Infini et effectivité, le point de vue fonctionnel

Francis Sergeraert

Institut Fourier, BP 74, 38402 St Martin d'Hères Cedex
[Draft of a paper published in *Images des Mathématiques*, 1989]

1 Introduction.

Infini et *effectivité* sont deux mots qui semblent appartenir à deux mondes bien distincts ; est-il possible de penser à l'*infini* en terme d'*effectivité* ? L'infini est une notion *abstraite* et paraît ainsi hors de portée de toute approche *effective*, on veut dire par là une approche se traduisant par une réalité concrète. On essaie d'expliquer au contraire dans cet article que l'ambition d'effectivité n'est pas à écarter d'emblée dans un travail où l'infini joue un rôle essentiel ; ambition qui conduit à des questions — et aussi à des réponses ! — bien passionnantes.

La capacité des mathématiciens à traiter avec efficacité de problèmes où l'*infini*, sous une forme ou une autre, intervient de façon essentielle leur donne vis-à-vis des autres populations des airs de sorciers un rien troublants. Il faut reconnaître qu'il y a là de belles prouesses à mettre en bonne place dans la liste des réussites humaines. Plusieurs mathématiciens gardent une notoriété directement attachée aux progrès substantiels qu'ils ont su réaliser quant à une meilleure compréhension de phénomènes où l'infini joue un rôle primordial. Citons Leibniz et Newton (calcul infinitésimal), Cantor (les ensembles infinis), Cauchy (analyse infinitésimale), Hilbert (espaces de dimension infinie), Gödel (incomplétude), Robinson (analyse non-standard).

Une réflexion superficielle pourrait faire penser que l'informaticien vit de ce point de vue dans un monde plus paisible et moins ésotérique que celui de son collègue mathématicien. L'informaticien travaille avec des machines (concrètes ou théoriques), par essence même *finies* ; ces machines ne peuvent travailler qu'avec des programmes, textes forcément *finis*, sur des données nécessairement *finies*, pendant un temps *fini*. Or cette réflexion n'est pas correcte, et ceci pour des raisons fort diverses. La plus simple consiste peut-être à dire que l'infini n'est pas étudié par le mathématicien seulement pour le plaisir, mais aussi parce que très souvent c'est un puissant outil de modélisation ; un exemple vient aussitôt à l'esprit : un calcul numérique non accompagné de calcul d'erreur est sans intérêt, mais la plupart du temps un tel calcul ne peut se faire commodément qu'à l'aide du calcul différentiel, nécessitant donc l'usage d'*infiniment petits*. Du côté des infiniment grands on peut

citer par exemple les études de complexité d'algorithmes, dont l'intérêt pratique n'a pas à être rappelé, et qui reposent bien souvent sur des méthodes asymptotiques.

On veut dans cet article étudier un lien d'une tout autre nature entre infini et programmation, le lien *fonctionnel*. Il s'agit d'un lien assez subtil, et dont pourtant la description est d'une simplicité extraordinaire ; si extraordinaire qu'il n'y a pas de raison de tourner autour du pot, décrivons-le immédiatement.

Un programme est un objet *fini* ; dans un cas concret, ce sera par exemple un texte, une chaîne de caractères de longueur finie, utilisant un ensemble fini de caractères, peut-être les caractères ASCII. Ce programme est susceptible de travailler sur des données (input) pour produire un résultat (output). Prenons par exemple un programme P calculant le carré d'un entier positif ; il est capable de travailler sur *tout* entier positif $n \in \mathbf{N}$ et nous tenons aussitôt notre lien : P est fini alors que l'ensemble $\mathbf{N}$ des entiers sur lesquels P peut travailler ne l'est pas. En général, si P est un programme, désignons par $I(P)$ (I pour *input*) l'ensemble des données sur lesquelles il peut travailler ; le programme P est par essence même fini, alors que $I(P)$ peut fort bien être infini.

On a mis ainsi en contact par ce tour de passe-passe l'univers des objets ordinaires, finis, et celui des mathématiciens, peuplé de monstres variés pas du tout finis. On pourrait ne voir là que considération de philosophe sans intérêt réel. Cet article a pour but de convaincre du contraire.

Il est organisé selon le plan suivant. L'exemple élémentaire du bon programme pour calculer le nombre chromatique d'un graphe, bien connu des graphistes, utilise le point de vue fonctionnel ; il sera soigneusement décrit afin d'isoler et de mettre en lumière les problèmes de programmation, d'*effectivité* donc, qu'on rencontre dans ce genre de situation ; on espère que la nature de la méthode et les difficultés à prévoir seront alors bien comprises. Il sera temps à ce moment d'examiner l'état de la science informatique en la matière ; on verra qu'il est excellent : nos collègues informaticiens ont justement tout ce qu'il faut (lambda-calcul, langages fonctionnels) pour que nous puissions travailler sur ces questions dans les meilleures conditions. La construction sur machine du foncteur *espace de lacets*, très facile à décrire, donnera un bon exemple des capacités étonnantes de programmation disponibles une fois qu'on dispose du bon point de vue et des bons outils : des programmes très simples, à la portée d'un étudiant de maîtrise, permettent de construire en un clin d'œil, *sur machine*, des espaces hautement infinis. Ces espaces sont commodes pour épater la galerie, mais ils ne servent pas qu'à cela : bien qu'infinis, Jean-Pierre Serre les avait inventés vers 1950 pour résoudre des problèmes de nature *finie* (groupes d'homotopie des sphères) ; on a le même genre d'intentions ; dans la dernière section on donnera une idée des résultats théoriques substantiels déjà obtenus en topologie algébrique et une description du beau champ de travail ouvert en programmation à qui veut appliquer concrètement ces méthodes sur machine.

L'encadré 1 donne aux lecteurs peu habitués à l'informatique une description plus précise de la nature éventuellement infinie de l'ensemble $I(P)$. Dans son fon-

dement même, l'approche de l'infini par la méthode fonctionnelle a déjà été souvent utilisée : l'approche de Cauchy de l'analyse infinitésimale est de cette nature et fait l'objet de l'Encadré 2.

Encadré 1.
De la non-finitude d'un ensemble de données.

On a considéré dans l'introduction le programme P faisant correspondre à un entier n son carré n^2. Ainsi, si le programme P est sommé de travailler sur l'entier 97, il doit répondre 9409. Il est élémentaire d'écrire dans un langage quelconque un tel programme, par exemple sur une calculette programmable en Basic.

On a noté $I(P)$ l'ensemble des données sur lesquelles le programme P est en mesure de travailler ; dans l'exemple du calcul du carré, l'ensemble $I(P)$ est l'ensemble $\mathbf{N}$ des entiers, connu comme infini.

Cette assertion peut mettre en éveil l'esprit critique d'un lecteur habitué au travail concret sur machine. L'affirmation que l'ensemble $I(P)$ est infini peut en effet laisser penser que son auteur est peu au fait des contraintes réelles d'utilisation des machines ; sur une calculette Basic par exemple, un *entier* n'est utilisable que s'il est plus petit qu'un certain entier défini par le constructeur de la calculette au moment de sa conception ; souvent c'est quelque chose comme 2^{31} ou 10^{10}, de sorte que le nombre d'entiers sur lesquels notre exemple de programme est *réellement* capable de travailler est en fait fini, et notre critique a raison. Ce genre de contrainte est présent dans la plupart des langages de programmation.

Cette difficulté peut néanmoins être contournée à l'aide de programmes connus comme capables de travailler *en multiprécision*. Ces programmes utilisent la technique suivante. Supposons que notre machine n'accepte que les entiers plus petits que 10^{10} ; appelons alors *petit entier* un entier plus petit que $b = 10^5$, b pour *base*. Si n est un entier, on peut toujours l'écrire, et d'ailleurs d'une seule façon, sous la forme $n = a_p b^p + a_{p-1} b^{p-1} + \ldots + a_0$ où les a_i sont des petits entiers ; ceci revient à découper l'écriture décimale de n en tranches de cinq chiffres ; c'est aussi l'écriture de n en base $b = 10^5$. Il est facile alors de voir que l'écriture analogue du carré n^2 de n peut être déterminée uniquement à l'aide d'une succession d'opérations sur de petits entiers, donc à portée de notre calculette. Techniquement on représentera n comme un *tableau* de petits entiers, le résultat n^2 sortira sous une forme analogue, et le reste est technique de manipulations d'indice, d'opérations élémentaires sur de petits entiers et de retenues.

On peut ainsi, même sur une modeste calculette, calculer le carré d'entiers déjà tout-à-fait conséquents, dont l'écriture peut demander plusieurs centaines de chiffres. Ce qui vient d'être décrit fut inventé il y a bien longtemps par nos ancêtres lorsqu'ils comprirent les surprenantes possibilités de l'écriture des nombres dans une base quelconque, 10 par exemple. Ils s'aperçurent alors que la même méthode, il faudrait mieux dire *le même programme*, est capable de venir à bout de la

multiplication de deux entiers admettant un nombre *quelconque* de chiffres ! C'est rigoureusement le même phénomène qui vient d'être décrit, à ceci près que la base est 10^5 au lieu de 10, mais ceci ne change rien à l'affaire ; et c'est grâce à cela que la table de multiplication usuelle ne va pas plus loin que $9 \times 9 = 81$. Les logiciels de calcul en multiprécision sont maintenant de plus en plus répandus ; ils sont même intégrés à certains langages (Lisp) et sont presque toujours incorporés aux systèmes de calcul formel.

Mais le critique toujours aux aguets trouve aussitôt un autre argument. Bien sûr on est maintenant capable de multiplier de grands entiers, mais il reste une limitation, celle de la place mémoire en machine. Le critique a encore raison. Si un petit entier occupe par exemple une cellule mémoire, alors le plus grand entier *logeable* dans la machine sous forme de tableau de petits entiers sera $N = 10^{5M} - 1$ si M est le nombre de cellules mémoires disponibles ; si un entier est plus grand que N, il est donc impossible de le rentrer dans la machine et a fortiori d'en calculer le carré.

Cependant on trouve de plus en plus facilement dans le commerce des machines *à mémoire extensible*, ce qui permet de travailler comme suit. Dans un premier temps on écrit un programme, par exemple un programme de travail sur entiers à multiprécision. Puis quelqu'un passe commande d'un travail à effectuer avec ce programme. Une étude permet alors de savoir quelle taille mémoire va être indispensable *pour cette utilisation particulière* ; au besoin on téléphone à son marchand de mémoire extensible pour donner à sa mémoire la taille nécessaire. Après quoi on lance le programme qui donne alors, tout a été fait pour, le bon résultat. Et on voit que, *en ce sens là*, notre programme peut travailler sur un entier quelconque.

Le critique pourra une dernière fois hausser les épaules : si le nombre de cellules mémoires indispensables pour un calcul particulier est plus grand que le nombre d'atomes de notre galaxie, on risque d'attendre longtemps la fourniture des extensions mémoires, fourniture souvent difficile pour bien moins que cela ! Mais peu importe, le théoricien se contente de ce fait : *potentiellement* son programme est capable de travailler sur un entier de taille quelconque.

Le terme *théoricien* qui vient d'être utilisé a un côté quelquefois un peu péjoratif : l'appellation de théoricien désigne souvent une personne surtout incapable de réalisations pratiques ! Dans cet article, on cherche à convaincre le lecteur que ces considérations théoriques sur les programmes aux potentialités infinies sont au contraire susceptibles d'applications tout-à-fait concrètes ! Puisqu'on parle ici de théorie, on ne peut pas ne pas rappeler que ce type de machine à mémoire extensible a été parfaitement modélisé par le mathématicien anglais Turing, bien avant l'existence même du mot *ordinateur* ; Turing avait inventé son modèle pour répondre négativement à la conjecture de Hilbert sur un algorithme universel solutionneur des problèmes mathématiques ; quand on sait que le travail de Turing a joué un rôle ô combien essentiel dans la genèse de l'informatique moderne, on a là un bon argument à utiliser à l'égard de ceux qui auraient encore des doutes sur l'intérêt de la recherche fondamentale. On conseille très vivement sur ces questions la lecture

du merveilleux ouvrage "*Alan Turing, the enigma*" de Andrew HODGES (Simon & Schuster, 1983) ; une traduction française intitulée "*Alan Turing ou l'énigme de l'intelligence*" est parue en 1987 à la Bibliothèque Scientifique Payot.

Encadré 2.
La solution de Cauchy pour l'analyse infinitésimale.

Le *truc fonctionnel* dont un type d'application fait l'objet de cet article est connu depuis longtemps. C'est par exemple l'outil essentiel de la formalisation moderne de l'analyse infinitésimale, habituellement attribuée à Cauchy. D'aprés Bourbaki, *Eléments d'histoire des mathématiques* (Masson, 1984), c'est en tout cas lui qui réussit à donner à cette façon de penser une forme suffisamment précise pour en faire un manuel d'enseignement raisonnablement utilisable, ce qui est un excellent critère. Bien sûr la réalité est plus complexe et le travail de Cauchy n'est que l'aboutissement d'une longue et difficile gestation à laquelle de nombreux prédécesseurs ont contribué de façon essentielle ; voir Bourbaki (op. cit.). Toujours est-il que Cauchy expliqua comment articuler les démonstrations d'analyse infinitésimale à l'aide de quantités très traditionnellement notées ε et η (ou δ) depuis Weierstrass, pouvant prendre des valeurs plus ou moins arbitraires, mais dont l'interdépendance est essentielle. Habituellement ε est arbitraire et, *étant donné* un tel ε, on doit être *capable* de démontrer l'existence d'un η vérifiant telle ou telle propriété. Par exemple une fonction $f : \mathbf{R} \to \mathbf{R}$ est continue en $x_0 \in \mathbf{R}$ si pour tout $\varepsilon > 0$ on peut démontrer l'existence d'un $\eta > 0$ tel que si $|x - x_0| < \eta$, alors $|f(x) - f(x_0)| < \varepsilon$.

On a souligné intentionnellement la locution "étant donné" et l'adjectif "capable" pour faire toucher du doigt qu'on se trouve exactement dans la situation d'un cahier des charges d'un programme. En détaillant un peu plus — de façon excessive, mais il faut enfoncer le clou — on peut dire que la démonstration de la continuité de la fonction f en x_0 n'est rien d'autre qu'un *programme* admettant en entrée (input) un réel strictement positif ε et donnant en sortie (output) un autre réel strictement positif η vérifiant la propriété indiquée. La continuité de f en x_0 équivaut ainsi à l'existence d'une *fonction* $\mu : \mathbf{R}_+ \to \mathbf{R}_+$ ($\mathbf{R}_+$ note l'ensemble des réels strictement positifs) telle que si $|x - x_0| < \mu(\varepsilon)$, alors $|f(x) - f(x_0)| < \varepsilon$. Une fonction μ vérifiant une telle propriété est appelée un *module de continuité* pour f en x_0. Les logiciens disent que si, de plus, μ est *récursive*, alors f est *effectivement continue*.

On a passé beaucoup de temps à raconter de deux façons essentiellement la même chose, ceci pour mettre en valeur différents points de vue ayant chacun leur intérêt propre. On voit qu'il y a correspondance à peu près canonique entre le point de vue *logique* qui consiste à utiliser des quantificateurs ($\forall\varepsilon$,$\exists\eta\ldots$) et le point de vue *fonctionnel* qui préfère affirmer l'existence d'*une* fonction telle qu'un argument quelconque pour cette fonction et la valeur qu'elle y prend vérifient une certaine propriété.

On notera que dans tout ce qui précède ni le mot *infini* ni aucun de ses dérivés n'est prononcé ! Alors qu'il s'agit pourtant bien de disserter sur "ce qui se passe" *infiniment près* de x_0. Le fait pour un x d'être très proche de x_0 n'a qu'un intérêt assez secondaire ; ce qui est capital c'est que l'examen d'un *nombre fini* de x proches de x_0 ne sera jamais suffisant pour conclure à la continuité ; une telle assertion doit porter sur une infinité de valeurs de x ; difficulté redoutable et essentielle que Cauchy a tourné par le biais *fonctionnel* : il a remplacé l'examen d'une infinité de x et de leurs propriétés par une assertion portant seulement sur *une fonction*, la fonction μ. Pourtant Cauchy a préféré utiliser une formulation logique (pour tout ε il existe un η tel que...), tout-à-fait équivalente, mais qui justement nous rend bien service car cette formulation souligne un aspect capital de l'affaire ; on avait souligné plus haut l'adjectif "capable" pour mettre en lumière le fait que l'infini est atteint à l'aide d'une affirmation sur l'aspect *potentiel* de travail : *si* vous me donniez un ε strictement positif, *alors* je serais *capable* de produire un η tel qu'une certaine propriété soit vérifiée.

C'est une situation à la Pierre Dac et Francis Blanche dans leur célèbre sketch de voyance : le mathématicien se contente de dire "Si vous me donniez un ε, alors trouver un η tel que... oui, je pourrais le faire !" Bien sûr, contrairement à Pierre Dac, le mathématicien argumente et démontre *pourquoi* il saurait le faire, mais toujours est-il que le plus souvent, il ne le fait pas !

Tout le monde sait à quel point la compréhension et l'apprentissage des méthodes $\varepsilon - \eta$ est difficile pour les débutants ; on devrait (mais ceci a peut-être déjà été fait) comparer avec la méthode fonctionnelle qui permet quelquefois de mesurer de façon plus palpable la difficulté d'une démonstration. Ainsi expliquer la continuité de f en x_0, c'est construire un module de continuité, fonction à argument et à valeur réels. Etudions la démonstration de la continuité de f^2 quand f est supposée continue. En terme de module de continuité, si $\mu : \mathbf{R}_+ \to \mathbf{R}_+$ est un module de continuité de f en x_0, la fonction $\mu'(\varepsilon) = \min(\mu(1), \mu(\varepsilon/(2|f(x_0)|+1)))$ est un module de continuité pour f^2 en x_0 et il en résulte que la continuité de f en x_0 implique la continuité de f^2 au même point. La difficulté nouvelle tient très précisément au fait qu'il s'agit ici de construire une fonction dont l'argument (μ) et la valeur (μ') sont eux-mêmes des fonctions. Pédagogiquement, la distinction entre argument et valeur d'une fonction (argument et/ou valeur pouvant à leur tour être des fonctions) ne serait-elle pas plus commode que la distinction entre quantificateur universel et quantificateur existentiel ? On notera que la structure arborescente des divers arguments et valeurs en question se lit très directement dans le formalisme fonctionnel alors que le formalisme logique demande un algorithme de conversion relativement sophistiqué. Devrait aussi être considérée la distance restant à parcourir pour aboutir à des calculs effectifs sur machine, où la méthode fonctionnelle est évidemment la bonne. Toujours est-il que cette question des fonctions dont argument et/ou valeur sont fonctionnels jouent un rôle capital dans cet article ; c'est la raison pour laquelle on a jugé utile ici ce complément à nos explications sur l'analyse infinitésimale.

2 Le nombre chromatique d'un graphe.

Un *graphe* est un couple (S, A) où S est un ensemble fini, l'ensemble des *sommets* du graphe et où A est un sous-ensemble de $S \times S$, l'ensemble des couples de sommets reliés par une *arête*. On dispose d'un ensemble de *couleurs* $c_1, c_2, \ldots, c_p$ et on cherche à obtenir un *bon coloriage* du graphe, c'est-à-dire à attribuer une couleur à chaque sommet, de façon que deux sommets reliés par une arête soient de couleurs *distinctes*. Notons $s_1, s_2, \ldots, s_n$ les sommets du graphe et $d_1, d_2, \ldots, d_n$ les couleurs qu'on leur attribue. La condition de bon coloriage est donc que si un couple (s_i, s_j) est un élément de A, alors $d_i \neq d_j$. Cette condition est assez restrictive, et si on ne dispose pas de suffisamment de couleurs, autrement dit si p est trop petit, on ne pourra trouver un bon coloriage. Le plus petit entier p qui permet d'obtenir un bon coloriage du graphe G avec p couleurs est appelé le *nombre chromatique* de G. Ce nombre a suscité et suscite toujours un très grand intérêt ; le *problème des quatre couleurs* consiste à savoir si quatre couleurs suffisent quand le graphe peut être dessiné dans un plan.

Un problème de programmation intéressant consiste à demander d'écrire un programme dont l'entrée est un graphe et qui donne en sortie son nombre chromatique. La méthode la plus simple, pour ne pas dire simpliste est la suivante : un coloriage (peut-être erroné) à p couleurs du graphe G est une suite $d_1, \ldots, d_n$ de couleurs choisies parmi $c_1, \ldots, c_p$. Il existe exactement p^n donc un nombre fini de tels coloriages. On classe tous ces coloriages d'une certaine façon, on les teste les uns après les autres jusqu'à constater que oui, un certain coloriage à p couleurs convient, ou bien au contraire que non, aucun ne convient. Dans le premier cas, le nombre chromatique est $\leq p$, dans le second, il est $> p$. On commence avec $p = 1$, puis $p = 2$, etc. jusqu'à trouver un entier p satisfaisant, c'est le nombre chromatique cherché.

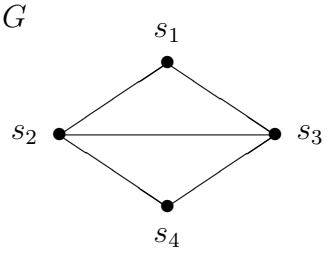
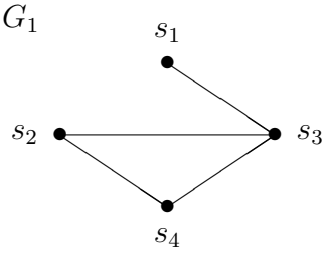
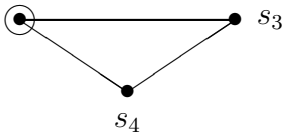
Ce programme naïf a le mérite d'exister pour démontrer la *calculabilité* du nombre chromatique, mais il serait tellement ruineux en temps de calcul que certainement personne n'a jamais réellement utilisé cette idée. Elle peut pourtant être améliorée comme suit. Prenons un sous-graphe G' de G contenant un certain nombre de sommets de G et toutes les arêtes correspondantes et supposons qu'on ait trouvé un bon coloriage à p couleurs de G'. Ajoutons à G' un sommet de G manquant dans G' et toutes les arêtes correspondantes : on obtient un graphe G'' qu'on tente de colorier, en utilisant le coloriage déjà trouvé pour G', et en donnant au nouveau sommet l'une des p couleurs considérées, de façon à ce que la condition sur les sommets reliés de G'' soit satisfaite. Il y a en général plusieurs façons de le faire ; pour la première d'entre elles on continue de même en ajoutant encore un sommet et des arêtes pour obtenir le graphe G'''... Si on n'aboutit pas, on essaiera une autre possibilité pour le dernier sommet de G'' et ainsi de suite. Si aucun de ces essais n'aboutit, c'est que le coloriage de G' n'était pas *prolongeable* et il faut réexaminer la question pour G', trouver un *autre* bon coloriage de G' en changeant la couleur de son *dernier* sommet, celui qui achevait de définir G', et réessayer la récurrence avec ce nouveau coloriage... Si un bon coloriage de G à p

couleurs existe, il peut être obtenu par ce procédé. Là encore on commence avec $p = 1$, puis $p = 2$, etc. jusqu'à trouver un entier p suffisant, le nombre chromatique cherché. Cette méthode, déjà plus rusée que la méthode naïve, est bien connue des informaticiens, qui l'appellent le *backtracking*, ou encore la méthode d'*essai et erreur*. Elle demande un peu de métier pour être programmée, voir à ce propos le chapitre 3 de l'excellent ouvrage de Niklaus Wirth, *Algorithms + data structures = programs* (Prentice-Hall, 1976).

Cette méthode, bien que meilleure que la méthode naïve est encore beaucoup trop longue pour être réellement utilisée pour des graphes un peu complexes. Les graphistes ont découvert une autre méthode d'une tout autre nature. Déjà dans les méthodes précédentes on essaie une *récurrence*, pour déterminer si le nombre p est suffisant. Dans cet ordre d'idées, la question suivante est naturelle : soit un graphe G ; enlevons-lui un sommet et les arêtes correspondantes pour obtenir un graphe un peu plus simple G', et supposons qu'on connaisse le nombre chromatique de G' ; peut-on déduire de cette seule information le nombre chromatique de G ? La réponse est négative : l'examen de quelques graphes simples montre qu'il ne peut pas y avoir de relation directe entre les nombres chromatiques des graphes G' et G et une récurrence simple basée sur le nombre de sommets du graphe ne peut donc être obtenue de la sorte.

A moins qu'on remplace l'information "nombre chromatique" par une autre information *considérablement plus riche*. C'est un phénomène un peu paradoxal mais fréquent en mathématiques, où l'on réussit assez facilement à trouver la solution d'un problème en apparence plus difficile, alors que le problème "simple" est sans solution. Associons à un graphe G la fonction $\chi_G : \mathbf{N} \to \mathbf{N}$ qui, à un entier p fait correspondre le nombre de bons coloriages de G à p couleurs. Cette fonction permet d'atteindre le nombre chromatique qui en apparaît comme un *sous-produit* : le nombre chromatique de G est le plus petit entier p tel que $\chi_G(p) \neq 0$; une fois connue la fonction χ_G, il est facile de trouver cet entier, d'autant plus qu'on va voir dans un instant que cette fonction est nécessairement un polynôme qu'on va appeler le *polynôme chromatique* de G.

On cherche donc à déduire le polynôme chromatique de G de polynômes chromatiques de graphes *immédiatement plus simples* que G. Si le graphe G n'a pas d'arête, (auquel cas c'est seulement un ensemble de n sommets non reliés), alors tout coloriage est un bon coloriage de G, de sorte que $\chi_G(p) = p^n$ et χ_G est donc déjà un polynôme. Sinon le graphe a au moins une arête, par exemple entre les sommets s_1 et s_2. Du graphe G on va déduire deux autres graphes G_1 et G_2 ; le premier, G_1, est obtenu simplement en supprimant l'arête entre s_1 et s_2 ; le second, G_2, est obtenu au contraire en *écrasant* cette même arête pour n'en faire, ainsi que de ses deux extrémités, qu'un seul sommet du graphe G_2 ; on notera aussi que si s_1 *et* s_2 étaient tous les deux reliés par des arêtes au sommet s_k, les *deux* anciennes arêtes n'en donnent qu'*une* dans le nouveau graphe.

G	G_1	G_2
		
$\chi_G(p) = p^4 - 5p^3 + 8p^2 - 4p$	$\chi_{G_1}(p) = p^4 - 4p^3 + 5p^2 - 2p$	$\chi_{G_2}(p) = p^3 - 3p^2 + 2p$
$\chi_G = \chi_{G_1} - \chi_{G_2}$		

Considérons alors un bon coloriage de G_1 ; ou bien les deux sommets s_1 et s_2 ont la même couleur auquel cas on déduit par "écrasement" un bon coloriage de G_2 et le coloriage correspondant de G n'est pas bon, ou bien au contraire ils sont de couleurs différentes, auquel cas on déduit un bon coloriage de G et par contre il n'est pas possible d'en déduire un coloriage de G_2. Il est facile de voir que tout bon coloriage de G et de G_2 peut être obtenu de la sorte. Si p est un nombre de couleurs, il en résulte aussitôt que $\chi_G(p) = \chi_{G_1}(p) - \chi_{G_2}(p)$ ou, si on préfère, puisque ceci est vrai pour tout entier p, que $\chi_G = \chi_{G_1} - \chi_{G_2}$, relation entre polynômes ; et on tient la relation de récurrence cherchée. On en déduit déjà, par récurrence, que le polynôme chromatique... est bien un polynôme ! Une fois ceci compris, c'est un jeu d'enfant d'en déduire un programme récursif sachant calculer χ_G quand on lui donne G en entrée. On peut alors en tirer comme sous-produit le nombre chromatique de G. Ce dernier programme, basé sur une récurrence simple, est considérablement meilleur que celui qui avait été précédemment expliqué.

Mais il est peut-être temps de revenir à notre sujet, *infini et effectivité*. Quel rapport avec notre problème de graphe ? On a insisté sur le fait que l'information *polynôme chromatique* est beaucoup plus riche que l'information *nombre chromatique* : d'une certaine façon elle contient une *infinité* d'informations puisque ce polynôme est défini pour *n'importe quel* nombre de couleurs p. Mais bien sûr cette *infinité* d'informations ne peut être décrite telle quelle : le *truc fonctionnel* est utilisé. Plutôt que de considérer l'*ensemble* de tous les couples $(p, \chi_G(p))$, on préfère considérer la *fonction* χ_G. Pour un mathématicien bourbakiste, il n'y a d'ailleurs aucune différence, puisqu'il définit une fonction comme un ensemble de couples argument-valeur ! mais pour l'informaticien c'est tout différent, car il peut coder cette *infinité* de couples comme un *polynôme*, donc sous forme d'une suite finie de coefficients et d'exposants *qu'il peut loger sur sa machine* ! Ce polynôme peut être considéré comme un *programme*, un texte fini, *capable*, si on le lui demande, de donner une réponse, une valeur, quelle que soit la donnée entière (ici un nombre de couleurs) qu'on lui communique. Ce schéma général de travail ne veut bien

fonctionner que parce qu'on est capable d'écrire et d'utiliser des programmes admettant comme données des polynômes (programmes) et fournissant comme sortie un autre polynôme (programme) ; ceci est indispensable pour utiliser la formule de récurrence. Ce genre de travail est facile pour les polynômes mais un peu plus difficile pour les algorithmes de nature beaucoup plus générale qu'on examinera plus loin.

Si on a décrit si soigneusement ce phénomène et sa solution, c'est parce qu'ils sont très proches de ce que l'on décrira dans la dernière section dans un domaine de recherche très actif mais plus ésotérique, celui de la *topologie algébrique* : les deux problèmes (et leur solution) sont à vrai dire de natures identiques.

3 Manipuler sur machine des objets fonctionnels.

Les sections précédentes et les encadrés 1 et 2 peuvent se résumer ainsi : un objet de nature apparemment infinie peut parfois être codé sous forme d'un texte fini représentant une certaine *fonction*. L'exemple du polynôme chromatique montre cependant que cette *astuce* n'est véritablement productive que si on dispose de moyens permettant, à partir de codages fonctionnels, d'en *calculer* d'autres. Précisons ce point.

Dans l'exemple du polynôme chromatique, on traite des fonctions dont la source et le but sont l'ensemble $\mathbf{N}$ des entiers. Il se trouve que ces fonctions sont en fait polynômiales, ce qui permet de les écrire — il faudrait mieux dire de les *coder* — sous forme d'un texte fini constitué de signes, de coefficients, d'une lettre à vrai dire sans importance (par exemple x), et d'exposants. Des conventions très simples permettent de représenter un tel polynôme par exemple comme une chaîne de caractères ASCII. Le programme *polynôme chromatique* doit alors être capable de déterminer le polynôme *différence* de deux polynômes donnés, afin de pouvoir exploiter la formule de récurrence expliquée. Plus précisément, on doit pouvoir écrire dans le langage de programmation utilisé une fonction à deux arguments (les deux *représentations* des polynômes), qui fournisse la représentation du polynôme différence. Ceci est un exercice facile, quel que soit le langage de programmation utilisé.

Facile parce que nos fonctions sont très particulières : ce sont des fonctions polynômiales. Cet exercice devient beaucoup plus difficile si on veut pouvoir traiter de la sorte des fonctions *quelconques*. La première question qui se pose est celle du *codage* : comment *coder*, sous forme d'une chaîne de caractères *finie* une fonction quelconque dont source et but sont l'ensemble $\mathbf{N}$ des entiers ? L'alphabet devant être fini, l'ensemble de ces chaînes de caractères est *dénombrable*, alors que, au contraire, l'ensemble des fonctions $\mathbf{N} \to \mathbf{N}$ a la puissance du continu ! Quelle que soit l'ingéniosité du système d'écriture, on ne pourra donc coder qu'une partie de l'ensemble de ces fonctions.

Quelles fonctions choisir ? comment les *écrire* ? comment écrire un programme

faisant correspondre par exemple aux textes de deux telles fonctions, le texte de la fonction *différence* ? Les logiciens avaient, bien avant les informaticiens (et même avant que l'informatique n'existe !) beaucoup réfléchi sur ces questions. Le bon ensemble de fonctions est l'ensemble des fonctions récursives, constituant en un certain sens un tout petit sous-ensemble de l'ensemble de toutes les fonctions, mais en un certain sens encore, ce sont les seules qui soient intéressantes ! Diverses méthodes ont été imaginées pour les écrire ; elles ont toutes été démontrées équivalentes quant à leurs possibilités.

L'une des plus intéressantes est celle du *lambda-calcul*. Par chance, le précédent numéro d'Images des Mathématiques contenait justement un excellent article de Michel Parigot intitulé *Preuves et programmes : les mathématiques comme langage de programmation*, expliquant entre autres choses ce qu'est le lambda-calcul ; voir en particulier la section *le lambda-calcul comme langage machine* et l'encadré 2 *la normalisation et le lambda-calcul*. S'il n'a pas le courage de s'y reporter, le lecteur pourra se contenter des indications données dans l'encadré 3 du présent article.

Le lambda-calcul fut un peu perdu de vue au début de ce demi-siècle, contrairement à la machine de Turing qui figure rituellement au début de tout cours d'informatique théorique. Il réapparut d'abord à la faveur des travaux de l'informaticien américain McCarthy qui créa le langage *Lisp* à la fin des années 50. Ce langage de programmation qui passait au début pour une simple curiosité sans possibilité d'applications concrètes dignes de ce nom est en effet directement inspiré du lambda-calcul. Comme tout système de mathématiques formelles, le lambda-calcul produit rapidement des *termes* dont la longueur est telle qu'elle exclut toute utilisation pratique : très vite, cette longueur dépasse par exemple le nombre d'atomes de notre galaxie ! McCarthy a donc réfléchi sur les méthodes (essentiellement l'utilisation des symboles comme abréviations) qui permettraient de tourner cette difficulté. Ce fut un travail long et difficile ; quarante ans plus tard il est clair que dans sa version la plus moderne, *Common Lisp*, c'est l'un des plus puissants langages de programmation qui soit actuellement disponible ! Par exemple les meilleurs systèmes de calcul formel tels que Macsyma, Reduce, Scratchpad, logiciels d'une très grande complexité, reposent sur Lisp.

Puisque Lisp est directement inspiré du lambda-calcul, il peut en effet parfaitement, *en cours d'exécution d'un programme*, créer des programmes capables à leur tour de travailler sur d'autres programmes pour en créer encore d'autres... Ceci lui donne des possibilités tout-à-fait inaccessibles aux langages ordinaires de programmation de style Pascal (Fortran, Ada, C, etc.). Et ce sont justement les possibilités dont on a besoin pour résoudre les problèmes de traitement d'objets *codes fonctionnels* que l'on considère ici. Par exemple la fonction Lisp qui à partir de deux fonctions de **Z** à valeurs dans **Z** fournit la fonction différence s'écrit assez simplement :

```
(setf sub-functions
  #'(lambda (f g)
      #'(lambda (n) (- (funcall f n) (funcall g n)))))
```

texte qu'on peut lire ainsi : la fonction `sub-functions` nécessite deux arguments `f` et `g` (deux fonctions) et leur fait correspondre la fonction qui, à un entier `n`, fait correspondre la différence de deux entiers obtenus en faisant travailler `f` puis `g` sur l'entier `n`. On ne cherchera pas ici à expliquer la présence des signes cabalistiques `#` et `'` ; ils permettent de résoudre des problèmes d'optimisation et de portée d'identificateurs inconnus des langages ordinaires et qu'on préfère laisser dans l'ombre.

Dans la suite d'instructions Lisp :

```
(setf f1
  #'(lambda (x) (* x 3)))
(setf f2 #'(lambda (x) (* x x)))
(setf f3
  (funcall sub-functions f-1 f-2))
```

on définit successivement la fonction $f_1(x) = 3x$, la fonction $f_2(x) = x^2$, puis, *à partir des codes de ces fonctions*, Lisp construit lui-même le code de la fonction différence $f_3 = f_1 - f_2$.

On peut continuer indéfiniment dans le même esprit et écrire par exemple une fonction dont l'argument est un *opérateur binaire sur les entiers* ; cette fonction va fournir la fonction capable de travailler sur les paires de fonctions $\mathbf{N} \to \mathbf{N}$ selon l'opérateur en question :

```
(setf create-op-function
  #'(lambda (operator)
      #'(lambda (f g)
          #'(lambda (n)
              (funcall operator (funcall f n) (funcall g n))))))
```

où on peut lire entre autres "fournira la fonction qui, à deux fonctions f et g, associera la fonction qui, à n,...". Et au lieu de définir notre fonction `sub-functions` comme plus haut, on peut alors tout simplement obtenir le même résultat par :

```
(setf sub-functions (funcall create-op-function #'-))
```

Encadré 3.
Le lambda-calcul.

Dans le monde du lambda-calcul n'existent que des *fonctions*, contrairement à celui de la programmation ordinaire où, notamment dans les cours pour débutants (penser aux cartes perforées d'antan), on distingue soigneusement le *programme* (un certain texte) des *données* (un autre texte sur lequel le programme

doit travailler). En lambda-calcul n'existent que des fonctions qui peuvent servir indifféremment de programmes ou de données. Un mécanisme, la *réduction* (décrit succintement dans l'article de M. Parigot) définit par quel processus le couplage de deux fonctions, la première considérée comme programme, la seconde comme donnée, met en marche une machine théorique qui, éventuellement, produit un résultat (encore une fonction) à considérer comme le résultat de la fonction-programme travaillant sur la fonction-donnée. Il peut se faire aussi que la machine tourne sans fin ; le résultat est alors indéfini.

Une fonction du lambda-calcul est un texte utilisant un alphabet tout-à-fait banal constitué de lettres et de quelques signes ad hoc, et obéissant à quelques règles (une grammaire) très simples. On peut, si on veut, considérer un tel texte comme un programme écrit dans le langage lambda-calcul. Cette unicité de nature, tout objet devant être une fonction, oblige à quelques acrobaties quand il faut traiter de données ordinaires telles qu'un nombre entier. Le truc du lambda-calcul consiste à coder un entier n comme la fonction faisant correspondre à une fonction quelconque f la fonction :

$$\underbrace{f \circ f \circ f \circ \ldots \circ f}_{n}$$

Cette façon de faire peut paraître très compliquée pour coder un objet aussi simple qu'un entier. Toujours est-il qu'il est parfaitement possible de programmer en lambda-calcul toutes les fonctions récursives ; l'encadré 2 de l'article de M. Parigot explique très en détail comment en lambda-calcul on réalise ainsi l'addition de 2 et de 2 !

L'intérêt du lambda-calcul consiste en ce que, considéré comme langage de programmation, on peut y écrire des programmes *capables de travailler sur des données-programmes et de produire des résultats-programmes.*

C'est une histoire très curieuse que celle du lambda-calcul. Il a été imaginé et mis au point par les logiciens des années 30 pour formaliser la composante algorithmique des mathématiques d'une façon suffisamment simple, permettant ainsi de répondre négativement à la conjecture de Hilbert sur l'existence d'un algorithme universel capable de résoudre les problèmes mathématiques. La démonstration s'inspire de celle du théorème d'incomplétude de Gödel et nécessite d'autoriser la possibilité d'énoncés pouvant travailler sur eux-mêmes. En terme d'algorithmique, il faut envisager des programmes capables de travailler sur eux-mêmes. Mais ceci est évidemment impossible si, dans son environnement de programmation, on sépare soigneusement programmes et données !

La solution de Church fut de créer un modèle algorithmique très ingénieux où ne figurent que des programmes : c'est le lambda-calcul ; Church put ainsi contredire la conjecture de Hilbert. Turing arriva au même résultat en construisant au contraire un modèle algorithmique (machine de Turing) où un programme n'est rien d'autre qu'une donnée ! C'est ainsi que Turing découvrit la notion même et la réalisation théorique d'une machine universelle, ce qui est à la base de toute la science informatique moderne. On peut par ailleurs démontrer qu'en un certain sens les solutions de Church et Turing sont équivalentes.

4 Complexes simpliciaux.

On espère que la section précédente aura rassuré le lecteur quant aux possibilités de traiter sur machine, même *au cours de* l'exécution d'un programme, des objets de nature fonctionnelle. Dans cette section on explique comment il est possible d'utiliser le truc fonctionnel pour coder des objets *géométriques* pouvant être tout-à-fait monstrueux. Jouer avec des monstres sur machine n'est pourtant pas un but en soi et les quelques monstres exhibés dans cette section n'ont en fait aucun intérêt réel ; dans la section suivante on expliquera comment les mêmes méthodes permettent de travailler sur machine avec des espaces hautement infinis et *vraiment utiles* (du moins pour les mathématiciens !). L'exemple des espaces de lacets, facile à comprendre, est idéal pour illustrer notre propos.

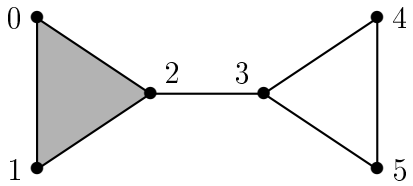
Expliquons d'abord ce qu'est un *complexe simplicial*. La définition en est *combinatoire*, mais à tout complexe simplicial est associé un objet *géométrique*, celui qu'on veut en fait modéliser par la définition combinatoire : un *complexe simplicial* K est un couple $K = (V, S)$ où V est un ensemble quelconque, l'ensemble des *sommets* de K, et S est un ensemble de parties *finies* de V, l'ensemble des *simplexes* de K. Ces données doivent satisfaire les propriétés suivantes :

1) Si v est un sommet de K, autrement dit si $v \in V$, alors $\{v\} \in S$;
2) Si s est un simplexe de K, autrement dit si $s \in S$, et si $s' \subset s$, alors $s' \in S$: tout sous-simplexe de K est encore un simplexe de K.

Posons par exemple :

$K_1 = (V_1, S_1)$, avec :
$V_1 = \{0,1,2,3,4,5\}$, et
$S_1 = \{\{\},$
$\{0\},\{1\},\{2\},\{3\},\ \{4\},\{5\},$
$\{0,1\},\{0,2\},\{1,2\},\{2,3\},\ \{3,4\},\{3,5\},\{4,5\},$
$\{0,1,2\}\}$.

Le complexe K_1 a six sommets et quinze simplexes. On peut intuitivement lui associer l'objet géométrique représenté sur la figure suivante :



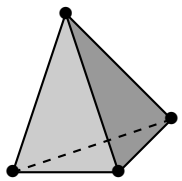
Le triangle 012 est *plein* alors que le triangle 345 est *creux*. La méthode de correspondance entre listes de sommets et simplexes, d'une part, et objet géométrique, d'autre part, est claire : la figure associée à un complexe simplicial a autant de "points marqués" qu'il y a de sommets dans le complexe ; deux points marqués sont reliés par un segment si l'ensemble de ces deux sommets figure dans la liste des simplexes ; trois sommets soustendent un triangle plein si l'ensemble de ces

trois sommets figure dans la liste des simplexes ; etc. On démontre que quitte à se placer dans un espace de dimension suffisamment grande, on peut toujours associer à un complexe simplicial, qu'on appelle quelquefois complexe simplicial *abstrait*, un objet géométrique de cette nature qu'on appelle alors complexe simplicial *géométrique*.

Autre exemple ; au complexe simplicial abstrait :

$K_2 = (V_2, S_2)$, avec :
$V_2 = \{0,1,2,3\}$, et
$S_2 = \{\{\},$
$\{0\},\{1\},\{2\},\{3\},$
$\{0,1\},\{0,2\},\{0,3\}, \{1,2\},\{1,3\},\{2,3\},$
$\{0,1,2\},\{0,1,3\}, \{0,2,3\},\{1,2,3\}\}$.

on peut associer le tétraèdre *creux* :



Il est creux parce que dans la liste de ses simplexes, *ne figure pas* le simplexe $\{0,1,2,3\}$; sinon l'objet géométrique associé aurait été un tétraèdre *plein*.

Un complexe simplicial *fini* peut facilement être codé sur machine sous forme de listes de simplexes ; en fait on peut se contenter de donner les simplexes maximaux, les autres pouvant en être déduits. Ainsi notre premier exemple de complexe simplicial pourrait être codé :

```
((0 1 2) (2 3) (3 4) (3 5) (4 5))
```

alors que le tétraèdre creux s'écrirait :

```
((0 1 2) (0 1 3) (0 2 3) (1 2 3))
```

et le tétraèdre plein :

```
((0 1 2 3))
```

Mais rien n'empêche de considérer des complexes simpliciaux *infinis*. Considérons par exemple le complexe $K_3 = (V_3, S_3)$ où V_3 est l'ensemble des entiers naturels et S_3 l'ensemble de toutes les parties finies de $\mathbf{N}$:

$K_3 = (V_3, S_3)$, avec :
$V_3 = \{0,1,2,3,4,5,6,7,8,\ldots\}$, et
$S_3 = \{\{\},$
$\{0\},\{1\},\{2\},\{3\},\{4\},\{5\},\{6\},\{7\},\ldots$
$\{0,1\},\{0,2\},\{0,3\},\{0,4\},\{0,5\},\{0,6\},\ldots$
$\{1,2\},\{1,3\},\{1,4\},\{1,5\},\{1,6\},\ldots$
$\{2,3\},\{2,4\},\{2,5\},\{2,6\},\ldots$
$\ldots$
$\{0,1,2\},\{0,1,3\},\{0,1,4\},\{0,1,5\},\ldots$
$\{0,2,3\},\ldots$
$\ldots$
$\{0,1,2,3\},\ldots$
$\ldots$
$\ldots\}$

L'objet géométrique associé contient un segment pour tout couple d'entiers différents, un triangle plein pour tout triplet d'entiers différents deux à deux, un tétraèdre plein pour tout quadruplet d'entiers différents deux à deux, etc. Bien entendu on ne peut représenter un tel objet dans $\mathbf{R}^3$, mais il n'est pas difficile de définir rigoureusement dans $\mathbf{R}^\infty$ un objet géométrique correspondant au complexe simplicial K_3. Autre exemple, en quelque sorte *sous-exemple* du précédent : on peut prendre $K_4 = (V_4, S_4)$ où V_4 est encore l'ensemble $\mathbf{N}$ des entiers naturels, et S_4 est l'ensemble des parties de $\mathbf{N}$ qui contiennent *au plus* deux éléments :

$K_4 = (V_4, S_4)$, avec :
$V_4 = \{0,1,2,3,4,5,6,7,8,\ldots\}$, et
$S_4 = \{\{\},$
$\{0\},\{1\},\{2\},\{3\},\{4\},\{5\},\{6\},\{7\},\ldots$
$\{0,1\},\{0,2\},\{0,3\},\{0,4\},\{0,5\},\{0,6\},\ldots$
$\{1,2\},\{1,3\},\{1,4\},\{1,5\},\{1,6\},\ldots$
$\{2,3\},\{2,4\},\{2,5\},\{2,6\},\ldots$
$\ldots$
$\ldots\}$

Cette fois l'objet géométrique associé contiendra une infinité de segments, mais par contre aucun triangle, aucun tétraèdre,...

Bien entendu il est absolument impossible de représenter sur machine de tels complexes simpliciaux à l'aide de listes de simplexes ; on ne peut installer sur machine que des listes de longueur finie et les listes dont on aurait besoin pour les complexes K_3 et K_4 sont infinies. Etant donné les préparatifs des sections précédentes, le lecteur devine certainement qu'on va utiliser le *truc fonctionnel* pour tourner cette difficulté. Comment procéder ?

On décide qu'un codage fonctionnel pour un complexe simplicial est une fonction f capable de travailler sur une liste quelconque d'objets de la machine et de répondre alors **vrai** ou **faux**. Cette fonction doit de plus satisfaire la condition suivante : si $f(l) =$ **vrai** et si $l' \subset l$, alors $f(l') =$ **vrai**. Il est facile d'associer à

une telle fonction f un complexe simplicial : soit V_f l'ensemble des objets v tels que $f(\{v\}) = \textbf{vrai}$, et S_f l'ensemble de toutes les listes pour lesquelles f répond **vrai**. Alors $K_f = (V_f, S_f)$ est un complexe simplicial appelé complexe simplicial associé à f. Par ce procédé très simple on code *fonctionnellement* les complexes simpliciaux. Puisque la fonction f peut, *potentiellement*, travailler sur une infinité d'objets (voir l'encadré 1), rien n'empêche de coder ainsi des complexes simpliciaux infinis.

Les différents exemples de complexes qui ont été donnés précédemment peuvent être codés en Lisp comme suit :

```
(setf K1
  #'(lambda (list)
      (or (subsetp list '(0 1 2))
          (subsetp list '(2 3))
          (subsetp list '(3 4))
          (subsetp list '(4 5))
          (subsetp list '(3 5)))))
```

Si on interroge `K1` pour la liste `(0 2)`, il répondra **vrai**, alors qu'il répondra **faux** par exemple pour la liste `(0 3)`. Le tétraèdre creux peut être codé :

```
(setf K2
  #'(lambda (list)
      (and      (subsetp  list     '(0 1 2 3))
           (not (subsetp '(0 1 2 3) list)))))
```

On demande là que tous les éléments de la liste argument soient extraits de la liste `(0 1 2 3)`, mais tous les éléments de cette dernière liste ne doivent pas y figurer simultanément ; si on enlevait cette dernière condition, on aurait le code fonctionnel du tétraèdre plein. Le code fonctionnel du complexe K_3, infini, n'est pas plus long ; il est même plutôt plus court :

```
(setf K3
  #'(lambda (list)
      (every #'integerp list)))
```

Il suffit de vérifier que tout élément de la liste argument est un entier, ce qui, comme on le voit, s'écrit très simplement. Ce que les topologues appellent le 1-squelette de K_3, c'est-à-dire le complexe K_3 d'où on retire tous les simplexes de dimension > 1, autrement dit notre complexe K_4, admet le code fonctionnel suivant :

```
(setf K4
   #'(lambda (list)
       (and (every #'integerp list)
            (< (length (remove-duplicates list))
               3))))
```

En effet il faut cette fois vérifier en outre que le nombre des sommets *différents* dans la liste est inférieur à 2.

Cette description de K_4 comme le 1-squelette de K_3 donne aussitôt une occasion d'illustrer les possibilités fonctionnelles de Lisp : on aimerait bien pouvoir disposer d'une fonction à laquelle on passe deux arguments, d'une part un code fonctionnel d'un complexe simplicial K, d'autre part une dimension d ; on voudrait que cette fonction *fabrique* un code fonctionnel du d-squelette de K, le nouveau complexe simplicial obtenu à partir de K en *supprimant* tous les simplexes de dimension $> d$. Ceci est très facile :

```
(setf squelette
  #'(lambda (complexe dimension)
      #'(lambda (list)
          (and (funcall complexe list)
               (< (length (remove-duplicates list))
                  (+ 2 dimension))))))
```

si bien qu'au lieu de se fatiguer à écrire un code fonctionnel de K_4, on aurait pu demander à la machine Lisp de le faire elle-même :

```
(setf K4 (funcall squelette K3 1))
```

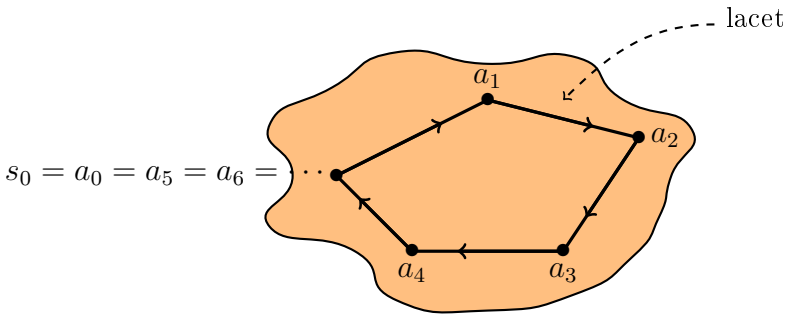
5 Construire un espace de lacets sur machine.

Soit $K = (V, S)$ un complexe simplicial et s_0 l'un de ses sommets auquel on va faire jouer un rôle particulier et qu'on appelle le *point base* de K. On suppose désormais que K est *connexe*, autrement dit *d'un seul tenant* : il faut comprendre ici que, en partant du point base, n'importe quel autre sommet est *accessible* en cheminant le long des arêtes de K. L'espace des lacets de (K, s_0), qu'on note $\Omega(K, s_0)$, est un autre complexe simplicial, défini à partir de K et s_0, qui est toujours (sauf si $V = \{s_0\}$) infini. Cet espace de lacets joue un rôle particulièrement important en topologie algébrique : il fut inventé par Jean-Pierre Serre au début des années 50 et joue en un certain sens le rôle d'*espace inverse* de l'espace K. Il n'est pas question de définir ici en quel sens, mais toujours est-il que c'est entre autres ce type de construction qui permit à Jean-Pierre Serre d'obtenir des progrès considérables en topologie algébrique (groupes d'homotopie des sphères), travaux qui lui valurent la médaille Fields.

Cet espace de lacets n'est pas difficile à définir et donne justement un exemple frappant des possibilités offertes par les méthodes de *codage fonctionnel*. Le complexe $K = (V, S)$ étant donné, ainsi qu'un sommet s_0 élément de V, le *complexe* $\Omega(K, s_0)$, *espace de lacets de* (K, s_0) est, comme tout complexe, défini par son ensemble de sommets V' et son ensemble de simplexes $S' : \Omega(K, s_0) = (V', S')$.

Décrivons d'abord l'ensemble des sommets V'. Un sommet de $\Omega(K, s_0)$ est un *lacet* de K *basé* en s_0, on veut dire par là une suite infinie de sommets $(a_0, a_1, a_2, \ldots, a_{k-1}, a_k, \ldots)$ vérifiant les conditions suivantes :

a) le premier sommet de cette suite doit être le point base s_0 de K : $a_0 = s_0$;
b) il doit exister un entier n tel que, si $k \geq n$, alors $a_k = s_0$;
c) pour tout entier $k > 0$, $\{a_{k-1}, a_k\}$ est un simplexe de K : $\{a_{k-1}, a_k\} \in S$ (répétition permise, auquel cas ce simplexe n'a en fait qu'un seul élément).



Cette suite doit être comprise comme un *cheminement* sur K, plus précisément le long des arêtes de K ; à l'instant 0, on part du point base : $a_0 = s_0$; à l'instant 1, on atteint le sommet a_1 de K, ce qui, pour être légitime nécessite que $\{a_0, a_1\}$ soit un simplexe de K ; intuitivement, entre les instants 0 et 1, le lacet parcourt l'*arête* de K située entre a_0 et a_1. De même, entre les instants $k-1$ et k, le lacet parcourt l'arête de K située entre a_{k-1} et a_k, ce qui nécessite que $\{a_{k-1}, a_k\}$ appartienne à S. On demande encore que ce voyage soit essentiellement fini, c'est le rôle de la condition b) : au bout d'un certain temps n, le lacet reste fixe au point base s_0 de K. Par exemple si K est le complexe exemple K_1 de la section précédente, muni du point base $s_0 = 2$ la suite suivante $\lambda = (2, 3, 4, 5, 3, 4, 5, 3, 2, 2, 2, \ldots)$ est un exemple de *sommet* de $\Omega(K_1, 2)$.

Ainsi la donnée d'*un seul* sommet de $\Omega(K, s_0)$ nécessite plusieurs sommets et arêtes de K. Définissons maintenant les simplexes de $\Omega(K, s_0)$. Par exemple soient $\lambda = (a_0, a_1, \ldots)$ et $\lambda' = (a'_0, a'_1, \ldots)$ deux sommets de $\Omega(K, s_0)$; dans quel cas le couple $\{\lambda, \lambda'\}$ est-il un simplexe de $\Omega(K, s_0)$? La condition qui doit être satisfaite est la suivante :

> Pour tout entier $k > 0$, l'ensemble $\{a_{k-1}, a_k, a'_{k-1}, a'_k\}$ (répétitions permises) doit être un simplexe de K : $\{a_{k-1}, a_k, a'_{k-1}, a'_k\} \in S$.

L'interprétation de cette condition n'est pas difficile ; elle demande qu'on soit capable de définir un chemin *intermédiaire* entre les chemins $\lambda = (a_0, a_1, \ldots)$ et $\lambda' = (a'_0, a'_1, \ldots)$. A l'instant $k \in \mathbf{N}$, ce chemin intermédiaire devrait passer au

milieu du segment $[a_k, a'_k]$, ce qui nécessite déjà pour être défini que $\{a_k, a'_k\}$ appartienne à S. Mieux, à l'instant $k-\frac{1}{2}$, le chemin intermédiaire doit passer au milieu du segment $[\lambda(k-\frac{1}{2}), \lambda'(k-\frac{1}{2})]$, qui est aussi le barycentre des quatre points $\lambda(k-1)$, $\lambda(k)$, $\lambda'(k-1)$ et $\lambda'(k)$, autrement dit a_{k-1}, a_k, a'_{k-1} et a'_k ; c'est pourquoi, pour que ce barycentre soit défini, on demande que ces quatre points définissent un simplexe du complexe K (répétitions permises). Par exemple, toujours avec le complexe exemple K_1 de la section précédente, si on prend $\lambda = (2, 3, 4, 5, 3, 2, 2, 2, \ldots)$ et $\lambda' = (2, 3, 3, 4, 5, 3, 2, 2, \ldots)$, cette condition n'est pas satisfaite au temps 3 puisque $\{3, 4, 5\}$ n'appartient pas à S et donc $\{\lambda, \lambda'\}$ *n'est pas* un simplexe de $\Omega(K_1, 2)$; au contraire, si on prend $\mu = (2, 0, 1, 2, 2, 2, \ldots)$ et $\mu' = (2, 2, 0, 1, 2, 2, \ldots)$, la condition est toujours satisfaite, essentiellement parce que $\{0, 1, 2\}$ appartient à S, et donc $\{\mu, \mu'\}$ est un élément de S' : $\{\mu, \mu'\}$ *est* un simplexe de $\Omega(K_1, 2)$.

Plus généralement et dans le même esprit, soit $\lambda^i = (a^i_0, a^i_1, a^i_2, \ldots)$, $1 \leq i \leq m$, une famille de lacets éléments de $\Omega(K, s_0)$; cette famille constituera un simplexe de $\Omega(K, s_0)$ si et seulement si, pour tout entier $k > 1$, la famille $\{a^1_{k-1}, a^1_k, a^2_{k-1}, a^2_k, \ldots, a^m_{k-1}, a^m_k\}$ appartient à S ; ce qui, intuitivement, permet de définir un chemin *barycentre* des chemins $\lambda^1, \ldots, \lambda^m$.

Examinons maintenant la possibilité de construire sur machine une fonction à deux arguments, d'une part le code fonctionnel d'un complexe simplicial, d'autre part l'un de ses sommets (point base), qui fournisse le code fonctionnel de son espace de lacets. Ce dernier code doit être capable de travailler sur des listes et doit répondre par oui ou par non à la question : *cette liste est-elle un simplexe du complexe espace de lacets* ? Chaque élément de liste doit au moins être un sommet, ce qui pose un petit problème, puisqu'on a défini un sommet de l'espace de lacets comme une suite *infinie*. Mais ce problème est facile à surmonter puisque la condition b) impose qu'à partir d'un certain rang, variable d'une suite à l'autre, tous les termes de cette suite sont égaux au point base ; il suffit dès lors de convenir qu'on représente une telle suite sous forme d'une liste *finie*, tous les termes manquants étant égaux au point base. L'examen des conditions à satisfaire pour qu'une liste de listes représente (modulo cette convention) un simplexe de notre espace de lacets est alors un petit programme utilisant entre autres le *code fonctionnel du complexe original* ; la transformation de programme du code fonctionnel d'un complexe vers le code fonctionnel de son espace de lacets peut elle-même être programmée, par exemple par le programme suivant qu'on montre seulement pour satisfaire l'éventuelle curiosité du lecteur :

```
(setf loop-space
  #'(lambda (K s0)
      #'(lambda (ll)
          (and (maplistp ll)
               (let ((ll (transpose (complete ll s0))))
                 (essential-test K ll))))))
```

La fonction `loop-space` utilise diverses fonctions auxiliaires de traitement de listes (`maplistp`, `transpose`, `complete`, `essential-test`) dont l'écriture est

Graphes	**Topologie algébrique**
Graphe G	Ensemble simplicial K
Nombre chromatique	Homologie ordinaire
Nombre chromatique connu N_G Construction simple G donne G' $N_{G'} = ? ? ?$	Homologie ordinaire connue $H_\star(K)$ Construction simple K donne K' $H_\star(K') = ? ? ?$
Polynôme chromatique χ_G	Homologie effective $EH_\star(K)$
Le polynôme chromatique contient une infinité d'informations	L'homologie effective contient une infinité d'informations
Informations codées sous forme *fonctionnelle* ; sur machine, ces informations apparaissent comme une chaîne *finie* de bits	Informations codées sous forme *fonctionnelle* ; sur machine, ces informations apparaissent comme une chaîne *finie* de bits
Le polynôme chromatique contient le nombre chromatique comme *sous-produit*	L'homologie effective contient l'homologie ordinaire comme *sous-produit*
Construction G_1, G_2 donne G Algorithme N_1, N_2 donne N	Construction K_1, K_2 donne K Algorithme EH_1, EH_2 donne EH

Tableau 1 : Polynôme chromatique — Homologie effective

un banal exercice de programmation. Moyennant quoi un très modeste micro-ordinateur *calcule* en moins d'un centième de seconde le code fonctionnel d'un espace de lacets, espace pourtant hautement infini !

6 Applications à la topologie algébrique.

On décrit très brièvement dans cette dernière section les applications que l'on peut obtenir à partir de cette méthode du codage fonctionnel quand on l'applique à la topologie algébrique. La situation est très analogue à celle qu'on a exposée pour le polynôme chromatique et le parallélisme est résumé dans le Tableau 1.

L'objet de la topologie algébrique est d'associer à des espaces des objets algébriques capables essentiellement de mesurer certaines de leurs propriétés. Dans cet ordre d'idées, des groupes très importants sont les *groupes d'homologie* dont la définition précise est trop ésotérique pour être expliquée ici. D'une certaine façon

ces groupes mesurent comment un espace est *troué*. Par exemple les topologues expliquent que le premier groupe d'homologie du plan euclidien $\mathbf{R}^2$, noté $H_1(\mathbf{R}^2)$, est nul, alors que si D est le disque unité de ce plan, $H_1(\mathbf{R}^2 - D)$ n'est pas nul, ce qui exprime que $\mathbf{R}^2 - D$ est *troué* ; dans le cas présent ceci se voit sur le groupe H_1 puisque le trou en question peut être enfermé dans un cercle, objet localement de dimension 1. Si on procédait de même pour l'espace euclidien $\mathbf{R}^3$ et sa boule unité B, ce serait cette fois le groupe H_2 qui ne serait pas nul, parce que le trou correspondant peut être enfermé dans une sphère, objet localement de dimension 2. Les topologues algébriques arrivent à définir très rigoureusement ces notions qui produisent pour un espace K des groupes d'homologie $H_n(K)$ pour chaque entier positif n.

Le calcul des groupes d'homologie des nombreux espaces intéressant les topologues est un sport difficile et qui reste aujourd'hui un sujet de recherche très actif. C'est à ce type de calcul que les méthodes de codage fonctionnel ont apporté récemment des progrès significatifs, un peu comme la méthode du polynôme chromatique permet d'améliorer le calcul du nombre chromatique. Le parallèle entre les deux situations est assez bien résumé dans le Tableau 1.

Il y a pourtant une petite différence dans le cas de la topologie algébrique : de nombreuses méthodes (suites exactes et suites spectrales notamment) avaient déjà été mises au point par les topologues, pour que, quand on construit un espace K à partir d'espaces K_1 et K_2 dont les homologies sont connues, on puisse avoir au moins quelques informations sur l'homologie de K ; souvent on arrive même à en déduire entièrement l'homologie de K, mais bien souvent aussi, ces méthodes se révèlent insuffisantes.

Les méthodes de l'*homologie effective*, mises au point par l'auteur en collaboration avec d'autres chercheurs (voir les références en fin d'article) permettent de tourner les difficultés que présentent les autres méthodes. Le cadre de cet article ne permet pas de donner beaucoup plus d'explications. Disons seulement que les objets de l'*homologie effective* contiennent, comme un polynôme chromatique, une infinité d'informations, mais le *truc fonctionnel* permet néanmoins de les manipuler sans difficulté particulière sur les machines théoriques et concrètes. Les méthodes classiques de la topologie algébrique sont donc transformées en vrais *algorithmes* capables de calculer les groupes convoités.

De nombreux résultats de calculabilité ont déjà été obtenus de la sorte, notamment pour les groupes d'homologie des espaces de lacets *itérés*. Le plus intéressant serait bien entendu de réussir à obtenir, par implémentation de ces méthodes sur des machines concrètes, le calcul de groupes d'homologie ou d'homotopie qui ont jusque là résisté. Un gros travail est en cours dans cette direction, précisément pour l'homologie des espaces de lacets itérés.

Le champ de recherche ainsi ouvert, qui va des mathématiques les plus pures (algèbre homologique et topologie algébrique), aux problèmes très concrets de réalisation d'algorithmes sur machine est passionnant. Il ouvre aussi des horizons insoupçonnés aux chercheurs en complexité (que dire de la complexité des algo-

rithmes à base de programmation fonctionnelle ?) et en calcul parallèle.

Pour en savoir plus :

- Francis SERGERAERT. *Homologie effective, I et II.* C. R. Acad Sc. Paris, Série I, 1987, vol. 304, pp 279-282 et 319-321.
- Julio RUBIO, Francis SERGERAERT. *Homologie effective et suites spectrales d'Eilenberg-Moore.* C. R. Acad. Sc. Paris, Série I, 1988, vol. 306, pp 723-726.
- Francis SERGERAERT. *The computability problem in algebraic topology.* Prépublication de l'Institut Fourier, n° 119, 1988.
- Julio RUBIO. *Homologie effective des espaces de lacets itérés.* Prépublication de l'Institut Fourier, n° 138, 1989.

-o-o-o-o-o-o-o-