

De la géométrie algorithmique au calcul géométrique

THÈSE

présentée et soutenue publiquement le 19 Novembre 1999

pour l'obtention du

Doctorat de l'université de Nice Sophia-Antipolis
(spécialité informatique)

par

Sylvain Pion

Composition du jury

<i>Président :</i>	Pierre Bernhard	Université de Nice Sophia-Antipolis
<i>Rapporteurs :</i>	Kurt Mehlhorn	Max Planck Institut, Saarbrücken
	Jean-Michel Muller	CNRS, LIP, ENS Lyon
	Franco P. Preparata	Brown University
<i>Examineurs :</i>	Jean-Daniel Boissonnat	INRIA, Sophia-Antipolis
	Hervé Brönnimann	INRIA, Sophia-Antipolis
	André Lieutier	Dassault Systèmes
	Jean-Michel Moreau	École des Mines de Saint-Etienne

Remerciements

Je tiens à remercier Jean-Daniel Boissonnat de m'avoir accueilli au sein de son équipe et de la confiance qu'il a eue en moi, Hervé Brönnimann pour toutes les discussions fructueuses que nous avons eues autour d'une bonne table, et enfin tous les autres membres de mon jury d'avoir pris le temps d'apprécier mon travail : Pierre Bernhard, Kurt Mehlhorn, Jean-Michel Muller, Franco P. Preparata, André Lieutier et Jean-Michel Moreau.

Je remercie également les membres du projet Prisme et les autres pour les bons moments que nous avons passés ensemble durant ces trois années : Anne, Agnès, Monique, Mariette, Olivier, Andreas, François, Alexandra, Frédéric, Stéphane, Julia, Cyrille, Frank, Pierre-Marie, Jack, Jeff, Bruce, Jean-Pierre, David, Ioannis...

Résumé

Dans cette thèse, nous définissons des méthodes efficaces et génériques dans le but de résoudre les problèmes de robustesse que pose la géométrie algorithmique, en se concentrant principalement sur l'évaluation exacte des prédicats géométriques. Nous avons exploré des méthodes basées sur l'arithmétique modulaire, ce qui nous a conduits à mettre au point des algorithmes simples et efficaces de reconstruction du signe dans cette représentation des nombres. Nous avons également mis au point de nouveaux types de filtres arithmétiques qui permettent d'accélérer le calcul des prédicats exacts, en contournant le coût des solutions traditionnelles basées sur des calculs multiprécision génériques. Nos méthodes sont basées sur l'utilisation de l'arithmétique d'intervalles, qui permet une utilisation souple et efficace, combinée à un outil de génération automatique de code des prédicats. Ces solutions sont maintenant disponibles dans la bibliothèque d'algorithmes géométriques CGAL.

Mots-clés: géométrie algorithmique, robustesse, calcul exact, prédicat géométrique, filtre arithmétique, arithmétique d'intervalles, arithmétique modulaire, C++, CGAL.

Abstract

In this thesis, we define efficient and generic methods in order to solve the robustness problems that arise in the field of computational geometry, and we concentrate especially on the exact evaluation of the geometric predicates. We investigated methods based on modular arithmetic, which led us to develop simple and efficient algorithms to reconstruct the sign in this number representation. We also developed new kinds of arithmetic filters, which allow to speed up the exact computation of predicates, working around the cost of traditional solutions based on generic multiprecision computations. Our methods are based on the use of interval arithmetic, which allows an efficient and simple use, combined to an automatic generation tool of the predicates code. These solutions are now available in the CGAL library of geometric algorithms.

Keywords: Computational geometry, exact computation, robustness, geometric predicate, arithmetic filter, interval arithmetic, modular arithmetic, C++, CGAL.

*À la mémoire d'Annabelle,
partie trop tôt.*

Table des matières

Table des figures	xi
Introduction	xiii
1 Modèles de calcul géométrique	1
1.1 Exemple d'algorithme géométrique	1
1.2 Le modèle de calculateur réel à accès aléatoire (Real-RAM)	2
1.3 La réalité	3
1.3.1 La norme IEEE 754	4
1.4 Problèmes de robustesse	6
1.4.1 Epsilon	7
1.4.2 Le paradigme du calcul exact	7
1.4.3 Arrondis sur une grille (snap rounding)	7
1.5 Dégénérescences	7
1.5.1 Traitement au cas par cas	8
1.5.2 Perturbations symboliques	8
1.6 Théorie des domaines	8
1.7 CGAL	9
1.8 Conclusion	10
2 Calcul exact	11
2.1 Arithmétique exacte	11
2.1.1 Bibliothèques existantes	12
2.1.2 Entiers multiprécision	12
2.1.3 Rationnels multiprécision	12
2.1.4 Flottants multiprécision	13
2.1.5 Types de nombre filtrés	13

2.1.6	Réels	14
2.1.7	Performances	15
2.2	Prédicats exacts	15
2.2.1	Qu'est-ce qu'un prédicat?	16
2.2.2	Notion de degré	17
2.2.3	Exemples	18
2.2.4	Abaissier le degré	22
2.2.5	Degré non borné	24
2.3	Solutions pour CGAL	26
2.4	Conclusion	26
3	Arithmétique entière modulaire	27
3.1	Éléments d'arithmétique modulaire	27
3.1.1	Relation de congruence	27
3.1.2	Division modulaire	28
3.1.3	Notations	29
3.1.4	Calcul modulaire	29
3.1.5	Test d'égalité	30
3.2	Implantation	31
3.2.1	Utiliser l'unité flottante	31
3.2.2	Choix des moduli	31
3.2.3	Réduction modulaire	31
3.2.4	Temps de calcul des primitives	32
3.2.5	Parallélisation des calculs	32
3.3	Calcul du signe	33
3.3.1	Méthode de Lagrange (élimination récursive des moduli)	33
3.3.2	Méthode de Lagrange généralisée	35
3.3.3	Variante éliminant ε_k dans la borne	36
3.3.4	Méthode de Newton	36
3.3.5	Variantes probabilistes	38
3.3.6	Parallélisation du calcul du signe	39
3.4	Calcul de la valeur approchée	39
3.5	Étude pratique	40
3.6	Conclusion	40

4	Filtres arithmétiques	41
4.1	Filtres simple précision	41
4.1.1	Filtres statiques	42
4.1.2	Filtres dynamiques	45
4.1.3	L'arithmétique d'intervalles	46
4.1.4	Filtres semi-statiques	50
4.1.5	Filtres statiques adaptatifs	51
4.2	Approches à précision adaptative	51
4.3	Génération automatique	52
4.3.1	LN (Little Numbers)	52
4.3.2	EXPCOMP (Expression Compiler)	53
4.3.3	L'approche prise pour CGAL	53
4.4	Temps de calcul	62
4.5	Conclusion	64
5	Calcul du signe des déterminants	65
5.1	Méthodes existantes	65
5.1.1	Élimination Gaussienne	65
5.1.2	Variante de Bareiss	66
5.1.3	Méthodes de Clarkson et ABDPY	66
5.2	En représentation modulaire	67
5.2.1	Calcul de la borne	67
5.2.2	Élimination Gaussienne	67
5.3	Filtres efficaces avec les intervalles	68
5.4	Temps de calcul	68
5.5	Conclusion	68
	Conclusion	71
	Annexe	73
	A Script PERL pour générer les prédicats filtrés	73
	B Plateformes utilisées pour les mesures	83
	Bibliographie	85

Table des figures

1.1	<i>Enveloppe convexe d'un ensemble de points du plan</i>	2
1.2	<i>Codage d'un nombre en double précision selon la norme IEEE 754.</i>	4
1.3	<i>Les prédicats sont le pont entre les parties numérique et combinatoire.</i>	6
2.1	<i>Codage symbolique et numérique d'un nombre</i>	14
2.2	<i>Prédicat générique</i>	16
2.3	<i>Prédicat $\text{sign}(ab - \sqrt{abcd})$</i>	17
2.4	<i>Le prédicat $\text{compare}_x(P, Q)$</i>	19
2.5	<i>Le prédicat $\text{compare}_x(P, S_1, S_2)$</i>	20
2.6	<i>Le prédicat $\text{orientation}(P, Q, R)$</i>	21
2.7	<i>Le prédicat $\text{in_circle}(P, Q, R, S)$</i>	22
2.8	<i>Delaunay est de degré 4</i>	23
2.9	<i>Marches classique et orthogonale</i>	24
2.10	<i>DAG de constructions géométriques</i>	25
3.1	<i>Temps des primitives modulaires</i>	32
3.2	<i>Le cercle modulaire</i>	37
4.1	<i>Temps des primitives sur les intervalles</i>	49
4.2	<i>Temps des prédicats sur les intervalles</i>	50
4.3	<i>Effectivité des filtres en fonction de l'algorithme</i>	62
4.4	<i>Comparaison des différents types de nombres</i>	63
4.5	<i>Taux d'échec des filtres en fonction de la distribution</i>	63

Introduction

Cette thèse rassemble le travail effectué durant les trois années que j'ai passées au sein du projet Prisme à l'INRIA. Cette équipe a pour thème de recherche principal la géométrie algorithmique, et s'est tournée récemment vers la programmation d'algorithmes géométriques dans le cadre du projet européen de construction d'une bibliothèque commune d'algorithmes géométriques CGAL.

Mon travail s'est orienté plus particulièrement vers la recherche de solutions efficaces et génériques qui garantissent la robustesse des algorithmes géométriques.

Nous introduisons tout d'abord au chapitre 1 les quelques modèles sur lesquels sont basées les études classiques des algorithmes géométriques, qui reflètent les applications de ces méthodes, et qui permettent de fixer le cadre de la suite de cette thèse.

Ensuite, le chapitre 2 montre les deux principales approches pratiques qui permettent de résoudre le problème de la robustesse, à savoir l'arithmétique exacte et les prédicats exacts.

Puis nous discutons d'arithmétique modulaire, méthode bien connue qui permet de résoudre les problèmes d'arithmétique entière exacte de manière simple et performante. Nous traitons principalement dans ce chapitre 3 des méthodes de calcul du signe des nombres en représentation modulaire que nous avons mises au point.

Le chapitre 4 est consacré à l'étude des filtres arithmétiques qui garantissent une très bonne efficacité de l'implantation des prédicats exacts. Nous y décrivons en détail nos travaux utilisant l'arithmétique d'intervalle, ainsi que les divers types de filtres que nous avons mis au point et implantés dans la bibliothèque CGAL.

Enfin, le dernier chapitre 5 se veut une collection des méthodes proposées ces dernières années par la communauté de géométrie algorithmique afin de calculer de manière exacte le signe des déterminants, problème que l'on trouve de manière récurrente dans les prédicats géométriques. Nous y détaillons en particulier les méthodes que nous avons développées, rattachées aux chapitres 3 et 4.

Chapitre 1

Modèles de calcul géométrique

De nombreuses applications informatiques ont besoin de manipuler des objets géométriques complexes, par exemple la robotique, la vision, la conception assistée par ordinateur ou l'infographie. La géométrie algorithmique est véritablement née il y a 25 ans pour étudier ces problèmes, et quelques ouvrages de référence couvrent ce domaine [PS85, Ede87, BY95, dBvKOS97, BY98]. Les objets qui y sont étudiés sont principalement des structures de données adaptées aux manipulations géométriques, et de nombreuses recherches ont porté sur les complexités des algorithmes mis en jeu pour créer ces structures, comme les enveloppes convexes ou les triangulations.

Les ordinateurs ne manipulent pas naturellement des objets géométriques de base, tels les points ou les droites, mais peuvent en traiter une représentation particulière, cette étude est l'objet du calcul géométrique. Nous présentons dans ce chapitre différents modèles utilisés, et la manière dont ils interagissent avec les particularités des ordinateurs réels.

1.1 Exemple d'algorithme géométrique

Un des problèmes géométriques les plus simples à poser est celui du calcul de l'enveloppe convexe d'un ensemble de n points dans le plan, représentés par leurs coordonnées cartésiennes (Fig. 1.1). L'algorithme de calcul doit produire en sortie la liste des points qui sont sur l'enveloppe convexe, c'est-à-dire que pour n'importe quelle paire de points consécutifs de cette liste, tous les points de l'ensemble de départ sont du même côté de la droite définie par ces deux points.

Il y a plusieurs méthodes qui permettent de déterminer cette liste. Un des algorithmes les plus simples est celui de Jarvis, qui fonctionne de la manière suivante :

- déterminer le point de l'ensemble d'abscisse minimale P_0 , ce point est sur l'enveloppe convexe.
- calculer P_{i+1} à partir de P_i de la manière suivante: parmi toutes les droites (P_iP) , pour tout P de l'ensemble de points, la droite (P_iP_{i+1}) est celle de pente minimale mais supérieure à celle de $(P_{i-1}P_i)$.
- s'arrêter lorsque $P_{i+1} = P_0$, on note h la valeur de i finale correspondante.

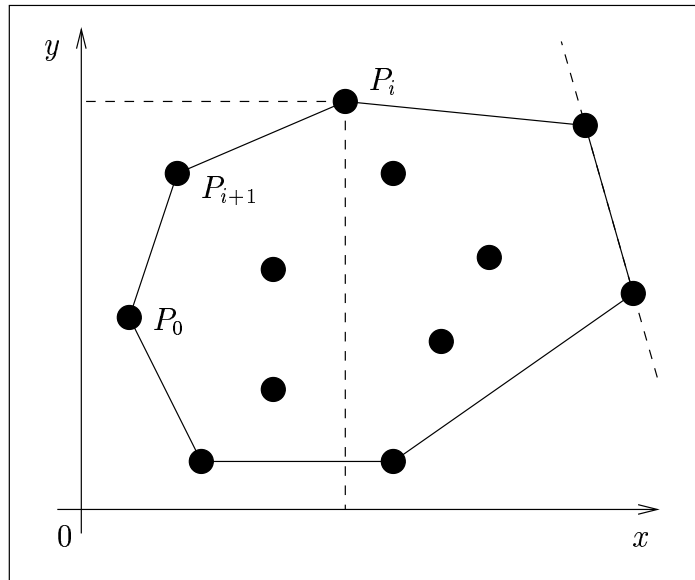


FIG. 1.1 – *Enveloppe convexe d'un ensemble de points du plan*

L'enveloppe convexe est alors la liste des points P_0, P_1, \dots, P_h . La complexité temporelle de l'algorithme est $O(nh)$, donc au pire quadratique par rapport à la taille de l'entrée, n .

Comme souvent en géométrie algorithmique, pour simplifier l'algorithme, nous n'avons pas traité les cas où des points de l'enveloppe convexe pouvaient être alignés : nous avons supposé qu'ils étaient *en position générale*, ce qui correspond à éviter de considérer les cas menant à des valeurs de prédicats qui ont une probabilité nulle d'arriver dans le cas d'une distribution aléatoire des données sur les réels (c'est donc une notion dépendante de l'algorithme). L'opération géométrique de base utilisée par cet algorithme est la comparaison des pentes des droites définies par des points, ce que l'on nomme le prédicat **orientation** (cf chapitre 2), et l'on suppose donc que l'on sait effectuer cette opération, qui est définie comme le signe d'un polynôme simple en les coordonnées des points définissant les droites.

Manipuler les objets géométriques revient donc à effectuer certaines opérations algébriques sur les coordonnées réelles définissant ces objets.

1.2 Le modèle de calculateur réel à accès aléatoire (Real-RAM)

Dans le but d'étudier les complexités des algorithmes, que ce soit en temps ou en espace mémoire, on a cherché à modéliser les ordinateurs de manière simple, et générale, par ce que l'on appelle le calculateur réel à accès aléatoire (Real-RAM).

Dans ce modèle, les propriétés suivantes sont supposées vérifiées par la machine :

- Les nombres réels sont représentés dans des cases mémoire de taille constante.
- L'accès à ces cases mémoire se fait en temps constant, aussi bien pour la lecture que pour l'écriture.
- Les comparaisons entre deux nombres réels sont effectuées en temps constant.

- Les opérations arithmétiques simples (addition, soustraction, multiplication, division, racines, logarithmes...) sont effectuées de manière exacte et en temps constant.

Le modèle Real-RAM offre un cadre simple pour tous les algorithmes géométriques, indépendamment de la représentation numérique des objets, c'est pourquoi il a été autant utilisé. Il modélise aussi relativement bien les machines actuelles, comme nous le verrons dans la section suivante.

L'adoption de ce modèle a permis de continuer à développer de nombreux algorithmes géométriques durant des années, et de nombreux résultats ont été obtenus en ce qui concerne les complexités de différents problèmes. Cependant, certains problèmes liés à l'implantation ont commencé à faire douter du fait que le modèle Real-RAM soit un cadre suffisant pour étudier les algorithmes géométriques. Nous allons maintenant souligner les différences entre ce modèle et les ordinateurs réels, qui peuvent poser des problèmes dans l'implantation pratique des algorithmes géométrique.

1.3 La réalité

La majorité des ordinateurs réels actuels implantent relativement bien le modèle Real-RAM, mais on peut noter tout de même qu'un modèle plus précis pourrait prendre en compte les remarques suivantes :

- Le temps d'accès mémoire dépend de la taille et de la localité des accès répétés aux données en mémoire. Ainsi, le temps d'accès aux registres du processeur se fait en un cycle, l'accès à une donnée dans le cache en quelques cycles, l'accès à la mémoire vive (RAM) se fait en quelques dizaines de cycles, et l'accès en mémoire virtuelle sur disque en quelques dizaines de milliers de cycles. On note aussi que ce schéma s'accroît au fil des années du fait que la vitesse des processeurs augmente plus rapidement que celle de la mémoire et des disques. Ce schéma a conduit à des algorithmes spécifiques au traitement de données de grande taille [GTVV93, AABV99].
- Les réels ne peuvent en aucun cas être stockés dans un espace mémoire de taille constant pour des raisons évidentes. Cependant, le modèle du calcul en virgule flottante, et en particulier de la norme IEEE 754 [IEE85] permet de mémoriser dans un espace constant (32 ou 64 bits) une valeur approchée d'un réel à une certaine précision. Cette représentation vérifie les contraintes de temps constant sur les opérations de base (addition, soustraction, multiplication, divisions, racines carrées), tout en remarquant qu'en pratique, une division est par exemple beaucoup plus coûteuse qu'une addition (d'un facteur variant entre 5 et 30 selon l'architecture).
- Si l'on désire manipuler des nombres réels de manière exacte et non approchée, alors on se restreint à ceux qui sont représentables en machine (de manière symbolique ou numérique). Dans ce cas, la représentation occupe un espace mémoire qui n'est plus borné, et les opérations arithmétiques ne se font plus non plus en temps constant (voir chapitre 2).

À première vue, il y a donc un dilemme entre le choix d'une représentation approchée, pour laquelle les calculs sont effectués de manière approchée mais en temps et espace

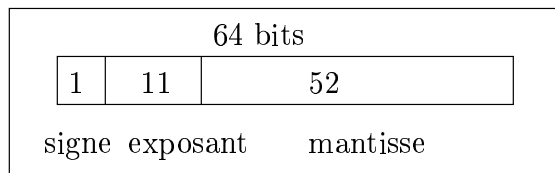


FIG. 1.2 – Codage d'un nombre en double précision selon la norme IEEE 754.

constant, et une représentation plus complexe qui permet de s'affranchir des approximations du calcul flottant au prix d'un coût en temps de calcul et espace mémoire qui n'est plus constant.

1.3.1 La norme IEEE 754

Nous présentons ici quelques notes sur la norme IEEE 754, qui seront utiles pour la compréhension du reste de cette thèse. Pour une vue d'ensemble, le lecteur pourra se référer à la norme elle-même [IEE85] ainsi qu'aux articles de synthèse de Goldberg [Gol91] et Kahan [Kah96].

Cette norme spécifie le codage des nombres à virgule flottante (ou nombres flottants) ainsi que le comportement exact des opérations qui les manipulent, et elle est implantée dans la quasi totalité des processeurs actuels.

Codage

La norme spécifie plusieurs précisions possibles, mais nous n'utiliserons que les flottants double précision, codés sur 64 bits (type `double` du langage C). La figure 1.2 illustre le fait que ces 64 bits sont décomposés en 3 zones : un bit de signe s , un exposant codé par un entier e sur 11 bits, et une mantisse de 52 bits m . La valeur réelle du nombre flottant correspondant vaut $(-1)^s \times 2^{e-1023} \times 1,m$. Les valeurs extrêmes de e (0 et 2047) sont utilisées pour coder les exceptions à cette règle : les valeurs $+0$ et -0 , les nombres dénormalisés, les infinis, et les NaNs (Not a Number). Nous ne détaillerons pas cela ici.

Opérations

La norme définit également le comportement des opérations de base : addition, soustraction, multiplication, division et racine carrée.

Pour chaque opération effectuée sur des nombres flottants, le résultat est calculé en suivant la propriété suivante :

- soit x le résultat réel de l'opération, et \mathbf{x} la valeur flottante à retourner.
- si x est représentable par un flottant \mathbf{x} , alors ce flottant est retourné.
- dans le cas contraire, alors il existe deux flottants consécutifs qui encadrent x ($\underline{\mathbf{x}}$ et $\overline{\mathbf{x}}$).

\mathbf{x} vaudra $\underline{\mathbf{x}}$ ou $\overline{\mathbf{x}}$, selon les règles suivantes.

Modes d'arrondi

L'utilisateur peut choisir un «mode d'arrondi» qui affectera les opérations flottantes dans le choix entre \underline{x} et \overline{x} . Ce choix est effectué en arrondissant x selon l'une des directions suivantes :

- au plus proche (comportement par défaut). C'est la valeur flottante la plus proche de x qui sera choisie (en cas d'égalité, une règle de parité est utilisée que nous ne détaillerons pas ici).
- vers zéro. C'est la valeur de plus petite valeur absolue qui est choisie.
- vers $+\infty$. C'est la valeur la plus grande qui est choisie.
- vers $-\infty$. C'est la valeur la plus petite qui est choisie.

On notera par la suite $\overline{\mp}$ l'opération d'addition entre deux flottants, utilisant le mode d'arrondi vers $+\infty$, $\underline{\pm}$ celle qui utilise le mode d'arrondi vers $-\infty$. On définit de manière similaire $\overline{=}$, $\underline{=}$, $\overline{\times}$, $\underline{\times}$, $\overline{/}$, $\underline{/}$, $\overline{\sqrt{\quad}}$ et $\underline{\sqrt{\quad}}$.

Unité dans la dernière position (ulp)

Pour un nombre flottant x , on note $\text{ulp}(x)$ l'«unité dans la dernière position» (Unit in the Last Place), c'est-à-dire la différence entre x et le flottant qui lui est immédiatement supérieur en valeur absolue.

Cette fonction peut se calculer de la façon suivante :

$$\text{ulp}(x) = (|x| \overline{\mp} \varepsilon_d) - |x|$$

où ε_d est le plus petit double strictement positif.

Notons que l'on peut aussi utiliser la fonction $\text{nextafter}(x,y)$, recommandée par la norme, qui renvoie le flottant suivant immédiatement x dans la direction de y . Ainsi,

$$\text{ulp}(x) = \text{nextafter}(|x|, +\infty) - |x|$$

Erreur d'arrondi

L'erreur commise sur une opération de base, différence entre la valeur réelle x du résultat et la valeur flottante \mathbf{x} , est majorée par $\frac{1}{2}\text{ulp}(\mathbf{x})$ dans le cas où le mode d'arrondi est au plus proche, et $\text{ulp}(\mathbf{x})$ pour les autres modes.

Cas de comparaison exacte

Pour toute opération de base \mathcal{OP} parmi $+$, $-$, \times , $/$, et quels que soient les flottants a , b , c , d , ayant pour valeurs réelles a , b , c , d :

$$(a \mathcal{OP} b < c \mathcal{OP} d) \Rightarrow (a \mathcal{OP} b < c \mathcal{OP} d)$$

C'est-à-dire que le sens de la comparaison donné par le calcul flottant est correct. Ceci est valable quel que soit le mode d'arrondi, pourvu qu'il soit le même pour les deux calculs du membre de gauche (le membre de droite est le calcul exact sur les réels). Cette propriété d'ordre est bien sûr perdue lorsque plusieurs opérations consécutives sont effectuées.

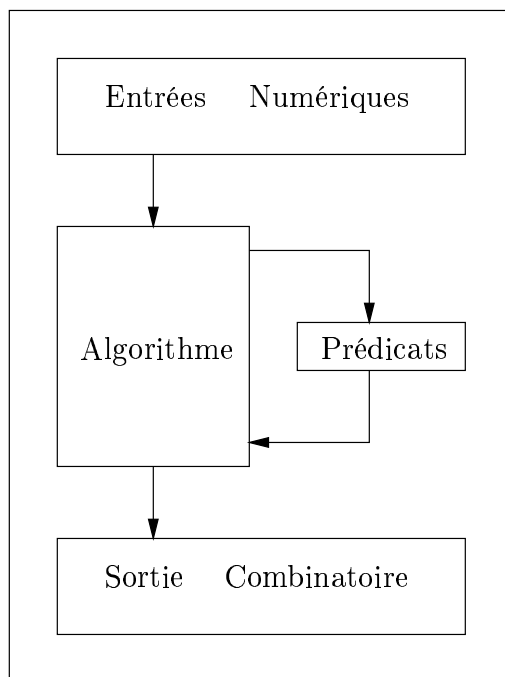


FIG. 1.3 – Les prédicats sont le pont entre les parties numérique et combinatoire.

1.4 Problèmes de robustesse

Les algorithmes géométriques peuvent en général être découpés en deux parties logiquement distinctes : l'une est combinatoire, qui manipule des structures discrètes telles que des graphes ou des arbres, et l'autre est numérique, qui manipule des nombres, tels les coordonnées des objets géométriques.

L'entrée d'un algorithme géométrique est généralement purement numérique, comme un ensemble non structuré de points. L'algorithme va consister dans ce cas à calculer une structure combinatoire à partir de ces données numériques. Puis éventuellement, l'algorithme produira une sortie numérique à partir des données de départ.

Le passage des objets numériques à la structure combinatoire peut être ramené à ce qu'on appelle des prédicats (cf Fig. 1.3), qui sont des fonctions qui prennent en paramètres des entrées numériques, et retournent une valeur booléenne correspondant à un test de comparaison. On peut étendre ce concept strict à une combinaison booléenne de prédicats, qui donc renvoie une valeur dans un ensemble fini de choix possibles.

Un exemple de prédicat est la fonction `orientation`, qui détermine de quel côté d'une droite orientée définie par deux points, un troisième point se situe (les points étant donnés par leurs coordonnées cartésiennes dans le plan). Cette fonction prend en entrée 6 valeurs numériques, et retourne une valeur dans un ensemble à trois possibilités. De plus amples détails sur ce sujet seront donnés dans le chapitre 2.

Les prédicats représentent une partie cruciale de l'algorithme en ce qui concerne la robustesse. En effet, la construction de la structure discrète qui en résulte peut être mal effectuée si les prédicats ne sont pas exacts. Et c'est ce qui peut se passer lorsque le prédicat est calculé avec des nombres flottants approchés. La mauvaise décision qui en

résulte peut provoquer, soit un arrêt inopiné du programme, une boucle infinie...

1.4.1 Epsilons

Les cas d'erreurs sont généralement provoqués par des situations géométriques dégénérées (points alignés par exemple), ou presque dégénérées. Une approche communément admise est de considérer qu'une valeur est nulle si elle est plus petite en valeur absolue qu'une certaine quantité fixe ε . On ne peut cependant pas déduire beaucoup de propriétés d'une telle méthode, même si quelques études ont été menées afin de prouver certains critères de robustesse [GSS89, Pri92].

Pour certains algorithmes, on peut étudier l'impact du point de vue de la cohérence entre les prédicats, des erreurs faites sur les calculs approchés [Sug92]. Par exemple, on peut tolérer que le calcul d'une enveloppe convexe soit approximatif dans le sens où quelques points sont à l'extérieur, s'ils sont en fait très près du polygone définissant l'enveloppe. La structure combinatoire est respectée dans ce cas. Certaines approches offrent un cadre au calcul flottant double précision pour certains problèmes comme les intersections de segments [Mil89a].

On peut aussi étudier la robustesse d'algorithmes dont les prédicats seraient évalués de manière à ce que l'erreur possible soit toujours du même côté [KW98].

1.4.2 Le paradigme du calcul exact

L'approche la plus directe [YD95, Yap93] pour prouver la robustesse des algorithmes, est de garantir l'exactitude des calculs numériques, et en particulier des prédicats (chapitre 2). Le point le plus délicat de cette approche est de la rendre efficace, et ce sera l'objet des filtres (chapitre 4).

1.4.3 Arrondis sur une grille (snap rounding)

Certains algorithmes géométriques produisent des constructions numériques (par opposition à une sortie purement combinatoire), qui est fonction de leurs données d'entrée. Si les calculs sont effectués de manière approchée, alors il se peut que l'ensemble des constructions perde certaines propriétés, comme l'alignement de points, la convexité de cellules, etc...

Des méthodes ont été mises au point au cas par cas [DG99] afin de permettre de plonger les constructions exactes dans un ensemble de valeurs flottantes, tout en conservant un certain nombre de propriétés topologiques qui garantissent la robustesse de l'algorithme [Mil95, Mil89b, GGHT97].

1.5 Dégénérescences

Un des points souvent éludés lors du calcul de la complexité des algorithmes géométriques est la prise en compte des cas dégénérés, c'est-à-dire les cas qui ne sont pas en position générale pour l'algorithme considéré. Cela implique par exemple la gestion des

points colinéaires, des ensembles de plus de deux droites qui s'intersectent en un même point, de points cocirculaires...

Chaque algorithme peut être sujet à ce genre de problèmes. Ne pas en tenir compte simplifie l'analyse, mais au risque d'avoir un déroulement faux du programme (boucle infinie ou arrêt inopiné) si un cas de dégénérescence se produit. En tenir compte nécessite une modification de l'algorithme qui peut entraîner une modification de la complexité.

1.5.1 Traitement au cas par cas

La première méthode qui vient à l'esprit pour traiter les cas dégénérés est de les traiter au cas par cas dans l'algorithme. Ainsi, si l'algorithme effectue un branchement selon la position d'un point par rapport à une droite, il faut aussi considérer le cas où le point se situe exactement sur la droite. Cette approche peut déboucher sur des implantations complexes, surtout lorsque la dimension augmente, ce qui de manière générale n'est pas souhaitable. C'est cependant le choix qui a été fait pour la majorité de la bibliothèque CGAL, car il n'est pas trop pénalisant pour les algorithmes simples en petite dimension.

1.5.2 Perturbations symboliques

Un autre type d'approche a été développé afin d'éliminer ces problèmes à la source, et traiter ainsi tous les cas de manière unique. Elle utilise l'analyse infinitésimale.

On associe à chaque coordonnée réelle une quantité infinitésimale, servant à éliminer les dégénérescences lorsqu'elles apparaissent, mais ne changeant en rien le déroulement de l'algorithme en l'absence de dégénérescences.

Par exemple, lorsque trois points dans le plan sont alignés au vu de leurs coordonnées réelles, par un choix judicieux de quantités infinitésimales associées aux points, on peut tout de même choisir une orientation stricte. Ce choix des epsilons infinitésimaux est bien sûr fixé à l'avance, et garantit la cohérence globale de l'algorithme, tout en éliminant virtuellement les dégénérescences.

La variante la plus efficace actuellement semble être celle des perturbations symboliques linéaires [ECS97], et elle a fait l'objet d'une implantation [CZ99] dans CGAL. Cette méthode est cependant moins générale que la méthode SoS («Simulation of Simplicity scheme») [EM90, Yap88]. Il existe aussi d'autres méthodes pour résoudre des problèmes spécifiques [ADS98].

1.6 Théorie des domaines

Les approches précédentes supposent que les données de départ sont réelles, calculables, et codées exactement en entrée de l'algorithme. Néanmoins, les données mesurées (en CAO, métrologie) ne le sont jamais qu'à une certaine précision, et il faudrait aussi savoir tenir compte de données imprécises, sujettes à certaines contraintes topologiques. Il n'existe que très peu de modèles sachant prendre en compte de telles imprécisions.

Une exception notoire est le modèle de Edalat et Lieutier [EL99] sur les opérations booléennes de solides, où les entrées sont représentables en précision arbitraire par une suite

convergente. La machine ne peut jamais que calculer un préfixe fini de cette suite, mais doit fournir un résultat compatible avec tous les éléments observés de la suite. La théorie des domaines fournit un cadre mathématique pour représenter le résultat (combinatoire) des opérations booléennes.

Une telle approche est essentiellement théorique, et s'intéresse surtout aux aspects de calculabilité de certaines constructions géométriques sur les réels définis comme des suites convergentes. Il serait intéressant d'explorer une implantation de cette approche basée sur une extension multi-dimensionnelle de l'arithmétique d'intervalles (chapitre 4), où les intervalles devraient être affinés lorsque le résultat ne permet pas de conclure. Nous n'avons pas suivi cette voie.

1.7 CGAL

La bibliothèque CGAL [CGA99] est le fruit d'une collaboration entre huit équipes de recherche financées par l'Union Européenne, qui vise à promouvoir l'usage des algorithmes géométriques au sein du monde industriel et scientifique grâce à la création d'une bibliothèque d'algorithmes écrite dans le langage C++. L'accent a été mis sur la modularité de l'ensemble et l'interchangeabilité de nombreuses structures, en utilisant le paradigme de la programmation générique. Cette bibliothèque est organisée en trois parties principales :

- le noyau (kernel library)
- la bibliothèque d'algorithmes fondamentaux (basic library)
- la bibliothèque support (support library)

Le noyau regroupe les objets géométriques de base de taille finie (points, droites, cercles, segments...), ainsi que les opérations qui les manipulent (constructions, prédicats, intersections...). Ces objets sont paramétrés par un type de représentation (cartésienne, homogène), qui lui-même est paramétré par le type de nombres utilisé pour coder les coordonnées (`double`, `int`...).

La bibliothèque d'algorithmes fondamentaux regroupe des structures de données et des algorithmes permettant de créer et de manipuler des objets tels que des enveloppes convexes, triangulations 2D et 3D (quelconques, de Delaunay, régulières), cartes planaires, polyèdres, etc... Tous ces algorithmes s'interfacent avec le noyau et utilisent de manière intensive le mécanisme des classes de traits [Mye95] afin d'offrir une plus grande flexibilité au programmeur.

Enfin, la bibliothèque support regroupe les interfaces avec les outils de visualisation externes, les outils arithmétiques, etc...

CGAL offre donc un cadre idéal pour étudier les différentes solutions aux problèmes de robustesse, génériques ou non, du fait de l'interchangeabilité des composants, et l'un des buts de cette thèse était de trouver des solutions génériques et efficaces aux problèmes posés par les algorithmes fondamentaux. CGAL a permis de mettre en avant des problèmes plus spécifiques au calcul géométrique, par rapport à la géométrie algorithmique en général.

Enfin, on peut mentionner que l'intérêt de regrouper un grand nombre d'algorithmes au sein d'une même bibliothèque est de pouvoir factoriser les solutions à des problèmes spécifiques, en les incluant dans un cadre plus général, ce qui stimule la recherche de

meilleures solutions, aussi bien sur le plan de la généralité que de l'efficacité. Les travaux de cette thèse n'auraient probablement pas eu autant d'applications directes sans CGAL, et certains choix ont été faits en faveur d'une approche applicable à CGAL, par opposition à des approches plus spécifiques à une application particulière.

1.8 Conclusion

Après une phase historique de développement d'algorithmes géométriques, la communauté de géométrie algorithmique s'est aperçu que des problèmes importants liés à l'implantation se posaient, et méritaient d'être traités rigoureusement. Des modèles prenant de plus en plus en compte les contraintes de la réalité des ordinateurs et des problèmes ont été mis en place. La création de CGAL permet d'étudier dans un cadre relativement général les diverses solutions aux problèmes de robustesse, c'est ce que nous allons détailler dans la suite de cette thèse.

Chapitre 2

Calcul exact

Le calcul exact regroupe différentes approches qui permettent de s'affranchir des approximations faites par la machine. Nous rejoignons ici le point de vue exprimé par Yap [YD95, Yap93], qui suppose que l'algorithme doit manipuler les nombres comme s'ils étaient des réels, et que le résultat ne doit pas dépendre d'éventuelles approximations lors du calcul. Cet aspect de la géométrie algorithmique est très important dès que l'on s'intéresse de près aux applications [C⁺96].

La manière la plus simple d'obtenir l'exactitude des calculs, et en particulier des prédicats, est d'utiliser un type de nombres dit exact, c'est-à-dire capable de coder et de manipuler de manière exacte (moyennant une restriction sur les opérations arithmétiques possibles) un sous-ensemble des réels (entiers, rationnels . . .). Plusieurs bibliothèques proposent de tels types de nombres.

Une autre approche consiste à traiter le problème au niveau des prédicats. En effet, des méthodes ont été développées afin de garantir l'exactitude du résultat d'un prédicat sans nécessairement se ramener à une exactitude systématique de chacune des opérations arithmétiques qui le composent. Le prédicat est ainsi considéré par l'algorithme comme une boîte noire dont la réponse est toujours exacte. Cette approche s'avère plus efficace que la première, et elle suffit à garantir la robustesse de la plupart des algorithmes géométriques (nous verrons des exceptions en fin de chapitre).

2.1 Arithmétique exacte

L'arithmétique exacte regroupe un ensemble de méthodes qui effectuent des calculs numériques de manière exacte. Pour cela, quelques bibliothèques proposent des types de nombres multiprécision, sur lesquels certaines opérations arithmétiques sont garanties sans erreur. Selon le type de codage utilisé, seulement certaines de ces opérations sont exactes (par exemple, la division ne peut pas être exacte tout le temps sur des entiers).

2.1.1 Bibliothèques existantes

Voici une liste non exhaustive de bibliothèques permettant de manipuler différents types de nombres :

- GMP [Gra96] est la bibliothèque GNU Multi Precision. Elle permet de manipuler des grands nombres très efficacement, avec une interface en C.
- CLN [Hai99] contient une interface C++ évoluée de GMP.
- LEDA [NU95, MN98] est la «Library of Efficient Data types and Algorithms», c'est une bibliothèque C++ qui contient entre autres des types de nombres.
- LAZY [MM97, Jai93, BJMM93] permet de manipuler des rationnels multiprécision de manière paresseuse.
- CORE [KLPY99b] est une bibliothèque offrant les mêmes fonctionnalités que LEDA en ce qui concerne les types de nombres exacts.

2.1.2 Entiers multiprécision

La majorité de ces bibliothèques représentent les entiers multiprécision par leur codage en binaire dans des tableaux de mots machines (32 ou 64 bits), sur lesquels ils appliquent des algorithmes qui effectuent les opérations arithmétiques de base suivantes :

- addition et soustraction exactes en temps linéaire sur la taille des entrées.
- multiplication exacte en temps quasi linéaire [Knu98] sur la taille des entrées en théorie, bien qu'en pratique on utilise souvent la méthode de Karatsuba, qui est en $O(n^{\log(3)/\log(2)})$.
- division dans le même temps, mais exacte uniquement lorsque le reste est nul.
- comparaison en temps constant en moyenne (linéaire au pire).

Noter qu'il n'y a pas de bibliothèque générique (à ma connaissance) basée sur l'arithmétique modulaire, ce qui n'est pas trop surprenant vues les difficultés de certaines opérations de base (comparaisons, divisions), et ce malgré des complexités optimales pour l'addition et la multiplication. Des efforts ont été entrepris pour inclure un type de nombres partiellement modulaire dans LEDA (version 4.0).

2.1.3 Rationnels multiprécision

L'extension naturelle des entiers multiprécision est le type rationnel, puisqu'il suffit de manipuler des couples (numérateur, dénominateur), eux-mêmes entiers multiprécision. Afin de réduire la taille des opérandes, il convient parfois de réduire la fraction de sorte que le numérateur et le dénominateur soient premiers entre eux.

Ils ajoutent comme opération exacte la division, et de fait couvrent une bonne part des besoins de la géométrie algorithmique. Par exemple, ces opérations suffisent pour calculer les coordonnées cartésiennes du centre d'un cercle circonscrit à trois points.

Exemple 1 *Les coordonnées du centre C d'un cercle passant par trois points P, Q, R non colinéaires sont données par les fractions suivantes :*

$$C_x = P_x + \frac{\begin{vmatrix} R_y - P_y & (R_x - P_x)^2 + (R_y - P_y)^2 \\ Q_y - P_y & (Q_x - P_x)^2 + (Q_y - P_y)^2 \end{vmatrix}}{2 \begin{vmatrix} Q_x - P_x & Q_y - P_y \\ R_x - P_x & R_y - P_y \end{vmatrix}}$$

$$C_y = P_y - \frac{\begin{vmatrix} R_x - P_x & (R_x - P_x)^2 + (R_y - P_y)^2 \\ Q_x - P_x & (Q_x - P_x)^2 + (Q_y - P_y)^2 \end{vmatrix}}{2 \begin{vmatrix} Q_x - P_x & Q_y - P_y \\ R_x - P_x & R_y - P_y \end{vmatrix}}$$

2.1.4 Flottants multiprécision

En couplant un entier multiprécision avec un exposant, on obtient des nombres flottants, dont l'entier multiprécision code la mantisse. Un tel type est généralement moins utile que les rationnels, du fait de l'inexactitude des divisions (on doit fixer une précision maximale a priori). Ils servent principalement de brique de base pour les nombres réels (décrits ci-dessous), dont on gère la précision à la demande. On trouve ces types de nombres dans LEDA, GMP et CLN.

2.1.5 Types de nombre filtrés

Dans le cas particulier des prédicats, c'est le signe d'une expression qui est recherché. On peut donc penser qu'avec un type de nombres adapté qui commencerait le calcul par les bits de poids fort d'abord (puisque ce sont ces bits qui donnent le signe), on aboutirait plus rapidement au résultat, sans calculer inutilement des bits de poids faible, non significatifs.

De nombreux travaux ont été effectués dans ce domaine, en particulier en vue d'une implantation matérielle [KM90, EL88], et dont nous avons fait une implantation logicielle basée sur l'arithmétique redondante [Pio95] qui permet d'effectuer des additions, soustractions et multiplications.

Le principe est de mémoriser les relations entre les nombres sous la forme d'un graphe acyclique dirigé (DAG, directed acyclic graph), qui code chaque nombre, soit sous la forme d'un entier dont on connaît la représentation binaire complète, soit sous la forme d'une opération et de pointeurs sur les opérandes de cette opération. Chaque nombre contient en outre une approximation de sa valeur à une certaine précision. La figure 2.1 montre le codage de l'expression $a = b + c$, où b est la constante 123, et c est une expression multiplicative.

Lorsque l'on désire obtenir le signe d'un nombre (ou bien une valeur approchée à une certaine précision), il suffit de vérifier si l'approximation déjà connue du nombre suffit à décider le signe. Dans le cas contraire, on demande un calcul plus précis de la valeur approchée, en demandant récursivement plus de précision aux opérandes qui définissent le nombre.

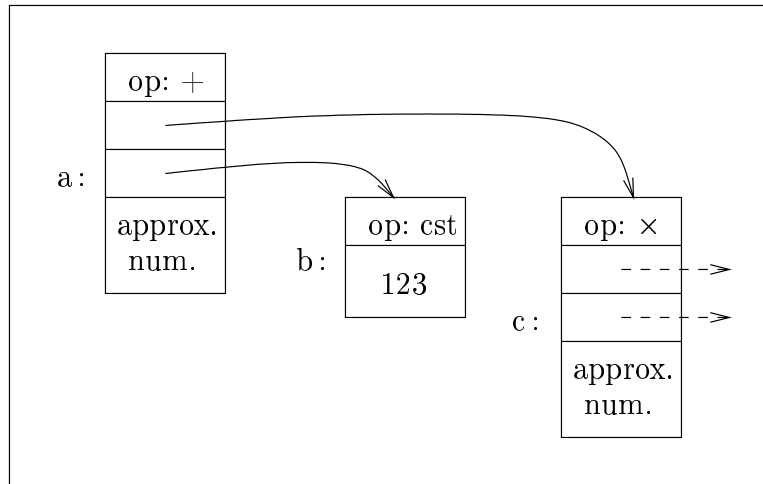


FIG. 2.1 – Codage symbolique et numérique d'un nombre

Selon ce schéma, la nullité d'une expression n'est décidable que lorsque la représentation binaire du résultat a été calculée entièrement, ce qui n'est possible qu'avec des entiers, et en n'utilisant que des additions, soustractions et multiplications.

La représentation binaire est ainsi calculée de manière paresseuse, tout en garantissant le calcul exact du signe de toute expression.

Il existe une implantation paresseuse [Jai93, MM97] des rationnels multiprécision utilisant comme valeur approchée un intervalle de doubles, et qui déclenche en cas de manque de précision un calcul exact en multiprécision. On peut parler de type de nombres «filtré».

De nombreuses variations sur ce thème sont possibles, nous allons maintenant voir que certaines permettent de manipuler plus que des rationnels.

2.1.6 Réels

Certaines bibliothèques proposent des classes de nombres irrationnels : elles autorisent les racines carrées (voire enièmes) en plus des quatre opérations arithmétiques de base, tout en garantissant l'exactitude du calcul du signe. C'est le cas des types `real` de LEDA [BMS96], et `REAL/EXPR` (ou `CORE` [KLPY99a, Yap98]), et des approches à base de «Straight-line Programs» [Reg95].

Elles sont basées sur la méthode du calcul par les poids forts d'abord décrite précédemment, étendue aux divisions et racines carrées. Le point délicat est de décider la nullité d'une expression, sachant que la méthode simple ci-dessus pourrait conduire à une demande de précision infinie, par exemple lorsque l'on demande de calculer le signe de l'expression $2 - \sqrt{2} \times \sqrt{2}$.

Il est nécessaire d'utiliser des bornes de séparation [Mig92, BFMS97], qui garantissent que la précision obtenue est suffisante pour garantir la nullité, en fonction de l'expression calculée.

Un inconvénient de cette approche est qu'elle nécessite une quantité importante de mémoire, ce qui peut la rendre impraticable dans certains cas. En effet, il est nécessaire

de stocker le graphe complet menant à chaque expression, et donc tous les calculs intermédiaires avec toutes leurs approximations respectives.

CORE utilise maintenant (version 1.2) les mêmes bornes de séparation que celles utilisées dans les `real` de LEDA, mais CORE est malheureusement, pour le moment, basée sur une bibliothèque ancienne d'entiers multiprécision de GNU qui est malheureusement inutilisable avec les compilateurs récents.

2.1.7 Performances

Les bibliothèques existantes couvrent donc a priori tous les besoins de la géométrie algorithmique.

Malheureusement, la multiprécision systématique a un coût en temps qui n'est pas négligeable : sur un même algorithme, il n'est pas rare d'observer un facteur 100 entre l'utilisation naïve de nombres multiprécision et de simples doubles.

Certaines approches (LAZY, LEDA `real`) font du calcul multiprécision de manière paresseuse (types «filtrés»), ce qui améliore un peu les choses, puisque le surcoût descend dans la plupart des cas à un facteur 10 par rapport au calcul flottant simple.

Mais on peut faire encore mieux en traitant le problème au niveau des prédicats comme nous allons le voir maintenant. En effet, le principal inconvénient des types de nombres filtrés est qu'ils doivent de toute façon gérer le DAG dynamiquement, même s'il s'avère inutilisé à l'exécution parce que les approximations auront suffi.

Sachant que le calcul flottant simple (inexact) s'avère correct dans l'immense majorité des cas, des méthodes intermédiaires ont été développées, dont les performances en moyenne approchent celles du calcul flottant, mais qui garantissent certaines propriétés d'exactitude, comme le calcul du signe des expressions, qui est la base de l'exactitude des prédicats.

2.2 Prédicats exacts

Les opérations de base sur les objets géométriques sont de deux ordres : celles qui se terminent par un choix (résultat booléen), et celles qui retournent une quantité numérique (résultat potentiellement réel). Les premières sont appelées des prédicats, et les secondes des constructions.

La robustesse de la plupart des algorithmes classiques (triangulations, enveloppes convexes, recherche d'intersections...) est fondée uniquement sur le principe suivant :

- Les coordonnées des objets en entrée sont connues avec une précision infinie (ce qui veut généralement dire que leur définition tient sur un nombre de bits fixé).
- Les prédicats doivent être exacts, ainsi toutes les décisions sont prises correctement, et aucune incohérence topologique ne peut apparaître.
- Soit il n'y a aucune sortie numérique, mais seulement une sortie combinatoire pointant sur les données numériques de départ, non modifiées.
- Soit la robustesse de l'algorithme ne dépend pas d'une sortie numérique qui peut être approximative (pour affichage, par exemple).

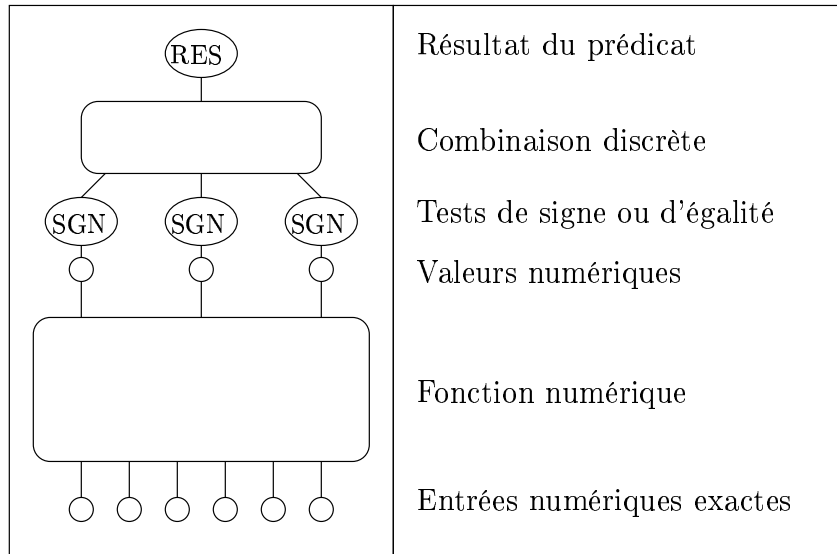


FIG. 2.2 – Prédicat générique

Par exemple, dans le cas d'un diagramme de Voronoï, où l'on désire calculer les coordonnées des cercles circonscrits aux triangles de la triangulation de Delaunay duale, la robustesse est assurée par les prédicats qui calculent la triangulation.

Garantir l'exactitude des prédicats uniquement permet donc de régler les problèmes de robustesse d'une grande partie des algorithmes géométriques.

2.2.1 Qu'est-ce qu'un prédicat ?

Définition 1 *Un prédicat géométrique peut être vu comme la passerelle entre les deux parties d'un algorithme géométrique :*

- la partie numérique, représentée par les coordonnées réelles des objets géométriques codant l'entrée d'un algorithme
- la partie combinatoire, codant par exemple un graphe reliant les sommets d'une triangulation.

Du point de vue de la programmation, les prédicats sont des fonctions :

- dont les arguments sont dans un sous-ensemble des réels, c'est-à-dire un type de nombres,
- et dont le résultat est d'un type discret, par exemple un booléen ou un type énuméré.

La valeur résultat est basée sur une combinaison booléenne de tests de comparaisons entre des valeurs numériques calculées à partir des entrées du prédicat à l'aide d'une formule algébrique correspondant à la propriété géométrique souhaitée (plusieurs formules différentes peuvent exprimer le même prédicat). La figure 2.2 illustre ce propos. La figure 2.3 donne un exemple simple.

Remarque 1 *Il est à noter que certains utilisent une définition plus restrictive d'un prédicat, qui ne peut avoir comme résultat qu'un booléen. La définition ci-dessus, qui*

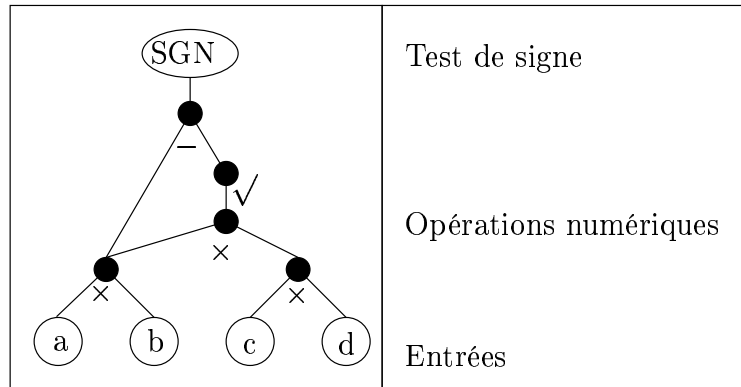


FIG. 2.3 – Prédicat $\text{sign}(ab - \sqrt{abcd})$

sera celle utilisée dans cette thèse, n'est qu'une extension de celle-ci (qui dit que toute combinaison de prédicats est un prédicat) et en aucun cas elle ne restreindra la généralité des remarques. Du point de vue pratique, elle permet de combiner plusieurs calculs dans une seule fonction, ce qui est utile du point de vue de la performance.

Remarque 2 La quasi-totalité des prédicats géométriques peuvent s'exprimer sous la forme de signes de polynômes à coefficients entiers en les entrées.

En effet, les constructions géométriques usuelles font appel à des fractions rationnelles ou des racines carrées, sur les entrées. Les prédicats sont basés sur le signe de telles expressions. Dans le cas des fractions rationnelles (expressions sans racines), il est évident que l'on peut se ramener à des signes de polynômes, puisque $\text{sign}(P/Q) = \text{sign}(P) \times \text{sign}(Q)$.

Dans le cas plus général où l'expression contient des racines carrées, on peut toujours les éliminer en théorie [Blö91]. En pratique, dans les cas simples, cette élimination est également faisable manuellement.

2.2.2 Notion de degré

Pour exprimer le même prédicat géométrique, plusieurs formules algébriques peuvent parfois être utilisées, et on aimerait trouver la plus efficace. Dans le modèle Real-RAM, la mesure serait le nombre d'opérations algébriques, mais dans l'optique d'un calcul exact multiprécision, une mesure communément admise est le maximum des degrés des polynômes intervenant dans une formule donnée correspondant au prédicat, et dont on prend le signe.

Cette mesure permet en effet de connaître approximativement à quelle précision maximale il faudra calculer les expressions pour avoir du calcul exact sur des entiers multiprécision. En effet, lorsque l'on multiplie deux entiers P et Q de b_P et b_Q bits respectivement, on obtient un entier de $b_P + b_Q$ bits. De même, le résultat d'un polynôme de degré d dont les entrées sont des entiers sur b bits sera un entier sur approximativement bd bits (si on néglige les effets dûs aux coefficients et aux additions). Il y a donc un lien direct entre le degré d'une expression et la complexité de son calcul en multiprécision.

Il ne s'agit là que d'une mesure qualitative, et il peut y avoir des cas où l'expression polynomiale de plus bas degré n'est pas la meilleure, par exemple quand elle correspond à un polynôme plein, alors qu'un polynôme creux de degré légèrement supérieur conviendrait. Il convient également de considérer la factorisation des polynômes avant de prendre leur signe, afin de diminuer le degré maximal.

Définition 2 *Le degré d'une formule exprimant un prédicat est le degré maximal des polynômes sur les entrées du prédicat dont la formule prend le signe.*

Définition 3 *Le degré d'un prédicat est le degré minimal des formules exprimant ce prédicat.*

On étend cette mesure aux algorithmes de la façon suivante :

Définition 4 *Le degré d'un algorithme géométrique est le degré maximal des prédicats qu'il utilise.*

Afin de minimiser les coûts au niveau de l'arithmétique nécessaire, le but est donc de trouver l'algorithme de plus petit degré qui résoud un problème. Notons que cela peut être en contradiction avec l'efficacité dans le modèle Real-RAM.

Définition 5 *Le degré d'un problème géométrique est le degré minimal des algorithmes qui le résolvent.*

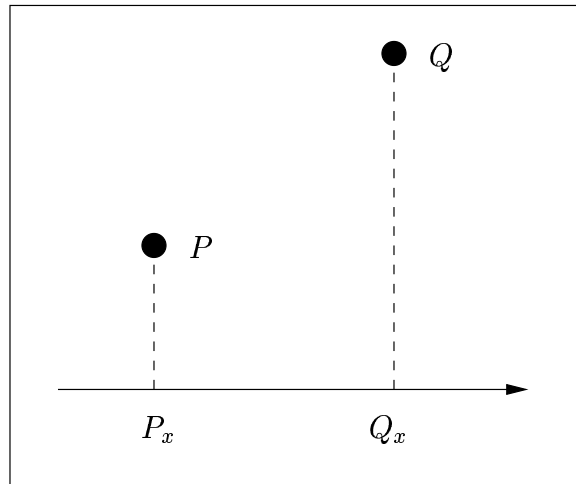
Remarque 3 *Le degré d'une expression est lié à une grandeur physique, par exemple, si les entrées d'un prédicat sont les coordonnées de points, alors ils sont assimilables à des longueurs (degré 1). Or on peut concevoir que certains prédicats prennent en entrée des mesures de surface, donc de degré 2 (voire davantage). Dans ces cas, on calcule le degré du prédicat en considérant que l'entrée correspondante est déjà un polynôme de degré supérieur à 1.*

On retiendra que la raison d'être de la notion de degré est de fournir un outil de comparaison pour l'étude des complexités qui interviennent dans le calcul exact multiprécision des prédicats.

2.2.3 Exemples

Voici quelques exemples de prédicats usuels. Ceux-ci sont exprimés en fonction des coordonnées cartésiennes des points d'entrée, on notera (P_x, P_y) les coordonnées du point P . On représente les segments par une paire de points.

Plusieurs représentations sont possibles. On pourrait utiliser les coordonnées homogènes des points, ou bien définir une droite par son équation cartésienne, ou bien d'autres représentations, nous avons choisi la plus courante afin d'illustrer notre propos. L'essentiel est que les valeurs d'entrées des prédicats ne soient pas déjà entâchées d'erreur de calcul.

FIG. 2.4 – Le prédicat $\text{compare}_x(P, Q)$

Changer la représentation des objets géométriques, de manière générale, modifie le degré des prédicats où ils interviennent.

Définition 6 Par la suite, on utilisera les fonctions sign et compare définies comme suit :

$$\text{sign}(a) = \text{compare}(a, 0) = \begin{cases} 1 & \text{si } a > 0 \\ 0 & \text{si } a = 0 \\ -1 & \text{si } a < 0 \end{cases}$$

$$\text{compare}(a, b) = \text{sign}(a - b) = \begin{cases} 1 & \text{si } a > b \\ 0 & \text{si } a = b \\ -1 & \text{si } a < b \end{cases}$$

Par définition, ces fonctions sont considérées comme des prédicats.

Propriété 1

$$\text{sign}(ab) = \text{sign}(a/b) = \text{sign}(a) \times \text{sign}(b)$$

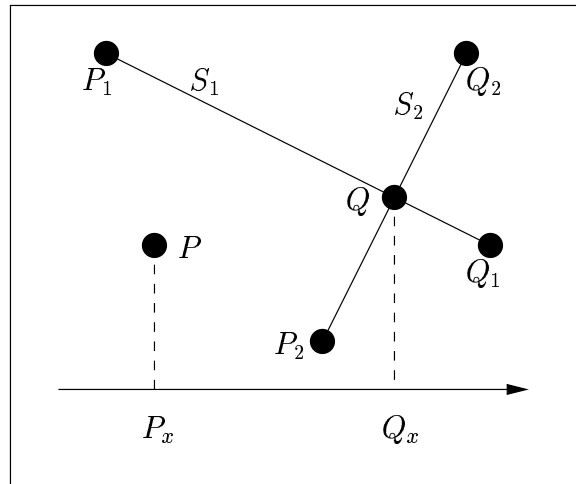
Cette propriété triviale est utile pour réduire le degré des prédicats.

Exemple 2 $\text{compare}_x(P, Q)$ (figure 2.4)

Les algorithmes basés sur un balayage du plan par une droite parallèle à l'axe des ordonnées commencent par trier les points selon leurs abscisses (voire lexicographiquement). Ils utilisent donc le prédicat compare_x (resp. compare_y) qui compare les abscisses (resp. ordonnées) de deux points du plan, ce qui est nécessaire et suffisant pour trier les points lexicographiquement. L'expression de ce prédicat, de degré 1, est la suivante :

$$\text{compare}(P_x, Q_x)$$

Exemple 3 $\text{compare}_x(P, S_1, S_2)$ (figure 2.5)

FIG. 2.5 – Le prédicat $\text{compare}_x(P, S_1, S_2)$

L'algorithme de Bentley-Ottman utilise aussi la comparaison des abscisses d'un point et de l'intersection de deux segments $S_1 : (P_1, Q_1)$ et $S_2 : (P_2, Q_2)$, qui sont supposés d'intersecter. Les équations cartésiennes des droites supports des segments sont de la forme : $a_1x + b_1y + c_1 = 0$, avec $a_1 = Q_{1y} - P_{1y}$, $b_1 = P_{1x} - Q_{1x}$ et $c_1 = Q_{1x}P_{1y} - Q_{1y}P_{1x}$.

L'abscisse du point d'intersection est donnée par :

$$Q_x = \frac{\begin{vmatrix} b_1 & c_1 \\ b_2 & c_2 \end{vmatrix}}{\begin{vmatrix} a_1 & b_1 \\ a_2 & b_2 \end{vmatrix}} = \frac{\begin{vmatrix} P_{1x} - Q_{1x} & Q_{1x}P_{1y} - Q_{1y}P_{1x} \\ P_{2x} - Q_{2x} & Q_{2x}P_{2y} - Q_{2y}P_{2x} \end{vmatrix}}{\begin{vmatrix} Q_{1y} - P_{1y} & P_{1x} - Q_{1x} \\ Q_{2y} - P_{2y} & P_{2x} - Q_{2x} \end{vmatrix}}$$

L'expression de ce prédicat, de degré 3, est donc la suivante :

$$\text{compare} \left(\begin{vmatrix} b_1 & c_1 \\ b_2 & c_2 \end{vmatrix}, P_x \begin{vmatrix} a_1 & b_1 \\ a_2 & b_2 \end{vmatrix} \right) \times \text{sign} \left(\begin{vmatrix} a_1 & b_1 \\ a_2 & b_2 \end{vmatrix} \right)$$

Exemple 4 orientation(P, Q, R) (figure 2.6)

Un autre exemple usuel est le prédicat *orientation*, qui détermine de quel côté de la droite orientée définie par deux points P et Q un troisième point R se situe. Il donne aussi l'orientation des trois points P, Q, R sur le cercle qui leur est circonscrit. Ce prédicat est utilisé intensivement dans les algorithmes de calcul d'enveloppe convexe et de triangulation. Une formule exprimant ce prédicat est la suivante :

$$\text{sign} \left(\begin{vmatrix} P_x & P_y & 1 \\ Q_x & Q_y & 1 \\ R_x & R_y & 1 \end{vmatrix} \right)$$

Soit encore :

$$\text{sign} \left(\begin{vmatrix} P_x - R_x & P_y - R_y \\ Q_x - R_x & Q_y - R_y \end{vmatrix} \right)$$

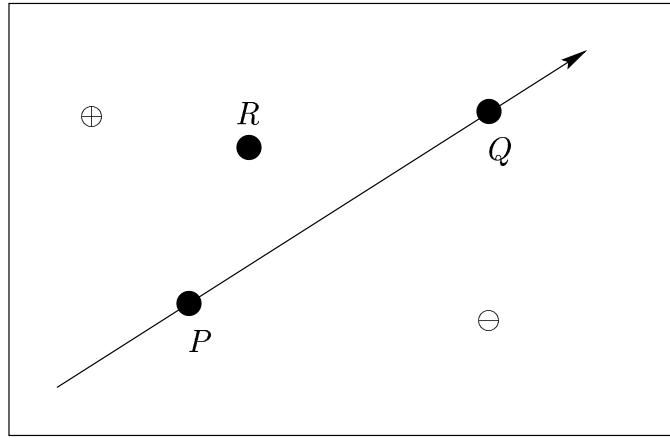


FIG. 2.6 – Le prédicat orientation(P,Q,R)

Cette expression est de degré 2. Elle correspond à un déterminant dont les entrées sont indépendantes, donc n'est pas factorisable. Cette propriété peut se démontrer facilement par récurrence sur la dimension.

Exemple 5 `in_circle(P,Q,R,S)` (figure 2.7)

Tout algorithme de construction (ou de vérification) d'une triangulation de Delaunay de points du plan utilise ce prédicat. Il détermine si un point S est à l'intérieur ou à l'extérieur du cercle circonscrit aux trois points orientés positivement P,Q,R .

La formule classique exprimant ce prédicat est la suivante :

$$\text{sign} \left(\begin{vmatrix} P_x & P_y & P_x^2 + P_y^2 & 1 \\ Q_x & Q_y & Q_x^2 + Q_y^2 & 1 \\ R_x & R_y & R_x^2 + R_y^2 & 1 \\ S_x & S_y & S_x^2 + S_y^2 & 1 \end{vmatrix} \right)$$

Ce qui est équivalent à :

$$\text{sign} \left(\begin{vmatrix} P_x - S_x & P_y - S_y & (P_x - S_x)^2 + (P_y - S_y)^2 \\ Q_x - S_x & Q_y - S_y & (Q_x - S_x)^2 + (Q_y - S_y)^2 \\ R_x - S_x & R_y - S_y & (R_x - S_x)^2 + (R_y - S_y)^2 \end{vmatrix} \right)$$

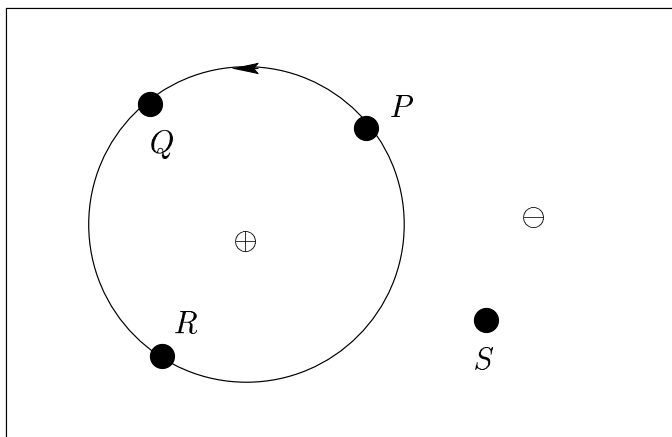
Soit, en notant P',Q',R',S' les translatés de P,Q,R,S tels que S' se retrouve à l'origine par cette translation :

$$\text{sign} \left(\begin{vmatrix} P'_x & P'_y & P_x'^2 + P_y'^2 \\ Q'_x & Q'_y & Q_x'^2 + Q_y'^2 \\ R'_x & R'_y & R_x'^2 + R_y'^2 \end{vmatrix} \right)$$

Soit encore :

$$\text{sign} \left(\begin{vmatrix} P'_x Q'_y - Q'_x P'_y & Q'_x (Q'_x - P'_x) + Q'_y (Q'_y - P'_y) \\ P'_x R'_y - R'_x P'_y & R'_x (R'_x - P'_x) + R'_y (R'_y - P'_y) \end{vmatrix} \right)$$

Ces expressions sont irréductibles et de degré 4 (on peut le montrer avec Maple).

FIG. 2.7 – Le prédicat $\text{in_circle}(P, Q, R, S)$

Remarque 4 *Tous ces prédicats sont invariants par translation (le résultat ne change pas si on translate tous les points d'un même vecteur), comme c'est souvent le cas en général. On note que l'expression d'un prédicat est souvent plus simple si l'on commence par traduire tous les points de sorte que l'un d'entre eux se retrouve à l'origine (il aura donc des coordonnées nulles, ce qui simplifie l'expression).*

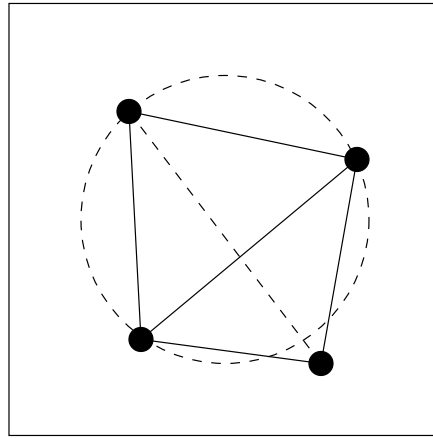
D'autres exemples peuvent être trouvés dans le noyau de la bibliothèque CGAL [FGK⁺96, CGA99]. On notera aussi le prédicat de degré 40 nécessaire au calcul du diagramme de Voronoï de segments dans le plan [Bur96]. Les algorithmes qui utilisent une méthode de balayage sont généralement de degré élevé, tel celui de Fortune qui calcule un diagramme de Voronoï [For86], qui est de degré 20. Enfin, des études ont été faites sur les prédicats utilisés pour calculer des arrangements d'arcs de cercles dans le plan [DFMT99].

2.2.4 Abaisser le degré

Des travaux ont été menés afin d'essayer de diminuer le degré de certains problèmes classiques, notamment pour le problème du calcul des intersections de segments [BP97a] et de courbes [BS99] dans le plan, ainsi que pour la recherche de plus proches voisins [LPT97]. Une étude du degré du calcul du diagramme de Voronoï de segments dans le plan a été effectuée dans [Bur96]. Ces algorithmes ont nécessairement une complexité plus grande puisqu'ils sont plus contraints, donc de manière générale, on essaye d'équilibrer les deux aspects : degré et complexité.

Il y a plusieurs méthodes, on peut soit essayer d'abaisser le degré d'un prédicat donné en trouvant une formule plus adaptée [DFMT99], soit modifier les algorithmes afin qu'ils utilisent des prédicats de plus petit degré [BP97a], comme c'est le cas pour l'exemple que nous donnons maintenant.

Nous montrons ici, sur un algorithme particulier, comment abaisser le degré «moyen», c'est-à-dire le degré du prédicat qui intervient le plus souvent, dans la génération d'une triangulation de Delaunay : le prédicat *orientation*. On peut montrer facilement que le degré du problème du calcul d'une triangulation de Delaunay dans le plan est de degré 4, en observant que le cas d'un ensemble de 4 points en position convexe est déterminé

FIG. 2.8 – *Delaunay est de degré 4*

exactement par le prédicat `in_circle`, qui est de degré 4 (Fig 2.8) (et d'autre part il existe des algorithmes de degré 4 qui résolvent ce problème).

Nous avons étudié le problème spécifique de la localisation dans une triangulation plane (ou même plus généralement multi-dimensionnelle). Les algorithmes incrémentaux pour construire une triangulation nécessitent la localisation d'un point dans la triangulation courante, c'est donc l'étape cruciale, et elle est souvent réalisée par une marche le long d'une droite qui coupe la triangulation. Des méthodes plus évoluées, basées sur des représentations hiérarchiques, utilisent le même genre de marche, mais en réduisent la longueur par diverses méthodes [Dev98].

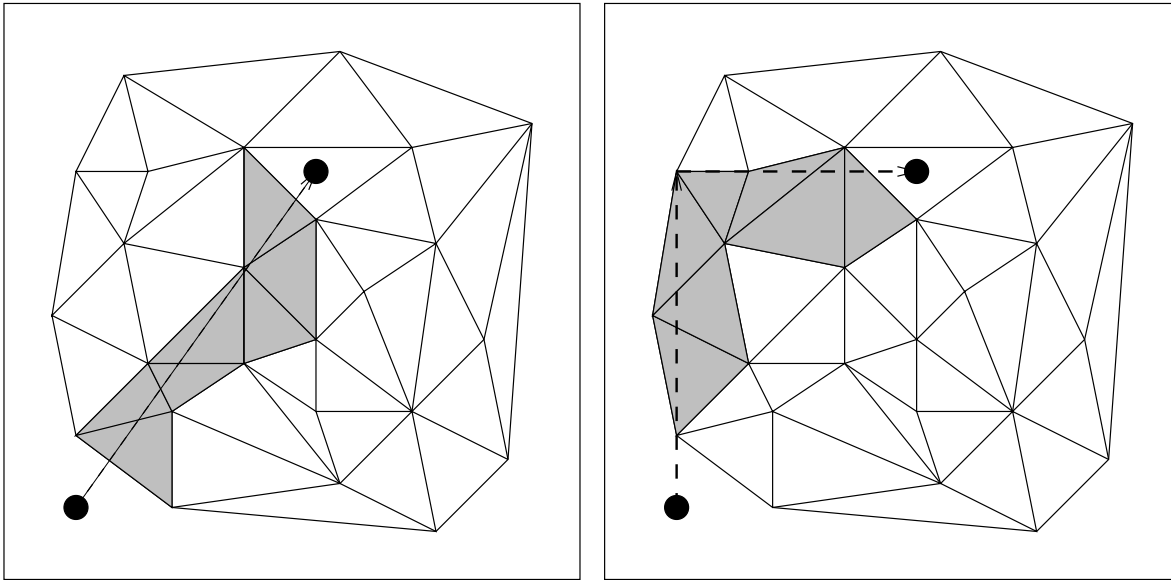
Cette marche est constituée d'une série d'appels au prédicat `orientation` (de degré 2). Or on remarque simplement qu'en suivant un tracé parallèle aux axes, seuls les prédicats `compare_x` et `compare_y` (de degré 1) sont appelés durant la majeure partie de la marche (Fig. 2.9).

Sur le plan théorique, on peut estimer que le nombre moyen de triangles intersectés (proportionnel au temps de calcul) est proportionnel à la longueur de la marche, qui est rallongée de 27% en moyenne :

$$\langle |\cos + \sin| \rangle = \frac{2}{\pi} \int_0^{\frac{\pi}{2}} (\cos \theta + \sin \theta) d\theta = \frac{4}{\pi} \approx 1.27$$

D'après nos mesures, environ 30% du temps que prend la construction d'une triangulation de Delaunay est passé dans le prédicat `orientation` utilisé lors de la marche classique, lorsque celui-ci est évalué de manière inexacte avec des doubles. Compte tenu du rallongement de la marche orthogonale, on peut espérer un temps de calcul global similaire. Des expérimentations effectuées par Olivier Devillers sur son programme de triangulation [Dev98] ont montré une différence d'au plus 5% par rapport à la marche classique, en plus ou en moins selon le jeu de données.

On peut espérer un meilleur rapport lorsque l'on utilise des types de nombres multi-précision, où une comparaison se fait généralement en temps constant, contrairement à un prédicat d'orientation (qui fait appel à 2 multiplications et 4 additions). Ces expérimentations n'ont pas été effectuées, le programme utilisé ne permettant pas d'interchanger

FIG. 2.9 – *Marches classique et orthogonale*

facilement le type de nombres (à l'époque des tests).

De manière générale, adopter le calcul exact coûte nécessairement plus cher que le calcul approché, puisqu'il s'agit d'une contrainte strictement rajoutée aux problèmes. Il faut donc essayer de minimiser la proportion de temps passé dans les prédicats, soit en diminuant leur degré, soit en changeant légèrement les algorithmes. La meilleure méthode n'est pas nécessairement de considérer le degré, car elle oblige à ne considérer que des polynômes, donc à se priver de divisions et de racines carrées. Éliminer les divisions et les racines carrées est toujours possible [Blö91], mais cela fait intervenir un nombre exponentiel de multiplications et d'additions, qui font augmenter le degré. Donc il peut parfois être avantageux de conserver les expressions d'origine, non polynomiales, en particulier dans le cas des filtres.

2.2.5 Degré non borné

Certains algorithmes ne calculent pas uniquement une structure combinatoire à partir de données numériques en entrée, mais effectuent aussi des constructions géométriques, par exemple on peut vouloir la liste des sommets d'un diagramme de Voronoï, et pas sa triangulation de Delaunay duale. Ces algorithmes, même s'ils utilisent des prédicats exacts, ont une sortie numérique potentiellement inexacte, si le type de nombres utilisé n'est pas exact. Le fait d'employer uniquement des prédicats exacts lors de l'algorithme ne garantit pas que les données numériques approchées en sortie vérifieront toujours certains critères topologiques (par exemple la convexité des cellules du diagramme de Voronoï).

Des incohérences peuvent alors apparaître si l'on utilise ces constructions numériques inexactes en entrée d'un autre algorithme, qui présuppose qu'elles vérifient toujours certaines propriétés, ou si elles sont simplement réutilisées à l'intérieur du même algorithme. On a alors des constructions en cascade.

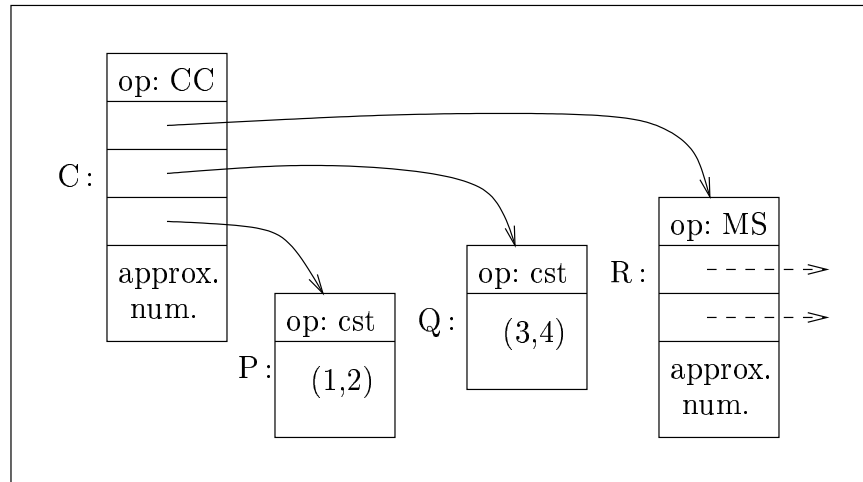


FIG. 2.10 – DAG de constructions géométriques

L'approche décrite dans cette section qui est basée sur la garantie de prédicats exacts est battue en brèche ici puisque les entrées des prédicats ne sont plus exactes. Il faut donc garantir l'exactitude des entrées, ce qui peut être fait grâce à un type de nombres exact adapté, qui peut être un type filtré pour plus d'efficacité.

Or dans ces conditions, on peut obtenir un algorithme dont le degré global n'est pas borné, et les solutions actuelles à base de types de nombres exacts ont donc un impact coûteux en ce qui concerne les performances, même pour les types de nombres filtrés, dont le risque d'échec du filtre augmente considérablement.

Ce type de problème peut également surgir sans qu'il y ait une cascade de constructions. Prenons par exemple un algorithme qui chercherait à minimiser la distance d'un parcours polygonal (plus court chemin dans un graphe). Afin de comparer la longueur de deux parcours, il effectuera une comparaison de deux sommes de racines carrées, ce qui est équivalent au signe d'un polynôme dont le degré dépend du nombre de points des parcours, donc n'est pas borné.

Une solution serait de définir symboliquement ces constructions en maintenant des pointeurs vers les objets exacts d'entrée les définissant, et en utilisant ces définitions symboliques dans les prédicats. On peut aussi penser effectuer un calcul filtré de ces constructions pour améliorer les performances. Il s'agit d'une approche similaire à celle des types de nombres filtrés, mis à part que les problèmes d'inexactitude seraient traités au niveau des constructions géométriques, à la place des opérations arithmétiques, ce qui devrait en réduire le coût. La figure 2.10 montre l'exemple de la définition du centre C du cercle circonscrit (opération «CC») à trois points P de coordonnées $(1,2)$, Q de coordonnées $(3,4)$, et R , défini comme le milieu du segment (opération «MS») de deux autres points. Nous avons là un graphe acyclique dirigé similaire à celui décrit précédemment pour les types de nombres.

À notre connaissance, cette approche n'a pas été implantée de manière générique, mais elle pourrait peut-être se révéler intéressante pour quelques algorithmes.

Les méthodes de «rounding» (introduites au chapitre 1), réduisent le degré des constructions à 1, en choisissant une valeur approchée de manière à conserver les pro-

propriétés topologiques voulues.

2.3 Solutions pour CGAL

La bibliothèque CGAL offre plusieurs possibilités pour traiter les problèmes de robustesse. D'une part, tous les algorithmes sont paramétrés par le type de nombres qui est utilisé pour représenter les coordonnées des objets géométriques. De cette façon, il est très facile d'utiliser une bibliothèque de types de nombres exacts.

De plus, CGAL permet de surcharger les fonctions génériques codant les prédicats, de manière à pouvoir utiliser les approches basées sur des prédicats exacts. Des détails concernant une implantation efficace de cette approche seront donnés dans le chapitre 4 sur les filtres, l'idée générale étant d'effectuer de manière approchée les calculs à l'intérieur du prédicat, tout en contrôlant l'erreur, et de ne faire appel au calcul multiprécision, que dans les cas (supposés rares) où l'approximation n'est pas suffisante pour déterminer le signe.

2.4 Conclusion

Le calcul exact est donc un concept général qui permet de garantir la robustesse des algorithmes géométriques, et donc leur déploiement par une utilisation fiable en pratique.

Des premières approches basées sur l'utilisation de types de nombres exacts, aux approches plus fines qui garantissent l'exactitude au niveau des prédicats, de nombreuses possibilités ont été développées, plus ou moins performantes et généralisables.

On note aussi le traitement particulier nécessaire aux algorithmes qui effectuent des constructions en cascade, qui nécessitent, afin d'être traités efficacement, l'usage de types de nombres filtrés.

Les deux chapitres qui suivent traitent de méthodes particulières pour l'implantation de prédicats exacts.

Chapitre 3

Arithmétique entière modulaire

Une méthode bien connue pour calculer de manière exacte le déterminant des matrices à coefficients entiers est l'utilisation de l'arithmétique modulaire [Knu98, Lau82, DST88]. La méthode s'applique aussi au calcul sur des polynômes, donc convient bien au calcul des prédicats en général. L'un des inconvénients traditionnels de cette méthode est le calcul du signe des expressions, pour lequel nous proposons plusieurs algorithmes simples et efficaces.

3.1 Éléments d'arithmétique modulaire

3.1.1 Relation de congruence

Soient a et b deux entiers et soit p un entier positif.

Définition 7 (Relation de Congruence) *On définit la relation de congruence de la manière suivante. On dit que a et b sont congrus modulo p et on note $a \equiv b \pmod{p}$ si l'une des propositions équivalentes suivantes est vérifiée :*

$a - b$ est divisible par p , ou bien

a et b ont même reste (dans $[0..p - 1]$) dans la division entière par p

La relation de congruence est une relation d'équivalence, et donc induit un ensemble quotient, qui regroupe les p classes de nombres qui sont congrus entre eux. On note ces classes $\overline{0}, \overline{1}, \dots, \overline{p - 1}$ (regroupant les entiers dont le reste de la division entière par p vaut respectivement $0, 1, \dots, p - 1$) et leur ensemble est noté $\mathbb{Z}/p\mathbb{Z}$.

Définition 8 (Addition) *On définit l'opération d'addition sur $\mathbb{Z}/p\mathbb{Z}$ de la façon suivante :*

$\overline{c} = \overline{a} + \overline{b}$, où c est le reste (dans $[0..p - 1]$) de la division entière de $a + b$ par p .

On remarque qu'on peut effectuer la somme $a + b$ en prenant à la place de a et b n'importe quel entier de la même classe, puisque tous les entiers de la classe de a sont de la forme $a + kp$ ($k \in \mathbb{Z}$), et que $(a + kp) + (b + k'p) = (a + b) + (k + k')p$, et que $(a + b)$ et $(a + b) + (k + k')p$ sont congrus modulo p .

On définit de la même manière la soustraction, qui est l'opération inverse de l'addition qui est commutative, donc $(\mathbb{Z}/p\mathbb{Z}, +)$ est un groupe abélien.

On définit aussi de la même manière la multiplication, qui s'avère être associative et commutative. On obtient donc l'anneau des entiers modulo p : $(\mathbb{Z}/p\mathbb{Z}, +, \times)$.

Propriété 2 *Tout élément de $\mathbb{Z}/p\mathbb{Z}$ dont un quelconque représentant est premier avec p est inversible dans $\mathbb{Z}/p\mathbb{Z}$.*

Cette propriété peut se démontrer de manière constructive en utilisant l'algorithme d'Euclide étendu que nous verrons plus loin. En corollaire, on remarque que si p est premier, alors $\mathbb{Z}/p\mathbb{Z}$ est un corps (tout élément non nul est inversible).

Dans la suite, nous considérons les représentants des classes de $\mathbb{Z}/p\mathbb{Z}$ qui sont dans $[-\frac{p-1}{2}.. \frac{p-1}{2}]$ si p est impair et $[-\frac{p}{2}.. \frac{p}{2} - 1]$ si p est pair (ensemble que nous notons $(-\frac{p}{2}; \frac{p}{2})$), plutôt que dans $[0..p-1]$, pour une raison liée à l'implantation que nous verrons bientôt. De plus, par commodité, on note $a = b \bmod p$, l'unique entier a dans $(-\frac{p}{2}; \frac{p}{2})$ tel que $a \equiv b \bmod p$.

3.1.2 Division modulaire

Comme on le verra plus tard pour le calcul du signe du déterminant, il est parfois utile d'effectuer des divisions modulaires, ce qui est possible pour tout résidu non nul si et seulement si p est premier. Les deux méthodes suivantes, qui donnent l'inverse modulaire, sont utilisables :

Petit théorème de Fermat

On peut utiliser le petit théorème de Fermat qui affirme que pour tout entier premier p , on a :

$$\forall a, a^p \equiv a \bmod p$$

Donc si $a \neq 0$, $a^{p-2} \equiv a^{-1} \bmod p$.

L'algorithme de division revient donc à une simple exponentiation, qui se calcule en temps $O(\log p)$, dès lors que la multiplication est effectuée en un temps constant.

Algorithme d'Euclide étendu

La deuxième méthode est basée sur l'égalité de Bézout (ou algorithme d'Euclide étendu), qui étant donnés a et b , détermine u et v tels que :

$$au + bv \equiv 1 \bmod p$$

Donc en prenant $b \equiv 0 \bmod p$, l'algorithme conduit à $u \equiv a^{-1} \bmod p$.

Cette méthode est aussi utilisable lorsque p n'est pas premier, mais a est premier avec p . La complexité est aussi en $O(\log p)$, mais cette deuxième méthode s'avère plus efficace en pratique comme nous le verrons plus loin.

3.1.3 Notations

Nous introduisons ici quelques notations qui serviront dans la suite de ce chapitre, pour un entier k que nous supposons fixé :

$$\begin{aligned} (m_i)_{i=1}^k &: \text{moduli, nombres entiers premiers entre eux} \\ m^{(j)} &= \prod_{i=1}^j m_i \\ m &= \prod_{i=1}^k m_i = m^{(k)} \\ v_i &= \prod_{j \neq i} m_j = \frac{m}{m_i} \\ w_i &= v_i^{-1} \pmod{m_i} \end{aligned}$$

Note: v_i est premier avec m_i , d'où l'existence de l'inverse w_i .

3.1.4 Calcul modulaire

On remarque que l'anneau $\mathbb{Z}/p\mathbb{Z}$ permet de calculer des expressions polynomiales dont on sait a priori que le résultat est dans $(-\frac{p}{2}; \frac{p}{2})$, et ce, même dans le cas où l'un des calculs intermédiaires sortirait de l'intervalle $(-\frac{p}{2}; \frac{p}{2})$ si l'on effectuait le calcul dans \mathbb{Z} .

Exemple 6 *Je désire calculer le résultat de l'expression polynomiale $2 + 3 - 4$, dont je sais qu'il est dans $[-1..1]$. Je considère donc la même opération dans $\mathbb{Z}/3\mathbb{Z}$:*

- je considère les classes des entrées : $2 \in \bar{2}$, $3 \in \bar{0}$, $4 \in \bar{1}$.
- j'effectue les opérations dans $\mathbb{Z}/3\mathbb{Z}$: $(\bar{2} + \bar{0}) - \bar{1} = \bar{2} - \bar{1} = \bar{1}$
- j'obtiens le résultat dans \mathbb{Z} en considérant le représentant dans $[-1..1]$ de $\bar{1}$, c'est-à-dire 1.

L'avantage est que les dépassements de capacité sur les calculs intermédiaires n'ont pas à être traités. Mais là où le calcul modulaire prend tout son intérêt est lorsque l'on prend en compte la propriété d'isomorphisme d'anneaux suivante :

$$(\mathbb{Z}/m\mathbb{Z}, +, \times) \sim \prod_{i=1}^k (\mathbb{Z}/m_i\mathbb{Z}, +, \times)$$

Cette propriété s'énonce aussi sous la forme du théorème suivant :

Théorème 1 (Restes Chinois) *Soit (x_1, \dots, x_k) un k -uplet tel que $\forall i, x_i \in (-\frac{m_i}{2}; \frac{m_i}{2})$. Alors il existe un unique entier $x \in (-\frac{m}{2}; \frac{m}{2})$, tel que $\forall i, x_i \equiv x \pmod{m_i}$.*

On dit que l'ensemble des **résidus** (x_1, \dots, x_k) est la représentation modulaire de x correspondant à l'ensemble des **moduli** (m_1, \dots, m_k) . Les correspondances entre les deux représentations sont données par les formules suivantes :

$$\forall i, x_i = x \bmod m_i$$

et la formule de Lagrange qui donne x en fonction des x_i :

$$x = \left(\sum_{i=1}^k ((x_i w_i) \bmod m_i) v_i \right) \bmod m$$

où v_i et w_i sont les quantités définies précédemment.

Si on désire calculer une expression polynomiale sur \mathbb{Z} dont on sait que le résultat est dans $(-\frac{m}{2}; \frac{m}{2})$, on peut la calculer sur $\mathbb{Z}/m\mathbb{Z}$. Le théorème des restes Chinois nous permet de dire qu'il suffit alors de la calculer sur les anneaux $\mathbb{Z}/m_i\mathbb{Z}$, grâce à l'isomorphisme. On peut ensuite obtenir une représentation traditionnelle du résultat en utilisant la formule de Lagrange.

Schématiquement, on peut manipuler des entiers x dont on sait qu'ils sont plus petits que $m/2$ en valeur absolue, par leurs résidus (x_1, \dots, x_k) modulo (m_1, \dots, m_k) . Comme nous le verrons dans la section suivante, nous considérons comme moduli en pratique des nombres premiers inférieurs à 2^{27} . Cela permet de manipuler des valeurs allant jusqu'au produit de tous ces nombres, qui est de l'ordre de 2^{108} (d'après la distribution asymptotique des nombres premiers), ce qui suffit amplement pour nos applications.

L'avantage de considérer des moduli de précision simple pour la machine est que les opérations de base (addition, multiplication moduli m_i) se font en temps constant (même si au niveau interne du processeur, une multiplication prendra $\log(27)$ étapes, le temps extérieur est généralement fixe, entre 1 et 3 cycles selon la machine). Donc le temps total pour une addition, soustraction et multiplication en représentation modulaire est exactement proportionnel à k , et le code est très facilement parallélisable.

Le principal inconvénient de cette méthode de calcul est qu'elle nécessite de connaître m a priori, c'est-à-dire une borne sur le résultat, car il est difficile de détecter les dépassements de capacité. C'est un sérieux obstacle à l'usage du calcul modulaire comme représentation interne pour un type de nombres entier multiprécision : il faut connaître une borne sur la valeur à calculer.

3.1.5 Test d'égalité

Certains prédicats (le test de colinéarité par exemple) ne font que tester la nullité d'une expression, ou l'égalité de deux expressions. Dans un tel cas, l'arithmétique modulaire est très efficace, car on remarque que :

$$x = 0 \iff \forall i, x_i = 0$$

Tester si x est nul revient donc à tester si tous les x_i sont nuls. Si l'un des x_i est non nul, alors x est non nul et de fait, on n'est pas tenu de calculer les autres x_i , ce qui augmente considérablement les performances, compte tenu des remarques probabilistes suivantes.

Si x est uniformément distribué dans $(-\frac{m}{2}; \frac{m}{2})$, alors x_i est uniformément distribué dans $(-\frac{m_i}{2}; \frac{m_i}{2})$, donc la probabilité que x_i soit nul vaut $1/m_i$. Comme les m_i sont premiers entre eux, cette remarque se généralise à : la probabilité que $\forall j \leq i, x_j = 0$ est de $1/m^{(i)}$.

On en déduit donc aussi un test d'égalité probabiliste. L'arithmétique modulaire fonctionne dans ce cas exactement comme une fonction de hachage.

3.2 Implantation

3.2.1 Utiliser l'unité flottante

Nous avons choisi d'implanter les primitives du calcul modulaire en utilisant le calcul flottant, plus précisément les doubles de la norme IEEE.

Les avantages par rapport à l'unité de calcul entière sont la portabilité, ainsi que l'observation que de plus en plus de constructeurs optimisent leur unité de calcul flottant, par rapport aux unités de calcul entier.

L'unité entière est généralement limitée à 32 bits de précision, alors que les doubles peuvent stocker 53 bits. On peut généralement étendre la précision des entiers à 64 bits pour certaines opérations sur certaines machines, mais le résultat n'est pas facilement portable.

3.2.2 Choix des moduli

Nous avons choisi comme moduli des nombres premiers, principalement pour que la division modulaire soit possible dans tous les cas. Ils sont choisis les plus grands possibles de manière à minimiser leur nombre k pour une borne de calcul donnée, et cependant majorés par 2^{27} , de sorte que les opérations suivantes soient optimisées (i.e. se calculent de manière exacte avec des doubles, avant d'effectuer une réduction modulaire) :

- l'addition et la soustraction
- la multiplication
- le déterminant 2×2 (l'expression $ad - bc$) qui est la brique de base de l'élimination Gaussienne, comme nous le verrons au chapitre 5.
- la réduction modulaire (l'opération qui calcule le résidu correspondant à la valeur donnée, dans $(-\frac{m_i}{2}; \frac{m_i}{2})$).

Nous avons considéré des résidus signés, plutôt que positifs, afin de gagner encore un bit de précision, ils sont donc majorés par 2^{26} en valeur absolue, ce qui permet d'effectuer un déterminant 2×2 de manière exacte avec des doubles (qui stockent des entiers exacts s'ils sont inférieurs à 2^{53}), ainsi que bien sûr les additions, soustractions et multiplications qui nécessitent moins de bits.

Considérer le déterminant 2×2 comme une opération de base permet d'effectuer moins de réductions modulaires durant son calcul.

3.2.3 Réduction modulaire

Une fois qu'une opération de base ci-dessus a été effectuée, et que l'on a donc son résultat exact dans un double, il faut effectuer une réduction modulaire afin que le résidu de l'opération soit plus petit que 2^{26} en valeur absolue.

	PC	Sun
Addition	2.6	6.7
Multiplication	6.4	8.5
Bézout	100	138
Fermat	260	310

FIG. 3.1 – Temps des primitives modulaires

Il y a deux méthodes possibles pour réduire x_i modulo m_i :

– Après une simple addition ou soustraction :

$$\begin{cases} x_i + m_i & \text{si } x_i < -2^{26} \\ x_i - m_i & \text{si } x_i > 2^{26} \\ x_i & \text{sinon} \end{cases}$$

– Après une multiplication ou un déterminant 2×2 ($|x_i| < 2^{53}$) :

$$x_i - m_i \times ((x_i/m_i + 3.2^{51}) - 3.2^{51})$$

Il est aussi possible de remplacer la division ci-dessus par une multiplication par l'inverse de m_i qui peut être pré-calculé (de manière approchée mais la précision est suffisante). L'addition puis la soustraction de la quantité 3.2^{51} , permet d'effectuer une troncature, qui calcule simplement la partie entière de l'expression, pourvue qu'elle soit plus petite que 2^{51} en valeur absolue (ce qui est le cas ici).

3.2.4 Temps de calcul des primitives

La figure 3.1 donne les temps de calcul en secondes pour 10^8 itérations des opérations modulaires suivantes, lorsque le modulo vaut $p = 134217649$ (environ 2^{27}) :

- une addition
- une multiplication
- une inversion selon la méthode de Bézout
- une inversion selon la méthode de Fermat

La complexité de l'algorithme de Bézout dépend des valeurs d'entrée, nous avons donné les temps maximaux observés. On se référera à l'annexe B pour les détails concernant les plate-formes de tests utilisées.

3.2.5 Parallélisation des calculs

Dans le cas des déterminants de grande dimension (supérieure à 10), comme on en trouve dans l'algorithme du simplexe par exemple, ou de grosses expressions de manière générale, il peut être utile de répartir le calcul des différents résidus de l'expression sur différents processeurs, lorsqu'on a une machine multi processeurs.

Il est facile de distribuer ainsi les calculs en utilisant du code réparti sur plusieurs processus travaillant en mémoire partagée («threads»).

3.3 Calcul du signe

Nous allons décrire maintenant plusieurs méthodes de calcul du signe d'une expression polynomiale représentée de manière modulaire [BEPP97b, BEPP99], étape importante dans le calcul des prédicats. Ce calcul est bien moins aisé qu'un test d'égalité; la méthode classique qui consiste à convertir la représentation modulaire en une représentation binaire est très coûteuse [HP94].

3.3.1 Méthode de Lagrange (élimination récursive des moduli)

Nous présentons ici les idées qui conduiront à l'algorithme 1.

La première méthode est basée sur le fait que si $x \in (-\frac{m}{2}; \frac{m}{2})$, alors sa valeur est donnée par la formule de Lagrange :

$$x = \left(\sum_{i=1}^k ((x_i w_i) \bmod m_i) v_i \right) \bmod m$$

Plutôt que de calculer directement x avec des entiers multiprécision, ce qui serait très coûteux, on peut remarquer qu'en divisant l'égalité par m , on obtient :

$$\frac{x}{m} = \text{frac} \left(\sum_{i=1}^k \frac{(x_i w_i) \bmod m_i}{m_i} \right)$$

où $\text{frac}(y)$ désigne la partie fractionnaire du réel y , comprise dans l'intervalle $(-\frac{1}{2}; \frac{1}{2})$.

Cette expression a le même signe que x . Pour aller plus vite, le principe consiste à évaluer cette expression de manière approchée en précision fixe, avec b bits de précision ($b = 53$ en pratique puisqu'on utilise des doubles). On note S la valeur approchée ainsi calculée. Tous les termes de la somme sont dans l'intervalle $(-\frac{1}{2}; \frac{1}{2})$, et on note ε_i l'erreur absolue commise sur le calcul de la somme partielle à l'étape i . On obtient donc récursivement :

$$\begin{aligned} \varepsilon_1 &= 2^{-b-1} \\ \varepsilon_i &= \varepsilon_{i-1} + 2^{-b-1} + 2^{-b} \end{aligned}$$

où le terme 2^{-b-1} représente l'erreur commise durant le calcul d'un terme de la somme, et 2^{-b} représente l'erreur rajoutée en faisant la somme. Donc l'erreur absolue finale, qui borne la différence entre S et x/m est :

$$\varepsilon_k = (3k - 2)2^{-b-1}$$

Alors, de trois choses l'une :

- soit $\varepsilon_k < |S| < \frac{1}{2} - \varepsilon_k$, alors le signe de x est donné par le signe de S . C'est ce cas qui est attendu le plus souvent, et qui permet de conclure immédiatement sur le signe de x .

- soit $\frac{1}{2} - \varepsilon_k \leq |S|$. On peut supprimer cette possibilité en rajoutant une légère contrainte sur les hypothèses de départ sur x , on admet que :

$$|x| < m \left(\frac{1}{2} - \varepsilon_k \right)$$

On verra plus tard une méthode pour traiter ce cas sans rajouter cette hypothèse supplémentaire.

- soit $|S| \leq \varepsilon_k$, alors on ne peut pas en déduire directement le signe de x . Mais on peut en revanche en déduire que $|\frac{x}{m}| < 2\varepsilon_k$, soit $|x| < 2\varepsilon_k m_k m^{(k-1)}$. On remarque que si $2\varepsilon_k m_k < \frac{1}{2} - \varepsilon_{k-1}$ (ce qui est toujours le cas dans notre implantation), alors $|x| < m^{(k-1)}(\frac{1}{2} - \varepsilon_{k-1})$, ce qui signifie que $k - 1$ moduli auraient en fait suffit pour calculer le signe de x , et que la borne sur x a été surévaluée. On peut donc reprendre la même méthode, mais en remplaçant k par $k - 1$, et ainsi de suite. Au pire, on termine lorsque $k = 1$, $x = x_1$, où l'on obtient le signe directement.

Nous présentons donc l'algorithme en utilisant les notations suivantes :

$$\begin{aligned} v_i^{(j)} &= \prod_{\substack{1 \leq \ell \leq j \\ \ell \neq i}} m_\ell \\ w_i^{(j)} &= \left(v_i^{(j)} \right)^{-1} \bmod m_i \\ S^{(j)} &= \text{frac} \left(\sum_{i=1}^j \frac{x_i w_i^{(j)} \bmod m_i}{m_i} \right) \end{aligned}$$

de sorte que $v_i = v_i^{(k)}$, $w_i = w_i^{(k)}$ et $S = S^{(k)}$. Tous les calculs de cet algorithme sont effectués en utilisant le calcul flottant avec une précision de b bits.

On note que $w_i^{(j)} = w_i^{(j+1)} m_{j+1} \bmod m_i$, donc le i -ème terme de $S^{(j)}$ peut être calculé à partir du i -ème terme de $S^{(j+1)}$. Donc uniquement les $w_i^{(k)}$ sont utiles à l'algorithme. Comme k est spécifié dans les entrées, une table de taille quadratique semble nécessaire pour stocker les valeurs pré-calculées, mais elle peut cependant être évitée au prix d'un surcoût en temps négligeable (voir la remarque plus loin).

Algorithme 1 (Élimination récursive des moduli) : Étant donnés (x_1, \dots, x_k) , calcule le signe de x .

Données pré-calculées: $m_i, w_i^{(k)}, \varepsilon_i$, pour tout $1 \leq i \leq k$

Entrée: entiers k et $x_i \in \left(-\frac{m_i}{2}, \frac{m_i}{2}\right)$, pour tout $1 \leq i \leq k$

Sortie: signe de x , l'unique solution de $x_i \equiv x \bmod m_i$ telle que $|x| \leq \frac{m^{(k)}}{2}(1 - \varepsilon_k)$

1. Soit $j \leftarrow k + 1$, $z_i = x_i w_i^{(k)} \bmod m_i$ pour tout $1 \leq i \leq k$

2. Répéter $j \leftarrow j - 1$, $S^{(j)} \leftarrow \text{frac} \left(\sum_{i=1}^j \frac{z_i}{m_i} \right)$

si $|S^{(j)}| < \varepsilon_j$ et $j > 0$ alors $z_i \leftarrow z_i m_j \bmod m_i$ pour tout $1 \leq i < j$ jusqu'à ce que $|S^{(j)}| > \varepsilon_j$ ou $j = 0$

3. Si $j = 0$ retourner " $x = 0$ "
4. Si $S^{(j)} > 0$ retourner " $x > 0$ "
5. Si $S^{(j)} < 0$ retourner " $x < 0$ "

Le lemme suivant borne la complexité de l'algorithme 1.

Lemme 1 *L'algorithme 1 calcule le signe de x à partir de ses résidus x_i en utilisant au pire $\frac{k(k+1)}{2}$ multiplications modulaires, additions et divisions flottantes, ainsi que $k + 2$ comparaisons flottantes.*

Preuve : Les m_i et les $w_i^{(j)}$ sont pré-calculés dans une table, et supposés être accessibles en temps constant. À l'étape 2, un total de j multiplications modulaires, additions et divisions flottantes (incluant le calcul de la partie fractionnaire) et une comparaison sont effectuées. \square

Dans de nombreux cas pratiques, on essaiera de calculer au mieux une borne sur $|x|$ qui permettra de minimiser k . Si x est du même ordre de grandeur que $m^{(k)}$, alors très peu d'itérations de l'algorithme seront nécessaires, et on obtient alors un comportement linéaire.

On évite ainsi le temps quadratique d'une reconstruction binaire complète.

3.3.2 Méthode de Lagrange généralisée

Nous montrons que la méthode précédente est en fait un cas particulier de la suivante. Soit

$$\Sigma^{(0)} = S = S^{(k)} = \text{frac} \left(\sum_{i=1}^k \frac{(x_i w_i) \bmod m_i}{m_i} \right)$$

Cette quantité est calculée à la première étape de l'algorithme 1. Si la valeur calculée de $\Sigma^{(0)}$ est plus petite que ε_k , alors $|\Sigma^{(0)}| < 2\varepsilon_k$. Donc $|x| < 2m\varepsilon_k$, et nous pouvons donc multiplier $x_i w_i$ par

$$\alpha_k = \left\lfloor \frac{\frac{1}{2}(1 - \varepsilon_k)}{2\varepsilon_k} \right\rfloor,$$

afin d'obtenir $(x_i w_i \alpha_k) \bmod m_i$ pour tout $i = 1, \dots, k$. Ceci peut se faire facilement en pré-calculant $\alpha_k \bmod m_i$ pour tout i . Nous calculons donc ensuite :

$$\Sigma^{(1)} = \text{frac} \left(\sum_{i=1}^k \frac{(x_i w_i \alpha_k) \bmod m_i}{m_i} \right),$$

et plus généralement :

$$\Sigma^{(j)} = \text{frac} \left(\sum_{i=1}^k \frac{(x_i w_i \alpha_k^j) \bmod m_i}{m_i} \right),$$

où l'on suppose que $\alpha_k \bmod m_i$ est pré-calculé pour tout $i = 1, \dots, k$. Il est aisé de voir que le nombre d'itérations de ce processus est $\lceil \log m / \log \alpha_k \rceil \leq k$, parce que $1 \leq |x| \leq$

$m^{(k)} \leq 2^{k(b/2+1)}$, et x_i est multiplié par α_k à chaque itération. Ce nombre est plus petit que $\lceil k/2 \rceil + 1$ pour tous les cas pratiques. Donc l'algorithme 2 utilise encore $\Theta(k^2)$ opérations dans le cas le pire, mais en pratique dans les cas courants (x du même ordre de grandeur que $m^{(k)}$), seulement k opérations.

On obtient donc l'algorithme suivant :

Algorithme 2 : Méthode de Lagrange généralisée.

Connaissant $x_i = x \bmod m_i$ pour tout $1 \leq i \leq k$, calcule le signe de x

Données pré-calculées: $m_i, w_i, \varepsilon_k, \alpha_k \bmod m_i$, pour tout $i = 1, \dots, k$

Entrées: entiers k et $x_i \in \left(-\frac{m_i}{2}; \frac{m_i}{2}\right)$ pour tout $i = 1, \dots, k$

Sortie: signe de x , l'unique solution de $x_i = x \bmod m_i$ dans $\left(-\frac{m^{(k)}}{2}; \frac{m^{(k)}}{2}\right)$

Pré-conditions: $|x| \leq \frac{m^{(k)}}{2}(1 - \varepsilon_k)$ et $x \neq 0$

1. Soit $j \leftarrow -1$, $z_i = x_i w_i^{(j)} \bmod m_i$ pour tout $1 \leq i \leq k$

2. Répéter $j \leftarrow j + 1$, $\Sigma^{(j)} \leftarrow \text{frac} \left(\sum_{i=1}^k \frac{z_i}{m_i} \right)$

si $|\Sigma^{(j)}| \leq \varepsilon_k$ alors $z_i \leftarrow z_i \alpha_k \bmod m_i$ pour tout $1 \leq i \leq k$, jusqu'à ce que $|\Sigma^{(j)}| > \varepsilon_k$

3. Si $\Sigma^{(j)} > 0$ retourner " $x > 0$ "

4. Si $\Sigma^{(j)} < 0$ retourner " $x < 0$ "

Remarque 5 L'algorithme 1 correspond à un choix de m_j à la place de α_k à l'étape j . Cela simplifie le calcul en éliminant un modulo à chaque itération, mais cela fait plus d'itérations. En multipliant par α_k , on effectue moins d'itérations, mais chaque itération est faite sur les k moduli. C'est pourquoi nous appelons l'algorithme 2 une généralisation.

3.3.3 Variante éliminant ε_k dans la borne

Les précédents algorithmes 1 et 2 ont le (petit) inconvénient de requérir que $|x| \leq m(\frac{1}{2} - \varepsilon_k)$.

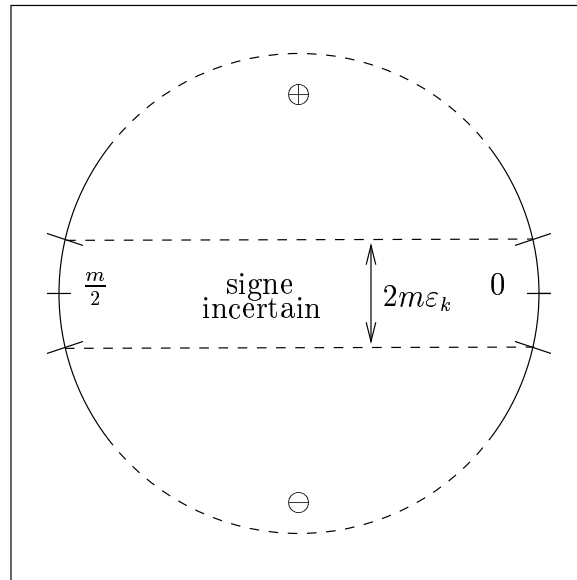
Pour pouvoir se ramener à l'hypothèse simple $|x| < \frac{m^{(k)}}{2}$, il faut traiter séparément le cas où $|S| > 1/2 - \varepsilon_k$, car alors on ne peut pas garantir que x et S ont le même signe (voir figure 3.2).

Dans ce cas, cela signifie que $|x| > m^{(k)}(1/2 - 2\varepsilon_k)$. En posant $x' = (m^{(k)}/2 - x) \bmod m^{(k)}$, on remarque que $\text{sign}(x') = \text{sign}(x)$ et que $|x'| < m^{(k)}\varepsilon_k$, donc on se ramène à effectuer l'algorithme précédent pour x' . Il suffit de calculer $x'_i = (m^{(k)}/2 - x_i) \bmod m_i$, ce qui est rapide puisque les $m^{(k)}/2 \bmod m_i$ pour tout i peuvent être pré-calculés.

Note : dans la version généralisée, cette remarque permet de prendre $\alpha_k = \lfloor \frac{\varepsilon_k}{4} \rfloor$ directement.

3.3.4 Méthode de Newton

Une version incrémentale de la démonstration constructive du théorème des restes Chinois, nommée méthode de Newton, est décrite ici.

FIG. 3.2 – *Le cercle modulaire*

Son avantage principal est que la variante probabiliste correspondante ne requiert pas de borne *a priori* sur la magnitude de x , c'est-à-dire le nombre de moduli à employer. Elle ne nécessite aussi qu'une quantité linéaire (en k) de données pré-calculées. Un autre avantage est que tous les calculs se font modulo les m_i , et que donc on n'a pas à solliciter l'unité flottante si on choisit d'implanter les primitives modulaires en utilisant le calcul entier, ce qui peut être un avantage.

Soit $x^{(j)} = x \bmod m^{(j)}$, pour $j = 1, \dots, k$, de sorte que $x^{(1)} = x_1$ et $x = x^{(k)}$. Soit $y_1 = x_1$, et pour tout $j = 2, \dots, k$,

$$\begin{aligned} t_j &= w_j^{(j)} = (m^{(j-1)})^{-1} \bmod m_j, \\ y_j &= (x_j - x^{(j-1)}) t_j \bmod m_j \in \left(-\frac{m_j}{2}; \frac{m_j}{2}\right). \end{aligned}$$

Alors (voir, par exemple, [Knu98, Lau82]), pour tout $j = 2, \dots, k$,

$$x^{(j)} = (x^{(j-1)} + y_j m^{(j-1)}) \bmod m^{(j)}. \quad (3.1)$$

Clairement, cela mène à un calcul incrémental de la solution $x = x^{(k)}$; nous allons voir ci-dessous comment on peut exploiter cela pour terminer cette interpolation le plus tôt possible.

Il est évident que lorsque $y_j \neq 0$, alors le signe de $x^{(j)}$ est le même que celui de y_j , puisque $|x^{(j-1)}| \leq m^{(j-1)}/2$. Si $y_j = 0$, le signe de $x^{(j)}$ est le même que celui de $x^{(j-1)}$, pour $j \geq 2$, et le signe de $x^{(1)} = x_1 = y_1$ est connu directement. Si $y_j = 0$ pour tout j , cela implique évidemment que $x = 0$.

En déroulant l'équation (3.1) dans la définition de y_j , on montre que les y_j vérifient l'identité de type Horner pour tout $j = 2, \dots, k$:

$$y_j = (x_j - (x^{(j-2)} + y_{j-2} m^{(j-2)})) t_j \bmod m_j$$

$$\begin{aligned} & \vdots \\ & = (x_j - x_1 - m_1(y_2 + m_2(\cdots(y_{j-2} + m_{j-2}y_{j-1})\cdots)))t_j \bmod m_j \end{aligned}$$

Tous les calculs sont faits modulo m_j . En conséquence, ils peuvent être effectués en utilisant uniquement l'arithmétique modulaire de précision fixe. Donc il suffit de supposer que $|x| < m^{(k)}/2$.

Algorithme 3 : Calcule le signe de x , connaissant $x \bmod m_i$, par la méthode de Newton incrémentale

Données pré-calculées : m_j, t_j , pour tout $1 \leq j \leq k$

Entrée : entiers k et $x_i \in (-\frac{m_i}{2}, \frac{m_i}{2})$ pour tout $i = 1, \dots, k$

Sortie : signe de x , où x est l'unique solution de $x_i = x \bmod m_i$ dans $(-\frac{m^{(k)}}{2}, \frac{m^{(k)}}{2})$

1. Soit $y_1 \leftarrow x_1, j \leftarrow 1$ et $s \leftarrow \text{sign}(y_1)$

2. Répéter $j \leftarrow j + 1$,

$$y_j \leftarrow (x_j - x_1 - m_1(y_2 + m_2(\cdots(y_{j-2} + m_{j-2}y_{j-1})\cdots)))t_j \bmod m_j,$$

jusqu'à ce que $j = k$. Pour tout j , $s \leftarrow \text{sign}(y_j)$

3. Selon la valeur de s ($-1, 0$, ou 1), retourner " $x < 0$ ", " $x = 0$ ", ou " $x > 0$ ", respectivement.

Lemme 2 L'algorithme 3 calcule le signe de x connaissant ses résidus x_i en utilisant au pire $\frac{k(k-1)}{2}$ multiplications modulaires, $\frac{k(k-1)}{2}$ additions modulaires, et k comparaisons.

Preuve : Pour tout $j = 2, \dots, k$, il y a $j - 1$ additions et multiplications modulaires. Il y a une comparaison pour tout $j = 1, \dots, k$. \square

On obtient donc un algorithme de complexité quadratique, ce qui est moins bon que la méthode de Lagrange, du moins dans le cas (attendu) où celle-ci est quasi linéaire.

On remarque que lorsque $|x|$ est nettement plus petit que m , c'est-à-dire lorsque l'on a surestimé la borne, les derniers y_j (ceux pour lesquels $|x| < m^{(j-1)}$) vont être nuls. La variante probabiliste de cette méthode s'appuie sur cette remarque pour arrêter le calcul plus tôt, dès qu'un certain nombre d' y_j sont nuls (elle présume que tous les suivants le seront aussi). Elle permet ainsi de ne pas requérir de borne sur x . De plus, c'est dans ces mêmes conditions ($|x|$ surestimé) que la méthode de Lagrange a une moins bonne complexité, les deux méthodes présentées sont ainsi complémentaires.

3.3.5 Variantes probabilistes

D'une manière similaire à ce que nous avons vu pour le test d'égalité, nous avons développé des variantes probabilistes des méthodes décrites précédemment pour la reconstruction du signe en représentation modulaire.

Ces méthodes ont une meilleure complexité, acquise au prix d'un risque d'erreur mesuré (fonction de la distribution des données) et généralement très faible. Comme tous les

algorithmes probabilistes, ils peuvent soulever des doutes quant à leur usage dans le cadre des méthodes robustes où s'inscrit cette thèse.

Nous avons pris le parti de ne pas décrire ces méthodes en détail ici, mais nous engageons le lecteur intéressé à se référer aux publications pour plus de détails [BEPP97a, BEPP99].

3.3.6 Parallélisation du calcul du signe

On se reportera à [BEPP97b, BEPP99, BEPP97a] pour l'utilisation possible de plusieurs processeurs dans le calcul du signe. L'utilité de ces méthodes est négligeable puisque grâce aux méthodes de reconstruction de signe que nous avons développées, c'est maintenant principalement le calcul des x_i qui domine le temps de calcul global, les méthodes précédemment décrites étant suffisantes dans la plupart des cas, pour ne plus nécessiter d'améliorations.

3.4 Calcul de la valeur approchée

Retrouver la valeur exacte en représentation binaire d'une expression dont on connaît la représentation modulaire n'est pas utile dans le cadre des prédicats, mais elle peut être utile pour les constructions géométriques.

En particulier, certaines méthodes (snap rounding [Hob93, Mil95]) garantissent certaines propriétés de robustesse topologiques lorsque les constructions sont correctement arrondies aux sommets les plus proches sur une grille (typiquement des coordonnées à virgule flottante). En cela, ces méthodes nécessitent des garanties similaires à celles offertes par la norme IEEE 754 concernant les opérations arithmétiques de base. Cependant, elles requièrent ces arrondis au plus proche pour le groupement d'opérations définissant la construction.

Nous avons dérivé des méthodes de reconstruction de signe en représentation modulaire décrites précédemment, des méthodes permettant d'obtenir ces garanties de constructions arrondies au plus proche [BP97b].

Le but est donc de calculer l'approximation à 2^{-b} près de x représenté par (x_1, \dots, x_n) . Pour cela, on remarque que l'algorithme de la méthode de Lagrange n'est pas spécifique au signe, il permet de calculer une approximation de x/m à une erreur de ε_k près.

Une fois cette approximation connue, de deux choses l'une :

- soit elle est suffisante pour en déduire une approximation de x à la précision souhaitée (en multipliant par une approximation de m),
- soit on peut soustraire cette première approximation de x en représentation modulaire (i.e. calculer la représentation modulaire de l'approximation, et la retrancher aux x_i), et calculer une approximation plus précise de la différence en utilisant la même méthode. Il n'y a plus ensuite qu'à additionner les différentes approximations flottantes dans l'ordre croissant, ce qui garantit, via les propriétés des additions de la norme IEEE 754, que le résultat final sera l'approximation au plus proche de la valeur.

3.5 Étude pratique

Pour le cas particulier des déterminants, nous renvoyons le lecteur au chapitre 5.

3.6 Conclusion

Une des applications principales du calcul modulaire est le calcul des déterminants, que nous étudierons plus en détail au chapitre 5. La bibliothèque C que nous avons écrite, et qui implante les algorithmes décrits précédemment, est utilisée pour manipuler des objets courbes de manière exacte et efficace [CKKM99].

L'autre propriété intéressante de cette représentation des entiers multiprécision est sa qualité de hachage des valeurs (des valeurs numériquement proches n'ont pas des résidus proches), ce qui en fait une bonne méthode pour les tests d'égalité de certains prédicats géométriques. Nous avons étendu le domaine d'intérêt des méthodes modulaires aux prédicats en général en introduisant des algorithmes performants de calcul du signe.

Nous avons utilisé le calcul modulaire pour représenter des entiers multiprécision, mais il est possible d'étendre ce schéma au calcul sur les rationnels [DST88].

Chapitre 4

Filtres arithmétiques

Pour pallier la lenteur du calcul exact dans l'évaluation des prédicats, et en observant que le calcul flottant approché mais rapide donne avec une grande probabilité la réponse exacte, ont été mises au point des méthodes de calcul paresseuses, dont le principe général est le suivant :

- On effectue un calcul approché des expressions numériques,
- mais en validant ou invalidant les comparaisons.
- En cas d'invalidité, il y a échec du filtre, et on a recours à un calcul plus coûteux.

Les filtres ont été étudiés suivant leur vitesse, mais aussi suivant leur probabilité de succès en fonction de la distribution des données, en particulier pour les filtres statiques dans le cas de distributions uniformes des entrées [DP98, DP99].

Nous décrivons dans ce chapitre comment ces filtres fonctionnent, en commençant par une énumération des types de filtres classiques, ainsi que par la présentation de filtres plus performants que nous avons mis au point. Puis nous parlerons des outils existants permettant de générer le code des prédicats filtrés de manière automatique, et nous présenterons notre propre outil, disponible dans CGAL.

4.1 Filtres simple précision

Le principe est donc de calculer une expression de manière approchée, mais en pouvant déterminer une précision suffisante pour confirmer ou infirmer que le signe de l'expression calculée de manière approchée est le même que celui de l'expression réelle exacte.

Pour calculer cette expression approchée, on utilise généralement une arithmétique flottante, car elle est rapide. La précision utilisée peut varier, mais toujours pour des raisons d'efficacité, on choisit de se baser sur les `double` de la norme IEEE, ce qui donne une précision de 53 bits. Les filtres basés sur une telle précision, et n'utilisant aucun calcul intermédiaire multiprécision, sont dits «simple précision».

On peut classer les différentes approches existantes en les catégories suivantes :

- les filtres statiques
- les filtres dynamiques

- les filtres semi-statiques (ou semi-dynamiques, intermédiaires)

Un filtre doit donc, à partir de données initiales, calculer une expression approchée d'une valeur réelle (ou plusieurs), et fournir une borne d'erreur permettant de garantir ou non que le signe de la valeur approchée calculée correspond au signe de la valeur réelle de l'expression.

Les deux critères importants pour juger de la qualité d'un filtre sont :

- la vitesse de calcul du prédicat en cas de succès du filtre, et
- la probabilité de succès, qui est directement reliée à la borne d'erreur que le filtre va donner.

En cas d'échec du filtre, une autre méthode doit être employée pour trouver de manière sûre le signe de l'expression. Cette méthode peut être constituée directement d'une arithmétique exacte, ou par l'introduction d'un autre type de filtre, plus lent, mais dont la probabilité de succès serait plus grande.

4.1.1 Filtres statiques

Les filtres dits *statiques* considèrent que :

- l'expression est calculée avec des `double`.
- la borne d'erreur de ce calcul est une constante.

De ce fait, le surcoût à l'exécution du prédicat par rapport à un simple calcul flottant est que le calcul du signe (comparaisons par rapport à zéro) est remplacé par une comparaison par rapport à une constante. Ainsi, le code du prédicat, qui pour du simple calcul flottant ferait :

```
...
if (x>0) return 1;
if (x<0) return -1;
return 0;
```

serait remplacé par :

```
...
if (x> C) return 1;
if (x<-C) return -1;
/* Traitement de l'échec du filtre: */
...
```

où `C` est la borne d'erreur associée au calcul de l'expression `x` dont le prédicat retourne le signe.

Bien entendu, ce type de filtre ne peut fonctionner que sous certaines conditions, celles qui permettent de majorer l'erreur dans tous les cas par une constante. Ces conditions sont :

- les divisions ne sont pas autorisées,
- les entrées du prédicat doivent être majorées par une constante.

En effet, comme nous le verrons, majorer l'erreur absolue d'une division nécessite de minorer le dénominateur, ce qui nécessiterait des contraintes supplémentaires. De plus, comme nous l'avons déjà remarqué, il est toujours possible de se passer de divisions dans les prédicats, ce n'est donc pas une importante restriction.

La restriction concernant le majorant sur les bornes des entrées peut être gênante pour certaines applications, mais convenir à d'autres.

On calcule la borne d'erreur de l'expression en utilisant les propriétés de la norme IEEE 754 qui garantissent, pour chacune des opérations de base (+, -, ×, /, √), une erreur absolue maximale égale à $\frac{1}{2}\text{ulp}()$. Nous détaillons maintenant ce calcul des bornes d'erreur, en fonction de l'expression à calculer et des bornes sur les entrées.

Notations: x est une variable réelle, \mathbf{x} sa valeur calculée avec des doubles, \mathbf{e}_x et \mathbf{b}_x sont des doubles tels que :

$$\begin{cases} \mathbf{e}_x \geq |x - \mathbf{x}| \\ \mathbf{b}_x \geq |\mathbf{x}| \end{cases}$$

Initialement, on peut obtenir par un arrondi au plus proche (si les valeurs ne tiennent pas exactement dans un double) :

$$\begin{cases} \mathbf{b}_x = |\mathbf{x}| \\ \mathbf{e}_x = \frac{1}{2}\text{ulp}(\mathbf{x}) \end{cases}$$

Addition et soustraction

La propagation d'erreur sur une addition $z = x + y$ est la suivante :

$$\begin{cases} \mathbf{b}_z = \mathbf{b}_x + \mathbf{b}_y \\ \mathbf{e}_z = \mathbf{e}_x \bar{+} \mathbf{e}_y \bar{+} \frac{1}{2}\text{ulp}(z) \end{cases}$$

En effet :

$$\begin{aligned} |z - \mathbf{z}| &= \underbrace{|(z - (x + y))|}_{=0} + \underbrace{|((x + y) - (\mathbf{x} + \mathbf{y}))|}_{\leq \mathbf{e}_x + \mathbf{e}_y} + \underbrace{|((\mathbf{x} + \mathbf{y}) - \mathbf{z})|}_{\leq \frac{1}{2}\text{ulp}(z)} \\ &\leq \mathbf{e}_x \bar{+} \mathbf{e}_y \bar{+} \frac{1}{2}\text{ulp}(z) \end{aligned}$$

De même pour la soustraction.

Multiplication

La propagation d'erreur sur une multiplication $z = x \times y$ est la suivante :

$$\begin{cases} \mathbf{b}_z = \mathbf{b}_x \times \mathbf{b}_y \\ \mathbf{e}_z = \mathbf{e}_x \bar{\times} \mathbf{e}_y \bar{+} \mathbf{e}_y \bar{\times} |\mathbf{x}| \bar{+} \mathbf{e}_x \bar{\times} |\mathbf{y}| \bar{+} \frac{1}{2}\text{ulp}(z) \end{cases}$$

En effet :

$$\begin{aligned} |z - \mathbf{z}| &= \underbrace{|(z - (x \times y))|}_{=0} + \underbrace{|((x \times y) - (\mathbf{x} \times \mathbf{y}))|}_{=(x-x)(y-y) - (x-x) \times y - (y-y) \times x} + \underbrace{|((\mathbf{x} \times \mathbf{y}) - \mathbf{z})|}_{\leq \frac{1}{2}\text{ulp}(z)} \end{aligned}$$

$$\leq e_x \bar{\times} e_y \bar{+} e_x \bar{\times} y \bar{+} e_y \bar{\times} x \bar{+} \frac{1}{2} \text{ulp}(z)$$

Division

La propagation d'erreur sur une division $z = x/y$ est la suivante :

$$\begin{cases} b_z = b_x / (|y| - e_y) \\ e_z = (e_x \bar{\times} |y| \bar{+} e_y \bar{\times} |x|) \bar{+} (|y| \bar{\times} (|y| - e_y)) \bar{+} \frac{1}{2} \text{ulp}(z) \end{cases}$$

En effet :

$$\begin{aligned} |z - z| &= \left| \underbrace{(z - (x/y))}_{=0} + \underbrace{((x/y) - (\mathbf{x}/\mathbf{y}))}_{= \frac{(x-x)y - (y-y)x}{yy}} + \underbrace{((\mathbf{x}/\mathbf{y}) - z)}_{\leq \frac{1}{2} \text{ulp}(z)} \right| \\ &\leq (e_x \bar{\times} y \bar{+} e_y \bar{\times} x) \bar{+} (y \bar{\times} (y - e_y)) \bar{+} \frac{1}{2} \text{ulp}(z) \end{aligned}$$

Dans ces conditions, il est nécessaire d'avoir une borne inférieure sur $(|y| - e_y)$.

Racine carrée

La propagation d'erreur sur une racine carrée $z = \sqrt{x}$ est la suivante :

$$\begin{cases} b_z = \sqrt{b_x} \\ e_z = \sqrt{e_x} \bar{+} \frac{1}{2} \text{ulp}(z) \end{cases}$$

En effet :

$$\begin{aligned} |z - z| &= \left| \underbrace{(z - \sqrt{x})}_{=0} + \underbrace{(\sqrt{x} - \sqrt{\mathbf{x}})}_{\leq \sqrt{|x-\mathbf{x}|}} + \underbrace{(\sqrt{\mathbf{x}} - z)}_{\leq \frac{1}{2} \text{ulp}(z)} \right| \\ &\leq \sqrt{e_x} \bar{+} \frac{1}{2} \text{ulp}(z) \end{aligned}$$

On notera que $\sqrt{e_x} \bar{+} \frac{1}{2} \text{ulp}(z)$ n'est pas une très bonne borne d'erreur relative. On peut faire mieux en connaissant une borne inférieure sur \mathbf{x} , puisque dans ce cas :

$$\begin{aligned} |\sqrt{x} - \sqrt{\mathbf{x}}| &= \sqrt{\mathbf{x}} \left(\sqrt{\frac{x}{\mathbf{x}}} - 1 \right) \\ &\leq \sqrt{\mathbf{x}} \left(\sqrt{\frac{\mathbf{x} + e_x}{\mathbf{x}}} - 1 \right) \\ &\leq \sqrt{\mathbf{x}} \left(\sqrt{1 + \frac{e_x}{\mathbf{x}}} - 1 \right) \\ &\leq \frac{e_x}{2\sqrt{\mathbf{x}}} \end{aligned}$$

Ce qui donne finalement : $e_z = e_x \bar{+} 2\sqrt{\mathbf{x}} \bar{+} \frac{1}{2} \text{ulp}(z)$.

Automatiser le calcul des bornes d'erreur

Afin de faciliter le calcul de ces bornes d'erreur statiques, nous utilisons un type de nombres (classe C++) qui contient comme données la borne b et l'erreur e dont nous venons de parler. Les opérateurs sont surchargés pour ce type de nombres, et utilisent les formules ci-dessus. Lorsque l'on exécute le code d'un prédicat sur ce type de nombres, en initialisant les valeurs aux bornes correspondant aux majorants des entrées, l'appel à un opérateur de comparaison affiche la borne d'erreur correspondante (C).

Cette méthode a ses limites, par exemple il est difficile de gérer les branchements à l'exécution du prédicat. C'est néanmoins une méthode qui est un premier pas vers une automatisation complète pour certains prédicats. Ce type d'outil sera utilisé aussi pour les filtres statiques adaptatifs.

Bilan

Finalement, on constate que le filtre statique est optimal du point de vue de la vitesse, puisque son surcoût par rapport au calcul flottant se résume à comparer à une constante au lieu de zéro. Par contre, le calcul de la borne d'erreur peut être trop grossier pour certaines applications, et il requiert des majorants sur les entrées. Il n'est pas non plus très facile d'usage puisque le calcul de la borne dépend vraiment de l'application, et doit être déterminée par le programmeur.

Une étude des probabilités de succès des filtres statiques dans des cas simples a été effectuée [DP98, DP99]. Elle montre que les cas d'échecs du filtre pour les prédicats *orientation* et *in_circle* 2d et 3d sont négligeables dans le cas d'une distribution aléatoire bornée des données.

Nous passons maintenant à un type de filtre bien plus performant pour le calcul de la borne d'erreur et plus souple d'utilisation, mais qui est plus lent.

4.1.2 Filtres dynamiques

Afin d'obtenir une estimation plus fine de l'erreur, mais surtout afin d'avoir un outil facilement utilisable, les filtres *dynamiques* ont été mis au point. Le principe de base est que l'erreur est calculée à l'exécution pour chaque opération de base (+, -, ×, /, √). Ainsi, chaque variable est en fait une paire de valeurs : une valeur approchée, à laquelle on associe une erreur, et c'est ce type de nombres qui est utilisé à l'exécution du prédicat.

Deux codages possibles existent :

- $X = (x, \varepsilon_X)$, tel qu'utilisé par le filtre dynamique incorporé aux `leda_real`.
- $X = [\underline{x}; \bar{x}]$, qui est la base du calcul par intervalles.

L'avantage de ce type de filtre est qu'il n'est pas nécessaire d'étudier la formule auparavant pour produire une borne d'erreur, et de fait l'usage en est beaucoup plus aisé.

La borne d'erreur peut être aussi petite qu'on le souhaite, tout dépend de la méthode utilisée pour son calcul, mais pour chaque opération de base, il est possible de propager la plus petite erreur codable en précision simple. Les divisions sont aussi gérées.

L'inconvénient est que cette approche est plus coûteuse que les filtres statiques puisque tout le calcul d'erreur a lieu durant l'exécution de chaque appel au prédicat.

Le premier codage ($X = (x, \varepsilon_X)$) revient simplement à effectuer le calcul d'erreur décrit pour les filtres statiques, à l'exécution.

Le deuxième codage ($X = [\underline{x}; \bar{x}]$), qui utilise le calcul par intervalles décrit plus loin (cf 4.1.3), améliore encore la vitesse ainsi que la borne d'erreur. Certains cas dégénérés peuvent même être certifiés sans coût supplémentaire, puisque l'arithmétique d'intervalles permet de détecter automatiquement, dans certains cas, la nullité d'une expression de manière naturelle (les deux bornes de l'intervalle sont égales à 0), alors que pour le même cas, le premier codage ne permet pas naturellement de faire ça (puisqu'il fait appel à la fonction `ulp` qui n'est jamais nulle).

Les deux codages sont bien sûr à peu près équivalents, et il paraît difficile d'en imaginer un troisième tout en conservant une précision unitaire. La différence majeure est que l'un est plus adapté au calcul sur les intervalles, alors que l'autre est plus classique pour le calcul d'erreur.

Nous allons maintenant détailler le fonctionnement du filtre dynamique basé sur l'arithmétique d'intervalles que nous avons codé dans CGAL.

4.1.3 L'arithmétique d'intervalles

L'analyse par intervalles date des années 60 avec la thèse de Moore [Moo66], puis elle s'est développée [HHKR95] en se basant sur un outil qui est l'arithmétique d'intervalles.

Nous ne parlerons pas en détail ici de l'analyse par intervalles en général car ce n'est pas exactement ce qui nous concerne. Le but de notre utilisation de l'arithmétique d'intervalles est de contrôler les propagations d'erreurs faites durant un calcul flottant.

Description

Nous utilisons une arithmétique sur les intervalles que nous notons :

$$X = [\underline{x}; \bar{x}]$$

Pour des intervalles dont les bornes sont des réels, la définition des opérations (parmi $+$, $-$, \times) est la suivante :

$$A \mathcal{OP} B = \{a \mathcal{OP} b \mid a \in A, b \in B\}$$

La propriété fondamentale vérifiée par cette définition est la suivante :

Propriété 3 (Inclusion) *Pour toute opération arithmétique \mathcal{OP} (parmi $+$, $-$, \times) effectuée sur les intervalles A et B , on a :*

$$a \in A, b \in B \Rightarrow (a \mathcal{OP} b) \in (A \mathcal{OP} B)$$

Cette propriété est importante car en la propageant à une composition de telles opérations (définissant une expression polynomiale), on peut conclure que si le résultat d'un tel calcul est un intervalle ne contenant pas zéro, alors on peut être sûr du signe du résultat réel exact.

C'est cette propriété que l'implantation vérifiera, plutôt que la définition mathématique initiale, qui nécessiterait de stocker des bornes de manière exacte.

Utilisation des modes d'arrondi de la norme IEEE 754

On remarque qu'en effectuant successivement une même opération sur les mêmes arguments, d'abord en arrondissant vers $-\infty$, puis vers $+\infty$, on obtient deux valeurs flottantes qui forment un intervalle contenant le résultat de l'opération sur les réels. Dans le cas où l'intervalle est réduit à une valeur, on connaît même ce réel de manière exacte.

On en déduit une manière simple de calculer sur les intervalles, qui garantit la propriété d'inclusion et qui de plus offre des intervalles de taille minimale pour chaque opération.

Nous définissons ainsi les opérations sur les intervalles $X = [\underline{x}; \bar{x}]$ et $Y = [\underline{y}; \bar{y}]$:

$$\begin{aligned}
 X + Y &= [\underline{x} + \underline{y}; \bar{x} + \bar{y}] \\
 X - Y &= [\underline{x} - \bar{y}; \bar{x} - \underline{y}] \\
 X * Y &= [\min(\underline{x}\underline{y}, \underline{x}\bar{y}, \bar{x}\underline{y}, \bar{x}\bar{y}); \max(\underline{x}\bar{y}, \bar{x}\underline{y}, \bar{x}\bar{y}, \bar{x}\bar{y})] \\
 X/Y &= \begin{cases} [-\infty; +\infty] & \text{si } 0 \in Y \\ [\min(\underline{x}/\underline{y}, \underline{x}/\bar{y}, \bar{x}/\underline{y}, \bar{x}/\bar{y}); \max(\underline{x}/\bar{y}, \bar{x}/\underline{y}, \bar{x}/\bar{y}, \bar{x}/\bar{y})] & \text{sinon} \end{cases} \\
 \sqrt{X} &= \begin{cases} [\sqrt{\underline{x}}; \sqrt{\bar{x}}] & \text{si } 0 \leq x \\ [0; \sqrt{\bar{x}}] & \text{si } \underline{x} < 0 \leq \bar{x} \\ \text{non défini} & \text{sinon} \end{cases}
 \end{aligned}$$

Remarque 6 *On n'implante pas la multiplication et la division selon les formules mathématiques ci-dessus, mais en faisant des tests préliminaires sur les signes des bornes de X et Y , qui vont fournir directement les multiplications ou divisions à effectuer dans chaque cas.*

Nous avons donc défini un ensemble d'opérations vérifiant la propriété d'inclusion. En appliquant ceci au calcul des prédicats, il suffit de remplacer dans le code du prédicat, le calcul du signe de l'exemple précédent par :

```

...
if (x.inf > 0) return 1;
if (x.sup < 0) return -1;
if ( (x.sup == 0) && (x.inf == 0) ) return 0;
/* Traitement de l'échec du filtre: */
...

```

où $\mathbf{x} = [\mathbf{x}.inf; \mathbf{x}.sup]$.

Ceci se fait de manière très simple en C++, en définissant un type de nombres (classe C++) codant l'intervalle par ses deux bornes en `double`, et en surchargeant les opérateurs.

Propriété de symétrie

En pratique, changer le mode d'arrondi correspond à une instruction particulière de la machine, qu'il peut s'avérer coûteux d'appeler souvent, ce qui serait le cas si on appliquait

les définitions ci-dessus directement. On peut contourner ce problème dans la quasi totalité des cas en utilisant les propriétés de symétrie suivantes :

$$\begin{aligned} a \underline{+} b &= -((-a) \overline{-} b) \\ a \underline{-} b &= -(b \overline{-} a) \\ a \underline{\times} b &= -((-a) \overline{\times} b) \\ a \underline{/} b &= -((-a) \overline{/} b) \end{aligned}$$

Ces propriétés triviales permettent d'enchaîner les calculs sur les intervalles en conservant d'un bout à l'autre le mode d'arrondi dirigé vers $+\infty$. Il est en effet beaucoup moins coûteux de calculer l'opposé d'une valeur plutôt que de changer de mode d'arrondi. Cela laisse aussi un champ beaucoup plus grand au compilateur pour optimiser le code.

Cette remarque ne s'applique pas à la racine carrée, ce qui n'est pas très gênant, puisque cette opération prend de toute façon un temps non négligeable par rapport aux changements de modes d'arrondis, et qu'elle est utilisée moins souvent.

Précision des calculs sur les intervalles

Le but du calcul sur les intervalles est de pouvoir certifier le signe d'une valeur, ce qui est possible si cet intervalle ne contient pas zéro. On a donc intérêt à essayer de minimiser la taille des intervalles pour le calcul d'une expression donnée, afin d'avoir une probabilité maximale qu'il ne contienne pas zéro, si cette expression n'est pas réellement nulle.

Pour de petites expressions comme les prédicats classiques, la propagation d'erreur est rarement importante. Par contre, certains calculs comme les signes de déterminants de grandes dimensions, qui peuvent intervenir dans notre domaine, doivent faire l'objet d'algorithmes spéciaux afin d'éviter que les intervalles ne soient trop gros pour avoir un quelconque intérêt.

Nous introduisons maintenant une mesure qualitative de la taille des intervalles obtenus lors du calcul d'une expression. On note $\text{Ideg}(\epsilon) \in \mathbb{Z}$, le degré d'une expression (note : ce degré n'est pas le même que le degré algébrique introduit au chapitre 2). On le définit inductivement en utilisant les règles suivantes :

$$\begin{aligned} \text{Ideg}(\text{constante}) &= 0 \\ \text{Ideg}(X + Y) &= \max(\text{Ideg}(X), \text{Ideg}(Y)) \\ \text{Ideg}(X - Y) &= \max(\text{Ideg}(X), \text{Ideg}(Y)) \\ \text{Ideg}(X \times Y) &= 1 + \max(\text{Ideg}(X), \text{Ideg}(Y)) \\ \text{Ideg}(X/Y) &= 1 + \max(\text{Ideg}(X), \text{Ideg}(Y)) \\ \text{Ideg}(\sqrt{X}) &= \text{Ideg}(X) \end{aligned}$$

Le degré est une mesure approximative du nombre de bits de précision perdus lors du calcul d'une expression en utilisant les intervalles. On peut s'en convaincre en déduisant des formules données précédemment pour le calcul des bornes d'erreur statiques, des bornes d'erreur relatives.

Si le degré d'une expression est du même ordre de grandeur que le nombre de bits de précision du type de nombres représentant les bornes des intervalles (53 pour des double),

	PC			Sun		
	IA	Adv	Adv/FP	IA	Adv	Adv/FP
Addition	16.7	5.8	2.7	39.2	11.5	3.7
Soustraction	15.2	5.9	2.7	37.1	11.6	4.1
Multiplication	18.8	8.1	3.8	40.9	13.5	4.3
Division	29.6	19.2	3.0	52.3	29.0	3.6
Racine carrée	41.3	35.2	3.1	66.4	43.1	2.7

FIG. 4.1 – Temps des primitives sur les intervalles

alors on peut considérer que le calcul par intervalles risque de donner trop souvent un résultat trop gros pour être utile.

Temps de calcul des primitives

La figure 4.1 donne les temps de calcul en secondes pour 10^8 itérations des opérations arithmétiques de base sur les intervalles. Les colonnes "IA" concernent le type de nombres qui inclus les changements de modes d'arrondis (type `Interval_nt` de CGAL). Les colonnes "Adv" concernent le type de nombres qui ne fait pas ces changements, et suppose que le mode d'arrondi est déjà positionné vers $+\infty$ (type `Interval_nt_advanced` de CGAL). Les colonnes "Adv/FP" donnent le rapport entre le temps "Adv" et le temps de la même opération faite sur des doubles.

Notons que les temps de calcul sur de si petites boucles peuvent montrer quelques effets de bord dus aux optimisations du compilateur ou du processeur. En particulier, nous avons préféré rajouter un appel de fonction, plutôt que d'avoir l'opération en ligne dans la boucle, afin d'éviter les propagations de constante à l'intérieur du code, et l'expulsion de certains invariants hors de la boucle. Nous avons aussi rajouté l'option de compilation `-fomit-frame-pointer` qui est adaptée aux petites fonctions. Ces mesures sont donc données surtout à titre indicatif, et nous accordons plus d'importance aux mesures qui englobent d'avantage de code, comme un prédicat ou un algorithme complet.

On constate néanmoins que les opérations de base sont au pire cinq fois plus lentes. Sur PC, nous avons aussi activé une protection qui est inutile dans les cas où il n'y a pas de risque de dépassement de capacité de l'exposant, ce qui résulte en un ralentissement d'environ 10%, pour les deux colonnes concernant les intervalles.

Temps de calcul des prédicats

La figure 4.2 donne les temps de calcul en secondes pour 10^7 itérations des prédicats classiques sur les intervalles. La signification des colonnes est la même que précédemment. Les prédicats utilisés sont les suivants, nous donnons le nombre d'opérations arithmétiques qu'ils contiennent dans l'implantation de CGAL :

- le prédicat orientation 2D (4 +, 2 ×, 1 compare)
- le prédicat orientation 3D (14 +, 9 ×, 1 sign)
- le prédicat in_circle 2D (14 +, 10 ×, 1 compare)
- le prédicat in_sphere 3D (37 +, 40 ×, 1 sign)

	PC			Sun		
	IA	Adv	Adv/FP	IA	Adv	Adv/FP
Orientation 2D	9.3	3.0	3.5	22.6	6.6	3.8
Orientation 3D	37.3	13.4	3.9	83.9	21.6	3.6
In_circle 2D	36.6	11.1	4.0	89.7	17.5	4.3
In_sphere 3D	343	38.9	6.21	382	79.3	8.5
In_sphere 3D(*)	136	42.0	6.70	291	79.4	8.5

FIG. 4.2 – Temps des prédicats sur les intervalles

On constate que pour le prédicat `in_sphere 3D`, le rapport entre "IA" et "Adv" passe à environ 10 alors qu'il n'est que de 3 ou 4 pour les autres prédicats. Ceci est dû à la grande taille de la fonction (plus de 20Ko), qui fait que le code qui doit être exécuté durant la boucle ne tient plus dans le cache de niveau 1 du processeur, d'où un ralentissement significatif. La dernière ligne "(*)" représente le temps de calcul du même prédicat, mais lorsque le code de la multiplication n'est pas mis en ligne, ce qui réduit considérablement la taille de la fonction, et rétablit un niveau de performance normal.

Bilan

Les filtres dynamiques sont à l'opposé des filtres statiques. Ils sont moins rapides, mais produisent une borne d'erreur beaucoup plus fine, en particulier le filtre basé sur l'arithmétique d'intervalles.

Il est à noter que d'autres implantations de l'arithmétique d'intervalles existent [Knü]. Nous avons préféré recoder un tel type de nombres, parce que les implantations courantes n'étaient pas en C++, ou n'utilisaient pas la propriété de symétrie. De plus, nous devions modifier le code afin de gérer le comportement en cas de comparaison non sûre. L'implantation dans CGAL occupe moins de 1000 lignes de code et supporte plusieurs architectures.

Nous présentons maintenant des approches intermédiaires entre les deux principaux types de filtres que nous venons de détailler.

4.1.4 Filtres semi-statiques

Les filtres *semi-statiques* ont été utilisés pour pallier l'inconvénient principal des filtres statiques : pour fonctionner, ils doivent connaître un majorant des entrées, afin d'évaluer une borne sur l'erreur absolue, ce qui n'est pas très souple à l'utilisation.

Le principe de fonctionnement du filtre semi-statique est le même que celui du filtre statique, excepté qu'il ne requiert pas de borne sur les entrées du prédicat, il la calcule à l'exécution. Il en déduit la borne d'erreur correspondante (toujours à l'exécution) de manière simple (généralement, c'est un polynôme simple).

Les calculs de bornes d'erreurs sont exactement les mêmes que pour les filtres statiques, exceptés qu'ils sont basés sur des données connues à l'exécution. Ce calcul peut être simplifié pour être rendu plus rapide, au prix d'une borne d'erreur un peu plus large.

Certains outils [BFS98] permettent de générer le code des prédicats basés sur ce type de filtre.

Nous avons choisi de ne pas implanter cette approche dans CGAL, au profit de la suivante.

4.1.5 Filtres statiques adaptatifs

L'inconvénient des filtres semi-statiques est qu'ils doivent recalculer la borne d'erreur à chaque appel du prédicat, or cela peut être contourné en faisant la remarque suivante : la borne d'erreur est une fonction croissante des majorants des entrées, donc si on a calculé les majorants et la borne d'erreur correspondant à un appel particulier du prédicat, et qu'à l'appel suivant, les entrées sont plus petites que les majorants précédemment calculés, alors la borne d'erreur de l'appel précédent est toujours valable (mais peut-être un peu trop large) pour les entrées actuelles.

De ce fait, on n'est pas obligé de calculer à chaque appel du prédicat un nouveau majorant sur les entrées et une borne d'erreur associée, il suffit de vérifier que les entrées vérifient toujours la condition sur le majorant précédent. Dans le cas contraire, il faut recalculer le majorant réel (le maximum des valeurs absolues des entrées), et la nouvelle borne d'erreur associée.

Un tel schéma conduit à une borne d'erreur croissante au fil des appels du prédicat, ce qui peut conduire à un taux important d'échecs du filtre. Si cela se produit, il convient de recalculer un majorant plus petit.

Donc, en fonction de la distribution des données : au pire, cette approche aura la complexité d'un filtre semi-statique, et au mieux, elle aura celle d'un filtre statique (c'est-à-dire optimale) plus une vérification des majorants (une valeur absolue et une comparaison par argument), et c'est évidemment cette dernière qu'on attend en moyenne.

Cette méthode est aussi souple d'emploi qu'un filtre semi-statique, elle ne requiert pas d'initialisation des bornes des entrées. On ne peut cependant pas utiliser de divisions.

On peut donc considérer que ce type de filtre est adaptatif (aux bornes d'entrée), par rapport à un filtre statique, et qu'il fait un calcul de bornes d'erreur paresseux par rapport à un filtre semi-statique.

Nous avons implanté cette approche dans CGAL. L'opération de réajustement est délicate à automatiser dans le cadre générique de CGAL, elle est décrite plus loin.

4.2 Approches à précision adaptative

Les premiers résultats dans ce domaine ont essayé de mélanger un filtre et une arithmétique exacte. On notera que c'est l'approche suivie par LN, les flottants à précision adaptative de Shewchuk [She96] et les expansions flottantes [Dau99, DF99], les `Fixed_precision_nt` programmés par Olivier Devillers dans CGAL, l'arithmétique redondante [Pio95], ainsi que les méthodes spécifiques au calcul de signe de déterminant [ABD⁺95, Cla92, BY97, BY96].

On peut considérer que ces approches rejoignent les types de nombres filtrés comme `leda_real`, si ce n'est qu'elles peuvent tirer parti de la nature isolée du code des prédicats,

et donc il n'y a pas nécessairement besoin de gérer la mémoire de façon dynamique. En clair, le code du filtre et de la partie exacte utilisée en cas d'échec du filtre sont en ligne dans la même fonction.

Les approches que nous avons implantées dans cette thèse sont plus radicales, dans le sens où elles perdent un éventuel pré-calcul effectué durant le filtre qui pourrait resservir en cas d'échec du filtre. Le raisonnement derrière cette position est que :

- le filtre est censé suffire dans pratiquement tous les cas, et conserver un résultat intermédiaire dans tous les cas a un certain coût, qui peut être désavantageux globalement.
- d'autre part, le temps de calcul du filtre est censé être bien inférieur au temps de calcul en cas d'échec, donc on peut se permettre de refaire les mêmes calculs que ceux qui sont faits dans le filtre, sans surcoût global significatif.
- la simplicité de la programmation est un élément important, et séparer clairement les concepts facilite l'implantation.

Bien évidemment, ces critères ne sont pas universels, et il se peut que certaines conditions pratiques liées à l'application, fassent qu'une méthode devienne meilleure qu'une autre.

4.3 Génération automatique

Pour éviter la tâche fastidieuse d'évaluation des bornes d'erreurs des filtres statiques, ont été mis au point des programmes d'aide ou *compilateurs de prédicats*. Ces outils aident au calcul des bornes d'erreur, et à la production du code des prédicats filtrés. LN et EXPCOMP en sont des exemples. Nous avons développé dans CGAL des méthodes aussi simples que possible pour les types de filtres dynamiques (basés sur l'arithmétique d'intervalles), statique adaptatif et statique, sachant qu'il est intéressant dans le cadre de CGAL de trouver des solutions aussi génériques que possible.

L'avantage d'une approche basée sur une étude avant l'exécution est que le temps de calcul est amoindri à l'exécution, par rapport à des méthodes qui stockent de manière dynamique les informations nécessaires à un éventuel calcul exact en cas d'échec du filtre, comme c'est le cas pour les `leda_real` de LEDA, ou bien les rationnels paresseux [BJMM93].

4.3.1 LN (Little Numbers)

C'est le premier générateur de filtres statiques, fait par Fortune et Van Wyk [FW93, FV93, FV96]. Le principe de son fonctionnement est le suivant :

LN prend en entrée le code d'un prédicat, exprimé dans un langage particulier, similaire à du C restreint, limité aux opérations $+$, $-$, \times . Les divisions, racines carrées et boucles ne sont pas autorisées. LN lit ce code, et génère en sortie une fonction représentant le prédicat filtré et exact, grâce à l'inclusion d'une arithmétique exacte entière paresseuse, qui calcule les poids forts d'abord.

Le filtre implanté par LN est d'abord de type statique, il requiert une borne sur les entrées, qui de plus doivent être des entiers. Puis c'est une approche à précision adaptative

qui est utilisée, c'est-à-dire qui calcule de plus en plus précisément la valeur du résultat tant qu'il ne peut pas décider son signe.

La méthode ne fonctionne pas pour un déterminant de dimension supérieure ou égale à 6, car elle produit un code gigantesque : c'est comme si le calcul multiprécision était entièrement mis en ligne (inline) dans le prédicat, avec les boucles déroulées.

4.3.2 EXPCOMP (Expression Compiler)

Dans la lignée de LN, Stefan Funke a développé (pour LEDA) un programme similaire, EXPCOMP [BFS98], avec les fonctionnalités supplémentaires :

- il accepte du code C++ en entrée, mais toujours pas de boucles.
- plusieurs types de filtres sont implantés, et pour certains, les divisions et les racines carrées sont autorisées.

Ce travail n'est cependant pas encore terminé.

4.3.3 L'approche prise pour CGAL

Les approches précédentes ont l'inconvénient pratique suivant : elles doivent comprendre le code des prédicats jusqu'au niveau des opérations arithmétiques de plus bas niveau, ce qui fait qu'elles ont à peu près la complexité d'un compilateur. Sous une forme ou sous une autre, elles nécessitent un « parser » (EXPCOMP utilise Yacc).

Pour CGAL, nous avons cherché une solution plus simple, au risque d'être moins efficace. La solution trouvée est finalement basée sur un script simple en PERL d'environ 500 lignes donné en Annexe A, et permet de générer le code des prédicats du noyau de CGAL, ainsi que les prédicats de l'utilisateur.

Les filtres disponibles sont au choix parmi ceux décrits précédemment :

- statique
- statique adaptatif
- dynamique (basé sur l'arithmétique d'intervalles décrite précédemment).

Nous allons détailler maintenant le fonctionnement de ces filtres.

Prédicats dans CGAL

Les prédicats dans CGAL sont sous la forme de fonctions génériques (template), par exemple le code du prédicat d'orientation 2-dimensionnel est :

```

template < class FT >
inline
Orientation
orientationC2(const FT &px, const FT &py,
              const FT &qx, const FT &qy,
              const FT &rx, const FT &ry)
{
    return sign_of_determinant2x2(px-rx,py-ry,qx-rx,qy-ry);
}

```

Les codes de ces prédicats sont isolés dans des fichiers particuliers, afin de faciliter leur traitement par des outils automatisés.

Gestion des cas d'échec du filtre

C++ permet de gérer les cas d'échec des filtres de manière très pratique et efficace, via le mécanisme des exceptions [Str86]. Ce mécanisme n'est pas spécifique à C++, il est aussi disponible dans d'autres langages comme C via `setjmp/longjmp`.

Considérons l'exemple suivant :

```
void f(double x)
{
    if (fabs(x) < C)
        throw exception_t();
}

void g(...)
{
    ...
    try {
        f(x);
    } catch (exception_t) {
        ...
    }
    ...
}
```

Dans la fonction `g`, l'appel à `f` peut provoquer le lancement d'une exception, si la condition ($|x| < C$) est vérifiée. Dans ce cas, l'exécution du bloc de programme situé dans le `try` est arrêtée, la pile d'exécution est dépilée, et l'exécution reprend au début du bloc `catch`. Si la condition sur `x` n'est pas vérifiée, `throw` n'est pas appelé, et l'exécution termine le bloc `try`, et ne rentre pas dans le `catch`.

Donc nous utilisons schématiquement :

- un bloc `try` pour représenter le calcul du filtre (qui peut appeler des sous-fonctions),
- s'il y a impossibilité de garantir le signe d'une expression, alors une exception est lancée, c'est le cas d'échec du filtre,
- un bloc `catch` qui code la méthode utilisée pour traiter l'échec, par exemple le recours à un type de nombres exact.

L'alternative pour gérer les échecs du filtre serait d'avoir une valeur de retour particulière de la fonction `sign()`, signifiant une erreur. Cela nécessiterait de changer beaucoup de code, car il faudrait gérer cette erreur même dans les sous-fonctions appelées. L'utilisation des exceptions permet de localiser les changements uniquement au niveau des calculs de signe.

Classes annexes

Des types de nombres annexes sont utilisés.

Le filtre dynamique utilise la classe `Interval_nt_advanced`, qui représente l'arithmétique d'intervalles décrite précédemment. Les opérateurs arithmétiques sont donc surchargés afin de vérifier la propriété d'inclusion, et les opérateurs de comparaison lancent une exception (de type `unsafe_comparison`), lorsque la comparaison ne peut pas être certifiée (échec du filtre).

Les filtres statiques et statiques adaptatifs doivent être capables de calculer des bornes d'erreurs en fonction des majorants. Bien sûr, le filtre statique n'effectue ce calcul qu'une seule fois à l'initialisation du programme, avec une borne spécifiée par le programmeur, et le filtre statique adaptatif doit pouvoir effectuer ce calcul quand c'est nécessaire. Pour cela, la classe `Static_filter_error` est un type de nombres qui permet de calculer les bornes d'erreurs décrites dans la section 4.1.1. L'utilisation exacte de cette classe sera décrite en détail plus loin.

Interface d'utilisation

Le script PERL va donc fournir, pour chaque prédicat générique d'origine, un ensemble de fonctions correspondant à ces mêmes prédicats, mais filtrés. Plutôt que de donner un nom différent à ces prédicats filtrés, ce qui nécessiterait de changer partout le code, on utilise le même nom que le prédicat générique, mais on utilise le mécanisme de spécialisation partielle, en utilisant un type de nombres différent : `Filtered_exact`.

Ainsi, l'utilisateur n'a qu'à utiliser un type de nombres différent pour spécifier qu'il veut utiliser des prédicats filtrés et exacts à la place de la version générique.

Le type de nombres `Filtered_exact` est en fait lui-même générique (template), et a les paramètres `<CT, ET, Type, Protection>`, qui ont la signification suivante :

- `CT` est le type de la valeur contenue dans un `Filtered_exact`. Il stocke la valeur numérique de la variable. Toutes les opérations effectuées à l'extérieur des prédicats sont propagées à ce type. Et il n'a pas besoin d'être exact pour garantir l'exactitude des prédicats.
- `ET` est un type de nombres exact qui sera utilisé lorsque le filtre échoue. Dans le bloc `catch`, on appelle le prédicat sur ce type. Donc soit c'est un type de nombres exact sur lequel le prédicat générique sera appelé, et fournira la réponse exacte, soit c'est lui-même un type pour lequel le prédicat générique est surchargé.
- `Type` peut prendre les deux valeurs `Static` ou `Dynamic` (valeur par défaut).
- `Protection` peut prendre deux valeurs `Protected` (valeur par défaut) ou `Advanced`.

Exemples :

- `Filtered_exact<double, leda_real, Static, Protected>` est un type de nombres qui a les mêmes propriétés qu'un `double` à l'extérieur des prédicats, et sur lequel les prédicats sont filtrés par un filtre statique adaptatif. En cas d'échec du filtre, le prédicat est appelé sur le type `leda_real`.
- `Filtered_exact<double, leda_real, Static, Advanced>` est la même chose, mais le filtre est purement statique.

- `Filtered_exact<double, leda_real, Dynamic>` est la même chose, mais le filtre est dynamique.
- `Filtered_exact<double, Filtered_exact<double, leda_real>, Static>` est un filtre cascadé: tout d'abord un filtre statique est essayé, en cas d'échec un filtre dynamique est utilisé, et en cas de deuxième échec, le type de nombres exact `leda_real` est utilisé.

Ce schéma nécessite des fonctions de conversion du type CT vers ET. De plus, pour le calcul filtré, le filtre dynamique a besoin de convertir le type CT vers le type `Interval_nt_advanced` (c'est-à-dire donner un encadrement), sur lequel le prédicat générique est appelé. Les détails de l'interface et les contraintes exactes sont spécifiées dans la documentation.

Ce schéma permet donc d'avoir un usage complètement générique. On peut même l'utiliser dans le cas où CT est un type exact multiprécision: on obtient ainsi une évaluation plus rapide des prédicats (même s'ils sont déjà exacts si CT est un type exact).

Filtre dynamique

Le principe de base est de faire exécuter la fonction template originale du prédicat sur le type de nombres spécial codant l'arithmétique d'intervalles, qui détecte les calculs de signes non sûrs. Le code produit par le script à partir du prédicat générique `orientation` est le suivant :

```

template < class CT, class ET>
Orientation
orientationC2(
    const Filtered_exact <CT, ET, Dynamic, Protected> &px,
    const Filtered_exact <CT, ET, Dynamic, Protected> &py,
    const Filtered_exact <CT, ET, Dynamic, Protected> &qx,
    const Filtered_exact <CT, ET, Dynamic, Protected> &qy,
    const Filtered_exact <CT, ET, Dynamic, Protected> &rx,
    const Filtered_exact <CT, ET, Dynamic, Protected> &ry)
{
    // Save and set the rounding mode.
    FPU_CW_t backup = FPU_get_and_set_cw(FPU_cw_up);
    try
    {
        Orientation result = orientationC2(
            px.interval(),
            py.interval(),
            qx.interval(),
            qy.interval(),
            rx.interval(),
            ry.interval());
        // Restore the rounding mode.
        FPU_set_cw(backup);
        return result;
    }
    catch (Interval_nt_advanced::unsafe_comparison)
    {
        FPU_set_cw(backup);
    }
}

```

10

20

```

    return orientationC2(
        px.exact(),
        py.exact(),
        qx.exact(),
        qy.exact(),
        rx.exact(),
        ry.exact());
}
}

```

Pour ce type de filtre, le script n'a pas du tout besoin de connaître le corps de la fonction générique, mais juste l'en-tête (nom de la fonction, type de retour, nombre d'arguments...). Il est donc très simple à mettre en œuvre en utilisant les expressions régulières de PERL.

Dans le code produit, on note le changement de mode d'arrondi effectué au début (et la sauvegarde du mode courant), afin de profiter de la propriété de symétrie. Le filtre dynamique `Advanced` permet de s'affranchir de ces changements de mode d'arrondi à l'intérieur du prédicat, en demandant au programmeur de s'en occuper à l'extérieur, ce qui peut être un avantage si aucun calcul qui nécessiterait un autre mode d'arrondi n'a besoin d'être effectué pendant tout le déroulement de l'algorithme.

Les fonctions membres `.interval()` et `.exact()` du type `Filtered_exact` effectuent les conversions du type `CT`, vers les types `Interval_nt_advanced` et `ET` respectivement.

Les filtres dynamiques sont donc implantés de manière très simple.

Filtre statique adaptatif

Les filtres statiques et statiques adaptatifs utilisent des structures annexes pour chaque prédicat (`Static_Filtered_orientationC2_6`), qui servent à stocker le contexte, c'est-à-dire les bornes sur les entrées courantes (constantes dans le cas du filtre statique), ainsi que les bornes d'erreur associées (epsilons). Elles servent aussi à regrouper des variantes du prédicat qui servent à mettre à jour les bornes d'erreur. Voici le contenu de cette structure (dont le nom est formé de la façon suivante : `Static_Filtered_nom` du prédicat_nombre d'arguments) :

```

struct Static_Filtered_orientationC2_6
{
    static double _bound;
    static double _epsilon_0;

    static Orientation update_epsilon(
        const Static_filter_error &px,
        const Static_filter_error &py,
        const Static_filter_error &qx,
        const Static_filter_error &qy,
        const Static_filter_error &rx,
        const Static_filter_error &ry,
        double &epsilon_0)
    {
        typedef Static_filter_error FT;

```

```

    return Static_Filtered_sign_of_determinant2x2_4::
        update_epsilon(px-rx,py-ry,qx-rx,qy-ry, epsilon_0);
}

```

20

```

static void new_bound (const double b)
{
    _bound = b;
    (void) update_epsilon(b,b,b,b,b,b,_epsilon_0);
}

```

```

static Orientation epsilon_variant(
    const Restricted_double &px,
    const Restricted_double &py,
    const Restricted_double &qx,
    const Restricted_double &qy,
    const Restricted_double &rx,
    const Restricted_double &ry,
    const double & epsilon_0)
{
    typedef Restricted_double FT;

    return Static_Filtered_sign_of_determinant2x2_4::
        epsilon_variant(px-rx,py-ry,qx-rx,qy-ry, epsilon_0);
}
};

```

30

40

Le principe de fonctionnement du script dans le cas des filtres statiques (adaptatifs) est qu'il détermine les endroits de la fonction où sont effectuées les comparaisons, et assigne un «epsilon» pour chacun de ces endroits. Si un sous-prédicat est appelé, il compte pour le nombre d'epsilons total de sa fonction (c'est le cas de `sign_of_determinant_2x2` ici). Note: par souci de simplicité (dans un premier temps, mais ce n'est pas un obstacle fondamental de la méthode), le script ne cherche pas les comparaisons en elles-mêmes "<", "≤", etc... Mais il cherche récursivement les appels aux autres prédicats, en utilisant comme prédicats de base les fonctions `sign` et `compare`.

Pour chaque prédicat, une version «à epsilons près» est créée (`Static_Filtered_orientationC2_6::epsilon_variant`), dont le corps est le même que celui de la version générique, mais où les appels aux sous-prédicats ont reçu en paramètre les valeurs des epsilons courants correspondants au contexte du prédicat.

La version courante permet de gérer une borne unique sur toutes les entrées du prédicat (`Static_Filtered_orientationC2_6::_bound`), et un nombre de bornes d'erreur variable (une par comparaison) (`Static_Filtered_orientationC2_6::_epsilon_0...`). On peut évidemment penser à une variante qui utiliserait plusieurs majorants, dans les cas où les entrées seraient de différents ordres de grandeur.

Le code du filtre statique adaptatif lui-même est le suivant :

```

template < class CT, class ET>
Orientation
orientationC2(
    const Filtered_exact <CT, ET, Static, Protected> &px,

```

```

const Filtered_exact <CT, ET, Static, Protected> &py,
const Filtered_exact <CT, ET, Static, Protected> &qx,
const Filtered_exact <CT, ET, Static, Protected> &qy,
const Filtered_exact <CT, ET, Static, Protected> &rx,
const Filtered_exact <CT, ET, Static, Protected> &ry)
{
    10
    bool re_adjusted = false;
    const double SAF_bound = Static_Filtered_orientationC2_6::_bound;

    // Check the bounds. All arguments must be <= SAF_bound.
    if (
        fabs(px.to_double()) > SAF_bound ||
        fabs(py.to_double()) > SAF_bound ||
        fabs(qx.to_double()) > SAF_bound ||
        fabs(qy.to_double()) > SAF_bound ||
        fabs(rx.to_double()) > SAF_bound ||
        20
        fabs(ry.to_double()) > SAF_bound)
    {
    re_adjust:
        // Compute the new bound.
        double NEW_bound = 0.0;
        NEW_bound = std::max(NEW_bound, fabs(px.to_double()));
        NEW_bound = std::max(NEW_bound, fabs(py.to_double()));
        NEW_bound = std::max(NEW_bound, fabs(qx.to_double()));
        NEW_bound = std::max(NEW_bound, fabs(qy.to_double()));
        NEW_bound = std::max(NEW_bound, fabs(rx.to_double()));
        30
        NEW_bound = std::max(NEW_bound, fabs(ry.to_double()));
        // Re-adjust the context.
        Static_Filtered_orientationC2_6::new_bound(NEW_bound);
    }

    try
    {
        return Static_Filtered_orientationC2_6::epsilon_variant(
            px.dbl(),
            py.dbl(),
            qx.dbl(),
            qy.dbl(),
            rx.dbl(),
            ry.dbl(),
            Static_Filtered_orientationC2_6::_epsilon_0);
    }
    catch (Restricted_double::unsafe_comparison)
    {
        // It failed, we re-adjust once.
        if (!re_adjusted) {
            re_adjusted = true;
            goto re_adjust;
        }
        // This scheme definitely fails => exact computation.
        return orientationC2(
            px.exact(),
            py.exact(),
            qx.exact(),
            40
            50

```

```

        qy.exact(),
        rx.exact(),
        ry.exact());
    }
}

```

60

Enfin, la structure associée possède une fonction membre `update_bound()` qui prend une borne sur les entrées du prédicat en argument, et met à jour les epsilons (bornes d'erreurs) de la structure associée au prédicat.

La mise à jour est effectuée par une autre variante du code du prédicat (`Static_Filtered_orientationC2_6::update_epsilon`), qui est la même que la version `epsilon_variant`, excepté que les arguments supplémentaires sont passés par référence, et donc mis à jour récursivement.

Le travail de détection des erreurs et de mise à jour effective est donc effectué dans une version surchargée des fonctions `sign()` et `compare()` pour les types `Static_filter_error` et `Restricted_double`. Voici la structure associée au prédicat `sign`:

```

struct Static_Filtered_sign_1
{
    static double _bound;
    static double _epsilon_0;
    static Sign update_epsilon( const Static_filter_error &a, double & epsilon_0)
    {
        epsilon_0 = a.error();
        return ZERO;
    }

    static void new_bound (const double b)
    {
        _bound = b;
        (void) update_epsilon(b,_epsilon_0);
    }

    static Sign epsilon_variant( const Restricted_double &a,
                                const double & epsilon_0)
    {
        if (a.dbl()> epsilon_0) return POSITIVE;
        if (a.dbl()<=-epsilon_0) return NEGATIVE;
        if (a.dbl()==0 && epsilon_0==0) return ZERO;
        throw Restricted_double::unsafe_comparison();
    }
};

```

10

20

Cette méthode a cependant une limitation principale, qui n'est pas trop gênante en pratique : quelques rares prédicats ont des branchements dans leur code, ce qui empêche la méthode décrite plus haut de fonctionner tout le temps, puisque pour que la mise à jour des epsilons fonctionne, il faut que le flot d'exécution passe sur tout le code. Pour ces prédicats, il faut soit concevoir une méthode automatique qui exécute plusieurs fois le prédicat en passant dans toutes les branches, ce qui n'est pas évident, soit demander à

	IA/FP	SA/FP	FP
Enveloppe Convexe 2D (10 ⁶ pts)	1.36	1.24	4.52
Triangulation 2D (10 ⁴ pts)	2.40	1.71	2.29
Triangulation 2D (10 ⁵ pts)	2.00	1.52	156
Delaunay 2D (10 ⁵ pts)	1.86	1.45	29.3
Triangulation Régulière 2D (10 ⁵ pts)	1.88	—	32.1
Triangulation 3D (10 ⁴ pts)	2.27	1.28	3.51
Delaunay 3D (10 ⁴ pts)	2.51	1.28	3.35
Triangulation Régulière 3D (10 ⁴ pts)	2.15	—	4.68

FIG. 4.3 – *Effectivité des filtres en fonction de l’algorithme*

4.4 Temps de calcul

Des études ont été effectuées pour comparer le coût de diverses approches dans CGAL [Sch99]. Nous présentons ici des résultats comparatifs de différentes méthodes existantes et celles que nous avons implantées dans CGAL, sur différents algorithmes.

Nous ne donnons ces mesures que pour la plate-forme PC, car ils sont très similaires sur les deux plate-formes.

La figure 4.3 met l’accent sur la différence d’impact des filtres sur différents algorithmes de CGAL, sur des points dont les coordonnées ont des valeurs aléatoires entières sur 30 bits, uniformément distribuées.

- FP: temps total en secondes de l’algorithme, lorsqu’on utilise des doubles.
- IA/FP: rapport du temps total lorsque l’on utilise les filtres dynamiques (`Filtered_exact < double, leda_real >`), sur le temps FP.
- SA/FP: rapport du temps total lorsque l’on utilise les filtres statiques adaptatifs (`Filtered_exact < double, leda_real, Static >`), sur le temps FP.

Sur ces données particulières, le filtre dynamique n’a pas échoué une seule fois, ce qui implique a posteriori que le calcul flottant n’a pas créé d’erreur. Par contre, le filtre statique adaptatif a un taux d’échec croissant avec le nombre de points, pour les prédicats complexes (`in_circle`), ce qui s’explique en considérant que la distance moyenne entre les points diminue avec le nombre de points, ce qui augmente le taux d’échec du filtre. Le filtre dynamique est insensible à cet effet, ce qui fait qu’en le mettant en position intermédiaire entre le filtre statique et le calcul exact coûteux, on obtient un temps de calcul très bon.

Pour résumer, on peut dire le filtre statique adaptatif a un surcoût qui varie entre 1.2 et 1.7, alors que le filtre dynamique varie entre 1.3 et 2.5 selon l’algorithme, lorsque les données sont distribuées de manière à avoir un taux d’échec du filtre négligeable.

La figure 4.4 compare sur trois algorithmes, avec la même distribution de points que précédemment, la performance de différents types de nombres, par rapport aux filtres précédents. La colonne `Real/FP` mesure le rapport en temps du calcul avec le type de nombres `real` de LEDA, par rapport au calcul flottant. `Gmpz/FP` mesure la même chose

	IA/FP	SA/FP	Real/FP	Gmpz/FP
Enveloppe Convexe 2D (10^6 pts)	1.36	1.24	5.63	11.0
Delaunay 2D (10^5 pts)	1.86	1.45	9.3	19.9
Delaunay 3D (10^4 pts)	2.51	1.28	13.0	40.6

FIG. 4.4 – Comparaison des différents types de nombres

	IA	SA
Aléatoire	0	870
$\varepsilon = 2^{-5}$	0	1942
$\varepsilon = 2^{-10}$	0	662
$\varepsilon = 2^{-15}$	0	8833
$\varepsilon = 2^{-20}$	0	132153
$\varepsilon = 2^{-25}$	10	192011
$\varepsilon = 2^{-30}$	19536	308522
Grille	49756	299505

FIG. 4.5 – Taux d'échec des filtres en fonction de la distribution

pour le type Gmpz, qui est une sur-couche C++ de CGAL sur les entiers multiprécision de la bibliothèque GMP.

On constate que la performance d'un type de nombres exact et filtré (`real`) est entre 5 et 10 fois moins bonne que celle qu'on peut obtenir avec des prédicats filtrés. Ceci montre le coût de la gestion dynamique de la mémoire et des types d'opérations, par rapport à l'approche des filtres pour les prédicats entiers. Cette différence est d'autant plus marquée que la complexité du prédicat augmente. Cela dit, cette approche est elle-même meilleure qu'une approche classique utilisant des types de nombres multiprécision (`Gmpz`).

La figure 4.5 illustre le nombre d'échecs du filtre au cours du calcul d'une triangulation de Delaunay 2D de 10^5 points, pour les filtres dynamiques (colonne IA) et statiques adaptatifs (colonne SA). Les points ont des coordonnées à valeur entière sur 30 bits, dont la distribution est aléatoire pour la première ligne, sur une grille à maille carrée exactement pour la dernière ligne, et pour les lignes intermédiaires, les points sont sur la grille, à une perturbation aléatoire près bornée de manière relative par rapport à la coordonnée maximale (2^{30}). Par exemple, $\varepsilon = 2^{-30}$ correspond à une perturbation d'au plus 1. Le nombre d'échecs est la somme des échecs pour les deux prédicats utilisés (`orientation` et `in_circle`), la majorité provenant du test `in_circle`.

On constate que le taux d'échec du filtre dynamique est très faible, même pour des données très dégénérées, voire exactement dégénérées. Le filtre statique adaptatif par contre est beaucoup plus sensible aux cas presque dégénérés.

Il est donc souhaitable pour ce type de distributions de pouvoir utiliser des filtres en cascade : tout d'abord un filtre rapide statique, suivi en cas d'échec d'un filtre dynamique avant de passer au calcul plus coûteux.

4.5 Conclusion

L'approche du calcul exact qui consiste à n'utiliser uniquement qu'un type de nombres exact induit nécessairement un traitement dynamique de l'erreur, ce qui n'est pas optimal du point de vue de l'efficacité, même si cela permet de l'être du point de vue de l'utilisation.

En revanche, traiter le problème au niveau des prédicats permet de rendre robustes une majorité d'algorithmes géométriques, et grâce à l'usage de filtres arithmétiques, cette approche peut être rendue presque aussi efficace que le calcul inexact flottant.

Nous avons intégré à la bibliothèque CGAL un outil permettant de générer des prédicats filtrés de manière automatique, en cela similaire aux outils existants, tout en étant plus simple sur le plan conceptuel, et se basant sur des types de filtres nouveaux que nous avons mis au point (dynamique basé sur l'arithmétique d'intervalles et statique adaptatif).

Chapitre 5

Calcul du signe des déterminants

Le problème du calcul du signe d'un déterminant est récurrent dans les prédicats géométriques. De ce fait, de nombreuses méthodes ont vu le jour afin de résoudre spécifiquement ce problème le plus rapidement possible.

Certaines méthodes s'attachent à calculer le déterminant lui-même puis en prennent le signe, d'autres utilisent des propriétés de conservation du signe par certaines manipulations.

5.1 Méthodes existantes

Nous présentons ici quelques méthodes connues de calcul de déterminants à coefficients entiers (sur b bits en multiprécision), en dimension d .

5.1.1 Élimination Gaussienne

L'élimination Gaussienne [CLR90] est une des méthodes les plus classiques pour calculer un déterminant. L'algorithme se déroule en $d - 1$ étapes, chacune modifiant la matrice, et multipliant le déterminant par un facteur connu. L'étape 1 correspond à la matrice d'origine, et l'étape $d - 1$ correspond à une matrice triangulaire dont on peut calculer facilement le déterminant.

À l'étape k de l'algorithme, on note le pivot $p_k = a_{k,k}^{(k)}$ (si $a_{k,k}^{(k)}$ est non nul, sinon on permute les lignes). Puis on calcule pour tout $i > k$ et $j > k$, $a_{i,j}^{(k+1)} = p_k \times a_{i,j}^{(k)} - a_{k,j}^{(k)} \times a_{i,k}^{(k)}$.

$$\begin{vmatrix} p_1 & & \cdots & & \\ 0 & \ddots & & & \\ \vdots & \vdots & & p_k & \\ 0 & 0 & & & a_{i,j}^{(k)} \end{vmatrix}$$

À chaque étape, la valeur du déterminant est multipliée par p_k^{d-k-2} , on obtient donc facilement le signe du déterminant final à partir des signes des pivots successifs.

Dimension	3	4	5	6
Nombre de bits	50	49	47	46

TAB. 5.1 – Nombre maximum de bits des entrées pour la méthode Lattice

Dans le modèle Real-RAM, sa complexité est de $O(d^3)$ opérations arithmétiques. Son inconvénient majeur est qu'elle nécessite des entiers de taille $O(b2^d)$ dès lors qu'on l'applique à des entiers multiprécision, si on évite les divisions.

5.1.2 Variante de Bareiss

Il existe une variante de l'élimination Gaussienne basée sur la remarque que des divisions exactes sont possibles sur les entiers par les pivots précédents, au fur et à mesure de l'algorithme : les $a_{i,j}^{(k)}$ sont divisibles par p_{k-2} . De ce fait, la taille des entiers multiprécision n'est plus exponentielle en la dimension, et devient $O(bd)$, linéaire par rapport à la dimension. Cependant, cette variante a l'inconvénient d'utiliser $O(d^3)$ divisions sur des entiers de taille $O(bd)$.

5.1.3 Méthodes de Clarkson et ABDPY

Le premier algorithme spécifique aux signes de déterminants est dû à Clarkson [Cla92], il a été suivi par des études plus détaillées [BY96], ainsi que par d'autres algorithmes comme ABDPY (Lattice) [ABD⁺95]. Ces méthodes sont basées sur l'idée de modifier légèrement les algorithmes classiques de manipulation de vecteurs de la matrice, afin de ne requérir que très peu de bits supplémentaires pour garantir le calcul exact du signe du déterminant.

Ainsi, Clarkson utilise le processus d'orthogonalisation de Gram-Schmidt, afin de mettre la matrice sous une forme suffisamment éloignée des dégénérescences pour que le signe de son déterminant puisse être calculé de manière sûre. Le nombre de bits utilisés par la méthode est $b + O(d)$.

La méthode ABDPY [ABD⁺95] et son extension aux dimensions supérieures à 3 [BY97, BY96] utilisent le même genre de technique, et nécessitent un nombre de bits additionnels équivalent.

Dans le cas des petites dimensions, ces méthodes permettent d'effectuer des calculs de signes de déterminants exacts avec des doubles, pour des entrées ayant un petit nombre de bits, allant jusqu'à 50. Elles s'avèrent bien plus efficaces que les méthodes multiprécision traditionnelles, et sont également complémentées efficacement par des filtres.

Nous donnons dans la figure 5.1 le nombre de bits maximum que peuvent avoir les entrées entières, dans la méthode Lattice [BY97], pour que le calcul soit effectué de manière exacte avec des doubles.

5.2 En représentation modulaire

Le calcul du déterminant en représentation modulaire est bien connu [Knu98]. Nous nous sommes attachés au calcul du signe des expressions modulaires en général dans le chapitre 3, et nous allons voir comment l'appliquer au calcul du signe du déterminant.

Le calcul de la valeur du déterminant à coefficients entiers peut se décomposer sur un certain nombre de moduli, puis on peut calculer le signe en utilisant les algorithmes du chapitre 3.

5.2.1 Calcul de la borne

Pour effectuer ce calcul, il faut connaître une borne sur la valeur du déterminant, afin d'en déduire le nombre de moduli nécessaires. On utilise généralement la borne d'Hadamard : $2^{bd} \times d^{d/2}$. Cette borne est suffisamment fine pour la plupart des besoins en petite dimension, cependant on notera qu'il est possible de faire mieux en grande dimension, comme en témoignent des travaux récents qui permettent de déterminer une meilleure borne et ont donc de meilleurs résultats que les nôtres pour les très grandes dimensions (supérieures à 50) [ABM99]. On notera également que les bornes des majorants des déterminants dont les coefficients sont plus petits que 1 sont connus de manière exacte jusqu'à la dimension 14 [Slo99], et que la borne d'Hadamard est optimale pour les dimensions qui sont des puissances de 2.

Une autre manière permettant d'acquérir une borne est d'utiliser un filtre dynamique, qui pourra même directement certifier le signe dans certains cas, et qui sinon offrira une borne probablement meilleure que celle d'Hadamard.

5.2.2 Élimination Gaussienne

La méthode qui semble la plus adaptée pour calculer le déterminant modulo m_i est l'élimination Gaussienne. Contrairement au calcul sur des entiers multiprécision, il n'y a aucun problème d'explosion de la taille des coefficients, puisqu'ils sont tous des résidus modulo m_i , la variante de Bareiss est donc sans objet. La méthode nécessite néanmoins une division modulaire finale par modulo, ce qui implique de prendre m_i premier si on ne veut pas trop compliquer l'algorithme.

On obtient donc les valeurs du déterminant modulo les m_i . Cette partie du calcul est dans le cas général la plus coûteuse, elle est aussi facilement parallélisable.

Dans le cas d'un calcul séquentiel, un avantage du calcul modulaire est qu'il fait les différents calculs du déterminant dans un emplacement mémoire de taille constante ($O(d^2)$), c'est-à-dire que l'occupation mémoire maximale est bien inférieure à celle d'un calcul en multiprécision. La complexité totale de l'algorithme est donc en espace $O(d^2)$ et en temps $O(bd \log(d))$.

5.3 Filtres efficaces avec les intervalles

On montre facilement que le calcul du signe d'un déterminant de dimension d en utilisant l'élimination Gaussienne est de degré $2d$ (cf 4.1.3.0). De ce fait, on peut dire qu'en pratique, pour une dimension supérieure à 25, il faut trouver une autre méthode de calcul que des intervalles dont les bornes sont des flottants en double précision.

Il existe d'autres méthodes de calcul de déterminant plus stables numériquement, et donc plus adaptées aux calculs sur les intervalles [BBP98b, BBP98a].

5.4 Temps de calcul

Nous présentons ici quelques mesures de temps de calcul de nos implantations en C du calcul du signe d'un déterminant en calcul modulaire, et la comparons avec différentes bibliothèques existantes.

Les coefficients sont des entiers de 53 bits, et les tests ont été effectués sur une UltraSparc à 200MHz. Nous avons étudié 3 classes de matrices: les matrices aléatoires, dont le déterminant est normalement très éloigné de zéro (Fig. 5.2), les matrices quasi singulières dont le déterminant est de l'ordre de 2^{50} (Fig. 5.3), et les matrices singulières dont le déterminant est nul (Fig. 5.4).

- La méthode FP est une implantation directe de l'élimination Gaussienne en calcul flottant.
- La méthode MOD correspond à l'élimination Gaussienne, suivie de la méthode de Lagrange de calcul du signe.
- La méthode GMP est l'implantation basée sur la bibliothèque GMP de l'élimination Gaussienne pour les petites dimensions, et la variante de Bareiss pour les plus grandes.
- La méthode LEDA utilise la routine `sign_of_determinant(integer_matrix)` de LEDA [BKM⁺95].

On constate que la méthode modulaire est très efficace, et que le coût de la reconstruction de signe avec notre méthode est négligeable, y compris sur les déterminants de petites dimensions. On constate néanmoins une légère différence pour la dimension 2, entre les déterminants nuls, où la méthode de reconstruction de signe est en temps quadratique, et les déterminants aléatoires, où typiquement elle est linéaire.

5.5 Conclusion

Le calcul du signe d'un déterminant est une opération particulière, mais elle est suffisamment utilisée pour mériter une étude approfondie. Elle a fait l'objet de méthodes de calcul exact dédiées [Cla92, ABD⁺95, BY97], surtout orientées vers les besoins des prédicats de la géométrie algorithmique, ainsi que diverses extensions aux dimensions plus grandes motivées par les besoins du calcul formel [ABM99].

n	FP	MOD	GMP	LEDA
2	0.1	7.6	6.1	88.4
3	0.8	15.9	37.9	336
4	2.1	36.7	146	988
5	3.8	125	538	2440
6	6.3	223	1300	5170
7	9.7	374	2790	10200
8	14.1	568	5120	18100
9	19.5	861	9400	30300
10	26.4	1270	15400	48300
12	44.0	2460	37500	123000
14	68.3	4400	75700	251000

TAB. 5.2 – *Déterminants aléatoires*

n	FP	MOD	GMP	LEDA
2	0.1	8.2	6.3	84.6
3	0.8	21.6	27.0	232
4	2.0	36.9	98.9	699
5	3.8	119	221	1100
6	6.3	234	731	2880
7	9.6	384	1110	3590
8	14.3	586	2800	6610
9	19.4	891	3430	10700
10	26.3	1320	6100	18300
12	43.9	2460	13300	38400
14	68.1	4460	27200	51800

TAB. 5.3 – *Petits déterminants*

n	FP	MOD	GMP	LEDA
2	0.1	9.3	6.4	78.7
3	0.8	19.8	38.5	284
4	2.1	39.7	149	717
5	3.8	134	547	1750
6	6.3	239	920	4850
7	9.7	397	2650	7300
8	13.9	594	4720	13800
9	19.6	894	7700	25500
10	26.3	1330	12600	46800
12	43.7	2550	30800	87700
14	67.9	4460	68100	209000

TAB. 5.4 – *Déterminants nuls.*

Nous avons présenté ici nos résultats de la méthode classique basée sur le calcul modulaire, à laquelle nous avons adjoint nos algorithmes de calcul de signe. Nous pensons enfin que des études particulières sur les filtres peuvent aboutir à des temps de calcul moyens bien meilleurs, si on arrive à avoir une stabilité numérique importante.

Conclusion

Les travaux de cette thèse ont porté sur la recherche de méthodes efficaces en pratique pour résoudre les problèmes de robustesse posés par la géométrie algorithmique. La bibliothèque CGAL a été un banc d'essai idéal pour valider des méthodes générales.

Le paradigme du calcul exact a été adopté pour sa généralité, et plus particulièrement l'approche qui vise à garantir l'exactitude des prédicats, plutôt qu'une approche simplement basée sur l'usage d'un type de nombres exact.

Grâce à l'usage des filtres arithmétiques, dont nous avons implanté plusieurs types, nous avons pu montrer que le surcoût induit par la prise en compte des problèmes de robustesse de la plupart des algorithmes travaillant sur des objets de petite dimension est négligeable, face au calcul flottant approché. Nous avons également conçu un outil qui permet de générer le code de ces prédicats filtrés automatiquement, et qui s'applique à tous les prédicats.

Nous avons pour cela utilisé des outils classiques tels que l'arithmétique d'intervalles et l'arithmétique modulaire.

Cependant, des problèmes de robustesse demeurent, particulièrement dans le cas des grandes dimensions, où les méthodes à base de filtres ne sont que naissantes et nécessitent un traitement particulier. On peut également se poser la question de savoir s'il existe de meilleures méthodes que celles existantes, pour traiter les cas de constructions en cascade.

Annexe A

Script PERL pour générer les prédicats filtrés

Cette annexe contient le code du script qui permet de produire des prédicats filtrés, écrit en PERL. Ce script fait partie de CGAL.

```
#!/usr/local/bin/perl -w
#
# Copyright (c) 1998-1999 The CGAL Consortium
# Author: Sylvain Pion <Sylvain.Pion@sophia.inria.fr>.
#
# This script takes as input a source code file containing template predicates.
# It outputs the filtered specializations.
#
# See the CGAL documentation: Support Library, Number Types.
#
# TODO:
# - Split the lines only if the result is larger than 80 characters?
# - Separate the class in other files? (they are needed by libCGAL)
# - while parsing, remove CGAL_assertion() and co from the original code.
# - Add assertions about the epsilons being updated.

use vars qw($opt_p $opt_h $opt_d $opt_v $opt_l); # Suppress warnings
require 'getopt.pl';

# Global variables
%known_ret_types=("bool", 1); # Known allowed return types (see main())
@predicates=( # list of predicates, with the built-ins.
# CGAL::, template_type, inline, ret_type, fct_name, #eps, body, new_body, args
[ "", "NT", "CGAL_KERNEL_INLINE", "Sign", "lexicographical_sign", 1, "",
  "", "x", "y" ],
[ "", "NT", "inline", "Comparison_result", "compare", 1, "", "", "a", "b"],
[ "", "NT", "inline", "Sign", "sign", 1, "", "", "x" ] );
# Note: we don't actually care about a few fields ($body...) for these.
$num_built_in_predicates=$#predicates+1; # Number of built-in predicates.
$pred_list_re="sign|compare|lexicographical_sign";

# We _must_ manipulate the fields of @predicate using these constants:
```

10

20

30

```

# (aren't there structs in Perl? Are packs the way to go? )
# ($CGAL_pos, $template_type_pos, $inline_pos, $ret_type_pos)=(0..3); # unused
($fct_name_pos, $seps_pos, $body_pos, $new_body_pos, $args_pos)=(4..8);

# Useful regexps.
$C_symbol_re='[a-zA-Z_]\w*';           # A pure C symbol name.
$CGAL_symbol_re='(?:CGAL:|)'.$C_symbol_re; # Idem with eventually 'CGAL:.'

# Parse the command line options.
sub parse_command_line {
  Getopt('iodl');
  if ($opt_h) {
    warn "Usage: filtered_predicate_converter [options]
-i file : specify the main input source file [default is stdin]
-o file : specify the main output file      [default is stdout]
-l file : indicate the output file that will receive the static part.
-d files : list of dependant predicates file (concatenated with \":\")
          by default, only the built-in predicates are known.
          (typically, it's just sign_of_determinant.h that you want)
          (it's only useful when \"-s\" is here too).
-p      : be pedantic (change warnings to errors)
-h      : print this help\n";
    exit;
  }
  @dependency_files = split(':', $opt_d) if ($opt_d);
  $opt_i = "-" unless $opt_i;
  $opt_o = "-" unless $opt_o;
  if (not $opt_l) {
    warning("The -l option is compulsory for the static filters.");
  }
  $pedantic = $opt_p;
}

# Auxiliary routine to emit a warning and die if pedantic.
sub warning {
  warn "// Warning: @_ \n";
  die if $pedantic;
}

# Parse the CGAL header
sub parse_CGAL_header {
  local ($) = @_;
  my $file_name = $1 if m#// file[\s]*: include/CGAL/(.*)#m;
  my $rest      = $' if /^#ifndef[\s]*([\S]*_H)\s*?/m;
  $_ = $1;
  $_ = "CGAL_$_" unless /CGAL_/;
  s/CGAL_/CGAL_ARITHMETIC_FILTER_/;
  return ($_, $file_name, $rest);
}

# Print the new CGAL header
sub print_CGAL_header {
  my ($output_file, $new_protect_name, $file_name) = @_;
  print $output_file

```

40

50

60

70

80

```

// =====
//
// Copyright (c) 1999 The CGAL Consortium
//
// This software and related documentation is part of an INTERNAL release
// of the Computational Geometry Algorithms Library (CGAL). It is not
// intended for general use.
//
// -----
//
// release      :
// release_date :
//
// file         : include/CGAL/Arithmetic_filter/$file_name
// package      : Interval_arithmetic
// author(s)    : Sylvain Pion <Sylvain.Pion@sophia.inria.fr>
//
// coordinator  : INRIA Sophia-Antipolis (<Marianne.Yvinec@sophia.inria.fr>)
// =====

// This file is automatically generated by
// scripts/filtered_predicates_generator.pl

#ifndef $new_protect_name
#define $new_protect_name\n\n";
}

# Print dynamic versions
sub print_dynamic {
  my ($adv, $CGAL, $t, $inline, $ret_type, $fct_name, $e, $b, $n, @args)=@_;
  my $type = "const ${CGAL}Filtered_exact <CGAL_IA_CT, CGAL_IA_ET, Dynamic,"
    ." $adv, CGAL_IA_CACHE>";
  my $args_call = join ",", map "\n  $type &$_", @args;
  my $args_inter = join ",", map "\n\t\t$.interval()", @args;
  my $args_exact = join ",", map "\n\t\t$.exact()", @args;

  print F0
  "#ifndef CGAL_CFG_MATCHING_BUG_2
template < class CGAL_IA_CT, class CGAL_IA_ET, class CGAL_IA_CACHE >
#else
template <>
#endif
/* $inline */
$ret_type
$fct_name($args_call)
{";

  if ($adv eq "Advanced") {
    print F0 "
CGAL_expensive_assertion(FPU_empiric_test() == FPU_cw_up);
try
{
  return $fct_name($args_inter);
}

```

```

catch (${CGAL}Interval_nt_advanced::unsafe_comparison)
{
  ${CGAL}FPU_CW_t backup = ${CGAL}FPU_get_and_set_cw(${CGAL}FPU_cw_near);
  $ret_type result = $fct_name($args_exact);
  ${CGAL}FPU_set_cw(backup);
  return result;
}";
} else {
  print FO "
${CGAL}FPU_CW_t backup = ${CGAL}FPU_get_and_set_cw(${CGAL}FPU_cw_up);
try
{
  $ret_type result = $fct_name($args_inter);
  ${CGAL}FPU_set_cw(backup);
  return result;
}
catch (${CGAL}Interval_nt_advanced::unsafe_comparison)
{
  ${CGAL}FPU_set_cw(backup);
  return $fct_name($args_exact);
}";
}
print FO "\n}\n\n";
}

# Print static infos
sub print_static_infos {
  my ($CGAL, $t, $inline, $ret_type, $fct_name, $eps, $b, $new_body, @args)=@_;
  my $predicate_class_name = "Static_Filtered_$fct_name\_".($#args+1);

  print FL map("double $predicate_class_name\::_epsilon_$_\n", 0..$eps-1);
  print FL "double $predicate_class_name\::_bound = -1.0;\n\n";
}

# Print the epsilon variant of the function
sub epsilon_function {
  my ($ret_type, $fct_name, $body, $t, $eps, $new_t, $const, @args) = @_;
  my $args_call = join ",", map "\n\tconst $new_t &$_", @args;
  my $eps_call = join ",", map "\n\t${const}double & epsilon_$_", 0..$eps-1;
  # We just typedef the original template type at the beginning of the body.
  $_ = $body;
  s/{/{\n typedef $new_t $t;\n/s;
  s/\n/\n /msg; # Indent
  s/update_epsilon/$fct_name/msg;
  return "static $ret_type $fct_name($args_call,$eps_call)$_";
}

# Print the struct corresponding to the predicate
sub print_predicate_struct {
  my ($CGAL, $t, $inline, $ret_type, $fct_name, $eps, $b, $new_body, @args)=@_;
  my $predicate_class_name = "Static_Filtered_$fct_name\_".($#args+1);
  my $epsilon_list = join ",", map "_epsilon_$_", 0..$eps-1;
  my $bound_list = "b," x ($#args+1);
  my $update_epsilon = epsilon_function($ret_type, "update_epsilon",

```

```

        $new_body, $t, $eps, "${CGAL}Static_filter_error", "", @args);

my $epsilon_variant = epsilon_function($ret_type, "epsilon_variant",
    $new_body, $t, $eps, "Restricted_double", "const ", @args);
print F0
"struct $predicate_class_name
{
    static double _bound;
    static double $epsilon_list;
    // static unsigned number_of_failures; //?

    $update_epsilon

    // Call this function from the outside to update the context.
    static void new_bound (const double b) // , const double error = 0)
    {
        _bound = b;
        // recompute the epsilons: \"just\" call it over Static_filter_error.
        // That's the tricky part that might not work for everything.
        (void) update_epsilon($bound_list$epsilon_list);
        // TODO: We should verify that all epsilons have really been updated.
    }

    $epsilon_variant
};\n\n";
}
# Print static versions
sub print_static {
    my ($adv, $CGAL, $t, $inline, $ret_type, $fct_name, $eps, $b, $new_body,
        @args)=@_;

    my $predicate_class_name = "Static_Filtered_{$fct_name}\_".($#args+1);
    my $type = "const ${CGAL}Filtered_exact <CGAL_IA_CT, CGAL_IA_ET, Static,"
        . " $adv, CGAL_IA_CACHE>";

    my $args_call    = join ",", map "\n    $type &$_", @args;
    my $args_dbl     = join ",", map "\n\t\t\t$_dbl()", @args;
    my $args_exact   = join ",", map "\n\t\t\t$_exact()", @args;
    my $args_epsilons = join ",",
        map "\n\t\t\t$predicate_class_name\::_epsilon_$_", 0..$eps-1;
    my $args_error   = "\n\t\t\tStatic_filter_error(SAF_bound)," x ($#args+1);
    my $bounds_check = join "||",
        map "\n\t\t\tfabs($_to_double()) > SAF_bound", @args;
    my $compute_new_bound = join "",
        map "\n    NEW_bound = std::max(NEW_bound, fabs($_to_double()));", @args;

    print F0
    "#ifndef CGAL_CFG_MATCHING_BUG_2
template < class CGAL_IA_CT, class CGAL_IA_ET, class CGAL_IA_CACHE >
#else
template <>
#endif
/* $inline */
$ret_type

```

```

$fct_name($args_call)
{";
250

    if ($adv eq "Advanced") {
        print FO "
CGAL_assertion_code(
    const double SAF_bound = $predicate_class_name\::_bound; )
CGAL_assertion(!($bounds_check));

    try
    {
        return $predicate_class_name\::epsilon_variant($args_dbl,$args_epsilons);
260
    }
    catch ($){CGAL}Restricted_double::unsafe_comparison)
    {
        return $fct_name($args_exact);
    };
} else {
    print FO "
bool re_adjusted = false;
const double SAF_bound = $predicate_class_name\::_bound;
270

    // Check the bounds. All arguments must be <= SAF_bound.
    if ($bounds_check)
    {
re_adjust:
        // Compute the new bound.
        double NEW_bound = 0.0;$compute_new_bound
        // Re-adjust the context.
        $predicate_class_name\::new_bound(NEW_bound);
    }
280

    try
    {
        return $predicate_class_name\::epsilon_variant($args_dbl,$args_epsilons);
    }
    catch ($){CGAL}Restricted_double::unsafe_comparison)
    {
        if (!re_adjusted) { // It failed, we re-adjust once.
            re_adjusted = true;
            goto re_adjust;
290
        }
        return $fct_name($args_exact);
    };
}
print FO "\n}\n\n";
}

# Parse from "template" to end of body
sub parse_function_definition {
    local ($) = @_;
    /^\\W*template\s*\<\s*(?:class|typename)\s*(\\S*)\s*\> # template type name
300
    \\W*(CGAL_\\w*INLINE|inline|) # eventual inline directive
    \\W*($CGAL_symbol_re) # return type
}

```

```

    \W*($C_symbol_re)          # function name
    .*?\((.*?)\) /smx;        # argument list
my ($x, $body, $after) = extract_balanced("{", "}", $');
my @pred = ($1, $2, $3, $4, -1, $body."}", "");
my $fct_name = $4;
if (not $known_ret_types{$3}) {
    warning("Return type \"\$3\" of function \"\$4\" is unknown");
}
foreach (split (',', $5)) {
    if (not /const/) {
        warning("Non const argument \"$_\" in function \"\$fct_name\".");
    }
    /($C_symbol_re)\s*$/;
    push @pred, $1;
}
return ($after, @pred);
}

```

310

```

# Main parsing subroutine
sub parse_input_code {
    local ($) = @_;
    my $CGAL = "CGAL::";
    # Treat NO_FILTER parts
    s//CGAL_NO_FILTER_BEGIN.*?//CGAL_NO_FILTER_END##msg;
    # Note that the following are buggy if they appear in strings (cf Perl FAQ).
    s//.*##mg;      # Remove C++ “//” comments
    s#/\*.*?\/##sg; # Remove C “/**/” comments
    while (/((CGAL_(?:BEGIN|END)_NAMESPACE|template\s*\(<.*?\>)/sm) {
        if ($1 eq "CGAL_BEGIN_NAMESPACE") { $CGAL=""; $_=$'; }
        elsif ($1 eq "CGAL_END_NAMESPACE") { $CGAL="CGAL::"; $_=$'; }
        else {
            my ($after, @pred) = parse_function_definition($1.$');
            push @predicates, [ $CGAL, @pred ];
            $pred_list_re.="|".$pred[$fct_name_pos-1];
            treat_predicate($#predicates);
            $_ = $after;
        }
    }
}

```

330

```

}

```

340

```

# Print the code of the overloaded predicates
sub print_predicates {
    my $was_in_CGAL = 0;
    # We skip the built-in predicates.
    for $i ( $num_built_in_predicates .. $#predicates ) {
        my ($CGAL) = @{$predicates[$i]};
        print FO "CGAL_BEGIN_NAMESPACE\n\n" if $CGAL eq "" && not $was_in_CGAL;
        print FO "CGAL_END_NAMESPACE\n\n"   if $CGAL ne "" && $was_in_CGAL;
        print_dynamic("Protected", @{$predicates[$i]});
        print_dynamic("Advanced", @{$predicates[$i]});
        print_predicate_struct(@{$predicates[$i]});
        print_static("Protected", @{$predicates[$i]});
        print_static("Advanced", @{$predicates[$i]});
        print_static_infos(@{$predicates[$i]});
    }
}

```

350

```

    $was_in_CGAL = $CGAL eq "";
}
print FO "CGAL_END_NAMESPACE\n\n" if $was_in_CGAL;
}
360

# Matches balanced $beg $end (say "(" and ")"), and counts the number
# of zones separated by a comma at level 1.
# Returns number of zones found, text before, and after.
# Perl 5.6 will support extended regexp that will understand balanced exprs...
sub extract_balanced {
    local ($beg, $end, $_) = @_;
    my $num_args = 1; # We don't handle 0 argument functions.
    my $before = "";
    my $quote_level = 0;
    while ( /./sm ) {
        if ($& eq $beg) { ++$quote_level; }
        elsif ($& eq $end) { --$quote_level; last if ($quote_level == 0); }
        elsif ($& eq "," && $quote_level == 1) { ++$num_args; }
        $before .= $&;
        $_ = $';
    }
    # The original text was "$before$end$_" (the last closing $end).
    return ($num_args, $before, $_);
}
370
380

# Find predicate calls in the body
# Returns the number of epsilons that this predicate needs, and the new body.
sub match_calls_in_body {
    local ($) = @_;
    my $new_body = "";
    my $num_eps = 0;

    # We match the first call to a known predicate function.
    while ( /(\W)($pred_list_re)(\s*\()/sm ) {
        my $predicate_name = $2;
        my ($num_args, $before, $after) = extract_balanced ("(", ")", "(.$'");
        # Recognize the actual predicate, with same number of arguments.
        my $p;
        foreach ( @predicates ) {
            $p = $_;
            last if (scalar(@$p) - $args_pos == $num_args) &&
                (@$p[$fct_name_pos] eq $predicate_name);
        }
        if (not $p) {
            warning("We matched \"\$predicate_name\" with an unknown number of"
                ." arguments");
        };
        # We replace $predicate_name by
        # Static_Filtered_$predicate_name_#args::update_epsilon.
        $new_body .= "$'$1Static_Filtered_$2_$num_args\::update_epsilon$before,"
            . "\n\t\ttepsilon_";
    }
}
390
400
410

```

```

    # We add the epsilon arguments to the call in $new_body.
    $new_body.= join ",\n\t\tepsilon-", map $num_eps+$_, 0..@p[$seps_pos]-1;
    $num_eps += @p[$seps_pos];
    $_ = $after;
}

return ($num_eps, $new_body.$_);
}

# Calls match_calls_in_body over @predicates[$_], and updates $seps & $new_body.
sub treat_predicate {
    my ($i) = @_;
    my ($seps, $new_body) = match_calls_in_body($predicates[$i][$body_pos]);
    $predicates[$i][$new_body_pos] = $new_body;
    $predicates[$i][$seps_pos] = $seps;
}

# Parse the list of dependancy files, and populate @predicates with them.
sub parse_dependancy_files {
    foreach (@dependancy_files) {
        open(FI, "<$_") || die "Couldn't open dependancy file \"$_\"";
        parse_input_code(<FI>);
        close(FI);
    }
    # Consider them as built-in now (i.e. don't output specializations for them)
    $num_built_in_predicates=$#predicates+1;
}

# Main program
sub main {
    for ("Sign","Comparison_result","Orientation","Oriented_side","Bounded_side")
    {
        $known_ret_types{$_}=1;
        $known_ret_types{"CGAL:::$_"}=1;
    }
    parse_command_line();
    undef $/; # We parse the _whole_ input files as strings, not line by line.
    if ($opt_d) { parse_dependancy_files(); }
    open(FI, "<$opt_i") || die "Couldn't open input file \"\$opt_i\"";
    my ($new_protect_name, $file_name, $rest) = parse_CGAL_header(<FI>);
    close(FI);
    parse_input_code($rest);
    open(FO, ">$opt_o") || die "Couldn't open output file \"\$opt_o\"";
    open(FL, ">$opt_l") || die "Couldn't open lib file \"\$opt_l\"";
    print_CGAL_header(FO, $new_protect_name, $file_name);
    print_CGAL_header(FL, $new_protect_name."_STATIC_INFO_H",
        "static_infos/\".$file_name);
    print_predicates();
    print FO "#endif // $new_protect_name\n";
    print FL "#endif // $new_protect_name."_STATIC_INFO_H\n";
    close(FO);
    close(FL);
}

```

main();

Annexe B

Plateformes utilisées pour les mesures

Cette annexe décrit les environnements qui ont été utilisés pour effectuer les divers bancs d'essai de cette thèse. Dans un souci d'homogénéité, tous les tests (sauf mention contraire) ont été effectués sur chacune des deux plateformes suivantes. Les mesures sont arrondies à deux chiffres significatifs.

Compilateur

Dans tous les cas, nous avons utilisé le compilateur GNU, GCC version 2.95 (soit en C ou en C++ selon le programme), avec l'option d'optimisation `-O2`.

Plateforme PC

Il s'agit d'un PC, bi processeurs Pentium III cadencés à 500 MHz, équipés de 512Ko de cache et 256 Mo de mémoire vive. Le système d'exploitation est Linux 2.2. Dans la plupart des cas, un seul des deux processeurs est utilisé. Nous avons rajouté les options de compilation spécifiques : `-mcpu=pentiumpro -march=pentiumpro`.

Plateforme Sun

Il s'agit d'une machine Sun Ultra-10, mono processeur UltraSparc cadencé à 300MHz, équipé de 128Mo de mémoire vive. Le système d'exploitation est Solaris 2.6. Nous avons rajouté les options de compilation spécifiques : `-mcpu=ultrasparc -mtune=ultrasparc`.

Bibliographie

- [AABV99] Pankaj K. Agarwal, Lars Arge, G. Brodal, and Jeffrey S. Vitter. I/o-efficient dynamic point location in monotone subdivisions. In *Proc. 10th ACM-SIAM Sympos. Discrete Algorithms*, pages 11–20, 1999.
- [ABD⁺95] Francis Avnaim, Jean-Daniel Boissonnat, Olivier Devillers, Franco P. Preparata, and Mariette Yvinec. Evaluation of a new method to compute signs of determinants. In *Proc. 11th Annu. ACM Sympos. Comput. Geom.*, pages C16–C17, 1995.
- [ABM99] J. Abbott, M. Bronstein, and T. Mulders. Fast deterministic computation of determinants of dense matrices. In *ISSAC*, 1999.
- [ADS98] Pierre Alliez, Olivier Devillers, and Jack Snoeyink. Removing degeneracies by perturbing the problem or the world. In *Proc. 10th Canad. Conf. Comput. Geom.*, 1998.
- [BBP98a] Hervé Brönnimann, Christoph Burnikel, and Sylvain Pion. Interval arithmetic yields efficient arithmetic filters for computational geometry. In *Abstracts 14th European Workshop Comput. Geom.*, pages 49–54, 1998.
- [BBP98b] Hervé Brönnimann, Christoph Burnikel, and Sylvain Pion. Interval arithmetic yields efficient dynamic filters for computational geometry. In *Proc. 14th Annu. ACM Sympos. Comput. Geom.*, pages 165–174, 1998.
- [BEPP97a] H. Brönnimann, I. Emiris, V. Pan, and S. Pion. Computing exact geometric predicates using modular arithmetic with single precision. Research Report 3213, INRIA, 1997.
- [BEPP97b] Hervé Brönnimann, Ioannis Emiris, Victor Pan, and Sylvain Pion. Computing exact geometric predicates using modular arithmetic with single precision. In *Proc. 13th Annu. ACM Sympos. Comput. Geom.*, pages 174–182, 1997.
- [BEPP99] Hervé Brönnimann, Ioannis Emiris, Victor Pan, and Sylvain Pion. Sign determination in Residue Number Systems. *Theoret. Comput. Sci.*, 210(1):173–197, 1999. Special Issue on Real Numbers and Computers.
- [BFMS97] C. Burnikel, R. Fleischer, K. Mehlhorn, and S. Schirra. A strong and easily computable separation bound for arithmetic expressions involving square roots. In *Proc. 8th ACM-SIAM Sympos. Discrete Algorithms*, pages 702–709, 1997.
- [BFS98] C. Burnikel, S. Funke, and M. Seel. Exact geometric predicates using cascaded computation. In *Proc. 14th Annu. ACM Sympos. Comput. Geom.*, pages 175–183, 1998.

- [BJMM93] M. Benouamer, P. Jaillon, D. Michelucci, and J.-M. Moreau. A lazy solution to imprecision in computational geometry. In *Proc. 5th Canad. Conf. Comput. Geom.*, pages 73–78, 1993.
- [BKM⁺95] Christoph Burnikel, Jochen Könnemann, Kurt Mehlhorn, Stefan Näher, Stefan Schirra, and Christian Uhrig. Exact geometric computation in LEDA. In *Proc. 11th Annu. ACM Sympos. Comput. Geom.*, pages C18–C19, 1995.
- [Blö91] J. Blömer. Computing sums of radicals in polynomial time. In *Proc. 32nd Annu. IEEE Sympos. Found. Comput. Sci.*, pages 670–677, 1991.
- [BMS96] C. Burnikel, K. Mehlhorn, and S. Schirra. The LEDA class real number. Technical Report MPI-I-96-1-001, Max-Planck Institut Inform., Saarbrücken, Germany, January 1996.
- [BP97a] Jean-Daniel Boissonnat and Franco P. Preparata. Robust plane sweep for intersecting segments. Research Report 3270, INRIA, Sophia Antipolis, September 1997.
- [BP97b] Hervé Brönnimann and Sylvain Pion. Exact rounding for geometric constructions. In *GAMM/IMACS International Symposium on Scientific Computing, Computer Arithmetic and Validated Numerics*, 1997.
- [BS99] J.-D. Boissonnat and J. Snoeyink. Efficient algorithms for line and curve segment intersection using restricted predicates. In *Proc. 15th Annu. ACM Sympos. Comput. Geom.*, pages 370–379, 1999.
- [Bur96] C. Burnikel. *Exact Computation of Voronoi Diagrams and Line Segment Intersections*. Ph.D thesis, Universität des Saarlandes, March 1996.
- [BY95] Jean-Daniel Boissonnat and Mariette Yvinec. *Géométrie Algorithmique*. Ediscience international, Paris, 1995.
- [BY96] Hervé Brönnimann and Mariette Yvinec. A complete analysis of Clarkson’s algorithm for safe determinant evaluation. Research Report 3051, INRIA, 1996.
- [BY97] Hervé Brönnimann and Mariette Yvinec. Efficient exact evaluation of signs of determinants. In *Proc. 13th Annu. ACM Sympos. Comput. Geom.*, pages 166–173, 1997.
- [BY98] Jean-Daniel Boissonnat and Mariette Yvinec. *Algorithmic Geometry*. Cambridge University Press, UK, 1998. Translated by Hervé Brönnimann.
- [C⁺96] Bernard Chazelle et al. Application challenges to computational geometry: CG impact task force report. Technical Report TR-521-96, Princeton Univ., April 1996.
- [CGA99] *The CGAL Reference Manual*, 1999. Release 2.0.
- [CKKM99] T. Culver, J. Keyser, S. Krishnan, and D. Manocha. MAPC: A library for efficient and exact manipulation of algebraic points and curves. Technical Report TR98-038, Department of Computer Science, University of N. Carolina, Chapel Hill, 1999. <http://www.cs.unc.edu/~geom/MAPC/>.
- [Cla92] K. L. Clarkson. Safe and effective determinant evaluation. In *Proc. 33rd Annu. IEEE Sympos. Found. Comput. Sci.*, pages 387–395, October 1992.

- [CLR90] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 1990.
- [CZ99] Jochen Comes and Mark Ziegelmann. An easy to use implementation of linear perturbations within CGAL. In *Proceedings of the 3rd Workshop on Algorithm Engineering*, volume 1668 of *Lecture Notes in Computer Science*, pages 169–182. Springer-Verlag, 1999.
- [Dau99] Marc Daumas. Multiplications of floating point expansions. In *Proceedings of the 14th Symposium on Computer Arithmetic*, Adelaide, Australia, 1999.
- [dBvKOS97] Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, Berlin, 1997.
- [Dev98] Olivier Devillers. Improved incremental randomized Delaunay triangulation. In *Proc. 14th Annu. ACM Sympos. Comput. Geom.*, pages 106–115, 1998.
- [DF99] Marc Daumas and Claire Finot. Division of floating point expansions with an application to the computation of a determinant. *Journal of Universal Computer Science*, 5(6):323–338, 1999.
- [DFMT99] Olivier Devillers, Alexandra Fronville, Bernard Mourrain, and Monique Teillaud. Exact predicates for circle arcs arrangements. Research Report 37??, INRIA, 1999. to appear.
- [DG99] Olivier Devillers and Pierre-Marie Gandoin. Rounding voronoi diagram. In *Proc. 8th Discrete Geometry and Computational Imagery conference (DGCI99)*, volume 1568 of *Lecture Notes in Computer Science*, pages 375–387. Springer-Verlag, 1999.
- [DP98] Olivier Devillers and Franco P. Preparata. A probabilistic analysis of the power of arithmetic filters. *Discrete Comput. Geom.*, 20:523–547, 1998.
- [DP99] Olivier Devillers and Franco P. Preparata. Further results on arithmetic filters for geometric predicates. *Comput. Geom. Theory Appl.*, 13:141–148, 1999.
- [DST88] J.H. Davenport, Y. Siret, and E. Tournier. *Computer Algebra*. Academic Press, London, 1988.
- [ECS97] I. Z. Emiris, J. F. Canny, and R. Seidel. Efficient perturbations for handling geometric degeneracies. *Algorithmica*, 19(1–2):219–242, September 1997.
- [Ede87] H. Edelsbrunner. *Algorithms in Combinatorial Geometry*, volume 10 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, Heidelberg, West Germany, 1987.
- [EL88] Milos D. Ercegovic and Thomas Lang. On-line arithmetic: A design methodology and applications in digital signal processing, 1988.
- [EL99] Abbas Edalat and André Lieutier. Foundation of a computable solid modeling. In *Solid Modeling*, 1999.
- [EM90] H. Edelsbrunner and E. P. Mücke. Simulation of simplicity: A technique to cope with degenerate cases in geometric algorithms. *ACM Trans. Graph.*, 9(1):66–104, 1990.

- [FGK⁺96] Andreas Fabri, Geert-Jan Giezeman, Lutz Kettner, Stefan Schirra, and Sven Schönherr. The CGAL kernel: A basis for geometric computation. In M. C. Lin and D. Manocha, editors, *Proc. 1st ACM Workshop on Appl. Comput. Geom.*, volume 1148 of *Lecture Notes Comput. Sci.*, pages 191–202. Springer-Verlag, 1996.
- [For86] S. Fortune. A sweepline algorithm for Voronoi diagrams. In *Proc. 2nd Annu. ACM Sympos. Comput. Geom.*, pages 313–322, 1986.
- [FV93] S. Fortune and C. J. Van Wyk. Efficient exact arithmetic for computational geometry. In *Proc. 9th Annu. ACM Sympos. Comput. Geom.*, pages 163–172, May 1993.
- [FV96] S. Fortune and C. J. Van Wyk. Static analysis yields efficient exact integer arithmetic for computational geometry. *ACM Trans. Graph.*, 15(3):223–248, July 1996.
- [FW93] S. Fortune and C. Van Wyk. *LN User Manual*. AT&T Bell Laboratories, 1993.
- [GGHT97] M. Goodrich, L. J. Guibas, J. Hershberger, and P. Tanenbaum. Snap rounding line segments efficiently in two and three dimensions. In *Proc. 13th Annu. ACM Sympos. Comput. Geom.*, pages 284–293, 1997.
- [Gol91] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.*, 23(1):5–48, March 1991.
- [Gra96] Torbjörn Granlund. *GMP, The GNU Multiple Precision Arithmetic Library*, 2.0.2 edition, June 1996. <http://www.swox.com/gmp/>.
- [GSS89] Leonidas J. Guibas, D. Salesin, and J. Stolfi. Epsilon geometry: building robust algorithms from imprecise computations. In *Proc. 5th Annu. ACM Sympos. Comput. Geom.*, pages 208–217, 1989.
- [GTVV93] Michael T. Goodrich, Jyh-Jong Tsay, Darren E. Vengroff, and Jeffrey Scott Vitter. External-memory computational geometry. In *Proc. 34th Annu. IEEE Sympos. Found. Comput. Sci.*, pages 714–723, 1993.
- [Hai99] Bruno Haible. *CLN, The Class Library for Numbers*, 1.0.1 edition, June 1999. <http://clisp.cons.org/~haible/packages-cln.html>.
- [HHKR95] R. Hammer, M. Hocks, U. Kulisch, and D. Ratz. *C++ Toolbox for Verified Computing*. Springer, 1995.
- [Hob93] J. Hobby. Practical segment intersection with finite precision output. Technical Report 93/2-27, Bell Laboratories, 1993.
- [HP94] C. Y. Hung and B. Parhami. An approximate sign detection method for residue numbers and its applications to RNS division. *Comput. Math. Appl.*, 27(4):23–35, 1994.
- [IEE85] *IEEE Standard for binary floating point arithmetic, ANSI/IEEE Std 754 – 1985*. New York, NY, 1985. Reprinted in SIGPLAN Notices, 22(2):9–25, 1987.
- [Jai93] P. Jaillon. LEA, a lazy exact arithmetic: Implementation and related problems. Technical Report in preparation, École Nationale Supérieure des Mines de Saint-Étienne, 1993.

- [Kah96] William Kahan. Lecture notes on the status of IEEE standard 754 for binary floating-point arithmetic, 1996.
- [KLPY99a] V. Karamcheti, C. Li, I. Pechtchanski, and C. Yap. A core library for robust numeric and geometric computation, 1999. ACM Symp. on Computational Geometry.
- [KLPY99b] Vijay Karamcheti, Chen Li, Igor Pechtchanski, and Chee Yap. *The CORE Library Project*, 1.2 edition, September 1999. <http://www.cs.nyu.edu/exact/core/>.
- [KM90] Peter Kornerup and David W. Matula. An algorithm for redundant binary bit-pipelined rational arithmetic, 1990.
- [Knü] O. Knüppel. *The PROFIL/BIAS library*. <http://www.ti3.tu-harburg.de/Software/PROFILEnglisch.html>.
- [Knu98] D. E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*. Addison-Wesley, Reading, MA, 3rd edition, 1998.
- [KW98] Lutz Kettner and Emo Welzl. One sided error predicates in geometric computing. In *Proceedings of the 15th IFIP World Computer Congress, Fundamentals - Foundations of Computer Science*, 1998.
- [Lau82] M. Lauer. Computing by homomorphic images. In B. Buchberger, G. E. Collins, and R. Loos, editors, *Computer Algebra: Symbolic and Algebraic Computation*, pages 139–168. Springer-Verlag, 1982.
- [LPT97] Giuseppe Liotta, Franco P. Preparata, and Roberto Tamassia. Robust proximity queries: an illustration of degree-driven algorithm design. In *Proc. 13th Annu. ACM Sympos. Comput. Geom.*, pages 156–165, 1997.
- [Mig92] M. Mignotte. *Mathematics for Computer Algebra*. Springer-Verlag, 1992.
- [Mil89a] V. Milenkovic. Double precision geometry: A general technique for calculating line and segment intersections using rounded arithmetic. In *Proc. 30th Annu. IEEE Sympos. Found. Comput. Sci.*, pages 500–505, 1989.
- [Mil89b] V. Milenkovic. Rounding face lattices in the plane. In *Abstracts 1st Canad. Conf. Comput. Geom.*, page 12, 1989.
- [Mil95] Victor J. Milenkovic. Practical methods for set operations on polygons using exact arithmetic. In *Proc. 7th Canad. Conf. Comput. Geom.*, pages 55–60, 1995.
- [MM97] Dominique Michelucci and Jean-Michel Moreau. Lazy arithmetic. In *IEEE Transactions on Computers*, volume 46, no. 9, pages 961–975, 1997.
- [MN98] K. Mehlhorn and S. Näher. *LEDA: A Platform for Combinatorial and Geometric Computing*. Cambridge University Press, New York, 1998.
- [Moo66] R. E. Moore. *Interval Analysis*. Prentice Hall, Englewood Cliffs, NJ, 1966.
- [Mye95] N. C. Myers. Traits: a new and useful template technique. *C++ Report*, June 1995.
- [NU95] S. Näher and C. Uhrig. *The LEDA Manual User Manual*, 1995.
- [Pio95] Sylvain Pion. Arithmétique redondante : paresseuse et exacte. Application au calcul du signe des déterminants. Rapport de magistère, École Normale Supérieure, France, 1995.

- [Pri92] D. Priest. *On properties of floating point arithmetics: numerical stability and the cost of accurate computations*. Ph.D. thesis, Dept. of mathematics, Univ. of California at Berkeley, 1992.
- [PS85] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, New York, NY, 1985.
- [Reg95] Ashutosh Rege. A complete and practical algorithm for geometric theorem proving. In *Proc. 11th Annu. ACM Sympos. Comput. Geom.*, pages 277–286, 1995.
- [Sch99] Stefan Schirra. A case study on the cost of geometric computing. In *Proceedings of Algorithm Engineering and Experimentation (ALENEX99)*, volume 1619 of *Lecture Notes in Computer Science*, pages 156–176. Springer-Verlag, 1999.
- [She96] Jonathan R. Shewchuk. Robust adaptive floating-point geometric predicates. In *Proc. 12th Annu. ACM Sympos. Comput. Geom.*, pages 141–150, 1996.
- [Slo99] N. J. A. Sloane. *The On-Line Encyclopedia of Integer Sequences*, 1999. <http://www.research.att.com/~njas/sequences/>.
- [Str86] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1986.
- [Sug92] K. Sugihara. Topologically consistent algorithms related to convex polyhedra. In *Proc. 3rd Annu. Internat. Sympos. Algorithms Comput.*, volume 650 of *Lecture Notes Comput. Sci.*, pages 209–218. Springer-Verlag, 1992.
- [Yap88] C. K. Yap. A geometric consistency theorem for a symbolic perturbation scheme. In *Proc. 4th Annu. ACM Sympos. Comput. Geom.*, pages 134–142, 1988.
- [Yap93] C. K. Yap. Towards exact geometric computation. In *Proc. 5th Canad. Conf. Comput. Geom.*, pages 405–419, 1993.
- [Yap98] Chee Yap. A new number core for robust numerical and geometric libraries. In *3rd CGC Workshop on Geometric Computing*, 1998. Invited Talk. Brown University, Oct 11–12.
- [YD95] C. K. Yap and T. Dubé. The exact computation paradigm. In D.-Z. Du and F. K. Hwang, editors, *Computing in Euclidean Geometry*, volume 4 of *Lecture Notes Series on Computing*, pages 452–492. World Scientific, Singapore, 2nd edition, 1995.