CALCULS EN THÉORIE DES GROUPES, ET INTRODUCTION AU LANGAGE GAP (GROUPS, ALGORITHMS AND PROGRAMMING)

par

Jean Michel

1. Introduction

Les ordinateurs sont devenus indispensables pour calculer dans les groupes (finis essentiellement, mais aussi infinis) depuis une trentaine d'années. Une première période où ils ont joué un rôle important a été lors de la construction des groupes simples sporadiques, dans les années 70. La classification des groupes simples, qui s'est achevée au début des années 80, les classifie en un certain nombre de familles infinies, qui sont les groupes alternés, les groupes de Chevalley (les groupes de points rationnels sur un corps fini de groupes algébriques simples) et des groupes dérivés de constructions analogues (par Steinberg, Suzuki, Ree ...), plus 26 groupes dits « sporadiques ».

L'existence d'un certain nombre de ces derniers ne fut pendant longtemps prouvée que par leur construction explicite sur ordinateur. Citons ainsi le « bébé-monstre » (ou groupe F_2), d'ordre $2^{41} \cdot 3^{13} \cdot 5^6 \cdot 7^2 \cdot 11 \cdot 13 \cdot 17 \cdot 19 \cdot 23 \cdot 31 \cdot 47$, qui fut construit comme sous-groupe d'un groupe de permutations d'environ $13 \cdot 10^9$ lettres [**Sims80**], ou encore le groupe de Janko J_4, d'ordre $2^{21} \cdot 3^5 \cdot 5 \cdot 7 \cdot 11^3 \cdot 23 \cdot 31 \cdot 37 \cdot 43$, qui fut construit comme groupe de matrices 112×112 sur le corps à 2 éléments [**Norton**].

La théorie algorithmique des groupes se retrouve aussi au détour de questions telles que : peut-on faire tourner un seul coin du Rubik's cube ? Quel est le groupe de Galois du polynôme $x^8 + 2x^7 + 28x^6 + 1728x + 3456$? Quels sont les groupes de symétrie possibles des cristaux ?

La plupart des systèmes de calcul symboliques peuvent être utilisés pour certaines de ces questions, mais dans le domaine de la théorie des groupes deux systèmes prédominent, GAP

`http://www-gap.dcs.st-and.ac.uk/~gap/`

et MAGMA

`http://www.maths.usyd.edu.au:8000/u/magma.`

Ces deux systèmes sont précisément *non symboliques* en ce sens que, contrairement au systèmes symboliques où l'interprétation des symboles manipulés dépend du contexte et seules leurs règles de manipulation sont spécifiées, ici tous les objets et fonctions du langage ont un sens mathématique précis. Dans la suite, nous présenterons le système GAP qui est celui que l'auteur connaît le mieux, et qui a l'avantage d'être disponible gratuitement sur le Web, (en versions Mac, Windows et Unix) ainsi que ses sources (en langage C). Les exemples sont donnés pour la version 3.4.4 qui est une version stable depuis 1997, mais la plupart d'entre eux tourneront sans modification sur la version « expérimentale » 4.2 qui est aussi disponible sur le site indiqué ci-dessus.

2. Représentation des groupes en machine

Le problème de base est de représenter les éléments d'un groupe de façon à pouvoir multiplier deux éléments, et reconnaître leur égalité. La *théorie algorithmique abstraite des groupes* consiste à étudier divers algorithmes sur les groupes et à mesurer leur coût en prenant comme mesure juste le nombre effectué des opérations ci-dessus. Chaque représentation effective d'un groupe correspond à une réalisation mathématique particulière de ce groupe. Passons en revue les plus courantes :

2.1. Présentation par générateurs et relations. Une présentation de la forme $\langle \{x_i\}_{i\in I} \mid \{r_j\}_{j\in J}\rangle$, où les r_j sont des mots en les symboles x_i et leurs inverses x_i^{-1}, représente le quotient du groupe libre sur les générateurs $\{x_i\}_{i\in I}$ par le sous-groupe distingué engendré par les éléments r_j (c'est-à-dire le sous-groupe engendré par les r_j et tous leurs conjugués). Par exemple, le groupe symétrique $\mathfrak{S}_3$ sur trois lettres peut être décrit par la présentation $\langle x_1, x_2 \mid x_1^2, x_2^2, (x_1x_2)^3\rangle$ (où x_1 correspond à la transposition $(1,2)$ et x_2 à la transposition $(2,3)$), ce qui signifie qu'on peut manipuler ses éléments en prenant toutes les suites de symboles $x_1, x_1^{-1}, x_2, x_2^{-1}$ modulo les relations $x_1^2 = x_2^2 = (x_1x_2)^3 = 1$ (où 1 est représenté par la suite vide). En fait, dans cet exemple, les générateurs étant de carré 1, tout élément du groupe peut donc être représenté par une suite de x_1 et x_2 telle qu'aucun symbole ne soit répété deux fois consécutives, même après application de l'égalité $x_1x_2x_1 = x_2x_1x_2$ (qui exprime la troisième relation $(x_1x_2)^3 = 1$). En GAP cet exemple devient :

```
gap> f:=FreeGroup(2);
Group( f.1, f.2 )
gap> S3:=f/[f.1^2,f.2^2,(f.1*f.2)^3];
Group( f.1, f.2 )
gap> Size(S3);
6
gap> Elements(S3);
[ IdWord, f.1, f.2, f.1*f.2, f.2*f.1, f.1*f.2*f.1 ]
gap> Size(f);
"infinity"
```

La commande `Size` peut déterminer l'ordre du groupe s'il est fini et déterminer dans un certain nombre de cas si le groupe est infini (et dans les autres cas d'infinité se contentera de provoquer une réflexion infinie de la machine...). Le principal intérêt de la représentation par générateurs et relations est de pouvoir représenter des groupes infinis ; son principal inconvénient est son inefficacité. Outre la place prise en machine par les mots à représenter, certains algorithmes sont malaisés sous cette forme et sont trop longs sur des groupes de plus de quelque milliers d'éléments.

2.2. Groupes de matrices. Soit V un espace vectoriel sur un corps k. Une *représentation linéaire* ρ d'un groupe G dans V est un morphisme de groupes $\rho : G \to GL(V)$. Elle est *fidèle* si ρ est injectif. C'est une méthode particulièrement importante d'étude des groupes : de nombreux groupes sont connus par le biais d'une représentation fidèle (si G est simple et ρ non triviale, elle est fidèle car le noyau de ρ est un sous-groupe distingué), et de plus si k est de caractéristique 0 et G fini, une représentation est déterminée à isomorphisme près par son *caractère* qui est la fonction de classe (*i.e.* fonction qui ne dépend que de la classe de conjugaison) donnée sur G par $g \mapsto \mathrm{Trace}(\rho(g))$. La détermination de la *table des caractères*, c'est-à-dire la liste des caractères des *représentations irréductibles* (*i.e.* où l'espace V n'a pas de sous-espace propre non trivial stable sous l'action de G) est pour un spécialiste un pas important vers la compréhension de la structure d'un groupe. En pratique, sur ordinateur, on prendra pour k un corps dont les éléments sont susceptibles d'une représentation exacte en machine : le corps $\mathbb{Q}$ des nombres rationnels, ou un corps fini à q éléments $\mathbb{F}_q$. Les *corps cyclotomiques* (extension de $\mathbb{Q}$ par des racines de l'unité) jouent aussi un rôle important.

Poursuivons notre exemple du groupe symétrique : le groupe $\mathfrak{S}_3$ dans son action naturelle par matrices de permutation sur l'espace vectoriel de base e_1, e_2, e_3 laisse fixe le vecteur $e_1 + e_2 + e_3$, donc aussi l'espace orthogonal à ce vecteur. Dans cet espace orthogonal, dont nous prendrons pour base les vecteurs $e_1 - e_2$ et $e_2 - e_3$, le groupe $\mathfrak{S}_3$ admet une représentation fidèle où la transposition $(1,2)$ agit par la matrice $\begin{pmatrix} -1 & 0 \\ 1 & 1 \end{pmatrix}$ et la transposition $(2,3)$ par la matrice $\begin{pmatrix} 1 & 1 \\ 0 & -1 \end{pmatrix}$. En GAP, cela donne :

```
gap> S3:=Group([[-1,0],[1,1]],[[1,1],[0,-1]]);
Group( [ [ -1, 0 ], [ 1, 1 ] ], [ [ 1, 1 ], [ 0, -1 ] ] )
gap> Size(S3);
6
gap> Elements(S3);
[ [ [ -1, -1 ], [ 1, 0 ] ], [ [ -1, 0 ], [ 1, 1 ] ],
[ [ 0, -1 ], [ -1, 0 ] ], [ [ 0, 1 ], [ -1, -1 ] ],
[ [ 1, 0 ], [ 0, 1 ] ], [ [ 1, 1 ], [ 0, -1 ] ] ]
```

L'avantage des représentations matricielles est qu'un groupe est souvent donné sous cette forme. Sur $\mathbb{Q}$, cette méthode peut représenter des groupes infinis. L'inconvénient est une fois encore la difficulté des algorithmes sous cette forme. Toutefois, des progrès récents ont été faits. Il n'est pas évident à priori qu'une matrice sur $\mathbb{Z}$ est d'ordre fini ; cependant [**BBR**] ont mis au point un algorithme pour tester la finitude d'un groupe de matrices sur $\mathbb{Z}$. Notons toutefois que [**Mihailova**] a montré que l'appartenance à un groupe de matrices sur $\mathbb{Z}$ est indécidable en dimension ≥ 4. Beaucoup d'efforts portent en ce moment sur le problème de la reconnaissance d'un groupe donné comme groupe de matrices sur un corps fini : [**Aschbacher**] a classé les sous-groupes maximaux de $GL_n(\mathbb{F}_q)$ en 9 classes, et le problème est de déterminer un tel sous-groupe contenant le groupe donné, ou de montrer que le groupe donné est GL_n tout entier. De tels algorithmes ont été implantés en GAP, utilisant des statistiques sur l'ordre d'éléments aléatoires du groupe donné ([**CelLG**]).

2.3. Groupes de Permutations. Une représentation de permutation peut être vue comme un type particulier de représentation linéaire (par matrices de permutation). Tout groupe en possède (par exemple la représentation régulière, où le groupe est considéré comme groupe de permutations de ses éléments), mais il se peut qu'un groupe ne possède qu'une représentation de permutation sur un très grand nombre de points alors qu'il possède d'autres représentations plus maniables. Nous allons voir que ce type de représentations est très avantageux pour le calcul en machine car l'on dispose de nombreux algorithmes efficaces. Notre exemple de $\mathfrak{S}_3$ comme groupe de permutations se présente en GAP comme suit :

```
gap> S3:=Group((1,2),(2,3));
Group( (1,2), (2,3) )
gap> Size(S3);
6
gap> Elements(S3);
[ (), (2,3), (1,2), (1,2,3), (1,3,2), (1,3) ]
```

Comme on le voit sur la dernière ligne, les permutations sont représentées en GAP comme produit de cycles distincts.

2.4. Groupes polycycliques. Finalement mentionnons une classe de groupes qui a un intérêt « technique » en ce qu'ils se prêtent bien à la représentation par ordinateur. Un groupe est dit *polycyclique* s'il possède une suite de sous-groupes $G = G_0 \supsetneq G_1 \supsetneq \cdots \supsetneq G_n = 1$ tels que G_{i+1} soit distingué dans G_i et tels que G_i/G_{i+1} soit cyclique (pour les groupes finis cette notion est équivalent à demander que G soit résoluble. En général, elle est équivalente à demander qu'il existe n tel que G soit un sous-groupe résoluble de $GL_n(\mathbb{Z})$). Si a_i est un représentant dans G_i d'un générateur de G_i/G_{i+1}, alors tout élément de G possède une écriture unique de la forme $a_1^{e_1} \cdots a_n^{e_n}$ (dite « expression réduite » de l'élément). Le groupe G lui-même admet une présentation de générateurs a_i et relations obtenues en écrivant les expressions réduites de $a_i^{|G_i/G_{i+1}|}$ (de a_i^{-1} si G_i/G_{i+1} est infini) et de $a_j a_i$ pour $j < i$, dite « présentation pcp » (power-conjugate presentation). La méthode pour trouver l'expression réduite d'un élément est semblable à l'élimination gaussienne.

Les techniques de groupes polycycliques ont permis de traiter des groupes résolubles de grande taille, par exemple le groupe de Burnside $B(4,4)$ d'ordre 2^{422} (le plus grand groupe d'exposant 4 a 4 générateurs) [**AHN**].

3. Quelques algorithmes

3.1. L'algorithme de Todd-Coxeter. C'est un algorithme qui, étant donné un groupe donné par générateurs et relations, $G = \langle \{x_i\}_{i\in I} \mid \{r_j\}_{j\in J}\rangle$, et un sous-groupe H défini par la liste de ses générateurs $\{h_k\}_{k\in K}$ (de même que les r_j, les h_k sont des mots en les x_i et leurs inverses), énumère les classes $H\backslash G$. Le cas particulier $H = 1$ est évidemment très utile pour énumérer les éléments de G, mais le cas général l'est aussi car il se peut que $H\backslash G$ soit fini alors que H ne l'est pas.

L'algorithme consiste à remplir un certain nombre de tables, dont les colonnes sont indexées par des éléments de l'ensemble X des x_i et des x_i^{-1}, et les lignes, construites au fur et à mesure, sont indexées par des entiers qui représentent les classes $H\backslash G$. On démarre avec la convention que 1 représente la classe $H = H \cdot 1$, et à chaque fois qu'on considérera une nouvelle classe $H \cdot g$ on lui attribuera un nouvel entier.

Les tables sont : la table T de multiplication des classes ; pour chaque r_j la « table de relation » correspondante, $R^{(j)}$; et pour chaque générateur h_k, la « table de générateur » correspondante $H^{(k)}$.

T a ses colonnes indexées par les x_i suivis des x_i^{-1}, et pour $\varepsilon \in \{-1,1\}$ on aura $T_{j,x_i^\varepsilon} = k$ si et seulement si $j.x_i^\varepsilon = k$ (ici nous avons noté les classes par les entiers correspondants ; l'égalité ci-dessus, si j représente Hg et k représente Hg' signifie que $Hgx_i^{\varepsilon} = Hg'$).

Si $r_j = u_1 \cdots u_{l_j}$ est la décomposition de r_j en $u_k \in X$, les colonnes de $R^{(j)}$ sont indexées par $u_1, \ldots, u_{l_j}$ dans l'ordre, et on aura $R^{(j)}_{i,u_k} = l$ si et seulement si $i \cdot u_1 \cdots u_k = l$. Par commodité pour la description de l'algorithme, on supposera que $R^{(j)}$ a une colonne supplémentaire avant toutes les autres u_0 telle que $R^{(j)}_{i,u_0} = i$.

Enfin, si $h_k = u_1 \cdots u_{l_k}$ est la décomposition de h_k en $u_j \in X$, on construit la table $H^{(k)}$ exactement de la même façon que pour $R^{(j)}$, sauf que cette table n'aura qu'une seule ligne, indexée par l'entier 1 (qui représente la classe triviale $H \cdot 1$).

Au départ, T n'aura qu'une ligne indexée par 1 qui sera vide, les $R^{(j)}$ n'auront que la ligne 1 avec pour seules cases remplies $R^{(j)}_{1,u_0} = R^{(j)}_{1,u_{l_j}} = 1$, et il en sera de même des $H^{(k)}$.

L'étape générale de l'algorithme de remplissage de ces tables est comme suit : on choisit dans une table $R^{(j)}$ ou $H^{(k)}$ une case vide qui est immédiatement à droite ou immédiatement à gauche d'une case déjà remplie. Disons par exemple que $R^{(j)}_{i,u_k} = l$ et que $R^{(j)}_{i,u_{k+1}}$ est vide (au départ cette situation a lieu pour $i = 1$ et $k = 0$). Alors on choisit un nouvel entier n et écrit $R^{(j)}_{i,u_{k+1}} = n$. Puis on reflète ce choix en créant une ligne n dans toutes les tables $R^{(j')}$ (où on initialise juste les cases $R^{(j')}_{n,u_0} = R^{(j')}_{n,u_{l_j}} = n$) et dans la table T (où cette ligne n est au départ créée vide). On note dans la table T les nouvelles informations $T_{l,u_{k+1}} = n$ et $T_{n,u_{k+1}^{-1}} = l$. Enfin, les informations de la table T sont répercutées (récursivement) dans toutes les tables $R^{(j)}$ et $H^{(k)}$: toutes les fois qu'on trouve dans une $R^{(j')}$ une case remplie $R^{(j')}_{i',u_{k'}} = l'$, telle que $R^{(j')}_{i',u_{k'+1}}$ soit vide et telle que $T_{l',u_{k'+1}}$ soit remplie, alors on pose $R^{(j')}_{i',u_{k'+1}} = T_{l',u_{k'+1}}$, et de même si on trouve une case remplie $R^{(j')}_{i',u_{k'}} = l'$, telle que $R^{(j')}_{i',u_{k'-1}}$ soit vide et telle que $T_{l',u_{k'}^{-1}}$ soit remplie, alors on pose $R^{(j')}_{i',u_{k'-1}} = T_{l',u_{k'}^{-1}}$ (au départ de cette récursion seules les informations qu'on vient de mettre $T_{l,u_{k+1}} = n$ et $T_{n,u_{k+1}^{-1}} = l$ vont permettre de remplir des trous dans les $R^{(j)}$ et $H^{(k)}$, mais dès qu'on aura rempli un trou, des informations plus anciennes de T pourront servir).

Au cours de cet algorithme, il se passe un phénomène particulier si on remplit la dernière case vide d'une ligne d'un $R^{(j)}$ ou d'un $H^{(k)}$: si, par exemple, $R^{(j)}_{i,u_k} = l$, que $R^{(j)}_{i,u_{k+1}}$ est vide et que $R^{(j)}_{i,u_{k+2}} = m$ alors en remplissant $R^{(j)}_{i,u_{k+1}} = n$ on déduit une information supplémentaire nouvelle $T_{m,u_{k+2}^{-1}} = n$. Si la case $T_{m,u_{k+2}^{-1}}$ était vide ceci la remplit juste et on procède comme d'habitude, mais si on avait déjà une valeur $T_{m,u_{k+2}^{-1}} = n'$ alors ce « conflit » nous apprend en fait que les classes n et n' sont égales. Il faut alors reporter cette information partout : on remplace dans toutes les tables le plus grand des entiers n, n' (disons n) par le plus petit (qui est donc n'). Puis on essaye de supprimer dans toutes les tables la ligne n : si cette ligne a une entrée qui diffère de l'entrée correspondante de la ligne n', on a un nouveau conflit qu'il faut régler récursivement...

On peut démontrer que, pourvu qu'on remplisse les entrées vides dans un ordre raisonnable (tel, par exemple, que tous les générateurs et leurs inverses soient considérés à leur tour), l'algorithme est garanti de se terminer si $|H\backslash G|$ est fini. On ne peut toutefois fixer aucune borne à son temps d'exécution : il existe des présentations du groupe trivial où on énumérera des milliards de classes avant de trouver un conflit ! On pourra consulter [**Johnson**], § 11 pour des détails.

3.2. Le lemme de Schreier. Ce lemme permet de réaliser en quelque sorte l'inverse de ce que fait l'algorithme de Todd-Coxeter. Étant donné un groupe G de générateurs $\{x_i\}_{i\in I}$, et un ensemble de représentants $\{t\}_{t\in T}$ des classes $H\backslash G$, pour un sous-groupe H, il décrit des générateurs de H. Pour cela, nous avons besoin d'une notation : soit $\pi(g)$ l'élément de T dans la même classe que g (c'est-à-dire que $Hg = H\pi(g)$). Alors le lemme dit que les $tx_i\pi(tx_i)^{-1}$, pour i parcourant I et t parcourant T, engendrent H.

3.3. Base et « strong generators » d'un groupe de permutation. Nous décrivons maintenant les méthodes (inventées au départ par [**Sims**]) qui rendent les calculs efficaces dans les groupes de permutations.

Nous supposons que G est un groupe de permutations d'un ensemble Ω et nous notons $C_G(\omega_1,\ldots,\omega_n)$ le sous-groupe des éléments de G qui fixent les points $\omega_1,\ldots,\omega_n$.

On appelle *base* pour G une suite $\omega_1,\ldots,\omega_n$ de points tels que si on pose $G^{(i)} = C_G(\omega_1,\ldots,\omega_i)$ alors on a $G = G^{(0)} \supsetneq G^{(1)} \supsetneq \cdots \supsetneq G^{(n)} = 1$. On appelle « strong generators » (relatifs à cette base) un ensemble S de générateurs de G tels que pour tout i, l'ensemble $S\cap G^{(i)}$ est un ensemble de générateurs de $G^{(i)}$.

La donnée d'un base et de « strong generators » permet de répondre efficacement à toutes sortes de questions sur G, par exemple pour déterminer l'ordre de G il suffit de connaître $|G^{(i)}/G^{(i+1)}|$, qui est égal à l'ordre de l'orbite de ω_i sous $G^{(i)}$ (et cette orbite peut être déterminée facilement en appliquant les générateurs à tour de rôle). La donnée de la base permet aussi de de représenter de façon compacte les éléments de G, qui sont déterminés par l'image de la base.

Il est facile de trouver une base et des « strong generators » pour un groupe donné par des permutations qui l'engendrent : on commence par choisir un point ω_1 qui n'est pas fixe par G. On calcule l'orbite de ω_1 sous G en appliquant les générateurs à tour de rôle ; au passage on aura trouvé des éléments qui envoient ω_1 sur ses transformés, c'est-à-dire des représentants des classes $G^{(0)}/G^{(1)}$. Par le lemme de Schreier on trouve alors des générateurs de $G^{(1)}$. On recommence alors le procédé à partir de $G^{(1)}$, ...

Bien que cet exemple soit trop simple pour être vraiment intéressant, voici les fonctions GAP correspondantes pour $\mathfrak{S}_3$:

```
gap> Base(S3);
[ 1, 2 ]
gap> PermGroupOps.StrongGenerators(S3);
[ (2,3), (1,2) ]
```

À la fin du calcul on garde les représentants obtenus de $G^{(i)}/G^{(i+1)}$. Ils permettent d'obtenir facilement une écriture unique de tout élément $g\in G$ de la forme $g = r_1\cdots r_n$ où r_i est un représentant de $G^{(i)}/G^{(i+1)}$, ce qui est un analogue de l'élimination Gaussienne dans le cadre des groupes de permutations ; ils permettent aussi de tester facilement l'appartenance d'un élément $g\in\mathfrak{S}_\Omega$ à G : dans les deux cas, étant donné un élément g, on trouve r_1 en regardant l'image de ω_1 par g, et on continue en regardant l'image de ω_2 par $r_1^{-1}g$ (qui est dans $G^{(1)}$) ...

3.4. L'algorithme de Dixon-Schneider. Nous ne faisons qu'esquisser cette méthode, dont la première idée remonte à Burnside, pour trouver la table des caractères d'un groupe fini ; le lecteur consultera [**Dixon**] et [**Schneider**] pour plus de détails. Soit χ le caractère d'une représentation irréductible $\rho : H \to GL_n(\mathbb{C})$. C'est une application $G \to \mathbb{C}$, constante sur les classes de conjugaison. On peut étendre χ et ρ à *l'algèbre* $\mathbb{C}G$ du groupe G, formée des combinaisons linéaires formelles à coefficients dans $\mathbb{C}$ d'éléments de G, où la multiplication est héritée de celle de G. Il est facile de montrer que la représentation étant irréductible, ρ étendue à $\mathbb{C}G$ est surjective sur l'algèbre de toutes les matrices $M_n(\mathbb{C})$, donc l'image d'un élément z du centre $Z\mathbb{C}G$ de l'algèbre du groupe est centrale dans $M_n(\mathbb{C})$, donc est une matrice scalaire. Le scalaire est noté $\omega_\chi(z)$, et est égal à $\chi(z)/\chi(1)$; l'application ω_χ est donc un homomorphisme d'algèbres $Z\mathbb{C}G \to \mathbb{C}$. Une base de $Z\mathbb{C}G$ est formée des éléments $S_C = \sum_{g\in C} g$ où C parcourt l'ensemble $\mathcal{C}$ des classes de conjugaison de G. Soit M_C la matrice de la multiplication par S_C dans $Z\mathbb{C}G$, exprimée dans la base $\{S_{C'}\}_{C'\in\mathcal{C}}$. Le fait que ω_χ soit un homomorphisme d'algèbres implique que le vecteur $V_\chi = \{\omega_\chi(S_{C'})\}_{C'\in\mathcal{C}}$ est un vecteur propre de M_C (pour la valeur propre $\omega_\chi(S_C)$). On peut montrer que les vecteurs V_χ constituent la seule base de vecteurs propres simultanés des matrices $\{M_C\}_{C\in\mathcal{C}}$ dont le coefficient sur S_1 est 1.

L'algorithme comprend les étapes suivantes :

(1) Déterminer les classes de conjugaison de G. Il existe des méthodes relativement efficaces pour cette tâche dans les groupes de permutation, basées sur un parcours aléatoire de G et le calcul d'invariants des classes de conjugaison (par exemple, pour un groupe de permutations, la décomposition en cycles est un tel invariant).

(2) Déterminer les matrices M_C. Ici le problème essentiel est de déterminer à quelle classe un élément arbitraire appartient. Le calcul d'invariants pour les classes est ici encore essentiel.

(3) Diagonaliser simultanément les $\{M_C\}_{C\in\mathcal{C}}$ et déterminer ainsi les V_χ. C'est la partie la plus dure du calcul *a priori* et c'est ici qu'apparaissent les contributions de Dixon et Schneider. Dixon effectue le calcul modulo p, où p est choisi de sorte que toutes les valeurs des caractères de G vivent dans $\mathbb{Q}[e^{2i\pi/(p-1)}]$; les calculs sont plus faciles modulo p et le choix de p permet de relever les résultats en caractéristique 0. Schneider montre comment obtenir les vecteurs propres en n'utilisant qu'une partie des M_C.

(4) Une fois trouvés les V_χ, puisque le coefficient $V_{\chi,C}$ de V_χ sur S_C est $|C|\chi(g)/\chi(1)$, pour $g \in C$, il suffit de déterminer $\chi(1)$ pour avoir χ. On obtient $\chi(1)$ par la formule d'orthogonalité des caractères qui donne $\sum_{C\in\mathcal{C}} V_{\chi,C}\overline{V_{\chi,C}}/|C| = |G|/\chi(1)^2$.

Montrons maintenant une table des caractères obtenue en GAP par l'algorithme de Dixon-Schneider. Nous prenons cette fois comme exemple le groupe $\mathfrak{S}_4$.

```
gap> S4:=Group((1,2),(2,3),(3,4));
Group( (1,2), (2,3), (3,4) )
gap> Display(CharTable(S4));

     2  3  2  .  3  2
     3  1  .  1  .  .

       1a 2a 3a 2b 4a
    2P 1a 1a 3a 1a 2b
    3P 1a 2a 1a 2b 4a

X.1     1  1  1  1  1
X.2     1 -1  1  1 -1
X.3     2  0 -1  2  0
X.4     3 -1  0 -1  1
X.5     3  1  0 -1 -1
```

Ici `1a`, `2a`, `3a`, `2b`, `4a` représentent des noms arbitrairement attribués aux classes de conjugaison (le chiffre toutefois note l'ordre d'un élément de la classe), et `X.1`, `X.2`, `X.3`, `X.4`, `X.5` sont des noms arbitrairement attribués aux caractères. Les lignes `2P` (resp. `3P`) sont les « powermaps » indiquant dans quelle classe tombe le carré (resp. le cube) d'une classe. Les deux premières lignes encodent l'ordre du centralisateur d'un élément, décomposé suivant les puissances des nombres premiers (puissance de 2 pour la ligne `2` et puissance de 3 pour la ligne `3`).

Cette exemple illustre une difficulté avec les tables de caractères obtenues ainsi de façon « abstraite » : il est difficile de faire un lien entre les étiquettes arbitrairement données et une description plus structurelle des classes et caractères du groupe. Dans le cas du groupe symétrique, on peut faire appel à un package qui « connaît la théorie des groupes symétriques » :

```
gap> RequirePackage("chevie");

WELCOME to the CHEVIE package, Version 3
http://www.math.rwth-aachen.de/~CHEVIE

Meinolf Geck, Frank Luebeck, Gerhard Hiss,
Gunter Malle, Jean Michel, Goetz Pfeiffer
Lehrstuhl D fuer Mathematik, RWTH Aachen
Universit'e Paris VII
AG Computational Mathematics Universit"at Kassel
Galway University

For first help type
?CHEVIE Version 3 -- a short introduction

gap> S4:=CoxeterGroup("A",3);
CoxeterGroup("A", 3)
```

```
gap> Display(CharTable(S4));
A3

      2    3    2    3    .  2
      3    1    .    .    1  .

        1111  211   22   31  4
     2P 1111 1111 1111   31 22
     3P 1111  211   22 1111  4

1111       1   -1    1    1 -1
211        3   -1   -1    0  1
22         2    0    2   -1  0
31         3    1   -1    0 -1
4          1    1    1    1  1
```

Ici le groupe symétrique est connu dans le cadre de la théorie des groupes de Coxeter. Les classes sont indexées par des partitions de 4 (décomposition en cycles) et les caractères aussi (diagrammes de Young). Les caractères ont été obtenus par la formule de Murnaghan-Nakayama.

4. Synopsis de GAP

GAP est un langage « quelque part à mi-chemin entre PASCAL et MAPLE », très riche et contenant de nombreux domaines des mathématiques. Nous présentons ci-dessous juste un échantillon minimal. Nous conseillons au lecteur de télé-charger GAP et de s'entraîner avec les rudiments présentés ci-dessous.

4.1. Constantes

```
gap> 123^12;
11991163848716906297072721
gap> Factorial(9);
362880
gap> 456/123;
152/41
gap> QuoInt(456,123);
3
gap> 456 mod 123;
87
gap> Factors(60);
[ 2, 2, 3, 5 ]
```

GAP comprend les entiers et rationnels de taille arbitraire. `QuoInt` et `mod` sont la partie entière du quotient et le reste, respectivement. `Factors` décompose en facteurs premiers.

```
gap> GaloisCyc(E(3),-1);
E(3)^2
```

Le symbole `E(k)` représente $e^{2i\pi/k}$. Si z est un nombre cyclotomique, `GaloisCyc(`z`,n)` effectue l'automorphisme de Galois qui envoie toute racine de l'unité `E(k)` sur `E(k)^n` (pour $n = -1$ on obtient la conjugaison complexe).

Les nombres cyclotomiques sont indispensables pour les tables de caractères :

```
gap> CharTable(Group((1,2,3))).irreducibles;
[ [ 1, 1, 1 ], [ 1, E(3), E(3)^2 ], [ 1, E(3)^2, E(3) ] ]
```

```
gap> 3*Z(9);
0*Z(3)
gap> last in GF(9); # last est le dernier resultat
true
gap> Elements(GF(9));
[ 0*Z(3), Z(3)^0, Z(3), Z(3^2), Z(3^2)^2, Z(3^2)^3,
Z(3^2)^5, Z(3^2)^6, Z(3^2)^7 ]
gap> 2 in [1,4,5];
false
gap> not true;true or false;
false
true
```

Le symbole `Z(q)` représente un générateur du corps fini $\mathbb{F}_q$. Le signe `#` annonce un commentaire. La fonction `GF(q)` retourne le corps fini $\mathbb{F}_q$ lui-même. Les valeurs vrai et faux sont représentées par les constantes booléennes `true` et `false`. Enfin l'opérateur `in` permet de tester l'appartenance d'un élément à un ensemble.

```
gap> Concatenation("une ","chaine");
"une chaine"
gap> "un"<"deux";"un"<="un";
false
true
gap> "un"<>"deux";
true
```

On voit ci-dessus diverses opérations sur des chaînes de caractères. Le symbole $\neq$ s'écrit <>.

```
gap> (1,3,5)(2,4,6)^2;(1,3,5)(2,4,6)^3;
(1,5,3)(2,6,4)
()
gap> 2^(1,2,3);
3
gap> (1,2,3)^(2,5);
(1,5,3)
gap> (2,5)^-1*(1,2,3)*(2,5);
(1,5,3)
```

Les permutations s'écrivent suivant leur décomposition en cycles. Le symbole `^` est aussi utilisé pour l'action d'une permutation (deuxième ligne ci-dessus) et pour la conjugaison dans un groupe (troisième ligne ci-dessus).

4.2. Fonctions. Les fonctions à un argument peuvent s'écrire en « λ-notation » :

```
gap> cube:=x->x^3;
function ( x ) ... end
gap> cube(5);
125
```

La forme générale est comme suit :

```
gap> cube:=function(x)local y;y:=x^3;return y;end;
function ( x ) ... end
gap> commutator:=function(x,y)return x*y*x^-1*y^-1;end;
function ( x, y ) ... end
gap> commutator((1,2),(2,3));
(1,2,3)
gap> hello:=function()Print("bonjour\n");end;
function ( ) ... end
gap> hello();
bonjour
```

Le premier exemple définit la même fonction `cube` que ci-dessus mais utilise la forme générale, et une variable locale. Les exemples qui suivent montrent comment définir une fonction à deux, ou 0 arguments (le caractère `\n` imprime un retour chariot).

On peut même définir une fonction à un nombre variable d'arguments (comme la fonction système `Print`) en utilisant l'argument spécial `arg` :

```
gap> Last:=function(arg)return arg[Length(arg)];end;
function ( arg ) ... end
gap> Last(59,77,33);
33
gap> Last("un",E(4),Z(3));
Z(3)
```

4.3. Listes. Les « types de données » fondamentaux en gap sont les listes et les « records ».

Les listes peuvent être emboîtées, et avoir des trous :

```
[ 2, [ 3, 4 ],,, "ert" ]
gap> Add(l,9);l;
[ 2, [ 3, 4 ],,, "ert", 9 ]
gap> Append(l,[3,4]);l;
[ 2, [ 3, 4 ],,, "ert", 9, 3, 4 ]
gap> Length(l);
8
gap> Position(l,9);
6
gap> IsBound(l[2]);
true
gap> Unbind(l[2]);l;
[ 2,,,, "ert", 9, 3, 4 ]
```

La fonction `Add` ajoute un élément à une liste, et `Append` concatène deux listes.

On peut indexer une liste par une autre liste grâce à la syntaxe `{}` :

```
gap> l1:=[1,5];
[ 1, 5 ]
gap> l{[1,5]};
[ 2, "ert" ]
gap> l{l1};
[ 2, "ert" ]
```

Les vecteurs et les matrices sont juste des listes : quand les dimensions sont conformes on peut :

```
gap> [1,2]+[3,4];
[ 4, 6 ]
```

Ajouter des vecteurs.

```
gap> [[1,0],[1,1]]+[[1,1],[-1,0]];
[ [ 2, 1 ], [ 0, 1 ] ]
gap> [[1,0],[1,1]]*[[1,1],[-1,0]];
[ [ 1, 1 ], [ 0, 1 ] ]
gap> PrintArray(last);
[ [ 1, 1 ],
[ 0, 1 ] ]
```

Ajouter et multiplier des matrices (`PrintArray` donne un meilleur affichage de ces dernières).

4.4. Ranges

```
gap> [2..5]=[2,3,4,5];
true
gap> [2,4..8]=[2,4,6,8];
true
```

Les « ranges » servent d'abréviation aux listes qui sont des progressions arithmétiques. La syntaxe `[`a`,`$a+b$`..` c`]` représente $[a, a+b, a+2b, \ldots, c]$ (et est acceptée quand $c-a$ est un multiple de b). Le nombre $a+b$ peut être omis quand $b=1$.

4.5. Programmation fonctionnelle. Les listes donnent avec les « λ-fonctions » un outil de programmation très efficace :

```
gap> List([1..10],x->x^3);
[ 1, 8, 27, 64, 125, 216, 343, 512, 729, 1000 ]
```

`List` applique son deuxième argument à tous les éléments du premier.

```
gap> Sum([1..10]);
55
gap> Sum([1..10],x->x^3);
3025
gap> Product([(1,2,3),(2,3,4)],x->x^2);
(1,2)(3,4)
```

La deuxième forme `Sum(l,f)` est équivalente à `Sum(List(l,f))`. La fonction `Product` se comporte de même.

Un certain nombre de fonctions prennent comme arguments une liste et une fonction à valeur booléenne (un prédicat) :

```
gap> Filtered([1..10],x->x mod 2=0);
[ 2, 4, 6, 8, 10 ]
gap> ForAll([3,5,7],IsPrime);
true
gap> ForAny([3,5,7],IsPrime);
true
gap> Number([1..10],IsPrime);
4
gap> First([1..10],IsPrime);
2
gap> PositionProperty([8..15],IsPrime);
4
```

La fonction `Filtered` retourne les éléments qui satisfont le prédicat. `Number` compte le nombre d'éléments qui satisfont le prédicat, `First` retourne le premier élément qui le satisfait, et `PositionProperty` retourne la position de celui-ci.

4.6. Ensembles. Un ensemble est juste une liste dont les éléments sont triés (un ordre total sur tous les objets GAP est défini par le langage une fois pour toutes). Les opérations habituelles sont définies sur les ensembles.

```
gap> Set([1/3,2,3,2]);
[ 1/3, 2, 3 ]
gap> Intersection([1..10],[4,6..16]);
[ 4, 6, 8, 10 ]
gap> IsSubset([1..10],[2..6]);
true
```

4.7. Records. Les « records » comprennent des champs variés repérés par leur nom. En GAP, ces champs peuvent être changés dynamiquement.

```
gap> r:=rec(l:=[4,5,6],s:="ert",perm:=());
rec(
l := [ 4, 5, 6 ],
s := "ert",
perm := () )
gap> r.l;
[ 4, 5, 6 ]
gap> RecFields(r);
[ "l", "s", "perm" ]
gap> IsBound(r.new);
false
gap> r.new:=rec(a:=4);
rec(
a := 4 )
gap> IsBound(r.new);
true
gap> RecFields(r);
[ "l", "s", "perm", "new" ]
```

```
gap> r;
rec(
l := [ 4, 5, 6 ],
s := "ert",
perm := (),
new := rec(
a := 4 ) )
```

4.8. Structures de contrôle : GAP dispose des structures de contrôle habituelles des langages de type PASCAL, avec quelques avantages :

```
gap> for i in [1..5] do Print("i=",i," i^3=",i^3,"\n");od;
i=1 i^3=1
i=2 i^3=8
i=3 i^3=27
i=4 i^3=64
i=5 i^3=125
```

Ici l'argument de `in` peut être un groupe, une liste, un corps...

```
gap> i:=3;;while i<1000 do Print("i=",i,"\n");i:=i*i;od;
i=3
i=9
i=81
```

Ici le double « ; » après la première instruction empêche l'affichage de son résultat.

4.9. Groupes. Ici nous donnons les exemples avec un groupe de permutations mais la plupart des exemples qui ne réfèrent pas explicitement à une permutation marchent avec tous les types de groupes.

```
gap> g:=Group([(1,2),(2,3),(3,4)],());
Group( (1,2), (2,3), (3,4) )
```

La forme ci-dessus, qui prend comme arguments la liste des générateurs, suivie de l'élément neutre, est un peu plus générale que la forme équivalente `Group((1,2),(2,3),(3,4))`.

```
gap> (1,2) in g;
true
gap> h:=Subgroup(g,[(2,3),(3,4)]);(1,2) in h;
Subgroup( Group( (1,2), (2,3), (3,4) ), [ (2,3), (3,4) ] )
false
gap> ConjugacyClasses(g);
[ConjugacyClass(Group((1,2), (2,3), (3,4)), ()),
 ConjugacyClass(Group((1,2), (2,3), (3,4)), (3,4)),
 ConjugacyClass(Group((1,2), (2,3), (3,4)), (2,3,4)),
 ConjugacyClass(Group((1,2), (2,3), (3,4)), (1,2)(3,4)),
 ConjugacyClass(Group((1,2), (2,3), (3,4)), (1,2,3,4))]
gap> Elements(h);
[ (), (3,4), (2,3), (2,3,4), (2,4,3), (2,4) ]
gap> d:=DerivedSubgroup(g);
Subgroup( Group( (1,2), (2,3), (3,4) ),
[ (1,3,2), (2,4,3) ] )
```

```
gap> s:=SylowSubgroup(g,2);
Subgroup( Group( (1,2), (2,3), (3,4) ),
[ (3,4), (1,2), (1,3)(2,4) ] )
gap> AbelianInvariants(g);
[ 2 ]
```

La fonction `AbelianInvariants(g);` décrit l'abélianisé de `g` (le quotient de `g` par son dérivé).

```
gap> k:=Stabilizer(g,[1,4],OnSets);
Subgroup( Group( (1,2), (2,3), (3,4) ),
[ (2,3), (1,4)(2,3) ] )
gap> orbits:=Orbits(g,[1..8]);
[ [ 1, 2, 3, 4 ], [ 5 ], [ 6 ], [ 7 ], [ 8 ] ]
gap> orbits:=Orbits(k,[1..8]);
[ [ 1, 4 ], [ 2, 3 ], [ 5 ], [ 6 ], [ 7 ], [ 8 ] ]
gap> orbits:=Orbits(k,[1..6]);
[ [ 1, 4 ], [ 2, 3 ], [ 5 ], [ 6 ] ]
gap> h:=Operation(k,orbits[1]);
Group( (1,2) )
gap> OnTuples([2,3],(1,3));
[ 2, 1 ]
gap> OnSets([2,3],(1,3));
[ 1, 2 ]
```

Les fonctions `Orbits`, `Operation`, `Stabilizer` ont un troisième argument optionnel qui décrit comment on fait agir les éléments du groupe sur : les éléments du deuxième argument pour `Orbits` et `Operation`, le deuxième argument pour `Stabilizer`. La fonction `OnSets`, utilisée comme argument à `Stabilizer`, fait agir sur les ensembles, et `OnTuples` sur les listes. Le défaut est de faire agir sur les points. L'ordre

```
Operation(k,orbits[1])
```

restreint l'opération de k à `orbits[1]` et donne un nouveau groupe agissant sur un ensemble en bijection avec cette orbite dont les éléments on été renumérotés consécutivement.

Après un appel a `Operation` (resp. `Stabilizer`) l'application quotient (resp. inclusion) est disponible. Elle peut être obtenue par

```
gap> hom:=OperationHomomorphism(k,h);
OperationHomomorphism(Subgroup(Group((1,2), (2,3), (3,4)),
[ (2,3), (1,4)(2,3) ] ), Group( (1,2) ) )
gap> Kernel(hom);
Subgroup( Group( (1,2), (2,3), (3,4) ), [ (2,3) ] )
```

Cela marche aussi avec un sous-groupe, ici par exemple :

```
gap> l:=Subgroup(k,[(2,3)]);
Subgroup( Group( (1,2), (2,3), (3,4) ), [ (2,3) ] )
gap> OperationHomomorphism(l,h);
OperationHomomorphism(Subgroup(Group((1,2), (2,3), (3,4)),
[ (2,3) ] ), Group( (1,2) ) )
```

Les opérations ensemblistes marchent aussi pour des groupes :

```
gap> Intersection(Subgroup(g,[(1,2),(2,3)]),
> Subgroup(g,[(2,3),(3,4)]));
Subgroup( Group( (1,2), (2,3), (3,4) ), [ (2,3) ] )
```

Enfin, signalons une fonction spécifique aux groupes de permutations :

```
gap> Blocks(s,[1..4]);
[ [ 1, 4 ], [ 2, 3 ] ]
```

(rappelons que `s` est le 2-Sylow de $\mathfrak{S}_4$). La fonction `Blocks(s,l)` suppose que `s` agit transitivement sur `l` et retourne alors la partition la plus fine non triviale stabilisée par `s`.

4.10. Groupes donnés par générateurs et relations

```
gap> g:=FreeGroup(3,"g");
Group( g.1, g.2, g.3 )
```

ou

```
gap> g:=FreeGroup("a","b","c");
Group( a, b, c )
```

Dans le deuxième cas il est recommandé de faire :

```
gap> a:=g.1;b:=g.2;c:=g.3;
a
b
c
```

On peut alors construire un quotient :

```
gap> S4:=g/[a^2,b^2,c^2,(a*b)^3,(b*c)^3,(a*c)^2];
Group( a, b, c )
gap> Size(S4);
24
```

4.11. Environnement. Finissons en donnant quelques commandes utiles :

```
gap> LogTo("3mai");
```

Demande de sauver toutes les commandes tapées et leur résultat sur le fichier `3mai`. Cette sauvegarde est terminée en tapant :

```
gap> LogTo();
```

```
gap>Read("essai.g");
```

lit dans la session courante les commandes qui sont dans le fichier `essai.g` (l'extension `.g` est traditionnelle pour les fichiers GAP), tandis que

```
Edit("essai.g");
```

appelle l'éditeur (celui défini par la variable d'environnement `$EDITOR` sous UNIX) puis relit le fichier (fournissant ainsi une méthode de développement).

L'aide en ligne est abondante et facile d'accès.

```
gap> ?group
```

donne l'aide sur la commande `Group` tandis que `??group` montre toutes les références a `group` ou `Group`. La page suivante de l'aide s'obtient par `?>` et la page précédente par `?<`.

5. Un exemple : le cube de Rubik $2 \times 2 \times 2$

Nous allons illustrer les commandes décrites à la section précédente par l'analyse d'un exemple en GAP : le Rubik's cube $2\times2\times2$. Nous utiliserons la notation suivante pour les facettes du cube :

		1	2				
		3	4				
5	6	9	10	13	14	17	18
7	8	11	12	15	16	19	20
		21	22				
		23	24				

Nous commençons par rentrer la description comme permutations des rotations d'un quart de tour de chacune des faces, et définir le groupe engendré par ces permutations (le groupe du cube) :

```
gap> Haut:=(1,2,4,3)(5,17,13,9)(6,18,14,10);;
gap> Gauche:=(5,6,8,7)(1,9,21,20)(3,11,23,18);;
gap> Avant:=(9,10,12,11)(3,13,22,8)(4,15,21,6);;
gap> Droit:=(13,14,16,15)(2,19,22,10)(4,17,24,12);;
gap> Arriere:=(17,18,20,19)(2,5,23,16)(1,7,24,14);;
gap> Bas:=(21,22,24,23)(7,11,15,19)(8,12,16,20);;
gap> cube:=Group(Haut,Gauche,Avant,Droit,Arriere,Bas);
Group( ( 1, 2, 4, 3)( 5,17,13, 9)( 6,18,14,10),
( 1, 9,21,20)( 3,11,23,18)( 5, 6, 8, 7),
( 3,13,22, 8)( 4,15,21, 6)( 9,10,12,11),
( 2,19,22,10)( 4,17,24,12)(13,14,16,15),
( 1, 7,24,14)( 2, 5,23,16)(17,18,20,19),
( 7,11,15,19)( 8,12,16,20)(21,22,24,23) )
gap> Size(cube);
88179840
gap> Factors(last);
[ 2, 2, 2, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 5, 7 ]
```

Nous vérifions maintenant que le groupe est transitif sur les facettes, et qu'il a une structure de « blocs » qui correspond à la préservation de l'intégrité physique de chacun des 8 petits cubes (les « coins ») :

```
gap> Orbits(cube,[1..24]);
[ [ 1, 2, 9, 7, 4, 19, 5, 21, 10, 24, 11, 3, 15, 17, 22, 6,
23, 20, 12, 14, 13, 18, 8, 16 ] ]
gap> coins:=Blocks(cube,[1..24]);
[ [ 1, 5, 18 ], [ 2, 14, 17 ], [ 3, 6, 9 ], [ 7, 20, 23 ],
[ 4, 10, 13 ], [ 16, 19, 24 ], [ 8, 11, 21 ], [ 12, 15, 22 ] ]
```

Étudions maintenant le quotient du cube qui consiste à ne retenir que l'action sur les coins (en oubliant leur orientation) :

```
gap> surcoins:=Operation(cube,coins,OnSets);
Group( (1,2,5,3), (1,3,7,4), (3,5,8,7), (2,6,8,5),
(1,4,6,2), (4,7,8,6) )
```

```
gap> Size(surcoins);
40320
gap> Factorial(8);
40320
```

Manifestement, toutes les permutations des 8 coins sont possibles. Étudions maintenant les changements d'orientations possibles en laissant les coins à leur place :

```
gap> homcoins:=OperationHomomorphism(cube,surcoins);
OperationHomomorphism( Group( ( 1, 2, 4, 3)( 5,17,13, 9)
( 6,18,14,10), ( 1, 9,21,20)( 3,11,23,18)( 5, 6, 8, 7),
( 3,13,22, 8)( 4,15,21, 6)( 9,10,12,11),
( 2,19,22,10)( 4,17,24,12)(13,14,16,15),
( 1, 7,24,14)( 2, 5,23,16)(17,18,20,19),
( 7,11,15,19)( 8,12,16,20)(21,22,24,23) ),
Group( (1,2,5,3), (1,3,7,4), (3,5,8,7),
(2,6,8,5), (1,4,6,2), (4,7,8,6) ) )
gap> stabcoins:=Kernel(homcoins);
Subgroup( Group( ( 1, 2, 4, 3)( 5,17,13, 9)( 6,18,14,10),
( 1, 9,21,20)( 3,11,23,18)( 5, 6, 8, 7),
( 3,13,22, 8)( 4,15,21, 6)( 9,10,12,11),
( 2,19,22,10)( 4,17,24,12)(13,14,16,15),
( 1, 7,24,14)( 2, 5,23,16)(17,18,20,19),
( 7,11,15,19)( 8,12,16,20)(21,22,24,23) ),
[ ( 1,18, 5)(12,15,22), ( 2,17,14)(12,22,15),
( 2,17,14)( 3, 9, 6)(12,15,22),
( 2,14,17)( 7,20,23)(12,22,15),
( 4,10,13)(12,15,22), (12,22,15)(16,19,24),
( 8,11,21)(12,22,15) ] )
gap> Size(stabcoins);
2187
```

Le groupe des orientations `stabcoins` est un sous-groupe du groupe de toutes les orientations possibles qui est $(\mathbb{Z}/3\mathbb{Z})^8$, donc il est commutatif. La commande pour étudier un groupe commutatif est

```
gap> AbelianInvariants(stabcoins);
[ 3, 3, 3, 3, 3, 3, 3 ]
```

Évidemment dans ce cas particulier nous n'en n'avons pas appris plus que si nous avions demandé `Factors(2187)`. Donc seul un tiers des orientations sont permises. Nous pouvons demander lesquelles :

```
gap> (3,6,9) in cube;
false
gap> (3,6,9)^2*(4,10,13) in cube;
true
```

Ainsi, si l'on tourne un coin d'un tiers de tour, on doit faire tourner dans la direction opposée un coin voisin. Le sous-groupe des orientations, étant le noyau d'un homomorphisme, est un sous-groupe distingué. Étudions si le groupe total est produit semi-direct de ce sous-groupe par un complément, c'est-à-dire voyons si on peut trouver un sous-groupe de `cube` isomorphe à $\mathfrak{S}_8$, permutant les coins comme le groupe symétrique, et préservant les orientations (pour une numérotation particulière de ces dernières).

```
gap> compl:=Stabilizer(cube,List(coins,x->x[1]),OnSets);
Subgroup( Group( ( 1, 2, 4, 3)( 5,17,13, 9)( 6,18,14,10),
( 1, 9,21,20)( 3,11,23,18)( 5, 6, 8, 7),
( 3,13,22, 8)( 4,15,21, 6)( 9,10,12,11),
( 2,19,22,10)( 4,17,24,12)(13,14,16,15),
( 1, 7,24,14)( 2, 5,23,16)(17,18,20,19),
( 7,11,15,19)( 8,12,16,20)(21,22,24,23) ),
[ (12,16)(15,19)(22,24), ( 8,12)(11,15)(21,22),
( 7, 8)(11,23)(20,21), ( 4, 7)(10,20)(13,23),
( 3, 4)( 6,10)( 9,13), ( 2, 3)( 6,14)( 9,17),
( 1, 2)( 5,17)(14,18) ] )
gap> Size(compl);
40320
gap> Intersection(compl,stabcoins);
Subgroup( Group( ( 1, 2, 4, 3)( 5,17,13, 9)( 6,18,14,10),
( 1, 9,21,20)( 3,11,23,18)( 5, 6, 8, 7),
( 3,13,22, 8)( 4,15,21, 6)( 9,10,12,11),
( 2,19,22,10)( 4,17,24,12)(13,14,16,15),
( 1, 7,24,14)( 2, 5,23,16)(17,18,20,19),
( 7,11,15,19)( 8,12,16,20)(21,22,24,23) ), [ ] )
gap> Size(last);
1
```

Cette structure de produit semi-direct, notée mathématiquement par

$$\texttt{cube} = \texttt{stabcoins} \rtimes \texttt{compl}$$

décrit complètement la loi de groupe sur `cube`. Si ϕ est l'homomorphisme $\texttt{compl} \to \mathrm{Aut}(\texttt{stabcoins})$ qui décrit l'action de `compl` sur `stabcoins`, on peut représenter les éléments de `cube` par un couple (g, h) où $g \in \texttt{stabcoins}$, $h \in \texttt{compl}$ avec pour loi de groupe $(g, h)(g', h') = (g\phi(h)(g'), hh')$.

D'autres propriétés de `cube` qu'on peut trouver en GAP sont par exemple :

(1) le fait que `cube` est déjà engendré par 3 des rotations (par exemple `haut`, `bas` et `droit`).

(2) déterminer le nombre minimum de mouvements nécessaires pour ramener toute position à une position donnée (ou encore la longueur minimale nécessaire pour écrire tout élément du groupe en les générateurs). On trouve 11.

6. Conclusion

On trouvera un excellent « survey » sur la théorie algorithmique des groupes contenant d'autres références en consultant [**Seress**], dont je me suis largement inspiré.

Références

[AHN] W.A. Alford, G. Havas and M.F. Newman. *Groups of exponent four*, Notices Amer. Math. Soc. **22**, 1975.

[Aschbacher] M. Aschbacher. *On the maximal subgroups of the finite classical groups.* Invent. Math. **76** (1984) 469–514.

[BBR] L. Babai, R. Beals and D. Rockmore. *Deciding finiteness of matrix groups in deterministic polynomial time.* In ISSAC'93 (Kiev) ACM Press, p. 117–126.

[CelLG] F. Celler and R. Leedham-Green. *A non-constructive recognition algorithm for the special linear and other classical groups* in Groups and Computation, DIMACS vol. 28 (AMS, 1997).

[Dixon] J.D. Dixon. *High speed computation of group characters.* Numer. Math. **10** (1967), 446–450.

[Johnson] D. L. Johnson. *Presentations of groups.* London Mathematical Society Student Texts **15**. Cambridge University Press, Cambridge, 1990.

[Mihailova] K.A. Mihailova. *The occurence problem for direct products of groups.* Dokl. Akad. Nauk. **119** (1958) p. 1103–1105.

[Norton] S. Norton. *The construction of J_4*, Proc. Symp. pure Math. **37** (1980), 271–277.

[Schneider] G.A. Schneider. *Dixon's character table algorithm revisited* in Computational group theory I, J. Symbolic Comp. **9** (1990) 601–606.

[Seress] A. Seress. *An introduction to computational group theory*, Notices Amer. Math. Soc. **44** (1997), 671–679.

[Sims] C. C. Sims. *Computational methods in the study of permutation groups*, in Computational problems in abstract algebra, Oxford, 1970 (J. Leech ed.).

[Sims80] C. C. Sims. *How to construct a baby monster.* In M.J.Collins, *Finite simple groups II*, Proc. Symp. Durham 1978, p. 339–345. Academic Press, 1980.

JEAN MICHEL, Équipe des groupes finis, Institut de Mathématiques UMR CNRS 7586, 175, rue du Chevaleret 75013 Paris • *E-mail :* `jmichel@math.jussieu.fr`
Url : `http://www.math.jussieu.fr/~jmichel/`