# LE SYSTÈME PARI/GP

GUILLAUME HANROT

Table des matières

1. Introduction 2
1.1. Présentation générale 2
1.2. Ce que Pari/GP sait bien faire 3
1.3. Ce que Pari/GP sait faire 3
1.4. Ce que Pari/GP ne sait pas faire 3
1.5. Pari/GP n'est pas un système de calcul formel 3
2. `gp` - ligne de commande 4
2.1. Premiers pas 4
2.2. Accès à l'aide 4
2.3. Les types GP 5
3. Fonctions sur les corps de nombres 15
4. Fonctions diverses 15
5. Scripts GP 16
5.1. Variables 16
5.2. Fonctions 16
5.3. Boucles `for` 17
5.4. Tests 17
5.5. Boucles `while` et `until` 17
5.6. `break`, `next` 18
5.7. Variables locales 18
5.8. Écriture de scripts 18
6. La pratique 19
7. Le mode bibliothèque 19
7.1. Fichiers include 19
7.2. Le type `GEN` 19
7.3. Philosophie générale 19
7.4. Gestion de la mémoire 20
8. `install` et `gp2c` 22
Références 22

Historique de ce document :
`version 1.1 : 20/06/2004 (G. Hanrot, document initial)`
`version 1.2 : 07/09/2004 (K. Belabas, mise à jour pari-2-2-8)`
`version 1.3 : 10/10/2005 (B. Allombert/K. Belabas, mise à jour pari-2-2-11)`

*Date*: 10 octobre 2005.

version 1.4 : 08/01/2010 (K. Belabas, mise à jour pari-2-4-3)

## 1. Introduction

L'objectif de ce texte est de présenter le système Pari/GP [?]. C'est un survol ne prétendant pas se substituer au volumineux manuel du système, ni au tutoriel inclus dans ce manuel. Pour toutes informations d'ordre général, téléchargement, documentation, on renvoie le lecteur à la page du logiciel,

`http://pari.math.u-bordeaux.fr/`

et aux Ressources Pari/GP, notamment une Foire aux Questions et une archive de scripts utiles,

`http://www.math.u-bordeaux.fr/~belabas/pari/`

Pari/GP a été développé à partir de 1986 par un petit groupe animé par Henri Cohen à l'université Bordeaux 1 (C. Batut, C. Bernardi, H. Cohen, M. Olivier), à l'origine comme outil d'expérience pour la recherche sur les algorithmes de la théorie des nombres. Il a été repris en 1995 par Karim Belabas (Paris XI, puis Bordeaux), qui le maintient depuis. Le développement est devenu plus ouvert : licence GPL, versions régulières, accès CVS aux versions de développement. Deux mailing-lists existent depuis novembre 1997 (`pari-users` et `pari-dev`), et sont archivées sur les sites sus-mentionnés.

La version stable courante est la version 2.3 (patchlevel 2 à la date où ce texte est écrit). La version 2.4(.2 au jour de l'écriture de ce texte) est une version de développement publique, accessible via CVS anonyme, voir `http://pari.math.u-bordeaux/CVS.html`.

Pari/GP nécessite environ 3 Mo d'espace disque[1] et 4 Mo de mémoire vive pour fonctionner correctement. Il est écrit en C, donc très portable[2], supplémenté d'un noyau assembleur pour l'arithmétique sur les architectures courantes, donc sans sacrifier l'efficacité. Pari/GP supporte l'utilisation de GMP [4] pour l'arithmétique entière à la place du noyau natif[3]. Des versions binaires sont disponibles pour les systèmes les plus courants (Linux, Windows).

### 1.1. Présentation générale.

Le système Pari/GP, comme son nom l'indique, comporte deux parties.

- La première, Pari, est une bibliothèque de fonctions C orientée vers la théorie des nombres ;
- La seconde, `gp`, est un interpréteur qui permet d'accéder depuis une ligne de commande de syntaxe simple à l'ensemble des fonctionnalités de Pari. Cet interpréteur est programmable, dans le langage de script GP.

[1] 15 Mo pour la version Windows

[2] Les développeurs n'ont connaissance d'aucun système moderne, disons de moins de 20 ans d'âge, sur lequel `gp` ne tourne pas.

[3] Le noyau GMP est légèrement plus lent à faible précision, légèrement plus rapide à partir de 100 chiffres décimaux, deux fois plus rapide à partir de 300 chiffres... cinq fois plus rapide vers 2000000 chiffres.

L'intérêt est d'offrir, d'une part, une calculette intuitive pour des non-experts en programmation, qui permet toutefois d'accéder à la quasi-totalité des fonctionnalités de PARI et dispose d'un langage de script permettant de réaliser des calculs moins élémentaires. Et d'autre part de préserver la possibilité de programmer en C une longue suite de calculs de bas niveau pour lesquels l'efficacité est primordiale.

Dans cette optique, signalons l'outil `gp2c`, écrit par Bill Allombert, qui traduit les scripts GP en code C compilable avec la bibliothèque PARI. L'interface `gp2c-run` réalise même ceci de façon transparente : elle traduit les scripts, les compile et exécute une session `gp` disposant au démarrage des nouvelles fonctions. Cela permet souvent un gain important en vitesse lors de l'exécution par rapport à l'interprétation du code GP, sans necéssiter d'intervention sur un code C.

**Remarque importante :** La syntaxe ainsi que les noms de fonction de GP ont énormément changé entre les versions 1.xx (avant 1996) et les versions 2.x.x. Ce document traite exclusivement des versions 2.x.x.

1.2. **Ce que** PARI/GP **sait bien faire.** Calculs élémentaires sur les entiers en précision arbitraire [factorisation des entiers (jusqu'à 80 chiffres environ)] ; fonctions transcendantes ; calculs sur les polynômes univariés (factorisation, recherche de racines complexes ou $p$-adiques) et les séries formelles (calculs élémentaires) ; calculs sur les corps de nombres (sa principale force) ; calculs sur les courbes elliptiques ; algèbre linéaire sur $\mathbb{Z}$, $\mathbb{Q}$, ou un corps fini ; réduction des réseaux et principales applications.

1.3. **Ce que** PARI/GP **sait faire.** Analyse numérique (calcul intégral, sommation de séries, algèbre linéaire) ; graphisme. Toutes ces choses sont présentes soit par souci de complétude, soit parce qu'elles ont un jour été nécessaires à une application particulière, mais ne constituent pas un objectif majeur de développement, et ne sont à ce titre généralement pas optimisées.

1.4. **Ce que** PARI/GP **ne sait pas faire.** Corps finis non premiers ; manipulation de polynômes multivariés ; manipulations d'objets gigantesques : milliards de chiffres, dimension ou degré $10^6$ (toutes choses possibles dans le principe, mais vite inutilisables)... et bien d'autres choses encore.

1.5. PARI/GP **n'est pas un système de calcul formel.** Il ne sait ni représenter, ni manipuler des expressions autres que des fractions rationnelles (ou séries formelles), ou des expressions algébriques sur ces domaines. En particulier, toute fonction transcendante est automatiquement remplacée par son développement limité au voisinage de 0.

```
gp > sin(x)
%1 = x - 1/6*x^3 + 1/120*x^5 - 1/5040*x^7 + (...)
```

En arguant en outre du fait que le traitement des polynômes multivariés ou des structures creuses dans PARI est rudimentaire, PARI n'est pas un système de

calcul formel à part entière, même s'il offre un certain nombre de fonctionnalités formelles.

**Remarque :** Dans ce document, les `(...)` sont utilisés pour couper les exemples qui dépassent la largeur de la page, mais `gp` présente les résultats intégralement. À moins qu'on ne lui demande de les couper à $n$ lignes, via `default(lines, n)`.

## 2. `gp` - LIGNE DE COMMANDE

Cette partie a pour objectif de présenter la calculette `gp`, d'en expliquer rapidement le fonctionnement et les fonctionnalités.

### 2.1. Premiers pas.

```
gp > 1 + 1
%1 = 2
gp > (1/2)^2
%2 = 1/4
gp > Pi
%3 = 3.1415926535897932384626433383
gp > cos(Pi)
%4 = -1.0000000000000000000000000000
gp > quit
Goodbye!
```

Jusqu'ici, pas de surprises.

2.2. **Accès à l'aide.** Premier objectif : accéder à la documentation. Taper `?` puis `Enter` affiche la liste des rubriques d'aide et le mode d'emploi de l'aide, tandis que `??` affiche l'intégralité du manuel. Trois niveaux de documentation existent sous `gp` :

- `?fun` donne une documentation minimale sur la fonction `fun`.
- `??fun` affiche la partie du manuel correspondant à la fonction `fun`, dans une fenêtre `xdvi` ou dans le terminal.
- `???"mot-clé"` indique les fonctions ayant un rapport avec le `mot-clé` (qui peut d'ailleurs être une expression arbitraire, contenant éventuellement des espaces ; les guillemets sont optionnels si elle n'en contient pas).

  Par défaut, la recherche est restreinte au Chapitre 3 du manuel, celui qui décrit les fonctions GP. Si un `@` est présent à la fin du mot-clé, au contraire, la recherche est étendue à l'intégralité du manuel.

```
gp > ???GRH
bnfcertify    bnfinit       bnfisintnorm  bnfisnorm     quadclassunit
rnfisnorm     thue          thueinit

See also:
  Functions related to general number fields

gp > ???"elliptic curve"@
```

```
Chapter 1:
==========
  Arithmetic functions

Chapter 2:
==========
?

See also:
  Member functions
  Specific Use of the \kbd{gp} Calculator

Chapter 3:
==========
elladd          ellak           ellan           ellap           ellbil
ellchangecurve  elleisnum       ellglobalred    ellheight       ellinit
ellisoncurve    elllocalred     ellminimalmodel ellorder        ellpointtoz
ellpow          ellrootno       ellsub          elltaniyama     elltors
ellwp

See also:
  Functions related to elliptic curves
  Functions related to general number fields
```

2.3. **Les types** GP**.** GP n'est pas typé au sens informatique du terme ; en particulier, les variables n'ont pas de type. Cependant, chaque objet GP a un type interne spécifique, qui peut être visualisé en utilisant la fonction `type()`. Par exemple, on obtient

```
gp > type(3)
%1 = "t_INT"
```

Chaque résultat renvoyé par gp se voit attribuer un numéro (`%1` ici). Un résultat précédent peut être rappelé par son numéro précédé d'un `%` ; par exemple, ici

```
gp > type(%1)
%2 = "t_STR"
```

Le dernier résultat peut être rappelé par `%`, l'avant-dernier par `%‘`, l'antépénultième par `%‘‘`, etc. Ci-dessus, `type(%)` aurait donné la même réponse.

Pour ce qui est des types proprement dits, la commande `\t` permet d'en obtenir une liste :

```
List of the PARI types:
  t_INT    : long integers     [ cod1 ] [ cod2 ] [ man_1 ] ... [ man_k ]
  t_REAL   : long real numbers [ cod1 ] [ cod2 ] [ man_1 ] ... [ man_k ]
  t_INTMOD : integermods       [ code ] [ mod  ] [ integer ]
  t_FRAC   : irred. rationals  [ code ] [ num. ] [ den. ]
  t_FFELT  : finite field elt. [ code ] [ cod2 ] [ elt ] [ mod ] [ p ]
  t_COMPLEX: complex numbers   [ code ] [ real ] [ imag ]
  t_PADIC  : p-adic numbers    [ cod1 ] [ cod2 ] [ p ] [ p^r ] [ int ]
  t_QUAD   : quadratic numbers [ cod1 ] [ mod  ] [ real ] [ imag ]
  t_POLMOD : poly mod          [ code ] [ mod  ] [ polynomial ]
```

```
  ------------------------------------------------------------
  t_POL    : polynomials       [ cod1 ] [ cod2 ] [ man_1 ] ... [ man_k ]
  t_SER    : power series      [ cod1 ] [ cod2 ] [ man_1 ] ... [ man_k ]
  t_RFRAC  : irred. rat. func. [ code ] [ num. ] [ den. ]
  t_QFR    : real qfb          [ code ] [ a ] [ b ] [ c ] [ del ]
  t_QFI    : imaginary qfb     [ code ] [ a ] [ b ] [ c ]
  t_VEC    : row vector        [ code ] [  x_1  ] ... [  x_k  ]
  t_COL    : column vector     [ code ] [  x_1  ] ... [  x_k  ]
  t_MAT    : matrix            [ code ] [ col_1 ] ... [ col_k ]
  t_LIST   : list              [ code ] [ n ] [ nmax ][ vec ]
  t_STR    : string            [ code ] [ man_1 ] ... [ man_k ]
  t_VECSMALL: vec. small ints  [ code ] [ x_1 ] ... [ x_k ]
  t_CLOSURE: functions [ code ] [ arity ] [ code ] [ operand ] [ data ] [ text ]
```

La fin de la ligne décrit la structure C des éléments du type donné : par exemple, pour le type `t_INT`, les deux premiers mots sont des mots de code, et les mots suivants représentent la mantisse. Pour le type `t_FRAC`, le premier mot est un mot de code, le second mot contient l'adresse du numérateur, et le troisième l'adresse du dénominateur.

En GP, cette structure C n'est pas pertinente : on accède aux composantes d'un objet naturellement pour les matrices (`M[1,2]`) ou les vecteurs (`v[1]`), par `polcoeff` pour les polynômes, les séries, ou les formes quadratiques binaires, par `imag` ou `real` pour les nombres complexes ou quadratiques, par `denominator` ou `numerator` pour les fractions et fonctions rationnelles, et finalement `lift(x)` ou `x.mod` pour le représentant canonique et le module associés à un type modulaire. Il n'y a pas de façon simple d'accéder aux mots de la mantisse d'un réel ou d'un entier (une première approche utiliserait `\x` ou `bitand`).

Examinons maintenant rapidement les différents types GP et les fonctions qui permettent de les manipuler.

2.3.1. `t_INT`. Ce type est utilisé pour représenter les entiers en précision arbitraire.

```
gp > 450!
%1 = 173336873311263265934471314610 45(...)
gp > 250!
%2 = 32328562609091077323208145520243(...)
gp > 200!
%3 = 78865786736479050355236321393218(...)
gp > %3/(%4*%5)
%4 = 67985441606155742983110298166171(...)
gp > binomial(450, 200)
%5 = 67985441606155742983110298166171(...)
gp > divrem(%3, %4*%5)
%6 = [67985441606155742983110298166171833 9(...), 0]~
gp > factorint(2^128+1)
%7 =
[59649589127497217 1]
```

```
[5704689200685129054721 1]
```

Signalons deux fonctionnalités utiles dans ce dernier cas : la commande `##` permet de connaître le temps CPU utilisé par la dernière commande :

```
gp > ##
  ***   last result computed in 310 ms.
```

De façon analogue, la commande `#` déclenche un chronomètre, de sorte que le résultat de chaque calcul s'affiche avec le temps qu'il a pris. Un nouveau `#` interrompt le chronomètre.

Enfin, la commande `\g`$n$, où $n$ est un entier (usuellement entre 0 et 8) déclenche des messages de diagnostic, qui peuvent permettre de suivre le déroulement d'un calcul un peu long. Dans la factorisation ci-dessus, `g3`, `g4`, `g5` `g6`, `g8`, permettent d'accéder à des niveaux différents. Dans le cas de la factorisation, on peut voir la suite des algorithmes utilisés (naïf, $\rho$, SQUFOF, ECM, MPQS). Les options de *factorint* permettent d'influer sur ces différentes méthodes :

```
gp > ?factorint
factorint(x,{flag=0}): factor the integer x. flag is optional, whose
binary digits mean 1: avoid MPQS, 2: avoid first-stage ECM (may fall
back on it later), 4: avoid Pollard-Brent Rho and Shanks SQUFOF,
8: skip final ECM (huge composites will be declared prime)
```

L'expression entre accolades dans le prototype de la fonction indique que cet argument est optionnel : la valeur transmise si l'argument `flag` est omis est 0.

À l'instar de `factorint`, un grand nombre de fonctions GP ont des arguments facultatifs, qui permettent de préciser l'algorithme utilisé, ou des paramètres utilisés par cet algorithme. Par exemple, la fonction `factor` :

```
gp > ?factor
factor(x,{lim}): factorization of x. lim is optional and can be set
whenever x is of (possibly recursive) rational type.  If lim is set
return partial factorization, using primes up to lim (up to
primelimit if lim=0)
```

Ainsi, `factor(x)` tente de factoriser complètement `x`, mais `factor(x, 10^5)` se contente de rechercher les facteurs premiers inférieurs à $10^5$.

Quelques autres fonctions GP manipulant les entiers sont : `bezout, core, coredisc, divisors, eulerphi, gcd, isprime, lcm, sigma, sqrtint`. Noter que la fonction `isprime`, contrairement à la majorité des systèmes de calcul formel, *prouve* la primalité du nombre (méthodes $N-1$ et APRCL) plutôt que d'utiliser une batterie de tests probabilistes (comme le fait `ispseudoprime`).

**Exercice :** Consulter l'aide de la fonction `sigma`. Calculer la somme des carrés des diviseurs de $2^{128}+1$. Est-il fait appel à la factorisation pour effectuer ce calcul ? Combien de temps faut-il ?

2.3.2. *Les types* `t_REAL, t_COMPLEX`. Le type `t_REAL` est utilisé pour représenter des nombres réels en précision arbitraire. Les fonctions transcendantes habituelles

peuvent être utilisées pour manipuler ces nombres. Faire `?3` pour obtenir une liste. Les constantes `Euler, Pi` sont prédéfinies.

```
gp > cos(Pi)
%1 = -1.0000000000000000000000000000
gp > gamma(1/2)^2
%2 = 3.1415926535897932384626433832
```

Il convient de dire un mot de la façon dont `gp` gère la précision. La précision courante s'obtient par `\p` :

```
gp > \p
   realprecision = 28 significant digits
```

Cette précision affecte l'affichage des résultats et la conversion de données exactes en flottants (dont les composantes de bas niveau sont de type `t_REAL`). Par contre, un calcul utilisant des données flottantes est effectué *à la précision des données*, indépendamment de la précision courante, même si seuls les chiffres correspondant à la précision courante sont affichés. De même, si un résultat a moins de chiffres significatifs que la précision courante, seuls ceux-ci sont affichés. Finalement, appliquer une fonction dont le résultat est flottant à des données exactes nécessite une conversion préliminaire et le calcul s'effectue donc à la précision courante (y compris les constantes `Pi, Euler`). Voyons comment ces règles s'appliquent sur quelques exemples :

```
gp > x = Pi/2 - exp(-10)
%1 = 1.5707509268651341343797861000
gp > \p 9
   realprecision = 9 significant digits
gp > y = Pi/2 - exp(-10)
%2 = 1.57075093
gp > x2 = tan(x)
%3 = 22026.4658
gp > y2 = tan(y)
%4 = 22026.6029
gp > \p 28
   realprecision = 28 significant digits
gp > x2
%5 = 22026.4657796734065940501528 7
gp > y2
%6 = 22026.60286
gp > tan(x)
%7 = 22026.4657796734065940501528 7
gp > cos(y2)
%8 = -0.624136994
gp > print(precision(x), "  ", precision(x2), "  ",\
           precision(y), "  ", precision(y2))
28  28  9  9
gp > y2 = precision(y2, 28)
%9 = 22026.6028747558593750000000 0
```

```
gp > cos(y2)
%10 = -0.624140633883919562593591230 8
gp > (1e20 + Pi) - 1e20
%11 = 3.141592653
```

Noter que la précision de `x2` est celle de `x`, et non la précision courante.

Le type `t_COMPLEX` est de même nature, les précisions des parties réelle et imaginaire sont gérées indépendamment.

2.3.3. *Le type* `t_PADIC`. C'est le type utilisé pour représenter les nombres $p$-adiques. On les entre par $x + O(p\hat{}n)$, où $p\hat{}n$ représente la précision $p$-adique, et $x$ un entier ou un rationnel.

On peut faire de l'arithmétique avec les $p$-adiques : quatre opérations, racine $n$-ème et plus généralement factorisation de polynômes dans $\mathbb{Q}_p[X]$ ; la plupart des fonctions transcendantes sont également disponibles, exp, log (logarithme d'Iwasawa), la fonction $\zeta$ (de Kubota-Leopoldt), etc.

```
gp > 2748 + O(2^12)
%1 = 2^2 + 2^3 + 2^4 + 2^5 + 2^7 + 2^9 + 2^11 + O(2^12)
gp > 1729 + O(2^15)
%2 = 1 + 2^6 + 2^7 + 2^9 + 2^10 + O(2^15)
gp > %1 / %2
%3 = 2^2 + 2^3 + 2^4 + 2^5 + 2^7 + 2^8 + 2^10 + O(2^12)
gp > 1 / %
%4 = 2^-2 + 2^-1 + 1 + 2 + 2^5 + 2^6 + O(2^8)
gp > factorpadic(x^3-x-1, 7, 10)
%5 =
[(1 + O(7^10))*x + (2 + 5*7 + 6*7^2 + 2*7^3 + (...) + O(7^10)) 1]

[(1 + O(7^10))*x^2 + (...) + (3 + 6*7 + 2*7^2 + (...) + O(7^10)) 1]
```

2.3.4. *Le type* `t_INTMOD`. C'est le type des éléments de $\mathbb{Z}/N\mathbb{Z}$. On crée un tel élément par `Mod(x, N)`. Diverses fonctions s'appliquent à ces éléments (`issquare`, `sqrt`, `chinese`, fonctions arithmétiques).

```
gp > issquare(Mod(2, 2^127-1))
%1 = 1
gp > lift( sqrt(Mod(2, 2^127-1)) )
%2 = 184467440737095516 16
gp > chinese(Mod(2, 10), Mod(3, 13))
%3 = Mod(42, 130)
gp > chinese(Mod(2, 10), Mod(3, 5))
  *** chinese: incompatible arguments in chinois.
gp > a = Mod(5, 143)^7
%4 = Mod(47, 143)
gp > b = 1/Mod(7, eulerphi(143))
%5 = Mod(103, 120)
gp > a^103
%6 = Mod(5, 143)
gp > a^lift(b)
```

```
%7 = Mod(5, 143)
```

Pour convertir un élément de type `t_INTMOD` en entier usuel, on utilise la fonction `lift` qui donne le relèvement canonique (dans $[0, N-1]$) de l'élément, ou `centerlift` qui donne le relèvement centré (dans $]-N/2, N/2]$). Calculons maintenant une racine primitive modulo 101 et résolvons un problème de log discret dans le (groupe multiplicatif du) corps premier $\mathbb{Z}/101\mathbb{Z}$ :

```
gp> g = znprimroot(101)
%1 = Mod(2, 101)
gp> g = znlog(7, g)
%2 = 9
```

La fonction `znstar` permet d'obtenir la structure de $(\mathbb{Z}/N\mathbb{Z})^*$ même quand il n'est pas cyclique.

2.3.5. *Le type* `t_FRAC`. Nombres rationnels ; la fraction est réduite au plus petit dénominateur.

La fraction continue d'un nombre réel ou d'une fraction peut être obtenue via `contfrac`, `contfracpnqn` : la première fonction donne les quotients partiels, la seconde les réduites partant des quotients partiels.

2.3.6. *Le type* `t_FFELT`. Éléments d'un corps fini. On les construit à l'aide de `ffgen` et du polynôme minimal d'un élément primitif sur le corps premier, souvent obtenu via `ffinit`.

```
gp > T = ffinit(3, 4, t)
%1 = Mod(1, 3)*t^4 + Mod(1, 3)*t^3 + Mod(1, 3)*t^2 + Mod(1, 3)*t + Mod(1, 3)
gp > t = ffgen(T)
%2 = t
gp > t^4
%3 = 2*t^3 + 2*t^2 + 2*t + 2
gp > g = ffprimroot(t)
%4 = 2*t^2 + t + 1
gp > fflog(t, g)
%5 = 48
gp > g^48
%6 = t
```

Ceci construit un modèle $\mathbb{F}_3[t]/(T)$ de $\mathbb{F}_{3^4}$ ; par abus de langage, on continue de noter $t$ l'image par la projection canonique de la variable de $\mathbb{F}_3[t]$, mais le résultat `%3` montre bien qu'il ne s'agit pas d'un polynôme ordinaire ! Puis on calcule un générateur $g$ de $\mathbb{F}_{3^4}^*$ et on résoud un problème de log discret dans ce groupe.

2.3.7. *Le type* `t_QUAD`. Le type `t_QUAD` permet de manipuler les éléments d'ordres quadratiques. On définit l'ordre par l'appel à la fonction `quadgen`, qui prend le discriminant en argument, et renvoie une valeur notée `w`. On peut alors calculer sur les éléments du corps des fractions, qui sont de la forme `a+b*w`.

```
gp > s97 = quadgen(97)
%1 = w
gp > charpoly(s97)
```

```
%2 = x^2 - x - 24
gp > 5 + 3*s97
%3 = 5 + 3*w
gp > %^2
%4 = 241 + 39*w
gp > s97^2
%5 = 24 + w
```

C'est un type obsolète. Il vaut mieux utiliser `t_POLMOD`, qui est un peu plus lent, mais bien plus général et prête moins à confusion.

2.3.8. *Le type* `t_POL`. Le type des polynômes univariés. Les coefficients peuvent avoir n'importe quel type. Les polynômes multivariés sont implantés comme des polynômes univariés à coefficients polynomiaux. L'ordre des variables fonctionne sur le principe "premier arrivé, premier servi", à l'exception de la variable `x`, prédéfinie. La fonction `variable()` fournit les priorités courantes des variables.

Une variable à laquelle a été affectée une valeur peut encore être appelée par `'x` ; on peut donc "récupérer" la variable par `x = 'x`.

```
gp > t; z; y;
gp > f = z^3*t+y*t+z^2*y
%2 = (z^3 + y)*t + y*z^2
```

La substitution d'une expression à une variable se fait par la fonction `subst`. Mais on peut faire des substitutions plus évoluées grâce à `substpol` et `substvec` :

```
gp > subst(f, z, t+y)
%3 = t^4 + 3*y*t^3 + (3*y^2 + y)*t^2 + (y^3 + 2*y^2 + y)*t + y^3
gp > substpol(x^4 + x^2*y + 1, x^2, x)
%4 = x^2 + y*x + 1
gp > substvec(x+y^2+z^3, [x,y,z], [1,x,y+x])
%5 = x^3 + (3*y + 1)*x^2 + 3*y^2*x + (y^3 + 1)
```

Les fonctions suivantes permettent de manipuler des polynômes univariés : `gcd`, `factor`, `factormod`, `polcyclo`, `poldisc`, `polresultant`, `polroots`, `polrootsmod`, `polrootspadic`, `polsturm`, `polsym`. Faire `?8` pour une liste.

Signalons une fonction utile, basée sur des techniques de réduction des réseaux : étant donné un polynôme irréductible $P \in \mathbb{Z}[X]$, `polred` et `polredabs` cherchent un polynôme $Q$ à petits coefficients qui définisse la même extension que $P$, c'est-à-dire tel que $\mathbb{Q}[X]/(P) \simeq \mathbb{Q}[X]/(Q)$. L'option 1 de `polred` ou `polredabs` fournit en plus l'isomorphisme permettant de passer d'une racine de $P$ à une racine de $Q$.

```
gp > P = x^3 + 371025000*x^2 + 45886547466011250*x\
     + 189167543773184802740625 0;
gp > Q = polredabs(P)
%2 = x^3 + 2*x - 2
gp > V = polredabs(P, 1)
%3 = [x^3 + 2*x - 2, Mod(123675*x - 123675000, x^3 + 2*x - 2)]
gp > subst(P, x, V[2])
%4 = 0
gp > modreverse(V[2])
```

```
%5 = Mod(1/123675*x + 1000, x^3 + 371025000*x^2 + 458865474(...)
gp > subst(Q, x, %)
%6 = 0
```

2.3.9. *Le type* `t_POLMOD`. C'est l'équivalent du type `t_INTMOD` pour les polynômes, qui permet de construire des extensions algébriques d'algèbres.

```
gp > Mod(x^3, x^7+1)
%1 = Mod(x^3, x^7 + 1)
gp > 1/%
%2 = Mod(-x^4, x^7 + 1)
gp > chinese(%, Mod(x^2, x^3-1))
%3 = Mod(x^8 - x^4 + x, x^10 - x^7 + x^3 - 1)
```

**Exercice :** Calculer l'inverse dans $\mathbb{F}_4 = \mathbb{F}_2[X]/(X^2+X+1)$ de l'élément $X$. En déduire qu'il est ardu d'utiliser `gp` pour les calculs dans les corps finis non premiers. Ceux qui ne sont pas convaincus peuvent calculer l'inverse de $Y+1$ dans $\mathbb{F}_{16} = \mathbb{F}_4[Y]/(Y^2+Y+X)$.

**Solution naïve :**

```
gp > T = Mod(1,2)*(X^2+X+1); 1 / Mod(X, T)
%1 = Mod(Mod(1,2)*X + Mod(1,2), Mod(1,2)*X^2 + Mod(1,2)*X + Mod(1,2))

gp > 1 / Mod(Y+1, (Y^2 + Y + X) * Mod(1,T))
  *** Mod: forbidden division t_POLMOD % t_POLMOD.
gp > 1 / Mod(x+1, (x^2 + x + X) * Mod(1,T))
%2 = Mod(
  Mod(Mod(1, 2)*X + Mod(1, 2),
      Mod(1, 2)*X^2 + Mod(1, 2)*X + Mod(1, 2))*x,

  Mod(1, Mod(1, 2)*X^2 + Mod(1, 2)*X + Mod(1, 2))*x^2
+ Mod(1, Mod(1, 2)*X^2 + Mod(1, 2)*X + Mod(1, 2))*x
+ Mod(X, Mod(1, 2)*X^2 + Mod(1, 2)*X + Mod(1, 2)))
```

Le premier essai échoue car les priorités relatives des variables sont incorrectes : comme "$X < Y$", `gp` interprète `(Y^2 - X) * Mod(1,T)` comme $-X+Y^2 \pmod T$ qui est un élément de $\big(\mathbb{F}_2(Y)[X]\big)/(T)$, et non comme un élément de $\big(\mathbb{F}_2[X]/(T)\big)[Y]$. Le deuxième essaie fonctionne car "$x < X$".

**Solution évoluée :** Plutôt qu'utiliser des `t_POLMOD` de `t_POLMOD`, il est préférable de réaliser un gros corps fini par un élément primitif sur son corps premier, et y plonger les corps intermédiaires grâce à `factorff` :

```
gp > T = Mod(1,2)*(x^2+x+1);
gp > U = ffinit(2, 4, y);
gp > X = - polcoeff( factorff(T, 2, U)[1,1], 0 ); \\ une racine de T dans F16
gp > Y = - polcoeff( factorff(x^2+x + X, 2, U)[1,1], 0 );
gp > 1/(Y+1)
```

2.3.10. *Le type* `t_SER`. Ce type permet de manipuler des séries formelles. À l'instar des nombres $p$-adiques, une série formelle s'entre comme $P(X)+O(X^n)$, où $P$ est une fonction rationnelle. On peut effectuer les 4 opérations sur les séries formelles, les composer (fonction `subst`), en prendre l'exponentielle, le logarithme, le cosinus, etc. On accède au développement limité des fonctions usuelles en les évaluant en un paramètre formel (la précision des séries formelles se gère comme pour les flottants, en utilisant `\ps` cette fois) :

```
gp > \ps
   seriesprecision = 16 significant terms
gp > cos(x)
%1 = 1 - 1/2*x^2 + 1/24*x^4 - 1/720*x^6 + 1/40320*x^8 \
  - 1/3628800*x^10 + 1/479001600*x^12 - 1/87178291200*x^14 \
  + 1/20922789888000*x^16 + O(x^18)
gp > \ps3
   seriesprecision = 3 significant terms
gp > sin(x)
%2 = x - 1/6*x^3 + O(x^4)
```

Le premier calcul commence par la conversion $x \to x + O(x^{17}) = x(1 + O(x^{16}))$ (soit 16 termes significatifs) ce qui permet d'obtenir 16 termes de $\cos(x) - 1 = x^2 + \ldots$, et donc 18 termes dans le résultat final.

2.3.11. *Le type* `t_RFRAC`. Ce type, similaire aux type `t_FRAC`, implante les fonctions rationnelles.

2.3.12. *Les types* `t_VEC, t_COL, t_MAT`. La façon la plus simple de construire un vecteur consiste à saisir ses composantes entre crochets. Un vecteur colonne est obtenu en transposant un vecteur ligne, ce qui peut se faire soit par la fonction `mattranspose`, soit par le suffixe ~.

```
gp > v = [1,2,3]
%1 = [1, 2, 3]
gp > type(%)
%2 = "t_VEC"
gp > mattranspose(%')
%3 = [1, 2, 3]~
gp > v~
%4 = [1, 2, 3]~
gp > type(%)
%5 = "t_COL"
```

Une matrice, quant à elle, se saisit de la façon suivante :

```
gp > [1,2,3; 4,5,6; 7,8,9]
%6 =
[1 2 3]

[4 5 6]

[7 8 9]
```

gp implante bon nombre d'opérations d'algèbre linéaire (sur un anneau euclidien ou un corps) : pivot de Gauss, polynôme caractéristique, déterminant, HNF, SNF, noyau, image, etc. Faire `?8` pour une liste.

L'accès à une composante d'une matrice ou d'un vecteur se fait via `v[i]` ou `m[i,j]` ; on peut également utiliser `m[i,]` (resp. `m[,j]`) pour accéder à la $i$-ème ligne, (resp. $j$-ème colonne) de la matrice `m`. La fonction `vecextract` permet en outre d'extraire une partie d'un vecteur ou d'une matrice. Les numéros de ligne et de colonne commencent à 1. La longueur d'un vecteur `v` s'obtient par `length(v)` ou `#v`.

Une matrice ou un vecteur doit être créé avant de pouvoir modifier ses entrées. Faire `v[1] = 2` sans avoir créé le vecteur `v` au préalable cause une erreur. On peut par exemple utiliser la commande `v = vector(15)`, ou `M = matrix(5,10)`.

```
gp > M = matrix(20, 20, j, k, gcd(j, k));
gp > matdet(M)
%2 = 46965467381760
gp > factorint(%)
%3 =
[2 32]

[3 7]

[5 1]

gp > N = M; N[2,2] = -1; matdet(N)
%4 = -346370321940480
gp > matsnf(M, 4)
%5 = [720, 24, 24, 24, 12, 12, 4, 4, 4, 2, 2, 2, 2, 2, 2, 2, 2, 2]
gp > M2 = vecextract(M, "-10..");  \\ 10 dernières colonnes
gp > matsize(matkerint(M2~))
%6 = [20, 10]
```

Un point à noter : une commande qui se termine par un point-virgule ne voit pas son résultat affiché ; toutefois, celui-ci reste numéroté et disponible par son numéro (`%1`, `%4`) ici pour utilisation ultérieure. Autre point à noter : les objets `%x` ne peuvent être modifiés.

gp offre en outre une implantation de l'algorithme LLL accessible via `qflll`. Par exemple :

```
gp > N = 10^20; [1,0,0; round(N*Pi),N,0; round(N*exp(1)),0,N]
%1 =
[1 0 0]

[314159265358979323846 100000000000000000000 0]

[271828182845904523536 0 100000000000000000000]
```

```
gp > U = qflll(%)
%2 =
[15431582173353 -4771327925432 -8876602352967]

[-48479745189073 14989568758405 27886668740919]

[-41947389406198 12969813997321 24129106874527]

gp > frac(U[1,1] * Pi)
%3 = 0.000000133262572
gp > frac(U[1,1] * exp(1))
%4 = 0.000000032444343
```

Noter que `qflll` renvoie une matrice $U$ de déterminant $\pm 1$, donnant une base LLL-réduite en termes de la base canonique ; les vecteurs courts du réseau proprement dits sont donnés par `M * qflll(M)`.

Parmi les autres applications, signalons les fonctions `algdep` et `lindep`, qui tentent de trouver une relation algébrique vérifiée par un réel ou un complexe, ou une relation entière vérifiée par un vecteur de flottants.

```
gp > lindep([log(15), log(3), log(5)])
%1 = [1, -1, -1]~
gp > polroots(x^3 - x + 1)
%2 = [-1.3247179572447460259609088 54 + 0.E-28*I, (...)]
gp > sqrt(%[1])
%3 = 1.150963925257758035680601218*I
gp > algdep(%, 8)
%4 = x^6 - x^2 + 1
```

Noter qu'il faut donner à `algdep` une borne sur le degré de la relation à trouver.

## 3. Fonctions sur les corps de nombres

Il semble important de consacrer une tête de section, fût-elle courte, à ces fonctionnalités qui sont le fer de lance de Pari et sa principale originalité. Toutefois, expliquer le rôle de ces différentes fonctions et la nature des objets qu'elles calculent dépasse largement le cadre de cet exposé. Signalons juste que `?6` permet d'accéder à une liste. Pour le reste, on renvoie par exemple au tutoriel de la distribution, aux livres d'Henri Cohen [3, 2], et au survol [1].

## 4. Fonctions diverses

GP offre un grand nombre de fonctions d'intégration et de sommation numérique, limitées aux cas univarié. La plupart d'entre elles sont étudiées pour le calcul à grande précision. Leur comportement est très convenable sur les fonctions simples :

```
gp > \p105
gp > intnum(x=1, 2, 1/x^2) - 1/2
```

```
%1 = 0.E-106
```

Par contre, elles sont très sensibles aux singularités, ainsi qu'au comportement asymptotique des fonctions oscillantes. Il est donc préférable d'indiquer explicitement les problèmes, sous peine de résultats ridicules :

```
? oo = [1];     \\ definition, pour alléger l'écriture
? \p105
? intnum(x = 0, oo, sin(x) / x) - Pi/2
%2 = 20.78..    \\ n'importe quoi
? intnum(x = 0, [oo,-I], sin(x)/x) - Pi/2
%3 = 0.E-105    \\ parfait
```

Ici le `-I` signale que la fonction est oscillante, de type $f(x)\sin(x)$, où $f(x)$ tend lentement vers 0. Regarder `??intnum` et `??sumnum` pour la syntaxe et de nombreux examples, puis `?9` pour une liste complète.

Finalement, quelques fonctions graphiques sont disponible, pour des tracés simples. Voir `?10` pour de plus amples renseignements.

## 5. Scripts GP

Cette partie décrit les structures de contrôle de GP qui permettent d'écrire de petits scripts. On peut accéder à une description complète par `?11`. Ces structures sont voisines de celles de C.

5.1. **Variables.** Les variables peuvent, hormis les noms réservés, porter n'importe quel nom composé de caractères alphanumériques (plus le caractère _) commençant par un caractère alphabétique. L'ensemble des noms réservés inclut toutes les fonctions et constantes prédéfinies.

5.2. **Fonctions.** Les fonctions utilisateur sont simplement définies par

```
f(args) = cmd
```

où `args` est une liste d'arguments (un argument peut être du type `n=5`, ce qui signifie que si l'argument correspondant n'est pas spécifié par l'utilisateur, la valeur par défaut de `n` est 5 ; si un argument pour lequel aucune valeur par défaut n'est donnée est omis par l'utilisateur, il prend la valeur 0).

```
gp > f(x,y=1,z) = x+2*y+4*z
gp > print( f(), ":", f(1,,1), ":", f(,1,1) )
2:7:6
```

La valeur de retour d'une fonction est celle de la dernière expression évaluée dans celle-ci. On peut aussi utiliser explicitement `return(val)`, qui interrompt l'exécution de la fonction et fixe la valeur de retour à `val`.

En fait, une fonction utilisateur n'est rien d'autre qu'un objet de type `t_CLOSURE`, assigné à une variable ordinaire. Plutôt que ce qui la construction ci-dessus, on peut utiliser la construction équivalente

```
gp > f = (x,y=1,z) -> x+2*y+4*z
```

(qui associe une fonction à la variable `f`) et même utiliser des «fonctions anonymes», sans leur donner de nom :

```
gp > vecsort([-3,2,-1])
%1 = [-3, -1, 2]
gp > vecsort([-3,2,-1], (x,y) -> abs(x) - abs(y))
%2 = [-1, 2, -3]
```

Dans le dernier exemple on appelle une fonction de tri en lui imposant une fonction de comparaison explicite. Le résultat est un vecteur trié par ordre croissant des *valeur absolues*, et non pas pour la relation d'ordre ordinaire.

5.3. **Boucles `for`.** La syntaxe GP d'une boucle `for` est `for(X = a, b, cmd)`. Elle permet d'exécuter la suite de commandes `cmd` pour une valeur de la variable `X` allant de `a` à `b`. Ici, `cmd` est une séquence de commandes séparées par des points-virgules. Signalons trois variantes utiles :

- `fordiv(n, X, cmd)`, qui affecte à la variable `X` les diviseurs successifs de `n` dans l'ordre croissant et exécute `cmd` ;
- `forprime(X = a, b, cmd)`, qui affecte à `X` les nombres premiers successifs de l'intervalle [`a`, `b`]
- `forvec(X = [m, M], cmd)` ; dans ce dernier cas, `m` et `M` sont des vecteurs de même dimension $k$ et $X$ parcourt le produit cartésien $\prod_{i=1}^{k} [\,\mathtt{m}[i], \mathtt{M}[i]\,]$.

On pourra aussi consulter l'aide de `forstep, forsubgroup`. Il est possible de modifier directement la valeur de `X` dans le corps de la boucle. De plus, `X` est locale à la boucle et sa valeur initiale est restaurée en sortie de boucle.

5.4. **Tests.** La syntaxe de `if` est `if(test, cmd1, cmd2)`. La mention de `cmd2` est facultative ; on peut donc écrire `if (test, cmd1)`. Si la condition `test` est vrai, alors `cmd1` est exécutée, sinon `cmd2` est exécuté s'il est présent.

GP, comme C, n'a pas de type booléen ; le résultat d'un test (`>`, `<`, `==`) est un entier ; sa valeur est `vrai` si cet entier est non nul, `faux` sinon. Ainsi

```
if (n % i, print(i));
```

affiche `i` ssi il ne divise pas `n`.

**Piège :** Il est facile de confondre `=` et `==`. Comme en C, la tournure `if(a = b, cmd1, cmd2)` exécute `cmd1` si `b` ne s'évalue pas à 0 (la valeur de retour de `a=b` est b). De même, `if (a=0, cmd1, cmd2)` exécute `cmd2`.

Les opérateurs ET, OU et NON se notent `&&, ||, !` (le NON est prefixe ; comme suffixe, il désigne la factorielle).

5.5. **Boucles `while` et `until`.** Ces deux boucles s'écrivent `while(a, cmd)` et `until(a, cmd)`. La suite de commandes `cmd` est exécutée tant que `a` est non nul, ou jusqu'à ce que `a` soit non nul. De plus `while` teste la condition avant d'exécuter `cmd`, alors que `until` exécute d'abord `cmd` avant de tester la condition.

5.6. `break, next.` Ces deux instructions permettent de sortir de la structure de contrôle courante, et de remonter au niveau supérieur. Dans le cas de `next`, on avance d'une itération au niveau atteint. Ces deux commandes admettent un argument entier qui permet de spécifier le nombre de niveaux à remonter.

5.7. **Variables locales.** Toutes les variables définies sont considérées comme globales, et traitées comme telles dans les fonctions :

```
gp > p = 3
%1 = 3
gp > f(x) = x+p
gp > f(2)
%2 = 5
gp > f(x) = t=2; g(3)
gp > g(x) = x+t
gp > f(5)
%3 = 5
```

Toutefois, l'utilisation d'un nom de variable comme paramètre d'une fonction (ou variable d'une boucle) cache sa valeur globale

Le mot-clé `my` permet de limiter la portée d'une variable à la fonction courante, ou même à la boucle courante. Une variable déclarée par `my` est initialisée à 0. Pour éviter cette initialisation, utiliser `my(x='x)`.

```
gp > f(x) = t=2
gp > t
%1 = t
gp > f(2); t
%3 = 2   \\ la variable globale t a été modifiée
gp > t = 't
%4 = t
gp > g(x) = my(t); t=2;
gp > g(2); t
%5 = t   \\ la variable globale t n'est pas affectée
```

Pour éviter les conflits de noms de variables, qui provoquent des erreurs rarement compréhensibles, un bon principe consiste à déclarer les variables locales comme telles, en leur donnant la portée minimale possible : si une variable temporaire n'est utile que dans le corps d'une boucle, la déclarer `my` pour la boucle. Enfin, pour utiliser une variable formelle `x` et éviter les mauvaises surprises, utiliser `'x`.

5.8. **Écriture de scripts.** Un script peut se saisir directement dans une fenêtre `gp`. Pour éviter que les retours chariots soient interprétés (et donc l'ensemble de l'expression évaluée telle quelle), on écrit usuellement une fonction GP de la façon suivante :

```
f(x) =
{ my(y = 1);
```

```
  ...
}
```

Les retours chariots après le = initial et à l'intérieur des accolades sont ignorés. Il est en général préférable de saisir le script dans un éditeur de texte, puis de lire le fichier correspondant par la commande `\r`.

L'appel à `?f` permet à tout moment de récupérer le code d'une fonction écrite précédemment.

## 6. La pratique

**Exercice 1 :** Écrire un script qui prend en argument un entier $n$ et renvoie le $n$-ème nombre de Fibonacci.

**Exercice 2 :** Écrire un script qui effectue un produit de matrices.

**Exercice 3 :** Écrire un script qui implante le test de primalité de Miller-Rabin : prenant en argument un entier $n$ et un paramètre $p$, on écrit $n-1 = 2^s t$, et on tire successivement $p$ valeurs de $a \in [1, n-1]$ au hasard, pour lesquelles on teste si $a^t = 1 \bmod n$, ou il existe $k \in [0, s-1]$ tel que $a^{2^k t} = -1 \bmod n$. Si un de ces tests est faux, on sait que $n$ n'est pas premier. On pourra utiliser `valuation` et `random` (voir l'aide).

## 7. Le mode bibliothèque

Il nous reste à décrire la façon d'accéder à toutes ces fonctions en mode bibliothèque.

### 7.1. Fichiers include.

Afin d'utiliser Pari en mode bibliothèque, il faut inclure le fichier `pari/pari.h`, et linker avec la bibliothèque `libpari`.

### 7.2. Le type `GEN`.

Tous les objets Pari sont d'un type unique, à savoir `GEN`, qui est défini comme un pointeur sur un `long`. Chaque objet `x` de type `GEN` contient un premier mot de code `x[0]` contenant des informations, principalement sur le type au sens GP de l'objet, ainsi que la taille, etc. Le contenu précis dépend du type considéré. On renvoie au manuel (et au code source) pour tous les détails.

### 7.3. Philosophie générale.

Pari utilise une pile pour gérer l'allocation dynamique de mémoire. Chaque fonction est responsable de l'allocation de la mémoire de la valeur de retour mais aussi de la libération de la memoire utilisée par le calcul intermédiaires. La pile rend la gestion mémoire très peu couteuse en temps et espace, mais doit être nettoyée manuellement.

Pari a choisi une philosophie d'écriture plus proche de l'habitude conventionnelle que celle de GMP [4]. Là où GMP écrit `mpz_add(x, y, z)`, Pari écrit `x = gadd(y, z)`. Ce choix implique deux choses :

- Dans la tournure `x = gadd(y, z)`, l'allocation du résultat est opérée par la fonction `gadd` elle-même. Ainsi l'utilisateur n'a-t-il pas à se soucier d'effectuer cette allocation. De façon générale, seuls les objets construits

"à la main" par l'utilisateur nécessitent d'effectuer une allocation, qui se fait alors par la commande `cgetg(taille, type)`. La taille désigne le nombre de mots à réserver pour l'objet (attention au(x) mot(s) de code !), et type peut être `t_INT`, `t_REAL`, etc.

- Dans la tournure `x = gadd(y, gadd(z, t))`, un objet intermédiaire est créé qui ne sera pas accessible à l'utilisateur. Il faut que l'utilisateur gère lui-même l'utilisation de la pile de mémoire bloquée par PARI.

7.4. **Gestion de la mémoire.** Avant tout appel à une fonction PARI, il faut appeler la fonction `pari_init`. Cette fonction se charge de bloquer un espace mémoire dédié à PARI, et de précalculer un certain nombre de nombres premiers. `pari_init`, en particulier, prend donc deux arguments, soit l'espace à réserver pour la pile, et la limite des nombres premiers à calculer. L'appel crée en plus de la pile des constantes `gen_0`, `gen_1`, `gen_m1`, `gen_2`, etc, et un vecteur de variables. En outre, un *tas* est créé, contenant les objets universels qui ne doivent pas être supprimés.

L'état courant de la mémoire est fourni à l'utilisateur par la variable globale `avma` (*available memory address*), de type `pari_sp` (stack pointer), qui décroît au fur et à mesure de l'exécution et pointe sur la fin de la zone de la pile encore disponible. Les variables `top` et `bot` pointent vers le bas et le haut de la pile.

Il appartient à l'utilisateur de veiller à "nettoyer" la pile périodiquement. À cette fin, plusieurs fonctions sont disponibles. Le cas le plus simple est le cas où tous les résultats calculés entre deux points du code peuvent être jetés : il suffit de sauvegarder la valeur de `avma` en début de section de code et de réaffecter cette valeur à `avma` à la fin de cette section.

```
void
affiche_add(GEN x, GEN y) {
  pari_sp ltop = avma;

  output( gadd(x, y) );
  avma = ltop;
}
```

Parfois des calculs intermédiaires sont nécessaires, dont les résultats doivent être jetés sans détruire le résultat final. Il faut alors utiliser la fonction `gerepile`, ou une de ses variantes. Par exemple :

```
GEN
add3(GEN x, GEN y, GEN z) {
  pari_sp ltop = avma, lbot;
  GEN t = gadd(y, z);

  lbot = avma;
  return gerepile(ltop, lbot, gadd(x, u));
}
```

La fonction `gerepile(ltop, lbot, t)` effectue les opérations suivantes :

- Elle ne modifie pas la zone mémoire entre `ltop` et `top` ;

- La zone mémoire comprise entre `avma` et `lbot` est déplacée en `ltop` ; les pointeurs sont mis à jour.
- La zone située entre `lbot` et `ltop` est purement et simplement supprimée.
- La valeur de retour est la nouvelle adresse de l'objet `t`.

Une difficulté à ne pas oublier : si un objet PARI est situé entre `avma` et `lbot`, mais si certains de ses composants sont situés entre `lbot` et `ltop`, une erreur est soulevée par `gerepile`. C'est possible par exemple avec un vecteur :

```
GEN
fun(GEN x, GEN y, GEN z) {
  pari_sp ltop = avma, lbot;
  GEN vec, t = gadd(x, gmul(y, z));

  lbot = avma;
  vec = cgetg(t_VEC, 4);
  gel(vec,1) = x;
  gel(vec,2) = t;
  gel(vec,3) = gadd(t, z);

  return gerepile(ltop, lbot, vec);
}
```

Ce code produit une erreur ("significant pointers lost in gerepile") ; en effet, si `vec` est préservé, sa deuxième composante est effacée.

**Exercice :** Comment corriger le code ci-dessus ?

Quelques variantes de `gerepile`, qui suffisent en général :

- `gerepileupto(pari_sp ltop, GEN q)` nettoie la pile entre `ltop` et `q`, et renvoie `q` mis à jour. Attention : pour que cette syntaxe fonctionne, `q` doit avoir été créé *avant* ses composants (comme dans l'exemple plus haut).
- `gerepilecopy(pari_sp ltop, GEN x)` est fonctionnellement équivalent à l'instruction `gerepileupto(ltop, gcopy(x))`, quoique plus rapide : on met `x` en sécurité, on nettoie la pile de `ltop` à la position courante, et on renvoie la nouvelle adresse de `x`.
- `gerepileall(pari_sp ltop, int n, ...)` permet de nettoyer la pile entre `ltop` et la position courante. Les `n` objets dont les adresses suivent sont préservés et leurs adresses mises à jour. Par exemple

  ```
  gerepileall(ltop, 3, &x, &y, &(z[1]));
  ```

Les deux fonctions les plus lentes sont `gerepilecopy` et `gerepileall`, par la recopie qu'elles impliquent. En contrepartie, elles ne supposent rien sur la structure des objets qu'elles mettent à jour.

**Exercice :** Écrire une fonction PARI qui prend en argument deux réels `x` et `y` et renvoie le vecteur `[x^2+y, y^2+x]`.

**Exercice :** Écrire une fonction PARI qui prend en argument un polynôme et un nombre réel, et qui évalue le polynôme au point correspondant.

**Exercice :** Écrire une fonction PARI qui implante le test de primalité de Miller-Rabin. On pourra utiliser la fonction `Fp_pow`.

## 8. `install` ET `gp2c`

Cette partie évoque deux "interfaces" entre PARI et `gp`. La fonction `install` permet d'accéder à un symbole d'un fichier `.so` depuis `gp`. En particulier, une fonction écrite en PARI peut être utilisée depuis `gp`. Cette possibilité n'existe pas sur certains vieux systèmes d'exploitation (par exemple Mac OS 9), et ne fonctionne généralement que dans le cas d'un binaire `gp` dynamique.

L'interface de cette fonction est délicate, en particulier parce que le type des arguments (au sens du C) doit être fourni à `install`. On renvoie au manuel pour les détails.

Le compilateur `gp2c` permet de compiler des scripts GP vers PARI. On constate des gains en efficacité significatifs dès lors que le code GP d'origine est un tant soit peu complexe. Noter que pour pouvoir compiler le langage de script GP, il a fallu l'affaiblir (certaines syntaxes, inutiles en principe en mode non interactif, ne sont pas valides). En contrepartie, pour permettre des gains en efficacité, certaines directives de compilation peuvent être intégrées aux scripts, précisant les types des objets manipulés (e.g., le cas où un entier devrait être stocké dans un entier long du C).

## RÉFÉRENCES

1. Karim Belabas, *Topics in computational algebraic number theory*, J. Théor. Nombres Bordeaux **16** (2004), no. 1, 19–63. MR MR2145572
2. H. Cohen, *Advanced topics in computational number theory*, Springer-Verlag, 2000. MR 1 728 313
3. Henri Cohen, *A course in computational algebraic number theory*, Graduate Texts in Mathematics, vol. 138, Springer-Verlag, Berlin, 1993. MR MR1228206 (94i :11105)
4. Torbjorn Granlund, *GMP : GNU Multi Precision library*, `http://swox.com/gmp/`.
5. *PARI/GP, version* `2.2.11`, Bordeaux, 2005, `http://pari.math.u-bordeaux.fr/`.