Quelques rappels sur l'utilisation de *Maple*

Le but de ce document est de faire quelques rappels sur les commandes de base du logiciel de calcul formel *Maple*. L'approche est heuristique et la liste des commandes décrites est très loin d'être exhaustive. On trouvera de nombreux guides d'utilisation de Maple plus complets et plus raisonnés sur la toile. Relevons par exemple :

- Jean-Louis Maltret de l'Université d'Aix-Marseille II, *Un guide en ligne*, http ://lumimath.univ-mrs.fr/∼jlm/cours/maple/maple.html
- Bruno Salvy de l'INRIA, *Petit guide de simplification en Maple* http ://algo.inria.fr/salvy/M1ENS/simplification.pdf
- Francis Sergeraert de l'Université de Grenoble I, *Maple expliqué*, http ://www-fourier.ujf-grenoble.fr/∼sergerar/Papers/Maple-explique.pdf

Il existe également des manuels. Par exemple citons :

- C. Gomez, B. Salvy, P. Zimmermann, *Calcul formel : mode d'emploi*, Masson, Paris, 1995.
- X. Jeanneau, D. Ligon, J.-L. Poss, *Exercices de mathématiques résolus à l'aide de Maple et Mathematica*, Ellipses, Paris, 1999.

Nous présentons un certain nombre de commandes en les regroupant en onze thèmes.

Assignation, égalité. L'assignation est la commande fondamentale des langages informatiques. Il s'agit d'affecter une valeur à une mémoire. Pour Maple, on se contentera d'y voir une manière de nommer un objet. Ainsi si l'on écrit :

```
> restart;
```

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

$$a := x + 1$$

Cela signifie qu'à partir de cette ligne `a` sera remplacé par `x+1` comme on le voit ci-dessous.

```
> b:= a+x;
```

$$b := 2\,x + 1$$

```
> b;
```

$$2\,x + 1$$

L'utilisation de `restart` efface toutes les mémoires et supprime toutes les assignations faites. Par ailleurs, il ne faut pas confondre l'assignation avec l'égalité.

```
> u= 2+x;
```

$$u = 2 + x$$

```
> u;
```

$$u$$

Dans l'exemple ci-dessus on voit que `u` n'a pas été remplacé par `2+x`. En fait l'égalité `u=2+x` est elle-même un objet de Maple et à ce titre, on peut lui donner un nom. Pour transformer une égalité en assignation, on utilise la commande `assign` :

```
> egalite:= u= 2+x;
```

$$egalite := u = 2 + x$$

```
> egalite;
```

$$u = 2 + x$$

```
> assign(egalite);
> u;
```

$$2 + x$$

Pour manipuler les égalités, rappelons l'usage de `rhs` (pour *right hand side*) et `lsh` (pour *left hand side*) qui rendent respectivement le membre de droite et de gauche d'une égalité :

```
> lhs(t=3);
```

$$t$$

```
> rhs(t=3);
```

$$3$$

Nombres. Le logiciel distingue les différents types de nombre, les entiers (`integer`), les rationnels (`rational`), les réels (`real`) et les complexes (`complex`). La commande `is` permet de tester la nature du nombre.

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

$$a := 2$$

```
> is(a, integer);
```

true

```
> b:= 2/3;
```

$b := 2/3$

```
> is(b,integer);
```

false

```
> is(b,rational);
```

true

```
> is(a,rational);
```

true

```
> c:= sqrt(2)-sqrt(3);
```

$c := \sqrt{2} - \sqrt{3}$

```
> is(c,real);
```

true

```
> Pi;
```

π

On notera ici que la constante d'Archimède se code `Pi` alors que la lettre grecque minuscule π se code `pi`. Malheureusement, les deux apparaissent sous la même graphie en sortie.

```
> is(Pi,real);
```

true

```
> is(Pi,rational);
```

false

Pour obtenir une valeur approchée décimale, on utilisera la commande `evalf`. Cette commande permet en général de faire des calculs approchées (d'intégrale, de sommes de série...). On note que si l'un des termes d'un calcul algébrique est un décimal, le résultat sera donné sous forme décimale approchée.

```
> evalf(Pi);
```

3.141592654

```
> evalf(b);
```

0.6666666667

```
> 1.5+b;
```

2.166666667

La valeur absolue est codée à l'aide de `abs`.

```
> abs(-Pi);
```

$$\pi$$

Maple permet également la manipulation des nombres complexes. Le nombre dont le carré est -1 est codé par `I` et l'on dispose des outils pour calculer les parties réelles et imaginaires (`Re` et `Im`), le modules et un argument (`abs` et `argument`). Le logiciel effectue aussi les calculs algébriques usuels dans $\mathbb{C}$.

```
> is(a,complex);
```

$$true$$

```
> z:= 2+3*I;
```

$$z := 2 + 3\,i$$

```
> Re(z); Im(z);
```

$$2$$
$$3$$

```
> 1/z;
```

$$2/13 - 3/13\,I$$

```
> abs(z);
```

$$\sqrt{13}$$

```
> conjugate(z); argument(z);
```

$$2 - 3\,I$$
$$\arctan\left(3/2\right)$$

```
> restart;
```

```
> is(A,real);
```

$$false$$

A priori, une lettre non affectée n'est pas considérée comme un nombre. Mais il est possible de le supposer pour faire certains calculs. On utilise alors la fonction `assume`. (Le verbe anglais *to assume* se traduit par *supposer* en français.)

```
> assume(A, real);
> A;
```

$$A\sim$$

```
> is(A,real);
```

$$true$$

Lorsqu'on fait une supposition sur une lettre, celle-ci sera suivie d'un tilde pour rappeler ce fait. On peut supprimer les tildes ainsi :

```
> interface(showassumed=0);
> A;
```

$$A$$

ou les remplacer par une phrase de rappel ainsi :

```
> interface(showassumed=2);
> A;
```

$$A$$
with assumptions on A

La commande `interface(showassumed=1);` repasse en mode « tildé ». Faire des suppositions sur la nature des objets permet de calculer à l'aide des commandes réservées aux nombres ou de faire des simplifications comme ci-dessous.

```
> Re(A+I*B);
```

$$\Re(A+IB)$$

```
> cos(n*Pi);
```

$$\cos(n\pi)$$

Les calculs ne se sont pas fait car les natures de `B` et de `n` ne sont pas précisées. Si l'on suppose que le premier est réel et le second entier on obtient les résultats suivants.

```
> assume(B,real);
> Re(A+I*B);
```

$$A\sim$$

```
> assume(n,integer);
> cos(n*Pi);
```

$$(-1)^{n\sim}$$

Séquences, listes et ensembles. Une séquence est un objet constitué de plusieurs objets séparés d'une virgule :

```
> sequ:= x, 1, a, b, 17, a, 2, 4;
```

$$sequ := x,\ 1,\ a,\ b,\ 17,\ a,\ 2,\ 4$$

La commande `seq` permet de construire des séquences d'objets dépendants d'un paramètre entier :

```
> seq( i^2, i=-3..5);
```

$$9,\ 4,\ 1,\ 0,\ 1,\ 4,\ 9,\ 16,\ 25$$

En soit les séquences sont d'un usage réduit. Pour manipuler des listes ou des ensembles (dans le sens du langage mathématique) d'objets, on utilise les listes et les ensembles (dans le sens de Maple).

Une liste est une séquence entre crochets. Les objets d'une liste sont ordonnés et numérotés. On peut ainsi disposer de chaque élément de la liste. La numérotation commence à 1.

```
> lst:= [sequ];
```

$$lst := [x, 1, a, b, 17, a, 2, 4]$$

```
> lst[1];
```

$$x$$

```
> lst[3];
```

$$a$$

Un ensemble est une séquence entre accolades. Il permet de coder les ensembles mathématiques. Les objets ne sont pas répétés.

```
> ens:={sequ};
```

$$ens := \{1, 2, 4, 17, x, a, b\}$$

On peut encore disposer de chaque élément à l'aide de la grammaire `ens[1]`. Néanmoins il faut faire attention car l'ordre de l'ensemble que l'on construit peut être changé par le logiciel comme c'est le cas dans l'exemple.

```
> ens[1];
```

$$1$$

Pour revenir à une séquence à partir d'un ensemble ou d'une liste, on peut utiliser la fonction `op` alors que la fonction `nops` donne le nombre d'éléments de la liste ou de l'ensemble. Cette dernière ne fonctionne pas pour les séquences.

```
> op(lst);
```

$$x,\ 1,\ a,\ b,\ 17,\ a,\ 2,\ 4$$

```
> op(ens);
```

$$1,\ 2,\ 4,\ 17,\ x,\ a,\ b$$

```
> nops(lst);
```

$$8$$

```
> nops(ens);
```

$$7$$

```
> nops(sequ);
```

```
Error, wrong number (or type) of parameters in function nops
```

Expressions, fonctions. Les expressions sont des objets fondamentaux de Maple. Il s'agit d'expressions (au sens mathématique) construites avec des nombres et des inconnues littérales à l'aide d'opérations algébriques. Par exemple :

```
> a:= (x +2)*(x-1)+(x^2+ 4*x+ 4);
```

$$a := (x+2)(x-1)+x^2+4x+4$$

Maple est capable de transformer les expressions, par exemple de les simplifier ou de les factoriser. Il peut également résoudre des équations.

```
> simplify(a);
```

$$2x^2+5x+2$$

```
> factor(a);
```

$$(x+2)(2x+1)$$

```
> solve(a=0,x);
```

$$-1/2,\ -2$$

Si l'on souhaite remplacer `x` par une valeur ou une autre expression, on utilise la fonction `subs` (pour *substitution*). On peut faire plusieurs substitutions simultanées en les codant sous la forme d'un ensemble :

```
> subs( x=1, a);
```

$$9$$

```
> subs( { A=x, B=y}, A*B+27);
```

$$xy+27$$

La fonction `subs` est un peu lourde en particulier si l'on considère l'expression comme une fonction de `x`. Il est possible de définir une fonction de la manière suivante :

```
> f:= x-> (x +2)*(x-1)+(x^2+ 4*x+ 4);
```

$$f := x \mapsto (x+2)(x-1)+x^2+4x+4$$

On a alors :

```
> f(1);
```

$$9$$

L'expression ne permet pas d'utiliser cette écriture en effet on a :

```
> a(1);
```

$$(x(1)+2)(x(1)-1)+(x(1))^2+4x(1)+4$$

Le nom de la variable d'une fonction est muet, on aurait de manière équivalente pu écrire :

```
> f:= p-> (p +2)*(p-1)+(p^2+ 4*p+ 4);
```

$$f := p \mapsto (p+2)\,(p-1) + p^2 + 4\,p + 4$$

En revanche les commandes de Maple vues ci-dessus et prévues pour des expressions ne fonctionnent plus pour les fonctions :

```
> solve(f=0,x);
```

```
> factor(f);
```

$$f$$

```
> simplify(f);
```

$$f$$

On peut facilement transformer une fonction en expression. Il suffit de l'évaluer avec une lettre :

```
> aa:= f(x);
```

$$aa := (x+2)\,(x-1) + x^2 + 4\,x + 4$$

Réciproquement, on fabrique une fonction à partir d'une expression à l'aide de `unapply` :

```
> ff:= unapply(a,x);
```

$$ff := x \mapsto (x+2)\,(x-1) + x^2 + 4\,x + 4$$

Pour dériver une expression, on utilisera `diff` (il faut préciser la variable). Pour dériver une fonction, on utilisera `D`. Dans le premier cas, la sortie sera une expression, dans le second, une fonction :

```
> da:= diff(a,x);
```

$$da := 4\,x + 5$$

```
> df:=D(f);
```

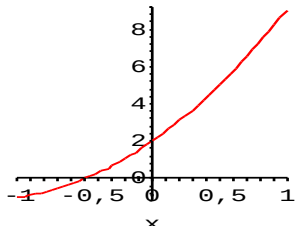
$$df := p \mapsto 5 + 4\,p$$

Certaines fonctions de Maple sont mixtes fonctionnant à la fois pour des fonctions et des expressions, à condition de les utiliser correctement. Les trois commandes suivantes donnent le même résultat. Remarquez que dans les deux premières, les arguments de `plot` sont des expressions et que dans la dernière, il s'agit d'une fonction.

```
> plot(a,x=-1..1);
```

```
> plot(f(x),x=-1..1);
```

```
> plot(f,-1..1);
```



Lettres indicées Nous avons vu que les listes permettaient de construire des suites de nombres ou d'objets :

```
> a:= [seq( -i,i=0..5)];
```

$$a := [0, -1, -2, -3, -4, -5]$$

```
> a[1]; a[2];
```

$$0$$
$$-1$$

Il est possible de définir directement les éléments d'une suite en utilisant les lettres indicées. Par exemple, si l'on veux définir trois nombres r_0, r_1 et r_2, on écrira :

```
> r[0]:= 1; r[1]:= 3; r[2]:= 4;
```

$$r_0 := 1$$
$$r_1 := 3$$
$$r_2 := 4$$

Néanmoins, `r` n'en devient pas pour autant une liste. Pour avoir la liste des éléments indicés, il faut utiliser `seq` :

```
> r;
```

$$r$$

```
> seq(r[i], i=0..2);
```

$$1, 3, 4$$

De plus, les éléments indicés ne sont pas des fonctions. Si l'on écrit

```
> u[n]:= n^2/ 2;
```

$$u_n := 1/2\, n^2$$

puis l'on tape :

```
> u[7];
```

$$u_7$$

on constate que Maple n'a pas remplacé `n` par 7 dans l'expression. Pour faire cela, il faut soit écrire `u[7]:= subs(n=7,u[n])`, soit définir la suite comme une fonction `v` de variable `n` :

```
> v:= n->n^2/ 2; v(7);
```

$$v := n \mapsto 1/2\, n^2$$

$$\frac{49}{2}$$

Boucles. Les lettres indicées se prêtent particulièrement au calcul des termes de suites définies par récurrence. On utilise alors des boucles itératives. Leur grammaire est :

`for` *l'indice* `from` *la première valeur de l'indice* `to` *sa dernière valeur* `do` *la ou les commandes à itérer* `od;`

Par exemple, si l'on souhaite calculer les premiers termes de la suite définie par $u_0 = 0$ et $u_{n+1} = u_n - \frac{1}{u_n^2+1}$, on écrira

```
> u[0]:= 0;
```

$$u_0 := 0$$

```
> for i from 0 to 5 do  u[i+1]:=  u[i]-1/( u[i]^2+ 1) ; od;
```

$$u_1 := -1$$

$$u_2 := -3/2$$

$$u_3 := -\frac{47}{26}$$

$$u_4 := -\frac{153171}{75010}$$

$$u_5 := -\frac{4877459662937311}{2181880029128410}$$

$$u_6 := -\frac{14963957244726236395991398564078769325546367133 1}{6229314006318155186174120505493171513064400461 0}$$

Il est possible d'introduire un test d'arrêt qui fera achever les calculs avant la dernière valeur de l'itération. La grammaire est alors :

`for` *l'indice* `from` *la première valeur de l'indice* `to` *sa dernière valeur* `while` *un test qui s'il est vérifié fait continuer la boucle* `do` *la ou les commandes à itérer* `od;`

Par exemple, pour la suite précédente, si l'on souhaite calculer les premiers termes tant qu'ils sont supérieurs à -2, on modifiera la boucle ainsi :

```
> for i from 0 to 5 while u[i]> -2 do
> u[i+1]:= u[i]-1/(u[i]^2+1);   od;
```

$$u_1 := -1$$

$$u_2 := -3/2$$

$$u_3 := -\frac{47}{26}$$

$$u_4 := -\frac{153171}{75010}$$

Le premier élément inférieur à -2 est u_4.

Conditionnelles. Un autre structure de programmation fondamentale est la fonction conditionnelle. Sa grammaire est

`if` *le test* `then` *la ou les commandes à exécuter si le test donne un résultat vrai* `else` *la ou les commandes à exécuter si le test donne un résultat faux* `fi;`

Dans l'exemple suivant nous avons inséré la commande conditionnelle dans une boucle pour calculer les premiers termes de la suite définie par la récurrence suivante. On pose

$$y_0 = 1 \quad \text{et} \quad y_{n+1} = \begin{cases} y_n^2 + 3 \text{ si } y_n < 3 \\ y_n - 1 \text{ sinon.} \end{cases}$$

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

$$y_0 := 1$$

```
> for i from 0 to 5 do if y[i] < 3 then y[i+1]:=y[i]^2+3;
> else y[i+1]:=y[i]-1; fi;    od;
> seq( y[i], i=0..6);
```

$$1, 4, 3, 2, 7, 6, 5$$

Fonction solve. L'une des fonctions de Maple la plus usitée est `solve` qui permet de résoudre des équations et parfois des inéquations. Elle prend en arguments l'équation à résoudre sous forme d'une égalité d'expressions (ou d'un expression, dans quel cas `=0` est sous-entendu) et de la variable pour laquelle on demande de résoudre. Quand l'expression ne comporte qu'une inconnue il est superflu de la préciser. Les trois commandes suivantes donnent le même résultat.

```
> solve(x+2=0, x);
> solve( x+2, x);
> solve(x+2);
```

$$-2$$

Lorsque l'équation comporte plusieurs inconnues, on peut traiter l'une d'entre-elles comme un paramètre :

```
> solve( x+y=1, x);
```

$$-y + 1$$

Il est possible de résoudre un système d'équations et pour plusieurs inconnues. Dans ce cas, on remplace équations et inconnues par des ensembles d'équations et d'inconnues.

```
> solve( { x+y=1, x-2*y=2 }, {x,y});
```

$$\{y = -1/3, x = 4/3\}$$

Dans certains cas, `solve` ne donne pas immédiatement les solutions explicites :

```
> S:=[solve(x*y=1, x+y=3,x,y)];
```

$$S := [\{y = RootOf\left(\_Z^2 - 3\,\_Z + 1, label = \_L1\right), x = -RootOf\left(\_Z^2 - 3\,\_Z + 1, label = \_L1\right) + 3\}]$$

La fonction `RootOf` est utilisée pour expliciter les solutions du système formé par $xy = 1$ et $x + y = 3$ comme les solutions de l'équation $Z^2 - 3Z + 1$. Pour obtenir les solutions explicites du système, on peut utiliser `allvalues` :

```
> allvalues(S);
```

$$[\{y = 3/2 + 1/2\sqrt{5}, x = 3/2 - 1/2\sqrt{5}\}], [\{x = 3/2 + 1/2\sqrt{5}, y = 3/2 - 1/2\sqrt{5}\}]$$

L'utilisateur peut lui aussi utiliser `RootOf` pour définir une solution abstraite d'une équation :

```
> a:= RootOf(Z^5-Z+4,Z);
```

$$a := RootOf\left(\_Z^5 - \_Z + 4\right)$$

```
> simplify(a^5-a);
```

$$-4$$

Mais il ne faut pas attendre de miracle. Le théorème d'Abel-Ruffini s'applique aussi au logiciel et certaines équations polynomiales ne sont simplement pas résolubles par radicaux !

```
> allvalues(a);
```

$$RootOf\left(\_Z^5 - \_Z + 4, index = 1\right), RootOf\left(\_Z^5 - \_Z + 4, index = 2\right),$$
$$RootOf\left(\_Z^5 - \_Z + 4, index = 3\right), RootOf\left(\_Z^5 - \_Z + 4, index = 4\right),$$
$$RootOf\left(\_Z^5 - \_Z + 4, index = 5\right)$$

Dans ce cas, il est parfois possible d'obtenir une solution approchée.

```
> evalf(a);
```

$$1.042678204 + 0.6871019391\,I$$

La fonction `solve` peut aussi, parfois, résoudre des inégalités.

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

$$RealRange\left(-\infty, Open\left(-1\right)\right), RealRange\left(Open\left(1\right), \infty\right)$$

```
> solve( x^2-1<0,x);
```

$$RealRange\left(Open\left(-1\right),Open\left(1\right)\right)$$

```
> solve( x^2+1>0,x);
```

$$x$$

Pour la première sortie, il faut comprendre $]-\infty,-1[\cup]1,+\infty[$, pour la seconde $]-1,1[$ et $\mathbb{R}$ pour la troisième. Lorsque Maple ne donne aucune réponse, cela peut signifier que l'ensemble des solutions est vide comme ci-dessous, mais aussi parfois que le logiciel ne parvient pas à résoudre l'inéquation. **Il faut donc être très prudent dans ce cas !**

```
> solve( x^2+2<0,x);
```

La résolution de l'inéquation inverse permet de trancher :

```
> solve( x^2+2>0,x);
```

$$x$$

Procédures. Une procédure n'est rien d'autre qu'une fonction qui nécessite plusieurs lignes de commandes. Sa grammaire est la suivante :

```
   nom:= proc( la ou les variables )
local les variables locales;
global les variables globales;
Les commandes successives en fonction des variables
end;
```

La dernière commande constitue la sortie de la procédure. Par exemple la procédure suivante n'est rien d'autre que la fonction `f` déjà définie plus haut :

```
> f:= proc(x)     x^2 + 2   end;
```

$$f := \mathbf{proc}(x)\quad x^2+2\ \mathbf{end\ proc};$$

La procédure `suite` suivante calcule les $n+1$ premiers termes de la suite (y_n) définie ci-dessus.

```
> suite:= proc(n)
> local i,y;
> y[0]:= 1;
> for i from 0 to n do
> if y[i] < 3 then y[i+1]:=y[i]^2+3; else y[i+1]:=y[i]-1; fi;
> od;
> [seq( y[i], i=0..n)];
> end;
```

$$f := \mathbf{proc}(n)\mathbf{local}\,i,\,y;\ y[0] := 1;\mathbf{for}\,i\,\mathbf{from}\,0\,\mathbf{to}\,n\,\mathbf{do}\ \mathbf{if}\,y[i] < 3\ \mathbf{then}\ y[i1] := y[i]^2+3\ \mathbf{else}\ y[i1] := y[i]-1\ \mathbf{end\ if};\ \mathbf{end\ do};\ [seq(y[i],\ i = 0..n)]\mathbf{end\ proc};$$

Pour calculer les six premiers termes, il suffit d'appliquer la fonction `suite` à 5 :

```
> suite(5);
```

$$[1, 4, 3, 2, 7, 6]$$

Expliquons maintenant la différence entre les variables locales et les variables globales. Une procédure peut utiliser un certain nombre de variables auxiliaires qui servent à effectuer les calculs. Il faut déclarer toutes les variables que l'on utilise en début de procédure. Si l'on les déclare avec `local`, les noms utilisés le seront uniquement « à l'intérieur » de la procédure et cela n'affectera pas les mêmes noms éventuellement utilisés avant et après l'utilisation de la fonction définie par la procédure. En revanche, quand on les déclare en `global`, les noms de variables utilisés seront affectés également dans le reste de la feuille de calcul. Donnons un exemple. La procédure suivante calcule les premiers termes de la suite définie par $u_0 = 1$ et $u_{n+1} = \frac{u_n}{2} + \frac{1}{u_n}$. Il est bien connu que cette suite tend vers $\sqrt{2}$. La procédure d'argument ε calculera les termes de la suite jusqu'au premier terme u_n vérifiant $|u_n - \sqrt{2}| < \varepsilon$.

```
> restart;
> racine:= proc(epsilon) local i,u,k;
> u[0]:=1;
> for k from 0 to 10
> while abs( evalf(u[k]-sqrt(2)) )  > epsilon
> do  u[k+1]:=   u[k]/2+1/u[k]; od;
> [seq(u[i], i=0..k)];
> end;
```

racine :=proc(ϵ) **local** i, u, k; $u[0] := 1$; **for** k **from** 0 **to** 10 **while** $\epsilon < abs(evalf(u[k] - sqrt(2)))$ **do** $u[k1] := 1/2 * u[k] + u[k]^{-1}$ **end do**; $[seq(u[i], i = 0..k)]$ **end proc**;

```
> racine(1/100000);
```

$$[1, 3/2, \tfrac{17}{12}, \tfrac{577}{408}]$$

La nouvelle fonction « marche ». On remarque que ni `u`, ni `i`, ni `k` qui ont été déclarées en variables locales ne sont modifiées par l'utilisation de la procédure :

```
> k;
```

$$k$$

```
> u[3];
```

$$u_3$$

```
> i;
```

$$i$$

A présent modifions les déclarations de variables.

```
> restart;
```

```
> racine:= proc(epsilon)  local i; global  u,k;
> u[0]:=1;
> for k from 0 to 10
> while abs( evalf( u[k]-sqrt(2) ))  > epsilon
> do  u[k+1]:=   u[k]/2+1/u[k]; od;
> [seq(u[i], i=0..k)];
> end;
```

$$\textbf{racine :=proc}(\epsilon)\ \textbf{local}\, i\,;\ \textbf{global}\, u,\, k;\ u[0] := 1;\ \ \textbf{for}\, k\ \textbf{from}\, 0\ \textbf{to}\, 10$$
$$\textbf{while}\, \epsilon < abs(evalf(u[k] - sqrt(2)))\ \textbf{do}\ u[k1] := 1/2 * u[k] + u[k]^{-1}\ \textbf{end do};$$
$$[seq(u[i],\ i = 0..k)]\ \textbf{end proc};$$

```
> racine(1/100000);
```

$$[1, 3/2, \tfrac{17}{12}, \tfrac{577}{408}]$$

L'utilisation de la procédure a modifié les valeurs des variables globales `u` et de `k` dans la feuille de calcul. En effet :

```
> k;
```

$$3$$

```
> u[3];
```

$$\frac{577}{408}$$

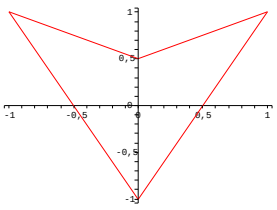
```
> i;
```

$$i$$

Tracer des courbes. La commande `plot` est la spécialiste du tracé de courbe. Nous avons vu qu'elle permettait de tracer les graphes des fonctions. Il est également possible de lui faire tracer des polygones. Il suffit de lui fournir la liste des coordonnées des points à relier sous la forme :

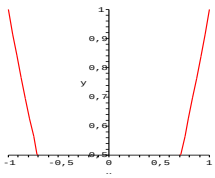
$$[[x_1, y_1], [x_2, y_2], \ldots, [x_n, y_n]].$$

```
> plot([[-1,1],[0,-1],[1,1], [0,1/2],[-1,1]]);
```



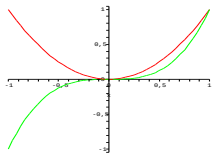
Il est également possible de restreindre la fenêtre d'un tracé en indiquant les bornes de l'axe des y.

```
> plot( x^2 , x=-1..1, y=1/2..1);
```



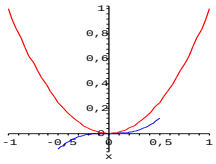
Pour tracer les graphes de deux fonctions sur le même dessin, on remplace l'expression de la fonction à tracer par un ensemble d'expressions :

```
> plot({x^2, x^3}, x=-1..1);
```



Une autre possibilité, pratique lorsque les domaines de définition sont différents, consiste à utiliser `display` de la bibliothèque `plots`. Il faut dans ce cas remplacer `;` par `:` à la fin de l'assignation de la sortie du graphique pour éviter une pollution de l'écran par les données numériques du tracé.

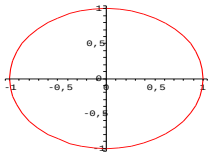
```
> with(plots):
> A:=plot( x^2 , x=-1..1, color=red):
> B:= plot( x^3 , x=-1/2..1/2, color=blue): display(A,B);
```



Le logiciel permet aussi de faire des tracés de courbes paramétrées. Il faut alors lui présenter l'argument de `plot` de la manière :

`[abs. en fonction du paramètre,ord. en fonction du paramètre, variation du paramètre]`

```
> plot( [cos(x),sin(x),x=0..2*Pi]);
```



On notera dans le dessins précédent la légère distorsion due à une différence d'échelle sur les deux axes.

Le logiciel possède plusieurs bibliothèques d'algèbre linéaire. Nous avons choisi d'utiliser la plus simple d'entre-elles. Ce document complète celui intitulé *Quelques rappels sur l'utilisation de Maple* à la bibliographie duquel nous renvoyons pour des références de manuels plus complets.

L'algèbre linéaire Le logiciel possède plusieurs bibliothèques d'algèbre linéaire. Nous avons choisi d'utiliser *linalg* la plus simple d'entre-elles.

On charge la bibliothèque. La liste des commandes contenues dans la bibliothèque apparaît. Nous ne parlerons que d'une toute petite partie d'entre-elle. L'aide du logiciel vous permettra de découvrir les autres.

```
> with(linalg);
```

```
Warning,the protected names norm and trace have been redefined
and unprotected
```

$[BlockDiagonal, GramSchmidt, JordanBlock, LUdecomp, QRdecomp, Wronskian, addcol, addrow, adj, adjoint, angle, augment, backsub, band, basis, bezout, blockmatrix, charmat, charpoly, cholesky, col, coldim, colspace, colspan, companion, concat, cond, copyinto, crossprod, curl, definite, delcols, delrows, det, diag, diverge, dotprod, eigenvals, eigenvalues, eigenvectors, eigenvects, entermatrix, equal, exponential, extend, ffgausselim, fibonacci, forwardsub, frobenius, gausselim, gaussjord, geneqns, genmatrix, grad, hadamard, hermite, hessian, hilbert, htranspose, ihermite, indexfunc, innerprod, intbasis, inverse, ismith, issimilar, iszero, jacobian, jordan, kernel, laplacian, leastsqrs, linsolve, matadd, matrix, minor, minpoly, mulcol, mulrow, multiply, norm, normalize, nullspace, orthog, permanent, pivot, potential, randmatrix, randvector, rank, ratform, row, rowdim, rowspace, rowspan, rref, scalarmul, singularvals, smith, stackmatrix, submatrix, subvector, sumbasis, swapcol, swaprow, sylvester, toeplitz, trace, transpose, vandermonde, vecpotent, vectdim, vector, wronskian]$

Il y a différentes façons de coder une matrice. Nous présentons différentes manières d'utiliser la fonction `matrix`. La première consiste à écrire en argument la liste des lignes de la matrices elles-aussi codées sous forme de liste :

```
> A:= matrix([[1,2,3],[4,5,6],[7,8,9]]);
```

$$A := \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

On peut aussi donner en argument la dimension de la matrice et la liste des coefficients dans l'ordre de lecture de l'écriture latine :

```
> A:=matrix(3,3,[1,2,3,4,5,6,7,8,9]);
```

$$A := \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

Lorsqu'on remplace la liste des coefficients par une fonction à deux variables f, on obtient :

```
> A:= matrix(3,3,f);
```

$$\begin{bmatrix} f(1,1) & f(1,2) & f(1,3) \\ f(2,1) & f(2,2) & f(2,3) \\ f(3,1) & f(3,2) & f(3,3) \end{bmatrix}$$

Cela a son intérêt lorsque les coeffients $a_{i,j}$ de la matrice sont donnés en fonction des indices i et j. Dans notre cas, nous pouvons définir une fonction à deux variables h :

```
> h:=(i,j)->3*(i-1)+j;
```

$$h := (i,j) \mapsto 3i - 3 + j$$

La fonction h est définie pour donner :

```
> A:= matrix(3,3,h);
```

$$A := \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

Enfin, la fonction `diag` permet de construire une matrice diagonale (même diagonale par blocs) à partir de la séquence de ses éléments diagonaux. C'est par exemple très utile pour définir la matrice identité !

```
> J:=diag(1,2,3);
```

$$J := \begin{bmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 3 \end{bmatrix}$$

La matrice `A` définie, on peut obtenir son (i,j)-ème coefficient en tapant `A[i,j]` :

```
> A[1,1];
```

$$1$$

```
> A[3,2];
```

$$8$$

Il est également possible d'extraire des sous-matrice d'une matrice à l'aide de la commande suivante :

```
> submatrix(A,1..2,2..3);
```

$$\begin{bmatrix} 2 & 3 \\ 5 & 6 \end{bmatrix}$$

On définit les vecteurs comme suit :

```
> V:= vector([1,2,3]);
```

$$V := [1,2,3]$$

Attention, bien que les vecteurs s'affichent comme des listes, il s'agit d'une structure différente. Comme on le voit :

```
> whattype(vector([1,2,3]));
```

$$array$$

```
> whattype([1,2,3]);
```

$$list$$

Remarquons la bizarrerie suivante. Lorsque nous entrons `A` et `V`, bien que ces lettres soient affectées, le logiciel ne semble pas les reconnaître.

```
> A;
```

$$A$$

```
> V;
```

$$V$$

Pour que le logiciel affiche la matrice ou le vecteur, il faut impérativement utiliser la commande `evalm` :

```
> evalm(A);
```

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

```
> evalm(V);
```

$$[1, 2, 3]$$

La commande `matrix` permet également de définir une « matrice inconnue » de dimension fixée :

```
> M:=matrix(3,3);
```

$$M := array(1..3, 1..3, [])$$

```
> evalm(M);
```

$$\begin{bmatrix} M_{1,1} & M_{1,2} & M_{1,3} \\ M_{2,1} & M_{2,2} & M_{2,3} \\ M_{3,1} & M_{3,2} & M_{3,3} \end{bmatrix}$$

Sommes, produits, produit par un scalaire, inversion

Introduisons une seconde matrice. La somme et le produit par un scalaire se fait avec les commandes usuelles `+` et `*` :

```
> B:=matrix(3,3,[2,1,-1,3,3,-4,3,1,-2]);
```

$$B := \begin{bmatrix} 2 & 1 & -1 \\ 3 & 3 & -4 \\ 3 & 1 & -2 \end{bmatrix}$$

```
> A+B;
```

$$A + B$$

```
> 3*B;
```

$$3B$$

Comme précédemment pour que les calculs soient effectués, il faut utiliser `evalm`.

```
> evalm(A+B);
```

$$\begin{bmatrix} 3 & 3 & 2 \\ 7 & 8 & 2 \\ 10 & 9 & 7 \end{bmatrix}$$

```
> evalm(3*B);
```

$$\begin{bmatrix} 6 & 3 & -3 \\ 9 & 9 & -12 \\ 9 & 3 & -6 \end{bmatrix}$$

Les commandes de puissance usuelles fonctionnent également :

```
> A^2; A^(-1) ;
```

$$A^2$$
$$A^{-1}$$

Evidemment, il faut évaluer et que la matrice soit inversible pour les puissances négatives :

```
> evalm(A^2);
```

$$\begin{bmatrix} 30 & 36 & 42 \\ 66 & 81 & 96 \\ 102 & 126 & 150 \end{bmatrix}$$

```
> evalm(A^(-1));
```

```
Error, (in inverse) singular matrix
```

```
> evalm(B^(-1));
```

$$\begin{bmatrix} \frac{1}{2} & \frac{-1}{4} & \frac{1}{4} \\ \frac{3}{2} & \frac{1}{4} & \frac{-5}{4} \\ \frac{3}{2} & \frac{-1}{4} & \frac{-3}{4} \end{bmatrix}$$

La multiplication des matrices s'effectue à l'aide du symbole `&*` :

```
> evalm(A&*B);
```

$$\begin{bmatrix} 17 & 10 & -15 \\ 41 & 25 & -36 \\ 65 & 40 & -57 \end{bmatrix}$$

Les règles usuelles de calcul dans l'anneau des matrices sont comprises par le logiciel :

```
> evalm(A^2- 2*(A-B)&*B );
```

$$\begin{bmatrix} 4 & 24 & 64 \\ -10 & 47 & 154 \\ -22 & 54 & 258 \end{bmatrix}$$

Nous citons encore les commandes `multiply` et `inverse` qui permettent également de multiplier et d'inverser sans avoir à évaluer en plus.

```
> multiply(A,B);
```

$$\begin{bmatrix} 17 & 10 & -15 \\ 41 & 25 & -36 \\ 65 & 40 & -57 \end{bmatrix}$$

```
> inverse(A);
```

```
Error, (in inverse) singular matrix
```

```
> inverse(B);
```

$$\begin{bmatrix} \frac{1}{2} & \frac{-1}{4} & \frac{1}{4} \\ \frac{3}{2} & \frac{1}{4} & \frac{-5}{4} \\ \frac{3}{2} & \frac{-1}{4} & \frac{-3}{4} \end{bmatrix}$$

Les noms des trois commandes utilisées ci-après parlent d'eux-mêmes.

```
> transpose(A);
```

$$\begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix}$$

```
> det(A);
```

$$0$$

```
> trace(A);
```

$$15$$

Résolution de système, noyau, image

Considérons le système linéaire donnée par $AX = V$ où $X \in \mathbb{R}^3$. Le logiciel la résout à l'aide de la commande `linsolve`.

```
> linsolve(A,V);
```

$$[-\tfrac{1}{3} + \_t_1, \tfrac{2}{3} - 2\_t_1, \_t_1]$$

La solution dépend d'un paramètre que le logiciel note $\_t_1$. Ce résultat n'est pas étonnant. Pour mieux comprendre le résultat nous pouvons demander de faire un pivot de Gauss au système (A, V) à l'aide de la commande `gausselim`. Dans un premier temps, on fabrique une matrice (A, V) en concaténant la matrice et le vecteur.

```
> U:=concat(A,V);
```

$$U := \begin{bmatrix} 1 & 2 & 3 & 1 \\ 4 & 5 & 6 & 2 \\ 7 & 8 & 9 & 3 \end{bmatrix}$$

```
> gausselim(U);
```

$$\begin{bmatrix} 1 & 2 & 3 & 1 \\ 0 & -3 & -6 & -2 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

La matrice obtenue est échelonnée et on constate sans peine que la solution proposée plus haut convient. Il y a un degré de liberté dans les solutions

car la dimension du noyau de A est 1. En effet, nous avons déjà vu que le déterminant de A est nul. la fonction `rank` calcule son rang :

```
> rank(A);
```

$$2$$

Le noyau de la matrice est calculé à l'aide de la fonction `kernel`. Celle-ci rend un ensemble de vecteurs constituant une base du noyau. Ici, le nombre de vecteurs est bien 1, la dimension attendue après le calcul du rang.

```
> kernel(A);
```

$$\{[1, -2, 1]\}$$

L'image de l'application associée à la matrice est l'espace engendré par ses colonnes. La commande `colspace` en donne une base :

```
> colspace(A);
```

$$\{[1, 0, -1], [0, 1, 2]\}$$

De même `rowspace` donne une base de l'espace engendré par les lignes de la matrice :

```
> rowspace(A);
```

$$\{[1, 0, -1], [0, 1, 2]\}$$

La matrice B est de déterminant non nul et donc son noyau sera réduit à $\{0\}$ et son image égale à l'espace $\mathbb{R}^3$. Jugez de la cohérence des calculs suivant :

```
> det(B);
```

$$-4$$

```
> kernel(B); colspace(B);
```

$$\{\}$$
$$\{[1, 0, 0], [0, 1, 0], [0, 0, 1]\}$$

Diagonalisation

Nous disposons de tous les outils pour étudier les éléments propres de la matrice A. Par exemple pour calculer ses valeurs propres il suffit de construire la matrice identité I et de calculer les racines de $\det(A - xI)$.

```
> Id:=diag(1,1,1);
```

$$Id := \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

```
> P:= factor(det(A-x*Id) );
```

$$P := -x\,(-18 - 15x + x^2)$$

```
> solve(P,x);
```

$$0, \frac{15}{2} + \frac{3}{2}\sqrt{33}, \frac{15}{2} - \frac{3}{2}\sqrt{33}$$

Néanmoins, Maple possède une commande `charploy` calculant directement le polynôme caractéristique, qu'il suffit de factoriser.

```
> charpoly(A,x);
```

$$-18x - 15x^2 + x^3$$

Néanmoins encore , Maple possède une commande `eigenvalues` calculant directement une séquence des valeurs propres.

```
> E:=[eigenvalues(A)];
```

$$E := \left[0, \frac{15}{2} + \frac{3}{2}\sqrt{33}, \frac{15}{2} - \frac{3}{2}\sqrt{33}\right]$$

Pour chaque valeur propre λ , il reste à déterminer le noyau de $A - \lambda I$.

```
> kernel(A-E[2]* Id);
```

$$\left\{\left[\frac{19}{4} - \frac{3}{4}\sqrt{33}, 1, -\frac{11}{4} + \frac{3}{4}\sqrt{33}\right]\right\}$$

Néanmoins toujours , Maple possède une commande `eigenvectors` calculant directement les valeurs propres et une base des espaces propres :

```
> eigenvectors(A);
```

$$[0, 1, \{[1, -2, 1]\}], \left[\tfrac{15}{2} + \tfrac{3}{2}\sqrt{33}, 1, \left\{[-1/2 + \tfrac{3}{22}\sqrt{33}, 1/4 + \tfrac{3}{44}\sqrt{33}, 1]\right\}\right],$$
$$\left[\tfrac{15}{2} - \tfrac{3}{2}\sqrt{33}, 1, \left\{[-\tfrac{1}{2} - \tfrac{3}{22}\sqrt{33}, \tfrac{1}{4} - \tfrac{3}{44}\sqrt{33}, 1]\right\}\right]$$

Le résultat est une séquence d'objets du type `[`λ`,`n`,{`$v_1, \ldots, v_k$`}]` où λ est une valeur propre, n sa multiplicité et $v_1, \ldots, v_k$ des vecteurs de base de son espace propre. Sur l'exemple précédent, chaque espace propre est de dimension 1, donc la matrice est diagonalisable. La nouvelle base est donnée par les vecteurs propres. Néanmoins, il est plus facile d'utiliser la commande `jordan` qui lorsque la matrice est diagonalisable, rend une matrice canonique de Jordan associée :

```
> jordan(A);
```

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & \frac{15}{2} - \frac{3}{2}\sqrt{33} & 0 \\ 0 & 0 & \frac{15}{2} + \frac{3}{2}\sqrt{33} \end{bmatrix}$$

De plus, si en second argument de `jordan`, on place un nom, celui-ci sera affecté avec la matrice de passage vers la nouvelle base :

```
> J:= jordan(A,'P');
```

$$J := \begin{bmatrix} 0 & 0 & 0 \\ 0 & \frac{15}{2} - \frac{3}{2}\sqrt{33} & 0 \\ 0 & 0 & \frac{15}{2} + \frac{3}{2}\sqrt{33} \end{bmatrix}$$

```
> evalm(P);
```

$$\begin{bmatrix} \frac{1}{6} & \frac{7}{132}\sqrt{33} + \frac{5}{12} & -\frac{7}{132}\sqrt{33} + \frac{5}{12} \\ -\frac{1}{3} & , -\frac{1}{66}\sqrt{33} + \frac{1}{6} & \frac{1}{66}\sqrt{33} + \frac{1}{6} \\ \frac{1}{6} & -\frac{1}{12}\sqrt{33} - \frac{1}{12} & \frac{1}{12}\sqrt{33} - \frac{1}{12} \end{bmatrix}$$

Un exercice classique consiste à diagonaliser pour calculer la puissance n-ième de la matrice A. Celle ci-sera donné par :

```
> M:= factor(evalm(P&*diag(seq(J[i,i]^n,i=1..3))&*P^(-1)));
```

$$M := [[\left(\tfrac{7}{132}\sqrt{33} + \tfrac{5}{12}\right)\left(15/2 - 3/2\sqrt{33}\right)^n + \left(-\tfrac{7}{132}\sqrt{33} + \tfrac{5}{12}\right)\left(15/2 + 3/2\sqrt{33}\right)^n$$
$$\tfrac{1}{132}\left(\tfrac{7}{132}\sqrt{33} + \tfrac{5}{12}\right)\left(15/2 - 3/2\sqrt{33}\right)^n\left(3\sqrt{33} - 11\right)\sqrt{33} + \tfrac{1}{132}\left(-\tfrac{7}{132}\sqrt{33} + \tfrac{5}{12}\right)\left(15/2 + 3/2\sqrt{33}\right)^n\left(3\sqrt{33} + 11\right)\sqrt{33}$$
$$\tfrac{1}{66}\left(\tfrac{7}{132}\sqrt{33} + \tfrac{5}{12}\right)\left(15/2 - 3/2\sqrt{33}\right)^n\left(\sqrt{33} - 11\right)\sqrt{33} + \tfrac{1}{66}\left(-\tfrac{7}{132}\sqrt{33} + \tfrac{5}{12}\right)\left(15/2 + 3/2\sqrt{33}\right)^n\left(\sqrt{33} + 11\right)\sqrt{33}]$$
$$[\left(-\tfrac{1}{66}\sqrt{33} + 1/6\right)\left(15/2 - 3/2\sqrt{33}\right)^n + \left(\tfrac{1}{66}\sqrt{33} + 1/6\right)\left(15/2 + 3/2\sqrt{33}\right)^n$$
$$\tfrac{1}{132}\left(-\tfrac{1}{66}\sqrt{33} + 1/6\right)\left(15/2 - 3/2\sqrt{33}\right)^n\left(3\sqrt{33} - 11\right)\sqrt{33} + \tfrac{1}{132}\left(\tfrac{1}{66}\sqrt{33} + 1/6\right)\left(15/2 + 3/2\sqrt{33}\right)^n\left(3\sqrt{33} + 11\right)\sqrt{33},$$
$$\tfrac{1}{66}\left(-\tfrac{1}{66}\sqrt{33} + 1/6\right)\left(15/2 - 3/2\sqrt{33}\right)^n\left(\sqrt{33} - 11\right)\sqrt{33} + \tfrac{1}{66}\left(\tfrac{1}{66}\sqrt{33} + 1/6\right)\left(15/2 + 3/2\sqrt{33}\right)^n\left(\sqrt{33} + 11\right)\sqrt{33}],$$
$$[\left(-1/12\sqrt{33} - 1/12\right)\left(15/2 - 3/2\sqrt{33}\right)^n + \left(1/12\sqrt{33} - 1/12\right)\left(15/2 + 3/2\sqrt{33}\right)^n,$$
$$\tfrac{1}{132}\left(-1/12\sqrt{33} - 1/12\right)\left(15/2 - 3/2\sqrt{33}\right)^n\left(3\sqrt{33} - 11\right)\sqrt{33} + \tfrac{1}{132}\left(1/12\sqrt{33} - 1/12\right)\left(15/2 + 3/2\sqrt{33}\right)^n\left(3\sqrt{33} + 11\right)\sqrt{33},$$
$$\tfrac{1}{66}\left(-1/12\sqrt{33} - 1/12\right)\left(15/2 - 3/2\sqrt{33}\right)^n\left(\sqrt{33} - 11\right)\sqrt{33} + \tfrac{1}{66}\left(1/12\sqrt{33} - 1/12\right)\left(15/2 + 3/2\sqrt{33}\right)^n\left(\sqrt{33} + 11\right)\sqrt{33}]]$$

On remarque, que lorsque la matrice est « trop grosse » pour être affichée sous forme de tableau, le logiciel l'écrit sous forme de liste de lignes. On peut vérifier que cette formule est valable pour $n = 1$:

```
> simplify(subs(n=1,evalm(M)) );
```

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

```
> eigenvalues(B);
```

$$-1, 2, 2$$

Si l'on étudie les éléments propres de B, on constate que la matrice n'est pas diagonalisable. En effet, elle a une valeur propre double et comme on le voit ci-dessous, l'espace propre associé à celle-ci est de dimension 1 seulement.

```
> eigenvectors(B);
```

$$[-1, 1, \{[0, 1, 1]\}], [2, 2, \{[1, 1, 1]\}]$$

Il est possible de calculer sa forme canonique de Jordan associée :

```
> jordan(B);
```

$$\begin{bmatrix} -1 & 0 & 0 \\ 0 & 2 & 1 \\ 0 & 0 & 2 \end{bmatrix}$$