ECOLE NATIONALE SUPERIEURE
DES MINES DE SAINT-ETIENNE

Numéro d'ordre **100ID**

THESE

Présentée par Philippe JAILLON

pour obtenir le grade de DOCTEUR DE L'UNIVERSITE DE SAINT-ETIENNE ET DE L'ECOLE NATIONALE SUPERIEURE DES MINES DE SAINT-ETIENNE.

PROPOSITION D'UNE ARITHMETIQUE RATIONNELLE PARESSEUSE ET D'UN OUTIL D'AIDE A LA SAISIE D'OBJETS EN SYNTHESE D'IMAGES.

Soutenue à Saint-Etienne le 30 septembre 1993

COMPOSITION DU JURY :

Messieurs	J. FRANÇON	Rapporteurs
	J.M. MULLER	
Messieurs	J.M. CHASSERI	Examinateurs
	F. GRAMAIN	
	D. MICHELLUCCI	
	B. PEROCHE	

Remerciements

Je tiens ici à exprimer mes remerciements aux personnes qui ont accepté de juger ce travail ainsi que celles qui m'ont aidé au long de cette thèse.

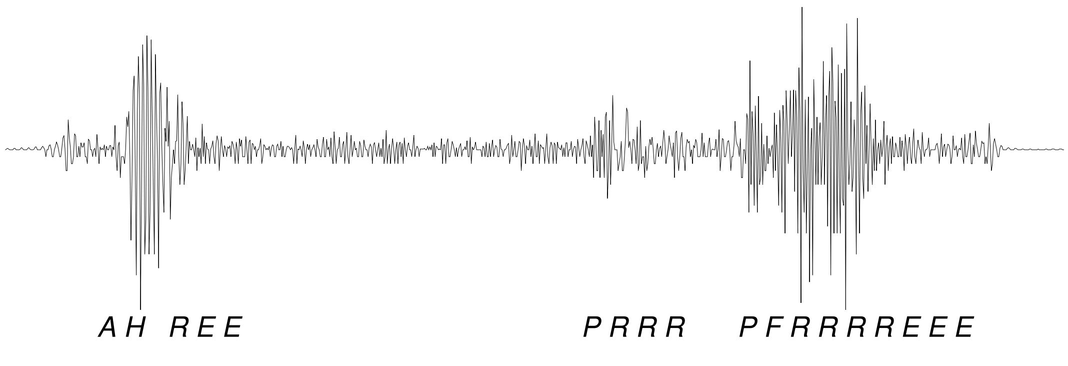
Merci à M. Jean Françon et M. Jean-Michel Muller d'avoir bien voulu être rapporteurs de ce travail, à M. Jean-Marc Chassery et M. François Gramain pour leur présence dans ce jury, à M. Dominique Michellucci pour le temps qu'il a consacré à me supporter (maudit fumeur que je suis) et à M. Bernard Péroche pour m'avoir accueilli dans son laboratoire et conseillé dans mes travaux.

Je tiens à exprimer mon amitié à l'ensemble du département informatique de l'Ecole des Mines de Saint-Etienne, *Marie Line*, *Antoine*, *Daniel*, *Clément* …, mais aussi *JJ*, *Ehoud*, *Michel*, *Mohand* et *Marc* du même bureau que moi. Après il faut penser à tout ceux qui sont partis : *Pascale*, *Elisabeth* et *Gabriel*… J'oublie sûrement du monde, que l'on me pardonne.

Je n'oublie pas le personnel du service de reprographie qui à fait des miracles et assuré la réalisation de ce rapport.

Une place toute particulière est réservée aux fumeurs[1] (*François, Jean-Michel*, *Jean-Luc*, *Roland*, *BK* et *Grégory*) avec qui j'ai eu l'occasion de mélanger les volutes bleues de nos chères cigarettes.

Je terminerai par un grand merci à mes parents, et surtout à **Evelyne** pour son aide de chaque instant et à ma petite **Agathe** dont voici un des premiers discours (sous-titré) :



1. espèce en voie de disparition et qu'il serait bon de protéger

Table des matières

2 Saisie d'objets en synthèse d'images

Introduction

Cette thèse traite de deux sujets bien distincts : une arithmétique exacte efficace utilisant le concept de la "paresse" et la reconstruction interactive en 3 dimensions de scènes polyèdriques à partir de photographies.

1 Des ordinateurs et des nombres

Les ordinateurs tels que nous les connaissons sont le résultat des travaux théoriques de Turing et de Von Neumann. S'ils permettent a priori de résoudre une vaste gamme de problèmes, tous ne le sont pas dans des temps "raisonnables". Certain d'entre eux réclament des temps polynomiaux, voire exponentiels. Quelques uns admettent des solutions avec une complexité plus faible (comme passer de $O(n^2)$ à $O(n \log n)$ par exemple) en utilisant des raisonnements topologiques dont doivent rendre compte les données manipulées (en général des nombres).

1.1 Les arithmétiques machine

Les ordinateurs manipulent des données binaires : 0-1, oui-non, vrai-faux,... L'arithmétique des ordinateurs s'appuie la plupart du temps sur une représentation des nombres en base 2 (les calculettes utilisent généralement la base 10 et certain gros calculateurs la base 16). Les nombres entiers et flottants des machines ont des tailles (nombre de chiffres binaires) qui ont été fixées lors de la construction des processeurs pour assurer un compromis entre la technologie, la rapidité des calculs et la plage d'utilisation (dynamique des nombres et précision). Ces outils de base, fournis sur toutes les machines, montrent vite leurs limitations : les entiers machines ont une dynamique trop limitée et les nombres flottants n'ont pas la puissance du continu (par exemple $1/3$ n'est pas représentable de manière exacte par un nombre flottant).

1.2 Les erreurs de calculs

Les limitations que nous venons de voir sur les arithmétiques des ordinateurs font que tous les calculs effectués avec des nombres flottants ne donnent comme résultat que des approximations de la valeur exacte. Le résultat que l'on obtient n'est pas exact, mais les arrondis étant réalisés tout au long des calculs, il dépend de l'ordre dans lequel sont effectuées les opérations : des expressions algébriquement équivalentes n'ont pas forcément la même valeur en machine.

Pour résoudre ce problème, la solution la plus évidente consiste à utiliser une arithmétique exacte. La plus simple à mettre en œuvre est une arithmétique rationnelle. Mais de telles arithmétiques doivent être simulées et sont donc très lentes sur des ordinateurs classiques et tout programmeur tentera d'en limiter l'utilisation (bien que certaines fois cela s'avère impossible).

Nous proposons dans la première partie de cette thèse une optimisation très puissante des arithmétiques rationnelles : l'arithmétique rationnelle paresseuse. L'idée principale de cette arithmétique est de retarder tous calculs exacts jusqu'à ce qu'ils deviennent soit inutiles (la plupart des cas), soit inévitables : ils sont alors effectués. Ainsi les calculs exacts qui ne sont pas nécessaires ne sont jamais faits. L'arithmétique paresseuse se présente sous la forme d'une bibliothèque autonome prenant à sa charge les problèmes de précision et qui est indépendante des programmes qui l'utilisent.

Cette arithmétique nous a permis de dégager un concept sous-jacent plus général : la paresse. Ce concept met en œuvre des représentations multiples des données manipulées : des informations grossières sous forme d'un encadrement d'un résultat et des informations symboliques permettant de retrouver la valeur exacte de ce résultat si cela s'avère nécessaire. Les opérations sur les encadrements permettent d'éliminer tous les cas triviaux qui ne nécessitent pas le recours à des informations exactes.

Pour être efficace, l'arithmétique paresseuse doit s'accompagner d'un certain nombre d'optimisations telles que l'utilisation de classes d'équivalence, la détection d'isomorphismes ainsi que de la gestion de clefs de hachage (très souvent utilisées par les algorithmes géométriques) sur des expressions symboliques.

L'ensemble de ces travaux nous a permis d'obtenir une arithmétique rationnelle qui est jusqu'à 200 fois plus rapide que les arithmétiques rationnelles standards et seulement 20 à 30 fois plus lente que les arithmétiques flottantes des machines.

2 La synthèse d'images

Le processus de création d'une image ou d'une animation de synthèse se décompose en deux grandes étapes. La création (la modélisation) de l'objet ou de la scène à synthétiser et la visualisation (le rendu) de cette scène. Si la phase de visualisation et de rendu est aujourd'hui bien maîtrisée, il existe toute une série d'algorithmes bien définis, il n'en est pas encore de même pour la modélisation.

2.1 Le modèle CSG

La description géométrique des objets modélisés peut être conservée de plusieurs façons différentes. La scène peut être décrite par une liste de faces ou de surfaces sans cohérence particulière, par la représentation de sa frontière (BRep) qui permet de différencier l'intérieur et l'extérieur des objets ou par un assemblage de solides. Cette dernière méthode est sans doute la plus riche et la plus concise dont on dispose actuellement.

On définit les opérations booléennes "Union", "Intersection" et "Différence" entre des primitives solides ainsi que les transformations qui, appliquées à un solide fournissent un autre solide comme résultat (translation, affinité, rotation,...). Chaque opération crée un nouvel objet solide. Cet objet peut à son tour être utilisé comme une primitive. Les opérations successives appliquées aux objets sont stockées de manière formelle dans un arbre où tous les nœuds sont des opérations booléennes et toutes les feuilles des solides de base, que l'on appelle aussi des "primitives". Cette structuration des données est un "arbre de construction" plus connu sous sa dénomination anglaise "Constructive Solid Geometry" ou CSG.

Les arbres de construction se prêtent facilement aux opérations de visualisation et de rendu. La richesse de la description CSG permet facilement d'en obtenir la représentation par frontières ou par facettes (le contraire étant bien plus délicat à réaliser), de cette manière il est possible d'utiliser tous les algorithmes de rendu existant pour ce type de données (Zbuffer, peintre…). Des méthodes de rendu telles que l'algorithme d'Atherton [ATHE 83] sont spécialement développées pour la visualisation des arbres de construction et permettent de résoudre les opérations booléennes à ce moment là. Les algorithmes de lancé de rayons, eux aussi, se sont facilement adaptés à la visualisation des arbres CSG.

2.2 La modélisation des objets

Avant de passer à la phase de rendu, il est obligatoire de passer par une phase de modélisation. Malheureusement, la modélisation est le parent pauvre de la synthèse d'images, les outils sont complexes à mettre en œuvre, un état de fait aggravé par la soif d'interactions des utilisateurs. Les objets que l'on cherche à modéliser appartiennent à deux grandes familles, soit ils sont le pur fruit de l'imagination, soit ils sont la copie d'objets réels.

Dans le cas de la modélisation d'objets réels, on ne tire pas parti de l'existence de ces objets : on les reconstruit de toutes pièces. Certains outils, tels que les scanners 3D ou les radars permettent d'utiliser l'objet réel pour fabriquer son pendant synthétique, mais ils sont onéreux et limités dans leur domaine d'application (volumes réduits, acquisition de terrains). L'utilisation de l'image comme support à la modélisation est assez peu pratiquée en synthèse d'images, alors qu'elle est le support principal du traitement d'images et de la vision par ordinateur.

Dans la deuxième partie de cette thèse nous nous intéresserons à un outil de modélisation dont le principal intérêt est d'utiliser l'image des objets comme support à leur modélisation sous forme d'arbre de construction. Cet outil est interactif, l'utilisateur pourra de cette manière ne modéliser que ce dont il a besoin et avec le niveau de détails le plus adapté à ses applications. Les extensions que nous proposons dans le domaine de l'incrustation d'images de synthèse dans des images naturelles (fixes ou animées) permettent de traiter correctement l'ombrage de la scène finale en tenant compte de la nature des éclairages, des ombres portées et des reflets.

1 Arithmétique paresseuse

Les nombres entiers ou flottants ne suffisent pas pour tous les types de calculs. Si les ordinateurs sont capables d'effectuer des opérations très vite sur ces nombres, les résultats peuvent s'avérer faux (ou plus ou moins imprécis) dans certains contextes. Pour pallier ce problème, on est tenté d'utiliser des arithmétiques "exactes". Le problème de telles arithmétiques est la lenteur d'exécution, les opérations élémentaires devant être simulées. Une idée simple, pour réduire le coût des calculs, est d'en faire le plus possible à l'aide de l'arithmétique native de la machine (qui est la plus efficace à notre disposition), tout en se laissant la possibilité d'avoir recours à une arithmétique exacte si cela s'avérait nécessaire.

Si à chaque instant on est capable de déterminer la façon la plus efficace (la moins pénalisante) de faire un calcul juste, on est devenu "paresseux".

Nous verrons au cours de cette première partie quels sont les problèmes que causent les imprécisions de calcul des arithmétiques flottantes standard, les solutions proposées ainsi qu'une solution originale : "les nombres paresseux".

Précision des nombres machine 1

Dans les années 1980, la programmation scientifique nécessitait un mode standard de représentation des nombres réels en machine. Toutes les représentations informatiques envisageables sont par nature de taille bornée, et l'architecture matérielle des ordinateurs impose même que la taille de la représentation des nombres réels soit constante (32 ou 64 bits actuellement). Si bien que l'ensemble Σ des nombres représentables en machine est fini (de l'ordre de 2^{32} ou 2^{64} nombres).

Quel que soit le mode de représentation choisi, la finitude de Σ fait qu'il n'est pas clos pour les quatre opérations arithmétiques élémentaires : addition, soustraction, multiplication et division : par exemple le double du nombre positif le plus grand (ou du nombre négatif le plus petit) de Σ n'est pas dans Σ ; de même la moyenne[1] de deux nombres successifs de Σ n'est pas dans Σ ; on prouve facilement que l'ensemble des nombres $x + y$, ainsi que l'ensemble des nombres xy avec $(x, y) \in \Sigma^2$ ont un nombre d'éléments de l'ordre de $\mathrm{card}(\Sigma)^2$. Si l'on ne veut pas que la moindre opération fasse sortir le résultat de Σ et interrompe le déroulement des programmes, on est donc bien forcé d'accepter que la représentation des nombres réels soit approximative.

Le choix de ce sous-ensemble de nombres pour les ordinateurs répond à des critères particuliers : les calculs sur ces nombres doivent être efficaces, la dynamique des nombres doit être grande, les opérations doivent préserver les ordres de grandeur. Il s'agit de trouver un bon compromis entre coût et qualité. Si un certain nombre de solutions existent pour répondre à ce problème, ce sont les nombres flottants ("floating point numbers") qui furent retenus et normalisés sous le nom IEEE 754 en 1985. Ces nombres offrent une grande dynamique, une implantation matérielle efficace, la portabilité entre divers matériels. Ils sont un compromis acceptable.

1. En supposant que Σ contienne 2, certes. Sinon 2 est résultat de $(a + a) / a$, où a est un élément non nul de Σ, et Σ n'est pas clos non plus ! Les seuls ensembles finis et clos pour les opérations arithmétiques élémentaires sont les corps finis.

Malheureusement, comme aucun des systèmes de représentation des nombres flottants ne permet de représenter de manière exacte tous les nombres réels, les nombres représentés en machine ne sont que des approximations. Le problème est de savoir si les erreurs que l'on va commettre tout au long des calculs vont rester négligeables, ou si elles vont s'accumuler et fournir un résultat irrémédiablement faux.

Nous allons illustrer avec quelques exemples les problèmes posés par les imprécisions des nombres flottants.

- La Société chaotique de banque [MULL 89] : Vous faites un apport de $e-1$ francs (e est la base des logarithmes népériens). La première année, vous êtes perdant, on multiplie votre capital par 1 et on retranche 1 franc pour frais de gestion. La deuxième année, c'est mieux, on multiplie le capital par deux et on retranche 1 franc pour la gestion. La troisième année on multiplie par 3 et on retranche 1 franc, et ainsi de suite. La $n^{\text{ième}}$ année on multiplie le capital par n et on retranche 1 franc. Vous retirez votre argent au bout de 25 ans. Les résultats obtenus dépendent beaucoup de la machine utilisée : sur une calculette on obtient -140 302 545 600 000 F et sur un SUN +1 201 807 247 F alors que le résultat n'est que de seulement 4 centimes (environ).

- Intersection de deux segments : On prend deux segments S1 et S2, dont les coordonnées des extrémités sont des nombres flottants. On calcule les coordonnées du point d'intersection de ces deux segments : P. On tente alors de vérifier que le point P que l'on vient de calculer est bien sur les deux segments étudiés. On s'aperçoit facilement que les cas où l'intersection des deux droites tombe sur la grille des nombres flottants sont rares, voire même exceptionnels (figure 1 p. 8). L'approximation du point d'intersection est bonne, mais la vérification de son appartenance aux segments S1 et S2 donnera trop souvent une réponse erronée [MORE 90].

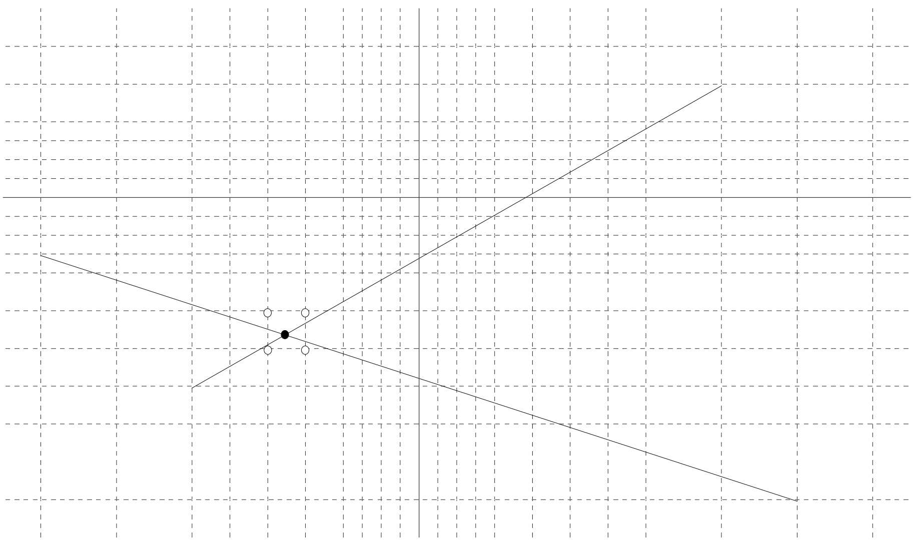


figure 1
Intersection de deux droites sur le plan des nombres flottants

1.1 Arithmétique sur les nombres flottants

Depuis l'apparition des ordinateurs, plusieurs principes de codage des nombres flottants ont été utilisés. Nous ne nous arrêterons pas sur les nombres entiers, qui ne posent qu'un problème : celui de leur taille. Nous verrons les techniques qui permettent de contourner ce problème dans les paragraphes suivants.

1.2 Norme IEEE 754

En 1985, l' "Américan National Standards Institute" et "The Institute of Electrical an Electronics Engineers" définissent la norme des nombres flottants [IEEE 85]. L'adoption de cette norme par une majorité de constructeurs informatiques permettra aux programmes utilisant les nombres flottants d'avoir un comportement identique (un même résultat pour un même calcul) quel que soit le type de machine. La norme IEEE 754 définit quatre formats de nombres flottants, répartis en deux groupes : le format de base (*basic*) et le format étendu (*extended*). Les valeurs représentables dans le format choisi sont spécifiées par les trois paramètres suivants :

- $p =$ nombre de bits significatifs de la mantisse (précision).
- $E_{max} =$ exposant maximum.
- $E_{min} =$ exposant minimum

Les nombres flottants sont de la forme $(-1)^s 2^E (b_0 \cdot b_1 b_2 \ldots b_{p-1})$ où

- $s =$ 0 ou 1
- $E =$ tout entier compris au sens large entre E_{min} et E_{max}.
- $b_i =$ 0 ou 1.

On a de plus les valeurs particulières suivantes :

- deux valeurs infinies $-\infty$ et $+\infty$
- Au moins un signal d'erreur NaN[1] ("floating exception").
- Au moins une valeur NaN (sans génération d'erreur).

La description précédente permet une représentation redondante des nombres, par exemple $2^0 (1 \cdot 0) = 2^1 (0 \cdot 1) = 2^2 (0 \cdot 01) = \ldots$. Les nombres de la forme $\pm 2^E (1 \cdot b_1 b_2 \ldots b_{p-1})$ (représentation canonique) sont appelés nombres normalisés. Les nombres différents de zéro codés sous la forme $\pm 2^{E_{min}} (0 \cdot b_1 b_2 \ldots b_{p-1})$ sont appelés des nombres dénormalisés. Des valeurs réservées de l'exposant permettent de coder $NaN, \pm\infty$, ± 0 et les nombres dénormalisés. Pour tout nombre qui aurait la valeur 0 (zéro), le bit s fournit une information supplémentaire, tous les formats ont une repré-

1. NaN : Not a Number (NaNs au pluriel).

1	8-11		23-52		…taille
s	e		f		
	msb	lsb	msb	lsb	…ordre

				Format
Paramètres	single	single extended	double	double extended
p	24	>32	53	>64
E_{max}	+127	>= +1023	+1023	>= +16383
E_{min}	-126	<= -1022	-1022	<= -16382
Exp. bias	+127	unspecified	+1023	unspecified

figure 2
Format des paramètres des nombres flottants

sentation différente de $+0$ et de -0, le signe étant significatif dans certaines circonstances (la division par ± 0 fournit comme résultat un ∞ correctement signé, le signe de ce nombre peut être testé…).

1.2.1 Formats des nombres

Les nombres en simple et double précision sont composés des trois champs suivants :

Un bit de signe s.

Un exposant biaisé[1] $e = E + biais$

Une fraction ou mantisse $f = 0 \cdot b_1 b_2 \ldots b_{p-1}$

La dynamique de l'exposant E non biaisé doit inclure tous les entiers entre les deux valeurs E_{min} et E_{max}, et doit réserver deux autres valeurs $E_{min} - 1$ pour coder les nombres dénormalisés[2], ainsi que $E_{max} + 1$ pour représenter $\pm\infty$ et NaNs.

La valeur des champs est interprétée de la manière suivante pour les nombres en simple et double précision :

La valeur v d'un nombre X est déduite à partir de ses champs :

si $e = E_{max} + 1 + biais$ et $f \neq 0$ alors v est un NaN, quelle que soit la valeur de s.

si $e = E_{max} + 1 + biais$ et $f = 0$ alors $v = (-1)^s \infty$

si $0 < e < E_{max} + 1 + biais$ et $f \neq 0$ alors $v = (-1)^s 2^{e - biais}(1 \cdot f)$

si $e = 0$ et $f \neq 0$ alors $v = (-1)^s 2^{E_{min} - 1}(0 \cdot f)$ (nombre dénormalisé).

si $e = 0$ et $f = 0$, alors $v = (-1)^s 0$ (zéro plus ou zéro moins).

1. Le Biais est la valeur ajoutée à l'exposant pour qu'il reste toujours positif.
2. Les nombres dénormalisés permettent de représenter des nombres compris entre $-2^{E_{min}-1}$ et $2^{E_{min}-1}$ et de déterminer le signe d'un résultat lorsque celui-ci est zéro après normalisation.

Le principe de codage des nombres est le même, quel que soit le type de flottant, simple ou double précision, “basic” ou “extended”. Seule la taille de la mantisse et de l’exposant change (ainsi que les valeurs particulières associées aux infinis, et au NaNs).

Il est à remarquer que l’écart qui existe entre deux nombres flottants consécutifs de même exposant est constant ([1]) et qu’il double lors de l’incrément de cet exposant ([2]). La valeur de cet écart est :

$$\begin{aligned} \delta_e &= (-1)^s 2^{e-E_{bias}}(1\cdot f) - (-1)^s 2^{e-E_{bias}}(1\cdot f') \\ \delta_e &= (-1)^s 2^{e-E_{bias}}(0\cdot(f-f')) = (-1)^s 2^{e-E_{bias}} 0\cdot 00\ldots01 \end{aligned} \qquad [1]$$

$$\begin{aligned} \delta_{e+1} &= (-1)^s 2^{e+1-E_{bias}}(1\cdot f) - (-1)^s 2^{e+1-E_{bias}}(1\cdot f') \\ \delta_{e+1} &= (-1)^s 2^{e+1-E_{bias}}(0\cdot(f-f')) = 2(-1)^s 2^{e-E_{bias}}(0\cdot(f-f')) = 2\delta_e \end{aligned} \qquad [2]$$

1.2.2 Opérations

Les opérations définies sont l’addition, la soustraction, la multiplication, la division, le reste d’une division et la racine carrée. Le résultat de chacune de ces opérations est le même que celui que l’on obtiendrait par arrondi du résultat exact de cette opération.

1.2.3 Arrondis

Une opération entre deux nombres flottants ne fournit pas forcément un résultat directement représentable dans le format choisi ($1./3. = 0.33333\ldots$). Il est nécessaire de transformer ce résultat pour le rendre représentable, c’est l’opération d’arrondi.

L’arrondi prend un nombre supposé de précision infinie et, si nécessaire, le modifie pour le mettre au format requis. Toutes les opérations produisent un résultat qui est la valeur arrondie, dans le mode choisi, de ce résultat en précision infinie.

La norme IEEE 754 fournit deux types d’arrondi : l’arrondi au plus près et l’arrondi dirigé. Les deux types d’arrondis affectent toutes les opérations arithmétiques, excepté les comparaisons et le reste. Le mode d’arrondi peut modifier le signe d’une somme égale à zéro, et le seuil pour lequel les “overflows” et les “underflows” sont signalés.

L’arrondi au plus près fournit la valeur représentable la plus proche de la valeur en précision infinie comme résultat ; si deux valeurs représentables sont à la même distance, la valeur rendue est celle qui a son bit le moins significatif égal à zéro. Un résultat en précision infinie avec une magnitude $2^{E_{\max}}(2-2^{-p})$ est arrondi vers $\pm\infty$ sans changement de signe ($E_{\max}$ et p sont déterminés par le format choisi).

L’arrondi direct se décompose en trois types : arrondi vers $+\infty$, vers $-\infty$ et vers 0. L’arrondi vers $+\infty$ fournit comme résultat la valeur représentable (éventuellement $+\infty$)

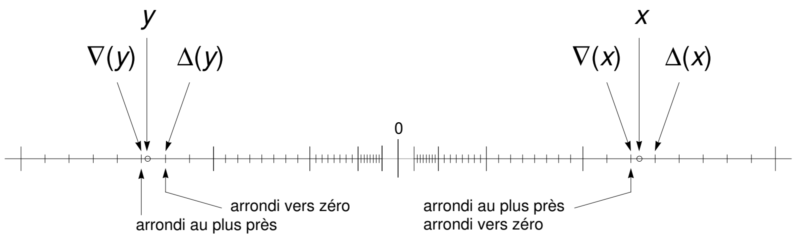


figure 3
Types d'arrondis.

supérieure ou égale la plus proche du résultat en précision infinie, l'arrondi vers $-\infty$, la valeur représentable (éventuellement $-\infty$) inférieure ou égale la plus proche du résultat et l'arrondi vers 0, la valeur représentable la plus proche dont la valeur absolue est inférieure ou égale au résultat.

On définit l'arrondi par défaut et l'arrondi par excès d'un nombre réel x notés respectivement $\nabla(x)$ et $\Delta(x)$ comme étant les nombres représentables dans un format donné immédiatement inférieur ou égal et immédiatement supérieur ou égal à x respectivement. On peut définir les différents modes d'arrondis de x de la manière suivante :

au plus près : $\nabla(x)$ si $x - \nabla(x) \leq \Delta(x) - x$, $\Delta(x)$ sinon.

vers $+\infty$: $\Delta(x)$

vers $-\infty$: $\nabla(x)$

vers 0 : $\Delta(x)$ si $x \leq 0$, $\nabla(x)$ si $x \geq 0$.

Normalement, un nombre est arrondi à la précision de sa représentation. Mais il existe des unités de calculs flottants (FPU) qui ne fournissent que des résultats sous forme “double” ou “extended”. Sur ce type de machine l'utilisateur peut spécifier le fait que l'arrondi soit celui d'un nombre simple précision bien que celui-ci soit stocké en format “double” ou “extended”.

Il est possible de convertir les nombres flottants dans les divers formats définis. Si la conversion entraîne une perte de précision, le résultat est arrondi de la manière spécifiée, en revanche la conversion d'un nombre représentable vers un format plus riche est exacte.

1.2.4 Comparaisons

Elles se font dans tous les formats définis, même si les opérandes sont de formats différents. Les comparaisons sont exactes et n'entraînent jamais d'“overflow” ni d'“underflow”. Il existe quatre relations mutuellement exclusives : inférieur, égal, supérieur et incomparable. Ce dernier cas se rencontre si l'un des opérandes se trouve être un NaN (un NaN ne peut pas être comparé à autre chose sans provoquer d'erreur).

Opération sur les infinis : $-\infty < \{\text{ensemble des nombres finis représentables}\} < +\infty$. L'arithmétique sur les infinis est exacte et ne génère pas d'erreur, sauf dans le cas d'opérations illicites. Un signal d'exception est généré seulement dans les cas suivants : une valeur ∞ est créée par débordement à partir d'opérandes finis ou par division par zéro ou lorsque ∞ est un opérande invalide.

1.2.5 Exceptions et débordements

Il y a cinq types d'exceptions qui sont signalées lorsqu'elles sont détectées : opération illicite, division par zéro, "overflow", "underflow" et inexacte.

Les opérations illicites sont toutes les opérations dont le résultat est sans signification ou impossible à calculer. Elles apparaissent dans les cas suivants :

Toutes les opérations ayant un NaN comme opérande.

Addition ou soustraction $(+\infty) + (-\infty)$.

Multiplication $0 \times \infty$.

Division $0/0$ ou ∞/∞.

Reste $x\%y$ avec $y = 0$ ou $y = \infty$.

Racine carrée d'un nombre négatif.

Si le diviseur est zéro et le dividende un nombre fini différent de zéro, la division par zéro est signalée, et le résultat est un ∞ correctement signé.

Une exception du type "overflow" est signalée quand la valeur absolue du résultat d'une opération excède la magnitude du plus grand nombre représentable. Le résultat de cette opération est déterminé par le mode d'arrondi choisi :

Au plus près : $\pm\infty$ en fonction du signe du résultat intermédiaire.

vers $+\infty$: si le résultat est négatif, le plus petit des nombres négatifs représentables, sinon $+\infty$.

vers $-\infty$: si le résultat est positif, le plus grand nombre positif représentable, sinon $-\infty$

vers 0 : le plus grand nombre représentable, positif ou négatif en fonction du signe du résultat intermédiaire.

Deux événements provoquent un "underflow" : la création d'un résultat différent de zéro compris entre $-2^{E_{\min}}$ et $+2^{E_{\min}}$, ou une perte de précision liée à l'approximation de tels nombres par dénormalisation.

Problèmes de géométrie 2

Comme nous l'avons vu à titre d'exemple au § 1, il est très rare que l'intersection de deux segments se trouve sur la grille des nombres flottants. Ce ne serait pas très important si les algorithmes géométriques pouvaient s'affranchir des erreurs de calcul. Le cas de l'algorithme de Bentley-Ottmann [BENT 79] est exemplaire à ce sujet. Cet algorithme se propose de trouver les k points d'intersection de n segments du plan avec une complexité en $O((n+k)\log n)$ (les algorithmes naïfs résolvent ce problème en $O(n^2)$). Pour obtenir une telle complexité, cet algorithme repose sur une forte cohérence topologique dont voici le principe général :

- Mettre les événements (extrémités des segments ou points d'intersections) en place dans une file de priorité triée par ordre lexicographique. Le xy-ordre est défini de la manière suivante : $(x, y) < (x', y') \Leftrightarrow x < x' \text{ ou } ((x = x') \text{ et } (y < y'))$.
- On balaie le plan par une droite verticale (parallèle à l'axe des Y) de $x = -\infty$ à $x = +\infty$. Une structure de données appelée barre de balayage rend compte de l'ordre vertical des points d'intersections des segments avec la position courante de la barre de balayage.
- On extrait un à un les événements de la file de priorité.
- Si la file de priorité délivre l'extrémité initiale d'un segment, on introduit en bonne place ce segment dans la barre de balayage. On regarde ensuite quels sont les segments directement adjacents (en dessous et au-dessus). On teste l'intersection du nouveau segment avec ses voisins, s'il y en a. S'il y a une intersection, celle-ci doit se trouver à droite de la barre de balayage, on insère ce point d'intersection dans la file de priorité des événements.
- Si la file de priorité délivre l'extrémité finale d'un segment, on supprime alors ce segment de la barre de balayage. Il se peut que l'on ait rendu adjacents des segments qui ne l'étaient pas précédemment ; on teste alors leur intersection de la même manière que lors de l'étape précédente.

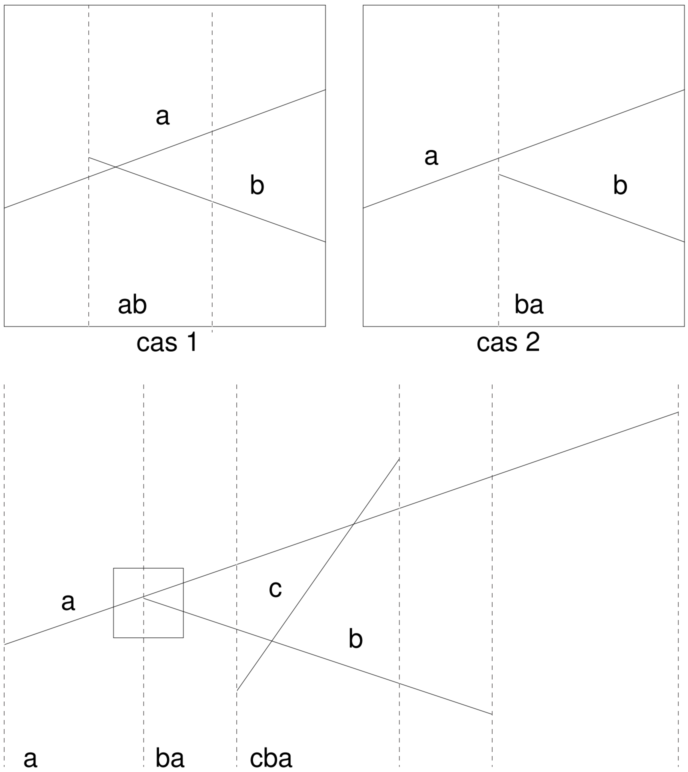


figure 4
Incohérences topologiques

L'intersection de deux segments nécessite d'inverser l'ordre d'apparition des demi-segments qui se trouvent à droite et à gauche du point d'intersection.

Le problème principal d'un tel algorithme est de dépendre de l'invariant suivant : après le traitement de l'événement courant, le graphe se trouvant à gauche de la barre de balayage est planaire (toutes les intersections à gauche de la barre de balayage ont dû être trouvées, et l'ordre dans cette même barre de balayage doit être correct). Cela implique qu'à tout moment, la topologie des segments est conforme à l'interprétation numérique qu'en fait l'algorithme. Ceci n'est plus vrai si la précision des calculs ne permet pas de déterminer de manière sûre la position de l'intersection de deux segments.

2.1 Prise de décision sûre

Presque tout algorithme doit prendre des décisions lors de son déroulement. Le cas le plus simple et le plus commun est sans doute celui de “l'aiguillage” :

si (a>b) alors … sinon …

Dans ce genre de circonstances, il est plus important de pouvoir déterminer la relation d'ordre qui lie les deux quantités *a* et *b* que de savoir exactement de combien l'une est supérieure à l'autre. En géométrie, la plupart des décisions que l'on est amené à prendre sont de l'ordre de la comparaison pure (les segments se coupent-t-ils ? ce point est-il à l'intérieur de ce cercle ? …).

Dans l'exemple précédent, illustré par la figure 4 page 16, on doit prendre une décision (B coupe-t-il A ?) lors de l'insertion du segment B dans la droite de balayage. A partir de cet instant précis, il est impératif de ne faire aucune erreur lors du calcul de l'intersection entre A et B. Si le calcul numérique de l'intersection entre A et B fournit un résultat topologiquement inexact (pas d'intersection dans le cas 1, intersection dans le cas 2), l'algorithme de Bentley-Ottmann sera pris en défaut lors du traitement du segment C. Dans les deux cas on supposera que A et C sont adjacents, le calcul d'intersection entre A et C fournira un nouvel événement E, et lorsque l'on traitera cet événement, on intersectera B et C, malheureusement ce point d'intersection se trouve en arrière de la ligne de balayage et ne pourra jamais être traité correctement. La propagation de cette erreur entraînera une solution erronée au problème : le graphe que l'on obtient ne sera pas planaire; il se peut même que le programme n'arrive pas à son terme à la suite de cette erreur.

Comme nous venons de le voir, un algorithme peut être pris en défaut par une décision erronée consécutive à une imprécision de calcul. Cette imprécision, même insignifiante numériquement, a modifié la topologie locale et entraîné une erreur dont les conséquences seront fatales pour l'algorithme. Le problème que nous venons de rencontrer n'est pas l'inexactitude du résultat de l'intersection en lui même, mais les contradictions provoquées entre les diverses décisions de l'algorithme. On trouve une analyse détaillée de ces problèmes dans [MICH 87] et [MORE 90] dont le souci était d'implanter de manière fiable des algorithmes géométriques de ce type.

2.2 Les solutions connues

Pour tenter de résoudre correctement les problèmes que nous venons d'énoncer, nous trouvons dans la littérature deux grandes familles apparentes : modifier la géométrie du problème ou modifier l'arithmétique. Dans tous les cas, les auteurs s'attachent à conserver cohérente la topologie de leurs données. Nous présenterons dans ce paragraphe les trois approches qui nous ont semblé les plus représentatives : *utilisation d'un epsilon*, *recalage* et *calcul exact*. Une infinité de variantes et de combinaisons de ces trois principes est possible. On peut aussi remarquer que très souvent, les solutions à caractère géométrique peuvent se traduire sous la forme d'une solution de type arithmétique et réciproquement, ce qui rend leur classification complexe.

2.2.1 Epsilon

On considère qu'il existe un seuil numérique (epsilon) en dessous duquel aucune décision fiable ne peut plus être prise par la machine et cette décision doit être remplacée par un choix empirique qui dépend du problème à traiter et du contexte. La solution la plus simple est de considérer qu'en dessous d'un certain seuil, les quantités comparées sont identiques. Mais si ce type de solution est possible dans le cas où le comportement de l'algorithme ne dépend pas des résultats (en général dans les algorithmes naïfs), il est très hasardeux de s'en servir dès que l'algorithme exploite une quelconque cohérence géométrique, comme c'est le cas de l'algorithme de Bentley-Ottmann : on se met à regrouper dans la même classe des quantités d'influences très différentes.

M. Segal et H. Sequin proposent dans [SEGA 85] de construire autour de chaque objet étudié une sorte de zone d'influence qui privilégie cet objet. Par exemple, un segment [AB] se voit dilater d'une valeur ε et l'extrémité C du segment [CD] qui se trouve à

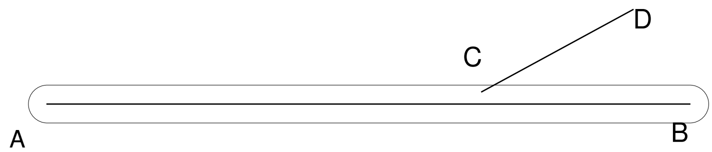


figure 5
Zone d'influence du segment AB

l'intérieur de la zone d'influence du segment [AB] est considéré comme appartenant à ce segment (figure 5 p. 18). Cette approche permet de créer ce que les auteurs appellent une consolidation de la scène étudiée. Son avantage est de permettre des opérations mathématiques constructives tout en assurant une cohérence globale de la topologie. On peut remarquer que cette méthode s'apparente à une extension au plan des arithmétiques d'intervalles.

2.2.2 Recalage

Les données initiales sont supposées appartenir ou avoir été recalées sur une grille entière de taille suffisante (G). L'équation de droite en résultant sera de la forme $ax + by + c = 0$ avec $a = y_1 - y_0$, $b = x_0 - x_1$ et $c = y_0 x_1 - x_0 y_1$. Dans ce cas tous les résultats des calculs impliqués lors de la détection d'une intersection entre deux segments sont des rationnels de taille bornée. Comme les points définissant les segments sont sur la grille, on est assuré que $|a| \le G$, $|b| \le G$ et que $|c| \le G^2$ et que les nombres peuvent être fidèlement représentés par des entiers de la machine. Le point d'intersection M de deux droites “recalées” est de coordonnées $(\Delta_x/\Delta, \Delta_y/\Delta)$ si $\Delta \ne 0$, $|\Delta_x|$ et $|\Delta_y| \le 4G^3$ et $|\Delta| \le 2G^2$. En machine, ce point peut être représenté en coordonnées homogènes entières $(\Delta_x, \Delta_y, \Delta)$, ou sous sa forme euclidienne : $x_e, y_e, x_r, y_r, \Delta$ avec

$\Delta_x = x_e\Delta + x_r$ et $\Delta_y = y_e\Delta + y_r$ $(0 \le x_e, y_e \le G$ et $0 \le x_r, y_r \le \Delta \le 2G^2)$ de manière à permettre une plus grande taille pour G et donc d'améliorer la précision de grille.

Pour que l'arithmétique entière native de la machine suffise, il suffit de choisir G de manière à ce que $2G^2$ soit inférieur ou égal au plus grand nombre entier représentable. Si ces entiers sont codés sur n bits, ce nombre est $N = 2^n - 1$ et l'on doit choisir G de la sorte que :

$$G \le \sqrt{\frac{2^n - 1}{2}} \qquad n = 32 \Leftrightarrow G \le 32768$$

Ceci n'est valable que pour l'intersection des segments initiaux : la valeur de G doit être réduite si l'on veut intersecter des segments dont les extrémités sont le résultat des calculs précédents, ce qui limitera fortement la taille de la grille initiale (si $(x, y) \le 2G^2$ alors $\Delta \le 2(2G^2)^2$). Les précisions quant à l'implantation et au traitement du recalage et des cas particuliers sont décrits dans [MICH 87].

2.2.3 Exact

La dernière solution que nous allons exposer dans ce paragraphe est la plus simple, la plus sûre, mais aussi la plus lente. Pour obtenir un résultat exact, prendre des décisions fiables, il suffit juste d'avoir une arithmétique exacte et fiable … Dans le cas de problèmes géométriques linéaires, une arithmétique rationnelle construite avec des nombres entiers de taille quelconque résoudra le problème. Le monde n'étant vraiment pas parfait, cette exactitude des résultats se paye par les délais nécessaires pour les obtenir. Karasick évoque ses déboires avec les arithmétiques exactes dans [KARA 89] où les différences de temps de calculs entre deux implantations d'une triangulation de Delaunay, l'une en flottant et l'autre en rationnel atteignaient un facteur 10.000 (de 0,1 à 1200 secondes pour un même échantillon de dix points).

Arithmétiques exactes 3

Comme nous venons de le voir, les arithmétiques machines sont incapables de représenter tous les nombres (par exemple le nombre 1/3 = 0,333333…), la façon de représenter les nombres et leurs tailles étant fixées lors de la construction du processeur ou de l'unité de calcul flottant. Une méthode simple pour obtenir des calculs exacts (dans certaines limites, pas de fonction transcendante, pas de racine carrée) est d'utiliser une arithmétique rationnelle dont le numérateur et le dénominateur sont des entiers de taille non bornée.

Ce type d'arithmétique a l'avantage de l'exactitude et de la fiabilité, mais le fait d'être simulée la rend inefficace à la fois en termes de temps nécessaire pour effectuer les opérations, mais aussi de place mémoire pour stocker les nombres qui peuvent facilement devenir très encombrants.

3.1 Les nombres entiers non bornés

Les nombres entiers de taille arbitraire sont représentés par des polynômes d'une base b judicieusement choisie en fonction des caractéristiques de la machine utilisée (en général une puissance de 2) sous la forme :

$$x = s_x(n_n b^n + n_{n-1} b^{n-1} + \ldots + n_2 b^2 + n_1 b^1 + n_0 b^0) \qquad s_x = \pm 1 \qquad [3]$$

Les n_i sont les chiffres du nombre en base b (coefficients du polynôme), ils sont tous positifs ou nuls et strictement inférieurs à b. Le signe de l'entier s_x est conservé à part. Sur l'ensemble des entiers non bornés on définit les opérations d'addition, soustraction, multiplication et division euclidienne ainsi que les opérateurs de comparaison > et = (les autres comparaisons s'en déduisent facilement). Toutes ces opérations ne sont pas très complexes si on les programme de manière naturelle (seule la division réclame un peu d'attention) et sont décrites de manière complète dans [KNUT 81].

3.2 Les nombres rationnels

Les nombres de l'ensemble des rationnels $\mathbb{Q}$ se représentent par une paire d'entiers non bornés : le numérateur et le dénominateur. Ces nombres rationnels vont pouvoir se représenter sous la forme :

$$x = \frac{\text{num}}{\text{den}} = \frac{s_x(n_n b^n + n_{n-1} b^{n-1} + \ldots + n_2 b^2 + n_1 b^1 + n_0 b^0)}{d_p b^p + d_{p-1} b^{p-1} + \ldots + d_2 b^2 + d_1 b^1 + d_0 b^0} \quad [\,4\,]$$

Le signe d'un rationnel est soit conservé à part avec un numérateur et un dénominateur positif, soit de manière conventionnelle, noté sur le numérateur avec un dénominateur positif.

Sur $\mathbb{Q}$ sont définies les opérations suivantes : l'addition, la multiplication, l'opposé et l'inverse notées respectivement +, *, opp, inv et les opérateurs d'égalité (=) et de comparaison (>). On définit ces opérations de la manière suivante :

$$\text{soit } x = \frac{n_x}{d_x} \text{ et } y = \frac{n_y}{d_y} \quad (d_x, d_y > 0) \qquad \begin{cases} x + y = \dfrac{n_x d_y + n_y d_x}{d_x d_y} \\ x * y = \dfrac{n_x n_y}{d_x d_y} \\ \text{opp}(x) = -\dfrac{n_x}{d_x} \\ \text{inv}(x) = \dfrac{d_x}{n_x} \text{ si } n_x \neq 0 \\ x = y \Leftrightarrow n_x d_y = d_x n_y \\ x > y \Leftrightarrow n_x d_y > d_x n_y \end{cases} \quad [\,5\,]$$

Les nombres rationnels sont supposés irréductibles (le numérateur et le dénomiteur sont premiers entre eux). Pour conserver cette propriété après chaque opération (addition ou multiplication) la fraction résultat doit être réduite (division du numérateur et du dénominateur par leur pgcd).

3.3 Coût des calculs

Le coût des calculs sur des fractions rationnelles est lié à la taille des polynômes représentant les numérateurs et les dénominateurs de chaque fraction, et au coût d'une opération entre deux polynômes (elle aussi liée à la taille des nombres). Seule l'addition et la multiplication de deux fractions ont des coûts non négligeables, l'opposé et l'inverse ont un coût constant : changer le signe dans le cas de l'opposé (si celui-ci est conservé à part), permuter numérateurs et dénominateurs pour l'inverse.

3.3.1 Addition et multiplication de deux entiers de taille arbitraire

Soient deux polynomes N et M de degrés respectifs n et m avec $n \leq m$.

L'addition des deux polynômes réclame l'addition de n termes et la propagation de $m+1$ retenues pour fournir un résultat de degré $m+1$ dans le pire des cas. La complexité d'un tel algorithme est O(m). Il faut remarquer qu'il existe des algorithmes et des représentations plus complexe des nombres [MULL 91], comme par exemple la représentation redondante d'Avizienis, qui permettent d'ajouter deux nombres de p chiffres en $O(1)$ si l'on dispose d'une architecture dédiée (avec $O(p)$ processeurs).

La multiplication, quant à elle, est plus gourmande. Un algorithme naïf demande de multiplier les n coefficients du polynôme N par le polynôme M et de faire la somme de tous les polynômes intermédiaires pour obtenir un résultat de degré $m+n+1$ dans le pire des cas. Le produit d'un polynôme de degré m par un scalaire se fait en O(m) opérations et la somme des n polynômes de degré m en nO(m) soit globalement en O(mn). Il existe d'autre techniques de multiplication des polynômes dont la complexité est asymptotiquement meilleure que la méthode naïve en utilisant les techniques diviser pour régner ("divide and conquer"), la complexité passe de $'\mathrm{O}(n^2)$ à $'\mathrm{O}(n(\log n)\log\log n)$ [AHO 74]. Mais ces méthodes ne deviennent vraiment intéressantes que pour de très grands polynômes (degré > 500).

3.3.2 Addition et multiplication de deux fractions rationnelles

Soient deux fractions rationnelles $x = \dfrac{n_n b^n + \dots}{d_p b^p + \dots}$ et $y = \dfrac{n_m b^m + \dots}{d_q b^q + \dots}$.

L'addition de deux fractions telle que nous la définissons en [5] se fait par la somme des produits croisés au numérateur et par le produit des dénominateurs qui donnent comme résultat un polynôme de degré au plus égal à $\max((n+q+1), (m+p+1))+1$ au numérateur (soit $\max((n+q), (m+p))+2$) et $p+q+1$ au dénominateur.

La multiplication de ces deux mêmes fractions donne comme résultat une fraction dont le numérateur est de degré $n+m+1$ au numérateur et $p+q+1$ au dénominateur.

L'enchaînement des opérations fait croître rapidement la taille des nombres du résultat si l'on n'y prend pas garde. A titre d'exemple, l'évaluation d'un déterminant 2x2 (2 multiplications et 1 soustraction en utilisant l'algorithme naïf) dont toutes les fractions rationnelles sont constituées de polynômes de degré n fournit un résultat de degré $4n+4$ au numérateur et $4n+3$ au dénominateur. Si l'on passe à un déterminant de taille 3x3 (9 multiplications et 5 additions ou soustractions) qui est très utile en géométrie, le degré du numérateur devient $15n+17$ et celui du dénominateur $15n+14$, dans le pire des cas.

Remarque : le degré de la somme de k fractions rationnelles ($k-1$ opérations) dont les polynômes sont de degré n est de la forme $k(n+2)-2$ pour le numérateur et $k(n+1)-1$ pour le dénominateur. L'algorithme naïf de calcul d'un déterminant nxn (règle de Sarrus) est en $'O(n(n!))$ opérations, celui de Bareiss en $'O(n^3)$. Le calcul d'un déterminant 10x10 par la première méthode donnerait un numérateur de degré approximatif 108 863 995 (10! = 3 628 800) et la seconde méthode un numérateur de degré 2 995 pour des polynômes initiaux de degré 1… Bien sur, ceci n'est vrai que si l'on ne réduit jamais les fractions rationnelles : si les calculs de déterminants sont justes, quelle que soit la méthode utilisée, ils auront même valeur, donc même degré après réduction.

Ces considérations posent le problème de la réduction des fractions rationnelles de manière à ce que des fractions du type $n/d = n'p/d'p$ (n' et d' sont premiers entre eux) soient réduites à n'/d' si $p \neq 1$. Cette opération nécessite de découvrir le plus grand facteur commun p de n et de d et de diviser n et d par p. La découverte de p se fait de manière habituelle par un algorithme vieux de 20 siècles : l'algorithme d'Euclide. Sa complexité est bornée en nombre d'itérations par la relation :

$$\text{si } 0 \le v \le u \text{, le nombre d'étapes } n \text{ pour obtenir } \mathrm{pgcd}(u, v) \text{ est } n \le \frac{\log(\sqrt{5}u)}{\log\left(\frac{1+\sqrt{5}}{2}\right)} - 2$$

où à chaque étape est effectué un calcul de modulo ou une division selon les versions de l'algorithme [MIGN 89]. La complexité de la division en nombre d'opérations est $'O(n^3)$ pour un algorithme simple et $'O(n\log^2 n \log\log n)$ selon [AHO 74]. D'un strict point de vue théorique la réduction des fractions rationnelles ne s'impose pas. Le théorème de Lejeune-Dirichlet [KNUT 81] dit que dans 60% des cas, deux nombres entiers pris au hasard sont premiers entre eux et que dans 94% des cas ils ont un pgcd inférieur à 10. La simplification qu'entraînerait la réduction serait donc dérisoire. Il semblerait qu'en moyenne il ne soit pas nécessaire de rendre irréductible les fractions rationnelles [MENI 93]. Mais l'expérience, par contre, nous prouve souvent le contraire : les calculs et les nombres que nous utilisons ne sont pas le fruit du hasard.

Si l'on décide de toujours rendre irréductibles les fractions résultats d'une addition ou d'une multiplication entre deux rationnels, on a intérêt à faire quelques optimisations pour tenter de réduire le coût du pgcd [DAVE 87].

- Cas de la multiplication : $\frac{a}{b} \times \frac{c}{d} = \frac{p}{q}$

  La façon la plus évidente est de calculer $p = ac$, $q = bd$, et de diviser, p et q par $\mathrm{pgcd}(p, q)$. Mais, si l'on suppose que a/b et c/d sont déjà réduits, on peut conclure que :

$$\mathrm{pgcd}(p, q) = \mathrm{pgcd}(a, d)\mathrm{pgcd}(b, c)$$

Il est alors plus efficace de calculer les deux pgcd de droite que le pgcd de gauche, car les données sont plus petites (si a, b, c et d sont des polynômes de degré n,p et q sont des polynômes de degré $2n+1$. Le calcul de $\mathrm{pgcd}(p, q)$ se fait en $(2n+1)^2$ opérations, alors que $\mathrm{pgcd}(a, d)$ et $\mathrm{pgcd}(b, c)$ se fait en $2n^2$ opérations, soit au moins 2 fois plus vite.

- Cas de l'addition : $\frac{a}{b}+\frac{c}{d} = \frac{p}{q}$

  On peut calculer $p = ad+bc$ et $q = bd$, mais il est plus efficace de calculer

$$q = \frac{bd}{\mathrm{pgcd}(b, d)} \quad \text{et} \quad p = a\frac{d}{\mathrm{pgcd}(b, d)} + c\frac{b}{\mathrm{pgcd}(b, d)}$$

  Ces deux relations fourniront des valeurs de p et q plus petites que celles obtenues de manière naïve. Le calcul de $\mathrm{pgcd}(p, q)$ permettra la réduction complète de la fraction.

La comparaison de deux fractions rationnelles se fait de manière conventionnelle en les ramenant au même dénominateur :

$$\frac{a}{b} = \frac{c}{d} \Leftrightarrow ad = bc \qquad \frac{a}{b} < \frac{c}{d} \Leftrightarrow ad < bc$$

Dans le cas où les fractions rationnelles sont irréductibles on peut avantageusement remplacer le test d'égalité des produits croisés par la comparaison des numérateurs et dénominateurs :

$$\frac{a}{b} = \frac{c}{d} \Leftrightarrow a = c \text{ et } b = d$$

Le lecteur trouvera tout ce qui est nécessaire à l'implantation d'une arithmétique sur de grands entiers et sur les arithmétiques rationnelles dans [AHO 74], [SEDG 89], [MORE 90] et surtout dans [KNUT 81]. Il est à remarquer que de telles arithmétiques existent dans le commerce et dans le domaine public, tel que [BIGN 89] ou [HANS 90].

3.4 Réticence à faire des calculs exacts

Les résultats que l'on vient de mettre en évidence au § 3.3 laissent à penser qu'il n'est pas complètement ridicule de se demander si le calcul que l'on va effectuer est bien nécessaire, et si la méthode choisie est la mieux adaptée. Les cas où il est opportun d'avoir recours à un calcul exact dépendent surtout du contexte dans lequel le résultat sera utilisé. Si l'on compare le résultat d'une fonction f avec 0, si f n'est pas d'une instabilité reconnue, on a peu de chance de se tromper tant que la valeur recueillie se trouve au dessus d'un certain seuil. Mais dans le cas contraire les arrondis successifs de la machine ont pu perturber le résultat et la valeur rendue par la fonction *f* n'est plus digne de foi.

La réticence à faire des calculs exacts est une façon naturelle d'étendre les solutions qui ont été exposées au § 2.2 . Dans le cas des solutions utilisant un epsilon, on peut admettre facilement que si les calculs fournissent un résultat qui ne semble pas fiable ($\leq \varepsilon$), on décide d'utiliser un calcul exact pour résoudre le problème. Il est clair que ce type de solution est en fait une approche intermédiaire entre le calcul exact et le calcul approché. La seule ombre qui reste au tableau est celle du calcul exact. Avec de telles solutions, il est impératif de connaître explicitement (avant la compilation) le calcul à effectuer pour pouvoir le faire à la fois de manière approché (en utilisant l'arithmétique native et efficace de la machine) et de manière exacte lorsque cela s'avère nécessaire. Lors de l'implantation d'un algorithme, le programmeur doit prendre à sa charge ce type de considérations.

3.4.1 Prise en charge par le programmeur

Deux auteurs nous on paru représentatifs de ce type de technique. Le premier [KARA 89], propose une utilisation ponctuelle et adaptative d'une arithmétique exacte pour déterminer le signe d'un déterminant. Le second, [MORE 90], décrit dans sa thèse une généralisation des arithmétiques à epsilon près avec recours réticent aux calculs exacts pour résoudre des problèmes de triangulations contraintes avec des données initiales de magnitude très différentes. Dans les deux cas, il faut remarquer que les solutions apportées ne sont pas indépendantes des problèmes à résoudre.

Karasick, Lieber et Nackman se sont trouvés, eux aussi, confrontés au fameux problème des “imprécisions numériques entraînant des erreurs topologiques” lors de l'implantation d'un algorithme de triangulation dû à Guibas et Stolfi [GUIB 85]. Cet algorithme a la particularité de n'utiliser que seulement deux tests géométriques simples, mais dont les résultats sont prépondérants pour son bon déroulement. Le prédicat *InCircle*,

$$\begin{vmatrix} 1 & x_a & y_a & x_a^2 + y_a^2 \\ 1 & x_b & y_b & x_b^2 + y_b^2 \\ 1 & x_c & y_c & x_c^2 + y_c^2 \\ 1 & x_d & y_d & x_d^2 + y_d^2 \end{vmatrix} < 0$$

qui est vrai si pour quatre points a, b, c et d, un de ces points est inclus dans le cercle circonscrit aux trois autres.

Le prédicat *CCW*,

$$\begin{vmatrix} 1 & x_a & y_a \\ 1 & x_b & y_b \\ 1 & x_c & y_c \end{vmatrix} > 0$$

qui est vrai si a, b et c sont orientés dans le sens trigonométrique direct. L'algorithme de Guibas et Stolfi effectue $'O(n\log n)$ *InCircle* et *CCW* tests pour trianguler n points.

Karasick, Lieber et Nackman se proposent de calculer de manière fiable ces deux déterminants. Leur première approche fut de les calculer de manière exacte à l'aide d'une arithmétique rationnelle. Les résultats en termes de temps furent tellement déplorables (10.000 fois plus lent qu'en arithmétiaue à virgule flottante) qu'ils décidèrent de rechercher une optimisation du calcul du signe de ces déterminants.

La première étape a consisté à transformer les matrices de nombres rationnels en matrices d'intervalles à bornes entières encadrant de manière grossière les valeurs initiales en utilisant les chiffres les plus significatifs des nombres ($1 < 123 \times 10^{-2} < 2$, $12 < 123 \times 10^{-1} < 13$ …). Si le résultat du calcul de déterminant sur ces matrices est un intervalle qui ne contient pas zéro, le signe est alors facile à détecter. Dans le cas contraire la précision des bornes des intervalles est améliorée et le calcul est répété. De cette manière on est sûr de découvrir le signe exact du déterminant, et de n'avoir recours à une grande précision sur les bornes des intervalles que lorsque l'on se trouve dans des cas litigieux. Ce principe et quelques optimisations sur le calcul des déterminants ont permis aux auteurs d'obtenir des temps d'exécution raisonnables, soit environ trente fois plus lents que leur meilleure implantation en nombres flottants.

Leur approche est une réticence à faire des calculs exacts à l'endroit le plus critique de leur algorithme (les décisions importantes se prennent au vu du signe de ces déterminants). Les données à traiter sont toujours des valeurs initiales du programme (génération zéro), et leur précision n'a que peu d'importance dans ce cas. Ils ne font appel au calcul exact que si cela s'avère nécessaire et seulement dans des endroits bien définis de leur programme.

Avec Moreau, on entre de plain-pied dans le monde de la réticence et de la mixité. Les idées et les travaux que nous allons exposer sont décrits avec beaucoup de détails dans [MORE 90]. L'idée générale en est : l'inexactitude des calculs effectués n'aura d'influence que lorsqu'il sera nécessaire de prendre une décision. Une décision ne pourra être prise à la suite de calculs flottants auxquels on ne peut accorder suffisamment de crédit , on devra alors avoir recours à la décision exacte. De manière claire, cela veut dire que pour toute suite de calculs on est capable de déterminer un seuil positif en dessous duquel il est impossible de prendre une décision sûre sans avoir recours à un calcul exact.

Le premier problème que l'on rencontre avec ce type d'approche est de déterminer la valeur du fameux *epsilon*, seuil de fiabilité. Deux choix s'offrent à nous : le premier

```
Comparaison(a,b)
    si(a ≥ b + ε)
      a est vraiment supérieur à b
    sinon
      si (a ≤ b − ε)
        a est vraiment inférieur à b
      sinon
        ComparaisonExacte(a, b)
```

figure 6
Algorithme mixte et réticent typique.

est de le déterminer pour chaque type de calcul aboutissant à une comparaison. Ce principe demande au programmeur de manipuler une grande quantité de seuils distincts et surtout de ne pas les mélanger. La deuxième solution est de déterminer un epsilon global à l'algorithme. Là aussi, il existe un problème, cet epsilon devra être supérieur ou égal à tous les epsilons maximum locaux, et imposera une utilisation exagérée de la représentation exacte. On doit remarquer que la valeur d'epsilon n'est pas indépendante de la dynamique des données à traiter. Comme nous l'avons vu au § 1.2 , la précision relative des nombres flottants varie avec leur magnitude. Moreau donne à titre d'exemple dans le cadre d'une application particulière (triangulation de terrains avec contraintes) une valeur d'epsilon égale à 10^{-1} pour manipuler de manière fiable des données dont la magnitude n'excède pas 10^{12}, ce qui assure que tous les résultats compris entre 10^{-1} et 10^{12} permettront de prendre une décision identique à celle qui aurait été prise à l'aide d'une arithmétique exacte. Si le résultat est inférieur à 10^{-1}, il faut avoir recours à un calcul exact, et si il est supérieur à 10^{12}, on s'est trompé lorsque l'on a déterminé la valeur de epsilon.

Pour que les comparaisons puissent être menées, à la fois en nombres flottants et en nombres rationnels, on doit avoir simultanément les deux représentations (flottants/rationnels) quand c'est nécessaire et sinon être capable de calculer la représentation rationnelle des valeurs manipulées, pour minimiser un recours trop fréquent à la librairie exacte. En fait, les données initiales (génération zéro) sont définies comme étant de précision maximale ; il n'est donc jamais nécessaire d'avoir recours à leur représentation rationnelle. Par contre, s'ils sont utilisés pour des calculs ultérieurs tous les résultats intermédiaires (génération 1, voire plus : les coordonnées d'un point d'intersection par exemple) doivent être conservés de manière mixte (flottants et rationnels) car on ne garde pas l'historique de leur création.

Dans les cas où l'on doit avoir recours à l'arithmétique exacte, le fait d'avoir d'abord fait les calculs en flottant n'a pas d'incidence notoire sur les performances des algorithmes, la probabilité de rencontrer de tels cas est assez faible (1% environ pour l'intersection de deux segments pris au hasard).

Dans les deux solutions que l'on vient d'exposer, le programmeur doit prendre en charge lui même les problèmes de précision. Ces deux exemples permettent quand même de dégager des principes généraux à mettre en œuvre pour obtenir une implantation à la fois fiable et efficace de certaines classes d'algorithmes. L'utilisation d'epsilon ou d'une arithmétique d'intervalles associée à une arithmétique lente mais exacte semble être un embryon de solution.

3.4.2 Arithmétique ad hoc

Une autre approche, toujours pour pallier la déficience des arithmétiques flottantes, consiste à définir l'ensemble des opérations élémentaires que l'on veut pouvoir utiliser sans risque d'erreur pour déterminer les caractéristiques minimales d'une arithmétique exacte. Yamaguchi propose dans [YAMA 93] ce type de solution. L'intérêt est, l'étape d'analyse effectuée, de fournir au programmeur, une arithmétique transparente et adaptée au problème.

Yamaguchi propose dans le contexte d'un outil de modélisation en trois dimensions décrit dans [YAMA 87] d'utiliser une arithmétique exacte adaptée au calcul des déterminants 4x4. Les interprétations géométriques des déterminants qu'il propose sont suffisantes pour résoudre toutes les opérations de modélisation standard, avec les conventions suivantes :

Soit un point $V_A(X_A, Y_A, Z_A, w_A)$ en coordonnées homogènes et les trois sommets d'un triangle dans le sens trigonométrique direct $V_0(X_0, Y_0, Z_0, w_0)$, $V_1(X_1, Y_1, Z_1, w_1)$ et $V_2(X_2, Y_2, Z_2, w_2)$.

$$S_{A012} \equiv \begin{vmatrix} X_A & Y_A & Z_A & w_A \\ X_0 & Y_0 & Z_0 & w_0 \\ X_1 & Y_1 & Z_1 & w_1 \\ X_2 & Y_2 & Z_2 & w_2 \end{vmatrix} = N_x X_A + N_y Y_A + N_z Z_A + D w_A$$

$$\text{avec} \quad N_x = \begin{vmatrix} Y_0 & Z_0 & w_0 \\ Y_1 & Z_1 & w_1 \\ Y_2 & Z_2 & w_2 \end{vmatrix}, N_y = \begin{vmatrix} Z_0 & X_0 & w_0 \\ Z_1 & X_1 & w_1 \\ Z_2 & X_2 & w_2 \end{vmatrix}, N_z = \begin{vmatrix} X_0 & Y_0 & w_0 \\ X_1 & Y_1 & w_1 \\ X_2 & Y_2 & w_2 \end{vmatrix}, D = -\begin{vmatrix} X_0 & Y_0 & Z_0 \\ X_1 & Y_1 & Z_1 \\ X_2 & Y_2 & Z_2 \end{vmatrix}$$

En notant $N_{012} = [N_x, N_y, N_z, D]$ et $V_N(X_N, Y_N, Z_N, 0)$ le vecteur orthogonal au triangle, on peut définir les opérations suivantes :

- Test de convexité : si V_i, V_j et V_k sont trois sommets consécutifs dans le sens trigonométrique direct , V_j est un point de convexité si $S_{Nijk} > 0$ et de concavité si $S_{Nijk} < 0$.
- Point d'intersection d'un segment de droite avec un plan : V_i, V_j et V_k ne sont pas colinéaires si $S_{Nijk} \neq 0$. Dans ce cas le point d'intersection du segment de droite $V_A V_B$ avec le plan défini par V_i, V_j et V_k est $P = S_{Aijk} V_B - S_{Bijk} V_B$ en coordonnées homogènes.

- Test d'intersection de deux segments de droite : soient V_iV_j et V_AV_B deux segments de droite en trois dimensions, si $S_{ABij} = 0$ (coplanarité), $S_{AijN} \cdot S_{BijN} \le 0$ et $S_{IABN} \cdot S_{JABN} \le 0$, les segments de droite se coupent. Le point d'intersection est donné par l'équation précédente où l'on substitue l'indice k par N.

- Test d'appartenance d'un point à une face : On considère les sommets successifs V_i, V_j, V_k. Le point V_A appartient à la région triangulaire infinie définie par les droites V_iV_j et V_jV_k si $S_{AijN} < 0$ et $S_{AjkN} > 0$ pour les régions convexes et $S_{AijN} > 0$ et $S_{AjkN} < 0$ pour les régions concaves.

```
count = 1;
pour tous les sommets du polygone
      si S_ijkN < 0 et S_AijN ≤ 0 et S_AjkN > 0
         count = count - 1
      sinon
         si S_ijkN > 0 et S_AijN > 0 et S_AjkN ≤ 0
            count = count + 1
si count == 0
      V_A est hors du polygone
sinon
      V_A est dans le polygone
```

figure 7
Appartenance d'un point à une face (algorithme)

L'auteur assure que pour représenter des données contenues dans un cube de 1 mètre de coté avec une précision de 1 micron, seuls 20 bits sont nécessaires pour chacune des coordonnées. Dans ce cas le résultat le plus complexe (calcul des coordonnées d'une intersection) tiendra de manière exacte sur 141 bits (soit environ 5x32bits = 160 bits). Les opérations élémentaires décrites précédemment fournissent deux types de résultat : un point dans le cas des intersections, une valeur logique pour les tests d'appartenance. Dans le deuxième cas, seul le signe des déterminants est important. Cette remarque permet à Yamaguchi de rendre efficace le calcul du signe des déterminants en utilisant ce qu'il appelle une arithmétique adaptative. Cette arithmétique consiste à n'utiliser que les valeurs de poids le plus fort de l'arithmétique exacte et à vérifier que le résultat ainsi obtenu est supérieur à un seuil constant égal à l'erreur maximale que l'on puisse commettre (Yamaguchi évalue cette erreur à 7FFFFFFFC), dans ce cas le signe du déterminant est déterminé, sinon il est nécessaire de faire le calcul exact complet.

Yamaguchi ne fournit que des informations incomplètes sur les temps d'exécution. Il montre seulement que lorsque le nombre de bits servant à représenter une valeur exacte augmente, le coût du signe d'un déterminant reste statistiquement constant dans le cas de l'utilisation de son arithmétique adaptative.

3.4.3 Prise en charge par la compilation

Une solution récente due à S. Fortune et C. Van Wyk [FORT 93] propose de calculer de manière statique l'erreur (ou epsilon) tout au long des calculs. Cette solution, pour être efficace et se démarquer de ce que l'on trouve dans la littérature, passe par l'utilisation d'un "expression compiler". Ce pré-compilateur traduit les expressions arithmétiques du code source des programmes en un ensemble d'opérations qui évaluent de manière fiable ces expressions. Après avoir déterminé les fonctions sensibles du programme, les commandes données au pré-compilateur permettent automatiquement de modifier et d'adapter les opérations en fonction de la dynamique des données initiales. Ce pré-compilateur génère de manière explicite dans le programme les opérations nécessaires (et suffisantes) à une arithmétique exacte.

Les auteurs proposent aussi des calculs en deux étapes, le "floating-point filter", à la manière du calcul réticent de Moreau. L' "expression compiler" se charge de générer les tests et les deux parties du code adéquates :

- Calcul de l'erreur maximum.
- Calcul flottant direct.
- Test de validité du résultat par comparaison du résultat flottant et de l'erreur.
- Calcul exact en ligne si nécessaire.

Les auteurs annoncent des résultats encourageants (seulement deux fois plus lent que les flottants) pour des exemples ou les données initiales ont une faible dynamique. Ils mettent aussi en avant que leur méthode, en développant les calculs dans le corps du programme, gagne en efficacité par l'absence d'appels de fonctions et d'allocation dynamique.

La seule restriction du procédé de S. Fortune et C. Van Wyk semble être, pour l'instant, l'impossibilité de conserver la valeur des variables intermédiaires ou de génération supérieure à zéro.

Il s'agit là, peut-être, des bases d'un langage ou d'un compilateur qui prendrait enfin à sa charge les problèmes d'imprécision numérique.

3.5 Que penser de tout ça ?

Karasick, Yamaguchi, Fortune et Moreau utilisent, semble-t-il, des concepts proches : rendre fiables les calculs dont dépend la bonne marche de l'algorithme. Yamaguchi, Fortune et Moreau utilisent tous des valeurs de seuils qui permettent de décider s'il est ou non nécessaire d'utiliser les calculs exacts. Yamaguchi et Moreau stockent de manière exacte les résultats intermédiaires qui sont susceptibles d'être réutilisés. Karasick et Moreau doivent tous les deux tenir à jour au moins deux représentations des valeurs manipulées. Karasick et Yamaguchi se sont limités volontairement à des calculs de déterminants. Si Karasick utilise de manière explicite une arithmétique d'intervalles et utilise un processus itératif pour obtenir un résultat fiable (le nombre d'itérations dans le pire des cas dépend des données), Yamaguchi, de manière implicite, en tentant d'abord un calcul grossier avec les chiffres de poids fort de ces nombres et le calcul exact en cas d'échec, utilise le même principe. Les valeurs grossières initiales peuvent être vues comme les bornes inférieures d'un intervalle et dans son cas, le processus d'évaluation se fait en seulement deux étapes.

Dans tous les cas, le choix de l'endroit où doit intervenir l'arithmétique exacte est à la charge du programmeur. Si l'on compare ces quatre méthodes du point de vue de la représentation des données, Karazik ne peut manipuler que des données initiales (les seules qui sont fiables), S. Fortune et et C. Van Wyk ne savent pas encore stocker de manière exacte les résultats intermédiaires, Moreau est obligé de conserver continuellement les deux représentations (exacte et flottante), et Yamaguchi n'utilise que des données exactes. Aucun de ces auteurs ne fournit vraiment une solution souple. Aucun d'entre eux n'est capable de fournir un résultat intermédiaire qui pourra resservir, de manière fiable et efficace, sans effectuer un calcul exact.

Si l'on bâtit la liste des éléments indépendants des applications mis en œuvre par ces auteurs pour résoudre leurs problèmes d'imprécisions, on obtient les concepts généraux suivants :

- Utilisation d'un encadrement des résultats pour déterminer le seuil de fiabilité.
- Utilisation de représentations multiples : approchées/efficaces et exactes/lentes.

Dans tous les cas, le recours à des optimisations locales est lié au fait que c'est dans l'algorithme lui-même que l'on doit déterminer les endroits sensibles, en aucun cas l'arithmétique ne le prend à sa charge. C'est la principale critique que l'on puisse faire à ce type de solution bien que Fortune et Van Wyk aillent plus loin en déplaçant le problème vers la phase de compilation.

La paresse 4

La paresse est un concept assez général, qu'il est possible d'appliquer dans de nombreux domaines. Son principe général est simple : pour un problème ou une opération donnée, on dispose de multiples descriptions. Chaque description est capable de fournir, soit une réponse fiable à la question posée (oui, non), soit pas de réponse du tout (je ne sais pas). Chaque représentation permettra d'obtenir des informations de plus en plus précises, mais il doit obligatoirement subsister une représentation exacte pour permettre de lever toutes les ambiguïtés en cas d'échec. Pour que la paresse soit efficace, plus les renseignements fournis par une représentation sont évasifs, plus ils doivent être obtenus de manière efficace. Ainsi, le recours éventuel aux autres représentations n'est pas pénalisé par les calculs précédents.

Le recours à de telles méthodes est en fait assez commun de manière implicite. Tous les algorithmes géométriques utilisant un englobant quelconque (Lancer de rayon, Cameron [CAME 91],…) sont paresseux jusqu'à un certain point (d'intersection par exemple). D'autres, comme l'optimisation $\alpha\beta$ des algorithmes de recherche du *minmax* (pour les arbres de jeux), poussent plus loin la paresse en évitant de parcourir les sous-arbres de solutions sans intérêt. Dans tous les cas, le programmeur aura préféré une redondance d'informations ou un supplément de contrôle pour éliminer le plus possible les calculs inutiles quand ceux-ci sont longs et coûteux.

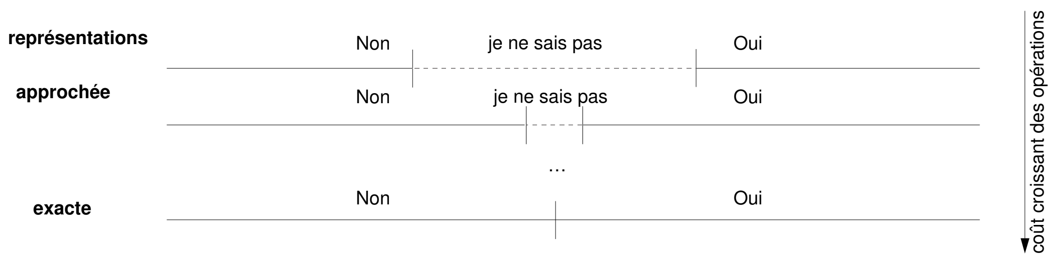


figure 8
Réponse paresseuse à une question

Le concept de paresse est un puissant outil d'optimisation applicable, au sens algorithmique, chaque fois que l'on peut utiliser des approximations du résultat ou des valeurs intermédiaires pour déterminer de manière précoce si l'on est en train de faire fausse route, ou pour fournir des informations sans ambiguïté dans les cas triviaux.

4.1 Une arithmétique exacte paresseuse

Les calculs effectués avec une arithmétique exacte sont coûteux en temps, ce n'est un secret pour personne. Comme nous l'avons vu depuis le début de ce chapitre, de nombreux auteurs ont tenté de fournir avec des fortunes diverses des solutions au problème suivant :

comment réconcilier l'exactitude et l'efficacité des calculs ?

Plaçons-nous dans le cas d'une arithmétique exacte rationnelle. Le coût des opérations a été étudié au § 3.3 et nous sommes bien forcés de constater l'inefficacité de ce type d'arithmétique. Par contre, comme l'explique [MORE 90], le seul endroit où il est vraiment indispensable de disposer d'une arithmétique exacte est lors d'une prise de décision (et peut-être pour le résultat final). Dans ce cas particulier, la décision doit être fiable mais les valeurs des données utilisées pour le test ne nous intéressent pas, seul le résultat de ce dernier compte. Ces remarques permettent de définir le mode de fonctionnement de l'arithmétique exacte paresseuse de la manière suivante :

Retarder le calcul exact jusqu'à ce qu'il devienne soit inutile, soit inévitable.

De cette manière les calculs exacts inutiles ne seront jamais effectués. Mais ce mode de fonctionnement induit deux choses :

- C'est l'arithmétique qui prend la décision de faire ou de ne pas faire un calcul exact.
- L'arithmétique doit tenir à jour l'historique de "fabrication" des nombres.

Toutes ces remarques nous permettent de mettre en place les principes généraux de l'arithmétique rationnelle paresseuse :

- Un nombre paresseux est une définition symbolique associée à un intervalle de deux nombres flottants qui est assuré de contenir ce nombre paresseux.
- Une expression symbolique est soit un nombre rationnel représenté sous la forme décrite au § 3.1 , soit une opération paresseuse.
- Une opération paresseuse est soit la somme ou le produit de deux nombres paresseux, soit l'opposé ou l'inverse d'un nombre paresseux.

Un nombre paresseux est donc la racine d'un arbre de nombres paresseux, dont les nœuds sont des opérations binaires (somme, produit) ou unaires (opposé, inverse) et les feuilles des nombres rationnels. Les nœuds et les feuilles de cet arbre peuvent être partagés entre plusieurs nombres paresseux. Ils forment plutôt un graphe orienté sans circuit ("Directed Acyclic Graph" ou DAG) qu'un arbre. Les nombres paresseux peuvent à tout moment connaître leur définition mais en aucun cas savoir dans quelles expressions ils interviennent.

Effectuer une opération paresseuse revient à construire la définition symbolique de cette opération et à calculer l'intervalle du résultat à l'aide d'une arithmétique d'intervalles. Ceci se fait en général en temps constant (sauf dans certain cas particuliers de l'inverse). L'opération exacte en nombres rationnels n'est pas effectuée et ne le sera jamais tant que l'intervalle associé au nombre paresseux est suffisant pour la suite des calculs.

Les calculs exacts rationnels sont seulement effectués dans deux situations bien définies :

- On tente de calculer l'inverse d'un nombre paresseux dont l'intervalle associé contient zéro.
- On compare deux nombres paresseux dont les intervalles associés se chevauchent (les nombres sont peut-être égaux).

Dans ces deux cas, l'arithmétique d'intervalles est impuissante à fournir un résultat fiable et on doit avoir recours aux calculs exacts pour lever toute ambiguïté. Nous appellerons cette opération l'évaluation d'une expression symbolique. Dans tous les autres cas, l'utilisation de l'arithmétique exacte est inutile pour les calculs ; seules les opérations sur les intervalles et les constructions symboliques sont effectuées. La gestion de la précision ainsi que de la paresse sont prises en charge au niveau de l'arithmétique elle-même, ce qui la rend, de fait, indépendante des algorithmes qui l'utilisent.

4.2 Les représentations multiples

Comme nous venons de le voir, pour être paresseuse, notre arithmétique doit disposer à chaque instant d'au moins deux représentations des nombres. Une représentation approchée mais efficace (les intervalles) et une représentation exacte, ou tout au moins une représentation lui permettant de retrouver la valeur exacte (les expressions symboliques). Le choix des représentations est dicté par l'efficacité de chacune d'entre elles : le coût de la construction des expressions symboliques et des opérations sur les intervalles doit être très inférieur à celui des calculs exacts si l'on veut espérer un gain de temps non négligeable.

Chaque représentation doit servir un but ultime : accélérer les calculs tout en restant fiable. L'arithmétique d'intervalles permet, de manière grossière, mais rapide, de répondre à tout un ensemble de questions, elle nous épargne un grand nombre de calculs exacts (qui sont très lents). Quand les intervalles ne suffisent plus, les expressions symboliques permettent de calculer partiellement ou complètement les nombres rationnels qui sont en cause, et en dernier ressort les opérations ou les comparaisons sont effectuées en rationnels.

Dans un premier temps, nous allons nous intéresser aux trois représentations essentielles dont doit disposer l'arithmétique paresseuse :

- Une arithmétique exacte rationnelle.
- Une arithmétique d'intervalles.
- Une arithmétique d'expressions symboliques.

Ces trois arithmétiques sont indispensables, mais il est évident que toutes les informations qui permettront de réduire le nombre de calculs exacts sont les bienvenues. Nous verrons dans les paragraphes suivants comment d'autres informations participent à l'amélioration des performances de l'arithmétique paresseuse.

4.3 Les intervalles

Les intervalles sont une extension des nombres réels et permettent de résoudre une large plage de problèmes. Un des plus anciens que l'on connaisse est sans doute l'encadrement du nombre π par la méthode d'Archimède. Pour déterminer le rapport entre la circonférence et le diamètre d'un cercle, Archimède considère les polygones à n cotés inscrits et circonscrits au cercle. Lorsque l'on augmente le nombre de cotés de chaque polygone, on obtient une suite de valeurs croissantes pour le périmètre du polygone inscrit et décroissante pour le polygone circonscrit. On a obtenu un intervalle encadrant de manière aussi fine que voulue la valeur π. Bien sûr, depuis Archimède, les arithmétiques d'intervalles ont trouvé bien d'autres applications comme l'encadrement des racines d'un polynôme, l'encadrement de fonctions. Le lecteur peut se reporter à [MOOR 66] pour des explications détaillées et à [SUFF 91] pour un exemple d'utilisation assez séduisant.

4.3.1 Arithmétique sur les intervalles réels

On définit un “intervalle réel” (“interval number”) comme étant une paire ordonnée de nombres réels, $[a, b]$, avec $a \le b$. L'"intervalle réel” dégénéré de la forme $[a, a]$ est équivalent au nombre réel a.

Un “intervalle réel” est aussi un ensemble de nombres réels. L'“intervalle réel” $[a, b]$ est l'ensemble des nombres x tels que $a \le x \le b$. Nous utiliserons la notation $\{x | P(x)\}$ pour “l'ensemble des nombres x qui vérifient la proposition $P(x)$”. On peut alors écrire :

$$[a, b] = \{x | a \le x \le b\} \qquad [6]$$

Nous utiliserons fréquemment les symboles $\in, \subset, \cup, \cap,$ dans leur sens commun de la théorie des ensembles.

- $x \in [a, b]$, le nombre réel x est dans l'intervalle $[a, b]$ $(a \le x \le b)$.
- $[a, b] \subset [c, d]$, l'intervalle $[a, b]$ est contenu au sens ensembliste dans l'intervalle $[c, d]$ mais pas nécessairement au sens strict $(c \le a \le b \le d)$.
- $[a, b] \cup [b, c]$ et $[a, b] \cap [c, d]$ sont pris au sens ensembliste classique.

Nous noterons φ, l'ensemble des "intervalles réels" fermés[1] (dont les bornes appartiennent à l'intervalle). Si $I \in \varphi$ alors il existe deux nombres réels a et b tel que $a \le b$ et $I = [a, b]$.

- On note la "largeur" d'un intervalle $[a, b]$ par $w([a, b]) = b - a$.
- On note la "magnitude" d'un intervalle par $|[a, b]| = \max(|a|, |b|)$.
- On définit un ordre partiel sur les éléments de φ par $[a, b] < [c, d]$ si et seulement si $b < c$.
- $[a, b] = [c, d]$ si et seulement si $a = c$ et $b = d$.

Nous introduisons maintenant une arithmétique pour les éléments de φ, les "intervalles réels". Cette arithmétique est une extension de l'arithmétique sur l'ensemble des réels et suppose que l'on utilise des nombres de précision infinie [MOOR 66].

- Si * est un des symboles +, -, ., /, nous définissons les opérations arithmétiques sur les intervalles par :

$$\begin{gathered} [a, b] * [c, d] = \{x * y \mid a \le x \le b, c \le y \le d\} \\ \text{nous ne définissons pas } [a, b] / [c, d] \text{ si } 0 \in [c, d] \end{gathered} \qquad [\,7\,]$$

- La définition des opérations sur une arithmétique d'intervalles donnée par l'équation [7] est théorique. Le calcul des bornes de l'intervalle résultat se fait de la manière suivante pour les quatre opérateurs (+, -, . et /) :

$$\begin{gathered} [a, b] + [c, d] = [a + c, b + d] \\ [a, b] - [c, d] = [a - d, b - c] \\ [a, b] \cdot [c, d] = [\min(ac, ad, bc, bd), \max(ac, ad, bc, bd)] \\ \text{si } 0 \notin [c, d], \text{ alors } [a, b] / [c, d] = [a, b] \cdot [1/d, 1/c] \end{gathered} \qquad [\,8\,]$$

- Nous pouvons identifier les intervalles dégénérés de la forme $[a, a]$ aux nombres réels. A l'aide de la définition [7], il est évident que dans ce cas particulier, l'arithmétique d'intervalles se réduit à une arithmétique ordinaire sur les nombres réels.
- La commutativité et l'associativité sont directement déduites de [7] pour l'addition et la multiplication d'intervalles. Si I, J et K sont des intervalles on a :

1. Nous n'utiliserons dans ce chapitre que des intervalles fermés, mais il existe aussi des intervalles ouverts (les bornes n'appartiennent pas à l'intervalle) et des intervalles partiellement ouverts ou fermés (à droite ou à gauche).

$$\begin{aligned} I + (J + K) &= (I + J) + K \\ I \cdot (J \cdot K) &= (I \cdot J) \cdot K \\ I + J &= J + I \\ I \cdot J &= J \cdot I \end{aligned} \qquad [\,9\,]$$

- Les nombres réels 0 et 1 sont respectivement les éléments neutres pour l'addition et la multiplication des intervalles.

$$\begin{aligned} 0 + I = I + 0 &= I \\ 1 \cdot I = I \cdot 1 &= I \end{aligned} \qquad [\,10\,]$$

- La loi de distributivité n'est pas applicable à l'arithmétique d'intervalle, un exemple suffit à le prouver :

$$[1, 2] \cdot ([1, 2] - [1, 2]) = [1, 2] \cdot [-1, 1] = [-2, 2]$$
$$[1, 2] \cdot [1, 2] - [1, 2] \cdot [1, 2] = [1, 4] - [1, 4] = [-3, 3]$$

  En utilisant le fait que les "intervalles réels" sont aussi des ensembles (de réels), on a une relation importante entre addition et multiplication d'intervalles : quels que soient les intervalles I, J et K on a :

  $I \cdot (J + K) \subset I \cdot J + I \cdot K$, cette propriété est la "sous-distributivité" [11]

- On a une propriété de distributivité. Si t est un réel : $t(I + J) = tI + tJ$

- Si J et K contiennent seulement des nombres réels de même signe ($J \cdot K > 0$), alors l'inclusion de [11] peut être remplacée par une égalité $I \cdot (J + K) = I \cdot J + I \cdot K$.

- Les inverses pour l'addition et la multiplication des intervalles existent seulement dans le cas des intervalles dégénérés (nombres réels).

- L'arithmétique d'intervalles est monotone au sens de l'inclusion, ce qui veut dire que si $I \subset K$ et $J \subset L$, alors

$$\begin{aligned} I + J &\subset K + L \\ I - J &\subset K - L \\ I \cdot J &\subset K \cdot L \\ I / J &\subset K / L \qquad (\text{si } 0 \notin L) \end{aligned} \qquad [\,12\,]$$

Cet ensemble de relations découle directement de la définition [7]. A partir de la transitivité de la relation d'inclusion ($I \subset J$ et $J \subset K \Rightarrow I \subset K$) et de l'ensemble de relations [12] nous obtenons le résultat suivant :

Théorème : Si $F(X_1, X_2, \ldots, X_n)$ est une expression *rationnelle* dans les variables de type intervalle $X_1, X_2, \ldots, X_n$, c'est à dire, une combinaison finie de $X_1, X_2, \ldots, X_n$ et un ensemble fini d'intervalles constants avec une arithmétique d'intervalles, alors

$$X'_1 \subset X_1, \ldots, X'_n \subset X_n \Rightarrow F(X'_1, \ldots, X'_n) \subset F(X_1, \ldots, X_n)$$

pour tous les ensembles d'intervalles $X_1, X_2, \ldots, X_n$ où les opérations arithmétiques sur les intervalles dans F sont définies.

4.3.2 Arithmétique sur intervalles arrondis

La représentation des nombres en virgule flottante impose que l'on prenne quelques précautions lors de la manipulation des intervalles. Nous avons vu au § 1.1 que le choix de l'arrondi va décaler le résultat toujours de la même manière.

Soient a et b deux nombres flottants (représentables en machine). Le nombre $x = a \operatorname{op} b$ n'est pas forcément un nombre représentable ($\nabla(x) \le x$ et $\Delta(x) \ge x$). Ainsi, le résultat du calcul en nombres flottants des bornes de l'intervalle d'une opération entre deux intervalles ne contiendra pas forcément le résultat exact de l'opération.

Soit $x \in [a, b]$ et $y \in [c, d]$, si l'arrondi se fait vers $+\infty$ le résultat de l'opération $x + y$ donnera l'intervalle $[\Delta(a + c), \Delta(b + d)]$ qui ne contient pas les valeurs qui se trouvent entre le nombre réel $a + c$ et le nombre machine $\Delta(a + c)$. Le résultat représentable englobant et le plus étroit possible de cette opération étant l'intervalle $[\nabla(a + c), \Delta(b + d)]$, on remarque que le type d'arrondi à utiliser dépend de la borne de l'intervalle que l'on manipule. Cette remarque met aussi en évidence le fait que lors des opérations, les arrondis opposés appliqués aux bornes de l'intervalle ont tendance à les éloigner, à faire grossir l'intervalle.

Pour tout nombre x appartenant à $\mathfrak{R}$ (ensemble des nombres réels), il existe deux valeurs $\nabla(x)$ et $\Delta(x)$ appartenant à l'ensemble des nombres représentables dans une arithmétique flottante (noté Σ) qui sont respectivement l'arrondi par défaut (vers $-\infty$) et l'arrondi par excès (vers $+\infty$) du nombre réel x.

$$\begin{gathered} \forall x \in \mathfrak{R}, \exists \nabla(x), \Delta(x) \in \Sigma, \nabla(x) \le x \le \Delta(x) \\ [\nabla(x), \Delta(x)] = \{ x | (\nabla(x) \le x \le \Delta(x)) \} \end{gathered} \qquad [13]$$

Cet intervalle est le plus petit intervalle encadrant la valeur réelle x qui soit représentable dans l'ensemble des nombres flottants Σ[1].

1. Par soucis de simplicité, nous ne tenons pas pas compte des éventuelles débordements dans ce paragraphe.

Soient deux nombres réels x et y encadrés respectivement par les intervalles $[\nabla(x), \Delta(x)]$ et $[\nabla(y), \Delta(y)]$. L'intervalle encadrant le résultat d'une opération entre x et y sera de la forme suivante d'après [7] :

$$[\nabla(x), \Delta(x)] \bullet [\nabla(y), \Delta(y)] = \{x \bullet y | \nabla(x) \le x \le \Delta(x), \nabla(y) \le y \le \Delta(y)\} \qquad [14]$$

Si l'on applique cette relation au cas de l'addition on obtient le résultat suivant:

$$[\nabla(x), \Delta(x)] + [\nabla(y), \Delta(y)] = [\nabla(x) + \nabla(y), \Delta(x) + \Delta(y)]$$

Les bornes de l'intervalle résultat n'ont aucune raison particulière d'appartenir à Σ (cf Norme IEEE 754 page 9)et pour obtenir un résultat convenable :

$$\begin{array}{cc} & \nabla(x) + \nabla(y) \le x + y \le \Delta(x) + \Delta(y) \\ & \text{et} \\ \nabla(x) + \nabla(y), \Delta(x) + \Delta(y) \in \Re & \left(\begin{array}{l} \nabla(\nabla(x) + \nabla(y)) \le \nabla(x) + \nabla(y) \le \Delta(\nabla(x) + \nabla(y)) \\ \nabla(\Delta(x) + \Delta(y)) \le \Delta(x) + \Delta(y) \le \Delta(\Delta(x) + \Delta(y)) \end{array}\right. \\ & \text{donc} \\ & \nabla(\nabla(x) + \nabla(y)) \le x + y \le \Delta(\Delta(x) + \Delta(y)) \end{array} \qquad [15]$$

$$[\nabla(x), \Delta(x)] + [\nabla(y), \Delta(y)] = [\nabla(\nabla(x) + \nabla(y)), \Delta(\Delta(x) + \Delta(y))] \qquad [16]$$

On remarque que l'encadrement du résultat n'est plus l'encadrement minimal dans Σ de la valeur $x + y$.

Les intervalles résultats des quatre opérations élémentaires pour les intervalles arrondis s'obtiennent de la même manière que dans le cas des intervalles exacts, avec les précautions à prendre pour le calcul des bornes comme nous venons de le voir en [15] et [16] :

$$\begin{array}{c} [a, b] + [c, d] = [\nabla(a + c), \Delta(b + d)] \\ [a, b] - [c, d] = [\nabla(a - c), \Delta(b - d)] \\ [a, b] \cdot [c, d] = [\nabla(\min(ac, ad, bc, bd)), \Delta(\max(ac, ad, bc, bd))] \\ \text{si} 0 \notin [c, d], \text{ alors } [a, b] / [c, d] = [a, b] \cdot [\nabla(1/d), \Delta(1/c)] \end{array} \qquad [17]$$

Soit un nombre flottant représentable de la forme $2^e m$, le nombre flottant immédiatement inférieur est $2^e(m - 2^{-p})$ et le nombre flottant immédiatement supérieur $2^e(m + 2^{-p})$ où p est la taille de la mantisse du nombre flottant manipulé.

$$\nabla(2^e m) = 2^e(m - 2^{-p}) \qquad \text{et} \qquad \Delta(2^e m) = 2^e(m + 2^{-p}) \qquad [18]$$

La largeur minimale d'un intervalle sera celle d'un intervalle de la forme $[\nabla(x),\Delta(x)]$ et aura pour valeur :

$$\text{si } x = 2^e m \qquad w([\nabla(2^e m),\Delta(2^e m)]) = \Delta(2^e m) - \nabla(2^e m) = 2^e 2^{1-p}$$

La façon dont la largeur des intervalles s'accroît dépend du type des opérations effectuées. L'étude du cas de l'addition et de la multiplication permet de tirer quelques enseignements.

Les formules suivantes ne sont valables que dans les cas où les deux intervalles ont leurs bornes supérieures à zéro, mais quelle que soit la configuration des intervalles, il est possible de trouver des relations analogues.

Addition :

$$w([a,b]+[c,d]) = w([\nabla(a+c),\Delta(b+d)]) = |\Delta(b+d) - \nabla(a+c)|$$

$$\text{si } [a,b] > [c,d] \quad \begin{cases} \Delta(b+d) = 2^{e_b}(m_b + 2^{e_d - e_b} m_d + 2^{-p}) = 2^{e_b} m_b + 2^{e_d} m_d + 2^{e_b} 2^{-p} \\ \nabla(a+c) = 2^{e_a}(m_a + 2^{e_c - e_a} m_c - 2^{-p}) = 2^{e_a} m_a + 2^{e_c} m_c - 2^{e_a} 2^{-p} \end{cases}$$

$$|\Delta(b+d) - \nabla(a+c)| = \left|w([a,b]) + w([c,d]) + 2^{-p}(2^{e_b} + 2^{e_a})\right| \cong \left|w([a,b]) + w([c,d]) + 2^{-p} 2^{e_b}\right|$$

Multiplication :

$$\text{si } [a,b] > [c,d] > [0,0] \qquad w([a,b]\cdot[c,d]) = w([\nabla(ac),\Delta(bd)]) = |\Delta(bd) - \nabla(ac)|$$

$$\Delta(bd) = 2^{e_b + e_d}(m_b m_d + 2^{-p}) \qquad \nabla(ac) = 2^{e_a + e_c}(m_a m_c - 2^{-p})$$

$$|\Delta(b\cdot d) - \nabla(a\cdot c)| = \left|2^{e_b + e_d} m_b m_d - 2^{e_a + e_c} m_a m_c + 2^{-p}(2^{e_b + e_d} + 2^{e_a + e_c})\right|$$

$$|\Delta(b\cdot d) - \nabla(a\cdot c)| \cong \left|w([a,b])\,w([c,d]) + 2^{e_c} m_c w([a,b]) + 2^{e_a} m_a w([c,d]) + 2^{-p} 2^{e_b + e_d}\right|$$

Dans le cas de l'addition, la largeur de l'intervalle du résultat est supérieure à la somme des largeurs des intervalles des opérandes. Pour la multiplication, la largeur de l'intervalle du résultat a un comportement qui dépend de manière plus complexe des opérandes, mais il aura toujours une largeur supérieure au produit des largeurs de ces opérandes.

Lorsque l'on utilise une représentation par des nombres flottants, la largeur réelle de l'intervalle est un mauvais indicateur de la précision du nombre que l'on manipule. La largeur de l'intervalle $[2^{-200},2^{-100}]$ est très inférieure à la largeur de $[2^{100},2^{101}]$, pourtant il y a beaucoup plus de nombres flottants représentables entre les bornes du premier intervalle qu'entre les bornes du second ($w([2^{-200},2^{-100}]) \approx 2^{-100}$ et $w([2^{100},2^{101}]) \approx 2^{100}$), signe que l'intervalle encadrant le résultat exact est devenu très imprécis.

Si on additionne ces deux intervalles on obtient $[2^{100},2^{101}]$, si on les multiplie $[2^{-100},2^{1}]$. Il paraît nécessaire de définir un indicateur plus adapté aux arithmétiques flottantes. Nous définissons la largeur d'un intervalle en fonction du nombre de flottants représentables qu'il contient et que nous noterons W.

Soit un intervalle $[a, b]$ avec $a = (-1)^{s_a} 2^{E_{bias} + e_a} m_a$ et $b = (-1)^{s_b} 2^{E_{bias} + e_b} m_b$, le nombre de flottants représentables appartenant à $[a, b]$, noté $W([a, b])$ est en base 2 :

- Soit $W([0, a])$ le nombre de flottants représentables compris entre 0 et a au sens large

$$W[0, a] = 2^p (E_{bias} + e_a) + m_a + 1$$

$$W([a, b]) = W([0, b]) - W([0, a])$$

$$W([a, b]) = 2^p ((-1)^{s_b} (E_{bias} + e_b) - (-1)^{s_a} (E_{bias} + e_a)) + (m_b - m_a) \qquad [19]$$

Pour les deux exemples précédents nous obtenons :

$$W([2^{-200}, 2^{-100}]) = 2^p ((E_{bias} - 100) - (E_{bias} - 200)) = 2^p \times 100$$

$$W([2^{100}, 2^{101}]) = 2^p ((E_{bias} + 101) - (E_{bias} + 100)) = 2^p$$

Cette nouvelle notion de largeur pour un intervalle flottant nous permettra plus tard de discriminer de manière correcte les intervalles très imprécis qui sont le résultat de nombreuses opérations de ceux dont la largeur est influencée par la magnitude des nombres flottants.

Des précautions particulières sont à prendre lors de l'implantation d'une telle arithmétique pour gérer de manière correcte les dépassements de capacité (overflows et underflows) si l'on ne dispose pas d'une arithmétique en nombres flottants respectant la norme IEEE 754.

4.4 Arithmétique sur les graphes orientés sans circuit

En plus de l'intervalle encadrant la valeur exacte de chaque nombre manipulé, il est nécessaire de conserver une trace de son histoire, de manière à pouvoir, le cas échéant, être capable de calculer sa valeur exacte. Cet historique est tout naturellement conservé sous sa forme symbolique la plus simple : un arbre d'expressions. Tous les nombres manipulés sont soit une donnée initiale du programme, soit le résultat d'une opération.

Toute valeur manipulée par l'arithmétique paresseuse est vue comme étant la racine d'un arbre d'expressions arithmétiques dont chaque nœud est une opération (unaire ou binaire) et chaque feuille un nombre rationnel (donnée initiale du programme ou calculé).

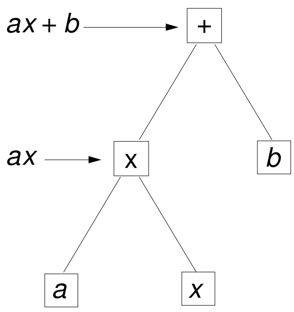


figure 9
arbre d'expression élémentaire

Tous les nombres sont d'un des types suivants :

- un nombre rationnel
- une opération binaire (addition ou multiplication) avec comme opérandes deux sous arbres, et comme valeur un intervalle encadrant le résultat de cette opération.
- une opération unaire (inverse ou opposé) avec comme opérande un sous arbre et comme valeur un intervalle encadrant le résultat de cette opération.

Toutes les entités que l'arithmétique manipule sont des nombres d'un des trois types vus précédemment et il est tout à fait possible de construire des expressions du type $ab + c/b$ sans être obligé de dupliquer le nombre b, ou le nombre n dans le calcul de factorielle n si on l'écrit de manière récursive ($\mathrm{fact}(n) = n \times \mathrm{fact}(n-1)$).

Le fait de pouvoir partager des expressions transforme l'arbre d'expressions initialement utilisé en un graphe sans circuit (il est impossible d'utiliser un résultat qui n'a pas encore été calculé). Une telle structure de données porte le nom de graphes orientés sans circuit ("direct acyclic graph" ou DAG en anglais). Son intérêt principal, outre le fait que son utilisation diminue le nombre de nœuds présents dans les expressions, est de faire bénéficier de l'évaluation d'un sous-arbre partagé tous les arbres qui l'utilisent.

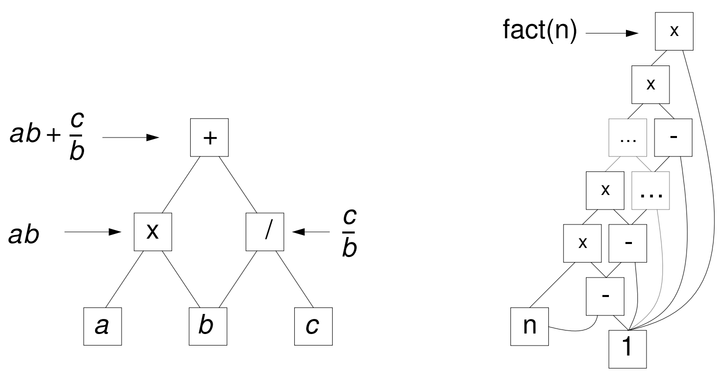


figure 10
Partage de valeurs dans une expression

Les diverses représentations de notre arithmétique en place, il s'agit maintenant de l'utiliser. Chaque nombre, comme nous venons de le voir, est soit une représentation exacte, soit une expression symbolique. Les opérations entre ces deux types de valeurs sont banalisées et se décomposent en trois cas dont le résultat est toujours une expression symbolique :

- Opération entre deux représentations exactes.
- Opération entre deux expressions symboliques.
- Opération entre une expression symbolique et une représentation exacte.

A aucun moment, pour l'instant il n'a été nécessaire de faire des calculs exacts, on se contente simplement de maintenir à jour de manière adéquate les encadrements des résultats exacts tant que cela s'avère possible. Notre arithmétique paresseuse se comporte, vue de l'extérieur, comme une arithmétique d'intervalles.

Le coût des opérations (+, -, *) est constant. Lors de chaque opération il est seulement nécessaire de créer le nœud du DAG représentant cette opération et de mettre à jour de manière correcte l'intervalle encadrant le résultat. La division, quant à elle, se fait aussi en temps constant si l'encadrement du diviseur ne contient pas zéro, mais provoque en revanche l'évaluation de tout ou partie de l'arbre d'expression dans le cas contraire. Le coût de l'opération dépendra de la complexité de l'expression représentant le dénominateur ; nous étudierons ce cas particulier dans le § 4.4.1 . Les calculs d'encadrement du résultat de l'opération sont réalisés par l'arithmétique d'intervalles en utilisant uniquement les encadrements des opérandes de l'opération.

Il faut remarquer que toutes les opérations s'accompagnent de l'occupation de places mémoire constantes. A chaque opération on doit créer un nouveau nœud dans le DAG (l'expression $ax+b$ utilise 1 nœud *, un nœud + et 3 feuilles si a, x et b sont des nombres rationnels). A la complexité en nombre d'opérations d'un algorithme vient s'ajouter une complexité en nombre de nœuds utilisés dans l'arbre qui est du même ordre. Ainsi un algorithme qui serait en $O(n^3)$ opérations sera aussi en $O(n^3)$ pour ce qui est de l'espace mémoire utilisé par les nœuds. Ce cas extrême n'est atteint que lorsque aucun calcul exact n'a été effectué, chaque évaluation faisant disparaître un certain nombre de nœuds dans le DAG. Dans l'autre cas extrême, celui où tout est évalué, la complexité en nombre de nœuds se rapproche de la complexité habituelle en place mémoire de l'algorithme. Il faut se garder de conclure hativement : la quantité de mémoire utilisée par les nœuds de l'arbre peut être importante, mais, lors d'une évaluation, ces nœuds seront remplacés par des nombres rationnels qui, eux aussi, peuvent occuper beaucoup de place mémoire.

4.4.1 Cas particulier de l'inverse d'un nombre paresseux

Les DAG étant une manière commode de conserver les informations nécessaires, ils doivent tenir compte des contraintes des arithmétiques et surtout ne jamais entrer en conflit avec elles si l'on veut pouvoir utiliser les expressions symboliques plus tard. Dans le cas des rationnels une division par zéro est illicite. Dans le cas des intervalles, une division par un intervalle $[a, b]$ dont les bornes ont des signes opposés donne comme résultat la réunion de $[-\infty, 1/b]$ et $[1/a, +\infty]$; l'intervalle connexe contenant cette réunion est $[-\infty, +\infty]$: cet encadrement ne nous serait pas d'un grand secours pour les calculs à venir. L'apparition d'un encadrement de nombre paresseux contenant zéro peut avoir deux causes :

- Les opérations qui ont fourni ce nombre ont fait grossir l'intervalle encadrant la solution exacte et maintenant il contient zéro.
- L'intervalle en question est bien celui du nombre zéro.

Dans tous les cas une évaluation s'impose. Il faut vérifier que la valeur mise en cause n'est pas zéro (dans le cas contraire on aurait affaire à une véritable division par zéro). Pour effectuer cette vérification on doit évaluer l'expression symbolique du nombre paresseux concerné. Cette opération consiste à calculer récursivement la valeur exacte de ce nombre et à remplacer la description symbolique par la valeur rationnelle. Une comparaison exacte entre ce nouveau rationnel et la valeur zéro permet de savoir si l'opération est licite ou non. Dans le cas où l'on tente une division par zéro, on se trouve face à une erreur et une exception de type “divide by zero” est générée (en général le programme s'arrête brutalement).

Pour la première fois, nous venons de rencontrer un cas où il est nécessaire de faire des calculs exacts. Il s'agit de la façon la plus simple de résoudre le problème posé, nous verrons dans les paragraphes suivants que d'autres méthodes peuvent être envisagées. L'arithmétique paresseuse prend d'elle même la décision d'évaluer un arbre d'expressions, dans ce contexte précis, pour être sûre de faire une opération valide. Cette décision est prise alors que les intervalles ne sont plus à même de fournir de manière fiable une réponse à la question : le diviseur est-il différent de zéro ?

4.4.2 Comparaison de deux nombres

Une arithmétique sans les opérateurs de comparaison est utilisable, bien sûr, mais limiterait fortement son domaine d'application. Pour pouvoir effectuer tous les tests entre deux nombres, il suffit d'avoir à sa disposition les deux opérateurs : supérieur et égal. Tous les autres opérateurs se construisent sans difficulté à partir de ces deux relations:

$a \neq b$	si $a = b$ alors rendre faux sinon rendre vrai
$a \leq b$	si $a > b$ alors rendre faux sinon rendre vrai
$a \geq b$	si $a > b$ ou $a = b$ alors rendre vrai sinon rendre faux
$a < b$	si $a \geq b$ alors rendre faux sinon rendre vrai

Dans tous les cas, la première étape de la comparaison se limite simplement à la comparaison des intervalles des deux opérandes. Si l'intersection entre les intervalles

$[a_m, a_M]$ et $[b_m, b_M]$ encadrant respectivement les valeurs a et b est vide, on est certain que ces deux valeurs sont différentes. Pour déterminer leur ordre, il faut comparer leurs bornes de la manière décrite au § 4.3 soit :

$$\text{si } ([a_m, a_M] \cap [b_m, b_M] = \phi) \text{ alors } \begin{cases} \text{si } a_M < b_m \text{ alors } a < b \\ \text{si } b_M < a_m \text{ alors } b < a \end{cases}.$$

Dans le cas où l'intersection entre les deux intervalles n'est pas vide, l'arithmétique d'intervalles est incapable de nous apprendre quoi que ce soit sur les deux nombres paresseux. Il faut avoir recours à une évaluation des expressions symboliques. Dans le cas où les opérandes sont des descriptions symboliques, on calcule les valeurs rationnelles qu'elles représentent puis on les compare de manière à fournir une réponse exacte.

Encore une fois, c'est le contexte qui dicte le comportement de l'arithmétique. Aucune évaluation ne sera faite si ce n'est pas nécessaire : c'est être paresseux.

On peut remarquer que l'utilisation des intervalles ne permet pas de garantir l'égalité de deux nombres paresseux et pour distinguer deux valeurs leurs intervalles ne doivent pas se recouvrir.

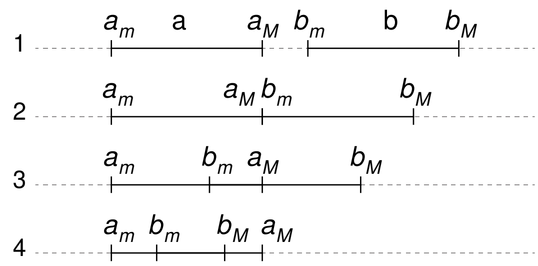


figure 11
Configuration des intervalles

4.5 Stratégies d'évaluation

Nous venons de voir dans les deux paragraphes précédents les deux seules situations dans lesquelles il est nécessaire d'évaluer des nombres paresseux. Cette évaluation doit permettre de pallier les insuffisances de précision des intervalles. Nous nous trouvons donc dans le cas où les deux nombres à comparer ont des intervalles qui se chevauchent (déterminer qu'un intervalle contient zéro est un problème équivalent : l'encadrement du nombre zéro est l'intervalle dégénéré $[0, 0]$).

4.5.1 Evaluation naïve

La version naïve de l'évaluation consiste à remplacer la définition symbolique des nombres paresseux à évaluer par leurs valeurs rationnelles. Cette opération se fait en évaluant récursivement l'arbre d'expressions. Pour chaque nœud de l'arbre, on rend le résultat de l'opération courante entre ces opérandes évalués. Certains nœud de l'arbre étant partagés, on peut faire profiter de cette évaluation tous les nombres paresseux qui partageaient un ou plusieurs nœuds de l'expression en cours d'évaluation en substituant les résultats intermédiaires aux sous-expressions correspondantes.

```
Rationnel Evaluer(noeud)
{
      if(Opérateur(noeud) == "+")
         return Evaluer(Opérande_gch(noeud)) + Evaluer(Opérande_dt(noeud));
      if(Opérateur(noeud) == "*")
         return Evaluer(Opérande_gch(noeud)) * Evaluer(Opérande_dt(noeud));
      …
}
```

figure 12
Evaluation d'une expression symbolique, algorithme de base.

Des versions plus sophistiquées de l'évaluation peuvent être envisagées si l'on tient compte de la façon dont sont propagés les calculs dans l'arbre et d'une propriété élémentaire des DAG.

4.5.2 Le partage

Par construction, certains nœuds des arbres d'expressions peuvent être partagés. Il se peut donc que l'on soit en train de tenter une comparaison entre deux nombres paresseux référençant la même expression symbolique. Dans ce cas, il suffit de comparer simplement l'adresse mémoire de ces deux arbres. Si une réponse positive nous est donnée, nous venons de déterminer trivialement l'égalité des nombres paresseux et nous pouvons fournir une réponse correcte à la comparaison. Bien entendu, dans le cas contraire, nous ne savons rien de plus, si ce n'est que les deux nombres ne partagent pas la même expression symbolique.

4.5.3 Le rafraîchissement

Nous avons vu que lors de l'évaluation naïve d'une expression, les nœuds partagés du DAG profitaient des calculs exacts entrepris. Il se peut que la représentation symbolique que nous tentons d'évaluer maintenant ait eu des sous expressions partiellement ou totalement évaluées dans le passé (à la suite d'une comparaison ou d'une division). Le principe de construction des DAG ne permettant pas d'avertir les expressions qui utilisent un sous-arbre partagé, les nouveaux encadrements des résultats n'ont pu être mis à jour et ne sont donc pas aussi précis que possible. Une étape de rafraîchissement de la valeur des intervalles peut permettre de préciser l'intervalle des nombres paresseux et les rendre disjoints si certains nœuds ont été évalués. Cette étape de rafraîchissement se réalise de manière élémentaire par la mise à jour récursive de tous les intervalles des nœuds de l'arbre.

4.5.4 Les aller-retour (YoYo)

La largeur des intervalles encadrant les valeurs exactes des nombres, quand ceux-ci sont représentés par des expressions symboliques, dépend du nombre et du type des opérations effectuées. De plus, seul l'intervalle encadrant une valeur exacte (un nombre rationnel) est de largeur minimale. Chaque fois que l'on réduira la profondeur de l'arbre (le nombre d'opérations) l'intervalle encadrant la nouvelle expression sera plus étroit, jusqu'à être égal à l'encadrement minimum du résultat rationnel. Cette convergence s'obtient facilement en évaluant de manière exacte les opérations les plus profondes de l'arbre d'expressions et en mettant à jour l'encadrement du résultat lors de chaque itération.

Cette remarque nous permet d'envisager une évaluation partielle de nos expressions par un processus itératif. Au lieu d'évaluer de manière naïve toute l'expression, on ne calcule de manière exacte que les nœuds les plus profonds de l'arbre (les nœuds dont les opérandes sont des rationnels) puis on rafraichit les intervalles. Cette opération réduit la largeur des encadrements des expressions, qui peuvent maintenant être disjoints. Ce processus est itéré tant que les intervalles ne sont pas disjoints ou tant que l'on n'a pas complètement évalué les expressions. On se ramène donc, dans un cas comme dans l'autre aux situations déjà étudiées. Nous avons donné le nom évocateur de YoYo à cette technique d'évaluation par va-et-vient.

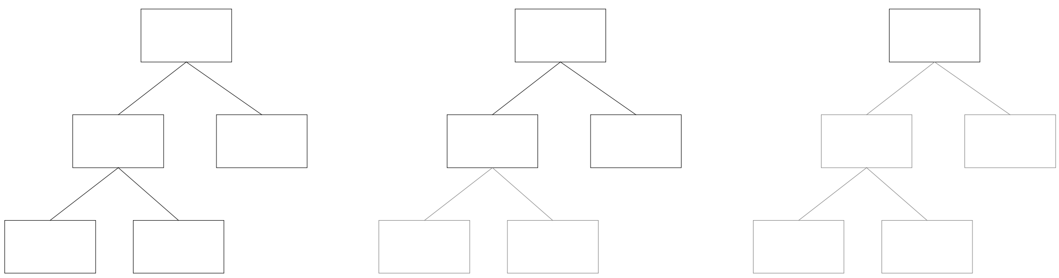


figure 13
évaluation par aller-retour (YoYo)

Cette dernière stratégie d'évaluation parait beaucoup plus coûteuse que la méthode naïve, mais il n'en est rien. Dans le pire des cas, on aura complètement évalué l'arbre d'expressions concerné et le seul surcoût est celui du rafraîchissement des intervalles. Empiriquement, les évaluations naïves ou de type YoYo ont des coûts pratiquement identiques. C'est sans doute dû au fait que lorsque tous les autres tests ont échoué, il se trouve que le plus souvent, on est en train de comparer des quantités numériquement identiques.

4.5.5 Utilisation de la largeur des intervalles

Quelle que soit la méthode d'évaluation, à chaque étape nous disposons d'un intervalle plus précis pour chacun des nombres. Il est évident que, avant de se lancer dans une comparaison exacte, nous utilisons ces informations pour vérifier que les nombres ne sont pas devenus distinguables par leurs intervalles. Comme l'évaluation de deux nombres ne peut se faire que de manière séquentielle (sur un machine séquentielle), le résultat de l'évaluation du premier arbre va peut-être permettre de rendre vide l'intersection des deux intervalles.

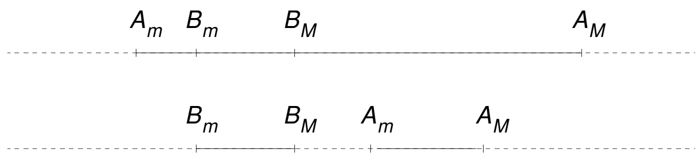


figure 14
stratégie d'évaluation : le plus large d'abord

La sequentialité des opérations implique un ordre dans les diverses méthodes d'évaluation. On a intérêt à tenter en premier l'évaluation du nombre dont l'encadrement est le plus large, sa réduction aura plus de chance d'entraîner une différenciation rapide. Cette méthode pour le choix du nombre à évaluer en premier est applicable à la stratégie d'évaluation par YoYo et doit être appliquée à chaque itération. L'utilisation de la largeur d'un intervalle décrite au § 4.3.1 ($w([A_m, A_M]) = A_M - A_m$) n'est pas très adaptée aux intervalles de nombres flottants que nous manipulons. Nous lui préférerons la définition du § 4.3.2 qui permet de connaître le nombre de flottants représentables entre les bornes de l'intervalle et ainsi de s'affranchir des problèmes de magnitude des nombres.

4.6 Problème de l'égalité de deux nombres

Le cas le plus défavorable que l'on puisse rencontrer dans le fonctionnement de l'arithmétique paresseuse tel que nous venons de la décrire est celui de la comparaison de deux nombres numériquement identiques mais dont les définitions sont différentes. En effet, dans ce cas, les intervalles encadrant les deux valeurs ont une intersection non vide et le recours au calcul exact sera inévitable. Il s'agit donc d'un point qu'il faudra tout particulièrement étudier. C'est à cet endroit précis que la paresse doit faire des miracles.

4.7 Isomorphisme

Dans les cas où l'arithmétique d'intervalles est devenue insuffisante pour déterminer l'ordre entre deux nombres, nous venons de voir qu'il était nécessaire d'évaluer les expressions représentant les valeurs exactes. Mais on peut détecter cette égalité sans aucun calcul exact dans le cas où les deux expressions symboliques sont identiques. Ce cas se présente assez fréquemment lors des calculs géométriques. On peut citer à titre d'exemple la comparaison de la pente de deux segment.

```
if(pente(s1) == pente(s2))…
```

Les deux appels de la fonction `pente(s)` fabriquent deux expressions symboliques identiques qui ne se différencient que par la valeur de leurs feuilles. Nous appellerons "clones" deux expressions de définitions identiques. Si les feuilles sont identiques (cas de deux segments confondus), la comparaison telle que nous l'avons définie dans les paragraphes précédents a recours à une évaluation complète des expressions pour déterminer leur égalité.

La détection de l'égalité de ce type d'expressions peut se faire sans évaluation en comparant de manière récursive les nœuds de chaque expression (figure 15 p. 50). Le nombre de comparaisons que nécessite cet algorithme est proportionnel au nombre de nœuds présents dans les expressions, soit une complexité en $O(n)$.

```
booléen Clone(a,b : Paresseux)
{
      si adresse(a) == adresse(b)
         rendre vrai;
      si intersection(encadrement(a),encadrement(b)) == vide
         rendre faux;
      si feuille(a) et feuille(b)
         rendre a == b;
      si opérateur(a) == opérateur(b)
         si opérateur(a) == "binaire"
            rendre
               (Clone(filsG(a),filsG(b)) et Clone(filsD(a),filsD(b))
         si opérateur(a) == "unaire"
            rendre
               (Clone(fils(a),fils(b))
      sinon rendre faux;
}
```

figure 15
Détection de deux arbres d'expressions identiques (clones).

Si l'algorithme que nous venons de présenter permet de détecter des clones, il est incapable de détecter les permutations sur les opérateurs commutatifs (addition et multiplication). Nous appellerons isomorphisme d'expressions l'égalité de deux expressions symboliques à la commutativité près ($a + b = b + a$ et $ab = ba$).

Nous définissons l'isomorphisme de deux DAG de la manière suivante :

Soient A et B deux DAG, A est isomorphe à B si et seulement si, il existe une bijection f des sommets de A dans les sommets B qui conserve la sémantique des sommets, et qui induise une bijection des arètes de A dans les arètes de B [BERG 83].

La sémantique des sommets est définie de la manière suivante :

Quel que soit s un sommet de A :

- Si s est un rationnel, $f(s)$ est un rationnel égal à s.
- Si s est une opération choisie dans {+, *, inv, opp}, $f(s)$ est la même opération.

La détection de l'isomorphisme d'arbres lorsque ceux-ci n'ont pas de valeur sémantique attachée à leurs sommets se fait en temps proportionnel au nombre de sommets ($O(n)$), et il en est de même dans le cas où chaque sommet porte une valeur unique dans l'arbre [AHO 74]. Une version plus simple à implanter que l'algorithme de [AHO 74] dans le cas d'un arbre où les sommets portent une valeur unique permet d'obtenir une complexité linéaire dans le cas des arbres binaires. Pour chaque sommet de l'arbre, il suffit de trouver les fils qui se correspondent en $O(d^2)$ comparaisons (d est le degré des sommets) ou en $d \log d$ si on les trie au préalable. Si tous les sommets sont en correspondance on continue récursivement, sinon les arbres ne sont pas isomorphes. La détection de l'isomorphisme se fait alors en $O(nd^2)$ (d^2 est une constante égale à quatre dans le cas des arbres binaires).

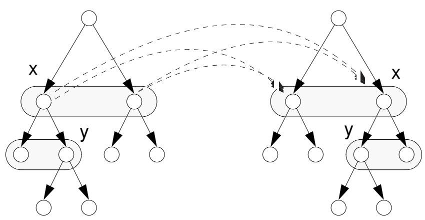


figure 16
Détection de l'isomorphisme de deux arbres.

Le problème que nous avons à résoudre est intermédiaire entre les deux problèmes précédents. Dans le cas général, la sémantique[1] (et la commutativité de l'addition et de la multiplication) ajoutée à chaque sommet nous impose de parcourir l'ensemble des permutations de l'arbre et rend la détection de l'isomorphisme exponentielle

1. il est impossible de différencier deux nœuds + ou deux nœuds *.

($O(2^{2n-1})$ où n est le nombre de feuilles d'un arbre binaire). Mais l'ajout des comparaisons entre les opérateurs, les intervalles associés aux sommets et les rationnels nous rapproche du problème où chaque sommet porte une valeur unique dans l'arbre. Sans prétendre transformer la complexité du problème, expérimentalement la détection de l'isomorphisme se fait en général avec un coût linéaire (il subsistera toujours des cas où la détection de l'isomorphisme sera de coût exponentiel). Un prétraitement des expressions par une recherche d'isomorphisme d'arbres permettrait d'éliminer tous les cas d'arbres non isomorphes en $O(n)$ opérations.

Les algorithmes que nous venons de décrire s'appliquent à des arbres ou à des DAG si ceux-ci sont parcourus comme des arbres. Si, lors de la détection de l'isomorphisme, deux sommets en correspondance partagent la même sous-expression il est inutile de vérifier que les deux sous-arbres sont isomorphes : ils sont identiques par construction. L'utilisation des opérateurs unaires opposé et inverse permet de traduire l'absence de commutativité des opérations soustraction et division et réduit le nombre des permutations possibles dans l'arbre.

Le coût de la vérification symbolique de cette égalité est très inférieur à celui de l'évaluation rationnelle des nombres malgré son aspect très récursif et elle ne fait aucun calcul exact. Si les deux arbres d'expressions sont très différents, on s'en aperçoit rapidement : il suffit seulement d'un nœud ou d'une feuille qui différent. Pour accélérer les comparaisons, on a intérêt à partager le plus possible les expressions identiques. Les résultats expérimentaux obtenus avec les algorithmes décrits au chapitre 6 ont montré que les configurations particulières des arbres d'expressions permettaient de détecter rapidement leur égalité ou leur inégalité, et que le fait de faire le test complet

```
booléen Isomorphe(a,b : Paresseux)
{
      si adresse(a) == adresse(b)
         rendre vrai;
      si intersection(encadrement(a),encadrement(b)) == vide
         rendre faux;
      si feuille(a) et feuille(b)
         rendre a == b;
      si opérateur(a) == opérateur(b)
         si opérateur(a) == "binaire"
            rendre
               (Isomorphe(filsG(a),filsG(b)) et Isomorphe(filsD(a),filsD(b)
                  ou
               (Isomorphe(filsG(a),filsD(b)) et Isomorphe(filsD(a),filsG(b)
         sinon
            rendre Isomorphe(fils(a), fils(b));
      sinon rendre faux;
}
```

figure 17
Détection de l'isomorphisme de deux arbres d'expressions.

d'isomorphisme plutôt que celui des clones ne ralentissait pas l'arithmétique et détectait plus d'expressions identiques.

4.8 Appartenance-Union (Union-Find)

Après avoir fait des calculs coûteux pour déterminer que $a = b$ et que $b = c$, nous aimerions que la comparaison de a et de c donne immédiatement $a = c$. Pour obtenir ce résultat, il faut que l'arithmétique paresseuse se souvienne des deux premiers résultats. Ce problème de regroupement de quantités égales est celui de la gestion des classes d'équivalence (dans notre cas, des classes d'équivalence pour l'égalité). Nous avons vu dans les paragraphes précédents (Comparaison de deux nombres et Isomorphisme) comment le regroupement par partage de toutes les expressions symboliques identiques permettait de rendre plus efficaces les comparaisons entre nombres et la détection d'isomorphisme.

La technique classique de gestion de classes d'équivalence est celle de l'appartenance-union ou “Union-Find”. Les résultats théoriques dûs à R. E. Tarjan [TARJ 72] montrent que le regroupement des classes d'équivalence se fait en $O(n\alpha(n))$ opérations, où α est l'inverse de la fonction d'Ackermann. Cette complexité est presque linéaire.

La solution classique à ce problème [SEDG 89] consiste à attacher à chaque nœud un représentant, initialement lui même. On définit l'archétype d'un nœud comme lui même s'il est son propre représentant, sinon comme l'archétype de son représentant. Lors d'une comparaison entre deux nœuds a et b, on vérifie d'abord (récursivement) que l'archétype de a n'est pas l'archétype de b auquel cas ils appartiennent déjà à la même classe d'équivalence, sinon on vérifie par d'autre moyens s'ils appartiennent à la même classe d'équivalence. En cas de réponse positive, on définit le représentant de b, le représentant de l'archétype de b, le représentant de a comme étant l'archétype de a (ou le contraire, c'est sans importance). Cette compression des chemins assure que l'archétype d'un nœud est toujours trouvé en temps quasi constant (figure 18).

L'application à l'arithmétique paresseuse de cette technique est évidente. Lors de la comparaison de deux expressions symboliques, on vérifie par tous les moyens disponibles que ces expressions ne sont pas égales (partage, union-find, isomorphisme, évaluation…). Si les deux expressions ont le même archétype, elles sont égales et

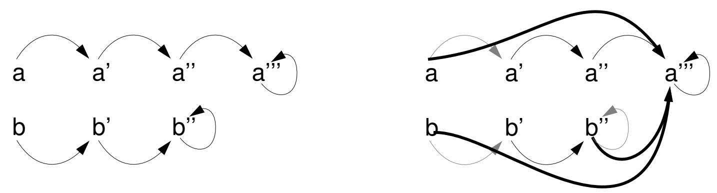


figure 18
Regroupement des classes d'équivalence

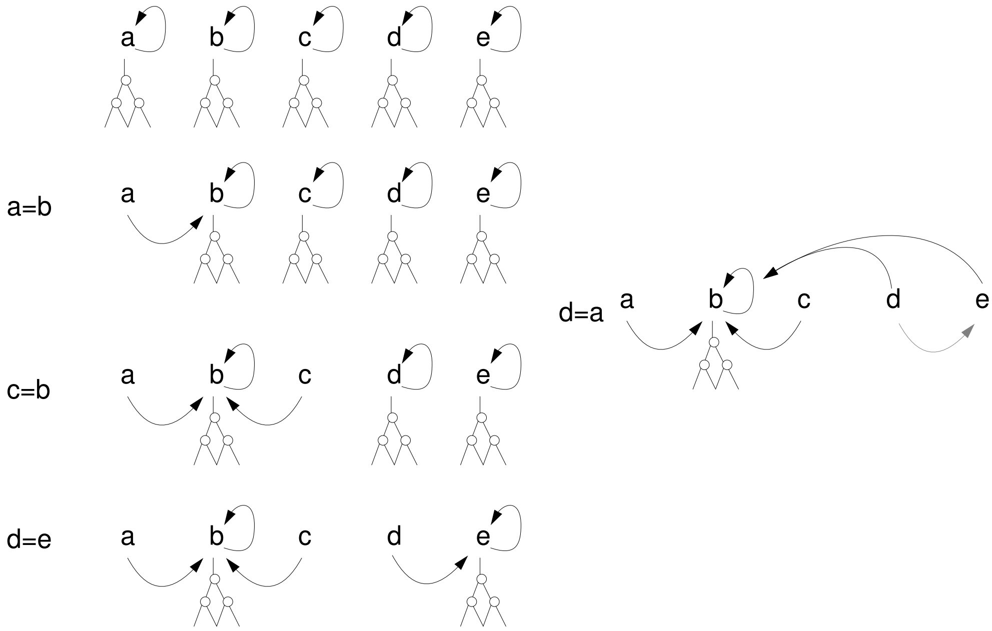


figure 19
Union-Find, déroulement de l'algorithme

appartiennent déjà à la même classe d'équivalence. Si on a déterminé leur égalité d'une autre manière, on abandonne la définition d'un des deux nombres (ils ont des définitions égales mais qui ne sont pas partagées) et on définit son représentant comme nous venons de le voir (figure 19).

Un inconvénient de cette méthode est dû au principe de détection de l'égalité des nombres. Si deux nombres sont trouvés égaux après une étape d'évaluation[1], ils seront bien regroupés dans la même classe d'équivalence, mais si survient une nouvelle comparaison avec une expression isomorphe à l'une des expressions précédentes avant évaluation, nous serons à nouveau forcés de procéder à une évaluation (l'expression précédente ayant été évaluée lors de la première comparaison). Une solution radicale à ce problème est proposée au paragraphe 8.1 page 95.

Les techniques de d'Appartenance-Union (Union-Find) nous permettent de rajouter un test efficace pour déterminer que deux nombres sont identiques, et de construire dynamiquement et incrémentalement les classes d'équivalence de nombres égaux.

Une méthode inspirée des techniques de l'Union-Find, le Pseudo-Union-Find, nous permet de nous rapprocher du DAG idéal en utilisant uniquement les informations du partage de données dans les expressions (figure 20). A chaque expression est associée un compteur qui indique par combien d'autres expressions elle est partagée. Lors de chaque comparaison, si l'on rencontre deux expressions identiques, on remplace la

1. Lors d'une évaluation, l'expression symbolique est remplacée par sa valeur rationnelle (cf § 4.5).

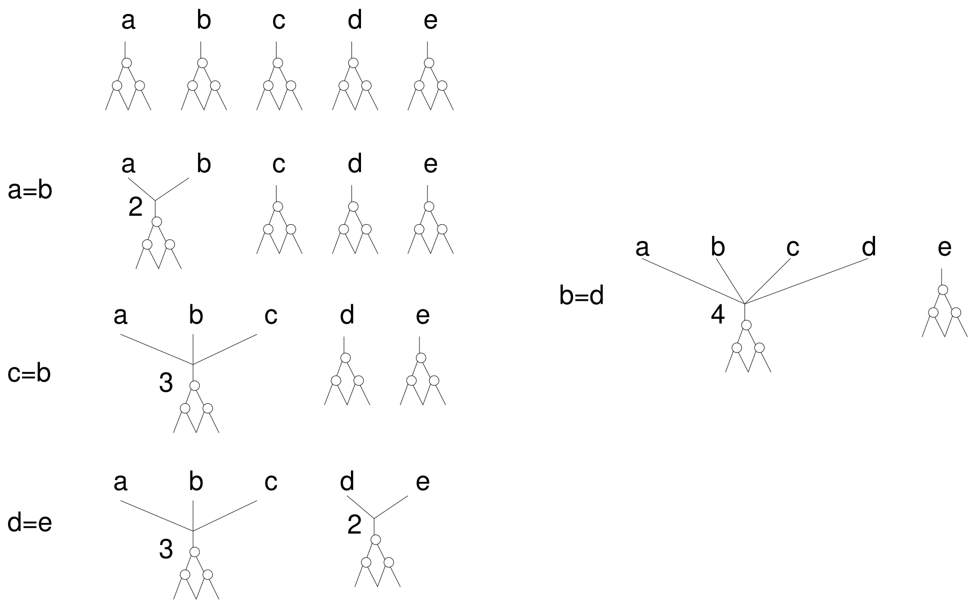


figure 20
Pseudo Union-Find

définition de celle dont la valeur du compteur est la plus petite par celle dont la valeur du compteur est la plus grande puis ce compteur est incrémenté. De cette manière, on optimise (de manière lente) le partage des expressions, améliorant par la même occasion l'efficacité de la comparaison et la gestion de la mémoire. Le pseudo Union-Find est implantable de manière triviale sur les DAG lorsque le partage est géré par des compteurs de référence. Lors des comparaisons, la détection de l'égalité se fait par simple comparaison d'adresse. Cet algorithme converge toujours (il ne peut pas apparaître de cycles), plus lentement que l'Appartenance-Union car dans certains cas il détruit en partie le travail précédent. Son principal avantage est d'utiliser uniquement les informations déjà disponibles dans le DAG et, lors des comparaisons, il n'est pas nécessaire de comparer les archétypes, le test d'égalité des adresses en mémoire le prenant en charge.

4.9 Optimisations symboliques

Les problèmes soulevés par la comparaison des nombres paresseux ou la détermination de leur signe nous poussent à envisager certains traitements symboliques des opérations. Ces traitements, toutefois, ne doivent pas dégrader l'efficacité générale de l'arithmétique. Le traitement symbolique des expressions est d'un coût rapidement excessif. On doit se limiter aux simplifications, qui sans entraîner un surcoût important, vont permettre une amélioration notable des performances. L'arithmétique doit rester paresseuse et ne pas devenir symbolique.

Nous avons délibérément choisi de nous restreindre aux simplifications élémentaires liées aux éléments neutres de chaque opération.

- Addition : $$\left\{\begin{aligned} a+0 &= a \\ a+(-a) &= 0 \end{aligned}\right.$$
- Opposé : $$-(-a) = a$$
- Multiplication : $$\left\{\begin{aligned} a\times 1 &= a \\ a\times(-1) &= -a \\ a\times 0 &= 0 \\ a\times\left(\frac{1}{a}\right) &= 1 \end{aligned}\right.$$
- Inverse : $$\frac{1}{\left(\frac{1}{a}\right)} = a$$

Toutes ces simplifications sont efficaces si les éléments neutres sont uniques et propres à l'arithmétique. De cette manière, les comparaisons entre les éléments neutres s'arrêteront lors de la comparaison des adresses mémoires. Le choix des éléments particuliers que nous traitons à part se limite aux valeurs 1, 0 et -1.

4.10 Clefs et arithmétique modulaire

Fréquemment, les algorithmes géométriques ont recours aux tables d'adressage dispersé (ou hashtables) pour accélérer la recherche d'éléments, tels que des points, des segments ou des plans. A chacun de ces éléments est associé une clef calculée à partir des valeurs numériques qui le représentent (coordonnées des points, pente des droites…). Lorsque l'on utilise l'arithmétique paresseuse, on est confronté au problème du calcul de la clef , la valeur exacte des nombres manipulés n'étant pas forcément disponible.

Les tables à adressage dispersé sont une façon efficace pour retrouver des informations qui ne peuvent pas être conservées dans des structures à accès direct comme les tableaux et dont la recherche fréquente interdit le stockage sous forme de listes. L'utilisation des tableaux est limitée par la place mémoire disponible pour permettre un accès en O(1), chaque valeur devant implicitement être un index unique. Dans le cas de l'utilisation d'une liste, l'accès aux données est une fonction du nombre de données utilisées ($O(n)$) ce qui rend son utilisation quadratique lors d'accès répétés aux données (n recherches dans une liste de longueur n ont une complexité en $O(n^2)$).

A chaque entité à conserver dans une table à adressage dispersé est associée une clef (la clef de hashage ou Hkey) qui doit avoir la propriété suivante:

$$\mathrm{clef}(a) \neq \mathrm{clef}(b) \Rightarrow a \neq b$$

Ces clefs doivent avoir une bonne dispersion (deux clefs identiques n'impliquant pas l'égalité des éléments) pour obtenir un accés optimal aux éléments de la table. Une table à adressage dispersé bien conçue permet de retrouver un élément quelconque en un temps en moyenne constant [KNUT 81] (en général 3 ou 4 comparaisons suffisent).

Soit $\mathbb{Z}$ l'ensemble des entiers et $p \in \mathbb{Z}_+^*$ un grand nombre premier. A chaque nombre paresseux z on peut associer une clef $\Psi(z) \in [\,0, p\,[$ telle que $\Psi(z) \neq \Psi(z') \Rightarrow z \neq z'$ pour tous les nombres paresseux z et z'.

La relation binaire définie dans $\mathbb{Z}$ par $x \equiv y\,[p]$ si et seulement si $\exists k \in \mathbb{Z}$ tel que $x - y = k \times p$ nous transporte dans $\mathbb{Z}_p = \mathbb{Z}/p\mathbb{Z}$. Nous noterons $x\%p$ l'unique entier dans $[\,0, p\,[$ reste de la division euclidienne de x par p.

Nous pouvons maintenant définir la clef d'un nombre rationnel. Par définition, à chaque entier $x \in \mathbb{Z}$ correspond une clef de hashage $\Psi(x) = x\%p$. Les nombres rationnels sont représentés sous la forme canonique x/y avec $x \in \mathbb{Z}$, $y \in \mathbb{Z}_+^*$ et $\mathrm{pgcd}(x, y) = 1$. La clef de hashage $\Psi(\frac{x}{y})$ est définie de la manière suivante :

- Si $\mathrm{pgcd}(y, p) = 1$, il existe une unique valeur, notée y^{-1} appartenant à $]0,p\,[$ qui est l'inverse de y dans $\mathbb{Z}_p$. Dans ce cas, on définit

$$\Psi(\frac{x}{y}) = (x \cdot y^{-1})\,\%p \equiv (\,(x\%p) \cdot (y\%p)^{-1})\,\%p \qquad [\,20\,]$$

  Seule la seconde forme de cette expression est utilisée par souci d'efficacité, et ne peut s'appliquer qu'à des fractions dont le pgcd n'est pas un multiple de p.

- Si $\mathrm{pgcd}(y, p) \neq 1$ ($y = k \cdot p$ pour $k \in \mathbb{Z}$), y n'admet pas d'inverse dans $\mathbb{Z}$. Nous définissons $0^{-1} = \Omega$, $\Omega^{-1} = 0$, $x \cdot 0 = 0$ et $x \cdot \Omega = \Omega$ si $x \neq 0$ avec Ω ayant une valeur en dehors de $]0,p\,[$ ($\Omega = p$ par exemple). En appliquant la relation [20] nous obtenons :

$$\Psi(\frac{x}{y}) = (x \cdot 0^{-1})\,\%p = (x \cdot \Omega)\,\%p = \Omega \qquad [\,21\,]$$

  La probabilité qu'un tel cas se présente $(\frac{1}{p})$ est faible et dépend de la taille du nombre p.

Nous définissons maintenant une clef de hashage pour une expression symbolique. La clef de hashage $\Psi(z)$ d'une expression symbolique z se calcule en appliquant les propriétés bien connues de $\mathbb{Z}_p$ dans les cas courants :

$$
\begin{aligned}
\Psi(z+z') &= (\Psi(z)+\Psi(z'))\,\%p \\
\Psi(z\cdot z') &= (\Psi(z)\cdot\Psi(z'))\,\%p \\
\Psi(-z) &= (-\Psi(z))\,\%p = p-\Psi(z) \\
\Psi(z^{-1}) &= \Psi(z)^{-1}\%p \\
\Psi(0^{-1}) &= \Omega
\end{aligned}
\qquad [\,22\,]
$$

Ce principe permet de calculer la clef de hashage d'un nombre qui n'a pas été évalué, mais surtout fournit des clefs de hashage identiques pour tous les nombres paresseux qui donneront des résultats rationnels identiques après évaluation (à l'exception des cas indéterminés que nous étudierons) :

$$\Psi\left(\frac{1}{2}+\frac{11}{3}\right) = \Psi\left(\frac{5}{3}\times\frac{5}{2}\right) = \Psi\left(\frac{25}{6}\right)$$

Dans les relations précédentes, $\Psi(z)$ et $\Psi(z')$ sont différents de Ω. Si ce n'est pas le cas, il est nécessaire d'utiliser des règles d'usage simples pour calculer les clefs :

$$
\begin{aligned}
\Omega\times\Omega &= \Omega \\
\Psi(z)\times\Omega = \Omega\times\Psi(z) &= \Omega \qquad \forall\Psi(z)\in[\,1,p\,[\\
\Psi(z)+\Omega = \Omega+\Psi(z) &= \Omega \qquad \forall\Psi(z)\in[\,0,p\,[\\
-\Omega &= \Omega \\
0^{-1} = \Omega \qquad \text{et} \qquad \Omega^{-1} &= 0
\end{aligned}
\qquad [\,23\,]
$$

Ces relations laissent indéterminés deux cas :

- $0\times\Omega = \Omega\times 0 = ?$

  Si on considère le nombre rationnel $a\times b$ avec $a = p/1$ et $b = k/p$ avec $k\in[\,1,p\,[$:

  $$\Psi(a) = 0 \qquad \Psi(b) = \Omega \qquad \Psi(a\times b)\equiv\Psi(k) = k$$

  $\Psi(a\times b)$ peut prendre n'importe quelle valeur k dans $[\,1,p\,[$. Si on choisit $a = p$ et $b = 1/p^2$, alors $\Psi(a)=0, \Psi(b)=\Omega$ et $\Psi(a\times b) = \Psi(1/p) = \Omega$.

- $\Omega+\Omega = ?$

  Si on considère le nombre rationnel $a+b$ avec $a = 1/p$ et $b = (k-1/p)$ ($k\in[\,0,p\,[$) :

  $$\Psi(a) = \Psi(b) = \Omega \qquad \Psi(a+b)\equiv\Psi(k) = k$$

  $\Psi(a+b)$ peut prendre n'importe quelle valeur k dans $[\,0,p\,[$. Si on choisit $a = 1/p$ et $b = 1/p$, dans ce cas $\Psi(a) = \Omega$, $\Psi(b) = \Omega$ et $\Psi(a+b) = \Psi(2/p) = \Omega$.

Dans ces deux cas particuliers, la solution la plus simple pour calculer la clef de hashage des expressions est d'évaluer de manière exacte z et de déduire $\Psi(z)$ du résultat rationnel comme nous l'avons vu précédemment. Ces cas d'indétermination de la clef de hashage ne sont pas très fréquents et n'ont pas une influence notable sur les performances de l'arithmétique paresseuse (ce cas ne s'est jamais produit lors de nos expérimentations avec $p = 65521$).

4.10.1 Arithmétique dans $\mathbb{Z}_p$

Si la somme, le produit ou l'opposé se calculent en un temps constant en utilisant les propriétés de $\mathbb{Z}_p$, l'inverse est moins simple à obtenir. Le calcul de la valeur u^{-1} pour tout u tel que $u \not\equiv 0\,[p]$ se fait en appliquant un des théorèmes suivants :

- Théorème de Fermat :

  Soit p un nombre premier, si u est un entier tel que $\mathrm{pgcd}(p, u) = 1$ (u n'est pas un multiple de p), alors $u^{p-1} \% p = 1$. On en déduit :

$$u^{-1} = u^{p-2} \% p \qquad [\ 24\ ]$$

- Théorème de Bezout :

  Pour deux nombres premiers entre eux $u, v \in \mathbb{Z}, \exists x, y \in \mathbb{Z}$ tel que $ux + vy = 1$. En appliquant l'algorithme d'Euclide étendu ([KNUT 81]) pour calculer le $\mathrm{pgcd}(u, v)$ avec $v = p$, on trouve x et y tel que $ux + py = 1$ et

$$u^{-1} = x \% p \qquad [\ 25\ ]$$

Les deux méthodes que nous venons de voir ont toutes les deux une complexité en temps de O(log(p)). En pratique, il est plus efficace de précalculer une table de tous les inverses des nombres de $\mathbb{Z}_p$. Mais avec $\forall i \in\,]0, p[,\ (p-i)^{-1} \equiv (-i)^{-1} \equiv -(i^{-1})$, il est suffisant de conserver $q = (p-1)/2$ inverses, de 1^{-1} à q^{-1}. Si p est un nombre premier fixé (une constante du programme), cette table peut être calculée une fois pour toute (en $O(p\log(p))$) et incluse comme donnée dans le programme. De cette manière, toutes les opérations arithmétiques élémentaires dans $\mathbb{Z}_p$ sont effectuées en temps constant et nécessitent $O(p)$ emplacements mémoire pour stocker la table des inverses.

Le choix de la constante p est un facteur important dans les performances de cette technique pour plusieurs raisons. La valeur de p doit être la plus grande possible pour réduire la fréquence des cas indéterminés et éviter les évaluations exactes. D'un autre coté, p ne doit pas être trop grand pour éviter les dépassements de capacité lors des calculs dans $\mathbb{Z}_p$ et surtout pour limiter à des valeurs raisonnables la taille de la table des inverses.

Un choix raisonnable, sur une machine 32 bits est de prendre p comme étant le plus grand nombre premier inférieur à 2^{16} soit 65521, la table des inverses ne prendra alors que 64 Kilo-octets de mémoire.

4.10.2 Couple de clefs

Le principal inconvénient de la méthode précédente est que la place nécessaire pour stocker les inverses de nombres de $\mathbb{Z}_p$ est proportionnelle à p. L'idée est de remplacer la clef de hashage par des informations plus faciles à manipuler.

Soit z un nombre paresseux quelconque, on définit un couple $(\eta, \delta) \in \mathbb{Z}_p^2$ de la manière suivante :

$$\eta \times \delta^{-1} = \Psi(z) \text{ avec } \delta \neq 0 \qquad [\,26\,]$$

Tous les couples de la forme $(k \times \eta, k \times \delta), k \neq 0$ représentent la même clef $\Psi(z)\%p$, et les couples $(k, 0), k \neq 0$ représentent la clef infinie Ω. Le couple $(0, 0)$ représente les cas indéterminés.

Une manière naturelle de définir le couple associé à un nombre rationnel irréductible $\frac{a}{b}$ est de choisir le couple $(a\%p, b\%p)$. On peut remarquer que cette définition peut aussi s'appliquer à une fraction non réduite si $\mathrm{pgcd}(a, b)$ n'est pas un multiple de p.

Si les couples associés aux nombres paresseux z et z' sont (η, δ) et (η', δ') , nous pouvons définir ceux associés aux résultats des opérations élémentaires (+, *, opposé et inverse) de la manière suivante :

$$\begin{aligned}
(\eta, \delta) + (\eta', \delta') &= ((\eta \times \delta' + \eta' \times \delta)\%p,\ (\delta \times \delta')\%p) \\
(\eta, \delta) \times (\eta', \delta') &= ((\eta \times \eta')\%p,\ (\delta \times \delta')\%p) \\
\mathrm{opp}(\eta, \delta) &= (-\eta\%p, \delta) \equiv (p - \eta, \delta) \\
\mathrm{inv}(\eta, \delta) &= (\delta, \eta)
\end{aligned} \qquad [\,27\,]$$

Toutes ces opérations sont effectuées en temps constant, et la division ne requiert plus le calcul explicite de l'inverse, il se traduit simplement par la permutation des éléments du couple. Cette représentation et ces opérations sur les clefs sont strictement équivalentes à celles de la méthode décrite au paragraphe précédent. Si nous supposons que :

$$\Psi(z + z') = ((\eta \times \delta' + \eta' \times \delta)\%p) \times ((\delta \times \delta')\%p)^{-1}$$

alors

$$\begin{aligned}
\Psi(z + z') &= (\eta \times \delta^{-1} + \eta' \times \delta'^{-1})\%p \\
\Psi(z + z') &= ((\Psi(z) + \Psi(z'))\%p)
\end{aligned}$$

Malgré tout, les cas indéterminés n'ont pas disparu et apparaissent dans les trois cas suivants :

$$\begin{aligned}(\eta, 0) + (\eta', 0) &= (0, 0)\\ (\eta, 0) \times (0, \delta') &= (0, 0)\\ (0, \eta') \times (\delta, 0) &= (0, 0)\end{aligned}$$

La méthode pour lever l'indétermination reste la même que précédemment et consiste à calculer le nombre rationnel sous-jacent.

Dans le cas de la comparaison de deux clefs, il n'est pas nécessaire de connaître leurs valeurs explicitement et on peut éviter de calculer les inverses :

$$\Psi(z) = \Psi(z') \Leftrightarrow \eta \times \delta' = \eta' \times \delta$$

Cette deuxième solution est plus simple que la précédente. En particulier il n'y a pas de traitements spéciaux pour les clef infinies Ω, et le calcul de l'inverse d'un nombre de $\mathbb{Z}_p$ n'est nécessaire que lorsque l'on veut la valeur explicite de la clef. La librairie paresseuse ne la calculera jamais pour ses besoins propres.

Dans le cas général, la clef correspondant au couple $(\eta, \delta), \delta \neq 0$ est $\Psi(z) = (\eta \times \delta^{-1}) \% p$ et le calcul de δ^{-1} n'est nécessaire que lorsque l'on veut disposer explicitement de la clef. Comme des requêtes de ce type ne sont pas très fréquentes, il est possible d'utiliser un des deux algorithmes que l'on a vus au § 4.10 . Bien entendu, à chaque fois que l'on effectue le calcul de δ^{-1}, on en profite pour remplacer le couple (η, δ) par $(\eta \times \delta^{-1}, 1)$ qui est équivalent et qui permettra de répondre immédiatement à une nouvelle requête de $\Psi(z)$. Le seul cas particulier à traiter est celui d'un calcul de clef avec un couple de la forme $(\eta, 0), \eta \neq 0$ où le résultat est Ω.

Le fait de pouvoir se passer de la table des inverses dans $\mathbb{Z}_p$ permet de choisir une valeur de p beaucoup plus grande (donc une probabilité de cas indéterminés plus faible). Sur une machine 32 bits, on peut choisir p comme étant le nombre de *Mersenne* :

$$M = 2^{31} - 1 = 2147483647$$

Dans ce cas, les dépassements de capacité lors des additions et des multiplications peuvent être traitées par les techniques standards décrites dans [KNUT 81] pp 272. De plus le choix de M rend la probabilité[1] $(\frac{1}{M})$ de voir apparaître des cas indéterminés négligeable.

Les clefs de hashage, si elles fournissent des outils efficaces pour le programmeur peuvent aussi participer à la paresse de l'arithmétique. Comme nous venons de le voir, il est nécessaire de conserver une clef pour chaque nombre paresseux pour, soit

1. Cette probabilité est un constat empirique, théoriquement on pourrait s'attendre à une probabilité de $1/p^2$ si l'on ne tient pas compte de l'élément absorbant $(0, 0)$.

la fournir à l'utilisateur, soit calculer la clef du résultat d'une opération. La propriété des clefs ($\Psi(a) \neq \Psi(b) \Rightarrow a \neq b$) peut être mise à contribution pour déterminer efficacement que deux nombres sont différents si les encadrements de ces nombres se recouvrent.

4.11 Recalage des valeurs initiales

Il existe plusieurs possibilités pour stocker dans un programme les valeurs initiales d'un problème. Ces dernières peuvent être conservées sous diverses formes : des entiers, des nombres flottants, des chaînes de caractères, des rationnels, etc. Dans tous les cas, ces valeurs doivent être prises comme étant exactes et de toute façon ce sont les seules dont nous disposons.

L'arithmétique paresseuse effectuant ses opérations exactes en rationnels, on doit être capable de convertir les données numériques initiales sous cette forme. Plusieurs solutions sont envisageables avec un encombrement ou une précision variée.

Si les nombres initiaux ont une représentation flottante, il est possible de transcrire la valeur en un nombre rationnel où le numérateur et le dénominateur sont choisis de manière à représenter de manière exacte le nombre flottant :

$$f = m \times 2^e = \begin{cases} \text{si } e \geq p & \dfrac{m \times 2^e}{1} \\ \text{si } e < p & \dfrac{m \times 2^p}{2^{p-e}} \end{cases} \quad \text{où } p \text{ est la taille de la mantisse du nombre.}$$

ce qui fournira par exemple pour le nombre 6,25 le rationnel $25/4$. Si cette solution a l'avantage d'être parfaitement exacte, elle fabrique des rationnels inutilement encombrants qui ralentiront beaucoup les calculs exacts nécessaires.

Une autre solution, elle aussi très simple, consiste à recaler chaque nombre sur une grille régulière dont on choisit le pas (précision). Dans ce cas le nombre rationnel créé sera de la forme :

$$(f = m \times 2^e) \to r = \frac{\mathrm{round}((m \times 2^e) \times G)}{G}$$

Cette technique a l'avantage de fournir des nombres rationnels de taille bornée et dont la précision peut être choisie par avance (valeur de G).

La dernière solution, de loin la plus intéressante, est l'utilisation du développement en fraction continue (DFC) des valeurs initiales qui peut se faire sous la forme :

$$x = a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{a_3 + \dots}}}$$

où les a_i sont les termes d'une suite d'entiers, finie si x est un nombre rationnel [HARD 60] (ce qui est le cas de tous les nombres flottants).

Si l'on note $\lfloor x \rfloor$ la partie entière d'un réel x, les termes successifs de la suite $(a_n)_n$ sont donnés par la relation de récurrence :

$$x_0 = x \qquad \dots \qquad a_n = \lfloor x_n \rfloor \qquad x_{n+1} = \frac{1}{x_n - a_n}$$

et par les deux suites entières $(p_n)_n$ et $(q_n)_n$:

$$\begin{cases} p_0 = a_0 \\ p_1 = a_0 \times a_1 + 1 \\ \qquad \dots \\ p_n = a_n \times p_{n-1} + p_{n-2} \quad (n \ge 2) \end{cases} \quad \text{et} \quad \begin{cases} q_0 = 1 \\ q_1 = a_1 \\ \qquad \dots \\ q_n = a_n \times q_{n-1} + q_{n-2} \quad (n \ge 2) \end{cases}$$

Ces valeurs nous permettent de construire une suite $(r_n)_n$ de rationnels, convergeant vers x et dont les termes successifs $r_n = p_n / q_n$ constituent des approximations rationnelles irréductibles, de précision et d'encombrement croissants. On démontre que la suite $(|x - r_n|)_n$ est décroissante et qu'elle vérifie :

$$\frac{1}{q_n \times q_{n+2}} < |x - r_n| < \frac{1}{q_n^2} \qquad \forall n \ge 0$$

D'après la théorie des fractions continues, ces approximations sont les meilleures possibles, au sens de la taille du numérateur et du dénominateur, pour une précision donnée.

4.12 Comparaison paresseuse

Le processus de comparaison de deux nombres paresseux peut se traduire maintenant de la manière suivante :

- Les nombres sont-ils les mêmes ?
- Les encadrements se recouvrent-t-ils ?

- Les clefs sont-elles identiques ?
- Les archétypes sont-ils les mêmes (si l'on utilise l'Union-Find standard)?
- Les arbres sont-ils isomorphes ?
- Les nombres évalués sont-ils égaux ?

Tous ces tests participent à la paresse et s'enchaînent du plus simple et rapide (comparaison des adresses mémoires) au plus long et compliqué (évaluation et comparaison exacte de deux rationnels).

Implantation 5

Après la description théorique de la solution à un problème, il faut passer à sa réalisation et montrer que les idées exposées ont un intérêt pratique. La mise en œuvre informatique n'est pas toujours ce qui est le plus facile. Le premier choix à faire est celui du langage, qui doit permettre un développement souple, présenter une bonne adéquation au problème mais aussi de bonnes performances et des chances d'être utilisable de manière industrielle.

Notre choix s'est porté vers C++ qui est un langage orienté objet, extension du langage C. Il est disponible sur une large gamme de matériels, soit directement chez les constructeurs, soit dans le domaine public. De plus, il semble devenir le successeur de C et on trouve de plus en plus de développements dans ce langage. Certaines facilités syntaxiques, telles que la redéfinition des opérateurs, sont très appréciables dans le cas d'une arithmétique : cela simplifie l'écriture et rend plus lisibles les programmes.

5.1 C++, langage objet efficace

C++ a été développé sur les bases du langage C [KERN 78] par Bjarne Stroustrup [STRO 87] qui s'est beaucoup inspiré de deux anciens langages : Simula67 et Algol68. B. Stroustrup était à la recherche d'un langage efficace pour le munir des concepts de Simula dont il avait besoin à des fins de simulation. Son choix s'était alors porté sur C.

C++ n'a rien de révolutionnaire en soi, il introduit la vérification de type et les facilités de la programmation objet. La conception de C++ a été guidée pour répondre à quatre objectifs principaux :

- Améliorer le langage C le plus possible tout en restant compatible avec lui.
- Obtenir un support pour la création de structures de données formelles.
- Ouvrir le langage à la programmation par objets.
- Conserver un compilateur efficace, produisant du code compact et rapide.

C++ se présente aussi comme une extension objet du langage C. Le langage met en œuvre les concepts de classe, les méthodes appliquées à ces classes et l'héritage (multiple depuis sa version 2). Ces concepts conduisent à une véritable programmation par objets. Malgré tout C++ reste efficace. Cette efficacité est préservée par le fait que le langage n'implémente pas de méta-classe et que tout n'est pas vraiment objet : les types de base du langage (entiers, nombres flottants, caractères…) ne sont pas des objets. Mais c'est à ce prix que le langage va rester utilisable. Le typage des données permet de déterminer statiquement à la compilation les méthodes applicables aux objets et c'est seulement si l'utilisateur le spécifie explicitement que ce choix sera fait dynamiquement lors de l'exécution. Ce principe de mise en œuvre permet à la fois le recours à la puissance d'expression de la programmation par objets tout en conservant l'efficacité de C.

Le concept de classe permet d'associer données et méthodes de manipulation des objets dans la définition même des objets. L'encapsulation, quant à elle, laisse libre choix au concepteur de la classe pour décrire la façon dont on doit utiliser ses objets en spécifiant des parties privées et des parties publiques[1] de l'interface. La partie privée de la classe contient en général les données dont on veut interdire l'accès sans contrôle et des méthodes de gestion internes à la classe. La partie publique est par contre une base contractuelle entre le concepteur et l'utilisateur, c'est elle qui permet de manipuler les instances de la classe. De cette manière, tout utilisateur d'une classe est assuré, en utilisant l'interface publique de la classe, d'être indépendant de l'implantation de cette classe. C'est un grand pas vers le génie logiciel.

Si C est le langage C,
l'acronyme C++ évoque malicieusement l'amélioration qu'il apporte au langage C.

5.2 Une bibliothèque transparente

L'implantation de l'arithmétique paresseuse est faite sous la forme d'une bibliothèque autonome qui fournit à l'utilisateur final les quatre opérations élémentaires (+, -, *, /), les comparaisons sous la forme des opérateurs standards de C, ainsi que diverses routines d'entrée-sorties (conversion d'un entier ou d'un flottant vers un rationnel, conversion d'un nombre paresseux vers un flottant…).

La bibliothèque prend à sa charge la gestion des variables temporaires et le partage des données, de manière à ce que vue de l'extérieur, elle se comporte comme une arithmétique du langage (il n'est pas plus compliqué d'utiliser des nombres paresseux

1. Depuis la version 2 de C++, il existe des zones protégées qui sont publiques pour des classes héritières, mais privées pour les utilisateurs.

que des entiers ou des flottants). Tout ceci est rendu facilement possible par les caractéristiques de C++ qui met à la disposition du programmeur :

- La surcharge des opérateurs définis dans le langage.
- Des mécanismes de construction et de destruction automatiques des objets.

Ce dernier point permet une gestion plus simple de la mémoire, ce qui est très appréciable lors de la manipulation des variables temporaires créées soit par l'utilisateur soit par des fonctions.

Cette indépendance vis-à-vis du programme permet de substituer l'arithmétique paresseuse à toute arithmétique interne au langage. Les caractéristiques de C++ nous ont permis de tester diverses arithmétiques sur un même programme (en l'occurrence une implantation de l'algorithme de Bentley-Ottmann) en ne changeant que le type des nombres à utiliser :

- Arithmétique flottante.
- Arithmétique rationnelle "pure".
- Arithmétique rationnelle paresseuse.

5.3 Classes et héritage

Comme nous l'avons vu depuis le début de ce chapitre, l'arithmétique paresseuse utilise conjointement plusieurs arithmétiques pour tenter de limiter les calculs. Ces arithmétiques sont implantées sous forme de classes indépendantes. Ces classes redéfinissent les opérations élémentaires, les comparaisons ainsi qu'un certain nombre de méthodes de gestions particulières à chaque classe.

```
#ifdef LAZY
#include <LazyNumber.h>
typedef LazyNumber Nombre;
#endif

#ifdef FLOTTANT
typedef double Nombre;
#endif

Nombre Y(Nombre x,Nombre a,Nombre b)
{
      return a*x + b;
}
```

figure 21
Arithmétique bien intégrée au langage

5.3.1 Arithmétique d'entiers non bornés

Les entiers non bornés sont implantés sur le modèle décrit au § 3.1 , sous la forme d'un polynôme de degré quelconque dans une base bien adaptée aux manipulations informatiques, en l'occurrence la base 2^{16} qui permet le stockage des chiffres sur des entiers de deux octets et des calculs intermédiaires sans débordement sur des entiers de quatre octets. Le choix de la structure de données pour l'implantation des entiers longs est dicté par son efficacité et sa simplicité d'utilisation lors des diverses opérations, mais aussi par son comportement en mémoire. La représentation des polynômes par des listes a l'avantage de ne manipuler que des données de taille constante mais est lourde et peu efficace pour les opérations qui font un accès intensif aux divers coefficients. La représentation par tableau est efficace pour les manipulations de coefficients mais par contre a tendance à émietter la mémoire par suite d'allocations et déallocations de tableaux de tailles souvent différentes. Mais ce problème peut être facilement résolu avec une gestion mémoire adaptée (simple à mettre en œuvre avec C++ qui autorise la redéfinition des opérateurs *new* et *delete*). La gestion des variables temporaires anonymes qui sont créées dans des expressions du type $ax + b$ doit être prise en compte au niveau le plus bas. De cette manière l'utilisation est plus transparente et plus efficace. Ce type de problème se résout facilement lorsque l'on utilise des compteurs de références associés aux données. Plutôt que de recopier les valeurs temporaires des polynômes, on les partage et lors de l'appel aux destructeurs, on vérifie qu'elles ne sont plus référencées. L'utilisation des compteurs de référence permet aussi d'obtenir l'opposé d'un entier non borné en temps constant par simple partage des données (création et initialisation d'une cellule BigInt, incrémentation du compteur de référence) et inversion du signe.

La création et la destruction continuelle des polynômes est un des facteurs de ralentissement des opérations sur les grands entiers. Lors d'une opération telle que l'addition de nombres de même signes, la taille du résultat n'est connue qu'à un chiffre près avant l'opération et s'il s'avère que cette opération ne génère pas de retenue, le polynôme créé aura un chiffre de trop. Dans un tel cas, il est préférable de conserver la

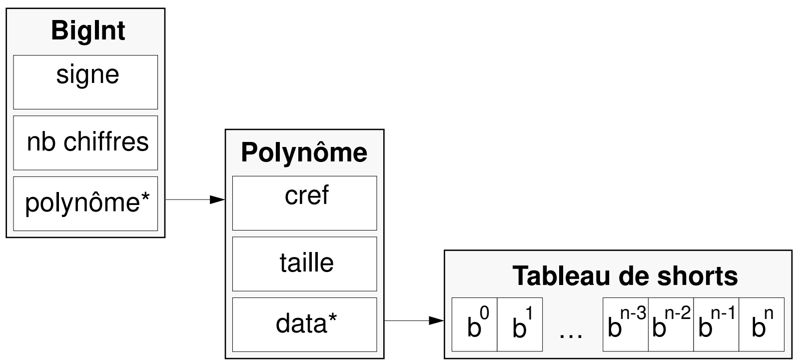


figure 22
Structure d'un entier non borné

taille maximale que pourra prendre le polynôme et le degré effectif de celui-ci plutôt que de le recopier dans un autre tableau qui aurait la taille minimale requise. Le petit surcoût mémoire que l'on engendre sera facilement récupéré lors d'opérations ultérieures et par le partage des informations (les rationnels ont toujours tendance à grossir, et plutôt que de se lancer dans une allocation aveugle, on vérifie que le résultat peut entrer dans le tableau existant).

La classe des entiers non bornés (BigInt) implante les opérations d'addition, soustraction, multiplication, division entière et modulo ainsi que le calcul du pgcd. La division et le modulo sont une vue extérieure d'une division euclidienne qui fournit et conserve les valeurs du quotient et du reste de la dernière opération. De cette manière, après une division, le modulo est immédiatement accessible dans le cas d'opérations du type $q = a/b$; $r = a\%b$ qui sont très souvent utilisées lorsque l'on manipule des arithmétiques entières.

Un modulo particulier est implanté dans le cas où on effectue le modulo d'un entier non borné par un entier machine. Dans ce cas il n'est pas nécessaire d'avoir recours à une division euclidienne, et le calcul peut se faire en utilisant les propriétés des arithmétiques modulaires et des évaluations de polynômes suivant le schéma de Horner :

$$\text{si } N = \sum_{i=0}^{n} x_i b^i \text{ alors } \begin{cases} N\%P = (x_0\%\mathrm{p} + (b\%p) \times (N'\%P))\,\%P \\ N' = \displaystyle\sum_{i=0}^{n-1} x_{i+1} b^i \end{cases}$$

Pour une efficacité maximale, dans le cas du calcul des clefs de hashages, le nombre premier utilisé doit être supérieur à la base des polynômes et tous les calculs doivent s'effectuer sans débordement dans un entier 32 bits. Cette limitation permet de réduire le nombre d'opérations nécessaires pour le calcul. Pour les autres cas, il faut avoir recours à des algorithmes plus complexes que l'on trouve dans [KNUT 81].

5.3.2 Arithmétique rationnelle

Les nombres rationnels sont définis comme une classe contenant deux instances de grands entiers pour représenter le numérateur et le dénominateur. Le fait d'utiliser des instances de la classe BigInt plutôt que des pointeurs simplifie la gestion mémoire, celle-ci étant prise en charge par les BigInts. Par construction, le numérateur sera porteur du signe du rationnel et le dénominateur sera toujours positif. Les opérations arithmétiques élémentaires sur les rationnels sont limitées à la somme, au produit, à l'opposé et à l'inverse. La soustraction et la division sont construites à partir des opérations précédentes. Les comparaisons sont aussi limitées à égal et supérieur. Les rationnels sont toujours irréductibles, ce qui permet d'accélérer les comparaisons (vérification des signes, des degrés,…).

La conversion d'un rationnel vers un nombre flottant peut être effectuée de manière triviale par conversion du numérateur et du dénominateur tant que ces deux valeurs n'excèdent pas la dynamique d'un flottant en double précision. Malheureusement c'est rarement le cas, et la conversion doit se faire de manière plus fiable.

Pour un nombre flottant en double précision si k est le degré du polynôme à convertir en flottant, k ne doit pas dépasser la valeur suivante : $2^{E_{max}} = b^k \Leftrightarrow k = 2^6 = 64$ avec $E_{max} = 2^{10}$ et $b = 2^{16}$ ($k = 2^3 = 8$ en simple précision), ce qui est relativement faible. Pour éviter le dépassement de capacité on ne calcule que les k premiers coefficients de poids fort du numérateur et du dénominateur de manière à ce qu'il n'y ait jamais de débordement arithmétique flottante, on les divise puis on multiplie le résultat par une valeur représentant leur différence de degré (une sorte d'exposant).

$$x = \frac{n_n b^n + n_{n-1} b^{n-1} + \ldots + n_1 b^1 + n_0 b^0}{d_p b^p + d_{p-1} b^{p-1} + \ldots + d_1 b^1 + d_0 b^0} \approx \frac{n_n b^n + n_{n-1} b^{n-1} + \ldots + n_{n-k} b^{n-k}}{d_p b^p + d_{p-1} b^{p-1} + \ldots + d_{p-k} b^{p-k}} \times b^{n-p}$$

Cette méthode permet d'obtenir des résultats suffisament précis mais oblige à l'évaluation des n premiers chiffres de chaque nombre et à l'élévation de la base à la puissance n-p. Si la différence de degré entre les deux nombres est trop importante ($|n-p| > k$), l'opération de conversion renvoie comme résultat la valeur HUGE correctement signée.

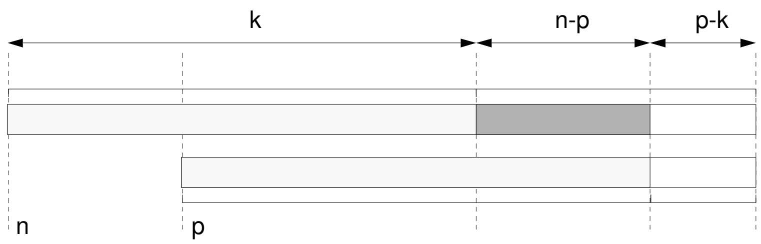


figure 23
Conversion des rationnels : mantisse et exposant.

L'erreur relative que l'on commet sur le résultat de la conversion de la fraction rationnelle $r = \mathrm{num}/\mathrm{den}$ peut s'écrire de la manière suivante :

$$r_{\text{exact}} = \frac{N_0 b^{n-k} + N_1}{D_0 b^{p-k} + D_1} \qquad r_{\text{flottant}} = \frac{N_0}{D_0} \times b^{n-p}$$

$$\frac{r_{\text{exact}} - r_{\text{flottant}}}{r_{\text{exact}}} = 1 - \frac{N_0 b^{n-p}}{D_0} \times \frac{D_0 b^{p-k} + D_1}{N_0 b^{n-k} + N_1} = \frac{D_0 N_1 - D_1 N_0 b^{n-p}}{D_0 (N_0 b^{n-k} + N_1)} \qquad [\,28\,]$$

et elle est bornée par les valeurs suivantes (cf. Annexe A pour la preuve) :

$$\frac{-1}{b^{k-2}} < \frac{r_{\text{exact}} - r_{\text{flottant}}}{r_{\text{exact}}} < \frac{1}{b^{k-2}} \qquad \text{avec} \qquad \begin{cases} b^{k-1} \le n_0 < b^k \\ b^{k-1} \le d_0 < b^k \\ 0 \le n_1 < b^{n-k} \\ 0 \le d_1 < b^{p-k} \\ 0 \le |n-p| \le k \end{cases} \qquad [29]$$

Ce calcul d'erreur permet de limiter la taille de k. L'erreur relative que l'on a sur les nombres flottants est de 2^{-p} (p est la taille de la mantisse du nombre), dans ces conditions et en utilisant les résultats de [29] il n'est pas nécessaire de calculer plus de $k_{min} = \lceil p/\log_2 b + 2 \rceil$ ($k = 6$ pour les flottants double précision avec $b = 2^{16}$) chiffres du nombre rationnel. Il faut remarquer que dans le cas où la base est une puissance de deux ($b = 2^{16}$), le calcul des premiers chiffres des grands entiers $((x_n b + x_{n-1})\, b + x_{n-2})\, b\ldots$ se traduit par la concaténation des représentations binaires des x_i. Lorsque le nombre de bits juxtaposés dépasse la taille de la mantisse p du nombre flottant, seul l'exposant sera modifié. Dans le cas d'une représentation en flottants double précision ($p = 53$), chaque chiffre des entiers non bornés est stocké sur 16 bits, le calcul sur quatre chiffres suffit à remplir la mantisse du nombre flottant ($4 \times 16 = 64$). Il n'est donc pas utile de calculer plus de chiffres, l'erreur théorique par contre va être plus importante ($b^{-2} = 2^{-32}$) et en pratique, on doit en tenir compte pour les opérations d'encadrement.

5.3.3 Arithmétique d'intervalles

L'arithmétique d'intervalles est implantée de manière naturelle. C'est une classe contenant deux champs flottants double précision *min* et *max* représentant les bornes de l'intervalle (figure 24 p. 74). Les opérations élémentaires (+, -, *, /) ainsi que les comparaisons sont définies. Pour assurer que le résultat d'une opération entre deux intervalles est correct, il faut prendre garde aux arrondis effectués par la machine (voir le § 4.3.2). Les opérations $\nabla(x)$ et $\Delta(x)$ peuvent être implantées de deux manières différentes. La version portable de ces opérations ajoute ou soustrait une valeur ε à la mantisse des nombres flottants manipulés[1]. Cette opération est simple à réaliser si l'on utilise les fonctions C *frexp*()[2] et *ldexp*()[3]. Une deuxième solution, moins portable mais plus efficace, manipule directement le type d'arrondi machine (vers plus ou moins l'infini suivant les bornes traitées) à l'aide des routines fournies par le constructeur. Lors de l'encadrement d'un nombre rationnel, la valeur de ε doit être supérieure à l'erreur relative maximum que l'on commet au moment de la conversion en flottant ($\varepsilon \ge 2^{-32}$ pour les flottants double précision).

1. On obtient dans ce cas des approximations par défaut et par excès des valeurs d'arrondi.
2. *frexp*() permet d'extraire la mantisse et l'exposant d'un nombre flottant.
3. *ldexp*() permet de construire un nombre flottant à partir d'une mantisse et d'un exposant.

La division de deux intervalles n'est pas définie dans le cas où l'intervalle au dénominateur contient zéro (bornes de signes opposés) et génère un signal “*floating Exception*”.

Si on dispose d'une arithmétique à la norme IEEE 754, il n'est pas nécessaire de traiter de manière particulière les dépassements de capacité. Par contre, dans le cas contraire, on est obligé de simuler de manière correcte les infinis. La méthode la plus simple consiste à utiliser des flottants simple précision pour les bornes des intervalles et à effectuer les opérations sur des flottants double précision. Dans ce cas, aucun résultat ne provoquera de débordement, et tous ceux dont la valeur dépasse la capacité des flottants simple précision seront considérés comme infinis. Par contre, la précision des intervalles sera plus faible et chaque opération va demander un test, ce qui ralentira l'arithmétique.

On peut ajouter qu'il existe des machines fournissant une arithmétique d'intervalles native et que la norme IEEE 754 est en cours de modification pour un meilleur support de ce type d'arithmétique.

5.3.4 Arithmétique modulaire

L'arithmétique modulaire est implantée sous la forme d'une classe redéfinissant les opérations élémentaires et l'égalité (les autres comparaisons n'ont pas de sens dans ce cas précis) selon la méthode exposée au § 4.10 . La classe stocke la valeur modulo *P* (*lModulo* par la suite, figure 24page 74) des nombres qui est calculée soit à partir de valeurs rationnelles, soit par opérations. La valeur de *P* (celui de $\mathbb{Z}_p$) est une variable statique de classe[1] paramétrable. Le calcul des inverses dans la version actuelle de l'arithmétique paresseuse est tabulé et permet de faire cette opération en temps constant.

Pour avertir les fonctions ou les méthodes qui ont demandé l'exécution d'un calcul modulaire, dans le cas où l'on obtient un résultat indéterminé, une gestion d'erreur particulière permet le déclenchement de l'évaluation exacte.

Le problème de cette implantation est la limitation en taille du nombre premier *P* par l'utilisation de la table des inverses. Il n'est pas envisageable du tout d'avoir une table de taille $2^{31}-1$ pour de simples raisons d'encombrement mémoire (il faudrait 4 Giga-octets de mémoire). Nous utilisons pour l'instant la table des inverses pour $P = 65521$ qui réclame quand même 64 Kilo-octets de mémoire (on peut utiliser des entiers codés sur deux octets) puisque que l'on ne stocke que $p/2$ entrées (cf. § 4.10). Sur les algorithmes que nous avons testés (tout particulièrement l'exemple qui est décrit au § 6.4 des erreurs liées aux indéterminés sur les modulos ne se sont jamais produites.

La représentation des clefs par des couples ne nécessite pas la tabulation des inverses dans $\mathbb{Z}_p$; elle s'impose donc pour de grandes valeurs de *p* ($p > 2^{16}$ sur les machines

1. Une variable statique de classe est une variable qui est partagée par toutes les instances de la classe.

32 bits). La dynamique de l'arithmétique entière des machines devient alors insuffisante ($(p-1)^2 > 2^{32}$) et la programation doit traiter les débordements de capacité. Dans le cas où p est un nombre de Mersenne $2^e - 1$, ces débordements peuvent être traités par les techniques classiques exposées dans [KNUT 81] :

$u+v$ représente la somme ($u \times v$ le produit) de deux entiers machines $u \oplus v$ la somme ($u \otimes v$ le produit) de deux entiers dans $\mathbb{Z}_p$ avec $p = 2^e - 1$:

$$u \oplus v = \begin{cases} u+v & \text{si } u+v < 2^e \\ ((u+v)\,\%2^e) + 1 & \text{si } u+v \geq 2^e \end{cases} \quad \text{(cf. Annexe B)}$$

$$u \otimes v = uv\%2^e \oplus \lfloor uv/2^e \rfloor$$

Ces opérations se traduisent par des décalages pour la division et des masquages dans le cas du modulo et sont donc très efficaces en machine. Le calcul de l'inverse d'une clef est réalisé à l'aide de l'algorithme d'Euclide étendu qui est présenté à l'Annexe B qui a l'avantage de ne pas provoquer de dépassements de capacité.

5.3.5 Arithmétique symbolique

Les expressions symboliques sont stockées sous la forme d'un arbre d'expressions dont les nœuds sont des opérations et les feuilles des nombres rationnels. Les opérations redéfinies sont l'addition. la multiplication, l'opposé et l'inverse d'une expression symbolique ou d'un rationnel. La division et la soustraction sont définies à partir des autres opérations :

$$a - b = a + \mathrm{opp}(b) \qquad a/b = a \times \mathrm{inv}(b)$$

L'arithmétique symbolique redéfinit aussi les opérateurs de comparaison. Pour faciliter la gestion de l'arbre d'expression, tous les nœuds et toutes les feuilles, quel que soit leur type, sont banalisés sous la forme d'une expression. Une hiérarchie de classes permet de les différencier et de définir les méthodes à associer à chaque classe (figure 24). L'approche objet et l'héritage permettent dans ce cas de simplifier les développements et de condenser le code source. Le principe de l'héritage est de regrouper les propriétés communes aux diverses classes à mettre en œuvre (factorisation) et de n'implanter que ce qui est particulier à chaque classe dérivée. L'utilisation des méthodes virtuelles permet d'adapter simplement et de manière transparente les opérations que doit effectuer chaque objet. Le graphe d'héritage présenté figure 25 permet de factoriser au mieux la définition des diverses classes en jeu. A titre d'exemple, la méthode permettant l'évaluation complète d'un arbre d'expressions se traduit par l'évaluation récursive de ces feuilles et le calcul exact que représente le nœud, quel que soit le type de nœud (la méthode permettant l'évaluation paresseuse partielle lui ressemble profondément). Cette manière de faire permet dans tous les cas de s'abstraire du type des éléments de l'arbre et de rajouter de manière très simple des opérations

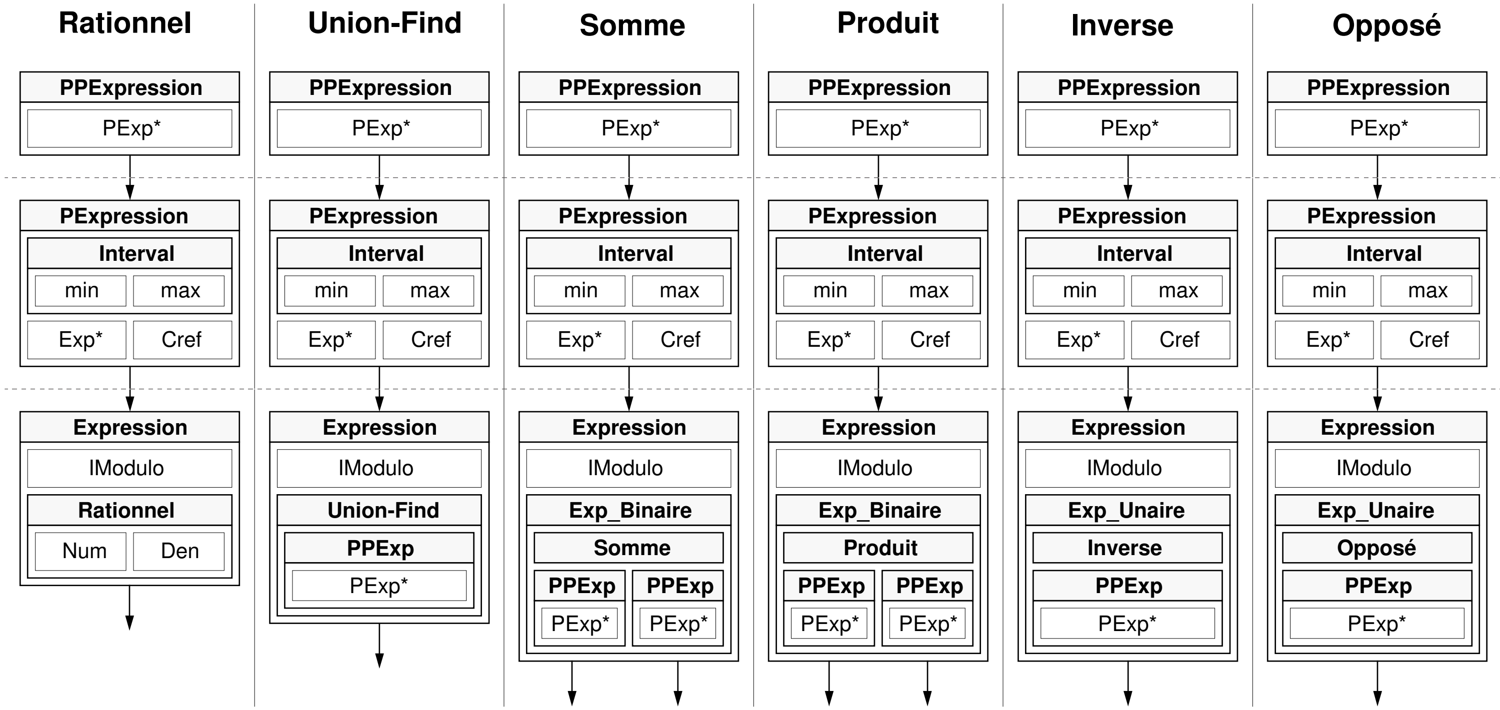


figure 24
Structure de données

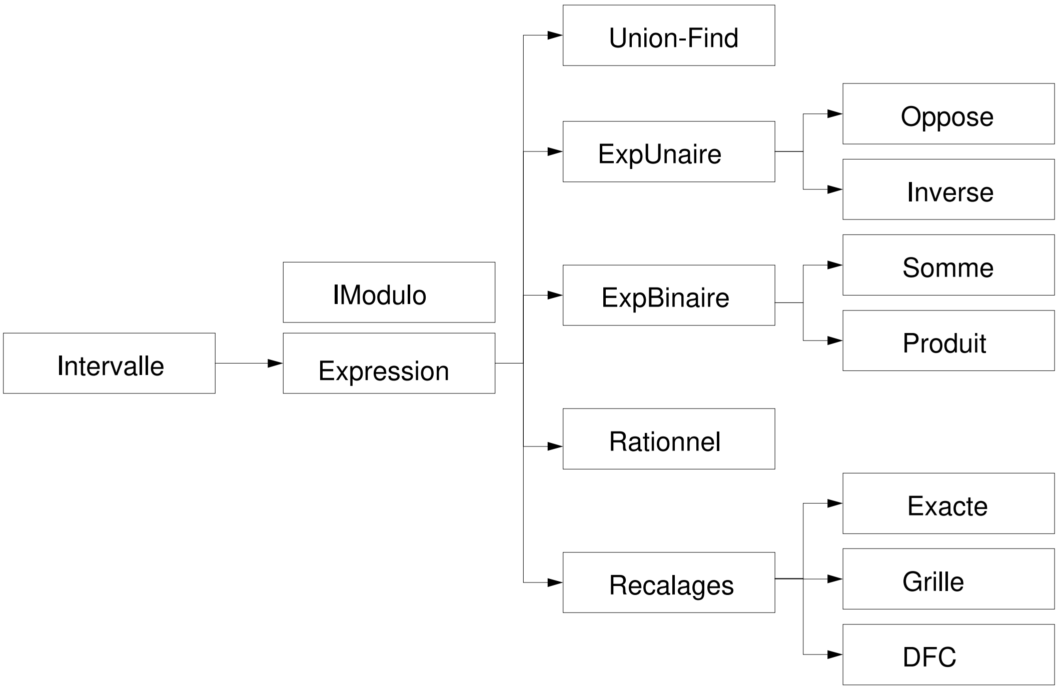


figure 25
Graphe d'héritage des expressions symboliques

supplémentaires (calcul de déterminants, puissances entières), sans qu'il soit nécessaire de modifier le source existant : il suffit de rajouter la définition des méthodes et la classe de la nouvelle opération.

La classe des expressions factorise le plus de choses possibles. C'est elle qui est détentrice du champs *IModulo* (commun à toutes les expressions) ; elle hérite de la classe Intervalle pour simplifier les manipulations (tout nombre est un intervalle ou une

```
Rationnel Rationnel::eval()
{
      return *this;
}

Rationnel Somme::eval()
{
      return a->eval() + b->eval();
}

Rationnel Oppose::eval()
{
      return -a->eval();
}
```

figure 26
Fonctions d'évaluation d'un arbre d'expression

expression symbolique) et définit les méthodes virtuelles[1] de manipulation des expressions : évaluer, évaluer de manière paresseuse, approximer, mettre à jour, imprimer, etc). Certaines méthodes, comme par exemple celle qui fournit à l'utilisateur la clef , n'ont pas besoin d'être virtuelles et ne sont définies que pour les Expressions génériques. Les Expressions Binaires et Unaires définissent le nombre de paramètres de l'expression et stockent ces paramètres. Toutes les autres classes quant à elles implantent effectivement les opérations. La classe *UnionFind* permet de gérer efficacement le partage des données en faisant le lien (tant que c'est nécessaire) entre les diverses expressions qui ont été reconnues identiques lors du déroulement du programme.

L'utilisation d'un DAG, plutôt que d'un arbre, permet de partager des informations au sein des expressions et ainsi de gagner de la place mémoire mais aussi des calculs exacts par une gestion adéquate. Lors d'une évaluation, on peut être amené à remplacer une définition symbolique partagée par sa valeur exacte et tous les nombres paresseux qui partagent cette expression doivent profiter de l'évaluation : il faut être paresseux et ne pas refaire plusieurs fois le même calcul. Pour pouvoir résoudre ce problème il est nécessaire d'utiliser une double indirection (*handle*) qui rend possible la modification d'une expression sans que l'on ait besoin d'avertir les nombres paresseux qui l'utilisent (figure 27 p. 77).

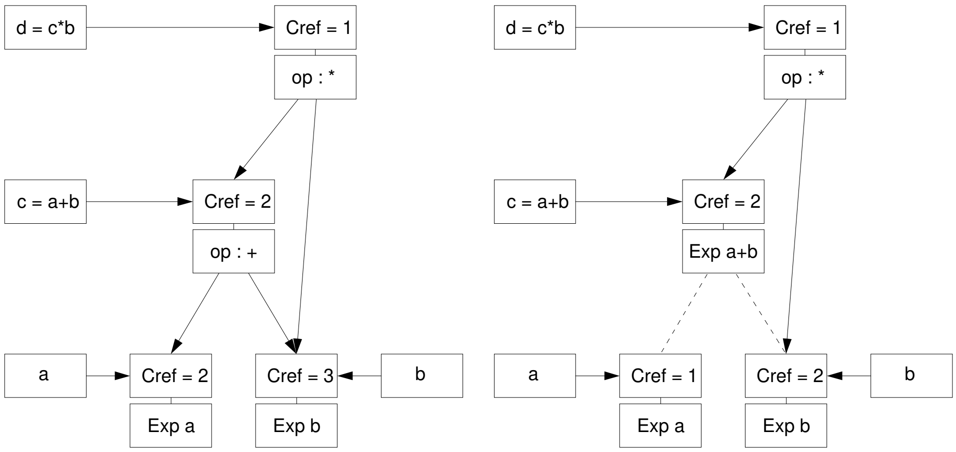
Pour permettre une utilisation simple des nombres, il est impératif de savoir manipuler les variables temporaires. Une arithmétique qui se veut transparente dans son utilisation doit permettre, sans que l'utilisateur ait à se préoccuper de la gestion mémoire, d'écrire des expressions du type :

```
Nombre f(Nombre a,Nombre b,Nombre c, Nombre d)
{
      …
      delta = a*d - b*c;
      …
}
```

Cette façon de faire est bien plus agréable que d'obliger le programmeur à prendre en charge ce type de problème :

```
Nombre f(Nombre a,Nombre b,Nombre c, Nombre d)
{
Nombre tmp1, tmp2;
      …
      tmp1 = a*b; tmp2 = b*c;
      delta = tmp1 - tmp2;
      …
}
```

1. Une méthode virtuelle est une méthode qui peut être redéfinie dans les classes dérivées, et lors de l'exécution, la méthode la plus appropriée sera choisie (sans connaissances a priori sur le type des données).



DAG avant et après une évaluation partielle

figure 27
Gestion mémoire, évaluation et partage

La résolution de ce problème se fait à l'aide d'un compteur de référence associé à chaque nombre. Lors de la création d'un nouveau nombre, le compteur est initialisé à 1. Sa valeur sera incrémentée lors de l'opération d'affectation (a = b) et les deux nombres partageront les mêmes données ; le compteur est aussi incrémenté lors de l'appel du constructeur de recopie[1]. Lors de la destruction d'un nombre, le compteur est décrémenté et, si sa valeur est zéro, on est en train de détruire le dernier représentant du nombre et on libère alors la place qu'il occupe en mémoire. Ce type de situation est très courant : on peut citer les variables temporaires que l'on crée en cours d'opération, les paramètres passés par valeur à des fonctions et les évaluations où des expressions sont remplacées par des valeurs exactes.

Les diverses méthodes de recalage ont été implantées sous la forme de classes associant une valeur (un nombre flottant en double précision) et une méthode de transformation en nombre rationnel. Pour chaque type de recalage, un intervalle encadrant la valeur (une fois recalée) est disponible pour permettre le bon fonctionnement de l'arithmétique. L'idée de ne transformer en nombres rationnels que les nombres qui le nécessitaient était séduisante, elle poussait la paresse encore plus loin. Malheureusement, les premières expériences ne furent pas à la hauteur de nos attentes : chaque fois qu'une expression nécessitait une évaluation, on détruisait de manière irrémédiable nos chances de trouver des arbres isomorphes. Ce problème était surtout dû au manque de précision des encadrements des nombres en instance de recalage, et bien souvent leur transformation en nombre rationnel permettait de réduire suffisament l'en-

1. Le constructeur de recopie est appelé automatiquement par C++ lors des passages et retours de paramètres par valeur et lors de l'initialisation des variables temporaires.

cadrement des expressions concernées pour répondre à la question posée (comparaison ou inverse) ; la grande majorité des nombres étaient transformés en rationnels.

Dans la version actuelle de l'arithmétique, les classes de recalage n'ont pas disparu mais elles servent uniquement à la transformation des nombres initiaux en rationnels. De cette manière il nous est facile de mélanger des valeurs recalées sur une grille avec des valeurs recalées par développement en fractions continues, et surtout de permettre de régler la précision du recalage. Tout ceci étant au choix de l'utilisateur final de l'arithmétique.

L'utilisateur ne manipule que des nombres (les *PPExpressions*). Cette classe est d'un faible encombrement (elle ne contient qu'un pointeur) et définit toutes les méthodes d'accès, les opérateurs élémentaires, les comparaisons et les opérations de conversion de types et d'entrées-sorties. Sa faible taille la rend très adaptée à l'utilisation en tableau ou dans des structures sans alourdir les manipulations. De cette manière, l'arithmétique est rendue transparente, la gestion du partage des données, des classes d'équivalences et des évaluations ne concerne l'utilisateur de la librairie en aucune manière.

5.4 Gestion mémoire

C++ permet la redéfinition des opérateurs *new* et *delete* (allocation et désallocation mémoire) pour chaque définition de classe. De plus, lors de l'héritage, les classes dérivées peuvent hériter de ces méthodes, ce qui permet l'implantation d'une classe de gestion mémoire plus ou moins générale suivant le degré de performance requis.

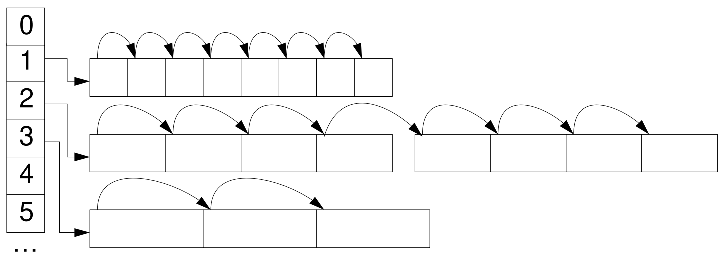


figure 28
Allocation mémoire

Le principe retenu permet de gérer de manière efficace les allocations et désallocations de grains mémoires en fonction de leur taille. On tient à jour un nombre fini de zones mémoires allouées par bloc dont la taille est un multiple de la taille requise pour les données. Lors de sa libération, le grain mémoire est restitué à sa zone. De cette manière, la mémoire n'est pas fragmentée. Pour limiter le nombre des zones mémoires de tailles de grain différents, les grains sont des multiples d'une taille de base (par exemple 1x8 octets, 2x8 octets, …) et chaque allocation se fait dans une zone dont les grains sont de taille supérieure ou égale à la taille demandée. Cette taille de base est paramétrable et doit être choisie en fonction de la taille réelle des données ainsi que des caractéristiques de la machine (certaines machines réclament des alignements mémoires sur des multiples de 8 octets pour être efficaces). Pour chaque taille disponible, on conserve l'adresse du prochain grain à fournir lors d'une allocation. La gestion de la mémoire libre se fait sous la forme d'une liste chaînée où les premiers octets de chaque grain contiennent l'adresse du grain suivant. Lors de la création d'une zone, les grains chaînés sont contigus. Lors d'une allocation, le grain courant est fourni comme emplacement mémoire vierge et le grain mémoire suivant devient le grain courant (pour l'allocation suivante). Lors d'une désallocation, le grain libéré sera replacé en tête de la liste dans sa zone. Si lors d'une allocation, on ne dispose plus de grains libres, une nouvelle zone est créée. La taille de grain zéro est réservée pour les allocations dépassant par leurs tailles la capacité de l'allocateur, ces zones mémoires sont allouées et désallouées de manière conventionnelle par le système.

Ce type d'allocateur mémoire est deux à trois fois plus efficace que l'allocation standard, et est très bien adapté aux allocations-désallocations continuelles de zones mémoire de taille assez faible (ce qui est le cas de l'arithmétique paresseuse). Évidemment, il est nécessaire de pondérer cet avantage par une taille accrue des processus : si les zones sont trop petites, on a recours trop souvent à l'allocateur système et les performances s'écroulent.

Résultats 6

L'arithmétique paresseuse a été utilisée avec succès dans trois programmes (deux géométriques et un numérique). Les nombres paresseux se sont montrés simples à utiliser lors de la programmation (gestion mémoire, opérateurs…). Par contre les performances des algorithmes ne sont pas complètement indépendantes de la manière d'effectuer les opérations. Malgré tout, la version devenue paresseuse des algorithmes s'est toujours montrée plus rapide que la version rationnelle (à notre grand regret, les flottants restent plus efficaces !).

Le choix des programmes que nous avons testés n'est pas détaché de nos préocupations et s'est porté sur des algorithmes dont la sensibilité aux imprécisions numériques est reconnue. Pour chacun des tests, nous avons établi la comparaison des résultats obtenus pour les trois arithmétiques suivantes : flottante, rationnelle et paresseuse.

6.1 Algorithme de Bentley-Ottmann

Comme nous l'avons vu au § 2 l'algorithme de Bentley-Ottmann est un digne représentant des algorithmes sensibles aux imprécisions numériques et tester l'arithmétique paresseuse dans ce contexte allait permettre de valider nos idées. Nous avons fabriqué trois versions de l'implantation de l'algorithme de Bentley-Ottmann, une version en flottants double précision, une version en arithmétique rationnelle, et une version avec l'arithmétique paresseuse. Ces trois versions utilisent toutes le même programme de l'algorithme et seule l'arithmétique varie ; cette prouesse est grandement facilitée par C++ et ne requiert que la compilation du programme avec à chaque fois la définition adéquate des nombres. L'arithmétique flottante que nous avons utilisée n'est pas la plus efficace que l'on puisse imaginer, pour rendre compatibles les trois versions du programme, nous avons dû simuler cette arithmétique sous la forme d'une classe qui ait la même interface que les paresseux et les rationnels. Cette contrainte[1], rend cette

1. Les nombres flottants ne sont pas des objets pour C++, il est impossible d'en redéfinir directement les opérateurs et d'y attacher des méthodes.

arithmétique flottante environ deux fois plus lente que l'arithmétique native de la machine.

L'algorithme a été testé sur un jeu de segments choisis de manière aléatoire dans le cercle unité. Dans ce cas, le nombre d'intersections est en moyenne de l'ordre du carré du nombre des segments, la complexité de l'algorithme ('$O((n+k)\log n)$) a tendance à devenir quadratique ($k \approx n^2$), ce qui explique la forme des courbes obtenues. Pour chacun des jeux de test nous avons aussi fait varier la précision de recalage des coordonnées (la taille des rationnels initiaux) pour en déterminer les implications dans les diverses arithmétiques. Le type de recalage (exact, grille, DFC) a assez peu d'influence sur les temps d'exécution : un gros rationnel issu d'un recalage sur une grille ou issu des DFC reste un gros rationnel.

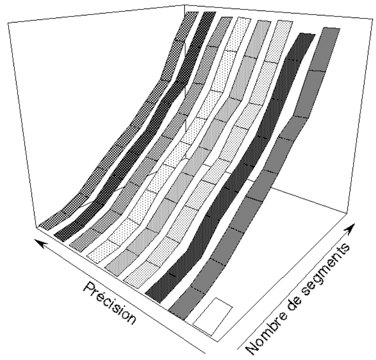
La version flottante de l'algorithme a été dans tous les cas (qui marchaient) plus rapide que les deux autres versions. Le graphe des résultats montre clairement l'influence quadratique des intersections sur les temps d'exécution. La précision du recalage n'a pas d'influence, si ce n'est sur le nombre de cas qui peuvent être résolus : avec un recalage des données initiales à 10^{-1} près, les cas de segments confondus ou partiellement confondus sont courants et les imprécisions numériques entraînent l'arrêt brutal de l'algorithme.

En pratique on se trouve confronté à deux grandes familles de données initiales : des ensembles de segments choisis au hasard (pour des tests) et des données fournies par l'utilisateur (qui correspondent à un problème concret). Dans le premier cas l'expérience montre que ces problèmes sont bien résolus (la probabilité de fabriquer un cas particulier de manière aléatoire est faible) alors qu'en général le traitement de données correspondants à un problème concret arrive rarement à son terme. Cette seconde famille de donnée contient très souvent de nombreux cas particuliers (segments confondus ou colinéaires, points de contacts) qui sont le reflet de la cohérence du problème.

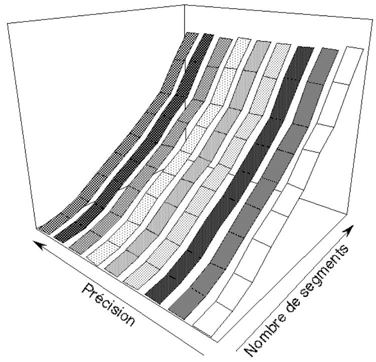
L'arithmétique paresseuse se comporte bien, elle est de 5 à 10 fois plus lente que l'arithmétique flottante (10 à 20 fois plus lente que l'arithmétique flottante native), fonctionne dans tous les cas et n'est pas influencée par la précision du recalage (taille des rationnels initiaux).

L'arithmétique rationnelle, quant à elle, est lente, mais c'était prévisible. Le fait important à remarquer est l'influence de la précision du recalage. Plus les rationnels initiaux sont gros, plus les calculs sont lents, mais en plus ce ralentissement a une composante exponentielle ; dans les mauvais cas, à chaque opération (addition ou multiplication), la taille des rationnels double (cf. § 3.3). Les facteurs de 300 à 500 (600 à 1000) qui apparaissent entre les temps d'exécution en flottant et en rationnels sur les tableaux de résultats sont tout à fait comparables à ceux obtenus par Karazick [KARA 89] lors de leurs premières implantations rationnelles de l'algorithme de [GUIB 85].

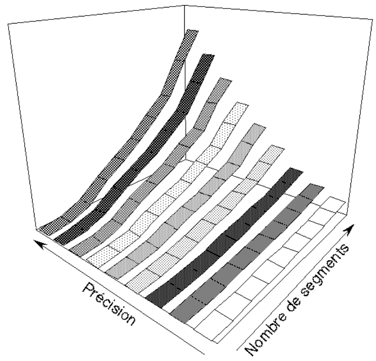
Flottants	10^{-1}	10^{-2}	10^{-3}	10^{-4}	10^{-5}	10^{-6}	10^{-7}	10^{-8}	10^{-9}
10	2	1	2	2	2	2	2	2	3
20	6	5	5	5	5	5	6	5	6
30		12	10	12	11	11	11	12	12
40		23	24	24	23	24	24	24	25
50		34	34	34	34	35	35	35	36
60		44	44	42	45	45	45	45	45
70		58	55	58	59	57	57	61	59
80		67	74	69	69	71	74	72	71
90		91	95	92	93	95	96	95	97
100		113	109	115	114	115	114	117	116



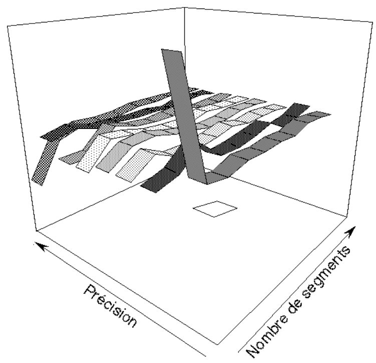
Lazy	10^{-1}	10^{-2}	10^{-3}	10^{-4}	10^{-5}	10^{-6}	10^{-7}	10^{-8}	10^{-9}
10	14	14	14	14	14	13	13	15	12
20	40	37	37	40	40	40	39	39	40
30	92	85	85	87	89	86	87	88	86
40	184	172	173	173	178	178	178	176	176
50	260	251	247	254	254	252	256	253	255
60	338	329	327	325	326	326	329	327	330
70	462	426	428	430	432	431	431	434	430
80	547	520	526	520	522	525	526	529	521
90	718	697	698	689	702	700	693	704	700
100	865	845	843	844	843	847	839	846	843



Rationnels	10^{-1}	10^{-2}	10^{-3}	10^{-4}	10^{-5}	10^{-6}	10^{-7}	10^{-8}	10^{-9}
10	73	78	121	196	239	305	377	466	527
20	200	248	374	605	797	1001	1250	1573	1739
30	438	655	927	1567	2091	2646	3370	4135	4717
40	933	1436	2155	3566	4735	6000	7897	9408	10979
50	1348	2133	3194	5196	7093	8965	11578	13994	16379
60	1782	2722	4198	6722	9292	11705	15168	18341	21397
70	2430	3668	5722	9082	12267	15856	20609	24794	28982
80	2920	4470	6914	11084	15090	19503	24993	30059	35525
90	3917	6040	9463	15017	20859	26974	33798	41113	48901
100	4760	7333	11701	18578	25677	33172	41843	51003	60277



Lazy/Float	10^{-1}	10^{-2}	10^{-3}	10^{-4}	10^{-5}	10^{-6}	10^{-7}	10^{-8}	10^{-9}
10	7,00	14,00	7,00	7,00	7,00	6,50	6,50	7,50	4,00
20	6,67	7,40	7,40	8,00	8,00	8,00	6,50	7,80	6,67
30		7,08	8,50	7,25	8,09	7,82	7,91	7,33	7,17
40		7,48	7,21	7,21	7,74	7,42	7,42	7,33	7,04
50		7,38	7,26	7,47	7,47	7,20	7,31	7,23	7,08
60		7,48	7,43	7,74	7,24	7,24	7,31	7,27	7,33
70		7,34	7,78	7,41	7,32	7,56	7,56	7,11	7,29
80		7,76	7,11	7,54	7,57	7,39	7,11	7,35	7,34
90		7,66	7,35	7,49	7,55	7,37	7,22	7,41	7,22
100		7,48	7,73	7,34	7,39	7,37	7,36	7,23	7,27



Rat./Lazy	10^{-1}	10^{-2}	10^{-3}	10^{-4}	10^{-5}	10^{-6}	10^{-7}	10^{-8}	10^{-9}
10	5,21	5,57	8,64	14,00	17,07	23,46	29,00	31,07	43,92
20	5,00	6,70	10,11	15,13	19,93	25,03	32,05	40,33	43,48
30	4,76	7,71	10,91	18,01	23,49	30,77	38,74	46,99	54,85
40	5,07	8,35	12,46	20,61	26,60	33,71	44,37	53,45	62,38
50	5,18	8,50	12,93	20,46	27,93	35,58	45,23	55,31	64,23
60	5,27	8,27	12,84	20,68	28,50	35,90	46,10	56,09	64,84
70	5,26	8,61	13,37	21,12	28,40	36,79	47,82	57,13	67,40
80	5,34	8,60	13,14	21,32	28,91	37,15	47,52	56,82	68,19
90	5,46	8,67	13,56	21,80	29,71	38,53	48,77	58,40	69,86
100	5,50	8,68	13,88	22,01	30,46	39,16	49,87	60,29	71,50

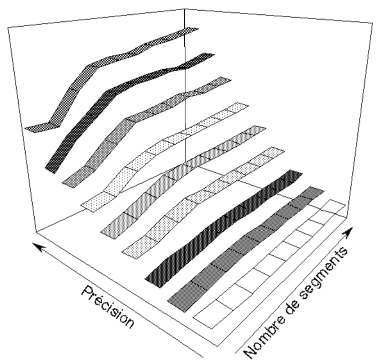


figure 29
Algorithme de Bentley-Ottman : résultats

6.2 Algorithme de Gauss

Un problème fréquent en géométrie est de déterminer le signe d'un déterminant (cf. § 3.4). L'utilisation de l'arithmétique flottante et l'imprécision qui en découle lors des calculs risquent de fournir des résultats erronés qui peuvent mettre en échec de nombreux algorithmes. A titre d'exemple, un simple déterminant 2×2 peut fournir un résultat différent selon la façon dont il est programmé :

Soit $\det\begin{bmatrix} 72450100 & 732698713 \\ 212345677 & 2147483637 \end{bmatrix} = -1$ en arithmétique exacte.

```
{
double a = 72450100.,  b = 732698713.;
double c = 212345677., d = 2147483637.;
double ad, bc, det;

     det = a*d - b*c;          // cas 1
     cout << det << "\n";      // resultat -1.

     ad = a*d;                 // cas 2
     cout << ad << "\n";       // résultat   155585404249013690.
                               // au lieu de 155585404249013700
     bc = b*c;
     cout << ad << "\n";       // résultat   155585404249013690.
                               // au lieu de 155585404249013701
     det = ad - bc;
     cout << det << "\n";      // résultat 0.
}
```

figure 30
Imprécisions numériques sur un calcul de déterminant

L'utilisation des registres du coprocesseur flottant (figure 30, cas 1) permet d'obtenir un résultat correct (104 bits sur les coprocesseurs flottants de la famille 68000 de Motorola) alors que le stockage des valeurs intermédiaires dans des flottants en double précision (figure 30, cas 2) entraîne une perte de précision (utilisation de 64 bits) et un résultat erroné. Il est difficile de se fier à l'arithmétique flottante dans un tel cas.

L'algorithme de Gauss permet un calcul efficace du déterminant d'une matrice carrée $n \times n$ avec une complexité de $O(n^3)$ opérations au lieu de $O(nn!)$ pour les algorithmes dérivés de l'utilisation de la règle de Sarrus. L'utilisation des nombres paresseux se fait de la manière la plus simple : chaque valeur de la matrice dont on cherche le déterminant est un nombre paresseux et le calcul du déterminant se fait avec l'algorithme classique.

Le résultat du calcul de déterminant va dépendre du conditionnement de la matrice. Soit la matrice est bien conditionnée et l'utilisation de l'arithmétique d'intervalles fournira un encadrement du résultat qui permet d'obtenir son signe sans calcul exact (l'intervalle ne contenant pas zéro), soit la matrice est mal conditionnée (ou son déterminant est nul) et l'encadrement du résultat contient la valeur zéro. Dans ce dernier cas il est nécessaire d'avoir recours à des calculs exacts pour déterminer le signe du déterminant. Empiriquement, on constate que dans de tels cas il est nécessaire de faire pratiquement tous les calculs de façon exacte pour obtenir le résultat.

Les opérations paresseuses s'accompagnant d'un encombrement mémoire proportionnel au nombre d'opérations, un algorithme tel que celui de Gauss voit sa complexité en espace mémoire passer de $O(n^2)$ (taille de la matrice) à $O(n^3)$ (nombre d'opérations nécessaires).

Ces dernières remarques nous permettent de mettre en avant les faits suivants :

- Si le signe du déterminant peut être calculé à partir des seuls encadrements, il n'était pas nécessaire de construire l'expression symbolique le représentant.
- Si le signe du déterminant ne peut pas être calculé à partir des seuls encadrements, il n'est pas nécessaire de construire l'expression symbolique le représentant puisque nous devrons l'évaluer.

Il semble donc judicieux d'envisager un déterminant paresseux, sous la forme d'un nouvel opérateur de l'arithmétique paresseuse, qui ferait d'abord les calculs sur les encadrements, puis si nécessaire les opérations exactes pour déterminer de manière fiable le signe (et la valeur). On élimine de cette façon la création inutile des expressions symboliques et on ramène par la même occasion la complexité en place mémoire de l'algorithme de Gauss à $O(n^2)$.

Il faut tout de même rappeler que pour résoudre ce type de problèmes, il existe d'autres solutions. On peut citer par exemple l'utilisation d'une arithmétique modulaire telle que décrite dans [KNUT 81]. Ce type d'arithmétique permet d'effectuer de manière très efficace les additions et les multiplications (en $O(1)$ sur une architecture spécialisée). Cependant les principaux défauts de ce type d'arithmétique sont de ne pas permettre une détection efficace des dépassements de capacité (il est nécessaire d'adapter a priori la représentation des nombres aux calculs à réaliser) et d'obliger à un changement de représentation pour effectuer les comparaisons. Pour déterminer le signe d'un nombre en représentation modulaire, en général on utilise une conversion dans un système de numération mixte ("mixed radix representation") dont le coût est proportionnel à la taille de la représentation modulaire.

Il faut noter que de toute façon, l'utilisation d'une arithmétique modulaire logicielle (sans utilisation de matériel spécialisé) nécessite de faire tous les calculs pour tous les modules alors que l'utilisation des encadrements flottants auraient peut-être suffit.

6.3 Suite convergente

L'exemple que nous présentons dans cette annexe est extrait d'un ouvrage de J.M Muller : *L'arithmétique des ordinateurs* [MULL 89]. Il met en avant un problème différent des incohérences topologiques que nous avons vu dans les paragraphes précédents. Cet exemple montre que le résultat d'un calcul peut être totalement faux alors que l'on observe une "bonne convergence" des calculs sur une machine.

Si l'on considère la suite a_n de nombres définis de la manière suivante :

$$a_{n+1} = 111 - \frac{1130}{a_n} + \frac{3000}{a_n a_{n-1}} \qquad \text{avec} \qquad a_0 = \frac{11}{2}, a_1 = \frac{61}{11}$$

cette suite de nombres converge de manière exacte vers 6. Pourtant sur n'importe quelle machine, on observe une convergence rapide vers 100. La figure 31 présente les premiers termes de cette suite calculée sur avec les flottants "extended" (80 bits) d'un Macintosh. L'utilisation des flottants sur 80 bits permet de "gagner" 3 itérations avant que la suite ne converge vers la valeur 100 par rapport à des calculs sur des flottants 64 bits (*double*).

```
a1  = 5.54545454545454          a16 = 6.65729723987226
a2  = 5.59016393442623          a17 = 16.6000501930843
a3  = 5.63343108504399          a18 = 70.0744205165950
a4  = 5.67464862051015          a19 = 97.4532928295079
a5  = 5.71332905238054          a20 = 99.8440057716986
a6  = 5.74912091970298          a21 = 99.9906658518466
a7  = 5.78181092049156          a22 = 99.9994419054612
a8  = 5.81131423839670          a23 = 99.9999666144729
a9  = 5.83765655072559          a24 = 99.9999980018843
a10 = 5.86095155277548          a25 = 99.9999998803639
a11 = 5.88137773201671          a26 = 99.9999999928344
a12 = 5.89916268056730          a27 = 99.9999999995707
a13 = 5.91467367252954          a28 = 99.9999999999743
a14 = 5.93025551028524          a29 = 99.9999999999985
a15 = 5.98143989459209          a30 = 99.9999999999999
```

figure 31
Convergence de la suite en nombre flottants

Le même calcul effectué à l'aide de l'arithmétique paresseuse montre que dans un premier temps l'arithmétique d'intervalles donne un bon encadrement du résultat, puis que celui-ci s'élargit démesurément et il a des bornes de signes opposés (itération a_{11} figure 32). L'arithmétique doit procéder à une évaluation du résultat de l'itération n ou $n-1$ pour pouvoir calculer la valeur de la suite à l'itération $n+1$: nous nous trouvons

```
a2  = [5.59016,  5.59016]     a17 = [-28.7712, 37.7121]
a3  = [5.63343,  5.63343]     a18 = [4.9484,   6.96642]
a4  = [5.67465,  5.67465]     a19 = [-45.3257, 50.9482]
a5  = [5.71333,  5.71333]     a20 = [5.97458,  5.97458]
a6  = [5.74911,  5.74913]     a21 = [5.97854,  5.97891]
a7  = [5.78145,  5.78218]     a22 = [5.97378,  5.99070]
a8  = [5.79318,  5.82945]     a23 = [5.59708,  6.37374]
a9  = [4.94564,  6.72760]     a24 = [-12.3223, 23.4344]
a10 = [-40.9888, 47.7436]     a25 = [5.95231,  6.02582]
a11 = [-164.194, 128.031]     a26 = [4.29523,  7.66091]
a12 = [5.89915,  5.89915]     a27 = [-87.096,  80.8389]
a13 = [5.91452,  5.91453]     a28 = [5.99385,  5.99409]
a14 = [5.92742,  5.92806]     a29 = [5.98953,  6.00048]
a15 = [5.92394,  5.95411]     a30 = [5.74612,  6.24648]
a16 = [5.24324,  6.65202]     a31 = [-5.6156,  17.2654]
```

figure 32
Convergence de la suite en nombres paresseux

dans un cas de division par un nombre dont l'encadrement contient zéro. La suite converge alors correctement vers 6. La figure 33 présente le résultat rationnel exact de la 20$^{\text{ème}}$ itération et sa valeur "paresseuse", il s'agit de l'arbre d'expressions complet sous une forme "LISPienne" et non du DAG (le partage des informations est difficilement représentable sous forme textuelle).

```
a20 = [5.97458, 5.97458] = 22413787798580981/3751525871703601

a20 = (+ (+ 111/1 (- (* 1130/1 (1/ (+ (+ 111/1 (- (* 1130/1 (1/ (+ (+ 111/1 (- (*
1130/1 (1/ (+ -1396733034168959/17689598897861 136555244373000/1608145354351)))))
(* 3000/1 (1/ (* (+ -1396733034168959/17689598897861 136555244373000/160814535435
1) 17689598897861/2973697798081)))))))) (* 3000/1 (1/ (* (+ (+ 111/1(- (* 1130/1
(1/(+ -1396733034168959/17689598897861 136555244373000/1608145354351)))))(* 3000/
1 (1/ (* (+ -1396733034168959/17689598897861 136555244373000/1608145354351) 17689
598897861/2973697798081)))) (+ -1396733034168959/17689598897861 136555244373000/1
608145354351))))))))) (* 3000/1 (1/ (* (+ (+  111/1 (- (* 1130/1 (1/ (+ (+  111/1
(- (*  1130/1 (1/ (+ -1396733034168959/17689598897861 136555244373000/16081453543
51))))) (* 3000/1 (1/ (* (+ -1396733034168959/17689598897861 136555244373000/1608
145354351) 17689598897861/2973697798081)))))))) (* 3000/1 (1/ (* (+ (+ 111/1 (- (
* 1130/1 (1/ (+ -1396733034168959/17689598897861 136555244373000/1608145354351)))
)) (* 3000/1 (1/ (* (+ -1396733034168959/17689598897861 136555244373000/160814535
4351) 17689598897861/2973697798081)))) (+ -1396733034168959/17689598897861 136555
244373000/1608145354351))))) (+ (+ 111/1 (- (* 1130/1 (1/ (+ -1396733034168959/17
689598897861 136555244373000/ 1608145354351))))) (* 3000/1 (1/ (* (+ -13967330341
68959/17689598897861 136555244373000/1608145354351) 17689598897861/2973697798081)
)))))))
```

figure 33
Résultat paresseux de la 20$^{\text{ème}}$ itération

6.4 Algorithme de Michelucci-Benouamer

Cet exemple traite des résultats obtenus avec un algorithme d'intersection de polyèdres représentés par leur frontière. Cet algorithme que l'on doit à Michelucci et Benouamer apporte une solution robuste et efficace au problème de la résolution des opérations booléennes lors d'une modélisation par arbre de construction (arbre CSG) [PERO 88]. Le lecteur intéressé trouvera une description complète de l'algorithme ainsi que les divers problèmes rencontrés lors de son implantation dans [MICH 87] et [BENO 93.2].

L'algorithme de Michelucci et Benouamer associe deux structures de donnée particulières : la face et le pan.

- Une face est le plus grand sous ensemble de la frontière d'un solide porté par un même plan orienté.
- Le pan est la généralisation tridimensionnelle de la notion de brin utilisé dans les cartes planaires [MICH 87]. Intuitivement un pan est une portion de plan, incidente à une arête donnée et d'un coté bien déterminé de cette arête.

Pour le bon déroulement de l'algorithme, il est nécessaire de définir les contraintes de cohérence suivantes :

- Deux arêtes distinctes ne peuvent se chevaucher, ni se couper en un point autre qu'une extrémité commune.
- Les faces sont maximales et deux faces distinctes ne peuvent s'intersecter que le long d'arêtes explicitement décrites dans la structure de données.
- Une arête doit avoir un nombre pair de pans incidents et il n'existe pas d'arêtes inutiles ayant deux pans incidents appartenant à la même face.

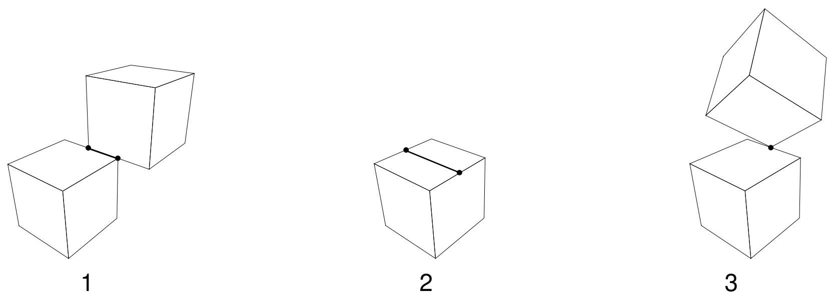


figure 34
Contraintes de cohérence

Tous les calculs nécessaires à la résolution des opérations boléennes sont supposés s'effectuer sans erreur, de sorte qu'aucune incohérence topologique ne puisse avoir lieu du seul fait des imprécisions numériques.

L'algorithme se déroule en 5 grandes étapes :

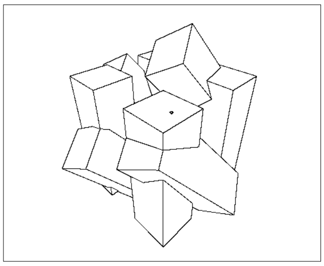
- Prétraitement : recherche des faces susceptibles de s'intersecter.
- Intersection des faces candidates, les intersections rencontrées sont stockées pour un traitement ultérieur.
- Eclatement des frontières des solides le long de leurs intersections.
- Classification des éléments et sélection de ceux appartenant à la frontière du résultat.
- Construction explicite du résultat.

Lors du déroulement de ces diverses étapes, seule la phase d'intersection est susceptible de construire de nouveaux points ou de nouveaux pans. Ces nouvelles informations seront utilisées lors des étapes de classification et de construction et devront être topologiquement cohérentes avec la solution (un point d'intersection doit appartenir à une face …). Par contre, il se peut que certaines de ces informations n'appartiennent pas à la solution et dans ce cas il aurait été inutile de les calculer de manière exacte.

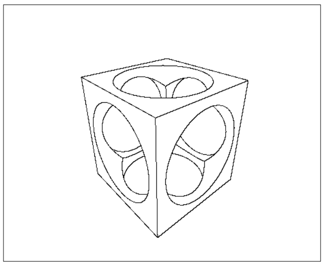
Plusieurs tests ont été effectués afin d'étudier l'influence des divers paramètres sur les performances globales de l'algorithme. Les deux tableaux, ci-dessous, synthétisent quelques résultats qui permettent d'estimer les coûts et les gains comparés des trois versions de l'algorithme (flottant, rationnel et paresseux). L'interprétation de ces résultats n'est pas toujours évidente, du fait de l'interaction de plusieurs paramètres contradictoires.

DFC $\varepsilon = 10^{-6}$	Nombre de facettes	Version rationnelle		Version paresseuse		Rapport R/P	
		Opérations	Temps	Opérations	Temps	Opérations	Temps
Exemple 1	96	72247	67145	629	1515	114	44
Exemple 2	252	549635	708435	1156	4270	475	150
Exemple 3	10300	6401412	1669743	90376	88441	71	20

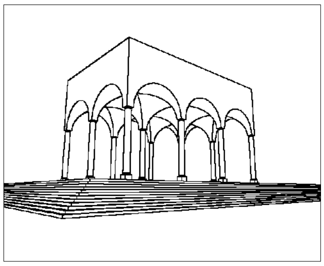
Les résultats du tableau précédent mettent en évidence l'influence d'un certain nombre de paramètres sur les temps d'exécution du programme. Il est évident, à la vue de ces résultats que le coût de l'algorithme est fonction du nombre d'opérations rationnelles qui sont effectuées.



Exemple 1



Exemple 2



Exemple 3

6.4.1 Surcoût lié à l'utilisation des nombres paresseux

Dans le cas ou le programme se déroule convenablement en nombres flottants, la version équivalente qui utilise les nombres paresseux ne devrait pas faire d'opération exacte. Dans ces conditions il est facile d'apprécier le surcoût lié à l'arithmétique paresseuse. Le test qui a été retenu consiste à évaluer l'intersection de *n* cubes centrés à l'origine et orientés de manière quelconque dans l'espace. Dans les tableaux "PARESSEUX" et "RATIONNELS" présentent la taille maximale (Max.), la taille moyennes (Moy,) des grands entiers manipulés par la librairie et le nombre d'opérations rationnelles effectuées (Op.) pour différentes valeurs de recalages (DFC). Les trois tableaux suivants (FLOT., PAR., RAT.) ainsi que les graphiques correspondants présentent les temps d'exécutions exprimées en secondes pour un nombre croissant de cubes (de 2 à 20) et une meilleure précision de recalage (DFC avec une précision de 10^{-3} à 10^{-12}).

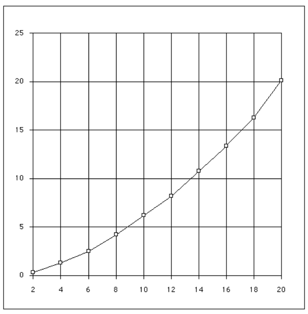
PARESSEUX			
Recalage	Max.	Moy.	Op.
1e-03	3	1,6	0
1e-06	3	1,6	0
1e-09	3	1,7	0
1e-12	7	1,8	114

RATIONNELS			
Recalage	Max.	Moy.	Op.
1e-03	44	9,2	912527
1e-06	86	18,2	912011
1e-09	128	27,4	912011
1e-12	170	37,3	640877

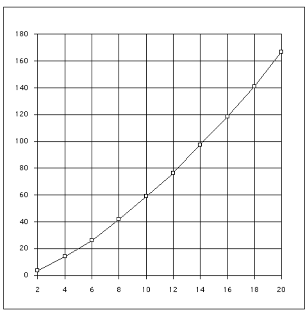
FLOT.	Nombre de cubes									
DFC	2	4	6	8	10	12	14	16	18	20
	0,3	1,3	2,5	4,2	6,2	8,2	10,8	13,4	16,3	20,1

PAR.	Nombre de cubes									
DFC	2	4	6	8	10	12	14	16	18	20
1e-03	3,5	13,8	26,1	42,2	59,4	75,9	96,7	117,8	140,6	166,8
1e-06	3,6	13,8	25,9	41,5	58,8	75,9	96,6	117,6	140,8	166,9
1e-09	3,6	13,9	25,8	41,6	59,1	76,1	96,7	117,8	140,8	166,7
1e-12	3,5	14	26,2	41,8	59,2	76,3	97,3	118,3	140,8	167

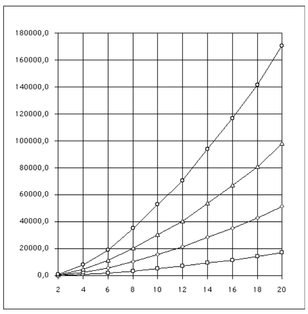
RAT.	Nombre de cubes									
DFC	2	4	6	8	10	12	14	16	18	20
1e-03	89	798	1839	3415	5171	6914	9269	11530	13943	16949
1e-06	234	2383	5609	10378	15692	21153	28254	35018	42566	51352
1e-09	434	4862	11197	20205	30309	40472	53869	66998	80792	98125
1e-12	727	8025	19076	34960	52660	70291	93904	116851	141602	170700



Flottants



Paresseux



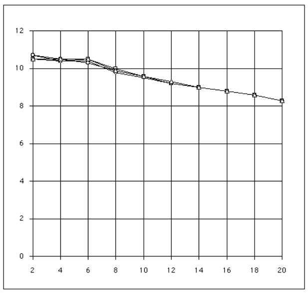
Rationnels

Les trois tableaux suivants présentent les rapports de performances entre les trois arithmétiques. Il est intéressant de constater que le rapport de performances entre la version paresseuse et flottante du programme est indépendant de la valeur de recalage choisie et que le surcoût entraîner par l'utilisation des nombres paresseux décroît vers une valeur asymptotique lorsque le nombre de valeurs initiales grossit. Ce comportement peut s'expliquer par une meilleure utilisation des classes d'équivalences. Le rapport de performances entre flottants et rationnels dominé par le coût des opérations entre grands entiers lorsque leur taille (nombre de chiffres) croît.

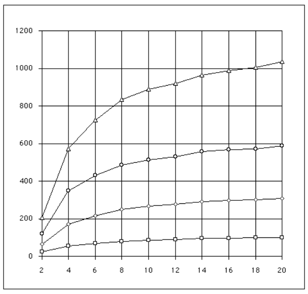
P/F	Nombre de cubes									
DFC	2	4	6	8	10	12	14	16	18	20
1e-03	10,5	10,4	10,5	10	9,6	9,2	9	8,8	8,6	8,3
1e-06	10,7	10,4	10,4	9,8	9,5	9,2	9	8,8	8,6	8,3
1e-09	10,7	10,5	10,3	9,9	9,6	9,2	9	8,8	8,6	8,3
1e-12	10,5	10,5	10,5	9,9	9,6	9,3	9	8,8	8,6	8,3

R/F	Nombre de cubes									
DFC	2	4	6	8	10	12	14	16	18	20
1e-03	266,5	598,5	735,5	810	836,3	839,7	859,6	859,4	855,4	844,7
1e-06	702,7	1787,1	2243,5	2461,2	2537,8	2569,2	2620,1	2610,1	2611,4	2559,1
1e-09	1301,8	3646,2	4478,8	4791,7	4901,8	4915,7	4995,6	4993,7	4956,6	4889,9
1e-12	2181,2	6018,8	7630,5	8291	8516,5	8537,4	8708,2	8709,4	8691,2	8637,9

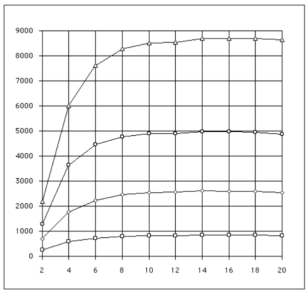
R/P	Nombre de cubes									
DFC	2	4	6	8	10	12	14	16	18	20
1e-03	25,5	57,8	70,4	80,9	87,1	91,1	95,9	97,9	99,2	101,6
1e-06	65,7	172,2	216,3	249,9	266,8	278,6	292,6	297,7	302,4	307,7
1e-09	121,7	348,9	433,4	485,7	513,1	531,7	557,2	568,6	573,9	588,6
1e-12	207,7	573,2	727,2	836,7	889,3	921,9	965,6	987,9	1006,4	1038,2



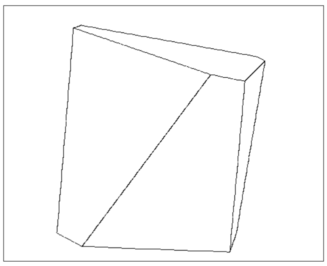
Paresseux/Flottants

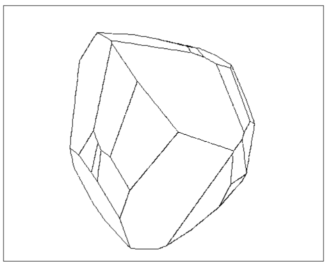


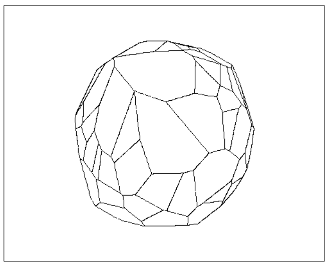
Rationnels/Paresseux



Rationnels/Flottants

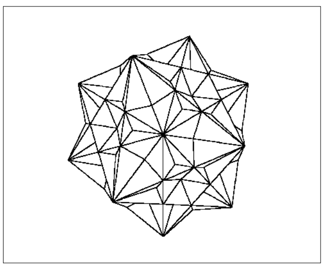

2 cubes

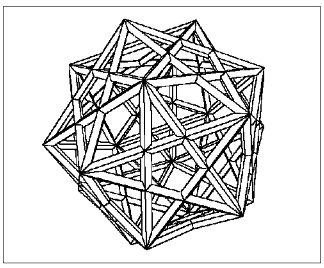

8 cubes

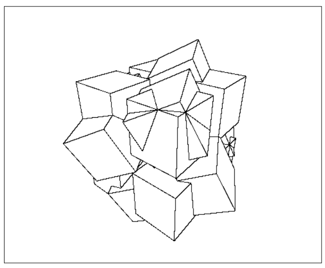

20 cubes

Lors d'une utilisation courante du programme, les bases de données contiennent de plus en plus de cas particuliers qui nécessitent le recours à des évaluations dans le cas de utilisation de l'arithmétique paresseuse ou entraînent une fin anormale du programme dans sa version flottante. Dans ce cas, les calculs rationnels qu'il est nécessaire d'effectuer vont dégrader les performances de l'algorithme. Ce ralentissement est directement fonction du nombre d'opérations exactes à effectuer et reste en général faible. Le tableau ci-dessous présente quelques cas qui n'aboutissent pas convenablement en flottants. La colonne "%op." donne la proportion d'opérations exactes effectuée par la version paresseuse du programme par rapport à la version entièrement rationnelle.

DFC $\varepsilon = 10^{-9}$	Nombre de facettes	Temps (Unités CPU)			Nb de chiffres maximum		Rapport (temps)		%op.
		Flot.	Rat.	Par.	Rat	Par.	P/F	R/P	
Test 1	96	88	97630	1263	76	3	14	77	0
Test 2	192	217	223202	2619	78	3	12	85	0
Test 3	252	399	2173277	9518	98	3	24	228	0
Test 4	600	862	3667275	20966	98	3	24	228	0
Test 5	372		21247	3806	26	13		69	12.79
Test 6	556		2150076	16336	96	44		132	0.15
Test 7	160		242592	1119	72	10		217	1.80







Limites 7

L'arithmétique paresseuse en tant que telle a atteint son principal but qui est de réduire le plus possible les calculs exacts dans les applications qui y ont recours. Des comparaisons entre des applications reconnues sensibles aux imprécisions numériques nous ont permis de voir que lorsque de tels programmes fonctionnaient avec une arithmétique flottante standard, en général, aucun calcul n'était effectué en arithmétique paresseuse[1], la précision des intervalles étaient suffisante.

En revanche, notre souhait de transparence de l'arithmétique vis à vis du programmeur a souvent été pris en défaut. Nous nous sommes longtemps interrogés sur des évaluations exactes qui n'aurait pas dû être déclenchées. Il apparait que les méthodes mises en œuvre pour éliminer les calculs exacts dans l'arithmétique paresseuse ne sont pas toujours sans influence sur la programmation. Une analyse détaillée de ces problèmes dans [BENO 93.2] met en évidence deux grandes classes de programmation : la programmation naïve et la programmation avertie.

Les remarques de M. O. Benouamer [BENO 93.2] s'appuient sur l'expérience du programmeur pour déjouer les pièges des arithmétiques flottantes où les opérations de multiplication et d'addition ne sont ni associatives ni distributives (le lecteur trouvera la description de ces phénomènes dans [GOLD 91] et [MULL 89]). Benouamer présente plusieurs exemples des méfaits de l'imprécision des nombres flottants ainsi que la manière d'y remédier lorsque c'est possible :

Soient D et D' deux droites du plan, d'équations respectives $D: ax + by + c = 0$ et $D': a'x + b'y + c' = 0$ et P leur point d'intersection. Calculées de manière analytique les coordonnées X et Y du point d'intersection sont données par la formule exacte :

$$X = \frac{bc' - b'c}{ab' - a'b} \text{ et } Y = \frac{ac' - a'c}{ab' - a'b}$$

1. Il se peut, que par chance, certaine fois l'algorithme arrive correctement à son terme et que l'on ait effectué quelques calculs exacts.

Malheureusement le calcul en nombres flottants est toujours entaché d'erreur. Ultérieurement, l'algorithme peut avoir besoin de déterminer la position de certains points par rapport à certaines droites, par un calcul de puissance. En particulier les puissances calculée $aX + bY + c$ et $a'X + b'Y + c$ du point P par rapport aux droites D et D' ont fort peut de chance d'être nulles, en dehors de quelques cas particuliers. Une erreur de ce type est généralement fatale au déroulement de l'algorithme.

Pour surmonter ce problème, le programmeur averti notera sur le point P des informations symboliques indiquant qu'il provient de l'intersection des droites D et D'. Ces informations supplémentaires lui serviront pour déterminer l'appartenance de P aux droites D ou D'. Un autre bon réflexe de programmeur averti consiste à ne pas utiliser des expressions différentes (même algébriquement équivalentes) pour calculer une coordonnée :

$$Y = \frac{ac' - a'c}{ab' - a'b}, \; Y = \frac{-(aX + c)}{b} \text{ ou } Y = \frac{-(a'x + c')}{b'}$$

Ces trois expressions, évaluées en arithmétique flottante, donneront bien souvent des résultats différents.

Nous venons de voir dans l'exemple précédent que la programmation avertie permet de limiter les erreurs numériques de l'arithmétique flottante dans les cas simples. Cependant les cas complexes requièrent une arithmétique exacte ou des solutions sophistiquées adéquates ([KARA 89]). Lorsque l'on utilise l'arithmétique paresseuse en programmation avertie, seuls les cas complexes nécessitent des calculs rationnels, alors que dans une implantation naïve même les cas simples provoqueront des évaluations exactes. Si on tente de comparer deux valeurs numériquement égales mais calculées de manières différentes (cf. exemple précédent) l'arithmétique paresseuse n'a pas d'autre solution que d'effectuer le calcul exact pour s'en rendre compte (tous les tests préliminaires à l'évaluation ayant échoués). Toutes ces remarques peuvent se traduire sous la forme suivante :

- Une programmation naïve conduit à des programmes sensibles aux erreurs de calculs des arithmétiques flottantes, fiables mais très lents en arithmétique paresseuse.
- Une programmation avertie permet d'obtenir des programmes relativement robustes en arithmétique flottante, fiables et efficaces en arithmétique paresseuse.

En conclusion, l'arithmétique paresseuse est aussi transparente que l'arithmétique flottante (mais pas plus) si l'on veut qu'elle soit efficace. En pratique il est préférable d'implanter d'abord une version flottante de l'algorithme qui résolve correctement les cas simples (programmation avertie) puis de faire appel à l'arithmétique paresseuse pour résoudre les cas complexes. Cette démarche obéit au principe fondamental de l'arithmétique paresseuse :

Ne recourir aux calculs exacts
que lorsque les calculs flottants ne fournissent plus un résultat fiable.

Perspectives et extensions 8

Comme nous venons de le voir dans les paragraphes précédents, l'arithmétique paresseuse dans sa description actuelle a certaines limites. Le plus gros problème que nous avons rencontré est celui de l'égalité de deux nombres. Certaines solutions, telles que l'utilisation de classes d'équivalences et l'isomorphisme d'arbres permettent en partie de les résoudre. La restriction actuelle de l'arithmétique aux opérations élémentaires (+,-,* et /) et aux nombres rationnels pour représenter les valeurs exactes limite l'étendue des problèmes que l'on peut résoudre.

Nous allons voir dans ce chapître quelles peuvent être les solutions envisageables et quelles sont les implications de ces solutions sur l'arithmétique paresseuse telle qu'elle est actuellement définie. Certaines de ces propositions ne sont que des extensions et sont simples à ajouter, d'autres nécessitent une remise en cause profonde de la l'implantation actuelle de l'arithmétique paresseuse.

8.1 Unicité des nombres

Pour le moment, des nombres identiques peuvent cohabiter, leur détection (après évaluation si nécessaire) n'est envisagée que lors des comparaisons et dans deux cas bien précis : les deux nombres sont des valeurs rationnelles égales ou les deux nombres sont représentés par des expressions isomorphes. Si deux valeurs sont déterminées comme étant isomorphes à un instant donné, le partage des informations assure qu'elles le resteront. Par contre, si on se place dans le cas où deux quantités avec des représentations identiques ne sont comparées qu'après une évaluation (totale ou partielle) de l'une des deux, ces deux valeurs ne seront déterminées égales qu'après une évaluation complète des deux expressions. L'instant de la comparaison n'est pas sans importance.

Cette remarque met en évidence le problème de l'unicité des nombres. Dire que les nombres doivent être uniques est une chose, mais il faudrait aussi tenir compte de leurs multiples représentations :

$$21 = 3 \times 7 = 17 + 4 = (6 \times 5) - (7 + 2) = \ldots$$

Le problème n'est pas si simple : on a vu que pour détecter l'égalité de deux nombres exprimés de manières différentes il fallait les évaluer de manière exacte. Mais on peut penser (bien qu'il existe une infinité de formes) qu'en conservant, pour chaque nombre, toutes les représentations rencontrées, on puisse souvent trouver des formes identiques et de cette manière éviter des calculs exacts. Les clefs associées aux nombres et les intervalles eux aussi peuvent participer à cette unicité.

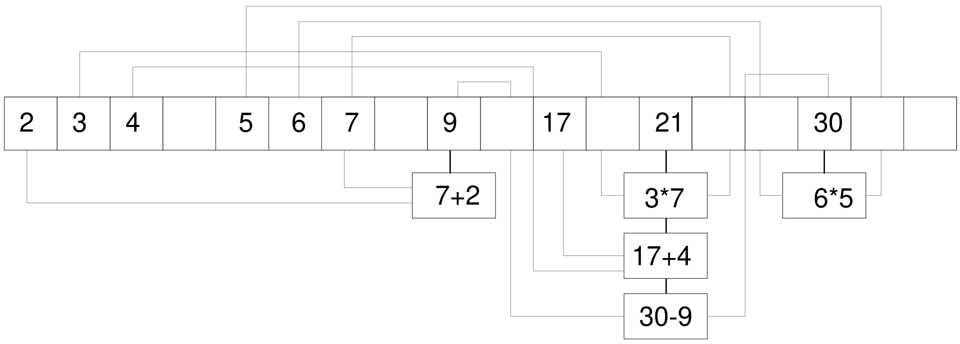


figure 35
Hashtable et unicité des nombres

On peut imaginer la solution suivante : on tient à jour une table d'adressage dispersé contenant tous les nombres déjà rencontrés, chacun de ces nombres est stocké dans la table à la place définie par sa clef. Lors de la création d'un nouveau nombre, on consulte l'entrée de la table correspondante, si l'entrée est vide on installe le nouveau nombre ainsi que sa définition (il est unique) ; si l'entrée n'est pas vide, on compare les intervalles des nombres de manière à éliminer les cas de collision banales dans la table, on se ramène alors au cas précédent si le nombre n'est pas présent dans la table, et dans le cas contraire on compare les différentes définitions du nombre présent dans la table d'adressage avec le nouveau nombre, si aucune définition ne correspond à celle du nouveau nombre il est nécessaire de l'évaluer et de vérifier que l'on est bien en présence de valeurs identiques, dans ce cas on ajoute la nouvelle définition à l'entrée de la table. Dans le cas contraire on a affaire à un nouveau nombre et on dispose maintenant de deux représentations (une expression et un rationnel) pour créer la nouvelle entrée de la table.

Bien que ce processus paraisse lourd, il ne faut pas oublier que pour chaque nombre mis en jeu dans le calcul du nouveau nombre, nous avons aussi de multiples représentations envisageables qui accéléreront beaucoup les détections d'égalité d'arbres car les nœuds sont eux aussi des nombres uniques. Cette technique sera très efficace pour pallier la destruction de l'isomorphisme des arbres qui apparaît lors d'évaluations partielles des expressions. Cette méthode permettra aussi l'utilisation des classes de recalage qui jusqu'à présent avaient le grave défaut de détruire trop souvent les propriétés d'isomorphisme des nombres. Les classes de recalages pourront être utilisées simplement comme d'autres définitions possibles des nombres.

8.2 Ordre

L'unicité des nombres comme que nous venons de le voir est une première étape. Si maintenant il est possible de répondre à la question de l'égalité de deux nombres, leur comparaison est toujours effectuée de la même manière, et il n'est nulle part tiré parti des calculs mis en œuvre pour les ordonner. Si au cours des calculs on est amené à comparer les nombres a et b puis b et c (ces trois nombres ayant des intervalles qui se chevauchent) avec les résultats suivant : $a<b$ et $b<c$, on doit répondre à sans hésitation et sans calcul $a<c$ pour la comparaison de a et c. Pour obtenir cette réponse immédiate il faut conserver d'une manière ou d'une autre le résultat des comparaisons préliminaires.

Ce simple exemple met en évidence l'intérêt de disposer d'une relation d'ordre sur les nombres paresseux. Par contre la mise en œuvre de cette technique réclame que l'on stocke de manière explicite l'ordre entre les nombres dès qu'il est connu, et que lors d'une comparaison qui ne peut être faite à l'aide des encadrements, on fasse une recherche pour tenter de le déterminer : par transitivité parcourir l'ensemble des nombres qui sont inférieur à c et vérifier que a ne se trouve pas sur le chemin, et dans le cas d'une réponse négative vérifier de la même manière que c n'est pas inférieur à a. Si cette recherche ne fournit pas de résultat, on est amené à faire la comparaison par d'autres moyens, et bien entendu on prend soin de conserver le résultat. Les opérations de recherche se font en un temps linéaire et il ne faut pas oublier que cette opération n'est effectuée que lorsque les encadrements des nombres ne permettent pas de répondre. La recherche par transitivité n'a de sens que dans ce contexte ce qui permet d'en limiter le coût moyen.

8.3 Fonctions paresseuses

Les opérateurs arithmétiques qui sont définis dans l'arithmétique paresseuse permettent de faire beaucoup de choses, mais certaines fois, ils ne tirent pas complètement parti des configurations rencontrées.

8.3.1 Puissance entière paresseuse

La multiplication est suffisante pour définir le carré ou le cube d'une valeur. Mais si on simule ainsi ces opérations on ne tient pas compte de certaines propriétés (le carré d'un nombre est toujours positif par exemple) et le résultat que l'on obtient est beaucoup moins précis que ce que l'on peut espérer :

$$[-2, 3] \times [-2, 3] = [-6, 9]$$

alors que l'on peut espérer beaucoup mieux :

$$[-2, 3]^2 = [0, 9]$$

Le cas de l'élévation au cube d'un nombre donne des résultats encore plus flagrants :

$$[-2, 3] \times [-2, 3] \times [-2, 3] = [-18, 27]$$
$$[-2, 3]^3 = [-8, 27]$$

Dans ce cas précis on tient compte du fait que les intervalles encadrent tous la même valeur et que l'on peut utiliser une méthode de calcul plus efficace que dans le cas général.

La création de l'opérateur spécifique puissance entière d'un nombre paresseux qui stockerait le nombre paresseux et la puissance entière permettrait de gagner de la place mémoire, de l'efficacité lors de l'évaluation et de la précision sur les encadrements.

L'élévation d'un nombre à une puissance entière est un exemple, mais il existe bien d'autres fonctions qui peuvent être optimisées de cette manière et rajoutées à l'arithmétique paresseuse. Pour ce faire, il suffit d'être capable de déterminer l'encadrement du résultat de la manière la plus précise possible en tenant compte des singularités du calcul.

8.3.2 Déterminant paresseux

Nous avons vu dans les paragraphes précédents que la complexité en espace mémoire des algorithmes utilisant l'arithmétique paresseuse était modifiée en fonction de la complexité en nombre d'opérations de ces algorithmes. Les opérations telles que les calculs de déterminants sont des cas pour lesquels, plutôt que de construire l'expression symbolique ('$O(n^3)$ nœuds) on peut fabriquer un nouvel opérateur (l'opérateur déterminant) qui conservera ses n^2 paramètres, calculera directement l'encadrement du déterminant à l'aide de l'arithmétique d'intervalles.

Le déterminant sera calculé de manière exacte seulement si une évaluation est demandée (les algorithmes géométriques utilisent le signe du déterminant). Ce nouvel opérateur nous permet de gagner de la place mémoire et le calcul de son encadrement ne faisant pas appel au parcours d'une expression symbolique est plus efficace. Bien entendu, il sera nécessaire de définir soit un opérateur déterminant pour chacune des tailles souhaitée, soit une classe manipulant et stockant des déterminants de taille quelconque (qui pourrait être associée à une classe de matrices). Cette opération est compatible avec le calcul immédiat des clefs de hashage.

8.3.3 Valeur absolue paresseuse

Il existe d'autres sortes de fonctions qui peuvent êtres rendues paresseuses. Le cas de la valeur absolue est simple : on est sûr dans tous les cas que le résultat de cette fonction sera supérieur ou égal à zéro. L'implantation de la valeur absolue sous la forme habituelle (*si x > 0 alors x sinon -x*) est très mauvaise, car cette formulation peut entraîner des évaluations exactes inutiles dans le cas où l'encadrement du résultat contient zéro. Une valeur absolue paresseuse doit simplement fournir un encadrement correct du résultat :

$$\mathrm{abs}([a, b]) = [0, \max(|a|, |b|)]$$

et sa méthode d'évaluation se borne simplement à fournir comme résultat une valeur rationnelle de signe positif (cette opération se fait en temps constant). Malheureusement, il n'est pas possible de calculer immédiatement la clef de hashage d'une valeur absolue : sa valeur est soit celle du nombre dont on prend la valeur absolue si celui-ci est positif, soit P moins la clef de ce nombre s'il est négatif.

8.3.4 Maximum et minimum paresseux

Les deux fonctions minimum et maximum peuvent, elles aussi, devenir paresseuses. Nous étudierons le seul cas du maximum paresseux, le minimum se déduisant facilement par symétrie. Lorsque l'on réclame le maximum de deux nombres paresseux, et que leurs intervalles sont disjoints, on est capable de fournir immédiatement le nombre le plus grand ; sinon, plutôt que d'évaluer les deux expressions paresseuses et de rendre le vrai maximum, on peut fabriquer un nouveau type d'opérateur (l'opérateur maximum) qui conserve de manière symbolique ses opérandes et qui a pour encadrement :

$$\max([a, a'], [b, b']) = [\max(a, b), \max(a', b')]$$

On peut, de cette manière espérer que l'encadrement du maximum paresseux sera suffisant pour la suite des calculs, avec la possibilité d'avoir recours à l'évaluation des expressions symboliques si cela s'avérait nécessaire. Malheureusement le maximum paresseux, comme la valeur absolue n'est pas compatible avec le calcul immédiat de la clef de hashage. La solution à ce problème serait d'avoir des clefs paresseuses elles aussi, qui ne seraient calculées que si elles sont explicitement demandées.

Il existe plusieurs stratégies d'évaluation des maximums paresseux. La méthode naïve évalue les expressions symboliques et rend le maximum. Une deuxième stratégie procède d'abord à la mise à jour des intervalles ou à leur affinement (YoYo). La troisième stratégie est plus intéressante mais aussi plus complexe. Lorsque le maximum des deux nombres doit être déterminé, l'intervalle dans lequel se trouve le maximum est connu, et dès que l'on constate qu'un des deux nombres candidats au maximum ne se trouve plus dans cet intervalle, on peut l'éliminer et arrêter l'évaluation. On appelle plage ce type d'intervalle, hors duquel l'évaluation d'un nombre peut être stoppée car devenue inutile. Dans l'exemple de la figure 36, pour évaluer la valeur du $\max(\max(A, B), C)$ on calcule d'abord le $\max(A, B)$ tout en sachant que le résultat doit se trouver dans la plage $[4, 6]$. Le $\max(A, B)$, initialement dans $[2, 5]$, doit être recherché dans la plage réduite $[2, 5] \cap [4, 6] = [4, 5]$. Cette réduction de plage nous permet d'affirmer que seul B a une chance d'être le maximum, alors que de manière naïve, il aurait été nécessaire d'évaluer A et B : les plages nous permettent d'éliminer A sans faire de calcul exact. On peut maintenant remplacer l'expression $\max(\max(A, B), C)$ par $\max(B, C)$, ou calculer explicitement le résultat si c'est nécessaire. Nous avons donné à ce principe d'évaluation le nom d'évaluation par propagation de plage.

8.3.5 Stratégie d'évaluation par plage

Le principe que nous venons d'énoncer pour la résolution du minimum ou du maximum paresseux peut être étendu. Les valeurs dont on tente de connaître le maximum (ou le minimum) sont des expressions symboliques que l'on doit dans les mauvais cas évaluer. La propagation des plages vers les opérandes va permettre de découvrir des branches de l'expression qu'il ne sera pas nécessaire d'évaluer. Cette méthode d'évaluation utilise le même principe que la clasique optimisation $\alpha\beta$ de la méthode du minimax [AHO 74].

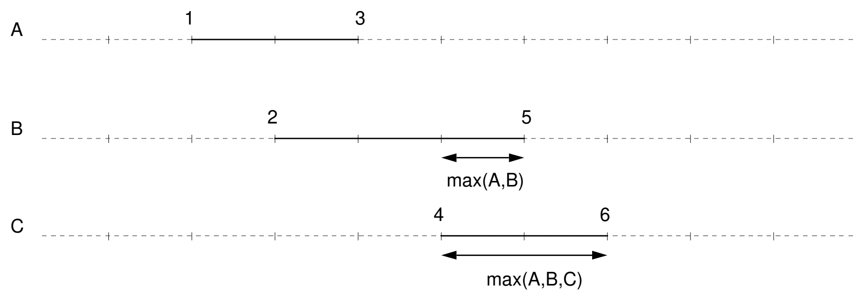


figure 36
Maximum paresseux

Les intervalles et les plages sont gérées de manières indépendantes. La plage d'un nombre est calculée lors de la propagation par intersection de l'intervalle avec la plage (celle qui est propagée) dans laquelle il doit se trouver.

Si on explicite la règle permettant de propager vers le bas la plage courante dans le cas des maximums paresseux on a :

- Soit $Z_p = [z_m, z_M]$ la plage de la valeur $z = \max(x, y)$, avec $X_i = [x_m, x_M]$ l'encadrement de x et $Y_i = [y_m, y_M]$ celui de y. Les plages X_p et Y_p pour x et y sont : $X_p = X_i \cap Z_p$ et $Y_p = Y_i \cap Z_p$.

On doit aussi pouvoir propager les plages pour les opérations élémentaires de manière à être capable de dire dans quelles plages doivent se trouver les opérandes pour obtenir le résultat demandé :

Soit $Z_p = [z_m, z_M]$ la plage de z, $X_i = [x_m, x_M]$ l'encadrement de x et $Y_i = [y_m, y_M]$ celui de y.

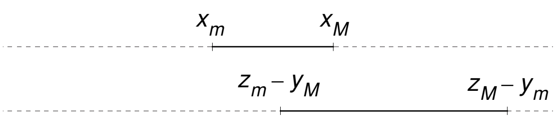
- Opposé : Soit $z = -x$, la plage de x est $X_p = X_i \cap (-Z_p)$ ($[x_m, x_M] \cap [-z_M, -z_m]$).
- Inverse : Soit $z = 1/x$, la plage de $1/x$ est $(1/X)_p = (1/X) \cap Z_p$. On en déduit que la plage de x est $X_p = X_i \cap (1/Z_p)$ si $0 \notin [z_m, z_M]$ (cette condition est toujours remplie, une plage est contenue dans l'encadrement du résultat, et dans le cas présent pour que $[z_m, z_M]$ contienne zéro, il aurait fallu que $[1/x_M, 1/x_m]$ et donc $[x_m, x_M]$ contienne zéro, ce qui aurait provoqué l'évaluation de x lors du calcul de l'inverse).
- Addition : Soit $z = x + y$, les plages des opérandes sont $X_p = X_i \cap (Z_p - Y_i)$ pour x et $Y_p = Y_i \cap (Z_p - X_i)$ pour y.
- Produit : Soit $z = x \times y$, les plages des opérandes sont $X_p = X_i \cap (Z_p / Y_i)$ pour x et $Y_p = Y_i \cap (Z_p / X_i)$ pour y.

Remarque : toutes les opérations sur les plages sont effectuées avec une arithmétique d'intervalles standard.

8.3.5.1 Réduction des plages des opérandes dans le cas de l'addition

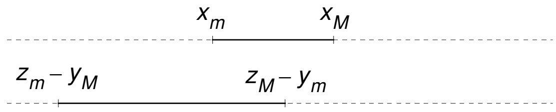
Soit $Z_p = [z_m, z_M]$ la plage associée à $z = x + y$ et $X_i = [x_m, x_M]$ et $Y_i = [y_m, y_M]$ les encadrements de x et y. La plage associée à x est $X_p = X_i \cap [z_m - y_M, z_M - y_m]$. La plage de x (X_p) peut se trouver réduite dans les cas suivants :

- Si $z_m - y_M > x_m \Leftrightarrow z_m > x_m + y_M$



La nouvelle plage de x, $X_p = [z_m - y_M, x_m]$ est réduite de $z_m - y_M - x_m$.

- Si $z_M - y_m < x_M \Leftrightarrow z_M < x_M + y_m$

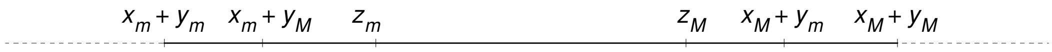


La nouvelle plage de x, $X_p = [x_m, z_M - y_m]$ est réduite de $x_M - (z_M - y_m)$.

et dans le cas où $[z_m - y_M, z_M - y_m] \subset X_i$ on peut définir la relation suivante :

$$x_m + y_m < x_m + y_M \le z_m < z_M \le x_M + y_m < x_M + y_M \text{ si } y_M - y_m < x_M - x_m$$

Dans ce cas cette relation peut se représenter graphiquement comme ci-dessous :



Si on effectue les mêmes calculs pour la plage de y on obtient : $z_m > y_m + x_M$ et $z_M < y_M + x_m$.

Ces remarques permettent de voir que les plages de x et y ne peuvent pas toujours être réduites de manière simultanée. Les conditions posées sur les bornes de Z_p et sur $w(X_i)$ et de $w(Y_i)$ sont contradictoires dans certains cas.

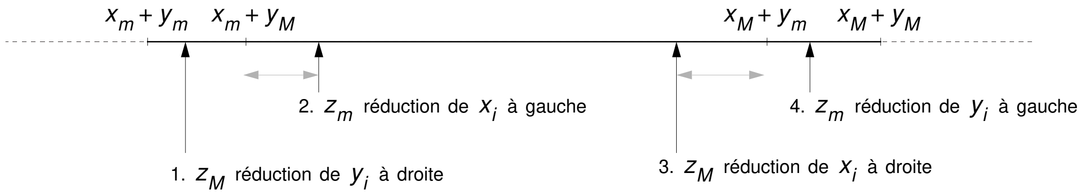


figure 37
Réductions des encadrements par propagation de plage

La réduction de taille que l'on peut obtenir pour la plage d'une valeur x est :

$$\delta_x = w(X_i) - w(X_p \cap X_i) .$$

La réduction des plages n'intervient pas dans toutes les configurations : si on se place dans le cas de l'exemple précédent, la plage X_p associée à x se réduit (est plus petite que l'intervalle X_i) si $z_m > x_m + y_M$ ($\delta = z_m - (x_m + y_M)$) ou si $z_M < x_M + y_m$ ($\delta = (x_M + y_m) - z_M$). Les valeurs de réduction sont symbolisées sur la figure 37 par les flèches en pointillés.

Si on se place dans le cas où $x_m + y_m < x_m + y_M$ qui est illustré par la figure 37, la réduction des plages X_p ou Y_p se décompose en quatre cas :

1. $x_m + y_m < z_M < x_m + y_M$:
2. $x_m + y_M < z_m < x_M + y_m$:
3. $x_m + y_M < z_M < x_M + y_m$:
4. $x_M + y_m < z_M < x_M + y_M$:

Nous avons supposé dans cet exemple que $x_m + y_m < x_m + y_M$. Mais si se n'est pas le cas, il suffit simplement d'échanger les rôles de x et de y. Les calculs précédents sont généraux.

Les plages dans lesquelles doivent se trouver les opérandes peuvent se réduire et vont permettre d'éliminer des calculs inutiles. Après la propagation des plages, si le résultat d'une évaluation se trouve à l'extérieur de la plage qui nous intéresse, on peut immédiatement abandonner la suite des calculs, le résultat ne pouvant pas se trouver dans la plage initiale.

Ce type d'approche est aussi envisageable lors d'une comparaison. On peut considérer l'intervalle le plus petit des deux nombres paresseux à comparer comme une plage, et propager cette plage sur les opérandes de la définition symbolique de l'autre nombre. La comparaison s'arrête lorsque la plage d'un opérande devient vide et l'ordre des nombres peut éventuellement être déduit.

Remarque : si l'on programme l'algorithme du minimax de manière simple (voire simplette), l'utilisation de la propagation de plage effectue de manière automatique l'optimisation $\alpha\beta$.

8.4 Les booléens paresseux

Certains tests de la forme “si (x<y) et (y<z) alors …” sont une source de paresse inexploitée. Il se peut que la première condition exige une évaluation coûteuse de x et de y alors que la seconde est manifestement fausse à la simple vue des intervalles de y et z. D'où l'idée de définir des booléens paresseux.

Les booléens paresseux se définissent par analogie avec les nombres paresseux. Un booléen paresseux est soit un booléen usuel (vrai ou faux), soit une définition symbolique de type et, ou, négation pour les opérateurs logiques, inférieur, supérieur, différent, etc, pour les comparaisons. Une définition symbolique fait référence à d'autres booléens dans le cas des opérateurs logiques et à des nombres paresseux dans le cas des comparaisons.

Les tests de comparaison (<, >, = …) et les opérateurs logiques retournent immédiatement le résultat vrai ou faux lorsque la réponse est évidente sans calculs coûteux ; sinon ils retournent un booléen paresseux du type adéquat avec les champs de la définition symbolique mis à jour. Les évaluations coûteuses sont ainsi retardées jusqu’a ce qu’elles deviennent soit inutiles, soit inévitables.

Pour chacun de ces nouveaux opérateurs paresseux, il existe une stratégie naturelle d’évaluation.

```
evaluer a && b  :  si ( a.eval() == VRAI ) rendre b.eval(); sinon rendre FAUX;
evaluer a || b  :  si ( a.eval() == VRAI ) rendre VRAI; sinon rendre b.eval();
evaluer !a      :  rendre !a.eval()

evaluer x < y   :  recours à l’arithmétique paresseuse
```

8.5 D’autres arithmétiques exactes

L’implantation proposée de l’arithmétique paresseuse utilise en dernier recours une arithmétique exacte rationnelle. Cette arithmétique ne nous permet de résoudre que les problèmes d’essence rationnelle. Il ne nous est pas possible, par exemple, de disposer de l’opérateur racine carrée. Si dans beaucoup de cas on peut contourner le problème, certaines fois, on bute sur les limites de l’arithmétique rationnelle.

Comme nous l’avons vu tout au long de ce chapitre, la paresse utilise de multiples représentations pour éviter les calculs exacts. Mais ces calculs exacts ne sont pas forcément des calculs rationnels. L’arithmétique exacte peut être remplacée par une arithmétique quelconque (algébrique, modulaire) à partir du moment où on est capable de l’implanter correctement et de fournir les deux représentations supplémentaires essentielles (encadrements et clefs de hashage) au bon fonctionnement de la paresse.

Conclusion 9

Nous avons vu dans cette première partie quelles étaient les problèmes dus à l'imprécision des arithmétiques flottantes, les défauts et les qualités des solutions déjà proposées dans la littérature pour résoudre ce problème (chapitre 2). L'arithmétique paresseuse ainsi que le concept plus général de la paresse nous ont permis d'apporter une solution efficace aux problèmes de l'exactitude et de la fiabilité des calculs.

L'utilisation de représentations multiples, les encadrements des résultats et les descriptions symboliques des nombres, nous permettent de calculer efficacement tant que les encadrements le permettent et de n'avoir recours à la description symbolique que pour effectuer les calculs exacts indispensables. L'arithmétique paresseuse gère par elle même la précision et la fiabilité des calculs et décharge le programmeur des problèmes d'imprécisions numériques (chapitre 4).

L'utilisation de l'arithmétique paresseuse nous a permis de mettre en évidence la notion de programmation avertie (chapitre 7). La programmation naïve d'un algorithme entraînera de nombreux calculs exacts (le programme naïf sera lent), alors qu'une programmation avertie les réduira le plus possible.

Bien que dans le pire des cas, l'ensemble des calculs doit être effectué de manière exacte (donc lentement), les résultats présentés au chapitre 6 montrent qu'en général très peu d'opérations rationnelles sont nécessaire pour conserver des résultats fiables. L'arithmétique paresseuse permet d'obtenir des gains de temps conséquents sur les arithmétiques exactes (jusqu'à 200 fois plus rapide) tout en restant suffisamment efficace : elle est seulement 20 à 30 fois plus lente que les nombres flottants.

L'arithmétique paresseuse offre de nombreuses possibilités d'extensions (chapitre 8). L'utilisation d'un langage objet pour son implantation (C++) permet de rajouter facilement des opérateurs supplémentaires spécialisés (déterminant). Une orientation possible de l'arithmétique paresseuse est d'utiliser des nombres algébriques pour les calculs exacts et ainsi de fournir de nouveaux opérateurs.

L'arithmétique paresseuse a fait l'objet de publications dans [BJMM 93.1], [BJMM 93.2] et [BJMM 93.3] ainsi que son utilisation dans [BENO 93.1].

2 Saisie d'objets en synthèse d'images

Le processus conduisant à la création d'images de synthèse tridimensionnelles est maintenant bien connu ([PERO 88], [FOLE 90]). Il s'agit d'une structure pipelinée comportant les quatre modules suivants : gestion du contenu de la scène, calculs géométriques (fenêtrage, projection,...), calculs de rendu de surface et affichage.

Le module de gestion du contenu de la scène comprend notamment une partie saisie qui constitue souvent un goulot d'étranglement du processus de synthèse : pour saisir une scène complexe (l'équivalent de plusieurs centaines de milliers de polygones), le processus est long et fastidieux.

L'objet de cette deuxième partie est de présenter une méthode permettant de faciliter la phase de saisie d'une scène 3D : à partir de données minimales sur un objet volumique (une ou plusieurs photographies digitalisées), l'objectif est de retrouver sa géométrie tridimensionnelle.

La méthode que nous allons décrire met en œuvre des concepts simples et efficaces : ceux de l'interaction homme-machine et ceux de l'assistance par ordinateur. L'idée est d'utiliser les compétences de l'opérateur pour la modélisation et de l'aider dans le placement des divers éléments par un jeu d'attracteurs. Cette méthode a l'avantage de résoudre le problème des éléments partiellement masqués ou incomplets sur la prise de vue. Cette particularité est très importante dans le cadre d'une utilisation industrielle, où les conditions de mise en œuvre sont loin d'être aussi strictes qu'en laboratoire.

Nous rappellerons d'abord, dans le paragraphe 1, le principe de la stéréovision qui est la méthode classique pour reconstruire un volume à partir de plusieurs images d'une scène. Nous exposerons notre méthode dans le paragraphe 2, en présentant les calculs de matrices de projection nécessaires pour résoudre notre problème ainsi qu'une solution originale au problème de la recherche des paramètres de prise de vue (§ 3). Nous terminerons cette partie en décrivant une application industrielle de notre méthode et en présentant des extensions de cette technique pour l'incrustation intelligente d'objets synthétiques dans un décor naturel fixe ou animé.

Solutions existantes : état de l’art 1

La perte de la troisième dimension sur une photographie empêche la reconstruction directe des divers éléments la constituant. Un point sur une image est la projection de la droite passant par ce point et par l'œil de l'observateur ; il existe donc une infinité de points qui se projettent au même endroit sur l'image (figure 38).

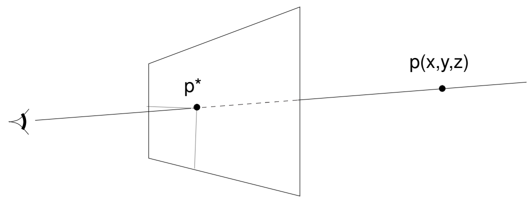


figure 38
Projection d’une droite passant par l’œil de l’observateur

Les méthodes actuelles de vision dans l'espace utilisent généralement deux caméras fixes fournissant deux images d'une même scène sous des angles de prise de vue différents. Avec cette méthode, l'angle et la distance entre les caméras sont fixés, ce qui permet de connaître par triangulation la position spatiale de tous les points visibles dans les deux images (figure 39). Une autre méthode utilise une caméra mobile qui fournit des images successives de la scène à partir de points de vue différents. Cette deuxième méthode se rapproche de celle à caméras fixes, les positions successives (angulaires et spatiales) de la caméra entre deux prises de vue permettant de calculer l'angle et la distance que font entre elles deux caméras virtuelles.

Après la saisie et le traitement des images (généralement une détection de contours), les algorithmes proposés jusqu'à ce jour tentent d'apparier des points, des segments ou des surfaces pour reconstruire de la manière la plus parfaite possible la scène photographiée [FAUG 84].

Dans tous les cas, les résultats obtenus sont insuffisants pour reconstruire un objet de façon cohérente. Les méthodes qui apparient les points fournissent une liste de points dont les coordonnées sont connues dans l'espace, mais qui n'ont aucune signification volumique (cette liste de points permet de construire plusieurs volumes différents). De plus, les objets qui ne sont pas présents sur les deux photographies, ceux qui sont partiellement masqués, les faces arrières d'un objet prismatique connu ne peuvent pas être reconstruits spatialement.

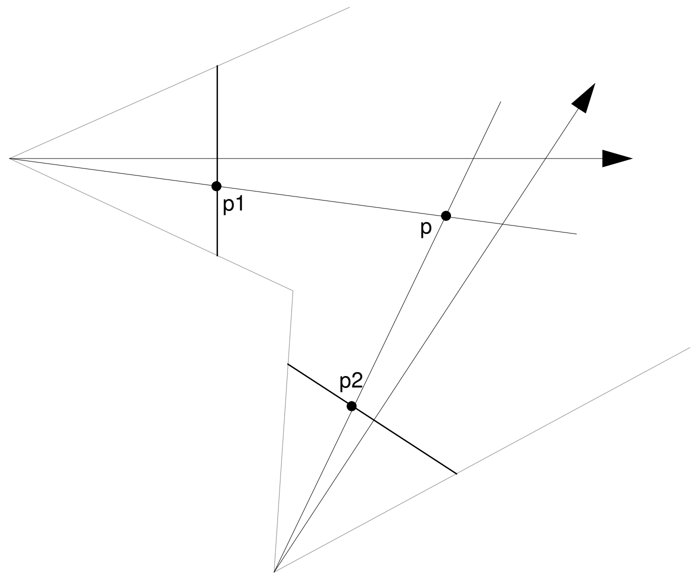


figure 39
Principe général de la stéréovision

Les techniques les plus avancées dans le domaine de la vision et de la reconstruction 3D d’objets utilisent aujourd’hui ce que l’on appelle la vision continue [FAUG 87]. Une caméra mobile permettra de fabriquer, de compléter et d’affiner la représentation 3D de l’objet au cours de son déplacement. Cette méthode est à rapprocher de la vision stéréoscopique, le mouvement de la caméra fournissant un grand nombre d’images différentes pour décrire l’objet. De cette manière, on pourra reconstruire des surfaces polyèdriques ou même quelconques ([ARBO 89]). Ces techniques très sophistiquées sont malgré tout limitées dans leur emploi par le matériel et surtout sa mise en œuvre, qui doit être réalisée dans des conditions proches de celles existant en laboratoire.

Ces différentes méthodes permettent par exemple à un petit robot de se déplacer sans heurter un obstacle à chacun de ses mouvements. Elles ne lui permettront pas de découvrir qu’il se trouve face à une fenêtre partiellement masquée par un arbre.

1.1 Vision stéréoscopique

Au sens étymologique du terme, stéréovision signifie “vision du solide”, c’est à dire vision tridimensionnelle. Cela revient à chercher à obtenir une représentation en trois dimensions d'un objet ou d'un environnement perçu par l'intermédiaire d'un capteur fournissant des représentations en deux dimensions du monde : par exemple des photographies.

Le principe de départ de la stéréovision consiste à tenter d'imiter le système de perception visuel humain. Avec une paire d'yeux, nous apprécions parfaitement les distances ainsi que les volumes.

Essayez de pointer rapidement avec un stylo des lettres de ce texte vu en biais. Les deux yeux ouverts, vous tomberez toujours sur les lettres que vous vous serez désignées. Avec un œil fermé, vous pourrez constater que c'est beaucoup moins évident et que les erreurs commises ne sont pas si négligeables. Evidemment, avec la pratique, vos performances vont s'améliorer. Nous disposons inconsciemment d'une foule d'outils très perfectionnés tenant compte de l'éclairage, des reflets, de nos connaissances préalables de la scène observée. Les machines peuvent encore difficilement utiliser ces outils ; c'est pourquoi les systèmes de vision utilisent deux ou trois prises de vue différentes pour la même scène.

L'étude géométrique du système est assez simple. Lors de l'enregistrement d'une image, on effectue une projection perspective d'un espace à trois dimensions sur un espace à deux dimensions (une photographie ou la rétine de l'œil). Un point quelconque de la photographie peut avoir comme antécédents toute la droite passant par l'œil et ce point.

1.1.1 Décomposition du problème

La résolution du problème que pose la stéréovision passe par le traitement de quatre sous-problèmes relativement indépendants :

- La calibration :

  Cette phase consiste à déterminer le plus exactement possible la manière dont les caméras ou les appareils photo interprètent ce qu'ils observent. En pratique, on étudie les transformations subies par une mire 3D de référence. Ces opérations permettent de connaître exactement les caractéristiques des périphériques de saisie : position des centres optiques, des distances focales, les défauts de projection (linéaires ou non) ainsi que les positions relatives des caméras si on en utilise plusieurs.

- Extraction des primitives :

  Pour reconstruire un objet, on va calculer les coordonnées des points de sa surface. La solution classique part d'une paire d'images stéréoscopiques. On ne peut dans la pratique calculer la position de tous les points des images. Les images que l'on manipule sont discrétisées pour être traitées et il est de manière générale à peu près impossible de trouver dans une image le point correspondant dans l'autre image, celui-ci n'existant pas forcément ou n'étant pas suffisamment caractéristique. On est donc amené à découvrir des correspondances entre des primitives plus riches telles que des contours, des segments de droites ou des surfaces homogènes. Des techniques de traitement d'images (détection de contours, segmentations et vectorisations) permettent de construire de telles primitives à partir des images originales.

- Appariement :

  Phase qui consiste à déterminer les correspondances entre les primitives des différentes images.

- Reconstruction 3D :

  L'étape précédente a fourni un ensemble de primitives 2D appariées. Il s'agit maintenant d'en déduire les primitives 3D correspondantes par une projection perspective inverse. De plus, il faut compléter le résultat obtenu en tentant de reconstruire correctement le volume observé.

1.1.2 Géométrie épipolaire

Afin de limiter la complexité des recherches de primitives se correspondant et les temps de calculs que cela implique, on est amené à utiliser une contrainte géométrique supplémentaire : la contrainte d'épipolarité. Cette contrainte permet de limiter la recherche d'un point m_b d'une image I_B, correspondant d'un point m_a d'une image I_A à une droite de I_B (à la place de l'image complète).

Considérons un système constitué de deux caméras dont on connaît les centres optiques C_A et C_B (figure 40). Les points E_A et E_B sont les images respectives des centres optiques C_B et C_A (E_A est l'image de C_B dans I_A et réciproquement). E_A et E_B sont appelés les épipoles du système. Etant donné un point M_A projection sur l'image I_A d'un point inconnu M, le point M_B, image de M sur I_B est à rechercher sur la droite qui est l'intersection du plan défini par C_A, E_A et M_A et de l'image I_B, de plus cette droite passe par le point E_B. Cette droite d'intersection est la droite épipolaire de I_B.

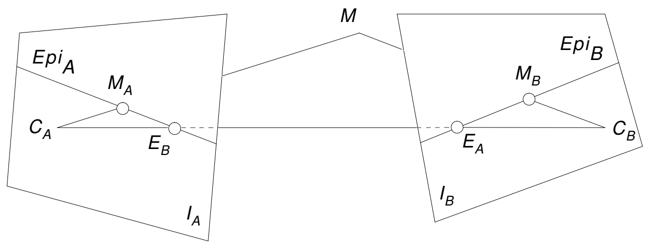


figure 40
Géométrie épipolaire

Cette propriété géométrique permet de limiter la recherche des points à mettre en correspondance à une droite de l'image voisine. L'équation de cette droite est simple à calculer si l'on possède des informations géométriques suffisantes sur les positions relatives des deux caméras (une étape de calibrage fournit ces informations).

La technique des droites épipolaires peut être utilisée avec des systèmes trinoculaires. Dans ce cas, en plus de faciliter l'appariement, les droites épipolaires construites pour chaque couple d'images permettent de vérifier la qualité de l'opération de mise en correspondance : sur chaque image, les points en correspondance doivent se trouver à l'intersection des trois droites épipolaires.

1.2 Vision monoculaire

La vision monoculaire tend à ce développer de plus en plus. Elle part du principe que l'on dispose de suffisamment d'informations sur une seule photographie pour y retrouver les informations géométriques (position de la caméra et des objets) qui nous intéressent. Les traitements s'appuient sur les invariants projectifs comme le birapport [MOHR 91] qui permettent de retrouver des informations spatiales à partir de données en deux dimensions.

Le déroulement du traitement en vision monoculaire est en partie identique à ceux que l'on emploie en stéréovision, à ceci près qu'il n'y a plus d'opérations d'appariement.

- La calibration :

  De nombreux auteurs se sont intéressés à ce sujet qui demeure fondamental pour obtenir un modèle correct de la caméra et ses coordonnées spatiales. On peut citer [ROGE 76] qui donne les notions élémentaires pour le positionnement de la caméra, [HARA 89] et [HORA 89] pour leurs solutions analytiques au problème et [PEUC 91] dont les travaux fournissent des solutions explicites à la position et aux paramètres de la caméra (orientation, focale). Nous fournissons au § 3.3 une solution originale à ce problème qui se base sur les angles sous lesquels on voit des segments de l'image.

- Extraction des primitives :

  Les procédés de traitement d'images employés sont identiques à ceux de l'extraction de primitives de la stéréovision.

- Reconstruction 3D :

  En général, on ne peut pas encore parler de reconstructions 3D automatiques. Si l'on possède certaines informations topologiques (tel point appartient à tel plan), il est possible de calculer de manière précise des postions spatiales [MORE 90] à partir de la seule photographie dont on dispose. Une autre approche est la reconnaissance et le positionnement d'objets particuliers dont on a une description géométrique. Les travaux de [BATA 91] lui permettent de reconnaître des objets polyèdriques présents sur une image (et dans sa base de donnée) en utilisant les propriétés des invariants projectifs comme la colinéarité, l’adjacence et l’ordre des sommets, mais aussi le bi-rapport. Cette technique permet de positionner et de reconnaître des objets même partiellement masqués.

Une nouvelle approche 2

Dans ce paragraphe nous allons présenter une approche innovante de la reconstruction tridimensionnelle de scènes polyèdriques à partir de photographies. Nous avons intentionnellement limité à une le nombre minimum de photographies nécessaires et nous n'imposons aucune contrainte particulière sur la prise de vue. L'outil que nous allons décrire présente l'avantage d'être souple et utilisable par des personnes sans compétences particulières dans le domaine de la vision 3D. Il s'appuie sur les interactions homme-machine pour résoudre des problèmes comme celui des objets partiellement masqués.

2.1 Utilisation des connaissances de l'opérateur

Nous avons vu, dans le paragraphe précédent, qu'aucune méthode n'était encore susceptible de fournir une reconstruction tridimensionnelle d'une scène quelconque de manière automatique. Toutes les limitations sont dues, semble-t-il, à l'absence de connaissance des machines sur les objets qu'elles tentent de reconstruire. Pour l'instant, seul l'homme reste capable de discerner les limites d'un balcon en partie masqué par les feuillages d'un arbre.

Les reconstructions 3D à partir de photographies peuvent avoir, selon l'utilisation que l'on veut en faire, des degrés de précision très variés (de la volumétrie générale à la reconstruction très minutieuse). Seul un opérateur humain sera à même de faire la différence entre le détail anecdotique et l'élément essentiel (la machine risque souvent de les mélanger) dans le cadre d'une application particulière. La présence d'un opérateur permet d'assurer que seulement ce qui est nécessaire sera reconstruit. Si la reconstruction va perdre un peu de son attrait (elle n'est plus automatique), cette manière de procéder va permettre d'obtenir des bases de données plus compactes mais aussi plus riches. L'opérateur peut mettre à profit la phase de saisie pour attribuer certaines propriétés aux objets qu'il manipule : il s'agit d'une fenêtre, l'objet est rouge, etc. Toutes ces informations supplémentaires, si elles étaient indispensables, auraient de toute façon dues être laborieusement saisies.

2.2 Le monde à l'envers

Nous avons vu au § 1 quel était l'état de l'art et quelles étaient les limites des diverses techniques employées en vision monoculaire et binoculaire. Il semble pour l'instant que l'on ne dispose pas d'une technologie suffisante pour reconstruire de manière cohérente les divers objets qui se trouvent sur une photographie. Par reconstruction cohérente, nous entendons la reconstruction d'un solide valide (avec un intérieur et un extérieur) dont nous n'avons que la représentation photographique. Nous avons vu qu'il était seulement envisageable de calculer des positions spatiales et dans le meilleur des cas de reconnaître des objets (la reconnaissance sous-entend que l'on connaisse ce que l'on cherche).

L'état actuel de la vision par ordinateur nous permet de connaître sans difficulté la position et les l'orientation de la caméra (étape de calibration). Les techniques de l'infographie et la synthèse d'images nous fournissent les outils de modélisation et de visualisation nécessaire pour la manipulation dans l'espace des solides élémentaires.

Le mélange de ces deux techniques va nous permettre de prendre le contre pied de ce que l'on pratique habituellement. Au lieux de rechercher les coordonnées en trois dimensions des objets qui sont présents sur la photographie, nous allons tenter de superposer la projection d'un objet en trois dimensions sur sa photographie.

2.3 Un outil interactif de reconstruction 3D

2.3.1 Présentation

Le problème que nous voulons résoudre est celui de la reconstruction, de manière interactive, d'un volume (approximation polyèdrique) décrit par sa représentation en perspective (sous forme de photographies). Pour ce faire, on utilisera une feuille de papier calque en trois dimensions qui contiendra des primitives géométriques (cubes, cônes, sphères…) que l'on projettera sur l'image et que l'on mettra en correspondance avec les volumes à déterminer. Pour éliminer toute ambiguïté sur la distance en profondeur de l'objet manipulé, les primitives qui le composent se déplaceront uniquement au contact d'objets déjà en place dans l'image. Les opérations que permet cette méthodologie sont les translations, les rotations et de manière générale toutes les déformations sur les objets qui pourront conserver au moins un point de contact avec un objet préalablement mis en place. De cette manière, il devient simple de construire un objet quelconque à partir d'une approximation polyèdrique. Le choix de la finesse de cette approximation est laissé à l'utilisateur.

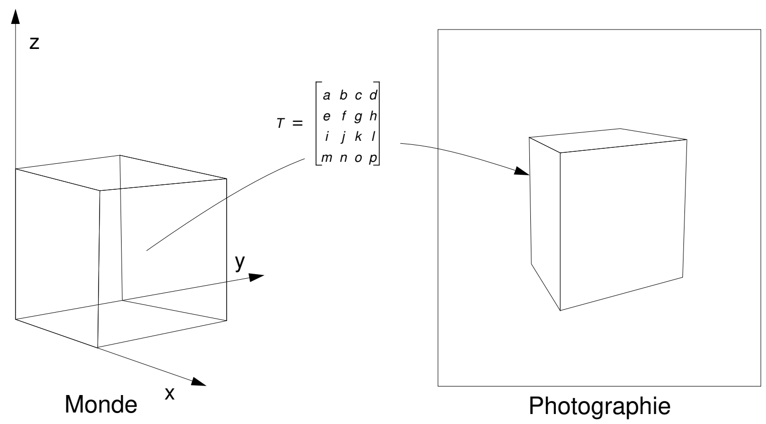


figure 41
Photographie : projection du monde sur un plan

L'idée la plus importante est de ne pas rechercher une transformation qui nous permettrait de passer directement de la photographie à l'espace tridimensionnel, mais au contraire de rechercher une transformation qui permettrait de projeter une primitive géométrique de la même manière que l'appareil photographique (cf. figure 41).

La méthode que nous proposons est adaptée de celle développée dans [ROGE 76]. Elle présente un énorme avantage si l'optique utilisée lors de la prise de vue est de bonne qualité (pas trop de déformations optiques) : elle permet d'éliminer tous les problèmes d'étalonnage, de recherche du centre optique, et surtout elle autorise toute latitude lors de la prise de vue (l'appareil photo peut être orienté de façon quelconque dans l'espace).

Par contre cette latitude pour la prise de vue se paye par le nombre de données numériques nécessaires pour la recherche de la matrice de transformation entre le monde et la photographie. Il faut disposer des coordonnées de six points non coplanaires dans le repère du monde et sur la photographie. Notre méthode ne cherchant à retrouver ni le point de vue, ni le point visé, nous allons obtenir une matrice 4x4 homogène qui est suffisante pour tous nos calculs.

Pour ce faire, nous utiliserons la matrice de projection qui a permis de passer de l'objet réel à sa photographie. Cette matrice existe et est unique ; malheureusement, elle n'est pas inversible (une colonne complète ne contient que des 0), ce qui nous empêche de passer directement de la photographie à la représentation tridimensionnelle. Le principe, pour contourner cette difficulté, consiste à passer du monde 3D à la photographie et à superposer la projection d'un objet 3D et sa représentation photographique.

Calcul de la matrice de projection 3

La littérature est pleine de “méthodes miracles” pour résoudre ce type de problèmes. Mais il faut constater que ces méthodes marchent en fait assez mal. En effet, dans tous les cas, les algorithmes proposés fournissent des résultats fortement instables ou pas de résultat du tout.

Il ne faut pas oublier que les photographies numérisées sur lesquelles nous travaillons ne font que 768 sur 576 points (format vidéo au standard PAL). Ce qui signifie qu’en dehors de la précision du relevé des points (à la souris !!!) l’erreur commise sur la position de chaque pixel est de l’ordre de un pour mille ; ce qui est nettement insuffisant au vu du nombre d’opérations à effectuer. Les divers algorithmes ne donnaient en définitive que des résultats avec au mieux une précision de l’ordre de 10%.

3.1 Méthode analytique

Il s'agit, à partir d'un nombre limité de données, de calculer les paramètres de la projection perspective qui ont été appliqués pour obtenir la photographie de notre objet. Ces paramètres peuvent se représenter sous la forme d'une matrice de 4 lignes et 4 colonnes qui permet la transformation d'un point $P(x, y, z, 1)$ en coordonnées homogènes en son homologue $P*(X, Y, Z, h)$ projeté sur la photographie.

Cette matrice T est obtenue à partir de deux matrices T_0 et T_1. T_0 est une matrice de transformation quelconque dans l’espace objet et T_1 est une matrice de projection perspective sur le plan $z = m$. On remarque que le produit d’un point par la matrice T_1 nous donne un nouveau point dont la composante z sera toujours égale à la distance m du plan de projection. Si ce type de transformation simule correctement le fonctionnement d’un appareil photographique, il se trouve que T_1 n’est pas inversible.

$$T = \begin{bmatrix} a & b & c & 0 \\ d & e & f & 0 \\ g & h & i & 0 \\ j & k & l & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & m^{-1} \\ 0 & 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} a & b & c & cm^{-1} \\ d & e & f & fm^{-1} \\ g & h & i & im^{-1} \\ j & k & l & lm^{-1} \end{bmatrix}$$

Nous lui préférerons une matrice de projection perspective inversible (telle qu'elles sont utilisées en synthèse d'images) en prenant garde de ne plus utiliser z qui n'a plus le sens que nous lui avions donné précédemment (ce n'est plus le plan de projection). La matrice T sera alors de la forme suivante :

$$T = \begin{bmatrix} a & b & c & 0 \\ d & e & f & 0 \\ g & h & i & 0 \\ j & k & l & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & m^{-1} \\ 0 & 0 & -1 & 0 \end{bmatrix} = \begin{bmatrix} a & b & 0 & cm^{-1} \\ d & e & 0 & fm^{-1} \\ g & h & 0 & im^{-1} \\ j & k & -1 & lm^{-1} \end{bmatrix}$$

La projection perspective d'un point quelconque de l'espace sera de la forme :

$$[x, y, z, 1] . T = [X, Y, Z, h]$$

L'utilisation des coordonnées homogènes nous permet de représenter un point sous la forme :

$$[x, y, z, 1] . T = h [x^*, y^*, z^*, 1] \qquad [30]$$

Ce qui donne après calcul :

$$\begin{bmatrix} x^* \\ y^* \\ z^* \\ 1 \end{bmatrix} = \begin{bmatrix} X/h \\ Y/h \\ Z/h \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{ax+dy+gz+j}{cx+fy+iz+l}m \\ \frac{bx+ye+hz+k}{cx+fy+iz+l}m \\ \frac{-1}{cx+fy+iz+l}m \\ 1 \end{bmatrix}$$

Après un changement de variables ($C = \frac{c}{m}, F = \frac{f}{m},\ I = \frac{i}{m},\ L = \frac{l}{m}$), on obtient :

$$\begin{aligned} x^* &= \frac{ax+dy+gz+j}{Cx+Fy+Iz+L} \\ y^* &= \frac{bx+ey+hz+k}{Cx+Fy+Iz+L} \\ z^* &= \frac{-1}{Cx+Fy+Iz+L} \end{aligned} \qquad [31]$$

ce qui nous donne :

$$Cxx^* + Fyx^* + Izx^* + Lx^* - ax - dy - gz - j = 0$$
$$Cxy^* + Fyy^* + Izy^* + Ly^* - bx - ey - hz - k = 0$$
$$Cxz^* + Fyz^* + Izz^* + Lz^* + 1 = 0$$

z^* (profondeur du point projeté) est une inconnue supplémentaire qui apparait avec les contraintes que nous nous sommes fixé sur l'inversibilité de la matrice de projection. L'équation $Cxz^* + Fyz^* + Izz^* + Lz^* + 1 = 0$ est artificielle et ne nous fournit aucun renseignement supplémentaire. On ne conserve donc que les deux premières équations et il ne nous reste plus que 12 inconnues. A l'aide de 6 points (par exemple 6 des sommets d'un cube) $P_i(x_i, y_i, z_i, 1)$ qui ont pour valeur projetée $P_i^*(x_i^*, y_i^*, 0, h_i)$ pour i variant de 1 à 6, on obtient le système d'équations suivant :

$$\begin{bmatrix} -x_1 & 0 & x_1x_1^* & -y_1 & 0 & y_1x_1^* & -z_1 & 0 & z_1x_1^* & -1 & 0 & x_1^* \\ 0 & -x_1 & x_1y_1^* & 0 & -y_1 & y_1y_1^* & 0 & -z_1 & z_1y_1^* & 0 & -1 & y_1^* \\ \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\ -x_6 & 0 & x_6x_6^* & -y_6 & 0 & y_6x_6^* & -z_6 & 0 & z_6x_6^* & -1 & 0 & x_6^* \\ 0 & -x_6 & x_6y_6^* & 0 & -y_6 & y_6y_6^* & 0 & -z_6 & z_6y_6^* & 0 & -1 & y_6^* \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \ldots \\ 0 \\ 0 \end{bmatrix}$$

Malheureusement, ce système est homogène. Un certain nombre d'équations sont donc liées entre elles ; il suffit de les trouver et le système résultant aura alors une solution autre que le vecteur nul.

Nous indiquons maintenant une méthode simple et efficace pour faire disparaître les équations liées entre elles. La diagonalisation de Gauss avec utilisation du pivot maximum donne une valeur très petite pour le dernier pivot (proche de zéro). Dans ce cas, cette valeur "epsilonnesque" fait que le résultat associé à ce pivot n'est pas significatif ($0.L = 0$) et peut donc être fixé (par exemple $L = 1$). On obtient alors un système de 11 équations à 11 inconnues qui n'est plus homogène et qui a donc une solution.

Cette méthode de résolution a pour principal avantage d'être simple et directe ; elle comporte cependant un inconvénient majeur : la matrice 12x12 représentant le système linéaire est très mal conditionnée et de fait très sensible à la qualité des données lors de son utilisation avec des coordonnées entières (coordonnées des pixels de l'écran). Pour pallier ce défaut, nous préférons rechercher le sous-système 11x11 le mieux conditionné et de cette manière retirer l'équation la moins indépendante du système. L'interaction est utilisée en supplément pour fournir à l'opérateur un retour graphique immédiat et lui permettre d'affiner le résultat obtenu. Une fois le calcul de la matrice effectué, on projète sur la photographie une grille en trois dimensions construite à partir des points du monde que l'opérateur a fourni lors de la phase de saisie. La projection de cette grille permet de se rendre compte visuellement et immédiatement de la qualité de la matrice de projection.

La matrice de projection que l'on obtient après avoir fixé l'inconnue L est de la forme :

$$T = \begin{bmatrix} a & b & 0 & C \\ d & e & 0 & F \\ g & h & 0 & I \\ j & k & -1 & 1 \end{bmatrix} \qquad [32]$$

Cette matrice est une projection perspective sur le plan $z = 0$, et comme toute bonne projection perspective qui se respecte en synthèse d'image, elle est inversible. Dans le cadre de notre application, cette particularité, simplifiera beaucoup les calculs pour les programmes interactifs qui auront besoin d'utiliser cette matrice.

3.2 Méthode des moindres carrés

Nous avons vu dans le paragraphe précédent que la méthode numérique employée, bien qu'exacte, était particulièrement instable. Cette instabilité pose un certain nombre de problèmes lors de l'utilisation, en fournissant souvent des résultats aberrants.

3.2.1 Ajustement de l'erreur par la méthode des moindres carrés

On écrit le système précédent sous la forme $XC = Y$ et on recherche le vecteur colonne B, approximation de C tel que :

$$Y = XB + E$$

E représente la différence entre les prévisions et les données expérimentales. C'est le vecteur erreur ou encore vecteur résidu.

L'erreur commise en prenant B comme approximation de C est égale à la somme des carrés des différences entre les données expérimentales et les prévisions, c'est à dire E^TE. Afin de minimiser cette erreur, on la différencie par rapport à B puis on égalise à zéro :

$$\begin{aligned} E^TE &= (Y - XB)^T(Y - XB) \\ &= Y^TY - B^TX^TY - Y^TXB + B^TX^TXB \\ &= Y^TY - 2B^TX^TY + B^TX^TXB \end{aligned}$$

$$\frac{d}{dB}E^TE = -2X^TY + 2X^TXB$$

$$\frac{d}{dB}E^TE = 0 \Leftrightarrow B = (X^TX)^{-1}X^TY \qquad [33]$$

On obtient ainsi la matrice B qui approche au mieux le système initial $XC = Y$.

Le principal intérêt de cette méthode est d'être stable, de prendre en compte les douze équations du système décrites au paragraphe précédent et surtout de permettre l'ajout de points supplémentaires (en nombre quelconque) pour améliorer la qualité et la fiabilité du résultat. La programmation de cet algorithme est élémentaire, et malgré une apparente débauche de moyens, il reste très efficace dans le cas qui nous intéresse. Bien entendu, si il était nécessaire de manipuler une cinquantaine de points (X^TX serait une matrice 100x100) il faudrait tirer partie des particularités de la matrice pour calculer efficassement son inverse (X^TX est une matrice symétrique).

3.3 Intersection de trois tores

Dans les deux paragraphes précédent, nous nous sommes intéressés à la résolution numérique du problème du positionnement de l'observateur. Rare sont les auteurs qui se sont penché sur une solution géométrique à ce problème. Dans [HORA 89] on trouve une solution au positionnement de la caméra en utilisant seulement quatre points. L'auteur s'appuie sur la géométrie des droites définies par ces points. Nul n'a semble-t-il songé à se demander quelles étaient les configurations géométriques d'où pouvait être vu un segment de l'espace.

Dans un espace à deux dimensions, il existe une propriété bien connue dérivée de la définition des arcs de cercle :

- Arc capable : un arc de cercle AB est défini comme l'ensemble des points M du cercle tel que l'angle α des vecteurs $\boldsymbol{MA}$ et $\boldsymbol{MB}$ soit constant : c'est l'arc relatif à cet angle. L'autre arc relatif à cet angle est capable de l'angle $\pi - \alpha$. Les angles sont dits inscrits en M.
- A des angles inscrits (ou au centre) égaux correspondent des cordes égales et des arcs égaux ; et réciproquement.

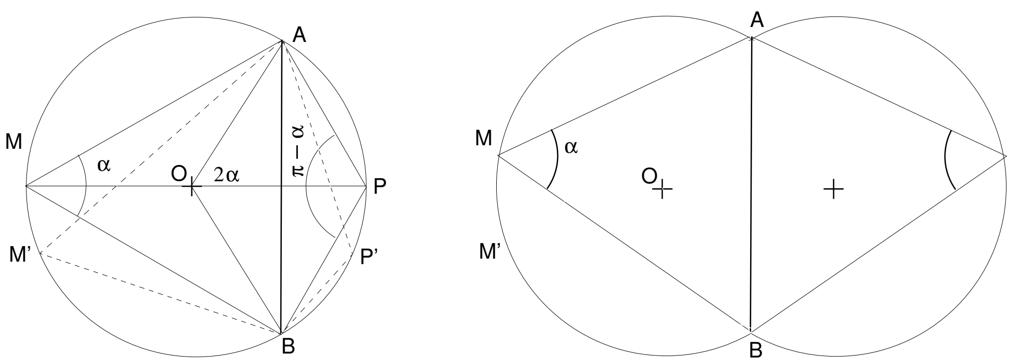


figure 42
Angles et cercles dans le plan

- L'ensemble des points du plan sous lesquels on voit un segment donné sous un angle donné est un arc de cercle (et son symétrique par rapport au segment).

Dans le plan, si l'on dispose de deux segments $[A, B]$ et $[C, D]$ ainsi que des l'angles α_{AB} et α_{CD} suivant lesquels ces segments sont vus. Il est alors possible de trouver les lieux géométriques où toutes les conditions précédentes sont remplies (figure 43).

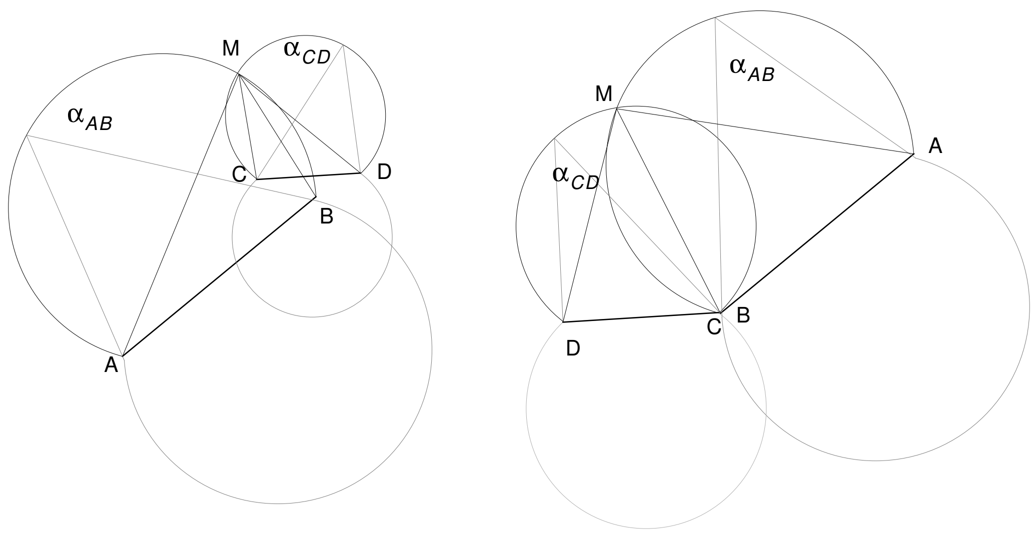


figure 43
Points du plan d'où sont vus deux segments

On peut remarquer sur la figure précédente que le problème a plusieurs solutions (jusqu'à quatre) et que si les deux segments ont une extrémité commune, le point de jonction des deux segments est une solution (double) triviale du problèmes, il ne reste donc plus que deux solutions intéressantes de part et d'autre des segments.

L'extension de ce principe à un espace à trois dimensions est évident. L'ensemble des points de l'espace sous lequel on voit un segment sous un angle donné est un tore dont l'axe est ce segment (il suffit de faire tourner l'arc (A, B) autour du segment $[A, B]$).

De cette manière, si on dispose de trois segments connus à la fois sur une photographie et dans le monde, le calcul du centre de projection (la position de l'observateur) se résume simplement à l'intersection des trois tores engendrés par les trois segments. Ces trois segments peuvent avoir des extrémités communes et former un triangle, dans ce cas, seuls trois points de l'espace suffisent pour retrouver la position de l'observateur.

3.3.1 Equations des tores

L'idée exploitée pour déterminer la position du centre de projection est de dire que c'est le point d'où on voit les divers segments $[A, B]$ sélectionnés sous les divers angles θ mesurés. L'équation des tores peut s'écrire sous la forme :

$$\|\boldsymbol{MA} \wedge \boldsymbol{MB}\| = \|\boldsymbol{MA}\| \times \|\boldsymbol{MB}\| \times \sin\theta$$

ou encore :

$$\boldsymbol{MA} \bullet \boldsymbol{MB} = \|\boldsymbol{MA}\| \times \|\boldsymbol{MB}\| \times \cos\theta$$

Nous choisirons la première solution en élevant toutefois les deux membres de l'égalité au carré. De plus comme :

$$\boldsymbol{MA} \wedge \boldsymbol{MB} = \boldsymbol{MA} \wedge \boldsymbol{AB}$$

l'équation finalement utilisée pour décrire un tore est :

$$\|\boldsymbol{MA} \wedge \boldsymbol{AB}\|^2 = \|\boldsymbol{MA}\|^2 \times \|\boldsymbol{MB}\|^2 \times \sin^2\theta$$

$$\|\boldsymbol{MA} \wedge \boldsymbol{AB}\|^2 - \|\boldsymbol{MA}\|^2 \times \|\boldsymbol{MB}\|^2 \times \sin^2\theta = 0 \qquad [\ 34\ ]$$

Remarque : Dans tous les cas, il est nécessaire d'utiliser un repère orthonormé. Si tel n'était pas le cas, les calculs d'angles obtenus grâce au produit scalaire ou au produit vectoriel seraient faux. Les mesures réelles de tous les segments choisis doivent être à la même échelle.

3.3.2 Newton

Le système d'équations non linéaire engendré par les trois tores décrits par l'équation [34] est facilement résolu si l'on utilise la méthode de Newton. L'inconnue de ce système est le point $M(x, y, z)$. Chaque équation de tore est de degré maximum égal à 4. Ce système admet donc potentiellement 64 solutions.

L'algorithme de Newton est l'un des plus efficaces en termes de rapidité et de stabilité parmi toutes les méthodes numériques de détermination de zéros de fonctions. Ainsi, si l'on a :

$$F = \begin{bmatrix} F_1 \\ F_2 \\ F_3 \end{bmatrix}$$

La méthode de Newton consiste à dire que le zéro de la fonction F recherché est la limite (si elle existe) de la suite définie par :

$$X_{n+1} = X_n - \frac{F(X_n)}{F'(X_n)}$$

Ce qui se traduit matriciellement de la façon suivante :

$$\begin{bmatrix} x_{n+1} \\ y_{n+1} \\ z_{n+1} \end{bmatrix} = \begin{bmatrix} x_n \\ y_n \\ z_n \end{bmatrix} - \begin{bmatrix} \frac{\partial}{\partial x}F_1(x_n, y_n, z_n) & \frac{\partial}{\partial y}F_1(x_n, y_n, z_n) & \frac{\partial}{\partial z}F_1(x_n, y_n, z_n) \\ \frac{\partial}{\partial x}F_2(x_n, y_n, z_n) & \frac{\partial}{\partial y}F_2(x_n, y_n, z_n) & \frac{\partial}{\partial z}F_2(x_n, y_n, z_n) \\ \frac{\partial}{\partial x}F_3(x_n, y_n, z_n) & \frac{\partial}{\partial y}F_3(x_n, y_n, z_n) & \frac{\partial}{\partial z}F_3(x_n, y_n, z_n) \end{bmatrix}^{-1} \times \begin{bmatrix} F_1(x_n, y_n, z_n) \\ F_2(x_n, y_n, z_n) \\ F_3(x_n, y_n, z_n) \end{bmatrix}$$

x_n, y_n et z_n sont les coordonnées des points de l'espace convergeant vers le centre optique. F_i étant l'équation du tore numéro i.

Les valeurs x_0, y_0 et z_0 peuvent être choisies arbitrairement si la fonction F est convexe et n'admet qu'un seul zéro. Sinon, la suite convergera vers un de ces zéros, si F est convexe. En général, la suite converge vers le zéro “le plus proche” de la valeur initiale de la suite. Si F n'est pas convexe sur son domaine de définition, la convergence n'est pas garantie (sauf localement). Néanmoins dans le cas des tores, la fonction F est convexe localement autour de chacun de ses zéros, ce qui est suffisant pourvu que x_0, y_0 et z_0 soient suffisamment proches du zéro cherché.

Les erreurs d'arrondis ne sont pas cumulées par la méthode de Newton à chaque itération et il est amusant de constater qu'elles sont fort utiles. Il existe des points de l'espace tels que si U_0 est l'un d'eux, la suite ne converge pas mais reste prisonnière d'un certain nombre de ces valeurs. La suite est cyclique. Mais grâce aux erreurs d'arrondis, la suite parvient en pratique à s'échapper de ces valeurs dangereuses, ce qui permet d'assurer le bon fonctionnement de la méthode.

3.3.3 Calcul du jacobien

Le segment $[A_i, B_i]$ et l'angle θ_i étant fixés, l'équation du tore d'indice i s'écrit :

$$F_i(M) = \|\boldsymbol{MA_i} \wedge \boldsymbol{AB_i}\|^2 - \|\boldsymbol{MA_i}\|^2 \times \|\boldsymbol{MB_i}\|^2 \times \sin^2\theta_i = 0$$

En outre on a le droit de dériver les produits scalaires et vectoriels de la même façon que les produits “normaux” de réels.

Ainsi, A et B étant fixés, en dérivant par rapport à $M(x, y, z)$, on obtient :

$$\frac{d}{dM}\|\boldsymbol{MA}\|^2 = \frac{d}{dM}(\boldsymbol{MA} \bullet \boldsymbol{MA}) = 2 \times \boldsymbol{MA} \bullet \frac{d}{dM}\boldsymbol{MA} \qquad [\,35\,]$$

$$\text{et } \frac{d}{dM}(\boldsymbol{MA} \wedge \boldsymbol{AB}) = \left(\frac{d}{dM}(\boldsymbol{MA}) \wedge \boldsymbol{AB}\right) + \left(\boldsymbol{MA} \wedge \frac{d}{dM}(\boldsymbol{AB})\right) = \frac{d}{dM}(\boldsymbol{MA}) \wedge \boldsymbol{AB} \qquad [\,36\,]$$

La dérivée par rapport à M de F en M, $\frac{d}{dM}F(M)$ est donc égale à :

$$\frac{d}{dM}F(M) = 2\,(\boldsymbol{MA} \wedge \boldsymbol{AB}) \bullet \left(\frac{d}{dM}(\boldsymbol{MA}) \wedge \boldsymbol{AB}\right) - 2\sin^2\theta\,(\boldsymbol{MA} \bullet \boldsymbol{MB})\,(\boldsymbol{MA} + \boldsymbol{MB}) \bullet \frac{d}{dM}(\boldsymbol{MA})$$

Bien sûr, on a :

$$\frac{\partial}{\partial x}(\boldsymbol{MA}) = \begin{bmatrix} -1 \\ 0 \\ 0 \end{bmatrix},\ \frac{\partial}{\partial y}(\boldsymbol{MA}) = \begin{bmatrix} 0 \\ -1 \\ 0 \end{bmatrix},\ \frac{\partial}{\partial z}(\boldsymbol{MA}) = \begin{bmatrix} 0 \\ 0 \\ -1 \end{bmatrix}, \text{ soit } \frac{\partial}{\partial M}(\boldsymbol{MA}) = \begin{bmatrix} -1 \\ -1 \\ -1 \end{bmatrix}$$

3.3.4 Calcul de l'angle sous lequel on voit un segment

Un minimum de connaissances des caractéristiques de l'appareil photo est nécessaire. Tout le principe repose sur l'hypothèse que la photographie est la projection perspective (avec comme foyer le centre optique de l'appareil) de la scène sur le “film”. Mais on a besoin des données suivantes :

La distance focale : distance entre le film et le centre optique.

L'angle de vue : angle entre la direction de visée et le bord droit de la photo.

La taille en pixels de la photographie.

Fort de ces renseignements, il est maintenant possible de calculer l'angle sous lequel on voit un segment $[A, B]$.

Soient $A(x_a, y_a, z_a)$ et $B(x_b, y_b, z_b)$ deux points dans le repère de l'espace. Ces points se projettent en $A'(x'_a, y'_a, z_0)$ et $B'(x'_b, y'_b, z_0)$ dans le repère de l'appareil (z_0 est la distance focale). On passe du repère du monde au repère de l'appareil photo par des rotations et des translations. Toutes ces transformations conservent les angles.

Si on note F le foyer de la projection (centre optique) :

$$\widehat{(FA, FB)} = \widehat{(FA', FB')} = \operatorname{acos}\left(\frac{\boldsymbol{FA'} \bullet \boldsymbol{FB'}}{\|\boldsymbol{FA'}\| \times \|\boldsymbol{FB'}\|}\right) \qquad [\,37\,]$$

Cependant, on ne connaît directement que les coordonnées x_a, y_a, x_b et y_b du segment $[A, B]$ exprimées en pixels. Les relations suivantes permettent de compléter les informations dont nous devons disposer :

$$\tan\theta_x = \frac{L_x}{2z_o}, \tan\theta_y = \frac{L_y}{2z_o}$$

avec :

z_o = Distance focale.

L_x = Largeur de la photographie exprimée en pixels.

L_y = Hauteur de la photographie exprimée en pixels.

θ_x = Angle entre la direction de visée et le bord droit (gauche) de la photographie.

θ_y = Angle entre la direction de visée et le bord haut (bas) de la photographie.

Toutefois, si la numérisation de la photographie ne se faisait pas “sans déformation” (si les pixels n’étaient pas carrés mais rectangulaires comme cela se produit sur certaines cartes d’acquisition vidéo), il serait nécessaire d’effectuer une affinité le long de l’un des 2 axes de coordonnées.

C’est dans cette phase que les erreurs de mesure peuvent influer. En effet, la projection d’un point peut tomber entre 2 (ou 4) pixels. Mais rien ne nous permet, en regardant la photographie, de dire que ce point n’est pas exactement à sa place. Rien ne permet de distinguer les points qui sont tombés juste au milieu du pixel de ceux qui sont tombés un tout petit peu à côté, mais suffisamment près pour ne pas être représentés sur un pixel voisin, ni surtout de dire de combien il en est tombé à côté.

3.3.5 Calcul de la solution

Nous avons vu dans les paragraphes précédents que le résultat de l’intersection de trois tores pouvait admettre jusqu’à 64 solutions. L’algorithme de Newton ne fournissant qu’une solution, il est donc assez peu probable d’obtenir le centre de projection directement. Des algorithmes de résolution numérique plus sophistiqués tel que celui de la bissection fournissent toutes les solutions réelles du système, mais il reste à choisir celle qui nous intéresse.

Une méthode simple permet de supprimer les points d’intersections qui appartiennent aux tores engendrés par les arcs capables d’angle $\pi - \alpha$ et qui ne font pas parties des solutions géométriques du système. Pour chaque couple angle/segment $(\alpha_i, [A_i, B_i])$, si $\alpha_i \geq \pi/2$ les solutions possibles appartiennent à la sphère S_i de diamètre $[A_i, B_i]$ et si $\alpha_i < \pi/2$, les solutions n’appartiennent pas à la sphère de diamètre $[A_i, B_i]$. Il faut remarquer l’intersection entre le tore et la sphère construite à partir de $[A_i, B_i]$ est réduite aux points A_i et B_i et que ces points ne peuvent pas être des solutions géométriques correctes.

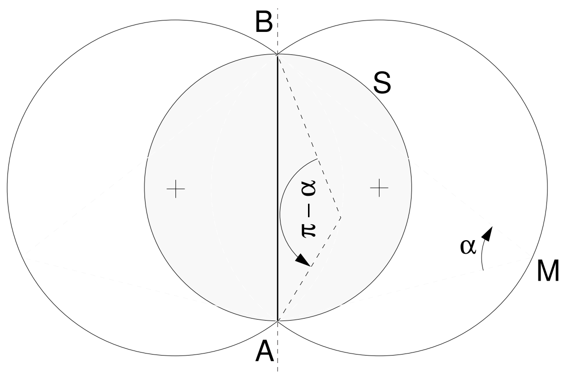


figure 44
Validité des solutions géométriques

Pour lever l'ambiguïté sur la position de l'observateur, une méthode consiste à calculer l'intersection d'un certain nombre de triplets de tores. La sélection de N points sur la photographie définit N.(N-1) / 2 tores différents. La résolution de tous les systèmes permet de choisir de manière plus fiable la position de l'observateur. Ce type d'approche permet aussi de limiter les effets des erreurs de mesure dues à la taille non nulle des pixels qui forment la photographie.

3.3.6 Evaluation des erreurs

Il était intéressant de se faire une idée quant à la précision des résultats donnés par l'algorithme de Newton en fonction des divers paramètres, tels que la distance de l'appareil par rapport aux objets, l'angle sous lequel on voit cet objet, la focale de l'appareil photo ou de la caméra.

On construit un cube orienté de manière quelconque dans l'espace. Pour ce faire, on applique des homothéties, des translations et des rotations à un cube "canonique" (de coté 1, dont un sommet est l'origine du repère et les arêtes issues de ce sommet s'appuient sur les axes de coordonnées). Les paramètres fournits par l'opérateur sont la longueur des arêtes (ce qui définit l'homothétie à appliquer), les angles dont on souhaite faire tourner le cube autour des trois axes de coordonnées, ainsi que la translation que l'on souhaite faire encore subir au cube. Le cube final est alors construit. A partir d'une position du foyer (position de l'observateur) entrée au clavier, et de la distance focale que l'on souhaite simuler, le cube est projeté sur un plan infini de normale (Oz) et à une distance du foyer égale à la "distance focale" choisie. Une procédure de perturbations modifie les coordonnées des points projetés. Le programme effectue alors la recherche de la solution non sans avoir demandé un vecteur de départ (une estimation de la solution pour initialiser l'algorithme de Newton) et les sommets à partir desquels on souhaite travailler. Le programme affiche alors la solution trouvée, la solution exacte (fournie au départ par l'utilisateur), et la distance entre ces deux points.

L'expérience montre que lorsque le programme trouve une solution celle ci n'est jamais à plus de 10 cm de la solution réelle. Et en général, elle est même à une distance inférieure à 3 cm.

- Comme on pouvait s'y attendre, la précision augmente au fur et à mesure que le foyer s'approche du cube (les écarts entre le milieu du pixel et le point où les sommets ont réellement été projetés deviennent négligeables). De même, plus le cube est gros et meilleure est la précision.
- L'influence de la distance focale est quant à elle beaucoup plus complexe. Dans la plupart des cas toutefois, augmenter la distance focale ("'zoomer") donne de meilleurs résultats.
- Comme on pouvait s'y attendre le résultats sont meilleurs lorsque le cube est plutôt de face que lorsque l'on choisit des sommets définissant des plans faisant un angle proche de 90 degrés de la direction de visée. Il vaut mieux travailler sur un cube au centre de la photo que loin sur le coté.

3.3.7 Conclusion

Cette méthode permet de retrouver de manière géométrique claire et précise la position du centre de projection avec un minimum de trois points. Nous n'avons aucune contrainte particulière sur le choix des points, ceux-ci peuvent même appartenir à un plan parallèle au plan de projection. Les autres paramètres de la prise de vue (orientation de la caméra) se déterminent facilement une fois que l'on connaît la position de l'observateur.

Le fait que cette méthode ne soit pas limitée à l'utilisation de trois tores permet d'obtenir des solutions robustes.

Contraintes 4

Nous avons vu au § 2 qu'il était indispensable de lever l'indétermination en Z des objets pour être capable de les positionner et de les dimensionner correctement. Dans le cas où l'on ne manipule qu'un seul objet, celui-ci n'est connu qu'à une affinité près si l'on ne dispose pas de points de repère fixes. Mais lorsqu'il est nécessaire de positionner précisément des objets supplémentaires (des détails) sur un volume général, on doit à tout prix assurer la cohérence de la base de données (l'arbre CSG).

4.1 Assurer un point de contact pour lever l'indétermination de Z

Le simple fait de fixer la position d'un objet de la photographie (cet objet doit bien entendu être connu) permet de déterminer de manière simple la position dans l'espace de tout point de cet objet. Si nous examinons le cas d'un plan, l'association des coordonnées dans le monde et sur la photographie de points appartenant à ce plan nous permet de trouver la position dans le monde de tous les points de l'image appartenant à ce plan. Une méthode classique consiste à utiliser les invariants projectifs (birapport), mais dans notre cas, le fait d'avoir choisi une matrice de projection perspective inversible nous permet, par simple produit par la matrice inverse de trouver les coordonnées de ce point dans l'espace. Il nous est alors facile de positionner correctement dans

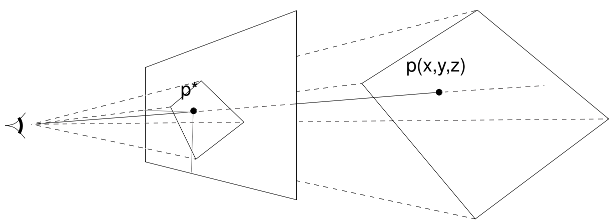


figure 45
Lever l'indétermination en Z

l'espace un objet quelconque dont on assure que, au moins un point de cet objet est en contact avec un plan déjà positionné. Par extension, tout nouvel objet ainsi placé dans l'espace peut à son tour servir de support à d'autres éléments. Un cube ainsi positionné aura une projection unique sur l'image (sa projection ne sera plus à une affinité près) et si sa projection se superpose exactement à sa représentation sur l'image, il est aux bonnes dimensions et à la bonne place. Pour que les relations (contacts) entre les objets puissent être préservées et vérifiées, nous les manipulerons comme des contraintes.

4.2 Entre objets

La mise en place et le dimensionnement des objets sur l'image supposent que l'on puisse leurs appliquer des transformations. Mais ces opérations ne doivent en aucun cas aller à l'encontre des contraintes qui unissent les objets. Nous allons définir des restrictions sur les transformations affines usuelles (translations, affinités et rotations) dans le cas où elles doivent s'appliquer à des objets contraints. Les solides que nous manipulons sont représentés par leurs frontières pour simplifier les opérations. Tous les objets sont donc des polyèdres à faces planes. Chaque face est délimitée par une liste de segments décrivant son contour. Les segments sont la portion de droite comprise entre leurs deux extrémités. Les contraintes de positionnement des objets telles que nous allons les définir assurent que les représentations par frontière des solides sont correctement positionnées, mais n'assurent pas que cette propriété est vérifiée dans le cas général pour les solides correspondants de la description CSG des primitives. Si on pose une sphère sur un plan, nous aurons un seul point de contact, alors que si l'on fait de même avec sa représentation polyèdrique on peut obtenir un point, une arête ou une face de contact.

Nous avons défini trois niveaux de contraintes entre deux solides polyèdriques :

Soit F_A une face de l'objet contraignant A et B un objet contraint par F_A.

- Contraintes de faces: B a une face F_B contrainte à rester dans le plan de F_A.
- Contraintes d'arêtes : B a un segment S_B contraint à rester dans le plan de F_A.
- Contraintes de sommets : B a un sommet P_B contraint à rester dans le plan de F_A.

Nous devons limiter les transformations affines standard applicables aux objets contraints de telle sorte que les contraintes restent satisfaites. Ces limitations sur les transformations dépendent du type de contrainte existant entre les objets. Il existe donc trois classes (figure 46 p. 134) :

- F_B doit rester posée sur F_A :

  Translations dans le plan défini par F_A.
  Rotations autour du vecteur normal à F_A.
  Affinités dans un repère construit à partir de F_A et du vecteur normal à F_A.

- S_B doit rester dans le plan de F_A :

  Translations dans le plan défini par F_A.
  Rotations autour du vecteur normal à F_A ou rotations autour de l'axe défini par S_B.
  Affinités dans un repère dont au moins un des axes est construit à partir du segment S_B.

- P_B doit rester dans le plan de F_A :

  Translations dans le plan défini par F_A.
  Rotations dont l'axe passe par P_B.
  Affinités dans un repère dont l'origine est le point P_B.

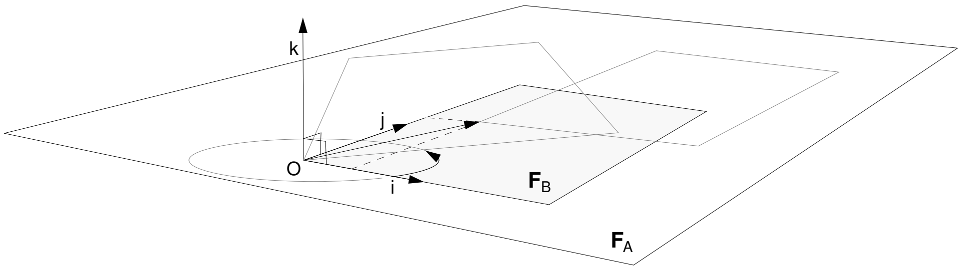
Une face étant constituée de segments et un segment de points, dans le cas où il est seulement nécessaire de préserver un point de contact entre les objets, les transformations que l'on peut appliquer aux objets contraints peuvent s'étendre de la manière suivante :

- A un objet contraint par une face on peut appliquer une transformation valide à un des segments de cette face. La contrainte de face est alors transformée en contrainte de segment.
- A un objet contraint par une face on peut appliquer une transformation valide à un des points des segments de cette face. La contrainte de face est alors transformée en contrainte de point.
- A un objet contraint par un segment on peut appliquer une transformation valide à un des points de ce segment. La contrainte de segment est alors transformée en contrainte de point.

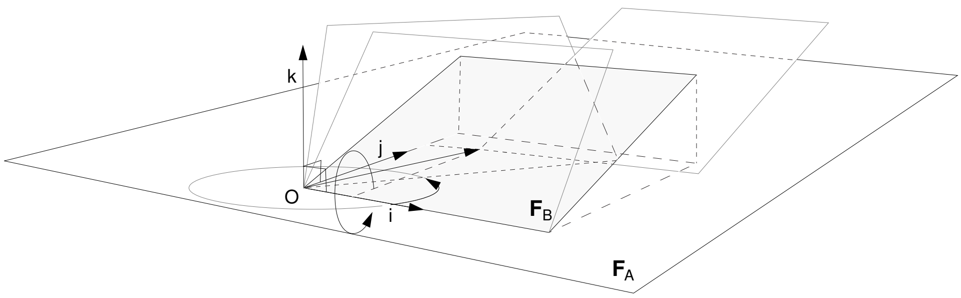
La construction de transformations préservant les contraintes entre les objets est relativement simple. Il suffit de disposer pour les objets contraignants, d'un repère associé à la face (F_A) et pour les objets contraints par F_A des informations sur le type de la contrainte ainsi que sur la face, l'arête ou le point qui est en jeu. Ces informations sont manipulées dans le repère associé à F_A et les transformations sont simplement construites en respectant les limitations que nous avons définies. Les transformations sont alors appliquées à l'objet contraint dans le repère local de l'objet contraignant (F_A).

4.3 Dans un arbre CSG

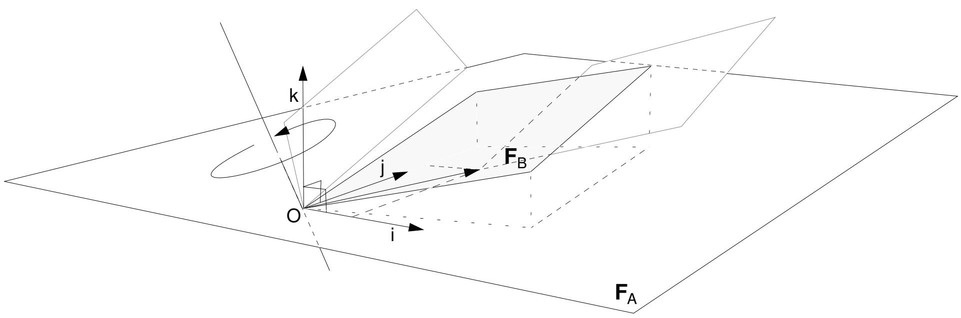
La description de la base de données que nous utilisons est réalisée à l'aide d'une représentation par un arbre de construction (CSG). Chaque feuille de l'arbre est une primitive géométrique et chaque nœud est une opération booléenne (union, intersection ou différence). Un objet est un nœud, racine d'un sous-arbre quelconque. Nous autorisons l'instauration de contraintes entre des objets de l'arbre (primitive ou sous-arbre) avec les conditions suivantes : à chaque objet contraint est associé un objet contrai-



Contrainte de faces



Contrainte de segments



Contrainte de points

figure 46
Transformation des objets contraints

gnant, la contrainte de position est définie par une face, un segment ou un point de l'objet contraint et une face de l'objet contraignant.

$$\texttt{InsererContrainte}(A, F_A, B, F_B, S_B, P_B)$$

Par souci de simplicité, nous interdisons par construction l'installation d'arcs de contraintes transitifs (arc *b* figure 47) et de circuits (arc *a* figure 47). La représentation des contraintes est strictement limitée à un arbre. Cette limitation ne serait pas envisageable dans le cadre d'un système de modélisation général, mais pour notre application, l'utilisation de ce type de contrainte est suffisant.

Chaque objet contraignant est la racine d'un arbre (ou d'un sous-arbre) de contraintes. Les contraintes entre objets dans un arbre CSG telles que nous les utilisons permettent les configurations suivantes pour chaque objet contraignant :

- Plusieurs objets sont contraints par la même face d'un objet contraignant.
- Plusieurs objets sont contraints par le même objet contraignant, mais par des faces différentes.

Lorsque l'on applique une transformation à un nœud de l'arbre CSG, on doit s'assurer que cette transformation est valide si cet objet est contraint et que l'ensemble des contraintes qu'il supporte reste satisfait. L'incidence d'une transformation valide sur un objet est décrite de la manière suivante :

- Une transformation appliquée à un objet contraint ne modifie pas l'objet contraignant et, après transformation, la contrainte doit rester satisfaite.
- Une transformation appliquée à un objet contraint modifie les objets qui sont éventuellement contraints (directement ou indirectement) par cet objet et les contraintes doivent rester satisfaites.

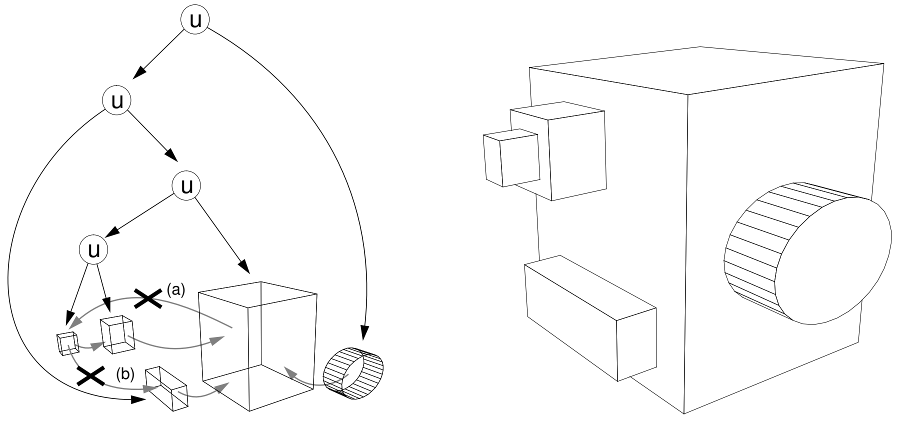


figure 47
Contraintes et arbre CSG

La transformation d'un objet contraignant, après avoir vérifié la validité de la transformation, se décompose en deux grandes étapes :

- Appliquer la transformation à l'objet (trivial).
- Calculer les transformations à appliquer aux objets contraints et appliquer ces transformations.

Dans le cas de la translation et de la rotation, on peut appliquer directement cette transformation aux objets contraints ; dans ce cas particulier, les contraintes associées aux objets restent satisfaites (figure 48). Les cas de l'affinité et des transformations affines quelconques sont plus complexes. Pour conserver des contraintes cohérentes, il est nécessaire de traduire ces transformations en une composition de translations et rotations à appliquer aux objets contraints.

Les transformations que nous manipulons sont banalisées (matrices 4×4 homogènes), et en général plusieurs transformations sont concaténées avant d'être appliquées aux objets. Une méthode simple permet de connaître la transformation à appliquer aux objets contraints dans le cas général. A chaque face des objets contraints servant de support à des contraintes est associé un repère orthonormé (trois vecteurs et un point origine). La transformation à appliquer aux objets contraints est la matrice qui permet de passer du repère associé à la face contraignante au repère construit à partir de la face transformée.

Chacune des faces d'un objet pouvant supporter plusieurs objets contraints, la mise à jour de tous les objets contraints s'effectue de la manière suivante :

```
Pour toutes les faces de l'objet à transformer {
   calculer la transformation à appliquer aux objets contraints.
   Pour tous les objets contraints par cette face
      appliquer récursivement la transformation à l'objet.
}
Transformer l'objet.
```

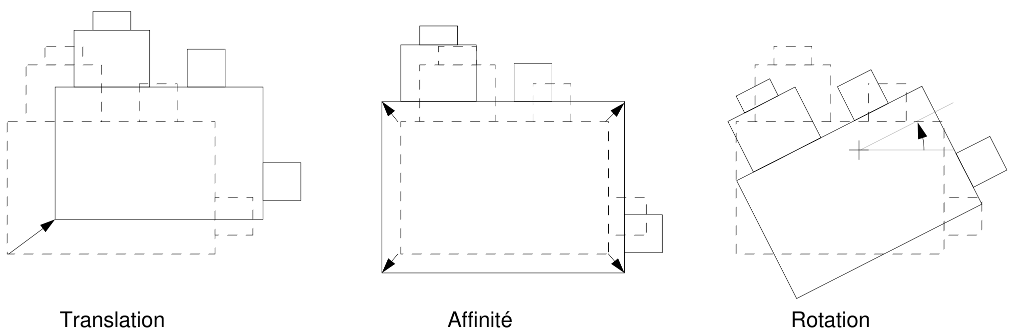


figure 48
Propagation des transformations sur les objets contraints

L'extension de la méthode que nous venons de présenter à des objets contraints par plusieurs objets (l'objet est accessible depuis deux arbres de contraintes distincts) est envisageable mais complexe. La mise en œuvre de ce type de contraintes suppose que l'on soit capable de déterminer des transformations préservant les contraintes.

La modélisation sous contrainte est un domaine en soi, la solution que nous proposons dans ce paragraphe ne fait que l'effleurer. Elle est par contre une solution raisonnable et adaptée au problème que nous avons à traiter.

4.4 Extensions aux surfaces

La gestion des contraintes telle que nous venons de la définir s'applique aux faces planes d'objets polyèdriques. Cette limitation engendre un certain nombre d'approximations qui peuvent être gênantes dans certains cas. La notion de face peut être étendue à une notion de surface. Les transformations affines usuelles n'ont plus vraiment de sens : que veut dire translater un objet sur une surface quelconque ? Une solution simple consisterait à associer à chaque objet une représentation paramétrique et à appliquer les transformations dans l'espace défini par ces représentations. Certains résultats risqueraient d'être surprenants.

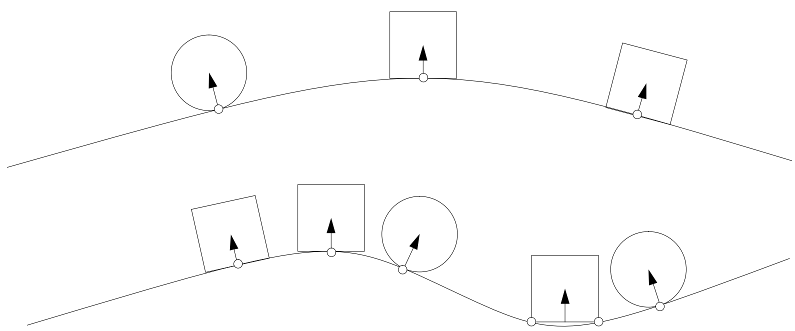


figure 49
Contraintes de contact sur une surface

Ce type de solution est facilement envisageable pour des contraintes ponctuelles. Mais l'utilisation de solides rend complexe un positionnement cohérent strict. Le simple exemple d'un carré posé sur une courbe quelconque suffit pour s'en convaincre. Sur la figure 49 le carré peut avoir un, deux points de contacts avec la courbe, bien plus si la courbe est très tourmentée et un segment si c'est une droite (ou si certaines parties de la courbe sont des droites). Un certain nombre d'auteurs traitent de ce sujet (surtout dans le domaine de l'animation), le lecteur pourra se reporter à [FERT 90], [EMME 91] et [ROSS 86].

Mise en œuvre de la méthode 5

5.1 Implantation

Cet outil s'appuie sur une description de la scène par arbre de construction (CSG : constructive solid geometry [PERO 88]) : la scène est décrite par une arborescence dont les noeuds sont des opérateurs booléens (union, intersection, différence) et dont les feuilles sont des primitives unitaires de modélisation. Les primitives géométriques à notre disposition sont le cube, le cône, la sphère, le cylindre et le prisme. Pour éliminer toute ambiguïté sur le positionnement des objets dans la scène, chaque nouvelle primitive sera insérée avec au moins un point de contact avec un objet déjà en place dans la scène. Dans la pratique (voir la figure 50), les objets seront collés les uns aux autres (ils auront une face commune) soit en saillie, soit en creux. Ceci nous permettra de différencier la réunion de la différence lorsque l'on insèrera la nouvelle primitive dans l'arbre de construction. Les opérations géométriques que nous autorisons pour un objet sont :

- la translation dans le plan de la face d'appui de cet objet.
- les affinités et les rotations permettant de conserver au moins un point de contact avec la face d'appui de l'objet.

Grâce à ces transformations géométriques, nous pouvons ainsi positionner interactivement des primitives dans la scène, en faisant coïncider la projection de ces primitives avec les objets apparaissant sur la photographie (figure 50 p. 141).

5.2 Méthodologie applicative

Comme nous l'avons vu précédemment, notre méthode suppose que l'opérateur prenne à sa charge la construction de l'arbre CSG représentant les objets qu'il doit modéliser.

Pour ce faire il doit respecter un certain nombre d'étapes dans l'élaboration de son modèle.

1 - L'étape de recherche de la matrice de transformation entre le monde et la photographie va lui fournir un repère orthogonal direct qui servira de base à la construction de l'arbre CSG (ce repère nous fournit un point origine O(0,0,0,1) et trois plans orthogonaux : x=0, y=0 et z=0).

2 - La première primitive insérée par l'opérateur sera unitaire et positionnée à l'origine du repère. Cette étape franchie, nous allons pouvoir déplacer et déformer cette primitive de manière à ce que sa projection perspective coïncide avec un élément de l'objet à modéliser et pour servir de support aux éléments suivants.

3 - Les deux étapes précédentes étant achevées, l'opérateur peut positionner de nouveaux éléments dans la scène. Les objets étant contraints par programme à conserver au moins un point de contact avec un élément déjà en place, il suffit de déplacer (translation, rotation) ou de déformer (affinité) interactivement chaque nouvelle primitive pour la positionner convenablement dans l'espace. On peut remarquer que chaque nouvel élément peut servir de support à d'autres primitives.

5.2.1 Avantages de la méthode

Ce principe de calcul de la matrice de projection permet de décrire un objet à partir de N photographies $(N \geq 2)$, même prises avec des objectifs de focales différentes. Pour cela (voir la figure 51), il suffit de placer correctement un objet de la scène à partir de la photographie $(N-1)$. Ses dimensions étant alors connues dans le repère $(N-1)$, il suffit de retrouver ce même objet sur la photographie (N), de rechercher à nouveau la matrice de projection avec, comme paramètres, les dimensions de l'objet sélectionné sur la photographie $(N-1)$ et sa représentation sur la photographie (N).

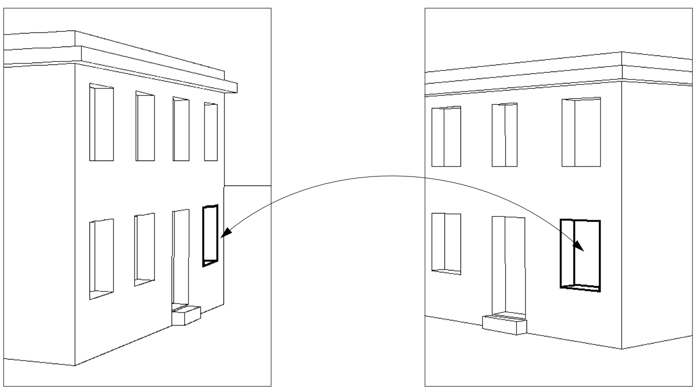


figure 51
Reconstruction à partir de photographies multiples

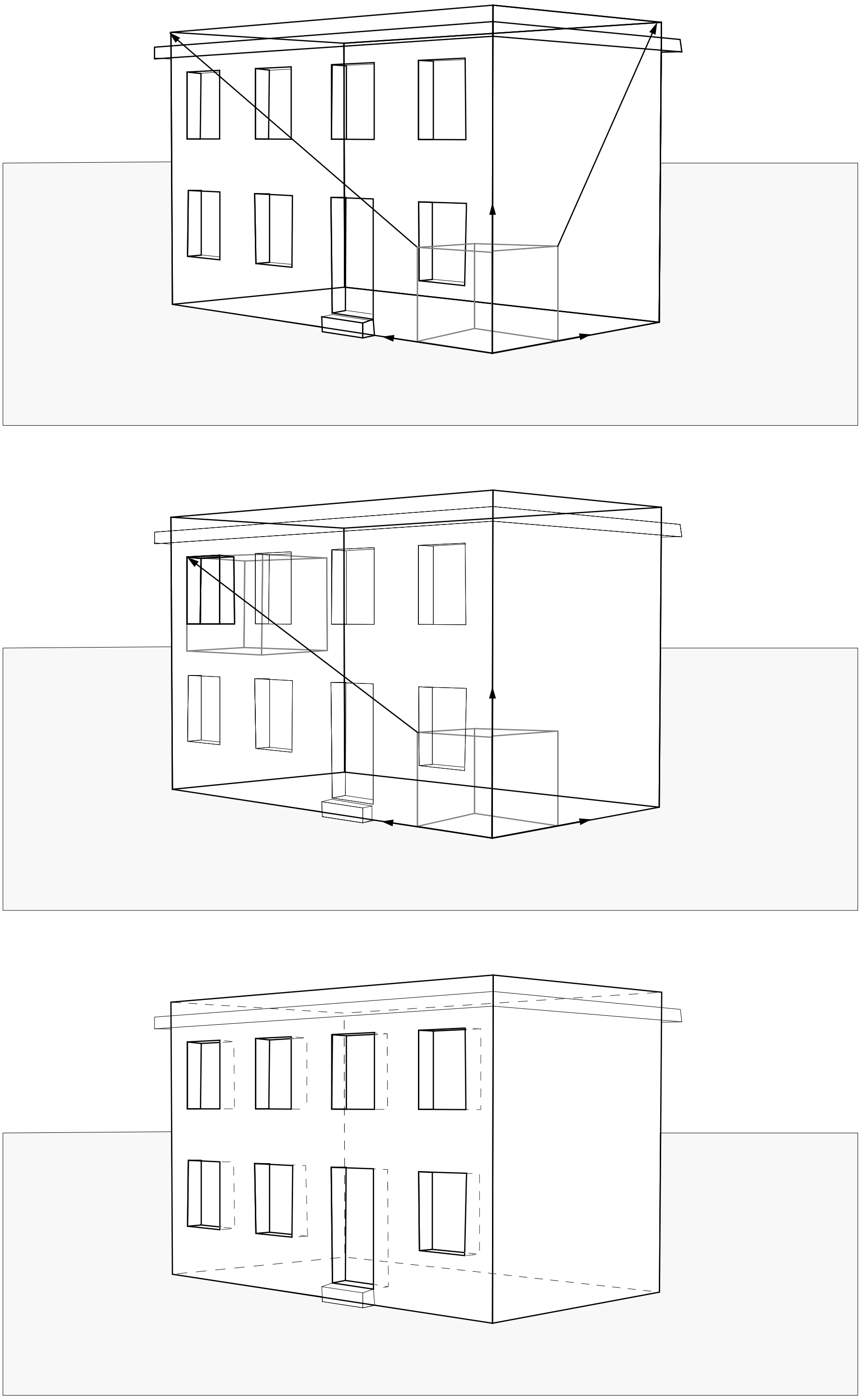


figure 50
Reconstructiuon interactive d'un bâtiment

On obtient alors un nouveau repère qui permet de continuer la construction de notre objet.

De cette manière, chaque élément ajouté à la scène est inséré dans l'arbre de construction. La définition d'attributs supplémentaires tels que faire un trou, conserver l'intersection de la primitive courante avec le reste de la scène,... permet la gestion des opérations booléennes du modèle (union, intersection et différence, l'union étant l'opération par défaut). Chaque nœud de l'arbre de construction peut être nommé et surtout réutilisé dans la suite de la modélisation. De cette manière, il devient concevable de fabriquer une représentation très précise d'un objet sans que cela soit fastidieux. Ainsi, nous pouvons modéliser de manière précise un détail (une fenêtre par exemple) à partir d'une photographie en gros plan, puis sur une vue plus générale, positionner cette fenêtre en chacun des endroits où elle devra se trouver.

Les bases de données créées de cette manière sont stockées sous forme d'un fichier au format de description de notre système de synthèse d'images tridimensionnelles “ILLUMINES” ([BEIG 89]), et sont de fait directement réutilisables pour la réalisation d'images de synthèse réalistes.

Lors de la saisie de chacun des objets de la scène, l'opérateur peut définir les propriétés des éléments qu'il met en place. Chacun des éléments comporte une liste d'attributs que l'on peut remplir de manière à conserver des informations qui simplifieront les traitements que l'on pourra faire subir à cette base de données. Ces attributs sont un complément d'informations non négligeable ; ils ne sont pas difficiles à mettre en œuvre et ils rentabilisent au mieux l'étape interactive de saisie.

Tout l'intérêt de cette méthode est de pouvoir s'affranchir des problèmes de reconstruction d'une scène à partir de plusieurs photographies, de faire abstraction des éléments partiellement masqués (faces cachées des balcons, parties masquées par des obstacles tels que des arbres, des véhicules…) et de s'appuyer sur l'expérience de l'opérateur pour la description des objets. On remarque que de cette manière et malgré une description incomplète d'un objet par les photographies, on est capable dans certaines conditions de le reconstruire complètement.

5.3 Applications industrielles

Nous allons présenter dans ce paragraphe les applications pour lesquelles nous avons développé cette méthode de reconstruction tridimensionnelle interactive.

Notre principal objectif était de modéliser des bâtiments, des ouvrages d'art, des monuments... de taille et de forme quelconques (de la maisonnette à la statue en passant par le viaduc …). La collecte des données permettant de reconstruire de tels ouvrages devait être simple à mettre en œuvre pour des gens qui ne seraient pas des spécialistes ni de l'informatique, ni de la photographie. Ces contraintes nous ont permis de nous orienter vers de la photographie de type magnétique pour la prise de vue, et vers l'interaction pour ce qui est de la reconstruction 3D. La photographie magnétique, malgré sa résolution assez réduite, nous permet d'automatiser de manière souple et rapide (plus de développement photographique) et pour un coût économique faible (moins de 20.000 Francs pour l'ensemble de la chaîne de saisie) la numérisation des photographies.

Dans des conditions d'utilisation courante, la description d'un bâtiment dans son ensemble est impossible sur une seule photographie : on est confronté presque systématiquement au problème de l'étroitesse des rues, à la présence d'obstacles (véhicules, arbres…). Notre méthode a donc l'avantage de répondre de manière simple à tous ces problèmes, l'opérateur prenant en charge la description des parties masquées du bâtiment, le logiciel l'aidant par contre à positionner les divers éléments constituant la scène par une carte d'attracteurs (sorte de grille dont les points attracteurs sont tirés de l'image numérisée par détection de contour) et lui fournissant les outils graphiques nécessaires à la reconstruction par vues multiples.

L'aspect interactif de notre méthode permet d'avoir divers degrés de précision lors de la modélisation et surtout de laisser cette précision de modélisation au choix de l'opérateur en fonction de l'utilisation future de sa base de données. De cette manière, on ne modélise pas d'objets inutiles qui pénaliseraient les traitements suivants. La saisie de la base de données étant interactive, l'opérateur peut à tout instant intervenir sur les différentes primitives qu'il manipule pour ajouter des informations supplémentaires à ces objets (ce cube représente une fenêtre, une porte ; il est en bois, de couleur rouge,...).

Notre méthode a fait l'objet d'une implantation logicielle sur station de travail Apollo DN 10000 et HP série 400. On trouvera au chapitre 7, un bref descriptif de la méthode opératoire accompagnée par des illustrations directement générées par notre outil aux étapes clés du processus de saisie. Notre outil fait l'objet actuellement d'une phase de test expérimental chez notre partenaire industriel.

Extensions 6

Nous venons de voir tout au long de cette partie de quelle manière il était possible de reconstruire un objet tridimensionnel à partir d’une ou plusieurs photographies. Le but principal de tout ce travail est de modéliser un objet en ne connaissant que des représentations photographiques, elle permet aussi de modéliser les objets environnants.

6.1 Incrustations vidéo

Une utilisation possible et intéressante de cette méthode est l'aide qu’elle peut apporter à l'incrustation d'images de synthèse dans un décor naturel (images fixes ou mobiles) ([NAKA 86],[NAKA 89] et [NAKA 91]).

Pour des raisons de simulation ou d’études d’impact, on peut vouloir incruster un bâtiment de synthèse dans son contexte urbain. Sur cet exemple, nous pouvons remarquer qu’il existe plusieurs problèmes à résoudre. Le premier est de retrouver la bonne transformation perspective à appliquer à notre modèle synthétique pour avoir une incrustation convenable. Le second est, dans le cas où notre bâtiment serait partiellement masqué par le contexte, d’être capable, de manière automatique, de ne conserver que les éléments réellement vus de notre modélisation dans la photographie. Enfin, par souci de réalisme, on peut vouloir calculer les reflets et les ombres portées produits par les constructions environnantes sur notre bâtiment.

L'outil que nous avons développé permet de répondre de manière satisfaisante à toutes ces questions. En effet, pour une photographie, nous sommes capables de retrouver la matrice de projection à appliquer à notre maquette synthétique pour l'incruster convenablement dans la scène. Quant aux problèmes de cohérence avec le contexte, il est simple de les éliminer en modélisant de manière adéquate (en général une modélisation grossière suffit) l'environnement immédiat du lieu d'implantation et de rajouter ces nouveaux éléments à la base de données qui décrit notre bâtiment. Ces nouveaux objets décrivant le contexte de l'implantation devront recevoir des attributs particuliers pour obtenir lors de la phase d'élimination des parties cachées un masque d'incrustation qui permettra de différencier les parties à incruster du reste de l'image. Pour cela, on peut utiliser une couleur particulière : du bleu vidéo ou un marqueur numérique quelconque. La modélisation du contexte que nous venons de réaliser va nous permettre de résoudre également les problèmes d'ensoleillement et d'ombres portées. A partir des données que nous possédons, nous pouvons trouver le vecteur de direction du soleil ; de manière naturelle, nous obtiendrons alors les ombres portées pour l'ensemble de la scène, celles du contexte sur notre maquette synthétique, mais aussi celles de notre maquette sur le contexte. Le masque d'incrustation est cette fois un peu plus sophistiqué puisqu'il faut tenir compte de l'ombrage du contexte lors du mélange des images. Pour ce faire, on peut utiliser une méthode du type alpha-buffer [DUFF 85] et [PORT 84].

6.1.1 Calcul de la direction d'éclairement

Une scène naturelle photographiée peut être éclairée globalement de deux façons différentes : de manière naturelle par le soleil ou par un éclairage artificiel.

Dans le cas d'un éclairage par le soleil, on peut considérer sans faire d'erreur flagrante, qu'il s'agit d'une source ponctuelle placée à l'infini. Nous avons à notre disposition deux méthodes pour calculer la direction de cet éclairage. La première méthode consiste à utiliser des techniques de type “héliodon”, ou à partir de la position (latitude et longitude) du lieu sur le globe terrestre et de la date (jour, heure, minute et seconde), un calcul astronomique simple permet de connaître les positions relatives terre-soleil (déclinaison et ascension droite de l'astre), donc la direction du soleil (cf. Annexe C). Une autre solution, ne dépendant que des informations disponibles sur la photographie, permet de découvrir facilement la direction du soleil. Sur la ou les photographies qui ont servi de base à la reconstruction du contexte de la scène à modéliser, il est en général possible de trouver des objets portant une ombre. Le vecteur soleil est alors directement déterminé par le point portant ombre et l'ombre de ce point. Bien entendu,

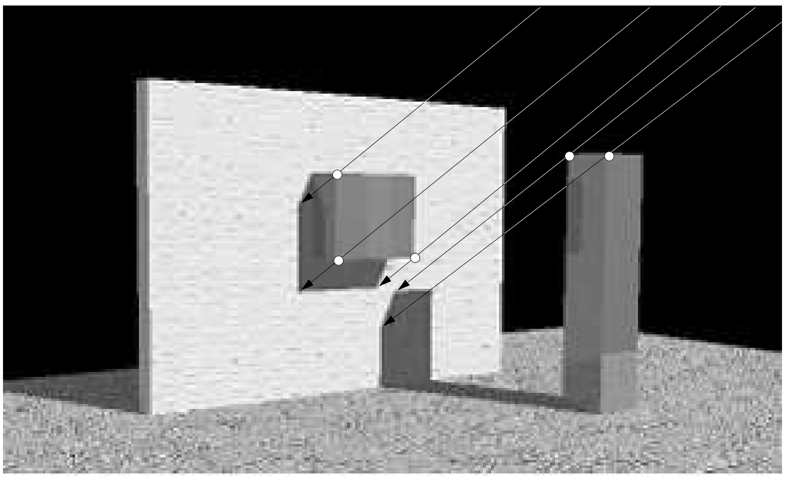


figure 52
Déterminer la direction du soleil

ce principe ne fonctionne que dans les cas où le soleil est suffisamment visible pour porter une ombre. Les jours très nuageux ou très brumeux, la lumière est trop diffuse, il est inutile de vouloir incruster un bâtiment avec des ombres franches, le réalisme en souffrirait. Il faut avoir recours à d'autres solutions comme des éclairages diffus ou des sources de lumière surfaciques de grandes dimensions.

La position des sources lumineuses dans le cas d'un éclairage artificiel n'est guère complexe à découvrir. Lors de la modélisation, certaines sources (lampes, lampadaires…) peuvent être visibles sur la photographie, leur position est alors explicite. Si les sources lumineuses ne peuvent être positionnées de cette façon, une solution d'un principe identique au positionnement du soleil peut être employée. Les sources lumineuses de type lampe (éclairage artificiel) peuvent être considérées comme ponctuelles dans le cas général et à une distance finie. Déterminer deux vecteurs (point portant ombre et l'ombre de ce point) permet d'estimer la position de la source. Les sources de lumière étant ponctuelles, les vecteurs sont dirigés vers cette source. Bien que les deux droites de l'espace ayant comme vecteurs directeurs les deux vecteurs précédents aient peu de chances de se couper, une solution approchée peut être trouvée (par le calcul de la distance entre les droites par exemple).

6.1.2 Textures et reflets

Lorsque l'on a reconstruit l'environnement, il est possible de coller sur toutes les faces qui sont orientées vers l'observateur[1] la portion d'image qui les représente. Cette portion d'image (un polygone quelconque en général) peut être associée à chaque facette et manipulée comme une texture plaquée.

Ces informations peuvent être utilisées pour visualiser la représentation synthétique du modèle sous un angle différent de celui de la photographie en plaquant les textures extraites de l'image. Les faces arrières (qui ne regardent pas l'observateur) de la représentation synthétique ne peuvent pas être texturées si l'on ne dispose que d'une seule image, et certains défauts peuvent apparaître avec le recouvrement des objets. Mais il est possible en général, à partir de plusieurs images, de trouver des portions de texture sans défauts. Peut-être, existe-t-il des méthodes automatiques pour résoudre ce problème.

Puisque nous sommes capables de fabriquer, à partir de plusieurs photographies, une représentation synthétique du modèle, il devient simple d'utiliser toutes ces informations pour le rendu d'une scène dans laquelle on incruste un objet purement de synthèse. S'il s'agit, par exemple d'un bâtiment, il sera possible de voir les reflets des bâtiments environnants dans les fenêtres en utilisant un simple lancé de rayons pour le rendu de la scène. La méthode peut même être étendue : voir le reflet du bâtiment de synthèse dans les fenêtres des ouvrages existants (présents sur la photographie). Pour obtenir un tel résultat, l'opérateur doit seulement donner un attribut particulier aux fe-

1. par convention, la normale à une face est tournée vers l'extérieur du solide.

nêtres des bâtiments de l'environnement pour que le processus de rendu puisse en tenir compte.

L'intérêt de cette démarche est de limiter le plus possible la modélisation du contexte. Le plaquage d'une texture permet de simuler rapidement des détails qui apporteront ou au moins participeront au réalisme global de l'image finale. Bien que plaquée sur un polygone plan, l'image d'une fenêtre (qui a du relief) est suffisante pour un reflet ou une vue éloignée, les problèmes liés au plaquage n'apparaissent que lorsque l'on s'approche trop de l'objet (le lecteur intéressé peut consulter [JAHA 91] et [BEIG 91] qui se sont intéressés au traitement des niveaux de détail en synthèse d'image).

6.1.3 Ombrage de la scène

Pour que l'image synthétique soit incrustée de manière correcte dans l'image naturelle, le seul souci géométrique n'est pas suffisant. La partie incrustée de l'image doit se fondre correctement en terme d'ombrage et d'éclairement. Nous avons vu dans le paragraphe précédent les diverses méthodes envisageables pour déterminer les positions ou les directions des divers éclairages. E. Nakamae & all. proposent une solution simple pour déterminer les coefficients de la fonction d'éclairement par une source lumineuse à l'infini (soleil). Leur solution s'applique seulement dans le cas d'objet strictement lambertiens observés dans un environnement diffus. Ils se limitent à l'étude de la fonction d'éclairement suivante :

$$I = I_a + I_d \cos\theta \qquad [\,38\,]$$

où i_d, $\cos\theta$ et I_a sont respectivement l'intensité du soleil, le cosinus de l'angle entre la normale à la face considérée et la direction du soleil et un coefficient d'éclairage ambiant. Pour que l'incrustation paraisse correcte le rapport entre I_a et I_d noté H, doit être calculé de manière à ce qu'il soit égal à celui de l'image naturelle. En choisissant deux murs apparaissant sur l'image naturelle, on calcule leur niveau d'intensité[1] moyen (par échantillonnage ou moyenne). H est alors déterminé de la manière suivante :

$$H = \frac{I_2 \cos\theta_1 - I_1 \cos\theta_2}{I_1 - I_2} \qquad (\cos\theta_{(1,2)} \geq 0)$$

I_1 et I_2 sont la luminance de chacun des murs. θ_1 et θ_2 sont les angles respectifs que font les normales des deux surfaces avec la direction du soleil. La valeur de H varie sensiblement avec le choix des faces (murs) qui permettent son calcul. Pour limiter le problème, un certain nombre de mesures sont effectuées, et une valeur moyenne servira de référence. On peut remarquer que le système d'équations composé à partir de l'équation [38] pour deux angles différents est simple à résoudre (système de deux équations à deux inconnues).

1. Cette quantité est proportionnelle à la luminance et à la réflectance

L'utilisation des textures extraites de l'image et plaquées sur la reconstruction de l'environnement permet le calcul automatique de H. En effet, pour un certain nombre de faces du contexte, nous disposons à la fois de leur orientation (normale à la face) et de leur texture (morceau d'image correspondant). Pour chaque face, il suffit de calculer une luminance moyenne et l'angle entre sa normale et la direction du soleil (qui est directement accessible). Le calcul de H peut alors se faire automatiquement comme moyenne des valeurs obtenues par les combinaisons des faces du contexte disposant des informations nécessaires.

Le modèle d'éclairement que nous venons de voir peut être étendu à de multiples sources lumineuses positionnées à l'infini comme le soleil, ou ponctuelles (éclairage artificiel). Pour traiter de manière correcte les paramètres associés à des sources de lumière ponctuelles, il est important de tenir compte de l'atténuation de l'éclairement avec la distance. L'équation devient de la forme suivante dans le cas d'une source de lumière ponctuelle [FOLE 90] :

$$I = I_a k_a + f_{att} I_p k_d \cos\theta$$

La fonction d'atténuation peut être choisie sous la forme $f_{att} = \frac{1}{d^2}$ où d est la distance du point traité à la source de lumière.

d et θ sont facilement déterminés (nous possédons toutes les informations géométriques nécessaires). Dans le cas d'une scène et d'un environnement particulièrement diffus l'approximation de E. Nakamae & all. ($k_a = 1$ et $k_d = 1$) est acceptable et rend le système linéaire et résolvable de la même manière dans le cas d'une source de lumière ponctuelle dont on connaît la position.

Le passage aux sources de lumière multiples ne pose que peu de problèmes en plus. Si on se restreint aux hypothèses précédentes, pour chaque source de lumière supplémentaire, il est nécessaire de fournir une équation de plus. Les sources de lumière étant ponctuelles (il est rare d'avoir plusieurs soleils), des luminances choisies sur la même face d'un objet permettent de construire les équations dont nous avons besoin (les équations ne seront pas liées entre elles, en chaque point, l'angle θ est différent).

$$I = I_a k_a + \sum_{i=1}^{m} f_{att_i} I_{pi} k_d \cos\theta_i$$

Le processus permettant de retrouver les paramètres de chaque source lumineuse peut aussi être traité de manière automatique. Des précautions supplémentaires sont à prendre pour le traitement des ombres portées. La multiplication des sources provoque la multiplication des ombres portées. Pour chaque point traité, il est nécessaire de déterminer quelles sont les sources qui participent à l'éclairage local. Ce problème est facilement résolu en vérifiant qu'aucun objet de la modélisation du contexte ne masque la source de lumière considérée (cf. calcul d'ombrage en lancé de rayons). Cette technique s'applique aussi dans le cas d'une source de lumière à l'infini.

L'extension de la recherche des paramètres d'éclairement à un modèle spéculaire est plus complexe si l'on ne dispose pas d'informations sur les paramètres de spécularité des objets que l'on manipule. L'équation du modèle d'éclairement de Phong [BUIT 75] est le suivant :

$$I = I_a k_a + \sum_{i=1}^{m} I_{pi}(k_d \cos\theta_i + k_s \cos^n \alpha_i)$$

L'apparition du cosinus à la puissance n rend délicate la résolution de tels systèmes (n variant entre 1 et 200 en pratique). Le cosinus de l'angle α à la puissance n (angle entre la direction du regard de l'observateur et la direction réfléchie de l'image de la source de lumière) décroît très rapidement pour les objets fortement spéculaires et le fait de ne pas tenir compte de ce phénomène risque d'engendrer des erreurs importantes sur les paramètres d'éclairement : soleil ou lampes trop fortes ou au contraire beaucoup trop faible.

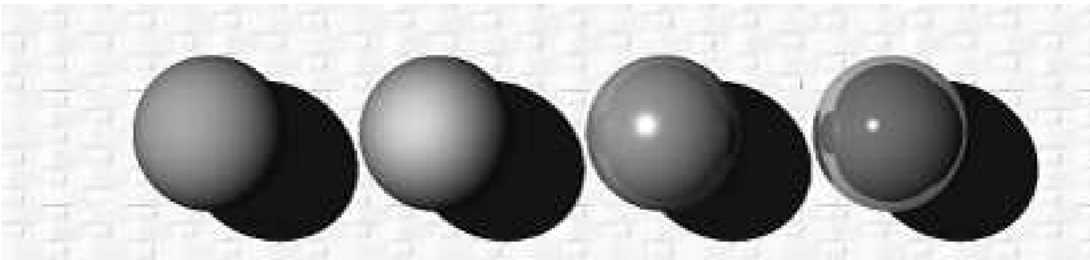


figure 53
Influence de la spécularité sur l'éclairement des objets

Les divers coefficients associés aux objets ne nous intéressent par particulièrement par eux-mêmes, puisque nous recherchons les caractéristiques de diverses sources lumineuses. Mais le fait de les ignorer peut amener à des incohérences lors de l'incrustation des objets synthétiques.

6.2 Animation

Notre méthode peut aussi s'appliquer efficacement à une séquence animée. La base de données représentant le contexte est construite une seule fois. Cette tâche, bien que lourde dans le cas d'environnement complexe, peut être assimilée à un prétraitement. Pour chaque image de la séquence que l'on veut réaliser, il nous suffit de déterminer la nouvelle matrice de projection associée à l'image, puis de ne refaire que les calculs d'élimination des parties cachées et d'ombrage comme nous l'avons vu précédemment, ceci sans avoir à reconstruire ni modifier la base de données. Le calcul automatique de la nouvelle matrice de projection est conditionné par la reconnaissance des points de référence. Il existe dans les outils de traitement d'images, un certain nombre de techniques qui résolvent ce problème (certaines permettent même de suivre des points quelconques choisis sur une image en temps réel). Les mouvements de caméra (travelings, panoramiques) peuvent faire disparaître du champ de la caméra certains, voire tous, points de références. Dans ces conditions, il est nécessaire de faire une remise à jour périodique des points (en ajouter, recaler des points qui s'éloignent…). Un article de Eltl, Muller-Seelich et Tabatabai [ERTL 91] met en évidence un problème d'aliassage temporel : l'incrustation synthétique semble bouger dans l'image naturelle lorsque l'on visionne la séquence. Ce phénomène est lié à la difficulté de calculer de manière stable la matrice de projection perspective associée à chaque image. Une erreur de l'ordre du pixel sur les coordonnées des points de référence influe significativement sur les coefficients de la matrice. Les auteurs de cet article proposent de filtrer les coordonnées des points de référence par une interpolation polynômiale ou une transformée de Fourier. Cette solution semble fournir d'excellents résultats : les mouvements de la caméra sont relativement souples dans la réalité.

Si la scène contient des objets mobiles, le problème est plus complexe. Un cas facilement résolu est celui où les objets mobiles n'interfèrent pas avec la partie synthétique de l'image, on les ignore simplement. Dans le cas contraire, pour chaque image, il est nécessaire de mettre à jour la position (et la modélisation) du mobile. Cette opération est réalisée manuellement si le mobile a un déplacement chaotique, et en partie automatiquement, s'il est possible de décrire une trajectoire interpolable à partir de position clefs (de nombreux outils de modélisation savent résoudre ce problème, une solution élégante permettant d'interpoler positions et orientations est proposée dans [HANO 93]). Nous atteignons ici les limites de notre méthode.

Bien que cette méthodologie ne soit pas entièrement automatique, le temps qu'elle fera gagner par rapport à des méthodes manuelles classiques sera important.

Résultats 7

Notre méthode de modélisation a fait l'objet d'une implantation logicielle complète sur station de travail APOLLO dn 10 000 et HP 733 s. L'ensemble des copies d'écrans présentées ici décrivent la méthode à suivre lors de la modélisation d'un bâtiment relativement conséquent.

1 Acquisition et numérisation de l'image.

Nous utilisons un appareil photographique magnétique Canon ION qui fournit une image vidéo en couleur avec une résolution de 450 lignes par 768 colonnes que nous numérisons avec une carte d'acquisition vidéo Imaging Technologie à la résolution de 512 x 768 pixels en 256 niveaux de gris.

figure 54
Calcul de la matrice de projection

figure 55
Mise en place du volume général

2 Calcul de la matrice de projection

Pour faire ce calcul il n'est pas nécessaire de disposer d'informations géométriques précises.

Le calcul de la matrice peut être fait à partir de points de l'image représentant les sommets d'un “cube” (figure 54) dont la longueur des arêtes est inconnue et temporairement fixée à 1 dans le cas de l’utilisation de la méthode de calcul présentée au paragraphe 3.1 page 119.

La matrice de transformation perspective obtenue le sera à une affinité près et cette affinité pourra être déterminée ultérieurement à partir d'objets présents dans la scène (fenêtre, porte ou balcon dans le cas d’un bâtiment). Il faut remarquer que la déformation due à la matrice de projection ne perturbe pas la saisie.

3 Mise en place du volume général

Le volume général du bâtiment à modéliser est mis en place par affinité d'une primitive unitaire (ici un cube). Ce volume doit être le plus grand possible pour simplifier les opérations suivantes (figure 55). Dans un premier temps, les dimensions qui ne peuvent être appréhendées sont laissées en attente (la profondeur du bâtiment reste inconue).

Ces informations pourront être complétées ultérieurement à partir d’autres photographies si cela s’avére nécessaire.

figure 56
Description de l'objet

figure 57
Facilités de description

4 Description de l'objet

En s'appuyant successivement sur les objets que l'on a déjà mis en place, on implante les détails significatifs du modèle (figure 56).

5 Facilités de description

On remarque, qu'à ce stade de la modélisation, un certain nombre d'éléments sont répétés. On peut, par simple duplication (ex : le balcon de l'étape 4) créer de nouveaux éléments et les positionner à leur place. On remarque que cette méthode permet de mettre en place facilement le balcon du bas de notre bâtiment alors qu'il est presque complètement masqué (figure 57).

Conclusion 8

Nous avons présenté dans cette deuxième partie, un outil d'aide à la modélisation pour la synthèse d'images. Cet outil nous permet de reconstruire de manière interactive un objet visible sur une ou plusieurs photographies par une approximation polyèdrique dont la richesse (résolution) est laissée à l'utilisateur.

Le fait de n'utiliser qu'une seule photographie pour le dimensionnement et le positionnement spatial des objets laisse une grande latitude lors des prises de vues. La reconstruction des objets peut se faire à partir de plusieurs vues prisent sous des angles différents et avec des focales différentes. Cette particularité permet à l'utilisateur de gérer de manière souple la précision de sa modélisation. La base de donnée est crée sous la forme d'un arbre de construction, ce qui la rend compacte (description symbolique) et directement réutilisable dans le système de synthèse d'image de l'Ecole des Mines de Saint-Etienne ("Illumine").

La méthode que nous employons (projection d'un objet sur l'image de la même manière que l'appareil photographique) nous a permis de mettre en évidence une solution géométrique claire au problème de la calibration de la caméra : intersection de trois tores dans l'espace.

Le contrôle du positionnement des objets est assuré par des contraintes entre les diverses primitives de l'arbre CSG. La gestion de ces contraintes permet d'assurer que tous les objets ont au moins un point de contact pour lever l'indéterminé en Z (distance de l'objet à l'observateur). Lorsque la projection d'une primitive se superpose à son image, il est correctement mis en place et dimensionné.

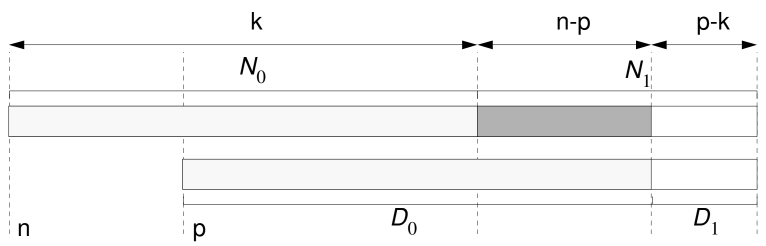
Les extensions que nous proposons à cet outil permettent de résoudre de façon élégante de nombreux problèmes liés à l'incrustation d'objets synthétiques dans des images naturelles : ombres portées par l'objet de synthèse sur le contexte dans lequel il est incrusté, ombres du contexte sur cet objet, calcul d'éclairement, reflets et simplification de l'animation de scènes composites.

Annexe A

1 Conversion d'un nombre rationnel en nombre flottant

Nous avons vu au § 5 quels étaient les problèmes de la conversion d'un nombre rationnel en nombre flottant. La solution que nous avons proposée est efficace et assure un résultat dont l'erreur relative reste inférieure à un seuil acceptable pour la suite des calculs.

1.1 Encadrement du résultat : preuve



Soit un nombre rationnel r représenté par la fraction num/den. num et den sont des entiers de taille non bornée respectivement de n et p chiffres en base b. Cette fraction, que nous appellerons r_{exact}, peut s'écrire de la manière suivante si l'on découpe numérateur et dénominateur au $k^{\text{ième}}$ chiffre significatif :

$$r_{\text{exact}} = \frac{num}{den} = \frac{N_0 b^{n-k} + N_1}{D_0 b^{p-k} + D_1}$$

$$r_{\text{flottant}} = \frac{N_0}{D_0} \times b^{n-p}$$

L'erreur relative que l'on commet sur le résultat flottant se calcule de la manière suivante :

$$\frac{r_{\text{exact}} - r_{\text{flottant}}}{r_{\text{exact}}} = 1 - \frac{N_0 b^{n-p}}{D_0} \times \frac{D_0 b^{p-k} + D_1}{N_0 b^{n-k} + N_1} = \frac{D_0 N_1 - D_1 N_0 b^{n-p}}{D_0 (N_0 b^{n-k} + N_1)}$$

L'encadrement de l'erreur relative de la conversion d'une fraction rationnelle en flottant se décompose en quatre cas qui sont fonctions de la taille respective du numérateur et du dénominateur de la fraction.

Si la taille du numérateur et du dénominateur est inférieure à k :

$$n_1 = 0, d_1 = 0 \Rightarrow \frac{r_{\text{exact}} - r_{flottant}}{r_{\text{exact}}} = \frac{D_0 N_1 - D_1 N_0 b^{n-p}}{D_0 (N_0 b^{n-k} + N_1)} = 0$$

Nous nous intéressons maintenant au cas des fractions rationnelles dont le numérateur ou le dénominateur ont plus de k chiffres. Les valeurs N_0 et D_0 doivent avoir k chiffres, elle doivent donc être comprises entre b^{k-1} et b^k (1000… et 9999… en décimal). N_1 et D_1 sont supérieures ou égales à 0 (nombres de la forme …0000) et strictement inférieures b^{n-k} pour N_1 et b^{p-k} pour D_1 (nombres de la forme …9999).

- Si le numérateur a moins de k chiffres ($N_1 = 0$) :

$$\frac{r_{\text{exact}} - r_{\text{flottant}}}{r_{\text{exact}}} = \frac{D_0 N_1 - D_1 N_0 b^{n-p}}{D_0 (N_0 b^{n-k} + N_1)} = \frac{-D_1 N_0 b^{n-p}}{D_0 N_0 b^{n-k}} = \frac{-D_1}{D_0} b^{k-p}$$

$$\frac{r_{\text{exact}} - r_{\text{flottant}}}{r_{\text{exact}}} \in \frac{-[0, b^{p-k}[}{[b^{k-1}, b^k[} b^{k-p} =]\frac{-1}{b^{k-1}}, 0[\quad \text{avec} \begin{cases} D_0 \in [b^{k-1}, b^k[\\ D_1 \in [0, b^{p-k}[\end{cases}$$

- Si le dénominateur a moins de k chiffres ($D_1 = 0$)

$$\frac{r_{\text{exact}} - r_{\text{flottant}}}{r_{\text{exact}}} = \frac{D_0 N_1 - D_1 N_0 b^{n-p}}{D_0 (N_0 b^{n-k} + N_1)} = \frac{D_0 N_1}{D_0 (N_0 b^{n-k} + N_1)} = \frac{N_1}{N_0 b^{n-k} + n_1}$$

$$\frac{r_{\text{exact}} - r_{\text{flottant}}}{r_{\text{exact}}} \in \frac{[0, b^{n-k}[}{[b^{k-1}, b^k[b^{n-k} + [0, b^{n-k}[} =]0, \frac{1}{b^{k-1}}[\quad \text{avec} \begin{cases} N_0 \in [b^{k-1}, b^k[\\ N_1 \in [0, b^{n-k}[\end{cases}$$

• Si le numérateur et le dénominateur de la fraction ont plus de k chiffres :

$$\frac{r_{\text{exact}} - r_{\text{flottant}}}{r_{\text{exact}}} \in \frac{[b^{k-1}, b^k[[0, b^{n-k}[- [0, b^{p-k}[[b^{k-1}, b^k[b^{n-p}}{[b^{k-1}, b^k[\,([b^{k-1}, b^k[b^{n-k} + [0, b^{n-k}[)}$$

$$\in \frac{[0, b^n[- [0, b^n[}{[b^{n-k-2}, b^{n+k} + b^n[} = \frac{]-b^n, b^n[}{[b^{n-k-2}, b^{n+k} + b^n[}$$

$$\frac{r_{\text{exact}} - r_{\text{flottant}}}{r_{\text{exact}}} \in \left]\frac{-1}{b^{k-2}}, \frac{1}{b^{k-2}}\right[\quad \text{avec} \begin{cases} N_0 \in [b^{k-1}, b^k[\\ N_1 \in [0, b^{n-k}[\end{cases} \text{et} \begin{cases} D_0 \in [b^{k-1}, b^k[\\ D_1 \in [0, b^{p-k}[\end{cases}$$

Nous avons vu au paragraphe [29] page 71 qu'il n'était pas nécessaire d'utiliser un $k > 4$ lors de la conversion et que dans ce cas l'erreur relative maximale que l'on pouvait commettre était de l'ordre de 2^{-32}.

2 Conversion optimale

Nous venons de voir dans le paragraphe précédent, que l'efficacité de la conversion se payait par une erreur relative importante (20 bits de la mantisse du résultat flottant inutilisés). Nous proposons maintenant une solution optimale au sens de la précision mais dont le défaut est de nécessiter des opérations coûteuses sur les entiers de taille non bornée.

Soit $r = \frac{A}{B}$ une fraction rationnelle positive et soit m le nombre de chiffres de la mantisse d'un nombre flottant (53 en double précision).

Le plus grand nombre flottant inférieur ou égal à $\frac{A}{B}$ est :

$$F = M2^k \qquad \text{avec} \qquad \begin{cases} 2^{m-1} \le M < 2^m \\ M2^k \le \frac{A}{B} < (M+1)\,2^k \end{cases} \qquad [\,39\,]$$

donc $M = \left\lfloor \frac{A}{B2^k} \right\rfloor$ (mais nécessite une division euclidienne)

en utilisant [39] : $2^{m+k-1} \le \frac{A}{B} < 2^{m+k}$

Il suffit donc de trouver $x = m + k$ tel que $2^{x-1} \le \frac{A}{B} < 2^x$:

On connaît le nombre de chiffres en base 2 de A : a et de B : b (le nombre de chiffres en base 2 du chiffre le plus à gauche du *BigInt* + le nombre de chiffres de la base b en base 2 * (nombre de chiffres du nombre en base b - 1)).

$$\text{donc } 2^{a-1} \leq A < 2^a \text{ et } 2^{b-1} \leq B < 2^b \text{ donc } 2^{a-b-1} < \frac{A}{B} < 2^{a-b+1}$$

On en déduit donc $x = a - b$ si $2^{a-b} \leq \frac{A}{B}$ sinon $x = a - b - 1$.

Une fois x déterminé et M calculé par division euclidienne, il suffit d'utiliser les m bits de poids fort du résultat pour fabriquer la mantisse et k pour l'exposant du résultat flottant de la conversion. Cette conversion est optimale au sens ou les m bits que l'on utilise pour construire la mantisse sont exacts et qu'il n'est donc pas possible d'obtenir une meilleure représentation en arithmétique flottante.

Annexe B

Nous avons vu au § 4.10.2 qu'il était possible de calculer l'inverse d'un nombre de $\mathbb{Z}_p$ à l'aide du théorème de Bezout :

> Pour tout couple de nombres premiers entre eux $u, v \in \mathbb{Z}, \exists u', v' \in \mathbb{Z}$ tel que $uu' + vv' = 1$.

Cette propriété permet de démontrer que $u^{-1} = u'\%p$ si l'on choisit $v = p$. La relation précédente s'écrit alors :

$$uu' + pv' = 1 \qquad \begin{cases} (uu' + pv')\,\%\mathrm{p} = 1 \\ (uu'\%p + pv'\%p)\,\%\mathrm{p} = 1 \\ uu'\%p = 1 \\ u^{-1} = u'\%p \end{cases}$$

1 Algorithme d'Euclide étendu

Si l'on choisit deux entiers positifs u et v, cet algorithme détermine le vecteur (u_1, u_2, u_3) tel que $uu_1 + vu_2 = u_3 = \mathrm{pgcd}(u, v)$. L'algorithme utilise deux vecteurs auxiliaires (v_1, v_2, v_3) et (t_1, t_2, t_3). L'algorithme entretient les relations suivantes tout au long des calculs :

$$ut_1 + vt_2 = t_3 \qquad uv_1 + vv_2 = v_3 \qquad uu_1 + vu_2 = u_3$$

l'implantation générale de l'algorithme se présente de la manière suivante :

```
PCGD(u,v)
{
      (u1,u2,u3) = (1,0,u);
      (v1,v2,v3) = (0,1,v);
      Tantque (v3 != 0 ) {
         q = u3/v3;
         (t1,t2,t3) = (u1,u2,u3) - (v1,v2,v3)*q;
         (u1,u2,u3) = (v1,v2,v3);
         (v1,v2,v3) = (t1,t2,t3);
      }
      rendre u3;
}
```

La relation $uu_1 + vu_2 = u_3$ permet de supprimer le deuxième terme des trois vecteurs et de réduire le nombre de calculs nécessaires. Les valeurs absolues de u_1, u_2, v_1 et v_2 restent inférieure à u et v ; cette particularité fait que l'algorithme ne produit pas de dépassements de capacités.

2 Application au calcul de l'inverse

Nous venons de voir que dans $\mathbb{Z}_p$ l'inverse d'un nombre u pouvait s'exprimer en fonction de u' : $u^{-1} = u'\%p$, si u est premier avec p. Dans ce contexte particulier nous pouvons simplifier l'algorithme précédent :

```
int InverseModulo(int u,int p)
{
int u1,u2,v1,v2,t,q;

      u1 = 1; u2 = u;
      v1 = 0; v2 = p;

      while (v2 != 0 ) {
         q = u2/v2;
         t = u1 - v1*q; u1 = v1; v1 = t;
         t = u2 - v2*q; u2 = v2; v2 = t;
      }
      return (u1 <0) ? p + u1 : u1;
}
```

Nous utilisons une arithmétique modulo p, u est donc strictement inférieur à p. Nous avons vu au paragraphe précédent que u_1 restait inférieure en valeure absolue aux valeurs initiales de l'algorithme, donc à p, ce qui implique que $u_1\%p = u_1$. Cela permet de rendre directement comme résultat u_1 si celui-ci est positif, l'opposé de u_1 dans $\mathbb{Z}_p$ si u_1 il est négatif. La valeur maximale que l'on peut choisir pour p est de $2^{31} - 1$ si on utilise une arithmétique machine sur 32 bits. Si on choisit cette valeur pour p, tous les calculs dans $\mathbb{Z}_p$ devront être réalisés en utilisant une arithmétique 32 bits non signée pour permettre le calcul de la somme de deux éléments de $\mathbb{Z}_p$ sans débordement. Le produit de deux clefs dépassant 2^{31}, ce calcul doit être simulé.

Le nombre d'itérations maximum nécessaire au calcul de u^{-1} est de 20 (cf paragraphe 3.3 page 22) ce qui correspond malgré tout à environ 250 d'opérations élémentaires.

3 Opérations modulo 2^{31}-1

Les arithmétiques entières non signées sur 32 bits permettent de représenter les valeurs comprises entre 0 et $2^{32}-1$. Si la somme de deux nombres compris entre 0 et $2^{31}-1$ (qui aura pour valeur maximale $2^{32}-2$) peut être calculée avec l'arithmétique entière de la machine, le produit de ces deux nombres sera compris entre 0 et $2^{62}-2^{32}+1$: il dépasse la capacité des entiers machines standards. L'opération doit donc être simulée.

Dans ce paragraphe nous nous intéresserons seulement au calcul de la multiplication modulo $2^{31}-1$, nous n'avons donc jamais besoin de la valeur du produit qui devrait être stockée sur 62 bits.

Soient x et y deux entiers positifs strictement inférieurs à $2^{31}-1$. x et y peuvent s'écrire de la manière suivante :

$$x = 2^{16}x_1 + x_0 \qquad y = 2^{16}y_1 + y_0 \qquad \begin{cases} x_0, y_0 < 2^{16} \\ x_1, y_1 < 2^{15} \end{cases}$$

$$\text{le produit} \quad xy = 2^{32}x_1y_1 + 2^{16}(x_1y_0 + x_0y_1) + x_0y_0 \qquad \begin{cases} x_0y_0 < 2^{32} \\ x_1y_0 \text{ et } x_0y_1 < 2^{31} \\ x_1y_0 + x_0y_1 < 2^{32} \\ x_1y_1 < 2^{30} \end{cases} \qquad [\ 40\ ]$$

Avec les relations décrites au paragraphe 5.3.4 page 72 et adaptées pour la circonstance à la valeur $2^{31}-1$

$$u \oplus v = \begin{cases} u+v & \text{si } u+v < 2^{31} \\ ((u+v)\%2^{31}) + 1 & \text{si } u+v \geq 2^{31} \end{cases}$$

$$u \otimes v = uv\%2^{31} \oplus \lfloor uv/2^{31} \rfloor$$

nous pouvons traduire l'expression [40] de la manière suivante :

$$x \otimes y = (2^{32} \otimes x_1y_1) \oplus (2^{16} \otimes (x_1y_0 + x_0y_1)) \oplus (x_0y_0)$$

Seuls les calculs $2^{32} \otimes x_1 y_1$ et $2^{16} \otimes (x_1 y_0 + x_0 y_1)$ peuvent occasionner des dépassements de capacité. Pour ce qui est du premier :

$$\begin{aligned} 2^{32} \otimes x_1 y_1 &= (2^{32} x_1 y_1) \% 2^{31} \oplus \lfloor 2^{32} x_1 y_1 / 2^{31} \rfloor \\ &= (2^{32} \% 2^{31})\ (x_1 y_1 \% 2^{31}) \oplus 2 x_1 y_1 \\ &= 2 x_1 y_1 \end{aligned}$$

Sur les entiers machine, le produit (respectivement la division entière) d'un nombre n par une puissance de deux (2^p) revient à effectuer un décalage vers la gauche (respectivement vers la droite) d'un nombre de bits égal à la puissance (p) :

$$n \times 2^p = n << p \qquad n \div 2^p = n >> p$$

Le modulo d'un nombre n par une puissance de deux (2^p) se réalise par un "et" logique (opération bit à bit notée &) entre ce nombre et la valeur $2^p - 1$:

$$n \% 2^p = n \ \& \ (2^p - 1)$$

ce qui nous permet de construire l'expression suivante :

$$\begin{aligned} 2^{16} \otimes (x_1 y_0 + x_0 y_1) &= (2^{16} (x_1 y_0 + x_0 y_1)) \% 2^{31} \oplus \lfloor 2^{16} (x_1 y_0 + x_0 y_1) / 2^{31} \rfloor \\ &= ((x_1 y_0 + x_0 y_1) << 16) \ \& \mathrm{0x7FFFFFFF} \oplus (x_1 y_0 + x_0 y_1) >> 15 \end{aligned}$$

Le calcul final ne pose pas de problème particulier, il suffit seulement de faire la somme de deux termes puis le modulo pour éliminer tout débordement.

$$\begin{aligned} x \otimes y = {} & (2 x_1 y_1) \\ & \oplus (((x_1 y_0 + x_0 y_1) << 16) \ \& \mathrm{0x7FFFFFFF} \oplus (x_1 y_0 + x_0 y_1) >> 15) \\ & \oplus (x_0 y_0) \end{aligned}$$

Annexe C

1 Calcul de la position du soleil

La position du soleil peut être déterminée à une date et une heure donnée pour un lieu particulier connu par ses coordonnées (latitude et longitude). Le vecteur indiquant la direction du soleil peut être calculé de la manière suivante [NAKA 86]:

$$\begin{bmatrix} x_\tau \\ y_\tau \\ z_\tau \end{bmatrix} = \begin{bmatrix} \cos\beta & -\sin\beta & 0 \\ \sin\beta & \cos\beta & 0 \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} -\sin\phi & 0 & \cos\phi \\ 0 & 1 & 0 \\ \cos\phi & 0 & \sin\phi \end{bmatrix} \times \begin{bmatrix} \cos\delta & 0 & 0 \\ 0 & \cos\delta & 0 \\ 0 & 0 & \sin\delta \end{bmatrix} \times \begin{bmatrix} \cos\tau \\ \sin\tau \\ 1 \end{bmatrix}$$

β : angle entre l'axe des x et la direction du sud.

τ : angle horaire ($\tau = 0$ correspond à midi et $\tau = 15$ correspond à une heure plus tard.

ϕ : longitude du soleil (ascension droite).

δ : déclinaison du soleil.

Cette solution suppose que l'on connaisse la position du soleil à une date et à une heure donnée. Bien qu'il existe des abaques éditées par le bureau des longitudes, la position du soleil peut être facilement calculée en utilisant des calculs astronomiques élémentaires. Des ouvrages tel que [BOUI 86] proposent des calculs simplifiés de la position des planètes et du soleil avec une précision très largement suffisante à notre utilisation : l'erreur que l'on commet est inférieure à 0,5 sur la déclinaison et l'ascension droite des astres (soit l'équivalent d'environ ± 2 minutes sur le choix de la date à laquelle on calcule la position du soleil).

Annexe D

1 Angles & Cercles dans le plan

<u>Théorème 1</u>: L'angle au centre est égale à deux fois l'angle inscrit.

C'est à dire, si ζ est un cercle de centre O, et A et B deux points distincts de ce cercle, quel que soit M, point de ζ (distinct de A et B), on a :

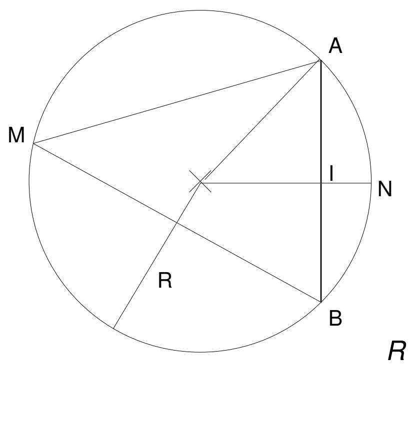
$$(\boldsymbol{OA},\boldsymbol{OB}) = 2\times(\boldsymbol{MA},\boldsymbol{MB})$$

<u>Théorème 2</u>: Soit le cercle ζ de centre O, et A et B deux points distincts de ζ, tels que :

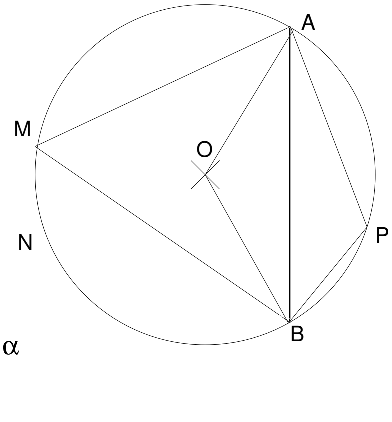
$$(\boldsymbol{OA},\boldsymbol{OB}) = 2\times\alpha$$

Alors, quel que soit M, point du cercle ζ, M voit $[A, B]$ sous l'angle α ou $\pi-\alpha$, suivant l'arc sur lequel il se trouve (d'un coté ou de l'autre segment $[A, B]$).

L'angle correspondant à $(\boldsymbol{OA}, \boldsymbol{OB})$ est l'angle au centre : c'est le double de l'angle inscrit.

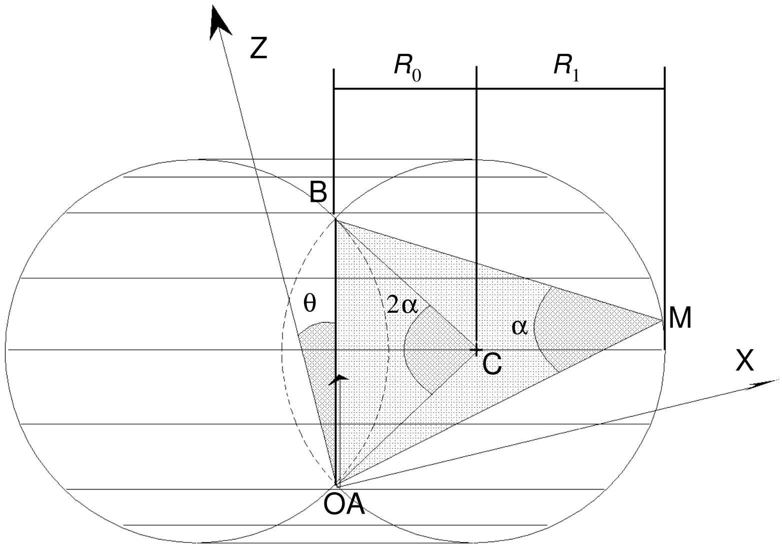


$$R = \boldsymbol{AB}\times\sin\alpha$$

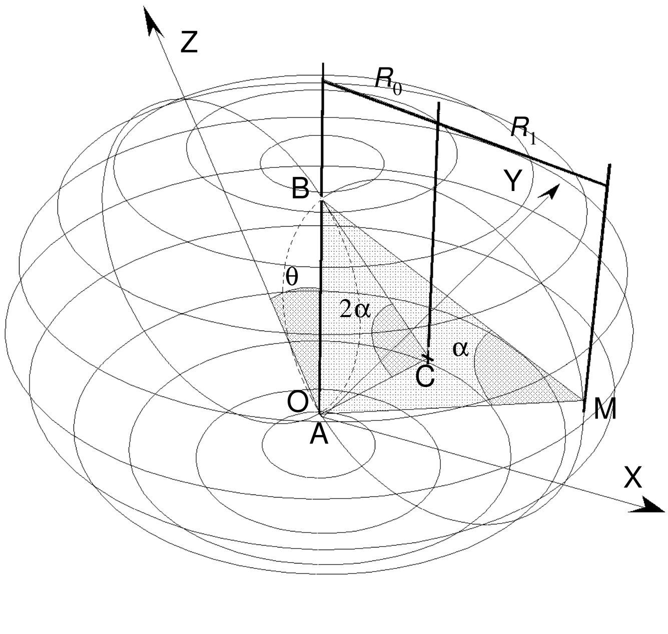


2 Angles & Tores dans l'espace

Dans l'espace, le théorème précédent est bien entendu toujours vrai. Si l'on fait tourner le plan du cercle C autour du segment [A,B], on obtient un tore d'axe [A,B].



$$R_0 = \frac{\|\boldsymbol{AB}\|}{2\tan\alpha},\ R_1 = \frac{\|\boldsymbol{AB}\|}{2\sin\alpha}$$



$$\cos\theta = \frac{\boldsymbol{AB}\cdot\boldsymbol{OZ}}{\|\boldsymbol{AB}\|\cdot\|\boldsymbol{OZ}\|},\ V_{axe} = \boldsymbol{AB}\wedge\boldsymbol{OZ}$$

Bibliographie

[AHO 74] A. V. Aho, J. E. Hopcroft, J. D. Ullman : *The design and analysis of computer algorithms*. Addison Wesley, Reading, Mass., 1974.

[AHO 83] A. V. Aho, J. E. Hopcroft, J. D. Ullman : *Data structures and algoithms*. Addison Wesley, Reading, Mass., 1983.

[APPL 88] Apple Computers : *Apple numerics manual*, Addison Wesley, Reading Mass., 1988.

[ARBO 89] E. Arbogast, R. Mohr : *Reconstruction de surfaces à partir des contours critiques, dans un contexte multi-images*. Rapport de Recherche 784-I-IMAG 94 LIFIA, 1989.

[ATHE 83] P. R. Atherton : *A scan-line hidden surface removal procedure for constructive solid geometry*. Computer Graphics, 17(3), pp. 73-82. 1983.

[AT&T 89] *UNIX System-V, AT&T C++ langage system, release 2.0, Library manual and Selected readings.* USA, AT&T. 1989.

[AYAC 90] N. Ayache, J.L. Jezouin : *Three Dimensional Structure from a Monocular Sequence of Images*. Rapports de recherche de l' INRIA n 1282. Août 1990.

[BATA 91] H. Batatia, A. Ayache, C. Krey : *Reconnaissance d'objets polyédriques à partir d'une seule image, utilisant des invariants projectifs*. RFIA. 25-29 Novembre 1991. Lyon.

[BEIG 88] M. Beigbeder : *Un développement pour la modélisation et la visualisation en synthèse d'images*. Thèse, Ecole des Mines de Saint-Etienne, 1988.

[BEIG 89] M. Beigbeder et B. Peroche, *Un système de synthèse d'images 3D : ILLUMINES*. Actes des Journées UNIX de Grenoble, 1989, 263-276.

[BEIG 91] M. Beigbeder, G. Jahami : *Managing levels of detail with textured polygons*. CompuGraphics'91, pp. 479-489. Sesimbra, Portugal. September 1991.

[BENO 93.1] M. O. Benouamer, D. Michelucci, B. Péroche : *Boundary evaluation using a lazy rational arithmetic*. Proceeding of the second ACM/IEEE symposium on solid modeling and applications. Montréal, Canada. 1993.

[BENO 93.2] M. O. Benouamer : *Opérations booléennes sur les polyèdres représentés par leurs frontières et imprécisions numériques*. Thèse, Ecole des Mines de Saint-Etienne, 1993.

[BENT 79] J.L. Bentley, T. Ottmann : *Algorithms for reporting and counting géométric intersections*. IEEE Trans. Comp. C-28, 9, Sep 1979.

[BERG 83] C. Berge : *Graphes*. 3ème édition. Gauthier-Villars, Paris 1983.

[BIGN 89] B. Serpette, J. Vuillemin, JC. Hervé : *BigNum : A portable and efficient package for arbitrary-precision arithmetic*. Digital Paris research laboratory, May 1989.

[BOUI 86] S. Bouiges : *Calcul astronomique pour amateurs*. Masson, Paris. 1986.

[BRIN 92] P. Brinch Hansen : *Householder reduction of linear equations*. ACM Computing Surveys, Vol. 24, n . 2, June 92.

[BUIT 75] Bui-Tuong Phong : *Illumination for computer generated pictures*. CACM, 18(6), pp. 311-317.June 1975.

[CAME 91] S. Cameron : *Efficient Bounds in constructive solid geometry*. IEEEE CG&A, May 1991, p 68-74.

[CHEN 89 Z. Chen, D.C. Tseng, J.Y. Lin : *A Simple Vision Algorithm For 3-D Position Determination Using A Single Calibration Object*. Pattern Recognition Vol 22, n 2, pp. 173,187. 1989.

[C++W 87] USENIX : *C++ workshop. Proceedings and additional paper*. Santa Fe, NM, 1987.

[DAVE 87] J.Davenport, Y. Siret, E. Tournier : *Calcul formel, sytème et algorithmes de manipulation algébriques*. Masson, Paris, 1987.

[DROM 89] M. Drome, M. Richetin, J.T. Lapresté, G. Rives : *Determination of the attitude of 3D objects from a single perspective view*. IEEE Transactions on Pattern Recognition and Machine Intelligence.Vol 11, n 12, Déc.1989.

[DUFF 85] T. Duff : *Compositing 3-D rendered images.* Computer Graphics. Vol. 19, n 3, 1985, 41-44.

[EDEL 90] H. Edelsbrunner, E. P. Mücke : *Simulation of simplicity : a technique to cope with degenerate cases in geometric algorithms*. ACM Transaction on Graphic, vol. 9, n 1, January 1990, p 66-104.

[EMME 91] M. J. G. M. Van Emmerik : *Interactive design of 3D models with geometric constraints*. The Visual Computer, n 7, pp. 309-325. 1991.

[ERTL 91] G. Ertl, H. Muller-Seelich, B. Tabatabai : *MOVE-X, a system for combining video films and computer animations*. Eurographics'91, Elsevier Science publishers B.V., North-Holland. 1991.

[EVEN 89] P. Even, L. Marcé : *PYRAMIDE, un outil interactif de modélisation de l'environement pour la téléopération assistée par ordinateur*. Actes de MICAD 1989. Paris. 1989.

[FAUG 84] O. D. Faugeras : *Vision par ordinateur en robotique : état de l'art*. Colloque du Cesta, Biarritz, mai 1984.

[FAUG 87] O. D. Faugeras, F. Lustman and G. Toscani : *Motion and structure from point and line matches*. in Proc. Int. Conf. Computer Vision, June 1987.

[FAUG 90] Y. Liu, T. Huang, O. Faugeras : *Determination of Camera Location from 2D to 3D Line and Point Correspondences*. IEEE Transactions on Pattern Analysis and Machine Intelligence. vol 12, n 1. Jan.1990.

[FERT 90] G. Fertey : *Deux problèmes en synthèse d'images : Les sources directionnelles de lumière et une interface évoluée*. Thèse, Ecole des Mines de Saint-Etienne, 1990.

[FLES 92] S. Flesia, M. Roche : *3D reconstruction of generalised cylinders from biplane projections*. Computer and Graphics. Vol 16, n 2. 1992.

[FOLE 90] J. D. Foley, A. van Dam, S. K. Feiner, J. F. Hughes : *Computer graphics, principles and practice, second edition*. Addison Wesley, Reading, Mass., 1990.

[FORT 93] S. Fortune, C.J. Van Wyk : *Efficient exact arithmetic for computational geometry*. 9th Annual Computational Geometry, Ca. USA. February 1993.

[GARC 93] C. Garcia, J.M. Corrieu, S. Bouakaz, D. Vandorpe : *Un système de stéréovision utilisant le réseau de Hopfield*. Actes de MICAD 93, pp. 205-224. Paris, 1993.

[GOLD 91] D. Goldberg : *What every computer scientist should know about floating-point arithmetic*. ACM Computing Surveys, vol. 23, n 1, March 91, pp. 5-48.

[GREC 88] *Perception de la distance par stéréovision*. Actes des journées du GRECO-PRC Communication homme-machine, novembre 1988.

[GRIM 91] J. Grimm : *L'arithmétique générique de Sisyphe*. Rapport technique No 136, Unité de recherche INRIA-Sophia Antipolis, 1991.

[GRIM 81] W. E. L. Grimson : *From images to surfaces, a computational studiy of the human early visual system*. The MIT Press, Massachusetts Institute of Technologiy, Cambrige, Massachusetts, 1981.

[GUIB 85] L. Guybas, J. Stolfi : *Primitives for manipulation of general subdivisions and the computation of Voronoi diagrams*. ACM transaction on Graphic, pp. 74-123. 1985.

[GUIB 89] L. Guibas, D. Salesin, J. Stolfi : *Epsilon geometry : Building robust algorithms from imprecise computations*. ACM 1989, p 208-217.

[GUYO 89] A. Guyot, Y. Herreros, J.M. Muller : *JANUS, an on-line multiplier/divider for manipuling large numbers*. 9th Symp. on Computer Arithemic. Santa Monica, USA, Sep 1989.

[HANO 93] G. Hanotaux : *Techniques de contrôle du mouvement pour l'animation*. Thèse, Ecole des Mines de Saint-Etienne, 1993.

[HANS 90] T. L. Hansen : *The C++ answer book*. Addison Wesley, Reading, Mass., 1990

[HARA 89] R. Haralick : *Determining Camera Parameters from the Perspective Projection of a Rectangle*. Pattern Recognition Vol 22, n 3 pp. 225-230. 1989.

[HARD 60] G.H. Hardy, E.M. Wright : *An introduction to the theory of numbers*. Oxford, University Press, 4^{th} ed. 1960.

[HERM 88] I. Herman : *Projective geometry and computer graphics*. Tutorial Eurographics'88.

[HOFF 88] C. M. Hoffmann, J. E. Hopcroft, M. S. Karasick : *Toward implementing robust geometric computation*. Proc. 4rd Annual ACM Symp. on Computational Geometry (1988), pp. 106-111.

[HOPT 92] J. E. Hoptcroft, P. J. Kahn : *A paradigm for robust geometric algorithms*. Algorithmica, pp. 339-380, Springer Verlag. New York. 1992.

[HORA 87] R. Horaud : *New Method for matching 3-D objects with single Perspective view*. IEEE Transaction on Pattern Analysis and Machine Intelligence, Vol PAMI-9, n 3, May 1987.

[HORA 89] R. Horaud, B. Conio, O. Leboulleux and B. Lacolle : *An analytic solution for the perspective 4-points problem*. Computer Vision, Graphics, and Image Processing 47, 1989, 33-44 .

[IEEE 85] American National Standard institute : *IEEE standard for floating-point arithmetic*. The Institute of Electrical and Electronic Engineers, New York 1985. Reprinted in Sigplan Notice vol. 22 #22, Feb 1987.

[JAHA 91] G. Jahami : *Pour un système de synthèse d'images flexible et évolutif*. Thèse, Ecole des Mines de Saint-Etienne, 1991.

[KAHA 80] W. Kahan : *Interval arithmetic option in proposed IEEE floating-point arithmetic standard*. In Interval Arithmetic 1980. K.E.L. Nickel, New York, Academic Press 1980.

[KANA 91] K. Kanatani : *Computational projective geometry*. Computer Vision, Graphics, and Image Processing : Image understanding. Vol 54, n 3. pp. 333-348. November 1991

[KARA 88] M. S. Karasick : *On the representation and manipulation of rigid solids*. PhD thesis, McGill University, Montréal 1988.

[KARA 89] M. Karasick, D. Lieber, L. R. Nackmann : *Efficient Delaunay triangulation using rational arithmetic*. IBM research report rc 14455 (#64722) August 89.

[KERN 78] B.W. Kernighan, D.M. Ritchie : *The C programming language*. Prentice-Hall, inc. Englewood cliffs, New Jersey. 1978.

[KNUT 81] D. E. Knuth : *The art of computer programming*. Vol. 2, Seminumerical algorithms. Second edittion, Addison Wesley, Reading, Mass., 1981.

[KULI 81] U. Kulisch, W.L. Milranker : *Computer arithmetic in theory and practice*, Academic press, New York 1981.

[KULI 83] U. Kulisch, W.L. Milranker (eds.) : *A new approach to scientific computation*, Academic press, New York 1983.

[MAGE 84] M.J. Magee, J.K. Aggarwal : *Determining vanishing points from perspectives images*. Computer Vision, Graphics, and Images Processing. No 26, pp. 256-267. 1984.

[MENI 93] V. Ménissier-Morain : *Une arithmétique rationnelle exacte efficace en CAML*. Journée francophones des langages applicatifs. Février 1993.

[MICH 87] D. Michelucci : *Les représentation par frontières : quelques constructions; difficultées rencontrées*. Thèse, Ecole des Mines de Saint-Etienne, 1987.

[MIGN 89] M. Mignotte : *Mathématique pour le calcul formel*. Presse Universitaire de France. Paris. 1989.

[MILE 88] V. J. Milenkovic : *Verifiable implementation of geometric algorithms using finite precision arithmetic*. PhD thesis, Carnegie Mellon University 1988.

[MILE 90] V. Milenkovic, Z. Li : *Constructing strongly convex hulls using exact or rounded arithmetic*. Extended abstract for SODA 1990.

[MULL 89] J.M. Muller : *Arithmétique des ordinateurs*. Masson, Paris. Avril 1989.

[MULL 91] J.Duprat, J.M. Muller : *Ecrire les nombres autrement pour calculer plus vite.* Technique et Science Informatiques (TSI) vol. 10, n 3, 1991.

[MOHR 91] R. Mohr, E. Arbogast : *It can be done without camera calibration*. Pattern recognition letters. pp. 39-43. january 1991.

[MOOR 66] R. E. Moore : *Interval analysis*, Prentice -Hall, Englewood Cliff, NJ 1966.

[MORE 90] J. M. Moreau : *Hiérarchisation et facétisation de la représentation par segments d'un graphe planaire dans le cadre d'une arithmétique mixte*. Thèse, Ecole des mines de Saint-Etienne, 1990.

[MORI 91] L. Morin, R. Mohr, E. Grosso : *Positionnement relatif à partir d,invariants projectif*. RFIA. 25-29 Novembre 1991. Lyon.

[NAKA 86] E. Nakamae, K. Harada , T. Ishizaki and T. Nishita : *A montage method : the overlaying of the computer generated images onto a background photograph*. Computer Graphics, Vol. 20, n 4, 1986, p. 207-214.

[NAKA 89] E. Nakamae, K. Harada , F. Kato, T. Nishita, H. Tanaka and T. Noguchi : *Three dimensional terrain modeling and display for environmental assessment*. Computer Graphics, Vol. 23, n 3, 1989, p. 207-214

[NAKA 91] E. Nakamae, K. Kaneda, K. Harada ,T. Miwa, T. Nishita and R. Saiki : *Reliability of computer graphic images for visual assessment*. The visual computer, Vol. 7, 1991, p. 138-148.

[OTTM 87] T. Ottman, G. Thiemt, Ch. Ullrich : *Numerical stability of geometric algorithms*. Proc. 3rd Annual ACM Symp. on Computational Geometry (1987), p. 119-125.

[PERO 88] B. Peroche, J.Argence, D. Ghazanfarpour, D. Michelucci : *La syntèse d'images*. Hermes, Paris (1988).

[PEUC 91] B. Peuchot, M. Saint-Andre, A. Tanguy : *Calibration d'une caméra en vision monoculaire avec zoom*. RFIA. Lyon, 25-29 Novembre 1991.

[PORT 84] T. Porter, T. Duff : *Composing digital images*. Computer Graphics. Vol. 18, n 3, 1984, pp. 253-259

[READ 89] C. Read : *Element of functionnal programming*. Addison Wesley, Reading, Mass., 1989.

[ROGE 76] D. F. Rogers and J.A. Adams : *Mathematical elements for computer graphics*. McGraw Hill, 1976.

[ROGE 85] D. F. Rogers : *Procedural elements for computer graphics*. McGraw-Hill, New York, 1985.

[ROSS 86] J. R. Rossignac : *Constraints in constructive solid geometry*. Proceeding of Workshop on Interacive 3D Graphics. Chapel Hill, ACM Press. pp. 93-110. Oct. 1986.

[SEDG 89] R. Sedgewick : *Algorithms*. Second edition, Addison-Wesley, Reading Mass., 1989.

[SEGA 85] M. Segal, H. Sequin : *Consistent calculations for solids modeling*. Proceedings of the first ACM Symposium on Computationnal Geometry. 1985.

[STRO 87] B. Stroustrup : *The C++ programming langage*. Addison Wesley, Reading, Mass., 1987.

[SUFF 91] K. G. Suffern, E. D. Fackerell : *Interval methods in computer graphics*. Computer & Graphics, Vol. 15, n 3, p 331-340, 1991.

[SUGI 89] K. Sugihara, M. Iri : *A solid modelling system free from topological inconsistency*. English translation of Japanese paper 'An approach of error-free solid modeling' in Transaction of Information Processing Society of Japan, vol. 28 (1987), p 962-974.

[TARJ 72] R. E. Tarjan : *Depth-first search and linear graph algorithms*. SIAM journal on computiong. vol. 1, n 2. 1972.

[ULUP 93] F. Ulupinar, R. Nevatia : *Perception of 3D surfaces from 2D contours*. IEEE Transaction on partern analysis and Machine intelligence. Vol 15, n 1, Junuary 1993.

[YAMA 87] F. Yamagushi : *Theorical foundations for the 4x4 determinant approach in computer graphics and geometric modeling*. The Visual Computer (1987), vol. 3, p 88-97.

[YAMA 93] F. Yamaguchi, K. Toshimitsu, H. Sato, J.Nakagawa : *An adaptative error-free computation based on the 4x4 determinant method*. The Visual Computer (1993), vol. 9, p 173-181.

Participation à publications

[JAIL 92] P. Jaillon, B. Peroche : *Aide à la saisie d'objet en synthèse d'images*. Actes de MICAD 92. Paris, 1992.

[JAIL 93] P. Jaillon : *"LEA", a lazy exact arithmetic : implémentation and related problems*. Technical report, Ecole Supérieure des Mines de Saint-Etienne, 1993.

[BJMM 93.1] M. O. Benouamer, P. Jaillon, D. Michelucci, J.M. Moreau : *A lazy arithmetic library*. Proceeding of the IEEE eleventh symposium on computer arithmetic. Windsor, Canada. July 1993.

[BJMM 93.2] M. O. Benouamer, P. Jaillon, D. Michelucci, J.M. Moreau : *A lazy solution to imprecision in computational geometry*. 5th CCCG, Waterloo Ontario, Canada. August 1993.

[BJMM 93.3] M. O. Benouamer, P. Jaillon, D. Michelucci, J.M. Moreau : *Hashing lazy numbers*. SCAN'93. Vienna, Austria. September 1993.

Index

Liste des figures

PROPOSITION D'UNE ARITHMETIQUE RATIONNELLE PARESSEUSE ET D'UN OUTIL D'AIDE A LA SAISIE D'OBJETS EN SYNTHESE D'IMAGES.

Philippe JAILLON

Mots clefs

Arithmétique, imprécisions, paresse, géométrie. Synthèse d'images, modélisation, interactions, incrustation.

Résumé

La solution la plus commune pour résoudre les problèmes de précision liés aux arithmétiques des ordinateurs est l'utilisation d'arithmétiques exactes. Nous proposons dans la première partie de cette thèse une optimisation très puissante des arithmétiques rationnelles : l'arithmétique rationnelle paresseuse. L'originalité de cette arithmétique est de retarder les calculs exacts jusqu'à ce qu'ils deviennent soit inutiles, soit inévitables. Ainsi les calculs exacts qui ne sont pas nécessaires ne sont jamais faits. L'arithmétique paresseuse se présente sous la forme d'une bibliothèque autonome prenant à sa charge les problèmes de précision et qui est indépendante des programmes qui l'utilisent.

La deuxième partie de cette thèse présente un outil de modélisation dont le principal intérêt est d'utiliser l'image des objets comme support à leur modélisation sous forme d'arbre de construction. Cet outil est interactif, l'utilisateur pourra de cette manière ne modéliser que ce dont il a besoin et avec le niveau de détails le plus adapté à ses applications. Les extensions que nous proposons dans le domaine de l'incrustation d'images de synthèse dans des images naturelles permettent de traiter correctement l'ombrage de la scène finale en tenant compte de la nature des éclairages, des ombres portées et des reflets.

Abstract

The most common solution to solve imprecision problems introduced by the arithmetic of computers is to use exact arithmetic. In the first part of this thesis, we suggest an extremely efficient optimization of rational arithmetic : the lazy rational arithmetic. The originality of this arithmetic is to postpone exact computations until they are either useless or inevitable. In that way, unnecessary exact computations are never made. Lazy arithmetic is an autonomous library which solves imprecision problems by itself, independently of computer programs using it.

The second part of this thesis describes a modelling tool using photographs of objects to model them in a constructive solid geometry description. It is an interactive tool with which the user may build what he needs with the most adapted detail level for his own application. Proposed extensions for image synthesis may be used to solve image incrustation consistently with shadows, lights and environment reflections.