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 de ne pas trouver des commandes graphiques sur les versions actuelles de la TI-Nspire CAS. On dispose tout de même d'un outil très performant, la 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 3.2. Il est important de le préciser car il est probable que la programmation sur TI-Nspire CAS sera encore amenée à évoluer dans les années à venir, et que d'autres fonctionnalités lui seront ajoutées, comme cela a été le cas depuis les premières versions. Des mises à jour de ce chapitre seront publiées si nécessaire lors de la sortie de ces futures versions.

La version 3.2 de TI-Nspire permet déjà de couvrir les besoins du programme des classes scientifiques, par contre, il est certain qu'elle ne permet pas l'écriture par exemple d'un jeu interactif ou d'une application utilisant toutes les ressources de l'affichage graphique ; traitement de l'image, pilotage du TI-Innovateur™ Rover ou même d'un drone… Pour cela on dispose depuis la version 5.2. d'un environnement Python qui sera abordé dans le **chapitre 17**.

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.

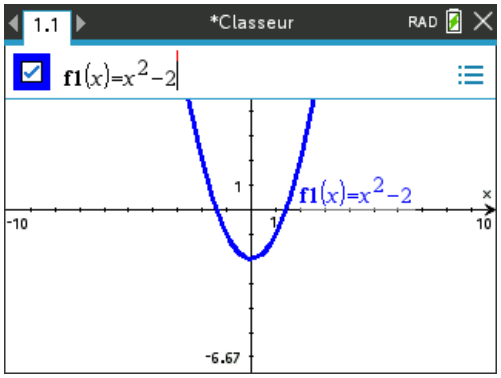
Nous recherchons ici une valeur approchée de $\sqrt{2}$. C'est la solution positive de l'équation $x^2-2=0$. On considère donc $f(x)=x^2-2$. Cette fonction est croissante sur $\mathbb{R}^+$. $f(1)<0$ et $f(2)>0$. On a donc $x_0\in]1,2[$. Voici les premières étapes de l'algorithme décrit ci-dessus.

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.



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. **Request** (**RequestStr** [2] pour une chaîne de caractères) permet de demander ces valeurs au cours de l'exécution du programme dans les versions récentes du logiciel TI-Nspire.

On va 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)**.

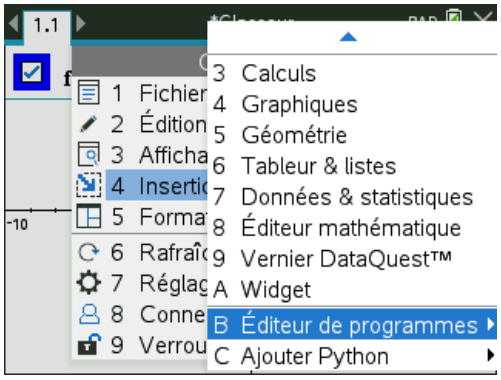
On retrouve ce mode de fonctionnement dans un logiciel comme Maple™, couramment utilisé en classe prépa. Cela ne constitue en rien une particularité de TI-Nspire CAS.

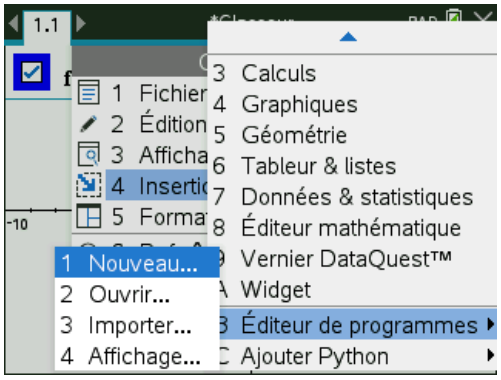
[2] Voir des exemples d'utilisation 4.2 page 17.

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

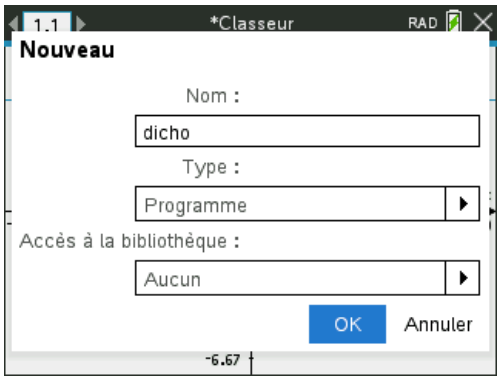
3.2 *Ouverture de l'éditeur de programme*

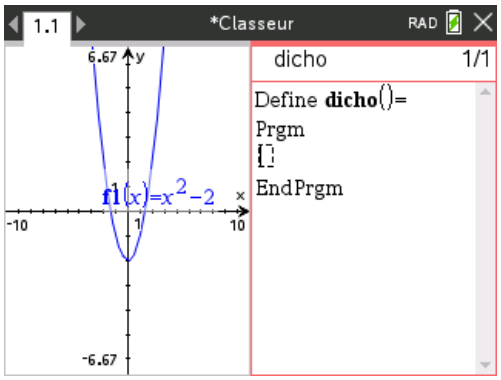
Commençons par ouvrir l'éditeur de programmes : on utilise la touche doc, puis 4 B 1.



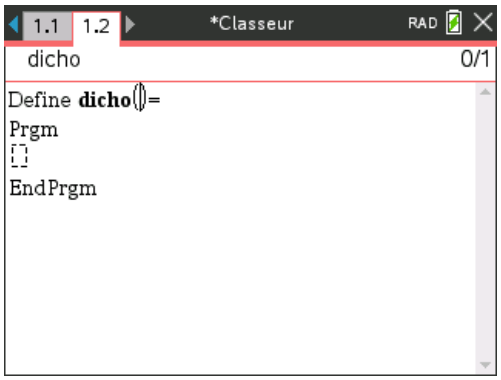


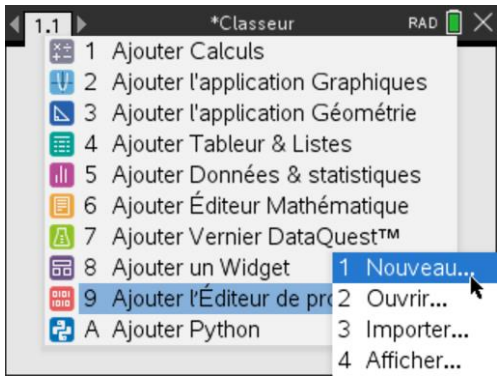
Saisir le nom choisi. Nous verrons par la suite à quoi correspondent les deux rubriques suivantes.



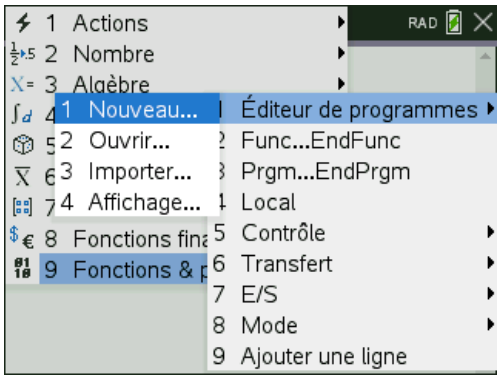


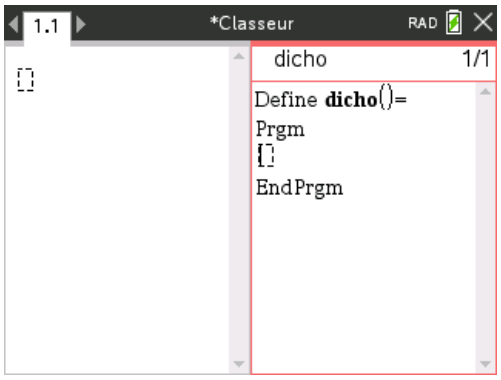
On obtient l'éditeur dans un partage d'écran, il est souvent préférable d'utiliser cet éditeur sur une pleine page, ctrl 6 permet d'obtenir un affichage plein écran. On peut également ouvrir un nouveau classeur directement avec l'**Éditeur de programmes** menu 9 1 (écran à droite ci-dessous).



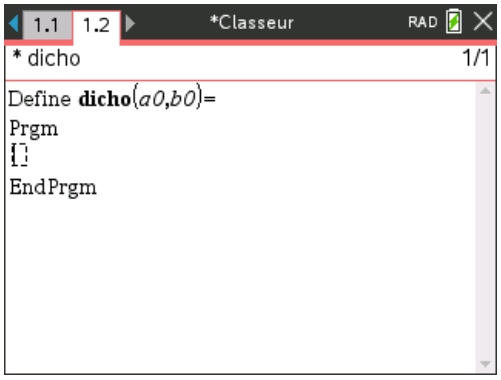


On peut aussi ouvrir l'éditeur de programme depuis l'application Calculs. Appuyer sur la touche menu puis sélectionner **Fonctions & programmes**, **Éditeur de programmes**, **Nouveau**. On obtient ensuite la même boîte de dialogue que précédemment, puis l'éditeur de programme s'ouvre dans un partage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





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**.

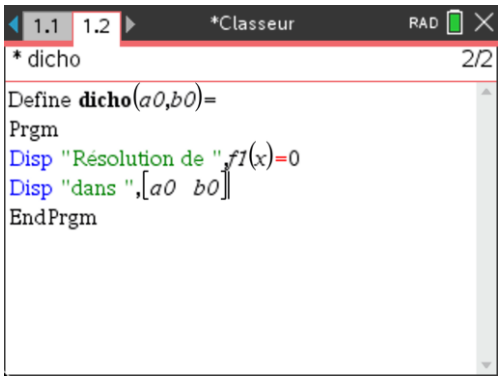


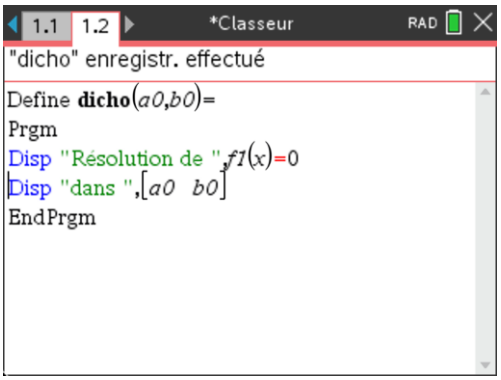
Observer la présence du caractère * précédant le nom **dicho** dans le haut de l'écran. Il indique que le contenu du programme a été modifié, mais n'a pas été encore sauvegardé dans le classeur.
Si vous souhaitez le faire dès maintenant, appuyez sur ctrl B (B : build).
Cela provoquera un contrôle de 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**, puis **1**.

3.3 *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.

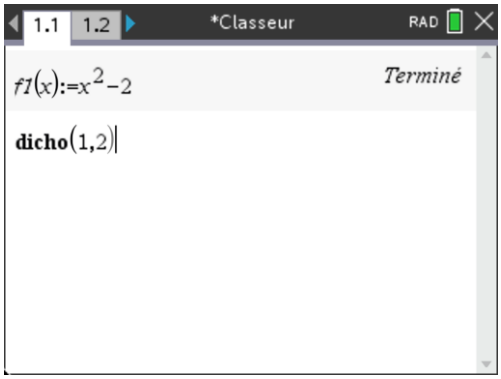
L'écran de gauche montre ce que l'on obtient après la saisie (avec le symbole * indiquant que le programme a été modifié), l'écran de droite est obtenu après avoir utilisé ctrl B.

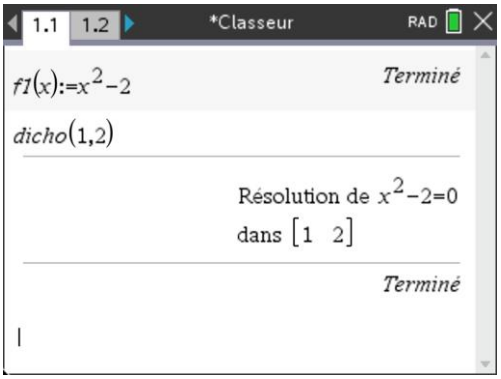




Pour entrer cette instruction, on peut la saisir directement ou appuyer sur la touche menu, puis utiliser le second choix **6:E/S** (entrées / sorties), qui contient l'instruction **Disp**.

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

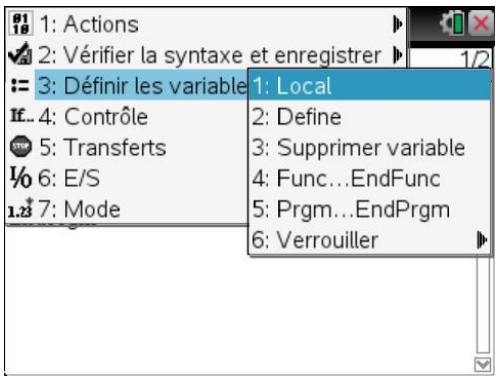




Jusqu'ici, tout va bien… Il nous reste à « apprendre » à notre programme à faire cette résolution.

3.4 *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*.

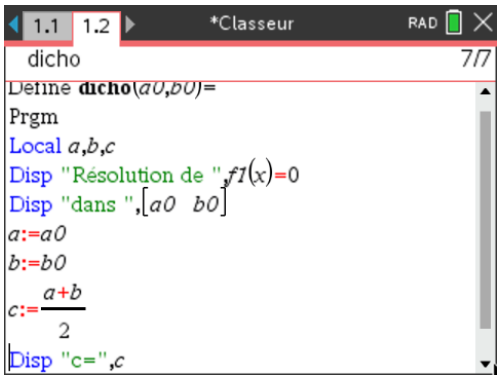
Un utilisateur de ce programme, ne connaissant pas le nom des variables choisies à l'intérieur de celui-ci risquerait donc de modifier accidentellement des valeurs mémorisées dans *a*, *b* et *c* en lançant un appel au programme **dicho**.

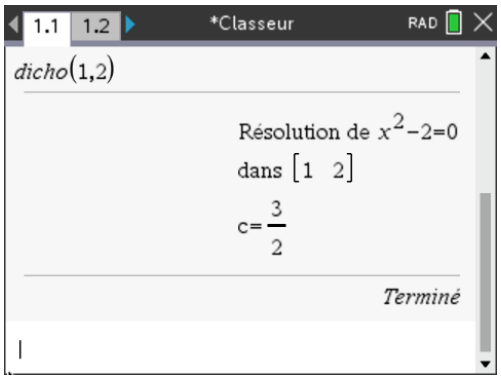
À l'inverse, si on utilise cette instruction **Local**, les variables *a*, *b* et *c* deviennent des **variables locales** à ce programme. Elles n'existent que pendant son exécution, et leur valeur n'a aucune influence sur celle des variables globales de même nom pouvant exister par ailleurs dans l'activité.

3.5 *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)





3.6 *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

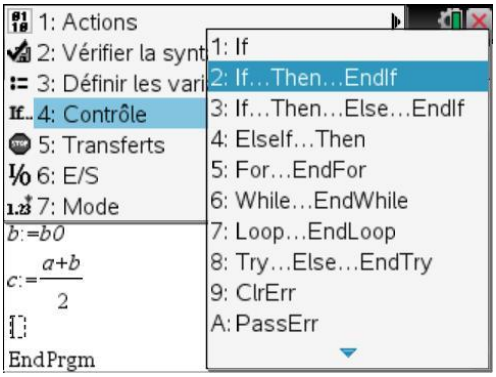
If *Condition* **Then**
    Instruction$_1$
    ...
    Instruction$_n$
Else
    Autre-Instruction$_1$
    ...
    Autre-Instruction$_n$
EndIf

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-Instruction$_1$
    ...
    Autre-Instruction$_n$
ElseIf *AutreCondition* **Then**
    Autre-Instruction$_1$
    ...
    Autre-Instruction$_n$
...
Else

Autre-Instruction$_1$
...
Autre-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]
```

3.7 *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.

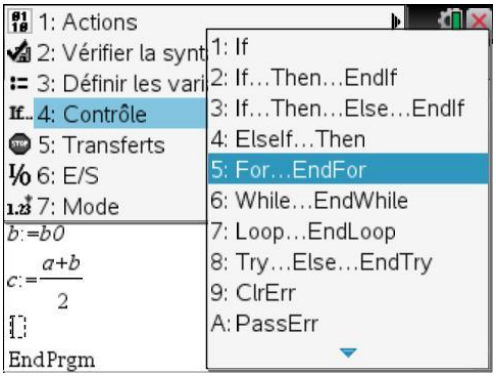
Une première approche consiste à fixer le nombre d'étapes à effectuer. Par exemple, on pourrait décider que le programme fera systématiquement 30 itérations.

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 :

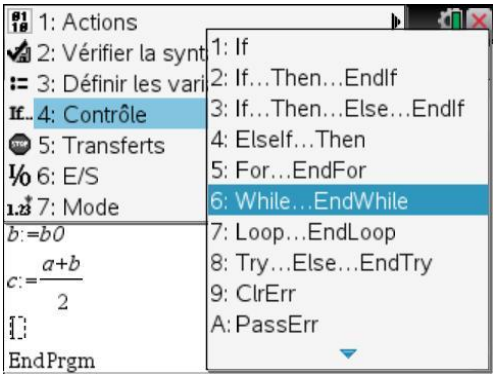
```
For i,1,30
 c:=(a+b)/2
 If f1(c)=0 then
  Disp c, " est solution  !"
  Exit
 ElseIf f1(a)*f1(c)>0 then
  a:=c
 Else
  b:=c
 EndIf
 Disp "[a,b]=",[a,b]
EndFor
```

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 à l'aide de la combinaison de touches : 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.

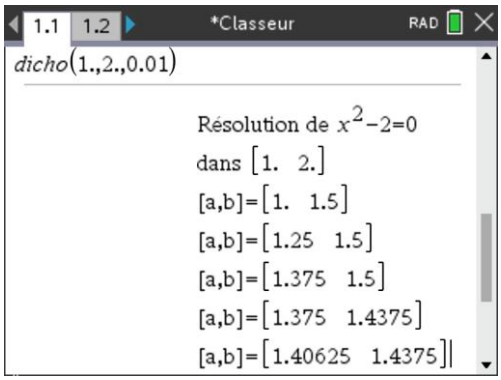
```
Define dicho(a0,b0,e)
Prgm
Local a,b,c
a:=a0
b:=b0
Disp "Résolution de ",f1(x)=0
Disp "Dans ",[a0,b0]
```

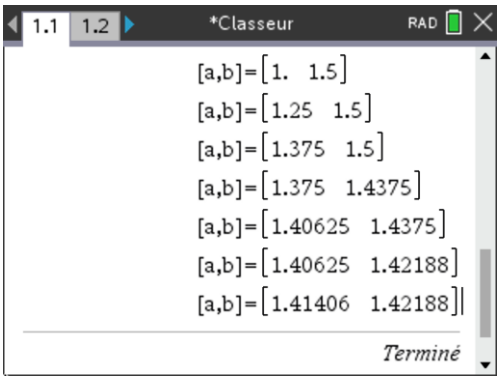
Le programme se poursuit sur la page suivante…

```
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} :





☞ *Remarquez les points décimaux après les deux premiers paramètres voir page 14.*

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 !

3.8 *Interruption d'un programme*

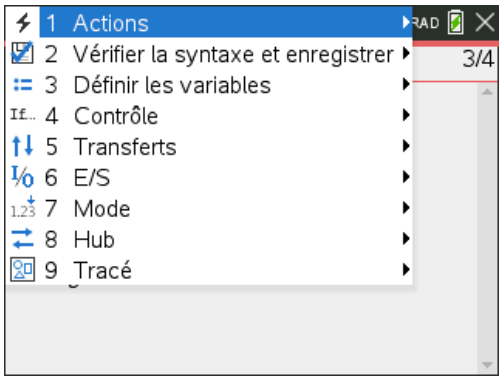
L'instruction **Stop** permet de mettre fin immédiatement à l'exécution d'un programme. Elle sera généralement placée dans une instruction conditionnelle, ce qui permettra de mettre fin de manière anticipée à l'exécution d'un programme quand certaines conditions sont vérifiées.

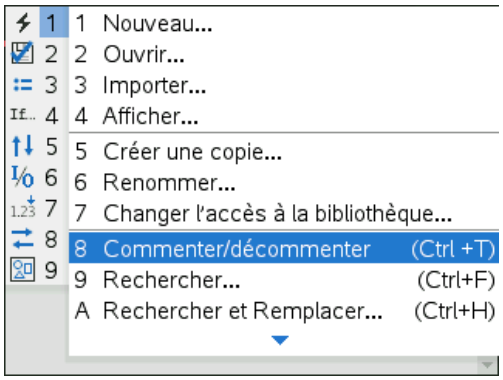
Dans le cas de la résolution par dichotomie, il serait prudent de commencer par tester que $f(a)f(b)<0$, et de mettre fin au programme après affichage d'un message d'erreur si ce n'est pas le cas. Pour cela, on peut rajouter les lignes suivantes au début du programme :

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

3.9 *Insérer un commentaire*

Pour ajouter un commentaire dans un programme, Appuyez sur [menu], utiliser le menu **Actions**, puis choisir **Commenter/décommenter** ([ctrl] [T]). 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****.*

3.10 *Quelques dernières retouches*

Éviter une boucle sans fin !

La version actuelle de notre programme présente un sérieux défaut …

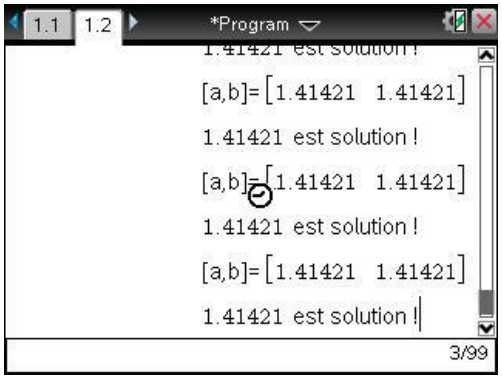
La condition d'arrêt de la boucle porte sur la comparaison $|b-a|>e$. Tant que cette propriété est vraie, on continue… A priori, cela peut sembler sans risque puisque la valeur de $|b-a|$ est divisée par 2 à chaque étape… On peut donc espérer que l'on finira bien par avoir $|b-a|$ plus petit que $e=10^{-12}$, 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 :

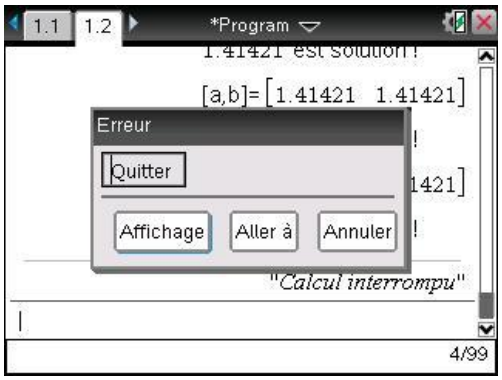
```
if e≤0 then
 Disp "Erreur, le troisième argument doit être positif"
 Stop
EndIf
```

☞ *On obtient le symbole ≤ à l'aide de la combinaison de touches* 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 on afin de quitter cette boucle sans fin.



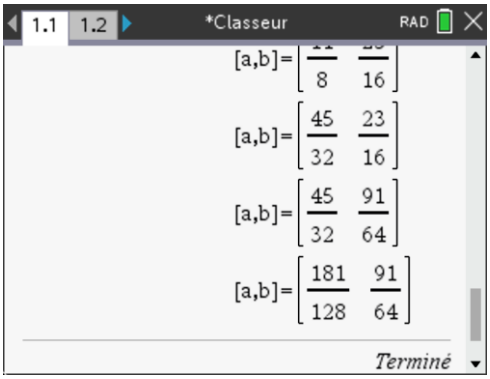
Optimisation

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 $f1(c)$ 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)`** :



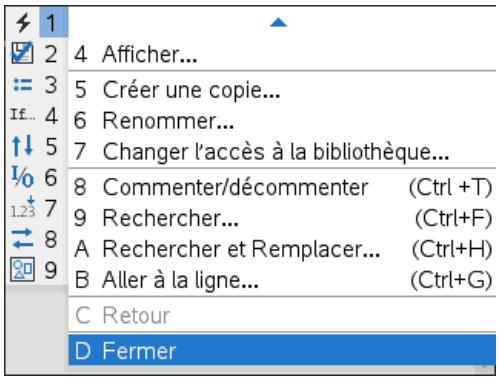
Tous les calculs ont été faits sous forme symbolique, mais il est bien peu probable que ce soit ce qui été attendu par l'utilisateur de ce programme de recherche de la valeur approchée de la solution d'une équation !

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

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

3.11 *Fermeture de l'éditeur de programme*

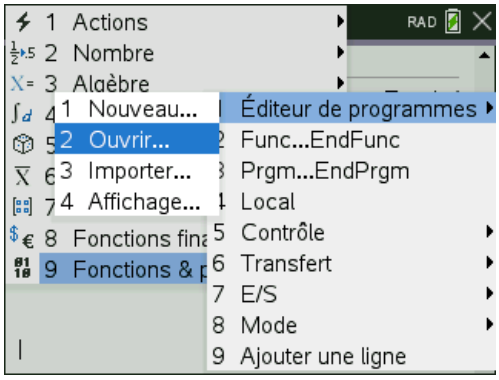
Pour refermer l'éditeur de programme appuyer sur menu et sélectionner **Actions, Fermer.**



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.

3.12 *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.

3.13 *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 peuvent soit être transmis lors de l'appel de ce programme, soit en utilisant **Request** (ou **RequestStr** s'il s'agit de chaîne de caractères, voir page suivante).	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[3].
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.

[3] 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* Request

Voici un exemple d'utilisation des instructions **Request**, **Disp** et **Text** (cette dernière marque une pause dans l'exécution d'un programme et affiche une chaîne de caractères dans une boîte de dialogue).

Programme de résolution d'une équation du second degré.

```
Define quad()=Prgm
  Local a,b,c,d
  Text "Résolution de ax²+bx+c=0 dans ℝ "
  Request "a=",a
  Request "b=",b
  Request "c=",c
  d:=b^(2)−4*a*c
  Text "Calculons Δ=b²−4ac"
  Disp "Δ=",d
  If d>0 Then
    Text "Δ>0, 2 racines distinctes"
    Disp "x1=",((-b− √ (d))/(2*a))
    Disp "x2=",((-b+ √ (d))/(2*a))
  ElseIf d=0 Then
    Text "Δ=0, Une racine double"
    Disp "x0=",((-b)/(2*a))
  Else
    Text "Δ<0, pas de racines"
  EndIf
EndPrgm
```

Un deuxième exemple : utilisation des fonctions **RequestStr** et **Text** :

```
Define hello()=Prgm
 RequestStr "Quel est votre nom ?",n
 Text "Bonjour "&n
EndPrgm
```

Attention les instructions **Request**, **RequestStr** et **Text** ne sont utilisables que dans un programme, contrairement à **Disp** qui peut être utilisé dans un programme ou une fonction.