Y+3`,c);
  SearchText(`y+3`,c);
                                         5
                                         0
```

3) La bibliothèque StringTools

On prendra garde de ne pas réinventer la roue en cherchant parmi les 153 fonctions de la bibliothèque **StringTools** dont on donne quelques exemples (on utilise ici l'une ou l'autre des deux formes "longues" d'appel, mais on peut aussi utiliser **with**)

```
> StringTools[Reverse]("12345");
                                         "54321"

> StringTools:-UpperCase("ab C,12x");
                                         "AB C,12X"

> L:=StringTools[Explode]("AB deF");
                                         L:=[A", "B", " ", "d", "e", "F"]
```

La fonction s'applique aussi aux symboles

```
> StringTools[Explode](`XY Z`);
                                         ["X", "Y", " ", "Z"]
```

Attention: sans préciser la chaîne de liaison (qui peut être vide), **Join** rajoute par défaut un espace entre les chaînes

```
> StringTools:-Join(L);
  StringTools:-Join(L,""); équivalent à > cat(op(L)); voir plus loin
  StringTools:-Join(L,"//");
```

```
"A B d e F"
"AB deF"
"A//B// //d//e//F"
```

On sélectionne les majuscules d'une chaîne par une combinaison de deux fonctions

```
> StringTools[Select](StringTools:-IsUpper, "AbcDe, *fH");
"ADH"
```

TrimLeft et **TrimRight** éliminent respectivement les espaces de début et de fin de chaîne (Left=gauche, Right=droite)

```
> StringTools:-TrimLeft(" A bc d ef      ");
" A bc d ef      "
> StringTools:-TrimRight("      A bc d ef      ");
"      A bc d ef"
```

4) Conversion

La fonction **convert** permet de passer d'une chaîne à un symbole et réciproquement

```
> convert(`aB_1`,string);
convert("aB_1",symbol);
"aB_1"
aB_1
```

ou de transformer un objet en chaîne de caractères ou en symbole

```
> p:=x^3-2*x+1;
convert(p,symbol), convert(p,string);
p := x3 - 2 x + 1
x3-2*x+1, "x3-2*x+1"
```

On peut aussi générer des chaînes de caractères en transformant des entiers suivant l'ordre ASCII. On rappelle qu'avec cette convention, à 32 (décimal) correspond un espace, à 64 correspond @, à 65 correspond A, etc.

```
> convert([66,111,110,106,111,117,114,32,33],bytes);
"Bonjour!"
```

Et réciproquement

```
> convert("Bonjour !",bytes);
[66, 111, 110, 106, 111, 117, 114, 32, 33]
```

On rappelle également les exemples donnés à la fin du chapitre 1.

```
> convert(MDXV,arabic), convert(2004,roman);
1515, "MMIV"
```

5) La fonction parse

Cette fonction permet de convertir en une expression, un symbole ou une chaîne de caractères comme si ces derniers avaient été entrés dans la zone "input" des commandes, mais, par défaut, sans procéder à l'évaluation. Il va de soi que le contenu de la chaîne ou du symbole doit être syntaxiquement correct.

```
> s:="x-3=0";
```

```

parse(s);
 $s := "x-3=0"$ 
 $x - 3 = 0$ 

> C:=1;
S:=`C+diff(x^3-2*x+1,x)+int(x,x=0..1)`;
parse(S);
 $S := C + \text{diff}(x^3 - 2x + 1, x) + \text{int}(x, x=0..1)$ 

$$C + \left( \frac{d}{dx} (x^3 - 2x + 1) \right) + \int_0^1 x \, dx$$


```

L'expression n'a pas été évaluée, ce que l'on fait maintenant

```

> %;

$$-\frac{1}{2} + 3x^2$$


```

Si l'on veut que le décodage soit considéré comme une véritable commande (et donc évaluée), il suffit de rajouter le mot clé **statement** en argument.

```

> parse(S, 'statement');

$$-\frac{1}{2} + 3x^2$$


```

Exercice: on veut transformer la chaîne suivante en une suite telle que $s := A, x, 0, s, t$. Expliquer la commande. Trouvez une autre solution. Utilisez la fonction **cat** ci-dessous pour passer de s à Ch .

```

> Ch:="AxOut":
s:=op(map(parse, StringTools[Explode](Ch)));
 $s := A, x, 0, u, t$ 

```

6) Concaténation

a) La fonction **cat**

On appelle concaténation la juxtaposition de chaînes de caractères ou de symboles. Cette opération est obtenue grâce à la fonction **cat**, les arguments étant des chaînes de caractères, des symboles ou des valeurs numériques:

```

> i:=3:
s1:="La valeur de j";
cat(s1," vaut ",2*i-1);
 $s1 := \text{"La valeur de j"}$ 
 $\text{"La valeur de j vaut 5"}$ 

```

Le même exemple avec des symboles

```

> s1:="La valeur de j";
cat(s1,` vaut `,2*i-1);
 $s1 := \text{La valeur de j}$ 
 $\text{La valeur de j vaut 5}$ 

```

Si les arguments sont de types différents, c'est **le type du premier argument** (chaîne ou symbole) qui détermine le type du résultat. Notez l'équivalence de *i* et `i` qui sont évalués alors que 'i' ne l'est pas par définition.

```
> cat("A",3,u, i );
  cat("A",3,u,`i`);
  cat("A",3,u,'i');
  cat(`A`,3,u,"i");
                                         "A3u3"
                                         "A3u3"
                                         "A3ui"
                                         A3ui
```

b) L'opérateur ||

On peut aussi utiliser l'opérateur || (doublement de la barre verticale) qui, comme nous le verrons, a un comportement particulier.

```
> s:="chaînes":
  "deux "||s;
z:=u||(i-3);
                                         "deux chaînes"
                                         z := u0

> "deux " || s || " plus une";
                                         "deux chaînes plus une"
```

Le type du résultat est déterminé par le type de la première des deux opérandes

```
> C:="deux "||`chaînes`||" plus une";
  whattype(C);
S:={`deux `||"chaînes"||" plus une";
  whattype(S);
                                         C := "deux chaînes plus une"
                                         string
                                         S := deux chaînes plus une
                                         symbol
```

mais attention: la première des deux opérandes, contrairement à **cat**, n'est *jamais évaluée* qu'elle soit assignée ou non. Ce comportement est *déliberé* (voir ci dessous).

```
> s:="2":
  C:=s || " chaines";
S:={`chaine ` || s;
                                         C := s chaines
                                         S := chaine 2
```

De plus, *l'opérateur || est autorisé à gauche du signe d'affectation* :=. Ceci permet de générer simplement et automatiquement des noms de variables. Une valeur entière positive ou nulle à droite de l'opérateur est évaluée et le résultat prend valeur de symbole. On comprend pourquoi *x* ne doit pas être évalué pour cet opérateur

```
> x:=4;
i:=2;
x||i:=x-1;# x à gauche de || n'est pas évalué
```

x := 4

i := 2

x2 := 3

```
> x||(i-2)^2; # Ces deux écritures sont différentes
x||i-2^2;
```

*x_0*²

x_2 - 4

Par exemple on veut donner à chacun des termes d'une liste un nom *Ln* où *n* est le numéro d'ordre de l'élément:

```
> L:=[-1,3,-4];
for i from 1 to nops(L) do
  L||i:=L[i];
end do;
```

L := [-1, 3, -4]

L1 := -1

L2 := 3

L3 := -4

Maintenant les noms *L1*, *L2* et *L3* existent et sont assignés

```
> L2=L[2],L3=L[-1];
3 = 3, -4 = -4
```

Enfin il existe d'autres possibilités utilisant les notions d'intervalles numériques ou d'intervalles de caractères. Ces derniers sont construits suivant l'ordre de la table ASCII.

```
> U||(0..5);
"A"||(6.."B");
`A`||(a..."h");
U0, U1, U2, U3, U4, U5
"A6", "A7", "A8", "A9", "A:", "A;", "A<", "A=", "A>", "A?", "A@", "AA", "AB"
Aa, Ab, Ac, Ad, Ae, Af, Ag, Ah
```

Exercice : Sans compter les caractères, remplacez dans la phrase suivante le mot " est " par " n'est pas "

```
> s:="La manipulation des chaines est tres simple avec MAPLE";
s := "La manipulation des chaines est tres simple avec MAPLE"
```


19 - Procédures et modules

Librairies

Procédures

Une procédure est un enchaînement d'instructions permettant de calculer un résultat en fonction d'une suite d'arguments. Elle commence par le mot clé **proc(arguments)**, se termine par **end proc** (ou **end**) et on lui affecte généralement un nom. Construisons une procédure simple G , pour définir une fonction telle que

$$G(x) = 1 \text{ si } x < 0$$

$$G(x) = \cos(x) \text{ si } x \geq 0$$

On rappelle que cette fonction peut aussi être construite en utilisant la fonction **piecewise** (voir le chapitre 7).

```
> restart;
```

```
G:=proc(x) if evalf(x) < 0 then 1 else cos(x) end if end proc;  
G := proc(x) if evalf(x) < 0 then 1 else cos(x) end if; end proc;
```

```
> G(-1),G(0),G(Pi/4);
```

$$1, 1, \frac{1}{2}\sqrt{2}$$

La procédure est ré-écrite sur la feuille de calcul sauf si les mots clés **end proc** sont suivis par ":" au lieu de ";"

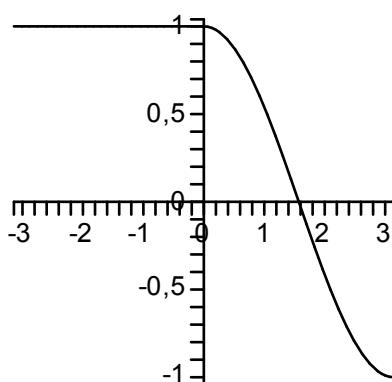
```
> G:=proc(x) if is(x < 0) then 1 else cos(x) end if end proc;
```

Rappel : Cette fonction peut être tracée avec la fonction **plot** mais **attention**, la notation de type opérateur doit être utilisée (pas d'argument pour G et pas de nom de variable pour l'intervalle). Ainsi la commande

```
> plot(G(x),x=-Pi..Pi);
```

ne serait pas correcte.

```
> plot(G,-Pi..Pi,color=black);
```



Mais on peut aussi écrire

```
> plot('G(x)',x=-Pi..Pi,color=black);
```

Cette syntaxe permet de tracer des procédures à **plusieurs variables**, disons par exemple $H(a, x, b)$

```
> plot('H(2*Pi/3,z,0.65)',z=0..Pi,color=blue);
```

Ponctuation et style d'écriture

On aura toujours intérêt à éviter d'écrire les instructions sur une même ligne. *On préférera une écriture mettant en évidence la structuration.*

En règle générale, une ligne se termine par une ponctuation ":" ou ";". Si une ligne se termine par un mot clé tel que **then**, **else**, **else if**, **do** ou si elle précède un des mots clés **else**, **else if** ou **elif**, **end if** ou **end do** la ponctuation est optionnelle. Les lignes contenant les mots clé de fermeture **end do** et **end if** seront systématiquement terminées par ":" ou ";", sauf peut-être celle qui précède **end proc:**. Si ces règles obéissent à une logique, on peut s'en dispenser et terminer toutes les lignes par une ponctuation pour être sûr de ne pas se tromper !

```
> G:=proc(x)
  if evalf(x) < 0 then;
    1;
  else;
    cos(x);
  end if;
end proc;
```

L'écriture précédente utilise la frappe **Shift-Return** à chaque fin de ligne pour passer à la suivante. Avec **Return** seul on obtient l'écriture suivante pour laquelle chaque ligne débute par un prompt. Le message d'avertissement (Warning) disparaîtra quand on donnera les mots clés **end proc:** ou **end proc;**. Il n'y a aucune différence sur le résultat entre les deux écritures (on peut d'ailleurs mélanger les deux formes).

```
> G:=proc(x)
> if evalf(x) < 0 then;
>
>
Warning, premature end of input, use <Shift> + <Enter> to avoid this message.
```

Contenus des procédures

Pour connaître le contenu d'une procédure il suffit d'écrire

```
> eval(G);
proc(x) if evalf(x)<0 then 1 else cos(x) end if; end proc;
```

On peut ainsi lire le contenu de certaines procédures internes de MAPLE. Il faudra toutefois exécuter au préalable la commande

```
> interface(verboseproc=2):
```

La variable **verboseproc** fixe le niveau des informations restituées. Notons au passage qu'il existe 34 variables de ce type qui permettent d'effectuer divers contrôles sur l'interface utilisateur (voit **interface** dans l'aide en ligne). Par exemple on pourra écrire

```
> interface(verboseproc=2):
#eval(exp); # Enlever le # pour exécuter la commande.
```

On n'a pas voulu afficher ici cette procédure qui est un peu longue. Avec de la réflexion et en possédant bien les éléments déjà présentés dans ce manuel, il est déjà possible d'en comprendre une bonne partie. On remarquera une utilisation très fréquente de la fonction **type**. On notera également que la fonction se redéfinie elle-même par $\exp(x)$ en fonction de la nature de l'argument, ce qui provoque des appels récursifs. L'écriture

'exp'(x) signifie au contraire que la fonction ne doit pas être évaluée par un appel récursif.

Seules peuvent être lues les procédures écrites en langage MAPLE. Certaines procédures sont écrites en langage C, sont compilées et ne peuvent pas être lues (option **builtin**)

```
> eval(evalf);
eval(`+`);

proc() option builtin, remember; 102 end proc;
proc() option builtin; 145 end proc;
```

On revient maintenant à un niveau standard d'informations.

```
> interface(verboseproc=1):
```

Avertissement

Les exemples qui suivent sont destinés à mettre en évidence point par point les éléments de construction et le fonctionnement d'une procédure. On a délibérément construit des exemples élémentaires qui seront souvent d'une utilité très marginale (ou même nulle!) et/ou remplaçables par de simples commandes. On ne cherchera donc pas quel intérêt, autre que pédagogique, ils pourraient présenter. Dans la réalité les procédures sont destinées à traiter des processus de calcul plus complexes.

Principes de fonctionnement et résultats rendus par les procédures

Une procédure est un opérateur qui renvoie toujours **un résultat unique** en fonction de un ou plusieurs arguments. **Attention**: quand on parle de "un" résultat renvoyé, il peut s'agir d'un objet MAPLE comme une suite, une liste, un vecteur, un ensemble, etc., contenant eux-mêmes plusieurs composantes. Le résultat peut aussi être une procédure (nous en donnerons un exemple dans un paragraphe de ce chapitre; voir § *Génération d'une procédure par une procédure*).

Le résultat d'une procédure est celui de la **dernière instruction effectuée** (qui n'est pas nécessairement la dernière instruction de la procédure!). **Attention**: "dernière instruction" et pas nécessairement "dernière évaluation" ou "dernière assignation". La procédure suivante est supposée avoir pour arguments une liste L de constantes de type *numeric* et un nombre (ou une variable) n . On voit que la boucle va explorer un à un les éléments de la liste et évaluer $L[i]+n$ ou $L[i]-n$ suivant le signe de l'élément $L[i]$. Aussi le résultat renvoyé par la procédure (d'un intérêt très relatif !) sera $L[\text{nops}(L)]+n$ ou $L[\text{nops}(L)]-n$ en fonction du signe de la valeur évaluée pour la dernière valeur de la liste. *Les autres évaluations auront été faites pour rien.*

Nous reviendrons ultérieurement sur le message "Warning..." renvoyé par MAPLE

```
> F:=proc(L,n)
  for i from 1 to nops(L) do
    if evalf(L[i]) < 0 then
      L[i]-n;
    else
      L[i]+n;
    end if
  end do
end proc:
```

Warning, `i` is implicitly declared local to procedure `F`

Le résultat de l'exécution de cette procédure correspond donc à la dernière évaluation, $-5-k$, correspondant à la valeur -5 de la liste.

```
> M:=[-3,0,1,-5];
F(M,k);
M := [-3, 0, 1, -5]
-k
```

Sans doute, ce que l'on cherche à obtenir est plutôt la liste $[-3 - k, k, 1 + k, -5 - k]$. On verra comment procéder, mais en attendant on a écrit une procédure qui ne présente strictement aucun intérêt sinon celui de montrer comment *le résultat est associé à la dernière instruction exécutée*. Elle calcule en fait le résultat de l'expression suivante (*Rappel*: à -1 correspond `nops(M)`).

```
> M[-1]+sign(M[-1])*k;
-k
```

Par défaut MAPLE affiche le résultat de la procédure. Si elle est utilisée dans une expression, le résultat est utilisé pour évaluer l'expression

```
> exp(-F(M,k)+y)-x^2;
e^(5+k+y) - x^2
```

On peut demander l'impression de certains résultats intermédiaires avec la fonction **print** pendant l'exécution d'une procédure

```
> F_prt:=proc(L,n)
  for i from 1 to nops(L) do
    if evalf(L[i]) < 0 then
      L[i]:=n;
      print(-n,L[i])
    else
      L[i]:=n;
      print(n,L[i])
    end if;
  end do;
end proc:
```

```
Warning, `i` is implicitly declared local to procedure `F_prt`
```

Exercice: examinez bien les résultats affichés. Quel est *le* résultat de **F_prt**? Pour une aide à cette réponse on se reportera à l'exemple suivant.

```
> F_prt(M,k);
-k, -3
k, 0
k, 1
-k, -5
```

Voici un autre exemple qui illustre, à travers une erreur, le principe de "dernière instruction exécutée". Dans ce qui suit on voit, avec l'impression des résultats intermédiaires, que la procédure est bien exécutée avant d'être transmise à la fonction **exp**. Cependant la dernière instruction exécutée, et donc le "résultat" de la procédure, est un ordre **print** et non un résultat symbolique. Il en résulte que l'exponentielle ne reçoit aucun argument (got 0)

```
> exp(F_prt(M,k));
```

```

-k, -3
k, 0
k, 1
-k, -5

```

```
Error, (in exp) expecting 1 argument, got 0
```

Exercice: pouvez-vous modifier la procédure **F_prt** pour qu'elle affiche toujours les résultats intermédiaires mais ne provoque pas ce message d'erreur ?

On veut maintenant construire une procédure donnant la liste $[-3 - k, k, 1 + k, -5 - k]$. On va, par exemple, créer une liste interne, **res**, remplie de 0 dans le but de lui donner le bon nombre d'arguments. On substituera ensuite les valeurs calculées à ces 0 (*ce n'est pas la meilleure solution*; voir plus loin). Quelque soit le signe du dernier élément de la liste la dernière évaluation sera associée à **res:=...** et le résultat sera la liste souhaitée. *Notez* que le résultat renvoyé par *F* n'est pas *res* mais *subsop(...)*.

```

> F:=proc(L,n)
  res:=[seq(0,i=1..nops(L))];
  for i from 1 to nops(L) do
    if evalf(L[i]) < 0 then
      res:=subsop(i=L[i]-n,res);
    else
      res:=subsop(i=L[i]+n,res);
    end if;
  end do;
end proc:

```

```
Warning, `res` is implicitly declared local to procedure `F`
```

```
Warning, `i` is implicitly declared local to procedure `F`
```

Il faut remarquer que malgré l'assignation à la variable *res* le résultat est une liste anonyme. De plus ni *i* ni *res* ne sont assignées en dehors du corps de la procédure. Comme le signale MAPLE, *res* et *i* sont des variables qui sont déclarées implicitement comme locales, c'est-à-dire connues seulement dans le corps de la procédure. Nous verrons que l'on peut déclarer explicitement ces variables internes, ce qui supprimera les messages. Insistons encore sur le fait que le résultat d'une procédure est celui de la dernière instruction exécutée et pas nécessairement celui de la dernière évaluation ou assignation.

```

> M;
F(M,k);
res;
i;
[-3, 0, 1, -5]
[-3 - k, k, 1 + k, -5 - k]
res
i

```

Arguments des procédures. Variables locales et globales

Par la suite nous appellerons **arguments formels** les noms de variables mis entre parenthèses pour la construction de la procédure. On appellera **arguments actuels**, ceux que l'on donne au moment de l'appel et qui prennent la place des arguments formels.

Les arguments actuels sont évalués avant d'être transmis à une procédure (sauf spécification par une mise entre ' des arguments actuels ou utilisation du pseudo type *uneval* pour les arguments formels; voir plus loin).

```
> F([-3+4, 'sin(Pi/4)', 0, eval(diff(ln(x), x), x=2)], k);
```

$$\left[1 + k, \sin\left(\frac{1}{4}\pi\right) + k, k, \frac{1}{2} + k \right]$$

L'exercice suivant propose une amélioration de l'écriture de la procédure précédente.

Exercice : Commentez cette écriture qui ne construit pas une liste *res* de départ remplie de 0, mais utilise une liste initiale vide. Modifiez-la en utilisant une suite (sequence) initialisée par **NULL**.

```
> F:=proc(L,n)
  res:=[];
  for i from 1 to nops(L) do
    if evalf(L[i]) < 0 then
      res:=[op(res),L[i]-n];
    else
      res:=[op(res),L[i]+n];
    end if;
  end do;
end proc:
```

```
Warning, `res` is implicitly declared local to procedure `F`
```

```
Warning, `i` is implicitly declared local to procedure `F`
```

```
> M;
F(M,k);
```

$$[-3, 0, 1, -5]$$

$$[-3 - k, k, 1 + k, -5 - k]$$

Exercice: pourquoi, lorsque la liste donnée en entrée est vide, la procédure ne répond-elle rien ?

Réponse: parce que la dernière instruction effectuée n'est pas l'évaluation de la liste vide [], mais la comparaison de 1 avec nops(L) qui vaut 0 et termine la boucle.

Exercice: que faut-il faire pour que, dans ce cas, la procédure renvoie comme résultat une liste vide ? La solution est donnée plus loin.

```
> F([],k);
```

Un argument actuel assigné ne peut pas être modifié dans une procédure.

```
> d:=proc(t)
  t:=t+1
end proc:
```



```
> x:=1:
d(x);
```

```
Error, (in d) illegal use of a formal parameter
```

Un argument actuel non assigné peut l'être dans une procédure.

Mais attention: l'assignation n'est pas évaluée pendant l'exécution de la procédure car *x n'est pas* une variable interne. *L'assignation est "retardée" et se produit après l'exécution.* Voir cependant (plus loin) le

comportement particulier des variables déclarées **global** ou les variables d'environnement telles que **Digits** ou **Order**.

```
> d:=proc(x,y)
  x:=3;
  x+y; # x ne sera pas évalué
end proc;

> x:='x';
d(x,y); # d'où le résultat
x;      # x prend la valeur assignée après l'exécution
        x := x
        x + y
        3
```

et maintenant...

```
> d(x,y);
Error, (in d) illegal use of a formal parameter
```

De même

```
> x:='x':z:='z':t:='t':
d(z,t);
x;
z;
        z + t
        x
        3
```

Ici l'assignation $x := x + 1$ conduit à une définition récursive de z qui ne peut pas aboutir. Notez que l'erreur apparaît après l'exécution de la procédure.

```
> x:='x':z:='z':t:='t':
d:=proc(x,y)
  x:=x+1;
  x+y;
end proc;

> d(z,t);
x;
z;
        z + t
        x
```

```
Error, too many levels of recursion
```

Pour éviter d'avoir à assigner un argument dans le corps de la procédure on créera une variable interne (locale).

```
> x:='x':z:='z':t:='t':
d:=proc(x,y)
  xi:=x;
  xi:=xi+1;
```

```

xi+y;
end proc:
Warning, `xi` is implicitly declared local to procedure `d`
> d(z,t);
z,t;
z + t
z, t

```

Variables locales et globales

Lorsqu'elles sont assignées explicitement ou implicitement (par exemple par une boucle), les variables internes à une procédure, telles que *res* ou *i*, (voir *F*) sont, par défaut, des variables que MAPLE déclare locales tout en envoyant le message "Warning..." (avertissement). Ceci signifie que ces variables ne communiquent pas avec l'environnement extérieur à la procédure. Les variables *i* et *res* dont on demande l'évaluation dans la commande suivante sont différentes de celles utilisées dans la procédure, même si elles portent les mêmes noms.

```

> i, res;
i, res

```

Déclaration local

De même toute assignation à des variables *i* ou à *res* en dehors de la procédure serait sans effet sur le fonctionnement de celle-ci. Pour supprimer le message "Warning..." il faut déclarer "locales" les variables internes avec l'instruction **local** placée en tête de la procédure. L'exercice qui suit montre un telle déclaration qui fait disparaître le message d'avertissement.

Exercice: On donne la solution à l'une des questions de l'exercice précédent, c'est-à-dire: comment écrire la procédure pour qu'elle renvoie une liste vide lorsque la liste argument est vide ? Expliquez pourquoi on obtient le résultat souhaité ? La réponse n'a aucun rapport avec la déclaration **local**.

```

> F:=proc(L,n)
local i,res;
res:=NULL;
for i from 1 to nops(L) do
  if evalf(L[i]) < 0 then
    res:=res,L[i]-n;
  else
    res:=res,L[i]+n;
  end if;
end do;
[res];
end:

> M;
F(M,k);
F([]);
[-3, 0, 1, -5]
[-3 - k, k, 1 + k, -5 - k]
[]

```

Au risque de se répéter, on rappelle que l'on peut faire plus simple et que l'écriture précédente n'est destinée

qu'à illustrer le fonctionnement des procédures...!

```
> Fs:=(L,n)->[seq(x+sign(x)*n,x in L)];  
Fs(M,k);  
Fs([],k);
```

$$Fs := (L, n) \rightarrow [seq(x + sign(x) n, x \in L)]$$

$$[-3 - k, k, 1 + k, -5 - k]$$

[]

ou encore

```
> Fs:=(L,n)->map(x->x+sign(x)*n,L);  
Fs(M,k);  
Fs([],k);
```

$$Fs := (L, n) \rightarrow map(x \rightarrow x + sign(x) n, L)$$

$$[-3 - k, k, 1 + k, -5 - k]$$

[]

Attention, Attention

Une variable déclarée **locale** et **exportée** n'a pas le même statut qu'une variable dite de "premier niveau". Ici les x sont tous identiques et de premier niveau (niveau d'entrée des commandes). Conformément à la théorie des ensembles, MAPLE simplifie pour donner

```
> {x,x,x};  
{x}
```

Construisons une procédure qui donne comme résultat le nom symbolique x **déclaré local**

```
> x_2:=proc()  
  local x;  
  x;  
end proc:
```

Contrairement aux apparences et malgré l'homonymie, la variable générée par cette procédure n'est pas identique à la précédente car issue d'un calcul de niveau 2 (locale à une procédure)

```
> x_2();  
x
```

Il en résulte des comportements qui peuvent paraître troublants et trompeurs...

```
> {x,x,x_2()};  
{x, x}
```

Par contre, si la variable n'est pas déclarée locale, l'exportation donne une variable de premier niveau

```
> x_1:=proc()  
  x  
end proc:  
  
> {x,x,x_1()};  
{x}
```

Et ainsi de suite... On génère ici un x de niveau encore supérieur sans même avoir fait de déclaration

```

> x_3:=proc()
  x_2();
end proc;

> x_3();
x

> {x, x_2(), x_3()};
{x, x, x}

```

Exercice: expliquer ces résultats a priori surprenants...

```

> x=x_2();
evalb(%);
x = x
false

> Int(x_2()^2,x=0..2):%=value(%);

$$\int_0^2 x^2 dx = 2x^2$$


```

Conclusion: n'exportez jamais, sinon en toute connaissance de cause, des variables déclarées locales.

Transfert de données dans une procédure sans utiliser les arguments.

On peut utiliser dans une procédure des variables définies hors de celle-ci sans les transmettre par arguments si elles ne reçoivent pas d'assignation dans la procédure. Il est cependant très vivement déconseillé d'utiliser une telle méthode de programmation, sauf cas particuliers : penser aux constantes comme **Pi** ou aux variables globales d'environnement telles que **Digits** ou **Order** (voir plus loin la déclaration explicite **global**).

Les **variables globales d'environnement** (telles que **Digits** ou **Order**) peuvent recevoir, contrairement aux variables ordinaires assignées, une assignation dans le corps d'une procédure. Celle-ci est immédiatement effective (non retardée), mais les modifications seront annulées dès la fin de l'exécution de la procédure pour conserver la valeur initiale.

Il n'y a pas ici de message "Warning ...implicitly declared local..." car *A*, bien que utilisée, n'est pas assignée dans la procédure. N'étant pas déclarée locale, la procédure va utiliser la variable *A* externe. **Cette méthode de transfert d'une information dans le corps d'une procédure est, dans la mesure du possible, à proscrire absolument.** Par contre **Digits** est une variable globale d'environnement et n'a pas à être déclarée locale.

```

> Digits:
A:=20;

> P:=proc()
Digits:=4;
evalf(Pi)+A;
end proc;
P := proc() Digits := 4; evalf(Pi) + A; end proc;

> 'A'=A;
P();      # On a bien Pi+20 évalué en décimal avec 4 chiffres
'A'=A;
Digits; # Digits reprend automatiquement sa valeur initiale

```

A = 20

23.14

A = 20

10

Maintenant *A* est déclarée locale et on n'observe plus d'interférences avec l'environnement externe de la procédure.

```
> P:=proc()
  local A;
  Digits:=4;
  evalf(Pi)+A;
end proc;
```

P := proc() local A; Digits := 4; evalf(Pi) + A; end proc;

```
> 'A'=A;
P();
'A'=A;
Digits; # Digits reprend sa valeur initiale
A = 20
3.142 + A
A = 20
10
```

Déclaration global

On peut choisir de définir ***explicitement*** des variables avec l'option **global**. Ces variables auront un comportement similaire aux variables globales d'environnement (telles **Digits** ou **Order**) à la différence que **leurs modifications dans une procédure seront conservées après exécution**. Ceci permet de transmettre dans un environnement de travail des informations d'une procédure à une autre sans passer par les arguments (cette méthode délibérée de programmation n'est pas en contradiction avec les recommandations faites plus haut!). On remarquera que l'assignation de 3 à *X* n'est pas "retardée" car la variable est déclarée globale.

```
> P:=proc()
  global X;
  X:=3;
  X:=X+1;
end proc;

> P();
X;
```

4

4

Variables globales d'environnement

Comme on vient de le voir ces variables, telles **Digits** ou **Order**, conservent, **contrairement aux variables globales ordinaires**, leurs valeurs initiales après l'exécution d'une procédure dans laquelle elles ont été modifiées. Pour créer des variables globales d'environnement, il suffit de leur donner un nom commençant par **_Env**

```
> _Env_x:=3;
```

```

Modifie_x:=proc()
  _Env_x:=_Env_x+1;
end proc:

```

_Env_x := 3

```

> Modifie_x();
  _Env_x;

```

4
3

On trouvera des informations complémentaires dans l'aide en ligne

```
> ?envvar
```

Arguments optionnels , variables locales args et nargs

Supposons que nous voulions, pour la procédure *F* définie plus haut, que l'argument *n* soit optionnel et prenne par défaut la valeur 1.

```

> eval(F); # Pour rappel de la définition de F

```

```

proc(L, n)
local i, res;
res := [];
for i to nops(L) do if evalf(L[i]) < 0 then
    res := [op(res), L[i] - n]
else
    res := [op(res), L[i] + n]
end if;
end do;
res;
end proc;

```

L'écriture suivante ne peut pas être utilisée car MAPLE contrôle le nombre d'arguments actuels présents lors de l'appel de la procédure.

```

> F([1,-2,3]);
Error, (in F) F uses a 2nd argument, n, which is missing

```

Lors de l'appel d'une procédure les arguments actuels sont automatiquement placés dans une suite nommée **args** : le premier argument sera **args[1]**, le deuxième **args[2]**, etc. Une variable automatique **nargs** donne le nombre d'arguments actuels transmis. Les variables **args** et **nargs** sont locales par défaut. Ces variables permettent une *autre forme d'écriture qui ne spécifie aucun argument formel*. La procédure peut alors s'écrire (en n'oubliant pas de rajouter *n* aux variables locales)

```

> F:=proc()
local res,i,n:
if nargs = 1 then
  n:=1 # valeur choisie pour le défaut
else
  n:=args[2]
end if;
res:=[];
for i from 1 to nops(args[1]) do
  if evalf(args[1][i]) < 0 then
    res:=[op(res),args[1][i]-n]
  else

```

```

    res:=[op(res),args[1][i]+n]
  end if;
end do;
res;
end:

```

Les appels suivants sont alors possibles

```

> F([-3,Pi],k),
F([-3,Pi]),
F([]),
F([],2);

```

[-3 - $k, \pi + k]$, [-4, $\pi + 1]$, [], []

Appels des fonctions de MAPLE dans une procédure

Afin d'éviter toute interférence, *l'utilisation de l'opérateur with est interdit dans le corps d'une procédure* (ou d'un module; voir plus loin dans ce chapitre). Pour appeler dans une procédure (ou un module) les fonctions de MAPLE contenues dans des bibliothèques on utilisera la forme longue:

```

> ...
s:=CurveFitting[Spline](xy,t):
...

```

Si la fonction est issue d'une bibliothèque/module on pourra aussi utiliser la syntaxe :-

```

> ...
x:=LinearAlgebra:-LinearSolve(A,b):
...

```

Gestion des erreurs

Pour illustrer la suite nous allons d'abord créer une procédure que nous appellerons *Zip* et dont le fonctionnement est semblable à la fonction **zip** de MAPLE. Rappelons par un exemple son fonctionnement (voir la fin du chapitre 7). La liste résultat possède automatiquement un nombre d'éléments identiques à celui de la liste la plus petite.

```

> zip((x,y)->x/y,[1,3,0,-2,3,5],[2,-1,2,-3]);

```

$$\left[\frac{1}{2}, -3, 0, \frac{2}{3} \right]$$

```

> Zip:=proc()
local x,i;
# args[1] désigne l'opérateur
# args[2] et args[3] les deux listes
x:=[];
for i to min(nops(args[2]),nops(args[3])) do
  x:=[op(x),args[1](args[2][i],args[3][i])];
end do;
x;
end proc:

```

Attention: **Zip** est le nom d'une procédure de la librairie **LinearAlgebra** qui provoquerait une difficulté si on

la rendait accessible (ce nom a été choisi pour pouvoir faire cette remarque..!). Même protégée par la fonction **protect**, notre procédure serait remplacée par celle de **LinearAlgebra** et on recevrait un message d'avertissement. Inversement si on cherchait à créer la procédure après avoir donné accès avec **with** à **LinearAlgebra**, la création serait refusée car *Zip* serait devenu un nom protégé. Néanmoins, même après la création de notre procédure *Zip*, on peut toujours accéder à celle de **LinearAlgebra** en utilisant l'une des deux formes "longues" d'appel.

```
> LinearAlgebra[Zip]((x,y)->x*y,<1,2>,<a,b>):
LinearAlgebra:-Zip((x,y)->x*y,<1,2>,<a,b>); # 2 écritures équivalentes

$$\begin{bmatrix} a \\ 2b \end{bmatrix}$$

```

Notre procédure semble fonctionner correctement

```
> Zip((x,y)->x/y,[1,3,0,-2,3,5],[2,-1,2,-3]);
Zip((x,y)->x/y,[1,3,0,-2,3,5],[]);

$$\left[ \frac{1}{2}, -3, 0, \frac{2}{3} \right]$$

[]
```

mais... ici l'opérateur est absent

```
> zip([1,3,0,-2,3,5],[2,1,2,-3]);
Error, (in Zip) invalid subscript selector
```

... ici l'opérateur n'est qu'à une variable

```
> Zip(x->exp(x),[1,3,0,-2,3,5],[2,1,2,-3]);
[e, e^3, 1, e^{(-2)}]
```

... ici les arguments ne sont pas dans le bon ordre

```
> Zip([1,3,0,-2,3,5],[2,1,2,-3],(x,y)->x/y);
[[1, 3, 0, -2, 3, 5], [1, 3, 0, -2, 3, 5], [1, 3, 0, -2, 3, 5], [1, 3, 0, -2, 3, 5]]
```

Ayant la définition de la procédure sous les yeux, il ne sera pas difficile d'apporter les corrections. Cependant les procédures (comme, par exemple, celles de MAPLE) peuvent être rangées dans une librairie (voir la fin de ce chapitre) et l'utilisateur n'y aura pas forcément un accès facile. Même quand on est l'utilisateur de ses propres procédures, on oublie vite... Sans compter les résultats faux et trompeurs, les messages renvoyés n'éclairent pas beaucoup sur la raison de ce qui ne va pas et on aura donc intérêt à écrire des procédures "robustes" qui sachent gérer un maximum d'erreurs avec des messages clairs.

1) La fonction **error**

La fonction **error** arrête le déroulement d'une procédure et permet au programmeur de générer ses messages d'erreurs ou de renvoyer un résultat. Sa syntaxe est la suivante

```
error "Message %i blabla...%k blabla...",a,b,c,...
```

- La chaîne de caractères (optionnelle) "..." est celle du message.
- Les éléments de la suite (optionnelle) *a, b, c,...* peuvent être des symboles, des chaînes de caractères, des expressions, des valeurs numériques, etc...
- Le symbole **%i** sera remplacé dans la chaîne du message par le *i*-ème élément de la suite précédente. Par exemple **%2** sera remplacé par l'évaluation de *b*.

- On peut également utiliser le symbole `%-i` qui remplace le i -ème élément de la suite, qui doit être dans ce cas un entier positif, par l'ordinal anglais correspondant. Par exemple si la suite est $x, y, 2, a, 3$, le symbole `%-3 (=2)` sera remplacé par 2nd et `%-5 (=3)` par 3rd, etc.

- Le mot clé `procname` contient le nom de la procédure en cours et peut être introduit dans la liste. Il permet de changer le nom de la procédure sans avoir à modifier les messages d'erreurs.

- Le caractère "`\`" indique une continuation de la chaîne de caractères. Un *Return* seul entraînerait un message sur deux lignes (voir le premier test sur le nombre d'arguments).

On va illustrer plusieurs cas d'utilisation de cette fonction en vérifiant dans la procédure `Zip` la validité des arguments reçus (voir aussi dans ce chapitre le paragraphe *Contrôle simplifié des arguments*).

```
> Zip:=proc()
local x,i,k1,k2;

# args[1] désignera l'opérateur
# args[2] et args[3] les deux listes

# On teste le nombre d'arguments transmis.
# procname contient le nom Zip.
if nargs <> 3 then
    error "Le nombre d'arguments de %1
          doit être 3, mais en a recu %2",procname,nargs;
end if;

# On teste le premier argument. Il n'y a
# aucune suite de variables associée au message.
if not type(args[1],operator(arrow)) then
    error "Le 1er argument doit être un opérateur \
du type arrow"
end if;

# On teste le nombre d'arguments de l'opérateur.
# Voir le § "opérandes" du chapitre "Fonctions".
if nops([op(1,args[1])]) <> 2 then
    error "L'opérateur doit avoir 2 arguments, \
mais en possède %1",nops([op(1,args[1])]);
end if;

# On vérifie que les deux autres arguments sont
# des listes. La programmation est un peu "tirée
# par les cheveux" mais illustre l'utilisation
# des symboles %-i.
if not type(args[2],list(algebraic)) then
    k1,k2:=2,3:
elif not type(args[3],list(algebraic)) then
    k1,k2:=3,2:
end if;
if assigned(k1) then
    error "The %-1 argument (and the %-2) must \
be of type list of algebraic terms",k1,k2:
end if;
```

```

x:=[];
for i to min(nops(args[2]),nops(args[3])) do
  x:=[op(x),args[1](args[2][i],args[3][i])];
end do;
x;
end proc:
```

On va tester quelques messages en provoquant les erreurs

```

> Zip([1,3,0,-2,3,5],[2,1,2,-3]);
Error, (in Zip) Le nombre d'arguments de Zip
doit être 3, mais en a recu 2

> Zip(binomial,[1,3,0,-2,3,5],[2,1,2,-3]);
Error, (in Zip) Le 1er argument doit être un opérateur du type arrow

> Zip((x,y,z)->x/(y+z),[1,3,0,-2,3,5],[2,1,2,-3]);
Error, (in Zip) L'opérateur doit avoir 2 arguments, mais en possède 3

> Zip((x,y)->x/y,[1,3,0,-2,3,5],3);
Error, (in Zip) The 3rd argument (and the 2nd) must be of type list of algebraic terms
```

3rd et 2nd sont échangés

```

> Zip((x,y)->x/y,3,[1,3,0,-2,3,5]);
Error, (in Zip) The 2nd argument (and the 3rd) must be of type list of algebraic terms
```

Pour le reste tout semble fonctionner normalement

```

> Zip((x,y)->x/y,[x,3,0,exp(x),x/y,4],[2,z,2,-3,-2]);

$$\left[ \frac{1}{2}x, \frac{3}{z}, 0, -\frac{1}{3}e^x, -\frac{x}{2y} \right]$$

```

2) Le bloc try/catch

Le message d'erreur renvoyé ici par MAPLE est très clair

```

> a:=1:      b:=0:
  a/b;
Error, numeric exception: division by zero
```

Mais supposons que nous voulions, par exemple, franciser et préciser le message (utile évidemment dans un autre contexte !). On pourrait construire un bloc **if** testant la nullité du dénominateur mais on peut aussi utiliser un bloc **try**. On voit que l'on "essaie" (to try=essayer) d'abord la division puis, si le message d'erreur survient, il est "attrapé" (to catch=attraper) et le message d'erreur souhaité est renvoyé

```

> a:=2:      b:=0:
  try
    a/b
  catch "numeric exception: division by zero":
    error "Division par 0 (%1/0)",a
  end try;
Error, Division par 0 (2/0)
```

Il est possible de "capturer" plusieurs types d'erreurs avec plusieurs ordres **catch** dans le même bloc

```
> a:=2:    b:=1..2:  
try  
  a/b  
catch "numeric exception: division by zero":  
  error "Division par 0 (%1/0)",a  
catch "invalid terms in product":  
  error "Division non conforme: (%1)/(%2)",a,b  
end try;  
  
Error, Division non conforme: (2)/(1 .. 2)
```

On peut aussi capturer toutes les erreurs sans préciser le message à capturer

```
> a:=2:    b:=1..2:  
try  
  a/b  
catch:  
  error "Division impossible"  
end try;  
  
Error, Division impossible
```

Attention: le message à capturer doit être écrit avec précision pour que le mécanisme fonctionne. Pour le connaître, il suffit de provoquer l'erreur et de récupérer la chaîne de caractères avec la variable **lasterror**

```
> a:=1:    b:=0:  
a/b:  
lasterror;  
  
Error, numeric exception: division by zero  
"numeric exception: division by zero"
```

Attention: respecter la syntaxe pour reproduire l'exemple ci-dessous. Si on écrit *a/a..b*, l'analyseur syntaxique comprendra *1..b*. Il n'y aura pas d'erreur et **lasterror** contiendra le message de la dernière erreur rencontrée, c'est-à-dire "numeric exception: division by zero".

```
> a/(a..b);  
lasterror;  
  
Error, invalid terms in product  
"invalid terms in product"
```

Pour simplifier la suite on va construire une procédure *Zip_div*, similaire à *Zip* mais qui ne réalise que la division terme à terme des deux listes. La présence de l'opérateur en argument n'est donc plus nécessaire

```
> Zip_div:=proc()  
local x,i;  
  
# On a supprimé tous les tests  
# pour raccourcir l'écriture.  
  
x:=[];  
for i to min(nops(args[1]),nops(args[2])) do  
  x:=[op(x),args[1][i]/args[2][i]];  
end do;  
x;
```

```
end proc:
```

Ca fonctionne...

```
> zip_div([1,-2,3,1],[3,5,-1]);
```

$$\left[\frac{1}{3}, \frac{-2}{5}, -3 \right]$$

mais évidemment, en raison de l'opération $-2/0$

```
> zip_div([1,-2,3,1],[3,0,-1]);
```

```
Error, (in Zip_div) numeric exception: division by zero
```

On souhaite modifier ce comportement pour que la procédure ne s'arrête pas sur cette erreur mais remplace la division $x/0$ par $\text{sign}(x) \infty$. On va pouvoir introduire un bloc **try** qui évitera l'utilisation d'un bloc **if**.

```
> Zip_div:=proc()
local x,i;
x:=[];
for i to min(nops(args[1]),nops(args[2])) do
  try
    x:=[op(x),args[1][i]/args[2][i]];
  catch "numeric exception: division by zero":
    x:=[op(x),sign(args[1][i])*infinity];
  catch:
    error "Division impossible";
  end try
end do;
x;
end proc:
```

```
> zip_div([1,-2,3,1],[3,0,-1]);
```

$$\left[\frac{1}{3}, -\infty, -3 \right]$$

```
> zip_div([1,-2,3,1],[3,0..1,-1]);
```

```
Error, (in Zip_div) Division impossible
```

Attention: introduire le bloc **try** dans la boucle **do** est différent d'introduire la boucle **do** dans le bloc **try**.

L'écriture suivante est correcte mais arrête la procédure sur l'occurrence de l'erreur

```
> Zip_div_2:=proc()
local x,i;
x:=[];
try
  for i to min(nops(args[1]),nops(args[2])) do
    x:=[op(x),args[1][i]/args[2][i]];
  end do;
catch "numeric exception: division by zero":
  x:=[op(x),sign(args[1][i])*infinity];
end try;
x;
end proc:
```

```
> zip_div_2([1,-2,3,1],[3,0,-1]);

$$\left[ \frac{1}{3}, -\infty \right]$$

```

Retour anticipé, instruction return

Il est toujours possible d'arrêter la procédure pendant son déroulement et de renvoyer un résultat avec l'instruction **return** dont la syntaxe est la suivante

```
return résultat
```

Cette instruction est en général associée à une condition définie par un bloc **if** ou un bloc **try**. Il peut bien sûr y avoir plusieurs instructions **return** dans une procédure.

Supposons que nous voulions que la procédure s'arrête dès qu'une division par 0 survient et qu'elle renvoie alors sa propre expression non évaluée. On pourra écrire

```
> Zip_div_3:=proc()
local x,i;
x:=[];
for i to min(nops(args[1]),nops(args[2])) do
try
  x:=[op(x),args[1][i]/args[2][i]];
catch "numeric exception: division by zero":
  return 'procname'(args);
end try
end do;
x;
end proc:
```

```
> Zip_div_3([1,-2,3,1],[3,3,-1]);
Zip_div_3([1,-2,3,1],[3,0,-1]);

$$\left[ \frac{1}{3}, \frac{-2}{3}, -3 \right]$$

Zip_div_3([1, -2, 3, 1], [3, 0, -1])
```

C'est ce que fait une procédure de MAPLE comme **sin** quand elle ne peut plus opérer de transformations internes ou récursives

```
> sin(Pi/4);
sin(x*y);

$$\frac{1}{2} \sqrt{2}$$

sin(x y)
```

Attention: on a écrit '**procname**' (**args**) pour que **procname** ne soit pas évaluée (plus exactement, l'évaluation est retardée et n'apparaît qu'après l'exécution de la procédure. Si on supprime les ', le mot **procname** sera évalué et provoquera (voir ci dessous) un nouvel appel de *Zip_div_rec(args)*. Cet appel en "miroir", dit récursif, peut être très utile en programmation et MAPLE s'en sert très souvent dans la définition de ses procédures. Ici rien ne permet d'arrêter le mécanisme et une erreur apparaît

```
> Zip_div_rec:=proc()
```

```

local x,i;
x:=[];
for i to min(nops(args[1]),nops(args[2])) do
  try
    x:=[op(x),args[1][i]/args[2][i]];
  catch "numeric exception: division by zero":
    return procname(args);
  end try
end do;
x;
end proc:

```

Le nombre d'appels récursifs étant limité, MAPLE affiche un message d'erreur.

```

> Zip_div_rec([1,-2,3,1],[3,0,-1]);
Error, (in Zip_div_rec) too many levels of recursion

```

Supposons maintenant que l'on veuille, pour une raison quelconque, que le résultat soit une liste de 0 de taille identique à la première liste quand une division par 0 apparaît. Il suffit d'écrire

```

> Zip_div_0:=proc()
local x,i;
x:=[];
for i to min(nops(args[1]),nops(args[2])) do
  try
    x:=[op(x),args[1][i]/args[2][i]];
  catch "numeric exception: division by zero":
    return [seq(0,i=args[1])];
  end try
end do;
x;
end proc:

> Zip_div_0([1,-2,3,1],[3,0,-1]);
[0, 0, 0, 0]

```

Exercice: on veut la même chose mais avec une liste de 0 dont le nombre est égal au nombre d'éléments de la plus grande des deux listes. Modifier la procédure.

Exercice: voici une petite procédure de "division modifiée". Etudiez son écriture et essayez la.

```

> Div:=proc(x,y,k)
# On pourrait faire ici des tests sur la nature des arguments
try
  x/y
catch "numeric exception: division by zero":
  if x = 0 then
    return k;
  else
    return sign(x)*infinity;
  end if;
end try
end proc:

```

Options cache, remember et trace

1) L'option cache

Avant la lecture de ce paragraphe on se reportera au chapitre 7, *Fonctions*, § *Opérandes*. Par défaut les procédures créées par un utilisateur n'activent pas de tables **Cache** ou **remember**. Elles valent **NULL** par défaut et c'est la raison pour laquelle MAPLE ne répond rien à cette commande.

```
> op(4,eval(Div));
```

Il existe 12 options possibles et **cache** et **remember** ne sont que deux d'entre elles.

```
> ?option
```

On peut activer ce mécanisme avec l'option **cache** (ou **remember**; voir plus loin) que l'on introduira en tête de la procédure.

```
> Div:=proc(x,y,k)
  option cache;
  try
    x/y
  catch "numeric exception: division by zero":
    if x = 0 then
      return k;
    else
      return sign(x)*infinity;
    end if;
  end try
end proc:
```

La table cache vaut toujours **NULL**

```
> op(4,eval(Div));
```

mais après chaque calcul le résultat sera enregistré dans la partie "temporaire" de la table

```
> Div(1,3,1): Div(-3,0,1): Div(-4,2,1):
  op(4,eval(Div));
```

$$\text{Cache} \left(512, \text{'temporary'} = \left[(1, 3, 1) = \frac{1}{3}, (-4, 2, 1) = -2, (-3, 0, 1) = -\infty \right] \right)$$

On peut assigner une image particulière pour des arguments particuliers

```
> Div(0,0,1):=0;
Div(0, 0, 1) := 0
```

qui sera enregistrée dans la partie temporaire de la table cache

```
> op(4,eval(Div));
```

$$\text{Cache} \left(512, \text{'temporary'} = \left[(1, 3, 1) = \frac{1}{3}, (-4, 2, 1) = -2, (0, 0, 1) = 0, (-3, 0, 1) = -\infty \right] \right)$$

A chaque appel de *Div*, la procédure va explorer la table en priorité pour savoir si le calcul a déjà été effectué et donner le résultat sans faire de calculs supplémentaires. On comprend que cette option puisse faire gagner du temps de calcul. La table est chargée jusqu'à concurrence de sa taille maximum, ici 512 entrées. Si le nombre d'entrées est dépassé les éléments "temporaires" sont éliminés au fur et à mesure.

Notez qu'il existe une **bibliothèque Cache** contenant 10 fonctions de gestion des tables de type *cache*. La commande suivante ajoute un élément dans la partie "permanente" de la table

```
> Cache:-AddPermanent(Div,[0,0,3],0);
op(4,eval(Div));
```

$$\text{Cache} \left(512, \text{'temporary'} = \left[(1, 3, 1) = \frac{1}{3}, (-4, 2, 1) = -2, (0, 0, 1) = 0, (-3, 0, 1) = -\infty \right], \right.$$

$$\left. \text{'permanent'} = [(0, 0, 3) = 0] \right)$$

Ici on modifie le nombre maximal d'entrées permises qui est fixé à la puissance de 2 immédiatement supérieure à la valeur prescrite

```
> Cache:-Resize(120,Div);
```

$$\text{Cache} \left(128, \text{'temporary'} = \left[(-4, 2, 1) = -2, (0, 0, 1) = 0, (1, 3, 1) = \frac{1}{3}, (-3, 0, 1) = -\infty \right], \right.$$

$$\left. \text{'permanent'} = [(0, 0, 3) = 0] \right)$$

Pour vider la table cache d'une procédure on utilisera la fonction **forget** (l'option **cache** reste néanmoins active)

```
> forget(Div);
op(4,eval(Div));
```

2) L'option remember

Elle fournit un comportement similaire, mais avec plusieurs différences

- La table est une table ordinaire.

- Sa taille n'est pas limitée. Ceci présente l'inconvénient de pouvoir consommer beaucoup d'espace mémoire. Si les calculs de la procédure sont simples, elle pourra passer plus de temps à explorer la table qu'à faire des calculs !

- Il n'y a pas, à l'exception de **forget**, d'outil de gestion de la table.

On préférera, sauf cas particulier, l'option **cache**.

3) L'option trace

Lors de la mise au point d'une procédure on peut demander le contrôle du déroulement de celle-ci. Il suffira de donner l'option **trace** (avec ou sans l'option **cache**, ces deux options étant indépendantes). La procédure affichera les diverses opérations effectuées.

```
> Zip_div:=proc()
  option cache, trace;
  local x,i;
  x:=[];
  for i to min(nops(args[1]),nops(args[2])) do
    try
      x:=[op(x),args[1][i]/args[2][i]];
    catch "numeric exception: division by zero":
      x:=[op(x),sign(args[1][i])*infinity];
    catch:
```

```

    error "Division impossible";
end try
end do;
x;
end proc:

> zip_div([1,3,-1],[-2,5,0]);
{--> enter Zip_div, args = [1, 3, -1], [-2, 5, 0]

          x := []
          x :=  $\left[ \frac{-1}{2} \right]$ 
          x :=  $\left[ \frac{-1}{2}, \frac{3}{5} \right]$ 
          x :=  $\left[ \frac{-1}{2}, \frac{3}{5}, -\infty \right]$ 
           $\left[ \frac{-1}{2}, \frac{3}{5}, -\infty \right]$ 

<-- exit Zip_div (now at top level) = [(-1)/2, 3/5, -infinity]}
```

$$\left[\frac{-1}{2}, \frac{3}{5}, -\infty \right]$$

Exercice : pourquoi les résultats intermédiaires de la commande suivante ne sont-ils pas affichés ?

```

> zip_div([1,3,-1],[-2,5,0]);
value remembered (at top level): Zip_div([1, 3, -1], [-2, 5, 0]) -> [(-1\
)/2, 3/5, -infinity]

           $\left[ \frac{-1}{2}, \frac{3}{5}, -\infty \right]$ 
```

Contrôle simplifié des arguments

MAPLE autorise une syntaxe simplifiée du contrôle des arguments réels en faisant suivre les arguments formels de " :: " puis, soit du type souhaité, soit d'un ensemble ou d'un profil de types (voir la fin du chapitre 2, *Assignations, Evaluations,...*). La non conformité d'un des arguments actuels transmis lors de l'appel entraîne l'abandon de la procédure et l'affichage d'un message

```

> restart:
Test_arg:=proc(x::algebraic,L::list)
  `Arguments valides`:
end proc:

> Test_arg(xy,[a,b]);
Arguments valides
```

```
> Test_arg(-2+exp(x^2),[a,{b,c}]);
                                         Arguments valides

> Test_arg(xy,{a,b});
Error, invalid input: Test_arg expects its 2nd argument, L, to be of type list, but
received {a, b}
```

L'utilisation d'ensembles sur les types permet des alternatives de choix. Le premier argument peut-être soit une constante réelle soit un nom, etc...

```
> Test_arg:=proc(x::{realcons,name},
                  y::posint,
                  z::{realcons..realcons,3,list(complex)})
  `Arguments valides`:
end proc:
```

```
> Test_arg(xy,3,2..Pi);
                                         Arguments valides
```

```
> Test_arg(Pi,-3,2..Pi);
Error, invalid input: Test_arg expects its 2nd argument, y, to be of type posint, but
received -3
```

```
> Test_arg(1,2,3);
                                         Arguments valides
```

```
> Test_arg(1,3,2);
Error, invalid input: Test_arg expects its 3rd argument, z, to be of type {3, realcons
.. realcons, list(complex)}, but received 2
```

etc... On propose une modification du calcul de certains types d'intégrales dans lequel on ajoute une constante symbolique aux primitives. On notera la possibilité, avec le mot clé **description** de donner une chaîne de caractères de description.

```
> Integrale:=proc(e::algebraic,
                  r::{name,name=algebraic..algebraic})
  description "Intégrale et/ou primitive avec constante additive";
  if type(r,name) then
    Int(e,r);
    %value(%)+_Cte
  else
    Int(e,r);
    %value(%)
  end if;
end proc:
```

Les tests sur les arguments des appels suivants sont corrects

```
> Integrale(exp(-t*s^2),t=0..1+sin(x));

$$\int_0^{1 + \sin(x)} e^{(-t s^2)} dt = -\frac{1 + e^{(-(1 + \sin(x)) s^2)}}{s^2}$$


> Integrale(exp(-t*s^2),t);
```

$$\int e^{(-t s^2)} dt = -\frac{e^{(-t s^2)}}{s^2} + Cte$$

Mais

```
> Integrale(exp(-t*s^2), t=[0, 1+sin(x))];
```

Error, invalid input: Integrale expects its 2nd argument, r, to be of type {name = algebraic .. algebraic, name}, but received t = [0, 1+sin(x)]

Non évaluation forcée

On peut forcer la non évaluation d'un argument actuel *dans la procédure* avec le pseudo type *uneval*

```
> x:=3;
t:=proc(z::uneval)
z+1;
end proc:
t(x+1); # x+2 au lieu de 5
x := 3
x + 2
```

On peut également forcer l'évaluation à celle d'un nom avec le pseudo type *evaln* (evaluate as a name). *evaln* est aussi une fonction de MAPLE: evaln(x) produit le même effet que 'x'.

```
> n:=proc(z::evaln)
z+1;
end proc:
n(x); # x+1 au lieu de 4
x + 1
```

Mais, comme $x + 1$ ne peut être évalué comme un nom

```
> n(x+1);
Error, illegal use of an object as a name
```

de même

```
> n(2);
Error, illegal use of an object as a name
```

On peut préciser que l'argument actuel doit posséder un type donné

```
> N:=proc(z::evaln(integer))
z+1;
end proc:
N(x); # x est un entier qui vaut 3
x + 1
```

```
> x:=3.2:
N(x);
Error, invalid input: N expects its 1st argument, z, to be of type evaln(integer), but
received x := 3.2
```

Génération d'une procédure par une procédure

Une procédure peut générer une procédure comme résultat. La fonction **dsolve** avec l'option **numeric** en est un exemple (voir chapitre 15, *Equations différentielles, § Résolutions numériques*). Dans l'exemple suivant on montre une illustration avec la méthode de *Newton-Raphson* pour la résolution itérative d'équations. On rappelle que pour résoudre $f(x) = g(x)$ par cette méthode on transforme cette équation en la mettant sous la forme $F(x) = 0$. Puis, partant d'une valeur x_0 , on calcule itérativement x_1, x_2, \dots avec le schéma

$$x_{n+1} = x_n - \frac{F(x_n)}{\left[\frac{dF(x)}{dx} \right]} \quad |_{x=x_n}$$

jusqu'à convergence. La procédure *N_R* suivante génère une procédure anonyme spécifique qui effectue ce calcul en fonction de F . *Ici la "dernière instruction" de N_R est le texte du corps de la procédure de calcul que l'on génère.*

```
> N_R:=proc(F::operator(arrow))
  local DF:
  DF:=D(F):
  proc(x_0::numeric,epsilon::numeric)
    local x_i,x_p,n_iter,n_iter_max;
    n_iter_max:=10:
    n_iter:=0:
    x_i:=x_0;
    x_p:=infinity:
    while abs((x_i-x_p)/x_i) > epsilon do
      if n_iter > n_iter_max then
        error "Le nombre d'itérations est \
plus grand que %1", n_iter_max:
      end if:
      x_p:=x_i:
      x_i:=evalf(x_p-F(x_p)/DF(x_p)):
      n_iter:=n_iter+1:
    end do:
    x_i:
  end proc:
end proc:
```

Le résultat de la procédure *N_R* est une procédure spécifique pour la fonction $F : x \rightarrow \cos(x) - x$ qui résout $\cos(x) = x$. On l'assigne à un nom

```
> H:=N_R(x->cos(x)-x);
H := proc(x_0::numeric, epsilon::numeric)
local x_i, x_p, n_iter, n_iter_max;
n_iter_max := 10;
n_iter := 0;
x_i := x_0;
x_p := infinity;
while epsilon < abs((x_i - x_p)/(x_i)) do if n_iter_max < n_iter then
  error "Le nombre d'itérations est plus grand que %1", n_iter_max
```

```

end if;
 $x\_p := x\_i;$ 
 $x\_i := evalf(x\_p - ((\text{proc}(x) \text{ option operator, arrow; } \cos(x) - x \text{ end proc; })(x\_p))/(DF(x\_p)));$ 
 $n\_iter := n\_iter + 1;$ 
end do;
 $x\_i;$ 
end proc;

```

Comme le demande la procédure générée on doit fixer en argument la valeur initiale x_0 des itérations et la précision relative minimale attendue

```

> H(0.5,1.0e-10);
0.7390851332

```

Le résultat ne doit pas dépendre de cette première valeur pourvu qu'elle ne soit pas trop éloignée de la solution (plus précisément x_0 doit appartenir au bassin d'attraction de la racine).

```

> H(1.2,1.0e-10);
0.7390851332

```

Le résultat est correct comme le montre la solution donnée par **fsolve** (voir chapitre 14, *Equations ordinaires*)

```

> fsolve(cos(z)-z=0,z=1);
0.7390851332

```

Opérandes d'une procédure

Rappel:

```

> whattype(sin), whattype(eval(sin)), type(sin,procedure);
symbol, procedure, true

```

Une procédure possède 8 opérandes.

```

> nops(eval(sin));
8

```

L'opérande 1 est la suite des noms des arguments formels utilisés pour construire la procédure et leurs types (ou profils de type) associés. La suite peut être **NULL** (pas d'argument ou option **builtin** par exemple)

```

> op(1,eval(zip));
f, a::{list, Vector, array, Array, Matrix}, b::{list, Vector, array, Array, Matrix}, d

```

```

> op(1,eval(Integrale)); # Voir plus haut
e::algebraic, r::{name = algebraic .. algebraic, name}

```

L'opérande 2 est la suite des variables déclarées locales dans la procédure

```

> op(2,eval(sin));
res, n, t, pull_out, keep_in

```

La liste peut être **NULL** (ici pas de variables locales)

```

> op(2,eval(Integrale));

```

L'opérande 3 est la suite des options

```
> op(3,eval(sin));
```

Copyright (c) 1992 by the University of Waterloo. All rights reserved.

```
> f:=x->2*x+1;  
op(3,eval(f));
```

$f := x \rightarrow 2x + 1$

operator, arrow

Il existe 12 options possibles dont *cache*, *remember*, 'Copyright...', *operator* ou *arrow*

```
> ?option
```

La table Cache est associée à l'opérande 4 (voir plus haut et aussi le chapitre 7, *Fonctions*).

```
> op(4,eval(sin)); # voir plus haut
```

L'opérande 5 donne la chaîne de description (voir plus haut l'écriture de la procédure "Integrale"). Cette chaîne peut être **NULL**, comme ici pour **sin**

```
> op(5,eval(sin));
```

```
op(5,eval(Integrale));
```

"Intégrale et/ou primitive avec constante additive"

On trouvera les définitions des deux autres opérandes dans l'aide en ligne

```
> ?procedure
```

Modules

Un module est une structure contenant une collection de procédures, de fonctions ou de simples instructions, réunies le plus souvent autour d'un thème. Beaucoup des librairies de MAPLE et notamment les plus récentes, comme **LinearAlgebra**, sont organisées par modules (les plus anciennes, comme **orthopoly**, sont organisées en table). Plutôt qu'une description détaillée, on donne ci-dessous un exemple d'illustration (volontairement très simple et par conséquent d'un intérêt très limité).

On suppose donnée une liste de listes $[\dots, [x_i, y_i], \dots, [x_j, y_j], \dots]$ contenant des abscisses et ordonnées d'une fonction et pour lesquelles les abscisses ne sont ni rangées dans un ordre croissant ou décroissant, ni équidistantes. On veut calculer une approximation numérique de l'intégrale de la fonction définie par une telle liste. Le module contient deux modes d'intégration, un basé sur une méthode de trapèzes, l'autre sur la construction d'une spline d'interpolation. On notera les points suivants:

- Le module est assigné à un nom, **nom_md:=module()** et se termine par **end module**;
- Un module peut contenir des variables locales (internes) et/ou globales. L'exemple suivant contient une variable locale (**Lt**) et un procedure locale (**Tri**). On retrouvera donc les mêmes déclarations que pour les procédures: **local**, **global**, etc.
- Sur les trois procédures contenues dans le module, deux seulement sont autorisées à être exportées (**Trapezes** et **Spl**) et sont précisées par la commande **export**. La troisième procédure (**Tri**) reconstruit la liste de listes par abscisses croissantes. Elle est considérée par le programmeur comme une procédure d'usage interne qui n'a pas à être exportée.
- Chaque procédure peut avoir son jeu de variables locales (ou globales).
- **Rappel:** l'utilisation de **with** est interdite dans les procédures et les modules. Pour accéder aux fonctions des bibliothèques de MAPLE on utilisera la forme longue

CurveFitting[Spline](...) ou **CurveFitting:-Spline(...)**

la deuxième ne fonctionnant que sur des bibliothèques construites par modules.

```
> Int_Num := module()
  local Tri,Lt;
  export Trapezes, Spl;

Trapezes:= proc(L)
  local i,S:
  Lt:=Tri(L);
  S:=0:
  for i to nops(L)-1 do
    S:=S+(Lt[i+1][2]+Lt[i][2])*(Lt[i+1][1]-Lt[i][1])/2
  end do;
end proc:

Spl:= proc(L)
  local i,S,x:
  Lt:=Tri(L);
  int(CurveFitting:-Spline(Lt,x),x=Lt[1][1]..Lt[-1][1]);
end proc:

Tri:=proc(L::list([realcons,realcons]))
  sort(evalf(L),(x1,x2)->`<`(op(1,x1),op(1,x2)));
end proc;

end module:
```

Pour faire un test, on construit une liste aléatoire de 20 abscisses dans l'intervalle [0,3] (voir chapitre 7, *Fonctions*) en y ajoutant les abscisses 0 et 3,

```
> x:=[0,3,seq(RandomTools[Generate](
  float(range=0..3,digits=5,method=uniform)),
  i=1..20)]:
```

puis la liste de listes $[(x_1, y_1), (x_2, y_2), \dots]$ en prenant pour ordonnées la fonction $\exp(-x^2)/\sqrt{\pi}$.

```
> Lp:=map(t->[t,evalf(exp(-t^2)/sqrt(Pi))),x):
```

Le résultat de l'intégrale devra donc être voisin de

```
> evalf(Int(exp(-t^2)/sqrt(Pi),t=0..3));
  0.4999889548
```

Pour utiliser une procédure d'un module on utilisera l'opérateur " :- "

```
> R[T]:=Int_Num:-Trapezes(Lp);
  R_T := 0.5000998499
```

Mais on peut aussi utiliser la forme standard [...].

```
> R[S]:=Int_Num[Spl](Lp);
  R_S := 0.4999099113
```

Remarques:

- Pour une liste avec beaucoup de points, la construction d'une spline devient gourmande en temps de calcul et en mémoire. Aucun test n'a été prévu dans le module pour une limitation, ce qui est une erreur...

- Une procédure peut être écrite dans le module, quand c'est possible, sous la forme d'une fonction. Par exemple *Tri* pourrait être remplacée par

```
Tri:=L->sort(evalf(L),(x1,x2)-><`(op(1,x1),op(1,x2)));
```

- La procédure *Tri* de notre module n'est pas définie comme exportable. On notera toutefois que pour faire le tracé 2D de la fonction à partir de la liste de listes, l'exportation de *Tri* serait bien utile et qu'il suffirait de la rajouter dans suite de **export** et de la supprimer de **local**. Une procédure n'est pas forcément exportable (si elle utilise par exemple des variables internes du module).

```
> Int_Num:-Tri(Lp);  
Error, module does not export `Tri`
```

On peut aussi utiliser l'opérateur **with** pour rendre accessible un ou des éléments exportés d'un module

```
> with(Int_Num,Trapezes);  
[Trapezes]
```

Ce qui évite l'appel par la forme longue (par contre on ne peut pas utiliser **with** dans le corps des procédures)

```
> Trapezes(Lp);  
0.5000998499
```

Noms exportés particuliers

Il existe des noms exportés prédéfinis qui entraînent des comportements particuliers

1) Le nom **ModuleApply**

Quand on donne ce nom à une des procédures, fonction ou assignation exportées d'un module, le nom du module peut être utilisé comme un nom de fonction sans utiliser l'appel `:-`. On aurait pu aussi bien donner ce nom à *f*(ModuleApply:=[3,2,1]): ou à *g* (ModuleApply:=(x::algebraic,y::algebraic)->*sqrt(x)*exp(y)*):.

```
> restart:  
F:=module()  
  export ModuleApply,f,g;  
  ModuleApply:=proc(x::algebraic)  
    x*g(-x,1);  
  end proc;  
  f:=[3,2,1];  
  g:=(x::algebraic,y::algebraic)->sqrt(x)*exp(y);  
end module;  
F := module () export ModuleApply, f, g; end module
```

Les deux écritures sont alors équivalentes

```
> F(z);  
F:-ModuleApply(z);  
z √-z e  
z √-z e
```

Un module exportant **ModuleApply** peut exporter d'autres objets. Cette fonctionnalité est surtout utile dans le cadre d'une programmation orientée objet.

```
> F:-f;  
F:-g(z,t);
```

[3, 2, 1]

$$\sqrt{z} e^t$$

2) Les noms ModuleLoad et ModuleUnload

Ces noms sont utilisés avec des modules lus dans une librairie. Nous en donnerons un exemple simple dans le paragraphe relatif à la création et l'utilisation de librairies

Variables structurées, fonction Record.

MAPLE permet de créer des variables structurées, c'est à dire d'agréger des données de natures variées et des procédures de calcul attachées à celles-ci sous l'enregistrement unique d'une variable. Supposons par exemple que nous voulions définir un objet A du type "point du plan cartésien". Il sera totalement défini par ses coordonnées, disons x et y , données sous forme de constantes ou d'expressions. On utilisera la fonction **Record** avec la syntaxe suivante. **Le résultat est un module** (avec option *record*) qui exporte les noms proposés, ici x et y

```
> restart;
A:=Record(x=-1,y=2);
A := module () export x, y; option record; end module
```

Conformément à la syntaxe associée aux modules on récupèrera les coordonnées en écrivant (`:-` est l'équivalent de l'opérateur de champ de structure `.` du C et C++ ou du `%` du Fortran).

```
> A:-x, A:-y;
-1, 2
```

La fonction **Record** associée à A est une forme simplifiée de l'écriture d'un module avec option *record*. Elle est écrite ici pour un autre "point" X

```
> X:=module()
  option record;
  export x, y;
  x:=1;
  y:=-2;
end module;
X := module () export x, y; option record; end module
```

```
> X:-x , X:-y;
1, -2
```

Les composantes de ce type de variables sont directement assignables. **Attention**, le x et le y utilisés pour exporter les composantes du module sont des noms qui **sont différents** de ceux de variables homonymes x et y . Dans l'exemple suivant le x de $\exp(x)$ est un simple nom de variable.

```
> A:-x:=4: A:-x;
X:-x:=exp(x): X:-x;
4
e^x
```

Aussi des assignations à des variables homonymes sont sans effet sur le fonctionnement du module

```
> x:=Pi: y:=10:   A:-x, A:-y;
4, 2
```

Par contre les règles d'évaluation (voir chapitre 2) sont toujours les mêmes ($\exp(x)$ pour $x = \pi$)

```
> X:=-x, X:=-y;  
eπ, -2
```

La nature des données d'une structure peut être quelconque y compris des structures ou des composantes de structures. C'est ce qui en fait l'intérêt.

```
> Truc:=Record('nom'='Machin', 'x'=exp(z),  
'm'=Matrix([[0,1],[-1,0]]), 'p'=A):
```

```
> Truc:-nom, Truc:-m;  
Machin,  
[ 0 1  
-1 0 ]
```

```
> Truc:-p:-x;  
4
```

Peut-être un exemple plus concret...

```
> Client[1]:=Record(nom='Dugenou', prenom='Jean', age=34,  
ad='9 rue Maple', ville='Waterloo', montant=2347.5):  
Client[1]:-nom, Client[1]:-montant;  
Dugenou, 2347.5
```

Revenons à nos "points du plan". On peut créer des procédures (de type *procedure* ou *operator(arrow)*) pour générer des objets de type *record*.

Remarques: *point* (*p* minuscule) est une mot réservé et protégé de MAPLE et donc inutilisable (c'est un type qui correspond aux objets de type *equation* de la forme *name=algebraic*; $a = x + 1$ est du type *point*). C'est aussi un nom de fonction de la bibliothèque **geometry**. **Point** (*P* majuscule) est un mot réservé de la bibliothèque **student** non accessible ici et donc utilisable.

```
> Point:=(u,v)->Record('x'=u, 'y'=v);  
Point := (u, v) → Record('x' = u, 'y' = v)
```

On écrit alors simplement

```
> A:=Point(sqrt(3),2):  
A:-x, A:-y;  
√3, 2
```

Le point *C* est ici celui d'un cercle de rayon 2 et d'argument θ

```
> C:=Point(2*cos(theta),2*sin(theta)):  
C:-x;  
2 cos(θ)
```

On peut transformer cet objet en une fonction qui possède deux composantes, l'abscisse et l'ordonnée (on peut bien sûr construire *C* directement avec **Record** sans passer par *Point*)

```
> C:=theta->Point(2*cos(theta),2*sin(theta));  
C(Pi/3):-x, C(Pi/3):-y;  
1, √3
```

Point étant une procédure, on peut contrôler ses arguments. Si on souhaite que les coordonnées ne puissent être que du type "constantes réelles" on écrira

```
> Point:=(u::realcons,v::realcons)->Record('x'=u,'y'=v);  
Point:=(u::realcons,v::realcons)→Record('x'=u,'y'=v)
```

Alors, par exemple des arguments symboliques, ne seront plus admis

```
> C:=Point(2*cos(theta),2*sin(theta));
```

```
Error, invalid input: Point expects its 1st argument, u, to be of type realcons, but received 2*cos(theta)
```

Attention: *O* majuscule est un nom réservé (développement en série)

```
> O:=Point(0,0);  
O:=Point(0,0);
```

```
Error, attempting to assign to `O` which is protected
```

On construit un point *T* de même abscisse que *A*.

```
> T:=Point(A:-x,sqrt(5));  
T:-x, T:-y;  
√3, √5
```

Type des objets construits avec Record ou par module avec option record

Le type de base, comme nous l'avons vu, est avant tout un module

```
> whattype(eval(A));  
module
```

mais qui possède le sous-type **record**. On rappelle (voir plus haut) que *X* n'est pas construit avec **Record**, mais avec un module contenant l'option **record**

```
> type(A,record), type(x,record);  
true, true
```

On peut associer des types spécifiques à des objets de type **record**. Ici *pt* est le nom d'un nouveau type associé à un profil de type et adapté à notre construction de "points". On protège aussi le nom ordinaire pour éviter toute confusion bien qu'ils n'interfèrent pas (voir chapitre 2). On pourrait aussi utiliser la fonction **AddType** de la bibliothèque **TypeTools** (voir chapitre 2).

```
> `type/pt`:=record(x::realcons,y::realcons);  
protect('pt');  
type/pt := record(x::realcons, y::realcons)
```

Les objets *A* et *T* sont bien du type *pt* mais pas *C* qui est une fonction.

```
> type(A,'pt'), type(C,'pt'), type(T,'pt');  
true, false, true
```

Voici un nouveau profil *pt_i* pour les "points" de coordonnées entières

```
> `type/pt_i`:=record(x::integer,y::integer):  
protect('pt_i');  
Ip:=Point(2,-3): type(Ip,pt_i);  
type(A,pt_i);  
true
```

false

Procédures avec variables structurées

On peut donc construire des procédures utilisant des variables structurées. Ici, *dist* calcule la distance euclidienne entre deux objets de type *pt*.

```
> dist:=proc(p1::pt,p2::pt)
   simplify(sqrt((p1:-x-p2:-x)^2+(p1:-y-p2:-y)^2));
end proc;
dist := proc(p1::pt, p2::pt) simplify(sqrt(((p1:-x) - (p2:-x))^2 + ((p1:-y) - (p2:-y))^2)) end proc;
```

La distance entre le point *A* et *B* est donc

```
> B:=Point(1,-3*sin(Pi/4)):
dist(A,B), evalf[5](dist(A,B));

$$\frac{1}{2} \sqrt{50 - 8 \sqrt{3} + 24 \sqrt{2}}, 4.1858$$

```

Bloc use...in/end use

Lorsque les procédures sont longues, l'écriture *NOM:-nom* peut devenir fastidieuse. On peut alors utiliser un bloc lexical d'alias **use** *a=A:-x, b=B:-y, ... in/end use:*. Les noms *a, b,...* du texte inclus dans le bloc sont alors remplacés par leurs équivalents *A:-x, B:-y,...* Comme le montre le résultat, la substitution est immédiate (pour une seule instruction son usage est ici d'un intérêt très limité).

```
> dist:=proc(p1::pt,p2::pt)
  use x1=p1:-x, x2=p2:-x, y1=p1:-y, y2=p2:-y  in
    simplify(sqrt((x1-x2)^2+(y1-y2)^2));
  end use;
end proc;
dist := proc(p1::pt, p2::pt) simplify(sqrt(((p1:-x) - (p2:-x))^2 + ((p1:-y) - (p2:-y))^2)) end proc;
```

```
> dist(A,o);

$$\sqrt{7}$$

```

Héritage

La fonction **Record** autorise un mécanisme de construction d'un type à partir d'un autre par *héritage*. On suppose ici que φ_1 décrit les composantes dans l'espace des phases d'une particule d'impulsion (px, py) , de nom *e+* et dont la position est donnée par les coordonnées du point *A*

```
> phi[1]:=Record[A] ('type'='`e+` , `px`=0, `py`=0);

$$\varphi_1 := \text{module } () \text{ export } x, y, \text{type}, px, py; \text{ option record; end module}$$

```

φ_1 est une extension de *A* et désigne un positron au repos dont les coordonnées de position sont $\sqrt{3}, 2$ (d'accord!, ce n'est pas de la mécanique quantique relativiste :-))

```
> phi[1]:=~`type~, [phi[1]:-px,phi[1]:-py], [phi[1]:-x,phi[1]:-y];

$$e+, [0, 0], [\sqrt{3}, 2]$$

```

Ecrivons une procédure qui construit un segment défini à partir des deux points extrémités. On exporte les cordonnées (*x_i,y_i*) de l'extrémité associée au premier point et (*x_f,y_f*) associées au second point.

```
> Seg:=(p1::pt,p2::pt)->
  Record('x_i'=p1:-x,'y_i'=p1:-y,'x_f'=p2:-x,'y_f'=p2:-y):
```

On définit le nouveau type associé

```
> `type/sg`:=record(x_i::realcons,y_i::realcons,
                     x_f::realcons,y_f::realcons):
    protect('sg');
```

On peut ainsi construire un objet "segment" qui a les propriétés attendues

```
> A=(A:-x,A:-y), B=(B:-x,B:-y);
S:=Seg(A,B):
type(S,sg);
S:-x_i, S:-y_f;
```

$$A = (\sqrt{3}, 2), B = \left(1, -\frac{3}{2} \sqrt{2} \right)$$

true

$$\sqrt{3}, -\frac{3}{2} \sqrt{2}$$

On peut, par héritage à partir de *Seg*, construire une nouvelle structure *Segment* qui contient la procédure de calcul de la longueur euclidienne du segment. On remarquera l'utilisation de l'opérateur d'auto-référence **self**: qui permet d'accéder aux composantes de la structure prototype *Seg(p1,p2)*. *Attention*, *long* ne peut pas être associé à une simple expression et doit être calculé à l'aide d'un opérateur de type *operator(arrow)*

```
> Segment:= proc(p1::pt,p2::pt)
  Record[Seg(p1,p2)](long=( ()->
    simplify(sqrt((self:-x_f-self:-x_i)^2 +
      (self:-y_f-self:-y_i)^2))) );
end proc:
```

ou, pour des cas plus compliqués, par un opérateur de type *procedure*. Notez que le **end proc** de ce dernier n'est pas suivi de ; ou :

```
> Segment:= proc(p1::pt,p2::pt)
  Record[Seg(p1,p2)]('long'=
    proc()
      simplify(sqrt((self:-x_f-self:-x_i)^2 +
        (self:-y_f-self:-y_i)^2))
    end proc);
end proc:
```

On définit le nouveau type associé

```
> `type/seg`:=record(x_i,y_i,x_f,y_f,long):
    protect('seg');
```

Cette nouvelle construction définit un nouvel objet "segment" *S*

```
> A=(A:-x,A:-y), O=(o:-x,o:-y);
S:=Segment(A,O):
type(S,seg);
```

$$A = (\sqrt{3}, 2), O = (0, 0)$$

S := module () export x_i, y_i, x_f, y_f, long; option record; end module

true

dont on peut extraire la longueur

```
> S:-long();
```

$$\sqrt{7}$$

Surcharge des opérateurs

On peut redéfinir les opérateurs de MAPLE comme `+`, de façon à ce qu'ils agissent aussi sur des objets ayant des types donnés en effectuant par convention des opérations particulières, mais tout en continuant de fonctionner normalement. Supposons que l'on veuille définir un objet *S* de type "segment" en disant qu'il est la "somme" de deux points *A* et *B*. On pourra écrire $S:=A+B$ en redéfinissant l'opérateur `+` dans un module écrit de la façon suivante

Attention: Si l'on veut surgager ` - ` il faut se rappeler qu'il est interprété par MAPLE comme un opérateur unaire, pas binaire ! ($A-B$ signifie $A+(-B)$; voir l'aide en ligne)

```
> Surcharge:=module()
  export `+`;
  `+`:=proc(p1::pt,p2::pt)
    option overload;
    Segment(p1,p2);
  end proc;
end module;
```

Après avoir donné accès à ce module

```
> with(Surcharge):
```

```
Warning, the protected name `+` has been redefined and unprotected
```

on obtiendra bien le résultat attendu.

```
> S:=A+B:
```

```
> S:-x_i, S:-y_f;
```

$$\sqrt{3}, -\frac{3}{2}\sqrt{2}$$

La surcharge étant construite avec *Segment*, on retrouve l'opérateur de calcul de longueur.

```
> (A+T):-long();
```

$$\sqrt{5} - 2$$

Exercice: expliquez cette erreur

```
> A+T:-long();
```

```
Error, module does not export `long`
```

L'**option overload** introduite dans la définition de `+` permet de ne pas altérer les autres fonctions de l'opérateur. **Autant dire que c'est l'option à ne pas oublier !!!**

```
> 3+z+3-5;
```

$$z + 1$$

On peut donner plusieurs significations à un opérateur avec la syntaxe suivante. On aurait pu créer ainsi un objet de type "triangle" et dire qu'il peut être défini par la "somme point+segment", etc. Mais pour faire plus court on a seulement défini l'opérateur `+` comme étant, soit la construction d'un segment, soit l'**union** de deux ensembles *e1* et *e2* par *e1+e2*. On a aussi défini le *produit cartésien de deux ensembles* en surchargeant l'opérateur `*`

Attention: dans la liste **overload([proc...end proc,...,proc...end proc])**, remarquez que les **end proc** ne se terminent pas par : ou ;

```
> Surcharge:=module()
    export `+`, `*`:
    option package:
`+` :=overload([
    proc(p1::pt,p2::pt)
        option overload:
            Segment(p1,p2);
    end proc,
    proc(e1::set,e2::set)
        option overload:
            e1 union e2;
    end proc]);
`*` :=proc(e1::set,e2::set)
    local X,EP:
    option overload:
        X:=combinat[cartprod]([e1,e2]);
        EP:={}:
        while not X[finished] do
            EP:={op(EP),convert(X[nextvalue](),set)}}
        end do:
        EP;
    end proc;
end module;
```

```
> with(Surcharge):
```

Warning, the protected names `*` and `+` have been redefined and unprotected

En déterminant la nature des objets sur lesquels il opère, MAPLE détecte la nature des opérations à effectuer. *A* et *B* étant de type *pt*, il sait quelle procédure il doit appliquer

```
> S:=A+B: type(S,seg), S:-x_i, 3+z+3-5;
                                         true,  $\sqrt{3}$ , z + 1
```

```
> (A+T):-long();
                                          $\sqrt{5} - 2$ 
```

Pour calculer l'union de deux ensembles il suffira maintenant d'écrire

```
> E1,E2:={a,b,{1,2}}, {u,v}:
  E1 + E2;
                                         {u, b, v, {1, 2}, a}
```

Pour le produit cartésien

```
> E1*E2;
                                         {{v, {1, 2}}, {u, a}, {v, a}, {u, b}, {b, v}, {u, {1, 2}}}}
```

Et les opérations standard sont toujours effectuées grâce à l'**option overload**

```
> 2*(z+1)-3*2;
                                         2 z - 4
```

La construction de modules, de variables structurées avec mécanisme d'héritage, la surcharge des opérateurs et le polymorphisme du langage permet une **programmation orienté objet**.

Enregistrement et lecture de librairies

1) *Création, sauvegarde et modification*

Quand MAPLE trouve un nom, une fonction, etc. dans une commande, il cherche sa définition dans une ou des librairies ou modules qui sont désignées par la variable globale prédéfinie **libname**. Par défaut, c'est la librairie MAPLE désignée ci dessous. La forme de la chaîne de caractères renvoyée est spécifique du système d'exploitation utilisé par l'ordinateur, des noms des disques et des répertoires utilisés.

```
> restart:  
libname;  
"/Applications/Maple 9.5/Maple 9.5.app/Contents/MacOS/lib"
```

L'utilisateur peut créer ses propres librairies d'objets tels des constantes, des expressions, des fonctions, des procédures, des modules etc. On ne donnera ici que quelques informations essentielles sur une façon de procéder parmi d'autres. Pour compléter ces informations sur la gestion et l'optimisation des librairies, voir l'aide en ligne **Repository/management** et l'aide de **march** (**march** est une contraction de *maple archive manager*)

```
> ?repository,management  
> ?march
```

Il faut tout d'abord *créer un répertoire autorisé en écriture* (vide, pour plus de clarté). Ceci peut se faire avec les outils du système d'exploitation ou avec la commande MAPLE **mkdir** (voir l'aide en ligne). Ci-dessous on suppose que le répertoire vide existe déjà (il est désigné par la commande **cat(...)**; pour plus de détails sur cette instruction, se reporter au chapitre 20, voir aussi **currentdir**). On crée la librairie, nommée ici **Ma_librairie.mla**, avec la commande **march('create',...)**. Il existe deux formats (convertibles de l'un vers l'autre avec **march('convert',...)**) que l'on définit par l'extension du nom de librairie, soit **.mla** (Maple Library Archive), soit **.lib**. Avec **.lib** on crée deux fichiers **Ma_librairie.lib** et **Ma_librairie.ind**. Avec **.mla** on crée seulement **Ma_librairie.mla**. On doit fixer la taille approximative du nombre maximum d'objets que l'on prévoit pour celle-ci (ici 100). Cette limite imposée n'est pas absolue et peut être dépassée jusqu'au double de la valeur donnée. Pour un module on comptera le nombre de procédures exportées. *Cette opération n'est à faire qu'une fois à la création d'une librairie.*

```
> march('create', cat(kernelopts(homedir),  
"/Desktop/Rep_lib/Ma_librairie.mla"), 100);
```

Construisons quelques objets que l'on veut sauvegarder dans une librairie. La gestion de librairies peut aussi se faire à partir de codes MAPLE contenus dans des fichiers ASCII. Nous n'aborderons pas ce point.

```
> Ma_constante:=Pi/exp(1):  
  
Ma_fonction :=(x,y)->x^2/y:  
  
Une_procedure:= proc(x)  
if x < 0 then  
exp(-x^2):  
else  
cos(x):  
end if:  
end proc:
```

```

Mon_module := module()
  export f, g;
  f:=x-> exp(-x)/x;
  g:=proc(x)
    if x=0 then;
      1:
    else
      sin(x)/x:
    end if:
  end proc:
end module:

Module_Div:=module()
export Div;
Div:=proc(x,y,k)
option cache;
try
  x/y
catch "numeric exception: division by zero":
  if x = 0 then
    return k;
  else
    return sign(x)*infinity;
  end if;
end try
end proc:
end module:

```

On crée aussi un module qui exporte les noms spéciaux **ModuleLoad** et **ModuleUnload** que nous avons déjà évoqués. Nous verrons, au paragraphe 2) *Utilisation* ci-dessous, comment fonctionnent les procédures associées à ces noms

```

> Init_Stop:=module()
  export f, ModuleLoad, ModuleUnload;
  ModuleLoad:=proc()
    global Cte;
    Cte:=Pi/3;
  end proc;
  ModuleUnload:=proc()
    print(`C'est fini`);
  end proc;
  f:=x->x^2+Cte;
end module:

```

Pour introduire ou modifier des objets dans une librairie il faudra préciser le répertoire qui la contient et son nom en l'assignant à la variable globale prédéfinie de MAPLE, **savelibname**. *S'il n'y a qu'une librairie dans le répertoire on peut se dispenser de préciser son nom en ne donnant que le répertoire* (comme ci-dessous). Il est préférable, pour plus de clarté, de n'avoir qu'une librairie par répertoire. On peut aussi donner sous forme de suite **savelibname:=cat(.../nom_1")**, **cat(.../nom_2")**,... plusieurs librairies. Si la sauvegarde est impossible dans la première, elle se fera dans la seconde ou dans les suivantes

```
> savelibname:=cat(kernelopts(homedir), "/Desktop/Rep_lib");
```

Pour sauvegarder des éléments dans une librairie qui vient d'être précisée avec **savelibname** on utilisera la fonction **savelib** avec la syntaxe suivante.

Attention: ne pas oublier les ' pour éviter toute évaluation.

Attention, une erreur dans un nom n'est pas signalée ("Ma_fonction" et non "Une_fonction").

```
> savelib('Une_fonction', 'Mon_module');
```

L'option '*list*' de **march** fournit la liste des membres de la librairie sous forme de liste de listes. Chaque liste contient le nom, la date et l'heure d'insertion ou de modification dans la librairie et deux entiers d'offset et de taille. On note que les procédures ou les fonctions exportées des modules sont ré-indexés par des entiers *n* sous la forme *:-n.m*. En effet, des modules différents (*M1, M2,...*) peuvent exporter des noms identiques (*f,g,...*). Au niveau du langage ils sont identifiés par des appels *M1:-f, M2:-f, M1:-g, M2:-g*, etc. et il n'y a pas d'ambiguïté. Au niveau de la librairie l'ambiguïté est levée par une ré-indexation par un entier.

```
> march('list', savelibname);
```

```
[[["Mon_module.m", [2005, 9, 1, 8, 13, 6], 41984, 88], ["-1.m", [2005, 9, 1, 8, 13, 6], 42072, 105],  
["-2.m", [2005, 9, 1, 8, 13, 6], 42177, 92]]]
```

Avec l'option '*prefix*' on peut aussi obtenir la liste de tous les noms commençant par un ou des caractères précisés sans une chaîne

```
> march('prefix', savelibname, ":" );  
["-1.m", "-2.m"]
```

On peut obtenir une liste simplifiée avec la commande suivante que l'on a transformée en fonction pour pouvoir la réutiliser.

```
> L_lib:=()>seq(op(1,i), i=march('list', savelibname));  
L_lib();  
"Mon_module.m", "-1.m", "-2.m"
```

On peut naturellement *rajouter ou modifier* un ou des éléments.

```
> Une_fonction:=(x,y)->x^2/y^2: # Modification  
savelib('Une_fonction', 'Ma_constante',  
'Une_procedure', 'Module_Div', 'Init_Stop');  
  
> L_lib();  
"Une_procedure.m", "Module_Div.m", "-3.m", "Ma_constante.m", "Init_Stop.m", "-4.m",  
"Mon_module.m", "-5.m", "-1.m", "-6.m", "-2.m", "Une_fonction.m"
```

Pour connaître l'entier associé à un nom exporté on écrira

```
> march('moduleref', savelibname, Mon_module:-f),  
march('moduleref', savelibname, Module_Div:-Div);  
:-1, :-3
```

alors que pour connaître les noms exportés par un module

```
> eval(Mon_module);  
module () export f, g; end module
```

Pour connaître la définition d'un nom exporté il suffit de préciser

```
> eval(Mon_module:-f);
```

```

eval(Mon_module:-g);

$$x \rightarrow \frac{e^{(-x)}}{x}$$

proc(x) if x = 0 then 1 else (sin(x))/(x) end if; end proc;

```

L'option '**delete**' de **march** permet d'effacer un ou plusieurs éléments de la librairie

```
> march('delete',savelibname,'Mon_module','Une_procedure');
```

Le module *Mon_module* et la procédure *Une_procedure* ont bien disparus

```

> march('prefix',savelibname,"M");
march('prefix',savelibname,"U");
      ["Module_Div.m", "Ma_constante.m" ]
      ["Une_fonction.m"]

```

Mais attention, les références des objets exportées par le module effacé n'ont pas pour autant disparues (*:-1.m* et *:-2.m* sont associées à *Mon_module* qui vient d'être effacé)

```
> march('prefix',savelibname,:');
      [":-3.m", ":-4.m", ":-5.m", ":-1.m", ":-6.m", ":-2.m"]
```

et il faut "purger les fossiles et scories" avec l'option '**gc**' de **march** (garbage collects)

```

> march('gc',savelibname):
> march('prefix',savelibname,:');
      [":-3.m", ":-4.m", ":-5.m", ":-6.m"]

```

Quand on pense en avoir terminé avec la construction de la librairie il est recommandé de purger, ré-indexer et compresser la librairie

```
> map(march,['gc','reindex','pack'],savelibname):
```

```
> L_lib();
":::-6.m", "Ma_constante.m", ":-3.m", ":-4.m", "Init_Stop.m", "Une_fonction.m", ":-5.m", "Module_Div.m"
```

et finalement protéger la librairie en écriture avec l'option '**setattribute**' et le '**mode**'="**READONLY**" de **march**. Pour pouvoir opérer à nouveau des modifications on devra appliquer l'option inverse '**setattribute**', '**mode**'="**WRITABLE**".

```
> march('setattribute',savelibname,'mode'='READONLY');
```

Pour obtenir le **nombre d'éléments** contenus dans la librairie on écrira

```
> nops(march('list',savelibname));
```

8

Exercice: voici un résultat plutôt impressionnant qui montre que ce manuel est bien loin d'avoir tout dit!
Interprétez

```
> nops(march('list',libname));
87940
```

Création de pages d'aide

L'utilisateur peut créer sa propre base de **pages d'aide** associées à sa librairie. Ces pages peuvent être écrites en texte ASCII mais le plus simple et le plus efficace est de les créer en enregistrant une feuille de

calcul que l'on aura composée spécialement (avec des exemples exécutés, c'est mieux !). On a composé une feuille de calcul avec des exemples pour la procédure *Div* du module *Module_Div* que l'on a enregistrée sous le nom *Div_help.mw* (on choisit le nom que l'on veut). A l'aide de la commande **makehelp** suivante la feuille sera intégrée au fichier **maple.hdb** de la librairie contenant *Module_Div/Div* qui est placée dans le répertoire précisé par **savelibname** (si le fichier **maple.hdb** n'existe pas, il sera créé automatiquement). Si vous êtes sûr de ne pas avoir à la modifier, vous pouvez détruire la feuille de calcul *Div_help.mw*.

Attention, savelibname, (voir le début du paragraphe), ne contient dans notre exemple que le nom du répertoire (c'est possible ici car ce dernier ne contient qu'une seule librairie). Si **savelibname** précise le nom du répertoire **et** le nom de la librairie on devra remplacer, dans la commande ci-dessous, **savelibname** par seulement le nom du répertoire qui contient la librairie.

```
> makehelp(`Module_Div/Div`,  
cat(kernelopts(homedir),"/Desktop/Div_help.mw"),  
savelibname):
```

Pour ajouter ou supprimer une feuille d'aide dans une librairie d'aide on pourra utiliser les items *Save to Database...* ou *Remove Topic* dans le menu **Help**.

L'information donnée ici est minimale. Pour la compléter, voir... l'aide en ligne

```
> ?worksheet,reference,addhelp
```

2) Utilisation

Lors d'une nouvelle session MAPLE ou après un ordre **restart**, toutes les définitions étant perdues, on pourra retrouver les objets sauvegardés. Au départ seule la bibliothèque par défaut est connue de MAPLE

```
> restart:  
libname;  
"/Applications/Maple 9.5/Maple 9.5.app/Contents/MacOS/lib"
```

Naturellement, les objets précédents sont devenus indéfinis

```
> Ma_constante;  
Ma_constante
```

Avec l'assignation suivante on va indiquer de chercher les définitions **d'abord** dans la bibliothèque désignée en premier **puis** dans la bibliothèque standard (encore désignée par *libname*). On peut bien sûr associer autant de bibliothèques que l'on souhaite. On rappelle que l'on peut se contenter de donner seulement le nom du répertoire de la librairie car il n'en contient qu'une.

```
> libname:=cat(kernelopts(homedir),"/Desktop/Rep_lib"),libname:  
> libname;  
"/Users/Jean/Desktop/Rep_lib", "/Applications/Maple 9.5/Maple 9.5.app/Contents/MacOS/lib"
```

Exercice: que se passerait-il si on exécutait à nouveau la commande > **libname:=cat(...),libname:** ?

Si cette opération doit être systématique on pourra la préciser dans le fichier d'initialisation de MAPLE (voir chapitre 1, partie 2, *Commentaires sur l'utilisation de Maple*, § IV-5), ce qui évitera de la répéter à chaque fois.

Et MAPLE retrouvera les définitions...

```
> Module_Div:-Div(-3,0);
```

-∞

```
> 2*Ma_constante+1;

$$\frac{2\pi}{e} + 1$$

```

```
> seq(op(1,i), i=march('list', libname[1]));
":-6.m", "Ma_constante.m", "-3.m", "-4.m", "Init_Stop.m", "Une_fonction.m", "-5.m", "Module_Div.m"
```

Exercice : pourquoi n'a-t-on pas intérêt à appliquer cette commande sur `libname[2]` ? (c'est sans danger, mais...)

On retrouvera aussi les pages d'aide créées par l'utilisateur pour sa bibliothèque.

```
> ?Module_Div/Div
```

Le nom `Cte` n'est pas défini

```
> Cte;
```

Cte

Mais le module `Init_Stop` de la librairie contient trois objets dont un porte le nom **ModuleLoad** qui assigne ce nom ($Cte := \pi/3$) (voir le paragraphe précédent). Le fait de faire appel à un des éléments de ce module ($f(x)$ calcule $x^2 + Cte$) active automatiquement l'objet **ModuleLoad**, d'où le résultat

```
> Init_Stop:-f(x);
```

$$x^2 + \frac{1}{3}\pi$$

Comme `Cte` a été définie globale, elle reste définie

```
> Cte;
```

$$\frac{1}{3}\pi$$

Quand le module n'est plus accessible, l'objet **ModuleUnload** de notre module `Init_Stop` s'active automatiquement.

```
> restart;
```

C'est fini

Sauvegarde simple et lecture d'un ou plusieurs objets

On peut aussi sauvegarder un ou plusieurs objets avec `save` dans un fichier sans avoir à créer une bibliothèque. Recréons quelques objets en les relisant dans notre librairie pour faire court

```
> libname:=cat(kernelopts(homedir), "/Desktop/Rep_lib"), libname:
Module_Div:-Div(-3,0);
Ma_constante;
```

$$\begin{aligned} & -\infty \\ & \frac{\pi}{e} \end{aligned}$$

On sauvegarde en écrivant (sans mettre les noms entre ')

```
> save Module_Div, Ma_constante,
      cat(kernelopts(homedir), "/Desktop/Sauvegarde.sav");
```

Lors d'une nouvelle session ou après un ordre `restart`, les définitions sont perdues, mais après lecture avec

read, on les retrouve.

```
> restart;
read cat(kernelopts(homedir), "/Desktop/Sauvegarde.sav"):
Ma_constante;
Module_Div:=Div(-3,0);
```

$$\frac{\pi}{e}$$

$-\infty$

20 - Lecture, écriture et fichiers

Il est important de pouvoir traiter avec MAPLE des données provenant de fichiers externes et réciproquement d'y écrire des résultats obtenus

```
> restart;
```

Lecture interactive de données au clavier

1) La fonction readstat

La fonction **readstat** permet d'introduire des données de façon interactive. Après exécution de l'ordre, une zone s'ouvre en mode "input" et attend la frappe du clavier. **Attention, ne pas oublier de terminer par ";".** Sinon une nouvelle zone s'ouvre et attend la suite; on pourra alors taper le ";" même seul.

```
> x:=readstat();
```

```
5.3456e-2;
```

```
x := 0.053456
```

Cette fonction autorise l'entrée de données symboliques et pas seulement numériques. On notera que les expressions sont évaluées (`expand(sin(2*b))`). On entre ici une *liste* "d'objets" divers:

```
> z:=readstat();
```

```
[a,expand(sin(2*b)), "Chaine de caracteres", {u,x}];  
z := [a, 2 sin(b) cos(b), "Chaine de caracteres", {u, 0.053456}]
```

Pour améliorer la lisibilité on peut utiliser un "prompt" introduit en premier argument par une chaîne de caractères.

```
> q:=readstat("q = ");
```

```
q = evalf(sin(Pi/4));
```

```
q := 0.7071067810
```

Afin d'éviter d'avoir à taper des entrées répétitives et/ou longues, on pourra utiliser 3 dito, introduits en arguments sous forme de listes (même pour un élément). L'ordre **impératif** des arguments des dito correspond à %%%, %% et %.

Attention, on doit toujours donner, soit aucun, soit 3 dito, même si on n'en utilise que un ou deux.

Attention, si on corrige directement la ligne de données, les dito deviennent ceux de la feuille de calcul et non ceux de **readstat** !

```
> L:=[readstat("--> ",[alpha,beta],[0,0,0],[q])];
```

```
--> -1,%, -2,%%%, a, b, %%, x, y;
```

```
L := [-1, 0.7071067810, -2,  $\alpha$ ,  $\beta$ ,  $a$ ,  $b$ , 0, 0, 0, 0.053456,  $y$ ]
```

Voici un exemple de procédure permettant de donner au clavier un à un les termes d'une matrice $n \times m$ en "promptant" chaque entrée avec l'identification de chaque élément. On considère ici que la valeur de q sera très souvent utilisée et on l'utilisera comme dito (rappel: **posint** => entier strictement positif).

```

> M_data:=proc(n::posint,m::posint)
  local i,j,Lm,L;
  Lm:=[];
  for i to n do
    L:=[];
    for j to m do
      L:= [op(L),readstat(cat("M[ ",i,",",j,"]= ")),[0],[0],[sqrt(2)/2]];
    end do;
    Lm:=[op(Lm),L];
  end do;
  Matrix(Lm);
end proc:

```

```

> A:=M_data(2,2);

```

```

M[1,1]= 1.0;

```

```

M[1,2]= 2%;

```

```

M[2,1]= -%/2+1;

```

```

M[2,2]= 1+exp(u);

```

$$A := \begin{bmatrix} 1.0 & 1.414213562 \\ 0.6464466095 & 1 + e^u \end{bmatrix}$$

2) La fonction readline

Bien que cette fonction soit plutôt utilisée pour lire des fichiers (voir plus loin dans ce chapitre), on peut en utilisant le mot clé **terminal**, entrer des données de façon interactive. Une fenêtre s'ouvre et attend la frappe. Inutile ici de terminer par un ";". **Le résultat est une chaîne de caractères**. Ici on a tapé dans la fenêtre: 3.987e-2,3.56 (sans les guillemets).

```

> restart:
S:=readline(terminal);

```

$$S := "3.987e-2,3.56"$$

Pour transformer cette chaîne de caractère, par exemple en liste, on pourra utiliser la fonction **parse** (voir le chapitre 18, *Eléments de programmation, § Symboles et chaîne de caractères*)

```

> s:=[parse(S)];

```

$$s := [0.03987, 3.56]$$

Une chaîne entrée peut être plus complexe

```

> S:=readline(terminal);

```

$$S := "x1=34 x2=3.675e-1"$$

et décodée partiellement avec **parse**

```

> L:=[parse(cat(S[4..5],",",S[10..-1]))];

```

$$L := [34, 0.3675]$$

ou avec la fonction **sscanf** (voir plus loin dans ce chapitre)

```

> L:=sscanf(S,"x%*d=%d x%*d=%g");

```

Lecture de données dans un fichier

Nous considérerons deux situations, une pour laquelle le fichier ne contient que des valeurs numériques et une autre, plus générale, pour laquelle le fichier contient des valeurs numériques mélangées à du texte.

1) Désignation du fichier

La première opération consiste à préciser à MAPLE le fichier en donnant sa localisation et son nom. Avec la commande **kernelopts(homedir)** on va récupérer, sous forme d'une chaîne de caractères, le nom du répertoire d'entrée de l'utilisateur (*kernelopts=kernel options*: cette fonction possède 42 possibilités). On va ensuite, avec **cat**, concaténer (juxtaposer) cette chaîne avec celle qui désigne le(s) sous-répertoire(s) et finalement le nom du fichier. *La syntaxe est évidemment dépendante du système d'exploitation de l'ordinateur utilisé.*

```
> restart;
fic:=cat(kernelopts(homedir), "/Desktop/Data_ch_20/Data_1.dat");
          fic := "/Users/Jean/Desktop/Data_ch_20/Data_1.dat"
```

En raison du rôle particulier de "continuation" joué dans MAPLE par le caractère "\", pour le système d'exploitation **Windows** on doublera le caractère \ (voir chapitre 18, § *Symboles et chaînes de caractères*)

```
> fic:=cat(kernelopts(homedir), "\\Bureau\\Data_ch_20\\Data_1.dat");
```

Pour les systèmes Unix, Linux ou MacOS X on peut aussi récupérer la variable d'environnement (MAPLE) **HOME** qui désigne aussi le nom du répertoire d'entrée de l'utilisateur

```
> fic:=cat(getenv(HOME), "/Desktop/Data_ch_20/Data_1.dat");
          fic := "/Users/Jean/Desktop/Data_ch_20/Data_1.dat"
```

Notez également la fonction de positionnement des répertoires, **currentdir**, qui permet d'imposer le répertoire courant. La réponse rendue par cette fonction donne le répertoire courant *avant* que le changement soit effectué. Il s'agit ici du répertoire qui contient l'application MAPLE sur l'ordinateur utilisé.

```
> currentdir(cat(kernelopts(homedir), "/Desktop/Data_ch_20"));
          /Applications/Maple 9.5/Maple 9.5.app/Contents/MacOS/bin.APPLE_PPC OSX
```

Le répertoire courant est maintenant celui qui a été imposé

```
> currentdir();
          "/Users/Jean/Desktop/Data_ch_20"
```

Il existe encore d'autres fonctions de gestion des répertoires comme **rmdir** ou **mkdir**.

2) Fonction **readdata**

On prendra pour exemple le fichier *fic* précédent dans lequel les données sont organisées en colonnes séparées par un ou plusieurs espaces. C'est la situation la plus fréquente.

-6.00	1.70	37	
-5.00	3.85	26	
-0.040e2	5.86	17	Commentaire_1
-3.00	7.83	10	
-2.00	9.71e0	5	

```

-1.00      11.8      2    Commentaire_2
0.000      13.7      1
1.00       15.7      1
20.E-1     17.8      2
3.00       19.7      7
4.0e0      21.3      13

```

Pour lire ce type de fichier, la fonction la plus pratique est **readdata**. La syntaxe suivante ne permet de lire que la première colonne, ce qui reste d'un intérêt limité...

```
> F:=readdata(fic);
F:=[-6.00, -5.00, -4.0, -3.00, -2.00, -1.00, 0., 1.00, 2.00, 3.00, 4.0]
```

Remarque: le répertoire courant que nous avons fixé avec **currentdir** étant celui qui contient le fichier, on peut aussi écrire

```
> F:=readdata("Data_1.dat");
F:=[-6.00, -5.00, -4.0, -3.00, -2.00, -1.00, 0., 1.00, 2.00, 3.00, 4.0]
```

Si l'on veut lire les deux premières colonnes, on écrira (**Attention:** on ne peut pas lire chaque ligne en commençant par la n -ième colonne et en ignorant les précédentes). Si le fichier contient beaucoup de valeurs, on prendra soin de terminer les commandes par ":" pour éviter un affichage vite encombrant

```
> F:=readdata(fic,2);
F:=[[ -6.00, 1.70], [-5.00, 3.85], [-4.0, 5.86], [-3.00, 7.83], [-2.00, 9.71], [-1.00, 11.8], [0., 13.7],
[1.00, 15.7], [2.00, 17.8], [3.00, 19.7], [4.0, 21.3]]
```

Il est **important** de noter que **readdata** restitue les valeurs lues sous la forme d'une **liste de listes**. A chaque liste de la liste correspondant une ligne du fichier. De plus on notera ici que la troisième colonne du fichier contient des entiers qui sont, **par défaut**, interprétés comme des décimaux par **readdata**

```
> F:=readdata(fic,3);
F:=[[ -6.00, 1.70, 37.], [-5.00, 3.85, 26.], [-4.0, 5.86, 17.], [-3.00, 7.83, 10.], [-2.00, 9.71, 5.],
[-1.00, 11.8, 2.], [0., 13.7, 1.], [1.00, 15.7, 1.], [2.00, 17.8, 2.], [3.00, 19.7, 7.], [4.0, 21.3, 13.]]
```

Si l'on veut respecter les types (entiers, décimaux, chaînes de caractères) on écrira

```
> F:=readdata(fic,[float,float,integer,string]);
F:=[[ -6.00, 1.70, 37], [-5.00, 3.85, 26], [-4.0, 5.86, 17, "Commentaire_1"], [-3.00, 7.83, 10],
[-2.00, 9.71, 5], [-1.00, 11.8, 2, "Commentaire_2"], [0., 13.7, 1], [1.00, 15.7, 1], [2.00, 17.8, 2],
[3.00, 19.7, 7], [4.0, 21.3, 13]]
```

Le troisième élément de chaque liste est maintenant un entier, comme dans la troisième colonne du fichier. On a même lu les commentaires avec **string**. **Attention:** comme ceux-ci ne sont pas présents sur chaque ligne, toutes les listes n'ont pas le même nombre de termes (ce serait aussi bien sûr le cas avec des nombres). Finalement on va ignorer les commentaires. On rappelle que l'on peut aussi utiliser l'opérateur de répétition \$.

```
> F:=readdata(fic,[float$2,integer]):
```

Pour lire ce fichier on pourra aussi utiliser les fonctions **readline** et **parse** (voir un des paragraphes ci-dessous)

Manipulation des données

Avec cette organisation en liste de listes, la manipulation des données est relativement facile. Voici quelques exemples. On veut connaître le nombre de lignes du fichier (on compte les opérandes de la liste principale)

```
> Nl:=nops(F);
```

$Nl := 11$

ou le nombre de colonnes (on compte les opérandes de la première liste)

```
> Nc:=nops(F[1]);  
Nc := 3
```

Si on se reporte au chapitre 16, *Graphisme 2D*, § *Tracés de données discrètes*, on verra que pour tracer des valeurs d'abscisses x_i et de leurs ordonnées correspondantes y_i , on doit les présenter sous la forme d'une liste de listes $[[x_1, y_1], [x_2, y_2], \dots, [x_n, y_n]]$. Si on veut tracer les valeurs de la **troisième colonne en fonction de la deuxième** on pourra écrire

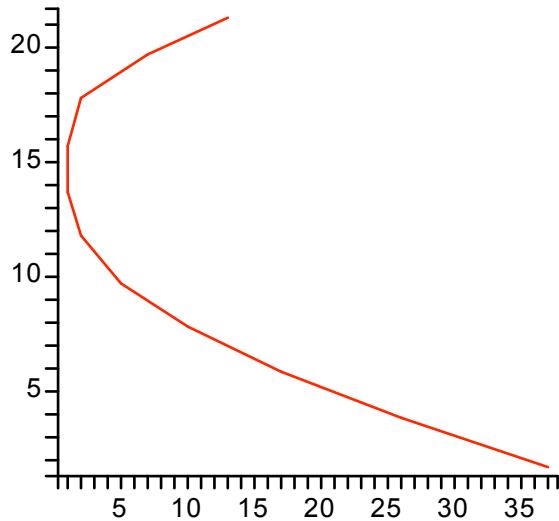
```
> A:=F[1..-1,2..3]; # ou bien  
A:=F[1..Nl,2..3]:  
A := [[1.70, 37], [3.85, 26], [5.86, 17], [7.83, 10], [9.71, 5], [11.8, 2], [13.7, 1], [15.7, 1], [17.8, 2],  
[19.7, 7], [21.3, 13]]
```

Si on veut tracer maintenant les valeurs de la **deuxième colonne en fonction de la troisième** on inversera les données de chacune des listes (voir chapitre 3, § *Listes*)

```
> A:=map(ListTools[Reverse],F[1..Nl,2..3]):  
A := [[37, 1.70], [26, 3.85], [17, 5.86], [10, 7.83], [5, 9.71], [2, 11.8], [1, 13.7], [1, 15.7], [2, 17.8],  
[7, 19.7], [13, 21.3]]
```

Exercice: ...ce qui est très différent de ceci. Que fait-on ?

```
> B:=ListTools[Reverse](F[1..Nl,2..3]):  
> plot(A,title="Tracé 2-ième col. vs 3-ième");  
Tracé 2-ième col. vs 3-ième
```



Mais on peut trouver bien d'autres façon de procéder. Les écritures précédentes ne fonctionnent que pour deux colonnes adjacentes. Si ce n'est pas le cas on pourra écrire pour sélectionner par exemple la troisième puis la première colonne

```
> A:=[seq([k[3],k[1]],k in F)];  
A := [[37, -6.00], [26, -5.00], [17, -4.0], [10, -3.00], [5, -2.00], [2, -1.00], [1, 0.], [1, 1.00], [2, 2.00],  
[7, 3.00], [13, 4.0]]
```

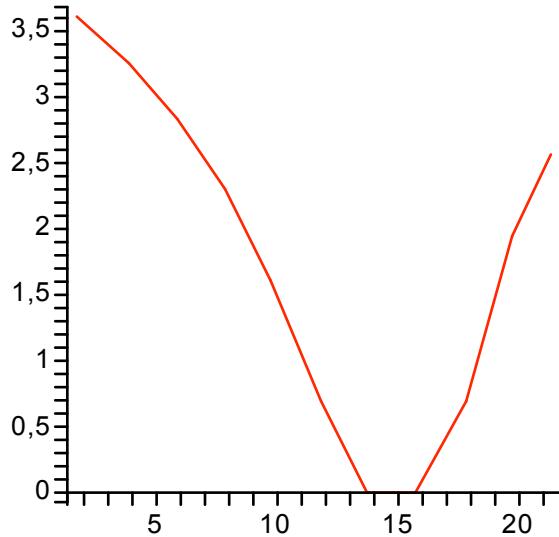
ou encore

```
> A:=[seq([F[i][3],F[i][1]],i=1..nops(F))];  
A := [[37, -6.00], [26, -5.00], [17, -4.0], [10, -3.00], [5, -2.00], [2, -1.00], [1, 0.], [1, 1.00], [2, 2.00],  
[7, 3.00], [13, 4.0]]
```

Si l'on doit faire de nombreuses manipulations on pourra créer des fonctions qui effectueront ces tris mais également contiendront des expressions mathématiques de transformation.

```
> Tri_log:=(L,y,x)->[seq([k[y],ln(k[x])],k in L)];  
Tri_log := (L, y, x) → [seq([ky, ln(kx)], k ∈ L)]
```

```
> plot(Tri_log(F,2,3),title="Tracé 2-ième col. vs ln(3-ième));  
Tracé 2-ième col. vs ln(3-ième)
```



La structure "liste de listes" permet également la construction de **matrices** (ou de **tableaux**), ce qui ouvre l'accès à toutes les fonctions de la bibliothèque **LinearAlgebra**. L'affichage sera alors conforme à ce type d'objet. On peut aussi introduire la lecture du fichier comme argument d'une fonction (sous réserve de compatibilité)

```
> M:=Matrix(readdata("Data_1.dat", [float$2,integer]));  
M := 
$$\begin{bmatrix} 11 \times 3 \text{ Matrix} \\ \text{Data Type: anything} \\ \text{Storage: rectangular} \\ \text{Order: Fortran_order} \end{bmatrix}$$

```

```
> m:=M[2..4,2..3];
```

```
m := 
$$\begin{bmatrix} 3.85 & 26 \\ 5.86 & 17 \\ 7.83 & 10 \end{bmatrix}$$

```

```
> v:=M[2..4,1];
```

$$v := \begin{bmatrix} -5.00 \\ -4.0 \\ -3.00 \end{bmatrix}$$

La conversion en listes de listes ou listes simples est toujours possible. On est alors ramené à la situation décrite plus haut

```
> LL:=convert(m,listlist);
L:=convert(v,list);
LL := [[3.85, 26], [5.86, 17], [7.83, 10]]
L := [-5.00, -4.0, -3.00]
```

On peut par exemple transformer en listes les colonnes de la matrice en leur donnant automatiquement à chacune un nom distinct

```
> for i to Nc do
  C||i:=convert(M[1..Nl,i],list);
end do;
C1 := [-6.00, -5.00, -4.0, -3.00, -2.00, -1.00, 0., 1.00, 2.00, 3.00, 4.0]
C2 := [1.70, 3.85, 5.86, 7.83, 9.71, 11.8, 13.7, 15.7, 17.8, 19.7, 21.3]
C3 := [37, 26, 17, 10, 5, 2, 1, 1, 2, 7, 13]
```

Si on veut calculer la moyenne ses éléments de la première colonne on écrira

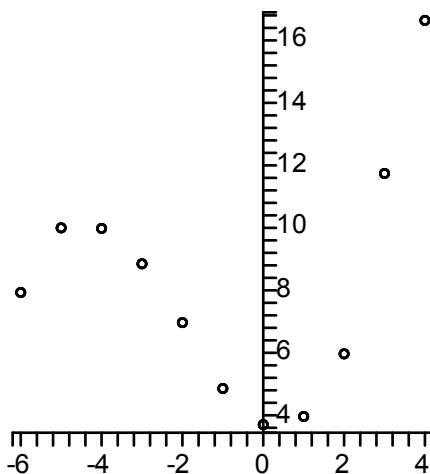
```
> add(i,i in C1)/nops(C1);
-1.000000000
```

Pour la liste des moyennes quadratiques des termes de la deuxième et dernière colonne on écrira

```
> Cq:=zip((x,y)->evalf[4](sqrt(x*y)),C2,C3);
Cq := [7.931, 10.00, 9.981, 8.849, 6.968, 4.858, 3.701, 3.962, 5.967, 11.74, 16.64]
```

que l'on peut tracer sur un graphique en fonction des valeurs de la première colonne

```
> P:=plot(zip((x,y)->[x,y],C1,Cq),style=point,
  symbol=circle,symbolsize=10,color=black):
P;
```



Exercice: analysez la syntaxe de cette fonction. Que fait-elle ?

```
> mu:=n->add(k,k=C||n)/nops(C||n);

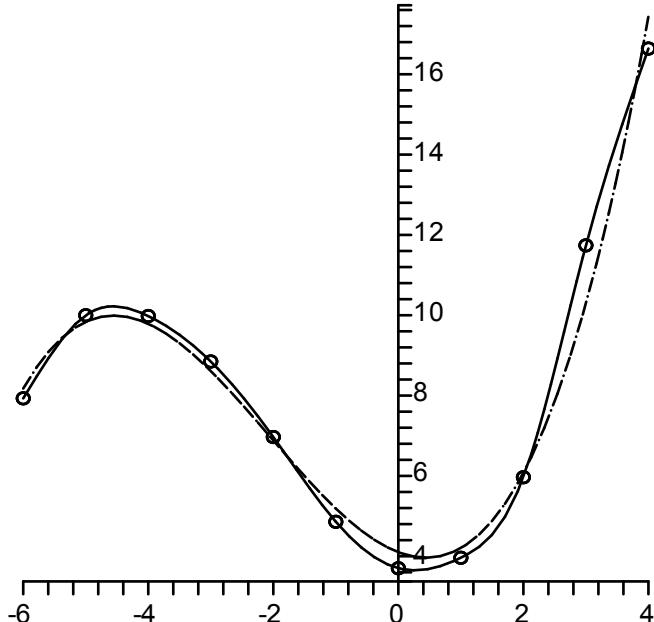
$$\mu := n \rightarrow \frac{\text{add}(k, k = C \parallel n)}{\text{nops}(C \parallel n)}$$

```

On calcule maintenant une spline et une cubique au sens des moindres carrés que l'on trace ensuite sur une même représentation graphique (voir chapitre 7, *Fonctions*).

Attention: si le fichier contient beaucoup de lignes, la spline sera très gourmande en mémoire.

```
> S:=CurveFitting[Spline](C1,Cq,x):
MC:=CurveFitting[LeastSquares](C1,Cq,x,
                                curve=a*x^3+b*x^2+c*x+d):
> plots[display]([P,
                  plot(S,x=C1[1]...C1[-1],color=black),
                  plot(MC,x=C1[1]...C1[-1],linestyle=2,color=black)]);
```



etc... On dispose ainsi de toute la puissance de MAPLE pour traiter les données avec la possibilité de créer aussi ses propres bibliothèques de fonctions spécialisées. On se reportera aussi au chapitre 10, *Intégration*, où l'on trouvera un exemple relatif aux FFT.

3) Fonction readline

Nous disposons d'un fichier *Data_2.dat* contenant les informations suivantes

valeurs mesurées

```
x1= 0      T= 3.e-3    P= 23. L1= 2.34
x2= 1      T= 2.5e1    P= 45.e5 L2= 25
x3= 3      T= 2.456    P= -1 L3= 2.57e-2    #Commentaires
x4= 5      T= -2.e4   P= 35 L4= -2.00
x5= 7      T= -2.e6   P= -25. L5= -45
```

Malgré son apparence ce fichier présente une certaine organisation dans son écriture: à l'exception des premières lignes, chaque élément d'information est composé d'une suite de caractères (par exemple `x1=` ou `3.e-3`) *tous séparés par un ou plusieurs espaces*. Il est donc possible de lire ce fichier avec **readdata** en spécifiant correctement la liste `[string,integer,string,float,...]`

```
> restart;
currentdir(cat(kernelopts(homedir), "/Desktop/Data_ch_20")):
A:=readdata("Data_2.dat",
  [string,integer,string,float,string,float,string,float]):
A[1];
A[2];
                                ["Valeurs"]
["x1=", 0, "T=", 0.0030, "P=", 23.0, "L1=", 2.34]
```

En prenant soin d'éviter la première ligne (`A[1]`) et en sélectionnant un élément sur deux dans chaque liste, on récupère les valeurs numériques dans une liste de listes. On se retrouve donc dans la situation du paragraphe précédent

```
> L:=[seq([seq(A[k,2*i],i=1..4)],k=2..nops(A))];
L := [[0, 0.0030, 23.0, 2.34], [1, 25., 4.50 106, 25.], [3, 2.456, -1., 0.0257], [5, -20000., 35., -2.00],
[7, -2.0 106, -25.0, -45.]]
```

Maintenant le fichier *Data_3.dat* est presque identique à *Data_2.dat*, mais les espaces ne séparent plus les zones critiques (par exemple `x1=0` au lieu de `x1= 0`) et **readdata** ne peut plus faire les séparations. On est obligé de tout lire en mode **string** ce qui rend l'interprétation des valeurs numériques plus difficile.

Valeurs mesurées

```
x1=0      T=3.e-3    P= 23. L1= 2.34
```

```
...
```

```
> A:=readdata("Data_3.dat", [string$4]):
A[1];
A[2];
                                ["Valeurs", "mesurées"]
["x1=0", "T=3.e-3", "P=23.", "L1=2.34"]
```

La fonction **readline** va lire, à chaque appel, une nouvelle ligne du fichier. Elle renvoie le résultat sous forme d'une chaîne de caractères, c'est-à-dire *un seul* objet du *type string* (même si le fichier ne contenait que des valeurs numériques).

```
> readline("Data_3.dat");# première ligne du fichier
type(% ,string);
                                "    Valeurs mesurées"
                                true

> readline("Data_3.dat"); # La ligne est vide
type(% ,string);
                                ""
```

```

true

> readline("Data_3.dat");
type(% ,string);
" x1=0 T=3.e-3 P=23. L1=2.34"

true

> readline("Data_3.dat");
readline("Data_3.dat");
readline("Data_3.dat");
readline("Data_3.dat");
" x2=1 T=2.5e1 P=45.e5 L2=25"
" x3=3 T=2.456 P=-1 L3=2.57e-2 #Commentaires"
" x4=5 T=-2.e4 P=35 L4=-2.00"
" x5=7 T=-2.e6 P=-25. L5=-45"

```

Quand **readline** atteint la fin du fichier elle renvoie un 0 qui devient un objet de type **numeric**.

```

> readline("Data_3.dat");
type(% ,numeric);
0
true

```

Le fichier peut ainsi être lu par une boucle comme celle-ci qui affiche aussi chaque ligne lue avec un **print**. On notera que la variable "ligne" doit être, soit non assignée, soit assignée à "quelque chose" qui ne soit pas 0 si l'on veut que la boucle s'exécute..!

```

> ligne:='ligne':
while ligne <> 0 do
  ligne:=readline("Data_3.dat"):
  print(ligne)
end do:
          "    Valeurs mesurées"
          ""
" x1=0 T=3.e-3 P=23. L1=2.34"
" x2=1 T=2.5e1 P=45.e5 L2=25"
" x3=3 T=2.456 P=-1 L3=2.57e-2 #Commentaires"
" x4=5 T=-2.e4 P=35 L4=-2.00"
" x5=7 T=-2.e6 P=-25. L5=-45"
0

```

Exercice: quand on essaie d'exécuter cette boucle, il ne se passe rien. Pourquoi ?

```

> while ligne <> 0 do
  ligne:=readline("Data_3.dat"):
  print(ligne)
end do:

```

4) Fonction sscanf

Ce qui nous intéresse ici ce sont plutôt les valeurs numériques alors que ce que nous obtenons à chaque lecture est simplement une chaîne de caractères. Il est donc nécessaire de décoder chacune de ces chaînes pour en extraire l'information. Cette opération se fait à l'aide de la fonction **sscanf** (attention au double s) qui reçoit pour premier argument la chaîne à décoder (ici *ligne*) et pour second argument un format de décodage. Celui-ci est constitué d'une chaîne de caractères inspirés du langage C (c'est l'équivalent de ce que l'on appelle un Format en Fortran). Elle indique comment doit être interprété chaque élément d'information de la chaîne *ligne*, c'est-à-dire chaque sous chaîne séparée par un ou plusieurs espaces.

Evidemment il faut lire ici les deux premières lignes sans les décoder.

```
> restart:  
currentdir(cat(kernelopts(homedir), "/Desktop/Data_ch_20")):  
for i to 2 do readline("Data_3.dat") end do:  
  
> ligne:=readline("Data_3.dat");  
ligne := "x1=0 T=3.e-3 P=23. L1=2.34"
```

Lorsque l'on veut décoder avec **sscanf** un élément comme une simple chaîne de caractères on utilise le descripteur **%s** (s pour string) alors que pour décoder un nombre décimal ou un entier on utilisera respectivement **%g** et **%d**. Le résultat est une *liste* constituées des éléments décodés. Comme on le voit sur l'exemple suivant, la liste de décodage peut être plus courte que le nombre d'éléments de la chaîne et la liste résultat sera remplie en conséquence. On sépare très simplement la partie texte et numérique à l'aide d'un gabarit, par exemple **T=%g**.

```
> sscanf(ligne, "%s T=%g P=%g");  
[x1=0, 0.0030, 23.0]
```

Un élément doit être décodé pour pouvoir accéder au suivant, mais il peut être omis de la liste résultat en insérant le caractère *. Ici **x1=0** et **T=3.e-3** sont décodés en tant que chaînes de caractères (**%s**) mais éliminés de la liste par l'introduction de *.

```
> sscanf(ligne, "%*s %*s P=%g L1=%g");  
[23.0, 2.34]
```

La ligne contient **x1** et **L1**, mais qui deviennent à la ligne suivante **x2**, **L2**, etc. Si l'on veut décoder les valeurs associées aux **xn**, il faut, pour pouvoir lire toutes les lignes, adapter le gabarit. Les **n** de **xn** et **Ln** seront décodés, mais éliminés par **%*d**. On pourrait bien sûr ne pas éliminer les valeurs **n** pour pouvoir, par exemple, effectuer un tri si les lignes étaient données dans le désordre dans le fichier.

```
> sscanf(ligne, "x%*d=%d T=%g P=%g L%*d=%g");  
[0, 0.0030, 23.0, 2.34]
```

Un pointeur a conservé la mémoire du numéro de la ligne lue dans le fichier et la boucle suivante continue à partir de la ligne **x2=...**

```
> while ligne <> 0 do  
  ligne:=readline("Data_3.dat"):  
  if ligne <> 0 then  
    print(sscanf(ligne, "x%*d=%d T=%g P=%g L%*d=%g")):  
  end if;  
end do:  
[1, 25., 4.50 106, 25.]  
[3, 2.456, -1., 0.0257]
```

```
[5, -20000., 35., -2.00]
```

```
[7, -2.0 106, -25.0, -45.]
```

Relisons le fichier en donnant comme résultat une liste L unique de listes (une par ligne). On élimine d'abord les deux premières lignes. Le résultat obtenu nous ramène donc à la situation d'une lecture avec **readdata**

```
> for i to 2 do
    ligne:=readline("Data_3.dat")
end do:
L:=[];
while ligne <> 0 do
    ligne:=readline("Data_3.dat"):
    if (ligne <> 0) then
        L:=[op(L), sscanf(ligne,"x%*d=%d T=%g P=%g L%*d=%g")]:
    end if;
end do:
L;
```

[[0, 0.0030, 23.0, 2.34], [1, 25., 4.50 10⁶, 25.], [3, 2.456, -1., 0.0257], [5, -20000., 35., -2.00],
[7, -2.0 10⁶, -25.0, -45.]]

Les informations données précédemment à propos de la fonction **sscanf** sont incomplètes et on se reportera à l'aide en ligne.

Ecriture dans un fichier

Il existe plusieurs possibilités pour écrire dans un fichier. On se contentera ici de donner le minimum indispensable à connaître avec les fonctions **writedata** et **fprintf**.

1) Le mode d'écriture Maple Notation

Il est possible d'écrire dans un fichier ou à l'écran des valeurs numériques, du texte ou des résultats sous forme d'expressions mathématiques en mode caractères. Dans ce cas les expressions sont converties en chaînes de caractères en mode **MAPLE Notation**. Pour l'obtenir on sélectionnera **MAPLE Notation** dans les préférences de MAPLE, onglet *Display*, item *Output display*. Ce mode, certes moins agréable à lire, est cependant un moyen très efficace d'économiser la mémoire de l'ordinateur si on doit faire des calculs volumineux. Il donne un résultat identique à celui de la fonction **lprint**

```
> p:=expand((x-2)*(x-3)*(x+1)):
lprint(p);
x^3-4*x^2+x+6
```

Attention: ce qui précède n'est donné qu'à titre d'exemple et il n'est pas utile pour la suite d'activer le mode **MAPLE Notation** que **writedata** utilise automatiquement.

2) La fonction writedata

Cette parenthèse étant close, revenons à notre sujet principal. On ne montrera ici que des exemples d'écriture de valeurs numériques dans un fichier avec la fonction **writedata**. La caractéristique principale de **writedata** est de pouvoir écrire simplement les valeurs numériques lorsqu'elles sont rangées dans un tableau, une liste simple ou une liste de listes conforme à un tableau. Le premier argument de la fonction donne la destination d'écriture, généralement le nom d'un fichier, mais ce peut être aussi le mot clé **terminal** afin d'écrire sur l'écran. Le deuxième argument est un tableau ou une liste de listes à écrire et le troisième détermine si les nombres doivent être écrits sous forme décimale, **float**, ou entière, **integer**. Noter que le mode **float** conserve le type entier des nombres. Les décimaux "entiers" comme -1.0 sont aussi traduits en entiers.

```

> restart;
A:=array([[3,-2.5,-1.0],[9.9e-1,-3.9,5.2],[-1,3.0,2.0]]):
writedata(terminal,A,float);

3      -2.5      -1
0.99    -3.9      5.2
-1      3          2

```

Le mode *integer* convertit tous les nombres en entiers

```

> writedata(terminal,A,integer);

3      -2      -1
0      -3      5
-1      3      2

```

Pour écrire ces résultats dans un fichier il suffit donc de remplacer le mot clé *terminal* par le nom du fichier. La structure du fichier sera la même que celle présentée à l'écran

```

> fichier:=cat(kernelopts(homedir),"/Desktop/Data_ch_20/Data_4.dat"):
writedata(fichier,A,float);

```

Il est fréquent que les valeurs numériques à écrire soient produites par une boucle. Il faut alors se rappeler que les résultats seront écrits conformément à la syntaxe de la liste qui contient ces données. Pour l'exemple suivant la liste est conforme à un tableau de trois lignes et une colonne. Chaque appel de **writedata** écrira donc trois nombres, chacun sur une nouvelle ligne. *Attention*: l'exemple ne fonctionne que parce que $\cos(i\pi)$ donne toujours une valeur de type *numeric* (+1 ou -1). Si on écrit $\cos(i\pi/4)$ on aura une erreur et il faudra écrire `evalf(cos(i*pi/4))`.

```

> for i to 3 do
  writedata(terminal,[i,evalf[5](exp(i)),cos(i*pi)],float)
end do:

1
2.7183
-1
2
7.3891
1
3
20.086
-1

```

La liste est maintenant conforme à un tableau à une ligne et trois colonnes, donc chaque ligne écrite contiendra trois nombres séparés par un ou des espaces

```

> for i to 3 do
  writedata(terminal,[[i,evalf(exp(i)),cos(i*pi)]],float)
end do:

1      2.718281828      -1
2      7.389056099      1
3      20.08553692      -1

```

Maintenant la liste est conforme à un tableau à deux lignes dont la deuxième ne contient qu'un nombre, etc.

```

> for i to 3 do
  writedata(terminal,[[i,evalf(exp(i))],[cos(i*pi)]],float)

```

```

end do:
1      2.718281828
-1
2      7.389056099
1
3      20.08553692
-1

```

Attention, Attention

Lorsqu'on écrit dans un fichier avec **writedata**, chaque appel ouvre le fichier et écrit en tête de celui-ci en "écrasant" ce qu'il pouvait déjà contenir. Ainsi la boucle suivante produira un fichier qui ne contiendra qu'une ligne avec les résultats obtenus pour $i = 3$

```

> for i to 3 do
    writedata(fichier, [[i,evalf(exp(i)),cos(i*Pi)]],float)
end do:

```

Pour résoudre ce problème il suffira d'utiliser la fonction **writedata[APPEND]** qui ouvre le fichier et écrit à la suite des informations déjà existantes

```

> for i to 3 do
    writedata[APPEND]
        (Fichier, [[i,evalf(exp(i)),cos(i*Pi)]],float)
end do:

```

Encore que le résultat de cette boucle sera un fichier contenant quatre lignes, la première résultant de l'exécution de la boucle précédente ! Pour obtenir le résultat souhaité on fera un premier appel avec **writedata[NEW]** ou simplement **writedata** de façon à commencer l'écriture en tête du fichier, puis écrire la suite avec **writedata[APPEND]**.

```

> writedata(Fichier, [[1,evalf(exp(1)),cos(Pi)]],float):
  for i from 2 to 3 do
    writedata[APPEND](Fichier, [[i,evalf(exp(i)),cos(i*Pi)]],float)
end do:

```

Pour éviter ces complications on préférera construire la totalité de la liste le listes au préalable et conformément au profil du fichier souhaité (voir plus haut). Mais pour les très gros fichiers, cette méthode présente l'inconvénient d'utiliser beaucoup d'espace mémoire.

```

> T:=[seq([i,evalf(exp(i)),cos(i*Pi)],i=1..3)];
    T := [[1, 2.718281828, -1], [2, 7.389056099, 1], [3, 20.08553692, -1]]

```

ou encore, si on a besoin d'utiliser une boucle

```

> T:=[ ]:
  for i from 1 to 3 do
    T:=[op(T),[i,evalf(exp(i)),cos(i*Pi)]]:
end do:
T;
    [[1, 2.718281828, -1], [2, 7.389056099, 1], [3, 20.08553692, -1]]

```

Il suffira ensuite d'écrire

```
> writedata(fichier,T,float);
```

3) La fonction fprintf

Cette fonction, d'un usage plus général, permet de contrôler avec une syntaxe proche du langage C, l'écriture dans un fichier (voir aussi les fonctions cousines **sprintf**, **nprintf** ou **printf**)

```
> restart:  
fichier:=cat(kernelopts(homedir),"/Desktop/Data_ch_20/Data_5.dat"):
```

Avec **fopen** (file open) on ouvre le fichier et avec l'option **WRITE** on spécifie que c'est une ouverture seulement en écriture. Il existe aussi d'autres fonctions d'information ou de manipulation de fichiers comme **iostatus**, **filepose** ou **fremove**. La fonction **fopen** renvoie un entier identifiant le fichier ouvert.

```
> fopen(fichier,WRITE);  
0
```

La fonction **iostatus()**[4] renvoie les informations sur tous les fichiers ouverts (voir l'aide en ligne; ici il n'y a qu'un fichier associé au numéro 0)

```
> iostatus();  
[0, "/Users/Jean/Desktop/Data_ch_20/Data_5.dat" , STREAM, FP = -1535977264, WRITE, TEXT]
```

On écrit en précisant le format d'écriture **dans une chaîne** avec les gabarits **%d**, **%g**, etc. Conformément à la syntaxe du langage C, le **\n** précise que l'on doit passer à la ligne suivante pour écrire la suite (*n* pour *new line*). **Noter** que l'on peut utiliser dans **fprintf** soit l'identifiant du fichier (ici 0) soit son nom. La gestion des fichiers par identifiants ou par noms est une question de commodité en fonction des circonstances.

```
> for i to 3 do  
    fprintf(0,"%d %g %g \n",i,evalf(exp(i)),cos(i*Pi/4)):  
end do:
```

Puis on ferme le fichier

```
> fclose(0);
```

Le fichier aura la forme suivante

```
1 2.71828 0.707107  
2 7.38906 0  
3 20.0855 -0.707107
```

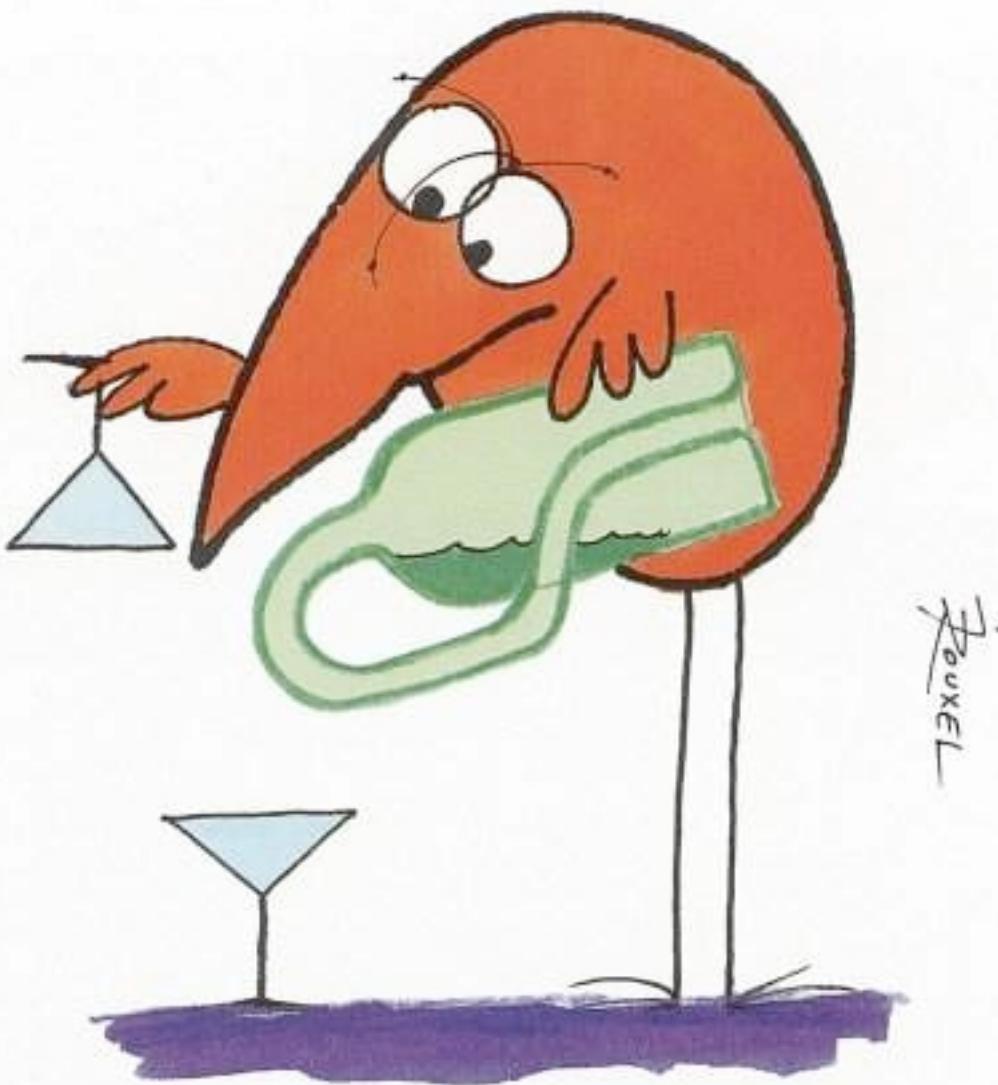
Il existe bien d'autres possibilités que l'on explorera avec l'aide en ligne...

```
> no_fichier:=fopen(fichier,WRITE):  
fprintf(no_fichier,"Résultats \n \n");  
for i to 3 do  
    fprintf(no_fichier,"x%d= %15.10f y%d=%g  
\n",i,evalf(exp(i)),i,cos(i*Pi/4));  
end do:  
fclose(no_fichier):
```

Ce fichier contiendra

Résultats

```
x1= 2.7182818280 y1=0.707107  
x2= 7.3890560990 y2=0  
x3= 20.0855369200 y3=-0.707107
```



S'IL N'Y A PAS DE SOLUTION
C'EST QU'IL N'Y A PAS DE PROBLÈME.

(voir la page de garde et la fin du chapitre 17 à propos de la bouteille de Klein)