TD Maple 2

Programmation

I - Avant de commencer

Quelques rappels avant de travailler. Commencez chaque nouvel exercice par un

```
restart;
```

pour « désaffecter » toutes les variables éventuellement utilisées dans un calcul précédent.
Si vous avez lancé un calcul qui a l'air de ne pas vouloir se terminer, cliquez sur l'icône STOP qui n'est active que lorsqu'un calcul est en cours.
Chacune de vos entrées débute après un *prompt* >, se termine par un **;**[a] ou un **:**[b] puis appuyez sur la touche Enter. Si vous voulez passer à la ligne sans valider la ligne précédente, tapez Shift ⇑ + Enter. Pour revenir plus haut, utilisez la souris.

II - Sept manières de calculer 7 !

1. Bien sûr, on peut commencer par faire

```
7!;
```

mais le calcul de 7! n'est qu'un prétexte pour découvrir sur un exemple simple la syntaxe de programmation MAPLE.

2. On pourrait alors taper

```
1*2*3*4*5*6*7;
```

mais les choses pourraient se compliquer au moment de calculer 3232!

3. Nous allons donc utiliser une boucle *for* avec indice numérique : nous partons de 1, puis nous multiplions par 2, puis nous multiplions le produit précédent par 3, etc. Cela donne

```
p:=1: for k from 1 to 7 by 1 do p:=p*k od;
```

Ce programme n'est pas optimum. En fait, dans une boucle *for*, les instructions *from* et *by* sont facultatives : elles valent par défaut 1. Ici, on peut se contenter de

```
p:=1 for k to 7 do p:=p*k od;
```

Il reste un inconvénient : MAPLE affiche tous les résultats intermédiaires. Rappelez-vous des rôles de **:** et **;** puis écrivez un programme qui n'affichera que la valeur finale.

4. Nous aurions pu créer la liste des entiers de 1 à 7 puis utiliser une boucle *for* indexée cette fois par les éléments de la liste. C'est plus compliqué, mais cela nous permet de découvrir comment MAPLE traite les listes. Il faudra bien distinguer

[a] si vous voulez que le résultat soit affiché.
[b] si vous voulez que MAPLE exécute sans afficher.

▷ les **ensembles** (*set*) qui sont des collections *non ordonnées* d'expressions toutes différentes séparées par des virgules et encadrées d'accolades.

```
ens1:={2,4,1}; ens2:={2,5,8,5};
```

On peut effectuer les opérations usuelles sur les ensembles

```
ens1 intersect ens2; ens1 union ens2; ens1 minus ens2;
```

l'ensemble vide se note

```
{ }
```

▷ les **suites** (*sequence*) qui sont des collections *ordonnées* d'expressions, différentes ou non, séparées par des virgules et encadrées ou non par des parenthèses.

```
5,7,5,1,2,3;
```

On peut aussi utiliser les opérateurs seq et $ pour des suites définies par une formule explicite

```
seq(k**2,k=1..5);
p**2 $ p=1..5;
```

```
m $ 5;
seq(i**2,i=1..n);      // problème...
j**2 $ j=1..n;          // plus de problème
```

On peut changer un élément de la suite

```
subs(n=5,'');       // remplacez '' par % dans les versions  ultérieures à MAPLE V.4
eval('');
```

▷ les **listes** (*list*) qui sont des collections *ordonnées* d'expressions séparées par des virgules et encadrées par des crochets. La différence, c'est qu'une suite, en tant que juxtaposition d'expressions, est en quelque sorte « en lecture seule », alors qu'une liste est une expression en elle-même et pourra donc « subir » des opérations algébriques.
Notez au passage quelques fonctions utiles

```
s1:=i $ i=-2..2; s2:=a,b,c,d,e;
L1:=[s1]; L2:=[s2];  // une liste est une suite entre crochets
nops(L2);     // nombre d'opérandes
L2[3]; op(3,L2);          // extrait le 3ème opérande
L2[3..5]; op(3..5,l2);
select (x->x>0,L1); remove(x->x<0,L1);
subs(b=32,L2);
subsop(2=z,L1); subsop(5=NULL,L2);   // pour substituer ou supprimer un opérande
map(cos,L2);
zip((x,y)->x+y,L1,L2);
```

▷ les **tableaux** (*array*) sont des objets plus complexes. Nous reviendrons plus tard dessus. Sachez seulement qu'une liste est un tableau d'une ligne

```
T:=array(1..10);   // T est un tableau d'un ligne et de 10 colonnes
T:=[k^2 $ k=1..10];   //  ... comme une liste
T[3];
```

Cela donne

```
L:=[k $ k=1..7]: p:=1: for i to 7 do p:=p*L[i] od:
p;
```

5. Nous pouvons utiliser une boucle *while*

```
p:=1: k:=1: while k<7 do k:=k+1: p:=p*k od:
p;
```

6. C'est bien beau, mais que ferons-nous quand il faudra calculer 32! ou 3232! et tous les autres ? Il faudrait créer un programme (on dira une **procédure**) qui donne $n!$ pour tout entier naturel n.

```
fact:=proc(n)
local p,k;  // nous aurons besoin de variables locales i.e. internes à la procédure
p:=1;
for k to n do p:=p*k od;
end:  // termine la procédure

fact(32);
```

7. Le meilleur pour la fin : la **procédure récursive**, qui s'appelle elle-même

```
factr:=proc(n)
if n=0 then 1 // une boucle $if...then...else$ pour régler le cas de 0!
else
n*factr(n-1) fi;  // symbolise la fin de la boucle
end:

factr(32);
```

En fait, ce mécanisme correspond à une suite numérique qui s'écrirait mathématiquement $u_n = n \times u_{n-1}$.

III - En guise d'échauffement...

Exercice 1 Partie entière

Déterminez une procédure `E :=proc(x)` qui, à un réel positif x, associe sa partie entière.

Exercice 2 Valeur absolue

Déterminez une procédure `ab :=proc(x)` qui, à un réel x, associe sa valeur absolue.

Exercice 3 Moyenne

Déterminez une procédure permettant de calculer la moyenne des éléments d'une famille de nombres réels.

Exercice 4 Somme

Vous savez peut-être que la suite $(S_n)_{n\in\mathbb{N}}$ de terme général

$$S_n = \sum_{k=0}^{n} \frac{1}{k!}$$

est croissante et converge vers e. Nous l'admettrons dans cet exercice.

1. Déterminez une procédure `S=proc(n)` qui, à un entier naturel n, associe S_n. N'oubliez pas les gages habituels : vous n'utiliserez ni la fonction prédéfinie `sum`, ni les procédures calulant $n!$ vues précédemment.

2. Déterminez une procédure `seuil :=proc(p)` qui, à un entier naturel p, associe le plus petit entier naturel n tel que $|S_n - e| \leqslant 10^{-p}$. Vous aurez besoin de savoir que e se dit `exp(1)` en MAPLE.

Exercice 5 Équation du second degré

Déterminez une procédure `sol :=proc(a,b,c)` qui, à une équation $ax^2 + bx + c = 0$, associe son ensemble des solutions.

Exercice 6 test sur une liste

Déterminez une procédure `test :=proc(L)`, `L` étant une liste d'entiers, qui teste si ses éléments forment une suite croissante.

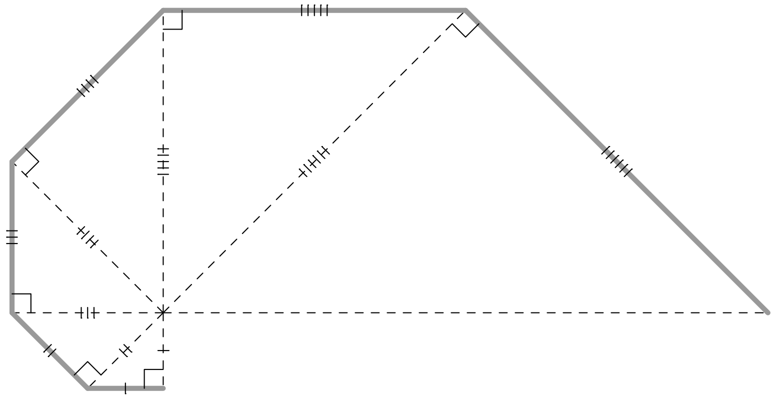
Exercice 7 Test d'appartenance

Créez une procédure pour tester l'appartenance d'un point dont on connaît les coordonnées à une courbe donnée.

IV - ♫ Dessinons en programmant . . . ♫

Exercice 8 Spirale

Comment dire à MAPLE de dessiner cette spirale ?



Pour ceux qui n'ont pas eu l'audace de choisir la spécialité maths l'année dernière, demandez à vos voisins ce qu'est une similitude.
Vous serez ensuite amenés à utiliser des complexes. Voici quelques clés utiles

```
z:=2+3*I;
Re(z);
Im(z);
Z_1:=z*exp(I*Pi/4);
Z_2:=Z_1*sqrt(13);
m:=[Re(z),Im(z)];
M_1:=[Re(Z_1),Im(Z_1)];
M_2:=[Re(Z_2),Im(Z_2)];
plot([m,M_1,M_2]);
```

Et voici un canevas pour produire la figure :

```
z:=array(1..7):  // tableau de longueur 7 pour construire les affixes
z[1]:=-I:
```

```
for i to 6 do ...
M:=array(1..7):   // tableau contenant les points
for j to 7 do ...
P:=[M[k] $ k=1..7]:
plot(P);
```

V - Les objets MAPLE - Étude des branches infinies

Pour vous, x est un réel, n un entier, $[1,2]$ un intervalle, etc. Pour MAPLE, cela n'a rien d'évident : il faut parfois tout lui expliquer. Pour illustrer notre propos, nous allons écrire un programme étudiant les branches infinies d'une fonction numérique. Un bref rappel de cours n'est peut-être pas inutile : schématisez-le sous forme d'un arbre.
Il va falloir utiliser pas mal de boucles if qui correspondent à chaque embranchement de l'arbre. Il va d'abord falloir demander à MAPLE de calculer des limites. Il sait faire :

```
limit(sin(x)/x,x=0); limit(1/sqrt(x),x=0);
limit(sin(x)/x,x=infinity); limit(sin(x),x=infinity);
```

Maintenant, il va falloir tester la réponse :a est réel ou infini ? La limite existe ou n'existe pas ? Pour cela, commençons par explorer la fonction `whattype(expression)`

```
whattype(32); whattype(1/2);
whattype(x->2*x); whattype(limit(sin(x),x=0));
```

```
whattype(limit(32*x,x=infinity)); whattype(limit(sin(x),x=infinity));
```

Ensuite, il existe une fonction qui teste si une expression est d'un type déterminé :`type(expression,domaine)`.

```
type(32,integer); type(32.1,integer);
type(limit(sin(x),x=infinity),infinity);
```

Nous allons maintenant construire une procédure `branche :=proc(f)` Qui renvoie l'éventuelle branche infinie de la fonction f. Elle contiendra deux variables locales a et b. Je vous livre le début

```
branche:=proc(f)}
local a,b;
a:=limit(f(x)/x,x=infinity):
if type(a,..)=true then lprint(`pas d'asymptote`);
elif type(a,infinity)=true then lprint(`pas d'asymptote`);
```

À vous d'imaginer la suite... Pour demander à MAPLE de envoyer un message mélangeant du texte et un résultat de calcul, on peut utiliser la commande

```
printf(`asymptote d'equation y=%f x+%f`,a,b);
```

où l'objet flottant `f` (c'est à dire le réel `f`) va prendre les valeurs de a et b.

Index des commandes vues aujourd'hui

Commandes