RAPPORT DE PROJET : GESTION DE LA MÉMOIRE DANS MAGMA

Objectif du projet

Mon travail consistait, dans un premier temps, en une évolution du code déjà écrit par Mme. Pieltant afin de permettre le stockage en mémoire de places après leur calcul, afin de permettre leur reconstruction en un temps inférieur dans une session ultérieure de Magma. Dans un second temps, il a été question de nettoyer la mémoire utilisée par Magma pendant l'exécution du calcul, afin d'éviter des problèmes de dépassement de la mémoire autorisée. Les sections 1 à 3 présentent le code initial de Mme. Pieltant. Les sections suivantes présentent les évolutions qui lui ont été apportées.

1. Définition de la tour

On considère une suite de corps de fonctions, appelée *tour*, définie sur $\mathbb{F}_{q^2}$ récursivement par l'équation :

$$Y^q + Y = \frac{X^q}{X^{q-1}+1}$$

Autrement dit, il s'agit d'une succession d'extensions $K_0 \subset K_1 \subset K_2 \subset \ldots \subset K_\ell \subset \ldots$ où

- $K_0 := \mathbb{F}_{q^2}(x_0)$ est le corps de fonctions rationnelles sur $\mathbb{F}_{q^2}$
- pour tout $\ell \geq 0$, on a $K_{\ell+1} := K_\ell(x_{\ell+1})$ avec $x_{\ell+1}$ satisfaisant l'équation :

$$x_{\ell+1}^q + x_{\ell+1} = \frac{x_\ell^q}{x_\ell^{q-1}+1}$$

Le programme commence donc par la définition du corps $\mathbb{F}_{q^2}$, puis des premiers étages de la tour :

```
FonctionConstructMatrixA := function(q,l:Verbose := false, Timings := false)

// q : tour sur $F_(q^2)$
// l : generation des etages K0, K1,..., Kl

tempsDExecution := Cputime() ;

k<delta> := FiniteField(q^2) ;

///////////////////////////////////////////////////
// DEFINITION DES ETAGES K_0, ..., K_l DE LA TOUR //
///////////////////////////////////////////////////

K0<x0> := FunctionField(k) ;
Pol<T> := PolynomialRing(K0) ;
K1<x1> := FunctionField(T^q+T-x0^q/(x0^(q-1)+1) : Check := false) ;
Pol<T> := ChangeRing(Pol, K1) ;

K := [K1] ;
x := AssociativeArray(Integers()) ;
x[0] := x0 ;
x[1] := x1 ;

for i in [2..l] do
    Insert(~K,i,FunctionField(T^q+T-x[i-1]^q/(x[i-1]^(q-1)+1) : Check := false)) ;
    AssignNames(~K[i],["x" cat IntegerToString(i)]);
    x[i] := K[i].1 ;
    Pol<T> := ChangeRing(Pol, K[i]) ;
end for ;
```

2. Décomposition des places rationnelles

But : Déterminer les places rationnelles dans les étages successifs.

Commençons par l'étage K_0 : les places rationnelles correspondent aux polynômes de degré 1 de $\mathbb{F}_{q^2}[x_0]$.
On note $P_\theta^{(0)}$ la place correspondant au polynôme $x_0 - \theta$, c'est-à-dire que la place $P_\theta^{(0)}$ est l'unique zéro de la fonction $x_0 - \theta$ dans K_0. De même, on note $P_\infty^{(0)}$ la place qui est l'unique pôle de x_0 dans K_0.

Une étude approfondie de la tour a permis de déterminer le comportement des places rationnelles, c'est-à-dire comment se décompose une place $P_i^{(0)}$ dans la tour (si une place $P_i^{(0)}$ donne une ou plusieurs places rationnelles dans les extensions successives de K_0). Ainsi on sait que :

- la place $P_\infty^{(0)}$ est *totalement ramifiée dans la tour*, ce qui veut dire qu'il y a dans chaque étage K_ℓ exactement une place rationnelle au dessus de $P_\infty^{(0)}$; on notera $P_\infty^{(\ell)}$ cette place de K_ℓ
- pour les places $P_\theta^{(0)}$, trois cas se présentent selon que θ soit 0 ou appartienne ou non à l'ensemble $\Omega^* := \Omega\backslash\{0\}$, avec
$$\Omega := \{\alpha \in \mathbb{F}_{q^2} \,|\, \alpha^q + \alpha = 0\} \quad \text{et donc} \quad \Omega^* = \{\alpha \in \mathbb{F}_{q^2} \,|\, \alpha^{q-1} + 1 = 0\}.$$
Alors :
  (a) les places $\{P_\theta^{(0)} ;\ \theta \in \Omega^*\}$ sont totalement ramifiées dans la tour. Pour chaque $\theta \in \Omega^*$, on notera $P_\theta^{(\ell)}$ l'unique place de K_ℓ correspondant à la place $P_\theta^{(0)}$ de K_0.
  (b) les places $\{P_\theta^{(0)} ;\ \theta \in \mathbb{F}_{q^2}\backslash\Omega\}$ sont *totalement décomposées dans la tour* : cela signifie que chacune de ces places se décompose en q places rationnelles distinctes dans K_1 (autant de places que le degré de l'extension $[K_1 : K_0]$) et qu'il en est ensuite de même pour chacune de ces q places dans l'extension suivante, et ainsi de suite. Les places d'un étage K_ℓ provenant de l'ensemble $\{P_\theta^{(0)} ;\ \theta \in \mathbb{F}_{q^2}\backslash\Omega\}$ sont appelées les places du code ("code places" en anglais) car ce sont sur ces places qu'on évaluera les fonctions qui permettront pour définir le code correcteur qui nous intéresse. Il y a donc $(q-1)q^\ell$ telles places dans K_ℓ.
  (c) la place $P_0^{(0)}$ a un comportement plus irrégulier qui sera détaillé ultérieurement (voir p.5).

Ce constat nous amène à distinguer les places rationnelles de K_0 selon le comportement qu'elles auront dans les extensions successives. C'est ce qui est fait dans la partie suivante du programme :

```
///////////////////////////////////////////////
// DEFINITION DES ENSEMBLES omega* ET k\omega //
///////////////////////////////////////////////

omegax := {@ k| @} ; // ensemble des elements de omega*
indexCP := {@ k| @} ; // ensemble des elements (non-nuls) de k\omega
for x in Exclude(Set(k),Zero(k)) do
    if (x^(q-1)+1 eq Zero(k)) then
       Include(~omegax, x) ;
    else
   Include(~indexCP, x) ;
    end if ;
end for ;
// affichage
if Verbose then
printf "Omega* = %o\n", omegax ;
    printf "indexCP = %o\n", indexCP ;
end if ;

//////////////////////////////////////////////
// DEFINITION DES PLACES RATIONNELLES DE K_0 //
//////////////////////////////////////////////

// DEFINITION DE LA PLACE A L'INFINI DANS K_0
Poo0 := Zeros(1/x0)[1] ;
```

```
// affichage
if Verbose then printf "Place à l'infini dans K_0 :\n P0(1/x0) = %o\n", Poo0 ;  end if ;

//DEFINITION DE P_0 DANS K_0
Po0 := Zeros(x0)[1] ;
// affichage
if Verbose then printf "Place P_0 dans K_0 :\n P0(x0) = %o\n", Po0 ; end if ;

// DETERMINATION DES CODES PLACES DE K_0
Pc0 := [ Zeros(x0-a)[1] : a in indexCP ] ;
// affichage
if Verbose then
print "Les codes places dans K_0 sont :\n" ;
for i in [1..#indexCP] do
     printf "\tP0(x0 - %o) = %o\n", indexCP[i], Pc0[i] ;
end for ;
end if ;

// DETERMINATION DES PLACES RELATIVES A omega* DANS K_0
Pm0 := [ Zeros(x0-alpha)[1] : alpha in omegax ] ;
// affichage
if Verbose then printf "Places relatives à omega* dans K_0 :\n %o\n\n", Pm0 ; end if ;

if Timings then printf "Cumulative time up to places in K0: %o\n\n", Cputime(tempsDExecution) ; end if ;
```

Ensuite, il s'agit de décomposer chacune des places rationnelles de K_0 jusqu'à l'étage final K_l qui nous intéresse (l'étage K_1 est traité indépendamment afin de pouvoir initier la récurrence) :

```
/////////////////////////////////////////////////////////////////////////////////
// DETERMINATION DES PLACES DANS K_l AU DESSUS DES PLACES RATIONNELLES DE K_0 //
/////////////////////////////////////////////////////////////////////////////////

// DETERMINATION DE LA PLACE A L'INFINI DANS K_l
Poo_temp := Decomposition(K1,Poo0)[1] ;
for i in [2..l] do
Poo_temp := Decomposition(K[i],Poo_temp)[1] ;
end for ;
Poo_l := Poo_temp ;
// affichage
if Verbose then printf "\nPlace à l'infini dans K_%o :\n P%o(1/x0) = %o\n\n", l, l, Poo_l ; end if ;

// DETERMINATION DES EXTENSIONS (TOTALEMENT RAMIFIEES) DANS K_l DES PLACES RELATIVES A omega*
Pm_temp := [ Decomposition(K1, P)[1] : P in Pm0 ] ;
for i in [2..l] do
Pm_l := [ Decomposition(K[i], P)[1] : P in Pm_temp ] ;
if Timings
then printf "Cumulative time up to totally ramified places in K_%o: %o\n", i, Cputime(tempsDExecution) ;
end if ;
Pm_temp := Pm_l ;
end for ;
// affichage
if Verbose
then printf "\n%o place(s) relative(s) à omega* dans K_%o :\n %o\n\n", #Pm_l, l, Pm_l ;
end if ;

// DETERMINATION DES CODES PLACES (TOTALEMENT DECOMPOSEES) DANS K_l
Pc_temp := [ Places(K1,1) | ] ;
pos := 0 ;
for i in [1..#Pc0] do
Stemp := Decomposition(K1, Pc0[i]) ;
Insert(~Pc_temp, pos+1, pos+1, Stemp) ;
pos := #Pc_temp ;
end for ;
```

```
for j in [2..l] do
Pc_j := Pc_temp ;
nb_it := #Pc_j ;
Pc_temp := [Places(K[j],1) | ] ;
for i in [1..nb_it] do
pos := #Pc_temp ;
Stemp := Decomposition(K[j], Pc_j[i]) ;
Insert(~Pc_temp, pos+1, pos+1, Stemp) ;
end for ;
if Timings
then printf "Cumulative time up to totally split places in K_%o: %o\n", j, Cputime(tempsDExecution) ;
end if ;
end for ;
Pc_l := Pc_temp ;
// affichage
if Verbose then printf "Les %o codes places de K_%o :\n %o\n\n", #Pc_l, l, Pc_l ; end if ;

// DETERMINATION DES PLACES AU DESSUS DE PoO DANS K_l
Potd_l := [ Zeros(K1, x1)[1] ] ;
pos := 1 ;
for alpha in omegax do
Insert(~Potd_l, pos+1, pos+1, Zeros(K1, x1-alpha)) ;
pos := #Potd_l ;
end for ;

Po_temp := Potd_l[1] ;
Poz_temp := Exclude(Potd_l, Po_temp) ;
Poz_l := [Places(K1) | ] ;
for j in [2..l] do
Potd_l := [ Zeros(K[j], x[j])[1] ] ;
pos := 1 ;
for alpha in omegax do
Insert(~Potd_l, pos+1, pos+1, Zeros(K[j], x[j]-alpha)) ;
pos := #Potd_l ;
end for ;
pos := 0 ;
Poz_l := [Places(K[j]) | ] ;
for i in [1..#Poz_temp] do
Stemp := Decomposition(K[j], Poz_temp[i]) ;
Insert(~Poz_l, pos+1, pos+1, Stemp) ;
pos := #Poz_l ;
end for ;
Po_temp := Potd_l[1] ;
Poz_temp := Insert(Exclude(Potd_l,Po_temp), q, q,Poz_l) ;
if Timings
then printf "Cumulative time up to places above P_0 in K_%o: %o\n", l, Cputime(tempsDExecution) ;
end if ;
end for ;
Po_l := Insert(Potd_l, #Potd_l+1, #Potd_l+1, Poz_l) ;
// affichage
if Verbose
then printf "\nLes %o places de K_%o au dessus de P_0 sont :\n %o\n\n", #Po_l, l, Po_l ;
end if ;

TempsTotal := Cputime(tempsDExecution) ;

if Timings
then printf "\nTemps d'execution pour q = %o et %o etages au dessus de K_0 : %o", q, l,
Cputime(tempsDExecution) ;
end if ;


TempsDecompPlaces := Cputime() ;
```

```
P := Poo_l ;
// affichage
if Verbose then printf "\nPlace a l'infini dans K_%o :\n P%o(1/x0) = %o\n\n", l, l, P ; end if ;
```

Seule cette partie du programme est l'objet du stage, on peut donc terminer la fonction (si on l'a définie au début) à cette endroit-là, par exemple en ajoutant les lignes suivantes :

```
return <omegax, indexCP, Poo0, Pm0, Pc0, Po0, Poo_l, Pm_l, Pc_l, Po_l, TempsDecompPlaces> ;
end function ;
```

Comportement des places au-dessus de $P_0^{(0)}$:

La place $P_0^{(0)}$ est totalement décomposée dans l'extention $K_0 \subset K_1$. Si on note $S^{(1)}$ l'ensemble constituée des q places de K_1 provenant de $P_0^{(0)}$ alors on sait que $S^{(1)}$ peut s'écrire sous la forme :

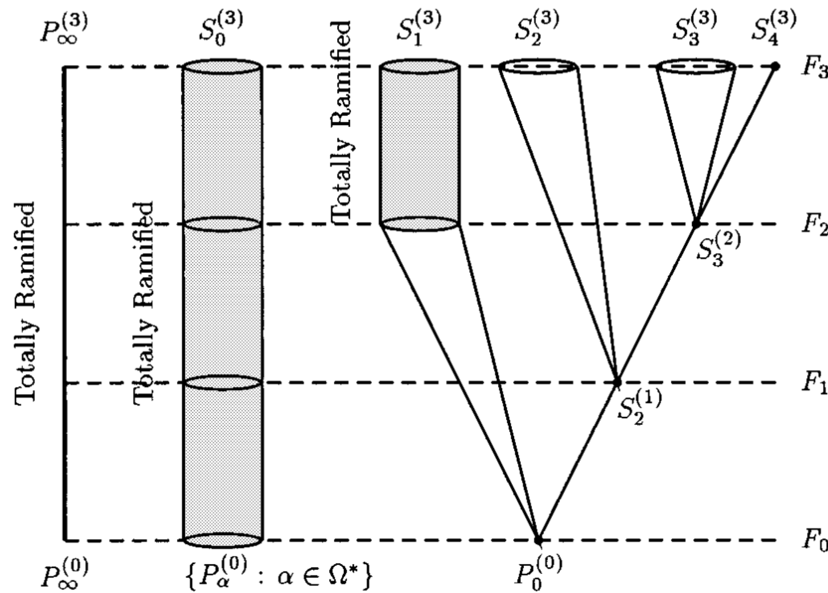
$$S^{(1)} = S_1^{(1)} \cup S_2^{(1)}$$

où $S_2^{(1)}$ est constitué d'une seule place (celle qui est également un zéro de x_1 dans K_1), et $S_1^{(1)}$ est constitué des $q-1$ places restantes (ce sont les places qui sont également les zéros respectifs que $x_1 - \alpha$ pour chacun des $q-1$ éléments $\alpha \in \Omega^*$). Notons que l'on sait que ces q places de $S^{(1)}$ sont également rationnelles.

Dans K_1 (puis $K_2, \dots$) :

- la même chose se reproduit pour la place de $S_2^{(1)}$: elle est totalement décomposée dans $K_1 \subset K_2$ et les q places qui sont ainsi définies dans K_2 sont toutes rationnelles et se répartissent en $q-1$ places dans un ensemble $S_2^{(2)}$ et une unique place dans un ensemble $S_3^{(2)}$ sur lequel on itère le processus.
- les $q-1$ places de l'ensemble $S_1^{(1)}$ ont un comportement moins facile à décrire, mais on sait qu'elles se décomposent en des places de différents degrés (pas seulement rationnelles !). Le même phénomène se produit à l'étage suivant pour les $q-1$ places de l'ensemble $S_2^{(2)}$ et ainsi de suite. Les places d'un ensemble $S_t^{(j)}$ retrouvent un comportement régulier à partir du moment où on atteint un étage K_j où $j \geq 2t$: dès l'extension $K_j \subset K_{j+1}$, toutes les places de l'ensemble $S_t^{(j)}$ sont alors totalement ramifiées (il y a donc exactement une place dans K_{j+1} provenant de chacune des places de $S_t^{(j)}$).

On peut se représenter les différents types de décomposition des places dans les étages de la tour de la façon visuelle suivante (seules les "code places" ne sont pas représentées ici) :



3. Exemple

Si l'on teste la fonction obtenue avec les paramètres suivants :

```
> q := 2 ; l := 3 ;
> FonctionConstructMatrixA(q,l : Verbose := true, Timings := true) ;
```

Alors voilà ce que Magma renvoie :

```
Omega* = {@ 1 @}
indexCP = {@ delta, delta^2 @}
Place  l'infini dans K_0 :
 P0(1/x0) = (1/$.1)
Place P_0 dans K_0 :
 P0(x0) = ($.1)
Les codes places dans K_0 sont :

        P0(x0 - delta) = ($.1 + delta)
        P0(x0 - delta^2) = ($.1 + delta^2)
Places relatives  omega* dans K_0 :
 [ ($.1 + 1) ]

Cumulative time up to places in K0: 0.000



Place  l'infini dans K_3 :
 P3(1/x0) = (1/x0, (((x0^3 + x0^2 + x0 + 1)/x0^4*x1 + (x0^2 + 1)/x0^3)*x2 + (x0 + 1)/x0^2)*x3 + ((x0^2 + 1)/x0^3*x1 +
(x0 + 1)/x0^2)*x2)

Cumulative time up to totally ramified places in K_2: 0.390
Cumulative time up to totally ramified places in K_3: 1.050

1 place(s) relative(s)  omega* dans K_3 :
 [ (x0 + 1, (((x0 + 1)*x1 + (x0 + 1))*x2 + ((x0^3 + x0^2 + x0 + 1)*x1 + (x0^5 + x0^3 + x0 + 1)))*x3 + ((x0^8 + x0^3)*x1
+ (x0^7 + x0^5 + x0^3 + x0^2))*x2 + (x0^8 + x0^7 + x0^5 + x0^4 + x0^3 + x0)*x1 + x0^8 + x0^6 + x0^3 + x0^2) ]

Cumulative time up to totally split places in K_2: 1.080
Cumulative time up to totally split places in K_3: 3.780
Les 16 codes places de K_3 :
 [ (x0 + delta, (((x0^2 + 1)*x1 + (delta*x0^3 + delta*x0^2 + x0 + 1))*x2 + ((delta*x0^6 + delta^2*x0^5 + x0)*x1 + (x0^6
+ delta^2*x0^5 + delta*x0^4 + x0 + 1)))*x3 + ((delta^2*x0^9 + delta*x0^7 + delta^2*x0^6 + delta*x0^5 + delta*x0^3 +
delta^2*x0^2 + 1)*x1 + (delta*x0^10 + x0^9 + x0^8 + delta^2*x0^7 + delta*x0^5 + delta*x0^4 + x0^3 + delta*x0^2 +
delta^2*x0 + 1))*x2 + (x0^9 + delta*x0^8 + x0^6 + delta^2*x0^4 + delta^2*x0^3 + delta*x0)*x1 + delta*x0^10 + delta*x0^9
+ x0^7 + x0^5 + x0^4 + delta^2*x0^3 + x0^2 + delta^2*x0 + delta), (x0 + delta, ((delta^2*x0^2 + delta^2*x0)*x2 +
((delta^2*x0^5 + delta^2*x0^4 + delta*x0^2 + delta*x0)*x1 + (delta^2*x0^4 + x0^3 + delta*x0^2 + delta^2*x0 +
delta^2)))*x3 + ((delta^2*x0^10 + x0^8 + delta^2*x0^7 + delta*x0^6 + x0^5 + delta^2*x0^4 + x0^2 + delta^2*x0 +
delta^2)/x0^2*x1 + (delta^2*x0^11 + x0^8 + delta^2*x0^6 + x0^5 + delta^2*x0^4 + delta^2*x0^3 + delta^2*x0 +
delta^2)/x0^2)*x2 + (delta^2*x0^8 + delta*x0^7 + x0^6 + x0^5 + delta*x0^3 + x0^2 + delta)/x0*x1 + delta^2*x0^9 +
delta^2*x0^8 + delta*x0^6 + delta*x0^5 + x0^4 + delta*x0^2 + x0 + delta^2), (x0 + delta, (((delta*x0^2 + x0 +
delta^2)*x1 + (delta^2*x0^2 + delta^2))*x2 + ((x0^5 + delta^2*x0^4 + x0^3 + x0^2 + delta*x0)*x1 + (delta*x0^6 +
delta^2*x0^5 + delta^2*x0^4 + delta^2*x0^3 + x0^2 + x0 + 1)))*x3 + ((delta*x0^11 + delta*x0^10 + delta*x0^7 +
delta^2*x0^5 + delta^2*x0^4 + x0^3 + delta*x0^2 + delta^2*x0 + delta)/x0^2*x1 + (x0^11 + delta*x0^10 + delta^2*x0^9 +
delta^2*x0^8 + delta^2*x0^6 + delta*x0^4 + delta^2*x0^2 + delta^2*x0 + delta)/x0^2)*x2 + (delta*x0^10 + x0^9 +
delta*x0^8 + delta^2*x0^7 + delta*x0^5 + x0^4 + delta*x0^3 + x0^2 + x0 + delta^2)/x0*x1 + delta^2*x0^9 + delta*x0^8 +
delta*x0^7 + delta*x0^6 + x0^4 + delta*x0^3 + x0^2 + delta*x0 + 1), (x0 + delta, (((delta^2*x0^2 + x0 + delta)*x1 +
(delta*x0^3 + delta^2*x0^2 + delta^2*x0 + delta))*x2 + ((delta*x0^6 + x0^4 + x0^3 + x0^2 + delta*x0 + 1)*x1 +
(delta^2*x0^6 + delta^2*x0^4 + x0^3 + delta*x0 + delta^2)))*x3 + ((x0^11 + delta*x0^9 + x0^8 + delta^2*x0^7 + x0^6 +
delta^2*x0^5 + x0^4 + delta*x0^3 + x0^2 + 1)/x0^2*x1 + (delta*x0^12 + delta*x0^11 + delta^2*x0^10 + x0^8 + delta^2*x0^7
+ delta^2*x0^6 + delta^2*x0^5 + x0^4 + delta*x0^3 + delta^2*x0^2 + 1)/x0^2)*x2 + (delta^2*x0^10 + delta^2*x0^9 +
delta*x0^7 + delta^2*x0^6 + x0^4 + delta*x0^3 + delta^2*x0^2 + delta*x0 + delta^2)/x0*x1 + delta*x0^10 + delta^2*x0^9 +
delta^2*x0^7 + delta^2*x0^3 + delta^2*x0 + delta), (x0 + delta, (((delta*x0 + delta)*x1 + (delta*x0^2 + delta))*x2 +
((delta*x0^5 + delta*x0^4 + delta*x0^3 + delta^2*x0^2 + delta^2*x0 + delta)*x1 + (delta*x0^5 + delta*x0^4)))*x3 +
((delta*x0^8 + delta^2*x0^7 + x0^6 + delta^2*x0^4 + delta*x0^3 + x0^2 + delta*x0 + delta)/x0^2*x1 + (delta*x0^11 +
delta*x0^9 + delta^2*x0^8 + delta*x0^7 + delta*x0^6 + x0^5 + delta*x0^4 + delta*x0^3 + delta*x0^2 + delta*x0 +
delta)/x0^2)*x2 + (delta*x0^9 + delta^2*x0^6 + delta*x0^5 + delta^2*x0^3 + delta*x0^2 + delta^2*x0 + 1)/x0*x1 +
delta*x0^9 + delta^2*x0^6 + delta*x0^5 + delta*x0^3 + delta^2*x0^2 + delta*x0), (x0 + delta, (((delta*x0^2 + x0 +
delta^2)*x1 + (delta^2*x0^3 + delta^2))*x2 + ((delta^2*x0^6 + delta*x0^5 + delta^2*x0^4 + x0^3 + delta^2*x0^2 + delta*x0
+ delta)*x1 + (delta*x0^6 + x0^3 + delta^2)))*x3 + ((x0^9 + delta*x0^8 + delta^2*x0^7 + x0^6 + delta*x0^5 + x0^4 +
delta*x0^3 + delta*x0^2 + x0 + delta^2)*x1 + (delta^2*x0^10 + x0^9 + delta*x0^8 + x0^6 + x0^5 + delta*x0^3 +
delta*x0))*x2 + (delta*x0^10 + delta*x0^9 + delta*x0^8 + x0^7 + delta^2*x0^6 + delta^2*x0^3 + delta^2*x0^2 + delta^2*x0
+ delta^2)/x0*x1 + delta^2*x0^10 + delta*x0^8 + delta*x0^7 + x0^6 + delta*x0^5 + delta^2*x0^3 + delta), (x0 + delta,
(((delta^2*x0^2 + delta*x0 + 1)*x1 + (delta*x0^3 + delta^2*x0 + 1))*x2 + ((delta*x0^6 + delta^2*x0^5 + delta*x0^4 +
delta*x0^3 + delta^2*x0 + delta)*x1 + (delta^2*x0^6 + delta^2*x0^5 + delta^2*x0 + delta^2)))*x3 + ((x0^11 + delta*x0^9 +
delta^2*x0^8 + delta*x0^7 + delta*x0^6 + delta*x0^5 + delta^2*x0^4 + x0^2 + delta*x0 + delta)/x0^2*x1 + (delta*x0^12 +
x0^11 + delta^2*x0^10 + delta^2*x0^9 + x0^5 + delta^2*x0^3 + x0^2 + delta*x0 + delta)/x0^2)*x2 + (delta^2*x0^10 +
delta^2*x0^7 + delta*x0^6 + delta^2*x0^5 + delta^2*x0^4 + x0^2 + x0 + delta)/x0*x1 + delta*x0^10 + delta^2*x0^7 +
```

```
delta*x0^6 + x0^5 + delta^2*x0^4 + delta^2*x0^3 + x0^2 + x0 + delta), (x0 + delta, ((delta*x0^3 + x0^2 + delta^2*x0)*x2
+ ((delta*x0^6 + x0^5 + delta^2*x0^4 + delta^2*x0^2 + delta^2*x0)*x1 + (delta*x0^5 + x0^4 + delta^2)))*x3 +
((delta*x0^11 + delta^2*x0^10 + delta*x0^9 + delta*x0^8 + delta*x0^7 + delta*x0^5 + delta*x0^3 + delta^2)/x0^2*x1 +
(delta*x0^12 + delta^2*x0^11 + delta*x0^9 + delta*x0^7 + x0^6 + delta^2*x0^4 + delta^2*x0^3 + delta^2*x0^2 +
delta^2)/x0^2)*x2 + (delta*x0^8 + delta^2*x0^7 + delta^2*x0^6 + delta^2*x0^5 + delta^2*x0^4 + delta^2*x0^3 + x0^2 +
delta*x0 + delta)*x1 + delta*x0^10 + x0^9 + delta^2*x0^8 + x0^6 + delta*x0^4 + delta^2*x0^3 + delta^2*x0), (x0 +
delta^2, (((x0^2 + delta^2*x0 + delta)*x1 + (x0^3 + delta^2*x0 + delta))*x2 + ((x0^6 + x0^5 + x0^4 + delta^2*x0^3 +
delta^2*x0 + 1)*x1 + (x0^6 + delta^2*x0^5 + delta*x0^3 + delta*x0^2 + delta^2*x0 + 1)))*x3 + ((delta*x0^10 + x0^9 + x0^8
+ x0^7 + x0^6 + delta^2*x0^5 + delta*x0^4 + x0^3 + delta*x0^2 + delta^2*x0 + delta^2)/x0^2*x1 + (x0^12 + x0^10 +
delta^2*x0^9 + delta*x0^8 + delta*x0^4 + x0^3 + delta*x0^2 + delta^2*x0 + delta^2)/x0^2)*x2 + (x0^10 + delta*x0^9 +
delta*x0^8 + x0^7 + delta*x0^6 + delta^2*x0^5 + delta*x0^4 + delta^2*x0^3 + delta^2*x0 + delta^2)/x0*x1 + x0^10 +
delta*x0^8 + x0^7 + delta*x0^6 + delta*x0^5 + delta^2*x0^4 + x0^3 + x0^2 + x0 + delta), (x0 + delta^2, (((delta*x0^2 +
delta)*x1 + (x0^3 + delta^2*x0^2 + delta))*x2 + ((x0^6 + x0^5 + delta*x0^4 + delta*x0)*x1 + (delta*x0^6 + delta^2*x0^5 +
delta^2*x0^4 + x0^2 + delta^2*x0)))*x3 + ((delta^2*x0^11 + delta*x0^10 + x0^9 + delta^2*x0^8 + delta*x0^7 + delta^2*x0^6
+ x0^5 + delta*x0^4 + delta^2*x0^3 + x0^2 + x0 + delta)/x0^2*x1 + (x0^12 + delta*x0^10 + delta^2*x0^9 + delta*x0^8 +
x0^7 + delta^2*x0^6 + delta^2*x0^4 + delta^2*x0^3 + delta^2*x0^2 + x0 + delta)/x0^2)*x2 + (delta*x0^9 + x0^8 +
delta*x0^7 + delta^2*x0^5 + delta^2*x0^4 + delta^2*x0^2 + delta^2*x0 + 1)*x1 + x0^10 + delta^2*x0^9 + delta*x0^8 +
delta*x0^7 + delta*x0^6 + delta*x0^5 + delta*x0^4 + delta*x0^3 + x0^2 + delta^2*x0), (x0 + delta^2, (((x0 + 1)*x1 +
(x0^2 + 1))*x2 + ((x0^5 + x0^4 + x0^3 + delta^2*x0^2 + delta^2*x0 + 1)*x1 + (x0^5 + x0^4 + delta*x0^3 + x0^2 +
delta^2*x0)))*x3 + ((delta^2*x0^8 + delta*x0^7 + x0^6 + delta*x0^5 + delta*x0^4 + delta^2*x0^3 + delta*x0 + 1)/x0^2*x1 +
(x0^11 + x0^9 + delta^2*x0^8 + x0^7 + x0^6 + delta*x0^5 + delta^2*x0^4 + delta^2*x0^3 + delta*x0^2 + delta*x0 +
1)/x0^2)*x2 + (x0^9 + delta*x0^7 + delta*x0^6 + delta^2*x0^4 + delta*x0^3 + x0^2 + delta*x0 + delta^2)/x0*x1 + x0^9 +
delta^2*x0^6 + delta*x0^5 + delta*x0^2), (x0 + delta^2, (((delta*x0^2 + delta)*x1 + (delta^2*x0^3 + delta*x0^2 +
delta^2*x0 + delta))*x2 + ((delta^2*x0^6 + x0^4 + delta*x0^2 + delta*x0 + delta)*x1 + (delta*x0^6 + x0^5 + delta*x0^4 +
delta^2*x0^3 + delta*x0^2 + x0 + 1)))*x3 + ((x0^11 + x0^10 + delta^2*x0^9 + delta*x0^8 + delta*x0^7 + delta*x0^6 + x0^3
+ delta*x0^2 + x0 + delta^2)/x0^2*x1 + (delta^2*x0^12 + delta^2*x0^11 + delta*x0^10 + x0^9 + delta^2*x0^8 + x0^7 +
delta*x0^6 + x0^5 + delta^2*x0^4 + delta^2*x0^2 + x0 + delta^2)/x0^2)*x2 + (delta*x0^10 + delta^2*x0^9 + x0^8 +
delta^2*x0^5 + x0^3 + delta^2*x0^2 + delta^2*x0 + delta)/x0*x1 + delta^2*x0^10 + delta*x0^9 + x0^8 + delta*x0^7 +
delta*x0^6 + delta^2*x0^5 + delta*x0^4 + x0^3 + delta^2*x0), (x0 + delta^2, (((delta*x0^2 + delta)*x1 + (delta*x0^3 +
delta^2*x0^2 + delta^2*x0 + delta))*x2 + ((delta*x0^6 + x0^5 + x0^4 + delta^2*x0^2 + 1)*x1 + (delta*x0^6 + delta^2*x0^4
+ delta^2*x0^3 + delta^2*x0^2 + delta^2*x0 + delta)))*x3 + ((x0^8 + delta*x0^7 + delta^2*x0^6 + delta^2*x0^5 +
delta*x0^4 + x0^3 + delta^2*x0^2 + delta^2)*x1 + (delta*x0^10 + delta^2*x0^9 + delta*x0^8 + delta*x0^6 + delta^2*x0^5 +
delta^2*x0^4 + x0^2 + delta*x0 + delta))*x2 + (delta*x0^10 + delta*x0^9 + x0^8 + delta*x0^6 + delta^2*x0^5 + delta*x0^4
+ delta^2*x0^3 + delta^2*x0^2 + delta^2*x0 + 1)/x0*x1 + delta*x0^10 + delta^2*x0^9 + x0^8 + delta*x0^7 + x0^6 +
delta*x0^5 + delta*x0^4 + delta*x0^3 + x0^2), (x0 + delta^2, (((delta^2*x0^2 + delta*x0 + 1)*x1 + (delta^2*x0^3 +
delta*x0^2 + 1))*x2 + ((delta^2*x0^6 + x0^5 + x0^4 + delta*x0^3 + x0^2 + delta*x0 + delta)*x1 + (delta^2*x0^6 +
delta*x0^5 + delta*x0^4 + delta*x0^3 + x0)))*x3 + ((delta^2*x0^10 + delta^2*x0^9 + x0^7 + delta*x0^6 + x0^5 + x0^4 +
delta*x0^2 + 1)/x0^2*x1 + (delta^2*x0^12 + delta*x0^11 + delta^2*x0^10 + delta*x0^9 + delta*x0^8 + x0^7 + delta*x0^6 +
delta^2*x0^5 + delta^2*x0^3 + 1)/x0^2)*x2 + (delta^2*x0^10 + x0^9 + delta^2*x0^8 + delta*x0^7 + delta*x0^6 + delta*x0^5
+ delta*x0^4 + x0^3 + delta^2*x0^2 + delta^2)/x0*x1 + delta^2*x0^10 + delta*x0^9 + delta^2*x0^8 + delta^2*x0^7 +
delta*x0^6 + delta^2*x0^4 + x0^3 + delta*x0^2), (x0 + delta^2, (((delta*x0^2 + delta^2*x0 + 1)*x1 + (delta^2*x0^2 +
delta*x0 + 1))*x2 + ((x0^5 + delta^2*x0^4 + delta^2*x0^3 + x0^2 + delta^2*x0 + delta^2)*x1 + (delta*x0^6 + x0^5 +
delta^2*x0^4 + delta*x0 + delta)))*x3 + ((delta*x0^11 + x0^8 + x0^7 + delta*x0^6 + delta^2*x0^4 + delta^2*x0^3 +
delta^2*x0 + delta^2)/x0^2*x1 + (x0^11 + delta*x0^10 + x0^9 + x0^8 + delta^2*x0^7 + delta^2*x0^6 + delta*x0^4 + x0^2 +
delta^2*x0 + delta^2)/x0^2)*x2 + (delta*x0^10 + delta^2*x0^9 + delta*x0^7 + delta*x0^4 + delta^2*x0 + delta)/x0*x1 +
delta^2*x0^9 + delta*x0^7 + x0^6 + x0^5 + delta^2*x0^4 + delta*x0^2 + delta*x0), (x0 + delta^2, (((delta^2*x0^2 + x0 +
delta)*x1 + (delta^2*x0^3 + delta*x0^2 + delta^2*x0 + delta))*x2 + ((delta^2*x0^6 + x0^5 + x0^4 + x0^3 + delta*x0)*x1 +
(delta^2*x0^6 + x0^5 + delta*x0^4 + delta^2*x0^3 + delta^2*x0^2 + delta*x0 + delta)))*x3 + ((delta^2*x0^8 + delta*x0^7 +
x0^5 + delta^2*x0^3 + delta^2*x0^2 + x0 + 1)/x0*x1 + (delta^2*x0^11 + delta*x0^10 + delta^2*x0^9 + x0^8 + delta^2*x0^6 +
delta*x0^5 + delta*x0^4 + delta^2*x0^2 + delta*x0 + 1)/x0)*x2 + (delta^2*x0^10 + delta*x0^9 + delta^2*x0^6 + delta*x0^5
+ delta^2*x0 + delta^2)/x0*x1 + delta^2*x0^10 + delta*x0^9 + delta^2*x0^7 + delta*x0^5 + delta^2*x0^4 + delta^2*x0^3 +
x0^2 + delta*x0 + 1) ]

Cumulative time up to places above P_0 in K_3: 3.810
Cumulative time up to places above P_0 in K_3: 4.300

Les 6 places de K_3 au dessus de P_0 sont :
 [ (x0, ((x0 + 1)*x1 + (x0 + 1))*x3 + ((x0^2 + 1)/x0^2*x1 + (x0^7 + x0^4 + x0^3 + 1)/x0^2)*x2 + (x0^4 + 1)*x1 + x0^5 +
x0^3 + x0^2 + x0), (x0, (((delta*x0 + delta)*x1 + (x0^3 + delta^2*x0 + delta))*x2 + ((x0^6 + x0^4 + delta*x0^3 +
delta^2*x0^2 + delta^2*x0 + delta)*x1 + (delta^2*x0^5 + delta*x0^3 + delta^2*x0^2 + x0 + delta^2)))*x3 + ((x0^11 +
delta^2*x0^10 + x0^9 + x0^8 + delta*x0^7 + delta^2*x0^6 + delta^2*x0^5 + delta*x0^4 + delta*x0^3 + delta*x0^2 +
delta^2*x0 + 1)/x0^2*x1 + (x0^12 + x0^11 + delta^2*x0^9 + delta^2*x0^7 + delta*x0^5 + delta^2*x0^3 + delta^2*x0^2 +
delta^2*x0 + 1)/x0^2)*x2 + (delta^2*x0^8 + delta^2*x0^7 + x0^6 + delta*x0^5 + delta^2)*x1 + x0^10 + delta^2*x0^8 +
delta^2*x0^6 + x0^5 + delta*x0^4 + delta*x0^3 + delta*x0^2 + delta^2), (x0, ((x0 + 1)*x1 + (x0 + 1))*x3 + ((x0^2 +
1)/x0^2*x1 + (x0^7 + x0^4 + x0^3 + 1)/x0^2)*x2 + (x0^4 + 1)*x1 + x0^5 + x0^3 + x0^2 + delta^2), (x0, ((x0 + 1)*x1 + (x0
+ 1))*x3 + ((x0^2 + 1)/x0^2*x1 + (x0^7 + x0^4 + x0^3 + 1)/x0^2)*x2 + (x0^4 + 1)*x1 + x0^5 + x0^3 + x0^2 + delta), (x0,
(((delta^2*x0 + delta^2)*x1 + (x0^2 + delta*x0 + delta^2))*x2 + ((x0^5 + x0^4 + delta^2*x0^3 + delta^2*x0^2 + x0 + 1)*x1
+ (delta^2*x0^5 + x0^4 + delta*x0^3 + x0^2 + x0)))*x3 + ((delta*x0^10 + x0^8 + x0^6 + delta*x0^5 + delta*x0^3 + x0^2 +
delta^2)/x0^2*x1 + (x0^11 + delta^2*x0^9 + x0^8 + x0^7 + x0^6 + delta*x0^5 + x0^4 + delta^2*x0^2 + delta^2)/x0^2)*x2 +
```

```
(delta^2*x0^9 + delta*x0^8 + delta*x0^6 + delta*x0^3 + delta^2*x0^2 + delta^2*x0 + 1)/x0*x1 + x0^9 + delta*x0^8 +
delta^2*x0^6 + x0^5 + delta^2*x0^3 + delta^2*x0^2 + 1), (x0, (((delta*x0 + delta)*x1 + (x0^2 + delta^2*x0 + delta))*x2 +
((x0^5 + x0^4 + delta*x0^3 + delta*x0^2)*x1 + (delta*x0^5 + x0^4 + delta^2*x0^3 + delta*x0^2 + delta^2*x0 + 1)))*x3 +
((delta^2*x0^10 + x0^8 + delta^2*x0^7 + delta*x0^6 + delta^2*x0^5 + delta^2*x0^4 + x0^3 + delta*x0^2 + delta^2*x0 +
delta^2)/x0^2*x1 + (x0^11 + delta*x0^9 + x0^8 + x0^6 + delta^2*x0^5 + delta*x0^3 + delta*x0^2 + delta^2*x0 +
delta^2)/x0^2)*x2 + (delta*x0^9 + delta^2*x0^8 + delta^2*x0^5 + x0^3 + x0^2 + x0 + delta^2)/x0*x1 + x0^9 + delta^2*x0^8
+ delta*x0^6 + x0^5 + delta^2*x0^3 + delta^2*x0^2 + x0 + delta^2) ]


Temps d'execution pour q = 2 et 3 etages au dessus de K_0 : 4.310
Place a l'infini dans K_3 :
 P3(1/x0) = (1/x0, (((x0^3 + x0^2 + x0 + 1)/x0^4*x1 + (x0^2 + 1)/x0^3)*x2 + (x0 + 1)/x0^2)*x3 + ((x0^2 + 1)/x0^3*x1 +
(x0 + 1)/x0^2)*x2)
```

4. Première étape : stockage des places dans un fichier

La première évolution apportée au code consiste en un bloc rajouté à la fin du programme permettant de sauvegarder les places dans un fichier.

Une possibilité aurait été d'utiliser la fonction `save` de Magma. Cette fonction permet, en temps normal, de sauvegarder tout l'environnement courant dans un fichier, qui peut ensuite être chargé dans Magma à l'aide de la fonction `restore`. Malheureusement, cette fonctionnalité ne semble pas fonctionner et ne nous a pas permis de recharger les places désirées, d'où ce projet.

La solution choisie a été d'écrire, depuis le script initial, dans un fichier des commandes Magma permettant de reconstruire les places. Ainsi, le fichier généré peut être chargé dans Magma en utilisant la fonction native `load` qui permet d'exécuter du code Magma, sans qu'on ait besoin de coder une fonction de parsing.

La partie "stockage" du code commence comme ci-dessous :

```
file := "memoryStorageFile";

s := Sprintf("q := \%o;\nl:= \%o;",q,l);

fprintf file, s;

s := "\nVerbose := false;\nTimings := false;\nk<delta> := FiniteField(q^2) ;\n\n
    //////////////////////////////////////////////////////\n// DEFINITION
DES ETAGES K_0, ..., K_l DE LA TOUR //\n//////////////////////////////////////////////////////\n\nK0<x0>
    := FunctionField(k) ;\nPol<T> :=
PolynomialRing(K0) ;\nK1<x1> := FunctionField(T^q+T-x0^q/(x0^(q-1)+1) : Check := false) ;\nPol<T> :=
    ChangeRing(Pol, K1) ;
\n\nK := [K1] ;\nx := AssociativeArray(Integers()) ;\nx[0] := x0 ;\nx[1] := x1 ;\n\nfor i in [2..l] do\
    n Insert(~K,i,FunctionField(T^q
+T-x[i-1]^q/(x[i-1]^(q-1)+1) : Check := false)) ;\n AssignNames(~K[i],[\"x\" cat IntegerToString(i)]);\
    n x[i] := K[i].1 ;\n
Pol<T> := ChangeRing(Pol, K[i]) ;\nend for ;\n\n\n////////////////////////////////////////////////////\n//
    DEFINITION DES ENSEMBLES omega*
ET k\omega //\n////////////////////////////////////////////////////\n\nomegax := {@ k| @} ; // ensemble des
     elements de omega*\nindexCP := {@
k| @} ; // ensemble des elements (non-nuls) de k\omega\nfor x in Exclude(Set(k),Zero(k)) do\n if (x^(q
    -1)+1 eq Zero(k)) then
\n Include(~omegax, x) ;\n else\n Include(~indexCP, x) ;\n end if ;\nend for ;\n// affichage\nif
    Verbose then\nprintf
\"Omega* = \%\%o\n\", omegax ;\n printf \"indexCP = \%\%o\n\", indexCP ;\nend if ;\n\n
    ////////////////////////////////////////////////////\n//
DEFINITION DES PLACES RATIONNELLES DE K_0 //\n////////////////////////////////////////////////////\n\n//
    DEFINITION DE LA PLACE A
L INFINI DANS K_0\nPoo0 := Zeros(1/x0)[1] ;\n// affichage\nif Verbose then printf \"Place a l infini
    dans K_0 :\n P0(1/x0) = \%
\%o\n\", Poo0 ; end if ;\n\n//DEFINITION DE P_0 DANS K_0\nPo0 := Zeros(x0)[1] ;\n// affichage\nif
    Verbose then printf \"Place
P_0 dans K_0 :\n P0(x0) = \%\%o\n\", Po0 ; end if ;\n\n// DETERMINATION DES CODES PLACES DE K_0\nPc0 :=
[ Zeros(x0-a)[1] : a in indexCP ] ;\n// affichage\nif Verbose then\nprint \"Les codes places dans K_0
    sont :\n\" ;\nfor i in
[1..#indexCP] do\n printf \"\tP0(x0 - \%\%o) = \%\%o\n\", indexCP[i], Pc0[i] ;\nend for ;\nend if ;\n\n
    // DETERMINATION DES
PLACES RELATIVES A omega* DANS K_0\nPm0 := [ Zeros(x0-alpha)[1] : alpha in omegax ] ;\n\n";

fprintf file, s;
```

La fonction `fprintf file, s` permet d'écrire la chaîne de caractères `s` dans le fichier `file`.
La longue chaîne de caractères correspond au code original permettant de générer les élements initiaux (Ω, places de K_0). Ces éléments sont rapides à calculer et sont indispensables pour définir les places aux niveaux supérieurs.

```
Oi := MaximalOrderInfinite(K[l]);
Of := MaximalOrderFinite(K[l]);

fprintf file, "Oi := MaximalOrderInfinite(K[l]);\n";
fprintf file, "Of := MaximalOrderFinite(K[l]);\n\n";
```

Ces quelques lignes de codes sont essentielles à la reconstruction des places. Pour chaque place à sauvegarder, on enregistre dans le fichier de stockage une instruction Magma qui crée une nouvelle place étant donnés :

— L'anneau auquel cette place est relative (c'est toujours soit `MaximalOrderFinite(`K_l`)`, soit `MaximalOrderInfinite(`K_l`)`, appelés par la suite `Of` et `Oi`)
— Les générateurs de cette place

On a donc besoin des ordres maximaux du corps considérés dans le fichier de calcul et dans celui de restoration pour respectivement savoir avec quel ordre on doit écrire l'instruction de création de la place restorée, et pour que cette instruction connaisse l'anneau dont elle a besoin. Puisque chaque place est liée soit à l'un, soit à l'autre des deux ordres, on charge les deux en mémoire dès le début du traitement.

Ci-dessous, l'exemple du stockage d'une place pour `Pm_l`. Le code est sensiblement le même pour toutes les autres places.

```
// Chargement de Pm_l

fprintf file, "\n\n";
fprintf file, "Pm_l := [ Places(K[3],1) | ];\n\n";

for p in Pm_l do
        I := Ideal(p);
        B := Generators(I);
        s := &cat Split(Sprintf("\%o",B), "\n");

        fprintf file, "p := Place(ideal<";

        if Order(I) eq Oi
                then fprintf file, "Oi";
                else fprintf file, "Of";
        end if;

        fprintf file, ("|" cat s cat ">);\n");
        fprintf file, "Append(~Pm_l,p);\n";
end for;
```

L'instruction `Pm_l := [ Places(K[3],1) | ];";` permet d'indiquer à Magma que `Pm_l` est une collection, pour le moment vide, d'éléments vivant dans l'ensemble `Places(K[3],1) |]`, c'est à dire les places rationnelles de `K[3]`. Elle est reprise du code original.

La ligne `s := &cat Split(Sprintf("\%o",B),"\n");` est particulièrement importante. La fonction `Sprintf` appliquée à l'argument `B` retourne la chaîne de caractères qui aurait été affichée par Magma si on lui avait donné l'instruction `B;`, en l'occurence, la valeur des générateurs de l'idéal associé à la place considérée. Les fonctions `cat` et `Split` permettent d'éliminer les sauts de ligne de cette expression. On obtient ainsi une expression qui est acceptable en Input pour Magma. On effectue ensuite un test pour savoir lequel de `Oi` ou de `Of` est l'ordre associé à la place considérée. Une fois que Magma connaît l'ordre associé à la place ainsi que les générateurs de l'idéal associé à la place, on utilise la fonction `Place` pour la reconstruire.

Cette méthode de stockage fonctionne et le fichier ainsi créé permet de recréer les places calculées dans l'environnement de Magma.

5. Deuxième étape : division du calcul et du stockage en blocs

En raison de problèmes de mémoire expérimentés lors du calcul des places - le nombre de places augmentant avec la hauteur de la tour, la mémoire utilisée augmente également, jusqu'à saturation -, M. Rambaud et Mme. Pieltant m'ont demandé de séparer ce calcul en blocs, tout en nettoyant la mémoire entre les blocs, ce qui nécessite pour garder une trace des blocs déjà calculés de sauvegarder les blocs de places immédiatement après les avoir calculés, et juste avant de les supprimer de l'environnement.

La séparation en blocs est surtout significative pour les places au-dessus de P_0, dont le nombre augmente rapidement avec la hauteur de la tour. Le principe est le suivant : on considère les places au dessus de P^0_∞, les places au-dessus de $\{P^0_\alpha : \alpha \in \Omega^*\}$, et les "code places" i.e. les places au-dessus de $\{P^0_\alpha : \alpha \in \mathbb{F}_{q^2} \setminus \Omega\}$ comme constituant trois blocs distincts.

Le comportement des places au-dessus de P^0_0 est plus complexe (voir page 5). A chaque étage de la tour, une nouvelle branche de successeurs de P^0_0 apparaît. Chacune de ces branches est traitée comme un groupe distinct.

La solution utilise deux programmes : un programme Initialisation, à utiliser une fois. Initialisation calcule les places de K_l en partant de K_0 sur le même modèle que le programme original. Il faut donc choisir un l raisonnable en espace mémoire utilisé. Ce programme sauvegarde chaque bloc dans un fichier, ainsi que le nombre de blocs relatifs aux places au-dessus de P^0_0.

Le programme Iteration reconstruit, pour chaque bloc, les places de l'étage l et s'en sert pour calculer celles de l'étage $l+1$, après quoi il nettoie la mémoire (à l'aide de la fonction `delete` et passe au bloc suivant. Les places de l'étage $l+1$ sont stockées dans des fichiers exactement de la même façon que par le programme Initialisation. On peut donc réutiliser le même programme Iteration - après avoir modifié la valeur de l dans le code - pour calculer les places de l'étage $l+2$.

Nous allons expliquer l'algorithme et le code permettant de calculer les places au-dessus de P^0_0 à l'étage $l+1$ à partir de celles à l'étage l.

1 Charger le fichier `GrpNb_l` contenant le nombre de groupes de l'étage l. ;
2 Calculer les places correspondant à la nouvelle branche. ;
3 Stocker le groupe dans un nouveau fichier `GrpNb_l+1`.;
4 **for** $i \leftarrow 1$ **to** $GrpNb_l$ **do**
5 Charger le fichier du groupe i.;
6 Calculer les décompositions dans K_l des places du groupe i.;
7 Stocker les places obtenues dans le nouveau fichier du groupe i.;
8 **end**
9 Incrémenter $GrpNb$ et le stocker dans le fichier `GrpNb_l+1`;

Algorithme 1 : Algorithme permettant de calculer les places de l'étage $l+1$ à partir de celles de l'étage l et de les stocker dans des fichiers

Plusieurs problèmes ont émergé lors de l'implémentation de cet algorithme en langage Magma. Tout d'abord, le chargement d'un fichier se fait à l'aide de la fonction `load filename`, où `filename` est décrit dans le manuel de référence comme étant une chaîne de caractères. Dans la ligne 1 de l'algorithme précédent, on souhaite charger le fichier `GrpNb_l`, or le fichier ne s'appelle pas littéralement `GrpNb_l` mais bien `GrpNb_2`, `GrpNb_3`, ... selon la valeur de l. On souhaiterait donc charger le fichier ainsi :

```
filename := "GrpNb_" cat IntegerToString(l);
load filename;
```

ou encore directement

```
load "GrpNb_" cat IntegerToString(l);
```

Malheureusement, aucune de ces implémentations ne fonctionne : Magma exige que le nom du fichier soit écrit directement par l'utilisateur dans l'instruction, et non calculé en fonction d'un paramètre (ici l).

On a contourné la difficulté de la façon suivante : au lieu de charger directement `GrpNb_l`, on crée un fichier temporaire `tmpFile` dans lequel on écrit , à l'aide de la fonction `fprintf`, comme suit :

```
fprintf file, "load \"GrpNb_" cat IntegerToString(l-1) cat "\";";
```

Le fichier temporaire `tmpFile` contient donc l'instruction `load GrpNb_3`, si $l == 3$ par exemple. On peut donc, dans notre programme principale, charger indirectement le nombre de groupes à l'aide de l'instruction `load tmpFile;`.

Le code correspondant à ce procédé est ci-dessous :

```
file := "files/tmpFile";
s := "load \"files/GrpNb_" cat IntegerToString(l-1) cat "\";";
fprintf file, s;

load "files/tmpFile";
Write(file, "" : Overwrite := true);
```

On souhaite, bien sûr, réutiliser le même code pour la ligne 5 de l'algorithme 1. On a en effet le même problème puisque le nom du fichier qu'on souhaite charger dépend de la variable i. Mais il est impossible dans Magma d'utiliser la fonction `load` au sein d'une boucle `for` (ce qui n'est indiqué que par une erreur "invalid syntax" et qui n'est pas documenté dans le manuel de référence). La solution choisie pour contourner le problème et implémenter la boucle `for` de l'algorithme 1 est la suivante : on crée un fichier, nommé `loopFile`, dans lequel on va "dérouler" notre boucle for, c'est à dire écrire (à l'aide d'une autre boucle) le code de chacune des itérations. On commence donc par écrire le code comme s'il n'y avait aucun problème avec la fonction `load` :

```
Poz_l := [];
```

```
for i in [1..GrpNb] do

        file := "tmpFile";
        s := "load \"files/Poz_temp_" cat IntegerToString(i) cat "_" cat IntegerToString(l-1) cat "\";";
        fprintf file, s;
        load tmpFile;
        Write(file, "" : Overwrite := true);
        Poz_grp := [];
        pos := 0;
        for p in Poz_temp_elt do
                Stemp := Decomposition(K[l], p);
                Insert(~Poz_grp, pos+1, pos+1, Stemp);
                pos := #Poz_grp;
                delete Stemp;
        end for;

        // ecriture du nouveau groupe dans un fichier

        file := "files/Poz_temp_" cat IntegerToString(i) cat "_" cat IntegerToString(l);

        fprintf file, "Poz_temp_elt := [ ];\n\n";

        for p in Poz_grp do
                I := Ideal(p);
                B := Generators(I);
                s := &cat Split(Sprintf("\%o",B), "\n");

                fprintf file, "print \"new place\";\n";
                fprintf file, "p := Place(ideal<";

                if Order(I) eq nOi
                        then fprintf file, "Oi";
                        else fprintf file, "Of";
                end if;

                fprintf file, ("|" cat s cat ">);\n");
                fprintf file, "Append(~Poz_temp_elt,p);\n";
        end for;
        delete Poz_grp;
end for;
```

On transforme ensuite ce code en une chaîne de caractères que l'on pourra écrire dans un fichier : en s'occupant notamment des caractères spéciaux comme % et \, et en remplaçant les saut de ligne par des `\n`. On n'oublie pas, en écrivant la chaîne de caractères dans le fichier `loopFile`, de remplacer les apparitions de la variable i par sa valeur. On obtient donc le code final suivant :

```
Poz_l := [];

file := "files/loopFile";

for i in [1..GrpNb] do

        s1 := "file := \"tmpFile\";\n   s := \"load \\\"files/Poz_temp_\" cat IntegerToString(";
        s2 := ") cat \"_\" cat IntegerToString(l-1) cat \"\\\";\";\n    fprintf file, s;\n      load
        tmpFile;\n      Write(file, \"\" : Overwrite := true);\n        Poz_grp := [];\n        pos := 0;\n
                for p in Poz_temp_elt do\n              Stemp := Decomposition(K[l], p);\n
                Insert(~Poz_grp, pos+1, pos+1, Stemp);\n                pos := #Poz_grp;\n
                        delete Stemp;\n end for;
                        \n      \n      // ecriture du nouveau groupe dans
                        un fichier\n    \n      file :=
                         \"files/Poz_temp_\" cat IntegerToString(";
        s3 := ") cat \"_\" cat IntegerToString(l);\n
                \n      fprintf file, \"Poz_temp_elt := [ ];\\n
        \\n\"; \n       \n      for p in Poz_grp do\n           I := Ideal(p);\n
        B := Generators(I);\n           s := &cat Split(Sprintf(\"%%o\",B), \"\\n\");\n
```

```
        \n              fprintf file, \"print \\\"new place\\\";\\n\";\n
        fprintf file, \"p := Place(ideal<\";\n\n                if Order(I) eq nOi\n
                                then fprintf
        file, \"Oi\";\n                 else fprintf file, \"Of\";\n            end if;\n\n
        fprintf file, (\"|\" cat s cat \">);\\n\");\n           fprintf file, \"
        Append(~Poz_temp_elt,p);\\n\";\n        end for;\n      delete Poz_grp;\n";

        fprintf file, s1;
        fprintf file, IntegerToString(i);
        fprintf file, s2;
        fprintf file, IntegerToString(i);
        fprintf file, s3;
        fprintf file, "\n\n\n";


end for;

load "files/loopFile";
```

6. Bilan des performances

Si l'objectif du projet était de pouvoir recharger en mémoire les places calculées lors d'une session ultérieure, l'aspect des performances est également important. En effet, il est moins intéressant de recharger les places calculées à partir d'un fichier si cette reconstruction prend plus de temps que le calcul originel.

Lors de la première étape, on s'est rendu compte que l'instruction `Pm_l := [ Places(K[3],1) | ];` qui permet de déclarer `Pm_l` comme la liste vide tout en indiquant à Magma que cette liste va contenir des places rationnelles de `K[3]` occupait la plupart du temps de calcul pour $q = 2$ et $l \geq 3$. Une explication pourrait être que Magma évalue l'expression `Places(K[3],1)` avant d'en jeter le résultat. On a donc supprimé les instructions de cette forme, sauf lorsqu'elles sont utilisées avec `K[1]` ce qui prend alors très peu de temps, pour les remplacer par un simple `Pm_l := [ ];`.

Après avoir effectué ces modifications, on s'est rendu compte que la reconstruction des places à l'aide du stockage des générateurs et de l'ordre correspondant dans un fichier, et de la fonction `Place(ideal<...>);` prennait *plus* de temps que le calcul par décomposition des places de l'étage inférieur. Nous avons alors essayé de reconstruire les places en prenant l'intersection des listes définies par `Zeros(f)` et `Zeros(g);` où `f` et `g` sont le résultat de la fonction `TwoGenerators` appliquée à la place à reconstruire.
Cependant, le temps d'exécution de ce nouveau programme était encore bien supérieur à celui de la première reconstruction : cette piste a donc été abandonnée.

Quant à la version du programme qui stocke les places par groupes et qui nettoie la mémoire au fur et à mesure, son temps d'exécution est comparable à celui de la version originelle, ce qui est surprenant, car le procédé de reconstruction des places est le même que lors de la reconstruction en un seul bloc. Une hypothèse est que le nettoyage de la mémoire permet à Magma de procéder plus rapidement.

Table 1. Comparaison des temps d'exécution

	q = 2, l = 2	q = 2, l = 3	q = 2, l = 4	q = 3, l = 2	q = 3, l = 3
script original	0,140s	3,660s	878s	1,480s	2281s
reconstruction en un bloc	0,150s	7,040s	> 1h	3,050s	> 2h
reconstruction par blocs	0,140s	3,359s	850s	1,840s	2437s

Conclusion

Ce projet avait pour but de contourner un bug de Magma - celui de la fonction `restore` -, et nous avons tenté diverses approches afin d'y arriver. Cependant, nous nous sommes heurtés à de nouveaux bugs du logiciels, et nous n'avons pas réussi à comprendre pourquoi le gain de temps lorsqu'on reconstruit les places à partir de leurs générateurs est nul. Une évolution pourrait être de contacter le support de Magma afin de pouvoir éclairer ces points. L'apport de ce programme, qui calcule et sauvegarde les places du niveau l à partir du niveau $l - 1$, est donc surtout de nettoyer la mémoire au fur et à mesure.