4.3 *L'instruction* Return

Nous allons voir dans ce paragraphe comment écrire une *fonction* retournant la valeur de la racine d'une équation par dichotomie. La syntaxe est très proche de celle du programme, mais comporte des différences qu'il est nécessaire de bien comprendre.

La plus importante est que l'on renvoie une valeur, et qu'il faut donc un moyen d'indiquer quelle est cette valeur.

L'instruction Return permet de le faire de manière explicite. On l'utilise sous la forme **Return** *résultat*

Voici par exemple la « version minimale » du programme ou de la fonction utilisant la méthode de recherche par dichotomie (sans les contrôles de validité sur les différents arguments) :

Version programme

```
Define dicho(a0,b0,e)=
Prgm
Local a,b,c,fa,fc
    Disp "Résolution de ",f1(x)=0
    Disp "dans ",[a0,b0]
    a:=approx(a0)
    b:=approx(b0)
    fa:=f1(a)
    While abs(b−a)>e
     c:=(a+b)/2
     fc:=f1(c)
     If fc=0 Then
      Disp c," est solution"
      Stop
     ElseIf fa*fc>0 Then
      a:=c
     Else
      b:=c
     EndIf
     Disp [a,b]
    EndWhile
    EndPrgm
```

Version fonction

```
Define fdicho(a0,b0,e)=
Func
Local a,b,c,fa,fc
    Disp "Résolution de ",f1(x)=0
    Disp "dans ",[a0,b0]
    a:=approx(a0)
    b:=approx(b0)
    fa:=f1(a)
    While abs(b−a)>e
     c:=(a+b)/2
     fc:=f1(c)
     If fc=0 Then
      Disp c," est solution"
      Return c
     ElseIf fa*fc>0 Then
      a:=c
     Else
      b:=c
     EndIf
     Disp [a,b]
    EndWhile
    Return(a+b)/2
    EndFunc
```

En fait, il n'y a que deux différences entre la version *Programme* et la version *Fonction.*

1. Le bloc **Prgm** … **EndPrgm** est remplacé par un bloc **Func** … **EndFunc**
2. Une nouvelle instruction, spécifique aux fonctions, permet d'interrompre le déroulement de la fonction et de renvoyer la valeur calculée. Il s'agit de l'instruction **Return**. Dans le cas présent, ce résultat pourra être de différentes natures : une chaîne de caractères en cas d'erreur, ou une valeur numérique en cas de fonctionnement normal.

En fait, la différence majeure est qu'il est possible d'utiliser le résultat renvoyé par la fonction **fdicho** alors que l'on ne peut rien faire des résultats obtenus au cours de l'exécution du programme **dicho** (mis à part, éventuellement, les utiliser manuellement dans une opération de copier / coller).

Par exemple, l'instruction **s:=1+dicho(0,1,0.001)** provoque un message d'erreur, alors que **s:=1+fdicho(0,1,0.001)** est utilisable de la même manière que **s:=1+cos(5)**.

Dans le second cas, la fonction **fdicho** sera exécutée (avec affichage des étapes intermédiaires), puis la valeur obtenue sera ajoutée à 1, et le résultat stocké dans **s**.

4.4 Return *implicite*

Dans le cas d'une fonction plus simple, il n'est pas toujours nécessaire d'utiliser explicitement l'instruction **Return**. Par défaut, c'est le dernier résultat obtenu avant la fin de l'exécution de cette fonction qui est retourné.

Voici un exemple :

```
Define u(n)=
Func
if mod(n,2)=0 then
  2n+1
Else
  3n−1
EndIf
EndFunc
```

Cette fonction retourne la valeur $2n+1$ si n est un nombre pair, et la valeur $3n-1$ dans le cas contraire.

Ce qui est écrit ici est strictement équivalent à

```
Define u(n)=
Func
if mod(n,2)=0 then
  Return 2n+1
Else
  Return 3n−1
EndIf
EndFunc
```

4.5 *Syntaxe abrégée*

Dans le cas où la fonction ne nécessite qu'une seule instruction pour indiquer le calcul à effectuer, on peut au choix utiliser l'une des syntaxes suivantes :

Define *nomfonct*(*var1*,*var2*, ...)= **Func**
  Return *instruction*
EndFunc

Ou plus simplement,

Define *nomfonct*(*var1*,*var2*, ...)= **Func**
  instruction
EndFunc

Ou même, encore plus simplement,

nomfonct(*var1*,*var2*, ...) := *instruction*

Cette dernière écriture peut être privilégiée lorsque l'on travaille sur l'unité nomade TI-Nspire pour définir une fonction dont le calcul ne nécessite qu'une instruction.
Bien noter l'utilisation d'un simple signe = dans **Define**, mais de := dans le dernier cas.

Par exemple, la fonction de l'exemple précédent aurait aussi pu être définie par :

```
u(n):=when(mod(n,2)=0,2n+1,3n-1)
```

5. Récursivité

Une fonction qui s'appelle elle-même est appelée une fonction récursive. Cela permet d'obtenir des définitions qui peuvent être extrêmement simples.

5.1 *Un premier exemple*

Par exemple, si on souhaite définir la suite définie par

$$\begin{cases} u_0 = 1 \\ \forall n \in \mathbb{N}, u_{n+1} = \exp(-n.u_n) \end{cases}$$

On a le choix entre l'écriture d'une boucle (approche itérative) ou l'écriture d'une fonction récursive.

Voici la première version :

```
Define u(n)=
Func
Local v,k
v:=1
For k,1,n
 v:=exp(-(k-1)*v)
EndFor
Return v
EndFunc
```

L'idée consiste à calculer les termes de proche en proche. La variable *v* va contenir le terme de rang *k*, pour *k*, variant de 1 à *n*.

Si $n = 0$, la boucle ne sera pas du tout effectuée, et on en restera à la valeur de *v* fixée au début de la fonction. La valeur retournée sera donc 1. Ce qui est le résultat attendu.

Si $n > 0$, on va entrer dans la boucle, avec initialement u_0 dans *v*.

1. Pour $k = 1$, on va remplacer *v* par $\exp\left(-(1-1).v\right)$, ce qui correspondra au terme u_1.
2. Pour $k = 2$, on va remplacer *v* par $\exp\left(-(2-1).v\right)$, ce qui correspondra au terme u_2.
3. …

Pour comprendre le calcul effectué, il est nécessaire de voir que cette suite peut aussi être définie par :

$$\begin{cases} u_0 = 1 \\ \forall n \in \mathbb{N}^*, u_n = \exp(-(n-1).u_{n-1}) \end{cases}$$

À la fin de la boucle, la variable *v* contiendra bien la valeur de u_n.

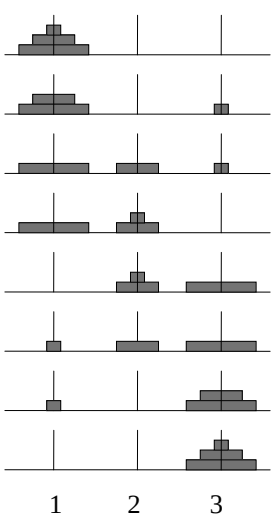
Voici à présent la version récursive, totalement immédiate :

```
Define u(n)=
Func
Local v
if n=0 then
 Return 1
Else
 Return exp(−(n−1)u(n−1))
EndIf
EndFunc
```

Cette version est la traduction exacte de la définition par récurrence du terme u_n

5.2 *Un exemple plus subtil : le problème des tours de Hanoï.*

On dispose de trois piquets. On veut déplacer une pile de n disques situés sur un premier piquet vers le troisième. Ces disques sont de tailles croissantes. On s'interdit de placer un disque sur un disque plus petit. On peut en revanche utiliser le deuxième piquet pour y placer les disques de façon temporaire. Quels sont les déplacements à effectuer ?



Description récursive de l'algorithme :

Pour déplacer *n* disques d'un piquet *a* vers un piquet *b*, on procède de la façon suivante :

S'il n'y a qu'un disque à déplacer, il suffit de noter le numéro du piquet de départ et le numéro du piquet d'arrivée.

Sinon,

1. On déplace les *n*-1 premiers disques du piquet *a* vers le piquet intermédiaire *c*.
2. On déplace le dernier disque (le plus gros) de *a* vers *b*
3. On termine en déplaçant les $n-1$ disques de *c* vers *b*.

Il suffit ensuite de remarquer que si *a* et *b* représentent les numéros de deux piquets, alors le troisième porte toujours le numéro $c = 6 - a - b$. (Par exemple, si $a = 1$ et $b = 3$, $c = 6 - 1 - 3 = 2$.)

On peut en déduire le texte du *programme* (à gauche) ou de la *fonction* (à droite) à utiliser :

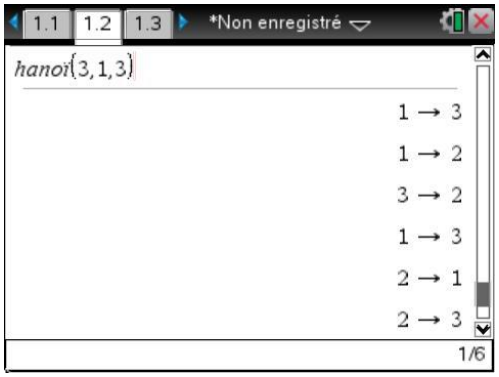
Ce *programme* affiche les déplacements.
Les arguments sont : le nombre de disques, le numéro du piquet, et le numéro de celui d'arrivée.

```
Define deplace(n,a,b) = Prgm
If n=1 Then
 Disp a," → ",b
Else
 deplace(n−1,a,6−a−b)
 deplace(1,a,b)
 deplace(n−1,6−a−b,b)
EndIf
```

Cette *fonction* utilise les mêmes arguments et affiche aussi les étapes. De plus, elle retourne également le nombre d'opérations nécessaires.

```
Define hanoi(n,a,b) = Func
Local n1,n2,n3
If n=1 Then
 Disp a," → ",b
 Return 1
Else
 n1:=hanoi(n−1,a,6−a−b)
 n2:=hanoi(1,a,b)
```

EndPrgm	n3:=hanoi(n−1,6−a−b,b) Return n1+n2+n3 EndIf EndFunc



On retrouve les 7 déplacements de l'illustration ci-dessus. La fonction permet de vérifier qu'avec une pile de 4 disques, 15 déplacements sont nécessaires, puis 31 pour 5 disques et 63 pour 6…
Vous aurez peut-être déjà reconnu le début d'une suite assez classique…

En fait, il est facile de déterminer le nombre de déplacements nécessaires. Pour déplacer $n+1$ disques, il est nécessaire d'en déplacer *n*, puis 1, puis encore *n*, d'où $u_{n+1}=u_n+1+u_n=2u_n+1$.

La suite est donc définie par les relations $\begin{cases} u_1=1 \\ \forall n\in\mathbb{N}, u_{n+1}=2u_n+1 \end{cases}$.

On peut montrer (par récurrence, ou en s'intéressant à $v_n=u_n+1$), que : $\forall n\in\mathbb{N}, u_n=2^n-1$.

5.3 *Les risques de la programmation récursive*

L'apparente simplicité de l'écriture d'un programme récursif peut en masquer la complexité d'exécution. Dans certains cas, le choix d'une programmation récursive peut s'avérer extrêmement pénalisante. Ceci est expliqué dans le **chapitre 8**, sur les suites et les séries. Voir le paragraphe 2.1

Il faut aussi faire attention au risque de « boucle sans fin » que peut présenter une écriture récursive. Reprenons l'exemple de la fonction calculant les termes d'une suite définie par récurrence :

```
Define u(n)=
Func
Local v
If n=0 then
  Return 1
Else
  Return exp(−(n−1)u(n−1))
EndIf
EndFunc
```

Que se passera-t-il si l'on demande par erreur le calcul de la valeur pour $n=-1$?
La fonction va chercher à calculer l'image de $n-1$, donc de -2, et donc celle de -3, etc…
Il ne restera plus que la solution d'appuyer sur la touche [on] pour interrompre ces calculs !

On retrouvera le même problème si on part d'une valeur non entière, comme par exemple 1.5 : la fonction a besoin de calculer l'image de 0.5, et donc celles de -0.5, -1.5, -2.5… On entre là aussi dans une suite d'appels sans fin. Il pourrait donc être prudent, même si ce n'est pas strictement indispensable de commencer à tester que l'argument donné lors de l'appel du programme est bien un entier positif… Cela peut par exemple être fait par la fonction suivante :

```
Define is_posint (n)=
Func
If getType(n)≠"NUM" then
  Return false
Else
  Return (n≥0 and int(n)=n)
EndIf
EndFunc
```

On commence par s'assurer que l'argument est bien un nombre, et, si c'est le cas, on renvoie la valeur obtenue en testant si ce nombre est positif ou nul, et s'il est égal à sa partie entière (donc entier).
Pour le premier test (est-ce bien un nombre ?), nous avons utilisé **GetType**. Si *var* désigne le nom d'une variable, l'appel de **GetType**(*var*) retournera la chaîne de caractères indiquée ci-dessous :

Nombres entiers, nombres décimaux, nombres rationnels	**"NUM"**
Nombres réels exprimés sous forme exacte et n'appartenant pas à la catégorie précédente (comme par exemple $1+\sqrt{2}$), nombres complexes, expressions mathématiques…	**"EXPR"**
Liste d'éléments	**"LIST"**
Matrices	**"MATRIX"**
Chaînes de caractères	**"STRING"**

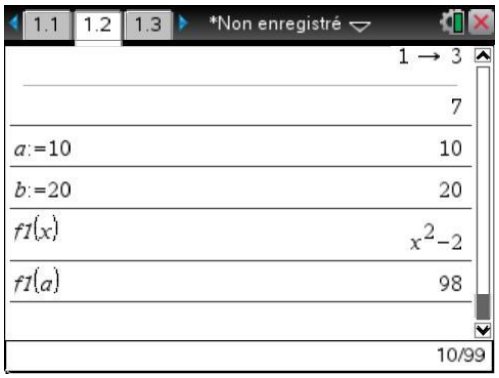
On peut à présent écrire notre fonction sous la forme :

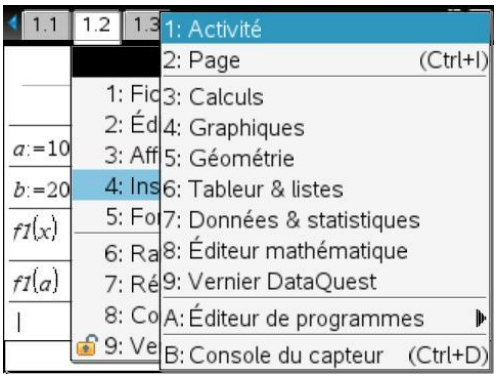
```
Define u(n)=
Func
Local v
if not is_posint(n) then
  Return "Erreur"
ElseIf n=0 then
  Return 1
Else
  Return exp(−(n−1)u(n−1))
EndIf
EndFunc
```

Dans l'exemple ci-dessus, nous avons d'abord « appris » à TI-Nspire à reconnaître les entiers positifs à l'aide d'une fonction auxiliaire, puis nous avons utilisé cette fonction pour simplifier l'écriture de la fonction principale. Ce type d'approche, où l'on décompose un traitement parfois complexe en opérations plus simples est un moyen efficace d'aborder la programmation d'un algorithme.

6. Domaine d'existence des programmes et des fonctions

Les calculatrices « ordinaires » travaillent dans un environnement unique, et toutes les variables, fonctions ou programmes qui ont été créés sont accessibles à tout instant. TI-Nspire CAS est un outil évolué qui travaille avec une logique différente. Les objets sont créés dans une *Activité* présente dans un *Classeur*, et n'existent plus en dehors[4]. Par exemple, si on crée deux variables *a* et *b* ainsi qu'une fonction dans la première activité d'un classeur, aucun de ces objets ne sera accessible dans une autre.



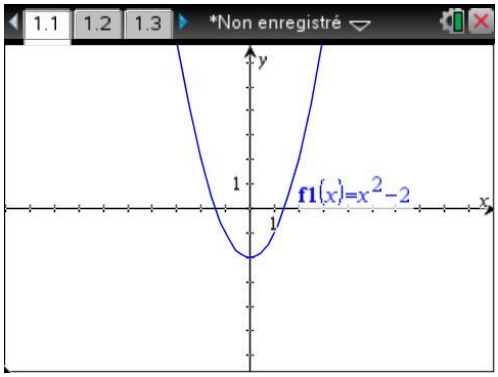


Il existe heureusement un moyen de rendre accessible un ensemble de fonctions ou de programmes utiles à tous les classeurs. La solution proposée par TI-Nspire CAS, la création de **bibliothèques de programmes**, sera étudiée dans le **chapitre 15**.

[4] Ce type d'organisation n'est pas spécifique à TI-Nspire, et tout ceci est sans doute déjà familier aux utilisateurs d'autres logiciels déjà utilisés dans les classes préparatoires, comme Maple.

7. L'utilisation de Tableur & listes

Dans de nombreux cas, l'utilisation de l'application Tableur & listes permet d'éviter l'écriture d'une fonction ou d'un programme ! Nous en avons vu un premier exemple dans le **chapitre 1**, avec le calcul des termes de la suite de Fibonacci. Pour mettre en place l'algorithme de dichotomie étudié dans ce chapitre, on ouvre un page Graphiques, et on y définit f1. On ouvre ensuite une page avec l'application Tableur & listes, et on place les deux valeurs initiales 1. et 2. dans les cellules **a1** et **b1**. Dans **c1**, on écrit **=(a1+b1)/2** et dans **d1**, on inscrit **=f1(c1)**.



La ligne suivante va contenir les formules nécessaires pour effectuer la première étape.

Oublions pour l'instant le cas particulier où l'on aurait la chance d'obtenir la bonne valeur directement (en fait, il est impossible que cela arrive ici : le nombre recherché est irrationnel).

- Si le signe de **d1** est le même que celui de **f1(a1)**, c'est-à-dire si le produit des deux est positif, on doit remplacer *a* par *c*, c'est-à-dire placer dans **a2** le contenu actuellement présent dans **c1**.
- Dans le cas contraire, on doit placer dans **a2** la valeur de *a* qui était présente dans **a1**.

Pour obtenir ce comportement, on écrit la formule suivante dans **a2** :

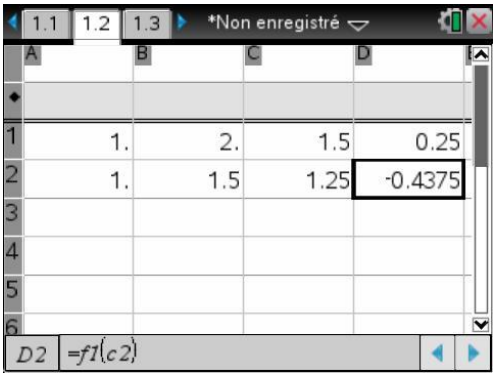
=when(f1(a1)*d1≥0,c1,a1)

On procède de manière analogue dans la cellule **b2**, où l'on écrit : **=when(f1(b1)*d1≥0,c1,b1)**

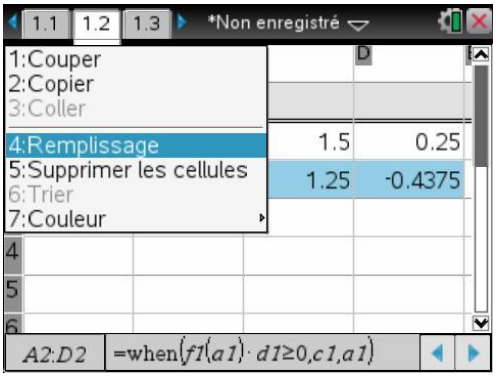
Dans la cellule **c2**, on place la formule **=(a2+b2)/2** et dans la cellule **d2**, on écrit **=f1(c2).**

On n'est pas véritablement obligé d'écrire ces deux formules. Il suffit de recopier les cellules **c1** et **d1** dans **c2** et **d2**. Comme sur tout autre tableur, le contenu initial (qui faisait référence à des cellules situées sur la ligne 1) est adapté pour faire maintenant référence à des cellules situées sur la ligne 2.

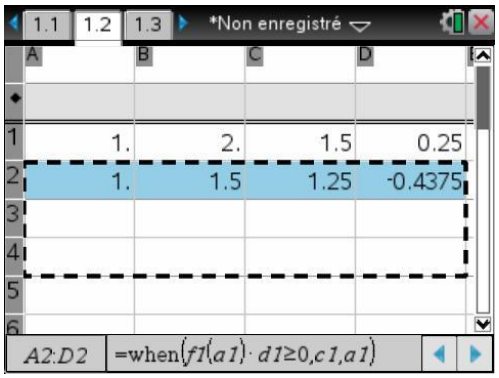
On obtient ainsi un tableau comportant les deux lignes suivantes :

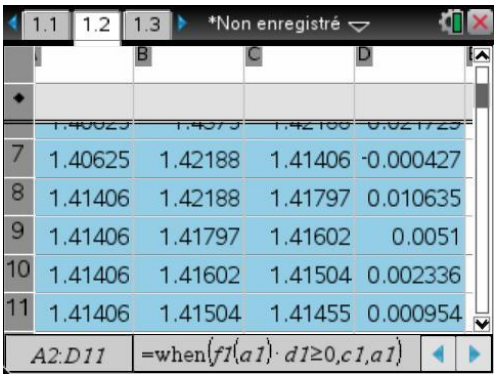


Placez le curseur en **a2**, appuyez sur la touche [⇧shift], et tout en maintenant cette touche enfoncée, déplacez-vous vers la droite de manière à sélectionner les cellules **a2** à **d2**. Ouvrez alors le menu contextuel par [ctrl] [menu], et sélectionnez **Saisie rapide**, de manière à obtenir l'écran ci-dessous :

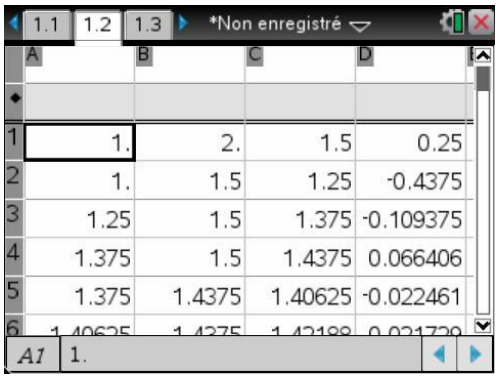


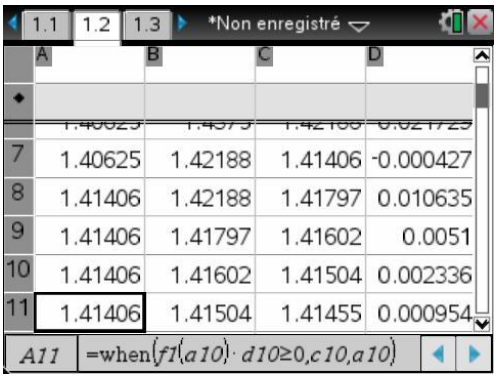
Il suffit ensuite de se déplacer vers le bas avec le TouchPad pour définir la zone dans laquelle on souhaite recopier les formules. On termine en appuyant sur le bouton central :





Remontez ensuite sur les premières lignes du tableur pour revoir les étapes de l'algorithme.

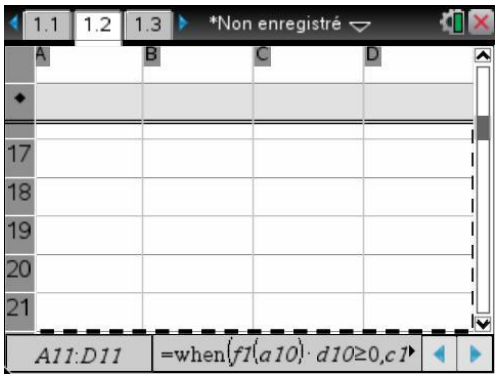


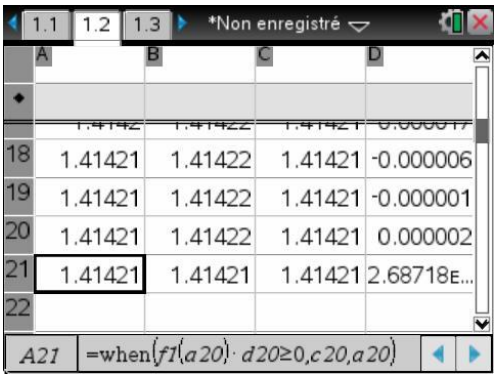


Les résultats obtenus dans les cellules des 2 premières colonnes de la ligne 11 correspondent à la fin de la dixième étape.

Ils montrent que $1.41406 < x_0 < 1.41504$.

Pour plus de précision, étendez le contenu des cellules a11, b11, c11 et d11 aux 10 lignes suivantes.

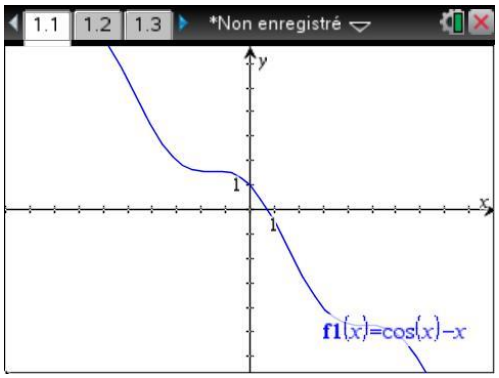


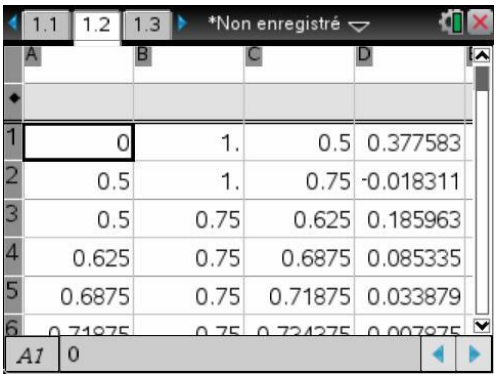


On obtient ainsi $x_0 \simeq 1.41421$

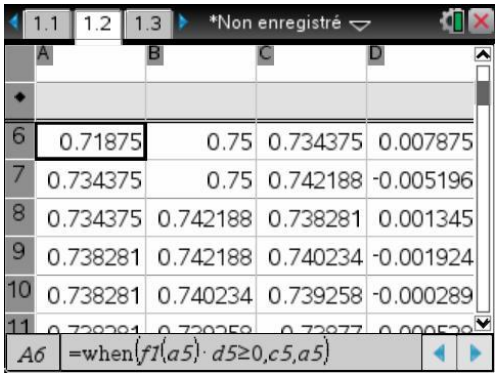
Une fois créée, cette feuille de calcul est utilisable avec toute autre fonction continue, strictement monotone sur un intervalle $[a,b]$, et prenant en *a* et en *b* des valeurs de signes opposés.

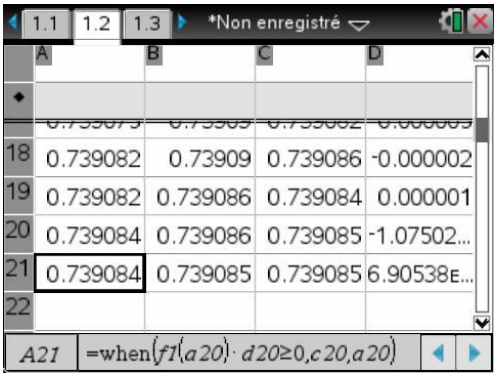
Cherchons par exemple la racine de $x-\cos(x)=0$. Compte-tenu de la représentation graphique, nous partons cette fois de l'intervalle $[0,1]$. Il suffit de modifier le contenu des cellules **a1** et **b1**. L'ensemble du tableau est mis à jour, et on peut suivre les étapes en faisant défiler les écrans vers le bas.





Comme dans l'application Calculs, utilisez ctrl 3 et ctrl 9 pour parcourir les calculs :





Annexe A
Interactions avec l'application *Graphiques & géométrie*

La lecture de cette section n'est absolument pas indispensable pour une utilisation classique de la TI-Nspire.

Son contenu s'adresse plutôt à des utilisateurs déjà expérimentés, souhaitant aller plus loin dans l'utilisation de ce produit.

A priori, la TI-Nspire ne dispose pas d'instructions permettant de faire des constructions graphiques. Il semble donc difficile de pouvoir parvenir au résultat annoncé dans le titre !

Il est cependant possible d'avoir certaines interactions en utilisant les idées suivantes :

- Si une courbe représentant une fonction a été construite dans un écran Graphiques & géométrie, alors il sera possible de modifier le tracé de cette courbe en redéfinissant cette fonction depuis un programme.
- Si un nuage de points, isolés ou reliés par des segments de droites, est défini par 2 listes, et est représenté dans Graphiques & géométrie, alors il sera possible d'agir sur cette représentation en redéfinissant le contenu de ces deux listes.
- Il est possible de lier les variables définissant les valeurs extrêmes sur les axes, ou les graduations. Il sera possible de modifier ces valeurs depuis un programme, et donc de modifier le cadrage.

Nous allons à présent voir un programme utilisant ces différentes méthodes.

Celui-ci est destiné à illustrer la méthode des rectangles.

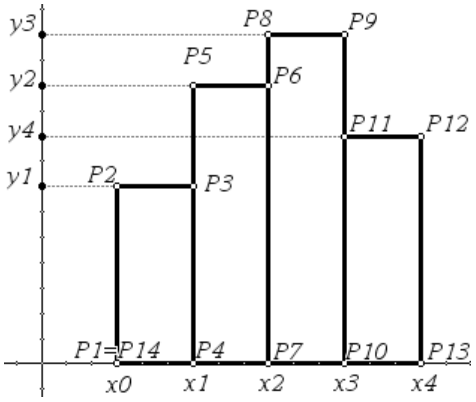
L'appel de ce programme devra provoquer une mise à jour de l'écran Graphiques & Géométrie, afin d'afficher la fonction que l'on souhaite étudier, ainsi que les rectangles utilisés.

Sur une TI-89 ou une Voyage 200, on utiliserait des instructions graphiques spécialisées pour faire ce travail. La méthode que l'on va utiliser avec la TI-Nspire, qui ne dispose pas de ces instructions, est radicalement différente.

On va construire ces rectangles sous la forme d'un *nuage de points* reliés par des segments de droites.

Pour cela, nous devons déterminer les coordonnées des sommets, et les placer dans deux listes (une pour les abscisses, l'autre pour les ordonnées).

Voici un exemple, avec 4 rectangles, permettant de comprendre la nature de ces listes.



Le tableau ci-dessous donne les coordonnées des différents points :

$$\begin{bmatrix} x_0 & x_0 & x_1 & x_1 & x_1 & x_2 & x_2 & x_2 & x_3 & x_3 & x_3 & x_4 & x_4 & x_0 \\ 0 & y_1 & y_1 & 0 & y_2 & y_2 & 0 & y_3 & y_3 & 0 & y_4 & y_4 & 0 & 0 \end{bmatrix}$$

Le nuage de points sera donc défini par les listes :

$l1 = \{x_0, x_0, x_1, x_1, x_1, x_2, x_2, x_2, x_3, x_3, x_3, x_4, x_4, x_0\}$

$l1 = \{0, y_1, y_1, 0, y_2, y_2, 0, y_3, y_3, 0, y_4, y_4, 0, 0\}$

Voici à présent un programme qui va se charger de la construction de ces listes, et qui définira également l'expression que l'on souhaite placer dans **f1**.

```
Define rectangles(ex,a,b,n)=
Prgm
Local h,lx,ly
©Définition de la fonction f1
expr("Define f1(x)="&string(ex))
©Calcul du pas de la subdivision
h :=(b-a)/n
©Construction de la liste des valeurs Xk de la subdivision
lx:=seq(a+k*h,k,0,n)
©Liste des images des Xk pour k compris entre 0 et n-1
ly:=f1(seq(a+k*h,k,0,n-1))
©Construction de la liste des abscisses des points
l1:={lx[1],lx[1]}
For i,2,n+1
l1:=augment(l1,{lx[i],lx[i],lx[i]})
EndFor
©Construction de la liste des ordonnées des points
l2:={}
For i,1,n
l2:=augment(l2,{0,ly[i],ly[i]})
EndFor
l2:=augment(l2,{0,0})
EndPrgm
```

Voici quelques explications sur les instructions utilisées par ce programme.

1. Définition de f1

La première instruction, qui fait appel à **expr** et à **string**, est plutôt mystérieuse ! Supposons par exemple que l'on entre la commande **rectangles(cos(x),0,1,10)**. La variable *ex* contiendra donc l'expression $\cos(x)$.

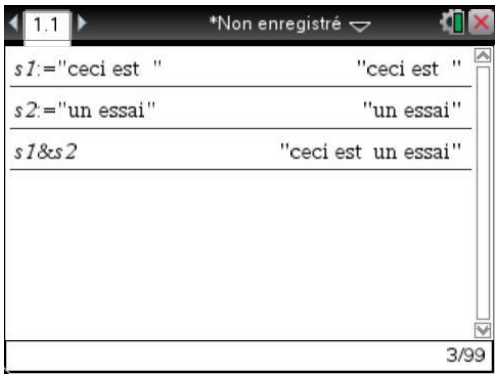
La fonction **string** opère la conversion en une chaine de caractères, l'utilisation de **string(ex)** permet donc d'obtenir la chaine de caractères **"cos(x)".**

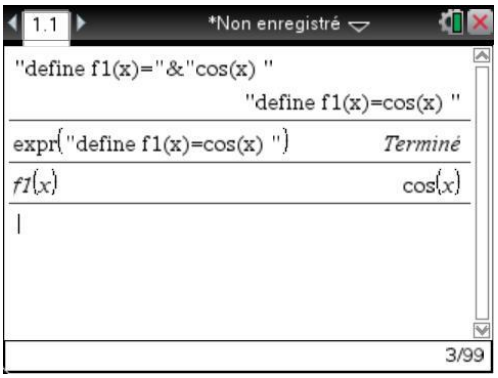
L'opérateur **&** permet de rassembler deux chaines de caractères en une chaine unique.

"Define f1(x)="&string(ex)

construit la chaine de caractères "Define f1(x)=cos(x))".

La commande **expr** permet ensuite d'évaluer le contenu de cette chaine de caractères : tout se passe comme si on entrait directement la commande qu'elle contient : **Define f1(x)=cos(x)**.

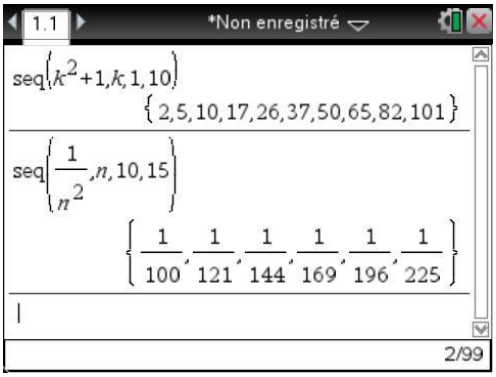




En résumé, on a bien défini la fonction f1 à partir du premier argument transmis au programme.

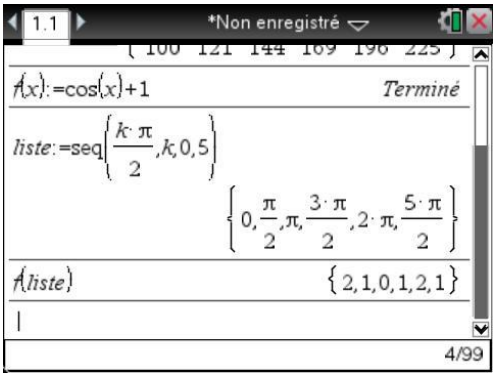
2. Liste des x_i

On utilise le constructeur **seq(***expression*, *var*, *val1*, *val2***)** qui permet de construire la liste définie par *expression* dépendant de la variable *var* quand *var* varie entre *val1* et *val2* avec un pas de 1.



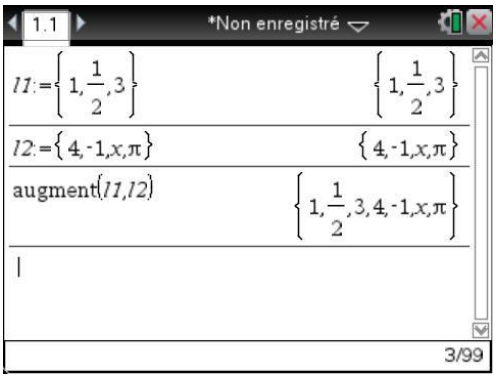
3. Liste des y_i

TI-Nspire permet le calcul direct de l'image d'une liste. En particulier **f1**(*liste*) retourne la liste des images des éléments de *liste*.



4. Construction des listes des abscisses et des ordonnées des points P_i

L'utilisation de **augment(***liste1*,*liste2***)** retourne la liste obtenue en ajoutant les éléments de *liste2* à la suite de ceux de *liste1*.



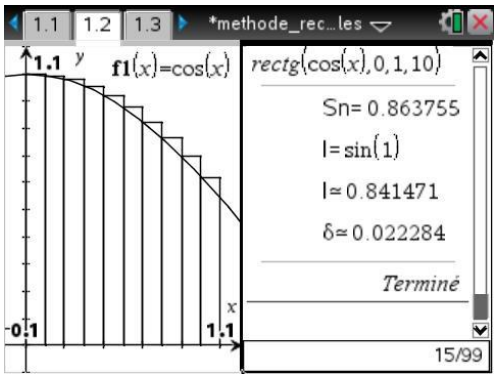
Nous allons maintenant pouvoir construire un classeur illustrant la méthode des rectangles.

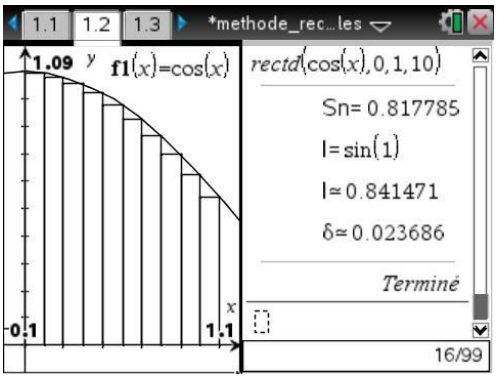
1. On définit le programme ci-dessus, en y ajoutant les instructions permettant de calculer la somme $h \cdot \sum_{k=0}^{n-1} f(x_k)$, avec $h = \frac{b-a}{n}$ ainsi que la valeur de l'intégrale.
2. On insère une page Calculs, et on y lance une première fois l'exécution du programme.
3. On partage l'écran en 2 et on ouvre l'application Graphiques.
4. Dans cette application Graphiques, on demande la représentation de la fonction f1. On obtient ainsi la représentation de la fonction définie lors de l'appel du programme.
5. On choisit le mode « Nuages de points », et on demande la représentation des listes l1 et l2.
6. On obtient un nuage de points non connectés. On sélectionne ce nuage, affiche le menu contextuel, et choisit **Attributs**. Le premier attribut permet de contrôler la taille des points (choisir la plus petite), le réglage du second attribut permet de choisir de les relier.
7. On a ainsi obtenu une première construction de la courbe et de la première série de rectangles dans ce classeur. On peut le sauver en vue d'un usage ultérieur.
8. Par la suite, il suffira d'ouvrir à nouveau ce classeur, et de relancer le programme **rectangles** avec les arguments souhaités. Tous les éléments de la partie graphique seront alors reconstruits : la courbe, et les rectangles associés.

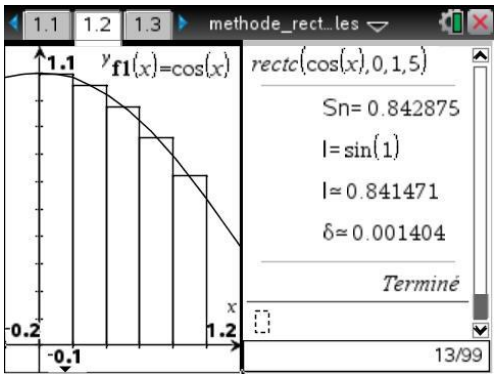
Vous retrouverez cette activité téléchargeable avec ce chapitre. La version proposée permet de visualiser le calcul des rectangles définis à partir de la valeur de l'image de l'extrémité gauche ou droite de l'intervalle $[x_i, x_{i+1}]$, ou de l'image du centre de cet intervalle.

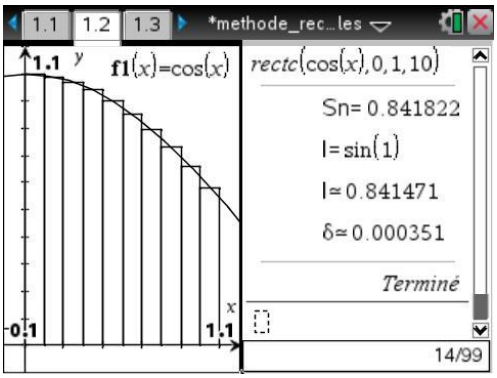
On peut constater expérimentalement que l'approximation est bien meilleure dans ce dernier cas[5].

[5] Une étude théorique permet de montrer que, si la fonction vérifie certaines hypothèses, l'erreur est en $\frac{A}{n}$ si on choisit l'extrémité gauche ou droite, et en $\frac{B}{n^2}$ si on choisit les centres des intervalles.
En particulier, à partir d'une certaine valeur de *n*, si on double le nombre de points, l'erreur est approximativement divisée par 2 dans le premier cas, et par 4 dans le second.



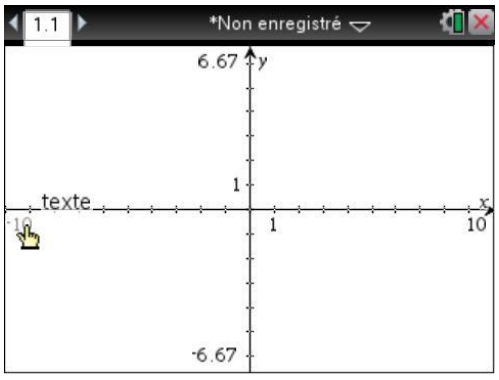


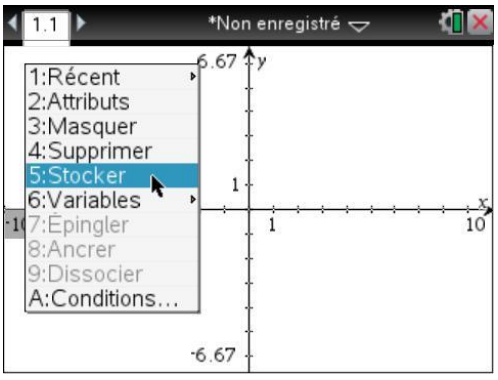


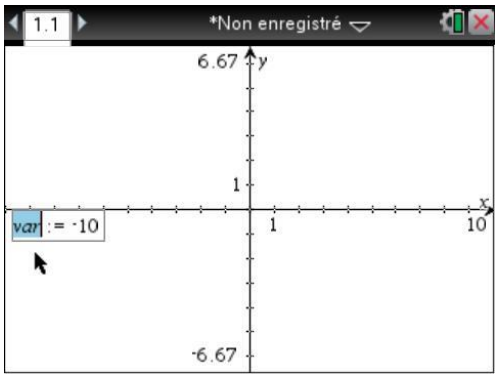


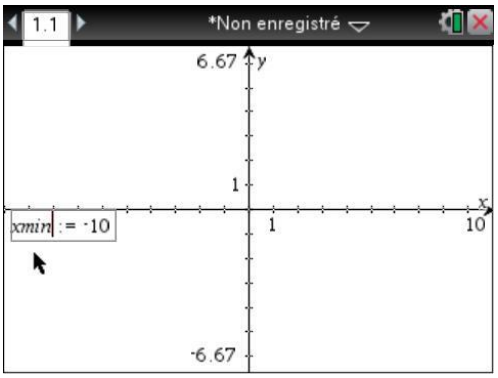
Ces programmes sont capables d'adapter automatiquement le cadrage. La méthode utilisée est décrite sur la page suivante.

1. Dans l'écran Graphiques avec lequel on souhaite interagir, on se place successivement sur chacune des 4 valeurs indiquant les valeurs minimales et maximales sur chaque axe. Si ces valeurs ne sont pas affichées, placer le curseur sur l'un des axes, ouvrir le menu contextuel accessible par [ctrl] [menu], sélectionner **Attributs**, et modifier la valeur du dernier, qui permet de masquer ou d'afficher les valeurs extrêmes.
2. Pour chacune d'elles, on ouvre le menu contextuel. On utilise ensuite l'option **Stocker** ce qui permet de placer des valeurs dans des variables que l'on pourra par exemple nommer **xmin**, **xmax**, **ymin** et **ymax**.

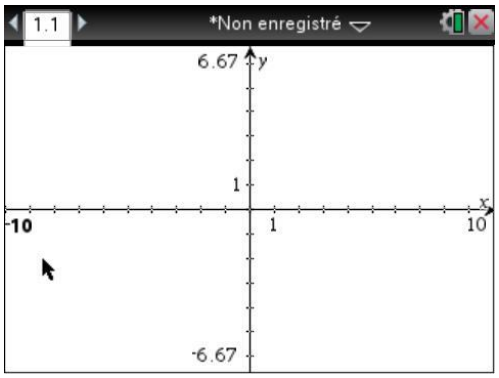


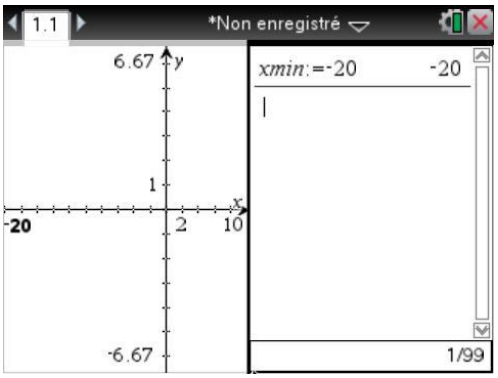






3. Une fois la liaison effectuée, le nombre sélectionné passe en gras, et on peut vérifier que tout se passe bien en modifiant ces différentes valeurs depuis un écran Calculs.





Une fois que cette liaison a été établie, il devient possible de piloter le cadrage depuis un programme en modifiant la valeur de ces variables. C'est la méthode qui est utilisée dans le classeur methode_rectangles.tns que vous pourrez télécharger avec ce chapitre.

Annexe B. Syntaxes TI-Nspire CAS et Maple[6]

Définition d'une fonction

TI-Nspire CAS	Maple
Define f(var1, var2, …) = **func** **local** *var1, var2, var3…* *instruction*$_1$ *instruction*$_2$ … *instruction*$_k$ **EndFunc**	**f:=proc(var1, var2, …)** **local** *var1, var2, var3…* **;** *instruction*$_1$ **;** *instruction*$_2$ **;** … *instruction*$_k$ **End**
Return *val*	**RETURN** (*val*)

Affichage d'un ou plusieurs résultats

TI-Nspire CAS	Maple
Disp *expr1, expr2, expr3…*	**Print** (*expr1, expr2, expr3…*)

Structure de boucles

TI-Nspire CAS	Maple
Loop *instruction*$_1$ … *instruction*$_k$ **Endloop**	**Do** *instruction*$_1$ **;** … *instruction*$_k$ **od**
For *var, début, fin* *instruction*$_1$ … *instruction*$_k$ **EndFor**	**for** *var* **from** *début* **to** *fin* **do** *instruction*$_1$ **;** … *instruction*$_k$ **od**
For *var, début, fin, pas* *instruction*$_1$ … *instruction*$_k$ **EndFor**	**for** *var* **from** *début* **to** *fin* **by** *pas* **do** *instruction*$_1$ **;** … *instruction*$_k$ **od**
While *condition* *instruction*$_1$ … *instruction*$_k$ **EndWhile**	**while** *condition* **do** *instruction*$_1$ **;** … *instruction*$_k$ **od**
For *var, début, fin, pas* **if not** *condition* : **Exit** *instruction*$_1$ … *instruction*$_k$ **EndFor**	**for** *var* **from** *début* **to** *fin* **by** *pas* **while** *condition* **do** *instruction*$_1$ **;** … *instruction*$_k$ **od**

[6] Maple est une marque déposée de Waterloo Maple Inc.

Structure conditionnelles

TI-Nspire CAS	Maple

if *condition* **then**
instruction$_1$
instruction$_2$
…
instruction$_k$
EndIf
	if *condition* **then**
	instruction$_1$ **;**
	instruction$_2$ **;**
	…
	instruction$_k$
	fi

if *condition* **then**
instruction$_1$
…
instruction$_k$
Else
instruction$_1$
…
instruction$_k$
EndIf
	if *condition* **then**
	instruction$_1$ **;**
	…
	instruction$_k$
	else
	instruction$_1$ **;**
	…
	instruction$_k$
	fi

if *condition*$_1$ **then**
instruction$_1$ …
…
instruction$_k$
ElseIf *condition*$_2$ **then**
instruction$_1$
…
instruction$_k$
ElseIf …
…
Else
instruction$_1$
…
instruction$_k$
EndIf
	if *condition*$_1$ **then**
	instruction$_1$ **;** …
	…
	instruction$_k$
	elif *condition*$_2$ **then**
	instruction$_1$ **;**
	…
	instruction$_k$
	elif …
	…
	else
	instruction$_1$ **;**
	…
	instruction$_k$
	fi

Fonctions et programmes

Dans Maple, il n'y a pas de distinction entre *fonctions* et *programmes*. On définit des procédures qui retournent une valeur, et qui peuvent agir sur des variables globales, sous réserve que celles-ci figurent dans une instruction **global** *var1, var2, var3…* **;**

Si on ne souhaite pas qu'une valeur soit retournée et affichée lors de la fin d'exécution de la procédure quand elle est directement utilisée dans une feuille de calcul, il faut retourner la valeur NULL.

Voici, juste à titre d'exemple, une procédure Maple, affichant la liste des carrés des entiers de 1 à *n*, mémorisant dans la variable globale **sc** la somme de ces entiers et ne retournant pas de valeur, ainsi que le programme équivalent pour TI-Nspire CAS.

```
Define u(n)= Prgm
Local i
sc:= 0
For i,1,n
disp i^2
sc:=sc+i^2
EndFor
EndPrgm
```

```
carres:=proc(n)
local i;
global sc;
sc:= 0;
for i from 1 to n do
print (i^2);
sc:= sc+i^2;
od;
RETURN(NULL)
end:
```