Chapitre 14

Programmation sur TI-Nspire CAS

Ce chapitre permet de découvrir les fonctionnalités de la programmation sur TI-Nspire CAS. Il est complété par le **chapitre 15** qui donne des informations sur les bibliothèques de programmes.

Sommaire

Philippe Fortin (Lycée Louis Barthou – Pau) / Roland Pomès (Lycée René Cassin – Bayonne)

1. Introduction

Les utilisateurs déjà habitués à la programmation sur calculatrice seront sans doute un peu surpris par les possibilités offertes par la version 1.4 (été 2008) de la TI-Nspire CAS. On dispose d'un outil très performant, mais dans le même temps on ne retrouve pas certaines instructions plutôt classiques sur les calculatrices comme par exemple **Input** ou **Prompt**, ce qui peut nécessiter un petit apprentissage.

TI-Nspire CAS rend possible la construction de fonctions ou de programmes très évolués, comme on peut le voir par exemple dans le **chapitre 11** sur les séries de Fourier. En particulier, **ces programmes pourront utiliser toutes les ressources du calcul formel, tout en étant capables d'interagir avec le contenu d'un écran graphique**[1].

Le contenu de ce chapitre décrit l'utilisation de la version 1.4, sortie pendant l'été 2008. Il est important de le préciser car il est probable que la programmation sur TI-Nspire CAS sera encore amenée à évoluer dans les mois ou années à venir, et que d'autres fonctionnalités lui seront ajoutées. Des mises à jour de ce chapitre seront publiées si nécessaire lors de la sortie de ces futures versions.

La version 1.4 de TI-Nspire permet déjà de couvrir les besoins du programme des classes scientifiques, par contre il est certain qu'elle ne permet pas (pas encore !) l'écriture d'un jeu interactif utilisant toutes les ressources de l'affichage graphique…

2. Un exemple d'algorithme

Connaissez-vous la méthode de résolution d'une équation par dichotomie ? On considère une fonction f continue, strictement monotone sur un segment $[a,b]$, telle que $f(a)$ et $f(b)$ soient non nuls, et de signes différents. Il s'agit donc d'une fonction strictement croissante telle que $f(a)<0$ et $f(b)>0$, ou d'une fonction strictement décroissante telle que $f(a)>0$ et $f(b)<0$.

Dans ce cas, il existe une unique valeur $x_0 \in]a,b[$ telle que $f(x_0)=0$.

Comment la trouver ? L'idée est de tester ce qui se passe au point $c=\dfrac{a+b}{2}$ afin de réduire la taille de l'intervalle dans lequel se trouve cette solution.

- Si on a beaucoup de chance, $f(c)=0$ et notre problème est résolu !
- Dans le cas contraire, on peut être dans l'une des deux situations suivantes :
  - $f(a)$ et $f(c)$ sont de signes opposés, et $x_0 \in]a,c[$
  - $f(a)$ et $f(c)$ sont de même signes, et $x_0 \in]b,c[$

Le nouvel intervalle dans lequel se trouve x_0 ($[a,c]$ ou $[c,b]$) est deux fois plus petit que $[a,b]$.

Si on réitère le procédé, on obtiendra un intervalle dont la taille sera à nouveau divisée par 2 (et donc 4 fois plus petit que l'intervalle de départ), la fois suivante, la taille aura été divisée par 8.

Après n étapes, on aura trouvé x_0 (si on a eu de la chance), ou on pourra affirmer qu'il se trouve dans un intervalle dont l'amplitude sera égale à $\dfrac{|b-a|}{2^n}$

[1] Ce dernier point est traité en détail à la fin de ce chapitre.

En seulement 10 étapes, l'intervalle de recherche initial aura été remplacé par un intervalle $2^{10} = 1024$ fois plus petit. En 10 étapes supplémentaires, on aura un intervalle à nouveau divisé en 1024. Par exemple, si l'écart initial entre a et b était égal à 1, on est certain d'avoir en 20 étapes un encadrement de x_0 à 10^{-6} près.

Voici un exemple d'utilisation de cet algorithme.

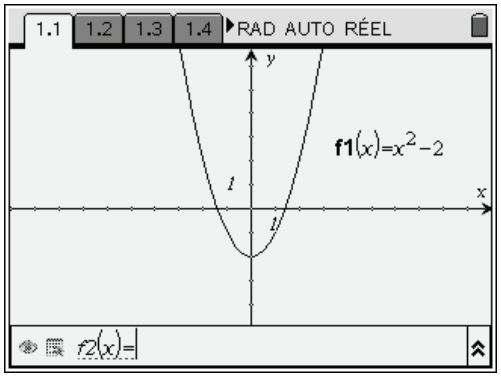
Nous recherchons ici une valeur approchée de $\sqrt{2}$. C'est la solution positive de l'équation $x^2 - 2 = 0$. On considère donc $f(x) = x^2 - 2$. Cette fonction est croissante sur $\mathbb{R}^+$. $f(1) < 0$ et $f(2) > 0$.
On a donc $x_0 \in]1, 2[$. Voici les premières étapes de l'algorithme décrit ci-dessus.

a	b	c	Signe de $f(c)$	Conclusion
1	2	1.50	$1.50^2 - 2 = 0.25 > 0$	$1 < x_0 < 1.5$
1	1.5	1.25	$1.25^2 - 2 = -0.4375 < 0$	$1.25 < x_0 < 1.5$
1.25	1.5	1.375	$1.375^2 - 2 = -0.109375 < 0$	$1.375 < x_0 < 1.5$
1.375	1.5	1.4375	$1.4375^2 - 2 = 0.066406 > 0$	$1.375 < x_0 < 1.4375$

3. L'écriture d'un programme

Nous allons voir à présent comment cela peut être traité par un programme.

Nous supposerons ici que la fonction à utiliser a préalablement été définie dans f1. Cela peut par exemple se faire dans l'écran Graphiques & géométrie.



3.1 *Transmission des arguments*

Avant d'aller plus loin, il peut être nécessaire de préciser un point. Nous devons communiquer au programme les deux valeurs initiales à utiliser pour *a* et *b*. Il n'existe pas de moyen de demander ces valeurs au cours de l'exécution de celui-ci, car des instructions comme **Input** ou **Prompt** ne sont pas disponibles dans les versions actuelles de TI-Nspire.

On va donc utiliser une autre méthode : le passage de ces valeurs en arguments lors de l'appel.

Par exemple, pour commencer l'algorithme avec les valeurs 1 et 2, si le nom de notre programme est par exemple **dicho**, nous entrerons la commande **dicho(1,2)**.

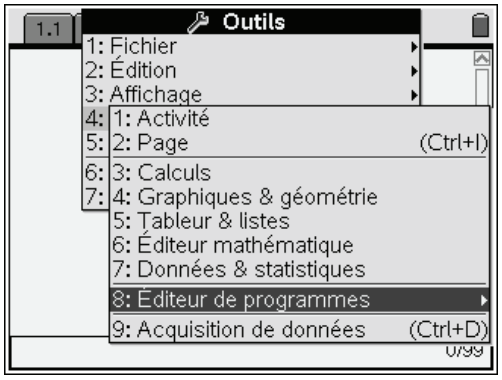
On retrouve ce mode de fonctionnement dans un logiciel comme Maple™, couramment utilisé en classe prépa. Cela ne constitue en rien une particularité de TI-Nspire CAS.

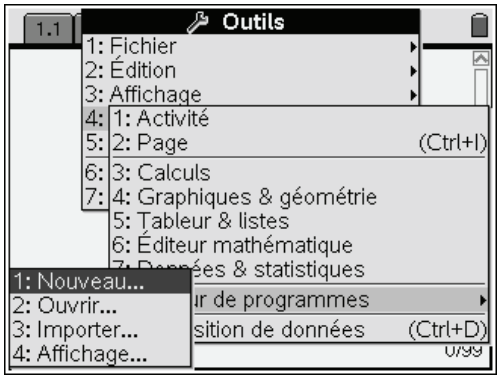
Ce point étant éclairci, nous pouvons passer à la suite !

3.2 *Ouverture de l'éditeur de programme sur une nouvelle page*

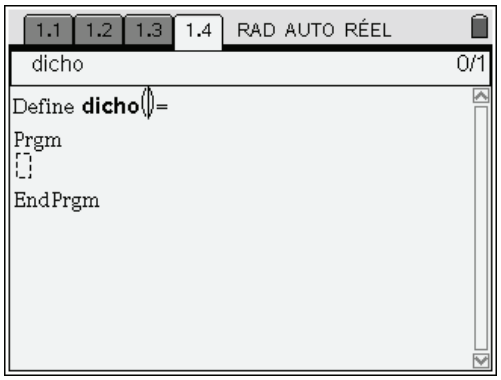
Commençons par ouvrir l'éditeur de programmes. Lorsque l'on travaille sur l'unité nomade, et non sur l'ordinateur, il est souvent préférable d'utiliser cet éditeur sur une pleine page (mais il est aussi possible de le faire dans un partage d'écran – voir paragraphe 3.3, page 5).

Nous allons donc insérer une première page avec l'application Calculs, puis à partir de cette page, utiliser le menu **Outils**, accessible par (ctrl) (⌂) pour insérer une autre page avec l'éditeur de programmes.

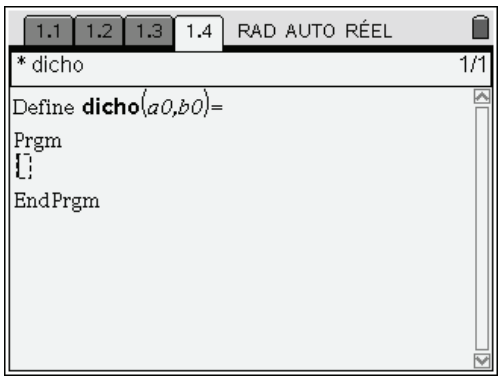




Saisir le nom choisi. Nous verrons par la suite à quoi correspondent les deux rubriques suivantes.



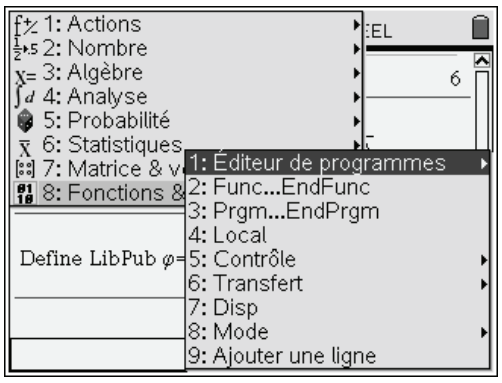
Nous devons ensuite indiquer le nom des paramètres utilisés lors de l'appel de ce programme. Ici, il y en a deux, contenant les valeurs initiales de *a* et *b*. Nous pouvons choisir les noms **a0** et **b0**.

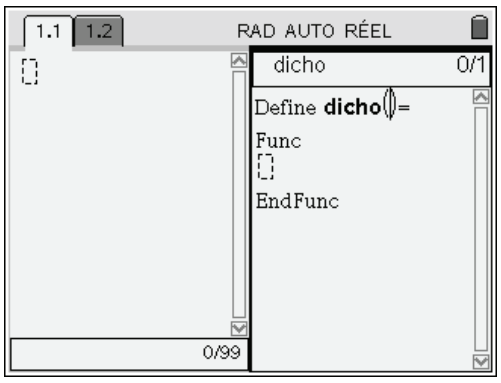


Observer la présence du caractère * à droite du nom **dicho** dans le haut de l'écran. Il indique que le contenu du programme a été modifié, mais n'a pas été encore sauvegardé dans le classeur.
Si vous souhaitez le faire dès maintenant, appuyez sur (ctrl)(B) (B : build).
Cela provoquera un contrôle de la syntaxe utilisée et la sauvegarde si tout est en ordre. Dans le cas contraire, on obtient un message d'erreur. En cas d'oubli du raccourci à utiliser, appuyer sur la touche (menu), puis utiliser le second choix **2:Vérifier la syntaxe et enregistrer**.

3.3 *Ouverture de l'éditeur de programme dans un partage d'écran*

On peut aussi ouvrir l'éditeur de programme depuis l'application Calculs, en provoquant un partage d'écran. Appuyer sur la touche (menu) puis sélectionner **Fonctions et programmes**, **Éditeur de programmes**. On obtient ensuite la même boîte de dialogue que précédemment, puis l'éditeur de programme s'ouvre dans un partage d'écran.

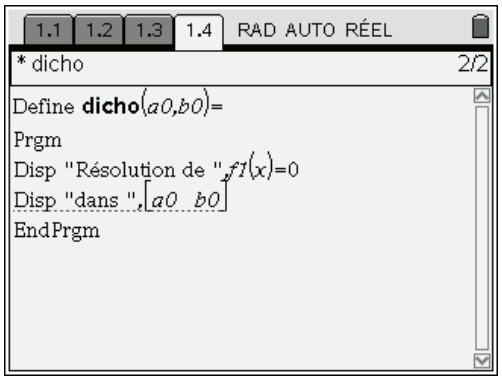


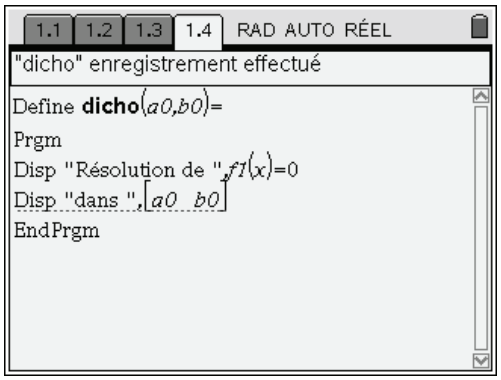


3.4 *Affichage d'un texte ou d'un résultat*

L'instruction **Disp** permet d'afficher un texte (à placer entre guillemets) ou une expression. Il est possible d'afficher plusieurs éléments dans une même instruction : il suffit de les séparer par une virgule.

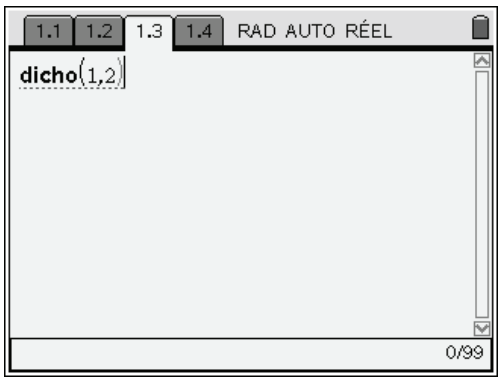
L'écran de gauche montre ce que l'on obtient après la saisie (avec le symbole * indiquant que le programme a été modifié), l'écran de droite est obtenu après avoir utilisé (ctrl) (B).

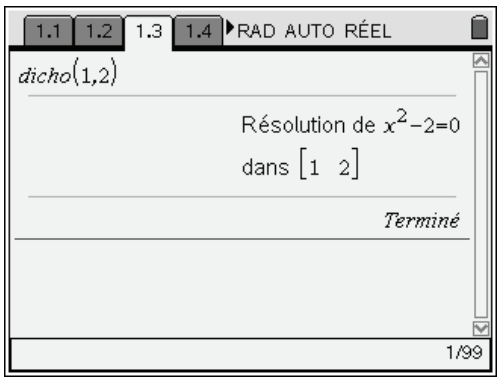




Pour entrer cette instruction, on peut la saisir directement ou appuyer sur la touche (menu), puis utiliser le second choix **6:E/S** (entrées / sorties), qui contient pour l'instant seulement l'instruction **Disp**.

Revenons sur la page précédente (Calculs) pour faire un premier essai :

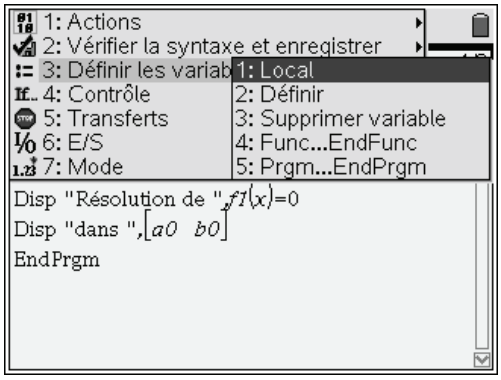


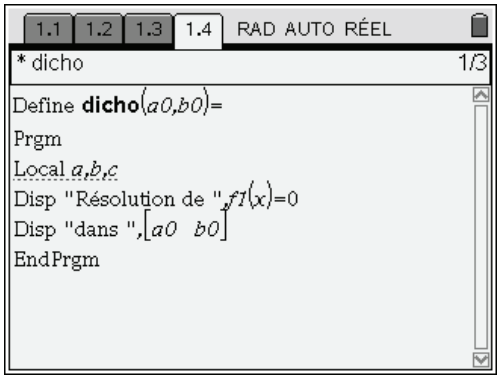


Jusqu'ici, tout va bien… Il nous reste à « apprendre » à notre programme à faire cette résolution.

3.5 *Variables locales*

Lorsque l'on construit un programme, on doit décider des variables que nous allons utiliser, et les indiquer au programme. Cela se fait au moyen de l'instruction **Local**. Ici, nous allons utiliser 3 variables : *a* et *b* seront les bornes de l'intervalle, et *c* le point situé au centre de celui-ci.





Le programme pourrait fonctionner sans cette instruction **Local**, mais dans ce cas toutes les opérations faites sur *a*, *b* et *c* affecteraient les variables de même nom pouvant exister par ailleurs dans cette activité, à l'extérieur de ce programme.

On dit dans ce cas que le programme manipule les **variables globales** *a*, *b* et *c*.

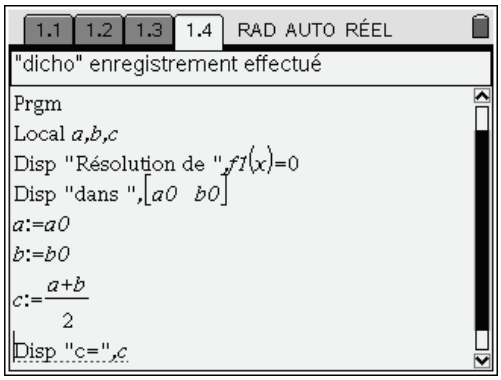
Un utilisateur de ce programme, ne connaissant pas le nom des variables choisies à l'intérieur de celui-ci risquerait donc de modifier accidentellement des valeurs mémorisées dans *a*, *b* et *c* en lançant un appel au programme **dicho**.

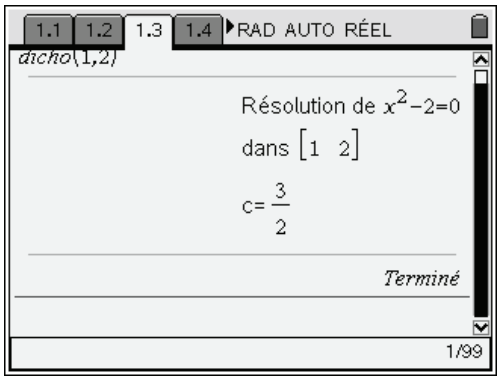
A l'inverse, si on utilise cette instruction **Local**, les variables *a*, *b* et *c* deviennent des **variables locales** à ce programme. Elles n'existent que pendant son exécution, et leur valeur n'a aucune influence sur celle des variables globales de même nom pouvant exister par ailleurs dans l'activité.

3.6 *Premières affectations*

La première chose à faire consiste à placer dans *a* et *b* les valeurs transmises en paramètres à ce programme. Nous allons ensuite calculer la valeur de *c*, et l'afficher.

Nous pouvons tester ces premières modifications (ne pas oublier d'utiliser ctrl B pour les valider)





3.7 *Structures conditionnelles*

TI-Nspire permet l'utilisation des structures suivantes :

If *Condition* **Then**
    $Instruction_1$
    ...
    $Instruction_n$
EndIf

Dans le cas où une seule instruction doit être exécutée lorsque la condition est vérifiée, il est possible d'utiliser la syntaxe abrégée :

If *Condition* **:** *Instruction*

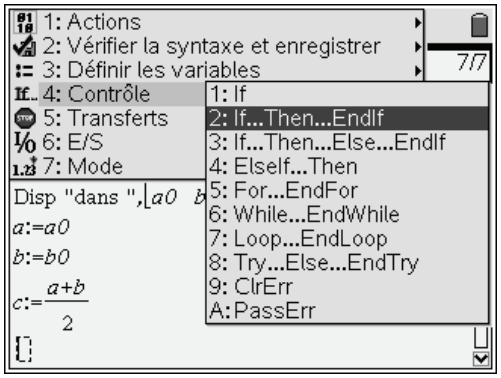
Dans le cas où un traitement spécifique doit être réservé au cas où la condition n'est pas vérifiée, on utilise la structure

If *Condition* **Then**
    $Instruction_1$
    ...
    $Instruction_n$
Else
    $Autre\text{-}Instruction_1$
    ...
    $Autre\text{-}Instruction_n$
EndIf

Enfin, dans le cas où plusieurs cas sont possibles, on peut utiliser la structure :

If *Condition* **Then**
    $Instruction_1$
    ...
    $Instruction_n$
ElseIf *AutreCondition* **Then**
    $Autre\text{-}Instruction_1$
    ...
    $Autre\text{-}Instruction_n$
ElseIf *AutreCondition* **Then**
    $Autre\text{-}Instruction_1$
    ...
    $Autre\text{-}Instruction_n$
...
Else
    $Autre\text{-}Instruction_1$
    ...
    $Autre\text{-}Instruction_n$
EndIf

On trouvera ces différentes structures dans le menu **Contrôle** :



L'utilisation de ces modèles est très pratique car elle insère directement l'ensemble du bloc et donc représente un réel gain de temps par rapport à l'utilisation du clavier de l'unité nomade. Elle évite aussi les erreurs qui peuvent se produire quand on oublie l'un des éléments (comme par exemple un **EndIf**). Pour entrer la construction avec **ElseIf**, on utilise le modèle **If**…**Then**…**Else**…**EndIf** puis on insère autant de blocs **ElseIf**… **then** que nécessaire.

Pour notre programme, il y a 3 cas possibles. Nous devons calculer l'image de c, puis,

- Si $f(c)=0$, alors c'est terminé
- Si le signe de $f(c)$ est le même que celui de $f(a)$, alors on remplace a par c.
- Si le signe de $f(c)$ est le même que celui de $f(b)$, alors on remplace b par c.

☞ *Comment tester que $f(x)$ et $f(y)$ sont de même signe ? Une méthode possible consiste à faire le produit, puis de tester si ce produit est bien positif.*

Les lignes suivantes de notre programme pourraient donc s'écrire :

```
If f1(c)=0 then
  Disp c, " est solution !"
ElseIf f1(a)f1(c)>0 then
  a:=c
Else
  b:=c
EndIf
Disp "[a,b]=",[a,b]
```

3.8 *Structures itératives*

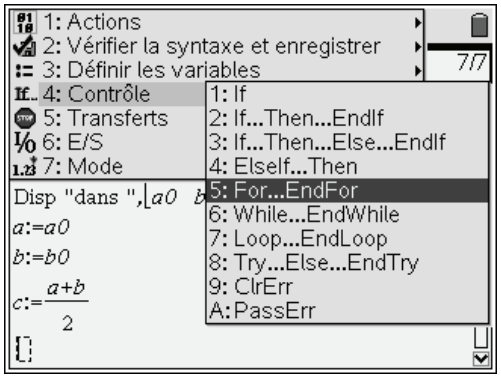
En fait, on ne veut pas faire l'opération qu'une seule fois, on veut la répéter, jusqu'à ce que l'intervalle fasse une taille suffisamment réduite.

Une première approche consiste à fixer le nombre d'étapes à effectuer. Par exemple, on pourrait décider que le programme fera systématiquement 30 itérations.

Dans ce cas, on peut utiliser la structure suivante :

For *compteur*, *début*, *fin*
  $Instruction_1$
  ...
  $Instruction_n$
EndFor

On trouvera également cette structure dans le menu **Contrôle** ce qui en facilite la saisie sur l'unité nomade.



Par exemple, si nous souhaitons faire 30 itérations, nous pouvons écrire :

```
For i,1,30
 c:=(a+b)/2
 If f1(c)=0 then
  Disp c, " est solution !"
 ElseIf f1(a)f1(c)>0 then
  a:=c
 Else
  b:=c
 EndIf
 Disp "[a,b]=",[a,b]
EndFor
```

Attention, si on choisit cette option, on introduit une nouvelle variable *i* qu'il faut penser à rajouter à la liste de nos variables locales. Il reste à régler un autre point : lorsque l'on a la chance de trouver la bonne solution, il est bien sûr inutile de continuer, et il faut sortir immédiatement de la boucle.

Cela peut se faire à l'aide de l'instruction **Exit**, que vous trouverez dans le menu **Transferts**.

On va donc écrire :

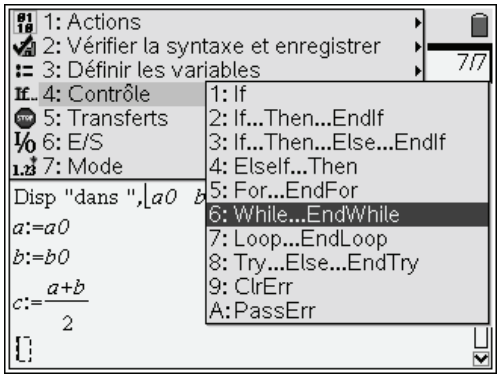
```
For i,1,30
 c:=(a+b)/2
 If f1(c)=0 then
  Disp c, " est solution !"
  Exit
 ElseIf f1(a)f1(c)>0 then
  a:=c
 Else
  b:=c
 EndIf
 Disp "[a,b]=",[a,b]
EndFor
```

Une autre approche consiste à continuer à répéter les opérations tant qu'une condition est vérifiée (taille de l'intervalle trop grande et solution non trouvée…).

Dans ce cas, on utilise la structure suivante :

While *condition*
    $Instruction_1$
    ...
    $Instruction_n$
EndWhile

On trouvera cette structure dans le menu **Contrôle** :



La condition peut s'exprimer avec les connecteurs logiques habituels **and**, **or**, **not**…
Rappelons également que la TI-Nspire CAS permet également d'utiliser directement des conditions s'exprimant sous la forme d'une encadrement, comme par exemple $v1 \le x < v2$.
Sur l'unité nomade, les symboles ≤, ≠ et ≥ s'obtiennent par (ctrl)(<), (ctrl)(=) et (ctrl)(>).

Ici, on continue tant que $|b-a| > \varepsilon$

On va donc utiliser les lignes suivantes :

```
While abs(b-a)>e
 c:=(a+b)/2
 If f1(c)=0 then
  Disp c, " est solution !"
  a:=c
  b:=c
 ElseIf f1(a)f1(b)>0 then
  a:=c
 Else
  b:=c
 EndIf
 Disp "[a,b]=",[a,b]
EndWhile
```

Lorsque l'on a la chance que c soit solution, on remplace a et b par celle-ci, et la condition $|b-a| > \varepsilon$ n'est plus satisfaite. Il devient donc inutile d'utiliser **Exit** pour mettre fin à la boucle.

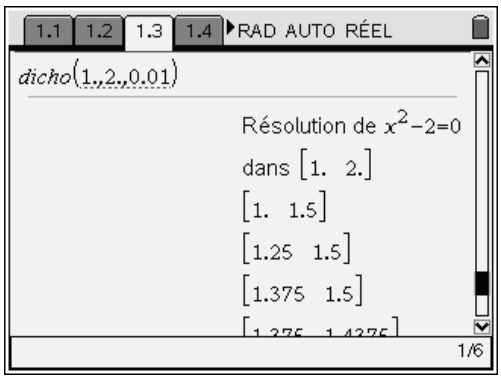
```
Define dicho(a0,b0,e)
Prgm
Local a,b,c
a:=a0
b:=b0
Disp "Résolution de ",f1(x)=0
Disp "Dans ",[a0,b0]
```

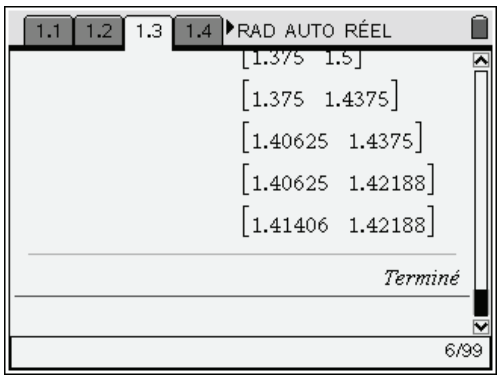
Le programme se poursuit sur la page suivante…

```
While abs(b-a)>e
 c:=(a+b)/2
 If f1(c)=0 then
  Disp c, " est solution !"
  a:=c
  b:=c
 ElseIf f1(a)f1(c)>0 then
  a:=c
 Else
  b:=c
 EndIf
 Disp "[a,b]=",[a,b]
  c:=(a+b)/2
EndWhile
EndPrgm
```

Il s'utilise à présent avec trois arguments : la valeur initiale de *a*, celle de *b*, et la précision souhaitée.

Voici par exemple la résolution de l'équation avec une précision de 10^{-2} :





Informations complémentaires sur les structures itératives

La syntaxe générale de la boucle **For** est la suivante :

For *compteur*, *début*, *fin, pas*
    $Instruction_1$
    ...
    $Instruction_n$
EndFor

Elle permet de faire varier le compteur de la valeur fixée par *pas*.

Par exemple :

```
For i,0,30,5
 Disp i
EndFor
```

affiche les nombres 0, 5, 10, 15, 20, 25, 30.

Les boucles **For** … **EndFor** et **While** … **EndWhile** devraient permettre de couvrir vos besoins courants. Il existe également une autre structure **Loop** … **EndLoop**.

Elle s'utilise sous la forme :

Loop
    $Instruction_1$
    ...
    $Instruction_n$
EndLoop

A priori, cela crée une boucle sans fin, et il est nécessaire de prévoir au moins une instruction conditionnelle permettant de sortir de cette boucle !

Par exemple

```
i:=0
Loop
 Disp i
 i:=i+5
 if i=30:Exit
EndLoop
```

est totalement équivalent à

```
For i,0,30,5
 Disp i
EndFor
```

Il est naturellement plus simple d'utiliser cette seconde syntaxe !

3.9 *Interruption d'un programme*

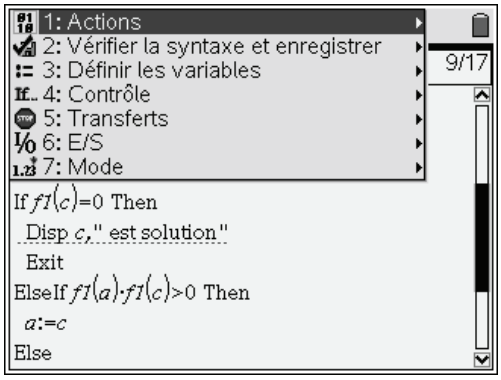
L'instruction **Stop** permet de mettre fin immédiatement à l'exécution d'un programme. Elle sera généralement placée dans une instruction conditionnelle, ce qui permettra de mettre fin de manière anticipée à l'exécution d'un programme quand certaines conditions sont vérifiées.

Dans le cas de la résolution par dichotomie, il serait prudent de commencer par tester que $f(a)f(b)<0$, et de mettre fin au programme après affichage d'un message d'erreur si ce n'est pas le cas. Pour cela, on peut rajouter les lignes suivantes au début du programme :

```
if f1(a0)f1(b0)>0 then
 Disp "Erreur, f(a) et f(b) sont de même signe"
 Stop
EndIf
```

3.10 *Insérer un commentaire*

Pour ajouter un commentaire dans un programme, Appuyez sur (menu), utiliser le menu **Action**, puis choisir **Insérer une annotation**. On peut aussi sélectionner le caractère © dans la table des symboles. Une ligne commençant par © sera ignorée. Cela permet de placer des explications dans le texte d'un programme, sans que cela intervienne sur son exécution.



☞ *Nous verrons une utilisation particulièrement intéressante de © dans le* ***chapitre 15****.*

3.11 *Quelques dernières retouches*

Éviter une boucle sans fin !

La version actuelle de notre programme présente un sérieux défaut …

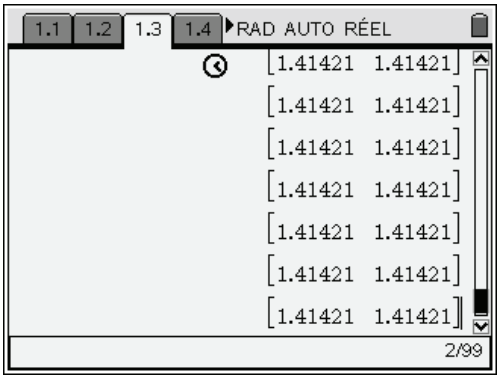
La condition d'arrêt de la boucle porte sur la comparaison $|b-a|>e$. Tant que cette propriété est vraie, on continue… A priori, cela peut sembler sans risque puisque la valeur de $|b-a|$ est divisée par 2 à chaque étape… On peut donc espérer que l'on finira bien par avoir $|b-a|$ plus petit que $e=10^{-12}$, ou même que $e=10^{-50}$. Mais que se passera-t-il si on lance ce programme avec un troisième argument nul, ou négatif. Dans ce cas, la condition $|b-a|>e$ sera en permanence vérifiée, et la boucle continuera donc indéfiniment, ce qui est bien sûr très gênant !

Comment l'éviter ? Tout simplement en ajoutant un test au début du programme :

```
if e≤0 then
 Disp "Erreur, le troisième argument doit être positif"
 Stop
EndIf
```

☞ *On obtient le symbole* ≤ *en utilisant* (ctrl)(<)

Si vous oubliez d'ajouter cette ligne, et lancez l'exécution du programme avec un appel du type **`dicho(1.,2.,0)`**, vous obtiendrez l'écran suivant :



Appuyez sur la touche (off on) afin de quitter cette boucle sans fin.

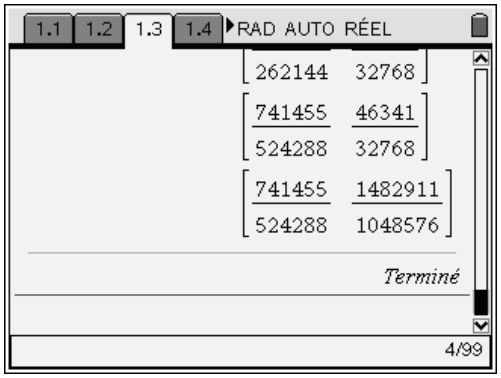
Optimisation

La version actuelle du programme calcule $f1(a)$ à chaque étape, pour simplement tester le signe. Il suffirait pour cela d'utiliser la valeur de $f1(a_0)$ puisque le signe sera ensuite toujours le même. En utilisant une variable locale supplémentaire dans laquelle on pourra mémoriser la valeur de $f1(a_0)$ on peut éviter ces calculs inutiles. On peut de même éviter de calculer deux fois $f(c)$ à chaque itération.

Sécurisation du fonctionnement du programme

Pour l'instant, rien n'interdit de donner des valeurs exactes aux arguments d'appel.

Voici par exemple le type de résultat obtenu si on utilise **dicho(1,2,0.000001)** (sans point décimal pour les valeurs initiales de *a* et *b*) au lieu de **dicho(1.,2.,0.000001)** :



Tous les calculs ont été faits sous forme symbolique, mais il est bien peu probable que ce soit ce qui été attendu par l'utilisateur de ce programme de recherche de la valeur approchée de la solution d'une équation !

Pour l'éviter, il suffit de modifier deux lignes dans le début du programme en écrivant

```
a:=approx(a0)
b:=approx(b0)
```

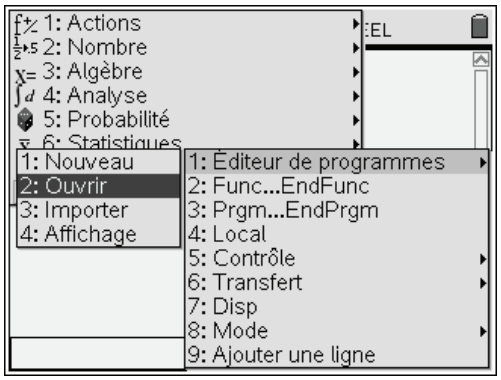
3.12 *Fermeture de l'éditeur de programme*

Pour refermer l'éditeur de programme appuyer sur (menu) et sélectionner **Action**, **Fermer**.

Cela referme l'éditeur en supprimant la page dans laquelle cet éditeur était utilisé, ou le partage d'écran, dans le cas où l'on avait choisi d'ouvrir cet éditeur sur une page contenant d'autres applications.

3.13 *Modification ultérieure d'un programme*

Pour éditer à nouveau un programme (afin d'en modifier le contenu) demander l'ouverture de l'éditeur de programme (dans une nouvelle page, ou dans un partage d'écran), en choisissant cette fois l'option **Ouvrir**.



On obtient ensuite une liste permettant de choisir le programme que l'on souhaite ouvrir.

3.14 *Ouverture automatique de l'éditeur en cas d'erreur*

Si l'utilisation d'un programme provoque une erreur, on obtient une boîte de dialogue proposant l'édition du programme. Il suffit de sélectionner ce choix pour

- Si le programme est déjà en cours d'édition, aller sur la page correspondante.
- Si ce n'est pas le cas, ouvrir un partage d'écran avec l'éditeur de programmes.

Dans les deux cas le curseur sera placé à l'endroit où l'erreur s'est produite.

4. Programme ou fonction ?

Dans ce qui précède, nous avons défini un *programme* dont le rôle était d'afficher les différentes étapes d'une dichotomie. Il est également possible de définir des *fonctions*.

4.1 *Différence entre fonctions et programmes*

Le tableau ci-dessous permet de mettre en évidence certaines similitudes mais aussi certaines différences.

Programmes	Fonctions
Un programme permet d'effectuer une suite d'opérations de façon automatique. Les arguments nécessaires au fonctionnement de ce programme doivent être transmis lors de l'appel de ce programme.	Une fonction effectue une ou plusieurs opérations à partir des arguments qui lui sont transmis, mais retourne de plus un résultat destiné à une utilisation ultérieure comme le font toutes les fonctions usuelles prédéfinies : sinus, cosinus, racine carrée ou autres. Une fonction retourne un résultat unique, dépendant des arguments utilisés. Il est cependant possible de retourner plusieurs valeurs à condition de les placer dans une liste, tout comme le fait la fonction **zeros.**
L'instruction **Disp** permet d'afficher des résultats obtenus pendant l'exécution du programme.	Une fonction peut aussi utiliser **Disp** pour afficher les étapes du calcul[2].
Il est généralement recommandé d'utiliser des variables locales, mais il reste possible d'agir sur des variables globales dans les cas où l'on souhaite effectivement le faire. Par exemple, le programme sur les séries de Fourier présenté dans le **chapitre 11** définit des variables globales qui seront utilisables une fois l'exécution du programme terminée.	Une fonction ne peut pas modifier la valeur d'une variable globale. Toute variable dont la valeur est modifiée dans une fonction doit donc être obligatoirement déclarée comme étant locale. Par exemple, une variable servant d'index dans une boucle **For**… **EndFor** devra obligatoirement être déclarée dans une instruction **Local**.
Un programme peut aussi inclure des instructions permettant de définir le mode de fonctionnement de la TI-Nspire (calcul exact ou approché, degrés ou radians, nombre de décimales) ou contenir des instructions **delvar** permettant d'effacer certaines variables.	Les instructions composant une *fonction* ne peuvent pas comporter un appel à une commande agissant sur le mode de fonctionnement de la TI-Nspire, d'appel à une procédure intégrée au système pouvant modifier des variables globales, comme **LU**, **QR**, **SortA**, **SortD**, ni comporter d'appel à un *programme*.

Le choix du type *fonction* s'impose chaque fois que l'on cherche à construire un algorithme permettant d'obtenir un résultat qui pourra être utilisé dans un calcul ultérieur.

[2] Ce point constitue une différence majeure par rapport aux calculatrices de type TI-89 ou Voyage 200. C'est une amélioration particulièrement intéressante.

4.2 *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**.

4.3 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
```

4.4 *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 $\exp(-(1-1).v)$, ce qui correspondra au terme u_1.
2. Pour $k = 2$, on va remplacer *v* par $\exp(-(2-1).v)$, 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 :

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

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

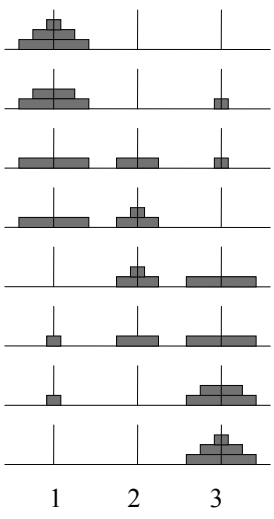
Voici à présent la version récursive, totalement immédiate :

```
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 $b = 3$, $c = 6 - 1 - 3 = 2$.)

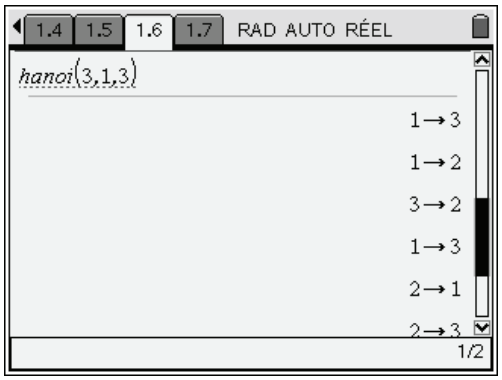
On peut en déduire le texte du *programme* (à gauche) ou de la *fonction* (à droite) à utiliser :

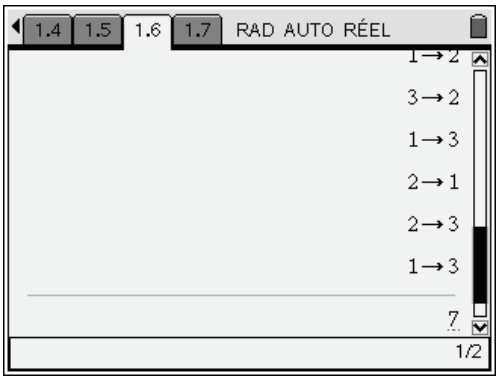
Ce *programme* affiche les déplacements.
Les arguments sont : le nombre de disques, le numéro du piquet, et le numéro de celui d'arrivée.

```
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
EndPrgm
```

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)
 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 (off 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"**

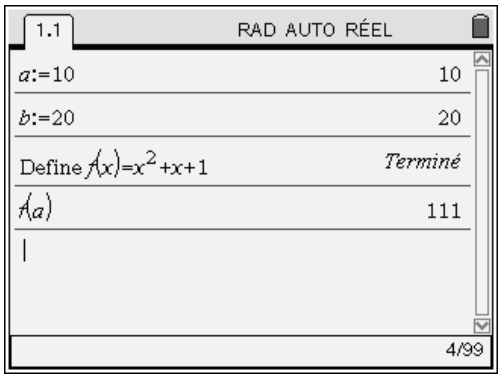
On peut à présent écrire notre fonction sous la forme :

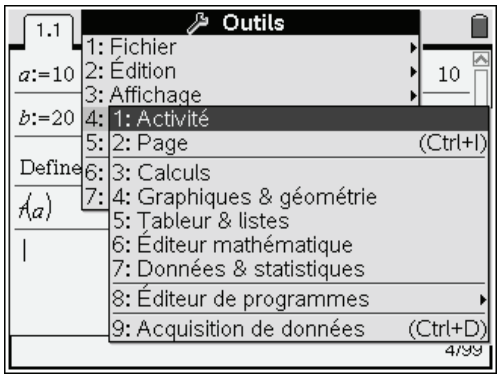
```
Define u(n)=
Func
Local v
if not is_posint(n) then
  Return "Erreur"
ElseIf n=0 then
  Return 1
Else
  Return exp(-(n-1)u(n-1))
EndIf
EndFunc
```

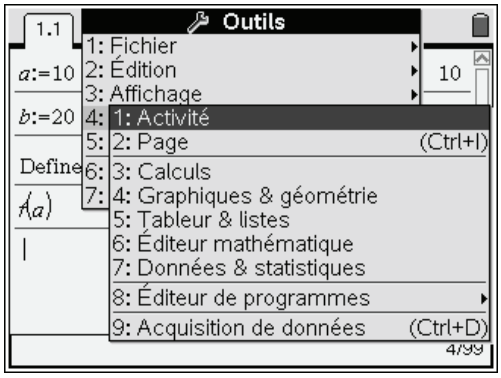
Dans l'exemple ci-dessus, nous avons d'abord « appris » à TI-Nspire à reconnaître les entiers positifs à l'aide d'une fonction auxiliaire, puis nous avons utilisé cette fonction pour simplifier l'écriture de la fonction principale. Ce type d'approche, où l'on décompose un traitement parfois complexe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

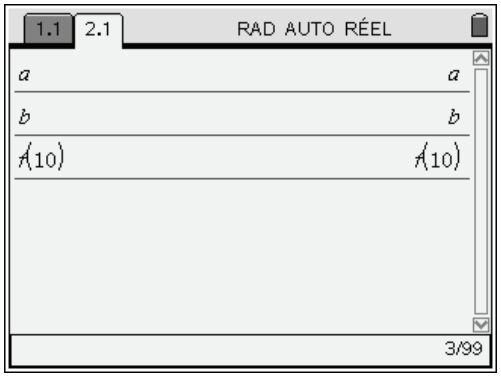
6. Domaine d'existence des programmes et des fonctions

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[3]. 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.







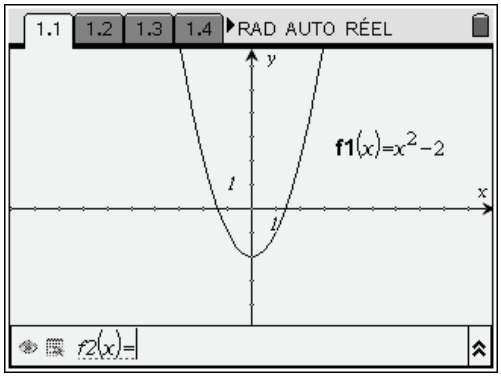


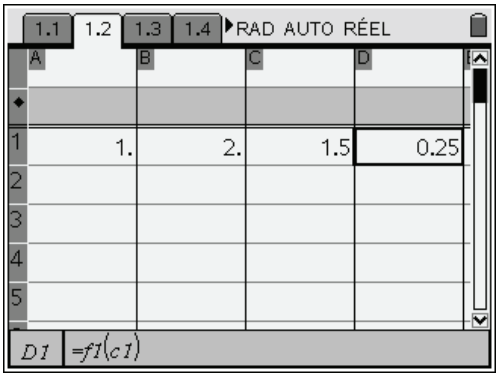
[3] Ce type d'organisation n'est pas spécifique à TI-Nspire, et tout ceci est sans doute déjà familier aux utilisateurs d'autres logiciels déjà utilisés dans les classes préparatoires, comme Maple.

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**.

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 & géométrie, 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** :

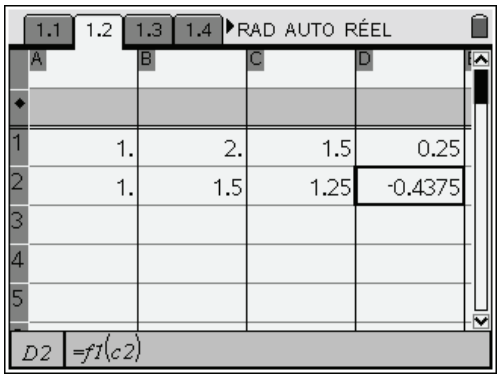
=when(f1(a1)d1≥0,c1,a1)

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

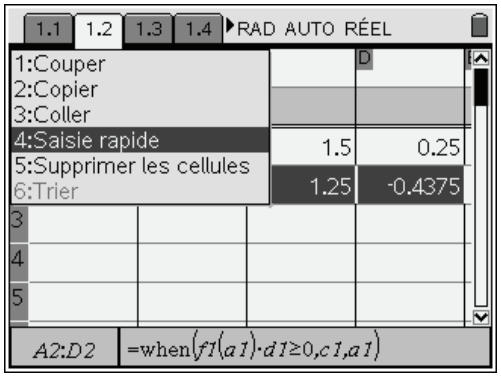
Dans la cellule **c2**, on place la formule **=(a2+b2)/2** et dans la cellule **d2**, on écrit **=f1(c2).**

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.

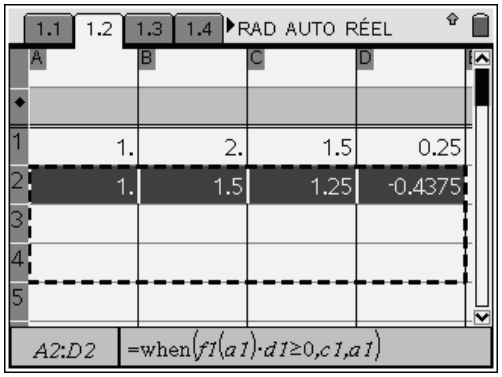
On obtient ainsi un tableau comportant les deux lignes suivantes :

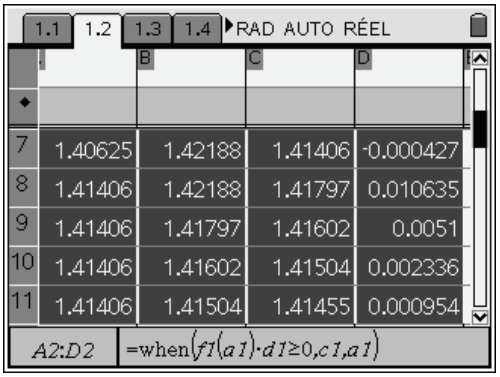


Placez le curseur en **a2**, appuyez sur la touche ⇧, et tout en maintenant cette touche enfoncée, déplacez-vous vers la droite de manière à sélectionner les cellules **a2** à **d2**. Ouvrez alors le menu contextuel par ctrl menu, et sélectionnez **Saisie rapide**, de manière à obtenir l'écran ci-dessous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

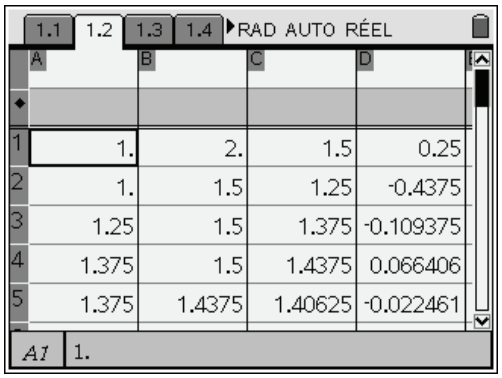


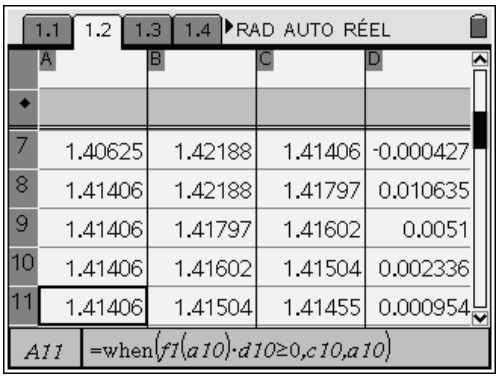
Il suffit ensuite de se déplacer vers le bas avec le Nav Pad pour définir la zone dans laquelle on souhaite recopier les formules. On termine en appuyant sur le bouton central :





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

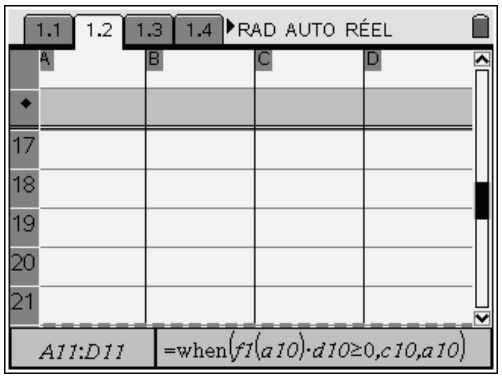


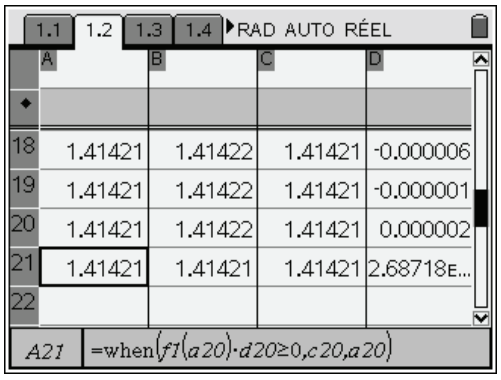


Les résultats obtenus dans les cellules des 2 premières colonnes de la ligne 11 correspondent à la fin de la dixième étape.

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.

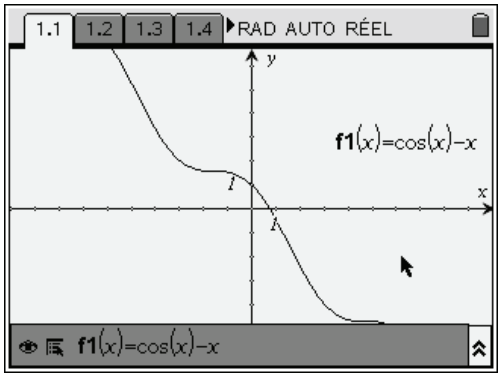


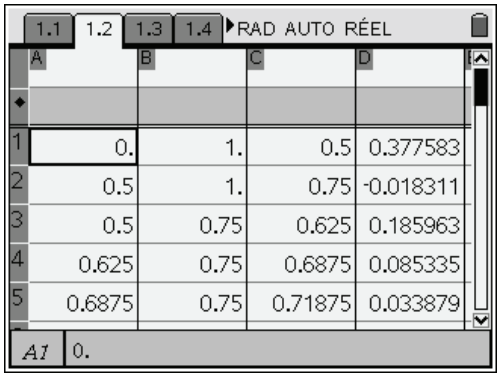


On obtient ainsi $x_0 \simeq 1.41421$

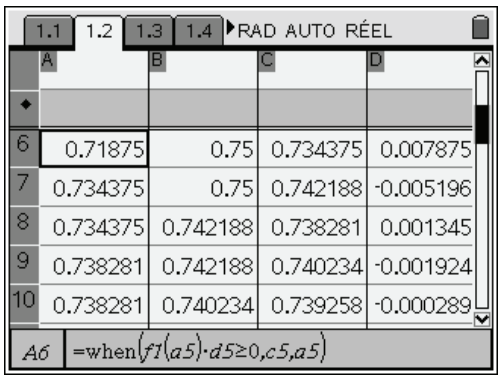
Une fois créée, cette feuille de calcul est utilisable avec toute autre fonction continue, strictement monotone sur un intervalle $[a,b]$, et prenant en *a* et en *b* des valeurs de signes opposés.

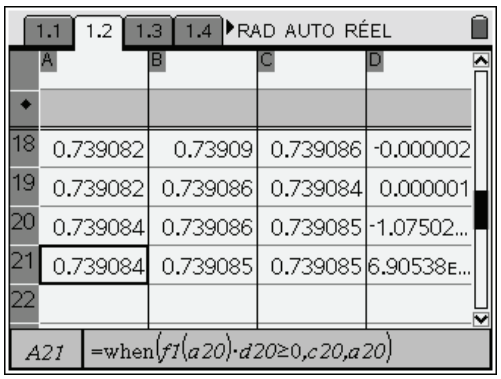
Cherchons par exemple la racine de $x - \cos(x) = 0$. Compte-tenu de la représentation graphique, nous partons cette fois de l'intervalle $[0,1]$. Il suffit de modifier le contenu des cellules **a1** et **b1**. L'ensemble du tableau est mis à jour, et on peut suivre les étapes en faisant défiler les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

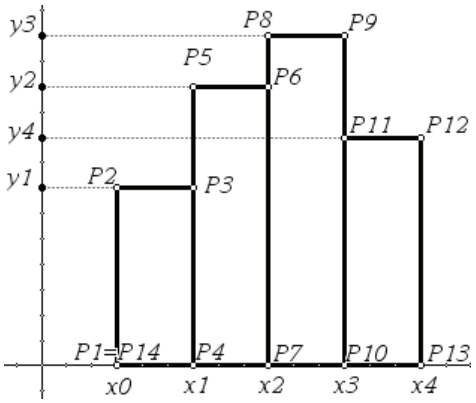
L'appel de ce programme devra provoquer une mise à jour de l'écran Graphiques & Géométrie, afin d'afficher la fonction que l'on souhaite étudier, ainsi que les rectangles utilisés.

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 version 1.4 de 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 $\cos(x)$.

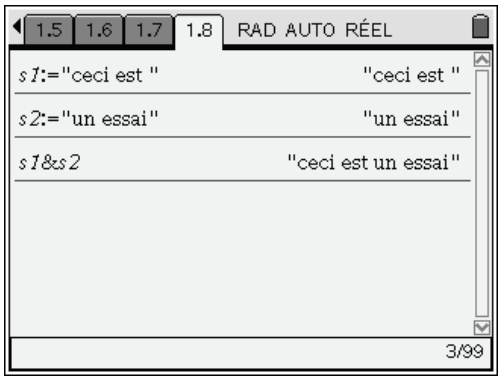
La fonction **string** opère la conversion en une chaine de caractères, l'utilisation de **`string(ex)`** permet donc d'obtenir la chaine de caractère*s* **`"cos(x)".`**

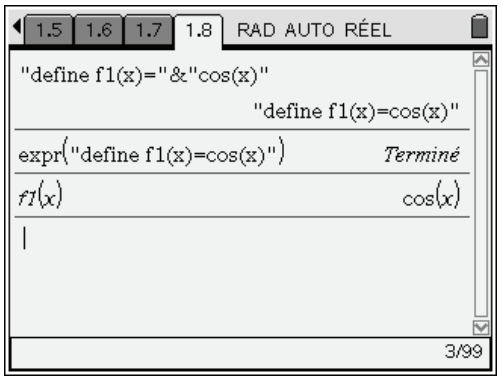
L'opérateur **&** permet de rassembler deux chaines de caractères en une chaine unique.

`"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)`**.

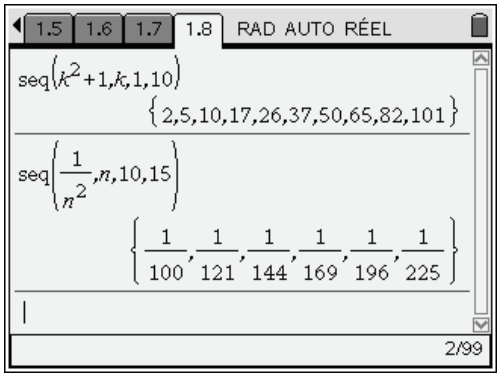




En résumé, on a bien défini la fonction f1 à partir du premier argument transmis au programme.

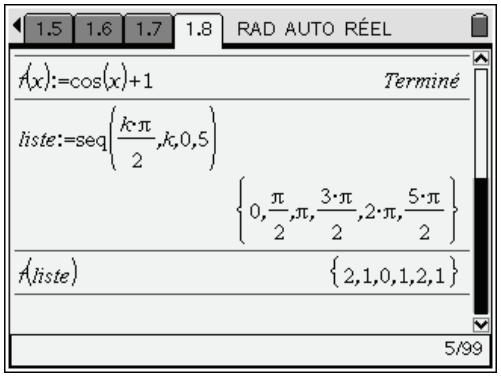
2. Liste des x_i

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.



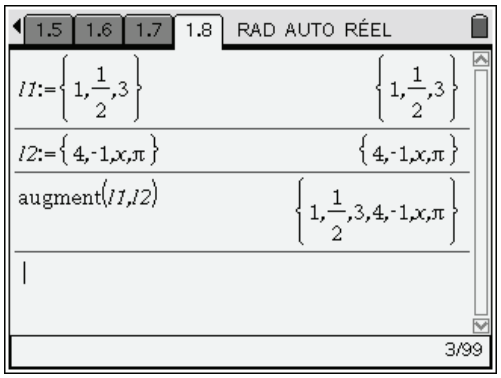
3. Liste des y_i

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*.



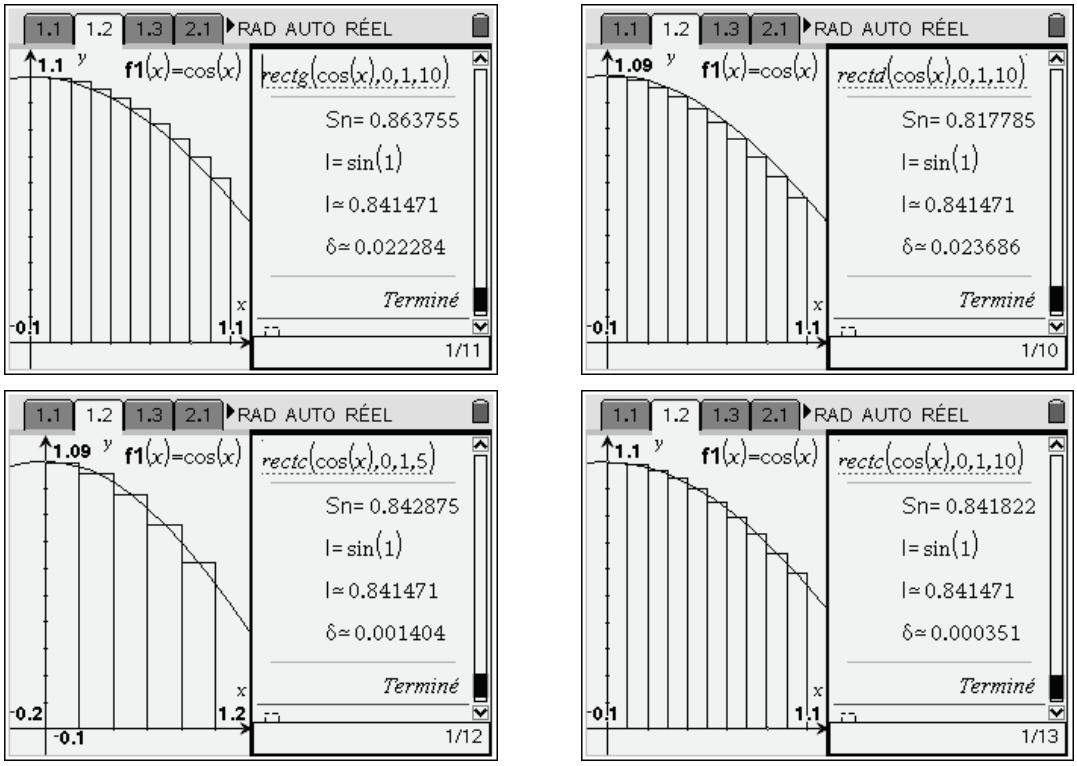
Nous allons maintenant pouvoir construire un classeur illustrant la méthode des rectangles.

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 & géométrie.

4. Dans cette application Graphiques & géométrie, 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 sur le site Internet **www.univers-ti-nspire.fr**. La version proposée sur ce site 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.

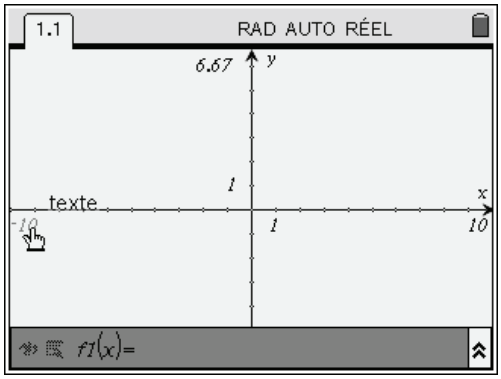
On peut constater expérimentalement que l'approximation est bien meilleure dans ce dernier cas[4].

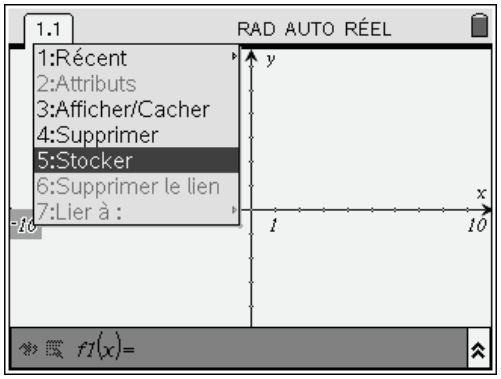


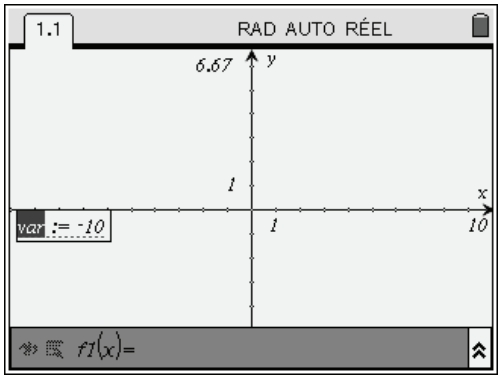
Ces programmes sont capables d'adapter automatiquement le cadrage. La méthode utilisée est décrite sur la page suivante.

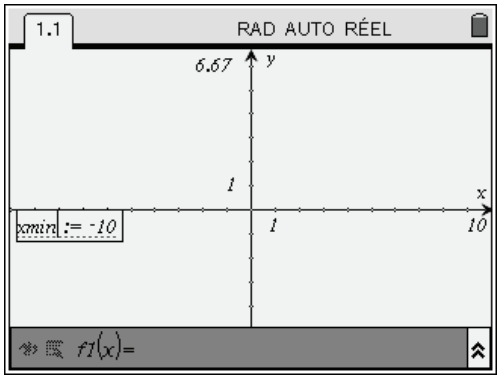
[4] Une étude théorique permet de montrer que, si la fonction vérifie certaines hypothèses, l'erreur est en $\frac{A}{n}$ si on choisit l'extrémité gauche ou droite, et en $\frac{B}{n^2}$ si on choisit les centres des intervalles.
En particulier, à partir d'une certaine valeur de n, si on double le nombre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

1. Dans l'écran Graphiques & géométrie 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**.

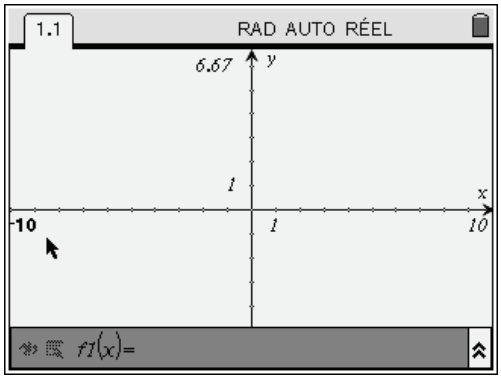


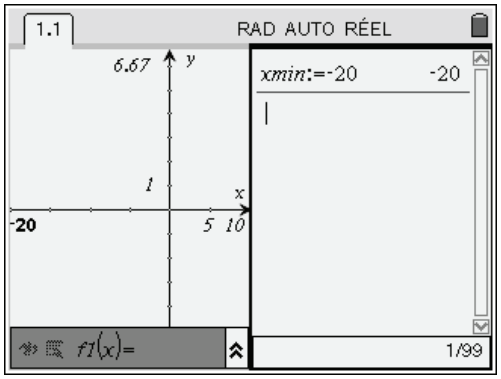






3. Une fois la liaison effectuée, le nombre sélectionné passe en gras, et on peut vérifier que tout se passe bien en modifiant ces différentes valeurs depuis un écran Calculs.





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 sur le site **www.univers-ti-nspire.fr**.

Vous trouverez également sur ce site un émulateur 3D (en mode *fil de fer*) qui a été créé en utilisant ces différentes techniques.

Annexe B. Syntaxes TI-Nspire CAS et Maple[5]

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**

[5] 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 For i,1,n disp i^2 sc:=sc+i^2 EndIf EndPrgm	carres:=proc(n) local i; global sumcarres for i from 1 to n do print (i^2) sc:= sc+i^2 od; RETURN(NULL) end