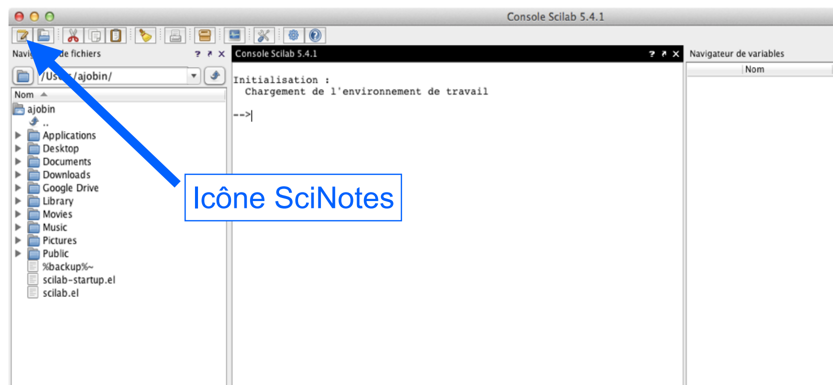
CH 3 : Premiers programmes en **Scilab**

I. L'éditeur de texte SciNotes

I.1. Première ouverture

SciNotes est l'éditeur de texte intégré à **Scilab**. Pour y accéder il faut au préalable choisir la console comme fenêtre de travail. On dispose alors de deux méthodes :

- soit cliquer sur l'icône en haut à gauche,



- soit sélectionner « **SciNotes** » dans le menu « Applications ».

Une fenêtre s'ouvre alors. **SciNotes** étant un outil essentiel, nous allons de suite l'intégrer à notre environnement de travail. Pour ce faire, on arrime la fenêtre **SciNotes** en lieu et place de la fenêtre « Navigateur de variables » (*cf docking* et explications du CH 1).

I.2. Un mot sur l'éditeur

SciNotes dispose de quelques fonctionnalités intéressantes :

- **coloration syntaxique** : les mots clés du langage apparaissent en couleur.
- **indentation** : afin d'améliorer la lisibilité du code, il est nécessaire de l'aérer. Il s'agit alors de sauter des lignes et d'utiliser des espaces pour faire apparaître des blocs. **SciNotes** permet de corriger l'indentation d'un programme. Pour ce faire :
  - × sélectionner toutes les lignes du programme considéré,
    (« Sélectionner tout » dans le menu « Édition » ou raccourci associé)
  - × choisir « Corriger l'indentation » dans le menu « Format ».
- **autocomplétion** : lors de l'écriture d'un mot clé, **SciNotes** peut automatiquement intégrer de nouveaux éléments (tels que des couples de parenthèses ou des mots clés associés au premier) dans le but de faire gagner du temps au programmeur. Cette fonctionnalité peut être désactivée en désélectionnant les items proposés par « Complétion automatique sur » dans le menu « Option » lorsque **SciNotes** est la fenêtre de travail.
  (par exemple, dans ***SciNotes****, taper « function » suivi d'un espace)*

I.3. Quelques raccourcis intéressants pour gagner du temps

Connaître certains raccourcis claviers permet de gagner en vitesse en évitant notamment l'utilisation un peu lourde de la souris. Voici quelques raccourcis qui pourront servir en **Scilab** (voire pour certains autres logiciels).

	Windows/Linux	Mac
Sélectionner tout	`CTRL-A`	`CMD-A`
Sauvegarder	`CTRL-S`	`CMD-S`
Ouvrir un nouvel onglet	`CTRL-N`	`CMD-N`
Enregistrer et exécuter	`F5`	`fn-F5`
Corriger l'indentation	`CTRL-I`	`CMD-I`

II. Les scripts Scilab

II.1. Introduction

Le terme **script** est utilisé ici pour désigner un fichier de commandes. On distingue cette notion de celle de **fonction** qui sera présentée dans le paragraphe suivant.

II.2. Un premier script

On s'intéresse au script suivant dont le but est de calculer le discriminant du polynôme $\texttt{a}X^2 + \texttt{b}X + \texttt{c}$, avec `a = 3`, `b = 5`, `c = 2`.

```
// Un calcul de discriminant
a = 3;
b = 5;
c = 2;
delta = b^2-4*a*c
```

La première ligne est précédée de deux barres obliques /. Ceci permet de pouvoir écrire un **commentaire** qui ne sera pas interprété lorsque le script sera exécuté. Un commentaire contient des précisions sur les lignes de code qui lui succèdent : on explicite ce que l'on cherche à calculer ou le rôle de telle ou telle variable. Les commentaires jouent un rôle primordial en programmation : ils permettent de s'y retrouver plus facilement dans du code écrit. Il faut avoir en tête les situations suivantes :

- reprise d'un code que l'on a écrit précédemment,
  À quoi servait ce bout de code déjà ?
- reprise d'un code écrit par un autre programmeur,
  Que voulait-il calculer ?
- écriture d'un code volumineux[1].
  À quoi sert cette variable utilisée 100 lignes plus bas ?

L'écriture de commentaires est donc une nouvelle règle de bonne conduite que l'on devra respecter.

II.3. Sauvegarder un script

Une fois ce premier script écrit dans une fenêtre **SciNotes**, il convient de le sauvegarder. Pour ce faire, une fois **SciNotes** sélectionné comme fenêtre de travail, on peut :

- soit sélectionner « Enregistrer » dans le menu « Fichier »,
- soit utiliser le raccourci clavier correspondant :
  `CTRL-S` (Windows/Linux) ou `CMD-S` (Mac).

Il faudra évidemment sélectionner le dossier adéquat pour la sauvegarde.

Remarque : par défaut, un script sera sauvegardé avec l'extension **.sce**.

Dès que vous modifiez un fichier, il faut penser à le sauvegarder. Dans le cas contraire, vous serez à la merci d'un bug de l'ordinateur, d'une mauvaise manipulation ou même d'une simple panne de courant. Sauvegarder doit donc devenir un réflexe !

II.4. Exécuter un script

En **Scilab**, il y a essentiellement deux modes d'exécution :

- **Exécuter fichier sans écho** : le fichier est exécuté mais il n'y a pas d'affichage dans la console autre que l'appel à la fonction d'exécution.

```
--> exec('/Info/Exemple_cours/discriminant.sce', -1)
```

- **Exécuter fichier avec écho** : chaque ligne du fichier est affichée et exécutée dans la console.

```
--> // Un calcul de discriminant
--> a = 3;
--> b = 5;
--> c = 2;
--> delta = b^2-4*a*c
  delta  =
     1.
```

[1] À titre d'exemple, le logiciel qui dirige une ligne automatique de métro parisien (ligne 1 et 14) contient environ 150 000 lignes de code.

Ces deux modes sont proposés dans le menu « Exécuter », visible lorsque **SciNotes** est choisi comme fenêtre de travail. Pour ce type de scripts, seul le mode d'exécution avec écho fait sens. Il est accessible depuis le raccourci : `CTRL-L` sous Windows/Linux, et `CMD-L` sous Mac.

II.5. Ouvrir un ancien script

Lors d'une nouvelle séance, il peut être utile de charger un ancien fichier. Pour ce faire, il y a une nouvelle fois deux solutions :

- soit utiliser le « Navigateur de fichiers » : on se déplace jusqu'au fichier concerné. L'ouverture dans **SciNotes** se fait par un double-clic.
- soit sélectionner « Ouvrir un fichier » dans le menu « Fichier » (lorsque la fenêtre de travail est **SciNotes** ou la console).

III. Les fonctions Scilab

III.1. Analogie avec la recette de cuisine

On a précédemment défini la notion de programme comme la traduction dans un langage informatique d'un algorithme. Pour bien comprendre la notion d'algorithme, il est fréquent de l'illustrer par un algorithme de la vie quotidienne : la recette de cuisine.

Une recette de cuisine est structurée comme suit.

1) Des ingrédients.
Ce sont les entrées de l'algorithme

2) Les différentes étapes de réalisation.
C'est la suite finie d'instructions

3) Le plat obtenu.
C'est la sortie de l'algorithme

Comme nous allons le voir, une fonction **Scilab** *i.e.* un programme écrit en **Scilab** respecte cette structure en trois points.

III.2. Syntaxe d'une fonction

La syntaxe d'une fonction est la suivante.

```
// Description
function sortie = maFonction(entrée)
    corps
endfunction
```

Détaillons les différents éléments de cette syntaxe.

- *// Description* : ce sont les commentaires détaillant les spécificités de la fonction qui suit.
- `function` ... `endfunction` : ce sont les mots clés du langage utilisés pour signifier le début et la fin de la fonction.
- `sortie` : c'est ce qui est calculé par la fonction. Ces arguments de sortie seront, selon les cas, zéro, une ou plusieurs variables.
- `maFonction` : c'est le nom choisi pour la fonction. Comme pour les variables, on choisira un nom explicite et, de préférence, assez court.
- `entrée` : c'est ce qui est pris comme entrée de la fonction. Ces arguments d'entrée seront, selon les cas, zéro, une ou plusieurs variables.
- `corps` : c'est la suite finie d'instructions qui permet à la fonction, à l'aide des arguments d'entrée, de calculer les arguments de sortie.

À titre d'illustration, reprenons l'exemple précédent du calcul de discriminant. En prenant comme arguments d'entrée `a`, `b` et `c` on peut calculer le discriminant de n'importe quel polynôme du second degré $\mathtt{a}X^2 + \mathtt{b}X + \mathtt{c}$. On stocke ce discriminant dans une variable nommée `delta`.

```
// discrim(a, b, c) permet de calculer le
// discriminant du polynôme aX^2+bX+c
function delta = discrim(a, b, c)
    delta = b^2-4*a*c
endfunction
```

III.3. Sauvegarder une fonction

Le principe de sauvegarde est le même que pour les scripts. Comme précisé précédemment, on peut utiliser le raccourci clavier :
`CTRL-S` (Windows/Linux) ou `CMD-S` (Mac).

Remarque :

- par défaut, les fonctions sont sauvegardées avec l'extension `.sci`. Il y a donc une différence de traitement entre les scripts (sauvegardés en .sce) et les fonctions (sauvegardées en .sci).
- **Scilab** propose, par défaut, d'utiliser le nom de la fonction pour nommer le fichier sauvegardé. Ceci incite à respecter la règle suivante :

« 1 fonction = 1 fichier »

III.4. Exécuter une fonction

Concernant les modes d'exécution, il n'y a pas de différence avec les scripts. Pour rappel, il y a deux modes d'exécution :

- **Exécuter fichier sans écho** : exécution sans affichage console.

```
--> exec('/Info/Exemple_cours/discrim.sci', -1)
```

- **Exécuter fichier avec écho** : exécution avec rappel de chaque ligne de la fonction dans la console.

```
--> // discrim(a,b,c) permet de calculer le
--> // discriminant du polynôme aX^2+bX+c
--> function delta = discrim(a, b, c)
-->     delta = b^2-4*a*c
--> endfunction
```

L'exécution du code d'une fonction ne produit pas de résultat. En fait, cette action correspond seulement à faire apprendre à **Scilab** une nouvelle définition que l'on pourra utiliser par la suite.

Pour les fonctions, on utilisera toujours le mode d'exécution sans écho. On peut d'ailleurs retenir le raccourci `F5` qui permet à la fois de sauvegarder et d'exécuter une fonction.

III.5. Tester une fonction

Insistons tout d'abord sur le fait que, lors de l'exécution d'une fonction, aucune analyse n'est effectuée par **Scilab**. En conséquence, aucune erreur ne sera détectée lors de cette phase. Par contre, lors de l'appel d'une fonction sur un jeu de données (`discrim(3, 5, 2)` par exemple), la fonction est **interprétée** et un message d'erreur peut apparaître. Ces messages d'erreurs sont mis en place pour aider le programmeur à débugger sa fonction. De ce fait, ces messages précisent les règles du langage qui, selon l'interpréteur, ont été violées.

Une fois la fonction sauvegardée et exécutée, il est primordial de la tester sur plusieurs données différentes. C'est, encore une fois, une règle de bonne conduite que l'on devra respecter. Pour chacun de ces tests, on vérifiera si :

- l'appel ne produit pas de message d'erreur.
- le résultat correspond aux spécifications de la fonction.

Par exemple, on peut tester la fonction `discrim` sur le jeu de données suivant.

```
--> discrim(3, 5, 2)
   ans  =
      1.
--> discrim(-7, -4, 0)
   ans  =
      16.
--> discrim(1, 4, 3)
   ans  =
      4.
```

Les résultats sont bien ceux attendus et aucun message d'erreur n'est levé. Cela permet d'accroître notre confiance en le calcul réalisé par cette fonction.

III.6. Ouverture d'une précédente fonction

L'ouverture d'une fonction s'effectue de la même manière que pour les scripts. Afin de pouvoir utiliser une fonction écrite dans une précédente séance, il faut l'ouvrir dans **SciNotes** et l'exécuter de nouveau. Il faut bien comprendre que lors de la fermeture, **Scilab** oublie toutes les définitions de fonctions que l'on vient de lui apprendre. Plus précisément, la mémoire utilisée par ces définitions est libérée lors de la fermeture. Cette mesure permet de s'assurer que **Scilab** n'utilisera pas la mémoire de l'ordinateur de manière incontrôlée.

Notez que si plusieurs fonctions sont ouvertes dans **SciNotes**, il est possible de toutes les exécuter en une fois. Pour ce faire, une fois **SciNotes** choisi comme fenêtre de travail, on peut :

- soit sélectionner « Enregistrer et exécuter tous les fichiers » dans le menu « Exécuter ».
- soit directement utiliser le raccourci correspondant :
  `CTRL-F5` (Windows/Linux) ou `CMD-F5` (Mac).

Il est aussi possible d'enregistrer toutes les variables et fonctions créées lors d'une séance. Pour ce faire, une fois la console choisie comme fenêtre de travail, il faut sélectionner « Enregistrer l'environnement » dans le menu « Fichier ». Lors de la séance suivante, il faudra alors « Charger l'environnement » depuis le même menu.

III.7. Fonction à plusieurs arguments de sortie

La fonction qu'on a pris pour exemple possède plusieurs arguments en entrée et un seul en sortie. Il n'y a toutefois pas de restriction sur le nombre de ces arguments. Comme précisé précédemment, une fonction peut posséder :

- 0, 1 ou plusieurs arguments d'entrée,
- 0, 1 ou plusieurs arguments de sortie.

Nous nous intéressons ici particulièrement aux fonctions prenant plusieurs arguments de sortie. Pour comprendre ce mécanisme, nous l'illustrons en développant l'exemple précédent. Considérons donc la fonction `racP` dont le but est de calculer les racines d'un polynôme du second degré $\texttt{a}X^2 + \texttt{b}X + \texttt{c}$ pris en paramètre. La plus grande racine sera stockée dans la variable `xPlus` et la plus petite dans la variable `xMoins`.

```
// racP(a, b, c) permet de calculer les
// racines du polynôme aX ^ 2+bX+c
function [xPlus, xMoins] = racP(a, b, c)
    delta = b^2-4*a*c;
    xPlus = (-b + sqrt(delta))/(2*a)
    xMoins = (-b - sqrt(delta))/(2*a)
endfunction
```

Du point de vue de la syntaxe, on remarque que l'on a encadré les deux variables de sortie `xPlus` et `xMoins` par une paire de crochets `[ ]`. On détaillera le sens de ces symboles dans le « CH 5 : Matrices ».

Testons maintenant la fonction `racP`.

```
--> racP(3, 5, 2)
 ans  =
   - 0.6666667
```

Le résultat n'est pas satisfaisant puisqu'une seule des deux racines est affichée. Afin de permettre l'affichage des deux racines, on stocke le résultat de l'appel dans deux variables que l'on nomme (par exemple) `x1` et `x2`. Pour ce faire, on procède comme suit.

```
--> [x1, x2] = racP(3, 5, 2)
 x2  =
   - 1.
 x1  =
   - 0.6666667
```

III.8. Fonction utilisant le résultat d'une autre fonction

La définition d'une fonction peut contenir l'appel à d'autres fonctions déjà implémentées dans **Scilab**. Par exemple, la fonction `discrim` utilise les opérateurs `+`, `-`, `⋆`, qui sont des fonctions prédéfinies de **Scilab**. Il est aussi possible de faire appel à une fonction que nous avons nous-mêmes définie. C'est même l'une des fonctionnalités les plus importantes des fonctions !

Reprenons l'exemple précédent de la fonction `racP`. Nous l'avons implémentée sans nous servir de la fonction `discrim` et avons donc dû copier/coller le code de la fonction `discrim` dans le code de la fonction `racP`. Cette manière de faire est envisageable ici car la fonction `discrim` est très courte (1 ligne de code) mais n'est pas très satisfaisante. On préférera l'implémentation suivante.

```
// racP2 est une alternative à la fonction racP
function [xPlus, xMoins] = racP2(a, b, c)
    delta = discrim(a, b, c);
    xPlus = (-b + sqrt(delta))/(2⋆a)
    xMoins = (-b - sqrt(delta))/(2⋆a)
endfunction
```

On teste la fonction `racP2` de la même façon que la fonction `racP`.

```
--> [x1, x2] = racP2(3, 5, 2)
    x2  =
      - 1.
    x1  =
      - 0.6666667
```

Évidemment, on veillera à ce que la fonction `discrim` ait été sauvegardée et exécutée avant de réaliser ce test.

IV. Les fonctions d'affichage

IV.1. Affichage des sorties : la fonction `disp`

Le fonction `disp`, abréviation du terme anglais *display* est utilisée pour réaliser des affichages dans le but d'améliorer l'expérience utilisateur. Afin de comprendre comment utiliser cette fonction, reprenons notre exemple.

Tout d'abord, on peut estimer que ce que l'on obtient lors de l'appel de la fonction `racP` n'est pas très lisible. Soulignons notamment que l'ordre d'affichage des racines (`x2` suivi de `x1`) ne correspond pas à l'ordre du calcul (`x1` suivi de `x2`), ce qui pourrait être source de confusion. On se propose donc que la valeur de chaque racine soit intégrée à une interface graphique plus explicite tel que l'affichage : « La plus grande racine vaut . . . ».

```
// racPDisp permet d'afficher les valeurs des
// deux racines d'un polynôme pris en paramètre
function racPDisp(a, b, c)
    delta = discrim(a, b, c);
    xPlus = (-b + sqrt(delta))/(2⋆a);
    xMoins = (-b - sqrt(delta))/(2⋆a);
    disp("La plus grande racine vaut " + string(xPlus));
    disp("La plus petite racine vaut " + string(xMoins));
endfunction
```

- ☞ En première ligne de la fonction, on se dispense d'arguments de sortie. Ces arguments sont a priori inutiles : on souhaite réaliser un affichage, pas utiliser les résultats de la fonction dans une autre.
- ☞ La phrase d'affichage apparaît entre guillemets `" "`, et forme une **chaîne de caractères** (*string* en anglais). L'opérateur « + » permet de coller la valeur des variables `xPlus` et `xMoins` à la suite de cette phrase. Enfin, l'opérateur `string` est utilisé pour transformer la valeur des variables en chaîne de caractères. Cette construction sera développée au « CH 4 : Types de données ».

Le test de la fonction `racPDisp` fait apparaître l'intérêt de la fonction `disp` : l'affichage du résultat est plus agréable.

```
--> racPDisp(3, 5, 2)
La plus grande racine vaut -0.6666667
La plus grande racine vaut -1.
```

IV.2. Affichage en entrée : la fonction `input`

Il est aussi possible de créer une interface graphique en entrée. Ceci est effectué par la fonction `input`, terme anglais signifiant *entrée*. Plus précisément, la fonction `input` permet de proposer à l'utilisateur de choisir la valeur des variables en entrée.

Illustrons de nouveau cette fonctionnalité à l'aide de la fonction de calcul de racines.

```
// Le script suivant demande la valeur d'un polynôme
// et affiche ses deux racines
a = input("Entrez la valeur du coefficient a : ");
b = input("Entrez la valeur du coefficient b : ");
c = input("Entrez la valeur du coefficient c : ");
delta = discrim(a, b, c);
xPlus = (-b + sqrt(delta))/(2*a);
xMoins = (-b - sqrt(delta))/(2*a);
disp("La plus grande racine vaut " + string(xPlus));
disp("La plus petite racine vaut " + string(xMoins));
```

- Nous n'avons plus affaire à une fonction puisqu'il n'y a plus d'arguments en entrée. La valeur de ces variables est entrée au clavier par l'utilisateur.
- Chaque phrase de demande de valeur apparaît entre guillemets " ". Ce sont toutes des chaînes de caractères. Chaque valeur entrée par l'utilisateur est stockée dans une variable que l'on peut utiliser par la suite.

Tester ce script correspond à l'exécuter. Pour ce type de scripts, on préférera le mode *exécution sans écho* (`F5`), qui permet d'afficher les effets des fonctions `input` et `disp` tout en évitant la recopie des lignes de code, inutile ici.

```
--> exec('/Info/Exemple_cours/racPDispInp.sce', -1)
Entrez la valeur du coefficient a : 3
Entrez la valeur du coefficient b : 5
Entrez la valeur du coefficient c : 2

 La plus grande racine vaut -0.6666667

 La plus petite racine vaut -1.
```

Les valeurs 3, 5 et 2 sont celles entrées au clavier par l'utilisateur. La première ligne témoigne du fait que l'on exécute le script nommé `racPDispInp` et détaille son adresse de sauvegarde.

IV.3. Script ou fonction ?

IV.3.a) D'un point de vue intérêt en informatique

Avec l'utilisation de la fonction `input`, on a vu comment intégrer des variables d'entrée à un script. On obtient alors des scripts dont le fonctionnement se rapproche de celui des fonctions. Il est donc légitime de se poser la question : doit-on préférer l'écriture sous forme de script ou de fonction ? Pour y répondre, on compare le code de la fonction `racP` avec sa version script enregistrée sous le nom `racPDispInp`.

- **D'un point de vue algorithmique**, la version script d'une fonction n'a pas beaucoup de sens. En effet, en privilégiant l'affichage graphique des sorties à l'aide de la fonction `disp`, on supprime une fonctionnalité primordiale de la fonction : il n'est pas possible de réutiliser directement le calcul des racines dans une autre fonction.
- **D'un point de vue utilisateur**, la version script, plus graphique, sera certainement la plus appréciée.

En résumé, utiliser des fonctions d'affichage, c'est se projeter du côté de l'utilisateur qui apprécie généralement de disposer d'une interface graphique. Au contraire, utiliser des fonctions c'est s'intéresser purement au calcul qui est réalisé. On peut ainsi concevoir d'utiliser la version script lorsque l'on fait un calcul qui n'a pas pour finalité d'être utilisé par une autre fonction.

Enfin il existe une manière simple de bénéficier à la fois des avantages de la fonction tout en améliorant l'expérience utilisateur :

1) on code d'abord la fonction (`racP` ou `racP2` dans notre exemple),

2) on lui ajoute une surcouche graphique.

Le script suivant fournit une interface graphique que l'on peut utiliser pour tester la fonction `racP`.

```
// Test de la fonction racP avec demande des entrées
// et affichage des sorties
a = input("Entrez la valeur du coefficient a : ");
b = input("Entrez la valeur du coefficient b : ");
c = input("Entrez la valeur du coefficient c : ");
[xPlus, xMoins] = racP(a, b, c);
disp("La plus grande racine vaut " + string(xPlus));
disp("La plus petite racine vaut " + string(xMoins));
```

Cette manière de faire est généralisable pour toute fonction. Il s'agit tout d'abord de demander à l'utilisateur la valeur des variables d'entrée grâce à des `input` ; puis de calculer la valeur des sorties en appelant la fonction considérée ; et enfin d'afficher la valeur de ces sorties grâce à des `disp`.

IV.3.b) D'un point de vue concours

Nous avons encore peu de visibilité sur les questions informatiques qui seront posées au concours. En effet, les premières questions **Scilab** n'apparaîtront que lors de la session 2015. Toutefois, il semblerait que la distinction script / fonction ne s'applique pas aux concours. Une des raisons principales est que les programmes informatiques que l'on vous demandera d'écrire ou simplement de comprendre seront généralement courts, sans appel à une autre fonction et auront pour but de réaliser un affichage (notamment le tracé d'une courbe ou d'un histogramme). Autrement dit, la version script risque d'être à chaque fois privilégiée aux concours.

IV.3.c) Pour les exercices de cette année

Lors des TP ou des examens, la distinction script / fonction sera effectuée même si, au vu des exigences des concours, elle ne sera pas sanctionnée. Il faudra alors choisir le code approprié en fonction de la formulation des questions. En guise d'exemple :

1) Écrire un programme **Scilab** qui prend en paramètre un polynôme du second degré et renvoie ses deux racines.
☞ *on souhaite le code de* `racP`

2) Écrire un programme **Scilab** qui prend en paramètre un polynôme du second degré et affiche ses deux racines.
☞ *on souhaite le code de* `racPDisp`

3) Écrire un programme **Scilab** qui demande un polynôme du second degré et affiche ses deux racines.
☞ *on souhaite le code du script enregistré sous le nom* `racPDispInp.sce`

Polynômes et racines ...

Lors de ce cours, nous avons volontairement parlé des « deux racines d'un polynôme du second degré » et même de la « plus grande » de ces deux racines. Cette dernière formulation est évidemment fausse car elle sous-entend que tout polynôme du second degré admet des racines réelles. Or, ce n'est le cas que pour les polynômes dont le discriminant est positif. Il faudrait donc disposer d'un mécanisme de test permettant de s'assurer du caractère positif du discriminant avant d'en calculer la racine carrée. Patience, on y viendra au « CH 7 : Conditionnelles ».