Compte rendu du projet de LM206

Thème : La cryptographie

Sommaire :

1. Présentation

Scilab est un environnement de calcul numérique qui permet d'effectuer rapidement toutes les résolutions et représentations graphiques couramment rencontrées en mathématiques appliquées. L'utilisation d'un tel environnement est désormais inséparable de l'activité du mathématicien : c'est un intermédiaire incontournable entre la calculatrice et un langage compilé comme C.

L'invite de la ligne de commande (prompt) est constituée d'une « flèche » : deux tirets et un signe supérieur -->. L'instruction est tapée puis validée avec la touche de retour (Enter, Return). Le résultat est affiché à la suite, sauf si la ligne se termine par un point-virgule auquel cas le résultat est caché.

Scilab utilise les fonctions et opérateurs classiques (+, -, *, /, ^ ou **, sqrt( ) pour la racine carrée, cos( ) pour le cosinus, int( ) pour la partie entière, round( ) pour l'arrondi au plus proche, abs( ) pour la valeur absolue…) et quelques autres (par exemple rand( ) pour avoir un nombre aléatoire entre 0 et 1). La fonction who affiche les variables déclarées. La variable ans contient le dernier résultat.

Le séparateur décimal est le point. Pour entrer l'imaginaire i, il faut utiliser %i ; il figure simplement sous la forme « i » dans les résultats. Pour entrer l'infini ∞, il faut utiliser %inf ; il figure simplement sous la forme « inf » dans les résultats. La valeur de π s'obtient par %pi, et la constante de Neper e par %e.

Scilab accepte un certain nombre d'instructions :

- <u>exécution conditionnelle</u> :

```
if condition then
  instruction
else
  instruction
end
```

- <u>boucle itérative</u> :

```
for variable = début : fin          OU          for variable = début : pas : fin
  instruction                                     instruction
end                                             end
```

- **boucle itérative avec condition :**

```
while condition do          OU          while condition do
  instruction                               instruction
end                                       else
                                            instruction
                                          end
```

Il est possible de définir des fonctions avec passage de paramètres. La fonction est un sous-programme avec ses variables propres, et qui peut contenir des boucles, des branchements conditionnels.

On peut écrire les instructions dans un fichier texte (avec scipad), puis faire exécuter dans Scilab. Le code source peut contenir des commentaires introduits par deux barres de fraction //.

2. Calcul du PGCD et du PPCM

Avec le logiciel scilab, nous pouvons créer des fonctions récursives (qui consiste à faire des rappels de fonctions jusqu'à ce que l'on atteigne la condition d'arrêt), itératives (pour cela on utilise des boucles for ou while) ou encore des fonctions qui font appels à d'autres fonctions dans leurs instructions.

Nous pouvons ici voir avec l'exemple du calcul du PGCD, que l'on peut l'écrire de deux manières différentes en récursif puis en itératif :

```
function [pgcd] = pgcdREC (a, b)
  if (a == 0) then                    //première condition d'arrêt
    [pgcd] = b
  elseif (b == 0) then                //deuxième condition d'arrêt
    [pgcd] = a
  elseif (a < b) then
    [pgcd] = pgcdREC (a, b-a)         //premier appel de la fonction
  else
    [pgcd] = pgcdREC (a-b, b)         //deuxième appel de la fonction
  end
endfunction
```

```
function [a] = pgcdITE (a, b)
  while (b <> 0)          //boucle while (tant que la condition n'est pas vérifier)
    aux = b
    b = modulo (a, b)
    a = aux
  end
endfunction
```

En faisant un appel de la première et de la deuxième fonction avec deux très grands nombres, puis en faisant appel à la fonction timer( ), qui est intégrée au logiciel nous pouvons voir que pgcdREC met du temps pour effectuer sont calcul alors que pour pgcdITE le résultat est instantané.

Nous avons par ailleurs pour le calcul du PPCM utiliser un appel de la fonction pgcdITE qui fait un calcul direct après avoir calculé la valeur du pgcd:

```
function [X] = ppcm (a, b)
  AB = a * b
  X = AB / pgcdITE (a, b)     //appel de la fonction qui calcul le pgcd itérativement
Endfunction
```

Cependant, dans scilab, il existe déjà des fonctions intégrées au logiciel. C'est-à-dire, que pour calculer le PGCD d'entiers ou de polynômes, on utilise la fonction gcd qui prend en argument un tableau à n éléments, par exemple :

```
x = int32 ( [12,18,24] )
[pgcd] = gcd (x)
      // Ces instructions renvoient :
pgcd =
  6
```

De même, nous pouvons calculer le PPCM grâce à la fonction lcm :

```
x = int32 ( [12,18,24] )
[ppcm] = lcm (x)
      // Ces instructions renvoient :
ppcm =
  114
```

Scilab nous offre donc des outils pour que l'on effectue nos calculs plus simplement (moins de lignes de programmes) et plus rapidement au niveau du temps d'exécution.

3. Algorithme RSA

RSA est un algorithme asymétrique de cryptographie à clé publique, très utilisé dans le commerce électronique, et plus généralement pour échanger des données confidentielles sur Internet. Cet algorithme a été décrit en 1977 par Ron Rivest, Adi Shamir et Len Adleman, d'où le sigle RSA.

RSA repose sur le calcul dans les groupes $\mathbb{Z}/n\mathbb{Z}$.Grace à cet algorithme, une personne X envoie une clé publique qui contient deux éléments : n (produits de deux nombres premiers p et q) et e (un nombre qui est premier à l'indicatrice d'Euler : $\varphi(n)$). Seule cette personne connaît la clé privée qui elle aussi à deux éléments : n (qui correspond au n de la clé publique) et d (un entier tel que $ed \equiv 1 \pmod{\varphi(n)}$). Grace a la clé publique, un personne Y en voie un message codé M tel que $M = m^e \pmod n$. X reçoit donc ce message et le décode grâce à la clé privée et obtient le message m tel que $m = M^d \pmod n$.

La fonction suivante permet de codé un message. Tout d'abord, on initialise les variables globales à la fonction que l'on va stocker dans un tableau ce qui nous permettra d'accéder plus rapidement à l'élément que l'on veut:

```
p = 11;                                  // Nombre premier
q = 23;                                  // Nombre premier
n = p * q;                               // Produit des deux nombres
euler = (p - 1) * (q - 1);               // Indicatrice d'euler
e = 13;                                  // Nombre premier à Euler
d = 17;                                  // Inverse de e
u = [p, q, n, euler, e, d];              // Tableau avec les données
m = 21;                                  // Message à codé
```

On écrit alors le programme de la fonction qui va coder le message m, nous utiliserons une fonction de la bibliothèque scilab qui est modulo (a, b) qui calcule a modulo b :

```
function [mes_signe] = signature (a, tab)
  N = tab(5)                                    // Affectation de la valeur de e
  mes_signe = 1
  for i = 1 : N
    mes_signe = modulo (mes_signe * a, tab(3)) // Calcul de m^e (mod n)
  end
endfunction

[mes_signe] = signature (m, u)                  // Renvoie la valeur de M
```

La valeur renvoyée par ce programme et pour ces valeurs est :

```
mes_signe =
    65
```

De la même manière, nous pouvons décoder ce message avec un programme très ressemblant mais on calculera alors M^d (mod n). Cette fonction a les mêmes variables que la fonction précédente :

```
x1 = 22
x2 = 21
function [mes_verifie, m] = verification (x, a, tab)
  N = tab (6)                                   // Affectation de la valeur de e
  m = 1
  for i = 1 : N
    m = modulo (m * a, tab (3))                 // Calcul de M^d (mod n)
  end
  if (m == x) then
    mes_verifie = 'C''est le bon message'
  else
    mes_verifie = 'Ce n''est pas le bon message'
  end
endfunction

[mes_verifie, m] = verification (x1, mes_signe, u)     // Renvoie la valeur de m
[mes_verifie, m] = verification (x2, mes_signe, u)     // Renvoie la valeur de m
```

Les valeurs renvoyées par ces deux appels de fonctions sont :

```
m =
   21                                  // Résultat du premier appel
mes_verifie =
     Ce n'est pas le bon message

m =
   21                                  // Résultat du deuxième appel
mes_verifie =
     C'est le bon message
```

Nous avons ici codé cette algorithme en effectuant une boucle for pour chacune des fonctions, qui est indispensable car si l'on effectue le calcul modulo (m^e, n) ou modulo (M^d, n), nous n'aurons pas de résultats exacts car le nombre mis à la puissance est très grand. C'est pour cela, que no faisons chacune des multiplications et en même temps nous calculons le modulons pour pouvoir rester dans l'intervalle [1 ; n].

Même avec l'utilisation de boucle, la fonction est rapide à l'exécution.

4. Algorithme de Vigenère

C'est un système de substitution poly-alphabétique ou de chiffrement poly-alphabétique. Cela signifie qu'il permet de remplacer une lettre par une autre qui n'est pas toujours la même, contrairement au Chiffre de César ou à ROT13 qui se contentaient d'utiliser la même lettre de substitution.

Ce chiffrement introduit la notion de clé. Une clé se présente généralement sous la forme d'un mot ou d'une phrase. Pour pouvoir chiffrer notre texte, à chaque caractère nous utilisons une lettre de la clé pour effectuer la substitution. Nous devons donc répéter la clé autant de fois qu'il y a de mots. Par exemple :

Clé : SCILAB
Texte : LE THEME EST LA CRYPTOGRAPHIE

Texte en clair : LE THEME EST LA CRYPTOGRAPHIE
Clé répétée : SC ILABS SCI LA BSCILABSCILAB

Puis par la suite, dans un tableau de 26 lignes et 26 colonnes nous plaçons les lettres telles que nous obtenons :

a	b	c	d	e	f	g
b	c	d	e	f	g	h
c	d	e	f	g	h	i
d	e	f	g	h	i	j
e	f	g	h	i	j	k
f	g	h	i	j	k	l
g	h	i	j	k	l	m

Nous pouvons voir que pour crypter la lettre E (5ème colonne) par C (3ème ligne) nous obtenons la lettre G, de même nous avons pour la lettre E (5ème colonne) par A (1ère ligne) la lettre E. puis nous avons à la fin une chaine de caractère de même longueur que la chaîne que l'on veut crypter. Par ailleurs, les espaces sont conserver dans nos fonctions que l'on à programmées.

Tout d'abord, nous créons un tableau à 26 éléments qui contient toutes les lettres de l'alphabet. Sachant que a se trouve à la position 1 et z à la position 26.

```
alphabet = ['a', 'b', 'c', ' d', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o',
'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z'] ;
```

Il sera plus simple de coder le message écrit en lettre par un message écrit en chiffre qui correspondront aux positions de chacune des lettres dans le tableau alphabet définit précédemment, car pour trouver la lettre codée, nous devons faire certains calculs.

La fonction ci-dessous permet donc de coder une chaine de caractère en un tableau d'entiers compris entre 0 et 26 (car 0 correspond à un espace entre 2 mots). Pour cela, nous utilisons plusieurs fonctions de scilab :

length(msg) calcule la longueur d'un tableau ou d'une chaîne de caractère.

strsplit(msg,[1 :(n-1)]) qui permet de casser une chaine de caractère (msg) de longueur n en un tableau contenant chacune des lettres comme un élément.

zeros(1, n) crée un tableau contenant n zéros.

find(alphabet == x) trouve la position de l'élément x dans alphabet (si celui-ci existe)

```
function [msg_numerique] = codeur_msg (msg)
  msg_casse = strsplit (msg, [1 : length (msg) - 1])  // Créer un tableau de lettres
  longueur = length (msg)                             // Calcul la longueur du message
  msg_numerique = zeros (1, longueur)                 // Créer un tableau de n zéros
  for i = 1 : longueur
    msg_numerique (i) = find (alphabet == msg_casse (i))
  end
endfunction
```

```
[msg_numerique] = codeur_msg ('scilab')
msg_numerique  =                                      // Résultat de la fonction
   19.    3.    9.    12.    1.    2.
```

A partir de cette fonction, nous allons créer notre programme qui codera un message grâce à une clé :

```
function [msg_num_code] = vigenere_codage (msg, cle)
  [msg_numerique] = codeur_msg (msg)          // Appel de la fonction précédente
  [cle_numerique] = codeur_msg (cle)          // Appel de la fonction précédente
  longueur_msg = length (msg)                 // Calcul de la longueur du message
  longueur_cle = length (cle)                 // Calcul de la longueur de la clé
  msg_num_code = ''                           // Initialisation du résultat
  j = 0                                     // Valeur pour les décalages des espaces
  for i = 1 : longueur_msg                  // i est la position dans msg_numerique
    if (msg_numerique (i) == 0) then
      msg_num_code = msg_num_code + ' '     // Espace si la valeur est égale à 0
    else
      k = modulo (j, longueur_cle) + 1   // k est la position dans cle_numerique
      a = cle_numerique (k) + msg_numerique (i) - 1
      position = modulo (a, 26)                 // Position de la lettre codée
      if (position == 0) then
        position = 26                           // La valeur 0 est 26 (mod 26)
      end
      lettre = alphabet (position)              // Lettre codée
      msg_num_code = msg_num_code + lettre        // Concaténation des lettres
      j = j + 1
    end
  end
endfunction
```

```
[msg_num_code] = vigenere_codage ('je suis en cours','scilab')
msg_num_code  =                                 // Résultat de la fonction
    bg afit wp kzusk
```

Nous pouvons voir, que nous devons faire attention a fait qu'un multiple de 26 modulo 26 est égale à 0, or si l'on veut prendre l'élément placé à la position 0, alors on a un problème. Pour cela, nous avons fait intervenir des -1 et +1 à certains endroits du la fonction.

Pour le décodage, il suffit d'effectuer les calculs inverses, et au lieu de faire -1, on fait +27, à cause du même problème :

```
function [msg_num_decode] = vigenere_decodage (msg_num_code, cle)
  [msg_numerique] = codeur_msg (msg_num_code)
  [cle_numerique] = codeur_msg (cle)
  longueur_msg = length (msg_num_code)        // Calcul de la longueur du message
  longueur_cle = length (cle)                 // Calcul de la longueur de la clé
  msg_num_code = ''                           // Initialisation du résultat
  j = 0                                     // Valeur pour les décalages des espaces
  for i = 1 : longueur_msg                  // i est la position dans msg_numerique
    if (msg_numerique (i) == 0) then
      msg_num_decode = msg_num_decode + ' '// Espace si la valeur est égale à 0
    else
      k = modulo (j, longueur_cle) + 1    // k est la position dans cle_numerique
      a = msg_numerique (i) - cle_numerique (k) + 27
      position = modulo (a, 26)                // Position de la lettre codée
      if (position == 0) then
        position = 26                          // La valeur 0 est 26 (mod 26)
      end
      lettre = alphabet (position)             // Lettre codée
      msg_num_decode = msg_num_decode + lettre   // Concaténation des lettres
      j = j + 1
    end
  end
endfunction
```

```
function [msg_num_decode] = vigenere_decodage (msg_num_code,'scilab')
msg_num_decode  =                             // Résultat de la fonction
      je suis en cours
```

Pour programmer l'algorithme, nous avons quelques petits soucis pour utiliser les chaines de caractères. Grâce à la fonction strsplit(msg,[1 :(n-1)]), nous avons compris le système puis réussis à trouver un algorithme efficace.

5. Conclusion

En conclusion, nous pouvons dire que scilab nous permet de faire les algorithmes mathématiques que l'on souhaite, sans compter qu'il nous offre beaucoup de fonctions pour faire des calculs, même parfois complexes.

Ce thème nous à attiré par le fait que nous avons étudié en LM220 l'algorithme RSA, mais aussi c'est très intéressant de comprendre la logique mathématique d'un sujet qui n'utilise pas forcément des calculs et pouvoir l'appliquer en informatique, pour pouvoir faire nos programmes de cryptages et de décryptages.

Par ailleurs, ce cours fut enrichissant du point de vue de la connaissance en informatique. En effet, étant étudiants en maths et intéressés par l'informatique, nous avons pu découvrir un nouveau logiciel de programmation mais qui est aussi un logiciel de calcul qui pourra nous être utile plus tard.