Université Claude Bernard LYON 1
Département de Mathématiques
Master Mathématiques Générales - Préparation à l'Agrégation

MEMENTO MAPLE

Michel CRETIN

Michel.Cretin@math.univ-lyon1.fr
http ://ufr-mathematiques.univ-lyon1.fr/Agregation/MG2ACF

Chapitre 1

Introduction à *MAPLE*.

1.1 Généralités

Les logiciels permettant le traitement par ordinateurs de problèmes mathématiques, et plus généralement, de problèmes de modélisation ou de simulation sont de deux sortes :

1. les logiciels de calcul *numérique* tels que *MATLAB* qui reposent sur des *approximations* aussi bien pour la représentation des données que pour les méthodes de résolution. Naturellement la validité de ces représentations et méthodes doit faire l'objet d'une justification particulière : *précision*, *stabilité numérique*.
2. les logiciels de calcul *formel* tels que *MAPLE* qui utilisent des représentations *symboliques* des objets qu'ils manipulent (*expressions*) au moyen d'algorihmes *exacts* qui naturellement doivent être validés : *terminaison*, *correction*.

Evidemment ces deux types d'approche sont complémentaires : par exemple on pourra trouver une solution numérique approchée d'une équation différentielle dont il n'est pas possible d'exprimer symboliquement la solution. D'ailleurs *MAPLE* possède de nombreuses possibilités de calcul numérique et est donc un logiciel assez complet.
Il existe beucoup d'autres logiciels plus spécialisés, souvent disponibles librement sur internet tels que *GAP* et *Singular* pour l'algèbre.
Signalons enfin le sytème libre récent *SAGE* qui organise sous une interface graphique commune un ensemble de logiciels libres.

L'utilisation efficace d'un logiciel de Calcul Formel dépend de manière essentielle des connaissances algorithmiques, et plus généralement mathématiques de l'utilisateur, et exige le même niveau de rigueur que tout autre travail mathématique.

1.2 Les expressions et les séquences d'expressions

Les expressions sont les objets manipulés par *MAPLE*. *La formation d'une expression suivie de son évaluation et de l'affectation du résultat (qui est encore une expression) à une variable constitue l'opération de base en MAPLE.* Aussi l'analyse de la structure des expressions (aussi bien pour les former que pour en extraire les composants), l'étude de leurs règles de transformation et d'évaluation, les principes de leur classification (`types`) sont-ils nécessaires pour une utilisation efficace du logiciel.

1.2.1 Expressions

Une expression est constituée d'un *opérateur appliqué à une séquence d'opérandes*. Les expressions peuvent contenir des *variables libres*.

`nops`	nombre d'opérandes
`op`	séquence des opérandes
`whattype`	opérateur
`indets(`$\cdots$`,name)`	variables *libres* de l'expression
`has`	teste si une expression contient une sous-expression

Dans le cas des relations [`type(`$\cdots$`,relation)`] on a aussi les fonctions :

`lhs`	membre de gauche
`rhs`	membre de droite

Les opérations suivantes permettent de transformer ou de simplifier une expression e et comportent de nombreux paramères optionnels :

`expand`(e)	développe e
`normal`$(e,$ `expanded`$)$	normalise une expression rationnelle
`convert`$(e, <$ *modèle* $>)$	transforme e selon un modèle
`combine`$(e, <$ *options* $>)$	regoupe certaines sous-expressions
`simplify`$(e, <$ *règles* $>)$	simplifie e selon des règles
`collect`$(e,$ *vars*, $<$*form*$>$, $<$*func*$>)$	regroupe des termes dans une expression

En particulier, la nécessité de *simplifier* convenablement une expression apparaît dans le *probléme de décision de l'égalité*, un test sommaire tel que `evalb`$(e1 = e2)$ ne décide que de l'*identité* des deux expressions $e1$ et $e2$. On distinguera donc l'*identité* de deux expressions (*égalité syntaxique*) de l'*égalité* entre les objets mathématiques qu'elles représentent (*égalité sémantique*). Ces deux notions coïncident lorsque les expressions sont *sous forme normale*.
Les types de données permettent de former des *classes d'expressions* selon leur forme, mais aussi selon la nature des objets mathématiques qu'elles représentent.
Une *classe d'expressions* (un `type`) représentant une *catégorie* d'objets mathématiques admet une *forme normale* s'il existe une opération de réduction $e \longrightarrow \overline{e}$ telle que l'égalité mathématique de deux objets $e1$, $e2$ de cette classe soit équivalente à l'identité de leurs formes normales $\overline{e1}$, $\overline{e2}$. Une classe d'expressions ne possède pas nécessairement de forme normale.

1.2.2 Séquences d'expressions

Ce sont des suites finies d'expressions, admettant des répétitions mais pas d'imbrications. On les construit par énumération des éléments séparés par des virgules, au moyen de l'opérateur de répétition `$` ou à l'aide du constructeur `seq` (*i.e.* la suite $(x_i)_{1 \leq i \leq n}$ où n est une *valeur* entière sera construite par l'une des formes `seq`$(x[i], i = 1..n)$ ou $x[i]$ `$` $i = 1..n$. La séquence vide est `NULL`.
On accède au $i^{\text{ème}}$ élément ($i \geq 1$) d'une séquence d'expressions S *via* l'*opérateur de sélection* $S[i]$; le *sélecteur* peut être *négatif* : *e.g.* $S[-1]$ est le dernier élément de la séquence ou un *intervalle* : $S[i..j]$ est la séquence extraite de S du $i^{\text{ème}}$ au $j^{\text{ème}}$ élément.
Le nombre d'éléments de S est donné par `nops`$([S])$.

1.2.3 Sommes et produits

La somme (*resp.* le produit) d'une suite de valeurs s'effectue au moyen des fonctions `add` (*resp.* `mul`) équivalentes à l'écriture de boucles *for*. Dans le cas de sommes (*resp.* de produits) explicites l'emploi de ces fonctions est préférable à celui de `sum` et `product`.
Pour les sommes et les produits symboliques, *i.e.* lorsque le calcul nécessite des transformations

(fonctions symétriques, fonctions hypergéométriques, *etc.*), que les bornes sont variables ou infinies (*ie.* séries ou produits infinis) on utilisera les fonctions `sum()` (resp. `product`).

Notons que ces dernières possèdent les formes inertes `Sum` (resp. `Product`). Une forme inerte permet de construire une expression tout en différant son calcul que l'on déclanche par l'opération `value` .

1.3 Les variables et l'affectation

1.3.1 Variables

Les variables servent à nommer les expressions. Une variable est *caractérisée* par :

1. son *nom*, de l'une des formes `symbol` ou `indexed` (regroupées dans le type `name`). La fonction `evaln` donne le nom de la variable.
   Un nom de type `symbol` est formé d'une suite de lettres majuscules ou minuscules, de chiffres, ou des signes _ et ? et commence par une lettre. On peut aussi utiliser n'importe quel signe à condition de le délimiter par des accents graves ‘···‘. L'opérateur de concaténation `||` est utile pour former des noms.
   Un nom de type `indexed` est un nom suivi d'une séquence d'indices délimitée par des crochets (comme $xyz[i,j]$) ; on a une variable différente pour chacune des valeurs des indices (aucune séquence, liste ou table n'est créée).
2. son *état* (libre ou affecté) renvoyé par la fonction `assigned`. Les variables affectées sont des noms pour les expressions tandis que les variables libres sont des *indéterminées*. La fonction `unassign` permet de libérer une ou plusieurs variables.
3. sa *valeur* (qui est évidemment une expression). La valeur d'une variable libre est son nom. Le `type` d'une variable est *dynamique* : c'est automatiquement celui de sa valeur ; il ne nécessite pas de déclaration préalable et il peut évoluer en cours d'exécution.

1.3.2 L'affectation

Les *variables* permettent de *nommer* les expressions au moyen de l'opération d'*affectation* :

$$Nom_1, \cdots, Nom_n := expr_1, \cdots, expr_n$$

ou bien

$$\texttt{assign}(Nom_1 = expr_1, \cdots, Nom_n = expr_n)$$

Cependant si e est une expression contenant des variables libres $x_1, \cdots, x_n$, on peut former une nouvelle expression en effectuant les substitutions $x_1 = v_1, \cdots, x_n = v_n$ dans e par l'opération :

$$\texttt{eval}(e, \{x_1 = v_1, \cdots, x_n = v_n\})$$

plutôt que d'utiliser les affectations $x_1 := v_1, \cdots, x_n := v_n$. Cela évite de modifier l'expression initiale e (on peut aussi utiliser `subs`$(\{x_1 = v_1, \cdots, x_n = v_n\}, e)$ mais attention à la présence ou non des accolades $\{\cdots\}$).
Souvent l'évaluation de cette nouvelle expression devra être complétée au moyen de l'un des *évaluateurs spécifiques* :

`evala`	nombres et fonctions algébriques
`evalb`	expressions booléennes
`evalc`	expressions complexes
`evalf`	nombres flottants
`evaln`	nom d'une variable
`evalp`	nombres p-adiques

Noter aussi la *forme inerte* :

$$\texttt{Eval}(e, x_1 = v1, \cdots, x_n = v_n)$$

à utiliser dans le cas d'*expressions algébriques* à évaluer `mod` p.
Rappelons que *la formation d'une expression, son évaluation et l'affectation du résultat à une variable* constituent l'opération de base en *MAPLE*.

1.3.3 Les opérateurs fonctionnels

Considérons de nouveau une expression e contenant des variables libres $x_1, \cdots, x_n$; on peut associer à e un opérateur fonctionnel $\widetilde{e}$ par l'instruction :

$$\widetilde{e} := (x_1, \cdots, x_n) -> e$$

On peut aussi utiliser la forme :

$$\widetilde{e} := \texttt{unapply}(e, x_1, \cdots, x_n)$$

Les opérateurs fonctionnels sont une forme simplifiée de *procédure* de sorte que l'évaluation de e en $x_1 = v_1, \cdots, x_n = v_n$ s'obtiendra par $\widetilde{e}(v_1, \cdots, v_n)$

1.4 Listes et Ensembles

1.4.1 Listes : `list`

Une liste est une séquence d'expressions comprise entre les crochets $[\cdots]$; [] est la liste vide. Dans une liste, les répétitions sont admises, l'ordre est préservé et plusieurs niveaux d'imbrication sont possibles.
En particulier le type `listlist` représente des listes dont tous les éléments sont des *listes* ayant le même nombre d'éléments.
Plus généralement si t est un type quelconque, `list`(t) (resp. `listlist`(t)) représente les listes dont les éléments sont de type t (resp. dont les éléments sont des *listes* d'éléments de type t).

On *accède* au $i^{\text{ème}}$ élément d'une liste L *via* l'*opérateur de sélection* $L[i]$; le *sélecteur* peut être négatif : *e.g.* $L[-1]$ est le dernier élément de la liste ou un *intervalle* : $L[i..j]$ est la liste extraite de L du $i^{\text{ème}}$ au $j^{\text{ème}}$ élément.
Pour *supprimer* le $i^{\text{ème}}$ (i pouvant être négatif) élément de L on écrira `subsop`$(i = \texttt{NULL}, L)$
Voici quelques opérations de base :

`in`	appartenance
`nops`	nombre d'éléments
`member`	test l'appartenance
`map`	applique une fonction à chaque élément d'une liste
`select`	sélectionne les éléments qui vérifient une propriété
`remove`	supprime les éléments qu vérifient une propriété
`selectremove`	partitionne une liste selon une propriété
`ListTools[Reverse]`	inverse l'ordre des éléments
`ListTools[MakeUnique]`	supprime les répétitions
`ListTools[Flatten]`	applatit une liste de listes
`ListTools[Enumerate]`	numérote les éléments
`ListTools[Categorize]`	construit les classes d'équivalence
`combinat[carprod]`	énumère les éléments d'un produit cartésien
`combinat[choose]`	forme les listes extraites
`combinat[permute]`	liste des permutations d'une liste
`combinat[randperm]`	une permutation aléatoire d'une liste

1.4.2 Ensembles : `set`

Un ensemble est une séquence d'expressions comprise entre les accolades $\{\cdots\}$; $\{\}$ est l'ensemble vide.
Dans un ensemble, il n'y a pas de répétition, ni d'ordre pour les éléments (*i.e.* l'ordre d'écriture des éléments peut varier à chaque exécution) et plusieurs niveaux d'imbrication sont possibles.
Voici quelques opérations sur les ensembles :

`in`	appartenance
`nops`	nombre d'éléments
`member`	test l'appartenance
`map`	applique une fonction à chaque élément d'un ensemblr
`select`	sélectionne les éléments qui vérifient une propriété
`remove`	supprime les éléments qu vérifient une propriété
`selectremove`	partitionne un ensemble selon une propriété
`union`	réunion
`intersect`	intersection
`minus`	différence
`subset`	test l'inclusion
`combinat[carprod]`	énumère les éléments d'un produit cartésien
`combinat[choose]`	forme les combinaisons
`combinat[subsets]`	énumère les parties d'un ensemble

1.5 Les nombres

1.5.1 Les entiers : `integer`

Ils représentent les éléments de $\mathbb{Z}$. Les opérations arithmétiques sont automatiquement effectuées (*simplification automatique*).
On dispose de sous-types tels que :

`posint`	$\mathbb{N}^\star$
`nonnegint`	$\mathbb{N}$
`even`	entiers pairs
`odd`	entiers impairs
`prime`	nombres premiers

Quelques fonctions disponibles :

`convert/base`	conversion d'une base dans une autre
`abs`	valeur absolue
`sign`	signe
`factorial` ou `!`	factorielle
`irem` ou `mod`	reste division euclidienne
`iquo`	quotient division euclidienne
`igcd` , `igcdex`	pgcd, pgcd et coefficients de Bezout
`ilcm`	ppcm
`min` , `max`	minimum, maximum
`ifactor` , `ifactors`	factorisation en nombres premiers
`numtheory[factorset]`	ensemble des facteurs premiers
`prevprim`	nombre premier immédiatement inférieur
`nextprim`	nombre premier immédiatement supérieur
`ithprime`	$i^{\text{ème}}$ nombre premier
`isprime`	test de primalité
`numtheory[divisors]`	ensemble des diviseurs
`numtheory[phi]`	fonction d'Euler
`numtheory[mobius]`	fonction de Möbius

1.5.2 Les nombres rationnels : `rational`

Ils représentent les éléments de $\mathbb{Q}$. Les fractions sont réduites, le dénominateur est strictement positif et les opérations arithmétiques sont effectuées (*simplification automatique*) .
Les fonctions `numer` et `denom` donnent numérateur et dénominateur.
A noter un type `extended_rational` admettant en plus des nombres rationnels les valeurs `infinity`, `-infinity` et `undefined`.

1.5.3 Les nombres flottants : `float`

Le nombre `Float`(M, n) est défini par deux entiers la *mantisse* M (d'un nombre de chiffres fixé par la *variable d'environnement* `Digits` qui vaut 10 par défaut) et l'*exposant* n (voir `Maple_floats` pour connaitre ses limites).
On peut représenter un nombre flottant en *notation décimale* suite de chiffres séparés par un point "." on en *notation scientifique* M`e`n ou M`E`n.
Dès qu'un expression contient un nombre flottant, l'évaluation de celle-ci se poursuit dans ce contexte. La fonction `evalf` force l'évaluation d'une expression en nombres flottants.
Les fonctions `SFloatMantissa` et `SFloatExponent` donnent la mantisse et l'exposant d'un nombre flottant.
Enfin le type `numeric` (`extended_numeric`) regroupe le type `rational` (`extended_rational`) avec le type `float`.

1.5.4 Les nombres réels

Hors mis quelques constantes prédéfinies telle que le nombre π (`Pi`), la constante d'*Euler* $\lim_{n\to\infty}(\sum_{k=1}^{n}\frac{1}{k}-\ln(n))$ (`gamma`) et la constante de *Catalan* $\sum_{n=0}^{\infty}\frac{(-1)^n}{(2n+1)^2}$ les nombres réels sont obtenus par application de fonction réelles à des nombres réels (*e.g.* $\sin(\sqrt{2}),\cdots$).
De telles expressions pourraient être testées par le *type* :

```
And(realcons,Not(float),Not(infinity))
```

où `realcons` est un type permettant de tester si une expression est une *constante réelle i.e.* `infinity`, `-infinity` ou a une valeur retournée par `evalf` qui est du type `float`.

La plupart des fonctions *élémentaires* prédéfinies en *MAPLE* sont valides dans le *domaine complexe* ; cependant le *package* `RealDomain` permet d'en restreindre la portée au *domaine réel*. Voici la liste des fonctions concernées :

Re	Im		arccos
arccosh	arccot	arccoth	arccsc
arccsch	arcsec	arcsech	arcsin
arcsinh	arctan	arctanh	cos
cosh	cot	coth	csc
eval	csch	exp	expand
limit	ln	log	sec
sech	signum	simplify	sin
sinh	solve	sqrt	surd
tan	tanh		

On a aussi les fonctions entières suivantes :

`trunc`	entier le plus proche vers 0
`round`	entier le plus proche
`frac`	partie fractionnaire
`floor`	plus grand entier inférieur ou égal
`ceil`	plus petit entier supérieur ou égal

1.5.5 Les nombres complexes : `complex`

Les nombres complexes comprenant aussi bien les nombres exacts que flottants. Le nombre imaginaire de carré -1 est noté I. On dispose, en plus des fonctions élémentaires, des fonctions suivantes :

`abs`	module
`argument`	argument
`polar`	forme trigonométrique
`Re`	partie réelle
`Im`	partie imaginaire
`evalc`	forme cartésienne
`conjugate`	conjugué

Les nombres complexes *flottants* (*i.e.* de la forme $a+I*b$ où a et b sont de type `float`) forment le type `complex(float)`. La fonction `evalf` force l'évaluation d'une expression en nombres flottants.

Les éléments de l'ensemble $\mathbb{C}$ pourraient être caractérisés par le *type* :

`And(complexcons,Not(complex(float)),Not(infinity))`

où `complexcons` est un type permettant de tester si une expression est une *constante complexe i.e.* `infinity`, `-infinity` ou a une valeur retournée par `evalf` qui est du type `complex(float)`.

Le type `complex` teste si une expression est de la forme $a + b * I$ avec a et b finis et du type `realcons`. De manière générale, si t est un type quelconque, `complex`(t) est le type formé par les éléments de la forme $a + I * b$ où a et b sont du type t ; par exemple les éléments de $\mathbb{Z}[i]$ (*entiers de Gauss*) (*resp.* $\mathbb{Q}[i]$) forment le type `complex(integer)` (*resp.* `complex(rational)`).
Lorsque l'*évaluateur* `evalc` est appliqué à une expression algébrique (type `algebraic`) (de manière à la ramener à la forme `complex(algebraic)`) contenant des *variables libres* ces dernières sont *supposées être réelles*.

Les *nombres algébriques* sous représentés par le type `radalgnum` (le type `algnum` représentant les nombres algébriques sous la forme `RootOf`, le type `radnum` ceux d'entre eux exprimés sous forme de radicaux).
Quelques opérations sur les radicaux :

`rationalize`	rend rationnelle une expression
`radsimp`	simplifie les radicaux
`radnormal`	simplifie les radicaux

1.5.6 Les nombres algébriques : `algnum`

Ils représentent les éléments de $\overline{\mathbb{Q}}$, ce sont les nombres rationnels (type `rational`) ou les racines des polynômes $P \in \mathbb{Z}[X]$ *primitifs* et *irréductibles* représentées par la *forme* `RootOf` (types `RootOf`, `algext`) :

$$alpha := \texttt{RootOf}(P, X)$$

(ou, en *mode inter-actif*, en utilisant l'*abréviation* `alias`$(alpha = \texttt{RootOf}(P, X))$) ainsi que les sommes, produits ou quotient de telles expressions.
Ces expressions seront évaluées *via* l'évaluateur `evala`.
Les nombres algébriques exprimés par radicaux (type `radnum`) doivent être convertis par l'opération `convert(`$\cdots$`,RootOf)`.

La forme `RootOf`(P, X) représente une *racine générique* de P *i.e.* l'extension $\mathbb{Q} \hookrightarrow \mathbb{Q}[X]/(P)$ dans laquelle la classe $\overline{X}$ est une racine de P. Les racines de P peuvent être *individualisées* au moyen d'un *sélecteur* optionnel de la forme RootOf :

- ✓ `RootOf`(P, X, c) où c est une *valeur approchée* de la racine à particulariser.
- ✓ `RootOf`$(P, X, a..b)$ où a et b sont les sommets inférieur-gauche et supérieur-droit d'un rectangle du plan complexe isolant la racine.
- ✓ `RootOf`$(P, X,$ `index` $= i)$ où i est l'*indice* de la racine dans l'énumération obtenue par arguments dans le sens trigonométriques et par modules croissants.

Quelques fonctions (*formes inertes* auxquelles il faut appliquer l'évaluateur `evala`) :

`Algfield`	représentation irréductible d'une extension
`Indep`	indépendance des racines
`Norm`	norme d'un nombre algébrique
`Primfield`	élément primitif d'une extension
`Subfields`	sous-corps d'une extension
`Trace`	trace d'un nombre algébrique

1.5.7 Les classes modulaires

Les éléments de $\mathbb{Z}/n\mathbb{Z}$ sont représentés par les entiers $\{0, \cdots, n-1\}$ ou $\{-[\frac{n-1}{2}], \cdots, [\frac{n}{2}]\}$ selon que la *variable d'environnement* `'mod'` a les valeurs `modp` (*défaut*) ou `mods`.
L'évaluation *modulo* n d'une expression s'effectue par l'opérateur `mod` : $\cdots$ `mod` n.
Noter la *forme inerte* `Power` pour le calcul des puissances *modulo* n.

1.5.8 Les corps finis

Soit p premier ; les éléments du corps fini $\mathbb{F}_p$ sont évidemment les classes modulaires modulo p. Par contre, pour $n \geq 2$, les éléments du corps fini $\mathbb{F}_{p^n}$ sont représentés comme polynômes en la forme `RootOf`(P, X) où $P \in \mathbb{F}_p[X]$ est un polynôme unitaire irréductible de degré n. On obtiendra de tels polynômes au moyen des fonctions :

`Prevpoly`	polynôme précédent
`Nextpoly`	polynôme suivant
`Prevprime`	polynôme irréductible précédent
`Nextprime`	polynôme irréductible suivant

Les expressions sont évaluées par l'opérateur `mod` éventuellement en liaison avec l'évaluateur (sous forme inerte) `Eval` (particulièrement si l'expression contient des variables libres auxquelles on veut substituer des valeurs qui contiennent la forme `RootOf`) :

$\cdots$ `mod` p^n ou bien `Eval`$(\cdots)$ `mod` p^n

1.5.9 Les nombres p-adiques

Le *package* `padic` permet d'utiliser les nombres p-adic. Ceux-ci sont représentés sous la forme d'un *développement limité*

$$x = a_k p^k + a_{k+1} p^{k+1} + \cdots + a_{k+h-1} p^{k+h-1} + 0(p^{k+h})$$

avec $0 \leq a_j \leq p-1$ pour tout j, $a_k \neq 0$, $k \in \mathbb{Z}$ étant l'ordre de x et $|x|_p = p^{-k}$ sa valeur absolue.
Ce développement généralise l'*écriture* d'un entier *en base* p (`convert`$(\cdots, \mathtt{base}, p)$).
La variable globale `Digitsp` fixant le nombre de chiffres significatifs du développement.
On dispose des fonctions suivantes :

`evalp`	développement p-adique
`ordp`	ordre
`valuep`	valeur absolue
`ratvaluep`	somme de la partie régulière
`rootp`	racines p-adiques

Noter que les *fonctions élémentaires* p-adiques sont disponibles par la *composition* `evalp`@f ou en *notation abrégée* fp et que les *séries* peuvent se calculer en appliquant l'*évaluateur* p-adique `evalp` à la *forme in inerte* `Sum`.

1.6 Groupes de permutation

Le *package* `group` introduit quelques opérations sur les groupes de permutation. Une permutation peut être définie sous forme de la liste de ses valeurs $[\sigma(1), \cdots, \sigma(n)]$ ou sous la forme de sa décomposition en cycles (`type/disjcyc`). On passe à l'une ou l'autre de ces formes au moyen

des fonctions `convert/permlist` ou `convert/disjcyc`.
Les opérations sur les permutations supposent que celles-ci sous la forme de décomposition en cycles.

`invperm`	inverse une permutation
`mulperms`	compose σ_1 et σ_2 *i.e.* calcule $\sigma_2 \circ \sigma_1$

On définit un groupe de permutations *i.e.* un sous groupe de $\mathfrak{S}_n$ par :

pg :=`permgroup`(n,*ens_gen*)

où *ens_gen* est l'ensemble des générateurs. On a alors les opérations suivantes :

`grouporder`	ordre d'un groupe
`elements`	liste des éléments
`groupmember`	test d'appartenance
`derived`	groupe dérivé
`issubgroup`	test d'inclusion
`normalizer`	normalisateur d'un sous-groupe dans un groupe
`Sylow`	sous-groupe de Sylow
`areconjugate`	test si deux sous-groupes sont conjugués dans un groupe

1.7 Les polynômes et les fractions rationnelles

Soient $X_1, \cdots, X_n$ des *variables libres* et t un type représentant les éléments d'un anneau *calculable* (*e.g.* les divers types de nombres introduits précédemment mais aussi les anneaux de polynômes ou les corps de fractions rationnelles dont les indéterminées distinctes de $X_1, \cdots, X_n$ joueront le rôle de paramètres) ;

- ✓ Un monôme (type `monomial`$(t, \{X_1, \cdots, X_n\})$) est une expression de la forme $cX_1^{\alpha_1} \cdots X_n^{\alpha_n}$ où $(\alpha_1, \cdots, \alpha_n) \in \mathbb{N}^n$ et c une expression de type t ne contenant pas l'une des variables $X_1, \cdots, X_n$.

- ✓ Un polynôme (type `polynom`$(t, \{X_1, \cdots, X_n\})$) est une expression qui est une somme de monômes de type `monomial`$(t, X_1, \cdots, X_n)$.

- ✓ Une fraction rationnelle (type `ratpoly`$(t, X_1, \cdots, X_n)$) est une expression qui est un quotient de polynômes de type `polynom`$(t, \{X_1, \cdots, X_n\})$.

Quelques fonctions concernant les polynômes (un *évaluateur* devra être appliqué aux f*ormes inertes*) :

`eval, Eval`	valeur en un point
`expand, Expand`	développer
`collect`	regrouper
`coeffs`	les coefficients
`degree`	degré
`ldegree`	plus bas degré
`factor, Factor`	factoriser
`irreduc, Irreduc`	test d'irréductibilité
`AIrreduc, AFactor`	irréductibilité, factorisation sur $\mathbb{C}$
`divide, Divide`	divisibilité
`lcoeff`	coefficient dominant
`tcoeff`	coefficient de plus bas degré
`content, Content`	contenu
`primpart, Primpart`	partie primitive
`randpoly,Randpoly`	polynômes aléatoire
`eliminate`	élimine des variables

Quelques fonctions concernant les fractions rationnelles :

`numer`	numérateur
`demon`	dénominateur
`normal, Normal`	simplification

Quelques fonctions spécifiques au cas univarié :

`sort`	ordonner
`coeff`	un coefficient
`quo, rem`	quotient et reste euclidiens
`gcdex`	pgcd et coefficients de Bezout
`resultant, Resultant`	résultant
`convert/polynom`	partie régulière d'une développement limité
`series`	développement limité
`convert/parfrac`	décomposition partielle en éléments simples
`convert/fullparfrac`	décomposition complète en éléments simples

De nombreux polynômes tels que ceux de Bernoulli (`bernoulli`), Berstein, (`bernstein`), les polynômes cyclotomiques (`numtheory[cyclotomic]`), la plupart des polynômes orthogonaux (*package* `orthopoly`) sont disponibles.

Enfin, le package `Groebner` contient des fonctions permettant d'utiliser les bases de Gröbner :

`Basis`	calcul d'une base de Gröbner
`LeadingMonomial`	monôme dominant
`LeadingTerm`	terme dominant
`LeadingCoefficient`	coefficient dominant
`NormalForm`	calcul de la forme normale
`IsProper`	détermine si un système est compatible
`IsZeroDimentional`	détermine si un système a un nombre fini de solutions
`SPolynomial`	calcul du S-polynôme

1.8 Calculus

1.8.1 Limites et continuité

`limit`	détermine un limite
`iscont`	teste la continuité
`discont`	détermine les points de discontinuité

Le calcul de limites n'est souvent possible qu'en introduisant des hypothèses sur les variables au moyen de la directive `assume` (*e.g.* `assume`(x>0)).

La fonction `limit` possède la *forme inerte* `Limit`.

1.8.2 Séries et produits infinis

les fonctions `sum()` (resp. `product`) permettent de calculer les sommes de séries et les porduits infinis et possèdent les formes inertes les fonctions `Sum` (resp. `Product`).

1.8.3 Dérivées et développements limités

Soit e une expression algébrique contenant la *variable libre* x ; la dérivée de e par rapport à x est obtenue par `diff`(e,x) et les dérivées d'ordre supérieur par `diff`(e,x$n).
Un développement limité de e, en $x = a$, à l'ordre o s'obtient par l'opération `series`(e,x=a,o+1) ;
la partie régulière de ce développement s'obtenant par `convert`(···,`polynom`).
Les développements asymptotiques sont obtenus par `asympt`$(e, x, o+1)$.

Plus généralement, si `e` contient les *variables libres* $x_1, \cdots, x_n$, la dérivée partielle $\dfrac{\partial^{i_1+\cdots+i_n}\mathtt{e}}{\partial x_1^{i_1}\cdots\partial x_n^{i_n}}$
est obtenue par `diff`$(e, [x_1\$i_1, \cdots, x_n\$i_n])$.
La fonction `diff` possède la *forme inerte* `Diff`.
Les développements de Taylor *multivariés* s'obtiennent par `mtaylor`.

1.8.4 Formes différentielles

Le *package* `difforms` permet l'utilisation des *formes différentielles*. Il est nécessaire de déclarer au préalable constantes symboliques, variables indépendantes et formes par :

`defform`$(x = \mathtt{scalar}, y = \mathtt{scalar}, \cdots, \lambda = \mathtt{const}, \phi =< deg >, \cdots);$

On dispose alors des opérations suivantes :

`wdegree`	degré
`d`	différentielle extérieure
	produit extérieur

1.8.5 Intégrales et Primitives

Soit e une expression algébrique contenant la *variable libre* x ; une primitive de e par rapport à x est obtenue par `int`(e,x) et l'intégrale définie $\int_a^b e\,dx$ par `int`(e,x=a..b).
La fonction `int` possède la *forme inerte* `Int`.
Les fonctions suivantes figurent dans le *package* `student` :

`changevar`	changement de variables
`intpars`	intégration par parties
`Doubleint,Tripleint`	formes inertes pour intégrales doubles, triples
`Lineint`	forme inerte pour intégrale curviligne

Parmi les fonctions du *package* `inttrans` on trouve :

`fourier`	transformation de Fourier
`invfourier`	transformation de Fourier inverse
`laplace`	transformation de Leplace
`invlaplace`	transformation de Laplace inverse

1.8.6 Equations

Les équations et les inéquations sont résolues par la fonction `solve`. Pour une *équation algébrique*, (en utilisant éventuellement la fonction `allvalues`) les solutions sont retournées sous la forme `RootOf`, ou dans certains cas *sous forme de radicaux*. La fonction `eliminate` permet d'éliminer une variable entre les équations d'un système.
Pour les *fonctions trancendantes*, l'obtention de l'ensemble des solutions se fait en mettant la *variable d'environnement* `_EnvAllSolutions` à `true`.
Pour les cas spécifiques des fonctions adaptées sont disponibles.

`fsolve`	résolution en nb flottants
`isolve`	résolution en nb entiers
`msolve`	résolution de congruences
`LinearAlgebra[LinearSolve]`	résolution des systèmes linéaires

Les relations de récurrence peuvent être résolues par l'opération `rsolve`.
Les options `makeproc` ou `genfunc` permettent d'exprimer les solutions sous forme d'une procédure ou d'une fonction génératrice.

1.8.7 Equations différentielles

Les équations *différentielles* sont résolues par la fonction `dsolve` ; on peut rechercher les solutions sous forme *symbolique*, sous forme *numérique* (option `type=numeric`) (avec le choix possible pour la méthode d'approximation), sous forme de *séries* (`type=series`), en utilisant des *séries de Fourier* (`method=fourier`) ou la *transformation de Laplace* (`method=laplace`).
Le *package* `DEtools` contient des opérations sur les équations différentielles en particulier la fonction `DEplot` permettant de représenter le portrait de phases d'une équation.
Les équations *aux dérivées partielles* sont résolues par la fonction `pdsolve`.

1.8.8 Analyse vectorielle

Le *package* `VectorCalculus` contient des fonctions permettant l'étude des courbes et surfaces de $\mathbb{R}^2$ et $\mathbb{R}^3$ ainsi que l'analyse vectorielle dans de nombreux systèmes de coordonnées. Il vient compléter le *package* `LinearAlgebra`. Parmi celles-ci :

`SetCoordinates`	fixe un système de coordonnées
`VectorField`	définit un champ de vecteurs
`Gradient`	gradient
`Divergence`	divergence
`Rotational`	rotationnel
`int`	intégration des fonctions de plusieurs variables
`PathInt`	intégrales curvilignes
`SurfInt`	intégrales de surface
`Flux`	flux d'un champ de vecteurs

1.9 Les tableaux, les vecteurs et les matrices

Soit t un type représentant les éléments d'un anneau *calculable* R ou l'ensemble des *nombres flottants* réels ou complexes ;

1.9.1 Les tableaux

Les tableaux à composantes dans t (type `'Array'`(t), *ne pas omettre les accents aigus*) représentent les suites *multi-indexés* d'éléments de t. La suite $A = (x_{i_1,\cdots,i_n})_{a_1\leq i_1\leq b_n,\cdots,a_n\leq i_n\leq b_n}$ est obtenue par l'opération :

$$A := \texttt{Array}(a_1..b_1,\cdots,a_n..b_n,(i_1,\cdots,i_n)->x_{i_1,\cdots,i_n})$$

On accède aux composantes *via* l'opérateur de *sélection* $A[i_1,\cdots,i_n]$ et on dispose des opérations :

`ArrayNumDims`	nombre de dimensions
`ArrayDims`	intervalles des indices
`ArrayNumElems`	nombre d'éléments
`ArrayElems`	ensembles des égalités $(i_1,\cdots,i_n) = x_{i_1,\cdots,i_n}$

1.9.2 Les vecteurs

Les vecteurs à composantes dans t (type `'Vector'`(t), *ne pas omettre les accents aigus*) représentent les éléments de l'ensemble $\bigcup_{n\in\mathbb{N}} R^n$.

Le vecteur (*colonne*)

$$V = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix}$$

est construit par l'une des opérations :

$$V :=< v_1,\cdots,v_n >; \quad \text{ou} \quad V := \texttt{Vector}([v_1,\cdots,v_n]);$$

On peut aussi utiliser le *constructeur* `Vector`.

On *accède* à la $i^{\text{ème}}$ composante d'un vecteur V *via* l'*opérateur de sélection* $V[i]$; le *sélecteur* peut être négatif : *e.g.* $V[-1]$ est la dernière composante du vecteur ou un *intervalle* : $V[i..j]$ est le vecteur extrait de V de la $i^{\text{ème}}$ à la $j^{\text{ème}}$ composante.

Les opérations d'algèbre linéaire nécessitent le *package* `LinearAlgebra`.

`Dimension`	nombre de composantes
`Equal`	test de l'égalité
`UnitVector`	vecteur unité
`ZeroVector`	vecteur nul
`DotProduct`	produit hermitien
`CrossProduct`	produit vectoriel
`VectorAdd`	combinaison linéaire
`VectorScalarMultiply`	multiplication par un scalaire
`VectorNorm`	norme
`VectorAngle`	angle
`Normalize`	vecteur unitaire associé

Opérations sur les *espaces vectoriels* :

`Basis`	base d'un ev
`IntersectionBasis`	base d'une intersection d'ev
`SumBasis`	base d'une somme d'ev
`RowSpace`	ev engendré par les lignes d'une matrice
`ColumnSpace`	ev engendré par les colonnes d'une matrice

1.9.3 Les matrices

Construction.

Les matrices à composantes dans t (type `'Matrix'`(t), *ne pas omettre les accents aigus*) représentent les éléments de l'ensemble $\bigcup_{n,p\in\mathbb{N}} \mathbf{M}_{n,p}(R)$.

La matrice à n lignes et p colonnes :

$$M = \begin{pmatrix} a_{1,1} & \cdots & a_{1,n} \\ \vdots & & \vdots \\ a_{n,1} & \cdots & a_{n,n} \end{pmatrix}$$

peut se construire ligne par ligne :

$$M :=<< a_{1,1}|\cdots|a_{1,n}>,\cdots,<a_{n,1}|\cdots|a_{n,n}>>;$$

ou bien colonne par colonne :

$$M :=<< a_{1,1},\cdots,a_{n,1}>|\cdots|<a_{1,n},\cdots,a_{n,n}>>;$$

On peut aussi utiliser le *constructeur* `Matrix` :

$$\texttt{Matrix}(n,p,(i,j)->f(i,j))$$

dans lequel il est possible de préciser le *format* de la matrice par l'option $\texttt{shape} = \cdots$:

$$\texttt{Matrix}(n,p,(i,j)->f(i,j),\texttt{shape}=\cdots)$$

On peut encore fournir comme argument au constructeur `Matrix` la *liste* des vecteurs colonnes de la matrice.

Accès aux coefficients.

On *accède* au coefficient (i, j) d'une matrice M *via* l'*opérateur de sélection* $M[i, j]$; les *sélecteurs* peuvent être négatif : *e.g.* $M[i, -1]$ est le coefficient de la $i^{\text{ème}}$ ligne et de la dernière colonne ou des *intervalles* : $M[a..b, c..d]$ est la matrice extraite en prenant les lignes i de l'intervalle $a..b$ et les colonnes j de l'intervalle $c..d$ ou même des listes d'indices L_1,L_2 : $M[L_1, L_2]$ est la matrice extraite en prenant les indices $i \in L_1$ et les indices $j \in L_2$.
On peut d'ailleurs assigner des sous-matrices par blocs.

Opérations.

1. *Opérations algébriques.*
   - ✓ $+$ ou `MatrixAdd` pour l'addition
   - ✓ $*$ pour la multiplication par un scalaire ;
     si celui-ci est symbolique, utiliser `simplify`$(\cdots * \cdots$, `symbolic`$)$
     ou l'opération `MatrixScalarMultiply`.
   - ✓ . `MatrixMatrixMultiply` pour le produit matriciel
   - ✓ $.^{-1}$ ou `MatrixInverse` pour l'inverse.

   Les opérations d'algèbre linéaire nécessitent le *package* `LinearAlgebra`.
2. *Test du format.*
   Le *format* d'une matrice peut être testé à l'aide de l'opération `IsMatrixShape` et l'un des ses paramètres :

`zero`	`identity`	`scalar`	`diagonal`
`constant`	`band`	`symmetric`	`antisymmetric`
`hermitian`	`antihermitian`	`triangular[upper]`	`triangular[lower]`

   On peut introduire d'autres formats de matrices à l'aide d'une procédure booléenne dont le nom devra être '`IsMatrixShape`/ $< nom\_format >$ '.
3. *Opérations de base :*

`Equal`	test de l'égalité
`Dimension`	nb des lignes et de colonnes
`RowDimension`	nb de lignes
`ColumnDimension`	nb de colonnes
`Row`	extrait une ligne
`Column`	extrait une colonne
`Minor`	mineur
`NullSpace`	noyau
`Rank`	rang
`Transpose`	transposée
`Determinant`	déterminant
`Trace`	trace
`CharacteristicMatrix`	matrice caractéristique
`CharacteristicPolynomial`	polynôme caractéristique
`Eigenvalues`	valeurs propres
`Eigenvectors`	vecteurs propres
`MinimalPolynomial`	polynôme minimal
`JordanForm`	forme de Jordan

4. *Construction de matrices particulières :*

`ZeroMatrix`	matrice nulle
`IdentityMatrix`	matrice identité
`ConstantMatrix`	matrice constante
`ScalarMatrix`	matrice scalaire
`DiagonalMatrix`	matrice diagonale
`BandMatrix`	matrice par bandes
`TridiagonalForm`	matrice tridiagonale
`VandermondeMatrix`	matrice de Van der Monde
`JordanBlockMatrix`	bloc de Jordan

5. *Opérations sur les lignes et les colonnes :*

`DeleteRow`	supprimer une ligne
`DeleteColumn`	supprimer une colonne
`RowOperation`	ajoute à une ligne un multiple d'une autre
`ColumnOperation`	ajoute à une colonne un multiple d'une autre

6. *Equations linéaires :*

`GenerateMatrix`	met un système linéaire sosu forme matricielle
`GenerateEquations`	forme des équations linéaires à partir d'une matrice
`LinearSolve`	résolution d'un système linéaire

7. *Formes réduites :*

`GaussianElimination`	élimination de Gauss
`ReducedRowEchelonForm`	forme échelonnée
`LUDecomposition`	décomposition LU
`FrobeniusForm`	forme de Frobenius
`SmithForm`	forme de Smith
`HermiteForm`	forme de Hermite

1.10 Les constantes et les expressions booléennes

Les *constantes booléennes* sont `true`, `false` mais aussi `FAIL`.
Les *opérateurs booléens* (type `logical`) sont :

`and`	conjonction
`or`	disjonction inclusive
`xor`	disjonction exclusive
`not`	négation

Constantes et opérateurs booléens permettent de former avec les *relations* (type `relation`) les expressions booléennes (type `boolean`).
La fonction `evalb` permet de forcer l'évaluation d'une expression booléenne (lorsque celle-ci n'est pas en position d'être évaluée automatiquement).
De plus les opérations `andmap` (resp. `ormap`) permettent d'évaluer si une propriété est vraie pour tous les opérandes (resp. pour au moins un opérande) d'une expression (le plus souvent une liste ou un ensemble).

1.11 Caractères et chaînes

Les *caractères* (type `character`) et les *chaînes* (séquences de caractères- type `string`) sont délimités par des *guillemets* "···". La fonction `cat` permet la *concaténation* des chaînes et la fonction `substring` permet d'extraire un sous-chaïne. Le *package* `StringTools` contient de nombreuses autre opérations.

1.12 Les instructions

La résolution d'un problème se décompose en une *succession* d'étapes (ou d'instructions). De plus le passage d'une étape à la suivante devra être *automatisé*, aussi bien en ce qui concerne *l'enchaînement* des étapes entre elles, que la *transmission des données* d'une étape à la suivante. On aboutit ainsi à la construction de *séquences de calcul* (computation sequences `CompSeq`) formalisées en *procédures*.
L'instruction élémentaire consiste en *la formation d'une expression, son évaluation et en l'affectation du résultat à une variable*.
L'enchaînement de base est la *séquence d'instructions*, suite finie d'instructions, séparées par des point-virgule ; et s'exécutant l'une après l'autre, les résultats de l'une des instructions de la séquence coïncidant avec les données de l'instruction suivante. Il est souhaitable de réunir ses instructions en *un bloc d'exécution* de sorte que toute demande d'exécution de l'une des instructions du bloc déclanche l'exécution du bloc complet.
Les autres enchaînements possibles sont la sélection et par répétition.

1.12.1 L'instruction de sélection

L'*instruction de sélection* permet le choix entre l'exécution de deux (séquences d')instructions selon qu'une condition est vérifiée ou non (sélection simple). Sa syntaxe est :

```
if < condition >  then  < seq_instruction 1 >
                  else  < seq_instruction 2 >
                                end if;
```

La *condition de sélection* est une expression booléenne.
Parfois il peut être utile de combiner plusieurs instructions conditionnelles. On utilise alors la syntaxe :

```
if < condition >  then     < seq_instruction 1 >
                  elif     < seq_instruction 2 >
                    ⋮
                  else   < autre_seq_instruction >
                                end if;
```

1.12.2 La répétition

La boucle générale :

Une *instruction de répétition* (ou *boucle*) permet de répéter *indéfiniment* une suite d'instructions. Sa syntaxe est

```
do  < seq_instruction >  end do ;
```

Naturellement il est indispensable d'ajouter à cette structure de base *une condition d'arrêt.* La fonction `break` permet de quitter une boucle ; la boucle la plus générale aura donc la forme :

```
do
    < seq_ instruction 1 >;
if   < condition > then   break  end if;
    < seq_ intruction 2 >;
          end do;
```

La fonction `next` permet de passer à l'itération suivante dans la boucle :

```
do
    < seq_ instruction 1 >;
if   < condition > then   next  end if;
    < seq_ intruction 2 >;
          end do;
```

La répétition calculée - boucle `for` :

lorsque le nombre d'itérations est connu à priori, on utilise une variable numérique pour en contrôler le nombre (le pas peut être négatif) : la syntaxe est :

```
for < indice >     from < début >     to < fin >  [by < pas >]
                do  < seq_ instruction >
                      end do;
```

Souvent cette boucle peut être évitée par l'emploi de séquences ou de listes ainsi que par les fonction `add`, `sum`, `mul`, `product`.

La boucle `while` :

la répétition se poursuit tant qu'une condition reste vraie :sa syntaxe est :

```
while     < condition >     do
          < seq_ instruction >
                end do;
```

Les boucles avec condition d'arrêt multiple :

On peut introduire *plusieurs conditions d'arrêt* dans une boucle et c'est, bien sûr, la première qui sera vérifiée qui provoquera la sortie de la boucle :

```
for  < indice >  from < début >      [to < fin >]      [by < pas >]  while  < condition >
                    do
                              < seq_ instruction 1 >;
                     if         < condition > then      break     end if;
                              < seq_ intruction 2 >;
                end do;
```

1.13 Les algorithmes et les procédures

Une procédure est la traduction d'un algorithme dans un langage de programmation donné.

1.13.1 Algorithmes

Un *algorithme* est un enchaînement fini d'actions permettant à partir d'objets donnés : *les entrées* de construire de manière *effective* d'autres objets : *les sorties*. L'algorithme est, en quelque sorte, la *description de cette construction caractérisée par son effectivité et sa finitude*. Les enchaînements admis dans cette description sont :

- la *séquence d'instructions*
- la *répétition* d'une séquence d'instructions
- la *sélection* entre plusieurs séquences d'instructions.

Les actions sont des sous-algorithmes déjà disponibles.

Il faut ensuite étudier (pour les algorithmes "exacts" du calcul formel) les points suivants :

- *terminaison* quelques soient les entrées, l'algorithme doit se terminer en un nombre fini d'étapes. Cette vérification est indispensable lorsque l'enchaînement comprend des *boucles* non calculées.
- *correction* il faut démontrer que, quelques soient les données, le résultat obtenu est exact
- *complexité* il s'agit d'estimer *a priori* le temps que mettra l'algorithme pour fournir un résultat à partir des données initiales. La quantité de mémoire nécessaire à l'exécution de l'algorithme est aussi à prendre en compte.

Pour les algorithmes numériques il y a d'autres problèmes tels que la *précision* et la *stabilité* numériques qui se posent et dont l'étude relève de l'analyse numérique.

1.13.2 Procédures

Les données en entrée sont transmises par l'intermédiaire des *paramètres*. Le domaine de validité de la procédure doit être strictement contrôlé par une vérification précise du type des paramètres. Le résultat peut être une expression de type quelconque (souvent une séquence).
En aucun cas (sauf dans le cas un arrêt sur erreur via l'instruction `error`*, ou, provisoirement, dans le cas d'un débogage) une procédure ne doit faire afficher directement des messages ou des résultats (via une instruction* `print` *par exemple). En effet le résultat obtenu en sortie de la procédure doit toujours pouvoir être transmis comme donnée en entrée à une autre procédure ; c'est le principe même de la construction des séquences de calcul, base de la programmation.*
Un exemple *test* doit toujours compléter l'écriture d'une procédure. La syntaxe, dans le langage *MAPLE*, est la suivante :

```
< nom_proc > := proc (< séquence des paramètres >
                description " < texte >";
                local < séquence des variables locales >;
                options < séquence des options >;
                        < seq_instruction >;
                        end proc ;
```

Voici quelques principes concernant l'écriture et l'utilisation des procédures :

1. L'exécution d'une procédure se fait par une instruction de la forme :

$$result := nom\_proc(p_1, \cdots, p_n);$$

où $p_1, \cdots, p_n$ sont les paramètres *effectifs* que l'on substitue aux paramètres *formels* figurant dans la déclaration de la procédure. La *déclaration* d'une procédure est purement *descriptive* ; c'est l'*appel* de la procédure qui provoque son *exécution*.

2. Le type des paramètres doit être contrôlé. *C'est la garantie que la procédure ne sera utilisée que pour des données valides.* Le type peut être précisé dans l'en-tête sous la la forme

   $< param >::< type >$

   ($< param >$ équivaut à $< param >::$ *anything* c'est à dire que le paramètre peut être d'un type quelconque) ou bien avec des instructions de la forme :

   `if not type(`$< param >, < type >$`) then error`$"< message >"$ `end if;`

3. (a) Le plus souvent, les paramètres d'une procédure sont en mode *entrée* : lors de l'exécution de la procédure les paramètres *effectifs* sont d'abord évalués et les valeurs sont substitués aux paramètres *formels* corrrespondants.

   (b) Il est possible d'utiliser des paramètres en mode *sortie* : lors de l'exécution de la procédure p devra être une variable libre qui sera affectée dans le corps de la procédure, en utilisant une déclaration de la forme :

   $test :=$ `proc(`p `::symbol)`

   (c) Il est aussi possible d'utiliser des paramètres en mode *entrée-sortie* en utilisant une déclaration de la forme :

   $test :=$ `proc(`p `::uneval)`

   ou bien

   $test :=$ `proc(`p `::name(`t`))`

   ou t est un type quelconque. Dans la première forme le paramètre effectif peut-être quelconque tandis que dans le second il devra être une variable *affectée* de type t, l'appel s'exécutant sous la forme

   test('p')

4. Les fonctions `args` et `nargs` utilisées dans le corps d'une procédure, qui renvoient les paramètres *effectifs* et leur nombre, permettent d'utiliser des procédures à un nombre variable d'arguments.
5. L'action que réalise la procédure devra être précisée succinctement dans le champ `description`.
6. 0n ne peut pas utiliser un paramètre à gauche d'un signe d'affectation := à l'intérieur d'une procédure. Si nécessaire il faut en *créer une copie* à l'aide d'une variable locale.
7. Dans le cas d'un paramètre de type `Array`, `Vector` ou `Matrix`, il est possible de modifier des composantes à l'intérieur d'une procédure, ; si l'on veut que le paramètre initial ne soit pas modifié, il faut en faire une copie locale au moyen de l'opération `copy`.
8. La valeur renvoyée est la *dernière expression évaluée* (il est recommandé de terminer toute procédure en rappelant cette valeur). La fonction `return` permet de renvoyer une valeur et de quitter la procédure.
9. Lorsque la valeur de `kernelopts(assertlevel)` est égale à 2, le type de retour de la procédure peut-être spécifié dans l'en-tête :

   nom_proc`:=proc(`$< séquence\ des\ paramètres >$ `::`$<type\_retour>$ `;`

10. Les *variables locales* ont une portée limitée à l'intérieur de la procédure. Il faut éviter l'utilisation de *variables globales* à l'intérieur d'une procédure (les *effets de bord* que de telles variables provoquent sont incontrôlables). Si on utilise de telles variables, il est est souhaitable des les déclarer dans le champ `global` de la procédure.

11. Une procédure peut-être *récursive* (i.e. comporter un appel à elle-même). L'*option* `remember` permet de construire la *table des appels* permettant de mémoriser les valeurs déjà calculées. Cette rable est accessible par `op`(4,`eval`(*nom_proc*)). On peut y inscrire des valeurs par l'opération *nom_proc*$(\cdots) = \cdots$. La table des appels peut-être effacée par `forget`(nom_proc).

1.13.3 Les types

Les types permettent de constituer des classes d'expressions ; les *types superficiels* donnent l'opérateur d'une expression tandis que les *types structurés* permettent d'analyser plus finement une expression ou de représenter une structure mathématique.
De nombreux types sont prédéfinis dans *MAPLE* (on en trouve la liste dans l'aide de la fonction `type`) ; à partir de ceux-ci il est possible de former de nombreux types structurés tels que *listes*, *ensembles*, *sommes* '$\&+$', *produits* '$\&*$', *etc.* de types. Enfin la fonction `AddType` du *package* `TypeTools` permet de définir de nouveaux types.
Exemple : Définir le type *ExprMath(x)* des expresions fonctionnelles en une variable x :

```
> AddType(ExprMath,proc(e,x::name)
>               if type(e,realcons) then return true end if;
>               if e=x then return true end if;
>               if indets(evalf(e),name)={x} and type(e,{'+','*','^',function})
>                           then return andmap(type,[op(e)],ExprMath(x)) end if;
>                   false;
>                   end proc);
```

Enfin si e est une expression d'un type t donné et si nt est un nouveau type la fonction `convert`(*e*,*nt*) permet de convertir l'expression à la nouvelle forme nt (*eg.* pour calculer la somme ou le produit des éléments d'une liste ou d'un ensemble ou en *trigonométrie*).

D'autre part, le *constructeur* `SetOf` appliqué à un type permet de construire l'ensemble abstrait associé (qui n'est pas du type `set`) On peut appliquer à ces *ensembles abstraits* les opérations suivantes :

`in`	appartenance
`union`	réunion
`intersect`	intersection
`minus`	différence

1.14 Les visualisations

Le *package* `plots` contient la plupart des opérations de visualisation. Chacune de ces opérations détermine une *structure de tracé* (`plot structure`) en $2D$ (`PLOT`) ou en $3D$ (`PLOT3D`) ce sont des structures de données incluant le type de tracé, les coordonnées (en nombres flottants) des points du dessin ainsi que des éléments de style du tracé.
Les dessins s'exécutent par défaut dans la feuille de calcul active, cependant on peut le visualiser dans une nouvelle fenêtre par

```
plotsetup(window)
```

On revient à la feuille de calcul par

```
plotsetup(default)
```

De plus le sytème graphique de *MAPLE* propose de nombreux outils pour élaborer les dessins *en mode interactif*, accessibles par des `menus contextuels`.

1.14.1 Représentations de base

Graphes des fonctions usuelles

On considère une expression à valeurs numériques e contenant une unique variable libre x (*cf.* l'exemple de définition du type *ExprMath(x)*)

Soit $a..b$ (type `range`) un intervalle ; le graphe de e pour $x \in a..b$ est obtenu par l'opération (qui crée la *structure de tracé* correspondant et visualise le dessin correspondant) :

$$\texttt{plot}(e, x = a..b);$$

dans laquelle on peut spécifier de nombreuses options parmi lesquelles :

`title`	titre du dessin
`discont=true`	discontinuités
`color`	couleur du tracé
`scaling=constrained`	axes orthonormés
`numpoints`	nb de points du partage
`view`	fenêtre de tracé
`thickness`	épaisseur du trait
`labels`	nom des axes
`labeldirections`	disposition des noms des axes
`legend`	légende d'une courbe

Dans le cas de plusieurs expressions $e_1, \cdots, e_n$ représentant des fonctions, on peut représenter leurs graphes sur un même dessin par l'opération :

$$\texttt{plot}([e_1, \cdots, e_n], x = a..b);$$

dans laquelle l'option color peut prendre la forme $color = [c_1, \cdots, c_n]$.
Pour construire un dessin à partir de plusieurs opérations différentes, on doit *dissocier le calcul de la structure de tracé et sa visualisation* ; la structure est calculée par :

$$G := \texttt{plot}(e, x = rh) :$$

Les différentes structures $G_1, \cdots, G_n$ seront ensuite visualisées par *superposition* sur le même dessin par :

$$\texttt{display}(G_1, \cdots, G_n);$$

ou *successivement*, créant ainsi une animation, par :

$$\texttt{display}(G_1, \cdots, G_n, \texttt{insequence=true});$$

Listes

Une liste L de couples de valeurs numériques est représentée dans le plan par l'opération :

$$\texttt{listplot}(L)$$

On peut spécifier les options `style=line/point` et symbol=circle/cross/box/diamond.
Une liste $[a_1, \cdots, a_n]$ de valeurs numériques est visualisée comme la liste de points $[[1, a_1], \cdots, [n, a_n]]$.

Figures géométriques

Le package `plottools` contient les fonctions permettant de tracer les figures usuelles aussi bien du plan (*e.g.* `point`, `line`, `circle`, `disk`,`rectangle`, *etc.*) que de l'espace (*e.g.* `sphere`, `cylinder`, `torus`, `tetrahedron`, *etc.*)

1.14.2 Courbes

Courbes paramétriques et polaires.

Une courbe paramétrique est représentée par une liste de la forme $[x, y, t = a..b]$ où x et y sont deux expressions à valeurs numériques contenant une unique variable libre t et représentant les coordonnées cartésiennes du point de la courbe de paramètre t. Le calcul de la structure de tracé et sa visualisation sont obtenues par l'opération

`plot`$([x, y, t = a..b])$;

à laquelle on peut rajouter les différentes options de tracé.

Une courbe polaire est représentée par une liste de la forme $[\rho, \theta, t]$ où ρ et θ sont deux expressions à valeurs numériques contenant une unique variable libre t et représentant le rayon vecteur et l'angle polaire du point de paramètre t de la courbe. On trace cette courbe par :

`plot`$([\rho, \theta, t]$,`coords=polar`$)$; ou `polarplot`$([\rho, \theta, t])$;

avec les options habituelles.
Dans le cas particulier où $\theta = t$ on a la forme abrégée :

`polarplot`$(\rho, \theta = a..b)$;

Courbes cartésiennes

Une courbe cartésienne est représentée par une expression à valeurs numériques F contenant deux variables libres X et Y, représentant la relation qui lie les cordonnées cartésiennes X et Y d'un point de la courbe $F(X, Y) = 0$. Le tracé est obtenu par l'opération :

`implicitplot`$(F, X = a..b, Y = c..d,$`grid`$= [nx, ny])$

où $nx \times ny$ est la *résolution de la grille* utilisée pour le tracé. On dispose encore des options habituelles.

Le package `algcurves` contient des fonctions permettant d'étudier les courbes algébriques :

`homogeneous`	calcul de l'équation homogénéisée
`singularities`	détermine les singularités
`puiseux`	calcul des développements de Puiseux
`genus`	calcul du genre
`parametrization`	paramétrisation d'une courbe de genre 0

Equations polaires implicites

Pour une courbe polaire ce sont le rayon vecteur ρ et l'angle polaire θ d'un point de la courbe qui sont reliés par une relation. L'opération de tracé est alors :

`implicitplot`$(F, \rho = a..b, \theta = c..d,$`coords=polar`$,$`grid`$= [nx, ny])$

1.14.3 Courbes gauches et surfaces

Courbes gauches

Une courbe gauche est représentée par une liste de la forme $[x, y, z, t = a..b]$ où x, y et z sont des expressions à valeurs numériques contenant une unique variable libre t et représentant les coordonnées cartésiennes du point de la courbe de paramètre t. Le calcul de la structure de tracé et sa visualisation sont obtenues par l'opération

`spacecurve`$([x, y, z], t = a..b);$

L'opération `display3d` permet de visualiser les structures de tracé en 3d. Voici quelques une des options :

`title`	titre du dessin
`orientation`	orientation dans l'espace
`light`	éclairage
`color`	couleur du tracé
`coords`	système de coordonnées

Graphe d'une fonction de deux variables

Etant donnée une expression numérique contenant deux variables libres X et Y, le tracé de la surface d'équation $Z = F(X, Y)$ est obtenu par l'opération :

`plot3d`$(F, X = a..b, Y = c..d)$

Signalons l'opération :

`densityplot`$(F, X = a..b, Y = c..d, \texttt{grid} = [rh, rv], color = \cdots)$

permettant de représenter une *plaque* $[a, b] \times [c, d]$ ou chaque point (X, Y) a la densité $F(X, Y)$.

Surfaces paramétrées

Une surface paramétrée est représentée par une liste [x,y,z] de trois expressions à valeurs numériques contenant deux variables libres u et v ; le tracé de la surface est obtenu par :

`plot3d`$([x, y, z], u = a..b, v = c..d)$

Surfaces implicites

Une surface cartésienne est représentée par une expression à valeurs numériques F contenant trois variables libres X, Y et Z représentant la relation qui lie les cordonnées cartésiennes X, Y et Z d'un point de la surface $F(X, Y, Z) = 0$. Le tracé est obtenu par l'opération :

`implicitplot3d`$(F, X = a..b, Y = c..d, Z = e..f)$