AIDE-MEMOIRE pour MAPLE

Expressions

Assignation, évaluation

`nom := expr`
: assigne à *nom* la valeur de *expr*.
L'évaluation des expressions est complète, sauf dans les procédures où l'évaluation se fait à un niveau.

`eval(expr)`
: force l'évaluation complète de *expr*

`'expr'`
: est évalué en *expr*

`nom := 'nom'`
: désassigne *nom*.

`x,y,z := a,b,c`
: exemple d'assignation multiple

`value(expr)`
: où *expr* est l'appel d'une fonction inerte (par exemple `Sum` ou `Int`), provoque l'appel de la fonction active correspondante (ici `sum` ou `int`)

Structure

`nops(expr)`
: nombre d'opérandes de *expr*

`op(expr)`
: séquence des opérandes de *expr*

`op(i,expr)`
: $i^{\text{ème}}$ opérande de *expr*

`has(expr, ssexpr)`
: teste si *ssexpr* est une sous-expression de *expr*

`testeq(expr1, expr2)`
: teste si *expr1* et *expr2* ont la même valeur

Type

`whattype(expr)`
: donne le type de *expr*

`type(expr, type)`
: teste si *expr* a le type *type*. Types courants :
`'+'`, `'*'`, `'^'`

`integer, fraction, rational, float, realcons, complex`
`polynom, series`
`'='`, `'<>'`, `'<'`, `'<='`, `relation, and, or, not, boolean`
`symbol, range, indexed, table, array, exprseq, list, set, string`

`false true not or and`
: valeurs booléennes et opérateurs booléens

Expressions composées

`expr1..expr2`
: intervalle (`range`)

`nom[expr]`
: expression indicée (`indexed`)

`nom[expr1] := expr2`
: définit une table nommée *nom*.
Les tables sont évaluées au dernier nom (utiliser `eval` pour avoir le contenu de la table)

`T := array(a..b, liste)`
: définit un tableau T indexé de a à b, contenant les éléments de *liste*
Cette définition s'étend aux tableaux à plusieurs dimensions. Comme pour les tables, l'évaluation se fait au dernier nom.

`expr1,..., exprn`
: séquence (`exprseq`)

`s[i]`
: $i^{\text{ème}}$ élément de la séquence `s`

`NULL`
: séquence vide

`[sequence]`
: liste (`list`)

`L[i]`
: $i^{\text{ème}}$ élément de la liste `L`

`{sequence}`
: ensemble (`set`). Pas d'ordre sur les éléments.

`member(elt, ens)`
: teste si *elt* est élément de *ens* (ensemble ou liste)

`ens1 intersect ens2`
: intersection de deux ensembles

`ens1 union ens2`
: réunion de deux ensembles

`ens1 minus ens2`
: complémentaire de *ens2* dans *ens1*

`x -> expr_de_x`
: fonction d'une variable

`(x,y) -> expr_de_x_et_y`
: fonction de deux variables etc.

`unapply(expr_de_x, x)`
: donne la fonction : x `->` *expr_de_x*

`proc(paramètres) ... end proc`
procédure (`procedure`)

`eval(procedure)`
donne l'expression-procédure elle-même.

`"abc"`
chaîne de caractères

`cat(s1, s2)`
concaténation des chaînes de caractères *s1* et *s2*

Transformations

N.B.— Les fonctions suivantes donnent une nouvelle expression obtenue par transformation d'une expression de départ qui, elle, n'est pas modifiée.

`subs(sousexpr = autre, expr)`
expression obtenue en remplaçant *sousexpr* par *autre* dans *expr*
on peut mettre un ensemble de plusieurs substitutions

`seq(expr, i = a..b)`
séquence des *expr* pour i allant de a à b

`expr$n`
équivaut à la séquence `expr`, ..., `expr` (n fois)

`map(fonction, expr)`
expression obtenue en appliquant *fonction* à chaque opérande de *expr*. Très utile pour les listes. Ne s'applique pas aux séquences.

`sum(expr, i = a..b)`
calcule la somme des *expr* pour i allant de a à b
Pour éviter les effets d'une assignation préalable de i, il est conseillé d'écrire plutôt : `sum('expr', 'i' = a..b)`

`product(expr, i = a..b)`
même chose avec le produit
`Sum` et `Product` sont les versions inertes correspondantes (i.e ne provoquent pas le calcul).

`convert(expr, type)`
change le type d'une expression.
type est par exemple : `'+'`, `'*'` ...

`convert(expr, modèle)`
transforme une expression selon un modèle.
modèle est par exemple : `exp`, `log`, `trig`, `tan` (conversion en tangente de l'arc moitié) ...

`expand(expr)`
développe une expression.
Utile pour développer les polynômes, exprimer les lignes trigonométriques de nx en fonction de celles de x, transformer e^{x+y} en e^x e^y, $\ln(xy)$ en $\ln(x)+\ln(y)$

`combine(expr, {options})`
regroupe les différents termes d'une expression.
options : `exp`, `log`, `power`, `trig` (pour linéariser les expressions trigonométriques) ...

`factor(expr)`
factorise une expression

`normal(expr)`

met les expressions rationnelles sous forme canonique.
Le numérateur est développé, le dénominateur factorisé.

`simplify(expr, sequence_facultative_de_regles)`
simplifie une expression.
règles : `exp`, `log`, `power`, `radical`, `trig`, `symbolic` ...

`simplify(expr, equations, variables)`
permet d'éliminer les *variables* de *expr* grâce aux *équations*.

Nombres

Entiers et rationnels

`iquo(a, b)`
quotient entier de a par b

`irem(a, b)`
reste de a modulo b

`igcd(sequence_d_entiers)`
p.g.c.d d'une séquence d'entiers

`igcdex(a, b, u, v)`
p.g.c.d de a et b, avec calcul des coefficients de Bezout u et v

`ifactor(n)`
décomposition de n en produit de facteurs premiers.

`isprime(n)`
teste si n est premier.

`n!`
factorielle de n

`binomial(n,p)`
$\binom{n}{p}$

`rand()`
donne un entier aléatoire

`numer(fraction)` `denom(fraction)`
numérateur et dénominateur de *fraction*

Flottants

`evalf(nombre, nb_chiffres)`
approximation décimale de *nombre* avec *nb_chiffres* (facultatif)

`Digits := n`
modifie la valeur de la variable `Digits` qui contrôle le nombre de chiffres utilisés pour les calculs en flottants (par défaut : 10)

`floor(x)`
partie entière de x

`trunc(x)`
partie entière "informatique" de x

`round(x)`
entier le plus proche de x

`signum(x)`
: signe de x (1 pour 0)

Complexes

`evalc(expr)`
: met *expr* sous la forme $a + ib$, en supposant que les noms figurant dans *expr* représentent des réels.

`evalc(Re(z))` `evalc(Im(z))`
: parties réelle et imaginaire de z

`abs(z)`
: module de z

`argument(z)`
: argument de z

`conjugate(z)`
: conjugué de z

`sqrt(z)`
: racine carrée de z

`csgn(z)`
: "signe" de z, c'est-à-dire :
$+1$ si $\Re(z) > 0$ ou $\Re(z) = 0$ et $\Im(z) > 0$
-1 si $\Re(z) < 0$ ou $\Re(z) = 0$ et $\Im(z) < 0$
0 si $z = 0$

Constantes

`Pi` `infinity` `I`
: représentent respectivement π, ∞ et $i = \sqrt{-1}$

Polynômes

Ecriture

`expand(P)`
: développe `P`

`collect(P,X,fonction)`
: regroupe en puissances de l'indéterminée X, en appliquant *fonction* (facultative) à chaque coefficient

`sort(P,X)`
: arrange dans l'ordre des puissances décroissantes de X

`coeff(P,X,k)`
: coefficient en X^k dans $P(X)$ (supposé somme de termes de degrés différents)

`degree(P,X)`
: degré de $P(X)$ en X

Divisibilité et racines

`divide(P,Q)`
: teste si P est divisible par Q

`quo(A,B,X)`
: quotient de la division euclidienne de $A(X)$ par $B(X)$

`rem(A,B,X)`
: reste de la division euclidienne de $A(X)$ par $B(X)$

`gcd(A,B)`
: p.g.c.d de $A(X)$ et $B(X)$

`gcdex(A,B,X,U,V)`
: p.g.c.d de $A(X)$ et $B(X)$ et coefficients de Bezout

`resultant(A,B,T)`
: résultant des polynômes A et B par rapport à T

`discrim(P,X)`
: discriminant de $P(X)$

`irreduc(P)`
: teste l'irréductibilité sur le corps engendré par les coefficients de P

`factor(P)`
: factorisation sur le corps engendré par les coefficients de P

`solve(P)`
: séquence des racines de P dans $\mathbf{C}$

Fractions rationnelles

`normal(F)`
: écrit la fraction rationnelle F à coefficients rationnels sous forme réduite.

`factor(F)`
: factorise le numérateur et le dénominateur

`convert(F, parfrac, X)`
: donne la décomposition en éléments simples de la fraction rationnelle $F(X)$. Si le dénominateur de $F(X)$ est factorisé sur $\mathbf{Q}$ (resp. $\mathbf{R}$, $\mathbf{C}$), on obtient la décomposition sur $\mathbf{Q}$ (resp. $\mathbf{R}$, $\mathbf{C}$).

Nombres algébriques

`RootOf(P,X)`
: représente une racine arbitraire de $P(X)$ (supposé irréductible sur $\mathbf{Q}$)
Cette fonction permet de décrire les corps de nombres algébriques, i.e. les extensions de $\mathbf{Q}$ engendrées par les racines de polynômes irréductibles sur $\mathbf{Q}$.

`allvalues(expr)`
: donne la séquence des valeurs d'une expression contenant un `RootOf`

`evala(expr)`
: évaluation de *expr* dans un corps de nombres algébriques donné par des `RootOf`

`factor(P,ensemble_de_radicaux)`
: factorisation sur un corps de nombres algébriques défini par un ensemble de radicaux (racines $n^{\text{ièmes}}$ d'entiers).

`factor(P,ensemble_de_RootOf)`
: factorisation sur un corps de nombres algébriques défini par un ensemble de `RootOf`

Equations

`expr1 = expr2`
: équation

`lhs(equation)` `rhs(equation)`
: membres gauche et droit d'une équation

`assign(ensemble_d_equations)`
: où chaque équation est de la forme `nom = expr`, a pour effet d'effectuer les assignations correspondantes.

`solve(ensemble_d_equations, ensemble_de_variables)` ou

`solve(liste_d_equations, liste_de_variables)`
: résolution d'un système d'équations par rapport aux variables spécifiées
rend l'ensemble des solutions (chaque solution est une séquence d'équations de la forme *variable* `=` *expr*)

`fsolve(...)`
: même chose mais résolution numérique en nombre flottants

`isolve(équation)`
: résolution d'une équation sur $\mathbf{Z}$

`msolve(ensemble_d_equations, n)`
: résolution modulo n

`chrem(liste_d_entiers, liste_d_entiers)`
: résolution d'un système de congruences multiples (théorème chinois)

`rsolve({equation, conditions_initiales}, fonction)`
: resolution d'une récurrence linéaire.

Algèbre linéaire

`with(LinearAlgebra):`
: chargement du package d'algèbre linéaire

Constructions

`Vector(n, liste)` ou `Vector(n, fonction)`
: construit un vecteur de dimension n dont les composantes sont données par une liste ou une fonction `i -> expr`

`v[i]`
: composante d'indice i du vecteur v

`Dimension(v)`
: dimension du vecteur v

`Matrix(n, p, liste)` ou `Matrix(n, p, fonction)`
: construit une matrice à n lignes et p colonnes dont les termes sont donnés par une liste ou une fonction `(i,j) -> expr`.

`M[i,j]`
: terme d'indices i, j de la matrice M

`RowDimension(M)` `ColumnDimension(M)`
: nombre de lignes, nombre de colonnes de M

`RandomMatrix(n, p)`
: matrice aléatoire à coefficients entiers

`map(fonction, M)`
: applique *fonction* à tous les termes de la matrice M
`map(simplify, M)` permet de simplifier la matrice M

`Copy(M)`
: renvoie une copie de la matrice M.

`IndentityMatrix(n)`
: matrice identité d'ordre n

Opérations

`+`, `-`, `*` (produit *scalaire* $\times$ *matrice*), $\cdot$ (produit matriciel)
: opérations matricielles

`DotProduct(v1, v2)`
: produit scalaire (ou hermitien) de v_1 et v_2

`CrossProduct(v1, v2)`
: produit vectoriel de v_1 et v_2 dans $\mathbf{R}^3$

Fonctions

`Norm(v, p)`
: norme indice p de v (i.e. $(\sum |v_i|^p)^{1/p}$)

`Row(M, k)` `Column(M, k)`
: $k^{\text{ième}}$ vecteur ligne (resp. colonne) de M

`Transpose(M)`
: matrice transposée de M

`Rank(M)`
: rang de M

`NullSpace(M)`
: donne une base du noyau de M

`GaussianElimination(M)`
: triangulation de `M` par la méthode du pivot de Gauss

`Determinant(M)`
: déterminant de M

`MatrixInverse(M)` ou `M^(-1)`
: calcule la matrice inverse de M

`CharacteristicPolynomial(M, X)`
: polynôme caractéristique de M en l'indéterminée X

`Eigenvalues(M)`
: séquence des valeurs propres de M

`Eigenvectors(M)`
: séquence de listes de la forme :
`[`*valeur propre*`,` *multiplicité*`,` *base du ss-espace propre*`]`

`JordanForm(M)` ou `JordanForm(M,output='Q')`
: donne la réduite de jordan J de M
la deuxième forme donne la matrice de passage Q telle que $M = Q\,J\,Q^{-1}$

Systèmes linéaires

`GenerateMatrix(liste_d_expressions, liste_de_variables)`
où chaque *expression* est une combinaison linéaire des *variables*
génère la matrice associée

`LinearSolve(A, b)`
donne le vecteur x solution du système linéaire $A\,x = b$

Analyse

Fonctions élémentaires

`sin(x)`, `cos(x)`, `tan(x)`, `cot(x)`
fonctions circulaires

`arcsin(x)`, `arccos(x)`, `arctan(x)`
fonctions circulaires inverses

`arctan(y,x)`
argument de $x + iy$

`sinh(x)`, `cosh(x)`, `tanh(x)`
fonctions hyperboliques

`arcsinh(x)`, `arccosh(x)`, `artanh(x)`
et leurs inverses

`exp(x)`, `ln(x)`
exponentielle et logarithme néperien

`GAMMA(x)`
fonction $\Gamma(x)$

`Psi(x)`
dérivée logarithmique de $\Gamma(x)$

`Psi(n,x)`
dérivée $n^{\text{ième}}$ de la précédente

`Si(x)`
sinus intégral de `x`

`BesselJ(n,x)`
fonction de Bessel de première espèce

`LegendreF(x,k)`
fonction de Legendre

Limite, continuité, extrema

`limit(expr, x = a, option)`
limite de *expr* quand $x \to a$
option : `left`, `right`, `real`, `complex`
forme inerte : `Limit`

`iscont(expr, x = a..b, option)`
teste la continuité de *expr* sur un intervalle
option : `closed`
à charger au préalable par `readlib(iscont)`

`singular(expr)`
donne les points qui n'ont pas d'image

`minimize(expr, x, x = a..b)  maximize(expr, x, x = a..b)`
calcule le minimum et maximum de *expr* sur un intervalle (optionnel)

Différentiation

`diff(expr, x)`
dérivée de *expr* par rapport à x
forme inerte : `Diff`

`diff(expr, x, y)`
dérivée seconde $\frac{\partial(expr)}{\partial x\,\partial y}$ etc.

`diff(expr, x$n)`
dérivée $n^{\text{ième}}$ de *expr* par rapport à x

`D`
opérateur de différentiation. Par exemple :
`D(f)(x)` dérivée de la fonction f par rapport à x
`(D@@n)(f)(x)` dérivée $n^{\text{ième}}$ de la fonction f par rapport à x

`jacobian(y, x)`
calcule la matrice jacobienne de y (liste ou vecteur d'expressions) par rapport à x (liste ou vecteur de variables)

Développements limités

`taylor(expr, x = a, ordre)`
développement limité de *expr* au voisinage de a, à l'ordre *ordre* (par défaut : 6)
Le reste est en $\mathrm{O}((x-a)^{ordre})$
Il y a une version à plusieurs variables : `mtaylor`

`series(expr, x = a, ordre)`
idem, mais meilleur

`asympt(expr, x, ordre)`
développement asymptotique de *expr* au voisinage de $x = \infty$, à l'ordre *ordre* (par défaut : 6)

`convert(DL, polynom)`
donne la partie polynômiale d'un D.L. en éliminant le reste

`Order := n`
modifie l'ordre des développements ultérieurs (par défaut : 6)

`mtaylor(expr, [x = a, y = b], ordre)`
formule de Taylor à deux variables

Intégration

`int(expr, x)`
intégrale indéfinie, sans constante d'intégration

`int(expr, x = a..b)`
intégrale définie
forme inerte : `Int`

`evalf(Int(...))`
valeur approchée d'une intégrale définie

`intparts(expr, u)`
intégration par parties de *expr*, qui est l'intégrale inerte d'une expression de la forme $u\,dv$, en dérivant u

`changevar(x = f(u), expr, u)`
intégration de *expr*, qui est l'intégrale inerte d'une expression de x, grâce au changement de variable spécifié
`intparts` et `changevar` nécessitent le chargement du package `student` par : `with(student)`

Equations différentielles

N.B.— Dans une équation différentielle en y fonction de x, la fonction inconnue se note `y(x)` *; toute condition initiale doit s'écrire avec l'opérateur de différentiation* `D`.

`dsolve(equation_en_y(x), y(x))`
résolution exacte, sous la forme `y(x) =` *expr*
Maple note `_C1`, `_C2` etc. les différentes constantes d'intégration.
Plus généralement :

`dsolve({equations, conditions_initiales}, {fonctions_inconnues})`
pour une ou plusieurs équations et des conditions initiales (facultatives).
On peut aussi obtenir une résolution approchée :

`dsolve({equation_en_y(x), cond_init}, y(x), series)`
solution sous forme de D.L. au voisinage du point où l'on donne la condition initiale

`dsolve({equation_en_y(x), cond_init}, y(x), numeric)`
solution sous forme d'une procédure qui, lorsqu'on lui passe en paramètre une valeur numérique x_0, rend une liste de la forme : `[x =` x_0`, y(x) =` *val1*`, D(y)(x) =` *val2 etc.*`]` de laquelle on peut extraire les valeurs numériques de $y(x)$ et éventuellement de ses dérivées.

`odeplot(procedure, [x, y(x)], a..b)`
où *procédure* est le résultat de `dsolve` avec l'option `numeric`, permet de tracer une courbe intégrale sur l'intervalle *a..b*
(nécessite `with(plots)`)

`DEplot([equations],[t,x1,x2,...],t=a..b,{[t_0,x1_0,x2_0,...],...},options)`
résolution graphique d'un système différentiel d'ordre 1 :
$x_1' = f(t, x_1, x_2, \ldots)$, $x_2' = g(t, x_1, x_2, \ldots)$ etc.
trace les courbes intégrales définies par l'ensemble des conditions initiales spécifiées
options : `stepsize = h` (par défaut : $\frac{b-a}{20}$)
`scene = [t, x0, x1]` (par exemple) pour une vue en 3D ou :
`scene = [x0, x1]` (par exemple) pour une vue plane,
`arrows = NONE` pour empêcher l'affichage des lignes de champ
(nécessite `with(DEtools)`).

Graphisme

En 2 dimensions

`plot(y, x = a..b, options)` ou `plot(f, a..b, options)`
graphe d'une fonction $y = f(x)$
options : `view=[a..b,c..d]` pour préciser le pavé représenté
`numpoints = n` (par défaut 49)
`style = point` ou `line`
`scaling = CONSTRAINED` pour un repère orthonormé
`axes = NONE` pas d'axe
`discont = true` évite les verticales dues aux discontinuités
Ces options s'appliquent à la plupart des versions suivantes de `plot`
En premier paramètre de `plot`, on peut passer un ensemble de manière à tracer plusieurs courbes sur le même dessin.

`plot([x, y, t = a..b])` ou `plot([f, g, a..b])`
courbe paramétrique $x = f(t)$, $y = g(t)$

`plot([rho, theta, theta = a..b], coords = polar)`
courbe polaire $\rho = f(\theta)$

`implicitplot(equation, x = a..b, y = c..d)`
courbe donnée par une équation cartésienne de la forme $f(x, y) = 0$
option supplémentaire : `grid = [m,n]` (par défaut, [25,25])
nécessite `with(plots)`

`display([g1, g2, ... ])`
affichage simultané de plusieurs courbes, où chaque g_i est un `plot(...)`.

En 3 dimensions

`spacecurve([x, y, z, t = a..b]`
trace la courbe de l'espace paramétrée par $x = f(t)$, $y = g(t)$, $z = h(t)$
nécessite `with(plots)`

`plot3d(z, x = a..b, y = c..d))`
trace la surface $z = f(x, y)$
option : `grid = [m,n]` (par défaut, [25,25])

`contourplot(z, x = a..b, y = c..d))`
trace les courbes de niveau de la surface $z = f(x, y)$
nécessite `with(plots)`

`plot3d([x, y, z], u = a..b, v = c..d)`
trace la surface paramétrée par $x = f(u, v)$, $y = g(u, v)$, $z = h(u, v)$

`plot3d([rho, theta, z], u = a..b, v = c..d, coords = cylindrical)`

`plot3d([rho, theta, phi], u = a..b, v = c..d, coords = spherical)`
variantes en coordonnées cylindriques ou sphériques

`implicitplot3d(equation, x = a..b, y = c..d, z = e..f)`
courbe définie par une équation cartésienne $f(x, y, z) = 0$

`display3d({g1, g2, ...}, style = patch)`
tracé simultané de plusieurs structures `plot3d(...)`

Programmation

`f := proc(`*paramètres formels*`)`
`local` *variables*`;`
`global` *variables*`;`
`options` *options*`;`
instructions
`end proc`

définit une procédure appelée `f`
options : `remember` est la plus importante (la fonction garde en mémoire les valeurs déjà calculées)
instructions : séparées par des points-virgules
Au moment de l'appel `f(`*paramètres*`)`, les paramètres sont évalués et leurs valeurs sont transmises aux paramètres formels ("passage par valeur")
si un paramètre est entouré d'apostrophes, il peut être modifié pendant l'exécution de la procédure ("passage par variable")
la dernière expression évaluée au cours du calcul donne le résultat de la procédure

`RETURN(`*expr*`)`
renvoie la valeur de *expr* comme résultat et provoque l'abandon de la procédure

`ERROR("message")`
provoque l'abandon de la procédure et affiche `message`

`for i from a by pas to b while condition do instructions end do`
boucle générale ; tous les éléments sont facultatifs, sauf `do` et `end do`

`for x in K do ... end do`
variante où K est un ensemble ou une liste

`break`
provoque l'abandon de la boucle en cours

`if condition then instructions else instructions end if`
instruction conditionnelle

`if ... elif ... elif ...else... end if`
instruction conditionnelle multiple (équivaut à des `if` imbriqués)

`print(expr)`
affiche la valeur de *expr*

`op(4, eval(procedure))`
donne la table de remember de la *procédure* (qui contient les valeurs déjà calculées)

Utilités

`?mot`
affiche l'aide en ligne concernant *mot*

`alias(nom = expr)`
déclare `nom` comme synonyme de `expr`.

`assume(x, propriete)`
permet de particulariser la variable x; *propriété* est par exemple :
`integer`, `real`, `RealRange(a,b)`
on peut également écrire une relation, par exemple `assume(x>0)`

`interface(labelling = false)`
désactive l'usage des `%1`, `%2`, .. dans l'affichage des expressions compliquées

`interface(verboseproc = 2)`
permet l'affichage des procédures internes à Maple, grâce à `eval`

`read "nom_fichier"`
charge et exécute les commandes Maple contenues dans le fichier-texte dont le nom est spécifié

`with(package)`
charge en mémoire les procédures incluses dans le package désigné

Quelques packages

`student`
calcul intégral, intégrales multiples, sommes, limites etc.

`group`
groupes de permutations etc.

`numtheory`
théorie des nombres (fonction φ d'Euler, symbole de Jacobi, nombres premiers etc.)

`orthopoly`
polynômes orthogonaux (Hermite, Laguerre, Legendre, Jacobi, Chebyshev etc.)

`LinearAlgebra`
algèbre linéaire (`linalg` est obsolète)

`VectorCalculus`
calcul vectoriel et différentiel

`powseries`
séries formelles

`gfun`
fonctions génératrices

`plots`
fonctions graphiques spécialisées

`geometry`
géométrie plane