Pseudo-premiers

(F. ARNAULT, 6 Octobre 1993)

Ici on implémente en `Maple` des outils permettant de tester des propriétés de pseudo-primalité, c'est-à-dire des propriétés qui sont à la base des tests de primalité probabilistes. Ce package comporte six parties. la première traite des pseudo-premiers classiques (Fermat, Miller), la deuxième et la troisième concernent respectivement le calcul des suites de Lucas et les pseudo-premiers de Lucas. La quatrième traite de la suite de Perrin (cf [1]) et la dernière, des pseudo-premiers elliptiques (cf [3]).

1. **Pseudo-primalité classique** 1
2. Contrôle des paramètres 1
5. Pseudo-primalité simple 2
6. Pseudo-primalité forte 2
8. Pseudo-primalité forte dans plusieurs bases 2
10. **Suites de Lucas** 4
11. Contrôle des paramètres 4
13. Calcul du nombre d'étapes 4
14. Calcul des suites de Lucas 5
16. Calcul modulaire des suites de Lucas 5
18. **Pseudo-primalité de Lucas** 6
19. Contrôle des paramètres 6
20. Initialisation des constantes locales 6
21. Pseudo-primalité simple de Lucas 7
22. Pseudo-primalité d'Euler-Lucas 7
23. Pseudo-primalité forte de Lucas 7
26. La méthode A 8
27. La méthode B 9
28. Des tests plus sûrs 9
29. **La suite de Perrin** 10
31. Contrôle des paramètres 10
32. Initialisation 11
33. Calcul de la signature 11
37. Signatures admissibles 12
40. **Pseudo-premiers elliptiques** 14
41. Choix de la courbe et du point initial 14
42. Sécurité 14
43. Arithmétique sur une courbe elliptique 15

44. Opposé d'un point 15
45. Addition 15
47. Multiplication 16
49. Calcul du nombre d'étapes 17
50. Pseudo-premiers 17
54. **Vue générale** 19
55. Bibliographie 19

1. Pseudo-primalité classique

On définit d'abord deux fonctions testant respectivement la pseudo-primalité et la pseudo-primalité forte d'un entier n dans une base b. On définit ensuite une fonction testant la pseudo-primalité forte dans un ensemble de bases.

⟨ pseudo-primalité classique **1** ⟩ ≡

```
psp := proc(b,n)    # checks if n is a pseudoprime in base b
    ⟪ Corps de la fonction psp 5 ⟫
end :

spsp := proc(b,n)   # checks if n is a strong pseudoprime in base b
    ⟪ Corps de la fonction spsp 6 ⟫
end :

mspsp := proc(lb,n)
          # checks if n is a strong pseudoprime in the bases in set lb
    ⟪ Corps de la fonction mspsp 8 ⟫
end :
```

This code is used in section 54

2. *Contrôle des paramètres* — Pour les deux premières fonctions, les paramètres b et n doivent être de type entier. Dans le cas contraire, on stoppe tout. De plus, on impose $n >= 1$ (On admet généralement que 1 est pseudo-premier dans toute base).

⟨ Contrôle des paramètres de psp **2** ⟩ ≡

```
if not type(n, integer) or not type(b, integer)
              then ERROR(‘arguments must be integers‘) ;
fi ;
if n < 1 then ERROR(‘second argument must be > 0‘) fi ;
```

This code is used in section 5

3. Pour la pseudo-primalité forte, on a des contraintes supplémentaires sur l'entier n.

⟨ Contrôle des paramètres de spsp. **3** ⟩ ≡

```
if not type(n, integer) or not type(b, integer)
              then ERROR(‘arguments must be integers‘) ;
fi ;
if type(n, even) or n < 2
    then ERROR(‘second argument must be odd and > 2‘)
fi ;
```

This code is used in section 6

4. Pour la fonction qui teste la pseudo-primalité forte dans un ensemble de bases, on doit faire les contrôles suivants :

⟨ Contrôle des paramètres de mspsp **4** ⟩ ≡

```
if not type(n, integer) or not type(lb, set(integer))
     then ERROR(‘arguments must be a list(integer) and an integer‘) ;
fi ;
if type(n, even) or n < 2
              then ERROR(‘second argument must be odd and > 2‘) ;
fi ;
```

This code is used in section 8

5. *Pseudo-primalité simple* — Rappelons qu'un entier $n \in \mathbb{N}^*$ est pseudo-premier de base $b \in \mathbb{N}^*$ si l'on a la relation

$$b^{n-1} \equiv 1 \text{ modulo } n.$$

L'opérateur `mods` est un opérateur de calcul modulaire. Il utilise la représentation centrée, c'est-à-dire dans l'intervalle $]-n/2, n/2]$. L'opérateur `Maple` '`&^`' est l'opérateur "inerte" d'exponentiation, c'est-à-dire qu'il n'est évalué que dans le contexte de l'opérateur `mods`, par un algorithme entièrement modulaire. La fonction `evalb` est nécessaire pour retourner un résultat booléen car son paramètre est de type *expression*.

⟨ Corps de la fonction `psp` **5** ⟩ ≡

```
⟪ Contrôle des paramètres de psp 2 ⟫
if n = 1 then RETURN(true) fi ;
evalb(mods(b &^ (n-1), n) = 1) ;
```

This code is used in section 1

6. *Pseudo-primalité forte* — On rappelle qu'un entier $n > 2$ est fortement pseudo-premier de base $b \in \mathbb{N}^*$ si l'on a

$$b^q = 1 \qquad \text{ou} \qquad \text{s'il existe } i \text{ vérifiant } 0 \leq i < k \text{ tel que } b^{2^i q} \equiv -1 \text{ modulo } n.$$

⟨ Corps de la fonction `spsp` **6** ⟩ ≡

```
local k,q,power ;
⟪ Contrôle des paramètres de spsp 3 ⟫
⟪ Calcul de k et q 7 ⟫
power := mods(b &^ q, n) ;
if power = 1 then RETURN(true) fi ;
do
   if power = -1 then RETURN(true) fi ;
   if k = 1 then break fi ;
   k := k - 1 ;
   power := mods(power * power, n) ;
od ;
false ;
```

This code is used in section 1

7. Ici, on calcule les entiers k et q tels que $n - 1 = 2^k q$, avec q impair.

⟨ Calcul de k et q **7** ⟩ ≡

```
q := n - 1 ;
k := 0 ;
while type(q,even) do
    q := q/2 ;
    k := k + 1 ;
od ;
```

This code is used in sections 6, 8

8. *Pseudo-primalité forte dans plusieurs bases* — Une fois seulement, on calcule les entiers k et q, puis on teste la pseudo-primalité de n dans chacune des bases de la liste. Si tous les tests sont positifs, la fonction renvoie `true`.

⟨ Corps de la fonction `mspsp` **8** ⟩ ≡

```
local b,height,k,q,power ;
⟪ Contrôle des paramètres de mspsp 4 ⟫
⟪ Calcul de k et q 7 ⟫
for b in lb do
   ⟪ Test pour la pseudo-primalité forte de base b 9 ⟫
od ;
true ;
```

This code is used in section 1

9. On est ici obligé d'utiliser une variable `height` supplémentaire par rapport à `spsp`.

⟨ Test pour la pseudo-primalité forte de base b **9** ⟩ ≡

```
power := mods(b &^ q, n) ;
if power = 1 then next fi ;
height := 0 ;
do
   if power = -1 then break fi ;
   if height = k - 1 then RETURN(false) fi ;
   height := height + 1 ;
   power := mods(power * power, n) ;
od ;
```

This code is used in section 8

10. Suites de Lucas

Deux fonctions pour calculer les termes U_n et V_n des suites de Lucas de paramètres P et Q. La première les calcule entièrement, la deuxième les calcule modulo un entier p.

⟨ Suites de Lucas **10** ⟩ ≡

```
Lucas := proc(P,Q,n)
            # Shows nth term of Lucas sequences with parameters P et Q
   ⟪ Corps de la fonction Lucas 14 ⟫
end :

modLucas := proc(P,Q,n,p)
            # Shows nth term of Lucas sequences with parameters P et Q mod p
   ⟪ Corps de la fonction modLucas 16 ⟫
end :
```

This code is used in section 54

11. *Contrôle des paramètres* — ⟨ Contrôle des paramètres de Lucas **11** ⟩ ≡

```
if not type(n, integer) then ERROR(‘arguments must be integers‘) fi ;
if n < 0 then ERROR(‘third argument must be >= 0‘) fi ;
```

This code is used in section 14

12. On impose au modulus p d'être impair car l'algorithme comporte des divisions par 2 modulaires.

⟨ Contrôle des paramètres de modLucas **12** ⟩ ≡

```
if not type(n, integer) or not type(P, integer)
    or not type(Q, integer) or not type(p, integer)
    then ERROR(‘arguments must be integers‘)
fi ;
if n < 0 then ERROR(‘third argument must be >= 0‘) fi ;
if type(p, even) or p < 2
    then ERROR(‘fourth argument must be odd and > 2‘)
fi ;
```

This code is used in section 16

13. *Calcul du nombre d'étapes* — On calcule ici le nombre d'étapes v nécessaires pour calculer les termes de rang n des suites de Lucas. C'est le nombre de chiffres du développement binaire de n moins 1, c'est-à-dire le plus grand entier tel que $2^v \leq n$.

⟨ Calcul du nombre v d'étapes **13** ⟩ ≡

```
v := 0 ;  twopowerv := 1 ;
do
   twopowerv := 2*twopowerv ;
   if twopowerv > n then break fi ;
   v := v + 1 ;
od ;      # v is the greatest integer such that 2^v <= n
```

This code is used in sections 14, 16, 33

14. *Calcul des suites de Lucas* — On calcule en même temps les deux termes U_n et V_n par une méthode dichotomique. Le calcul d'un seul terme serait sensiblement du même coût.

⟨ Corps de la fonction `Lucas` **14** ⟩ ≡

```
local v, twopowerv, D, k, Qk, Uk, Vk, Utmp ;
⟪ Contrôle des paramètres de Lucas 11 ⟫
if n = 0 then RETURN([0, 2]) fi ;
⟪ Calcul du nombre v d'étapes 13 ⟫
D := P^2 - 4*Q ;
k := 1 ;   Uk := 1 ;   Vk := P ;      Qk := Q ;
while v >= 1 do
    ⟪ Passage de l'indice k à l'indice 2k ou 2k + 1 15 ⟫
od ;
[Uk, Vk]
```

This code is used in section 10

15. Connaissant les termes U_k et V_k, on calcule les termes U_{2k} et V_{2k} puis éventuellement U_{2k+1} et V_{2k+1} grâce aux formules

$$\begin{cases} U_{2k} &= U_kV_k \\ V_{2k} &= V_k^2 - 2Q^k \end{cases} \qquad \begin{cases} 2U_{k+1} &= PU_k + V_k \\ 2V_{k+1} &= DU_k + PV_k \end{cases} \qquad \text{pour tout } k \in \mathbb{N}.$$

La condition du test est vraie si le chiffre de poids 2^v dans le developpement binaire de n est 1, fausse sinon.

⟨ Passage de l'indice k à l'indice $2k$ ou $2k + 1$ **15** ⟩ ≡

```
Uk := Uk*Vk ;   Vk := Vk^2 - 2*Qk ;      Qk := Qk*Qk ;
k := 2*k ;  v := v - 1 ;
if 2^v * (k+1) <= n then
    Utmp := (P*Uk + Vk)/2 ;  Vk := (D*Uk + P*Vk)/2 ;  Uk := Utmp ;
    Qk := Qk*Q ;   k := (k + 1) ;
fi ;
```

This code is used in section 14

16. *Calcul modulaire des suites de Lucas* — L'algorithme employé est le même que pour le calcul exact. Le calcul modulaire se fait en un temps $O(\log n)$.

⟨ Corps de la fonction `modLucas` **16** ⟩ ≡

```
local v, twopowerv, D, k, Qk, Uk, Vk, Utmp ;
⟪ Contrôle des paramètres de modLucas 12 ⟫
if n = 0 then RETURN([0, 2]) fi ;
⟪ Calcul du nombre v d'étapes 13 ⟫
D := mods(P^2 - 4*Q, p) ;
k := 1 ;   Uk := 1 ;   Vk := mods(P, p) ;    Qk := mods(Q, p) ;
while v > 0 do
    ⟪ Passage modulaire de l'indice k à l'indice 2k ou 2k + 1 17 ⟫
od ;
[Uk, Vk]
```

This code is used in section 10

17. ⟨ Passage modulaire de l'indice k à l'indice $2k$ ou $2k + 1$ **17** ⟩ ≡

```
Uk := mods(Uk*Vk, p) ;   Vk := mods(Vk&^2 - 2*Qk, p) ;
Qk := mods(Qk*Qk, p) ;     k := 2*k ;       v := v - 1 ;
if 2^v * (k+1) <= n then
    Utmp := mods((P*Uk + Vk)/2, p) ;  Vk := mods((D*Uk + P*Vk)/2, p) ;
    Uk := Utmp ;    Qk := mods(Qk*Q, p) ;      k := (k + 1) ;
fi ;
```

This code is used in section 16

18. Pseudo-primalité de Lucas

Cinq fonctions sont implémentées ici. Les trois premières testent respectivement la pseudo-primalité simple de Lucas, la pseudo-primalité d'Euler-Lucas, et la pseudo-primalité forte de Lucas pour des paramètres P et Q donnés. Les deux dernières testent la pseudo-primalité de Lucas pour des paramètres déterminés automatiquement par deux méthodes préconisées par J.L. Selfridge et R. Baillie.

⟨ Pseudo-primalité de Lucas **18** ⟩ ≡

```
lpsp := proc(P,Q,n)
   # checks if n is a Lucas pseudoprime with parameters P et Q
   ⟪ Corps de la fonction lpsp 21 ⟫
end :

elpsp := proc(P,Q,n)
   # checks if n is a Euler-Lucas pseudoprime with parameters P et Q
   ⟪ Corps de la fonction elpsp 22 ⟫
end :

slpsp := proc(P,Q,n)
   # checks if n is a strong Lucas pseudoprime with parameters P et Q
   ⟪ Corps de la fonction slpsp 23 ⟫
end :

lpspA := proc(n)
   # checks if n is a Lucas pseudoprime, with parameters chosen by method A
   ⟪ Corps de la fonction lpspA 26 ⟫
end :

lpspB := proc(n)
   # checks if n is a Lucas pseudoprime, with parameters chosen by method B
   ⟪ Corps de la fonction lpspB 27 ⟫
end :
```

This code is used in section 54

19. *Contrôle des paramètres* — Les contraintes sur les paramètres sont identiques pour les trois premières fonctions.

⟨ Contrôle des paramètres P, Q, n **19** ⟩ ≡

```
if not type(n, integer) or not type(P, integer) or not type(Q, integer)
             then ERROR(‘arguments must be integers‘) ;
fi ;
if n < 2 or type(n, even)
     then ERROR(‘third argument must be odd and > 1‘) ;
fi ;
if gcd(n,Q) <> 1 then ERROR(‘n et Q must be relatively prime‘) fi ;
```

This code is used in sections 21, 22, 23

20. *Initialisation des constantes locales* — On choisit d'exclure le cas où D et n ne sont pas premiers entre eux.

⟨ Initialisation des constantes locales **20** ⟩ ≡

```
D := P^2 - 4*Q ;
eps := jacobi(D,n) ;
if eps = 0 then ERROR(‘we must have (D/n) <> 0‘) fi ;
```

This code is used in sections 21, 22, 23

21. *Pseudo-primalité simple de Lucas* — On rappelle que n est pseudo-premier de Lucas de paramètres P et Q s'il est impair, premier avec Q et vérifie $m \mid U_{m-\varepsilon(m)}$ où l'on désigne par $\varepsilon(m)$ le symbole de Jacobi $\left(\frac{D}{m}\right)$.

⟨ Corps de la fonction `lpsp` **21** ⟩ ≡

```
local D,eps ;
⟪ Contrôle des paramètres P, Q, n 19 ⟫
⟪ Initialisation des constantes locales 20 ⟫
evalb(modLucas(P,Q,n-eps,n)[1] = 0) ;
```

This code is used in section 18

22. *Pseudo-primalité d'Euler-Lucas* — On rappelle que n est pseudo-premier d'Euler-Lucas de paramètres P et Q s'il est premier avec $2QD$ et si

$$n \mid U_{(n-\varepsilon(n))/2} \quad \text{si } \left(\frac{Q}{n}\right) = 1, \qquad n \mid V_{(n-\varepsilon(n))/2} \quad \text{si } \left(\frac{Q}{n}\right) = -1$$

où $\varepsilon(n)$ désigne le symbole de Jacobi $\left(\frac{D}{n}\right)$.

⟨ Corps de la fonction `elpsp` **22** ⟩ ≡

```
local D,eps ;
⟪ Contrôle des paramètres P, Q, n 19 ⟫
⟪ Initialisation des constantes locales 20 ⟫
if jacobi(Q,n) = 1 then evalb(modLucas(P,Q,(n-eps)/2,n)[1] = 0) ;
else evalb(modLucas(P,Q,(n-eps)/2,n)[2] = 0) ;
fi ;
```

This code is used in section 18

23. *Pseudo-primalité forte de Lucas* — Enfin, on rappelle que n est fortement pseudo-premier de Lucas de paramètres P et Q s'il est premier avec $2QD$ et vérifie

$$n \mid U_q \qquad \text{ou} \qquad \text{il existe un entier } i \text{ tel que } 0 \le i < t \text{ et } n \mid V_{2^i q}$$

où l'on a posé $\varepsilon(n) = \left(\frac{D}{n}\right) = 2^t q$ avec q impair.

⟨ Corps de la fonction `slpsp` **23** ⟩ ≡

```
local D,eps,q,t,k,Qk,Uk,Vk,UkVk ;
⟪ Contrôle des paramètres P, Q, n 19 ⟫
⟪ Initialisation des constantes locales 20 ⟫
⟪ Calcul de t et q 24 ⟫
UkVk := modLucas(P,Q,q,n) ;
k := q ;    Qk := mods(Q &^ q, n) ;
Uk := UkVk[1] ;     Vk := UkVk[2] ;
if Uk = 0 then RETURN(true) fi ;
do
   if Vk = 0 then RETURN(true) fi ;
   if t = 1 then break fi ;
   t := t - 1 ;
   Uk := mods(Uk*Vk, n) ;   Vk := mods(Vk &^ 2 - 2*Qk, n) ;
   k := 2*k ;   Qk := mods(Qk*Qk, n) ;
od ;
false ;
```

This code is used in section 18

24. La fonction `slpsp` nécessite le calcul des entiers t et q tels que $n - \varepsilon = 2^t q$, avec q impair. Ce fragment s'en occupe.

⟨ Calcul de t et q **24** ⟩ ≡

```
q := n - eps ;
t := 0 ;
while type(q, even) do
   q := q/2 ;
   t := t + 1 ;
od ;
```

This code is used in section 23

25. Voici maintenant les deux dernières fonctions. Utilisées conjointement avec un test de pseudo-primalité classique, elles constituent un test probabiliste de primalité redoutable. Leur unique paramètre doit satisfaire les contraintes suivantes :

⟨ Contrôle du paramètre n **25** ⟩ ≡

```
if type(n, even) or n < 3 then
      ERROR(‘the argument must be odd and >= 3‘)
fi ;
if type(n, square) then RETURN(false) fi ;
```

This code is used in sections 26, 27

26. *La méthode A* — C'est celle préconisée par J.L. Selfridge. Dans cette méthode, on choisit D comme étant le plus petit élément de la liste $5, -7, 9, -11, 13, \ldots$ tel que

$$\varepsilon(n) = \left(\frac{D}{n}\right) = -1$$

puis on prend $P = 1$ et $Q = (1 - D)/4$. De plus, si un diviseur de n est trouvé au cours de la recherche de D, la fonction retourne `false`.

⟨ Corps de la fonction `lpspA` **26** ⟩ ≡

```
local D, Q, eps ;
⟪ Contrôle du paramètre n 25 ⟫
D := 5 ;
do
   eps := jacobi(D,n) ;
   if eps = 0 then RETURN(evalb(abs(D) = n)) fi ;
   if eps = -1 then break fi ;
   if D > 0 then D := - (D + 2) else D := - (D - 2) fi ;
od ;
Q := (1 - D)/4 ;
if gcd(Q,n) > 1 then RETURN(false) fi ;
lpsp(1,Q,n) ;
```

This code is used in section 18

27. *La méthode B* — C'est celle préconisée par R. Baillie. Dans cette méthode, on choisit D comme étant le plus petit élément de la liste $5, 9, 13, 17, \ldots$ tel que

$$\varepsilon(n) = \left(\frac{D}{n}\right) = -1$$

puis on prend pour P le plus petit entier impair tel que $P^2 \geq D$ et on pose $Q = (P^2 - D)/4$. De plus, si un diviseur de n est trouvé au cours de la recherche de D, la fonction retourne `false`.

⟨ Corps de la fonction `lpspB` **27** ⟩ ≡

```
local D, P, Q, eps ;
⟪ Contrôle du paramètre n 25 ⟫
D := 5 ;
do
   eps := jacobi(D,n) ;
   if eps = 0 then RETURN(evalb(D = n)) fi ;
   if eps = -1 then break fi ;
   D := D + 4 ;
od ;
P := isqrt(D) ;
if P^2 < D then P := P + 1 fi ;
if type(P, even) then P := P + 1 fi ;
Q := (P^2 - D)/4 ;
if gcd(Q,n) > 1 then RETURN(false) fi ;
lpsp(P,Q,n) ;
```

This code is used in section 18

28. *Des tests plus sûrs* — Les auteurs de [4] offrent une récompense à celui qui trouvera un nombre composé passant l'un des deux tests implémentés ici (ou bien qui prouverait que l'un de ces tests est infaillible).

⟨ Tests de primalité mixtes **28** ⟩ ≡

```
isprimeA := proc(n) ;
   if n < 2 then RETURN(false) fi ;
   if type(n, even) then RETURN(evalb(n = 2)) fi ;
   if not spsp(2, n) then RETURN(false) fi ;
   lpspA(n) ;
end :

isprimeB := proc(n) ;
   if n < 2 then RETURN(false) fi ;
   if type(n, even) then RETURN(evalb(n = 2)) fi ;
   if not spsp(2, n) then RETURN(false) fi ;
   lpspB(n) ;
end :
```

This code is used in section 54

29. La suite de Perrin

La suite de Perrin est la suite récurrente d'entiers définie par :

$$A_0 = 3, \quad A_1 = 0, \quad A_2 = 2, \qquad A_{n+3} = A_{n+1} + A_n \quad \text{pour tout } n \in \mathbb{N}.$$

Plus généralement, les auteurs de [1] considèrent les suites définies par :

$$A_0 = 3, \quad A_1 = r, \quad A_2 = r^2 - 2s, \qquad A_{n+3} = rA_{n+2} - sA_{n+1} + A_n \quad \text{pour tout } n \in \mathbb{N}. \tag{1}$$

où r et s sont des entiers fixés. Ces suites vérifient la propriété suivante

$$p \mid A_p \qquad \text{pour tout nombre premier } p. \tag{2}$$

ainsi que bien d'autres qui sont exposées dans [1].

30. Dans [1], on remarque que la formule (1) permet de prolonger la suite de Perrin de manière à l'indexer sur $\mathbb{Z}$. On appelle signature de n modulo p le sextuplet

$$(A_{-n-1}, A_{-n}, A_{-n+1}, A_{n-1}, A_n, A_{n+1}) \text{ modulo } p.$$

On trouve aussi dans [1] une description de l'algorithme ci-dessous permettant de calculer cette signature en un temps $O(\log n)$. Le calcul du terme A_n seul serait sensiblement du même coût.

⟨ La suite de Perrin **30** ⟩ ≡

```
signPerrin := proc(r, s, n)
       # Shows nth signature (terms of rank -n-1, -n, -n+1, n-1, n, n+1)
       # of Perrin sequence modulo an integer p.
       # If p is omitted then the default is n
   ⟪ Corps de la fonction signPerrin 33 ⟫
end :

PerrinSpsp := proc(r, s, n)   # Checks whether n has a S-signature
   ⟪ Corps de la fonction PerrinSpsp 37 ⟫
end :

PerrinQpsp := proc(r, s, n)   # Checks whether n has a Q-signature
   ⟪ Corps de la fonction PerrinQpsp 39 ⟫
end :

PerrinIpsp := proc(r, s, n)   # Checks whether n has a I-signature
   ⟪ Corps de la fonction PerrinIpsp 38 ⟫
end :
```

This code is used in section 54

31. *Contrôle des paramètres* — La fonction `signPerrin` calcule la signature de n modulo p ou, si le deuxième paramètre est absent, la signature de n modulo n.

⟨ Contrôle des paramètres de `signPerrin` **31** ⟩ ≡

```
if nargs < 3 or nargs > 4 then ERROR(‘3 or 4 arguments expected‘) fi ;
p := args[nargs] ;
if not type(n, integer) or not type(p, integer) or not type(r, integer)
     or not type(s, integer) then ERROR(‘arguments must be integers‘) ;
fi ;
if n <= 0 or p <= 0 then ERROR(‘third and fourth arguments must be > 0‘) fi ;
if gcd(r*s - 1, p) > 1 then ERROR(‘rs - 1 must be prime to p‘) fi ;
```

This code is used in section 33

32. *Initialisation* — Le calcul de la signature de n fait intervenir une suite croissante $(k_i)_{0\le i\le v}$ d'entiers telle que $k_0 = 1$ et $k_v = n$ et telle que l'on sache calculer la signature de k_i en fonction de celle de k_{i-1}. On commence donc par initialiser la variable k par k_0 et la signature $(S_1, S_2, S_3, S_4, S_5, S_6)$ par la signature de 1 qui est $(s^2 - 2r, s, 3, 3, r, r^2 - 2s)$.

⟨ Initialisation par la signature de 1 **32** ⟩ ≡

```
k := 1 ;
S1 := mods(s^2 - 2*r, p) ;   S2 := mods(s, p) ;   S3 := mods(3, p) ;
S4 := mods(3, p) ;           S5 := mods(r, p) ;   S6 := mods(r^2 - 2*s, p) ;
```

This code is used in section 33

33. *Calcul de la signature* — L'algorithme est très semblable à celui qui est utilisé pour le calcul des suites de Lucas.

⟨ Corps de la fonction `signPerrin` **33** ⟩ ≡

```
local p, v, twopowerv, k, S1, S2, S3, S4, S5, S6, Saux,
                          T1, T2, T3, T4, T5, T6, rsm1inv, r2ms, s2mr ;
⟪ Contrôle des paramètres de signPerrin 31 ⟫
⟪ Calcul du nombre v d'étapes 13 ⟫
⟪ Initialisation par la signature de 1 32 ⟫
r2ms := r^2 - s ;   s2mr := s^2 - r ;    rsm1inv := mods(1/(r*s - 1), p) ;
while v >= 1 do
   ⟪ Passage de l'indice k à l'indice 2k 34 ⟫
   if 2^v * (k+1) <= n then
        ⟪ Passage de l'indice 2k à l'indice 2k + 1 36 ⟫
   fi ;
od ;
[S1, S2, S3, S4, S5, S6] ;
```

This code is used in section 30

34. Connaissant la signature de k, on peut calculer les entiers A_{-2k-2}, A_{-2k}, A_{-2k+2}, A_{2k-2}, A_{2k} et A_{2k+2} en utilisant la formule

$$A_{2l} = A_l^2 - 2A_{-l}.$$

⟨ Passage de l'indice k à l'indice $2k$ **34** ⟩ ≡

```
T1 := mods(S1&^2 - 2*S6, p) ;   # term of rank -2k-2
T2 := mods(S2&^2 - 2*S5, p) ;   # term of rand -2k
T3 := mods(S3&^2 - 2*S4, p) ;   # term of rank -2k+2
T4 := mods(S4&^2 - 2*S3, p) ;   # term of rank  2k-2
T5 := mods(S5&^2 - 2*S2, p) ;   # term of rank  2k
T6 := mods(S6&^2 - 2*S1, p) ;   # term of rank  2k+2
```

This code is defined in sections 34, 35

This code is used in section 33

35. Pour obtenir la signature de $2k$, il ne reste alors plus qu'à calculer A_{-2k-1}, A_{-2k+1}, A_{2k-1} et A_{2k+1} par les formules suivantes, obtenues à partir de la relation de récurrence (1) :

$$(rs-1)A_{2k-1} = (r^2-s)A_{2k} + rA_{2k-2} - A_{2k+2}, \quad (rs-1)A_{-2k-1} = (r^2-s)A_{-2k} + rA_{-2k-2} - A_{-2k+2},$$
$$(rs-1)A_{2k+1} = (s^2-r)A_{2k} + sA_{2k+2} - A_{2k-2}, \quad (rs-1)A_{-2k+1} = (s^2-r)A_{-2k} + sA_{-2k+2} - A_{-2k-2}.$$

⟨ Passage de l'indice k à l'indice $2k$ **34** ⟩ +≡

```
S1 := mods((r2ms*T2 + r*T1 - T3)*rsm1inv, p) ;
S2 := T2 ;
S3 := mods((s2mr*T2 + s*T3 - T1)*rsm1inv, p) ;
S4 := mods((r2ms*T5 + r*T4 - T6)*rsm1inv, p) ;
S5 := T5 ;
S6 := mods((s2mr*T5 + s*T6 - T4)*rsm1inv, p) ;
k := 2*k ;   v := v - 1 ;
```

This code is defined in sections 34, 35

This code is used in section 33

36. On change la signature de $2k$ par la signature de $2k+1$ par l'affectation

$$(S_1, S_2, S_3, S_4, S_5, S_6) \quad \longleftarrow \quad (sS_1 - rS_2 + S_3, S_1, S_2, S_5, S_6, S_4 - sS_5 + rS_6).$$

⟨ Passage de l'indice $2k$ à l'indice $2k+1$ **36** ⟩ ≡

```
Saux := mods(S4 - s*S5 + r*S6, p) ;              # future value for S6
S4 := S5 ;   S5 := S6 ;   S6 := Saux ;
Saux := mods(s*S1 - r*S2 + S3, p) ;              # future value for S1
S3 := S2 ;   S2 := S1 ;   S1 := Saux ;
k := k + 1 ;
```

This code is used in section 33

37. *Signatures admissibles* — Une S-signature est une signature de la forme $(s^2-2r, s, 3, 3, r, r^2-2s)$.

⟨ Corps de la fonction `PerrinSpsp` **37** ⟩ ≡

```
local Ssignature ;
Ssignature := [mods(s^2 - 2*r, n), mods(s, n), mods(3, n),
               mods(3, n), mods(r, n), mods(r^2 - 2*s, n)] ;
evalb(signPerrin(r, s, n) = Ssignature) ;
```

This code is used in section 30

38. Une I-signature est une signature de la forme (r, s, D', D, r, s) où D et D' sont des racines modulo n distinctes du polynôme

$$X^2 - (rs - 3)X + r^3 + s^3 - 6rs + 9.$$

⟨ Corps de la fonction `PerrinIpsp` **38** ⟩ ≡

```
local D ;
signature := signPerrin(r, s, n) ;
if signature[1] <> mods(r, n) then RETURN(false) fi ;
if signature[2] <> mods(s, n) then RETURN(false) fi ;
if signature[5] <> mods(r, n) then RETURN(false) fi ;
if signature[6] <> mods(s, n) then RETURN(false) fi ;
D := signature[3] ;
if mods(D^2 - (r*s - 3)*D + r^3 + s^3 - 6*r*s + 9, n) <> 0
then RETURN(false) ;
fi ;
D := signature[4] ;
if mods(D^2 - (r*s - 3)*D + r^3 + s^3 - 6*r*s + 9, n) <> 0
then RETURN(false) ;
fi ;
evalb(mods(signature[3] - signature[4], n) <> 0) ;
```

This code is used in section 30

39. Une Q-signature est une signature de la forme (A, s, B, B, r, C) où l'on a les relations

$$(s^3 - r^3)A \equiv (s^2 - 3r)B^2 + s(4r^2 - s^2r - 3s)B + (s^5 - 3s^3r + 6sr^2 - 2r^4) \text{ modulo } n$$
$$B^3 - rsB^2 + (r^3 + s^3 - 3rs)B + (r^3 + s^3 - r^2s^2) \equiv 0 \text{ modulo } n$$
$$(r^3 - s^3)C \equiv (r^2 - 3s)B^2 + r(4s^2 - r^2s - 3r)B + (r^5 - 3r^3s + 6rs^2 - 2s^4) \text{ modulo } n.$$

⟨ Corps de la fonction `PerrinQpsp` **39** ⟩ ≡

```
local signature, A, B, C ;
signature := signPerrin(r, s, n) ;
if signature[3] <> signature[4] then RETURN(false) fi ;
if signature[5] <> mods(r, n) then RETURN(false) fi ;
if signature[2] <> mods(s, n) then RETURN(false) fi ;
B := signature[3] ;
if mods(B^3 - r*s*B^2 + (r^3 + s^3 - 3*r*s)*B + (r^3 + s^3 - r^2*s^2), n) <> 0
then RETURN(false) ;
fi ;
A := signature[1] ;
if mods((s^3 - r^3)*A, n) <> mods((s^2 - 3*r)*B^2 + s*(4*r^2 - s^2*r - 3*s)*B
                            + (s^5 - 3*s^3*r + 6*s*r^2 - 2*r^4), n)
then RETURN(false) ;
fi ;
C := signature[6] ;
if mods((r^3 - s^3)*C, n) <> mods((r^2 - 3*s)*B^2 + r*(4*s^2 - r^2*s - 3*r)*B
                            + (r^5 - 3*r^3*s + 6*r*s^2 - 2*s^4), n)
then RETURN(false) ;
fi ;
true ;
```

This code is used in section 30

40. Pseudo-premiers elliptiques

Enfin, la dernière partie concerne les pseudo-premiers elliptiques. L'équation de la courbe E utilisée ainsi que le point initial sont précisés par des variables globales `a`, `b` et `P0`. Un point non nul sur E est représenté par une liste `[x,y]` à deux éléments et le zéro de E est représenté par la liste vide `[]`.

⟨ Pseudo-primalité elliptique **40** ⟩ ≡

```
⟪ La fonction setcurve 41 ⟫
⟪ La fonction isoncurve 42 ⟫
⟪ Arithmétique 43 ⟫
⟪ Pseudo-premiers 50 ⟫
```

This code is used in section 54

41. *Choix de la courbe et du point initial* — Cette fonction permet d'affecter ces variables en puisant dans le catalogue suivant :

n°1	$Y^2 = X^3 - 5X,$	$P_0 = (5, 10)$
n°2	$Y^2 = X^3 - 120X - 448,$	$P_0 = (64, 504)$
n°3	$Y^2 = X^3 - 3500X - 98000,$	$P_0 = (84, 448)$
n°4	$Y^2 = X^3 - 2,$	$P_0 = (3, 5)$

⟨ La fonction `setcurve` **41** ⟩ ≡

```
setcurve := proc(c) ;
   if   c = 1 then a :=    -5;  b :=      0;  D := -1;  P0 :=   [5,10];
   elif c = 2 then a :=  -120;  b :=   -448;  D := -2;  P0 := [64,504];
   elif c = 3 then a := -3500;  b := -98000;  D := -7;  P0 := [84,448];
   elif c = 4 then a :=     0;  b :=     -2;  D := -3;  P0 :=    [3,5];
   else ERROR(‘Curve number is out of range‘)
   fi ;
end :

setcurve(1) ;
```

This code is used in section 40

42. *Sécurité* — Cette fonction vérifie que son premier paramètre est bien sur la courbe E modulo n.

⟨ La fonction `isoncurve` **42** ⟩ ≡

```
isoncurve := proc(P, n) local left, right ;
   if P=[] then RETURN(true) fi ;
   left  := mods(P[2]&^2 - b, n) ;
   right := mods((P[1]^2 + a) * P[1], n) ;
   evalb(left = right) ;
end :
```

This code is used in section 40

43. *Arithmétique sur une courbe elliptique* — L'addition est rendue délicate par le fait que la courbe utilisée est définie modulo un entier n qui n'est pas supposé premier. La stratégie employée est celle de Lenstra et consiste à suivre la procédure que l'on applique habituellement modulo les nombres premiers et à renoncer à calculer le résultat dès qu'une ambiguïté apparaît au cours de l'algorithme. En effet, une telle ambiguïté est provoquée par l'apparition d'un facteur non trivial de n et donc d'une preuve de la non-primalité de n.

⟨ Arithmétique **43** ⟩ ≡

```
⟪ La fonction ellminus 44 ⟫

⟪ La fonction elladd 46 ⟫

⟪ La fonction ellmul1 47 ⟫

⟪ La fonction ellmul 48 ⟫
```

This code is used in section 40

44. *Opposé d'un point* — L'opposé d'un point $[x, y]$ est $[x, -y]$.

⟨ La fonction `ellminus` **44** ⟩ ≡

```
ellminus := proc(P, n) ;
   if P=[] then RETURN(P) fi ;
   [P[1], mods(-P[2], n)] ;
end :
```

This code is used in section 43

45. *Addition* — Soient P et Q deux points non nuls à additionner. Nous devons calculer la pente m de la droite passant par P et Q (ou la tangente si $P = Q$). Elle est donnée par les formules

$$m = \frac{y_2 - y_1}{x_2 - x_1} \quad \text{si } P_1 \neq P_2, \qquad m = \frac{3x_1^2 + a}{2y_1} \quad \text{si } P_1 = P_2$$

quand elles ont un sens modulo n. Mais si le dénominateur n'est pas inversible modulo n, la fonction d'addition retourne l'élément neutre si $P_1 = P_2$ et retourne la valeur `FAIL` dans le cas contraire (on a alors un facteur non trivial de n). Nous traitons ici ces cas particuliers.

⟨ Calcul de m **45** ⟩ ≡

```
m := mods(Q[1] - P[1], n) ;          # first, only the denominator of m
sumy := mods(Q[2] + P[2], n) ;
if m = 0 then                        # we have (y1 + y2)(y1 - y2) = 0 mod n
    if sumy = 0 then RETURN([]) fi ;             # P = - Q mod n
    if igcd(sumy, n) > 1 then RETURN(FAIL) fi ;  # P = - Q for some p|n
    m := mods((3*P[1]^2 + a)/(2*P[2]), n) ;      # here, P = Q mod n
else
    if igcd(m, n)>1 then RETURN(FAIL) fi ;  # 2-division point mod some p|n
    m := mods((Q[2] - P[2])/m, n) ;         # P =/= +-Q mod all p|n
fi ;
```

This code is used in section 46

46. Si l'un de ces points à additionner est nul, l'addition est triviale. Sinon, $P = (x_1, y_1)$ et $Q = (x_2, y_2)$. Si le calcul du coefficient m "réussit", alors la somme (x_3, y_3) est donnée par

$$x_3 = m^2 - x_1 - x_2, \qquad y_3 = m(x_1 - x_3) - y_1.$$

⟨ La fonction `elladd` **46** ⟩ ≡

```
elladd := proc(P, Q, n) local m, sumy, x3, y3 ;
   if not isoncurve(P, n) or not isoncurve(Q, n) then
       ERROR(‘Arguments are not on curve‘) ;
   fi ;
   if P = [] then RETURN(Q) fi ;
   if Q = [] then RETURN(P) fi ;
   ⟪ Calcul de m 45 ⟫
   x3 := mods(m^2 - P[1] - Q[1], n) ;
   y3 := mods(m*(P[1] - x3) - P[2], n) ;
   [x3, y3] ;
end :
```

This code is used in section 43

47. *Multiplication* — Premier algorithme : exponentiation dichotomique classique (de droite à gauche).

⟨ La fonction `ellmul1` **47** ⟩ ≡

```
ellmul1 := proc(k, P, n) local Q, PP, kk ;
   if not type(k, integer) then ERROR(‘First argument must be integer‘) fi ;
   if not isoncurve(P, n) then ERROR(‘Second argument must be on curve‘) fi ;
   if k < 0 then RETURN(ellminus(ellmul1(-k, P))) fi ;
   Q := [] ;  PP := P ;  kk := k ;
   do
      if type(kk, odd) then
          Q := elladd(Q, PP, n) ;
          if Q = FAIL then RETURN(Q) fi ;
      fi ;
      if kk < 2 then break fi ;
      PP := elladd(PP, PP, n) ;
      if PP = FAIL then RETURN(PP) fi ;
      kk := iquo(kk, 2) ;
   od ;
   Q ;
end :
```

This code is used in section 43

48. Deuxième algorithme : schéma de Hörner (de gauche à droite). C'est celui utilisé par Gordon [3]. Le résultat peut être différent (rarement) de celui du premier algorithme, puisque la chaîne d'additions utilisée n'est pas la même.

⟨ La fonction `ellmul` **48** ⟩ ≡

```
ellmul := proc(k, P, n) local Q, v, twopowerv, kk ;
   if not type(k, integer) then ERROR(‘First arg must be integer‘) fi ;
   if not isoncurve(P, n) then ERROR(‘Second arg must be on curve‘) fi ;
   if k < 0 then RETURN(ellminus(ellmul(-k, P))) fi ;
   if k = 0 then RETURN([]) fi ;
   ⟪ Calcul du nombre v d'étapes pour la multiplication 49 ⟫
   Q := P ;  kk := 1 ;
   while v > 0 do
      Q := elladd(Q, Q, n) ;
      if Q = FAIL then RETURN(Q) fi ;
      kk := 2*kk ;   v := v - 1 ;
      if 2^v * (kk + 1) <= k then
           Q := elladd(Q, P, n) ;
           if Q = FAIL then RETURN(Q) fi ;
           kk := kk + 1 ;
      fi ;
   od ;
   Q ;
end :
```

This code is used in section 43

49. *Calcul du nombre d'étapes* — On calcule ici le nombre d'étapes v nécessaires dans l'algorithme de multiplication. C'est le nombre de chiffres du développement binaire de k moins 1, c'est-à-dire le plus grand entier tel que $2^v \leq k$.

⟨ Calcul du nombre v d'étapes pour la multiplication **49** ⟩ ≡

```
v := 0 ;  twopowerv := 1 ;
do
   twopowerv := 2*twopowerv ;
   if twopowerv > k then break fi ;
   v := v + 1 ;
od ;      # v is the greatest integer such that 2^v <= k
```

This code is used in section 48

50. *Pseudo-premiers* — Deux fonctions, pour tester la pseudo-primalité ou la pseudo-primalité forte d'un entier n sur la courbe sélectionnée.

⟨ Pseudo-premiers **50** ⟩ ≡

```
⟪ La fonction ellpsp 51 ⟫

⟪ La fonction ellspsp 52 ⟫
```

This code is used in section 40

51. Un entier n est pseudo-premier elliptique pour la courbe et le point initial selectionnés si l'on a $\left(\frac{D}{n}\right) = -1$ et la relation

$$(n+1)P_0 = 0.$$

⟨ La fonction `ellpsp` **51** ⟩ ≡

```
ellpsp := proc(n) ;
   if not type(n, posint) then ERROR(‘argument must be integer > 0‘) fi ;
   if jacobi(D, n) <> -1 then ERROR(‘p must be inert‘) fi ;
   evalb(ellmul(n + 1, P0, n) = []) ;
end :
```

This code is used in section 50

52. Un entier n est pseudo-premier elliptique fort pour la courbe et le point initial selectionnés si l'on a $\left(\frac{D}{n}\right) = -1$ et l'une des conditions

$$qP_0 = 0 \qquad \text{ou} \qquad 2^i qP_0 \text{ est d'ordonnée nulle} \quad \text{pour un } i \text{ tel que } 0 \le i < k$$

où l'on a posé $n + 1 = 2^k q$.

⟨ La fonction `ellspsp` **52** ⟩ ≡

```
ellspsp := proc(n) local k, q, P ;
   if not type(n, integer) or type(n, even) or n<3 then
       ERROR(‘argument must be an odd integer > 1‘) ;
   fi ;
   if jacobi(D, n) <> -1 then ERROR(‘p must be inert‘) fi ;
   ⟪ Calcul de k et q pour ellspsp 53 ⟫
   P := ellmul(q, P0, n) ;
   if P = [] then RETURN(true) fi ;
   if P = FAIL then RETURN(false) fi ;
   do
      if P[2] = 0 then RETURN(true) fi ;
      if k = 1 then break fi ;
      k := k - 1 ;
      P := ellmul(2, P, n) ;
      if (P = []) or (P = FAIL) then RETURN(false) fi ;
   od ;
   false ;
end :
```

This code is used in section 50

53. La fonction `ellspsp` nécessite le calcul des entiers k et q tels que $n+1 = 2^k q$, avec q impair. Ce fragment s'en occupe.

⟨ Calcul de k et q pour `ellspsp` **53** ⟩ ≡

```
q := n + 1 ;
k := 0 ;
while type(q, even) do
   q := q/2 ;
   k := k + 1 ;
od ;
```

This code is used in section 52

54. Vue générale

Enfin, voici l'architecture globale du package.

⟨ **File "psprimes.mpl"** ⟩
- ⟪ Pseudo-primalité classique **1** ⟫
- ⟪ Suites de Lucas **10** ⟫
- ⟪ Pseudo-primalité de Lucas **18** ⟫
- ⟪ Tests de primalité mixtes **28** ⟫
- ⟪ La suite de Perrin **30** ⟫
- ⟪ Pseudo-primalité elliptique **40** ⟫

55. *Bibliographie* —

[1] W.W. Adams, D. Shanks : *Strong primality tests that are not sufficient.* Mathematics of Computation, vol. 39, n° 159, juil. 1982, pp. 225–300.

[2] R. Baillie, S. Wagstaff Jr : *Lucas pseudoprimes.* Mathematics of Computation, vol. 35, n° 152, oct. 1980, pp. 1391–1417.

[3] D.M. Gordon : *On the number of elliptic pseudoprimes.* Mathematics of Computation. vol. 52, n° 185, janvier 1989, pp. 231–245.

[4] C. Pomerance, J.L. Selfridge, S.S. Wagstaff : *The pseudoprimes to* $25 \cdot 10^9$. Mathematics of Computation, vol. 35, n° 151, 1980, pp. 1003–1026.

[5] M.O. Rabin : *Probabilistic algorithms for testing primality.* Journal of Number Theory, vol. 12, 1980, pp. 128–138.