Le but de cette première feuille est de se familiariser avec les commandes de base de Maple. Maple est un système de calcul formel, i.e. un logiciel qui permet de faire des mathématiques en manipulant des expressions symboliques. Contrairement à la plupart des langages classiques de programmation, il peut traiter non seulement des quantités numériques (entières, réelles, complexes) mais aussi des polynômes, des fonctions, des séries,...et effectuer des opérations courantes : dérivation, intégration, limites, simplifications,...

De plus, Maple permet aussi de faire les calculs numériques classiques : solution d'équations ou de systèmes, résolution d'équations différentielles,...Dans toutes les versions actuelles des capacités graphiques évoluées autorisent aussi de faire des représentations graphiques ainsi que d'aborder une partie des questions de géométrie plane ou dans l'espace.

A tout moment vous pouvez avoir plus d'informations sur une fonction *fonc* en tapant :

```
> ?fonc;
```

Exercice 1 : Premiers pas

• Tapez successivement les commandes :

```
> 12+58;
> 12+59:
```

Question : expliquez la différence entre le " :" et le ";". Une expression Maple est complète si elle se termine par ";" ou " :". Si vous envoyez à l'évaluateur une expression incomplète, Maple se contente de réafficher l'invite et d'attendre la suite de l'expression jusqu'à ce qu'il rencontre ";" ou " :".

• Expressions arithmétiques.

Avec Maple on peut évidemment utiliser toutes les opérations élémentaires :

```
> (8^3+5*3)/12-27;
> 15!;
> I*I;           # nombre complexe
> sqrt(5);
> evalf(sqrt(5));
```

Les deux derniers calculs nous montrent que Maple fait des calculs exacts sauf si on lui demande la contraire avec la commande *evalf*.

Question : sachant que le nombre π s'écrit Pi, faites afficher 15 de ses décimales (on utilisera la fonction *Digits*).

• Variables et affectations.

L'affectation sert à associer un nom et une valeur. La syntaxe d'une affectation est nom :=expression. On peut ensuite utiliser les variables dans différents calculs. Exemple :

```
> a:=3;
> b:=sqrt(a);
> monindice:=1;
```

A la différence de bien d'autres langages informatiques, il n'est pas nécessaire d'avoir affecté une variable pour pouvoir l'utiliser dans des expressions. C'est cette possibilité de travailler symboliquement avec des noms de variables qui fait de Maple un logiciel de calcul formel.

```
> a:=m^2;
> sqrt(a);
> x:=3*y+5;
> y:=1;
> eval(x);
```

Dans ce dernier exemple, si on ne veut pas que y prenne la valeur 1, et que l'on veut évaluer x en $y = 1$, on peut faire :

```
> x:=3*y+5;
> eval(x,y=1);
```

Question : Mettez dans une variable x l'expression $\sin(\pi y)$ et évaluez la valeur exacte puis approchée de $\sin(\pi/4)$ et $\sin(3\pi/2)$.

Rem : pour désaffecter la valeur de x tapez x :='x'. Si on veut désaffecter toutes les variables, on pourra taper “restart”.

Exercice 2 : Manipulation d'expressions

• *factor.*

La commande *factor* factorise les expressions polynômiales. Exemple :

```
> factor(x^3+x^2);
```

• *expand.*

La commande *expand* développe les expressions. Exemple :

```
> expand(cos(3*t));
> expand((x-t)*(x+t));
```

• *simplify.*

La commande *simplify* simplifie les expressions. Exemple :

```
> simplify(exp(a+ln(b*exp(c))));
```

Exercice 3 : Structures de contrôle

• Structures répétitives

Boucle *for*

Pour répéter un nombre déterminé de fois une ou plusieurs commandes, on utilise la commande *for* dont voici la syntaxe :

```
for <indice> from <debut> to <fin> by <pas> do
  <commandes>
od
```

où <indice> est une variable, <debut> et <fin> les bornes de l'intervalle que doit parcourir la variable <indice> avec un pas <pas> . Exemple d'une boucle qui calcule le cube des nombres pairs compris entre 2 et 12.

```
> for k from 2 to 12 by 2 do
     print(k^3);
  od;
```

<u>Boucle *while*</u>

Pour répéter plusieurs fois une ou plusieurs commandes tant qu'une certaine condition est vérifiée, on utilise la commande *while* dont voici la syntaxe :

```
while <condition> do
 <commandes>
od
```

où <condition> est une expression booléenne. Exemple

```
> x:=0:
  while x<15 do
  x:=x+1:
  od:
```

• Structure conditionnelle (test logique) : *if*

Dans un programme, nous verrons que l'on a besoin de faire des tests.

```
 if <cond> then
<pr1>
else
<pr2>
 fi;
```

où <cond> est une expression dont l'évaluation est un booléen, <pr1> et <pr2> sont des suites d'instructions.

Exemple : Détermination de la valeur absolue d'un nombre x

```
if (x<0) then
x:=-x;
else
x:=x;
fi;
```

Exercice 4 : Les Procédures

Une procédure est un ensemble d'instructions regroupées qui effectue une tâche déterminée à partir de paramètres donnés. Concrètement une procédure sert à automatiser une suite d'instructions. La syntaxe est la suivante :

```
proc(<sequence_de_parametres>)
 local <sequence_de_noms> ;
 suite_d'instructions
end
```

où <*sequence_de_parametres*> est une suite de noms de variables à partir de laquelle sera calculé le résultat escompté et <*sequence_de_noms*> est une suite de noms de variables qui sera utilisé à l'intérieur de la procédure (et n'existe qu'à l'intérieur de celle-ci).

Le résultat renvoyé est le dernier résultat calculé par la procédure. Pour changer de ligne sans changer de section dans une feuille de calcul, appuyez simultanément sur " Shift "et " Return ".

Exemple d'une procédure qui calcule la somme des n premiers entiers :

```
> somme:=proc(n)
 local s,k;
 s:=0:
 for k from 1 to n do
 s:=s+k;
 od;
 print(s);
 end;

> somme(5);
```

Question : Ecrivez une procédure qui calcule les n premiers termes de la suite de Fibonacci définie par $T_0 = 0$, $T_1 = 1$ et $T_n = T_{n-1} + T_{n-2}$ pour $n > 1$. Etudiez le comportement de T_{n+1}/T_n quand n tend vers l'infini.

Exercice 5 : Les fonctions

On peut définir des fonctions telles que $f(x) = 3x^2 + 6x - 2$ par la commande :

```
> f:=x->3x^3+6*x-2;
```

On utilise f ensuite comme une fonction :

```
> f(2);
> f(a+b);
> expand(%);     # le signe % designe le dernier resultat caluclé
> (f(x+h)-f(x))/h;
> simplify(%);
```

Attention, les fonctions et expressions sont deux objets différents :

```
> f1:=x^2+6;     # ceci est une expression
> f2:=x-> x^2+6; # ceci est une fonction
> # et on ne les evalue pas de le meme maniere :
> eval(f1,x=3);
> f2(3);
```

mais on peut passer de l'un à l'autre :

```
> q:=f2(x);   # q est une expression
>
> p1:=x+1;    # p1 est une expression
> f:=unapply(p1,x); # f est une fonction
>
> f:=unapply(f2(x),x); # f est une fonction
```

On peut dériver une expression par rapport à une variable en utilisant la commande *diff*. Pour dériver une fonction on utilise l'opérateur *D*.

```
> g:=sin(x)+x;
> diff(g,x);
>
> f:=x->sin(x)+x;
> D(f)(x);
> (D@@2)(f);   # dérivée seconde
```

On peut également définir des fonctions à plusieurs variables :

```
> f:=(x,y)->x^2+y^2-5;
```

Exercice 6 : Les graphiques

Soit f une fonction d'une variable déjà définie par l'opérateur flèche. Pour la représenter sur l'intervalle $[a, b]$, on utilisera la syntaxe :

```
> plot(f(x),x=a..b);
```

Pour afficher plusieurs courbes sur le même graphique, on utilise la commande *display* et il est alors préférable d'attribuer une couleur différente pour chaque courbe (faire *?plot[options] ;* pour obtenir la liste des différentes couleurs et styles). Exemple, si $f1$ et $f2$ sont deux fonctions que l'on a précédemment définies :

```
> with(plots):   # on a besoin d'inclure le package plots pour utiliser display
  A:=plot(f1(x),x=-12..12,color=blue,style=point,symbol=cross,legend="courbe 1"):
  B:=plot(f2(x),x=-12..12,color=magenta,style=line,linestyle=3,legend="courbe 2"):
  display(A,B,title="Différentes courbes");
```

Si g est une fonction de deux variables, on la représentera sur le carré $[a, b] \times [c, d]$ par la syntaxe :

```
> plot3d(g(x,y),x=a..b,y=c..d);
```

Question : Représentez la surface de votre choix.

Exercice 7 : Les séquences et les listes

• Les séquences.

Une séquence est une suite d'objets séparés par des virgules. Pour créer une séquence on peut définir tous ses éléments :

```
> S:=1,6,8,3,89;
```

ou bien on utilise la commande *seq* :

```
> seq(i^2,i=0..1);
```

On accède ensuite au $i^{\text{ème}}$ élément de S par $S[i]$. Attention on ne peut pas modifier la $i^{\text{ème}}$ valeur de la séquence i.

• Les listes.

Une liste est une séquence entre crochets. Cette fois on peut modifier le $i^{\text{ème}}$ élément :

```
> l:=[2,3,7];
> s:=seq(i^2,i=0..6);  # s est une sequence
> l2:=[s];             # l2 est une liste
> l2[3]:=1;            # on modifie ici le troisieme élément de la liste l2
```

On vient de voir comment transformer une séquence en liste (passage de s à $l2$), on peut également créer une séquence à partir d'une liste par la commande *op* :

```
> S:=[1,5,8,9];
> op(S);
```

La fonction *map* permet d'appliquer une fonction à tous les éléments d'une liste :

```
> map(cos,[seq(i,i=1..3)]);
> map(x->x+1,[seq(i,i=1..3)]);
```

Rem : La différence fondamentale entre une liste et une séquence est que la première est une expression mais pas la deuxième. Par conséquent on pourra appliquer certaines fonctions à l'une et pas à l'autre.

Question : Expliquez chacune des instructions suivantes :

```
> with(plots):
> l:=[seq(i,i=0..10)];
> m:=map(x->x^2,l);
> coor:=zip((x,y)->[x,y],l,m);
> plot(coor,style=point,symbol=circle,symbolsize=15);
```

Question supplémentaire 1 : La suite de Syracuse. Ecrire une procédure qui teste la parité d'un nombre A, et calcule $A/2$ si A est pair et $3A+1$ si A est impair.

Question supplémentaire 2 : Racines d'un polynôme. Ecrire une procédure qui calcule les racines de $aX^2 + bX + c$.