INTRODUCTION À LA PROGRAMMATION EN MAPLE

Bernard Dupont

Bernard.Dupont@univ-lille1.fr

Maple met à la disposition de ses utilisateurs plusieurs milliers de commandes pré-programmées stockées dans la bibliothèque principale et dans les paquetages spécialisés. Cette profusion a deux inconvénients. D'une part, un cerveau humain normalement constitué ne peut pas dominer la totalité des fonctionnalités proposées et il est souvent plus rapide de programmer ses propres besoins plutôt que de se mettre à la recherche d'une ressouce nichée au fin fond du logiciel. D'autre part, il est fréquent qu'on ressente le besoin de programmer ses propres commandes en fonction de besoins spécifiques pointus. C'est particulièrement le cas quand on se rend compte qu'on utilise fréquemment la même succession de commandes et qu'on voudrait automatiser cette tâche répétitive pour gagner du temps. La programmation a précisément pour objectif de créer des procédures automatisées de traitements de tâches. Elle est devenue de nos jours une discipline intellectuelle à part entière dont il faut bien comprendre les particularités, les exigences et les méthodes de travail. La section 1 commence par préciser son vocabulaire: algorithme, programme, procédure et module. La section 2 expose les deux principales structures utilisées dans les algorithmes : structures itératives et structures conditionnelles. La section 3 montre comment mettre au point une procédure Maple élémentaire.

Définitions

L'activité d'un programmeur consiste à automatiser une tâche rencontrée fréquemment en pratique, c'est à dire à demander à un ordinateur de faire avec justesse et rapidité des opérations à la place d'un être humain pour atteindre un résultat. Par exemple, le calcul de la variance d'une variable statistique discrète nécessite le calcul prélable de la moyenne, le calcul du carré de l'écart entre chaque valeur et la moyenne, sa pondération par la fréquence relative, enfin la somme des produits des fréquences par les écarts quadratiques. S'il n'y a que quelques modalités, c'est jouable "à la main". Au delà de 20 modalités, des erreurs de calcul sont excusables mais impardonnables et il vaut mieux se munir d'une calculatrice. Pour des centaines voire des milliers de modalités, l'automatisation devient un bienfait, surtout s'il faut calculer la variance de centaines de séries statistiques. Le besoin d'un programme générique se fait sentir ...

Il n'est pas nécessaire d'avoir constamment à portée de main un ordinateur pour rédiger un programme générique. Au départ, une feuille de papier suffit. En substance, il s'agit en effet de réfléchir à son contenu qui dépend bien entendu de l'objectif qu'on lui assigne. Ce travail de réflexion aboutit à un **algorithme** qui se présente formellement comme une succession de lignes d'instructions réunies en un ou plusieurs blocs structurés et dont la logique d'enchaînement permet d'atteindre toujours le but recherché. Dans l'exemple du calcul de la variance, il n'est pas difficile de décomposer les étapes de son calcul qui vont donner chacune naissance à un bloc d'instructions : un bloc pour calculer la moyenne; un second bloc pour calculer le carré des écarts à la moyenne; un troisième bloc pour calculer le produit des écarts à la moyenne par la fréquence relative; un dernier bloc pour sommer les résultats obtenus dans l'étape précédente. Ainsi, un algorithme décompose sur un mode purement logique une tâche en sous-tâches élémentaires. Tout bloc a une charpente qui se présente sous la forme d'une structure itérative, encore appelée structure for, ou d'une structure conditionnelle, encore appelée structure if.

Stricto sensu, un **programme informatique** est un algorithme écrit dans un langage de programmation. Il s'ensuit qu'un algorithme donné peut être transcris dans plusieurs langages : basic, cobol, fortran, java, C, etc..Chaque langage a ses particularités et nécessite bien entendu un

apprentissage pour que les idées contenues dans l'algorithme ne soient pas dénaturées. Maple ne fait pas exception à cette règle. Pour programmer en Maple, il faut connaître les particularités de sa syntaxe dont ce chapitre va donner les bases essentielles.

Un programme Maple peut servir une fois, auquel cas il reste ... un programme de circonstance dans la feuille de travail. Mais si l'utilisateur pressent ou a la certitude que ce programme va servir plusieurs fois, il a intérêt à le transformer en **procédure**. Une procédure Maple est un programme à qui un nom et d'éventuels arguments ont été attribués et qui se déclenche dès que l'utilisateur l'appelle par son nom et fournit les arguments nécessaires. Autrement dit, c'est une commande au même titre que les commandes pré-programmées avec cette différence que c'est l'utilisateur qui en est le créateur et avec cette limite qu'elle n'est disponible que dans la séance en cours.

Certaines procédures ont une portée universelle aux yeux de leur auteur qui souhaite alors les invoquer dans ses futures feuilles de travail. Il peut alors les insérer dans un **module** qui est conservé par Maple en tant que paquetage localisé - au sens où il est résident de l'ordinateur du créateur mais n'existe pas à l'extérieur - et devient alors exploitable par simple chargement dans n'importe quel worksheet. Généralement, un module correspond à une thématique précise et regroupe de ce fait plusieurs procédures relevant de cette thématique. On peut bien entendu élargir son public par voie de diffusion.

Structures itératives et structures conditionnelles

On a vu que tout programme informatique repose sur un algorithme qui déroule des blocs d'instructions. Deux catégories de blocs élémentaires se dégagent dans la pratique.

Une première structure très fréquemment rencontrée se réfère à la notion de boucle. On la rencontre si une même tâche doit être effectuée un nombre fini donné de fois ou encore un nombre fini inconnu de fois qui dépend d'une condition. On parle alors de *structure itérative* ou *structure for*.

Une seconde structure, tout aussi nécessaire que la précédente, aiguillonne le déroulement du programme suivant le résultat d'un test. Par exemple, un économiste demandera que seules les solutions réelles positives d'une équation soient retenues, et non toutes les solutions complexes, de sorte que si une procédure doit résoudre une équation "économique", il faut introduire une condition supplémentaire et effectuer le test correspondant. Le programme contiendra alors une *structure conditionnelle*, appelée aussi *structure if.*

Structures itératives ou structures for

Une structure itérative est une boucle qui effectue une action un certain nombre de fois fixé à l'avance par l'utilisateur ou un certain nombre de fois inconnu a priori mais dépendant de la réalisation d'une condition. Deux syntaxes sont proposées par Maple suivant que la variable-compteur prend ses valeurs dans un ensemble de nombres ou dans une séquence ou une liste préexistante.

La variable-compteur décrit un ensemble de nombres

Si la variable servant de compteur a une valeur de départ et une valeur d'arrivée numérique, il faut utiliser la syntaxe générale suivante :

```
for nom_de_la_variable_compteur from valeur_de_départ by
valeur_du_pas to valeur_d'arrivée while condition_test do
          bloc d'instructions
end do;
```

L'ordre des termes est impératif. Chaque terme doit être bien compris, d'autant que certains sont obligatoires et d'autres optionnels.

Le premier terme **for** est obligatoire. Il signale qu'on rentre dans une boucle. Chaque

"round" est comptabilisé dans une variable-compteur dont il faut obligatoirement donner le nom (par exemple **i** ou **k**).

Le second terme **from** est optionnel. Si la variable compteur doit démarrer à la valeur 1, from est inutile (la valeur de départ est fixée à 1 par défaut). Dans tous les autres cas, il faut signaler après **from** la valeur de départ (par exemple, **from 0** ou **from 15**).

Le troisième terme **by** définit le pas d'une boucle. Il est optionnel. Le pas étant fixé par défaut à 1, l'utilisateur n'a pas à préciser la valeur du pas dans ce cas précis. Mais si le pas est différent de 1, il faut le signaler avec un **by** suivi d'un nombre égal à la longueur du pas (par exemple, **by 0.35**).

Les termes **to** et **while** sont là pour donner la condition de sortie de la boucle. La valeur d'arrivée du compteur est a priori infinie pour Maple, de sorte qu'il faut impérativement indiquer une valeur butoir au compteur. On utilise **to** pour arrêter le processus à cette valeur précise (par exemple, **to 25**). Alternativement, on utilise **while** (en français : tant que) pour arrêter le processus quand une condition logique est satisfaite (par exemple **while i<25**).

Le dernier terme **do** est absolument obligatoire, de même que l'instruction de clôture **end do**; (ne pas oublier le point-virgule).

Entre ces deux derniers termes se situe le bloc d'instructions qui doit être fait dans chaque "round".

Les exemples élémentaires suivants déclinent les possibilités offertes par ce type de structure itérative.

Exemple 1 : Affichage de nombres

On veut afficher les 5 premiers entiers naturels pairs, 0 étant considéré comme pair. La commande d'affichage est **print**. Le compteur démarre à 0. Le pas est 2. La valeur d'arrêt est 8.

```
> for i from 0 by 2 to 8 do
  print(i)
  end do;
```

0
2
4
6
8 **(2.1.1.1)**

On veut maintenant afficher les 5 premiers nombres impairs. Comme on démarre à 1, le terme **from** est superflu.

```
> for i by 2 to 9 do
  print(i)
  end do;
```

1
3
5
7
9 **(2.1.1.2)**

On veut maintenant afficher tous les nombres impairs compris entre 7 et 15. Le compteur

démarre en 7 et s'arrête à la valeur 15. Le pas est encore de 2.

```
> for i from 7 by 2 to 15 do
  print(i)
  end do;
```

7
9
11
13
15 **(2.1.1.3)**

Pour afficher tous les entiers dont le carré est inférieur ou égal à un nombre donné, ici 49, on met en oeuvre une structure itérative avec **for** et **while** :

```
> for i while i^2<=49 do
  print(i)
  end do;
```

1
2
3
4
5
6
7 **(2.1.1.4)**

Exemple 2 : Modification des composantes d'un vecteur

Supposons qu'on ait besoin de remplacer les composantes d'un vecteur en prenant leur carré.

```
> V:=Vector(6,[2.3,3.4,4.5,5.6,6.7,7.8]);#création du
  vecteur
  for i to 6 do
  V[i]:=V[1]^2
  end do:#les "deux points" empêchent l'affichage des
  calculs
  V;#affichage du vecteur transformé
```

$$V := \begin{bmatrix} 2.3 \\ 3.4 \\ 4.5 \\ 5.6 \\ 6.7 \\ 7.8 \end{bmatrix}$$

(2.1.1.5)

$$\begin{bmatrix} 5.29 \\ 27.9841 \\ 27.9841 \\ 27.9841 \\ 27.9841 \\ 27.9841 \end{bmatrix} \quad \textbf{(2.1.1.5)}$$

Quand on travaille sur une liste, une séquence ou un vecteur dont le nombre de composantes est inconnu, on fixe la valeur d'arrivée du compteur par **nops**.

```
> V:=[0.1,1.2,2.3,3.4,4.5,5.6,6.7,7.8,8.9];
  for i to nops(V) do
  V[i]:=V[i]^2
  end do:
  V;
```

$$V := [0.1, 1.2, 2.3, 3.4, 4.5, 5.6, 6.7, 7.8, 8.9]$$

$$[0.01, 1.44, 5.29, 11.56, 20.25, 31.36, 44.89, 60.84, 79.21] \quad \textbf{(2.1.1.6)}$$

Exemple 3 : Composantes d'une matrice

Les éléments d'une matrice se situent au croisement d'une ligne et d'une colonne. Remplir ou modifier une matrice se fait en imbriquant deux boucles **for**. La première variable-compteur décrit le nombre de lignes et la seconde le nombre de colonnes. Ainsi, pour former une matrice dont les éléments m_{ij} valent $(-1)^{i+j}(i+j)$, on fera :

```
> M:=Matrix(5,4);#création d'une matrice à 5 lignes et 4
  colonnes
  for i to 5 do#la variable i décrit le nombre de lignes
       for j to 4 do;#la variable j décrit le nombre de
  colonnes
       M[i,j]:=(-1)^(i+j)*(i+j)
       end do;#fin de la boucle imbriquée
  end do;#fin de la boucle
  M;
```

$$M := \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

(2.1.1.7)

$$\begin{bmatrix} 2 & -3 & 4 & -5 \\ -3 & 4 & -5 & 6 \\ 4 & -5 & 6 & -7 \\ -5 & 6 & -7 & 8 \\ 6 & -7 & 8 & -9 \end{bmatrix} \quad \textbf{(2.1.1.7)}$$

Conformément à un usage établi et parfaitement justifié, la lecture du programme est rendue plus aisée en indentant les instructions relatives à la deuxième boucle par rapport aux instructions relatives à la première boucle.

La variable-compteur prend ses valeurs dans une liste ou une séquence

Si la variable-compteur doit prendre les valeurs successives d'une liste ou d'une séquence, qu'on notera ici **L**, la syntaxe est la suivante :

```
for nom_de_la_variable_compteur in L while condition_test do
          bloc d'instructions
end do;
```

Ici, seul le terme **while** est optionnel. Tous les autres termes et instructions sont obligatoires.

Dans cet exemple, on construit la table des cosinus d'angles rassemblés dans la liste **S**.

```
> S:=[0,Pi/6,Pi/4,Pi/3,Pi/2,Pi,7*Pi/6,5*Pi/4,4*Pi/3,3*Pi/2,
  2*Pi];
  for i in S do
  'cos'(i)=cos(i)
  end do;
```

$$S := \left[0, \frac{1}{6}\pi, \frac{1}{4}\pi, \frac{1}{3}\pi, \frac{1}{2}\pi, \pi, \frac{7}{6}\pi, \frac{5}{4}\pi, \frac{4}{3}\pi, \frac{3}{2}\pi, 2\pi\right]$$

$$\cos(0) = 1$$

$$\cos\left(\frac{1}{6}\pi\right) = \frac{1}{2}\sqrt{3}$$

$$\cos\left(\frac{1}{4}\pi\right) = \frac{1}{2}\sqrt{2}$$

$$\cos\left(\frac{1}{3}\pi\right) = \frac{1}{2}$$

$$\cos\left(\frac{1}{2}\pi\right) = 0$$

$$\cos(\pi) = -1$$

$$\cos\left(\frac{7}{6}\pi\right) = -\frac{1}{2}\sqrt{3}$$

$$\cos\left(\frac{5}{4}\pi\right) = -\frac{1}{2}\sqrt{2}$$

$$\cos\left(\frac{4}{3}\pi\right) = -\frac{1}{2}$$

$$\cos\left(\frac{3}{2}\pi\right) = 0$$

(2.1.2.1)

$$\cos(2\pi) = 1 \quad \textbf{(2.1.2.1)}$$

Structures conditionnelles ou structures if

Formellement, toute structure conditionnelle s'insère dans une boucle commençant par **if** et se terminant par **end if;** (noter la présence du point-virgule). Sa syntaxe générale est la suivante :

```
if condition_1 then instruction_1
elif condition_2 then instruction_2
elif condition_3 then instruction_3
.................................
elif condition_N then instruction_N
else instruction_(N+1)
end if;
```

Maple exécute linéairement une telle boucle. Il commence par tester la `condition_1`: si (**if**) elle est vraie, alors (**then**) Maple exécute `l'instruction_1` puis sort de la boucle; si elle n'est pas vraie, alors Maple passe à la `condition_2` qui commence par **elif** (pour : mais si). Si cette dernière est vraie, alors (**then**) il exécute l'`instruction_2` puis sort de la boucle, mais si elle n'est pas vraie, il passe à la troisième condition qui commence aussi par **elif**; etc.. Si laucune des N premières conditions n'est vraie, l'`instruction_(N+1)` précédée par **else** (sinon) est exécutée et Maple sort de la boucle.

Les tests sont des évaluations booléennes de conditions au sein d'une logique ternaire: les conditions sont donc toujours des énoncés dont on peut dire qu'ils sont vrais (true) ou faux (false) ou indécidables (FAIL). Une condition peut être un énoncé singulier ou être formée d'énoncés reliés par des connecteurs propositionnels tels que **and** (et) et **or** (ou). Si un énoncé n'est pas vrai, cela signifie qu'il est faux ou indécidable.

Seule la première et la dernière ligne sont obligatoires. On a alors une structure conditionnelle minimale : si le test est concluant, l'instruction correspondante est exécutée; sinon, rien n'est fait. Assez souvent, la boucle comprend la première ligne et les deux dernières car la structure conditionnelle porte sur une alternative (la `condition_1` est ou n'est pas respectée) mais dans chaque cas une action doit être entreprise.

Un premier exemple, assez artificiel, montre comment changer la valeur prise par une fonction en un point! Le critère est le suivant : si l'image est positive ou nulle, sa valeur est conservée; si elle est négative, elle doit prendre la valeur 0. Une seule condition suffit dans ce cas.

```
> P:=x->x^3-4*x^2+5;#définition de la fonction P
  'P(1)'=P(1);#cas d'une image positive
  'P(-2)'=P(-2);#cas d'une image négative
  if P(-2)<0 then P(-2):=0#test sur la positivité de P(-2)
  end if:
  'P(-2)'=P(-2);#la valeur de P(-2) est modifiée
  if P(1)<0 then P(1):=0#test sur la positivité de P(1)
  end if:
  'P(1)'=P(1);#la valeur de P(1) est conservée
```

$$P := x \to x^3 - 4\,x^2 + 5$$

$$P(1) = 2$$

$$P(-2) = -19$$
$$P(-2) = 0$$
$$P(1) = 2 \qquad (2.2.1)$$

Une structure if à deux conditions se présente souvent en pratique. Soit un nombrefixé a. Si le nombre b est supérieur ou égal à a, alors l'expression Q doit prendre la valeur 1 Si le nombre b est inférieur à a, alors l'expression Q vaut 0.

```
> a:=2;
  b:=5;
  if b<a then Q:=0
  else Q:=1
  end if;
```

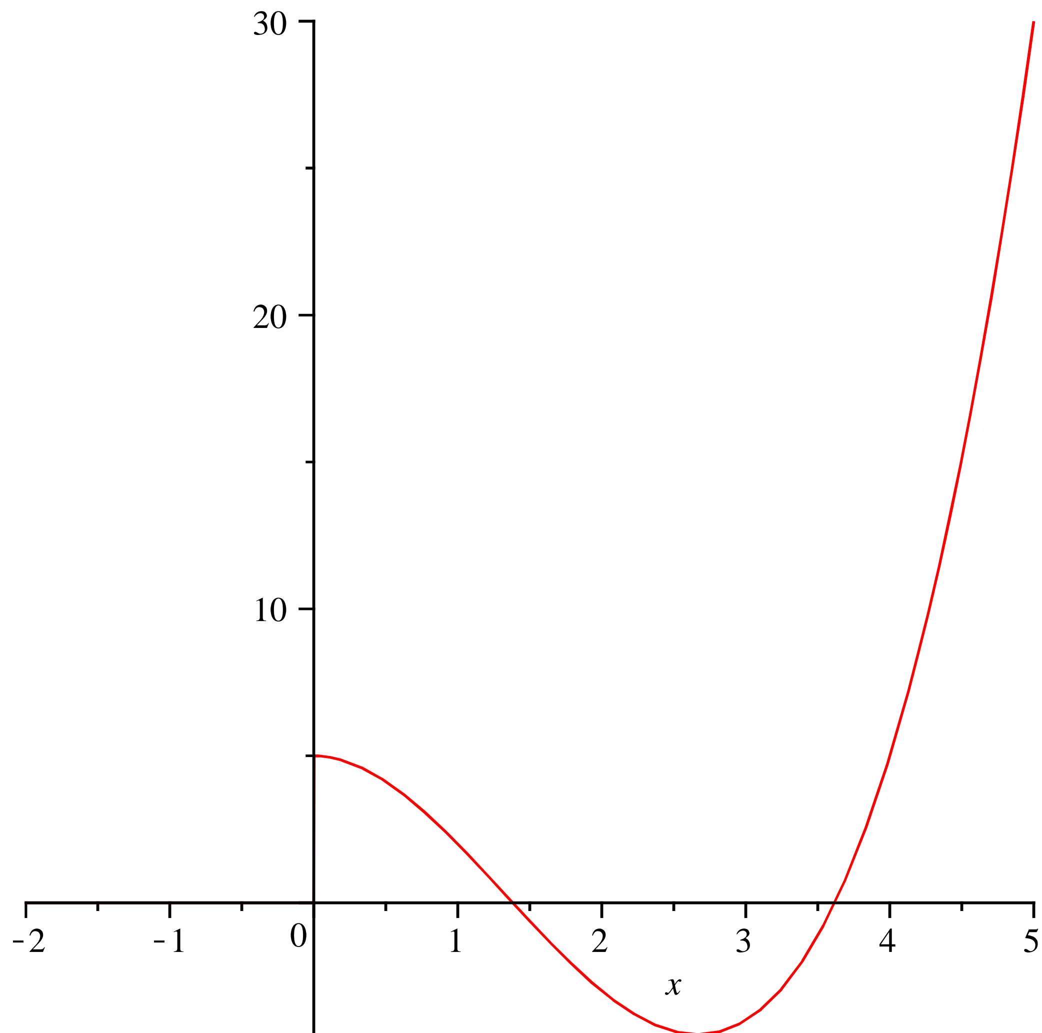
$$a := 2$$
$$b := 5$$
$$Q := 1 \qquad (2.2.2)$$

Signalons qu'une structure conditionnelle peut être intégrée dans la définition d'une fonction-procédure, ici une fonction à deux morceaux.

```
> F:=x->if x<0 or x=0 then F(x):=0 else F(x):=x^3-4*x^2+5 end
  if;
  F(-5);F(2);#il est toujours prudent de vérifier une ligne
  de commandes sur des exemples simples ...
  plot('F(x)',x=-2..5);#Demande de représentation graphique.
  Les apostrophes retardent l'évaluation de F(x).
```

$F := x \rightarrow \textbf{if } x < 0 \textbf{ or } x = 0 \textbf{ then } F(x) := 0 \textbf{ else } F(x) := x^3 - 4x^2 + 5 \textbf{ end if}$

$$0$$
$$-3$$



La généralisation à plus de deux alternatives est évidemment possible. On peut donc définir des fonctions à plus de deux morceaux. L'exemple suivant est un peu factice dans la mesure où il existe la commande dédiée **piecewise**, mais son intérêt pédagogique est évident.

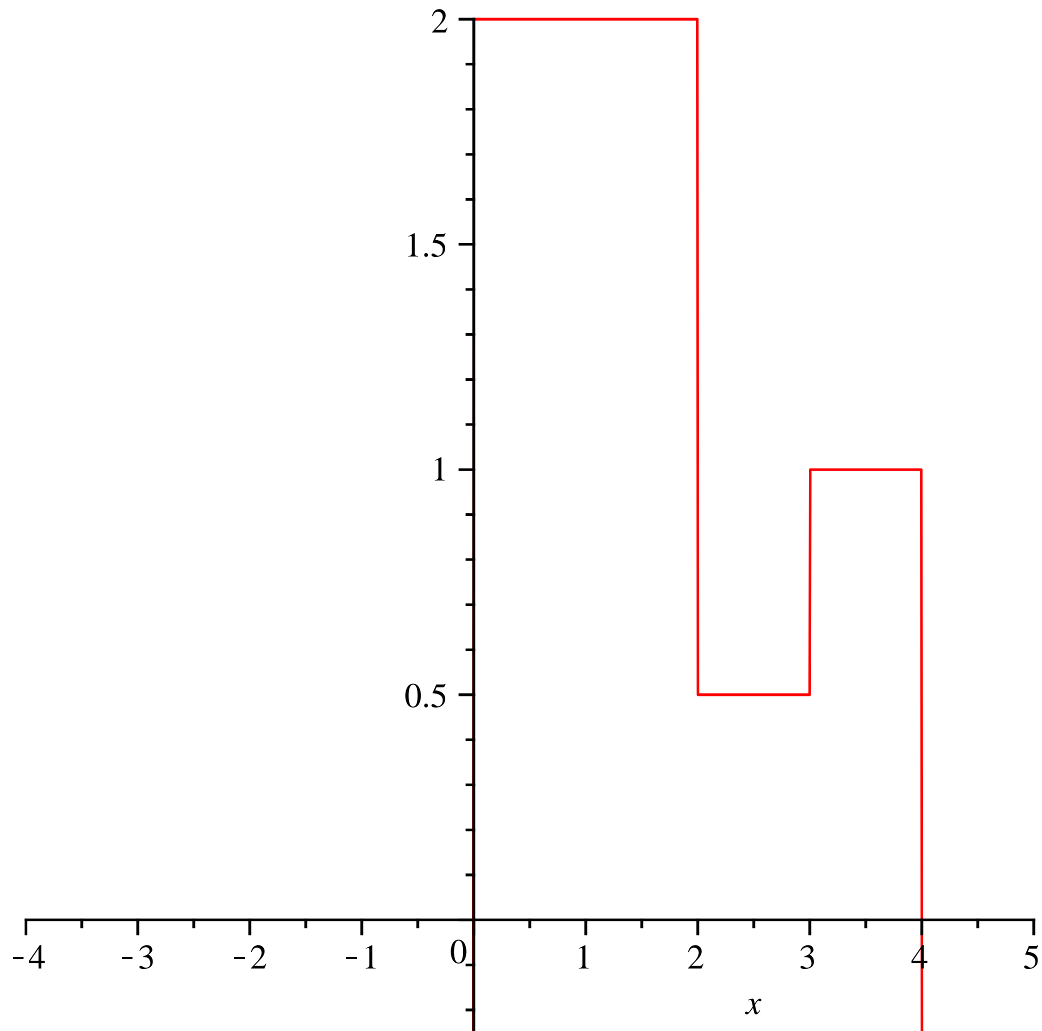
```
> G:=x->if x<0 or x>4 then G(x):=-0.25 elif x>=0 and x<2 then
  G(x):=2 elif x>=2 and x<3 then G(x):=1/2 else G(x):=1 end
  if;
  plot('G(x)',x=-4..5);#représentation graphique de la
  fonction G
```

$G := x \to$ **if** $x < 0$ **or** $4 < x$ **then** $G(x) := -0.25$ **elif** $0 \leq x$ **and** $x < 2$ **then** $G(x) := 2$

elif $2 \leq x$ **and** $x < 3$ **then**

$$G(x) := \frac{1}{2}$$

else $G(x) := 1$ **end if**



Combinaisons de structures

Les structures itératives et conditionnelles ne s'excluent pas mutuellement. Tout au contraire, une structure itérative peut abriter une structure conditionnelle et réciproquement. Deux exemples vont suffire pour faire comprendre ce point.

Exemple 1 :On se donne M une matrice à 3 lignes et 4 colonnes. On transforme ses éléments suivant la règle : s'il est négatif, sa nouvelle valeur est 0; s'il est nul, sa nouvelle valeur est 1; s'il est positif, sa nouvelle valeur est égale à la précédente augmentée de 1. Dans ce cas, l'algoriithme doit insérer une boucle conditionnelle dans une double boucle itérative.

```
> M:=Matrix([[-0.1,2.3,-3.4,0],[4.5,-5.6,0,6.7],[0,-8.9,9.8,
  -0.7]]);
  for i to 3 do
       for j to 4 do
         if M[i,j]<0 then M[i,j]:=0
         elif M[i,j]=0 then M[i,j]:=1
         else M[i,j]:=M[i,j]+1
         end if;
       end do;
```

```
end do;
M;
```

$$M := \begin{bmatrix} -0.1 & 2.3 & -3.4 & 0 \\ 4.5 & -5.6 & 0 & 6.7 \\ 0 & -8.9 & 9.8 & -0.7 \end{bmatrix}$$

$$\begin{bmatrix} 0 & 3.3 & 0 & 1 \\ 5.5 & 0 & 1 & 7.7 \\ 1 & 0 & 10.8 & 0 \end{bmatrix} \quad \textbf{(2.3.1)}$$

Exemple 2 : On se donne *M* une matrice carrée de format 3. On transforme ses éléments suivant la règle : si *M* est définie négative, toutes ses composantes sont augmentées de 1; sinon, toutes ses composantes sont diminuées de 1. L'algorithme va intégrer une double boucle itérative dans chaque branche de l'alternative if..then..else.

```
> M:=Matrix([[-2,0,0],[0,-1,0],[0,0,-5]]);#création d'une
  matrice définie négative
  with(LinearAlgebra):#chargement du paquetage contenant
  l'instruction IsDefinite
  if IsDefinite(M,query=negative_definite)=true then #cas où
  la matrice est définie négative
       for i to 3 do
            for j to 3 do
              M[i,j]:=M[i,j]+1
            end do;
       end do;
  else#cas où la matrice n'est pas définie négative
       for i to 3 do
            for j to 3 do
              M[i,j]:=M[i,j]-1
            end do;
       end do;
  end if;
  M;
```

$$M := \begin{bmatrix} -2 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -5 \end{bmatrix}$$

$$\begin{bmatrix} -1 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & -4 \end{bmatrix} \quad \textbf{(2.3.2)}$$

Quand *M* n'est pas définie négative, le programme exécute la deuxième branche de l'alternative.

```
> M:=Matrix([[-2,0,0],[0,1,0],[0,0,-5]]);#création d'une
```

```
matrice non définie
if IsDefinite(M,query=negative_definite)=true then
     for i to 3 do
          for j to 3 do
            M[i,j]:=M[i,j]+1
          end do;
     end do;
else
     for i to 3 do
          for j to 3 do
            M[i,j]:=M[i,j]-1
          end do;
     end do;
end if;
M;
```

$$M := \begin{bmatrix} -2 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -5 \end{bmatrix}$$

$$\begin{bmatrix} -3 & -1 & -1 \\ -1 & 0 & -1 \\ -1 & -1 & -6 \end{bmatrix} \quad \textbf{(2.3.3)}$$

Remarque : dans chaque exemple, la rédaction du programme a respecté le principe d'indentation des sous-blocs de sorte que le lecteur - et souvent le programmeur lui-même - puisse comprendre plus aisément la logique d'ensemble. On aura également noté que l'insertion de commentaires est toujours possible après un dièse **#**.

Procédures Maple

Les algorithmes présentés dans la section précédentes ont l'inconvénient majeur de ne concerner que la ou les variables assignées antérieurement. Toute modification de variable(s) oblige à réécrire tout le programme pour obtenir le nouveau résultat. Même si le copier-coller demande un effort minime, on peut s'épargner cette pratique en créant une procédure. On le fera d'autant plus que les inputs seront racourcis : plusieurs dizaines de lignes de commandes sont en effet remplacés par ... un simple nom éventuellement suivi de valeurs que doivent prendre les arguments. Il en ressort que, dans une feuille de travail, toute procédure valide se comporte exactement comme une commande Maple une fois qu'elle a été créée.

Les composants d'une procédure

Une procédure est un nom à qui est assignée une succession de lignes d'instructions comprises entre l'instruction d'ouverture `proc` et l'instruction de fermeture `end proc;`.
Sa structure générale est donc :

```
nom:=proc( )
...........
```

```
end proc;
```

Devant l'instruction d'ouverture, le **nom** est choisi librement par l'utilisateur, à condition de respecter les règles Maple usuelles (éviter les noms réservés en particulier). De préférence, il évoque le thème de la procédure : **moyenne** pour un calcul de moyenne; **variance** pour un calcul de variance; **hamilt** s'il faut former un hamiltoniel; etc. Il est suivi par la combinaison de touches **:=** signalant une assignation.

Le terme **proc** est un raccourci évident de *procedure*; il est suivi de parenthèses à l'intérieur desquelles on peut ne rien mettre ou au contraire - et c'est généralement le cas - donner un ou plusieurs arguments.

L'instruction de fermeture est impérative : le terme **end proc** doit être obligatoirement suivi d'un point-virgule.

Entre ces deux lignes, une procédure développe cinq catégories d'instructions, toutes closes par un point-virgule :

1 - déclaration par le mot **local** des variables locales qui seront utilisées dans la procédure,

2 - déclaration par le mot **global** des variables globales,

3 - déclaration par le mot **options** des options qui seront utilisées dans le programme,

4 - après le mot **description**, on peut insérer un texte (entre guillemets) qui définit l'objet du programme,

5 - un bloc d'instructions proprement dites, groupe de commandes qui constituent le programme.

En définitive, une procédure est structurée de la manière suivante :

```
nom:=proc(argument_1,argument_2, ...., argument_n)
local variable_locale_1,variable_locale_2, .....,
variable_locale_m;
global variable_globale 1,variable_globale_2, ....,
variable_globale_p;
options option_1,option_2,....,option_q
description "texte décrivant l'utilité du programme"
bloc de commandes;
end proc;
```

Si une procédure a impérativement besoin d'un bloc de commandes, les déclarations d'arguments, de variables locales et de variables globales, d'options et la description ne sont pas toujours nécessaires.

Les trois exemples qui suivent concrétisent la définition d'une procédure et montrent quelques particularités de son utilisation pratique.

Exemple 1 : Un enseignant calcule ainsi la moyenne semestrielle des étudiants : l'examen terminal et le contrôle continu représentent chacun 50% de la note terminale; le contrôle continu est la moyenne arithmétique des notes obtenues à 3 devoirs surveillés. On appelle la procédure de calcul de la moyenne trimestrielle **moytri**. Quatre arguments sont nécessaires, notés **x** (note à l'exament terminal), **y** (note au DS1), **z** (note au DS2) et **t** (note au DS3). Aucune variable locale ou globale n'est requise.

```
> moytri:=proc(x,y,z,t)
  0.5*x+0.5*(y+z+t)/3;
  end proc;
```

$$moytri := \textbf{proc}(x, y, z, t)\ \ 0.5 * x + 0.5 * (y + z + t) / 3\ \ \textbf{end proc}$$

Maple valide cette procédure en renvoyant en écho le programme. Si la procédure est syntaxiquement incorrecte, un message d'erreur est renvoyé. Par exemple, s'il manque un point-virgule après l'instruction de clôture :

```
> moytrierror:=proc(x,y,z,t)
  0.5*x+0.5*(y+z+t)/3;
  end proc;
```

Pour utiliser la procédure valide, il suffit de l'appeler par son nom, ici **moytri** et d'introduire entre parenthèses le nombre d'arguments nécessaires.

```
> moytri(12.1,10,14,18);
```

$$13.05000000$$

Si les arguments sont incorrects, un message d'erreur est renvoyé. Ici, il manque une note :

```
> moytri(6.5,10,9);
Error, invalid input: moytri uses a 4th argument, t, which is missing
```

Exemple 2 : La procédure suivante calcule la racine carrée de la valeur absolue du produit de 3 nombres. En poussant loin le détail des opérations, on utilise 3 variables locales en plus des arguments.

```
> calcul:=proc(x,y,z)
  local a,b,c;#déclaration de variables locales
  a:=x*y*z;#la variable a assigne le produit des 3 nombres
  b:=abs(a);#la variable b assigne la valeur absolue de a
  c:=sqrt(b);#la variable c assigne la racine carrée de b
  end proc;
```

$calcul := \mathbf{proc}(x, y, z)\ \mathbf{local}\ a, b, c;\ a := x*y*z;\ b := \mathrm{abs}(a);\ c := \mathrm{sqrt}(b)\ \mathbf{end\ proc}$

Examinons une requête précise.

```
> calcul(1.5,-2.6,3.7);
```

$$3.798683983$$

Maple renvoie le dernier résultat calculé, soit ici la valeur prise par la variable **c**. Ce principe d'affichage du dernier calcul est général : il n'est pas possible de récupérer des résultats intermédiares tels que **a** ou **b** dans l'exemple. Dans certains cas, ce principe peut être gênant; il existe heureusement des parades qui seront présentées plus bas.

Exemple 3 : Le calcul précédent ne mérite pas vraiment l'honneur d'une procédure. Plus simplement, on aurait pu créer une fonction à l'aide d'une flèche :

```
> calc:=(x,y,z)->sqrt(abs(x*y*z));
  calc(1.5,-2.6,3.7);
```

$$calc := (x, y, z) \to \sqrt{|x\,y\,z|}$$

$$3.798683983$$

Le résultat est le même que dans l'exemple 2. Normal. Ce qui est nouveau ici est que Maple reconnait la fonction **calc** comme une ... procédure.

```
> type(calc,procedure);
```

$$true$$

En d'autres termes, toutes les fonctions d'une ou de plusieurs variables construites avec des flèches ne sont rien d'autre que des procédures déguisées.

Arguments, variables locales et variables globales

D'un point de vue formel, une procédure est assimilable à une correspondance entre des expressions d'entrée et des expressions de sortie. Les expressions en entrée sont les arguments de la procédure. Ce sont des données de départ qu'il faut distinguer des variables utilisées au sein du programme qu'il faut plutôt interpréter comme des auxiliaires de calcul.

Arguments

Les arguments sont des expressions données au départ et se présentent comme une séquence, éventuellement vide, insérée entre parenthèses après la commande **proc.**

Tous les types sont possibles de sorte qu'un argument peut être un nombre, une fonction, un ensemble, une liste, un vecteur, une matrice, un tableau, une chaîne de caractères, et même .. une procédure. Si besoin est, le type de l'argument se déclare à l'aide de l'opérateur **::**. En cas d'incompatibilité entre le type entré par l'utilisateur et le type exigé par le programmeur, un message d'erreur s'affiche et explique la nature de l'erreur commise.

```
> sddnp:=proc(x::posint,y::negint)#la procédure doit s'appliquer à deux entiers, le premier positif, le second négatif.
x+y;
end proc;
```

$$sddnp := \textbf{proc}(x::posint,\ y::negint)\ \ x + y\ \ \textbf{end proc}$$

```
> sddnp(4,-3);
```

$$1$$

```
> sddnp(-1,-9);#le premier argument n'est pas du type déclaré
Error, invalid input: sddnp expects its 1st argument, x, to be of type posint, but received -1
```

Il est clair que les arguments seront utilisés dans le programme. Toutes les manipulations sont concevables sauf leur réaffectation. Par exemple, si la procédure a pour argument la fonction **F1:=x->exp(x^2)**, on ne peut pas dans la procédure affecter à **F1** la fonction **x->2*x** ou tout autre expression. Les arguments ne sont donc pas des variables, même s'ils en ont le statut dans le reste du worksheet.

Variables locales et variables globales

Par définition, une variable locale reste affectée à l'intérieur de la procédure. En conséquence, si une variable de même nom existe dans le worksheet en dehors de la procédure, sa valeur n'est pas modifiée après exécution de la procédure. A contrario, une variable globale conserve sa valeur après exécution de la procédure. Ces propriétés sont mises en évidence dans l'exemple suivant :

```
> manip:=proc(x,y)
local s;#s est déclarée variable locale
global q;#q est déclarée variable globale
s:=x+y;#s assigne la somme de x et y
```

```
q:=x/y;#q assigne la division de x par y
s+q;
end proc;
```
$manip := \mathbf{proc}(x, y)\ \mathbf{local}\ s;\ \mathbf{global}\ q;\ s := x + y;\ q := x/y;\ s + q\ \mathbf{end\ proc}$

```
> manip(5.3,6.6);
s;#la variable locale s a été désassignée
q;#la variable globale q reste assignée
```
$$12.70303030$$
$$s$$
$$0.8030303030$$

En pratique, les variables du programme sont le plus souvent déclarées locales - c'est d'ailleurs l'hypothèse que Maple retient par défaut lorsque l'utilisateur omet de délarer le statut des variables. L'intérêt des variables globales est évidemment la possibilité de les récupérer après exécution de la procédure.

Exemple : On considère la suite numérique définie par l'équation de récurrence :

$$\begin{cases} u_{n+1} = 2\,u_n \text{ si } u_n \leq 0.5 \\ u_{n+1} = -2\,u_n + 2 \text{ si } u_n > 0.5 \end{cases}$$

Pour afficher la liste des n premiers termes de la suite, on va élaborer une procédure dont les arguments sont d'une part **n** le nombre de termes demandé par l'utilisateur et d'autre part **u0** le premier terme de la suite. Les instructions vont intégrer une structure conditionnelle "classique" au sein d'une structure itérative. On récupère le résultat final en demandant la sortie de la variable locale **L** une fois toutes les itérations terminées. La variable locale **i** est le compteur et la variable locale **u** désigne la suite.

```
> restart:
suite:=proc(n,u0)
local i,u,L;
u[0]:=u0;#condition initiale
L:=[u[0]];#initialisation de la liste
for i from 0 to (n-2) do
          if u[i]<=0.5 then u[i+1]:=2*u[i];L:=[op(L),u
[i+1]];#première branche de l'alternative
          else u[i+1]:=-2*u[i]+2;L:=[op(L),u[i+1]];
#deuxième branche de l'alternative
          end if;
end do;
L;#pour afficher la liste finale
end proc;
```
$suite := \mathbf{proc}(n, u0)$ **(3.1.1.2.1)**

$\quad \mathbf{local}\ i, u, L;$

$\quad u[0] := u0;$

$\quad L := [u[0]];$

$\quad \mathbf{for}\ i\ \mathbf{from}\ 0\ \mathbf{to}\ n - 2\ \mathbf{do}$

```
        if u[i] <= 0.5 then
            u[i+1] := 2*u[i]; L := [op(L), u[i+1]]
        else
            u[i+1] := -2*u[i]+2; L := [op(L), u[i+1]]
        end if
    end do;
    L
end proc
```

```
> suite(25,0.15);#demande des 25 premiers termes de la
  suite démarrant en 0.15.
  %[2];#extraction du deuxième terme
```

[0.15, 0.30, 0.60, 0.80, 0.40, 0.80, 0.40, 0.80, 0.40, 0.80, 0.40, 0.80, 0.40, 0.80, 0.40, 0.80, 0.40, 0.80, 0.40, 0.80, 0.40, 0.80, 0.40, 0.80, 0.40]

0.30 **(3.1.1.2.2)**

Sorties de données

La question de la sortie des données mérite qu'on s'y attarde car on a vu que seule la dernière instruction génère un output, ce qui n'est pas forcément suffisant. On évoque ici les deux moyens principaux pour extraire d'une procédure plusieurs renseignements.

Utilisation de la commande `return`

En insérant dans les instructions d'un programme la commande :

```
return expression_1,expression_2,...,expression_N;
```

Maple affiche la suite d'expressions en question mais n'affiche plus en revanche le dernier résultat calculé.

Exemple 1 : Soit 3 nombres x, y et z. La procédure `prod` doit renvoyer la somme $xy + yz + xz$. Pour prendre connaissance des produits partiels xy, yz et xz, on crée des variables locales que la commande return va afficher.

```
> prod:=proc(x,y,z)
  local a,b,c;
  a:=x*y;
  b:=y*z;
  c:=x*z;
  return a,b,c;
  a*b*c;
  end proc;
```

```
prod := proc(x, y, z)
    local a, b, c;
    a := x*y; b := y*z; c := x*z; return a, b, c; a*b*c
end proc
```

```
> prod(2,3,4);
```

6, 12, 8

La commande **return** a permis l'affichage des produits partiels ... mais pas celui du résultat recherché! Une parade possible est de créer une variable globale dans le programme et demander à la fin du programme sa valeur :

```
> restart;
  prodreturn:=proc(x,y,z)
  local a,b,c;
  global r;#déclaration d'une variable globale
  a:=x*y;
  b:=y*z;
  c:=x*z;
  r:=a*b*c;#la variable globale r contient le résultat
  recherché
  return a,b,c,r;#pour afficher le contenu des variables
  locales et de la variable globale
  end proc;
```

prodreturn := **proc**(*x*, *y*, *z*) **(3.2.1.1)**

local *a*, *b*, *c*;

global *r*;

a := *x* * *y*; *b* := *y* * *z*; *c* := *x* * *z*; *r* := *a* * *b* * *c*; **return** *a*, *b*, *c*, *r*

end proc

```
> prodreturn(2,3,4);
  a;b;c;r;#a, b et c sont désassignées alors que r est
  assigné.
```

6, 12, 8, 576

a

b

c

576 **(3.2.1.2)**

Exemple 2 : on passe maintenant à un programme plus ambitieux. On cherche à bâtir une procédure nommée **suite** capable de calculer le n-ième terme d'une suite numérique définie par une équation de récurrence à coefficients constants et sans second membre, soit $u_n = \alpha\, u_{n-1} + \beta\, u_{n-2}$ avec u_0 et u_1 donnés. Les arguments de la procédure sont n, α et β, u_0 et u_1. Comme un utilisateur peut appeler le premier terme (n=0), le second (n=1) ou tout autre terme (n>1), une boucle conditionnelle est nécessaire et l'affichage des résultats se fait avec **return**. Pour obtenir un programme agréable, l'instruction principale, située après else, fait appel ... à la procédure **suite**, c'est à dire elle-même! On dit alors qu'on est en présence d'une structure récursive.

```
> restart;
  suite:=proc(n,alpha,beta,u0,u1)
  if n=0 then return u0;
  elif n=1 then return u1;
  else return alpha*suite(n-1,alpha,beta,u0,u1)+beta*suite
```

```
(n-2,alpha,beta,u0,u1);
end if;
end proc;
```

suite := **proc**(*n*, alpha, beta, *u0*, *u1*)
  if $n = 0$ **then**
    return *u0*
  elif $n = 1$ **then**
    return *u1*
  else
    return alpha * *suite*($n-1$, alpha, beta, *u0*, *u1*) + beta * *suite*($n-2$, alpha, beta, *u0*, *u1*)
  end if
end proc

Cette procédure est syntaxiquement correcte, mais elle est très gourmande en calculs, ce que confirment les temps de calcul (évitez de demander le calcul pour $n = 40$!!!).

```
> suite(0,1,1,1,1);#premier terme
  suite(1,1,1,1,1);#2° terme
  suite(2,1,1,1,1);#3° terme
  suite(5,1,1,1,1);#6° terme
  suite(10,1,1,1,1);#11° terme
  suite(20,1,1,1,1);#21° terme
  st:=time();suite(30,1,1,1,1);time()-st;#31° terme avec
  demande de temps de calcul
```

1

1

2

8

89

10946

st := 10.764

1346269

2.932

La lenteur de calcul vient de l'instruction après `else` : le calcul du *n*-ième terme par la procédure `suite` se fait en appelant la procédure `suite` à l'ordre *n*-1 et *n*-2, deux calculs nécessitant eux-même l'appel à cette même procédure à tous les ordres inférieurs...
Ce problème est évité en "calant" l'option `remember` au début du programme, ce qui force Maple à mettre en mémoire les termes de la suite une fois calculés au lieu de recalculer plusieurs fois les mêmes termes.

```
> suite2:=proc(n,alpha,beta,u0,u1)
  options remember;
  if n=0 then return u0;
  elif n=1 then return u1;
```

```
else return alpha*suite2(n-1,alpha,beta,u0,u1)+beta*
suite2(n-2,alpha,beta,u0,u1);
end if;
end proc;
```

suite2 := **proc**(*n*, alpha, beta, *u0*, *u1*)
option *remember*;
if $n = 0$ **then**
return *u0*
elif $n = 1$ **then**
return *u1*
else
return alpha * *suite2*($n - 1$, alpha, beta, *u0*, *u1*) + beta * *suite2*($n - 2$, alpha, beta, *u0*, *u1*)
end if
end proc

```
> st:=time();suite2(150,1,1,1,1);time()-st;#151° terme avec
demande de temps de calcul
```

$st := 13.696$

16130531424904581415797907386349

0.

La table `remember` est récupérable puisqu'elle constitue le quatrième opérande de la procédure.

```
> op(4,eval(suite2));
```

table([(44, 1, 1, 1, 1) = 1134903170, (73, 1, 1, 1, 1) = 1304969544928657, (38, 1, 1, 1, 1) = 63245986, (24, 1, 1, 1, 1) = 75025, (67, 1, 1, 1, 1) = 72723460248141, (128, 1, 1, 1, 1) = 407305795904080553832073954, (19, 1, 1, 1, 1) = 6765, (117, 1, 1, 1, 1) = 2046711111473984623691759, (122, 1, 1, 1, 1) = 22698374052006863956975682, (111, 1, 1, 1, 1) = 114059301025943970552219, (148, 1, 1, 1, 1) = 6161314747715278029583501626149, (49, 1, 1, 1, 1) = 12586269025, (62, 1, 1, 1, 1) = 6557470319842, (91, 1, 1, 1, 1) = 7540113804746346429, (16, 1, 1, 1, 1) = 1597, (104, 1, 1, 1, 1) = 3928413764606871165730, (93, 1, 1, 1, 1) = 19740274219868223167, (8, 1, 1, 1, 1) = 34, (146, 1, 1, 1, 1) = 2353412818241252672952597492098, (135, 1, 1, 1, 1) = 11825896447871834976429068427, (12, 1, 1, 1, 1) = 233, (92, 1, 1, 1, 1) = 12200160415121876738, (86, 1, 1, 1, 1) = 679891637638612258, (3, 1, 1, 1, 1) = 3, (147, 1, 1, 1, 1) = 3807901929474025356630904134051, (21, 1, 1, 1, 1) = 17711, (80, 1, 1, 1, 1) = 37889062373143906, (37, 1, 1, 1, 1) = 39088169, (74, 1, 1, 1, 1) = 2111485077978050, (31, 1, 1, 1, 1) = 2178309, (68, 1, 1, 1, 1) = 117669030460994, (129, 1, 1, 1, 1) = 659034621587630041982498215, (27, 1, 1, 1, 1) = 317811, (110, 1, 1, 1, 1) = 70492524767089125814114, (43, 1, 1, 1, 1) = 701408733, (56, 1, 1, 1, 1) = 365435296162, (1, 1, 1, 1, 1) = 1, (141, 1, 1,

1, 1) = 212207101440105399533740733471, (98, 1, 1, 1, 1)
= 218922995834555169026, (55, 1, 1, 1, 1) = 225851433717, (76, 1, 1, 1, 1)
= 5527939700884757, (105, 1, 1, 1, 1) = 6356306993006846248183, (134, 1, 1,
1, 1) = 7308805952221443105020355490, (7, 1, 1, 1, 1) = 21, (35, 1, 1, 1, 1)
= 14930352, (32, 1, 1, 1, 1) = 3524578, (149, 1, 1, 1, 1)
= 9969216677189303386214405760200, (90, 1, 1, 1, 1)
= 4660046610375530309, (79, 1, 1, 1, 1) = 23416728348467685, (116, 1, 1, 1,
1) = 1264937032042997393488322, (5, 1, 1, 1, 1) = 8, (145, 1, 1, 1, 1)
= 1454489111232772683678306641953, (94, 1, 1, 1, 1)
= 31940434634990099905, (17, 1, 1, 1, 1) = 2584, (123, 1, 1, 1, 1)
= 36726740705505779255899443, (29, 1, 1, 1, 1) = 832040, (72, 1, 1, 1, 1)
= 806515533049393, (61, 1, 1, 1, 1) = 4052739537881, (23, 1, 1, 1, 1) = 46368,
(50, 1, 1, 1, 1) = 20365011074, (39, 1, 1, 1, 1) = 102334155, (0, 1, 1, 1, 1) = 1,
(124, 1, 1, 1, 1) = 59425114757512643212875125, (57, 1, 1, 1, 1)
= 591286729879, (54, 1, 1, 1, 1) = 139583862445, (115, 1, 1, 1, 1)
= 781774079430987230203437, (112, 1, 1, 1, 1)
= 184551825793033096366333, (69, 1, 1, 1, 1) = 190392490709135, (18, 1, 1,
1, 1) = 4181, (42, 1, 1, 1, 1) = 433494437, (28, 1, 1, 1, 1) = 514229, (127, 1, 1,
1, 1) = 251728825683549488150424261, (6, 1, 1, 1, 1) = 13, (36, 1, 1, 1, 1)
= 24157817, (22, 1, 1, 1, 1) = 28657, (97, 1, 1, 1, 1)
= 135301852344706746049, (142, 1, 1, 1, 1)
= 343358302784187294870275058337, (75, 1, 1, 1, 1) = 3416454622906707,
(11, 1, 1, 1, 1) = 144, (109, 1, 1, 1, 1) = 43566776258854844738105, (4, 1, 1, 1,
1) = 5, (130, 1, 1, 1, 1) = 1066340417491710595814572169, (87, 1, 1, 1, 1)
= 1100087778366101931, (108, 1, 1, 1, 1) = 26925748508234281076009, (137,
1, 1, 1, 1) = 30960598847965113057878492344, (102, 1, 1, 1, 1)
= 1500520536206896083277, (131, 1, 1, 1, 1)
= 1725375039079340637797070384, (64, 1, 1, 1, 1) = 17167680177565, (53, 1,
1, 1, 1) = 86267571272, (58, 1, 1, 1, 1) = 956722026041, (13, 1, 1, 1, 1) = 377,
(47, 1, 1, 1, 1) = 4807526976, (84, 1, 1, 1, 1) = 259695496911122585, (113, 1,
1, 1, 1) = 298611126818977066918552, (2, 1, 1, 1, 1) = 2, (126, 1, 1, 1, 1)
= 155576970220531065681649693, (40, 1, 1, 1, 1) = 165580141, (26, 1, 1, 1, 1)
= 196418, (82, 1, 1, 1, 1) = 99194853094755497, (71, 1, 1, 1, 1)
= 498454011879264, (30, 1, 1, 1, 1) = 1346269, (89, 1, 1, 1, 1)
= 2880067194370816120, (150, 1, 1, 1, 1)
= 16130531424904581415797907386349, (83, 1, 1, 1, 1)
= 160500643816367088, (10, 1, 1, 1, 1) = 89, (144, 1, 1, 1, 1)
= 898923707008479989274290850145, (101, 1, 1, 1, 1)
= 927372692193078999176, (138, 1, 1, 1, 1)
= 50095301248058391139327916261, (95, 1, 1, 1, 1)
= 51680708854858323072, (132, 1, 1, 1, 1)
= 2791715456571051233611642553, (65, 1, 1, 1, 1) = 27777890035288, (46, 1,

1, 1, 1) = 2971215073, (107, 1, 1, 1, 1) = 16641027750620563662096, (120, 1, 1, 1, 1) = 8670007398507948658051921, (77, 1, 1, 1, 1) = 8944394323791464, (34, 1, 1, 1, 1) = 9227465, (119, 1, 1, 1, 1) = 5358359254990966640871840, (25, 1, 1, 1, 1) = 121393, (140, 1, 1, 1, 1) = 131151201344081895336534324866, (41, 1, 1, 1, 1) = 267914296, (70, 1, 1, 1, 1) = 308061521170129, (99, 1, 1, 1, 1) = 354224848179261915075, (96, 1, 1, 1, 1) = 83621143489848422977, (85, 1, 1, 1, 1) = 420196140727489673, (143, 1, 1, 1, 1) = 555565404224292694404015791808, (52, 1, 1, 1, 1) = 53316291173, (81, 1, 1, 1, 1) = 61305790721611591, (14, 1, 1, 1, 1) = 610, (59, 1, 1, 1, 1) = 1548008755920, (136, 1, 1, 1, 1) = 19134702400093278081449423917, (125, 1, 1, 1, 1) = 96151855463018422468774568, (114, 1, 1, 1, 1) = 483162952612010163284885, (103, 1, 1, 1, 1) = 2427893228399975082453, (60, 1, 1, 1, 1) = 2504730781961, (15, 1, 1, 1, 1) = 987, (121, 1, 1, 1, 1) = 14028366653498915298923761, (118, 1, 1, 1, 1) = 3311648143516982017180081, (51, 1, 1, 1, 1) = 32951280099, (48, 1, 1, 1, 1) = 7778742049, (133, 1, 1, 1, 1) = 4517090495650391871408712937, (106, 1, 1, 1, 1) = 10284720757613717413913, (63, 1, 1, 1, 1) = 10610209857723, (20, 1, 1, 1) = 10946, (100, 1, 1, 1, 1) = 573147844013817084101, (9, 1, 1, 1, 1) = 55, (33, 1, 1, 1, 1) = 5702887, (78, 1, 1, 1, 1) = 14472334024676221, (139, 1, 1, 1, 1) = 81055900096023504197206408605, (88, 1, 1, 1, 1) = 1779979416004714189, (45, 1, 1, 1, 1) = 1836311903, (66, 1, 1, 1, 1) = 44945570212853])

Utilisation de la commande `print`

En insérant dans les instructions d'un programme la commande

`print(expression_1,expression_2,...,expression_N)`

Maple affiche la suite d'expressions ainsi que le dernier résultat calculé. L'avantage de print est que l'utilisateur prend connaissance des calculs intermédiares et obtient le résultat final, ce qui permet de "suivre" le fonctionnement d'un algorithme lors de sa mise au point. On reprend l'exemple 1 du paragraphe précédent.

```
> restart;
  prodprint:=proc(x,y,z)
  local a,b,c;
  a:=x*y;
  b:=y*z;
  c:=x*z;
  print(a,b,c);
  a*b*c;
  end proc;
```

prodprint := **proc**(*x*, *y*, *z*)
  local *a*, *b*, *c*;
  a := *x* * *y*; *b* := *y* * *z*; *c* := *x* * *z*; *print*(*a*, *b*, *c*); *a* * *b* * *c*

end proc

```
> prodprint(2,3,4);
```

$$6, 12, 8$$
$$576$$

Débogage

Mettre au point une procédure ambitieuse relève souvent du parcours du combattant. Maple est intraitable sur la correction syntaxique d'un programme et le programmeur, même chevronné, se voit bien des fois refuser la validation de sa procédure pour des raisons qui le rendent perplexe sur le moment et qui peuvent lui échapper pendant des heures. Une chose est sûre dans ce domaine : Maple a toujours raison, et il faut trouver la faille dans son propre programme. Heureusement, toute faute grossière est signalée dès la validation de la procédure par un message d'erreur et la barre d'insertion clignote à l'endroit où la procédure cloche.
D'autre part, l'utilisateur peut suivre pas à pas les opérations suivies par Maple à l'intérieur d'une procédure en invoquant la fonction `trace`. L'exemple suivant permet de comprendre comment on peut prendre la mesure d'une erreur de conception (un "bug" ou encore un "bogue"). On désire afficher dans un ensemble les carrés des n premiers nombres entiers. Une procédure fautive est :

```
> restart;
  car:=proc(n)
  local i,c,L;
  for i to n do
  c:=i^2;
  L:=L union {c}
  end do;
  print(L);
  end proc;
```

car := **proc**(n)
local i, c, L;
for i **to** n **do** $c := i$^2; $L := union(L, \{c\})$ **end do**; $print(L)$
end proc

Le test n'est pas concluant :

```
> car(5);
```

$$L \cup \{1, 4, 9, 16, 25\} \qquad \textbf{(3.3.1)}$$

D'où provient le $L \cup$? On appelle la fonction `trace(nom_de_la_procedure)`:

```
> trace(car);
```

$$car$$

Puis on refait le test :

```
> car(5);
{--> enter car, args = 5
```

$$c := 1$$
$$L := L \cup \{1\}$$
$$c := 4$$
$$L := L \cup \{1, 4\}$$

$$c := 9$$
$$L := L \cup \{1, 4, 9\}$$
$$c := 16$$
$$L := L \cup \{1, 4, 9, 16\}$$
$$c := 25$$
$$L := L \cup \{1, 4, 9, 16, 25\}$$
$$L \cup \{1, 4, 9, 16, 25\}$$

```
<-- exit car (now at top level) = }
```

Dès la lecture des résultats du premier round, on constate que la variable locale **L** n'a pas été initialisée. Il aurait fallu commencer par la définir comme un ensemble et insérer l'instruction **L:={}**. Le programme devient alors :

```
> carc:=proc(n)
  local i,c,L;
  L:={};
  for i to n do
  c:=i^2;
  L:=L union {c}
  end do;
  print(L);
  end proc;
```

carc := **proc**(*n*)
  local *i*, *c*, *L*;
  L := { }; **for** *i* **to** *n* **do** *c* := *i*^2; *L* := *union*(*L*, {*c*}) **end do**; *print*(*L*)
end proc

```
> trace(carc);
```

carc

```
> carc(5);
{--> enter carc, args = 5
```

$$L := \{\ \}$$
$$c := 1$$
$$L := \{1\}$$
$$c := 4$$
$$L := \{1, 4\}$$
$$c := 9$$
$$L := \{1, 4, 9\}$$
$$c := 16$$
$$L := \{1, 4, 9, 16\}$$
$$c := 25$$
$$L := \{1, 4, 9, 16, 25\}$$
$$\{1, 4, 9, 16, 25\}$$

```
<-- exit carc (now at top level) = }
```

Le problème est éliminé.

Conclusion

Ce chapitre ne prétend pas couvrir tout le champ de la programmation en Maple. Pour ce faire, il faudrait un ouvrage complet. Le manuel Maple, quant à lui, tient en deux tomes. Comme le titre l'annonce, c'est une introduction qui répond à deux objectifs. Il s'agit tout d'abord d'initier à la mise au point de procédures assez simples qui répondent en pratique à la grande majorité des besoins d'un économiste. D'autre part, il veut pour ainsi dire déculpabiliser l'utilisateur moyen devant les difficultés supposées de la programmation et l'affranchir des contraintes que représentent les commandes pré programmées. Ces deux raisons nous ont poussés à placer ce chapitre en début d'ouvrage plutôt qu'à la fin comme on le fait ordinairement. La programmation est tenue pour un domaine difficile réservé à une élite rompue aux arcanes de l'informatique. Elle n'est qu'un supplément d'âme pour l'honnête homme curieux ou la chasse gardée de spécialistes. Ce n'est pas notre point de vue. Apprendre le plus tôt possible à faire ses propres programmes non seulement améliore la qualité d'un worksheet mais aussi aide à comprendre comment fonctionnent réellement les commandes disponibles dans la bibliothèque principale et les paquetages.