Notes complémentaires
au cours de NODAL de R. Cailliau

Sommaire du cours :

1. Commandes essentielles en NODAL (explication et exemple)

2. Analyse et écriture d'un programme à partir de l'exemple CURVE- PLOT.

3. Exercices par groupe et explications des points difficiles.

4. Adaptation à la structure d'un language et aux particularités d'un système.

Références

- Manuel NODAL (français ou anglais)
- Listing et Print-out de Curve-Plot
- Notes du cours (me demander celles qui vous manquent)

The Development of a program
Analys du probleme Curve-Plot
Liste des excercies (et résultats lorsque vous me les aurez données)

Annexes

- Utilisation de SINTRAN III et SYNTRON II
- Emploi du QED pour écrire et modifier les programmes
- Display layout sheet
1. **Commandes NODAL**

Dans tous les langages, on peut regrouper la syntaxe des ordres en quatre groupes :

1.1 Commande d'entrée/sortie
1.2 Création des variables
1.3 Opérateurs
1.4 Ordres de branchement

1.1 **Entrée/Sortie**

- **ASK**  
  Lit sur un terminal un tampon de 80 caractères ASCII et si c'est une variable numérique, le convertit en binaire

- **TYPE**  
  Envoie dans un terminal, une chaîne de caractères ASCII (80 au maximum) après l'avoir converti, si c'est une variable numérique, selon le format spécifié.

**Formats** :

* ! Reviens à la ligne suivante
  * % Utiliser le format E
  * $3 Laisser trois espaces
  * \X Taper l'équivalent ASCII de X
  * %5,.3 X Taper X selon 12.123 digits.

1.2 **Création des variables**

- **SET A = 10**  
  permet de créer une variable numérique A avec la valeur initiale 10.

- **$SET B = "NOM"**  
  permet de créer une chaine de caractère B de deux mots :
  
  \[ N \quad O \quad M \quad b \]

Si ces variables existent déjà, l'interprèteur NODAL les reconnaît dans sa liste de variables, et alors, ne fait que changer leurs valeurs.

La réservation de tableau se fait par l'ordre DIMENSION
- DIM A(10) crée un tableau numérique A de dix mots
- DIM-S NAME.(10) crée un tableau de caractères NAME de 10 chaînes.

1.3 Opérateurs
- Arithmétiques : ÷, *, /, −, + par hiérarchie décroissante
- Logiques : <, =, >, <>
- Deux fonctions utiles : INT(X) donne la partie entière de X
  (mais sans arrondir)
  MOD(X,3) donne le reste de la division \( \frac{X}{3} \)

Au lieu d'écrire des expressions illisibles utiliser des variables intermédiaires.

1.4 Ordres de branchement

a) Boucle + compteur + une sortie = WHILE (condition); ....

\[\text{Test} \quad \text{Faux} \rightarrow \text{Vrai} \]

ou compteur

\[\downarrow\]

Pour sortir, il faut donc que la condition du compteur soit modifiée dans la boucle :

**Exemple**  \( A = 100 \)

WHILE \( A > 1; T ! A; \) SET \( A = A - 1 \)
b) L'aiguillage IF, n'existe pas en NODAL et le IF se représente par

```
IF habituel

IF (condition)
  Then
  Else
```

```
IF en NODAL

```

c) DO permet de diviser le programme en blocs "sub-routines" ayant une action simple et bien définie et de les exécuter, là où c'est nécessaire.

```
Main programme

  Début
  
  DO 10 10 % lecture des valeurs
  Fin

```

d) FOR peut être considéré comme un cas particulier en WHILE.

2. Analyse d'un programme

Nos connaissances sont suffisantes maintenant pour essayer d'analyser un programme tel que CURVE- PLOT !

Quelques règles simples pour nous aider à

- comprendre le problème
- le diviser en parties simples
- comprendre le fonctionnement et l'articulation de ces parties ..

De plus, ce système indépendant du language de programmation utilisé, permet d'écrire des programmes précis, clairs, faciles à comprendre, et à modifier. Enfin, il permet une certaine standardisation des programmes.
3. Trois boîtes suffisent pour la représentation des actions

pour symboliser une action du programme

" " une boucle " "

" " une décision

Une autre boîte qui n'existe pas en NODAL mais peut être combiné par le DO X est :

Distr.

CRT's
Participants au cours
Using QED to maintain large NODAL programs.

The normal editor QED can be used to maintain NODAL programs in the same way as programs written in other languages. It is sufficient that the :SYMB file containing the Nodal program looks like and is used as a MODE-file to type the program to Nodal.

To build the program X:NOD, edit a file NOD-X:SYMB with QED, respecting the format:

```plaintext
@CC NOD-X:SYMB
@CC
@CC <comments describing the function of X>
@CC
@CC
@NODAL
---
-------
--
-------
------
------
------
------
------
-----

SAVE X ALLP ALLV
QUIT

--- calls Nodal,

the Nodal statements constituting
the program, with their line numbers,
and also all the direct commands
> to create special variables or load
sections from other Nodal programs
if needed.

the immediate command to
save X:NOD

leaves Nodal.
```

If X:NOD does not exist, then create it with CREATE-FILE X:NOD @ as a command to Sintran-III. To build the file after editing or modification, run the file with:

```
MODE NOD-X,TERMINAL
```
The following are the advantages of this technique:

-- QED can be used with all its powerful facilities for modification. This saves editing time, especially in the case where blocks have to be renumbered, names have be modified throughout the program etc.

-- The :SYMB file cannot be accidentally destroyed by a SAVE command.

-- The :SYMB file can be sent to the line-printer with a COPY-FILE command, it can be backed up by Archiver, and above all, it is in a standard Sintran-III format.

-- The :SYMB file being a normal file of characters can now be pushed through comment-removing programs or even automatic abbreviation programs. This will speed up execution of the :NOD program without losing the much needed documentation. There is NO LIMIT on the number of comment lines contained in the :SYMB file, since they can appear as immediate commands and do not even need line numbers!

-- If the program is conversational, then the command RUN and test input can also be put in the :SYMB file, so that after modification, rebuilding the :NOD file also makes the program go through a test. There is then always an example of use and test.

If you wish to convert an existing program Y:NOD to this sort of file, then, in Nodal, do:

1) LOAD Y
2) SET ODEV=OPEN("W","NOD-Y:SYMB"); LIST
3) With QED, add the parts to make NOD-Y:SYMB a MODE-file.

For obvious reasons, the filekind NOD is used, instead of the normal standard MODE (for mode-files).
Utilisation de SINTRAN III et SINTRON II

SYNTRON (Console et Service)  SINTRAN III (TSS)

1) Créer un fichier
   >IM(LIB)DEFFI (<IND>NOM",8)  @ CREATE FILE
   NAME : NOM : NOD
   Nb Page : 4 (par exemple)

2) Sauver un programme
   >SAVE(LIB)<IND>NOM ALLP  >SAVE NOM

3) Perforer une bande
   >BLANK; SAVE F-P; BLANK  > idem

4) Faire un listing
   >RUN LIST sur le SERV  >SET ODEV = OPEN ("w","L-P"); LIST

5) Charger le programme
   >OLD(LIB)<IND>NOM  >OLD NOM

6) Lire une bande perforée.
   Mettre la bande dans le lecteur et
   >LOAD T-R  > idem

7) Effacer un fichier
   >IM(LIB) DELFI("<IND>NOM")  @ DELETE FILE

8) Créer des routines
   DEF-C NOM (paramètres)

9) Sauver des routines dans un fichier
   >SDEF (LIB)<IND>NOM  >SDEF NOM:LIB
10) Charger le fichier des routines
   >LDEF (LIB)<IND>NOM >LDEF NOM

Noté : s'il y a plusieurs fichiers à charger
   >LDEF (LIB)<IND>NOM 1 >(LIB)<IND>NOM2 etc.

11) Sauver/charger sur bande perforée
    SDEF F-P LDEF T-R

NB pour entrer dans le système SINTRAN :

ESC
ENTER OPERATION
PASSWORD :)
© NODAL

Si on sort par erreur du NODAL
© apparaît, taper CONT), et on se retrouve en NODAL!
THE DEVELOPMENT OF A PROGRAM

Phase Ø: IS THERE A PROBLEM?

In this phase we try to find out whether the problem has not already been solved elsewhere, whether it can be solved by changing the equipment or the method of operation, whether a computer program is really the best way of achieving our goals. We assume this phase has at least been recognized, it has been worked through, and its outcome was the necessity of a program.

Phase 1: STATE THE PROBLEM AND CONVINCE YOURSELF THAT A REASONABLE SOLUTION EXISTS

We are still not very far, but it is necessary to think up a good, common sense and simple problem statement such as:

"make a program to list all the prime numbers less than 1000"
"make a program to list all the prime numbers less than 1000000"
"make a program that plays chess as well as a grandmaster"
"make a program that wins at the Tercé".

It is equally important to be sure that at least one of the programmers knows all the mathematics, physics, engineering science or whatever is needed in order to solve the problem. The best programmer will not be able to write a program to solve second-order equations if he does not know or remember the algebra needed. Therefore, the solution invariably starts from the knowledge about the objects we are dealing with in the program. Phase 1 is completely disconnected from programming. Take your time to live through phase 1, draw diagrams, discuss, look up literature ... Invent names (good ones) for the things you are trying to think about.

Phase 1.9: Repeat Phase 1 because I am sure you skipped it.
Phase 2: Decide how the program will interact with its environment

This is the most important step in program development. Most programs in our environment are interactive, or have at least some important influence on some machinery or other programs. If you know from phase 1 how to optimise a certain beam parameter, you will now have to decide when to activate the program, who or what will control it, what to do with its results and at which instance. During phase 2, the whole method of interaction will be designed.

Notice that we do not discuss any part of the solution to the original problem, we only try to fit a nice box with the right handles and controls around the real solution.

Phase 3: Design the program and data structures in a language independent way

Here we first meet the constraints of computing: we will have to express the complete solution in a certain notation which is somewhat more restrictive than everyday English!

Use program structure diagrams, a program design language or any such tool that helps think clearly and precisely. Do spend a lot of time on getting the data structures right. If Phase 1 was OK, then there should be no major problem. Try to restrict yourself to the following:

- use different data structures for different things (do not try to save memory, especially not in this phase). If you are worried about data space, then either something went wrong in phase 1 or you are attempting something too big for your system.

- use only the procedure, the if-then-else or case, and the while. (for, repeat and loop are permitted variants). If, in phase 3, you find the need for a jump (goto) then do not try to "program around it": something else is wrong: you have not got the ideas right or you did not do phase 1 well enough. At this level, the appearance of a goto is an indication of faulty design or more
serious troubles. (Later, the goto may be necessary to realize the control structures in the programming language you use, but that is a different matter).

- take ten minutes to think of a good name for each object.

- make sure you have proper ways of initializing everything, and also proper ways of terminating without leaving things lying around.

Phase 4 : DIGESTION

Skipping this phase usually results in "frigging" : one thinks the program is ready and starts "testing" it. This may work for trivial ten-line programs, but it will not do any good for complicated real-life programs. The best method to avoid wasting time in debugging and frigging is now to let the program (as yet not typed in, let alone run!) mature in your desk, for a week or so, like good wine. You will then have forgotten the details, and a fresh look at it will reveal any serious bugs (especially in design of interactions). When you take it out of your desk, try to explain it to somebody else in its fullest detail. This is extremely useful, since most people refuse to understand someone else's work and will put very relevant questions. If after ten minutes of discussion, you no longer understand it yourself, then go back to phase 1 (if you decide to persist at all).

Phase 5 : TRANSLATION INTO A PROGRAMMING LANGUAGE (CODING)

You have made it : it is now a trivial matter to choose a language and hand-translate your program into that language. Preferably you choose a language that is efficient, i.e. for which little translation effort has to be done. Most languages in wide use today however do not even support if-then-else and therefore you will have to translate such structures into allowed if's and goto's. The goto's you will meet now will be the result of translation, and are virtually harmless.
Be aware that translation always causes pollution: some of the peculiarities of the programming language will infect your way of thinking and this will influence your way of designing a program in the future.

**Phase 6: Desk and Machine Checking**

After having typed the coded program, get a clean listing. Do try not to run it (that really is somewhat like eating the forbidden fruit, especially with large programs) but study it on your desk first. Check syntax, structure, spelling. If the program is really large, then you will have designed it in parts anyway, so you can look at each part individually. Parts and methods can be tested. When you have reasonable confidence, reserve 30 minutes on the machine and make sure someone has booked it right after you: you are then safe from wasting more than 30 minutes on your first tests.
A simple curve-plotting program  
(R. Cailliac - SPI Model course)  
9/5/77

We wish to plot on the line printer curves of the form \( y = f(x) \), whereby the function \( f(x) \) is given by the user. For the purpose of the exercise, we will keep the program simple.

Requirements:

- the user must be able to give his function.
- the user must be asked for the interval \([x_0, x_1]\), \(x_0 < x_1\), in which the function must be displayed.
- we will always take 50 points in \([x_0, x_1]\), but must compute \(y_{\text{min}}\) and \(y_{\text{max}}\) in \([x_0, x_1]\) to be able to scale in the \(y\)-direction and to choose a starting point for \(y\).
- the \(y\) axis will run across the paper over 100 positions (simplify).

![Graph](image)

**Fig. 1:** plotting window and associated plot.

The plotting window is shown in fig. 1. The problem-solving phase goes as follows:

We must approximate points on the plane to available printing positions. Always taking 50 positions in \([x_0, x_1]\) means we step through \(x\) from \(x_0\) by \(\Delta x\) to \(x_1\), and the increment \(\Delta x = \frac{x_1 - x_0}{49}\).
Since \( y_{\text{min}} \) and \( y_{\text{max}} \) are not given, we must determine them. We can do this by looking at \( y_i = f(x_i) \) for all \( x_i = x_0 + \Delta x i \); \( 0 < i < 99 \). It would be a waste of storage and time to first compute all the \( y_i \) and then find the smallest and the largest. Instead, we will compare every \( y_i \) with \( y_{\text{min}} \) and \( y_{\text{max}} \), and if \( y_i < y_{\text{min}} \) then we will take \( y_i \) as the new \( y_{\text{min}} \), if \( y_i > y_{\text{max}} \) we will take it as the new \( y_{\text{max}} \).

To start off on this, we must put decent values in \( y_{\text{min}} \) and \( y_{\text{max}} \). Convince yourself that \( y_{\text{min}} = y_{\text{max}} = y_0 \) is the right choice, and compare this with the way your minimum-maximum thermometer works at home if you have one. (If you do not have such a thermometer, inspect one next time you go to a supermarket.)

For a given \( y_i \), we must choose one of the 100 printing positions. Now \( \Delta y = \frac{y_{\text{max}} - y_{\text{min}}}{99} \) and the distance \( y - y_{\text{min}} \) contains \((y - y_{\text{min}})/\Delta y \) such intervals. This last number varies between 0 and 99. We will round it, and then add 1 to get an integer between 1 and 100, which we call \( y_p \).

The \( x \) axis is printed at \( y = \phi \) (position 1), how many blanks must we output between it and the position for \( y_p \)? If \( y_p = 1 \) we should not even try to print the position. If \( y_p = 2 \), then there should be no blanks, if \( y_p = 3 \) then one etc.: if \( y_p > 2 \), \( y_p - 2 \) blanks.
Plotting is done by computing all \( y_i \) again and taking a new line right before output of a position. The first \( y_i \) (i.e., \( f(x_0) \)) is on the \( y \)-axis and must not be output.
The user interface can be made very fancy, to provide fixing of scaling, "friendly" scaling, tabulation etc., but the only extra we will allow here is this: after having made a plot of a particular function, the user may want to change \([x_0, x_1]\), because he is looking for a certain feature (such as a zero or a minimum) which happens to lie outside the interval he just chose. Therefore we will repeat the plotting until he indicates that he’s finished examining the curve.

Program structure development.

Since the whole program is a big loop until-done-will-examining, the first approximation PSD is: (fig 2)

```
CURVE- PLOT

initialize and get f()

while the user examines the function

make a plot according to his desires.

leave
```

Fig. 2

We will leave the initialization block until later and concentrate on the making of a single plot. For each plot, the values of \(x_0\) and \(x_1\) must be got; they must be verified \((x_1 > x_0)\), the values \(y_{\text{min}}\) and \(y_{\text{max}}\) must be determined from them, then the curve has to be printed in the window \([x_0, x_1], [y_{\text{min}}, y_{\text{max}}]\), and finally we must find out whether that was enough or whether he (the user) wants another go. This results in fig. 3.
Fig. 3.

The first block of fig. 3 is simple: fig. 4.

The second block is somewhat more complex but is well-known from our pondering phase, and the last block is not even worth expanding. Figure 5 represents determination of $y_{min}$ and $y_{max}$, this leaves the plotting itself as the most "difficult" bit.
When plotting the function, it is interesting to the user to print some information such as \( f(x) \), \( x_0, x_1 \), \( y_{\text{min}}, y_{\text{max}} \) so that he can keep his plot apart. We will print the x-axis using the character ":", the y-axis using "-", and the curve using "*". Remember that \( y_p \) is the position in \( 1 \) to \( 100 \) of a point of the curve. Then:

a) the y-axis: \( x = x_0 \), if \( y_p(x_0) = 1 \) then print "*" and \( 99 \) times "-"; else if \( y_p(x_0) > 1 \) then print \( y(x) - 1 \) times "-", one "*", and \( (99 - y(x_0)) \) times a "-". Print "y".

b) for all other points, i.e., for \( x = x_0 + \Delta x \) up to \( x_4 \), do the following:

1) take a new line, determine \( y_p(x) \)
2) if \( y_p = 1 \) then print "*",
   else if \( y_p = 2 \) then print ":", print "*",
   else if \( y_p > 2 \) then print ":", print \( (y_p - 2) \) times a space, print "*".

PSD in Fig. 6.
Obtaining the function \( f(x) \) from the user is not difficult in Nidal, we read its definition into a string, and make a Nidal line out of it. (The practice of constructing program lines as you go, i.e., during execution of a program, is very dangerous, unreliable and bad programming. It should never be done, except in such well-controlled cases as this, where only a functional relation between (two) variables is determined but it does not change the program logic.)

If you have noticed that we forgot a very important test during the reasoning on pages 2 and 5, then read on. If you have not noticed, then dig up your elementary algebra books and reread those pages!

What have we done? On pages 2 and 5 we use \( y \). But \( y \) is determined as the number of intervals \( \Delta y \) in \( y_{max} \). This is computed by...
dividing $y_{\min}$ by $\Delta y$. Suppose $\Delta y = 0$? Then our whole program goes haywire! The meaning of $\Delta y = 0$ actually is:

$$y_{\min} = y_{\max} = f(x),$$

i.e. the function does not vary with $x$, it is a constant!

Since this can happen, even with very complex functions, we should not try to plot if $y_{\min} = y_{\max}$. (But there is also the danger that $\Delta y$ gets very small, and division will become inaccurate, or results very large. Normally you should test for $\Delta y < e$ where $e$ is a function of your computations and the precision of your machine.)

The PSD of fig. 3 is therefore wrong, and must be replaced by that of fig. 7.

---

![Flowchart](chart.png)

The structured development of the program guarantees modularity, and it is therefore easy to re-design the erroneous BDD(s), usually without having to modify many blocks.

Every program should be developed structurally, and the paths of thought followed during development (with the errors!) should be written down as an essential part of the program's documentation.
Try the following extensions or modifications: (use PSL's and write down all your thoughts)

1) if the x-axis \((y=0)\) is within the window, then plot it too, using "+". (difficult)

2) allow for each plot the optional output of a table of values \(x, f(x)\) in \([x_0, x_1]\). (easy)

3) what happens when \(x_1-x_0=\Delta x\) is so small that division by 49 causes underflow, and \(\Delta x\) effectively becomes zero? (easy)

4) suppose \(f(x)\) is also dependent on a parameter \(p\) : \(y = f(x, p)\). Make the program able to output a set of curves \(f(x)\) in \([x_0, x_1]\), for \(n\) different values of \(p\) in \([p_0, p_1]\). \((n, p_0, p_1\) input by the user.) (not very difficult; but useful only if you have a digital plotter so that the \(n\) curves can be drawn on the same piece of paper.)

5) Make a simultaneous plot of \(y_1 = f_1(x), y_2 = f_2(x), ... y_n = f_n(x)\) all on the same window, using "1", "2", ..., "n" to recognize the different curves. (difficult?)

6) provide "friendly" scaling in both \(x\) and \(y\). Example: try to adjust the window so that the axes cross at points with "round" coordinates (not 5.475 but 5.5, for example) and also so that the widths in \(x\) and \(y\) correspond to a pleasant multiple of divisions on the axes. (such as 2.5 or 10 divisions, but not 7.) (difficult: at least a full day of mathematics.)

(THINK FIRST, PROGRAM LATER!)
1.01 % CURVE- PLOT PROGRAM
1.02 % ===========
1.03 %
1.10 DO 2
1.20 WHILE EXAM. = 1; DO 3
1.30 TYPE '!! 'BYE!'
1.40 END

2.01 % INITIALIZE AND GET F()
2.02 % =========================
2.03 %
2.10 TYPE '!! 'CURVE- PLOT' ! FUNCTION
2.20 $ASK F; IF SIZE(F)<3; TYPE '!! TYPE F(X) PLEASE. " ; GOTO 2.2
2.30 $SET NODLIN(99.1)="SET " F
2.40 SET EXAM. = 1
2.50 SET WX=50; SET WY=100

3.01 % MAKE A PLOT ACCORDING TO DESIRES
3.02 % ==================================
3.03 %
3.10 DO 4; DO 5
3.20 IF YMAX.-YMIN. > 0; DO 6; GOTO 3.4
3.30 TYPE '!! CONSTANT FUNCTION! NO PLOT MADE." !!!
3.40 DO 7

4.01 % GET XO, XI.
4.02 % =================
4.03 %
4.10 $ASK 'XO' ' XO ' XI' ' XI; IF XO>=XI; TYPE '!! XI MUST BE > XO!!! ; GOTO 4.1

5.01 % DETERMINE YMIN+, YMAX
5.02 % ========================
5.03 %
5.10 SET DX=(XI-XO)/(WX-1)
5.20 SET X=XO; DO 99; SET YMIN.=Y; SET YMAX.=Y;
5.30 FOR X=XO+DX;DX;XI; DO 8

6.01 % PLOT THE FUNCTION
6.02 % ====================
6.03 %
6.10 TYPE '!! 'FUNCTION ' F '!!!
6.20 TYPE 'XO' ' XO ' XI' ' XI' ' XI' ' XI' ' YMIN' ' YMIN. ' YMAX' ' YMAX. '!!!
6.30 $SET DY=(YMAX.-YMIN.)/(WY-1)
6.40 SET X=XO; DO 99; DO 9
6.50 IF YP=1; DO 10; GOTO 6.7
6.60 DO 11
6.70 FOR X=XO+DX;DX;XI; DO 12

7.01 % ASK IF MORE
7.02 % ==================
7.03 %
7.10 TYPE '!! 'DO YOU WANT ANOTHER PLOT? ' %
7.20 $ASK ANSW.;
7.30 $IF (ANSW."' Y") 7.4, 7.9, 7.4
7.40 $IF (ANSW."' N") 7.5, 7.8, 7.5
7.50 TYPE '!! 'ANSWER WITH 'Y' OR 'N' PLEASE! ; GOTO 7.1
7.60 SET EXAM. = 0
7.70 RETURN

8.10 DO 99
8.20 IF Y<YMIN.; SET YMIN.=Y
8.30 IF Y>YMAX.; SET YMAX.=Y

9.01 % COMPUTE YP
9.10 SET YP=INT((Y-YMIN.)/DY + 1.5)

10.10 TYPE '***'; FOR I=1;WP-1; TYPE '*' 
11.10 FOR I=1;WP-1; TYPE '="
11.20 TYPE '***
11.30 FOR I=1;WY-YP; TYPE '="
12.10 TYPE '!! ; DO 99; DO 9
12.20 IF YP=1; TYPE '**'; RETURN
12.30 IF YP=2; TYPE '***'; RETURN
12.40 TYPE '***'; FOR I=1;WP-2; TYPE '**
12.50 TYPE '***
Exercises:
Try your hand at the following small problems:
(the problem definitions are sometimes vague, to allow you to
observe that problem statements must be refined, and that often some
restrictions must be introduced to the program's possibilities.)

1) List the prime numbers from 1 to 1000.

2) Read words and determine whether they are palindromes.
   (palindromes are words like CAMAC, RADAR, TOT, XYZYX, that can be
   read backwards the same as forwards.)

3) Play the role of the observer in the game of Master-Mind.

4) Make a histogram of a list of numbers.

5) Find the $x_i$ for which $F(x_i)=0$ for a given $F()$ in $x_0, x_1$

6) For a pair of numbers, find their greatest common divisor and their
   smallest common multiple.

7) Multiply correctly (to the last digit) two 8-digit integers.

8) Write a program to teach the four operations to a child. The
   child types two numbers and an operator (+,-,*,/), along with the
   result of the operation. The program only verifies the result.

9) Generate a list of pseudo-random numbers. Verify with the program
   of problem 4.

10) Monitor the fuel consumption of your car. Output the consumption
    from fill-up to fill-up, and also the mean consumption over the last
    n km (say n=2000km) (interpolations!)

11) Put eight chess queens on a chess board, so that no queen can take
    any of the others.
Exercise:

The problem of Josephus.

A tyrant with a penchant for mathematics decides to execute all his prisoners. He conceives the following scheme: the prisoners will stand in a circle, then every third one will be executed. This selection does not stop after one round, but goes on and on until only one prisoner is left. This last man will be saved and set free.

(example: for 7 prisoners the execution sequence will be: 3,6,2,7,5,1; and the fourth is left over.) Josephus, a prisoner who is a professional programmer, finds out about the tyrant's intentions. When the prisoners have to take their place in the circle, where should Josephus stand in order to be saved?

Given that the number of prisoners is n, and the place of the saved prisoner is m, then:

1) can you find a function m=F(n)? (closed form)
2) if not, why not?
3) how many ways could you devise to compute F(n)?
4) did you write down all the "pondering" necessary to arrive at a solution? Was it useful? How many dead ends did you meet before hitting a solution?
5) Do you think this is a difficult problem? Did you think so when you started, and in how much time did you estimate to solve it when you first started? If you made no estimate, why not?
6) Did any knowledge about mathematics or other experience help you? Which? Did you consider recursion?
Utilisation de SINTRAN III
et SINTRON II

SYNTRON (Console et Service) | SINTRAN III (TSS)

1) Créer un fichier
   @ CREATE FILE
   NAME : NOM : NOD
   NB Page : 4 (par exemple)
   ou
   >SAVE "NOM" la première fois
   >IM(LIB)DEFFI ("<IND>NOM",4)
   dans le LIBRARY
   >DEFF("<NOM."",1)
   dans un GP

2) Sauver un programme
   >SAVE(LIB)<IND>NOM ALLP
   >SAVE NOM

3) Perforer une bande
   >SAVE F.P
   >SAVE F-P

4) Faire un listing
   >RUN LIST sur le SERV
   >SET ODEV = OPEN ("W","L=P"); LIST
   >RUN(OP)LIST

5) Charger le programme
   >OLD(LIB)<IND>NOM
   >OLD NOM

6) Lire une bande perforée.
   Mettre la bande dans le lecteur et
   >LOAD T.R
   >LOAD T-R

7) Effacer un fichier
   @DELETE-USER-FILE
   >IM(LIB)DELFI("<IND>NOM")

8) Créer des routines
   DEF-C NOM (paramètres)
   idem

9) Sauver des routines dans
   un fichier
   >SDEF(LIB)<IND>NOM
   >SDEF NOM:LIB
10) Charger le fichier des routines
   \( \text{LDEF} \ (\text{LIB})<\text{IND}>\text{NOM} \quad \text{LDEF NOM} \)

__Nota__ : s'il y a plusieurs fichiers à charger

\( \text{LDEF} \ (\text{LIB})<\text{IND}>\text{NOM} \ 1 \quad <(\text{LIB})<\text{IND}>\text{NOM2}, \text{etc.} \)

11) Sauver/charger sur bande perforée
   \( \text{SDEF} \ F.P \quad \text{SDEF} \ F-P \)
   \( \text{LDEF} \ T.R \quad \text{LDEF} \ T-R \)

__Nota__ : pour entrer dans le système SINTRAN :

ESC
ENTER OPERATION
PASSWORD :
a NODAL

Si on sort par erreur du NODAL

a apparaît, taper CONT, et on se retrouve en NODAL !

\( a \, \text{LI-FI} \) donne la liste des fichiers
\( a \, \text{US-STAT} \) vous donne l'espace qui reste