

INFORMATIQUE

ATS 1° SEMESTRE

Tiré de « Informatique Pour Tous en Classe Préparatoire »

L'informatique consiste à faire traiter de façon *automatique* de l'*information* à un ordinateur. Cela implique de gérer avec précaution la communication entre le programmeur et l'ordinateur (*choix du langage*), le stockage et la gestion des données (*format des données, export/lecture de fichier*). L'interface choisie est le logiciel Scilab, gratuit et téléchargeable ici : <https://www.scilab.org/download/>

I. Scilab

1. Description

Scilab se compose de deux fenêtres (Figure 1) :

- **L'éditeur** à gauche, dans lequel on écrira les programmes.
- **La console** se trouve au centre de la fenêtre de droite. On trouvera à gauche de la console un navigateur de fichier, et à droite le navigateur de variable (utile pour voir les variables affectées) et l'historique.

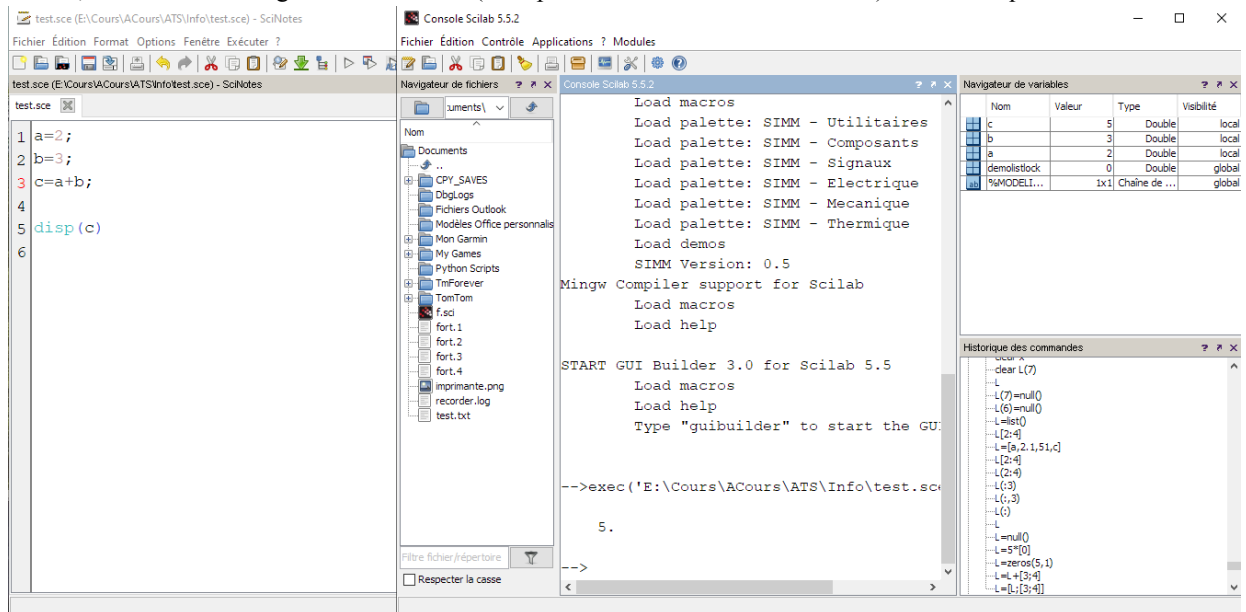


Figure 1 : Scilab

Au démarrage de Scilab, la console charge les module par défaut (voir TP), puis affiche l'invite de commande : le symbole `-->`. A la suite de ce symbole on peut taper une commande : c'est le mode *interactif* de Scilab, où chaque ligne tapée est immédiatement exécutée. Ainsi, si l'on tape une expression, sa valeur s'affiche (dans tout cet ouvrage, les entrées tapées par l'utilisateur dans l'interpréteur interactif seront notées en gras) :

```
--> 2+2
ans =
```

4.

Dans la suite du document on ne reportera plus le saut de ligne après la réponse de l'ordinateur.

Dans ce mode, on peut d'ores et déjà utiliser des variables pour stocker des valeurs. L'affectation s'écrit avec le symbole `=`. La variable mémorise la valeur qu'on lui a donnée et peut être utilisée dans la suite de la session. Le symbole `;` peut être utilisé en fin de ligne pour ne pas afficher le résultat de l'affectation :

```
--> a = 2      --> a = 2
ans = 2.      --> a+a
--> a+a      ans = 4.
ans = 4.
```

Notons que si l'on utilise dans une expression une variable à laquelle on n'a jamais donné de valeur, une erreur se produit.

```
--> z+1
!--error 4
Variable non définie : z
```

La dernière ligne de ce message indique plus précisément d'où vient l'erreur, ici de la variable `z` utilisée à tort. Enfin, il est possible de rappeler une ligne tapée précédemment à l'aide des flèches *haut* et *bas*, et de modifier cette ligne avant de relancer son calcul avec *Entrée*.

```
--> b=1;
```

```
--> b=b+1;  
--> b=b+1;  
--> b=b+1;  
--> b  
b = 4.
```

À chaque nouvelle session, les valeurs des variables sont perdues ; les instructions précédemment saisies peuvent toujours être rappelées, mais cela reste peu pratique et on n'imagine évidemment pas écrire un programme complet de la sorte. Les sessions interactives sont donc à réserver pour tester très rapidement l'évaluation de quelques expressions que l'on ne souhaite pas conserver par la suite.

2. Editeur

Dès que l'on veut écrire un programme, ou même tout simplement une suite d'instructions dont on veut garder une trace, on utilise l'éditeur. Voici un premier programme Scilab à tester :

```
disp("Bonjour !")  
x = 42  
disp(x)
```

On observe déjà plusieurs différences par rapport au mode interactif :

- Les mots-clés du langage (comme **disp**) et les nombres se colorent pour ressortir sur le reste du texte.
- Les chaînes de caractères (entre guillemets) se colorent également.
- Lorsque l'on tape une parenthèse ouvrante, la parenthèse fermante correspondante se crée automatiquement ; et lorsque l'on place le curseur sur une parenthèse, l'autre parenthèse (ouvrante ou fermante) est encadrée en rouge.

Cependant, à ce stade, le programme n'est encore qu'un texte, une suite de caractères qui n'a pas de sens pour l'ordinateur. Pour que la machine exécute (on dit aussi *interprète*) les instructions que l'on a tapées, il faut le lui demander par la commande *Enregistrer et exécuter* du menu *Exécuter* (raccourci clavier *F5* ou icône ). Les instructions sont alors lues et exécutées ; le résultat, lorsqu'il y en a un, s'affiche dans l'interpréteur interactif.

On peut noter que dans l'éditeur, les affectations sont muettes (pas d'affichage dans la console lors de l'exécution). Donc la notion de finir une ligne de programme par un ; n'a plus de sens.

II. Représentation des nombres

1. Représentation de l'information par des booléens

Les ordinateurs et les programmes mémorisent, transmettent et transforment des données aussi variées que des nombres, des textes, des images, des sons, etc. Pourtant, quand on les observe à une plus petite échelle, les ordinateurs ne manipulent que des objets beaucoup plus simples : des 0 et des 1.

La mémoire des ordinateurs est constituée d'une multitude de petits circuits électroniques qui, chacun, ne peuvent être que dans deux états : sous tension ou hors tension. Techniquement, il serait possible de construire des circuits ayant plus de deux états, correspondant à différentes valeurs de tensions, mais les risques d'erreurs dans le stockage et la transmission de ces états deviennent vite beaucoup plus importants que les avantages qu'on pourrait en tirer. Comme il fallait donner un nom à ces états, on a décidé de les appeler 0 et 1. Une telle valeur, 0 ou 1, est dite booléenne. On l'appellera booléen, chiffre binaire ou encore *bit* (*binary digit*). Un tel circuit à deux états s'appelle un circuit mémoire un bit.

L'état d'un circuit composé de plusieurs de ces circuits est représenté par une suite finie de 0 et de 1, que l'on appelle un mot. Par exemple, le mot 100 décrit l'état d'un circuit composé de 3 *bits*.

Exercice 1 Trouvez trois informations de la vie courante qui peuvent être exprimées par un booléen.

Exercice 2 On imagine un ordinateur dont la mémoire est constituée de quatre circuits mémoire un bit. Quel est le nombre d'états possibles de la mémoire de cet ordinateur ? Même question pour un ordinateur dont la mémoire est constituée de dix circuits mémoire un bit. Et pour un ordinateur dont la mémoire est constituée de 32 milliards de tels circuits ?

Exercice 3 On veut représenter chacune des sept couleurs de l'arc-en-ciel par un mot, les sept mots devant être distincts et de même longueur. Quelle est la longueur minimale de ces mots ?

2. Conversion de base k en base 10

SAVOIR-FAIRE Représenter en base k un entier naturel donné en base dix

Si l'on note $a_i \dots a_1 a_0$ l'écriture de n en base k , elle est obtenue par l'algorithme suivant

Données : n, k

$i \leftarrow 0$

tant que $n \neq 0$ **faire**

$a_i \leftarrow$ reste de la division euclidienne de n par k

$n \leftarrow$ quotient de la division euclidienne de n par k

$i \leftarrow i + 1$

Résultat : la représentation $a_i \dots a_1 a_0$.

SAVOIR-FAIRE Représenter en base dix un entier naturel donné en base k

Pour trouver la représentation en base dix d'un entier naturel donné en base k , on utilise le fait qu'en base k , le chiffre le plus à droite représente les unités, le précédent les paquets de k , le précédent les paquets de $k \times k = k^2$, le précédent les paquets de $k \times k \times k = k^3$, etc.

On retrouve les opérations réciproques de celles effectuées dans l'algorithme de conversion en base k : si $a_i \dots a_1 a_0$ est l'écriture de n en base k , alors

$$n = a_0 + a_1 k + a_2 k^2 + \dots + a_i k^i$$

Exercice 4 Représenter le nombre $(47)_{10}$ en base 2 (binaire) :

Exercice 5 Calculer en base 10 le nombre $(101101)_2$

3. Le binaire : « base 2 »

Dans la mémoire des ordinateurs, les circuits mémoire un bit sont souvent groupés par huit (les octets) et on utilise des nombres exprimés en notation binaire sur un, deux, quatre ou huit octets, soit 8, 16, 32 ou 64 bits. Cela permet de représenter les nombres de 0 à 1111 1111 = 255 sur un octet, de 0 à 1111 1111 1111 1111 = 65 535 sur deux octets, de 0 à 1111 1111 1111 1111 1111 1111 1111 1111 = 4 294 967 295 sur quatre octets, etc.

Lorsque l'on manipule des entiers naturels uniquement représentés de cette façon, dans certains langages on les appelle entiers *non signés*. Scilab est un logiciel de calcul numérique, alors même lorsque vous entrez un entier, il est stocké comme un flottant (type double dans le navigateur de variable).

Exercice 6 Trouver la représentation en base deux du nombre 100.

Exercice 7 On considère un disque dur de 1 To (1 Terra-octet). A combien de *bits* cela correspond-t-il ? On rappelle 1 To = 10^9 octets

Exercice 8 Donner les représentations en base deux des nombres 1, 3, 7, 15, 31 et 63. Expliquer le résultat par rapport aux nombres en 2^k .

Exercice 9 On cherche à mettre au point une procédure de calcul du produit de deux entiers naturels représentés en binaire. Dans tout cet exercice, on pourra supposer par commodité que les entiers sont représentés sur 32 bits ; la généralisation à un nombre de bits quelconque ne pose cependant pas de difficulté particulière.

Pour multiplier par dix un entier naturel exprimé en base dix, il suffit d'ajouter un 0 à sa droite, par exemple, $12 \times 10 = 120$. Quelle est l'opération équivalente pour les entiers naturels exprimés en base deux ? Illustrer cette remarque par des exemples.

On appelle cette opération *décalage à gauche*, car dans les microprocesseurs, au lieu d'ajouter un 0 à droite, on déplace tous les bits vers la gauche. Sur quels entiers cette opération n'est-elle pas possible sans commettre d'erreur ?

Quelle est l'opération effectuée sur un entier naturel en base deux si on lui fait subir deux décalages à gauche ? trois décalages ? n décalages ?

À l'aide des remarques précédentes, étant donnés deux entiers naturels n et m , exprimer le produit $n \times m$ en fonction de n et de la décomposition $m_j \dots m_1 m_0$ de m en base deux.

En déduire un algorithme pour effectuer le produit de deux entiers naturels représentés en base deux utilisant uniquement : des additions ; des décalages à gauche ; des tests sur un bit d'un nombre. Les multiplications effectuées par les microprocesseurs fonctionnent sur ce principe.

4. Représentation des entiers relatifs

Sur 1 octet (8 bits) on peut stocker les entiers naturels de 0 à 255, ou les entiers relatifs de -128 à +127. On définit le codage *binaire naturel signé* (*bns*) en réservant le bit de poids fort au signe : 0 pour positif, et 1 pour négatif.

De manière générale sur n bits, on peut coder de -2^{n-1} à $2^{n-1}-1$.

Pour trouver le codage d'un nombre négatif : exemple pour -62

- déterminer le nombre de bits nécessaire (8,16, 32 ou 64) : $2^{n-1} > 62 \Leftrightarrow n=8$
- partir du codage du nombre positif sur n bits : $62 = (00111110)_2$
- inverser les 0 et les 1 : (11000001)
- ajouter 1 : $-62 = (11000010)_{bns}$

Décimal	Codage bns
127	$(01111111)_{bns}$
126	$(01111110)_{bns}$
...	
0	$(00000000)_{bns}$
-1	$(11111111)_{bns}$
...	
-127	$(10000001)_{bns}$
-128	$(10000000)_{bns}$

Remarque : On parle de complément à 2 car si on ajoute les deux nombres :
$$\begin{array}{r} 00111110 \\ + 11000010 \\ \hline 100000000 \end{array}$$

= $(00000000)_{bns}$ sur 8 bits !

Exercice 10 Trouver les représentations décimales des entiers relatifs dont les représentations binaires sur huit bits sont 00010111 et 10001100.

5. Représentation des nombres à virgules

La conversion en binaire s'étend aussi aux nombres à virgules, avec les puissances négatives de 2 ! On retrouve donc les coefficients de la conversion par multiplications par 2 jusqu'à ce que la partie décimale soit nulle :

$$0,375 \times 2 = 0,75 \quad 0,75 \times 2 = 1,5 \quad 0,5 \times 2 = 1,0 \quad (\text{on attend que la partie décimale soit nulle})$$

Le nombre $(5,375)_{10}$ s'écrit ainsi en base 2 : $(101,011)_2$

Conversion IEEE 754

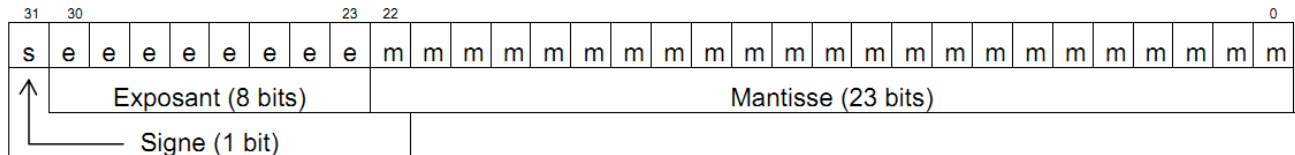
Le standard IEEE 754 définit le codage des réels sur 32 et 64 bits. Prenons le cas du nombre $x = -243,25$ à coder sur 32 bits. Il faut en premier lieu convertir sa valeur absolue en binaire :

Exercice 11 : Montrer que $243,25 = (11110011,01)_2$

On écrit ensuite ce nombre binaire sous forme « scientifique » : $1,mmmmm*2^n$. n est le nombre de décalage vers la gauche. Ici, $(11110011,01)_2 = (1, 111001101*2^7)_2$

Le chiffre 1 avant la virgule étant toujours présent, on ne code que la partie décimale : (111001101) c'est la **mantisse**.

Le code simple précision se met sous la forme suivante en 32 bits :



Le **signe** est codé sur 1 bits (de poids le plus fort) : « 0 » pour positif et « 1 » pour négatif.

L'exposant n est codé sur 8 bits en **binaire signé**, différent du binaire naturel signé. Il faut donc ajouter 127 à l'exposant à coder.

Ici, $n=7$. Il faut donc coder $127+7 = 134 = (1000\ 0110)_2$

La **mantisse** code la partie décimale (*Attention au 1 avant la virgule non codé !*) sur les 23 bits restants.

Ici, la mantisse est $(111001101)_2$ sur 23 bits soit : $(111001101000000000000000)$

Attention on complète les zéros à droite ici car on code la partie décimale !

La représentation de -243,25 est donc : (1 10000110 111001101000000000000000)

Exercice 12 : Calculer la valeur décimale de (0100 0010 0010 1110 1000 0000 0000 0000)

Dépassement ou soupassement

En 64 bits, le plus grand exposant représentable est +1023, le plus petit est -1022. On ne peut pas excéder ces valeurs !

Arrondi et erreurs numériques

Il est impossible de représenter de façon **exacte** sur un nombre fini de bit n'importe quel entier rationnel. Il suffit pour s'en convaincre de calculer la conversion en binaire du nombre $n = 0.1$: il est « irrationnel » en base 2 !

D'autres erreurs d'arrondis se présentent lorsque l'on effectue des calculs, notamment entre des nombres dont les ordres de grandeur sont très différents, par exemple, en Scilab, on peut observer :

--> 1 + 2^-54 - 1	--> 1 - 1 + 2^-54
ans = 0.0	ans = 5.551D-17

La principale conséquence est qu'un test d'égalité $a=b$ n'aura pas de sens si a et b sont deux nombres à virgule flottante. On le remplacera donc par une condition de la forme **abs(a-b)<ε** où ε est une valeur proche de zéro, choisie en fonction du problème à traiter et de l'ordre de grandeur des erreurs auxquelles on peut s'attendre sur a et b .

A chaque multiplication de deux nombres à virgule flottante, en ne gardant que 52 chiffres binaires après la virgule, on introduit une erreur relative de l'ordre de $2^{-52} \approx 2,2 \cdot 10^{-16}$. Ces erreurs s'ajoutent avec le nombre de calculs !

III. Les variables, les chaînes de caractères et les listes

1. Opération mathématiques sur les nombres entiers ou flottants

Les opérations mathématiques classiques sont disponibles + , - , * , /. La puissance x^y est notée x^y .

Le quotient et le reste de la division euclidienne sont accessibles par // et %, respectivement.

Le terme D (ou e) désigne une puissance de 10, on peut donc remplacer $1.2*10^{**4}$ par $1.2e4$.

On peut, en cas de besoin, obtenir la troncature de x par la fonction *int*. Attention *int(x)* ne calcule pas la partie entière de x mais le tronque, c'est-à-dire calcule la partie entière de $|x|$, puis l'affecte du signe de x .

2. Les booléens

Les booléens constituent un type spécial n'ayant que deux valeurs, *True* et *False*. Ils servent à représenter le résultat de l'évaluation d'expressions logiques : tests d'égalités, de comparaisons, d'appartenance, ... Il sont représentés par T ou F en Scilab.

Les expressions logiques réalisables sont le NON, le ET et le OU ; traduits par NOT, AND (&) et OR (|). Ces règles d'évaluations sont résumées dans le tableau suivant :

<i>b1</i>	<i>b2</i>	not b1	b1 and b2	b1 or b2
True	True	False	True	True
True	False	False	False	True
False	True	True	False	True
False	False	True	False	False

3. Comparaisons et égalités entre nombres flottants ou entiers

Le test d'égalité (réalisables uniquement sur des valeurs entières) s'écrit $a=b$. Le double égal permet de différencier de l'affectation ($a=2$).

Le test d'inégalité peut s'écrire *not* ($a=b$) ou plus simplement $a <> b$.

Les tests de comparaisons s'écrivent : $<$, $>$, $<=$, $>=$.

4. Les variables

Déclaration et affectation

En Scilab, il n'est pas nécessaire de déclarer le type de ses variables comme en C/C++ ou dans d'autres langages. Toutes les variables sont stockées en *flottant*.

Manipulations et erreurs classiques

On considère l'expression : $x=x+1$

```
--> x = x+1
      !--error 4
      Variable non définie : x
```

Si, lors de l'évaluation d'une expression, un nom de variable est utilisé alors qu'il n'apparaît pas dans la mémoire de l'ordinateur, il devient impossible d'attribuer une valeur à cette expression. On retrouve ainsi le message d'erreur, classique au début de programmation, que *la variable 'x' n'est pas définie..*

On rajoute donc avant l'instruction précédente l'expression $x=2$.

Lors de l'évaluation de $x=x+1$, la variable x est substituée par sa valeur 2 à droite du signe =, le résultat est stocké dans la mémoire de l'ordinateur ($2+1=3$). Puis ce résultat est affecté à la variable à gauche du =, donc x prendra la valeur 3.

Exercice 1 : Par quelles instructions peut-on incrémenter (augmenter de 1) ou décrémenter (diminuer de 1) la valeur de x ?

Exercice 2 : On commence un programme par les instructions suivantes :

```
x=1
y=2
x=y
y=x
```

Quelles seraient les valeurs de x et y à la fin de l'exécution ? Corriger ce programme pour échanger les valeurs des variables x et y.

Exercice 3 Partant de l'état initial vide, décrire l'évolution de l'état lors de l'exécution des instructions suivantes, évaluées du haut vers le bas :

```
x = 3
y = 2
x = 1 + y * x
y = y
y = 1.2 - x
```

5. Les listes

Pour construire une *liste*, il suffit de placer des expressions entre crochet séparées par des virgules. Comme pour les autres valeurs, il est possible de stocker une liste dans une variable :

```
--> L = [1, 2.2] ;
```

Pour accéder aux éléments de la liste, c'est-à-dire aux sous-valeurs qu'il contient, on utilise l'expression $L(i)$ où i est le numéro de la composante, on parle de son *indice*. Attention en Scilab les indices commencent à 1 !

```
--> L (1)          --> L(2)
ans = 1            ans = 2.2
```

Longueur d'une liste

On obtient la longueur, c'est-à-dire le nombre d'éléments, d'une liste à l'aide de la fonction **length** :

```
--> length( L )
ans = 2
```

Opérations mathématiques

On peut ajouter deux listes (de même longueur) entre elles, multiplier une liste par un nombre, ou ajouter un nombre à tous les éléments d'une liste par les instructions suivantes :

```
--> L + [1, -1]      --> L*2          --> L + 1
ans = [2, 1.2]      ans = [2, 4.4]    ans = [2, 3.2]
```

On ne peut pas multiplier deux listes entre elles car il n'y a pas de définition mathématique du produit !

Ajout d'un élément à une liste

On peut ajouter un élément à une liste en affectant directement l'indice, même s'il n'existe pas ! Ici on ajoute un terme à l'indice 3 :

```
--> L(3)=-2
L = [1, 2.2, -2]
```

On peut aussi concaténer à la liste l'élément à ajouter :

```
--> L=[L, 4]
L = [1, 2.2, -2, 4]
```

Liste vide

On définit une liste vide par l'instruction suivante :

```
--> L = [ ] ;
```

Liste nulle

Si l'on souhaite définir une liste d'une ligne et 10 colonnes de zéros, on utilise la fonction *zeros* :

```
--> C = zeros(1,10)
C = [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ]
```

Extraction d'une sous-liste

On peut extraire les éléments situés entre les indices 2 à 4 (inclus) par **L(2:4)**.

6. Chaînes de caractères : strings

Construction

Le type des chaînes de caractères, *string* en anglais et dans Scilab, est celui permettant de représenter des textes. On considère dans un premier temps des textes élémentaires, ceux composés d'une unique lettre ; on les appelle les *caractères*.

En Scilab, les caractères peuvent être n'importe quelle lettre de l'alphabet, mais aussi des symboles, comme les signes de ponctuation. Dans les exemples ci-dessous on a affecté les caractères à des variables :

```
--> x = 'a'           --> y = '?'
```

Une chaîne de caractères est une suite finie de caractères consécutifs, qu'on note entre apostrophes ' ' ou guillemets " " :

```
--> z = 'Ceci est une chaine'
```

La chaîne vide se note ' ' ou "" (sans espace entre les guillemets !)

Accès aux caractères

C'est différent d'une liste : **z(1)** ne correspond pas à 'C'. Il faut utiliser la fonction *part* :

```
--> x=part(z, 1)
x = 'C'
```

Longueur

La longueur s'obtient également par **length(z)** qui donnera 19 (car les espaces sont un caractère)

Concaténation

On peut également concaténer deux chaînes de caractères en utilisant le signe + (noter aussi le code de l'apostrophe) :

```
--> 'Scilab c' ' est' +' super'
ans = 'Scilab c' est super'
```

Sous-chaîne

Un ensemble de caractères consécutifs à l'intérieur d'une chaîne s'appelle une sous-chaîne. Pour extraire une sous-chaîne de *s*, utilise la fonction *part* avec *i* l'indice du premier caractère de la sous-chaîne et *j* l'indice du dernier caractère :

```
--> part(z, 2 :12)
ans = 'eci est une'
```

Conversion de chaîne de caractères en nombres

On peut convertir une valeur d'un type simple vers une chaîne de caractères à l'aide de la construction **str(e)** :

```
--> string(1.2)
ans = 1.2
```

La conversion n'est pas visible à l'affichage, mais le résultat est bien du texte !
--

Il est possible aussi de convertir une chaîne vers un nombre de type double par *strtod* :

```
--> strtod( '-1.52' )
ans = -1.52
```

IV. Algorithmique

Un algorithme est une procédure permettant de résoudre un problème, écrite de façon suffisamment détaillée pour être suivie sans posséder de compétence particulière ni même comprendre le problème que l'on est en train de résoudre. Pour faire fonctionner un algorithme, il faut donc lui fournir des données précisant l'instance du problème qu'il devra traiter. En retour, l'algorithme construira un résultat.

Nous allons voir ici les trois instructions qui suffisent, à elles seules, à exprimer tous les algorithmes (chapitres 3, 4 et 5). Les deux premiers chapitres sont nécessaires pour comprendre la structure d'un algorithme en Python.

1. Test simple

Une instruction conditionnelle n'est exécutée que si une condition donnée est vérifiée par l'état courant.

Pour traduire cela, on utilise l'instruction **if**, qui a en Scilab la syntaxe suivante :

```
if condition then
    bloc_d_instructions
end
```

Le *bloc d'instructions* est exécuté uniquement si *condition* est vérifiée. Dans l'exemple qui suit, on ajoute 1 à la variable *x* seulement si, dans l'état courant, elle a une valeur impaire ; si *x* a une valeur paire, l'instruction d'ajout est ignorée :

```
if x == 0 then
    x = x + 1
end
```

2. L'indentation

Un bloc est une suite d'instructions qui s'exécutent les unes après les autres. Pour identifier sans ambiguïté les instructions appartenant au bloc du *if*, **il est prudent de les indenter**.

Indenter signifie ajouter un même nombre d'espaces (ou de tabulations) devant chacune des lignes définissant les instructions. Les deux expressions suivantes auront le même résultat, mais l'un est plus lisible que l'autre !

```
y = 3
if x != 0 then
    y = y / x
end
```

```
y = 3
if x != 0 then
    y = y / x
end
```

Exercice 1 : Quel sont les résultats des programmes suivants dans l'état (a=0, b=2, c=1) ?

```
if a == 0 then
    b = 4
end
if a != 0 then
    c = 5
end
b = 1
```

```
if a == 0 then
    b = 4
    if a <> 0 then
        c = 5
    end
end
b = .1
```

3. Instruction conditionnelle : Test complet

Dans l'instruction conditionnelle complète, le *if* est forcément présent, il y aura autant de *else if* que nécessaire, et le *else* est optionnel. *elseif* correspond à « sinon si », donc une autre condition est nécessaire derrière, *else* correspond à « sinon », ou alors « dans tous les autres cas » et permet donc de réaliser une instruction si tous les autres cas sont faux.

```
if condition1 then
    bloc_instruction1
elseif condition2 then
    bloc_instruction2
else
    bloc_instruction3
end
```

Exercice 2 : Que fait ce programme ? a, b, c sont les entrées, m est le résultat.

```
if a > b
    if a > c
        m = a
    else
        m = c
    end
else
    if b > c
        m = b
    else
        m = c
    end
end
```

Exercice 3

1 Écrire un programme qui calcule la négation d' une variable booléenne sans utiliser l' opérateur *not*.

2 Écrire un programme qui calcule le *et* logique de deux variables booléennes sans utiliser l' opérateur *and*.

3 Écrire un programme qui calcule le *ou* logique de deux variables booléennes sans utiliser l' opérateur *or*.

4. Boucle conditionnelle

En français, si l'on peut reformuler un énoncé avec une phrase contenant « Tant que ... », alors on a besoin pour le programmer d'une boucle conditionnelle *while*. En Python, on l'écrit ainsi :

```
while condition          // Tant que la condition est vraie
    bloc_d_instructions //      Le bloc d' instructions est exécuté
end
```

Exercice 4 Faire afficher le rang n du dernier terme strictement positif de la suite récurrente définie par $u_{n+1} = u_n / 2 - 3n$, la valeur initiale étant donnée par l' initialisation de la variable u :

```
u=1027
n=0
while .....
    u=u/2-3*n

end
```

5. Boucle inconditionnelle

À l'aide de la construction **for** $i = 1 : n$ l'indice i variera de 1 à n par pas de 1 (par défaut), on effectuera donc un nombre d'itérations n donné. Ainsi le programme de calcul de 2^n sans avoir l'opérateur puissance, puisque $2^n = 2 * 2 * \dots * 2$, on aura juste à écrire :

```
p = 1
for i = 1:n
    p = 2 * p
end
disp(p)
```

Le compteur de boucle est ici entièrement géré par la boucle **for**. Il n'est pas besoin de l'initialiser, il est **interdit de l'incrémenter** et il n'est pas nécessaire d'évaluer de condition de sortie : la valeur du compteur i variera de 1 à n .

Exercice 5 : Ecrire un programme permettant de calculer la factorielle de n

La notion de pas

La structure de la boucle *for* s'écrit de façon complète **for** $i = \text{debut} : \text{pas} : \text{fin}$. Par défaut *pas* vaut 1 s'il n'est pas précisé. Les valeurs *debut* et *fin* sont incluses. Le pas peut avoir une valeur décimale, ou négative si l'on a $\text{debut} > \text{fin}$.

Exercice 6 : Ecrire une boucle qui fait varier x de 0 à 3 (inclus) par pas de 0,1.

Exercice 7 : Ecrire une boucle qui fait varier x de -2 à 1 (inclus) par pas de 0,05.

Parcourir les valeurs d'une liste ou d'une chaîne de caractère

Voici comment calculer la somme des éléments d'une liste en utilisant une boucle *for* :

```
t = [1, 5, 9]
s = 0
for i = 1 : length(t)    // length(t) vaudra 3 ici
    s = s + t(i)
end
print(s)
```

De même, voici comment compter les espaces dans une chaîne de caractères. La fonction *part(c,i)* permet d'accéder à la i-ème lettre de la chaîne de caractère *c*.

```
chaîne = 'Les itérables sont vraiment épatants.'
esp = 0
for i = 1 : length(chaîne)
    x=part(c, i)
    if x == ' ' // Avec un espace entre les guillemets !
        esp = esp + 1
    end
end
disp(esp) // Le résultat sera esp=4 ici
```

Arrêt d'une boucle for

On peut arrêter une boucle *for* par l'instruction *break*. Il faut en pratique éviter de l'utiliser car le programme devient difficile à comprendre. Il faut plutôt adapter son programme en utilisant une boucle *while* !

Exercice 8 Programmer les problèmes suivants :

Calculer la valeur absolue d'un nombre donné.

Calculer la norme d'un vecteur dans un espace vectoriel de dimension $n=3$ (pour une des normes usuelles).

Déterminer tous les diviseurs d'un nombre entier donné n .

Déterminer le plus petit diviseur (différent de 1) d'un nombre entier donné n .

Exercice 9 : Écrire un programme qui détermine si un automobiliste est en excès de vitesse, connaissant sa vitesse et le type de voie sur lequel il circule (donné sous forme d'une chaîne de caractères) : voie = 'autoroute', 'route' ou 'ville'.

Puis l'adapter pour qu'en cas d'excès de vitesse, il donne le montant de la contravention encourue. Celle-ci est de 68 € si l'excès commis est inférieur à 20 km/h, 135 € au-delà.

Exercice 10 Écrire un programme qui détermine l'ordre de grandeur d'un nombre x non nul donné, autrement dit l'entier relatif n tel que $10^n \leq |x| < 10^{n+1}$. On

n'utilisera pas les fonctions logarithmes du module *math*.

Exercice 11 Écrire un programme qui trouve le plus petit multiple commun à 2 entiers naturels m et n .

Exercice 12

Écrire un programme qui reçoit comme donnée un quadruplet (*jours, heures, minutes, secondes*) et vérifie que celui-ci respecte les conventions habituelles sur les durées (secondes entre 0 et 59, etc.).

Écrire un programme qui reçoit une durée exprimée en secondes et construit un quadruplet (*jours, heures, minutes, secondes*) désignant la même durée et respectant les conventions habituelles sur les durées.

Exercice 13 *

Écrire un programme qui calcule la valeur d'un entier connaissant son écriture binaire fournie sous forme d'une chaîne de caractères : par exemple $c = '110101'$.

Écrire un programme qui construit dans une chaîne de caractères l'écriture binaire d'un entier naturel donné.

V. Fonctions

On utilisera les fonctions pour des programmes qui pourront être exécutés plusieurs fois. Cela évitera les répétitions.

1. Définition d'une fonction

Le mot clé pour **définir** une fonction et le mot **function**. La structure est la suivante :

```
function param_sortie = nomdefonction (param_entree )
    bloc_instructions
endfunction
```

Le *nomdefonction* est à choisir librement par l'utilisateur. Les *param_entree* recensent les variables **externes** dont dépend la fonction. A l'intérieur de la fonction, vous pouvez définir autant de variables internes que nécessaire, ces variables ne seront connues que « dans » la fonction. Le but étant d'arriver au calcul du résultat qui est *param_sortie*. La variable *param_sortie* sera aussi interne à la fonction. On dissociera la **définition** de fonction (qui permet d'exprimer la méthodologie d'obtention du résultat) de son **exécution** (qui permettra d'obtenir la valeur du résultat).

On définit une fonction qui calcule le l'hypoténuse d'un triangle de côtés *x*, *y* et *z* (l'hypoténuse).

```
function z=hypotenuse(x, y)
    z = (x^2 + y^2)^0.5
endfunction
```

On l'utilise, par exemple pour calculer l'hypoténuse du triangle dont les côtés valent 3 et 4, de la manière suivante dans la console :

```
--> hypotenuse(3.0, 4.0)
Ans = 5.
```

Dans cette instruction, l'appel **hypotenuse**(3.0, 4.0) transmet les arguments effectifs 3.0 et 4.0 à la fonction, en lieu et place des arguments formels *x* et *y*. La fonction **hypotenuse** calcule alors la valeur $(x^2 + y^2)^{0.5}$ et l'affecte à *z*. En Scilab, si la fonction est exécutée dans la console, le résultat est affiché par défaut. Si vous voulez sauvegarder le résultat dans une variable, il faut utiliser l'expression :

```
c=hypothenuse(3,4)
```

2. Fonctions renvoyant plusieurs variables

Cette définition permet facilement de définir plusieurs variables en entrée, on peut le faire aussi facilement pour les paramètres de sortie. Exemple de calcul de la somme et de la différence de *a* et *b* :

```
function [x,y] = sd (a,b) :
    x=a+b
    y=a-b
endfunction
```

Si l'on exécute cette fonction dans la console, seul le premier argument est renvoyé :

```
--> sd(4,5)
ans = 9.
```

Si l'on souhaite utiliser (et exploiter) le résultat dans un script, on peut exécuter :

```
[somme,diff] = sd(4,5)
```

les variables *somme* et *diff* seront alors affectées aux valeurs 9 et -1, respectivement.

Exercice 1 Écrire en Python une fonction qui prend comme argument un entier *n* et renvoie l'entier $2*n$.

Exercice 2 * Écrire en Python une fonction qui prend comme argument un entier *n* et renvoie un booléen qui indique si cet entier est premier ou non. On pourra tester l'indivision de *n* par tous les entiers de 2 à *n*-1 en utilisant le reste de la division entière par la fonction *modulo*.

Exercice 3 ** Écrire une fonction qui calcule le PGCD des deux entiers naturels passés en arguments en suivant l'algorithme d'Euclide, à l'aide d'une boucle **while**.

Exercice 5.22 On écrit un programme qui gère automatiquement le score au tennis.

1. Écrire une condition booléenne qui permet de décider si un joueur a gagné un jeu.
2. Définir une fonction qui compte les points au cours d'un jeu. En entrée, on demande répétitivement quel joueur, 1 ou 2, gagne le point ; au fur et à mesure,

on calcule et on affiche le score. Le programme s'arrête dès qu'un joueur gagne le jeu, après avoir affiché le nom du vainqueur.

3. Pour faciliter les tests, écrire une seconde version de cette fonction avec pour seule entrée une chaîne de caractères qui contient les numéros successifs des joueurs marquant les points ; la fonction lit cette chaîne caractère par caractère pour compter les points. Ainsi, l'entrée '211222' sera comprise comme : le joueur 2 gagne un point, puis le joueur 1 en gagne deux, puis le joueur 2 en gagne trois et le joueur 2 gagne donc le jeu.
4. Écrire une deuxième fonction qui compte les jeux au cours d'un set et s'arrête lorsqu'un joueur gagne le set. Cette fonction fera appel à la précédente pour savoir qui gagne les jeux. On n'oubliera pas de prévoir le cas particulier du jeu décisif.
5. Écrire une troisième fonction qui compte les sets et s'arrête lorsqu'un joueur gagne le match. On pourra, avant de commencer le match, demander en combien de sets gagnants il est joué.

VI. Les listes (ou les tableaux), et les matrices

1. Les listes (ou tableaux)

Exercice 1 : Programmer une fonction $evaluer(x, A)$ réalisant l'évaluation du polynôme $P(x)$ où la valeur x est donnée en entrée, et la liste A contient les coefficients du polynôme :

$$P(x) = \sum_{i=0}^n a_i x^i \text{ avec } A = [a_0, a_1, \dots, a_n]$$

Exercice 2* : Écrire une fonction qui prend un tableau d'entiers t en argument et renvoie le tableau des sommes cumulées croissantes correspondantes, autrement dit un tableau de même taille dont la k -ième composante vaut $\sum_{i=0}^k t[i]$. Le tableau fourni en argument ne sera pas modifié.

Réfléchir à une version plus efficace ?

Exercice 3 : Écrire une fonction qui renvoie un tableau contenant les n premières valeurs de la suite de Fibonacci. On considère la suite de Fibonacci définie par :

$$\begin{cases} F_0 &= 0 \\ F_1 &= 1 \\ F_n &= F_{n-2} + F_{n-1} \text{ pour } n \geq 2. \end{cases}$$

2. Recherche dans un tableau trié

Recherche séquentielle

On cherche à déterminer si un tableau A contient une certaine valeur x . On écrit la fonction suivante :

```
function resultat = appartient(x, A)
    resultat = %F // %F correspond à False
    for i = 1:length(A)
        if A(i) == x:
            resultat = %T
        end
    end
endfunction
```

Le *resultat* est initialisé à la valeur *Faux*. Dès que l'élément x est trouvé, la valeur de *resultat* passe à *Vrai*. Dans le cas où l'élément x ne se trouve pas dans la liste A , l'instruction associée à la boucle conditionnelle *if* ne sera pas exécutée et la valeur renvoyée sera *Faux*. Pour qu'il n'y ait qu'un seul résultat renvoyé (*Vrai* ou *Faux*), il ne faut surtout pas réattribuer à la valeur *resultat* la valeur *Faux* dès que le terme $A(i)$ ne correspond pas à x .

On va maintenant écrire une fonction de recherche légèrement différente, qui renvoie le premier indice où la valeur x apparaît dans le tableau s .

```
function indice= valeur_ind(x, A)
    indice = -1 // indice est initialisé à une valeur impossible
    for i = 1:length(A)
        if A(i) == x:
            indice = i
        end
    end
endfunction
```

Recherche dichotomique dans un tableau trié

On note que les fonctions **appartient** et **valeur_ind** précédentes parcourent tout le tableau et effectuent donc n comparaisons, où n est la longueur du tableau. Dans le cas où le tableau est trié, la recherche d'un élément dans un tableau peut être réalisée de manière plus efficace. On peut alors exploiter l'idée suivante : on coupe le tableau en deux par le milieu et on détermine si la valeur x doit être recherchée dans la moitié gauche ou droite. En effet, il suffit pour cela de la comparer avec la valeur centrale. Puis, on répète le processus sur la portion sélectionnée.

On suppose par exemple que l'on cherche la valeur 9 dans le tableau [1, 3, 5, 6, 9, 12, 14]. La recherche s'effectue ainsi :

On cherche dans $a[1:7]$.	1	3	5	6	9	12	14
On compare $x=9$ avec $a[4]=6$.	1	3	5	6	9	12	14
On cherche dans $a[5:7]$.					9	12	14
On compare $x=9$ avec $a[6]$					9	12	14
On cherche dans $a[5:5]$.					9		
On compare $x=9$ avec $a[5]=9$.					9		

Seules trois comparaisons ont été nécessaires pour trouver la valeur. C'est une application du principe *diviser pour régner*. On retrouvera d'autres applications de ce principe, appelé *dichotomie* dans la résolution d'équations et les tris.

Pour écrire l'algorithme, on délimite la portion du tableau A dans laquelle la valeur x doit être recherchée à l'aide de deux indices g et d , pour *gauche* et *droite*. On commence par initialiser les variables g et d avec 1 et **length(a)**, respectivement :

```
function ind = recherche_dichotomique(x, A)
    g=1
    d=length(A)
    ind = -1      // On initialise l'indice à une valeur impossible
```

Tant que la portion à considérer contient au moins un élément :

```
while g <= d
```

on calcule l'indice de l'élément central, en faisant la moyenne de g et d :

```
    m = (g + d) // 2
```

Il est important de noter qu'on effectue ici une division *entière*. Qu'elle soit arrondie vers le bas ou vers le haut, on obtiendra toujours une valeur comprise entre g et d , ce qui assure d'une part que $A[m]$ existe et qu'il est bien situé entre g et d . Si $A[m]$ est l'élément recherché, on a terminé la recherche :

```
    if A[m] == x
        ind = m
```

Sinon, on détermine si la recherche doit être poursuivie à gauche ou à droite. Si $A[m] < x$, on poursuit à droite :

```
    elseif A[m] < x
        g = m + 1
```

Sinon, on poursuit à gauche :

```
    else:
        d = m - 1
    end
end      // fin du while
endfunction // fin de la fonction
```

Si on sort de la boucle **while**, c'est que l'élément ne se trouve pas dans le tableau, car il ne reste que des éléments strictement plus petits (à gauche de g) ou strictement plus grands (à droite de d). La valeur renvoyée pour l'indice sera alors le -1 affecté au début du programme

Le code complet est donné programme [1](#) ci-dessous.

PROGRAMME 1 Recherche dichotomique dans un tableau trié

```
function ind = recherche_dichotomique(x, A)
    g=1
    d=length(A)
    ind = -1
    while g<=d
        m = (d+g) // 2
        if A(m) == x
            ind = m
        elseif A(m)<x
            g = m+1
        else
            d = m-1
        end
    end
endfunction
```

Exercice 4 Écrire une fonction qui renvoie l'élément maximal d'un tableau d'entiers. On discutera des diverses solutions possibles pour traiter le cas d'un tableau de longueur 0. Faire attention aux tableaux dont les valeurs sont toutes négatives...

Exercice 5 * Écrire une fonction qui renvoie les deux plus grands éléments d'un tableau d'entiers. On supposera que le tableau est de longueur au moins 2 ; en revanche, on veillera à ne le parcourir qu'une seule fois.

Exercice 6 Écrire une fonction qui renvoie l'indice de la première occurrence de l'élément maximal d'un tableau d'entiers.

Exercice 7 Concevoir un algorithme vérifiant qu'une liste est rangée en ordre croissant.

Exercice 8 * Écrire une fonction qui vérifie qu'une chaîne de caractères est un palindrome, c'est-à-dire qu'elle est identique qu'on la lise de gauche à droite ou de droite à gauche.

Adapter cette fonction pour qu'elle ne tienne pas compte des espaces ni des signes de ponctuation.

3. Les matrices

En scilab, les listes ou les matrices sont définies comme les listes. On rajoute simplement un symbole « ; » pour changer de ligne.

Affectation

La matrice (2, 3) suivante $M = \begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \end{bmatrix}$ sera représentée par la liste suivante :

--> $M = [0, 1, 2 ; 3, 4, 5]$

En particulier, on accède à l'élément $M_{i,j}$ avec l'expression $M(i,j)$.

Initialisation

Si l'on souhaite initialiser une matrice M de dimensions (n,p) avec uniquement des 0, on utilise la fonction zeros :

--> $M = \text{zeros}(n, p)$

Dimensions

La fonction size permet d'obtenir en résultat le nombre de lignes, suivi du nombre de colonnes :

--> $\text{size}(M)$
ans = 2. 3.

Exercice 9 Créer une fonction *transposition(M)* renvoyant en sortie la transposée de M.

Exercice 10 Créer une fonction *produit(M,N)* renvoyant en sortie le produit des matrices $M \times N$. Dans le cas où les dimensions des matrices sont incompatibles, il faudra renvoyer *False*.

Exercice 11 * Le but de cet exercice est d' écrire une fonction qui détermine la médiane d' un tableau d' entiers. Programmer l' algorithme suivant : on recherche le minimum et le maximum du tableau, on les supprime et on recommence jusqu' à obtenir un tableau de longueur inférieure ou égale à 2. On déduit alors facilement la valeur de la médiane du ou des entiers restants.

Exercice 12 Écrire une fonction qui détermine si tous les éléments d' un tableau (d' entiers ou de flottants) sont positifs ou nuls. On veillera à sortir de la fonction dès qu' une valeur négative est rencontrée.

Exercice 13 Écrire une fonction qui prend en arguments trois entiers m , n et p et renvoie un tableau identique à celui désigné par l' expression `list(range(m, n, p))`, bien entendu sans utiliser cette dernière expression.

Exercice 14 Écrire une fonction qui prend un entier n en argument et renvoie un tableau de n entiers tirés aléatoirement dans l' intervalle $[0, n]$. On utilisera la fonction `random.randint`

Exercice 15 Écrire une fonction qui prend en argument un tableau et inverse l' ordre de ses éléments, sans utiliser la fonction `reversed`.

Exercice 16 Écrire une fonction `appliquer( t )` qui prend pour argument un tableau t et renvoie le tableau formé des images de chaque élément de t par la fonction `cosinus`. La fonction `map` de Python réalise une opération similaire.

Exercice 17 * Écrire une fonction qui vérifie si un tableau donné en argument est dépourvu de doublons (aucune valeur n' y apparaît deux fois).

Exercice 18 Écrire une fonction qui calcule le produit scalaire de deux vecteurs représentés par deux tableaux de même taille.

VII. Tracé de fonctions $y=f(x)$

1. Définition de la fonction f

Définissons une fonction $f(x)$:

```
function y = f(x)
    y=exp(x**2)
endfunction
```

2. Variation du paramètre d'entrée X

On définit les valeurs de X par *début* : *pas* : *fin* :

$X = 0 : 0.1 : 2$

3. Calcul des valeurs de Y

On dispose maintenant dans une liste X des valeurs d'entrée de la fonction, nous allons stocker chaque valeur $y=f(x)$ dans une liste Y :

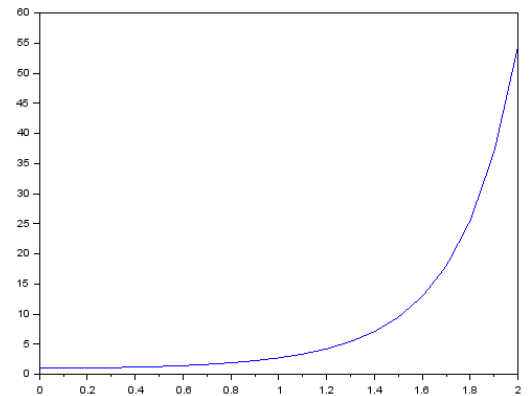
```
Y= []
for i = 1 : length(X)
    Y(i) = f(X(i))
end
```

4. Tracé

Puis on souhaite tracer une liste en fonction de l'autre par la fonction *plot*

```
plot( X , Y )
```

On obtient donc en sortie le graphe 'sans fioritures' de la fonction f sur l'intervalle $[0, 2]$.

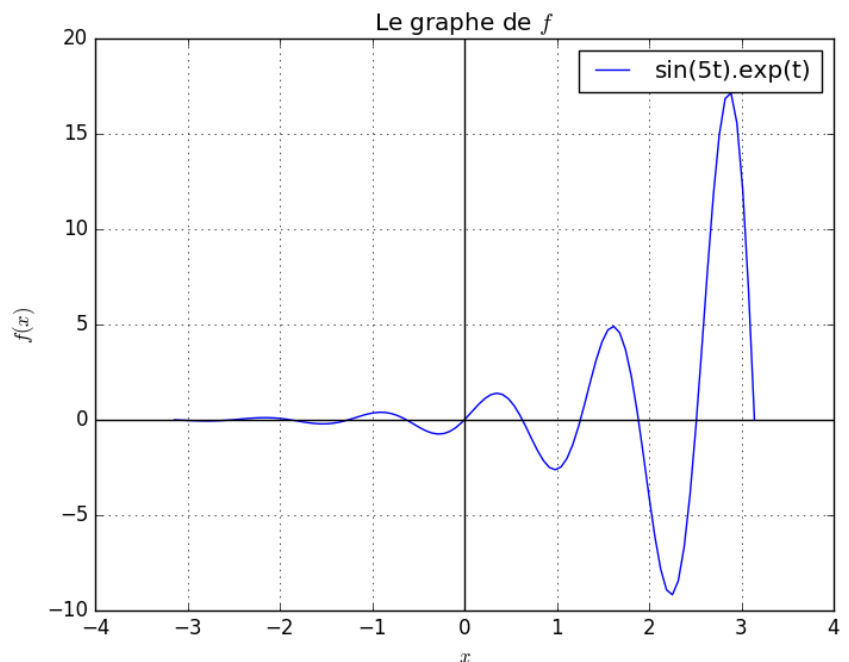


On peut rajouter des options en suivant le programme suivant :

PROGRAMME 3 Tracé

```
plot(x,y)

xgrid // pour créer une grille
title('Le graphe de f') // Le titre
xlabel('x') // le nom de l'axe horizontal
ylabel('y=f(x)') // et de l'axe vertical
legend('sin(5t).exp(t)')
```



VIII. Quelques algorithmes non vus en TD, pour s'exercer

1. Résolution d'une équation du second degré avec gestion de la comparaison à zéro

1. Écrire un programme qui, étant donnée une équation du second degré à coefficients entiers, détermine le nombre de ses solutions et leurs valeurs éventuelles.
2. Tester ce programme pour l'équation $x^2 + 6x + 9 = 0$.
3. Adapter ce programme pour qu'il accepte des coefficients non entiers.
4. Tester ce second programme sur les équations $x^2 + 10^{-10} = 0$ et $x^2 - 10^{-10} = 0$.
5. Montrer qu'une infime variation sur l'un des coefficients sépare les cas où l'équation a des solutions des cas où elle n'en a pas.
6. Tester le second programme sur l'équation $0, 1x^2 + 0, 6x + 0, 9 = 0$. Comparer les résultats avec ceux obtenus à la question 2. Que doit-on en penser ? Comment explique-t-on ce phénomène ?
7. Tester ce second programme sur l'équation $x^2 + (1 + 2^{-50})x + 0, 25 + 2^{-51} = 0$ et comparer les résultats obtenus avec ceux que prévoit la résolution exacte de cette équation. Expliquer encore une fois les différences constatées.
8. On travaille pour cette question sous l'hypothèse (raisonnable vus les calculs effectués) que la norme IEEE 754 assure les propriétés suivantes des arrondis :
Si le discriminant est strictement positif, alors sa valeur approchée calculée est positive ou nulle.
De même, si le discriminant est strictement négatif, alors sa valeur approchée calculée est négative ou nulle.
Enfin, si le discriminant est nul, sa valeur approchée calculée peut être quelconque. Adapter alors ce programme pour qu'il n'affiche que des affirmations certaines à propos des polynômes qui lui sont fournis.
9. Enfin, on peut quantifier l'erreur commise sur le calcul du discriminant. On a vu dans le chapitre II.5 que l'erreur relative introduite par une multiplication est de 2^{-52} : quelle est donc l'erreur relative maximale commise dans le calcul du discriminant ? En-dessous de quel seuil doit-on considérer que le discriminant pourrait être nul ? Adapter le programme en conséquence.

2. Algorithme d'Euclide

Algorithme d'Euclide original

Soient $u, v \in \mathbb{N}$. L'algorithme suivant, dit *algorithme d'Euclide*, calcule le plus grand diviseur commun (PGCD) de u et v :

A1 Si $v = 0$ alors la réponse est u .
A2 Faire $(u, v) \leftarrow (v, u \bmod v)$. Retourner en A1.

Le terme *mod* désigne le modulo. Écrire une fonction **euclide(u,v)** implantant cet algorithme.

Algorithme d'Euclide étendu

Soient $u, v \in \mathbb{N}$. L'algorithme d'Euclide peut être adapté pour calculer, en même temps que le PGCD de u et v , les coefficients de Bezout. L'algorithme suivant calcule un triplet (u_1, u_2, u_3) tel que $uu_1 + vu_2 = u_3 = u \wedge v$.

B1 $(u_1, u_2, u_3) \leftarrow (1, 0, u)$.
B2 $(v_1, v_2, v_3) \leftarrow (0, 1, v)$.
B3 Si $v_3 = 0$ alors la réponse est (u_1, u_2, u_3) .
B4 Soit $q = \lfloor u_3/v_3 \rfloor$. Faire
 $(t_1, t_2, t_3) \leftarrow (u_1, u_2, u_3) - q(v_1, v_2, v_3)$
 $(u_1, u_2, u_3) \leftarrow (v_1, v_2, v_3)$
 $(v_1, v_2, v_3) \leftarrow (t_1, t_2, t_3)$
Retourner en B3.

Écrire une fonction **bezout(u,v)** qui réalise cet algorithme.

Application : division modulo m

Soient $u, v, m \in \mathbb{N}^*$ tels que v et m soient premiers entre eux. On appelle quotient de u par v modulo m tout entier w tel que $0 \leq w < m$ et $u \equiv v.w \pmod{m}$.

Écrire une fonction calculant le quotient de u par v modulo m .

3. Décomposition en facteurs premiers

Étant donné $n \in \mathbb{N}$, n s'écrit de manière unique sous la forme $n = p_1 p_2 \dots p_k$ avec $p_1 \leq p_2 \leq \dots \leq p_k$ et p_i premier. On se propose ici de déterminer les facteurs p_i .

Méthode par division

La méthode la plus simple est la suivante : si $n > 1$, on teste sa divisibilité par les nombres premiers successifs $p = 2, 3, 5, \dots$ jusqu'à ce que $n \equiv 0 \pmod{p}$. On remplace alors n par n/p et on reprend à p . Lorsque $n \not\equiv 0 \pmod{p}$ avec $n/p \leq p$, on s'arrête, avec n premier.

Cette méthode a l'inconvénient qu'il faut déterminer la suite des nombres premiers. Toutefois, on peut simplifier cette méthode de la façon suivante. Soit $n \in \mathbb{N}^*$ dont on souhaite déterminer la décomposition en facteurs premiers. Soit (d_i) une suite d'entiers

$$2 = d_0 < d_1 < d_2 < \dots$$

qui inclut tous les nombres premiers $\leq \sqrt{n}$ et au moins un élément $d_k \geq \sqrt{n}$. Alors, l'algorithme suivant détermine la décomposition de n en facteurs premiers :

```
A1  $t \leftarrow 0, k \leftarrow 0$ .  
A2 Si  $n = 1$  c' est terminé.  
A3 On divise  $n$  par  $d_k$  :  $n = q.d_k + r$  avec  $(0 \leq r < d_k)$ .  
A4 Si  $r = 0$  alors  $t \leftarrow t + 1, p_t \leftarrow d_k, n \leftarrow q$ . Aller en A2.  
A5 Si  $q > d_k$  alors  $k \leftarrow k + 1$  et aller en A3.  
A6  $t \leftarrow t + 1, p_t \leftarrow n$ . C' est terminé.
```

Écrire une fonction **decomp** prenant en argument l'entier n , une suite (d_k) pour n , et renvoyant la suite croissante des facteurs premiers p_i de n .

On pourrait prendre pour (d_k) la suite $2, 3, 5, 7, \dots$, c'est-à-dire 2 puis tous les impairs à partir de 3. On prendra plutôt la suite $2, 3, 5, 7, 11, 13, 17, 19, 23, 25, \dots$, c'est-à-dire ajouter alternativement 2 et 4 à partir de 5 (on supprime ainsi tous les multiples de 2 et 3).

Écrire une fonction **dk** prenant n en argument d'entrée et renvoyant une telle suite $d_0, \dots, d_k$ avec $d_k \geq \sqrt{n}$.

Recherche de grands nombres premiers

On s'attache maintenant à trouver une méthode plus efficace pour tester la primalité d'un entier. On s'intéressera ici au test de Fermat :

Soit n un entier. On dit que n passe le test de Fermat de base a si $a^{n-1} \equiv 1 \pmod{n}$. On sait que si n est premier, alors pour tout entier $a \in]0, n[$, n passe le test de Fermat de base a . Donc, par contraposée, si n ne passe pas le test de Fermat de base a , alors n n'est pas premier. On dit dans ce cas que a est un témoin de non-primalité de n pour le test de Fermat. Si n passe le test de Fermat de base a mais n'est pas un nombre premier, on dit que n est un pseudo-premier de Fermat de base a .

Écrire une fonction **passe_fermat** (a, n) renvoyant une valeur booléenne indiquant si n passe le test de Fermat de base a .

Y a-t-il des entiers compris entre 1 et 1 000 qui soient des pseudo-premiers de Fermat de base 2 (on les appelle aussi nombres de Poulet) ? Lesquels ?

Y a-t-il des entiers n compris entre 1 et 1 000 qui soient des pseudo-premiers de Fermat de base a pour tout a ? Pour tout a appartenant à $]0, n[$?

Pour tester si a^{n-1} est congru à 1 modulo n , on peut calculer $a^{n-1} \% n$. Combien de temps ce calcul prend-il pour $n = 100\,000\,000$ et $a = 2$?

Python propose une fonction spécialisée pour effectuer ce calcul : **pow**. On peut calculer **pow**($a, n-1, n$). Comparer avec le temps d'exécution précédent.

Les nombres premiers sont très utiles en cryptographie. Le logiciel de chiffrement PGP utilise ainsi les tests de Fermat de bases 2, 3, 5 et 7 pour décider si un nombre est premier. Écrire une fonction **premier_PGP**(n) « testant » si un nombre est premier avec la méthode utilisée par PGP.

En considérant que ce test est parfait, écrire une fonction **premier_suivant**(n) rendant le plus petit nombre premier strictement supérieur à n .

Chercher expérimentalement jusqu'à quelle valeur de n on obtient une réponse en moins de dix secondes.

4. Cryptographie : la méthode RSA

Parmi les procédés cryptographiques, la méthode RSA, découverte par R. Rivest, A. Shamir et L. Adleman en 1978, est l'une des plus utilisées actuellement. Elle fait partie des méthodes dites à *clé publique* : chaque personne possède une clé *P publique* dont tout le monde peut avoir connaissance (par exemple dans un annuaire) et une clé *S secrète* qu'elle seule connaît. Lorsqu'on veut envoyer un message *M* à une personne *A*, on le code avec la clé publique P_A du destinataire. Ce dernier le décode alors avec sa clé secrète S_A et peut le lire.

Un tel système fonctionne donc si on a les propriétés suivantes :

1. $S(P(M)) = M$ pour tout message *M*.
2. Les paires (S, P) sont toutes distinctes.
3. Découvrir la clé secrète *S* à partir de la clé publique *P* est aussi difficile que déchiffrer le message codé.
4. Une paire (S, P) peut se calculer facilement.

La méthode RSA fonctionne de la manière suivante : soient *x*, *y* et *s* trois grands nombres premiers, tels que $x, y \leq s$. Soient $N = x.y$ et *p* tel que $p.s \bmod (x-1)(y-1) = 1$. On peut alors montrer que pour tout *M*, on a $(M)^{p.s} = M \pmod{N}$.

La clé publique est alors le couple (N, p) et la clé secrète le couple (N, s) . Pour coder un message, on commence par le découper en entiers inférieurs à *N* et on élève alors ces entiers à la puissance *p* modulo *N*. Pour le décoder, on élève les entiers composant le message à la puissance *s* modulo *N*.

En supposant que l'on dispose d'un générateur de grands nombres premiers, comment engendrer un couple clé publique-clé secrète pour la méthode RSA ?

Un message est codé de la manière suivante. Chaque caractère est d'abord remplacé par son code ASCII (65 pour A, 66 pour B, etc.). Le message est ensuite chiffré en utilisant la clé publique (N, p) où $N = 49808911$ et $p = 5685669$. Le résultat est le suivant :

49583279, 4553592, 17767401, 16172223, 33062955, 33599637,
17767401, 11607763, 17767401, 9275561, 11607763, 35959722, 17767401,
44022065, 3148857, 17767401, 40136246, 13922222, 16172223, 17767401,
4553592, 11607763, 16172223, 25708244, 11607763, 25708244, 3148857,
36260425, 17767401, 35959722, 3148857, 29735027, 4553592, 3148857,
29538442, 3148857, 16172223, 31299930, 17767401, 44022065, 3148857,
17767401, 40136246, 13922222, 16172223, 35959722, 17767401, 9275561,
45114532, 13922222, 25708244, 45114532, 11607763, 29538442, 29538442,
3148857, 29735027, 45114532, 35959722

« Craquer » ce codage RSA pour découvrir le message en clair. On rappelle que la fonction `chr` renvoie le caractère dont on donne le code ASCII et que l'on peut afficher le caractère *c* sans qu'il soit suivi d'un retour chariot avec `print(c, end="")`.

Donner une idée des tailles de nombres premiers qu'il faut choisir pour ne plus pouvoir casser ce codage par factorisation de *N*. Il est à noter qu'il existe d'autres méthodes pour casser le codage choisi ici sans factoriser *N*. En pratique la mise en œuvre de RSA est donc plus complexe.