Génération de codes et d'annotations prouvables d'algorithmes de points intérieurs à destination de systèmes embarqués critiques

Guillaume Davy

THÈSE

En vue de l'obtention du
DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par l'Institut Supérieur de l'Aéronautique et de l'Espace

Présentée et soutenue par

Guillaume DAVY

Le 6 décembre 2018

Génération de codes et d'annotations prouvables d'algorithmes de points intérieurs à destination de systèmes embarqués critiques.

Ecole doctorale : **EDMITT - Ecole Doctorale Mathématiques, Informatique et Télécommunications de Toulouse**

Spécialité : **Informatique et Télécommunications**

Unité de recherche :
ISAE-ONERA MOIS MOdélisation et Ingénierie des Systèmes

Thèse dirigée par
Didier HENRION

Jury

Mme Sylvie PUTOT, Rapportrice
M. Yves BERTOT, Rapporteur
M. Eric FERON, Examinateur
Mme Sylvie BOLDO, Présidente
M. Didier HENRION, Directeur de thèse
M. Pierre-Loïc GAROCHE, Co-directeur de thèse

ONERA
THE FRENCH AEROSPACE LAB

À mes grands-pères,

Remerciements

Tout d'abord, je souhaite remercier mes rapporteurs, Sylvie Putot et Yves Bertot, pour leur relecture attentive et les remarques constructives qui en ont découlé. Je voudrais aussi remercier Sylvie Boldo et Éric Féron d'avoir accepté de faire partie de mon jury. Je remercie, évidemment, mes deux directeurs de thèses : Pierre-Loïc Garoche et Didier Henrion pour leur encadrement, leur gentillesse et leur bonne humeur. Merci d'avoir cru en ce projet et d'avoir pu rendre ce manuscrit possible.

Merci à Tanguy Perennou et Christophe Garion de m'avoir fait confiance en me laissant enseigner au sein de l'ISAE Supaéro. Merci pour leurs conseils et leurs disponibilités lors de ces enseignements. Un merci particulier à Christophe pour avoir écouté et partagé mes déboires avec Frama-C.

Merci à tous les chercheurs que j'ai pu rencontrer lors de workshops, écoles d'été et conférences. En particulier, je voudrais remercier Behçet Açıkmeşe et son équipe pour leur accueil chaleureux à UW, Seattle. Je voudrais aussi remercier Éric Féron et toutes les personnes que j'ai rencontrées à Georgia Tech. Un merci spécial à Timothy Wang et Romain Jobredeaux pour leurs travaux précédant cette thèse, leur aide et les nombreuses discussions que nous avons pu avoir sur ce sujet ou d'autres. Un grand merci, aussi, à Raphaël Cohen pour tous les échanges qu'on a pu avoir, et les heures de travail passées ensemble.

Je souhaite remercie les professeurs de l'ENS Cachan pour leur enseignement et la transmission de leur passion. Je veux aussi remercier les professeurs et enseignants que j'ai eus tout au long de ma scolarité, ils ont su me faire aimer la science. Merci au Téléthon, à la Cité des Sciences et de l'Industrie et au Palais de la Découverte pour m'avoir donné, petit, le goût de la recherche.

Je veux remercier tous les membres du DTIS pour leur aide et leurs conseils. Plus particulièrement, un très grand merci au doctorant du DTIM pour la bonne entente les midis, aux pauses, mais aussi en dehors du bureau. Cette bonne ambiance est souvent salvatrice dans les moments difficiles. Merci à l'équipe MAC du LAAS, en particulier ces doctorants, pour leur accueil, lors de mes passages erratiques au LAAS.

Merci à tous les Nadines, qui malgré la distance sont toujours là, merci pour tous ces bons moments. Un grand merci, en particulier, aux Présidents pour constamment rester fidèle à leur surnom pendant toutes ces années. Je remercie aussi tous mes amis toulousains pour leur soutien, leur compréhension et pour tout les bons moments passés autour d'une table de jeu de rôle.

Enfin, je veux remercier ma famille, pour le terrain propice à la curiosité, la science et la culture dans laquelle mon enfance a baigné. Je remercie, en particulier, mes parents, mes beaux-parents, mon frère et ma soeur pour leur soutien, leurs conseils, et tout simplement pour avoir toujours été présent quand j'en ai eu besoin.

Abstracts

Short abstract

In the industry, the use of optimization is ubiquitous. Optimization consists of calculating the best solution subject to a number of constraints. However, this calculation is complex, long and not always reliable. This is why this task has long been confined to the design stages, which allowed time to do the computation and then check that the solution is correct and if necessary redo the computation.

In recent years, thanks to the ever-increasing power of computers, the industry has begun to integrate optimization computation at the heart of the systems. That is to say that optimization computation is carried out continuously within the system, sometimes dozens of times per second. Therefore, it is impossible to check a posteriori the solution or restart a calculation. That is why it is important to check that the program optimization is perfectly correct and bug-free.

The objective of this thesis was to develop tools and methods to meet this need. To do this we have used the theory of formal proof that is to consider a program as a mathematical object. This object takes input data and produces a result. We can then, under certain conditions on the inputs, prove that the result meets our requirements. Our job was to choose an optimization program and formally prove that the result of this program is correct.

Résumé court

Dans l'industrie, l'utilisation de l'optimisation est omniprésente. Elle consiste à calculer la meilleure solution tout en satisfaisant un certain nombre de contraintes. Cependant, ce calcul est complexe, long et pas toujours fiable. C'est pourquoi cette tâche est longtemps restée cantonnée aux étapes de conception, ce qui laissait le temps de faire les calculs puis de vérifier que la solution était correcte et si besoin refaire les calculs.

Ces dernières années, grâce à la puissance toujours grandissante des ordinateurs, l'industrie a commencé à intégrer des calculs d'optimisation au coeur des systèmes. C'est-à-dire que des calculs d'optimisation sont effectués en permanence au sein du système, parfois des dizaines de fois par seconde. Par conséquent, il est impossible de s'assurer a posteriori de la correction d'une solution ou de relancer un calcul. C'est pourquoi il est primordial de vérifier que le programme d'optimisation est parfaitement correct et exempt de bogue.

L'objectif de cette thèse a été de développer outils et méthodes pour répondre à ce besoin. Pour ce faire, nous avons utilisé la théorie de la preuve formelle qui consiste à considérer un programme comme un objet mathématique. Cet objet prend des informations en entrée et produit un résultat. On peut alors, sous certaines conditions sur les entrées, prouver que le résultat satisfait nos exigences. Notre travail a consisté à choisir un programme d'optimisation puis à prouver formellement que le résultat de ce programme est correct.

Abstract

The use of convex optimization within aeronautics critical systems has emerged for the last years. The benefits of optimization can range from reducing the consumption of fuel oil to new control techniques. For example, the rocket landing done by SpaceX is based on the online solving of convex optimization problem thanks to interior point algorithms. However, these algorithms are expensive in computing time, but the growing power of computer systems allows, nowadays, their use online. Moreover, in order to reduce computing times, manufacturers choose to develop specific implementations for the problems to be solved.

Optimization, however, remains limited to secondary computations, or uncritical systems. In order to generalize their use, manufacturers will have to guarantee the implementations of these algorithms. This consists in checking : 1. no runtime error, 2. that it answers in a finite time, ideally bounded, 3. the correction of the returned solution.

Regarding the last point, for an optimization algorithm, the solution must satisfy the constraint and be optimal.

In this work, we developed tools and methods that allow to formally guarantee an interior point algorithm generated code. We used methods and tools from the formal program verification to generate a code embeddable, but also provable from a given convex optimization problem. The main contributions made by this manuscript are : 1. the creation of a language, pySil, which allows to express simple algorithms with matrix computation, as well as their specifications. 2. the writing, in pySil, of an interior point algorithm. 3. the development of a pySil compiler to the C / ACSL language to obtain a code executable online, but also provable by Frama-C. 4. the enrichment of the specification of the algorithm written in pySil so that Frama-C can prove the correction of the C/ACSL compiled files.

This approach was evaluated on examples, in particular of predictive control. On the other hand, we conducted this work with the aim of clearly explaining and minimizing software and assumptions on which the proof of correction is based.

Résumé

L'aéronautique a vu émerger ces dernières années l'utilisation d'optimisation convexe au sein de ses systèmes critiques. Les avantages apportés par l'optimisation peuvent aller de la réduction de la consommation de fioul à de nouvelles techniques de contrôle. Par exemple, l'atterrissage de fusée comme pratiqué par SpaceX se base sur la résolution en ligne de problème d'optimisation convexe grâce à des algorithmes de points intérieurs. Toutefois, ceux-ci sont couteux en temps de calcul, mais la puissance grandissante des systèmes informatiques permet, désormais, leur utilisation en ligne. De plus, afin de réduire les temps de calcul au maximum, les industriels choisissent de développer des implémentations spécifiques aux problèmes à résoudre.

L'optimisation reste cependant aujourd'hui limitée à des calculs secondaires, ou à des systèmes peu critiques. Pour généraliser leur utilisation, au sein de systèmes grand public, les industriels devront pouvoir certifier les implémentations de ces algorithmes. Cela consiste à vérifier : 1. l'absence d'erreur à l'exécution, 2. qu'eles répondent en temps fini, idéalement borné 3. la correction de la solution retournée. Dans le cas de l'optimisation, ce dernier point correspond à vérifier que la solution respecte les contraintes et est optimale.

Ce travail a consisté à développer des outils et des méthodes qui permettent de certifier formellement un code généré d'algorithme de points intérieurs. Pour cela, nous avons utilisé des méthodes et outils issus de la vérification formelle de programmes afin de générer un code embarquable, mais aussi prouvable à partir de la donnée d'un problème d'optimisation convexe.

Les principales contributions apportées par ce manuscrit sont : 1. La création d'un langage, pySil, qui permet d'exprimer simplement des algorithmes numériques matriciels, ainsi que leurs spécifications. 2. L'écriture, en pySil, d'un algorithme de points intérieurs. 3. Le développement d'un compilateur de pySil vers le langage C/ACSL afin d'obtenir un code à la fois exécutable en ligne, mais aussi prouvable par Frama-C. 4. L'enrichissement de la spécification de l'algorithme écrit en pySil afin que Frama-C puisse prouver la correction de sa compilation en C/ACSL.

L'approche a pu être évaluée sur des exemples, en particulier de commande prédictive. D'autre part, nous avons mené ce travail avec le souci d'expliciter clairement et réduire au maximum les logiciels et hypothèses sur lesquels repose la preuve de correction.

Table des matières

Liste des figures

Notations

$\mathbb{B}$	Ensemble des booléens : $\mathbb{B} = \{\boldsymbol{V}, \boldsymbol{F}\}$ avec $\boldsymbol{V}$ vrai et $\boldsymbol{F}$ faux.
$\mathbb{N}, \mathbb{Z}, \mathbb{R}$	Respectivement ensemble des entiers naturels, relatifs et des réels.
$\mathbb{R}^+$	Ensemble des réels positifs.
$\mathbb{R}^{+*}$	Ensemble des réels positifs, non-nuls.
$[\![m, n[\![$	Ensemble des entiers entre m et n avec m inclus et n exclu.
$[a, b[$	Interval de $\mathbb{R}$ entre a et b avec a inclus et b exclu.
$\mathcal{P}(E)$	Ensemble des parties de E.
$E \to F$	Ensemble des fonctions de E dans F
$f \circ g$	Composition de g par f.
$\dot{y}$	Dérivée de $y : \mathbb{R} \to \mathbb{R}^n$.
$Df(x)$	Différentielle de f au point x.
$f(\bullet, y_0)$	Fonction définie par $\begin{array}{ccc} E & \to & G \\ x & \mapsto & f(x, y_0) \end{array}$ pour $f : (E \times F) \to G$.
$\mathbf{Id}_E$	Fonction identitée de E dans E
$f \sim_a g$	$f, g : A \subset \mathbb{R} \to \mathbb{R}$ sont équivalentes en a adhérent à A.
$\min_{x \in E} f(x)$	Minimum de f sur E.
$\inf_{x \in E} f(x)$	Borne inférieure de l'image de E par f.
$\arg\min_{x \in E} f(x)$	Ensemble des points de E où le minimum est atteint.
$\arg\inf_{x \in E} f(x)$	Ensemble des points de E où la borne inférieure est atteinte.
$f \uplus \{x_0 \mapsto y_0\}$	Fonction définie par $x \mapsto \begin{cases} y_0 \text{ si } x = x_0 \\ f(x) \text{ sinon} \end{cases}$.
$B(x, r)$	Boule de centre $x \in \mathbb{R}^n$ et de rayon $r \in \mathbb{R}$
x_i	i-ième composante du vecteur x.
$\langle x, y \rangle$	Produit scalaire entre x et y appartenant à $\mathbb{R}^n$.
$x < y$	Pour $x, y \in \mathbb{R}^n$, $x < y \Leftrightarrow \forall i \in [\![0, n]\!], x_i < y_i$.
$x[i \leftarrow v]$	Pour $x \in E^n$, $\forall j \in [\![0, n[\![, (x[i \leftarrow v])_j = \begin{cases} v & \text{si } i = j \\ (x)_j & \text{sinon} \end{cases}$.
$\mathbb{R}^{m \times n}$	Ensemble des matrices à m lignes et n colonnes à valeur dans $\mathbb{R}$.
$\mathcal{S}_n$	Ensemble des matrices symétriques de taille n à valeur dans $\mathbb{R}$.
$\mathcal{S}_n^+$	Ensemble des matrices semi-définies positives.
$\mathcal{S}_n^{++}$	Ensemble des matrices définies positives.
$[A]_{i,j}$	Élément de la matrice A situé à la $i$ième ligne de la j-ième colonne.
$B \succcurlyeq A$	$B - A$ est définie positive.
$E[x/M]$	Expression E dont on a remplacé les x par des M.

Première partie

Contexte

Chapitre 1

Introduction

Une des évolutions majeures de l'industrie aérospatiale de ces dernières années est sûrement la part grandissante du secteur privé dans cette industrie restée longtemps la chasse gardée des États. La montée la plus fulgurante est celle de SpaceX qui en 2017 représentait à elle seule 20% des lancements orbitaux de la planète contre à peine plus de 10% en 2016. Une grande partie de sa stratégie est basée sur la réutilisation des lanceurs. En particulier, depuis fin 2015, SpaceX fait atterrir le premier étage de ses lanceurs afin de pouvoir les réutiliser. L'atterrissage est possible grâce à l'utilisation dans le contrôleur de la fusée d'un algorithme d'optimisation convexe. Comme expliqué dans Blackmore, 2016, l'optimisation convexe permet un atterrissage avec une précision de l'ordre de la dizaine de mètres.

Un algorithme d'optimisation consiste à trouver parmi toutes les solutions d'un problème donné, celle minimisant une fonction donnée que l'on appelle coût. Les algorithmes permettant de résoudre de tels problèmes sont complexes. Or, le contrôleur est la partie la plus critique d'un engin astronautique ou aéronautique. C'est pourquoi les algorithmes d'optimisation sont utilisés en contrôle, mais, le plus souvent, avant le lancement du système. Ainsi, cela laisse le temps de vérifier la solution générée par l'algorithme. Utiliser ces algorithmes en ligne pourrait apporter un contrôle de bien meilleure qualité, permettre de réduire les consommations de carburant ou bien encore rendre possibles des manœuvres impossibles jusqu'alors. Cependant, si des risques importants apparaissent, monétaires ou humains, il faudra nécessairement passer par une phase de certification. En effet, tout logiciel embarqué dans un système aéronautique doit être certifié par une autorité étatique. Son but est de garantir, entre autres, que tous les logiciels utilisés au sein d'un système accueillant des passagers fonctionneront correctement. En d'autres termes, ils veulent s'assurer qu'un programme ne possède pas de bogue.

Les entreprises doivent donc apporter à ces autorités la preuve que leur code fonctionnera toujours correctement. Pour ce faire, l'informatique théorique apporte des solutions formelles. Par solution formelle, on entend des méthodes prouvant mathématiquement l'absence de bogue. Certaines de ces méthodes ont l'avantage d'être en grande partie automatisables grâce à l'informatique elle-même. Cela consiste donc à écrire des programmes manipulant d'autres programmes afin, en l'occurrence, de prouver leur fonctionnement.

Une autre spécificité liée à l'utilisation de l'optimisation convexe au sein d'un contrôleur est la nécessité de produire un code spécifique au problème d'optimisation. En effet, d'une part, une grande partie du problème est connu à l'avance. D'autre part, on a besoin d'une réponse très rapide. Ainsi, générer automatiquement le code permet d'obtenir un programme spécifique au problème et donc très efficace.

Notre problématique est de développer des méthodes de vérification d'implémentation d'algorithme numérique, plus précisément dans le cadre de la génération automatique de code d'optimisation. En absence de certification formelle du code embarqué, bien que les théories garantissent un parfait contrôle des systèmes, un simple bogue dans l'implémentation peut conduire aux pires catastrophes possibles pour le système. Une première source de bogue

potentiel est un dépassement du temps alloué par le système au calcul de la solution. Une seconde source est une réponse erronée du calcul. La troisième est une erreur d'exécution du programme.

Wang, Jobredeaux, Pantel et al., 2016 ont proposé un algorithme d'optimisation et son implémentation en Matlab™. Ils ont ensuite annoté chaque instruction du programme avec une formule logique représentant l'état de la mémoire avant et après l'instruction. Ils ont ensuite justifié informellement que chaque instruction transformait la mémoire comme spécifié par l'annotation. Cet article a permis d'ouvrir la voie à la vérification formelle d'algorithmes d'optimisation convexe. L'article, cependant, ne donne aucun moyen automatique de vérifier que les annotations données sont correctes. En effet, la méthode nécessite un travail humain trop conséquent pour être utilisée dans l'industrie. De plus, le programme est écrit en Matlab™ ce qui est éloigné des problématiques de l'embarqué critique. Enfin, les aspects génération de code ne sont pas abordés en profondeur.

Dans Wang, Jobredeaux, Herencia-Zapana et al., 2013, on peut trouver un cadre logiciel complet pour générer le code d'un contrôleur depuis une représentation haut niveau en Matlab™. En plus du code, des annotations sont aussi générées depuis la même représentation. Le code généré est ensuite prouvé automatiquement à travers l'utilisation de tactiques automatiques dans un assistant de preuve. Cela correspond presque parfaitement à notre problématique. L'article se situe complètement dans la thématique du contrôle et on y retrouve le concept de la génération de code et de la preuve formelle. En revanche, l'article traite principalement du contrôle linéaire et donc n'aborde pas d'algorithmes plus complexes de type numérique, comme ceux d'optimisation qui nous intéressent dans cette thèse.

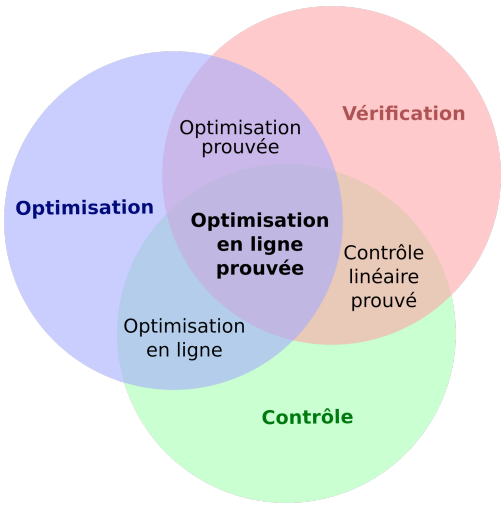
L'article Mattingley et Boyd, 2012 est, en quelque sorte, le complémentaire de l'article précédent. C'est un des articles majeurs de la génération de code d'optimisation convexe à destination de l'embarqué. Il présente cvxgen, un outil utilisé dans l'industrie permettant de prendre la représentation d'une famille de problèmes d'optimisation convexe en entrée et génèrant à partir de celui-ci un code spécifique et efficace les résolvant. Les auteurs ont effectué de nombreux tests sur le code généré qui est comparé aux meilleurs solveurs du marché. Les temps d'exécution sont bas : le gain est d'un facteur d'environ 100 pour tous les exemples proposés. Bien que l'article mentionne très clairement l'importance fondamentale de générer un code correct, il n'aborde pas plus en détail cette problématique.

L'article Bemporad, Borrelli et Morari, 2002 développe un algorithme permettant le calcul hors ligne de solutions explicites de problèmes d'optimisation linéaire paramétrés. Pour cela, l'algorithme découpe l'ensemble des paramètres en régions et résout le problème d'optimisation sur chaque région. Il ne reste plus ensuite qu'à embarquer la liste de toutes les régions et de leurs solutions. Quand le système reçoit un paramètre, il a juste à chercher dans quelle région il se trouve et à retourner la solution correspondante. Comme tous les calculs d'optimisation sont effectués hors ligne, on peut les vérifier hors ligne. L'algorithme embarqué est lui très simple et donc ne serait pas compliqué à certifier. Cependant, le nombre de régions est exponentiel en la taille du problème d'optimisation. Ainsi, le calcul hors ligne prend un temps exponentiel ce qui, bien que hors ligne, deviendra prohibitif pour les problèmes de grande taille. De plus, un espace mémoire exponentiel sera nécessaire pour stocker toutes les régions, ce qui rendra à partir d'une certaine taille impossible son utilisation, en particulier dans l'embarqué où la taille de la mémoire peut être une contrainte importante.

L'ensemble de ces études montre qu'il existe des méthodes de génération de code d'optimisation à destination des systèmes embarqués, mais aussi des méthodes de vérification

d'algorithme numérique. L'article Wang, Jobredeaux, Pantel et al., 2016, présente même un code d'optimisation Matlab™ annoté. Toutefois, le travail effectué sur la génération de code ne tient pas compte des problématiques de vérification. Dans le cas du travail sur la preuve formelle de code d'optimisation, l'utilisation de Matlab™ ne reflète pas toutes les problématiques d'implémentation, en particulier les aspects mémoires. De plus, la méthode repose sur une annotation manuelle d'un code donné alors que la génération automatique du code ainsi que sa validation automatique sont nécessaires pour que l'utilisation de telles méthodes se généralise. L'article Wang, Jobredeaux, Herencia-Zapana et al., 2013 se limite à la preuve d'algorithmes de contrôle très simple, éloigné des problématiques de l'optimisation convexe. Enfin, le dernier article Bemporad, Borrelli et Morari, 2002, propose une méthode alternative prometteuse, mais est limité à de petits exemples.

C'est ce qui justifie cette étude. Elle consiste à trouver les moyens de générer un code exempt de bogue résolvant des problèmes d'optimisation convexe dans le cadre des systèmes embarqués critiques.



Ce travail se situe à l'interface de trois domaines : l'optimisation qui représente notre matière de travail, la vérification qui représente nos outils et le contrôle qui représente notre motivation. Nous présentons dans le chapitre 2 les éléments nécessaires à l'intersection de chacun de ces trois domaines.

Dans le premier chapitre de contribution, le chapitre 3, l'idée est de créer un langage, pySil, permettant l'expression d'un code numérique pour l'embarqué mais aussi d'exprimer les propriétés que devra vérifier ce code. Dans un premier temps, nous choisissons un paradigme pour ce langage et une méthode de spécification. Ce choix nous permet, dans un second temps, de construire un langage adéquat pour écrire programmes et propriétés. Dans un troisième temps, l'idée est d'enrichir cette syntaxe avec le support du calcul matriciel afin de pouvoir exprimer des programmes plus complexes et conformes à notre objectif. Enfin, dans un quatrième temps, le but est de montrer que l'extension matricielle ne complexifie pas notre langage d'origine.

Fort du langage pySil créé dans le chapitre précédent, le deuxième chapitre de contribution, le chapitre 4, est consacré à l'écriture d'un algorithme d'optimisation annoté dans ce langage afin de l'utiliser par la suite pour générer un code embarquable et prouvable. Pour cela, nous

commençons par choisir l'algorithme d'optimisation le plus intéressant à prouver. Une fois choisi, nous implémentons l'algorithme en pySil et en profitons pour donner les grandes lignes de la preuve l'algorithme afin d'en déceler les points importants. Une fois l'algorithme écrit, nous étudions comment on peut utiliser cet algorithme au sein d'un problème de contrôle. Une partie des résultats de ce chapitre ont fait l'objet de deux publications : Feron et al., 2017 et Cohen et al., 2017.

Le chapitre 5 est consacré à la génération automatique à partir d'un algorithme écrit en pySil d'un code embarquable et prouvable automatiquement. La première étape consiste à choisir un cadre logiciel permettant la validation d'annotations sur un code embarquable. L'idée est ensuite de construire un compilateur pySil permettant à la fois d'obtenir un code embarquable, mais aussi un code annoté pouvant être analysé et prouvé. Enfin, l'objectif de la dernière section est de nous assurer que notre compilateur fonctionne correctement en l'exécutant sur des exemples.

Dans le dernier chapitre de contribution, le chapitre 6, l'idée est de s'assurer que le code généré est automatiquement prouvable par le cadre logiciel choisi. Cela permet d'obtenir une méthode sans interaction humaine allant du problème d'optimisation pySil à un code prouvé. Dans un premier temps, nous mettons en place une méthode itérative d'enrichissement du compilateur et des annotations afin de l'appliquer à notre algorithme écrit en pySil. Dans un second temps, nous cherchons à automatiser les interactions entre notre compilateur et le cadre logiciel de preuve afin de simplifier l'utilisation de la méthode développée. Dans un troisième temps, l'idée est d'utiliser cette méthode afin d'enrichir le compilateur puis d'enrichir l'algorithme afin d'obtenir un code totalement prouvable. Dans un dernier temps, l'objectif est de tester le cadre logiciel complet sur des exemples afin de voir comment il passe à l'échelle. Les résultats de ce chapitre ont été accepté à LPAR-22 : Davy et al., 2018.

Une fois les quatre chapitres de contribution terminés, nous aborderons, dans le chapitre 7 , 5 points importants présentant des limites de l'approche ou des possibilités d'amélioration. Dans une première section, nous étudions comment borner le temps d'exécution dans le pire des cas. Dans une deuxième section, nous abordons le sujet de l'optimisation du code généré. On présente ensuite les moyens permettant d'étendre notre travail à d'autres algorithmes d'optimisation convexe. La quatrième section est l'occasion de s'intéresser au problème du calcul flottant. Nous reviendrons enfin, dans une dernière section, sur tous les éléments qu'il faut supposer corrects afin d'avoir la garantie que le code exécuté est correct.

Le code développé pour cette thèse peut être trouvé à l'adresse suivante : `https://github.com/davyg/pyOptimToC`.

Bibliographie

Bemporad, Alberto, Francesco Borrelli et Manfred Morari (déc. 2002). "Model predictive control based on linear programming - the explicit solution". In : *IEEE Transactions on Automatic Control* 47.12, p. 1974-1985 (cf. p. 4, 5, 76).

Blackmore, Lars (2016). In : *IEEE Control Systems Magazine* 36.6, p. 24-26 (cf. p. 3, 76).

Cohen, Raphael, Guillaume Davy, Eric Feron et Pierre-Loic Garoche (2017). "Formal Verification for Embedded Implementation of Convex Optimization Algorithms". In : *IFAC-PapersOnLine* 50.1, p. 5867-5874 (cf. p. 6, 94, 164).

Davy, Guillaume, Eric Feron, Pierre-Loic Garoche et Didier Henrion (nov. 2018). "Formal verification of an interior point algorithm instanciation". In : *LPAR-22* (cf. p. 6, 139, 164).

Feron, Eric M., Raphael P. Cohen, Guillaume Davy et Pierre-Loic Garoche (jan. 2017). "Validation of Convex Optimization Algorithms and Credible Implementation for Model Predictive Control". In : *AIAA Scitex* (cf. p. 6, 94, 164).

Mattingley, Jacob et Stephen Boyd (mar. 2012). "CVXGEN : a code generator for embedded convex optimization". In : *Optimization and Engineering* 13.1, p. 1-27 (cf. p. 4, 16, 147).

Wang, Timothy, Romain Jobredeaux, Heber Herencia-Zapana et al. (2013). "From Design to Implementation : an Automated, Credible Autocoding Chain for Control Systems". In : *CoRR* abs/1307.2641. arXiv : 1307.2641 (cf. p. 4, 5, 103, 135).

Wang, Timothy, Romain Jobredeaux, Marc Pantel et al. (déc. 2016). "Credible autocoding of convex optimization algorithms". In : *Optimization and Engineering* 17.4, p. 781-812 (cf. p. 4, 5).

Chapitre 2

Outils et Méthodes

Sommaire

2.1 Contrôle

Dans cette section, nous allons introduire ce qui motive ce travail à savoir l'utilisation en ligne de l'optimisation convexe. Nous commencerons par quelques considérations générales sur le contrôle de manière générale puis nous aborderons la commande prédictive à horizon fini.

Les notations utilisées dans cette section ne sont pas toutes canoniques. Nous avons fait ce choix afin que les noms restent les mêmes quand nous utiliserons l'optimisation convexe pour le contrôle.

2.1.1 Système physique

Le domaine du contrôle est l'étude des moyens permettant de générer des commandes pour un système physique afin qu'il remplisse une tâche décidée au préalable. Le plus souvent cela consiste à commander des systèmes créés par l'homme. Cela peut aller d'un radiateur à une fusée en passant par une voiture, un avion ou une machine à laver. Avant de présenter ce qu'est le contrôle, il nous faut définir ce qu'est un système à contrôler.

Définition abstraite

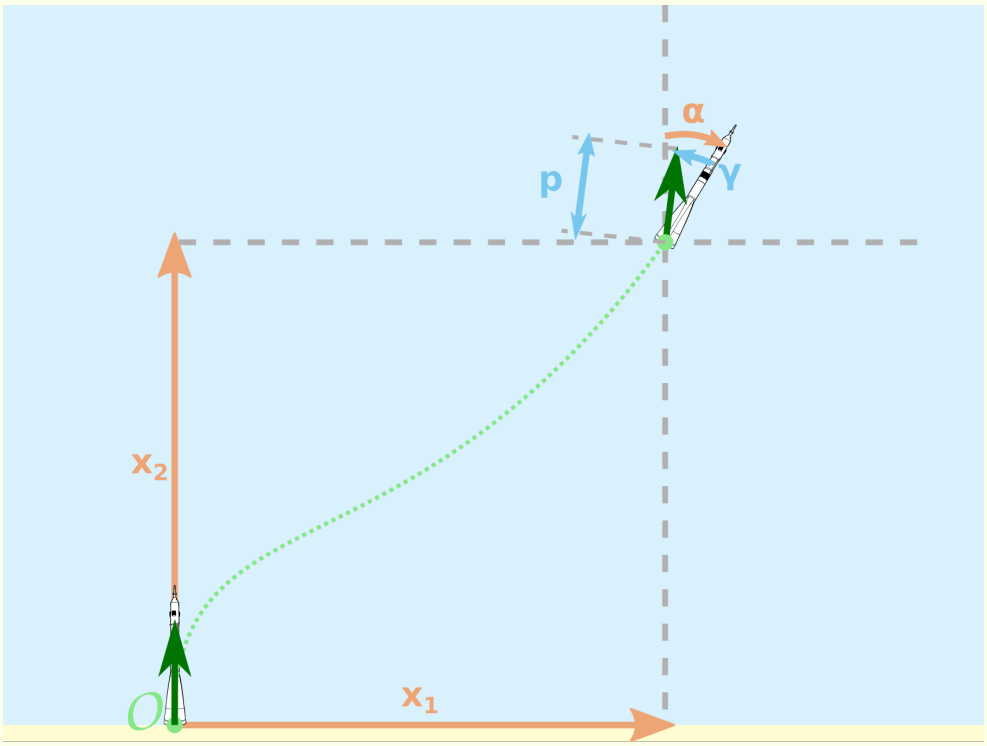
On va représenter un système physique par une boîte noire prenant une grandeur en entrée et produisant une grandeur en sortie et on le nomme système :

Ces deux grandeurs évoluent dans le temps. La grandeur de sortie se note $y : \mathbb{R} \to E$ et la grandeur d'entrée se note $u : \mathbb{R} \to C$. La première varie dans E l'ensemble des sorties et la seconde dans C l'ensemble des commandes. E et C sont généralement des vecteurs de réels et on prendra donc toujours dans la suite : $E = \mathbb{R}^e$ et $C = \mathbb{R}^c$ avec e et c des entiers. Étant donné une commande u et une sortie y, le couple (u, y) est appelé une trajectoire du système.

Un système pourra représenter l'objet physique que l'on souhaite contrôler, comme une fusée, mais aussi l'objet que l'on souhaite contrôler accompagné de son environnement. Ce sera un choix de conception. Dans ce travail, l'idée est que le système représente ce que l'on va contrôler. Les entrées du système, aussi appelées commande, représenteront les grandeurs du système que l'on peut contrôler, et, les sorties, les grandeurs qui permettront au contrôleur d'effectuer le calcul de la commande. Ces dernières peuvent donc être des grandeurs du système à contrôler comme sa vitesse ou son accélération, mais aussi des grandeurs de l'environnement comme la pression de l'air ou la force du vent.

Exemple :

Nous allons prendre l'exemple d'une fusée dans un plan en deux dimensions. La sortie de ce système est composée de sa position : (x_1, x_2) et l'angle α de la fusée avec la verticale. Sa commande est composée de deux grandeurs : l'angle γ représentant la direction de la poussée des moteurs et la puissance délivrée que l'on nommera p. Ainsi on a $e = 3$ et $c = 2$. La figure 2.1 illustre cet exemple.
Nous avons proposé une sortie très simple pour notre système, mais on peut très bien l'enrichir en fonction des besoins. On pourrait rajouter la masse totale qui représente aussi la quantité de carburant ou bien sa vitesse, ou encore son accélération. On peut aussi rajouter à la sortie de notre système des variables externes à la fusée comme le vent ou la pression atmosphérique. Ces variables devront alors être mesurées ou calculées par la fusée.



En orange, on a représenté les sorties et en bleu les commandes. Le vecteur vert représente la poussée de la fusée.

FIGURE 2.1 – Exemple de système physique

Modélisation

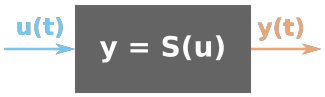
On parle de boîte noire, car par nature un système physique ne peut être exprimé parfaitement. On peut en revanche le modéliser. Cela consiste à se donner un opérateur S reliant la commande du système à sa sortie : $y = S(u)$. S est un opérateur qui, si on lui donne la fonction représentant l'évolution des entrées au cours du temps, produira une fonction représentant l'évolution des sorties.

Définition 2.1.1 (Modèle) *Soit e et c des entiers.*

Un modèle déterministe S est une fonction qui prend en entrée une fonction de $\mathbb{R} \to \mathbb{R}^c$ et produit en sortie une fonction de $\mathbb{R} \to \mathbb{R}^e$.

Un modèle non déterministe S est une fonction qui prend en entrée une fonction de $\mathbb{R} \to \mathbb{R}^c$ et produit un sous-ensemble de fonction de $\mathbb{R} \to \mathbb{R}^e$.

Si rien n'est précisé, le modèle est supposé déterministe et ce sera toujours le cas dans ce document.



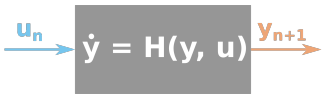
Ainsi, bien qu'à chaque instant t_0 le système prenne une commande $u(t_0)$ et produise une sortie $y(t_0)$, $y(t_0)$ ne dépend pas uniquement de $u(t_0)$, il peut dépendre de la valeur de u à n'importe quel moment. Cependant, tant que l'on modélise un système physique, $y(t_0)$ dépendra uniquement des $u(t)$ avec $t < t_0$ (principe de causalité).

En général, on introduit une notion d'état du système distinct de sa sortie. S génère alors l'évolution de l'état du système et y ne représente qu'une partie de l'état, celle à laquelle le contrôleur a accès. Nous supposerons que l'on a toujours accès à l'intégralité de l'état du système. Ainsi, on pourra parler pour y indépendamment de sortie ou d'état du système. On peut maintenant définir plus formellement la notion de trajectoire.

Définition 2.1.2 (Trajectoire) *Une trajectoire est une fonction $T : \mathbb{R}^+ \to \mathbb{R}^c \times \mathbb{R}^e$ telle qu'il existe $u : \mathbb{R} \to \mathbb{R}^c$ et $y : \mathbb{R} \to \mathbb{R}^e$, tel que $\forall t \in I, T(t) = (u(t), y(t))$ et $y = S(u)$.*

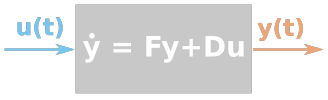
Équation différentielle

Pour définir un modèle S, on peut se donner une équation différentielle ordinaire de la forme $\dot{y} = H(y, u)$ où $\dot{y}$ est la dérivé temporelle de y et S associera à chaque u les solutions y de $\dot{y} = H(y, u)$. Le modèle sera donc déterministe si et seulement si l'équation différentielle admet une unique solution.



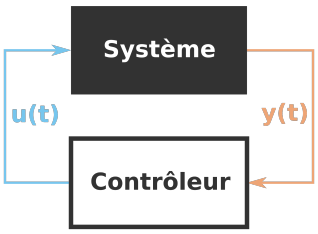
Modèle linéaire

La modélisation la plus courante est la modélisation par une équation différentielle linéaire. Cela consiste à choisir H linéaire. On peut alors exprimer l'équation différentielle sous la forme : $\dot{y} = Fy + Du$ avec $F \in \mathbb{M}_{s,s}$ et $D \in \mathbb{M}_{c,s}$.



2.1.2 Contrôleur

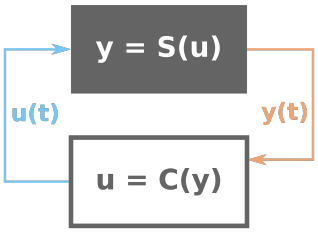
Un contrôleur est en général un calculateur qui va générer à partir de l'état actuel du système, une suite de commande pour celui-ci. Il peut lui aussi être vu comme une boîte noire. C'est en quelque sorte le complémentaire du système puisqu'il prend sa sortie en entrée et produit une commande pour celui-ci.



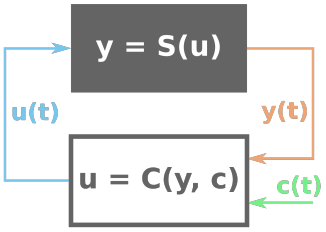
On peut définir une notion de contrôleur très générale :

Définition 2.1.3 (Contrôleur) *Soit s et c des entiers. S un système.*

On appelle contrôleur de S toute fonction C qui prend en entrée une fonction $\mathbb{R} \to \mathbb{R}^e$ et produit une fonction $\mathbb{R} \to \mathbb{R}^c$.



Le contrôleur peut aussi prendre en entrée une consigne issue d'un processus informatique de plus haut niveau et/ou de l'utilisateur. C'est ce qui se passe lorsqu'on a un thermostat. On donne une consigne de température à un contrôleur qui va ensuite faire en sorte que la température évolue vers cette consigne.



Une des familles de contrôleurs les plus répandues est celle des correcteurs. Les correcteurs sont des contrôleurs prenant une consigne en entrée. On peut leur associer une projection linéaire P qui permet de définir le terme d'erreur $e(t) = |Py(t) - c(t)|$ qu'ils doivent minimiser.

Ils permettent dans un premier temps de stabiliser un système dans un état donné. Par exemple, on peut vouloir stabiliser un quadrirotor à une hauteur h donnée en consigne. Toute variation autour de la hauteur h devra entraîner une réaction proportionnée du contrôleur qui visera à replacer le quadrirotor à sa position h. On peut aussi se servir d'un correcteur pour commander le système. Dans le cas du quadrirotor, si on change la consigne de hauteur de h à $h + \Delta h$, le correcteur détectera une erreur de Δh et la corrigera, ce qui déplacera le quadrirotor à la hauteur $h + \Delta h$.

2.1.3 Commande prédictive

Les correcteurs, bien que très répandus, sont des contrôleurs qui se contentent de regarder le passé pour contrôler le système. Cela n'est pas toujours suffisant. On peut faire une analogie avec un humain marchant en ville. Dans cette analogie, un correcteur serait un humain qui se

déplace sur un trottoir en ne regardant que la distance entre ses pieds et le bord du trottoir. Il s'assurerait alors de toujours rester à une distance constante du bord du trottoir. S'il est sur une route vide, cela est suffisant. En revanche, s'il y a un poteau devant le piéton et qu'il ne regarde que ses pieds, il rentrera en collision avec celui-ci. Aussi, si le trottoir est à sa gauche et que le trottoir fait un virage à droite, le piéton n'aura pas une trajectoire optimale. En pratique, un piéton ne regarde pas ses pieds, mais devant lui. Ceci lui permet d'anticiper les obstacles et d'optimiser sa trajectoire. La différence entre contrôleur correcteur et commande prédictive est exactement la même.

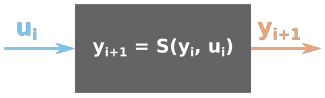
La commande prédictive consiste à produire des contrôleurs qui vont utiliser le modèle du système que l'on souhaite contrôler pour chercher une trajectoire à suivre. Cette étape de construction de la trajectoire se passe au sein même du contrôleur et sera donc exécutée régulièrement par le système. L'intérêt repose dans le fait que l'on va pouvoir générer une trajectoire qui respecte un certain nombre de contraintes et éventuellement optimiser certains critères.

Nous allons voir comment mettre en place un tel contrôleur, que l'on appellera dans la suite contrôleur MPC (pour *Model Predictive Control*). Mais avant cela, nous allons définir un système discret.

Modèle linéaire discret

Les contrôleurs dans la grande majorité des cas sont exécutés par des systèmes informatiques qui sont par nature discrets. Cela se traduit par le fait que la fonction u sera une fonction constante par morceau et non une fonction continue. Concrètement, au sein du système, une fonction est appelée toutes les $\frac{1}{f}$ secondes. Cette fonction prend en argument les valeurs des capteurs, c'est la sortie du système. La fonction calcule alors la commande du système et la retourne. Cette valeur restera alors la même jusqu'au prochain appel de la fonction. f est appelée la fréquence d'échantillonnage. Ainsi, au lieu d'utiliser un modèle linéaire continu on va utiliser un modèle discret qui reflétera mieux le comportement du contrôleur du système.

Dans un modèle discret, on remplace les fonctions $u : \mathbb{R} \to \mathbb{R}^c$ et $y : \mathbb{R} \to \mathbb{R}^e$ par des suites $u_i \in (\mathbb{R}^c)^{\mathbb{N}}$ et $y_i \in (\mathbb{R}^e)^{\mathbb{N}}$:



La version discrète d'un modèle linéaire reste très proche de sa version continue :

Définition 2.1.4 (Modèle linéaire discret) *Un modèle linéaire discret est un couple de matrices $(F, D) \in \mathbb{R}^{s\times s} \times \mathbb{R}^{c\times s}$. Il associe à une suite u_i la suite y_i définie par : $y_{i+1} = Fy_i + Du_i$.*

Pour la suite de cette section, nous nous donnons (F, D) un modèle linéaire discret.

Remarque : Si le modèle discret correspond mieux aux problématiques informatiques, quand on cherche à modéliser un système physique, il est, la plupart du temps, beaucoup plus naturel de travailler en continu. Fort heureusement on peut facilement passer d'un modèle linéaire continu à un modèle linéaire discret. Soit $f \in \mathbb{R}$ une fréquence d'échantillonnage, nous définissons la discrétisation (F, D) d'un modèle continu (F_c, D_c) par :

$$F = \mathrm{e}^{\frac{1}{f}F_c} \tag{2.1}$$

$$D = \left(\int_{\tau=0}^{\frac{1}{f}} \mathrm{e}^{\tau F_c} \, d\tau \right) D_c \tag{2.2}$$

Contrôleur MPC

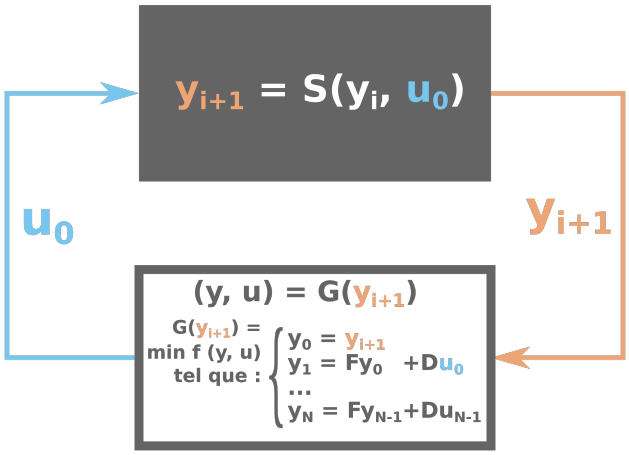
Avant de pouvoir définir un contrôleur MPC, nous avons besoin de la notion de trajectoire et d'horizon.

Définition 2.1.5 (Trajectoire) *Soit $N \in \mathbb{N}$, $u_n \in (\mathbb{R}^c)^{N-1}$ et $y_n \in (\mathbb{R}^e)^N$.*

(u_n, y_n) est une trajectoire pour le modèle linéaire (F, D) si pour tout $n \in [0, N[$, on a $y_{n+1} = Fy_n + Du_n$.

N sera appelé l'horizon du système.

Un contrôleur MPC pour un modèle S est une fonction, qui prenant en argument une sortie **y0** du système, génère une trajectoire pour S d'horizon fixe et renvoie la première commande de cette trajectoire. Ceci forme un contrôleur et cette procédure est donc réitérée à chaque itération de la boucle de contrôle.



Définition 2.1.6 (Contrôleur MPC) *Une fonction $G : \mathbb{R}^e \to (\mathbb{R}^c)^{N-1} \times (\mathbb{R}^e)^N$ est un générateur MPC d'horizon N pour le système (F, D) si pour chaque valeur $\mathbf{y0} \in \mathbb{R}^e$ $(u_n, y_n) = G(\mathbf{y0})$ est une trajectoire de (F, D) et $y_0 = \mathbf{y0}$.*

Le contrôleur MPC associé au générateur G est la fonction $C = P \circ G$ avec P une fonction linéaire de projection qui renvoie la composante u_0 d'un vecteur de l'image de G.

On a défini une version minimale d'un MPC, en pratique on pourra rajouter d'autres contraintes et optimiser la trajectoire. Par exemple, on peut vouloir borner les commandes, optimiser la consommation d'énergie voir imposer un ensemble de position acceptable au système.

Ainsi, en général, on pourra écrire la fonction G comme un problème d'optimisation. Soit $f : \mathbb{R}^{N \times e + (N-1) \times c} \to \mathbb{R}$ une fonction de coût et $E \subset \mathbb{R}^{N \times e + (N-1) \times c}$ l'ensemble des trajectoires admises. G est la fonction qui à **y0** associe la solution du problème suivant :

$$\inf_{X \in \mathbb{R}^{N \times e + (N-1) \times c}} f(X) \text{ soumis à } \begin{cases} X \in E \\ y_0 = \mathbf{y0} \\ y_1 = Fy_0 + Du_0 \\ \dots \\ y_N = Fy_{N-1} + Du_{N-1} \end{cases} \quad \text{avec } X = \begin{pmatrix} y_0 \\ \dots \\ y_N \\ u_0 \\ \dots \\ u_{N-1} \end{pmatrix}. \tag{2.3}$$

Implémenter un contrôleur reposant sur une telle fonction G implique de devoir embarquer au sein du système le plus critique, c'est-à-dire le contrôleur, un algorithme d'optimisation permettant de résoudre le problème de l'équation (2.3). Ce sont en général des algorithmes complexes et très coûteux en puissance de calcul. On peut cependant remarquer qu'une grande partie du problème d'optimisation est connu à l'avance. Seul l'état actuel du système, **y0**, est nécessaire à l'exécution pour construire le problème. Cela veut dire qu'au lieu d'embarquer un solveur générique on peut embarquer un solveur spécifique et très optimisé pour notre problème. C'est l'idée derrière Mattingley et Boyd, 2012 qui prend un problème d'optimisation convexe paramétré par une variable (dans notre cas, cela serait **y0**) et produit un code C.

Bilan

On a vu qu'une branche de la théorie du contrôle reposait sur l'exécution en ligne d'algorithme d'optimisation afin de calculer la prochaine commande du système. La solution du problème d'optimisation contient entre autres une trajectoire complète du système pour un horizon donné. Un MPC classique n'utilise que la première commande de la trajectoire. À chaque itération, il calcule une nouvelle trajectoire partant du nouvel état du système et ne garde que la première commande. Si générer une nouvelle trajectoire à chaque itération est trop couteux, on peut aussi envisager d'utiliser plus de commandes de la trajectoire générée. Cela permet d'utiliser l'algorithme d'optimisation moins souvent, mais entraîne une moins bonne réactivité.

Si E et f sont convexes dans l'équation (2.3) le problème est convexe, car les seules autres contraintes sont linéaires. En pratique, c'est souvent le choix qui est retenu, car comme on va le voir dans la section suivante, des algorithmes efficaces existent pour résoudre les problèmes convexes.

2.2 Optimisation

L'optimisation est un ensemble de techniques mathématiques visant à trouver les maximums et minimums de fonctions. On parle d'optimisation linéaire si la fonction à optimiser, aussi appelée fonction de coût, est linéaire, et si son domaine est borné par des fonctions linéaires. Si on a des fonctions quadratiques ou convexes à la place de linéaires, on parlera respectivement d'optimisation quadratique ou convexe. On construit ainsi des ensembles de problèmes d'optimisation. On appellera ces ensembles des classes et on parlera de sous-classe pour signifier qu'une classe est incluse dans une autre. Par exemple la classe des problèmes linéaires est une sous-classe de la classe des problèmes convexes.

Dans cette section, nous allons présenter dans un premier temps les domaines d'optimisation qui seront abordés dans la suite de ce travail : l'optimisation linéaire puis sa généralisation aux problèmes semi-définis. Dans un second temps, nous présenterons brièvement des méthodes pour les résoudre. Nous commencerons par une présentation de l'algorithme du simplexe puis après quelques rappels sur la méthode de Newton nous aborderons les algorithmes de points intérieurs. Tous les résultats présentés dans cette section sont classiques et peuvent être trouvés dans les ouvrages traitant d'optimisation convexe comme Boyd et Vandenberghe, 2004.

2.2.1 Présentation de l'optimisation linéaire

L'optimisation linéaire consiste à résoudre les problèmes suivants :

Définition 2.2.1 *Soit m un entier représentant le nombre de contraintes, soit n un entier représentant le nombre de variables. Soit $A \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^m$ et $c \in \mathbb{R}^n$. La solution d'un problème linéaire $P(A, b, c)$ est :*

$$S(A, b, c) = \inf_{x \in E_f(A,b)} \langle c,\, x \rangle \tag{2.4}$$

avec $E_f(A, b) = \{\, x \in \mathbb{R}^n \mid Ax < b \,\}$. Un problème écrit sous cette forme sera qualifié de primal.

On peut omettre les arguments de E_f, P et S s'il n'y a pas d'ambiguïté.

$E_f(A, b)$ est un polytope de dimension n c'est-à-dire l'intersection de m demi-espaces formés par les hyperplans $< A_i, x >= b_i$. La figure 2.2 présente un exemple.

Résoudre un problème linéaire peut se représenter par la recherche du dernier (dans la direction de $-c$) hyperplan de normale c qui intersecte l'ensemble E_f. La figure 2.3 permet de visualiser cette représentation.

L'existence d'une solution est garantie si E_f est d'intérieur non vide et borné et l'on se placera toujours dans ce cas sauf mention du contraire.

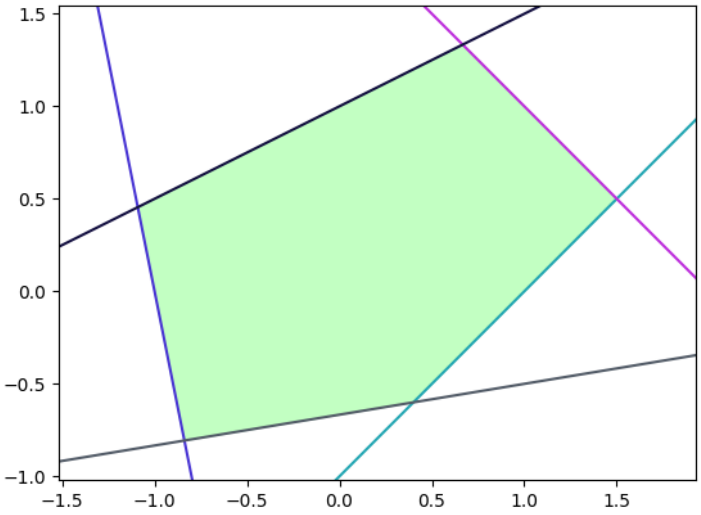
2.2.2 Optimisation Semi-Définie

On peut généraliser cette définition aux matrices semi-définies positives. C'est-à-dire que notre contrainte ne sera plus une inégalité vectorielle, mais demandera qu'une matrice dépendant des variables d'optimisation soit semi-définie positive. Pour commencer, nous allons définir ces notions :

Notation 1 *Soit $A, B \in \mathcal{S}^n$, $\mathcal{S}^n$ représentant l'ensemble des matrices carrées symétriques de taille n. On notera $B \succ A$ respectivement $B \succcurlyeq A$ quand $B - A$ est définie positive respectivement semi-définie positive. C'est à dire lorsque pour tout $x \in \mathbb{R}^n$ non-nul, $x^t \cdot (B - A) \cdot x > 0$ respectivement $x^t \cdot (B - A) \cdot x \geq 0$*

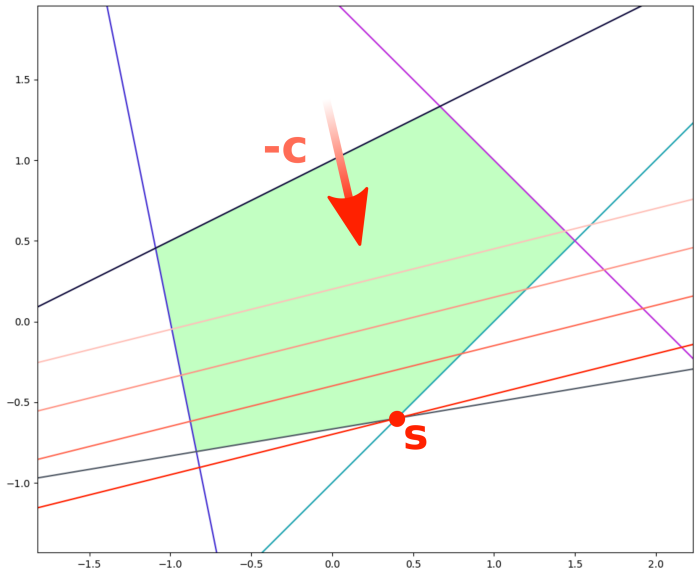
$$A = \begin{pmatrix} 3 & -3 \\ 1 & 1 \\ -5 & -1 \\ -1 & -2 \\ 1 & -6 \end{pmatrix}$$

$$b = \begin{pmatrix} 3 \\ 2 \\ 5 \\ 2 \\ 4 \end{pmatrix}$$



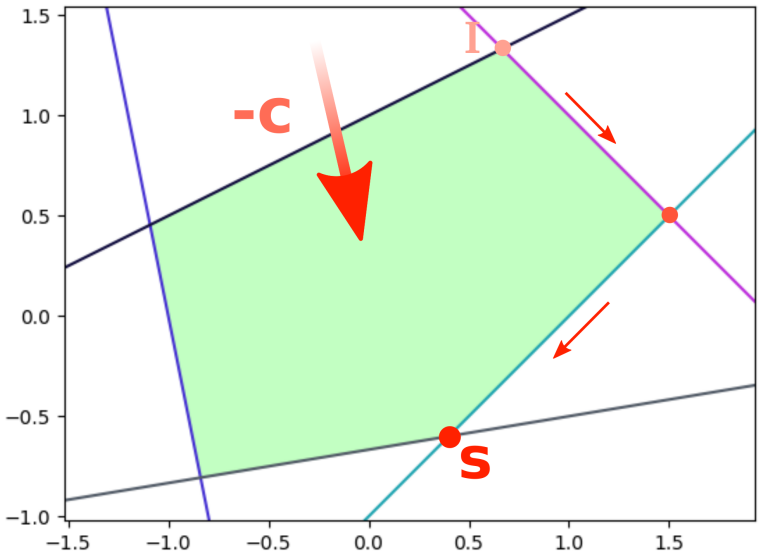
L'ensemble faisable, $E_f(A, b)$, est représenté en vert clair. Chaque droite correspond à une ligne de (A, b) avec, du haut vers le bas, les couleurs suivantes : cyan, violet, bleu, noir, gris.

FIGURE 2.2 – L'ensemble faisable en dimension deux



On reprend l'ensemble faisable de la figure 2.2 et on cherche à optimiser $\langle c, x \rangle$ avec $c = \begin{pmatrix} -1 \\ 4 \end{pmatrix}$. Pour cela on a tracé des courbes de niveaux de la fonction de coût. Ce sont des droites de normale c. On cherche alors la dernière courbe de niveau dans la direction de $-c$ qui intersecte l'ensemble faisable. Les points à l'intersection seront les optimums recherchés. En l'occurence le point S.

FIGURE 2.3 – Recherche manuelle de l'optimum



On reprend l'exemple de la figure 2.3. L'algorithme part du point I pour arriver au point S. La flèche rouge représente $-c$, qui est le vecteur normal aux droites $< c, x >= d$ avec d réel constant, dirigé vers la recherche du minimum.

FIGURE 2.4 – Itérations de l'algorithme du simplexe.

L'optimisation semi-définie positive (SDP) consiste en la résolution des problèmes suivants :

Définition 2.2.2 *Soit n, m des entiers. Soit $(A_0, ...A_{m-1}) \in (\mathcal{S}^n)^m$, $B \in \mathcal{S}^n$ et $c \in \mathbb{R}^n$. La solution d'un problème SDP, $P_{SDP}((A_i), B, C)$, est $S_{SDP}((A_i), b, C) \in \mathbb{R}$:*

$$S((A_i), b, C) = \inf_{x \in E_{SDP}((A_i), B)} \langle c, x \rangle \tag{2.5}$$

avec $E_{SDP}((A_i), B) = \left\{ x \in \mathbb{R}^n \;\middle|\; B \succcurlyeq \sum_{i=0}^{m} x_i A_i \right\}$

2.2.3 Algorithme du simplexe

Le premier algorithme historiquement introduit pour résoudre un problème linéaire est celui du simplexe : DANTZIG, 1990. Il part de la constatation suivante : l'optimum recherché est un sommet du polytope $E_f(A, b)$. L'algorithme consiste à partir d'un sommet quelconque du polytope et à se déplacer le long des arêtes de manière à toujours réduire le coût. Cela permet de converger vers l'optimum. La figure 2.4 représente une exécution de cet algorithme.

Cet algorithme requiert cependant de connaître un sommet du polytope. Pour trouver un sommet à un programme linéaire P, on peut construire un problème modifié P' dont on connaîtra un sommet du polytope et dont la solution sera un sommet de P.

Ainsi l'algorithme se décompose en deux phases, la première trouve un sommet et la seconde l'optimum. On peut aussi écrire des algorithmes combinant les deux phases en une seule.

L'algorithme du simplexe bien que de complexité exponentielle se révèle très efficace en pratique. Cependant il repose profondément sur la structure de polytope de l'ensemble faisable. Ainsi il se limite à la résolution de problèmes linéaires.

Autres méthodes : Un autre moyen pour résoudre un problème linéaire est la méthode des ellipsoïdes, qui possède un pire temps théorique polynomial, mais qui est bien moins efficace en pratique. Les méthodes alternatives utilisées le plus souvent en pratique sont celles dites de points intérieurs. Elles possèdent un pire coût polynomial et sont aussi efficaces que le simplexe en pratique. C'est un algorithme issu de cette dernière méthode qui sera utilisé dans ce travail. Avant de présenter de manière générale les méthodes de points intérieurs, il nous faut effectuer quelques rappels sur la méthode de Newton.

2.2.4 Méthode de Newton

Pour résoudre des problèmes d'optimisation, on peut partir d'une méthode classique de recherche d'optimum : la méthode de Newton. Cette méthode permet avant de trouver le zéro d'une fonction mais elle peut aussi être utilisé pour chercher les extrema d'une fonction, c'est-à-dire les points de valeurs minimales ou maximales d'une fonction. Pour cela on utilise le théorème suivant qui fait le lien entre les zéros de la dérivées d'une fonction et ses extrema :

Théorème 2.2.1 (Condition nécessaire pour un extremum local) *Soit $f : E \to F$ et x_0 un point appartenant à l'intérieur de E où f est différentiable. Si x_0 est un extremum de f alors toutes les dérivées partielles de f sont nulles en ce point.*

Un point où toutes les dérivées partielles de f s'annulent est appelé point critique, ce n'est pas forcément un optimum : cette condition est donc bien seulement nécessaire. Il faut donc avoir des informations tierces pour s'assurer que l'on a bien l'optimum recherché. Cela permet de remplacer le problème de trouver un extremum par la recherche du point d'annulation d'une fonction ce qui peut s'effectuer grâce à l'algorithme de Newton.

Définition 2.2.3 (Suite de Newton) *Soit U un ouvert de $\mathbb{R}^n$ et $f : U \to \mathbb{R}^m$ une fonction de classe $\mathcal{C}^2$. Si $Df(x)$, la différentielle de f au point x, est inversible, on définit le pas de Newton $N_f(x) = x - Df(x)^{-1}f(x)$. On appelle suite de Newton la suite (x_n) définie par $x_0 \in U$ et $x_{n+1} = N_f(x_n)$.*

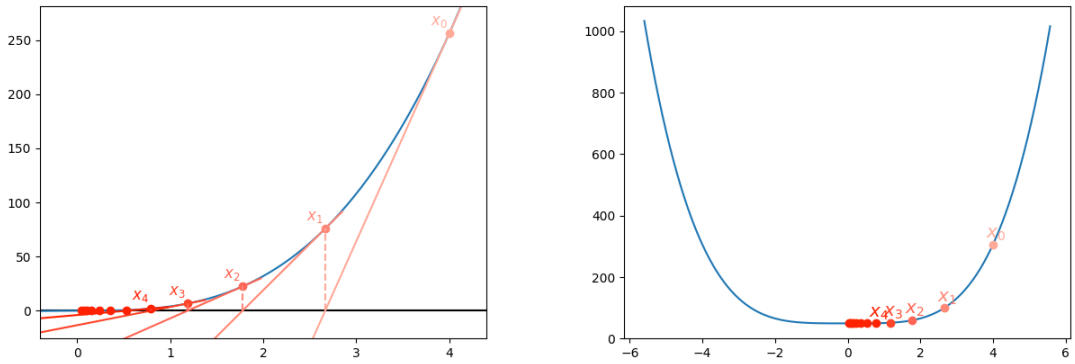
Nous présentons seulement une des formalisations de la méthode de Newton. Cependant, l'histoire de cette méthode est riche et nous renvoyons le lecteur vers Ypma, 1995 pour la découvrir.

Nous avons définit une suite de point, N_f, le théorème suivant va montrer que, sous certaines hypothèses, elle va converger vers un point d'annulation de f.

Théorème 2.2.2 (Convergence de N_f) *Soit U un ouvert de $\mathbb{R}^n$, $f : U \to \mathbb{R}^m$ une fonction de classe $\mathcal{C}^2$. Soit $x^* \in U$ tel que $f(x^*) = 0$, supposons $Df(x^*)$ inversible alors il existe $r > 0$ tel que pour tout $x_0 \in B(x^*, r)$ la suite $x_{k+1} = N_f(x_k)$ est définie et converge vers x^*.*

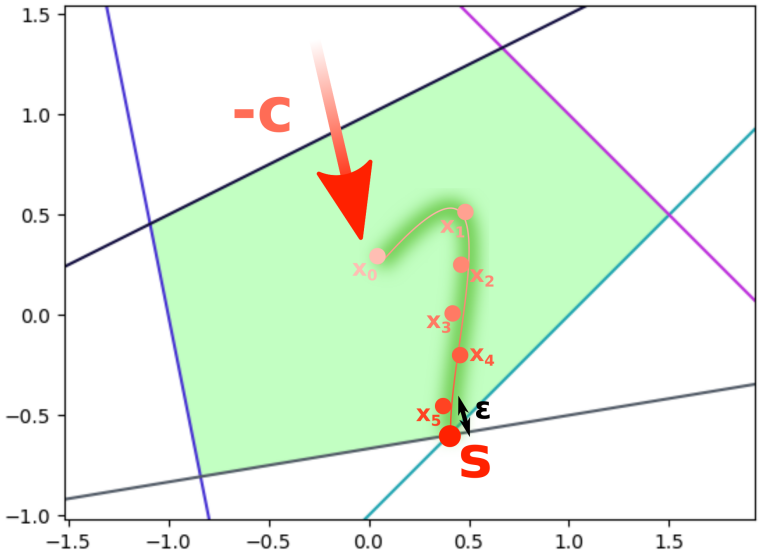
De plus $\|x_k - x^*\| \leq \left(\frac{1}{2}\right)^{2^k-1} \|x_0 - x^*\|$

Ainsi, en calculant les itérés successives de cette suite on obtient une méthode convergeant vers le point d'annulation d'une fonction. En l'appliquant sur la dérivée d'une fonction f, on peut alors, en utilisant le théorème 2.2.1, converger vers un point critique de f. Cette procédure est représentée dans la figure 2.5. Il faut ensuite d'autres informations pour montrer que ce point est l'extremum recherché. En optimisation convexe, on ne pourra pas utiliser la méthode de Newton directement car celui-ci ne fonctionne que pour un point à l'intérieur du domaine. Or, dans les cas que nous abordons dans ce manuscrit, les extrema recherché sont sur la frontière. On va donc voir dans la sous-section suivante comment on peut ramener un problème d'optimisation à un problème de recherche d'extrema.



On cherche un minimum de la fonction de droite. Pour cela, on peut utiliser l'algorithme de Newton sur la dérivée de cette fonction représenté à gauche. Les x_i correspondent aux itérés de Newton sur la dérivé. On a représenté les tangentes à la courbe qui permettent le calcul, par intersection avec l'axe des abscisses, des itérés suivantes. Les tangentes sont calculé grace à la dérivée seconde de la fonction d'origine.

FIGURE 2.5 – L'algorithme de Newton



Lex x_i représente un ensemble de point proche du chemin central et convergeant vers l'optimum s.

FIGURE 2.6 – Le chemin central

2.2.5 Points intérieurs

Les algorithmes de points intérieurs sont des algorithmes permettant la résolution de problèmes d'optimisation convexe. Ils sont en particulier utilisés pour résoudre des problèmes convexes. Ils portent ce nom, car ils produisent une suite de points tous inclus dans l'intérieur de l'ensemble faisable. La figure 2.6 illustre ce procédé.

Cette méthode est en opposition avec le simplexe qui lui aussi possède une suite d'itérées, mais qui est à la frontière de l'ensemble faisable et non dans son intérieur. On a vu que la solution d'un problème linéaire se situait sur sa frontière. On peut même généraliser cette propriété à tout ensemble faisable convexe de fonctions de coût convexes (voir Rockafellar, 2015, Chapitre 32). Ainsi, dans le cadre des points intérieurs et contrairement au simplexe, la suite d'itérés n'arrivera jamais sur l'optimum. En revanche, on pourra garantir que la suite des itérés converge vers l'optimum et le contrôle de cette convergence pourra donner une ϵ-optimalité.

Définition 2.2.4 (ϵ-optimalité) *Soit p un problème d'optimisation définit par $f^* = \inf_{x \in E} f(x)$.*

On dit qu'un point $x \in \overline{E}$ est une solution ϵ-optimale de p si $f(x) - f^* \leq \epsilon$ [1].

En informatique, ce n'est pas un problème d'avoir un algorithme seulement ϵ-optimal, car on représente le plus souvent les réels par des nombres flottants les approximant. Ainsi, que l'algorithme génère une solution ϵ-optimale ou pas, le programme ne pourra que manipuler des solutions ϵ-optimales. De plus, il existe certaines méthodes, comme dans Mehrotra, 1991, qui permettent de trouver une solution exacte à partir d'une solution ϵ-optimale pour un problème linéaire.

L'ensemble des points produits par l'algorithme formera ce qu'on appellera chemin central. Ce chemin aura pour extrémité l'optimum recherché d'un côté et de l'autre un point que l'on nommera centre analytique.

Nous aborderons dans ce travail des méthodes de points intérieurs basés sur l'algorithme de Newton. On a vu que la méthode de Newton permettait de trouver l'optimum d'une fonction. Cependant, cet optimum doit se situer dans l'intérieur du domaine de la fonction. Or dans le cas de l'optimisation convexe, l'optimum se situe sur la frontière du domaine. Il est donc impossible d'utiliser directement l'algorithme de Newton. C'est pourquoi nous allons modifier la fonction de coût afin que l'optimum se situe dans l'intérieur du domaine en lui ajoutant une fonction barrière.

Fonction barrière

Pour forcer les itérés de Newton à rester dans l'ensemble faisable on rajoute une fonction de pénalisation à la fonction de coût. Cette fonction pénalise les points qui se rapprochent de la frontière de l'ensemble faisable. En d'autres termes, c'est une fonction définie sur l'ensemble faisable et qui tend vers l'infini quand on se rapproche de la frontière. Une telle fonction est appelée une fonction barrière. Un exemple de fonction barrière peut être trouvé dans la figure 2.7. Cette notion sera définie dans la sous-section 4.2.2.

Algorithme primal

Un algorithme primal est l'application de l'algorithme de Newton sur une combinaison linéaire de la fonction de coût et d'une fonction barrière dont le coefficient diminue au fur et à mesure des itérations. Les difficultés de l'algorithme résident dans la vitesse à laquelle on fait décroître l'impact de la barrière et sa preuve de correction. Cet algorithme sera décrit précisément dans la partie 4.2.

1. $f(x^*)$ est bien indépendant du x^* choisi

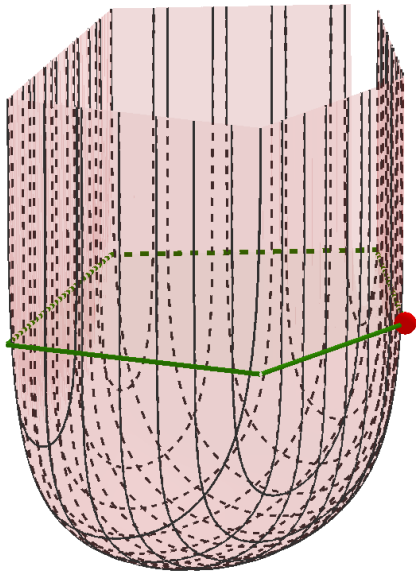


FIGURE 2.7 – Fonction barrière en dimension deux

Algorithme primal dual

Une autre méthode de points intérieurs repose sur le calcul de l'optimum de deux problèmes au lieu d'un unique problème. Le premier problème est le problème original défini dans 2.2.1, on l'appelle problème primal.

Le second problème se construit à partir du problème primal, on l'appelle le problème dual[2].

Définition 2.2.5 (Problème dual) *Soit $P(A,b,c)$ un problème linéaire, $D(A,b,c)$ est le problème dual de $P(A,b,c)$. Il est défini par la recherche de la solution $SD(A,b,c) \in \mathbb{R}$:*

$$SD(A,b,c) = \sup_{y \in ED_f(A,c)} \langle b,\, y \rangle \tag{2.6}$$

$$\text{avec } ED_f(A,c) = \left\{ y \in \mathbb{R}^m \;\middle|\; y \geq 0 \wedge A^T y = c \right\}$$

L'introduction d'un problème dual présente un intérêt à travers les propriétés de la fonction $dg(x,y) = \langle b,\, y \rangle - \langle c,\, x \rangle$ avec $x \in E_f(A,b)$ et $y \in ED_f(A,c)$. On appelle cette fonction le saut de dualité, car elle représente la différence entre le coût du problème primal et le coût du problème dual. Cette fonction est positive et elle s'annule aux points (x,y) où x est un point optimum de $P(A,b,c)$ et y de $D(A,b,c)$. Ainsi, pour trouver une solution à $P(A,b,c)$ on peut chercher un point d'annulation de cette fonction. Cela entraine l'utilisation de l'algorithme de Newton où on devra faire attention à toujours respecter les contraintes d'inégalité. On devra aussi connaitre un point initial dans l'ensemble faisable. Pour cela, on peut utiliser une méthode dite phase 1.

2. La dénomination primale/duale peut être inversée vu que chacun est le dual de l'autre. Le choix pris dans ce document permet d'être cohérent avec l'algorithme primal défini dans le chapitre 4.

Phase 1

Comme dans le cadre du simplexe qui a besoin d'un sommet initial, on a besoin dans le cadre des méthodes de points intérieurs d'un élément à l'intérieur de l'ensemble faisable. Nous allons présenter ici le moyen d'obtenir un tel point dans le cadre d'un algorithme primal. Des variantes reposant sur le même principe existent pour le simplexe et les méthodes primales/duales. Dans tous les cas, cela consiste à construire un problème possédant un point initial trivial et dont la solution est un point initial pour le problème d'origine.

Dans le cas d'un problème de points intérieurs, le point initial est un point dans l'intérieur du domaine : un point faisable. La définition suivante présente un exemple de problème phase 1 permettant de trouver un point faisable.

Définition 2.2.6 (Problème Phase 1) *Soit $P(A, b, c)$ un problème linéaire. Le problème $P_1(A, b, c) = P\left(\begin{pmatrix} A & \mathbf{Id}_m \end{pmatrix}, b, \begin{pmatrix} 0 \\ \mathbf{1}_m \end{pmatrix}\right)$ est appelé problème phase 1.*

On peut aisément calculer un point initial à un problème phase 1 :

Propriété 2.2.1 (Point initial phase 1) *Le point $\begin{pmatrix} \mathbf{1}_n \\ b - A\mathbf{1}_n - \mathbf{1}_m \end{pmatrix}$ appartient à $E_f(P_1(A, b, c))$*

De plus, les solutions de P_1 sont des points faisables de P :

Théorème 2.2.3 (Phase 1) *$\exists (x, s) \in \mathbb{R}^n \times \mathbb{R}^m$ solution ϵ-optimale de P_1 et $s < \mathbf{0}_m$ si et seulement si p admet un point faisable. De plus si (x, s) est une solution ϵ-optimale de P_1 alors x est un point faisable de P.*

2.2.6 Bilan

On a donc vu qu'il existait deux grandes familles de méthodes pour résoudre les problèmes linéaires : le simplexe et les méthodes de points intérieurs. En pratique, les deux sont utilisées. On a aussi vu que les problèmes linéaires se généralisaient aux problèmes SDP. Ces problèmes ne sont pas solubles par des algorithmes du simplexe. En revanche, les méthodes de points intérieurs s'y adaptent très bien.

Il existe une troisième méthode, appelée méthode des ellipsoïdes. Elle permet de résoudre les problèmes linéaires et SDP. Elle n'est pas utilisée en pratique car nécessitant un très grand nombre d'itérations. En quelques mots, cette méthode consiste à construire une suite d'ellipsoïdes contenant le point optimal. Le volume de ces ellipsoïdes décroît à chaque itération. Ainsi, la suite des centres des ellipsoïdes converge vers l'optimum.

On a vu, dans la section précédente, que la résolution en ligne de problème d'optimisation convexe permettait la mise en place de contrôleurs performants. Si l'on souhaite utiliser de tels contrôleurs dans des systèmes critiques, il faudra certifier ces programmes. Il ne suffira pas alors de donner une preuve de la correction de l'algorithme. Il faudra prouver la correction de leurs implémentations bas niveau, par exemple en C. C'est pourquoi, nous allons présenter dans la section suivante des méthodes permettant de prouver formellement des codes sources écrits dans des langages bas niveau.

2.3 Vérification

On se place dans un cadre abstrait de programme, on suppose donné un ensemble de mémoire $\mathbb{M}$, l'ensemble des programmes $\mathbb{P}$ est l'ensemble des fonctions transformant une mémoire en une autre : $\mathbb{P} = \mathbb{M} \to \mathbb{M}$.

2.3.1 Triplet de Hoare

Définition

Les triplets de Hoare sont un moyen de spécifier des propriétés qu'un programme doit vérifier. Ils sont composés d'une précondition, d'un programme et d'une postcondition. La précondition représentant une hypothèse sur la mémoire du programme avant son exécution. La postcondition est la propriété que doit vérifier la mémoire du programme après son exécution.

Définition 2.3.1 (Triplet de Hoare) *Soit* `C` $: \mathbb{M} \to \mathbb{M}$ *un programme que l'on souhaite spécifier. Soit* P *et* Q*, deux prédicats sur l'ensemble* $\mathbb{M}$*.* $\{P\}$ `C` $\{Q\}$ *est appelé un triplet de Hoare. Il est valide quand*

$$\forall m \in \mathbb{M}, P(m) \Rightarrow Q(C(m)). \tag{2.7}$$

P *est appelée une précondition et* Q *une postcondition.*

C peut représenter n'importe quel objet transformant une mémoire en une autre. Cela peut être, par exemple une fonction écrite dans un langage donné, une instruction de ce même langage, un script ou un programme compilé. Lorsque C représente une fonction f, le couple (P, Q) sera appelé le contrat de la fonction f.

Les triplets de Hoare peuvent être vus comme une sémantique axiomatique, c'est-à-dire qu'ils permettent de donner le sens de `C` . Plus précisément, ils considèrent l'exécution de `C` comme la transformation de sa mémoire en une autre et permettent ainsi de spécifier des propriétés sur cette transformation.

Exemple

Afin d'illustrer la définition d'un triplet de Hoare, nous allons spécifier une fonction *fibonnaci* permettant de construire un tableau de N éléments contenant la suite de Fibonacci. Une telle fonction est donnée dans la figure 2.8.

Pour exprimer les préconditions et les postconditions on utilise la logique du premier ordre munit d'un symbole de fonction $fib : \mathbb{N} \to \mathbb{N}$, définit axiomatiquement par $fib(0) = 0$, $fib(1) = 1$ et $\forall k > 1, fib(k) = fib(k-1) + fib(k-2)$. On peut alors écrire le triplet de Hoare suivant $\{n >= 2\}$ `ar = fibonnaci(n)` $\{\forall k \in [\![0, n[\![, ar[k] = fib(k)\}$.

Cet exemple sera réutilisé régulièrement dans la suite de ce document comme illustration à certains concepts abordés.

fibonnaci

$ar := [0, 1]$
$i := 0$
while $i < n - 2$ **do**
  $ar[i + 2] := ar[i + 1] + ar[i]$
  $i := i + 1$

FIGURE 2.8 – Algorithme calculant la suite de Fibonacci

```
1  unsigned int n;
2  unsigned int ar[100];
3
4  /*@ requires 2 <= n < 100;
5      ensures \forall integer i; (0 <= i < n) ==> ar[i] == fib(i);
6      assigns ar[0 .. n-1];
7      */
8  void fibonacci() {
9    unsigned int i = 0;
10   ar[0] = 0;
11   ar[1] = 1;
12   while (i < n - 2) {
13     ar[i+2] = ar[i+1] + ar[i];
14     i += 1;
15   }
16 }
```

Les tableaux étant de taille fixe en C nous avons dû enrichir notre précondition.

FIGURE 2.9 – Fonction annotée en ACSL calculant la suite de Fibonacci

ACSL : ANSI/ISO C Specification Language

Le langage ACSL est un langage permettant d'annoter un code C. Ces annotations se placent dans les commentaires du code C et donc se contentent de l'étendre. Plus précisément, tout texte au sein d'un commentaire qui est précédé d'une arobase sera considéré comme de l'ACSL. La particularité de ces annotations est, qu'au contraire de commentaires classiques, leur structure est définie par une grammaire et peut donc être interprétée par un programme, en l'occurrence Frama-C. Ces annotations peuvent servir à spécifier n'importe quelle information sur le programme, en particulier sa sémantique. Le choix retenu pour exprimer la sémantique en ACSL est le triplet de Hoare. Ainsi ACSL permet d'exprimer des triplets de Hoare sur un code C.

Pour chaque bloc de code, en particulier les fonctions, on peut définir une précondition à l'aide du mot-clef `requires` et une postcondition avec `ensures`. La figure 2.9 reprend l'exemple de la suite de Fibonacci en C annoté en ACSL. On peut y trouver une annotation supplémentaire par rapport à un triplet de Hoare classique : `assigns`. Cela permet de spécifier quelles parties de la mémoire sont modifiées par la fonction, et donc de déduire que toutes les autres parties sont invariantes ce qui sera utile lors des appels de fonction.

La logique du premier ordre est utilisée pour les préconditions et les postconditions en ACSL. Les variables quantifiées peuvent être de n'importe quel type existant en C. ACSL supporte, en plus, certains types comme le type `integer` pour les entiers naturels et le type `real` pour les réels. Les opérations classiques,comme les additions ou les racines carrées, sont aussi supportées pour tous ces types. Une des particularités de ACSL est de pouvoir enrichir le langage de base avec de nouveaux types. On peut ensuite déclarer des opérateurs sur ces nouveaux types qui peuvent être définis soit par leur expression directe, soit par des axiomes. Toutes ces définitions sont faites au sein d'un objet appelé axiomatique. On peut trouver l'exemple de la définition de l'opérateur `fib` représentant la suite de Fibonacci dans la figure 2.10. La deuxième ligne introduit une nouvelle axiomatique nommée `Fibonacci`. Ensuite à la quatrième ligne on déclare un nouvel opérateur `fib` prenant un entier en argument et retournant un entier. Les deux lignes suivantes introduisent deux axiomes sur ce nouvel opérateur spécifiant les deux premiers éléments de la liste. Enfin, à la septième ligne,

```
/*@
axiomatic Fibonacci
{
    logic integer fib(integer x);
    axiom fib_0: fib(0) == 0;
    axiom fib_1: fib(1) == 1;
    axiom fib_n:
        \forall integer n;
          (n >= 2) ==>
            fib(n) == fib(n-1) + fib(n-2);
}
*/
```

FIGURE 2.10 – Définition de la suite de Fibonacci en ACSL

on introduit un troisième axiome donnant la relation de récurrence définissant la suite de Fibonacci. On pourra ensuite utiliser l'opérateur `fib` dans n'importe quelle expression ACSL comme effectué dans la figure 2.9.

Comment valider des triplets de Hoare ?

Les triplets de Hoare permettent d'énoncer les propriétés attendues d'un programme. Pour prouver qu'un programme est correct vis-à-vis de ces propriétés, il faut montrer que tous les triplets de Hoare définis sont valides. Pour cela, si on se donne un langage, on va pouvoir définir pour chaque construction du langage une règle d'inférence. Par exemple pour une affectation, la règle d'inférence peut s'écrire :

$$\text{Affectation}\ \frac{Q[x/e] = P}{\{P\}\ \texttt{x := E}\ \{Q\}}$$

La règle d'inférence pour la séquence peut quant à elle s'écrire :

$$\text{Sequence}\ \frac{\{P\}\ \texttt{A}\ \{R\} \qquad \{R\}\ \texttt{B}\ \{Q\}}{\{P\}\ \texttt{A;B}\ \{Q\}}$$

Autant l'application sur le code de la règle d'inférence pour une affectation peut être automatisée, autant la règle pour la séquence nécessite d'exhiber un R pour être prouvé. Ainsi, pour prouver un programme, il faut exhiber un R pour chaque ligne de code. Ce travail serait fastidieux et peu robuste, car tout changement d'une ligne de code nécessiterait de changer les prédicats l'encadrant. On peut remplacer ce travail par le calcul de plus faible précondition introduit par Dijkstra dans DIJKSTRA, 1975.

2.3.2 Calcul de plus faible précondition de Dijkstra

Le calcul de plus faible précondition repose sur la règle d'inférence suivante :

$$\frac{\{P'\}\ \texttt{C}\ \{Q\} \qquad P \Rightarrow P'}{\{P\}\ \texttt{C}\ \{Q\}}$$

Cette règle montre que pour prouver un triplet, on peut prendre une autre précondition et si on montre que le triplet est valide avec celle-ci et que cette nouvelle précondition est plus faible que la première alors le triplet original est valide.

Le calcul de plus faible précondition va, à partir d'une postcondition et d'un programme, calculer la plus faible précondition formant un triplet valide. Ainsi, il ne permet pas d'obtenir directement la validité d'un triplet, mais construit une précondition formant un triplet valide. Ainsi, pour prouver un triplet, il ne reste plus qu'à appliquer la règle précédente et prouver que la précondition implique la plus faible précondition qui a été calculée. Comme, parmi toutes les préconditions formant un triplet valide, on va calculer la plus faible. Si le triplet que l'on souhaite prouver est valide alors nécessaire l'implication sera vraie.

Définition 2.3.2 (Plus faible précondition) *La plus faible précondition d'un programme* `C` *et d'une postcondition* Q *est le prédicat* $\mathbf{WP}(\texttt{C}, Q)$ *tel que :*

1. $\{\mathbf{WP}(\texttt{C}, Q)\}\ \texttt{C}\ \{Q\}$ *est valide*
2. *Quelque soit* P, $\{P\}\ \texttt{C}\ \{Q\}$ *valide entraine nécessairement* $P \Rightarrow \mathbf{WP}(\texttt{C}, Q)$

On peut alors réduire le problème de validité d'un triplet de Hoare à un problème de satisfaction de formule logique grâce au théorème 2.3.1.

Théorème 2.3.1 (Prover un triplet de Hoare) $$\frac{\mathbf{WP}(\texttt{C}, Q) = R \qquad P \Rightarrow R}{\{P\}\ \texttt{C}\ \{Q\}}$$

Ainsi on peut réduire la validité d'un triplet de Hoare à la preuve mathématique d'une implication et à un calcul de plus faible précondition pour lequel on va présenter une méthode automatique de calcul dans la section suivante.

Calculer la plus faible précondition

On peut donner pour chaque instruction du langage une manière mécanique de calculer la plus faible précondition. La figure 2.11 présente ces règles pour un langage pseudo-code afin de se donner une idée. Parmi toutes les règles de calcul de plus faible précondition, les règles associées aux structures de boucle empêchent une automatisation complète. En effet, il est nécessaire de leur adjoindre un prédicat I, appelé invariant de boucle. Un invariant de boucle est un prédicat qui doit être vrai avant et après chaque itération de la boucle.

Définition 2.3.3 (Invariant de boucle) *Soit* I *et* E *des prédicats sur* $\mathbb{M}$ *et* $C : \mathbb{M} \to \mathbb{M}$. *Soit* $p \in \mathbb{P}$ *un programme contenant une boucle* `while (E) C` .

Soit $M \subset \mathbb{M}$ *l'ensemble des mémoires de* p *possible avant l'exécution de la boucle.*

I *est un invariant de la boucle* `while (E) C` *si les deux propriétés suivantes sont vérifiées :*

$$\forall m \in M, I(m) \tag{2.8}$$

$$\forall n \in \mathbb{N}, m \in M, \Big(\forall i \in [\![0, n[\![, E(C^i(m))\Big) \Rightarrow I(C^n(m)) \tag{2.9}$$

Pour prouver qu'un prédicat est un invariant de boucle, on peut faire une récurrence, cela se traduit en logique de Hoare par la règle suivante :

$$\text{Conclusion}\ \frac{P \Rightarrow I \qquad \{I\}\ \texttt{C}\ \{I\} \qquad I \Rightarrow Q}{\{P\}\ \texttt{while (E) inv}\ I\ \texttt{C}\ \{Q\}}$$

Il est souvent difficile de trouver automatiquement un invariant. Même si certaines solutions existent comme dans Bensalem, Lakhnech et Saidi, 1996, elles ne sont jamais systématiques. C'est pourquoi, pour le calcul de plus faible précondition, on impose pour chaque boucle de spécifier un invariant de boucle.

$$\frac{}{\mathbf{WP}(\texttt{ x := E }, Q) = \forall y, y = \texttt{ E } \Rightarrow Q[x/y]}$$

(a) Affectation

$$\frac{\mathbf{WP}(\texttt{ S2 }, Q) = O \qquad \mathbf{WP}(\texttt{ S1 }, O) = R}{\mathbf{WP}(\texttt{ S1;S2 }, Q) = R}$$

(b) Séquence

$$\frac{\mathbf{WP}(\texttt{ S1 }, Q) = P_1 \qquad \mathbf{WP}(\texttt{ S2 }, Q) = P_2}{\mathbf{WP}(\texttt{ if (E) S1 else S2 }, Q) = \texttt{ E } \Rightarrow P_1 \wedge \neg \texttt{ E } \Rightarrow P_2}$$

(c) Conditionnelle

$$\frac{R \Rightarrow Q \qquad \{P\}\ \texttt{C}\ \{R\}}{\mathbf{WP}(\texttt{ f() }, Q) = P} \text{ avec } C \text{ le corps de } f$$

(d) Appel de fonction

$$\frac{\mathbf{WP}(\texttt{ E }, I) = P \qquad (\neg \texttt{ F } \wedge I) \Rightarrow Q \qquad \{\texttt{ F } \wedge I\}\ \texttt{C;G}\ \{I\}}{\mathbf{WP}(\texttt{ for (E;F;G) inv } I \texttt{ C }, Q) = P}$$

(e) Boucle For

$$\frac{(\neg \texttt{ E } \wedge I) \Rightarrow Q \qquad \{\texttt{ E } \wedge I\}\ \texttt{C}\ \{I\}}{\mathbf{WP}(\texttt{ while (E) inv } I \texttt{ C }, Q) = I}$$

(f) Boucle While

FIGURE 2.11 – Règles de calcul de plus faible précondition

On impose aussi de donner un contrat pour chaque fonction. Ceci pourrait être évité en remplaçant chaque appel de fonction par le corps de la fonction appelé. Cependant, les fonctions représentant généralement une unité sémantique naturelle il semble plus naturel de les prouver une fois pour toutes.

De plus, comme nous le verrons dans cette thèse, la preuve de l'implication finale $P \Rightarrow R$ peut se révéler ardue. Ainsi, séparer le calcul en plusieurs sous-problèmes plutôt qu'un unique problème monolithique se révèle souvent la bonne solution. Cela s'applique pour les appels de fonctions, mais aussi pour les boucles. En effet, une solution alternative à la donnée d'un invariant de boucle pour le calcul de plus faible précondition serait de demander une borne sur le nombre d'itérations de la boucle [3]. On pourrait ainsi déplier les boucles, mais le calcul de plus faible précondition générerait une formule logique de taille au moins proportionnelle au nombre d'itérations de la boucle.

3. On serait contraint de spécifier manuellement cette borne, car, à cause de l'indécidabilité du problème de l'arrêt des machines de Turing, il serait impossible de calculer automatiquement cette borne.

Exemple

Dans cette sous-section nous allons illustrer le fonctionnement du calcul de plus faible précondition en l'appliquant à la fonction Fibonacci de la figure 2.8. On a déjà écrit le contrat de la fonction à la sous-section précédente. Comme il y a une boucle dans le code de la fonction, il faut lui donner un invariant. Nous savons qu'à chaque itération de la boucle tous les éléments du tableau à une position inférieure à l'indice courant contiennent la suite de Fibonacci. Ainsi, on peut prendre l'invariant :

$$I = \forall k \in [\![0, i+2[\![, ar[k] = fib(k) \tag{2.10}$$

Cet invariant sera vrai au début de la boucle, car le tableau a été rempli correctement pour ces deux premiers éléments. À chaque itération, on rajoute un nouvel élément de la suite au tableau, ce qui permet de garantir la préservation de I.

On rappelle que la précondition et la postcondition, que l'on nommera respectivement P et Q étaient :

$$P = n \geq 2 \tag{2.11}$$
$$Q = \forall k \in [\![0, n[\![, ar[k] = fib(k) \tag{2.12}$$

On peut désormais effectuer le calcul de plus faible précondition sur notre algorithme. On va utiliser les notations suivantes :

$$\texttt{C1} = \texttt{ar := [0, 1]} \tag{2.13}$$
$$\texttt{C2} = \texttt{i := 0} \tag{2.14}$$
$$\texttt{C3} = \texttt{ar[i+2] := ar[i+1] + ar[i]} \tag{2.15}$$
$$\texttt{C4} = \texttt{i := i + 1} \tag{2.16}$$
$$E = i < n - 2 \tag{2.17}$$
$$\texttt{C} = \texttt{C1 ; C2 ; while (E) inv } I \texttt{ C3; C4} \tag{2.18}$$

C représente le programme défini dans la figure 2.8. On souhaite prouver $\{P\}$ C $\{Q\}$. Pour cela on utilise le théorème 2.3.1 afin de se ramener à un calcul de plus faible précondition.

$$\frac{\mathbf{WP}(\texttt{ C }, Q) = R \qquad P \Rightarrow R}{\{P\}\ \texttt{C}\ \{Q\}}$$

On remet à plus tard le calcul de $G_0 = P \Rightarrow R$. Il faut ensuite effectuer un calcul de plus faible précondition. La première étape du calcul consiste à appliquer la règle de séquence 2.11b :

$$\frac{\mathbf{WP}(\texttt{ while (E) inv } I \texttt{ C3; C4; }, Q) = O \qquad \mathbf{WP}(\texttt{ C1; C2; }, O) = R}{\mathbf{WP}(\texttt{ C }, Q) = \mathrm{R}}$$

Il faut ensuite calculer $\mathbf{WP}(\texttt{ while (E) inv } I \texttt{ C3; C4; }, Q)$ avec la règle de la boucle 2.11f.

$$\frac{(\neg\ \texttt{E}\ \wedge I) \Rightarrow Q \qquad \{\ \texttt{E}\ \wedge I\}\ \texttt{C3; C4;}\ \{I\}}{\mathbf{WP}(\texttt{ while (E) inv } I \texttt{ C3; C4; }, Q) = I}$$

Ainsi $O = I$ [4] et on obtient alors deux buts à prouver, le premier $G_1 = (\neg\ \texttt{E}\ \wedge I) \Rightarrow Q$ est purement logique, nous le traiterons plus tard en même temps que tous les buts logiques. Le second est un triplet de Hoare que l'on peut réduire grâce au théorème 2.3.1 :

4. voir l'équation (2.10) pour sa définition

$$\frac{\mathbf{WP}(\texttt{ C3; C4; }, I) = T \qquad (\texttt{ E } \wedge I) \Rightarrow T}{\{\texttt{ E } \wedge I\}\ \texttt{C3; C4;}\ \{I\}}$$

On remet à plus tard la preuve du nouveau but logique $G_2 = (\texttt{ E } \wedge I) \Rightarrow S$. On doit, par contre, calculer $S = \mathbf{WP}(\texttt{ C3; C4; }, I)$. Pour cela, nous utilisons la règle de séquence :

$$\frac{\mathbf{WP}(\texttt{ C4 }, I) = V \qquad \mathbf{WP}(\texttt{ C3 }, V) = S}{\mathbf{WP}(\texttt{ C3; C4; }, I) = S}$$

Puis nous appliquons la règle d'affectation pour $\mathbf{WP}(\texttt{ C4 }, I)$ ce qui donne :

$$\overline{\mathbf{WP}(\texttt{ i := i + 1 }, I) = \forall y, (y = i + 1) \Rightarrow \forall k \in [0, y + 2[, ar[k] = fib(k)}$$

Ce qui donne après simplification :

$$V = \forall k \in [\![0, y + 3[\![, ar[k] = fib(k) \tag{2.19}$$

On peut à nouveau appliquer l'affectation à la deuxième partie de la séquence :

$$\frac{S = \forall y, (y = ar[i + 1] + ar[i]) \Rightarrow \forall k \in [\![0, i + 3[\![, ar[i + 2 \rightarrow y][k] = fib(k)}{\mathbf{WP}(\texttt{ ar[i+2] := ar[i+1] + ar[i] }, V) = S}$$

L'affectation se fait ici dans un tableau ce qui la rend un peu plus compliquée. C'est pourquoi, on utilise la notation $x[k \rightarrow x]$ qui est égale au tableau x dont on a remplacé la k-ième valeur par x. Ainsi on a :

$$S = \forall k \in [\![0, i + 3[\![, ar[i + 2 \rightarrow ar[i + 1] + ar[i]][k] = fib(k) \tag{2.20}$$

Ceci termine cette branche du calcul. On peut donc revenir à l'autre branche de la première séquence : $\mathbf{WP}(\texttt{ C1; C2; }, O) = R$ à laquelle on applique à nouveau la règle de séquence :

$$\frac{\mathbf{WP}(\texttt{ C2 }, O) = U \qquad \mathbf{WP}(\texttt{ C1 }, U) = R}{\mathbf{WP}(\texttt{ C1; C2; }, O) = R}$$

Chacune des branches se prouve par la règle d'affectation :

$$\overline{\mathbf{WP}(\texttt{ i := 0 }, I) = \forall y, y = \texttt{ 0 } \Rightarrow \forall k \in [\![0, y + 2[\![, ar[k] = fib(k)}$$

Ainsi après simplification :

$$U = \forall k \in [0, 2[, ar[k] = fib(k) \tag{2.21}$$

et enfin la dernière affectation se calcule ainsi :

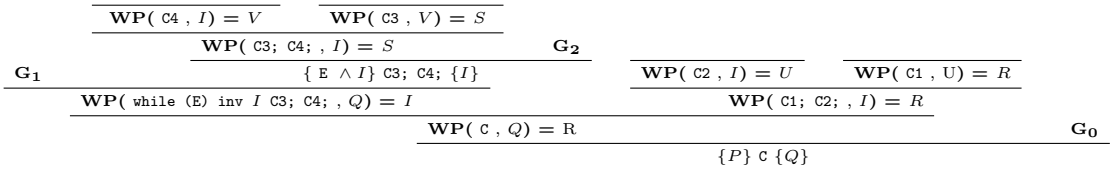
$$\overline{\mathbf{WP}(\texttt{ ar := [0, 1] }, U) = \forall y, z, y = \texttt{ 0 } \Rightarrow z = \texttt{ 1 } \Rightarrow U\,[ar/ar[0 \leftarrow y][1 \leftarrow z]]}$$

L'affectation de deux éléments simultanément dans le tableau `ar` a été modélisé comme l'affectation successive du premier élément puis du deuxième.

Après simplification, on peut exprimer complètement R :

$$R = \forall k \in [\![0, y[\![, U\,[ar/ar[0 \leftarrow 0][1 \leftarrow 1]] \tag{2.22}$$

Il ne reste plus de calcul de plus faible précondition, seulement 3 formules logiques à prouver : G_0, G_1 et G_2. On appelle ces formules des buts. La figure 2.12 présente l'arbre



On applique le calcul de plus faible précondition sur la fonction définie dans la figure 2.8, pour rappel :
$V = \forall k \in [\![0, i+3[\![, ar[k] = fib(k)$
$S = \forall k \in [\![0, i+3[\![, ar[i+2 \rightarrow ar[i+1] + ar[i]][k] = fib(k)$
$U = \forall k \in [\![0, 2[\![, ar[k] = fib(k)$
$R = \forall k \in [\![0, 2[\![, ar[0 \leftarrow 0][1 \leftarrow 1][k] = fib(k)$

FIGURE 2.12 – Arbre de dérivation d'un calcul de plus faible précondition

complet du calcul de plus faible précondition, les feuilles non vides de cet arbre correspondent aux 3 buts logiques qu'il reste à prouver.

Le premier but G_0 s'écrit :

$$n \geq 2 \Rightarrow \forall k \in [\![0, 2[\![, ar[0 \leftarrow 0][1 \leftarrow 1][k] = fib(k) \tag{2.23}$$

Que l'on prouva facilement en instanciant k avec 1 et 0. G_1 s'écrit :

$$(i \geq n-2 \wedge \forall k \in [0, i+2[, ar[k] = fib(k)) \Rightarrow \forall k \in [\![0, n[\![, ar[k] = fib(k) \tag{2.24}$$

Si on prend $k \in [0, n[$, on voit que l'on pourra instancier la partie droite du $\wedge$ grâce à la partie gauche et donc obtenir ce que l'on souhaite prouver.

Enfin G_3 s'écrit :

$$\begin{aligned}(i < n-2 \ \wedge \ \forall k \in [\![0, i+2[\![, ar[k] = fib(k)) \\ \Rightarrow \forall k \in [\![0, i+3[\![, ar[i+2 \rightarrow ar[i+1] + ar[i]][k] = fib(k)\end{aligned} \tag{2.25}$$

On prend $k \in [\![0, i+3[\![$, on peut faire deux cas, si $k < i+2$ alors on peut utiliser l'hypothèse et sinon on a directement le résultat par définition de la suite de Fibonacci.

Ce travail est déjà long et fastidieux avec un petit programme écrit en pseudo-code. Mais avec des langages bas niveau, comme le C, les accès mémoire rendent la chose encore plus complexe. C'est pourquoi des logiciels comme Frama-C permettent d'automatiser le processus.

2.3.3 Prouver la correction de code C

Frama-C

Frama-C est un logiciel permettant d'interpréter un code C annoté par ACSL. Il possède ensuite un certain nombre d'extensions permettant d'analyser le code. L'extension qui nous intéresse s'appelle WP. Comme son nom l'indique, elle permet d'effectuer un calcul de plus faible précondition sur un code C.

Comme on l'a vu, il est nécessaire pour un calcul de WP d'annoter les boucles par un invariant. Le langage ACSL supporte cette annotation avec le mot-clef `loop invariant`. Il est aussi nécessaire de spécifier les variables modifiées par la boucle via le mot-clef `loop assigns`. La figure 2.13 présente la boucle **`while`** annotée de la fonction de Fibonacci précédemment définie dans la figure 2.9.

```
1   /*@ loop invariant \forall integer k; (0 <= k < i + 2)
2                         ==> ar[k] == fib(k);
3       loop assigns i, ar[0 .. n-1]; */
4   while (i < n - 2) {
5     ar[i+2] = ar[i+1] + ar[i];
6     i += 1;
7   }
```

FIGURE 2.13 – Boucle while annotée par son invariant de boucle.

Le plug-in WP quant à lui, prend des fonctions C annotées par un contrat et des invariants de boucle et génère, à partir de celui-ci, une série de formules logiques, appelé but. Il garantit que si ces formules sont prouvées correctes alors le code C vérifie les contrats exprimés. Les formules peuvent être générées sous trois formats. Le premier format est celui d'Alt-Ergo qui est un solveur SMT (pour Satisfiabilité Modulo Théories). Le second est celui de Coq qui est un assistant de preuve. Enfin le troisième format est celui de Why3, c'est un logiciel permettant d'exporter un problème donné en entrée vers un grand nombre de prouveurs automatiques et assistants de preuve.

Modèle mémoire Un calcul de plus faible précondition repose toujours sur un modèle mémoire correspondant à la manière dont il représente dans la logique la mémoire et ces modifications. Le modèle le plus souvent utilisé est la logique de séparation introduite dans REYNOLDS, 2002. Elle supporte l'arithmétique des pointeurs tout en permettant de raisonner de manière locale dans un programme. Plus précisément, elle permet de ne tenir compte que de la mémoire utilisée par la portion de code en cours d'étude. Ainsi, on peut prouver une fonction sans avoir besoin de connaître dans quel contexte elle sera utilisée. Frama-C possède plusieurs modèles mémoires dont la plupart sont basés sur la logique de séparation. Ces modèles sont décrits dans le troisième chapitre de BAUDIN et al., p. d. Nous utiliserons dans ce travail le modèle par défaut : le modèle typé.

Prouver des formules

On peut séparer en deux grandes familles les outils informatiques permettant de prouver des formules logiques ou mathématiques : les prouveurs automatiques et les assistants de preuves

Les prouveurs automatiques prennent en entrée des formules logiques et statuent sans interaction humaine sur leur validité. Il n'y a aucune garantie qu'ils arriveront à prouver un but donné ou même qu'ils terminent dans un temps donné. Parmi les prouveurs automatiques, une grande famille se distingue, celle des solveurs SMT reposant sur un solveur SAT équipé d'un ensemble de théories permettant de décider de la validité des prédicats du premier ordre. On peut citer, par exemple, Alt-Ergo, CVC4, veriT, Yices ou Z3. Il existe aussi des SMT dédiés à des problèmes plus précis comme Boolector pour les vecteurs de bit ou Gappa pour les flottants. Parmi les autres prouveurs automatiques ne rentrant pas dans la catégorie des SMT, on peut mentionner Vampire ou bien encore E theorem prover.

Les prouveurs automatiques s'opposent aux assistants de preuve. Ces derniers permettent de vérifier formellement une preuve écrite par un humain. Ils apportent en général aussi un certain nombre d'outils pour aider la personne qui écrit la preuve. Ils possèdent l'avantage de pouvoir vérifier n'importe quelle preuve, mais le désavantage est qu'un humain doit l'écrire avant à la différence des prouveurs automatiques qui effectuent eux-mêmes cette tâche. Les assistants de preuves les plus connus sont Coq, Isabel, Mizar et PVS.

Sortie

```
[kernel] Parsing Fibonacci.c (with preprocessing)
[wp] Running WP plugin...
[wp] Warning: Missing RTE guards
[wp] 9 goals scheduled
[wp] [Qed] Goal typed_fibonacci_loop_assign_part1 : Valid
[wp] [Qed] Goal typed_fibonacci_loop_assign_part2 : Valid
[wp] [Alt-Ergo] Goal typed_fibonacci_loop_inv_established : Valid
[wp] [Alt-Ergo] Goal typed_fibonacci_post : Valid
[wp] [Qed] Goal typed_fibonacci_assign_part2 : Valid
[wp] [Alt-Ergo] Goal typed_fibonacci_loop_inv_preserved : Valid
[wp] [Qed] Goal typed_fibonacci_assign_part3 : Valid
[wp] [Alt-Ergo] Goal typed_fibonacci_loop_assign_part3 : Valid
[wp] [Alt-Ergo] Goal typed_fibonacci_assign_part1 : Valid
[wp] Proved goals:    9 / 9
  Qed:             4  (0.46ms-8ms-36ms)
  Alt-Ergo:        5  (7ms-27ms-91ms) (144)
```

FIGURE 2.14 – Sortie de Frama-C/WP sur l'exemple de Fibonacci

Prouver le code C de Fibonacci

On a exécuté Frama-C muni de son extension WP sur l'exemple de la fonction de Fibonacci précédemment écrite et annotée en C/ACSL que l'on peut trouver dans la figure 2.9. Le résultat de cette exécution peut être trouvé dans la figure 2.14. On voit que 9 buts ont été générés. Si l'on enlève les buts liés aux `assigns` on obtient bien 3 buts correspondant aux 3 buts que l'on a pu extraire lors du calcul de WP manuel. Parmi les 9 buts, 4 ont été prouvés automatiquement par Qed, un moteur de preuve simple interne à Frama-C. Les autres buts ont été prouvés par le solveur SMT Alt-Ergo dans des temps inférieurs à 100ms.

On a ainsi prouvé automatiquement que la fonction C, que nous avons écrite, calcule bien n itérés de la suite de Fibonacci.

2.3.4 Bilan

Dans cette section, nous avons vu que les triplets de Hoare permettaient de spécifier les propriétés attendues d'un code. Le calcul de plus faible précondition permet de vérifier automatiquement qu'un code vérifie sa spécification exprimée sous forme de triplet de Hoare. Cela est possible à deux conditions. La première est que les boucles soient annotées avec des invariants et les fonctions avec des contrats. La seconde condition est d'avoir des prouveurs automatiques assez puissants pour prouver les buts que génère un calcul de plus faible précondition.

Ces techniques ont été mises en place pour les programmes C grâce à Frama-C et son langage d'annotation ACSL. Les buts générés peuvent être ensuite prouvés par des prouveurs automatiques ou bien en utilisant un assistant de preuve.

Bilan

Dans ce chapitre, nous avons pu présenter les 3 domaines sur lesquels repose cette thèse. Nous avons, dans un premier temps, abordé la problématique du contrôle, en particulier la commande prédictive qui rend nécessaire la certification d'implémentation d'algorithme d'optimisation. Nous avons présenté, dans un second temps, différents problèmes d'optimisation et les algorithmes permettant de les résoudre. Dans un dernier temps, nous avons présenté la méthode de plus faible précondition de Dijkstra qui permet la preuve formelle de code et donc la certification formelle d'implémentations.

Est-ce qu'il est possible d'appliquer cette méthode à des algorithmes numériques complexes comme ceux d'optimisation, le tout dans le contexte de la génération de code pour le contrôle de système embarqué critique ? Question à laquelle nous allons apporter une réponse dans ce manuscrit.

Bibliographie

Baudin, Patrick, François Bobot, Loïc Correnson et Zaynah Dargaye (p. d.). *Frama-C/WP*. https://frama-c.com/download/frama-c-wp-manual.pdf (cf. p. 33).

Bensalem, Saddek, Yassine Lakhnech et Hassen Saidi (1996). "Powerful techniques for the automatic generation of invariants". In : *Computer Aided Verification*. Sous la dir. de Rajeev Alur et Thomas A. Henzinger. Berlin, Heidelberg : Springer Berlin Heidelberg, p. 323-335 (cf. p. 28).

Boyd, Stephen et Lieven Vandenberghe (2004). *Convex Optimization*. Cambridge University Press (cf. p. 17).

Dantzig, George B. (1990). "A History of Scientific Computing". In : sous la dir. de Stephen G. Nash. New York, NY, USA : ACM, p. 141-151 (cf. p. 19).

Dijkstra, Edsger W. (août 1975). "Guarded Commands, Nondeterminacy and Formal Derivation of Programs". In : *Commun. ACM* 18.8, p. 453-457 (cf. p. 27).

Mattingley, Jacob et Stephen Boyd (mar. 2012). "CVXGEN : a code generator for embedded convex optimization". In : *Optimization and Engineering* 13.1, p. 1-27 (cf. p. 4, 16, 147).

Mehrotra, Sanjay (1991). "On finding a vertex solution using interior point methods". In : *Linear Algebra and its Applications* 152, p. 233-253 (cf. p. 22).

Reynolds, John C. (2002). "Separation Logic : A Logic for Shared Mutable Data Structures". In : *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*. LICS '02. Washington, DC, USA : IEEE Computer Society, p. 55-74 (cf. p. 33).

Rockafellar, Ralph Tyrell (2015). *Convex analysis*. Princeton University Press (cf. p. 22).

Ypma, Tjalling J. (déc. 1995). "Historical Development of the Newton-Raphson Method". In : *SIAM Rev.* 37.4, p. 531-551 (cf. p. 20).

Deuxième partie

Contribution

Chapitre 3

Création de pySil, un langage pseudo-code capable d'exprimer un code impératif ainsi que des triplets de Hoare

Sommaire

Introduction

Ce chapitre est consacré au développement d'un langage de programmation que nous avons nommé PYSIL. Il doit permettre l'expression d'un code numérique destiné à l'embarqué. En plus de l'expression du programme, le langage doit permettre d'exprimer les propriétés que l'on attend de lui.

Dans une première section, nous choisirons le paradigme de programmation qui sera utilisé, ainsi que la méthode de spécification. Pour cela, nous établirons les critères que doit avoir le langage puis nous comparerons les différents paradigmes et les différentes méthodes de preuves.

Dans une deuxième section, nous développerons le cœur de notre langage qui aura donc un nombre limité de constructions facilement compilables pour de l'embarqué. Il devra aussi supporter la méthode de spécification choisie. Nous détaillerons en premier lieu le choix d'utiliser python comme métalangage. En second lieu, nous développerons la partie code du langage PYSIL. Enfin, en dernier lieu, nous incorporerons au sein du langage la partie annotation.

La troisième section sera consacrée à l'extension du langage pour supporter les opérations matricielles. Ceci permettra d'écrire de manière naturelle et lisible les algorithmes de points intérieurs dans PYSIL. Cette étape se décomposera en deux parties : l'introduction des opérations matricielles dans la partie code du langage puis le développement d'une librairie logique matricielle qui permettra de spécifier le code avec des annotations matricielles.

Enfin, dans une dernière section, nous plongerons cette extension matricielle dans le langage créé dans la deuxième section afin que l'introduction des matrices ne complexifie pas les étapes de compilation. Pour cela, nous introduirons une fonction de transformation puis prouverons sa correction en utilisant les sémantiques précédemment définies.

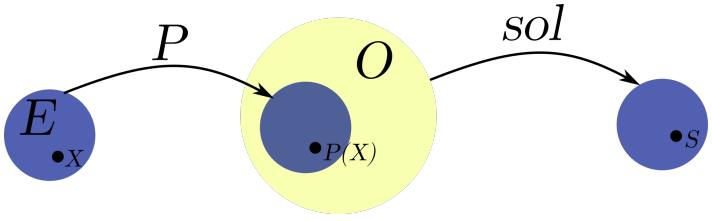
3.1 Choisir une méthode d'expression du code et de sa sémantique

Dans cette section, nous allons choisir le paradigme que devra suivre PYSIL. Pour cela, nous commencerons par établir les critères qu'il devra posséder. Après nous être intéressés aux langages existants, nous comparerons les différents paradigmes de langage de programmation. Enfin, nous comparerons différentes méthodes de vérification logicielle utilisable pour PYSIL.

But général

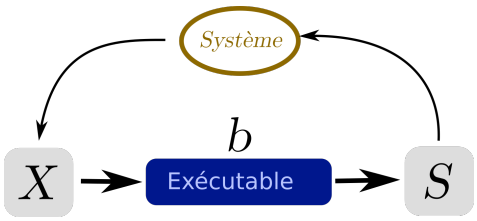
Avant de rentrer dans les détails, nous allons rappeler dans quel contexte sera utilisé le langage PYSIL que nous allons développer dans ce chapitre.

On souhaite résoudre un problème MPC en ligne(cf. section 2.1). Le problème MPC pourra s'exprimer comme un problème d'optimisation de classe O(cf. section 2.2) paramétré par un vecteur $X \in E$. E représente l'ensemble des états que peut prendre le système. On notera $sol(p)$ la fonction qui renvoit la solution d'un problème p de classe O.



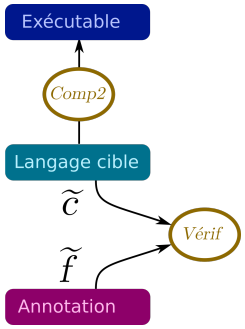
On veut pouvoir embarquer un binaire b qui prend en argument un $X \in E$ et dont on peut certifier qu'il renvoie $S = sol(P(X))$. Par ailleurs, on veut aussi que b soit très efficace

et donc pour cela qu'il soit aussi spécifique que possible à P.



Pour cela, on doit exprimer b dans un langage de programmation impératif, bas niveau que l'on nommera langage cible. On note $\tilde{c}$ l'expression de b dans le langage cible. On appellera $comp2$ l'étape de compilation de $\tilde{c}$ vers b et on la supposera certifiée.

Afin de garantir la correction du programme $\tilde{c}$, on souhaite avoir une spécification $\tilde{f}$ exprimant que $\tilde{c}$ renvoie bien une solution à $P(X)$. On veut alors avoir la preuve formelle que $\tilde{c}$ vérifie $\tilde{f}$. Cette étape se nommera $Verif$ et sera l'objet du chapitre 6.



Nous souhaitons automatiser le processus de génération de $\tilde{c}$ et $\tilde{f}$ à partir d'un problème P car c'est une étape longue et fastidieuse. Pour cela, on va se donner un algorithme que l'on appellera A qui permet de résoudre n'importe quel problème de O. Afin de proposer des outils qui ne sont spécifiques ni à A ni à O nous avons décidé d'utiliser un langage intermédiaire, PYSIL, pour y exprimer A.

Soit $\mathbf{P}$ l'ensemble des programmes écrit en PYSIL. Un programme PYSIL (c, f) sera formé de deux objets mathématiques [1]. Le premier objet, c, représente la partie algorithmique de p, celle qui effectue des calculs, modifie la mémoire ou affecte l'environnement extérieur. C'est ce qu'on appelle habituellement un programme. Le second objet, f, est la partie déclarative de p, elle représente sa spécification. Elle permet d'exprimer les propriétés que l'on veut que notre programme vérifie.

Comme b doit être spécifique à P, $\tilde{c}$ aussi et donc c aussi. Ainsi on générera programme c et son annotation f à partir de P. Cette étape se nommera Gen. Le programme p représentera une version de A spécifique à P. Le choix de O et A ainsi que l'expression de A en PYSIL seront l'objet du chapitre 4. Une étape de compilation $comp1$ s'occupera de la compilation de p en $(\tilde{c}, \tilde{f})$. Cette étape sera abordée au chapitre 5.

On peut remarquer que la plupart de ce qui est présenté ici sera traité plus tard dans le manuscrit, mais il est important d'avoir le schéma général à l'esprit afin de faire les bons choix de conception pour PYSIL. La figure 3.1 résume ces informations.

3.1.1 Définition de critères

Nous allons établir les critères que le langage devra satisfaire. Pour cela, nous commencerons par rappeler les quatre buts que nous nous sommes fixés et en extraire les critères souhaités.

1. En pratique, la syntaxe de PYSIL ne permettra pas de séparer physiquement ces deux parties.

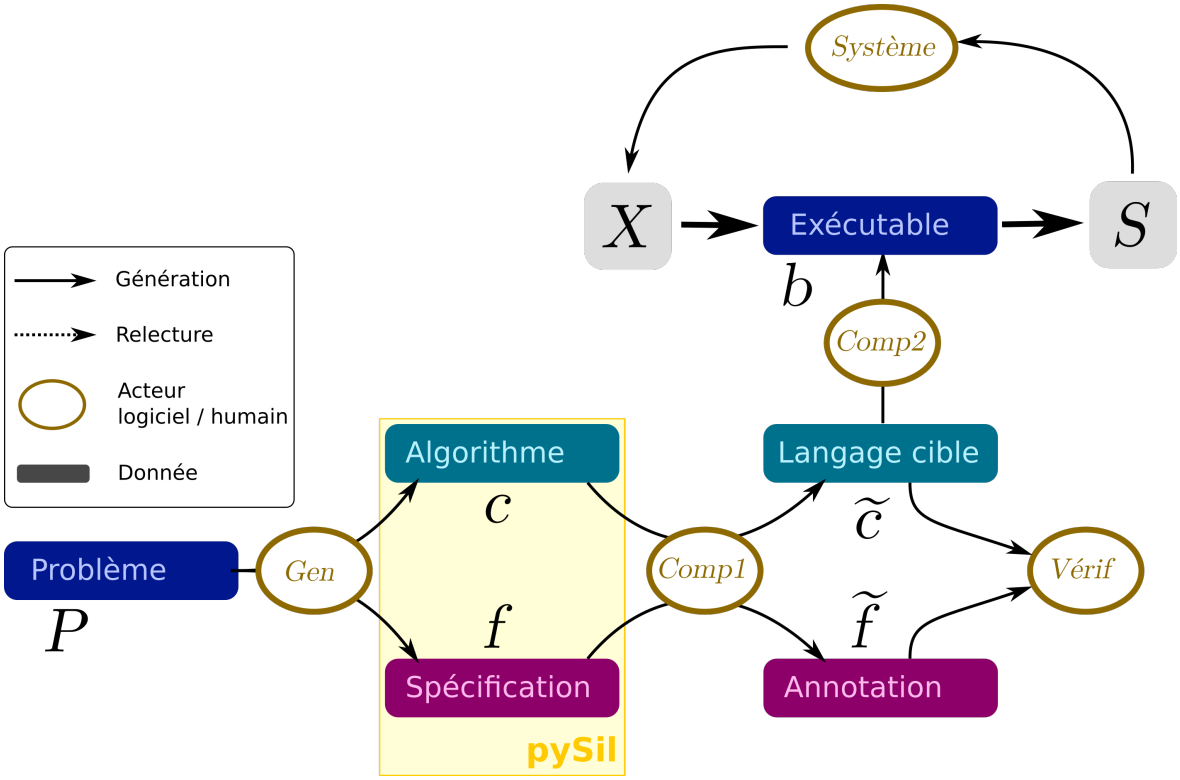


FIGURE 3.1 – Écosystème dans lequel se place PYSIL

Expression du programme Notre premier but est de pouvoir exprimer un algorithme dans notre langage. Cela correspond à la partie c de $p = (c, f)$. Plus précisément, ce sont les algorithmes de points intérieurs qui nous intéressent. Donc notre langage devra être capable de supporter facilement des opérations numériques, en particulier matricielles. Le langage devra être le plus proche possible d'un format de type pseudo-code que l'on pourrait trouver dans une publication scientifique de ce domaine. Cela doit permettre à quelqu'un qui n'est pas issu du monde des méthodes formelles de lire ou d'écrire des programmes dans ce langage.

Critère 3.1.1 *Le langage doit être accessible et cohérent avec les problématiques numériques.*

Expression des propriétés Le second point fondamental à notre travail est la capacité d'annoter ce code avec les propriétés qu'il doit vérifier. Cela correspond à la partie f de $p = (c, f)$. On souhaite écrire un algorithme de points intérieurs résolvant le problème $P(A, b, c)$. Les propriétés principales que l'on souhaite pouvoir écrire sont les suivantes :

Faisabilité de la solution :

$$f_1(A, b, c, x) = Ax \leq b \tag{3.1}$$

ϵ-optimalité de la solution :

$$f_2(A, b, c, x) = \langle c,\ x\rangle - S(A, b, c) < \epsilon \tag{3.2}$$

Critère 3.1.2 *Le langage doit être capable d'exprimer des propriétés mathématiques sur des variables du programme*

Génération de code embarquable Nous souhaitons que ce langage puisse ensuite être compilé vers un langage de plus bas niveau. Comme nous nous plaçons dans le cadre de l'embarqué, nous souhaitons aussi pouvoir traiter des problématiques d'optimisation du code généré. Un besoin classique quand il s'agit d'optimiser le code est l'utilisation d'un métalangage, comme le préprocesseur C, qui doit permettre d'effectuer des précalculs, simplifier les expressions ou encore introduire des macros.

Critère 3.1.3 *Le langage doit permettre l'optimisation du code généré.*

Preuve du programme généré On souhaite donc générer un code embarquable, mais pas seulement. On souhaite apporter la garantie formelle que ce code est exempt de tout bogue. Plus précisément, nous souhaitons prouver mathématiquement que le résultat du programme est correct ; qu'il satisfait sa spécification.

Critère 3.1.4 *Le langage doit faciliter le travail visant à garantir la correction du code généré*

Combiné avec le but précédant ce dernier induit un nouveau critère : pouvoir changer des parties du code à des fins d'optimisation. Le but est qu'une modification quelconque d'un code prouvé nécessite une modification proportionnée de la preuve. En effet, il est commun en informatique de commencer par travailler sur un code peu efficace. Ce code est ensuite affiné afin d'améliorer son efficacité. Ce processus doit être faisable sans avoir à recommencer intégralement le travail de preuve.

Critère 3.1.5 *Le langage doit permettre de modifier une partie du code sans que cela impacte plus que nécessaire la partie preuve.*

3.1.2 Solution existante

La problématique de générer un programme bas niveau annoté pour de l'embarqué critique n'est pas nouvelle, un certain nombre d'outils existe déjà.

On peut mentionner la méthode B introduite dans Abrial, 1996 utilisé dans l'industrie ferroviaire. Cette méthode permet d'exprimer un code de haut niveau et sa spécification puis, par raffinement, d'arriver à un code facilement transcodable en C. Cette méthode valide tous nos critères hormis le critère 3.1.1. La méthode B n'est pas facile d'accès. De plus, elle est principalement utilisée pour tenir compte des propriétés discrètes des programmes et peu des problématiques numériques.

Le langage Boogie spécifié dans Leino, 2008 propose d'exprimer un programme impératif et numérique muni d'annotation. Un logiciel éponyme permet ensuite de vérifier la correction des annotations automatiquement. Ce langage ne permet pas l'exportation native vers un code de bas niveau, donc il faudrait l'implémenter. De plus, le langage a été construit pour servir de langage intermédiaire entre un langage de programmation et des prouveurs automatiques. Ainsi, son essence première n'est pas la simplicité pour un humain, en particulier extérieur aux méthodes formelles (critère 3.1.1). En conclusion, utiliser ce langage pour notre problématique nécessiterait une assez grande charge de travail pour utiliser un langage assez éloigné du concept de pseudo-code.

Le langage WhyMl présenté dans Filliâtre et Paskevich, 2013 est assez proche du langage Boogie, et en possède les mêmes désavantages : un langage pas forcément simple d'utilisation et avant le début de cette thèse il ne possédait pas non plus de compilateur vers un langage de bas niveau. Entre temps, l'article Rieu-Helft, Marché et Melquiond, 2017 est sorti en 2017 et présente un compilateur de WhyMl vers le langage C. Leur approche correspond relativement bien à nos attentes et son utilisation peut être une alternative au langage présenté ici. En revanche leur compilateur ne supporte pas encore la génération des

annotations et les algorithmes qu'ils prennent pour exemple (calcul de grand nombre) sont relativement éloignés du domaine de l'optimisation convexe.

On pourrait aussi utiliser des assistants de preuve qui proposent en général l'expression de programmes. En particulier Reif, 1992 ou Appel, 2011 pourraient correspondre à nos attentes. Cependant, le passage par un assistant de preuve rend difficile l'utilisation par un public non familier avec les méthodes formelles. De plus, le but avec notre langage est l'expression simple et claire de l'algorithme et de ses annotations. Nous ne souhaitons pas prouver le code à ce niveau-là, mais une fois la compilation vers un langage bas niveau effectué.

Tous ces langages possèdent leur avantage, mais ont été conçus pour permettre leur preuve et donc se destinent au monde de la vérification logicielle. Avec notre langage pySil, nous souhaitons qu'il soit simple et naturel d'exprimer un algorithme d'optimisation ainsi que sa spécification. Aucun de ces langages ne nous satisfaisant, nous avons décidé d'en créer un nouveau correspondant à nos exigences.

3.1.3 Choix du paradigme de langage de programmation

Il existe trois grandes familles de paradigmes de langages de programmation :

— La programmation impérative

— La programmation déclarative

— La programmation orientée objet

Le troisième paradigme permet l'utilisation d'objet dans un langage. Les objets permettent de représenter un concept, une idée, ou tout objet physique. Cela est particulièrement utile pour de gros projets que l'on peut décomposer en sous-partie. La présence d'objet au sein d'un langage entraine une compilation plus complexe puisqu'ils finiront par être encodés dans un paradigme impératif. De plus, cela se fait en général au détriment de l'efficacité du code, car l'utilisation d'objets rajoute des couches de calcul. Le seul critère correspondant aux apports de l'orienté objet serait, 3.1.1 mais dans le cas d'algorithme numérique le gain ne serait pas énorme en lisibilité par contre la complexité que cela apporterait rendrait la tâche beaucoup plus difficile.

Dans la programmation déclarative, on peut éliminer les langages purement descriptifs qui ne correspondent pas à notre problématique et les langages par contraintes qui sont peu faciles d'accès et très éloignés des aspects embarqués et numériques. Il ne reste donc que la programmation fonctionnelle. D'un côté, elle possède l'avantage d'être plus facilement manipulable dans le cadre de la vérification logicielle, de l'autre côté, elle n'est pas toujours naturelle pour quelqu'un d'extérieur au monde de l'informatique théorique. De plus, sa compilation vers des langages de plus bas niveau est complexe, en particulier, si l'on souhaite un code optimisé. Enfin, nous souhaitons avant tout prouver le code qui sera généré. Or celui-ci est par nature dans un paradigme impératif. Pour ces raisons, nous avons éliminé la programmation fonctionnelle.

Si on regarde dans les publications scientifiques les algorithmes d'optimisation, on observe qu'ils sont dans un format impératif. Le langage cible sera lui aussi impératif, car aussi bas niveau que possible. C'est pourquoi utiliser un paradigme impératif pour pySil semble parfaitement adapté. En effet, il correspond bien au critère 3.1.1. Les phases de compilation seront plus simples qu'avec un langage fonctionnel et donc plus facile à optimiser ce qui simplifie le critère 3.1.3. On a donc fait le choix d'utiliser un paradigme impératif.

Nous allons voir dans la sous-section suivante quelle méthode de spécification nous allons utiliser pour exprimer les propriétés sur les programmes.

3.1.4 Choix de la méthode de spécification

Afin de satisfaire le critère 3.1.2, nous souhaitons, pouvoir exprimer au sein du langage de type pseudo-code, les propriétés que nous attendons du code. On appelle ces spécifications des annotations, car elles se rajoutent au code. Ces annotations doivent pouvoir exprimer des propriétés sur la mémoire du programme. Pour cela on utilise une sémantique. On distingue trois types de sémantique :

— La sémantique opérationnelle

— La sémantique dénotationelle

— La sémantique axiomatique

La sémantique opérationnelle considère le programme comme la suite d'états qu'il génère au niveau de la machine. Comme nous travaillons avec un langage de type pseudo-code, il faudrait construire une machine ad hoc. La compilation de ces annotations vers des annotations sur un langage bas niveau serait très complexe (critère 3.1.4) : il faudrait transformer la machine du pseudo-code en une machine bas-niveau puis transformer les propriétés. Cela reste cependant possible. Par contre, cela rendrait les annotations en PySil et leur compilation très complexe ce qui est incompatible avec le critère 3.1.1. De plus, elles seraient dépendantes de la structure du programme ce qui implique que chaque modification du code entrainerait des modifications des annotations ce qui est en contradiction avec le critère 3.1.5.

La sémantique dénotationelle considère le programme en un objet mathématique. On peut alors aisément exprimer n'importe quelles propriétés. Par contre, manipuler mathématiquement des programmes n'est pas quelque chose de naturel pour quelqu'un qui n'est pas issu du monde des méthodes formelles (critère 3.1.1). Concernant le critère 3.1.5, cette sémantique est plutôt adéquate, car en optimisant un programme on laisse le résultat du calcul inchangé et donc les objets mathématiques modélisant ce calcul devraient rester identiques. Pour l'aspect compilation et les problématiques issues du critère 3.1.4, cela nécessiterait d'avoir des sémantiques cohérentes entre notre langage de type pseudo-code et le langage cible ce qui ne serait pas aisé, mais faisable.

La sémantique axiomatique considère le programme comme une fonction mathématique transformant une mémoire en une autre. Les annotations seront des prédicats sur la mémoire donc, dans le cas d'un langage impératif de type pseudo-code, les variables. Parmi les trois sémantiques, c'est l'objet à manipuler le plus simple. Pour rappel, dans la première on manipulait des états d'une machine et dans la deuxième on manipulait des objets mathématiques représentant des programmes. Ainsi, elle est plus simple à appréhender ce qui la rend la plus proche du critère 3.1.1. Concernant le critère 3.1.5, les annotations reposent sur la mémoire donc tant que les optimisations sur le code ne changent pas la structure de la mémoire, les annotations n'auront pas besoin de changer. Pour la compilation du code, on devra transformer les variables du pseudo-code en variable du langage de bas niveau. Cette étape aura déjà été en grande partie effectuée pour la compilation du code ce qui satisfait bien le critère 3.1.4.

Au vu des critères validés par la sémantique axiomatique, nous avons décidé de l'utiliser pour annoter notre pseudo-code. Plus précisément, nous utiliserons la logique de Hoare introduite à la section 2.3.

3.2 Création d'un cœur impératif supportant la logique de Hoare

Dans cette section, nous allons présenter la construction du cœur impératif du langage PySil signifiant "Python Simple Imperative Language". Le langage PySil a été créé comme

un module python. Nous commencerons par justifier ce choix puis nous développerons la partie code du langage avant d'incorporer la partie annotation par triplet de Hoare.

Nous allons définir la grammaire de PYSIL dans un format basé sur la forme de Backus-Naur (BNF). Pour parler de ce format, on parlera de méta syntaxe dont nous allons maintenant donner les éléments importants. Chaque unité syntaxique portera un nom noté entre chevrons(**<unité>**). On définira une unité syntaxique grâce à l'opérateur `:=`. Tout élément entre accolades lors de la définition d'une unité syntaxique signifie que cet élément est optionnel. Le symbole `|` représentera le choix entre deux unités syntaxiques.

Dans PYSIL, toutes les unités syntaxiques représentant une expression sont typées. Nous reflétons ceci dans la méta syntaxe en rajoutant la possibilité de spécifier un type entre les chevrons par l'opérateur `:`, cela donne pour `unité` de type **Type** : **<unité:Type>**.

<integer>, **<float>**, **<bool>** représente respectivement un entier, un flottant et un booléen de python. Ils sont rassemblés dans l'unité syntaxique **<value>** :

```
<value> := <integer> | <float> | <bool>
```

<ident> représente un nom de variable python, il est composé d'une séquence de lettre, chiffre et blanc souligné conformément à la syntaxe python. Les variables python et donc les **<ident>** sont utilisés pour stocker les programmes, les fonctions et enfin les variables PYSIL.

<name> est une chaine de caractère python, c'est-à-dire entouré de guillemets (simple ou double), il représente le nom d'un élément PYSIL comme une variable ou une fonction.

<ident> est en général le nom d'une variable python qui contient un objet PYSIL alors que **<name>** est le nom de l'objet lui-même. Cette notion d'objet PYSIL et la présence de syntaxe python dans notre langage vont être abordées avec plus de détail dans la section suivante.

3.2.1 Utilisation de python comme métalangage

Les critères 3.1.4, 3.1.3 et 3.1.5 laissent entrevoir la nécessité d'instrumenter facilement le code que l'on écrit en PYSIL. En effet, on souhaite être en mesure de remplacer simplement des parties du code. On aura aussi besoin d'effectuer un certain nombre de calculs avant l'exécution du code lui-même. En particulier, lors de l'utilisation des algorithmes de points intérieurs pour du contrôle, on doit pouvoir générer le programme d'optimisation à partir de la donnée d'un modèle. On verra aussi dans le chapitre 6 que la preuve nécessitera aussi de pouvoir instrumenter code et annotation.

Afin de simplifier ces tâches, PYSIL a été construit comme un module PYTHON. Cela veut dire que chaque élément de la syntaxe PYSIL pourra être vu comme un objet PYTHON. Plus précisément, un script PYSIL pourra être exécuté en tant que script python et alors le programme PYSIL sera un objet PYTHON, ses fonctions et ses variables aussi. Cela permettra de construire de manière dynamique des objets PYSIL à partir d'objet PYTHON. Par exemple, une matrice PYSIL pourra être construire à partit du résultat d'un précalcul effectué en PYTHON. On pourra aussi utiliser les variables, les fonctions et structures de contrôles de PYTHON pour instrumenter le code. Par exemple, une boucle PYTHON **for** pour répéter une instruction PYSIL ou une fonction pour paramétriser un bout de code PYSIL. PYTHON est adapté pour cette tâche, car c'est un langage de script donc facile d'utilisation en particulier quand il sert d'outil. Il est de plus très peu verbeux ce qui permettra de ne pas alourdir la syntaxe PYSIL. De plus, il possède des librairies très complètes de calcul numérique comme NUMPY (OLIPHANT, 2006) ou de calcul symbolique comme SYMPY (MEURER et al., 2017).

Définir un langage comme un objet PYTHON signifie que les types, les structures de contrôle, les instructions et tous les éléments de notre langage seront des objets PYTHON. Cela aurait pu alourdir grandement la syntaxe et la faire s'éloigner de l'objectif de simplicité.

Cependant, avec les capacités de méta programmation, de polymorphisme et la simplicité de base de PYTHON, PYSIL pourra rester un langage proche du pseudo-code.

Dans cette section, nous avons rappelé les buts recherchés pour le développement de PYSIL. Par la suite, nous avons listé les critères qui en découlaient afin de choisir un paradigme de programmation et une méthode de spécification. Nous avons ainsi choisi un paradigme impératif et de la logique de Hoare pour construire PYSIL. Nous allons maintenant développer un cœur impératif pour PYSIL qui contiendra toutes les composantes de base pour écrire un programme de manière impérative, mais aussi l'annoter avec de la logique de Hoare.

3.2.2 Sémantique

En même temps que donner la syntaxe de notre langage, nous allons lui donner une sémantique dénotationelle. Celle-ci ne correspond pas à la sémantique qui sera ensuite utilisée pour l'annoter. Elle permettra principalement de préciser la signification des différents éléments du langage. L'ensemble des programmes que l'on peut écrire sera noté $\mathbf{P}$, il correspond à l'unité sémantique **<program>**. Soit $p \in \mathbf{P}$, calculer sa sémantique $[\![\texttt{p}]\!]$ se fera en deux étapes. La première étape consiste à extraire de p l'ensemble de ces variables $(\!| \texttt{p} |\!)$ afin de constituer l'état du programme p, on appellera cet ensemble le support (des états) de p. La seconde étape consiste à calculer la sémantique des différentes fonctions définies en PYSIL à travers une définition de $[\![\cdot]\!]$ par induction sur la syntaxe PYSIL. La sémantique d'une fonction associera à chaque état du programme un nouvel état.

Les variables étant typées en PYSIL il faut d'abord définir l'ensemble des types de PYSIL :

Définition 3.2.1 (L'ensemble des types) *On appelle l'ensemble des types* $\mathbf{T} = \{\mathbb{Z}, \mathbb{R}, \mathbb{B}\} \cup \{\mathbb{Z}^n \mid n \in \mathbb{N} \setminus \{0,1\}\} \cup \{\mathbb{R}^n \mid n \in \mathbb{N} \setminus \{0,1\}\} \cup \{\mathbb{B}^n \mid n \in \mathbb{N} \setminus \{0,1\}\}$.

C'est l'ensemble des variantes mathématiques des types définis en PYSIL (**<type>**) augmenté des types tableaux correspondants. On associera dans la suite les types **<type>** à leurs équivalents mathématiques.

$\mathbf{T}$, par construction est infini, cependant, cet aspect n'est pas utile et compliquerait les définitions. Ainsi il sera considéré comme fini. Pour cela, on peut borner les trois n de la définition de $\mathbf{T}$ par N. N doit être pris supérieur à toutes les tailles de tableaux présents dans le programme.

Le support de p, qui représente l'ensemble de ces variables, n'est pas en réalité un ensemble, mais un n-uplet d'ensembles, chaque composante correspondant à un type précis.

On notera $\mathbf{I}$, l'ensemble des identifiants que l'on peut écrire en PYSIL, noté **<ident>** dans la syntaxe.

Définition 3.2.2 (Ensemble des supports) *L'ensemble des supports existants en* PYSIL *est* $\mathbf{V} = \{V \in \prod_{\mathbb{T} \in \mathbf{T}} \mathcal{P}(\mathbf{I}) | \forall \mathbb{T}, \mathbb{U} \in \mathbf{T}, \mathbb{T} \neq \mathbb{U}, V_\mathbb{T} \cap V_\mathbb{U} = \emptyset \}$ *avec* $V_\mathbb{T}$ *la composante de* V *associé à* $\mathbb{T}$*, cette notation sera réutilisée dans la suite.*

On introduit aussi un opérateur d'union $\sqcup$ *des éléments de* $\mathbf{V}$ *comme étant l'union composante par composante ainsi que* $\boldsymbol{\emptyset} = (\emptyset, ..., \emptyset) \in \mathbf{V}$ *son élément neutre.*

Nous avons défini ce qu'était un support en PYSIL nous pouvons donc définir la notion d'état d'un programme :

Définition 3.2.3 (Ensemble des états) *Soit* $V \in \mathbf{V}$ *le support d'un programme* p*, on définit pour chaque* $\mathbb{T} \in \mathbf{T}$*,* $\Sigma^V_\mathbb{T} = V_\mathbb{T} \to \mathbb{T}$.

L'ensemble des états que peut prendre un programme de support V *est* $\Sigma^V = \prod_{\mathbb{T} \in \mathbf{T}} \Sigma^V_\mathbb{T}$.

Soit $\sigma \in \Sigma^V$*, pour chaque* $\mathbb{T} \in \mathbf{T}$ *on notera* $\sigma_\mathbb{T}$ *la composante de* σ *associé à* $\mathbb{T}$.

Nous avons donc défini la notion d'état qui associe à chaque variable du programme une valeur. Nous allons maintenant définir la notion d'environnement qui est une fonction qui associe à chaque nom la sémantique de la fonction correspondante. La sémantique d'une fonction, quant à elle, se définit par une fonction qui transforme l'état d'un programme en un autre état : $\Sigma \to \Sigma$

Définition 3.2.4 (Ensemble des environnements) *On définit $\bot$ un symbole qui représentera une fonction non définie.*

Soit $V \in \mathbf{V}$, l'ensemble des environnements associé à l'ensemble des états Σ^V est $\Gamma^{\Sigma^V} = \mathbf{I} \to ((\Sigma^V \to \Sigma^V) \cup \{\bot\})$.

On introduit l'environnement initial, $\mathbf{\bot}$, qui associe $\bot$ à tout les noms :

$$\begin{array}{rccc} \mathbf{\bot} & : & \mathbf{I} & \to & ((\Sigma^V \to \Sigma^V) \cup \{\bot\}) \\ & & x & \mapsto & \bot \end{array} \tag{3.3}$$

Maintenant que nous avons introduit quelques éléments de syntaxe et sémantique, nous allons pouvoir passer à la création du langage PYSIL.

3.2.3 Création de la partie code de pySil

Le but de PYSIL est d'être facile à appréhender via une syntaxe proche du pseudo-code, mais aussi facile à exporter vers un langage bas niveau. Nous avons choisi d'utiliser un paradigme impératif, mais nous garderons ce but à l'esprit durant la création du langage

Nous avons réparti les unités syntaxiques de PYSIL en trois groupes :

— les définitions

— les instructions

— les expressions

Chacune sont définies dans un fichier PYTHON correspondant du répertoire `pySil/base/`, respectivement `definitions.py`, `statements.py` et `mtype.py`. Ainsi, si on prend un code PYSIL et qu'on rajoute au début du fichier la ligne `from pySil import *`, le fichier devient un code PYTHON correct. L'exécution de ce fichier génère automatiquement un arbre abstrait (AST) représentant le programme. Pour rappel, le code peut être trouvé à l'adresse suivante : https://github.com/davyg/pyOptimToC.

Il nous faut choisir une manière de symboliser l'appartenance d'une instruction à un bloc comme une fonction ou une boucle. En C et dans beaucoup d'autre langage, on utilise des accolades. En BASIC on utilise souvent les `begin/end`, PYTHON utilise les indentations.

Le code PYSIL doit pouvoir être exécuté comme un code PYTHON afin de former son AST. Ainsi la manière de traduire l'appartenance d'une instruction à un bloc doit être choisie en accord avec la syntaxe PYTHON. Une solution simple serait d'introduire les mots clefs `begin()` et `end()` comme en BASIC. Cela fonctionnerait très bien, mais cette syntaxe est de base assez lourde par rapport aux accolades par exemple. De plus, la lecture d'un programme PYSIL deviendrait difficile, car il est impossible d'indenter librement un programme PYTHON. Ainsi tout code PYSIL devrait s'écrire sans indentation.

Ce n'est donc pas cette solution qui a été retenue, mais l'utilisation de la syntaxe **`with`**. Cette syntaxe reçoit un objet en argument et est suivie par un bloc d'instruction. Lorsque PYTHON exécute le with, il commence par appeler la fonction `__enter__` puis exécute le bloc et enfin appel la fonction `__exit__`. Cela correspond parfaitement à notre besoin, l'objet que l'on donnera au **`with`** correspondra au bloc que l'on crée, par exemple une fonction. Le bloc PYTHON qui suit contiendra les instructions PYSIL et les appels à `__enter__` et `__exit__` pourront être utilisés pour construire l'AST.

Par exemple pour une fonction `"ma_fonction"` composé des instructions `i1`, `i2` et `i3`, on pourra l'écrire :

```
mf = Function("ma_fonction")
with mf:
    i1
    i2
    i3
```

La manière compacte et proche d'un pseudo-code est la suivante :

```
with Function("ma_fonction"):
    i1
    i2
    i3
```

On peut aussi définir la fonction en plusieurs fois :

```
mf = Function("ma_fonction")

with mf:
    i1
    i2

with mf:
    i3
```

On peut utiliser l'opérateur **as** pour placer dans une variable un objet créé directement dans le **with**. Ainsi une quatrième manière de définir la fonction "ma_fonction" est la suivante :

```
with Function("ma_fonction") as mf:
    i1
    i2

with mf:
    i3
```

Le contenu d'un bloc **with** est donc une suite d'instructions python que nous exprimerons au niveau syntaxique à travers l'opérateur : **LIST**( ) :

```
LIST(<X>) :=
    pass |
    <X>
    LIST(<X>)
```

pass signifie que le bloc est vide. La sémantique associée à **LIST**( ) liste est la composition :

Définition 3.2.5 (Composition d'instruction) *Soit un type d'unité sémantique* **<unit>**, *u une unité syntaxique de type* **<unit>**, *us une unité syntaxique de type* **LIST**(**<unit>**).

On définit la sémantique de **LIST**(**<unit>**) *par* $[\![\cdot]\!]^V_{list}$ *avec* $[\![\textbf{pass}]\!]^V_{list} = \textbf{Id}$ *et* $[\![u\ us]\!]^V_{list} = [\![us]\!]^V_{list} \circ [\![u]\!]^V_{unit}$.

De même on définit $(\!| \cdot |\!)_{list}$ *par* $(\!| \textbf{pass} |\!)_{list} = \emptyset$ *et* $(\!| u\ us |\!)_{list} = (\!| u |\!)_{unit} \cup (\!| us |\!)_{list}$.

Maintenant que nous avons terminé ces quelques remarques préliminaires nous pouvons créer la syntaxe et la sémantique du coeur impératif du langage PYSIL. La syntaxe complète est donnée dans la figure 3.2 et sa sémantique dans la figure 3.3.

```
<value> := <integer> | <float> | <bool>
<ident> := [a-Z0-9]*
<name> := "[a-Z0-9]*"
<type> := Real | Int | Bool
<cmp> := > | >= | < | <= | == | !=
<unary> := - | ~
<binary> := + | - | * | /

<expr> := <left> | <type>.Cst(<value>)
    | <unary> <expr> | <expr> <cmp> <expr>
    | <expr> <binary> <expr>

<assign_stmt> :=
    <left:T> %= <expr:T>
<left:T> := <ident:T> | <ident:T>[<integer>]
<call_stmt> := <ident>()
<if_stmt> :=
    with If(<expr:Bool>):
        LIST(<stmt>)
    {with Else():
        LIST(<stmt>)}
<stmt> := <assign_stmt> | <call_stmt> | <if_stmt>
                        | <for_stmt> | <definition>

<fun_def> :=
    with Function(<name>) as <ident>:
        LIST(<stmt>)
<var_def> :=
    <ident:T> = <type:T>.Var(<name>)
<array_def> :=
    {<ident:T> =} <type:T>.Array(<name>, <integer>)
<definition> := <fun_def> | <var_def> | <array_def>

<program> :=
    with Program(<name>) {as <ident>}:
        LIST(<fun_def>)
```

FIGURE 3.2 – Grammaire du coeur impératif de PYSIL.

Soit $\mathbb{T} \in \mathbf{T}$, $V \in \mathbf{V}$, $\sigma \in \Sigma^V$ $y \in V_T$ et $\gamma \in \Gamma^{\Sigma^V}$.

$$
\begin{aligned}
[\![\texttt{pass}]\!]^V_{list} &= \boldsymbol{Id} \\
[\![\texttt{u us}]\!]^V_{list} &= [\![\texttt{us}]\!]^V_{list} \circ [\![\texttt{u}]\!]^V_{unit} \\
[\![\texttt{with Program(n): lines}]\!] &= (V, [\![\texttt{lines}]\!]^V_{list} \bot) \\
&\quad \text{avec } V = (\!| \texttt{lines} |\!)_{list} \\
[\![\texttt{with Function(n) as x: lines}]\!]^V_{fun}(\gamma) &= \gamma \uplus \{x \mapsto [\![\texttt{lines}]\!]^V_{list}(\gamma)\} \\
[\![\texttt{x = T.Var(n)}]\!]^V_{vars} &= \boldsymbol{Id}_{\Sigma^V} \\
[\![\texttt{x = T.Array(n, s)}]\!]^V_{vars} &= \boldsymbol{Id}_{\Sigma^V} \\
[\![\texttt{x \%= v}]\!]^V_{inst}(\gamma,\sigma)(y) &= \begin{cases} v & \text{si } x = y \text{ et } \mathcal{T}^V(x) = \mathbb{T} \\ \sigma_{\mathbb{T}}(y) & \text{sinon} \end{cases} \\
[\![\texttt{x[i] \%= v}]\!]^V_{inst}(\gamma,\sigma)(y) &= \begin{cases} \sigma_{\mathbb{T}}(y)[i \leftarrow v] & \text{si } x = y \text{ et } \mathcal{T}^V(x)^{\mathcal{S}^V(x)} = \mathbb{T} \\ \sigma_{\mathbb{T}}(y) & \text{sinon} \end{cases} \\
[\![\texttt{f()}]\!]^V_{inst}(\gamma,\sigma) &= \gamma(f)(\sigma) \\
[\![\texttt{with If(b): l}]\!]^V_{inst}(\gamma,\sigma) &= \begin{cases} [\![\texttt{l}]\!]^V_{list}(\gamma,\sigma) & \text{si } [\![\texttt{b}]\!]^V_{expr}(\sigma) = \boldsymbol{V} \\ \sigma & \text{sinon} \end{cases} \\
[\![\texttt{with If(b):l1 with Else():l2}]\!]^V_{inst}(\gamma,\sigma) &= \begin{cases} [\![\texttt{l1}]\!]^V_{inst}(\gamma,\sigma) & \text{si } [\![\texttt{b}]\!]^V_{expr}(\sigma) = \boldsymbol{V} \\ [\![\texttt{l2}]\!]^V_{inst}(\gamma,\sigma) & \text{sinon} \end{cases} \\
[\![\texttt{with For(d, f) as i: l}]\!]^V_{inst}(\gamma,\sigma) &= \begin{cases} [\![\texttt{l}]\!]^V_{inst}(\gamma,\sigma) & \text{si } [\![\texttt{i < f}]\!]^V_{expr}(\sigma) = \boldsymbol{V} \\ \sigma & \text{sinon} \end{cases} \\
[\![\texttt{x}]\!]^V_{\mathbb{T}}(\sigma) &= \sigma(x) \\
[\![\texttt{x[i]}]\!]^V_{\mathbb{T}}(\sigma) &= \sigma(x)_i \\
[\![\texttt{T.Cst(v)}]\!]^V_{\mathbb{T}}(\sigma) &= [\![\texttt{v}]\!] \\
[\![\texttt{f(a\_0, ..., a\_n)}]\!]^V_{\mathbb{T}}(\sigma) &= [\![\texttt{f}]\!]([\![\texttt{a\_0}]\!]^V_{\mathbb{T}}(\sigma), ..., [\![\texttt{a\_n}]\!]^V_{\mathbb{T}}(\sigma))
\end{aligned}
$$

$$
\begin{aligned}
(\!| \texttt{pass} |\!)_{list} &= \boldsymbol{\emptyset} \\
(\!| \texttt{u us} |\!)_{list} &= (\!| \texttt{u} |\!)_{unit} \cup (\!| \texttt{us} |\!)_{list} \\
(\!| \texttt{with f: l} |\!)_{fun} &= (\!| \texttt{l} |\!)_{list} \\
(\!| \texttt{x = T.Var(n)} |\!)_{vars} &= V \text{ avec } V_{\mathbb{T}} = \begin{cases} \{x\} & \text{si } \mathbb{T} = T. \\ \emptyset & \text{sinon.} \end{cases} \\
(\!| \texttt{x = T.Array(n, s)} |\!)_{vars} &= V \text{ avec } V_{\mathbb{T}} = \begin{cases} \{n\} & \text{si } \mathbb{T} = T^s. \\ \emptyset & \text{sinon.} \end{cases} \\
(\!| \texttt{x \%= v} |\!)_{inst} &= \boldsymbol{\emptyset} \\
(\!| \texttt{x[i] \%= v} |\!)_{inst} &= \boldsymbol{\emptyset} \\
(\!| \texttt{f()} |\!)_{inst} &= \boldsymbol{\emptyset} \\
(\!| \texttt{with If(b): l} |\!)_{inst} &= (\!| \texttt{l} |\!)_{list} \\
(\!| \texttt{with If(b): l1 with Else(): l2} |\!)_{inst} &= (\!| \texttt{l1} |\!)_{list} \sqcup (\!| \texttt{l2} |\!)_{list} \\
(\!| \texttt{with For(d, f, n) as i: l} |\!)_{inst} &= (\!| \texttt{l} |\!)_{list} \sqcup V \text{ avec } V_{\mathbb{T}} = \emptyset \text{ et } V_{\mathbb{Z}} = \{\texttt{i}\}
\end{aligned}
$$

FIGURE 3.3 – Sémantique du coeur impératif de PYSIL.

Création de la syntaxe et de la sémantique pySil des définitions

En PYSIL, on doit pouvoir déclarer des fonctions, des variables et des tableaux. À ces trois objets se rajoute un quatrième : le programme lui-même définit dans son propre fichier : `program.py`.

Programme Commençons par donner la syntaxe des programmes :

```
<program> :=
    with Program(<name>) {as <ident>}:
        LIST(<fun_def>)
```

On utilise donc l'instruction **with** comme présenté précédemment.

La sémantique du programme est composée de deux éléments. D'une part, l'état du programme représenté par l'ensemble des variables et leur type, et d'autre part un ensemble de fonctions renvoyant un nouvel état à partir d'un état donné :

Définition 3.2.6 (Sémantique d'un programme) *On définit la sémantique* $[\![\cdot]\!] : \mathbf{P} \to \{(V, \gamma) \mid V \in \mathbf{V} \ \wedge \ \gamma \in \Gamma^{\Sigma^V}\}$ *d'un programme par* $[\![\texttt{with Program(n): lines}]\!] = (V, [\![\texttt{lines}]\!]^V_{list} \bot)$ *avec* $V = (\!| \texttt{lines} |\!)_{list}$

$\bot$ *est défini dans la définition 3.2.4.*

Fonction On continue avec la syntaxe des fonctions :

```
<fun_def> :=
    with Function(<name>) as <ident>:
        LIST(<stmt>)
```

Le choix a été fait de ne pas mettre d'argument aux fonctions. Ceci a pour but de simplifier le code compilé. En effet, toutes les variables seront globales. Cela permet de connaître, à la compilation, l'ensemble des variables, et donc la mémoire qu'utilisera le programme. Ceci sera aussi avantageux pour la preuve, car la mémoire est plus simple.

On peut définir une fonction dans une autre fonction, cela n'a pas de signification particulière et donc cela est strictement équivalent à les définir en séquence. C'est pourquoi cette possibilité sera ignorée dans la sémantique. On pourra supposer l'existence d'une étape de transformation les plaçant avant la fonction où elles sont définies.

Définir une fonction en PYSIL entraine le rajout de sa sémantique à l'environnement courant. La sémantique de la fonction elle-même est celle de sa liste d'instruction et sera donc une fonction transformant un état du programme en un autre.

Définition 3.2.7 (Sémantique d'une fonction) *Soit* $\mathbf{F}$*, l'ensemble des programmes que l'on peut écrire en* PYSIL *:* **<fun_def>**.

On définit la sémantique $[\![\cdot]\!]^V_{fun} : \mathbf{F} \to (\Gamma_{\Sigma^V} \to \Gamma_{\Sigma^V})$ *d'une fonction par*

$$[\![\texttt{with Function(n) as x: lines}]\!]^V_{fun}(\gamma) = \gamma \uplus \{x \mapsto [\![\texttt{lines}]\!]^V_{list}(\gamma)\}$$

On définit aussi $(\!| \cdot |\!)_{fun} : \mathbf{F} \to \mathbf{V}$ *par* $(\!| \texttt{with f: lines} |\!)_{fun} = (\!| \texttt{lines} |\!)_{list}$.

On peut remarquer que le nom de la fonction est ignoré. En effet, il est utilisé uniquement pour rendre lisible le code qui sera généré dans le chapitre 5. Il en va de même pour la définition des variables et des tableaux.

Variable Rajout des variables à PYSIL :

```
<var_def> :=
    <ident:T> = <type:T>.Var(<name>)
```

Les variables sont toujours globales. Elles peuvent être définies n'importe où dans le programme, cela ne changera en rien leurs sémantiques. Ce choix a été fait pour la même raison que les arguments de fonction.

Définition 3.2.8 (Sémantique de la définition des variables) *La définition d'une variable n'apporte rien du point de vue de la sémantique :* $[\![\texttt{x = T.Var(n)}]\!]^V_{vars} = \boldsymbol{Id}_{\Sigma^V}$.

En revanche, c'est le cœur de l'extraction de l'état du programme : $(\!| \texttt{x = T.Var(n)} |\!)_{vars} = V$ *avec* V *définit par* $V_{\mathbb{T}} = \begin{cases} \{x\} & si\ \mathbb{T} = T. \\ \emptyset & sinon. \end{cases}$ *pour tout* $\mathbb{T} \in \mathbf{T}$

On introduit aussi deux opérateurs :

— $\mathcal{T}^V : \mathbf{I} \to \mathbf{T}$ *qui associe à une variable ou un tableau son type de base.*

— $\mathcal{S}^V : \mathbf{I} \to \mathbb{N}$ *qui associe à un tableau sa taille.*

Tableau Il reste à introduire les tableaux pour finir les définitions :

```
<array_def> :=
    {<ident:T> =} <type:T>.Array(<name>, <integer>)
```

Le second argument correspond à la taille du tableau, il possède donc une taille fixe. Les mêmes remarques que sur les variables sont applicables aux tableaux, ils sont globaux et appartiennent implicitement à une fonction. La définition de la sémantique est elle aussi similaire :

Définition 3.2.9 (Sémantique de la définition des tableaux)

$[\![\texttt{x = T.Array(n, s)}]\!]^V_{vars} = \boldsymbol{Id}_{\Sigma^V}$.

$(\!| \texttt{x = T.Array(n, s)} |\!)_{vars} = V$ *avec* V *définit par* $V_{\mathbb{T}} = \begin{cases} \{n\} & si\ \mathbb{T} = T^s. \\ \emptyset & sinon. \end{cases}$ *pour tout* $\mathbb{T} \in \mathbf{T}$

On peut remarquer qu'il ne peut y avoir de création de mémoire dynamique. Quoiqu'il arrive, toute la mémoire sera connue à la compilation. Ceci est un choix motivé à la fois par l'aspect preuve et par la problématique de l'embarqué. Concernant la preuve, la modélisation de la mémoire devient beaucoup plus simple si elle n'est pas dynamique. Concernant l'embarqué, la mémoire vive est souvent limitée et donc connaitre très précisément les besoins du programme à l'avance est avantageux et garantit qu'il n'y aura pas d'erreur à cause d'un manque de mémoire vive.

Il reste à introduire l'unité syntaxique **<definition>** déjà utilisée qui représente une des trois définitions que nous venons d'introduire :

```
<definition> := <fun_def> | <var_def> | <array_def>
```

Création de la syntaxe et de la sémantique pySil des instructions

Dans cette section, nous allons définir la syntaxe **<stmt>** des instructions.

La sémantique d'une instruction est une fonction associant à un environnement et un état du système un nouvel état du système. Une instruction est présente uniquement dans une fonction. De plus, nous avons supposé qu'il n'y avait pas de fonctions définie dans une autre

fonction. Ainsi, l'environnement, qui correspond en quelque sorte à la liste des fonctions, ne peut pas être modifié au sein d'une fonction et donc n'est présent qu'en argument.

Définition 3.2.10 (Types pour la sémantique des instructions) *Soit* $\mathbf{S}$, *l'ensemble des instructions que l'on peut écrire en* PYSIL *:* **<stmt>**.

$$[\![\cdot]\!]^V_{inst} : \mathbf{S} \to \Gamma^{\Sigma^V} \to \Sigma^V \to \Sigma^V$$
$$(\!| \cdot |\!)_{inst} : \mathbf{S} \to \mathbf{V}$$

Affectation En PYSIL on doit pouvoir affecter une valeur à une variable ainsi qu'un emplacement d'un tableau. Pour cela, on utilise l'opérateur `%=`, cet opérateur à l'origine est utilisé pour effectuer un modulo sur une variable, mais nous allons l'utiliser pour une simple affectation en le surchargeant là ou cela sera nécessaire.

```
<assign_stmt> :=
    <left:T> %= <expr:T>
```

La partie gauche de l'affectation peut donc être soit une variable, soit un élément d'un tableau :

```
<left:T> := <ident:T> | <ident:T>[<integer>]
```

L'accès au tableau se fait de manière naturelle avec l'opérateur python `[ ]`.

Définition 3.2.11 (Sémantique de l'affectation) *La sémantique de l'affectation consiste à modifier la valeur d'un des états de la mémoire :*

Soit $\gamma \in \Gamma^{\Sigma^V}$ $\mathbb{T} \in \mathbf{T}$, $y \in V_{\mathbb{T}}$,

$[\![\texttt{x \%= v}]\!]^V_{inst}$

$$[\![\texttt{x \%= v}]\!]^V_{inst}(\gamma, \sigma)(y) = \begin{cases} v & si\ x = y\ et\ \mathcal{T}^V(x) = \mathbb{T} \\ \sigma_{\mathbb{T}}(y) & sinon \end{cases}$$

$$[\![\texttt{x[i] \%= v}]\!]^V_{inst} instV(\gamma, \sigma)(y) = \begin{cases} \sigma_{\mathbb{T}}(y)[i \leftarrow v] & si\ x = y\ et\ \mathcal{T}^V(x)^{\mathcal{S}^V(x)} = \mathbb{T} \\ \sigma_{\mathbb{T}}(y) & sinon \end{cases} \quad \forall y \in V_T, \mathbb{T} \in \mathbf{T}$$

$(\!|$ `x %= v` $|\!)_{inst} = \emptyset$

$(\!|$ `x[i] %= v` $|\!)_{inst} = \emptyset$

Appel de fonction Comme on l'a vu plus tôt les fonctions n'ont pas d'arguments, ainsi l'appel se fait très simplement.

```
<call_stmt> := <ident>()
```

Si l'on souhaite transmettre une donnée à une fonction on doit passer par des variables globales.

Par exemple, on veut écrire une fonction f prenant deux entiers a, b en argument et retourne un entier $r = f(a, b)$. Pour cela, on définit trois variables globales `a`, `b` et `r` qui représenterons les arguments et la valeur de retour. Comme la portée des variables est globale, on peut accéder à ces variables sans problème dans la fonction qui appellera f :

```
1 with Function("f") as compute_f:
2     a = Int.Var('a')
3     b = Int.Var('b')
4     r = Int.Var('r')
```

```
    # Calcul de r = f(a, b)

with Function("main1"):
    c = Int.Var('c')
    a %= 3
    b %= 5
    compute_f()
    c %= r
    #c contient désormais f(3, 5)
```

Si l'on veut pouvoir écrire directement dans le code, $f(3,5)$ il est aisé grâce à l'utilisation du langage python de créer le sucre syntaxique pour pouvoir le faire :

```
with Function("f") as compute_f:
    a = Int.Var('a')
    b = Int.Var('b')
    r = Int.Var('r')
    # Calcul de r = f(a, b)

def f(a_val, b_val):
    a %= a_val
    b %= b_val
    compute_f()
    return r

with Function("main2"):
    c = Int.Var('c')
    c %= f(3, 5)
    #c contient désormais f(3, 5)
```

Définition 3.2.12 (Sémantique des appels de fonctions) *La définition de cette sémantique est rendue simple par l'absence d'argument :* $[\![\texttt{f()}]\!]^V_{inst}(\gamma,\sigma) = \gamma(f)(\sigma)$ *et* $(\!| \texttt{f()} |\!)_{inst} = \emptyset$

Structure de controle Nous utilisons l'opérateur `with` pour déclarer les structures de contrôle :

```
<if_stmt> :=
    with If(<expr:Bool>):
        LIST(<stmt>)
    {with Else():
        LIST(<stmt>)}
```

On se contente en PYSIL de définir les boucles *for*[2] :

```
<for_stmt> :=
    with For(<expr:Int>, <expr:Int>, <name>) as <ident>:
        LIST(<stmt>)
```

L'identifiant correspond à la variable de boucle et le troisième argument du `For` permet de la nommer.

Définition 3.2.13 (Sémantique des structures de contrôle)

$$[\![\texttt{with If(b): lines}]\!]^V_{inst}(\gamma,\sigma) = \begin{cases} [\![\texttt{lines}]\!]^V_{list}(\gamma,\sigma) & si\ [\![\texttt{b}]\!]^V_{expr}(\sigma) = \boldsymbol{V} \\ \sigma & sinon \end{cases}$$

2. on pourrait aisément déclarer une boucle *while* si un besoin se crée

$$[\![\texttt{with If(b): lines1 with Else(): lines2}]\!]^V_{inst}(\gamma,\sigma) = \begin{cases} [\![\texttt{lines1}]\!]^V_{inst}(\gamma,\sigma) & si\ [\![\texttt{b}]\!]^V_{expr}(\sigma) = \boldsymbol{V} \\ [\![\texttt{lines2}]\!]^V_{inst}(\gamma,\sigma) & sinon \end{cases}$$

$$[\![\texttt{with For(d, f) as i: l}]\!]^V_{inst}(\gamma,\sigma) = \begin{cases} [\![\texttt{l}]\!]^V_{inst}(\gamma,\sigma) & si\ [\![\texttt{i < f}]\!]^V_{expr}(\sigma) = \boldsymbol{V} \\ \sigma & sinon \end{cases}$$

$$(\!| \texttt{with If(b): lines} |\!)_{inst} = (\!| \texttt{lines} |\!)_{list}$$

$$(\!| \texttt{with If(b): lines1 with Else(): lines2} |\!)_{inst} = (\!| \texttt{lines1} |\!)_{list} \sqcup (\!| \texttt{lines2} |\!)_{list}$$

$$(\!| \texttt{with For(d, f, n) as i: lines} |\!)_{inst} = (\!| \texttt{lines} |\!)_{list} \sqcup V \text{ avec } V \text{ définit par } V_{\mathbb{T}} = \emptyset \text{ pour } \mathbb{T} \in \mathbf{T} \setminus \{\mathbb{Z}\} \text{ et } V_{\mathbb{Z}} = \{i\}$$

```
<stmt> := <assign_stmt> | <call_stmt> | <if_stmt>
                        | <for_stmt> | <definition>
```

Création de la syntaxe pySil des expressions

Expression. Une expression est constituée d'une variable, d'un accès tableau, d'une constante ou d'une opération. Notons que l'on peut utiliser directement des valeurs python à la place de la syntaxe des constantes. Les exemples que nous proposons à la fin de cette section utilisent ce sucre syntaxique lors du remplissage des matrices.

```
<expr> := <left> | <type>.Cst(<value>)
    | <unary> <expr> | <expr> <cmp> <expr>
    | <expr> <binary> <expr>
```

Opérateurs On retrouve en PYSIL les opérateurs classiques, la plupart parlent d'eux même, précisons juste que ~ représente la négation booléenne. Les opérateurs seront acceptés par le compilateur uniquement lorsqu'ils ont du sens vis-à-vis de leur type. Ceci est particulièrement vrai pour le type booléen.

```
<cmp> := > | >= | < | <= | == | !=
<unary> := - | ~
<binary> := + | - | * | /
```

Enfin, nous avons doté PYSIL de 3 types de bases : flottant, entier, booléen, qui sont définis dans `types.py`.

```
<type> := Real | Int | Bool
```

Définition 3.2.14 (Sémantique des expressions) *Soit* $\mathbf{E}_T$*, l'ensemble des expressions que l'on peut écrire en* PYSIL *:* `<expression:T>`*.*

La sémantique des expressions consiste à associer un état du programme à une valeur mathématique : $[\![\cdot]\!]^V_{\mathbb{T}} : \mathbf{E}_{\mathbb{T}} \to \left(\Sigma^V \to \mathbb{T}\right)$ *défini par :*

- $[\![\texttt{x}]\!]^V_{\mathbb{T}}(\sigma) = \sigma(x)$
- $[\![\texttt{x[i]}]\!]^V_{\mathbb{T}}(\sigma) = \sigma(x)_i$
- $[\![\texttt{T.Cst(v)}]\!]^V_{\mathbb{T}}(\sigma) = [\![\texttt{v}]\!]$
- $[\![\texttt{f(a\_0, ..., a\_n)}]\!]^V_{\mathbb{T}}(\sigma) = [\![\texttt{f}]\!]([\![\texttt{a\_0}]\!]^V_{\mathbb{T}}(\sigma), ..., [\![\texttt{a\_n}]\!]^V_{\mathbb{T}}(\sigma))$

  f regroupe les opérateurs préfixes `<unary>` *mais aussi infixe* `<cmp>` *et* `<binary>`*.* $[\![\texttt{f}]\!]$ *est la version mathématique de l'opérateur* `f`*, les correspondances sont naturelles. De même,* $[\![\texttt{v}]\!]$ *représente la version mathématique de la valeur* `v`*.*

Exemples d'utilisation de pySil

Nous allons présenter deux petits exemples de programme écrit en PYSIL :

Fibonacci Le premier exemple est une fonction remplissant un tableau avec les éléments de la suite de Fibonacci :

```
N = 10

with Program("fibonacci"):
    with Function("compute_fib") as fib:
        ar = Int.Array("ar", N)
        ar[0] %= 1
        ar[1] %= 1
        with For(0, N - 2) as i:
            ar[i + 2] = ar[i] + ar[i + 1]
```

Pour utiliser cette fonction, il suffit d'appeler la fonction : `fib()` et on obtient le résultat dans le tableau `ar`. `N`, le nombre d'éléments de la suite de Fibonacci que l'on doit calculer est fixé à l'avance. On aurait pu obtenir cette valeur à travers une variable assignée à l'exécution, mais celle-ci aurait été bornée par la taille du tableau qui est nécessairement fixée à l'avance.

Multiplication de matrice. Le second exemple est plus proche de ce que l'on souhaite exprimer en PYSIL : une multiplication de matrice. À nouveau, la taille des matrices est fixée à l'avance.

```
N = 2

with Program("matrix") as p:
    with Function("mat_mul") as mat_mul:
        A = Real.Array("A", N*N)
        B = Real.Array("B", N*N)
        R = Real.Array("R", N*N)
        x = Real.Var("x")
        with For(0, N) as i:
            with For(0, N) as j:
                R[i*N+j] %= 0
                with For(0, N) as k:
                    R[i*N+j] %= R[i*N+j] + A[i*N + k] * B[k*N + j]

    with Function("main"):
        A[0] %= 1
        A[1] %= 3
        A[2] %= 5
        A[3] %= 7
        B_val = [2, 4, 6, 8]
        for i in range(N*N):
            B[i] %= B_val[i]
        mat_mul()
```

La fonction `mat_mul` calcule la multiplication des matrices contenues dans `A` et `B` et place le résultat dans `R`.

La fonction `main` remplit les deux tableaux `A` et `B` puis appelle la fonction `mat_mul`. À la fin de la fonction, on a donc `R` qui contient leur multiplication :

$$R = \begin{pmatrix} 1 & 3 \\ 5 & 7 \end{pmatrix} \times \begin{pmatrix} 2 & 4 \\ 6 & 8 \end{pmatrix} = \begin{pmatrix} 20 & 28 \\ 52 & 76 \end{pmatrix} \tag{3.4}$$

```
<quantifier> := forall | exists
<expr_quant> := <quantifier>(<type>, ... , lambda <ident>, ... : <expr>)
<expr_old> := old(<ident>)
<expr_impl> := <expr:Bool>.implies(<expr:Bool>)
<expr> := <expr> | <expr_old> | <expr_quant> | <expr_impl>

<req_stmt> := <ident>.addRequire(<expr:Bool>)
<ens_stmt> := <ident>.addEnsure(<expr:Bool>)
<inv_stmt> := addInvariant(<expr:Bool>)
<assert_stmt> := Assert(<expr:Bool>)
<dep_stmt> := <ident>.addDep(<ident>)
<stmt> := <stmt> | <req_stmt> | <ens_stmt> | <inv_stmt>
                 | <assert_stmt> | <dep_stmt>

<type_def> := <ident> = Type(<name>)
<op_def> :=
    <ident> = Operator(
        <name>,
        <type:T>,
        (<type>, ...){,
        lambda <ident>, ... : <expr:T>})
<ax_def> := <ident> = Axiom(<name>, <expr:Bool>)
<lem_def> := <ident> = Lemma(<name>, <expr:Bool>)

<axiomatic> :=
    with Axiomatic(<name>):
        LIST(<type_def> | <ax_def> | <op_def> | <lem_def> )

<definition> := <definition> | <lem_def> | <op_def> | <axiomatic>
```

FIGURE 3.4 – Grammaire de la logique de Hoare de PYSIL.

On a utilisé deux méthodes pour remplir les tableaux. La première consiste à affecter chaque valeur une par une. La seconde utilise python pour le faire de manière bien plus facile. Ceci permet de voir l'intérêt d'avoir un langage embarqué dans python. Cela serait encore plus intéressant si le tableau `b_val` était issu d'un calcul numérique effectué en python.

Nous avons dans cette section présenté une sémantique dénotationelle de notre langage PYSIL. Cette sémantique ne permet pas d'aborder tout ce qu'il est possible de faire grâce à l'utilisation de python comme métalangage. Pour cela, on aurait pu plonger la sémantique proposée ici dans une sémantique du langage python, mais cela représentait un travail trop conséquent pour cette thèse.

3.2.4 Rajout du support de la logique de Hoare

Nous avons créé un langage permettant d'exprimer des programmes impératifs. Nous allons maintenant l'étendre pour supporter nativement la possibilité de l'annoter par la logique de Hoare. Ces nouveaux éléments sont incorporés dans les fichiers précédents quand ils leur sont directement liés. Pour les autres, on peut les trouver dans `hoare.py` pour la création du contrat de fonction et dans `logic.py` pour ce qui concerne la création de nouveaux objets logiques comme les opérateurs ou les lemmes. La syntaxe de la logique de Hoare est rappelée en totalité dans la figure 3.4.

Enrichissement des instructions avec la logique de Hoare

Triplet de Hoare Nous commençons par la possibilité de définir un triplet de Hoare :

```
<req_stmt> := <ident>.addRequire(<expr:Bool>)
<ens_stmt> := <ident>.addEnsure(<expr:Bool>)
```

L'identifiant devra être celui d'une fonction et l'expression une expression booléenne comme présentée dans la section précédente. Nous enrichirons, par la suite, la grammaire des expressions pour supporter quelques constructions logiques comme les quantificateurs. Les expressions peuvent faire référence aux variables et aux tableaux définis dans le programme.

Invariant La seconde construction de la logique de Hoare nécessaire à l'annotation complète d'un programme en logique de Hoare est l'invariant de boucle :

```
<inv_stmt> := addInvariant(<expr:Bool>)
```

Cette fonction ne peut être appelée que dans le contexte d'une boucle.

Assertion Enfin, nous incorporons la possibilité de rajouter des assertions à notre code. Elles peuvent être rajoutées n'importe où dans une fonction.

```
<assert_stmt> := Assert(<expr:Bool>)
```

Elles permettent de spécifier des points intermédiaires au cœur d'une fonction, c'est un équivalent du lemme en mathématique.

On rajoute ces nouvelles instructions à celle précédemment définie :

```
<stmt> := <stmt> | <req_stmt> | <ens_stmt> | <inv_stmt> | <assert_stmt>
```

Enrichissement des expressions

Il faut maintenant enrichir légèrement notre langage d'expression, en particulier les expressions booléennes avec des constructions propres à la logique.

Quantificateur. On commence par rajouter le support de la logique du premier ordre en incorporant la quantification :

```
<quantifier> := forall | exists
<expr_quant> := <quantifier>(<type>, ... , lambda <ident>, ... : <expr>)
```

Ainsi pour exprimer une formule F quantifiée existentiellement par n variables $x_1, ...x_n$ de type $T_1, ...T_n$: $\exists x_1 \in T_1, ...x_n \in T_n, F(x_1, ...x_n)$, on pourra écrire :

```
1 exists(T_1, ... , T_n,  lambda x_1, ... , x_n : F)
```

On utilise l'opérateur lambda de python afin de représenter les variables liées. On peut aussi décider de définir `G = lambda x_1, ... , x_n : F` avant dans le programme puis d'écrire simplement `exists(T_1, ... , T_n, G)`.

Référencer l'ancienne mémoire. Il est parfois nécessaire dans la logique de Hoare de faire référence à l'état de la mémoire juste avant l'appel à la fonction. Pour cela, on peut utiliser l'opérateur suivant :

```
<expr_old> := old(<ident>)
```

L'identifiant sera une variable ou un tableau.

Implication. Il faut aussi un nouvel opérateur représentant l'implication. Nous n'avons trouvé aucun opérateur de python pouvant représenter naturellement l'implication, donc nous utilisons une méthode :

```
<expr_impl> := <expr:Bool>.implies(<expr:Bool>)
```

Il reste à enrichir l'unité syntaxique des expressions avec ces nouveaux éléments :

```
<expr> := <expr> | <expr_old> | <expr_quant> | <expr_impl>
```

Enrichissement des définitions

Définition de nouveau type Nous avons un langage capable d'exprimer à la fois un programme impératif, mais aussi de l'annoter avec la logique de Hoare. Pour l'instant, cette logique est contrainte à utiliser les mêmes opérateurs que le programme. Cependant, il peut parfois être intéressant d'avoir des opérateurs propres à la logique comme le `old` que nous avons défini. Nous pourrions ainsi enrichir notre langage avec tous les opérateurs que l'on pensera nécessaire. Cependant, nous ne pourrons jamais être exhaustifs et nous souhaitons avoir un cœur le plus minimal possible. Ainsi nous allons enrichir notre langage avec la possibilité de définir de nouveaux types et opérateurs purement logique :

```
<type_def> :=
    <ident> = Type(<name>)
```

Pour déclarer un type, on se contente de lui donner un nom, il sera défini implicitement par les opérateurs le référençant.

Opérateur Un opérateur se définit en lui donnant un nom, son type de retour, un n-uplet contenant le type de ses arguments et enfin un lambda représentant l'opérateur lui-même.

```
<op_def> :=
    <ident> = Operator(
        <name>,
        <type:T>,
        (<type>, ...){,
        lambda <ident>, ... : <expr:T>})
```

On peut aussi déclarer des opérateurs implicites en ne donnant pas le quatrième argument. Afin que ces opérateurs puissent avoir un sens, il faut pouvoir les définir axiomatiquement.

Axiome et Lemme Nous avons aussi ajouté à PYSIL la possibilité d'exprimer des résultats mathématiques comme des lemmes ou des axiomes :

```
<ax_def> :=
    <ident> = Axiom(<name>, <expr:Bool>)
<lem_def> :=
    <ident> = Lemma(<name>, <expr:Bool>)
```

On appelle lemmes tous les résultats mathématiques qui nécessitent d'être prouvés, les axiomes eux sont supposés corrects. Les axiomes sont utilisés dans leur rôle mathématique

original d'axiomes, c'est-à-dire pour définir de nouveaux opérateurs principalement. On peut aussi les utiliser pour exprimer un résultat qui pourrait être démontré, mais dont la preuve est trop complexe.

Dans les deux cas, il faut être extrêmement attentif lors de la définition d'un axiome, car ils peuvent rendre la logique incohérente.

Afin de faciliter le travail de preuve, PYSIL contient aussi une manière de spécifier manuellement quels résultats sont nécessaires à la preuve d'un résultat mathématique ou d'un contrat :

```
<dep_stmt> :=
    <ident>.addDep(<ident>)
<stmt> :=
    <stmt> | <dep_stmt>
```

Le premier identifiant peut être un lemme ou une fonction, quand au second il peut être soit un lemme, soit un axiome. La ligne `f.addDep(a)` signifie que pour prouver `f` on a besoin de `a`.

Axiomatique. Les axiomatiques sont des environnements dans lequel doivent être définis les types, les axiomes et les opérateurs implicites.

```
<axiomatic> :=
    with Axiomatic(<name>):
        LIST(<type_def> | <ax_def> | <op_def> | <lem_def> )
```

On enrichit ainsi nos définitions avec uniquement les lemmes, les opérateurs (explicites) et les axiomatiques.

```
<definition> := <definition> | <lem_def> | <op_def> | <axiomatic>
```

Exemples d'utilisation de la logique de Hoare en pySil

Reprenons l'exemple de Fibonacci :

```
1  with Program("fibonacci") as p:
2      with Axiomatic("Fibonacci"):
3          fib = Operator("fib", Int, (Int, ))
4          Axiom("fib_0", fib(0) == 1)
5          Axiom("fib_1", fib(1) == 1)
6          Axiom("fib_n", forall(Int, lambda n:
7              (n >= 2).implies(
8                  fib(n) == fib(n-1) + fib(n-2))))
9
10     with Function("compute_fib") as compute_fib:
11         compute_fib.addEnsure(forall(Int, lambda x:
12             ((0 <= x) & (x < N)).implies(
13                 ar[x] == fib(x))))
14         ar = Int.Array("ar", N)
15         ar[0] %= 1
16         ar[1] %= 1
17         with For(0, N - 2) as i:
18             addInvariant(forall(Int, lambda x:
19                 ((0 <= x) & (x < i+2)).implies(
20                     ar[x] == fib(x))))
21             ar[i + 2] %= ar[i] + ar[i + 1]
```

On commence par définir l'opérateur `fib` de manière axiomatique. On déclare l'opérateur de manière implicite puis on rajoute 3 axiomes, deux pour les cas initiaux puis le cas récursif.

Le code reste le même, mais on rajoute une postcondition à la fonction qui garanti que `ar` contient la suite de Fibonacci. On rajoute aussi un invariant à la boucle **`For`** qui garantit que jusqu'à $i + 2$ le tableau est rempli de la suite de Fibonacci.

3.2.5 Bilan

Ainsi nous avons développé le cœur impératif de PYSIL, à la fois la partie code et la partie spécification à travers la logique de Hoare. On a présenté ici seulement les grandes lignes, il existe de nombreux détails d'implémentation qui permettent la génération d'un AST à la volée, de la vérification d'erreur ainsi qu'un certain nombre de fonctionnalités permettant l'enrichissement du langage. C'est particulièrement le cas de la création des types de bases qui se résume à quelques lignes et permet donc de facilement rajouter de nouveaux types. En effet, le cœur est simple et petit, mais est voué à être étendu. Cependant, nous souhaitons que toutes les extensions de ce cœur de base soient replongées dans le cœur lui-même. Ceci afin de rendre plus aisé le travail de compilation et garder un aspect modulaire simplifiant le développement. C'est le cas de l'extension matricielle qui est définie dans la section suivante.

3.3 Extension du langage pour supporter les matrices

Nous allons, dans cette partie, ajouter à PYSIL le support des matrices afin de pouvoir exprimer de manière naturelle des algorithmes matriciels. On doit à la fois supporter les matrices dans le code, mais aussi dans les annotations. Par exemple, on doit pouvoir exprimer un calcul comme P^tAP aussi simplement que :

```
1  B %= P.T() * A * P
```

On souhaite pouvoir faire de même pour les annotations. Par exemple l'appartenance à l'ensemble faisable devrait se traduire par :

```
1  f.addEnsure(A * x < b)
```

On pourra trouver les fichiers concernant la définition des matrices dans : `pySil/matrix/`. La figure 3.5 rassemble la syntaxe de l'extension matricielle.

3.3.1 Rajout des opérations matricielles

Il faut un nouveau type pour représenter les matrices. Un type est créé pour chaque taille de matrice. Ceci permet de vérifier les compatibilités entre les tailles lors des opérations matricielles.

```
<type> := <type> | LMat | Mat(<integer>, <integer>)
<expr> := <expr> |
    <ident:Mat(M, N)>[<integer>, <integer>] |
    <ident:Mat(M, N)>.T() |
    <type:Mat(M, N)>.fromArray(<array>) |
    <type:Mat(M, N)>.identity() |
    <type:Mat(M, N)>.uniform(<expr:Real>)
```

FIGURE 3.5 – Grammaire de l'extension matricielle de PYSIL.

```
<type> := <type> | Mat(<integer>, <integer>)
```

Concernant la sémantique on étend l'ensemble des types :

Définition 3.3.1 (Ensemble étendu) *On rajoute à l'ensemble des types les matrices.* $\widetilde{\mathbf{T}} = \mathbf{T} \cup \{\mathcal{M}_{m,n}(\mathbb{R}) \mid m, n \in \mathbb{N}\}$. *On construit ensuite* $\mathbf{V}'$ *en remplaçant* $\mathbf{T}$ *par* $\widetilde{\mathbf{T}}$ *dans la définition de* $\mathbf{V}$. *De manière générale, pour chaque ensemble* E *défini dans la section précédente, on construit* $\widetilde{E}$ *qui est le prolongement de* E *à la syntaxe matricielle.*

Les matrices sont composées exclusivement de nombres réels. On peut récupérer un de ses éléments grâce l'opérateur python dédié :

```
<expr> := <expr> | <ident:Mat(M, N)>[<integer>, <integer>]
```

Définition 3.3.2 (Sémantique de l'accès à un élément) $[\![\texttt{x[i, j]}]\!](\sigma) = [m]_{k,l}$ *avec* $m = [\![\texttt{x[i, j]}]\!](\sigma)$, $k = [\![\texttt{i}]\!](\sigma)$ *et* $l = [\![\texttt{j}]\!](\sigma)$.

On peut utiliser les symboles d'addition, soustraction, multiplication et d'égalité. On peut aussi utiliser les symboles de comparaison avec la sémantique suivante $A < B$ si et seulement si chaque élément de la matrice A est plus petit que l'élément correspondant dans B. On rajoute aussi l'opérateur de transposition grâce une fonction dédiée :

```
<expr> := <expr> | <ident:Mat(M, N)>.T()
```

Définition 3.3.3 (Sémantique des opérations) *Elle ne change pas par rapport à la définition de la section précédente, par exemple pour la transposition :*
$[\![\texttt{x.T()}]\!](\sigma) = [\![\texttt{x}]\!](\sigma)^T$.

Pour créer une variable matricielle, nous avons utilisé la même syntaxe que pour les autres variables. Pour créer une matrice constante en revanche il faut spécifier toutes ces valeurs. Nous avons donc nommé l'opérateur correspondant **fromArray**, car il prend un argument un tableau de tableau de nombres réels écrit dans la syntaxe python. Nous avons aussi rajouté la possibilité de créer des matrices identité ainsi que des matrices uniformes qui sont les matrices $\begin{pmatrix} a & \dots & a \\ \vdots & \ddots & \vdots \\ a & \dots & a \end{pmatrix}$ avec $a \in \mathbb{R}$.

```
<expr> := <expr> |
    <type:Mat(M, N)>.fromArray(<array>) |
    <type:Mat(M, N)>.identity() |
    <type:Mat(M, N)>.uniform(<expr:Real>)
```

L'unité syntaxique **<array>** représente une matrice sous forme de tableau de tableau python. **identity** retourne la matrice identité, si les dimensions ne sont pas égales, on construit la matrice identité carrée de la taille la plus petite et on complète avec des zéros. `uniform` retourne une matrice dont tous les éléments sont égaux au réel donné en argument de la fonction.

Définition 3.3.4 (Sémantique des constructeurs de matrices) *La définition des constructeurs de matrices possède une sémantique naturelle :*
$[\![\texttt{Mat(m, n).fromArray(ar)}]\!](\sigma) = A$ *avec* $A \in \mathcal{M}_{m,n}$ *tel que* $\forall i, j \in [\![0, n[\![\times [\![0, m[\![, [A]_{i,j} = ar[i][j]$

```
with Axiomatic("matrix") as ax:
    Type("LMat")

    LMat.get = Operator("get", Real, [LMat, Real, Real])
    LMat.getM = Operator("getM", Int, [LMat])
    LMat.getN = Operator("getN", Int, [LMat])
```

FIGURE 3.6 – Définition du type et des opérateurs de base de l'axiomatique matrice

$[\![$ `Mat(m, n).identity()` $]\!](\sigma) = A$ *avec* $A \in \mathcal{M}_{m,n}$ *tel que* $\forall i, j \in [\![0, n[\![[A] \times [\![0, m[\![, [A]_{i,j} = 1$ *si* $i == j$, 0 *sinon.*

$[\![$ `Mat(m, n).uniform(x)` $]\!](\sigma) = A$ *avec* $A \in \mathcal{M}_{m,n}$ *tel que* $\forall i, j \in [\![0, n[\![[A] \times [\![0, m[\![, [A]_{i,j} = x$

Dans la section précédente, nous avions présenté un code calculant la multiplication de deux matrices, ce code devient désormais très simple à exprimer :

```
with Program("matrix") as p:
    with Function("main"):
        A = Mat(2, 2).fromArray([[1, 3],
                                 [5, 7]])
        B = Mat(2, 2).fromArray([[2, 4,
                                  6, 8]])
        R = A * B
```

3.3.2 Écriture d'une librairie pour les matrices

Nous avons introduit une syntaxe permettant d'écrire des programmes faisant intervenir des matrices. La prochaine étape est de pouvoir faire de même dans les spécifications. Pour cela nous introduisons un nouveau type :

```
<type> := <type> | LMat
```

Il représente les matrices, mais dans le domaine logique. Ce type supporte donc toutes les mêmes opérations que pour le code. Cependant, chacune de ces opérations est reliée directement à un opérateur logique défini dans une axiomatique dédiée.

Définition d'une axiomatique des matrices en pySil En effet, on utilise la syntaxe logique définie à la partie précédente afin de créer axiomatique définissant un nouveau type logique **`LMat`**.

On trouve dans la figure 3.6 les définitions des opérateurs de base : `A.get(i, j)` qui renvoie l'élément à la ligne `i` et à la colonne `j` de la matrice `A`, on peut écrire plus simplement, comme dans le code, `A[i, j]`. `A.getM()` et `A.getN()` renvoient respectivement le nombre de lignes et le nombre de colonnes de la matrice `A`.

Variable matricielle Pour pouvoir parler d'une variable matricielle du code dans notre formalisme logique, il faut pouvoir créer un objet de type **`LMat`** à partir du type **`Mat`**`(M, N)`, c'est pourquoi on introduit l'opérateur `MatVar`.

Dans la figure 3.7 on peut voir que `MatVar` est déclarée implicitement. Il ne peut en être autrement, car il consiste à générer un objet de type LMat qui est lui-même implicite. On déclare ensuite 3 axiomes un pour chacun des opérateurs de base.

```
with ax:
    MatVar = Operator("MatVar", LMat, [Real.Array, Int, Int])
    Axiom("getM_MatVar",
        forall(Real.Array, Int, Int, lambda a, m, n:
            MatVar(a, m, n).getM() == m))
    Axiom("getN_MatVar",
        forall(Real.Array, Int, Int, lambda a, m, n:
            MatVar(a, m, n).getN() == n))
    Axiom("get_MatVar",
        forall(Real.Array, Int, Int, lambda a, m, n:
            forall(Int, Int, lambda i, j:
                ((i >= 0) & (i < m)).implies(
                ((j >= 0) & (j < n)).implies(
                    a[i + j * m]== MatVar(a, m, n)[i, j])))))
```

FIGURE 3.7 – Définition de l'opérateur `MatVar`

```
with ax:
    LMat.T = Operator("T", LMat, [LMat])
    Axiom("getM_T", forall(LMat, lambda A: A.T().getM() == A.getN()))
    Axiom("getN_T", forall(LMat, lambda A: A.T().getN() == A.getM()))
    Axiom("get_T", forall(LMat, lambda A: forall(Int, Int, lambda i, j:
                    A.T()[i, j] == A[j, i])))
```

FIGURE 3.8 – Définition de l'opérateur de transposition logique

Opérations matricielles Cette structure de définition sera la même pour tous les opérateurs définis dans l'axiomatique **`LMat`**. Nous n'allons pas présenter ici l'intégralité de l'axiomatique, mais seulement l'opérateur de transposition dans la figure 3.8 à des fins d'illustration. La définition des autres opérateurs pourra être trouvée dans le fichier `logic.py`.

Remarque sur la différence de typage entre les matrices code et logique Pour la logique, le choix retenu est que toutes les matrices sont du même type, quelque soit leur taille. En effet, dans le cas contraire il aurait fallu une définition de chaque opérateur pour chaque taille de matrice et les lemmes associé à chaque fois. On aurait pu générer tous ces lemmes à la volée et cela n'aurait pas alourdi ce code. Cependant, cela aurait grandement alourdi le code que l'on générera. C'est pourquoi toutes les matrices logiques appartiennent au même type. Cela ne pose pas de problème de correction, car la cohérence entre les différentes tailles sera vérifiée au moment de la preuve.

Une autre solution aurait été d'utiliser du sous-typage, mais cela n'est pas supporté par Frama-C/WP qui sera utilisé dans le chapitre 6 donc nous avons préféré ne pas l'introduire en PYSIL.

3.3.3 Résolution d'équation linéaire

Comme on a pu voir dans 2.2, les algorithmes de points intérieurs effectuent des pas de Newton qui nécessitent de résoudre un système linéaire. Nous avons rajouté cette fonctionalité comme un nouvel opérateur du langage : **`solve`** prenant en argument une matrice A et un vecteur b et qui retourne un x tel que $Ax = b$. On écrira donc :

```
x %= solve(A, b)
```

Cet opérateur est défini dans le fichier `pySil/solver/solver.py`.

3.3.4 Bilan

Nous avons, dans cette partie, ajouté le support des matrices à notre langage. Nous pouvons à la fois écrire des programmes faisant intervenir du calcul matriciel de manière très naturelle, mais aussi les annoter de façon tout aussi naturelle. Ceci représente un certain nombre d'extensions. Ce langage a pour vocation à être compilé vers un autre langage. Ainsi, afin d'éviter de complexifier cette étape, nous allons présenter dans la partie suivante comment plonger cette extension matricielle dans le cœur impératif défini à la section précédente.

3.4 Plongement des matrices dans le coeur impératif

Dans cette section, nous allons exhiber une transformation T prenant en argument un programme dans la syntaxe étendue de la section précédente et renvoyant un programme sémantiquement équivalent dans le coeur impératif défini dans la section 3.2.

Ainsi en implémentant une seule fois la transformation on pourra se contenter de compiler le coeur impératif dans un langage pour pouvoir compiler la syntaxe étendue dans ce langage.

3.4.1 Plongement des matrices dans le coeur impératif

Nous allons définir l'opérateur T en donnant la transcription de tous les nouveaux éléments de syntaxe introduite à la sous-section 3.3.1. Le support des matrices a introduit un nouveau type qui n'est donc plus disponible dans le coeur impératif. Ainsi, pour chaque expression ou instruction faisant intervenir ce type il faut que T transcrive cet élément quand il est appliqué à des matrices. Les éléments de syntaxes en question sont : les définitions de variable, l'instruction d'affectation et toutes les expressions. Dans ces expressions, il ne faut pas oublier les trois nouvelles expressions spécifiques aux matrices.

Définition 3.4.1 (Tranformation vers le coeur impératif) $\mathrm{T} : \left(\tilde{\mathbf{S}} \cup \tilde{\mathbf{E}}\right) \to (\mathbf{S} \cup \mathbf{E})$ *est défini par :*

$\mathrm{T}(x = \texttt{Mat(m, n).Var(s)}) = x = \texttt{Real.Array(s, m*n)}$

$\mathrm{T}(x \texttt{ \%= } y) =$

```
with For(0, m) as i:
    with For(0, n) as j:
        X(i, j) %= Y(i, j)
```

avec $X = \mathrm{T}(x)$ *et* $Y = \mathrm{T}(y)$

$\mathrm{T}(\texttt{x[i, j]}) = \texttt{T(x)(i, j)}$

$\mathrm{T}(\texttt{x.T()})(i,j) = \texttt{T(x)(j, i)}$

$\mathrm{T}(\texttt{Mat(m, n).fromArray(x)(i, j)}) = \texttt{x[i][j]}$

$$\mathrm{T}(\texttt{Mat(m, n).identity()(i, j)}) = \begin{cases} 1 & \text{si } i = j \\ 0 & \text{sinon} \end{cases}$$

$\mathrm{T}(\texttt{Mat(m, n).uniform(x)})(i,j) = x$

$\mathrm{T}(x)(i,j) = \texttt{x[N*i + j]}$ *avec* x *variable de type* `Mat(M, N)`

$\mathrm{T}(x * y)(i,j) = \texttt{X(i, 0) * Y(0, j) + ... + X(i, N) * Y(N, j)}$ *avec* N *le nombre de colonnes de* x *et de lignes de* y, $X = \mathrm{T}(x)$ *et* $Y = \mathrm{T}(y)$.

$\mathrm{T}(x) = x$ *pour tous les autres*

3.4.2 Preuve de préservation de sémantique

Dans cette sous-section nous exposons les moyens qui permettraient de montrer que la transformation T préserve la sémantique. Nous commençons par définir l'équivalence de sémantique entre un programme avec matrice et un programme sans matrice.

Définition 3.4.2 (Équivalence sémantique) *Soit* $p \in \mathbf{P}$ *et* $p' \in \tilde{\mathbf{P}}$, $(V, \gamma) = [\![\ p\ ]\!]$ *et* $(V', \gamma') = [\![\ p'\ ]\!]$.

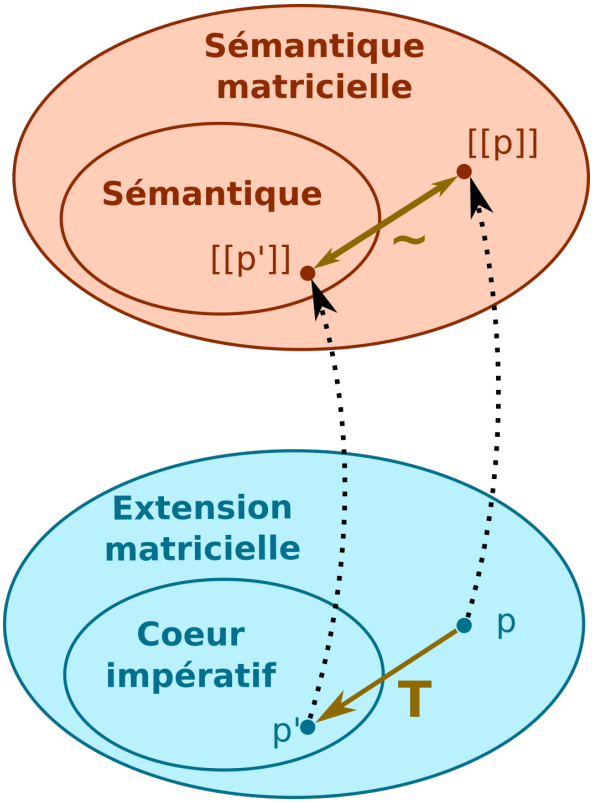


FIGURE 3.9 – Plongement des matrices dans le coeur impératif

Soit $\phi : \widetilde{\mathbf{\Sigma}} \to \mathbf{\Sigma}$, $\mathbb{T} \in \widetilde{\mathbf{T}}$, $v \in V'$, $\sigma \in \Sigma'^{V}$. *Si* $\mathbb{T} \neq \mathcal{M}_{m,n}$ *pour* $m, n \in \mathbb{N}$ *alors* $\phi(\sigma_{\mathbb{T}})(v) = \sigma_{\mathbb{T}}(v)$, *sinon* $\phi(\sigma_{\mathbb{R}^{m\times n}})(v) = \sigma_{\mathcal{M}_{m,n}}(v)$.

$p \sim q$ *si et seulement si* $\forall m, n \in \mathbb{N}$, $\forall x \in V'_{\mathcal{M}_{m,n}}$, $x \in V_{\mathbb{R}^{m\times n}}$ *et* $\forall f \in \gamma'$, $\forall \sigma \in \Sigma'^{V}$, $f(\phi(\sigma')) = \phi(f'(\sigma'))$

Maintenant que nous pouvons comparer la sémantique d'un programme avec matrice et sans matrice, il reste à montrer que la transformation T préserve la sémantique. Cet énoncé est illustrée dans la figure 3.9.

Propriété 3.4.1 (Coïncidence des sémantiques sur le support non matriciel) *Soit* $p \in \mathbf{P}$ *et* $p' \in \widetilde{\mathbf{P}}$, $(V, \gamma) = [\![\ p\ ]\!]$ *et* $(V', \gamma') = [\![\ p'\ ]\!]$ *tel que* $p \sim q$

Théorème 3.4.1 (Préservation de sémantique) *Soit* $p \in \widetilde{\mathbf{P}}$, $p \sim T(p)$

Ce théorème n'est pas encore prouvé mais une preuve par induction sur la syntaxe de p est la piste privilégiée pour le moment.

3.4.3 Solveur

Nous n'avons pas pris en compte l'opérateur `solve` défini dans la sous-section 3.3.3. En effet, celui-ci est directement implémenté comme une fonction en PYSIL donc il serait compliqué de l'intégrer dans notre sémantique. Nous avons implémenté l'algorithme de Cholesky ainsi que la méthode du pivot de Gauss. En pratique, comme `solve` est toujours appellé sur des matrices symétriques définies positives, on utilise toujours l'algorithme de Cholesky. Son implémentation en PYSIL est donnée dans la figure 3.10.

```
def cholesky_reduction(A):
    D = Mat(A.M, A.N).Var("D")
    L = Mat(A.M, A.N).Var("L")
    s = Real.Var("sumcho")
    with For(0, A.M) as j:
        s %= Real.Cst(0)
        with For(0, j) as k:
            s %= s + L[j, k] * L[j, k] * D[k, k]
        D[j,j] %= A[j, j] - s
        with For(j+1, A.M) as i:
            s %= Real.Cst(0)
            with For(0, j) as k:
                s %= s + L[i, k] * L[j, k] * D[k, k]
            L[i, j] %= (A[i, j] - s) / D[j, j]
    return (D, L)

def solveLTriangular(A, b):
    x = Mat(A.N, 1).Var("xLT")
    s = Real.Var("sumLT")
    with For(0, A.M) as i:
        s %= Real.Cst(0)
        with For(0, i) as k:
            s %= s + A[i, k] * x[k, 0]
        x[i, 0] %= (b[i, 0] - s)
    return x

def solveUTriangular(A, b):
    x = Mat(A.N, 1).Var("xUT")
    s = Real.Var("sumUT")
    with For(0, A.M) as i:
        s %= Real.Cst(0)
        with For(A.M-i, A.M) as k:
            s %= s + A[k, A.M-i-1] * x[k, 0]
        x[A.M-i-1, 0] %= (b[A.M-i-1, 0] - s)
    return x

class cholesky(Solver):
    def content(self):
        (m, n) = self.params
        z = Mat(n, 1).Var("z")
        self.A = Mat(m, n).Var("A")
        self.b = Mat(m, 1).Var("b")
        (D, L) = cholesky_reduction(self.A)
        y = solveLTriangular(L, self.b)
        with For(0, z.M) as i:
            z[i, 0] %= y[i, 0] / D[i, i]
        self.x = solveUTriangular(L, z)
        self.addRequire(self.A.defpos())
        self.addEnsure(self.A * self.x == self.b)
```

FIGURE 3.10 – Algorithme de Cholesky en PYSIL

3.4.4 Bilan

Dans cette section nous avons montré comment on pouvait plonger toute l'extension matricielle dans le coeur impératif. Ce plongement est automatisable ce qui veut dire qu'exporter un code pySil avec des matrices est strictement identique à exporter un code pySil sans matrices si on possède l'opérateur T. En pratique, la transformation T se fait à la volée quand le script pySil est exécuté.

Conclusion

Dans ce chapitre, nous avons créé un langage de haut niveau de type pseudo-code. Ce langage est impératif et permet d'exprimer des algorithmes numériques en particulier matriciels. Le langage supporte aussi nativement les annotations par triplet de Hoare y compris le support des matrices dans les annotations. Le langage possède aussi une sémantique ce qui permet de raisonner dessus. Il a aussi été conçu modulaire : il possède un cœur impératif qui peut ensuite être étendu. Le but lors d'une extension est de pouvoir la replonger dans le cœur impératif. Ceci a été en particulier effectué pour l'extension matricielle dont la correction de l'encodage a été prouvée correcte grâce à la sémantique précédemment définie. En bilan, on a un langage simple d'utilisation, haut niveau impératif dans lequel on peut exprimer des algorithmes numériques matriciels, mais aussi ses propriétés. Ainsi on va l'utiliser pour implémenter un algorithme de points intérieurs dans le chapitre suivant.

Bibliographie

ABRIAL, J.-R. (1996). *The B-book : Assigning Programs to Meanings*. New York, NY, USA : Cambridge University Press (cf. p. 45).

APPEL, Andrew W (2011). "Verified software toolchain". In : *European Symposium on Programming*. Springer, p. 1-17 (cf. p. 46).

FILLIÂTRE, Jean-Christophe et Andrei PASKEVICH (2013). "Why3 : Where Programs Meet Provers". In : *Proceedings of the 22Nd European Conference on Programming Languages and Systems*. ESOP'13. Rome, Italy : Springer-Verlag, p. 125-128 (cf. p. 45).

LEINO, Rustan (juin 2008). "This is Boogie 2". In : Microsoft Research (cf. p. 45).

MEURER, Aaron et al. (jan. 2017). "SymPy : symbolic computing in Python". In : *PeerJ Computer Science* 3, e103 (cf. p. 48, 151).

OLIPHANT, Travis E (2006). *A guide to NumPy*. T. 1 (cf. p. 48).

REIF, Wolfgang (1992). "The KIV system : Systematic construction of verified software". In : *International Conference on Automated Deduction*. Springer, p. 753-757 (cf. p. 46).

RIEU-HELFT, Raphaël, Claude MARCHÉ et Guillaume MELQUIOND (juil. 2017). "How to Get an Efficient yet Verified Arbitrary-Precision Integer Library". In : *9th Working Conference on Verified Software : Theories, Tools, and Experiments*. T. 10712. Lecture Notes in Computer Science. Heidelberg, Germany, p. 84-101 (cf. p. 45).

Chapitre 4

Écriture en pySil d'un algorithme de points intérieurs annoté

Sommaire

Introduction

Dans ce chapitre, nous allons montrer comment on peut écrire un algorithme de points intérieurs en pySil. Ce chapitre est une première étape vers la génération automatique d'un code d'optimisation annoté et prouvé pour l'embarqué critique. C'est pourquoi en plus d'écrire l'algorithme nous l'annoterons avec des triplets de Hoare.

Nous commencerons dans la première section par choisir une classe de problème d'optimisation ainsi qu'un algorithme correspondant. Pour cela, nous dresserons un panorama des différents algorithmes d'optimisation convexe avant de choisir le plus adapté à nos besoins.

La seconde section se concentre sur l'écriture de l'algorithme lui-même et surtout l'expression des propriétés qui permettront de garantir sa correction.

Dans la troisième section, nous utiliserons cet algorithme au sein d'un programme spécifique à l'embarqué critique. Pour cela, nous prendrons l'exemple de la commande prédictive et effectuerons quelques changements sur l'algorithme pour l'y adapter.

4.1 Choix de l'algorithme

L'objectif de ce chapitre est d'écrire en pySil un algorithme d'optimisation annoté. Pour cela, nous devons choisir l'algorithme à traduire en pySil. Avant cela, il convient même de choisir la classe de problème qu'il devra résoudre. Ces deux choix seront l'objet de cette section.

4.1.1 Choix du problème

Le but de cette thèse est d'apporter les moyens de valider avec le plus haut niveau de garantie des algorithmes numériques, en particulier d'optimisation. Obtenir ces niveaux de garantie permettra l'utilisation de tels algorithmes dans l'embarqué critique. L'une des classes de problème dont l'utilisation tend à se développer dans le domaine de l´embarqué critique est l'optimisation convexe. Des exemples de telles utilisations peuvent être trouvés dans Mattingley et Boyd, 2009 ou Açikmese, Carson et Blackmore, 2013. Ces algorithmes sont même déjà utilisés dans des systèmes embarqués critiques comme les fusées Falcon 9 de SpaceX pour la partie atterrissage, comme mentionné dans Blackmore, 2016. Notre choix s'est donc porté sur l'optimisation convexe. En fonction de la forme de la fonction de coût et des contraintes, plusieurs classes de problèmes existent. Une des plus petite, l'optimisation LP, et une des plus grande, l'optimisation SDP, ont été abordés dans la section 2.2. Des classes intermédiaires existent. Par example, QPQP dont les contraintes et la fonction de coût sont quadratiques. Ou bien encore SOCP dont le cout est linéaire mais dont la contrainte fait intervenir une norme euclidienne. On peut retrouver dans la figure 4.1 les relations d'inclusion entre ces classes.

L'objectif de cette thèse est avant tout de démontrer la faisabilité d'un tel processus. C'est pourquoi au lieu de chercher à résoudre un problème d'optimisation convexe quelconque nous allons nous limiter à la sous-classe des problèmes d'optimisations LP. Les problèmes seront plus simples et les algorithmes les résolvant aussi et donc la vérification sera moins complexe.

4.1.2 À propos de la programmation linéaire paramétrique

L'article Bemporad, Borrelli et Morari, 2002 présente une méthode pour résoudre des problèmes d'optimisation en ligne qui permettrait d'embarquer un algorithme très simple pour les résoudre : la programmation linéaire paramétrique. Elle repose sur un précalcul des solutions par région du vecteur paramètre. Le vecteur paramètre étant la partie du problème d'optimisation inconnu hors ligne, qui sera apporté par le système durant son fonctionnement.

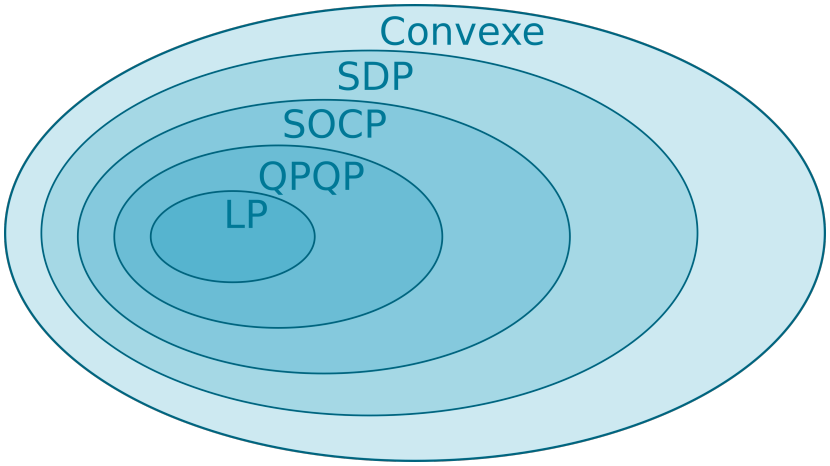
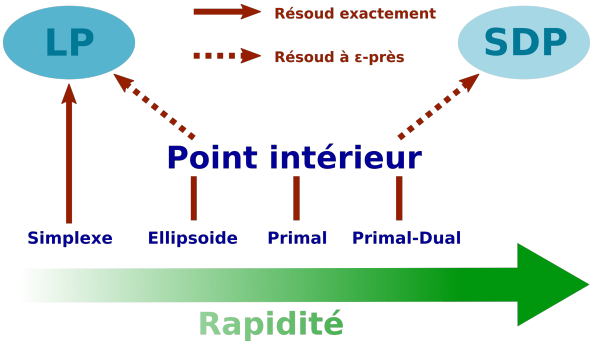


FIGURE 4.1 – Hiérarchie des problèmes d'optimisation convexe



Remarque : La complexité en pire cas du simplexe est bien exponentielle contrairement aux autres qui sont polynomiaux. Cependant, en pratique, l'algorithme est très efficace et est implémenté, conjointement avec un primal/dual, dans la plupart des solveurs.

FIGURE 4.2 – Hiérarchie des algorithmes de résolution de problème linéaire

La problématique de la vérification du code en ligne n'est pas directement abordée dans cet article. Cependant, elle serait grandement simplifiée, car une grande partie du calcul est effectuée hors-ligne. Cependant, cette solution nécessite d'embarquer des données de taille exponentielle par rapport au nombre de contraintes et est donc difficilement utilisable pour des problèmes de grande taille.

4.1.3 Choix de l'algorithme

Cette sous-section est dédiée au choix de l'algorithme que nous allons implémenter. Nous présenterons dans cette section plusieurs algorithmes résolvant des problèmes linéaires. Ainsi, pour plus de clarté, ils sont résumés dans la figure 4.2.

Simplexe ou Points intérieurs ? Nous avons donc décidé de résoudre un problème linéaire. Dans la section 2.2, on a vu qu'il existait deux grandes familles d'algorithmes qui permettent de résoudre un problème linéaire : les algorithmes du simplexe et les algorithmes de points intérieurs. Les premiers ont une complexité, dans le pire des cas, exponentielle, mais permettent d'obtenir une solution exacte. Les seconds ont une complexité polynomiale, en revanche la solution renvoyée n'est pas exacte, mais correcte à ϵ près. En pratique, la complexité

au cas le pire est rarement atteinte et une étude présentée dans Spielman et Teng, 2004 montre que la complexité lisse du simplexe est polynomiale. Cependant, les algorithmes de points intérieurs ont un avantage important pour nous : ils permettent de résoudre un grand nombre de problèmes convexes. En particulier, ils permettent de résoudre les problèmes SDP, une classe bien plus grande de problèmes d'optimisation. Cela nécessite quelques adaptations sur l'algorithme, mais la structure de celui-ci reste la même. Cette thèse se propose donc de travailler sur des problèmes LP, mais il sera aisé de passer ensuite à des problèmes plus complexes sans changer fondamentalement l'algorithme. On a vu que pySil était modulaire et facilement instrumentable ce qui simplifiera le travail de transposition le moment venu.

Méthode des ellipsoïdes, primale ou primale-duale ? Dans la famille des points intérieurs, on retrouve trois types d'algorithmes. Le premier, historiquement est la méthode des ellipsoïdes présentée dans Khachiyan, 1979. Elle est peu utilisée en pratique, car elle demande un grand nombre d'itérations. Une autre méthode peu utilisée pour la même raison est la méthode primale décrite dans la sous-section 2.2.5 aussi appelée méthode avec barrière. Elle a l'avantage d'être assez simple à implémenter. Enfin, les méthodes primales/duales, décrites dans la sous-section 2.2.5, sont les plus efficaces. Ce sont ces méthodes qui sont implémentées dans la plupart les logiciels d'optimisation [1]. Ce sont ces dernières qu'il serait intéressant de prouver, mais nous préférons, encore une fois, privilégier la simplicité à l'efficacité. Nous avons donc choisi d'utiliser un algorithme primal qui est plus simple à implémenter. Il reste cependant plus proche d'un algorithme primal/dual que la méthode des ellipsoïdes.

Conclusion. Nous allons implémenter un algorithme primal pour résoudre un problème linéaire. On limite, ainsi, l'ensemble des problèmes que l'on pourrait résoudre et l'algorithme n'est pas aussi efficace que les meilleurs algorithmes actuels. Cependant, l'algorithme sera plus simple à implémenter et surtout à prouver. De plus, l'algorithme choisi reste assez proche des méthodes les plus efficaces pour que le travail effectué sur cet algorithme puisse être poursuivi et étendu à ces méthodes.

1. On peut trouver dans Gearhart et al., 2013 une comparaison des différents logiciels libres dédiés à la programmation linéaire ainsi que les algorithmes utilisés.

4.2 Écrire un algorithme primal annoté

Dans cette partie, nous allons implémenter en PYSIL un algorithme de points intérieurs primal pour des problèmes linéaires. Le but est de pouvoir ensuite le compiler vers un langage de bas niveau et garantir la correction du code ainsi généré. Nous avons décidé de nous baser sur NESTEROV, 2004, Chapitre 4 pour écrire cette implémentation. La théorie nécessaire à l'écriture de l'algorithme et à sa preuve est redécrite dans cette partie. On se place dans le cadre, plus restreint, de l'optimisation LP bien que la plupart des résultats restent valides pour un problème SDP, voire convexe.

En plus du code, nous voulons spécifier les propriétés que l'on attend des différentes parties du code grâce à la logique de Hoare. Pour cela, nous devons introduire un certain nombre d'opérateurs spécifiques à notre problème d'optimisation. Nous les rassemblerons dans une axiomatique dédiée :

```
1 OptimAxiomatic = Axiomatic("Optim")
```

4.2.1 Définition d'un problème linéaire en pySil

Dans cette section, on se donne A, b, c comme présenté dans la définition 2.2.1. Nous supposerons $E_f(A, b)$ non vide et borné pour le reste de cette section. Ceci assure l'existence de solutions.

$E_f(A, b)$ se notera E_f si il n'y a pas d'ambiguïté et on introduit $f(x) = \langle c, x \rangle$.

La première étape est de définir en PYSIL un problème linéaire. Cela correspond à traduire la définition 2.2.1.

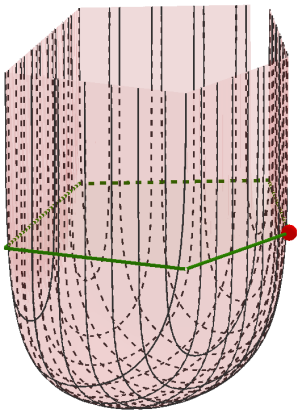
Cela nécessite d'introduire la notion de borne inférieure qui prend en argument un ensemble et une fonction. Ces deux dernières notions, bien que définissables dans la logique, rendraient la définition trop abstraite et donc les preuves plus compliquées. Afin de rester avec des définitions simples d'utilisation on va se contenter de directement définir l'opérateur `"sol"` comme une fonction non interprétée, représentant S.

```
1 with OptimAxiomatic:
2   sol = Operator("sol", Real, [LMat, LMat, LMat])
3   Axiom("sol_min",
4     forall(LMat, LMat, LMat, LMat, lambda A, b, c, x:
5       (A * x < b).implies(c.dot(x) >= sol(A, b, c))))
6   Axiom("sol_greater",
7     forall(LMat, LMat, LMat, Real, lambda A, b, c, m:
8       (forall(LMat, lambda x: (A * x < b).implies(c.dot(x) >= m))
9         ).implies(sol(A, b, c) >= m)))
```

À la seconde ligne on déclare l'opérateur `"sol"` renvoyant un réel et prenant en argument les 3 matrices A, b, c. Ensuite à la troisième on définit l'axiome `"sol_min"` qui assure que $S(A, b, c)$ est un minorant, puis à la sixième ligne `"sol_greater"` qui garantit que $S(A, b, c)$ est plus grand que tous les minorants.

On peut grâce à cette définition écrire le contrat de notre fonction principale. Supposons que l'on a `A`, `b`, `c` et `EPSILON`, définis dans notre programme PYSIL. On peut désormais écrire le contrat de notre fonction principale `pathfollowing`.

```
1 pathfollowing = Function("pathfollowing")
2 with pathfollowing:
3   x = Mat(N, 1).Var('X')
4 pathfollowing.addEnsure(A * x < b)
5 pathfollowing.addEnsure(x.dot(c) - sol(A, b, c) < EPSILON)
```



En vert on a représenté l'ensemble de contraintes issus de l'exemple 2.2 et en rouge la fonction barrière associée.

FIGURE 4.3 – Fonction barrière

`x` est la variable qui contiendra à la fin de la fonction `pathfollowing` le point recherché. C'est-à-dire une solution du problème d'optimisation correcte à `EPSILON` près.

4.2.2 Rajout d'une fonction barrière

On a vu dans la sous-section 2.2.4 que l'on pouvait utiliser la méthode de Newton pour chercher les extremums d'une fonction. Cependant, en l'état, le théorème 2.2.1 n'est pas utilisable, car il ne peut trouver un minimum que dans l'intérieur d'un domaine alors que dans le cas de l'optimisation convexe l'optimum se situe sur la frontière comme indiqué dans la sous-section 2.2.5. Cette limitation peut être contournée par l'utilisation d'une fonction barrière.

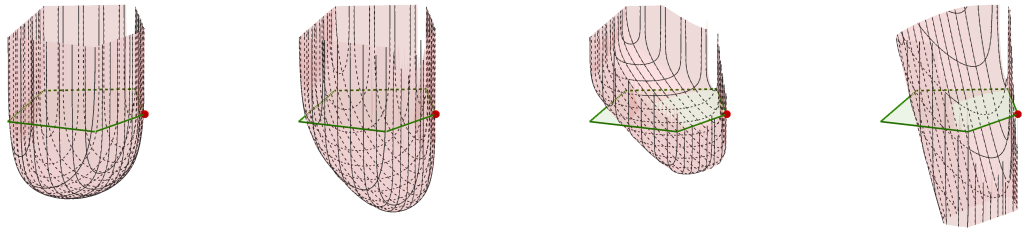
Définition 4.2.1 (Fonction barrière) *On appelle fonction barrière d'un ensemble ouvert U une fonction $B : U \to \mathbb{R}$ strictement convexe telle que pour toute $x_0 \in \partial U$, $\lim\limits_{x \to x_0} B(x) \to +\infty$*

Les fonctions barrières se construisent en général avec un logarithme appliqué à une mesure de la faisabilité. Pour le cas linéaire, on prend $B : E_f \to \mathbb{R}$ défini par $B(x) = -\sum\limits_{i=0}^{m} log(b_i - \langle a_i, x \rangle)$. On notera G son gradient et H sa hessienne définis sur $E_f(A, b)$.

Propriété 4.2.1 *B est une fonction barrière.*
De plus :
$[G(x)]_j = \sum\limits_{i=0}^{m} \frac{a_{i,j}}{b_i - \langle a_i, x \rangle}$,
$[H(x)]_{k,l} = \sum\limits_{i=0}^{m} \frac{a_{i,k} a_{i,l}}{(b_i - \langle a_i, x \rangle)^2}$.

B est dessiné dans la figure 4.3 pour un ensemble de 2 variables et 5 contraintes.

Définition 4.2.2 (Fonction de coût ajusté) *On appelle $\tilde{f} : E_f \times \mathbb{R}_+ \to \mathbb{R}$ la fonction de coût ajusté définie par $\tilde{f}(x, t) = t \times f(x) + B(x)$.*



(a) $t = 0$ (b) $t = 1$ (c) $t = 3$ (d) $t = 4$

Le minimum de la fonction barrière se rapproche de l'optimum représenté par un point rouge avec $t \to +\infty$.

FIGURE 4.4 – Évolution de la fonction de coût ajusté quand t augmente.

Propriété 4.2.2 *Cette fonction $\tilde{f}$ possède trois propriétés intéressantes pour la suite :*

1. *Pour tout $x \in E_f$, $\tilde{f}(x, 0) = B(x)$, $\tilde{f}(\cdot, 0)$ est donc indépendant de la fonction de coût.*
2. *Pour tout $x \in E_f$, $\tilde{f}(x, \cdot) \sim_{+\infty} g_x$, avec $g_x(t) = t \times f(x)$ qui est indépendant de la fonction barrière.*
3. *Soit $t > 0$, pour tout $x \in E_f$, $D^2\tilde{f}(x, t) = D^2B(x)$ ainsi $\tilde{f}(\cdot, t)$ est strictement convexe et admet donc un unique point critique, correspondant à un minimum.*

Ainsi pour chaque $t \in [0, +\infty[$ il existe un minimum à la fonction de coût ajustée et il est unique. Tous ces minimums forment un chemin que l'on va nommer chemin central. Ce chemin est intéressant, car on observe que, quand t devient grand, l'impact de la fonction barrière sur le coût ajusté diminue. Ainsi, à la limite on espère que le chemin central converge vers l'optimum recherché. On peut observer ce phénomène dans la figure 4.4.

4.2.3 Progression le long du chemin central

Le chemin central est un chemin défini par les points de E_f de coût ajusté minimal pour chaque t. Un exemple d'un tel chemin est représenté dans la figure 4.5.

Définition 4.2.3 (Chemin central)

$$\begin{aligned} x^* : \ \mathbb{R}^+ &\to E_f \\ t &\mapsto \arg\min_{x \in E_f} \tilde{f}(x, t) \end{aligned} \tag{4.1}$$

Le minimum existe et est bien unique par la propriété 4.2.2. $x^*(0)$ est appelé le centre analytique et se note x_{AC}, il est indépendant de la fonction de coût.

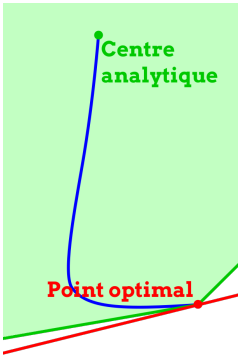
Ainsi à chaque $t \in \mathbb{R}^+$ est associé une position sur le chemin central de manière unique grâce à la propriété 4.2.2. Le chemin central possède une propriété fondamentale aux algorithmes de points intérieurs : il a pour extrémité la solution du problème d'optimisation associé :

Propriété 4.2.3 $\lim_{t \to +\infty} x^*(t) = S(A, b, c)$

L'algorithme va consister à suivre les points de ce chemin de $t = 0$ à $t = t_{fin}$ avec t_{fin} assez grand pour que $x^*(t_{fin})$ soit ϵ-proche de $S(A, b, c)$.

Pour cela il utilisera deux variables : `x` que nous avons déjà défini en section 4.2.1 et `t`. L'algorithme consistera en une suite d'itérations qui augmentent t et mettent à jour x afin de suivre le chemin central vers un point optimal.

```
with pathfollowing:
  t = Real.Var('t')
```



En vert sont représentés l'ensemble faisable et en bleu clair le chemin central.

FIGURE 4.5 – Exemple de chemin central pour un problème de dimension deux

Pour suivre le chemin central, on aura donc t qui représente le point $x^*(t)$. Idéalement x décrirait exactement $x^*(t)$ cependant informatiquement parlant cela n'est pas possible, car on ne peut représenter seulement qu'une partie des réels. Ainsi on se contentera d'avoir un x **suffisamment proche** $x^*(t)$. On verra que cela est suffisant pour obtenir l'ϵ-optimalité.

Au lancement de l'algorithme, `t` vaudra 0 et `x` représentera un point **suffisamment proche** du centre analytique : $x^*(0)$. Ensuite, à chaque itération `t` sera augmenté de `dt` et `x` sera translaté de `dx` afin d'être **suffisamment proche** de $x^*(t + dt)$. À la fin, on obtient donc un `x` **suffisamment proche** de $x^*(t_{fin})$.

Cette notion de **suffisamment proche** n'est pas encore bien définie. Ce prédicat dépend de notre problème A, b, c, de la position sur le chemin central t, de la position courante x et d'un réel β représentant la distance entre x et $x^*(t)$. On notera cette condition $acc(A, b, c, t, x, \beta)$ et on l'appellera *condition de centrage approché*.

```
with OptimAxiomatic:
  acc = Operator("acc", Bool, [LMat, LMat, LMat, Real, LMat, Real])
```

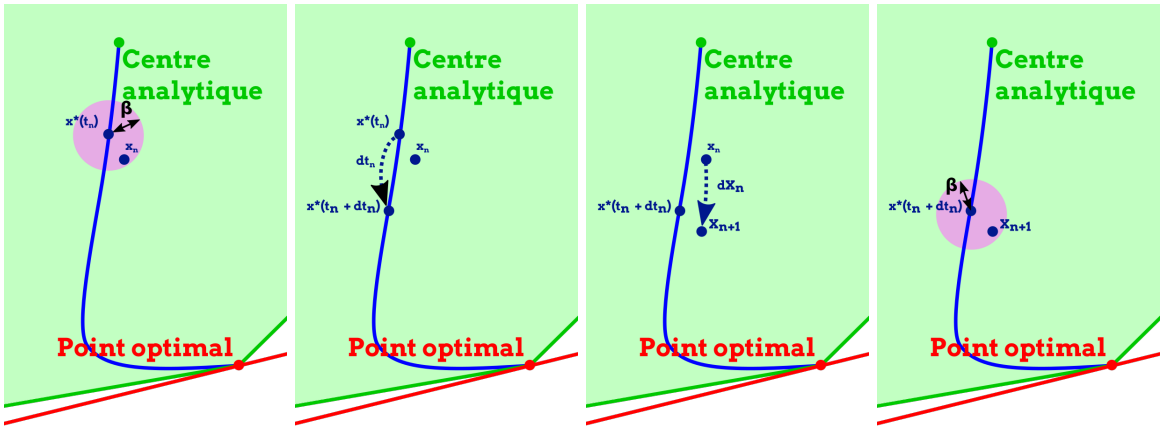
Ce prédicat doit être vrai avant et après chaque itération de la boucle, c'est donc un invariant selon la définition 2.3.3.

Ainsi, même si *acc* n'est pas pleinement défini, on peut déjà écrire le squelette de l'algorithme :

```
with pathhfollwing:
  with While(t < F_FIN):
    addInvariant(acc(A, b, c, t, x, BETA))
    compute_pre() # Calcul des éléments nécessaire aux deux fonctions
    update_t()
    update_x()
```

On suppose qu'une variable `T_FIN` contient la valeur de t à partir de laquelle la précision ϵ est atteinte. `BETA` contient un réel représentant la proximité entre x_n et $x^*(t_n)$. L'algorithme se décompose en trois phases. La première est un précalcul des éléments nécessaires aux deux fonctions dans `compute_pre`. La deuxième phase correspond à la mise à jour de t et la troisième à celle de x. Ces deux dernières étapes sont représentées dans la figure 4.6.

Il reste donc à trouver à quel moment la boucle doit s'arrêter, comment mettre à jour x et comment mettre à jour t. Au fur et à mesure que nous définirons ces éléments, nous définirons *acc* de manière à rendre correct l'algorithme. Nous allons commencer par calculer quand la boucle doit s'arrêter.



(a) *Avant :* x_n est **proche** à β du point $x^*(t_n)$ du chemin central

(b) *update_t :* on avance sur le chemin central vers l'optimum

(c) *update_x :* on calcule un nouveau x pour se rapprocher de $x^*(t_{n+1})$

(d) *Après :* x_{n+1} est **proche** à β d'un point $x^*(t_{n+1})$

FIGURE 4.6 – Décomposition d'une itération en plusieurs étapes

4.2.4 Expression de t_{fin}

Soit $t > 0$ et $x \in E_f$ vérifiant $acc(A, b, c, t, x, \beta)$. À la fin des itérations, on veut :

$$d = \langle c,\, x\rangle - S(A, b, c) < \epsilon \tag{4.2}$$

Pour contrôler cette distance, nous allons introduire un point du chemin central : $x^*(t)$, afin de pouvoir scinder en deux parties notre distance :

$$d = \langle c,\, x\rangle - \langle c,\, x^*(t)\rangle + \langle c,\, x^*(t)\rangle - \langle c,\, x^*\rangle \tag{4.3}$$

On va alors contrôler indépendamment la différence de coût d_1 entre x et le point courant du chemin central et d_2 la différence de cout entre le point courant du chemin central et le coût optimal.

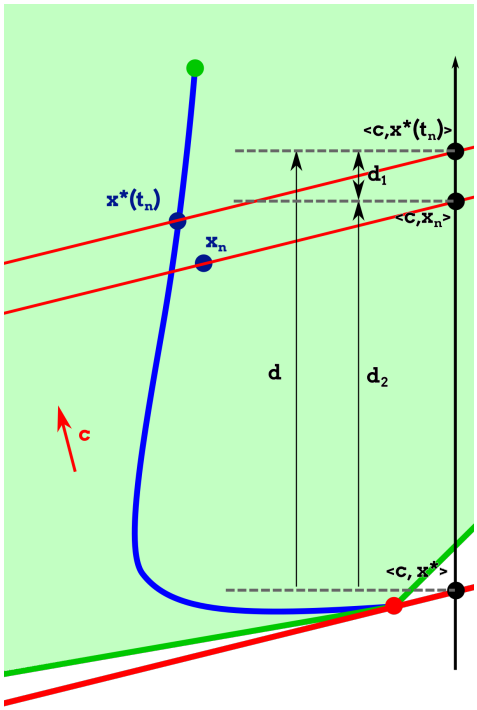
$$d_1 = |\langle c,\, x\rangle - \langle c,\, x^*(t)\rangle| \tag{4.4}$$
$$d_2 = \langle c,\, x^*(t)\rangle - \langle c,\, x^*\rangle \geq 0 \tag{4.5}$$

Intuitivement, la première quantité sera toujours faible tant qu'on reste proche du chemin central. En revanche, la seconde va diminuer tout au long de l'algorithme. La figure 4.7 illustre ce propos.

Contrôle de $|\langle c,\, x\rangle - \langle c,\, x^*(t)\rangle|$

On souhaite borner la différence de coût entre le point courant x et le point associé sur le chemin central $x^*(t)$ par une expression décroissante en t. Pour cela, on note $B'(x)$ le gradient de B au point x et $\tilde{f}'(x,t) = t \cdot c + B'(x)$ le gradient de $\tilde{f}'(\cdot, t)$. Ainsi on peut écrire :

$$|\langle c,\, x\rangle - \langle c,\, x^*(t)\rangle| = \frac{1}{t}|\langle \tilde{f}'(x,t) - B'(x),\, x - x^*(t)\rangle| \tag{4.6}$$
$$\leq \frac{1}{t}(|\langle \tilde{f}'(x,t),\, x - x^*(t)\rangle| + |\langle B'(x),\, x - x^*(t)\rangle|) \tag{4.7}$$



On retrouve sur l'axe y, la valeur de la fonction de coût ajusté. En rouge, on représente les droites $\langle c, x\rangle = \langle c, a\rangle$ pour $a \in \{x^*(t), x_n, x^*\}$. On cherche à partir de quel t, $d < \epsilon$, pour cela on doit majorer d_1 puis d_2.

FIGURE 4.7 – Majoration du coût ajusté

On va commencer par majorer le terme de droite. En effet, notre fonction barrière possède la propriété suivante [2] :

Propriété 4.2.4 (Condition de barrière auto-concordante) *Pour tout x appartenant à E_f et $u \in \mathbb{R}^n$ on a $\langle B'(x), u\rangle^2 < m\langle B''(x)u, u\rangle$ avec m le nombre de contraintes du problème et $B''(x)$ la hessienne de B au point x.*

En appliquant cette propriété au terme de droite de notre équation, on obtient :

$$|\langle B'(x), x - x^*(t)\rangle| \leq \sqrt{m}\sqrt{\langle B''(x)(x - x^*(t)), x - x^*(t)\rangle} \tag{4.8}$$

Notons que $B''(x)$ est une matrice symétrique définie positive, ainsi elle induit deux [3] normes locales que l'on notera pour tout $u \in \mathbb{R}^n$:

$$\|u\|_x = \sqrt{\langle B''(x)u, u\rangle} \tag{4.9}$$

$$\|u\|_x^* = \sqrt{\langle B''(x)^{-1}u, u\rangle} \tag{4.10}$$

La norme que nous utiliserons le plus est la seconde ainsi nous la définissions dans notre librairie d'optimisation :

```
with OptimAxiomatic:
    hess = Operator("hess", LMat, [LMat, LMat, LMat])
    grad = Operator("grad", LMat, [LMat, LMat, LMat])
    norm = Operator("norm", Real, [LMat, LMat, LMat, LMat],
        lambda A, b, y, x:
            (y.transpose() * (hess(A, b, x).inv() * y))[0, 0].sqrtl())
```

On en a profité pour déclarer les opérateurs de gradient et hessienne qui sont définis par leur expression directe donnée dans la propriété 4.2.1.

Ainsi on peut réécrire l'équation (4.7) :

$$|\langle c, x\rangle - \langle c, x^*(t)\rangle| \leq \frac{1}{t}(|\langle \tilde{f}'(x,t), x - x^*(t)\rangle| + \sqrt{m}\,\|x - x^*(t)\|_x) \tag{4.11}$$

On va maintenant borner le terme de gauche. Les normes $\|\cdot\|_x$ et $\|\cdot\|_x^*$ ont une propriété (cf. Nesterov, 2004, p. 4.1.4) proche de celle de Cauchy-Schwarz :

Propriété 4.2.5 (Cauchy-Schwarz local) *Pour tout u et v de $\mathbb{R}^n$ on a $|\langle u, v\rangle| < \|u\|_x^* \times \|v\|_x$*

On peut donc borner le terme de gauche par un terme de la même forme que le terme de droite :

$$|\langle c, x\rangle - \langle c, x^*(t)\rangle| \leq \frac{1}{t}\left(\left\|\tilde{f}'(x,t)\right\|_x^* \times \|x - x^*(t)\|_x + \sqrt{m}\,\|x - x^*(t)\|_x\right) \tag{4.12}$$

Il faut maintenant borner $\|x - x^*(t)\|_x$, ce qui nous est apporté par le théorème suivant (cf. Nesterov, 2004, Th4.1.13) :

2. Cette propriété peut sembler bien heureuse, mais elle est issue de la théorie de l'auto-concordance introduite dans Nesterov et Nemirovski, 1994. Cette théorie extrait les propriétés nécessaires pour contrôler la convergence des algorithmes de Newton, on se contentera ici d'en utiliser les résultats. C'est cette théorie qui nous a permis de choisir la fonction barrière par exemple.

3. Plus exactement la seconde est induite par $B''(x)^{-1}$ qui est elle aussi symétrique définie positive puisque $B''(x)$ l'est.

Théorème 4.2.1 *Pour tout x vérifiant $\left\|\tilde{f}'(x,t)\right\|_x^* < 1$, on a $\|x - x^*(t)\|_x < \frac{\left\|\tilde{f}'(x,t)\right\|_x^*}{1-\left\|\tilde{f}'(x,t)\right\|_x^*}$.*

Soit $x \in E_f$, $t > 0$ et $\beta \in \mathbb{R}^+$ vérifiant :

$$\left\|\tilde{f}'(x,t)\right\|_x^* < \beta \tag{4.13}$$

la norme de la dérivée de $\tilde{f}$ au point x est borné par β. Or, à t fixé si $\left\|\tilde{f}'(x,t)\right\|_x^*$ tend vers zéro, x tend vers $x^*(t)$ puisque c'est l'unique point critique de $\tilde{f}$. Ainsi, $\left\|\tilde{f}'(x,t)\right\|_x^*$ représente une notion de proximité entre x et $x^*(t)$, comme pour utiliser ce théorème on doit avoir $\left\|\tilde{f}'(x,t)\right\|_x^* < 1$, on va prendre $\beta \in [0,1]$ et $acc(A,b,x,t,\beta) \Rightarrow$ (4.13). Cette implication est raisonnable, car *acc* doit, elle aussi, représenter une notion de proximité entre x et $x^*(t)$. L'application du théorème 4.2.1 à l'équation (4.12) nous permet d'obtenir la majoration suivante :

$$|\langle c,\, x\rangle - \langle c,\, x^*(t)\rangle| \leq \frac{1}{t}\left(\beta + \sqrt{m}\right)\frac{\beta}{1-\beta} \tag{4.14}$$

Contrôle de $\langle c,\, x^*(t)\rangle - \langle c,\, x^*\rangle$

Il faut donc maintenant contrôler la distance entre un point du chemin central et l'optimal :

$$\langle c,\, x^*(t)\rangle - \langle c,\, x^*\rangle = \frac{1}{t}\langle B'(x^*(t)),\, x^*(t) - x^*\rangle \tag{4.15}$$

La propriété 4.2.4 permet de déduire le théorème suivant(cf. Nesterov, 2004, Th4.2.4) :

Théorème 4.2.2 *Pour tout x et y appartenant à E_f, $\langle B'(x),\, y - x\rangle < m$.*

Ce théorème permet donc d'écrire $\langle c,\, x^*(t)\rangle - \langle c,\, x^*\rangle < \frac{m}{t}$.

Bilan

Ainsi, on a $\langle c,\, x\rangle - S(A,b,c) < \frac{1}{t}\left(\beta + \sqrt{m}\right)\frac{\beta}{1-\beta} + \frac{m}{t}$. Si on prend :

$$t > \frac{1}{\epsilon}\left(\frac{\beta(\beta+\sqrt{m})}{1-\beta} + m\right) \tag{4.16}$$

on obtient $\langle c,\, x\rangle - S(A,b,c) < \epsilon$. Ceci nous donne donc à partir de quelle valeur de t on peut s'arrêter : $t_{fin} = \frac{1}{\epsilon}\left(\frac{\beta(\beta+\sqrt{m})}{1-\beta} + m\right)$ que l'on peut définir en PySil :

```
T_FIN = ( BETA*(BETA + sqrt(M))/(1-BETA) + M) / EPSILON
```

`BETA` représente β un réel de $]0,1[$ choisi par l'utilisateur, Nesterov, 2004, p. 4.2.22 conseille de prendre $\beta = \frac{1}{9}$. `M` représente m, le nombre de contrainte et `EPSILON` représente ϵ et sera choisi par l'utilisateur.

Dans cette sous-section nous avons dû supposer que $acc(A,b,x,t,\beta) \Rightarrow$ (4.13), nous allons maintenant montrer que prendre $acc(A,b,x,t,\beta) \Leftrightarrow$ (4.13) est suffisant. Pour cela, nous allons montrer que l'on peut augmenter t et trouver x en conséquence de manière à maintenir cet invariant.

```
with OptimAxiomatic:
  acc = Operator("acc", Bool, [LMat, LMat, LMat, Real, LMat, Real],
      lambda A, b, c, t, x, beta:
          norm(A, b, grad(A, b, x) + c * t , x) <= beta)
```

4.2.5 Calcul de `dx`

Le calcul de `dx` se situe juste après le calcul du nouveau t que nous noterons t_+. On cherche à calculer dx tel que $x_+ = x + dx$ vérifie $acc(A, b, c, t_+, x_+, \beta)$. Pour cela, on pourra émettre des hypothèses sur t_+ que nous vérifierons ensuite dans la sous-section suivante. Comme on veut montrer que $\left\|\tilde{f}'(x_+, t_+)\right\|_x^* < \beta$, on veut que x_+ se rapproche du point où $D\tilde{f}'(\cdot, t + dt)$ s'annule. Pour cela nous allons utiliser un pas de Newton :

$$dx = -\tilde{f}''(x, t_+)^{-1} \times \tilde{f}'(x, t_+) \tag{4.17}$$

Ce qui donne en PySil :

```
with Function("update_x") as update_x:
  update_x.addRequire(H == hess(A, b, x))
  update_x.addRequire(G == grad(A, b, x))
  update_x.addEnsure(acc(A, b, c, t, x, BETA))
  dx %= -1 * solve(H, G + c * t)
  x %= x + dx
```

On a introduit deux préconditions assurant que `G` et `H` contiennent respectivement $G(\mathtt{x})$ et $H(\mathtt{x})$. La fonction utilise l'opérateur `solve(W, h)` qui renvoie un vecteur égal à $W^{-1}h$. Ce calcul est effectué grâce à l'algorithme de Cholesky abordé dans la sous-section 3.4.3.

Il reste à trouver quelle précondition sur t_+ il faut rajouter à cette fonction pour obtenir la postcondition souhaitée. Pour cela, nous utilisons le théorème suivant(cf. Nesterov, 2004, Th4.1.14) :

Théorème 4.2.3 *Pour tout x vérifiant $\left\|\tilde{f}'(x,t)\right\|_x^* < 1$, si $x_+ = x - B''(x)^{-1}(B'(x) + t \times c)$ alors $\left\|\tilde{f}'(x_+,t)\right\|_{x_+}^* \leq \left(\frac{\left\|\tilde{f}'(x,t)\right\|_x^*}{1-\left\|\tilde{f}'(x,t)\right\|_x^*}\right)^2$.*

On peut alors montrer par le calcul que $\frac{\sqrt{\beta}}{1+\sqrt{\beta}} < 1$ est solution de l'inégalité en $z \in [0, 1[$ suivante : $\left(\frac{z}{1-z}\right)^2 < \beta$. Pour simplifier les calculs suivants on prend $\gamma = \frac{\sqrt{\beta}}{1+\sqrt{\beta}} - \beta$, ainsi la solution de l'inégalité se réécrit $\beta + \gamma$. Avec le théorème 4.2.3, on obtient alors :

$$\left\|\tilde{f}'(x, t_+)\right\|_x^* < \beta + \gamma \Rightarrow \left\|\tilde{f}'(x_+, t_+)\right\|_{x_+}^* \leq \beta. \tag{4.18}$$

Ainsi, on a trouvé une condition sur t_+ pour obtenir $acc(A, b, c, t_+, x_+, \beta)$:

$$\left\|\tilde{f}'(x, t_+)\right\|_x^* < \beta + \gamma \tag{4.19}$$

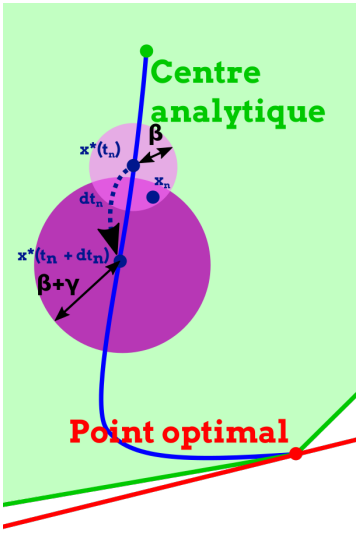
Cela nous permet d'introduire la précondition suivante au calcul du nouveau `x` :

```
GAMMA = sqrt(BETA) / (1 + sqrt(BETA)) - BETA
update_x.addRequire(acc(A, b, c, t, x, GAMMA + BETA))
```

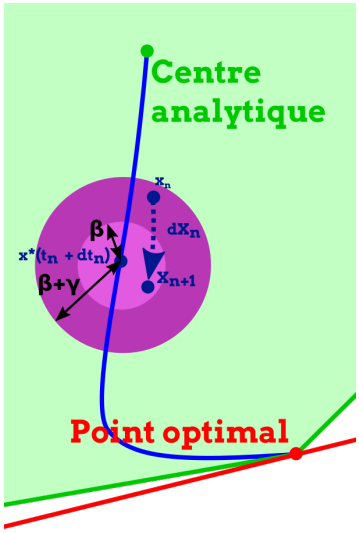
Une illustration de la mise à jour de x est présentée dans la figure 4.8b.

4.2.6 Calcul de `dt`

Pour cette étape on sait que l'on veut prouver l'équation (4.19) et on suppose que l'on a l'équation (4.13) vérifiée au début. Ceci est illustré dans la figure 4.8a.



(a) Mise à jour de t avec en violet clair la précondition et en violet foncé la postcondition



(b) Mise à jour de x avec en violet foncé la précondition et en violet clair la postcondition

FIGURE 4.8 – Mise à jour de t et x

Il reste à calculer l'augmentation de t de manière à ce que l'équation (4.19) soit vrai. Pour aider on sait que $\left\|\tilde{f}'(x,t)\right\|_x^* < \beta$ est vrai. Or

$$\left\|\tilde{f}'(x,t+dt)\right\|_x^* = \|(t+dt)c + B'(x)\|_x^* \leq \left\|\tilde{f}'(x,t)\right\|_x^* + dt \times \|c\|_x^* \leq \beta + dt \times \|c\|_x^* \quad (4.20)$$

Ainsi, si on prend $dt = \frac{\gamma}{\|c\|_x^*}$ on aura bien l'équation (4.19) vraie.

Ceci nous permet donc d'exprimer la fonction `"update_t"` :

```
with Function("update_t") as update_t:
  update_t.addRequire(H ==hess(A, b, x))
  update_t.addRequire(acc(A, b, c, t, x, BETA))
  update_t.addEnsure(acc(A, b, c, t, x, BETA + GAMMA))
  c_norm = Mat(1, 1).Var('c_norm')
  c_norm %= c.transpose() * solve(H, c)
  dt %= gamma / c_norm[0, 0].sqrt()
  t %= t + dt
```

4.2.7 Maintien de la faisabilité

Nous avons passé sous silence, le fait que x doit toujours rester dans l'ensemble faisable. Pour cela, il faut rajouter l'invariant correspondant, qui sera maintenu de facto grâce au théorème suivant :

Théorème 4.2.4 *Soit $x \in E_f(A,b)$ et $y \in \mathbb{R}^n$ tel que $\|y-x\|_x < 1$ alors $y \in E_f(A,b)$.*

En effet, pour prouver que $x+dx$ est faisable, il suffit d'appliquer le théorème 4.2.4 à x et $x+dx$ et prouver que $\|dx\|_x < 1$. Or on sait que

$$dx = -B''(x)^{-1} \times \tilde{f}'(x,t_+) \quad (4.21)$$

donc

$$\|dx\|_x = \sqrt{\langle B''(x)dx,\, dx\rangle} = \sqrt{\tilde{f}'(x,t_+)B''(x)^{-1} \times \tilde{f}'(x,t_+)} = \left\|\tilde{f}'(x,t)\right\|_x^*. \quad (4.22)$$

```
with Axiomatic("Optim"):
  sol = Operator("sol", Real, [LMat, LMat, LMat])
  Axiom("sol_min",
    forall(LMat, LMat, LMat, LMat, lambda A, b, c, x:
      (A * x < b).implies(c.dot(x) >= sol(A, b, c))))
  Axiom("sol_greater",
    forall(LMat, LMat, LMat, Real, lambda A, b, c, m:
      (forall(Real, lambda x: (A * x < b).implies(c.dot(x) >= m))
       ).implies(sol(A, b, c) >= m)))

  hess = Operator("hess", LMat, [LMat, LMat, LMat])
  grad = Operator("grad", LMat, [LMat, LMat, LMat])
  norm = Operator("norm", Real, [LMat, LMat, LMat, LMat],
      lambda A, b, y, x:
          (y.transpose() * (hess(A, b, x).inv() * y))[0, 0].sqrtl())

  acc = Operator("acc", Bool, [LMat, LMat, LMat, Real, LMat, Real],
      lambda A, b, c, t, x, beta:
          norm(A, b, grad(A, b, x) + c * t , x) <= beta)
```

FIGURE 4.9 – Axiomatique PYSIL utilisée pour l'annotation de l'algorithme primal

4.2.8 Bilan

On a défini les différentes étapes de l'algorithme, exprimé sa condition d'arrêt ainsi que l'invariant qui permet de prouver sa correction. On a aussi donné un contrat en logique de Hoare aux fonctions. Pour cela, on a dû définir une axiomatique dédiée à l'optimisation.

On peut retrouver la forme finale de l'axiomatique dans la figure 4.9 ainsi que celle de l'algorithme dans la figure 4.10. Nous allons maintenant voir comment on peut utiliser cet algorithme PYSIL dans un programme plus large comme un problème de commande prédictive.

```
T_FIN = ( BETA*(BETA + sqrt(M))/(1-BETA) + M) / EPSILON
GAMMA = sqrt(BETA) / (1 + sqrt(BETA)) - BETA

with Function("update_x") as update_x:
  update_x.addRequire(H == hess(A, b, x))
  update_x.addRequire(G == grad(A, b, x))
  update_x.addRequire(acc(A, b, c, t, x, GAMMA + BETA))
  update_x.addEnsure(acc(A, b, c, t, x, BETA))
  dx %= -1 * solve(H, G + c * t)
  x %= x + dx

with Function("update_t") as update_t:
  update_t.addRequire(H ==hess(A, b, x))
  update_t.addRequire(acc(A, b, c, t, x, BETA))
  update_t.addEnsure(acc(A, b, c, t, x, BETA + GAMMA))
  c_norm = Mat(1, 1).Var('c_norm')
  c_norm %= c.transpose() * solve(H, c)
  dt %= gamma / c_norm[0, 0].sqrt()
  t %= t + dt

with Function("pathfollowing") as pathfollowing:
    pathfollowing.addRequire(acc(A, b, c, 0, x, BETA))
    pathfollowing.addRequire(A * x < b)
    pathfollowing.addEnsure(A * x < b)
    pathfollowing.addEnsure(x.dot(c) - sol(A, b, c) < EPSILON)

    x = Mat(N, 1).Var('X')
    t = Real.Var('t')

    with While(t < F_FIN):
        addInvariant(acc(A, b, c, t, x, BETA))
        compute_pre()
        update_t()
        update_x()
```

FIGURE 4.10 – Algorithme primal, écrit en PYSIL, résolvant un problème linéaire

4.3 Exemple d'utilisation de l'algorithme primal

Dans la section précédente, nous avons écrit un algorithme de points intérieurs en pySil. C'est un algorithme primal résolvant un problème linéaire. Nous allons voir dans cette section comment l'utiliser dans le cadre du MPC. En particulier, nous verrons de quelle manière la paramétrisation impacte l'algorithme implémenté et comment borner de manière stricte le nombre d'itérations.

4.3.1 Utilisation dans le cadre de la commande prédictive

Une des motivations pour l'utilisation d'algorithme de points intérieurs est son utilisation dans le cadre de la commande prédictive présentée dans la section 2.1.3. Pour rappel, cela consiste à utiliser un modèle du système à contrôler pour simuler à chaque itération de la boucle de contrôle les N prochaines itérations. À partir de cette simulation, on extrait une trajectoire à suivre et on garde la première commande que l'on envoie au système puis on recommence. Nous allons dans cette partie montrer comment on peut implémenter un tel contrôleur en pySil.

Du problème MPC au problème primal

Soit S un système physique avec e variables de sortie et c variables d'entrée comment défini dans la section 2.1. On supposera, par souci de simplicité, que l'on a accès à l'intégralité des variables d'états. En pratique, elles s'obtiennent par ses capteurs et/ou le calcul, mais nous n'aborderons pas ce sujet.

On va maintenant introduire un problème MPC comme défini dans l'équation (2.3). Pour cela, on suppose donné un modèle linéaire discret $(F, D) \in \mathcal{M}_{s,s} \times \mathcal{M}_{c,s}$ de notre système S ainsi que $N \in \mathbb{N}$ l'horizon voulu. Soit $d \in \mathbb{R}^l$ le vecteur de coût et $(H, h) \in \mathcal{M}_{l,r} \times \mathbb{R}^r$ des contraintes linéaires avec $r, q \in \mathbb{N}$ et $l = (N+1) * n + N * m + q$. On appelle $MPC(F, D, d, H, h, N)$ la fonction qui associe à $\mathbf{x0}$, un état initial du système, la composante u_0 de la solution du problème linéaire suivant :

$$\inf_{X \in \mathbb{R}^{N \times s + (N-1) \times c}} \langle d,\, X \rangle \text{ soumis à } \begin{cases} HX < h \\ y_0 = \mathbf{y0} \\ y_1 = Fy_0 + Du_0 \\ \dots \\ y_N = Fy_{N-1} + Du_{N-1} \end{cases} \quad \text{avec } X = \begin{pmatrix} y_0 \\ \dots \\ y_N \\ u_0 \\ \dots \\ u_{N-1} \\ s_0 \\ \dots \\ s_q \end{pmatrix} \tag{4.23}$$

On reconnait bien dans 4.23 un problème linéaire, mais pour être résolu par notre algorithme il faut éliminer les contraintes d'égalité. Ceci peut se faire par la substitution successive des $y_0, ...y_N$ par ordre décroissant, en commençant par y_N. Cette procédure termine, car il n'y a pas de cycle dans les contraintes d'égalités. On écrit, $P(\mathbf{A}, \mathbf{B}, \mathbf{d}) = \mathbf{p} = Prim(p)$ le nouveau problème généré par cette procédure avec $A \in \mathcal{M}_{N \times m+q, l}$.

A quoi servent les s_i ? Les s_i sont des variables supplémentaires que l'on peut vouloir adjoindre au problème. Elles n'ont en général pas de sens physique, mais permettent de capturer des problèmes un peu plus généraux. Elles sont en général utilisées pour la linéarisation d'un modèle plus compliqué. Par exemple on peut vouloir minimiser la norme d'une variable d'entrée ou de sortie. Pour cela on introduit une nouvelle variable s_j et si on veut borner la

```
with Function("compute_pre") as compute_pre:
    x = Mat(N, 1).Var('X')
    (G, H) = dF(M, N, A, b, x)

with Function("computeAC") as newton:
    newton.addRequire(A * x < b)
    newton.addEnsure(acc(A, b, c, 0, x, BETA))
    w = Real.Var("w")

    with While(w > beta):
        compute_pre()
        v = solve(H, G)
        w %= (G.T() * v)[0, 0].sqrt()
        x %= x - v / ( 1 + w)
```

FIGURE 4.11 – Algorithme de Newton amorti pour le calcul du centre analytique.

norme de u_k on rajoute les contraintes $u_k < s_j$ et $-u_k < s_j$. Ainsi il suffira de minorer ensuite s_j pour minorer la norme de u_k. Avec de l'optimisation linéaire, on ne peut capturer que la norme une ou la norme infinie.

Calcul du centre analytique

Nous avons donc généré dans la sous-section précédente un problème primal que peut résoudre notre algorithme. Cependant, notre algorithme nécessite de connaitre un point proche du centre analytique. Pour être exact, on veut un $x_{AC} \in \mathbb{R}^{N\times m+q}$ qui vérifie $acc(\mathbf{A}, \mathbf{b}, \mathbf{c}, 0, x_{AC}, \beta)$. Cette condition se réécrit $\|G(x_{AC}\|^*_{x_{AC}} \leq \beta$. On rappelle que G est le gradient de B qui est convexe on peut donc utiliser un algorithme de Newton pour résoudre ce problème. On amortira l'algorithme par $\frac{1}{1+\|G(x_{AC}\|^*_{x_{AC}}}$ afin d'avoir des garanties sur le nombre d'itérations conformément à NESTEROV, 2004, p204. On peut trouver cet algorithme écrit en PYSIL dans la figure 4.11.

Trouver un point dans l'ensemble faisable

Le calcul du centre analytique nécessite lui-même un point dans le domaine. Pour obtenir un point faisable, il y a deux solutions possibles. Soit il existe une solution calculable facilement en exploitant la structure du problème, soit on doit résoudre un problème d'optimisation. Ce dernier devra lui-même avoir un point faisable facilement calculable. On utilise pour cela une méthode phase 1 comme présentée à la sous-section 2.2.5. Elle consiste, pour un problème $p = P(A, b, c)$, avec $A \in \mathcal{M}_{m,n}$, à construire un nouveau problème $P_1(p)$ ayant pour solution un point faisable de p.

On se retrouve donc avec 4 phases pour pouvoir résoudre un problème linéaire p :

— Calculer le centre analytique de $P_1(p)$.

— Calculer une solution de $P_1(p)$.

— Calculer le centre analytique de p.

— Calculer une solution de p.

Par contre seulement deux algorithmes sont nécessaires : l'algorithme primal de la section précédente et l'algorithme de calcul du centre analytique de la figure 4.11. Les deux algorithmes étant assez proches, cette thèse n'abordera pas l'algorithme de calcul du centre analytique.

Dans le cadre de la commande prédictive, on peut s'attendre à trouver une solution faisable via la structure du problème. Par exemple, si on reprend le problème 4.23, il arrive qu'en prenant $y_i = \mathbf{y0}$ pour tout $i \in [\![0, N[\![$, on puisse constuire automatiquement un point X faisable. Cela correspond à laisser le système dans un état stationnaire.

4.3.2 Borner le temps d'exécution

On a donc vu dans la section précédente comment utiliser un algorithme de points intérieurs dans le cadre de la commande prédictive. Cependant dans le cadre d'une utilisation embarquée critique, en particulier dans la boucle de contrôle, il peut être nécessaire de borner de manière précise le temps d'exécution des algorithmes embarqués. Pour cela, l'idéal est de borner le nombre d'itérations à la compilation afin de transformer la boucle **`while`** en une boucle **`for`**.

Le livre de Nesterov, 2004, p203 nous donne une minoration pour t. C'est lui qui représente l'avancement sur le chemin central et qui donne donc la condition d'arrêt (4.16) :

$$t > \frac{1}{\epsilon}\left(\frac{\beta(\beta+\sqrt{m})}{1-\beta} + m\right). \tag{4.24}$$

On peut alors extraire de la preuve du théorème de convergence 4.2.9 de Nesterov, 2004, p203 la propriété suivante :

Propriété 4.3.1 (Minoration de t_k) *Soit $k > 0$, et t_k la valeur de t à la k-ième itération. Alors*

$$t_k > \frac{\gamma(1-2\beta)}{(1-\beta)\,\|c\|^*_{x_{AC}}}\left(1 + \frac{\gamma}{\beta+\sqrt{m}}\right)^{k-1}. \tag{4.25}$$

Notons $a = \frac{\gamma(1-2\beta)}{1-\beta}$, $q = 1 + \frac{\gamma}{\beta+\sqrt{m}}$ et $\delta = \frac{\beta(\beta+\sqrt{m}}{1-\beta} + m$. On peut alors réécrire l'équation (4.25) avec ces nouvelles notations :

$$t_k > \frac{a}{\|c\|^*_{x_{AC}}} q^{k-1} \tag{4.26}$$

ainsi que l'équation (4.24) :

$$t > \frac{\delta}{\epsilon}. \tag{4.27}$$

On peut ainsi combiner les deux, en cherchant le N tel que

$$\frac{a}{\|c\|^*_{x_{AC}}} q^{N-1} > \frac{\delta}{\epsilon}. \tag{4.28}$$

Ainsi, si on effectue au moins N itérations on sera sûr que la condition d'arrêt est respectée et donc que l'on a atteint l'ϵ-optimalité.

En appliquant un logarithme aux deux côtés de l'inéquation (4.28), on obtient :

$$N > n_{max} = \lceil 1 + \frac{\ln(\delta) + \ln(\|c\|^*_{x_{AC}}) - \ln(a) - \ln(\epsilon)}{\ln(q)} \rceil \tag{4.29}$$

Ainsi, effectuer n_{max} itérations est suffisant pour que le x calculé soit ϵ-optimal.

On peut donc réécrire la fonction principale comme présentée dans la figure 4.12. On a besoin de rajouter un nouvel invariant représentant l'équation (4.25). Pour cela, on a eu besoin de définir la suite géométrique `LOWER` dans une axiomatique dédiée.

Le calcul de $\|c\|^*_{x_{AC}}$(`NCCA` dans le programme 4.12), ne pose pas de problème si A et b sont connus à l'avance. En revanche, si ceux-ci ne sont connus qu'au moment de l'exécution il n'est pas simple de les calculer à la compilation et donc de calculer le nombre d'itérations à l'avance. Cette problématique est restée ouverte et est discutée dans la section 7.1.

```
NCCA = ...
DELTA = v + ((beta + sqrt(v))*beta)/(1 - beta)
a = gamma*(1.0-2.0*beta) / ((1.0 - beta)*NCCA)
q = 1 + gamma/(beta + sqrt(v))
NBR = ceil(1.0 + (log(DELTA) - log(a) - log(epsilon)) / log(q))

with Axiomatic("AxLower"):
    lower = Operator("lower", Real, [Int], rec=
        lambda f, k:
            Real.Cst(-1).iif(k<1, (q * f(k-1)).iif(k>1, Real.Cst(a))))

with pathfollowing:
  with For(1, NBR) as l:
    addInvariant(acc(A, b, c, t, x, BETA))
    addInvariant(A*x < b)
    addInvariant(t > lower(l))
    compute_pre()
    update_t()
    update_x()
```

FIGURE 4.12 – Fonction principale avec le **While** remplacé par un **For**

Conclusion

Dans ce chapitre, nous avons fait le choix de nous limiter à l'optimisation linéaire et un algorithme primal afin de ne pas complexifier la preuve et ainsi de nous concentrer sur les problématiques essentielles. Nous avons ensuite écrit un code en PYSIL résolvant des problèmes linéaires et nous l'avons aussi spécifié avec la logique de Hoare. Pour cela la rédaction d'une axiomatique dédiée a été nécessaire. Elle définit entre autres dans la logique de PYSIL ce qu'est un problème linéaire. Enfin, nous avons vu comment ce code pouvait servir dans un contexte de commande prédictive qui nécessite la résolution régulière de problème d'optimisation en ligne.

Nous avons publié ces résultats dans COHEN et al., 2017 et FERON et al., 2017 avec la différence que le code était exprimé en C avec les triplets de Hoare au format ACSL. En effet, afin de pouvoir exécuter notre algorithme au sein d'un système critique il va être nécessaire de le compiler vers un langage de plus bas niveau comme le C. Il faut compiler le code, mais aussi les triplets de Hoare, car notre but est de justement certifier le code qui sera généré. C'est pourquoi dans le chapitre suivant nous construirons un compilateur permettant à la fois de générer du code, mais aussi ses annotations.

Bibliographie

Açikmese, Behçet, John M. III Carson et Lars Blackmore (2013). "Lossless Convexification of Nonconvex Control Bound and Pointing Constraints of the Soft Landing Optimal Control Problem". In : *IEEE Trans. Contr. Sys. Techn.* 21.6, p. 2104-2113 (cf. p. 76).

Bemporad, Alberto, Francesco Borrelli et Manfred Morari (déc. 2002). "Model predictive control based on linear programming - the explicit solution". In : *IEEE Transactions on Automatic Control* 47.12, p. 1974-1985 (cf. p. 4, 5, 76).

Blackmore, Lars (2016). In : *IEEE Control Systems Magazine* 36.6, p. 24-26 (cf. p. 3, 76).

Cohen, Raphael, Guillaume Davy, Eric Feron et Pierre-Loic Garoche (2017). "Formal Verification for Embedded Implementation of Convex Optimization Algorithms". In : *IFAC-PapersOnLine* 50.1, p. 5867-5874 (cf. p. 6, 94, 164).

Feron, Eric M., Raphael P. Cohen, Guillaume Davy et Pierre-Loic Garoche (jan. 2017). "Validation of Convex Optimization Algorithms and Credible Implementation for Model Predictive Control". In : *AIAA Scitex* (cf. p. 6, 94, 164).

Gearhart, Jared Lee et al. (oct. 2013). "Comparison of open-source linear programming solvers". In : *Technical report, Sandia National Laboratories* (cf. p. 78).

Khachiyan, Leonid (1979). "A polynomial algorithm in linear programming". In : *Akademiia Nauk SSSR. Doklady* 244, p. 1093-1096 (cf. p. 78).

Mattingley, Jacob et Stephen Boyd (2009). "Automatic code generation for real-time convex optimization". In : *Convex Optimization in Signal Processing and Communications*. Sous la dir. de Daniel P. Palomar et Yonina C.Editors Eldar. Cambridge University Press, p. 1-41 (cf. p. 76).

Nesterov, Yurii (2004). *Introductory lectures on convex optimization : a basic course*. Applied optimization. Boston, Dordrecht, London : Kluwer Academic Publ. (cf. p. 79, 85-87, 92, 93, 120-122).

Nesterov, Yurii et Arkadi Nemirovski (1994). *Interior-point Polynomial Algorithms in Convex Programming*. T. 13. Studies in Applied Mathematics. Society for Industrial et Applied Mathematics (cf. p. 85, 120).

Spielman, Daniel A et Shang-Hua Teng (2004). "Smoothed analysis of algorithms : Why the simplex algorithm usually takes polynomial time". In : *Journal of the ACM (JACM)* 51.3, p. 385-463 (cf. p. 78).

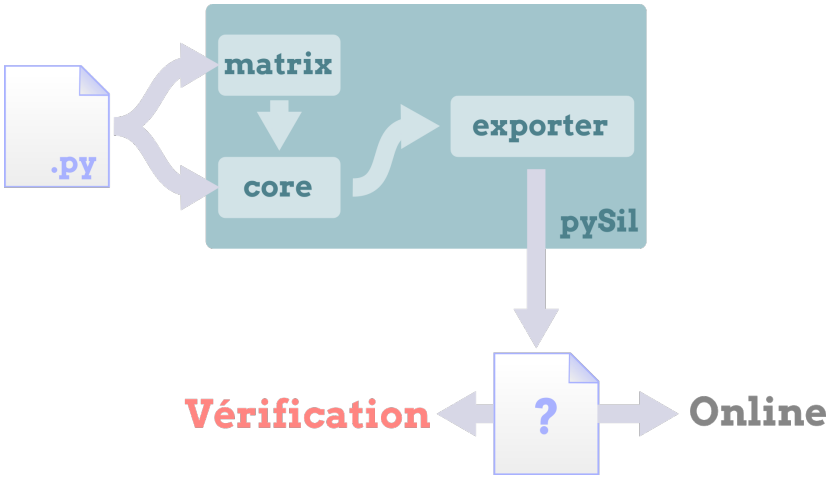
Chapitre 5

Développement d'un compilateur de pySil vers un langage annoté et embarquable

Sommaire

Introduction

Ce chapitre est consacré à la génération à partir d'un script pySil de code embarquable et prouvable comme représenté dans la figure 5.1. Le but est d'avoir toutes les garanties de corrections possibles pour embarquer un algorithme d'optimisation en ligne dans un contexte critique. La première étape consistera à choisir un cadre logiciel permettant la validation d'annotations sur un code embarquable. On commencera par choisir le langage cible puis la méthode d'annotation et enfin les outils permettant la preuve du code. Dans la deuxième section, on développera un générateur de pySil vers les langages choisis dans le but d'obtenir un code embarquable et prouvable. Pour cela, nous développerons un itérateur sur l'arbre syntaxique pySil. Cela permettra ensuite de construire une base modulaire pour l'exportation, ce qui simplifie le développement ainsi que l'export vers d'autres langages. Nous passerons ensuite à l´exportation proprement dite vers les langages choisis. Enfin, nous verrons comment améliorer cette exportation pour aider le travail de preuve. La troisième section sera consacrée à l'utilisation de ce compilateur sur le programme écrit au chapitre précédent. Cela permettra de tester le fonctionnement du compilateur, et la capacité à prouver le code généré. On en profitera aussi pour analyser l'efficacité des différentes étapes de la chaine de compilation.



Dans ce chapitre, on souhaite construire la partie *exportation* de PYSIL. Le but est de générer un code source annoté dont le format reste à déterminer, mais qui sera vérifiable et embarquable.

FIGURE 5.1 – Génération de codes et d'annotations prouvable

5.1 Choisir une méthode d'expression du code et de sa sémantique

Dans cette section, nous allons choisir un langage cible pour PYSIL et les outils permettant de prouver formellement du code dans ce langage. Ce langage devra être d'assez bas niveau pour faire confiance à la phase de compilation finale. Il devra aussi être doté d'un langage d'annotation par triplets de Hoare afin que l'on puisse compiler les annotations PYSIL dans ce langage et ainsi obtenir un code annoté. Le but est ensuite de prouver que le code généré satisfait les annotations générées et il faut donc qu'un framework logiciel le permettant soit disponible.

5.1.1 Choix du langage cible

Nous souhaitons compiler PYSIL vers un langage de plus bas niveau. Il n'est pas question ici de cibler l'assembleur. L'assembleur correspondrait plus à l'étape suivante et représente un travail indépendant qui a pu déjà être effectué dans LEROY, 2009b ou SEWELL, MYREEN et KLEIN, 2013. Ainsi nous allons cibler un langage intermédiaire dans lequel il est naturel d'écrire du code embarqué. Il devra être impératif afin de rester sur une structure proche du code exécuté par la machine et donc, surtout, des annotations. Les langages basés sur d'autres paradigmes, comme les langages orientés objet ou fonctionnels, n'ont pas été retenus, car ils rajoutaient des étapes de compilations après notre preuve. On a aussi éliminé des langages trop récents comme Rust, car ne possédant pas encore d'outils de vérification matures. Il est resté deux langages : C et Ada.

Le langage que nous avons retenu est le C. C'est le langage le plus utilisé pour l'embarqué ce qui lui permet d'avoir d'un écosystème très complet en particulier dans le domaine des méthodes formelles. Par exemple, LEROY, 2009b et SEWELL, MYREEN et KLEIN, 2013 proposent chacun une méthode pour vérifier formellement un binaire généré à partir de code C. L'écosystème inclut aussi un certain nombre d'outils permettant de valider du code écrit en C que nous allons comparer et choisir dans la sous-section suivante.

5.1.2 Choix du logiciel permettant l'analyse du code source

Il existe un grand nombre de logiciels permettant l'analyse statique de code C. Par analyse statique, on entend la vérification de propriétés sur le code source d'un programme. Statique par opposition à une analyse dynamique qui consisterait à vérifier ces propriétés durant l'exécution du programme. L'analyse dynamique repose sur des outils de vérification embarqués avec le code à exécuter ce qui nécessite un système informatique de grande puissance, mais aussi d'être capable de réagir correctement aux erreurs détectées. On préférera donc en général une analyse statique nous assurant, avant l'exécution même du code, sa correction.

Parmi ces logiciels d'analyse statique, les plus courants se concentrent sur la preuve d'absence d'erreur à l'exécution. Cela peut être une division par zéro ou un accès illégal à la mémoire pour les plus courantes. Ces erreurs entrainent en général l'arrêt inopiné du programme. Au minimum, il faudra relancer un calcul sans garantie que le problème ne se reproduira pas. Dans le pire des cas, ces erreurs peuvent être le point d'entrée d'utilisateurs malveillants. C'est pourquoi prouver leur absence est primordial et on comprend les efforts déployés contre ces erreurs. Cependant, on reste sur des garanties assez faibles du point de vue des fonctionnalités du code : on a prouvé l'absence de problèmes, mais nullement la présence d'un code fonctionnel, c'est-à-dire un code qui fait ce pour quoi il a été écrit. Par exemple, soit f une fonction calculant l'inverse d'une matrice. Il est important de savoir qu'aucune division par zéro n'aura lieu lors de l'exécution de, f mais il est tout aussi important de s'assurer que $f(A)$ renverra bien la matrice inverse de A .

Ainsi, il faut, dans un premier temps, pouvoir spécifier ce qu'on attend d'une fonction puis, dans un second temps, prouver que la fonction vérifie bien sa spécification. Spécifier un programme peut se faire à plusieurs niveaux. D'un côté le code lui-même peut être vu comme une forme de spécification, de l'autre côté un cahier des charges informelles peut lui aussi être considéré comme une spécification. Ainsi il faut trouver le juste milieu en revenant à ce qu'on attend d'une spécification. On attend deux choses de la spécification : être à la fois lisible par un humain et un programme. Elle doit être lisible par un programme pour permettre de prouver automatiquement que le code la satisfait. Mais elle doit être lisible par un humain pour qu'il puisse affirmer sans aucune hésitation qu'elle traduit bien ce qu'il attend du programme. C'est pourquoi les annotations sont écrites dans un formalisme logique qui satisfait bien ces deux aspects. Dans notre cas, ce formalisme logique doit pouvoir parler d'opérations matricielles et d'optimisation ou permettre de définir ces notions. En effet, sans cette capacité il est difficile de se convaincre de la correction d'une spécification matricielle.

Par exemple dans la figure 5.2, 5.2b est beaucoup plus facile à comprendre que, 5.2a car elle est plus courte et proche de ce qu'on écrirait en mathématique : $R = A \times B$.

Pour résumer, on veut un langage permettant de déclarer de nouveaux objets, de spécifier ce qu'on attend d'une fonction et d'un logiciel permettant de prouver que le code satisfait les spécifications de ce langage.

Deux logiciels semblent adaptés à ces besoins : Frama-C et son langage d'annotation ACSL décrit à la sous-section 2.3.1 et la Verified Software Toolchain (VST) reposant sur la transformation d'un programme C en Coq et son annotation directement en Coq. Frama-C possède une extension intégrée, nommé WP, qui permet d'effectuer des calculs de plus faible précondition sur les triplets exprimés en ACSL et l'exportation du résultat vers Coq, Why3 et Alt-Ergo. Il permet aussi d'utiliser d'autre méthode de vérification comme l'interprétation abstraite avec son extension Value Analysis. Du côté de VST, la transformation d'un programme C en Coq apporte une certaine souplesse dans ce qui peut être fait, mais nécessite en échange plus de travail. Le choix s'est porté sur Frama-C, car nous l'utilisions avant et utiliser Coq représente beaucoup de travail. De plus, les annotations ACSL sont plus lisibles pour une personne extérieure au domaine de la preuve formelle. Ceci est important, car la correction finale repose sur la relecture de la spécification principale comme cela est discuté dans la sous-section 7.5.2.

```
/*@ensures \forall int i, j; 0 <= i < 4 ==> 0 <= j < 2 ==>
      R[j*4+i] == A[j*4+0]*B[0*2+i] + A[j*4+1]*B[1*2+i];
*/
void mult();
```

(a) Spécification sans support des matrices de la fonction `mult`

```
/*@ensures Mat(4, 2, R) = Mat(4, 3, A)*Mat(3, 2, B); */
void mult();
```

(b) Spécification avec support des matrices de la fonction `mult`

```
void mult() {
    for (int i = 0; i < 4; i++) {
        for (int j = 0;j < 2; j++) {
            R[j*4+i] = 0;
            for (int k = 0;j < 2; j++) {
                R[j*4+i] += A[j*4+k]*B[k*2+i];
            }
        }
    }
}
```

(c) Code de la fonction `mult`

On souhaite spécifier de la fonction `mult` calculant $A \times B$ avec A de taille $4x3$ et B de taille $3x2$. À gauche on à annoté dans une logique ne supportant pas les matrices. À droite on utilise une logique possédant une définition des matrices. Une erreur s'est glissée dans le code de gauche, est-elle facile à détecter en une simple lecture ? Imaginez sur du code bien plus complexe et des matrices bien plus grandes.

FIGURE 5.2 – Comparaison entre deux logiques : avec et sans support des matrices

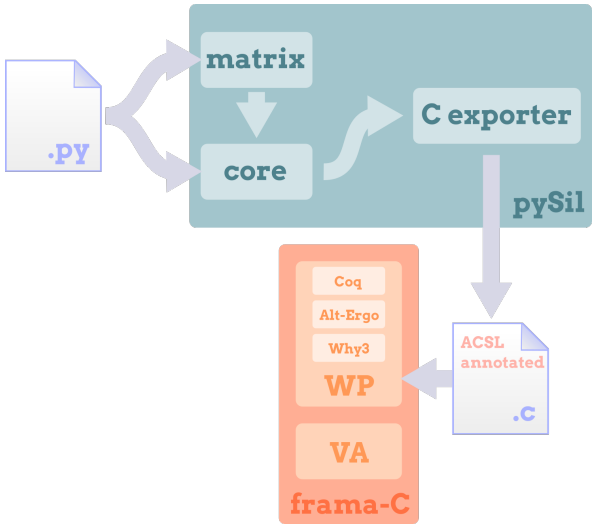


FIGURE 5.3 – Les différentes étapes menant à la génération des buts

Ainsi nous compilerons les triplets de Hoare PYSIL en annotations ACSL. Celles-ci seront ensuite analysées par le logiciel Frama-C et son extension WP permettra d'appliquer la méthode du calcul de plus faible précondition. Ces différentes étapes sont schématisées dans la figure 5.3. Le résultat est une formule logique qui, si elle est prouvée, garantit la validité du triplet. Une telle formule mathématique sera appelée un but. Il faut maintenant discuter des méthodes permettant de prouver ces buts.

5.1.3 Comment prouver des buts WP

Un but WP est une formule logique F issue du calcul de plus faible précondition d'un triplet de Hoare $\{P\}$ `C` $\{Q\} : F = P \Rightarrow \mathbf{WP}($ `C` $, Q)$. Cette formule est en général assez peu lisible et la prouver à la main sur papier serait long, fastidieux et assez peu digne de confiance. C'est pourquoi en général un tel but est exporté vers un système de preuve automatique ou assisté par ordinateur. Le plug-in WP permet l'export vers 3 systèmes : Coq, Why3 et Alt-Ergo.

Coq Coq est un assistant de preuve. Il permet d'écrire l'arbre de preuve d'un théorème et de vérifier sa correction. Un certain nombre de tactiques permettent de simplifier le travail de construction de l'arbre de preuve afin de rendre ce travail proche de l'écriture d'une preuve papier. Cependant cela nécessite quand même un travail manuel important. Ceci est en particulier vrai pour des propriétés issues d'un calcul de plus faible précondition peu naturel à prouver et nécessitant souvent de répéter un grand nombre de fois la même mécanique de preuve, par exemple pour chaque variable ou élément d'un tableau.

Alt-Ergo Alt-Ergo est un solveur SMT. Il prend donc en argument une formule puis décide de sa validité. Il peut se produire quatre évènements une fois le solveur lancé. Il peut répondre que la formule est correcte, fausse, qu'il ne sait pas ou ne pas répondre du tout. Pour éviter ce dernier cas on arrête automatique le programme après un certain temps. L'énorme avantage est que cela ne nécessite aucune intervention humaine et les solveurs SMT sont en général efficaces là où l'humain a le plus de difficulté. Par exemple, prouver quelque chose de simple un millier de fois ne posera pas plus de souci que le prouver une seule fois, au contraire de l'humain pour qui la tâche serait trop fastidieuse. En contrepartie, ils sont loin d'être capables

de tout prouver et surtout il est assez peu aisé de savoir à l'avance si une formule donnée sera prouvable ou non par un solveur donné.

Why3 Why3 est un logiciel permettant l'export d'une formule logique donnée (écrite dans un langage ad hoc) vers de nombreux solveurs SMT ou assistants de preuve comme Alt-Ergo ou Coq. Ainsi, il n'est pas un solveur par lui-même, mais apporte la possibilité d'essayer de prouver un but donné avec de nombreux solveurs différents ayant chacun leurs points forts et points faibles.

Bilan Un de nos buts est l'accessibilité de la génération de code prouvé à un public pas forcément familier avec les méthodes formelles. Ainsi quand c'est possible nous privilégions l'automatisation. C'est pourquoi nous avons fait le choix d'utiliser des solveurs SMT. Une autre solution aurait été la génération automatique de scripts de preuve comme pratiquée dans Wang, Jobredeaux, Herencia-Zapana et al., 2013. Cependant, cela représente un travail complexe par nature, et en particulier dans notre cas, il n'est pas toujours évident de prédire la structure qu'aura un but exporté vers Coq afin de générer un script correspondant. Ainsi, nous voulons utiliser exclusivement des solveurs SMT. Parmi tous les solveurs SMT nous souhaitons en utiliser un seul afin de réduire la base logicielle en laquelle nous devons avoir confiance. Toujours dans cette optique nous avons choisi le solveur SMT Alt-Ergo [1] car il est supporté nativement par Frama-C et donc ne nécessite pas d'utiliser Why3. Réduire la base logicielle possède aussi les avantages secondaires d'augmenter les chances que tous les logiciels restent compatibles et de simplifier leur installation et mise à jour.

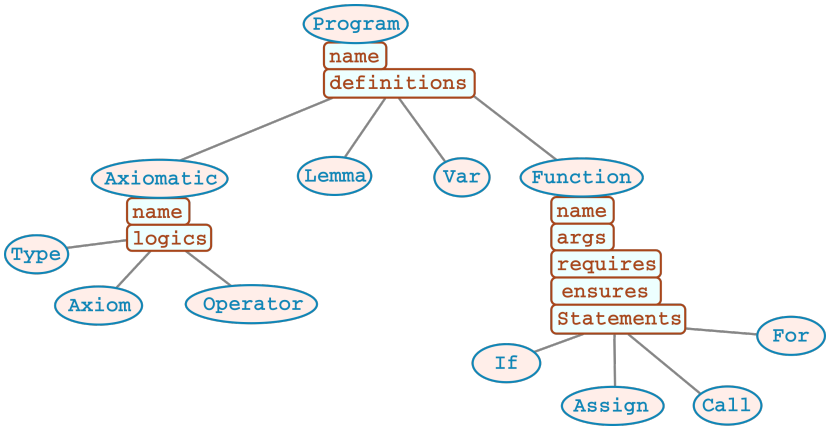
5.2 Exportation de code pySil vers le langage C/ACSL

Nous allons développer dans cette partie un générateur de code C/ACSL pour notre langage pySil. La partie code de pySil sera transformée en code C et les triplets de Hoare ainsi que les axiomatiques seront convertis en ACSL. Pour rappel, au chapitre 3 nous avons plongé l'extension matricielle dans le coeur impératif, il suffit donc d'exporter ce coeur vers le langage C. Le but est d'obtenir un code embarquable et que ce code puisse être prouvé en utilisant Frama-C/WP et Alt-Ergo. Dans cette partie, nous commencerons par présenter l'objet `Iterator` définit dans `misc/iterator.py`. Cet objet permet de parcourir un arbre syntaxique pySil. Nous présenterons ensuite le système d'exportation générique composé de deux objets : `Printer` et `Exporter` définis respectivement dans `printer.py` et `exporter.py` dans le dossier `pySil/exporter`. Nous présenterons ensuite les grandes lignes de notre exporteur vers C/ACSL.

5.2.1 Création de `Iterator`

Quand on écrit un code dans la syntaxe pySil développée au chapitre 3, on obtient un script python qui peut être exécuté. Cette exécution entraîne la génération d'un objet python associé à chaque unité syntaxique de la grammaire. Tous ces objets forment un arbre syntaxique représentant le programme pySil décrit dans le fichier python. Une version simplifiée de la structure de l'arbre est présentée dans la figure 5.4. Afin d'explorer cet arbre syntaxique, nous avons développé l'objet `Iterator` qui permet de parcourir cet arbre de manière générique. Pour utiliser cet itérateur, il faut créer une nouvelle classe, R, héritant de `Iterator` et y déclarer une méthode pour chaque classe d'objets que l'on souhaite parcourir. La méthode devra porter le nom de la classe qu'elle doit traiter et prend en argument le nœud en cours.

1. Cet objectif de n'utiliser qu'Alt-Ergo n'a pas toujours pu être respecté. Certains résultats ont été prouvés via un script Coq comme expliqué dans la section 6.4



Les ellipses représentent les classes et les rectangles les champs de ces classes.

FIGURE 5.4 – Arbre syntaxique d'un programme PYSIL

La méthode Assign de la figure 5.5 donne un exemple simple d'exploration d'un nœud de type `Assign` représentant l'opérateur `%=` défini à la sous-section 3.2.3.

Il est de la responsabilité de chaque méthode de R de lancer le parcours de ces enfants via la méthode `_iter` de la classe `Iterator`. Cette méthode prend en argument un nœud `n` de type `C` et appel automatiquement la méthode s'appellant `C`. Si cette méthode n'existe pas, elle tentera d'appeler une méthode portant le nom des classes parentes de `C` et ainsi de suite. Si aucune méthode n'est trouvée, une erreur sera renvoyée.

Ce système est utilisé pour parcourir toutes les unités syntaxiques y compris les expressions. Ceci peut entraîner un très grand nombre d'appels récursifs. Afin d'éviter les problèmes de débordement de la pile, nous avons utilisé le système de coroutine de python. Les appels récursifs se font à travers l'opérateur yield. Cet opérateur permet d'interrompre la fonction en cours tout en permettant de la reprendre plus tard. Ainsi, à chaque nouvel objet à explorer, au lieu de rentrer dans une nouvelle méthode on arrête l'exécution avec un **`yield`**. La main est alors rendue à `Iterator` qui s'occupe de lancer l'exploration du nouvel objet et de reprendre l'exploration de l'ancien une fois le nouveau terminé.

La figure 5.5 présente un exemple très simple d'exploration consistant à lister toutes les assignations d'une fonction. Pour cela, on définit **`Function`** qui explore les fonctions en les considérants comme un `BlockInner` grâce à un appel récursif. `BlockInner` est l'objet représentant une liste d'instruction, c'est un parent de **`Function`** mais aussi de toutes les structures de contrôle comme le **`For`**. C'est pourquoi il est suffisant d'explorer `BlockInner` pour explorer toutes les instructions d'une fonction. En l'occurrence, seules les assignations nous intéressent donc on ignore les autres instructions en définissant `Statement` comme ne faisant rien.

Une fois la classe héritant de `Iterator` définie on peut créer un objet de cette classe et l'appliquer directement à n'importe quel nœud syntaxique. Cela entraîne l'exploration de ce nœud et renvoie le résultat associé. La figure 5.6 montre l'utilisation de `Exploration` sur l'exemple de la fonction de Fibonacci présenté dans la sous-section 3.2.3.

La classe `Iterator` est utilisée pour implémenter une classe `Subst` qui parcourt l'arbre syntaxique afin de remplacer une ou plusieurs variables par des valeurs. Cette classe est implémentée dans **`pySil/misc/subst.py`**. `Iterator` est aussi utilisé dans la classe `Inline`(**`pySil/misc/inline.py`**) qui permet de remplacer un appel de fonction par le code de ladite fonction. Il permet aussi de dérouler les boucles ou les structures conditionnelles. La classe `Simul`(**`pySil/misc/simul.py`**) utilise aussi `Iterator`, mais cette fois pour exécuter le code

```python
from pySil.misc.iterator import Iterator

class Exploration(Iterator):
    def Function(self, noeud):
        print("Assignation dans la fonction {} :".format(noeud))
        yield self._iter(noeud, self.BlockInner)

    def BlockInner(self, noeud):
        for n in noeud.statements:
            yield self._iter(n)

    def Assign(self, noeud):
        print("  Assignation de la valeur {0} a {1}".format(
          noeud.expr, noeud.var))
```

Cette classe parcourt l'arbre syntaxique d'une fonction et affiche un message pour chaque assignation.

FIGURE 5.5 – Exploration des assignations du code PYSIL

```python
from fibonnacci import compute_fib
from exploration import Exploration
ex = Exploration()
ex(compute_fib)
```

Sortie

```
Assignation dans la fonction Function(compute_fib) :
  Assignation de la valeur C(1) a compute_fib_ar[0]
  Assignation de la valeur C(1) a compute_fib_ar[1]
  Assignation de la valeur
  ↪  __add__(compute_fib_ar[compute_fib_i0],compute_fib_ar[__add__(compute_fib_i0,C(1)
  ↪  => Int)] => Int) a compute_fib_ar[__add__(compute_fib_i0,C(2) => Int)]
```

Utilisation d'`Exploration` définie dans la figure 5.5 sur `compute_fib` définie dans la sous-section 3.2.3. On voit bien apparaitre les 3 assignations de la fonction `compute_fib`, les deux d'initialisation et celle de la boucle **`For`**.

FIGURE 5.6 – Exécution de la classe issue de la figure 5.5

PySil directement, sans le compiler. La dernière utilisation de `Iterator` se situe dans le code d'exportation que nous allons détailler dans la sous-section suivante.

5.2.2 Gestion générique de l'exportation

La classe `Printer` permet l'exportation vers un fichier texte. Les lignes sont écrites au fur et à mesure dans le fichier afin d'éviter toute concaténation de chaine de caractère. La classe gère aussi les indentations. Afin de simplifier son utilisation la classe surcharge certains opérateurs : `>>=` pour ajouter une ligne de texte et indenter le texte qui suivra, `<<=` pour faire l'inverse et `|=` pour simplement ajouter une ligne.

Une classe `Exporter` est aussi introduite. Elle sert de base pour toutes les classes d'exportation. Elle hérite de `Iterator` et `Printer` et s'occupe de l'affichage de l'avancement de l'exportation.

Trois exporteurs ont été créés chacun dans un dossier présent dans `pySil/exporter/`. Le premier `LatexExporter` permet d'exporter le code PySil vers LaTeX, ceci est principalement utile pour les définitions des opérateurs et lemmes. Il est utilisé pour faciliter leur relecture. Le deuxième est le `ReportExporter` qui permet de générer un rapport XML. Ce rapport contient une version colorée et structurée du code généré. De plus, couplé à la feuille de style `pySil/scripts/report.css` il permet d'afficher dans un navigateur web un affichage clair des fonctions et de leur contrat. Le dernier exportateur : `CExporter` est celui générant le C/ACSL et fait l'objet de la section suivante.

5.2.3 Exportation vers C/ACSL

Dans cette section nous allons présenter les grandes lignes de l'exportation vers C/ACSL. On ne rentrera pas dans la plupart des détails de l'exportation. En effet, nous partons d'un code impératif avec une sémantique définie donc la transcription en C se fait assez naturellement. On trouvera tous les fichiers relatifs à cet exportateur dans le dossier `pySil/exporter/c_exporter/`. La transcription des différentes unités syntaxiques se répartit dans 3 fichiers : `definitions.py`, `statements.py` et `expressions.py` et l'exportation de la partie logique : triplet et axiomatique se situent dans le fichier `hoare.py`. On peut regrouper les structures C/ACSL générées par ces fichiers en 5 types : les axiomatiques, les lemmes, les contrats de fonctions, les variables et les fonctions.

Dans la partie 2.3.1, on a vu qu'un contrat ACSL était composé de préconditions et de postconditions, mais aussi d'une directive *assigns*. Cette directive renseigne sur les variables qui ont été modifiées par la fonction en question. Ceci permet à FramaC/WP de savoir quelles variables n'ont pas été modifiées par l'appel d'une fonction. Nous n'avons pas introduit une telle syntaxe en PySil, car cette information peut se calculer automatiquement dans le cas des algorithmes que nous utilisons. On peut donc pour chaque fonction, regarder les `Assign` présents dans ses instructions et en déduire toutes les variables qui sont modifiées. Ce processus est assez simple hormis pour le cas des tableaux dont l'accès dépend de la valeur d'une variable. Pour l'instant, nous nous contentons de considérer que tout le tableau a été modifié. Cependant, la plupart du temps, ces variables sont issues d'une boucle **`For`**, ainsi on connait l'intervalle où elle va varier, donc on pourrait être beaucoup plus précis.

Nous avons, dans un premier temps, exporté toutes les structures logiques et les déclarations dans un fichier d'en-tête C et chaque fonction dans un fichier de code C. Indépendant qui inclut l'en-tête contenant la logique. On appellera ce découpage la configuration 1, elle est représentée dans la figure 5.7. Ce découpage, bien que satisfaisant pour la compilation, n'est pas suffisant quand il s'agit de prouver que le programme est correct.

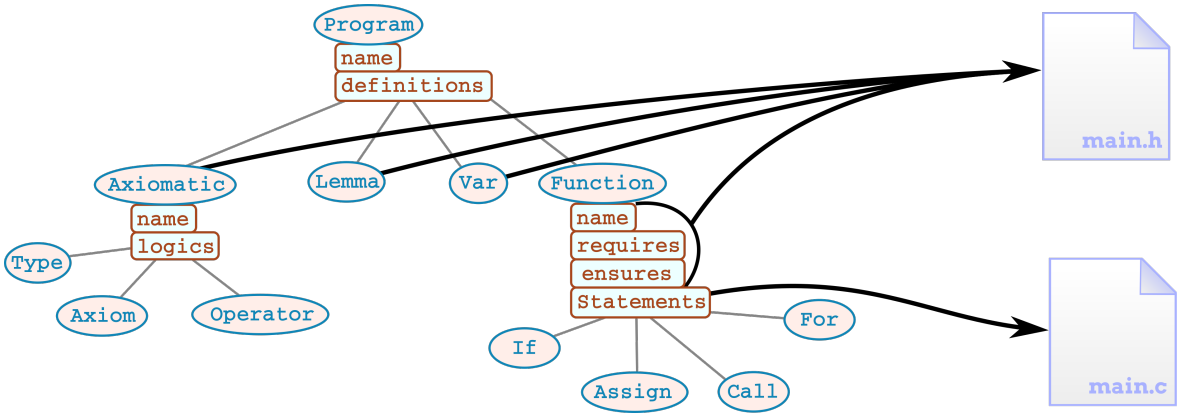


FIGURE 5.7 – Exportation monofichier de l'arbre syntaxique PYSIL (configuration 1)

Remarque :

À propos des matrices en ACSL Nous avons dans la section 3.4 développé une librairie matricielle en PYSIL qui est donc automatiquement exportée par la suite vers ACSL. Une autre solution aurait été d'utiliser une librairie ACSL des matrices comme celle proposée dans HERENCIA-ZAPANA et al., 2012. Cependant, cette dernière semblait trop spécifique aux problèmes que l'article traite et ne contenait pas tous les opérateurs qui nous étaient nécessaires. De plus, avoir la librairie complète écrite en PYSIL possède l'avantage d'être exportable dans d'autres langages. On verra aussi dans le chapitre suivant qu'il est possible d'utiliser les axiomes ou lemmes PYSIL comme des formules logiques afin de les décomposer ou de les combiner pour en former de nouvelles. Ceci n'aurait pas été possible avec une librairie directement écrite en ACSL.

5.2.4 Réduire le contexte

Dans cette partie, nous allons mettre en évidence que la configuration 1 n'est pas satisfaisante pour prouver la correction d'un programme et nous verrons comment contourner ce problème. Nous allons commencer par définir ce qu'on entend par un programme correct.

Définition 5.2.1 *Soit $F, F_1, ..., F_n$ des formules du premier ordre comme manipulées par Frama-C/WP. On notera $F_1, ..., F_n \vdash F$ la formule $F_1 \Rightarrow ... \Rightarrow F_n \Rightarrow F$.*

Dans notre cas on vérifie la correction de $F_1, ..., F_n \vdash F$ par un appel à Alt-Ergo.

Définition 5.2.2 (Programme à prouver) *On définit un programme $p(A, L, C, n, m, l)$ à prouver par Frama-C/WP comme la donnée des formules du premier ordre suivantes : $A_1, ..., A_n$ ses axiomes, $L_1, ..., L_m$ ses lemmes (dans l'ordre de définition) et $C_1, ..., C_l$ ses contrats, c'est-à-dire le résultat du calcul de plus faible précondition.*

Un programme est correct si $\forall i \in [1, l]$ $A_1, ..., A_n \vdash C_i$.

Soit $p(A, L, C, n, m, l)$ un programme à prouver. Si on génère les fichiers selon la configuration 1, Frama-C générera les buts suivants :

$$A_1, ..., A_n, L_1, ..., L_{i-1} \vdash L_i \text{ avec } i \in [1, m] \tag{5.1}$$

$$A_1, ..., A_n, L_1, ..., L_m \vdash C_i \text{ avec } i \in [1, l] \tag{5.2}$$

Théorème 5.2.1 *Si on suppose les buts* (5.1) *et* (5.2) *corrects alors p est un programme correct.*

Preuve: Si un des A_i est faux alors p est trivialement correct. Supposons les A_i vrais, alors par application du modus ponens on prouve de proche en proche que les L_i sont tous vrai. En appliquant m fois le modus ponens on arrive à prouver les C_i. □

On a donc $m + l$ buts de taille $m + n$. Ainsi, quand m augmente, la taille des buts augmente elle aussi. On pourra constater dans le chapitre 6 que m va rapidement augmenter, c'est-à-dire que l'on devra prendre en compte un grand nombre de lemmes. Or les solveurs SMT sont très sensibles à la taille du contexte. Un certain nombre de mécanismes comme les déclencheurs permettent de réduire le problème. Cependant, dans notre cas, cela ne s'est pas révélé suffisant en particulier quand nous avons augmenté la taille des problèmes.

Afin d'éviter ce problème, il faut réduire au maximum le contexte de preuve : le nombre de lemmes et axiomes utilisés pour réaliser la preuve. Pour cela, il faudrait que les fichiers donnés à Alt-ergo ne contiennent que les lemmes qui seront nécessaires à la preuve du but. Nous avons intégré un opérateur dans PYSIL qui permet de lister les lemmes ou axiomes dont dépend un lemme ou un contrat : l'opérateur `addDep` dont on peut trouver la définition dans la sous-section 3.2.4.

Ainsi pour chaque lemme L_i on suppose que l'on a $\lambda_i \subset [1, m]$ tel que L_i dépend des L_j pour $j \in \lambda_i$ de même pour chaque contrat C_i on a $\gamma_i \subset [1, m]$ représentant ses dépendances.

On souhaite maintenant pouvoir se contenter de prouver les buts suivants :

$$A_1, ..., A_n, (L_j)_{j\in\lambda_i} \vdash L_i \text{ avec } i \in [1, m] \tag{5.3}$$

$$A_1, ..., A_n, (L_j)_{j\in\gamma_i} \vdash C_i \text{ avec } i \in [1, l] \tag{5.4}$$

En émettant l'hypothèse qu'un lemme ou un contrat ne dépendent pas de plus de M lemmes avec M une constante, on se retrouve avec $m + l$ buts de taille au pire $M + n$. La taille des buts devient donc indépendante du nombre de lemmes. Ainsi, pour M pas trop grand, cela réduira grandement le temps de preuve de chaque but et, comme le nombre de buts n'a pas augmenté, le temps de preuve totale.

Il reste à prouver que les buts (5.3) et (5.4) suffisent pour prouver que p est correct, et, à défaut, trouver quelles hypothèses il faut faire pour qu'ils soient suffisants.

Définition 5.2.3 (Clôture transitive des dépendances)

$$\begin{aligned} D\colon \mathcal{P}([1,m]) &\to \mathcal{P}([1,m]) \\ X &\mapsto X \cup \bigcup_{x\in X} \lambda_x \end{aligned}$$

On définit la clôture transitive des dépendances d'un lemme L_i, l'ensemble $D_i = \lim_{n\to+\infty} D^n(\{i\})$.

D_i est bien définie, car la suite possède une limite. En effet, elle est croissante et bornée. Elle est même stationnaire, car nous travaillons dans des ensembles finis.

Théorème 5.2.2 *Si on suppose les buts* (5.3) *et* (5.4) *corrects et que pour tout i, $i \notin D_i$ alors p est un programme correct.*

Preuve: On peut se ramener au théorème 5.2.1. Ainsi, on suppose (5.3) et (5.3) et on veut prouver (5.1) et (5.2). Pour l'équation (5.2), on peut tout simplement enrichir les hypothèses avec tous les lemmes. Pour l'équation (5.1), il faut se donner un ordre sur les lemmes. La structure formée par les λ_i est un arbre grâce à l'hypothèse $i \notin D_i$, on peut alors effectuer un parcours en largeur de l'arbre pour former une suite de lemmes qui convient. □

On a donc montré qu'il suffisait de prouver les buts (5.3) et (5.4) pour prouver que p est correct à la condition qu'il n'y ait pas de cycle dans les dépendances écrites dans le programme pySil. En d'autres termes que les lemmes forment un arbre, dit de preuve.

Il reste à voir comment implémenter cette réduction du contexte. On peut le faire à postériori, cela est cependant dangereux, car il ne sera pas aisé de montrer que l'on a modifié correctement les buts générés par Frama-C. On peut aussi faire relire les buts par un humain ce qui est très loin de notre volonté d'automatisation. Une autre solution serait de demander à Frama-C de générer les buts conformément aux équations (5.3) et (5.4). Nous n'avons trouvé aucun moyen simple d'effectuer ceci, c'est-à-dire sans écrire un plug-in pour Frama-C. Par contre, au niveau de pySil, on peut générer un fichier par but voulu et prouver ainsi chaque fichier indépendamment.

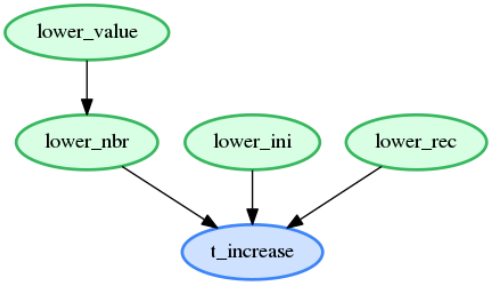
Cela consiste à sortir toutes les définitions de lemme du fichier en-tête. On rajoute ensuite à chaque fichier contenant une fonction les lemmes nécessaires à la preuve de son contrat. On génère aussi un fichier par lemmes avec, une nouvelle fois, les lemmes dont il dépend rajoutés comme axiome au fichier. On appellera configuration 2 ce découpage dont un exemple est présenté dans la figure 5.8.

Bilan La phase d'exportation vers le langage C/ACSL consiste dans un premier temps à générer un fichier en-tête contenant les axiomatiques, les définitions de variable et les contrats. Il n'y a aucun but à prouver dans ce fichier. Dans un second temps, on génère un fichier pour chaque lemme contenant le lemme à prouver ainsi qu'une axiomatique contenant les lemmes dont il dépend définis comme axiomes. Enfin, dans un troisième temps, on génère un fichier par fonction du programme qui contient, lui aussi, les lemmes dont il dépend définis comme axiomes. Cette étape est schématisée dans la figure 5.9.

Si Alt-Ergo arrive à prouver tous les buts générés par Frama-C/WP sur le programme exporté, s'il n'y a pas de boucle dans les dépendances et s'il y a bien un fichier pour chaque lemme alors le programme est correct. Ces 3 vérifications sont garanties par le compilateur pySil. Ainsi, cette partie du compilateur doit être vérifiée ou bien les 3 vérifications doivent être faites de manière indépendante. On a donc une condition suffisante pour la correction d'un programme. En revanche, la condition n'est pas nécessaire, car nous n'avons aucune garantie que Alt-Ergo pourra prouver les buts qu'il reçoit.

```python
with Program("lower") as p:
  with Axiomatic("AxLower"):
    lower = Operator("lower", Real, [Int],
      lambda x: a * pow(Real.Cst(q), x-1))

  lower_rec = Lemma("lower_rec", forall(Int, lambda k:
    (k > 1).implies(lower(k) == q * lower(k-1))))
  lower_ini = Lemma("lower_ini", lower(1) == a)
  lower_value = Lemma("lower_value",
                  lower(NBR) >= floor(a * pow(q, NBR-1)))
  lower_nbr = Lemma("lower_nbr", lower(NBR) >= delta/epsilon)
  lower_nbr.addDep(lower_value)

  with Function("t_increase") as t_increase:
    t = Real.Var('t')
    t %= gamma / NCCA

    with Function("update_t") as update_t:
      update_t.addRequire(t > 0)
      update_t.addEnsure(t > old(t) * (1 + gamma/(beta + sqrt(m))))
      t %= t + t * (gamma/(beta + sqrt(m)))

    with For(1, NBR) as l:
      addInvariant(t > lower(l))
      update_t()

    t_increase.addEnsure(t > delta / epsilon)
    t_increase.addDep(lower_nbr, lower_rec, lower_ini)
```

(a) Ce code augmente t à chaque itération de manière semblable à l'algorithme original. On souhaite prouver, qu'à la fin de la boucle **For**, $t > \frac{\delta}{epsilon}$. Pour cela, conformément à la sous-section 4.3.2, on montre qu'à chaque itération, t est toujours au-dessus de la suite géométrique `lower`.



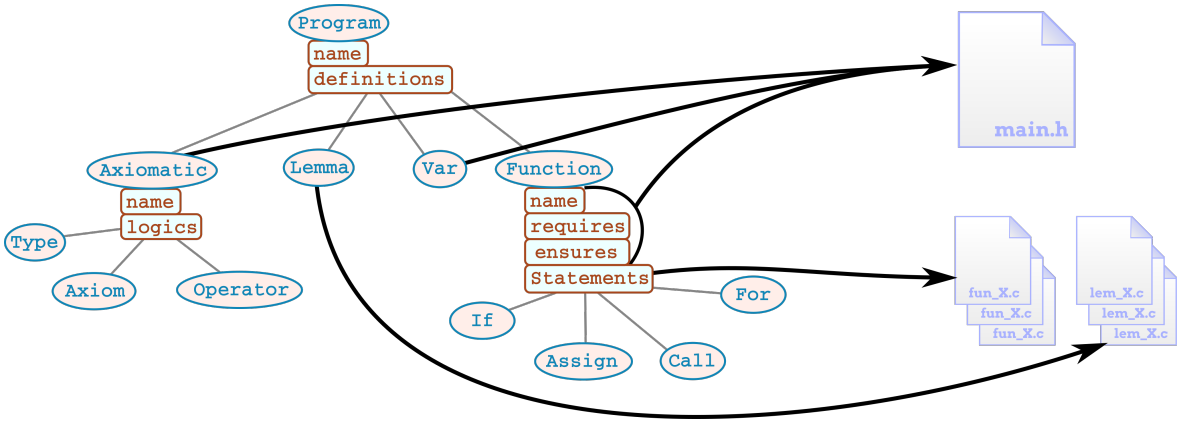
(b) Voici l'arbre de preuve généré par le script 5.8a. *lower_ini* est utile pour l'initialisation de l'invariant et *lower_rec* pour sa préservation. *lower_nbr* et *lower_value* correspondent à la minoration de t en fin de boucle.

```
generated
├── lem_lower_ini.c
├── lem_lower_value.c
├── lem_lower_nbr.c
├── lem_lower_rec.c
├── fun_t_increase.c
└── fun_update_t.c
```

(c) Liste des fichiers générés lors de l'exportation du script 5.8a

Le script 5.8a est la partie correspondant à l'évolution de t dans l'algorithme du chapitre 4. La preuve nécessite plusieurs lemmes dont les dépendances sont représentées dans la figure 5.8b. Ceci nous amène à la génération de 6 fichiers listée dans la figure 5.8c.

FIGURE 5.8 – Preuve d'une fonction PYSIL exporté en C



Chaque fonction et chaque lemme est exporté dans un fichier distinct.

FIGURE 5.9 – Exportation multifichier de l'arbre syntaxique PYSIL (configuration 2)

5.3 Exportation de l'algorithme de points intérieurs

On a donc développé dans ce chapitre un compilateur de PYSIL vers C/ACSL. Ce compilateur doit permettre, à la fois de produire un code embarquable et de rendre possible la preuve de ce code. Nous allons tester ces aspects sur le code PYSIL de l'algorithme de points intérieurs développé au chapitre 4 et résumé dans la figure 4.10.

Nous allons exécuter le code de génération sur des problèmes de tailles croissantes. Les résultats permettront de se donner une idée des temps de calcul pris par l'exportation vers C, la compilation du code C et enfin l'exécution. Nous utiliserons ensuite Frama-C afin de constater quels buts il peut prouver sans effort supplémentaire.

5.3.1 Expérimentation sur la génération du code

Le tableau 5.11 présente les résultats de l'expérimentation. Il n'aborde pas la partie preuve qui sera discutée dans la section suivante. On voit dans ce tableau qu'à partir d'un nombre de variables supérieur à 10 le temps de génération du code C dépasse la seconde. À partir d'un nombre de variables de 250 on dépasse les 5 minutes. Ces temps peuvent sembler important cependant ils s'effectuent hors-ligne donc les temps de calcul ne sont pas un problème en soit, mais cela complique toujours les tests. Cela est dû à l'utilisation de mécanisme du langage python comme les métaclasses qui complexifie l'exécution, et au précalcul/simplification de certaines expressions mathématiques. L'exporteur n'a pas été écrit dans l'optique d'être le plus rapide donc des efforts pourraient être menés dans cette direction.

On voit que la compilation du code C en $-O0$ prend plus d'une seconde à partir d'un nombre de variables de 50. Cette augmentation du temps de compilation est due aux opérations matricielles qui sont effectuées élément par élément et non dans une boucle. Cela permet de limiter les calculs inutiles. Les temps de compilation même longs restent cependant acceptables avec l'option d'optimisation minimale. En revanche, on risque de ne pas pouvoir utiliser l'option d'optimisation de gcc maximale pour de gros problèmes, car le temps de compilation deviendrait trop grand.

Les temps d'exécution du code final sont très importants comparés à ce que l'on pourrait attendre d'un algorithme de points intérieurs. En particulier, dans le cadre du MPC, il faudrait des temps au minimum inférieur à la seconde voir à la centième de seconde. Cela était un résultat attendu et discuté dans la sous-section 4.1.3 et il est dû à l'utilisation d'un algorithme primal et non primal/dual. On prend aussi plus de temps que ce qui pourrait être fait avec un

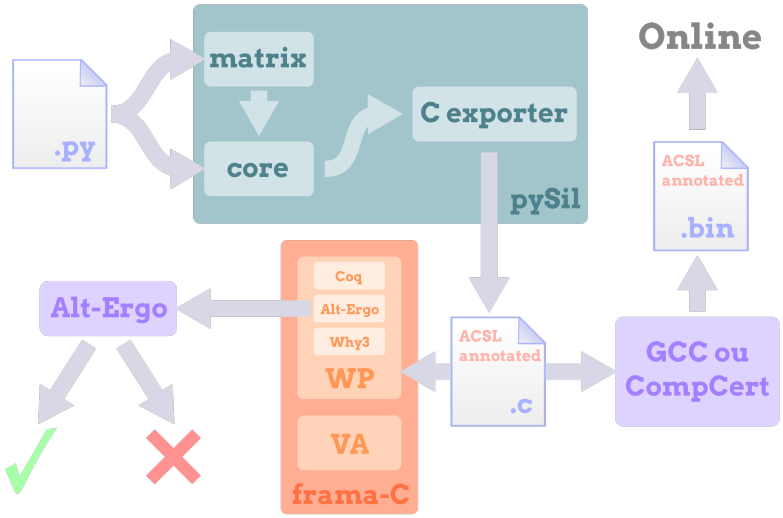


FIGURE 5.10 – Transformation d'un code PYSIL en exécutable dont le code C est prouvé

Taille du problème	2x5	2x40	10x25	10x200	50x125	50x1000	250x550
Parsing	0.080	0.111	0.182	0.830	2.966	19.987	64.441
Exportation	0.348	0.421	0.874	2.654	8.751	60.968	253.073
Compilation(gcc -O0)	0.282	0.285	0.317	0.474	1.252	7.019	41.032
Exécution (gcc -O0)	0.004	0.004	0.012	0.040	1.932	8.144	
Compilation(gcc -O3)	0.322	0.372	0.681	48.602	67.749		
Exécution (gcc -O3)	0.004	0.004	0.006	0.014	0.475		
Erreur optimalité	0	6e-07	6e-07	6e-07	1e-06	5e-02	
Faisabilité	0	-2e-04	-5e-05	-8e-04	-1e-03	-2e-03	

On a exporté vers C/ACSL l'algorithme primal pour 7 problèmes de taille différente puis on l'a compilé avec gcc et enfin exécuté. On a mesuré le temps en seconde de chaque étape. Les deux premières lignes correspondent à l'outil que nous avons développé. Les deux suivantes correspondent à la compilation et l'exécution du code C avec gcc sans optimisation. Il en va de même pour les deux suivantes, mais avec l'optimisation maximale de gcc. L'avant-dernière ligne donne la différence entre notre valeur optimale et celle trouvée par numpy. Enfin, la dernière ligne correspond à la plus petite valeur du vecteur $b - A * x_{sol}$. C'est un indicateur de la satisfaction des contraintes. S'il est positif ou nul, les contraintes sont satisfaites. À l'opposé, plus il est négatif, plus cela signifie que les contraintes sont violées. Tous ces tests ont été effectués pour $\epsilon = 0.0001$ et sur un ordinateur portable muni d'un processeur *Intel(R) Core(TM) i5-6300U CPU @ 2.40GHz* couplé à 8*Go* de RAM.

FIGURE 5.11 – Temps de génération/compilation/exécution pour différents problèmes.

Taille du problème	2x5	2x40	5x40	10x40
fun_cholesky_PF2	27/112	27/112	25/112	25/112
fun_compute	1/3	1/3	1/3	1/3
fun_compute_pre	18/77	18/77	17/77	17/77
fun_pathfollowing	9/17	9/17	9/17	9/17
fun_update_t	2/4	2/4	2/4	2/4
fun_update_x	3/5	3/5	3/5	3/5
Temps total	27s	31s	54s	178s

On remarque que peu de buts sont prouvés. C'est pourquoi, dans le chapitre on introduira des lemmes intermédiaires afin que ces buts soient prouvés.

FIGURE 5.12 – Nombre de buts prouvés pour différente taille de problème

algorithme primal, mais en contrepartie, on a la garantie mathématique que dans ce temps donné, une solution ϵ-optimale sera trouvée.

Concernant la correction de l'algorithme, on a calculé la différence entre la solution que notre programme retourne et celle retourné par le solveur de numpy. On voit que l'on a une erreur d'optimalité assez faible. Elle devient significative pour la taille 50×1000 mais elle peut aussi être dû au solveur de numpy. L'erreur sur la faisabilité reste aussi assez faible. Il est de toute manière délicat d'obtenir quelque chose de parfaitement faisable vu que l'optimal est justement sur la bordure. Ainsi, plus on s'approche de l'optimal, plus les erreurs de calcul dues à l'utilisation de nombres flottants vont être signigicatives.

5.3.2 Expérimentation sur la preuve du code généré

Nous avons vu comment évoluaient dans les grandes lignes la génération, la compilation et l'exécution du code. Dans cette section, nous allons reprendre la même expérimentation, mais nous concentrer sur l'aspect preuve. Plus précisément nous allons nous intéresser au nombre de buts prouvé par Alt-Ergo parmi tous les buts générés par Frama-C. Pour l'instant aucun lemme n'a été écrit et donc on essaye de prouver uniquement des contrats. Ces résultats sont présentés dans le tableau 5.12 qui complète le tableau 5.11.

Les tailles ont été adaptées, car au-delà de dix variables les temps de preuve deviennent prohibitifs. La première ligne indique la taille du problème et la dernière le temps total pris par cette étape. Les autres lignes correspondent au nombre de buts prouvés sur le nombre de buts à prouver généré par Frama-C pour chaque fichier. Chaque fichier est composé d'une unique fonction éponyme. `compute`, est la fonction principale qui appelle `pathfollowing` et `cholesky_PF2` est la fonction résolvant les équations linéaires. Les autres fonctions peuvent être retrouvées dans la figure 4.10. Les colonnes donnent le nombre de buts prouvés sur le nombre de buts totaux pour chaque taille de problème.

Remarque :

On avait considéré dans les sections précédentes qu'on associait un unique but à chaque contrat. Le tableau 5.12 nous montre qu'en pratique FramaC génère un certain nombre de buts pour chaque fichier et donc chaque contrat. Cela est dû à plusieurs phénomènes. Le premier est que Frama-C génère un but pour chaque postcondition, et un but en plus par contrat pour les `assigns`. Il génère deux buts pour chaque invariant de boucle : initialisation et préservation ainsi qu'un but en plus pour les `assigns` de chaque boucle. Il génère aussi un but pour chaque précondition de chaque appel de fonction.
Enfin Frama-C génère aussi des buts pour toutes les erreurs potentielles à l'exécution comme les accès mémoire ou les divisions. Par exemple, pour chaque division, on doit prouver que le diviseur ne peut pas être nul. Autre exemple, pour chaque entier, on doit vérifier qu'il ne dépasse pas la taille maximale de la variable. On appellera tous ces buts RTE pour *Real Time Error*.
Reprenons, par exemple, la fonction `increase_t` dont on a détaillé la preuve dans la figure 5.8. Si on exécute frama-C/WP sur le fichier **`fun_t_increase.c`** contenant la fonction `increase_t`, on obtient le retour suivant :

Sortie

```
[wp] [Qed] Goal typed_t_increase_loop_inv_established : Valid
[wp] [Qed] Goal typed_t_increase_loop_inv_preserved : Valid
[wp] [Alt-Ergo] Goal typed_t_increase_loop_inv_2_preserved : Valid
[wp] [Alt-Ergo] Goal typed_t_increase_post : Valid
[wp] [Qed] Goal typed_t_increase_assign_exit : Valid
[wp] [Qed] Goal typed_t_increase_loop_assign : Valid
[wp] [Alt-Ergo] Goal typed_t_increase_assert_rte_signed_overflow : Valid
[wp] [Alt-Ergo] Goal typed_t_increase_loop_inv_2_established : Valid
[wp] [Qed] Goal typed_t_increase_assign_normal : Valid
[wp] [Alt-Ergo] Goal typed_t_increase_call_update_t_pre : Valid
```

On voit que dix buts ont été générés, trois (lignes 5, 6 et 9) pour les `assigns`, deux pour chaque invariant (lignes 1, 2, 3 et 8) de boucle, une (ligne 4) pour la postcondition, une (ligne 10) pour la précondition de l'appel à *update_t* et enfin une (ligne 7) RTE qui vérifie qu'il n'y a pas de débordement sur la variable de boucle. Par ailleurs, on voit que la moitié des buts sont prouvés directement par Frama-C par simplification (*Qed*) et l'autre moitié par Alt-Ergo.

Si on revient au tableau 5.11, on voit qu'un certain nombre de buts sont prouvés. Les buts prouvés sont les buts `assigns` ainsi que certains buts RTE. Les contrats, les invariants de boucles et les appels de fonctions ne sont pas prouvés. Ceux-ci, ainsi que les buts RTE restants, reposent sur des résultats mathématiques complexes. Ceux-ci sont trop complexes pour être prouvé directement par un solveur SMT. Pour prouver ces buts on va devoir introduire des lemmes intermédiaires comme illustré dans la figure 5.8.

Conclusion

Dans ce chapitre, on a développé un générateur de code, prenant en entrée un code PySil et générant un code C annoté par des triplets de Hoare en ACSL. La génération se fait de manière modulaire ce qui pourrait permettre de développer des générateurs pour d'autres langages. Dans ce chapitre, on a aussi vu comment on pouvait simplifier la tâche des solveurs SMT en répartissant code et lemmes dans des fichiers indépendants. Nous avons terminé le chapitre en générant le programme écrit dans le chapitre précédent pour de nombreux problèmes linéaires. Le bilan de cette expérimentation est double. Le code fonctionne correc-

tement, mais est cependant lent. En revanche, pour la preuve, aucun des fichiers générés n'a pu être prouvé. C'est pourquoi le chapitre suivant sera consacré à enrichir le code pySil ainsi que le compilateur pour que le code d'optimisation généré soit prouvable automatiquement par Alt-Ergo.

Bibliographie

HERENCIA-ZAPANA, Heber et al. (2012). "PVS Linear Algebra Libraries for Verification of Control Software Algorithms in C/ACSL". In : *NASA Formal Methods - 4th International Symposium, NFM 2012, Norfolk, VA, USA, April 3-5, 2012. Proceedings*. Sous la dir. d'Alwyn GOODLOE et Suzette PERSON. T. 7226. Lecture Notes in Computer Science. Springer, p. 147-161 (cf. p. 107).

LEROY, Xavier (2009b). "Formal verification of a realistic compiler". In : *Communications of the ACM* 52.7, p. 107-115 (cf. p. 99).

SEWELL, Thomas Arthur Leck, Magnus O. MYREEN et Gerwin KLEIN (juin 2013). "Translation Validation for a Verified OS Kernel". In : *SIGPLAN Not.* 48.6, p. 471-482 (cf. p. 99).

WANG, Timothy, Romain JOBREDEAUX, Heber HERENCIA-ZAPANA et al. (2013). "From Design to Implementation : an Automated, Credible Autocoding Chain for Control Systems". In : *CoRR* abs/1307.2641. arXiv : 1307.2641 (cf. p. 4, 5, 103, 135).

Chapitre 6

Automatisation de la preuve du code source généré

Sommaire

Introduction

Dans ce chapitre, nous allons nous assurer que le code généré est automatiquement prouvable par le cadre logiciel choisi afin d'obtenir une méthode sans interaction humaine allant de l'expression du problème à un code prouvé. Dans la section 6.1, nous mettons en place une méthode itérative d'enrichissement du code et des lemmes afin de l'appliquer sur notre travail. Dans la section 6.2, nous allons automatiser les interactions entre notre compilateur et le cadre logiciel de preuve afin de simplifier l'utilisation de la méthode développée. Nous avons ainsi pu mettre en action la méthode. D'abord, dans la section 6.3 pour enrichir le compilateur afin de valider un maximum de preuve sur les aspects bas niveau du code, en particulier de calcul matriciel. Ensuite, dans la section 6.4, pour valider les aspects mathématiques de l'algorithme. Finalement, dans la section 6.5, nous avons expérimenté la preuve du code généré sur des exemples de tailles croissantes afin de voir comment la preuve se comporte avec de gros problèmes.

6.1 Recherche des moyens d'enrichissements de l'ACSL généré

On a vu qu'afin de prouver l'intégralité de l'algorithme de points intérieurs généré nous allons devoir enrichir nos annotations ACSL. Celles qui doivent nécessairement être présentes sont : les contrats sur les fonctions et les invariants de boucles. Cependant comme on l'a vu au chapitre précédent, cela n'est pas suffisant. On peut isoler deux grandes raisons à ce problème. La première raison est la complexité des résultats mathématiques en jeu. La seconde raison est la complexité du code source C, la différence entre le calcul au niveau matriciel des annotations et le calcul élément par élément au niveau du code C. Nous allons voir comment peut être abordée la résolution de ces deux problèmes.

6.1.1 Réduire la complexité des résultats mathématiques à prouver

Admettre certains résultats

Pour prouver la correction de l'algorithme, nous avons besoin des résultats mathématiques liés à celui-ci. En l'occurrence nous nous basons sur Nesterov, 2004 qui utilise la théorie des fonctions auto concordantes développée pour la première fois dans Nesterov et Nemirovski, 1994. Celle-ci est relativement compliquée et repose sur des résultats mathématiques fortement liés à du calcul d'intégrales. Prouver toute cette théorie de manière formelle serait une tâche à part entière que nous n'aborderons pas dans cette thèse. Ceci est discuté plus en détail dans la section 7.5.2. Nous avons donc décidé de ne pas prouver certains théorèmes. C'est-à-dire que nous allons placer comme axiome un certain nombre de résultats issus de Nesterov, 2004. Le nombre de ces théorèmes doit être aussi petit que possible. Ils devront, de plus, être assez généraux afin de rester facilement lisibles. De plus, autant que faire se peut, nous sélectionnerons des résultats présents comme lemmes ou théorèmes dans Nesterov, 2004. Cela permettra de vérifier plus facilement que le lemme généré en ACSL est correct. En effet, il sera primordial de vérifier manuellement que ces lemmes sont corrects, car rien d'autre ne le garantira. Cet aspect est discuté plus en détail dans la section 7.5.

Nous allons admettre quatre théorèmes, qui sont les quatre théorèmes de la section 4.2. Nous donnons pour chaque théorème, son numéro, le théorème correspondant dans Nesterov, 2004 et son expression en pySil.

Lien entre l'ACC et l'optimalité Le premier théorème que nous allons admettre est le théorème 4.2.1. C'est le théorème 4.1.13 de Nesterov, 2004, p.190. Il permet de lier la condi-

tion de centrage approché et l'optimalité ce qui est nécessaire pour prouver la postcondition principale.

Théorème 1 (Rappel théorème 4.2.1) *Pour tout x vérifiant $\left\|\tilde{f}'(x,t)\right\|_x^* < 1$, on a $\|x - x^*(t)\|_x < \frac{\left\|\tilde{f}'(x,t)\right\|_x^*}{1-\left\|\tilde{f}'(x,t)\right\|_x^*}$.*

```
theorem_4_1_13 = Axiom("theorem_4_1_13",
    forall(LMat, LMat, LMat, LMat, LMat, Real,
    lambda A, b, c, x, xp, t:
        (normI(A, b, grad(A, b, x) + c * t , x) < 1).implies(
        acc(A, b, c, t, xp, 0).implies(
        norm(A, b, x - xp, x) <
            normI(A, b, grad(A, b, x) + c * t , x)
            /(1 - normI(A, b, grad(A, b, x) + c * t , x))))))
```

`acc(A, b, c, t, xp, 0)` signifie que `xp` est le point du chemin central correspondant à $t : x^*(t)$. On rappelle aussi que $\tilde{f}'(x,t) = t \cdot c + B'(x)$.

Propriété de la fonction barrière Le second théorème est 4.2.2. C'est le théorème 4.2.4 de Nesterov, 2004, p.196. C'est une des propriétés fondamentales des fonctions barrières. Elle est en l'occurrence utile pour le calcul du nombre d'itérations, mais aussi pour borner $\|c\|_x^*$ à chaque itération.

Théorème 2 (Rappel théorème 4.2.2) *Pour tout x et y appartenant à E_f, $\langle B'(x), y - x\rangle < m$.*

```
theorem_4_2_4 = Axiom("theorem_4_2_4",
    forall(LMat, LMat, LMat, LMat, lambda A, b, x, y:
        grad(A, b, x).dot(y-x) < A.getM().cast(Real)))
```

Préservation de l'ACC Le troisième théorème est 4.2.3. C'est le théorème 4.1.14 de Nesterov, 2004, p.190. Il est utilisé pour démontrer la préservation de la condition de centrage approché après un pas de Newton.

Théorème 3 (Rappel théorème 4.2.3) *Pour tout x vérifiant $\left\|\tilde{f}'(x,t)\right\|_x^* < 1$, si $x_+ = x - B''(x)^{-1}(B'(x) + t \times c)$ alors $\left\|\tilde{f}'(x_+,t)\right\|_{x_+}^* \leq \left(\frac{\left\|\tilde{f}'(x,t)\right\|_x^*}{1-\left\|\tilde{f}'(x,t)\right\|_x^*}\right)^2$.*

```
theorem_4_1_14 = Axiom("theorem_4_1_14",
  forall(LMat, LMat, LMat, LMat, LMat, Real,
  lambda A, b, c, ax, xp, t:
    correct(A, b, c).implies(
    (A*ax < b).implies(
    (xp == ax - hess(A, b, ax).inv() * (grad(A, b, ax) + c*t)).implies(
    (normI(A, b, grad(A, b, ax) + c * t, ax) < Real.Cst(1.0)).implies(
      normI(A, b, grad(A, b, xp) + c * t, xp) <=
        omega(normI(A, b, grad(A, b, ax) + c * t, ax)))))))
```

`omega(x)` est définit comme $\omega(x) = \left(\frac{x}{1-x}\right)^2$

Préservation de la faisabilité Le dernier théorème est 4.2.4. C'est le théorème 4.1.5 de Nesterov, 2004, p.182. Il donne une condition suffisante pour l'appartenance à l'ensemble faisable. Il est utile pour prouver l'invariant de boucle garantissant qu'à chaque itération le point est faisable.

Théorème 4 (Rappel théorème 4.2.4) *Soit* $x \in E_f(A,b)$ *et* $y \in \mathbb{R}^n$ *tel que* $\|y-x\|_x < 1$ *alors* $y \in E_f(A,b)$.

```
theorem_4_1_5 = Axiom("theorem_4_1_5",
    forall(LMat, LMat, LMat, LMat, lambda A, b, x, y:
        (A * x < b).implies(
        (norm(A, b, y - x, x) < 1).implies(
        A*y < b))))
```

Construire un arbre de lemme

Il était donc impossible que lors des expérimentations de la section 5.3, les buts soient prouvés. En effet, les solveurs SMT, se basent énormément sur des recherches semi-exhaustives dans l'ensemble des hypothèses qu'on leur donne. Ainsi, prouver seul des buts dont la preuve repose sur des théorèmes d'analyse et d'algèbre en n'ayant quasiment qu'une axiomatique sur les réels est impossible. Il était donc nécessaire d'introduire des résultats intermédiaires. Nous avons fait le choix de les prendre assez généraux afin d'en limiter le nombre. Cela signifie que le chemin de preuve entre ceux-ci et les buts que l'on souhaite prouver sera long et encore une fois hors de portée des solveurs SMT. Cependant, on peut espérer qu'en écrivant manuellement un certain nombre de lemmes intermédiaire sur ce chemin de preuve, les solveurs SMT puissent construire automatiquement le chemin entre ces différents lemmes. On suit ainsi une méthode de preuve classique en mathématique : on veut prouver un résultat, pour cela on écrit un certain nombre de lemmes qui permettent de le prouver et on prouve ces lemmes avec les théorèmes que l'on connaissait déjà. La différence ici est que l'on se contente d'écrire les lemmes, les preuves seront faites automatiquement par les solveurs SMT.

Nous allons, pour chacun des buts à prouver, construire des lemmes intermédiaires. Ainsi, chaque but de la logique de Hoare devra reposer sur un ou plusieurs lemmes. Ces lemmes reposeront eux-mêmes sur un ou plusieurs lemmes. Et ainsi de suite jusqu'à arriver aux théorèmes que nous avons admis. Ces lemmes formeront une hiérarchie de lemme comme présenté dans la sous-section 5.2.4. Il est difficile de savoir à l'avance s'il sera possible de prouver le lemme avec Alt-Ergo(ou n'importe quel solveur SMT). C'est pourquoi nous effectuerons un travail itératif. Soit B un but(lemme ou contrat) et $L_1, ...L_n$ les lemmes nécessaires à sa preuve, on sait donc que $(L_1 \wedge ... \wedge L_n) \Rightarrow B$. Bien que ce résultat soit vrai, on suppose que les prouveurs SMT ne pourront pas prouver ce but [1].

On introduit donc un nouveau lemme L tel que $(L \wedge L_1 \wedge ... \wedge L_n) \Rightarrow B$ et $(L_1 \wedge ... \wedge L_n) \Rightarrow L$. On pourra élaguer dans les prémisses de B et de L, car, en général, seule une partie de $(L_1 \wedge ... \wedge L_n)$ est nécessaire à l'un et à l'autre. Cette étape est très importante, car on a vu à la sous-section 5.2.4 que les solveurs SMT ont des difficultés avec les contextes qui contiennent de nombreuses hypothèses. On teste alors si les deux nouveaux buts sont prouvés. Si c'est le cas, on a réussi notre objectif initial, sinon on recommence pour chaque sous-but qui n'a pas pu être prouvé. Il arrive aussi que l'on se soit trompé dans l'écriture de L, dans ce cas on devrait finir par s'en rendre compte, car à chaque itération $(L_1 \wedge ... \wedge L_n) \Rightarrow B$ est plus simple et donc à un moment l'erreur sera visible. Dans tous les cas, si une erreur se glisse dans l'écriture d'un lemme elle ne pourra mener à une erreur non détectée si on respecte bien les conditions énoncées dans la sous-section 5.2.4.

1. Ces expressions logiques sont exprimées en pySil, mais seront prouvées sur le code C/ACSL généré par notre exporteur

6.1.2 Réduire la complexité du code à prouver

On a vu, dans la section précédente, comment enrichir l'ACSL avec un certain nombre de lemmes permettant de simplifier les buts à prouver. Cependant, on ne s'est concentré que sur la simplification des problèmes mathématiques. Il reste maintenant à trouver comment réduire la complexité issue du code comme les accès dans des tableaux ou les racines carrées. Cette complexité se matérialise dans un but dans l'expression de la plus faible précondition. On pourrait, comme à la sous-section précédente, introduire de nouveaux lemmes permettant la simplification petit à petit de ce résultat. Cela possède deux désavantages. Le premier est qu'il est très complexe de prévoir la structure d'un calcul WP par Frama-C et, de plus, celui-ci change au fur et à mesure des versions. Le second désavantage est que toute modification du code entrainera une modification du calcul de plus faible précondition et il sera donc nécessaire de modifier tous les lemmes intermédiaires. Ainsi, travailler avec des lemmes pour le code est une tâche fastidieuse et assez peu souple. C'est pourquoi nous avons décidé d'utiliser des mécanismes de coupure comme à la sous-section précédente, mais sur le code directement.

Utilisation des `Assert`

Une coupure sur le code se fait traditionnellement via l'utilisation du mot clef `assert` à une ligne quelconque d'un code C.

```
1 {
2     instructions;
3     //@assert prop;
4     instructions;
5 }
```

Cela sépare le code en deux parties. La partie avant la ligne qui aura pour postcondition le `assert` et la partie après qui possède maintenant une nouvelle précondition. Cependant, la sémantique de ces derniers en Frama-C ne diminue pas assez le contexte comme nous allons le voir.

Sémantique du `assert` Soit $\{P\}$ `C1;C2` $\{Q\}$ un triplet à prouver. Un tel triplet se prouve grâce au calcul de plus faible précondition avec le but suivant :

$$P \Rightarrow \mathbf{WP}(\texttt{C1;C2}, Q) \tag{6.1}$$

On suppose l'équation (6.1) trop complexe à prouver ce qui nous amène à introduire une assertion R au milieu : $\{P\}$ `C1;Assert(R);C2` $\{Q\}$. On doit désormais prouver deux buts :

$$P \Rightarrow \mathbf{WP}(\texttt{C1}, R) \tag{6.2}$$

$$(P \wedge \mathbf{WP}(\texttt{C1}, R)) \Rightarrow \mathbf{WP}(\texttt{C1;C2}, Q) \tag{6.3}$$

On a donc des buts plus simples à prouver, car (6.2) fait intervenir moins de code et le but (6.3) possède un résultat en plus dans le contexte. On retrouve bien, par analogie, la même structure qu'au chapitre précédent : avec un nouveau lemme à prouver L qui ensuite intervient dans la preuve du but original B. Cependant, comme expliqué plus tôt, il est important d'élaguer le contexte, or avec les assertions, cela n'est pas possible simplement. En effet, nous ne souhaitons pas modifier la manière de générer les buts ni les buts après leurs générations. C'est pourquoi nous utiliserons les `Assert` avec parcimonie.

On préférera, en général, utiliser l'introduction d'une nouvelle fonction. On parlera alors de coupure fonctionnelle. Elle permettra une maitrise totale du contexte avant et après coupure grâce aux préconditions et postconditions.

Coupure fonctionnelle

La coupure fonctionnelle consiste à introduire une fonction $\{P'\}$ `f()` $\{R\}$ ayant pour corps C_1. Le triplet à prouver devient alors $\{P\}$ `f();C_2` $\{Q\}$ générant 3 buts :

$$P \Rightarrow P' \tag{6.4}$$

$$P' \Rightarrow \mathbf{WP}(\texttt{C1}, R) \tag{6.5}$$

$$(P' \wedge R) \Rightarrow \mathbf{WP}(\texttt{C2}, Q) \tag{6.6}$$

On voit alors que dans le but (6.6) C_1 n'apparait plus. Ceci le rend bien plus petit que le but (6.3). En contrepartie, le but (6.4) est apparu. On peut l'éviter en prenant, $P' = P$ mais il permet justement de réduire le contexte pour le but (6.5) donc cela a aussi son avantage. On a donc la possibilité d'écrire R et P' de manière à réduire le contexte au mieux, et une vraie séparation en deux du code à prouver. Par contre la mise en place est un peu plus complexe qu'un simple **`Assert`**. Ce dernier point est compensé par la modularité et la souplesse de pySIL qui simplifieront cette tâche.

6.1.3 Méthode itérative d'enrichissement de l'ACSL généré

On a développé d'un côté une méthode pour enrichir de lemmes la partie mathématique des buts à prouver. De l'autre côté, on a mis en évidence deux méthodes pour réduire la complexité des buts à prouver sur la partie code. Ces deux méthodes sont l'introduction d'**`Assert`** dans le code et la coupure fonctionnelle. On a aussi montré que la coupure fonctionnelle présentait un certain nombre d'avantages ce qui nous amènera à la privilégier. Comme pour la partie mathématique, il est difficile de savoir à l'avance si la coupure sélectionnée suffira pour la preuve des nouveaux buts. Il convient donc de mettre en place une méthode itérative de même nature que pour la partie mathématique : si un but ne peut pas être prouvé alors on rajoute une fonction intermédiaire et on regarde si on arrive à prouver les buts générés. Si on n'y arrive pas alors on recommence, à rajouter des fonctions intermédiaires munies de leur contrat.

On procède donc, dans les deux cas, de manière itérative, on rajoute petit à petit des résultats intermédiaires jusqu'à ce que les buts générés soient assez simples pour Alt-Ergo. Voici la procédure complète que nous avons suivi :

1. Écrire les définitions
2. Écrire les théorèmes
3. Écrire les lemmes intermédiaires
4. Vérifier que les lemmes sont prouvés par Frama-C
   ⇒ Sinon, on retourne à 3
5. Séparer le code en plusieurs fonctions
6. Écrire un contrat pour chaque fonction
7. Vérifier que tous les contrats sont prouvés
   ⇒ Sinon retourner à 3 ou 5
8. Le code est vérifié

```
À = [ [ 3, -3],
      [ 1,  1],
      [-5, -1],
      [-1,  2],
      [ 1, -6]]
b = [[3],
     [2],
     [5],
     [2],
     [4]]
c = [[-1],
     [ 4]]
usePrimal(
    "primalTest",           # Nom du programme
    (5, 2, A, b, c, 0.01), # Problème à générer
    True,                   # Effacement des résultats précédents
    (True, True, True),     # Options de générations
    None,                   # Génération de dessin
    (1, ),                  # Option pour gcc(optimisation)
    True,                   # Est-ce qu'on doit tester le binaire
    (1, 1, True, 'Real+Nat', [], False))
                            # Options pour la preuve
```

FIGURE 6.1 – Utilisation du script d'automatisation `usePrimal`

6.2 Automatisation complète de l'utilisation du cadre logiciel de preuve depuis l'expression du programme linéaire

Nous avons mis en place dans la section précédente une méthode ayant pour but de prouver le code généré. Cette méthode est itérative, elle repose sur des aller-retour entre le code PYSIL et la preuve des buts avec Alt-Ergo. Il faut donc à chaque itération regénérer le code C/ACSL, lancer Frama-C sur chaque fichier et prouver chaque but avec Alt-Ergo. Afin de simplifier ce travail, nous avons écrit un script. Ce script permet, à partir de la donnée de A, b, c et ϵ de générer le code PYSIL, mais aussi de le compiler, lancer le code compilé et bien entendu prouver le code généré. Ainsi en une commande, on peut générer le code et le prouver.

De plus, toujours dans un souci de simplifier le travail d'enrichissement, le statut de chaque fichier : prouvé/pas prouvé est stocké dans un fichier XML. Ainsi, une option permet de lire ce fichier et de ne lancer la preuve que pour les buts manquants. Le fichier XML contient aussi un certain nombre d'informations supplémentaires, comme le temps mis pour prouver chaque fichier, le nombre de buts de chaque fichier ainsi que les dépendances de chaque fichier. De cette dernière information, le script génère l'arbre des lemmes. Un exemple de cet arbre peut être trouvé dans la figure 6.11. Il existe aussi une possibilité de marquer un lemme comme *ignoré*. C'est-à-dire que lors de l'étape de preuve le script n'essayera pas de prouver le fichier en question. Cette possibilité est très pratique quand on travaille à prouver certaines parties alors que d'autres ne sont pas encore prouvées.

La figure 6.1 présente un exemple d'utilisation du script d'automatisation dont nous venons de parler, on y trouve un certain nombre d'options que nous allons décrire. À chaque fois, l'étape ne se produira que si des options sont données.

Génération Pour la génération, on a trois options booléennes. La première active la génération des annotations, la deuxième active la génération des lemmes, et enfin la troisième active la génération des fichiers de rapports.

Sortie

```
+ Proof
|--> [ 1/11] fun_cholesky_PF2                                  done in    1.681s
|-->            Failed                                           15/  16
|--> [ 2/11] fun_compute                                       done in    1.621s
|-->            Failed                                            1/   3
|--> [ 3/11] fun_compute_dt                                    done in    2.984s
|-->            Failed                                            2/   6
|--> [ 4/11] fun_compute_dx                                    done in    1.660s
|-->            Failed                                            1/   2
|--> [ 5/11] fun_compute_pre                                   done in    2.811s
|-->            Failed                                           15/  17
|--> [ 6/11] fun_main
|-->            Ignored
|--> [ 7/11] fun_pathfollowing                                 done in    3.306s
|-->            Failed                                           11/  15
|--> [ 8/11] fun_set_nc                                        done in    0.642s
|-->            Proven                                            3/   3
|--> [ 9/11] fun_update_t                                      done in    1.645s
|-->            Failed                                            2/   5
|--> [10/11] fun_update_x                                      done in    1.652s
|-->            Failed                                            3/   5
|--> [11/11] mymath
|-->            Ignored
|--> 1 more proven, total: 1+2/ 11 (8 remaining)
|__ done in 18.013s
```

FIGURE 6.2 – Sortie du script d'automatisation.

Compilation Une seule option existe pour le moment : le niveau d'optimisation de gcc

Preuve La première option contient le timeout à donner à Alt-Ergo. La seconde option est le nombre d'instances parallèles à lancer. La troisième indique si on doit réessayer les buts échoués. La quatrième option attend les modèles à utiliser. La cinquième option est la liste des lemmes à prouver, une liste vide entrainera la preuve de tous les lemmes. Enfin, la sixième option indique si on souhaite désactiver le système permettant d'ignorer la preuve d'un lemme.

La figure 6.2 présente une portion de la sortie du script pour l'algorithme primal sans les enrichissements développé dans ce chapitre. On peut y trouver la liste de chaque fichier à prouver. Pour chaque fichier est indiqué : le nombre de buts prouvés sur le nombre de buts total, le temps total de la preuve et le statut actuel du fichier. On retrouve à la fin, un bilan de ce qui est prouvé et de ce qui reste à prouver ainsi que le temps requis pour effectuer la preuve.

6.3 Enrichissement du compilateur

Dans cette partie, nous allons présenter les modifications que nous avons apportées au compilateur PYSIL afin de simplifier les buts de preuves générés par Frama-C. Elles sont issues du processus présenté dans la partie précédente. Les améliorations présentées dans cette section impactent principalement la partie code des preuves. Nous avons été amenés à modifier le compilateur et nous allons présenter dans cette sous-section les modifications les plus conséquentes. Elles vont principalement concerner les calculs matriciels.

6.3.1 Rappels sur les affectations matricielles

En PYSIL, les calculs matriciels sont effectués à chaque affectation de matrice. Quand le compilateur rencontre le symbole d'affectation, `%=` il va alors prendre l'expression de la partie

```
y %= A * x + b
```

(a) Code simple pySil contenant une affectation matricielle, A, b, x, c sont des matrices

```
y[0, 0] %= A[0, 0] * x[0, 0] + A[0, 1]*x[1, 0] + b[0, 0]
y[1, 0] %= A[1, 0] * x[0, 0] + A[1, 1]*x[1, 0] + b[1, 0]
```

(b) Signification réelle du code de la figure a, avec A de taille 2×2 et b de taille 2×1

```
f_y[0] = 5*f_x[0]+f_x[1]+1;
f_y[1] = 3*f_x[1]+2;
```

(c) Code résultat en C pour $A = \begin{pmatrix} 5 & 1 \\ 0 & 3 \end{pmatrix}$ et $b = \begin{pmatrix} 1 \\ 2 \end{pmatrix}$

FIGURE 6.3 – Transcription basique d'une affectation matricielle

droite et l'affecter à la partie gauche. Dans le cas des matrices, pour chaque élément de la matrice résultante, le compilateur calcule l'expression qu'il doit affecter à cet élément. Un exemple de cette étape est présenté dans la figure 6.3.

Remarque :

En pySil, il est possible d'utiliser l'opérateur d'égalité = pour affecter une matrice à une autre. Ceci n'est pas une vraie affectation de matrice. En effet, ce qu'on affecte, c'est le calcul lui-même et non la matrice. Il faut bien noter que le compilateur ne générera du code uniquement si l'opérateur %= est utilisé. Par exemple,

```
z = A * x
y %= z + b
```

est équivalent à

```
z = A * x + b
```

Si la taille des matrices est grande, ou si les calculs se multiplient, le but associé à des fonctions incorporant des affectations matricielles peut devenir beaucoup trop important pour être prouvé. De plus, du point de vue du langage C, nous n'avons qu'une collection d'affectations d'éléments de tableau sans sens particulier. Ainsi, quand il s'agira de prouver des propriétés matricielles, le code sera très différent des annotations et le triplet de Hoare sera donc difficile à prouver. Nous allons voir, dans la section suivante, comment réduire les buts correspondants aux fonctions contenant des opérations matricielles.

6.3.2 Enrobage des affectations matricielles dans des fonctions

Dans cette sous-section, nous souhaitons modifier le code afin de simplifier la preuve. Comme on l'a vu dans la section 6.1.2, nous allons utiliser une coupure fonctionnelle. Cela consiste donc à placer l'intégralité du code d'affectation dans une fonction et construire un contrat correspondant. La postcondition de ce contrat sera sa représentation directe en logique. La précondition doit représenter ce qui est nécessaire pour effectuer le calcul. Le seul cas qui peut arriver est une contrainte de domaine comme pour la division ou la racine carrée.

Afin de construire la postcondition, on adjoint à chaque expression en pySil l'expression équivalente en logique. Ainsi, lors d'une affectation, il suffit de prendre l'expression logique de la partie gauche et celle de la partie droite et de construire l'égalité entre les deux. Il

```
1  /*@ensures MatVar(((double*)f_y), 2, (1)) ==
2          \old(mat_add(mat_mult(A, MatVar(((double*)f_x),2,1)), b)));
3    @assigns f_y[1], f_y[0]; */
4  void set_f_y()
5  {
6      f_y[0] = 5*f_x[0]+f_x[1]+1;
7      f_y[1] = 3*f_x[1]+2;
8  }
9
10 void f()
11 {
12     set_f_y();
13 }
```

On a appliqué la transformation sur le programme de la figure 6.3a.

FIGURE 6.4 – Résultat d'une compilation avec enrobage de l'affectation

faut cependant faire attention à prendre l'expression dans la mémoire du programme avant l'exécution de la fonction(opérateur **old** en PYSIL). Concernant la précondition, elle est pour l'instant vide, car aucun calcul matriciel n'a pour l'instant nécessité d'hypothèses.

On donne alors le nom `set` concaténé avec le nom de la fonction puis le nom de la variable et éventuellement un numéro. Le numéro est utile si on affecte plusieurs fois cette variable. Tous ces éléments sont séparés par un blanc souligné. On peut voir dans la figure 6.4 le résultat sur le même exemple que précédemment.

Il reste à voir s'il est possible de prouver ces contrats directement ou si l'on doit introduire des lemmes pour le permettre. Le but à prouver repose principalement sur la preuve d'une égalité entre deux matrices.

6.3.3 Lemmes d'extensionalités

Toute preuve d'égalité entre deux matrices en ACSL doit passer par l'axiome d'extensionalité. Cet axiome définit que si deux matrices ont la même taille et les mêmes éléments alors elles sont égales. La figure 6.5 présente cet axiome en PYSIL et en ACSL.

Cet axiome est un exemple type d'axiome qui peut bloquer un solveur SMT. Leur simple présence dans le contexte d'un but impliquant des matrices entrainera le solveur SMT à essayer de l'instancier pour toutes les matrices qui apparaitront dans le but. Afin d'éviter ceci, l'axiome a été sorti de l'axiomatique des matrices afin de rentrer dans le système de lemmes présenté dans la sous-section 5.2.4. Cela permet de sélectionner quand il est nécessaire et donc de l'empêcher d'apparaitre dans le contexte de but où il serait superflu.

Un autre problème est la difficulté pour un solveur SMT d'instancier correctement le lemme. Cela concerne en général l'instanciation de la taille de la matrice. Afin de simplifier la tâche des solveurs SMT, le compilateur génère des lemmes d'extensionalité spécifiques à la taille des matrices. Ainsi, en plus de spécifier qu'un but dépend du lemme d'extensionalité, on doit préciser la taille des matrices pour lesquels il est nécessaire.

Les lemmes d'extensionalités sont rajoutés automatiquement par PYSIL dans le cadre des affectations matricielles. Ils sont parfois nécessaires à plus haut niveau, dans ce cas, on peut le rajouter manuellement aux dépendances en utilisant la fonction `LMat.getMatEqDef(M, N)`, par exemple :

```
Axiom("mat_eq_def",
    forall(LMat, LMat, lambda A, B:
        ((A.getM() == B.getM()) &
         (A.getN() == B.getN()) &
         forall(Int, Int, lambda i, j:
            ((i >= 0) & (i < A.getM())).implies(
            ((j >= 0) & (j < A.getN())).implies(
                (A[i, j] == B[i, j])
        ))))).implies(A == B)))
```

```
/*@axiom mat_eq_def:
    \forall LMat A, B;
        ((getM(A)==getM(B))&&
         (getN(A)==getN(B))&&
         (\forall integer i, j;
            0<=i<getM(A) ==>
            0<=j<getN(A) ==>
                mat_get(A, i, j) == mat_get(B, i, j))
        ) ==> A == B; */
```

FIGURE 6.5 – Axiome d'extensionalité

```
/*@axiom mat_eq_def_2_1:
    \forall LMat A, B;
        ((getM(A) == 2) ==> (getN(A) == 1) ==>
        (getM(B) == 2) ==> (getN(B))== 1) ==>
            (mat_get(A, 1, 0) == mat_get(B, 1, 0)) &&
            (mat_get(A, 0, 0) == mat_get(B, 0, 0))
        ) ==> (TRIGGER:A) == (TRIGGER:B);*/
```

FIGURE 6.6 – Axiome d'extensionalité pour une matrice de taille 2×1

```
update_x.addDep(LMat.getMatEqDef(N, 1))
```

Dans l'exemple 6.4, on doit prouver une égalité entre deux matrices de tailles 2×1. On doit donc rajouter le lemme correspondant qui est donné dans la figure 6.6. Comme cela n'est pas toujours suffisant, on rajoute une nouvelle fonction intermédiaire dont les postconditions correspondent exactement à l'instanciation des hypothèses du lemme d'extensionalité. On peut voir le résultat, toujours sur le même exemple dans la figure 6.7.

6.3.4 Enrobage des opérations élémentaires

Dans la sous-section précédente, nous avons enrobé les opérations élémentaires par une première fonction qui présente en postcondition toutes les hypothèses du lemme d'extensionalité. Puis, nous avons enrobé cette fonction par une fonction qui possède en postcondition la conséquence du lemme d'extensionalité c'est-à-dire une égalité entre la matrice qui reçoit l'expression et l'expression.

Ainsi de l'extérieur, on appelle une fonction pour chaque affectation avec une postcondition exprimant clairement l'égalité qui en résultera. À l'intérieur, on a un ensemble de fonctions dont chaque étape est de taille raisonnable. La dernière étape consiste à prouver chaque hypothèse du lemme d'extensionalité à partir du code C. Ce code C est long, car il y a une ligne par élément de la matrice et complexe, car comprenant des accès mémoire. Le but associé est donc assez long à être prouvé pour de grosses matrices.

```
/*@ensures getM(MatVar((double*)f_y, 2, 1)) == 2;
  @ensures getN(MatVar((double*)f_y, 2, 1)) == 1;
  @ensures getM(\at(mat_add(mat_mult(A,
    MatVar((double*)f_x, 2, 1)),b), Pre)) == 2;
  @ensures getN(\at(mat_add(mat_mult(A,
    MatVar((double*)f_x, 2, 1)), b), Pre)) == 1;
  @ensures mat_get(MatVar((double*)f_y, 2, 1), 1, 0)) == mat_get(
    \old(mat_add(mat_mult(A, MatVar((double*)f_x, 2, 1)), b)), 1, 0);
  @ensures mat_get(MatVar((double*)f_y, 2, 1), 0, 0) == mat_get(
    \old(mat_add(mat_mult(A, MatVar((double*)f_x, 2, 1)), b)), 0, 0);
  @assigns f_y[1], f_y[0]; */
void set_in_f_y1()
{
    f_y[0] = 5*f_x[0]+f_x[1]+1;
    f_y[1] = 3*f_x[1]+2;
}
```

FIGURE 6.7 – Fonction intermédiaire avec le lemme d'extensionalité en postcondition

Pour réduire ce temps de preuve, nous avons enrobé chaque affectation élémentaire par une fonction annotée avec les opérateurs logiques matriciels. Ainsi, on se retrouve avec un grand nombre de nouveaux buts très simple et le but original, toujours aussi long, mais sans accès mémoire. On peut trouver, dans la figure 6.8, le résultat sur l'exemple de la figure 6.3a.

On peut noter que nous avons rajouté des assertions dans la fonction `set_in_f_y2`. Elles permettent de prouver que les variables utilisées dans le calcul matriciel, si ce n'est pas la variable cible, restent identiques après chaque opération élémentaire.

6.3.5 Bilan

On a vu dans cette partie comment rendre prouvables les buts associés à des calculs matriciels. Les calculs matriciels se situent exclusivement dans l'affectation d'une matrice à une expression matricielle. Pour cela, on a introduit un certain nombre de fonctions intermédiaire. On a aussi vu l'importance de l'axiome d'extensionalité dans cette étape, les problèmes qu'il apportait ainsi que des solutions à ces problèmes. Nous allons maintenant voir comment nous avons enrichi notre code PYSIL afin de prouver le code qui en est généré.

```
/*@ ... */
void set_in_f_y2()
{
    set_f_y2_sub_0();
    /*@assert MatVar(((double*)f_x), 2, 1) ==
        \at(MatVar((double*) f_x, 2, 1), Pre);*/
    set_f_y2_sub_1();
    /*@assert MatVar(((double*)f_x), 2, 1) ==
        \at(MatVar((double*) f_x, 2, 1), Pre);*/
}

/*@ensures mat_get(MatVar((double*)f_y, 2, 1), 0, 0) ==
    \old(mat_get(mat_add(mat_mult(A,
        MatVar((double*) f_x, 2), 1)), b), 0, 0);
  @assigns f_y[0]; */
void set_f_y2_sub_0();

/*@ensures mat_get(MatVar((double*)f_y, 2, 1), 1, 0) ==
    \old(mat_get(mat_add(mat_mult(A,
        MatVar((double*) f_x, 2, 1)), b), 1, 0));
  @assigns f_y[1]; */
void set_f_y2_sub_1();
```

FIGURE 6.8 – Enrobage des opérations élémentaires

6.4 Enrichissement de l'algorithme de points intérieurs

On va, dans cette section, enrichir l'algorithme présenté dans le chapitre 4. Nous allons commencer par écrire des lemmes associés à chaque but issu d'un contrat. Ces lemmes doivent permettre de prouver le but d'origine très facilement. En général, cela consiste à construire un but par postcondition. On peut trouver dans la figure 6.9, l'exemple des lemmes associés au contrat de `update_t`, en voici leur version mathématique :

Propriété 6.4.1 (update_t_ensures1) *Soit $c \in \mathbb{R}^n$, $x \in \mathbb{R}^n$ et $t \in \mathbb{R}$.*
Supposons que c n'est pas le vecteur nul et que $\left\|\tilde{f}'(x,t)\right\|_x^* < \beta$, alors :

$$\left\|\tilde{f}'(x, t + \frac{\gamma}{\|c\|_x^*})\right\|_x^* < \beta + \gamma \tag{6.7}$$

Propriété 6.4.2 (update_t_ensures2) *Soit $c \in \mathbb{R}^n$, $x \in \mathbb{R}^n$ et $t \in \mathbb{R}$.*
Supposons que $\left\|\tilde{f}'(x,t)\right\|_x^* < \beta$, alors :

$$t + \frac{\gamma}{\|c\|_x^*} \geq t(1 + \frac{\gamma}{\beta + \sqrt{m}}) \tag{6.8}$$

Propriété 6.4.3 (update_t_ensures3) *Soit $c \in \mathbb{R}^n$ et $x \in \mathbb{R}^n$.*
Supposons que $\left\|\tilde{f}'(x,0)\right\|_x^* < \beta$, alors :

$$\|c\|_x^* < \|c\|_{x_{AC}}^* \tag{6.9}$$

Ces lemmes ne sont pas toujours suffisants et il faut renforcer les préconditions. Pour cela on introduit de nouvelles fonctions intermédiaires au sein de la fonction. C'est pourquoi nous avons introduit `compute_dt` dans la fonction `update_t`. La fonction `update_t` a elle aussi

```
1  Lemma("update_t_ensures1",
2    forall(LMat, LMat, LMat, LMat, Real, Real, lambda A, b, c, x, at, dt:
3      (c.isnull() == False).implies(
4      acc(A, b, c, at, x, beta).implies(
5      (dt == gamma / normI(A, b, c, x)).implies(
6        acc(A, b, c, at + dt, x, beta + gamma))))))
7  Lemma("update_t_ensures2",
8    forall(LMat, LMat, LMat, LMat, Real, Real, lambda A, b, c, x, at, dt:
9      acc(A, b, c, at, x, beta).implies(
10     (dt == gamma / normI(A, b, c, x)).implies(
11     at + dt >= at * (1 + gamma/(beta +
12       A.getM().cast(Real).sqrtl()))))))
13 Lemma("update_t_ensures3",
14   forall(LMat, LMat, LMat, LMat, lambda A, b, c, x:
15     theory.acc(A, b, c, 0, x, beta).implies(
16       theory.normI(A, b, c, x) <= NCCA)))
17
18 with Function("update_t") as update_t:
19   compute_dt()
20   t %= t + dt
21 update_t.addRequire(H ==theory.hess(ACst, bCst, x))
22 update_t.addRequire(theory.acc(ACst, bCst, cCst, t, x, beta))
23 update_t.addEnsure(theory.acc(ACst, bCst, cCst, t, x, beta + gamma))
24 update_t.addEnsure((old(t) > 0).implies(
25   t >= old(t) * (1 + gamma/(beta + v))))
26 update_t.addEnsure((old(t) == 0).implies(t >= gamma / NCCA))
27 update_t.addDep("update_t_ensures1")
28 update_t.addDep("update_t_ensures2")
29 update_t.addDep("update_t_ensures3")
```

Le premier lemme (l.1-6) correspond à la propriété 6.4.1. Il montre que la condition de centrage approché intermédiaire est correcte après mise à jour de `t`.
Ce lemme permet de prouver la première postcondition (l.23) de la fonction `update_t`. C'est pourquoi, on la retrouve en conclusion (l.6) du lemme. La première hypothèse du lemme (l.3) garantit que `c` est non nul afin de pouvoir faire la division. La deuxième hypothèse (l.4) correspond à la seconde précondition de la fonction (l.22). La troisième hypothèse (l.5) correspond à la postcondition issue de l'appel de la fonction `compute_dt` (l.19).
On peut retrouver le même genre de construction pour les deux lemmes suivants (l.7-12 et l.13-16) qui montrent respectivement que `t` augmente suffisamment à chaque itération, propriété 6.4.2 et que `t` est correctement initialisé, propriété 6.4.3.

FIGURE 6.9 – Lemmes PYSIL permettant la preuve d'un contrat ACSL

```
1  with Function("compute_dt") as compute_dt:
2    nc = Real.Var('nc')
3    pre_nc = Mat(1, 1).Var('pre_nc')
4    Hc = solve(name, H, cCst, 'cholesky')
5    Assert(Hc == theory.hess(ACst, bCst, x).inv() * cCst)
6    pre_nc %= (cCst.transpose() * Hc)
7    Assert(pre_nc == cCst.transpose().logic *
8      (theory.hess(ACst, bCst, x).inv() * cCst))
9    with Function("set_nc") as set_nc:
10     nc %= pre_nc[0, 0].sqrt()
11     compute_nc.addRequire(pre_nc.logic[0, 0] >= 0)
12     compute_nc.addEnsure(nc == (pre_nc.logic[0, 0]).sqrtl())
13   set_nc()
14   Assert(nc == theory.norm(ACst, bCst, cCst, x))
15   dt %= gamma / nc
```

Cette fonction calcule `dt` $= \frac{\gamma}{\|c\|_x^*} = \frac{\gamma}{\|c\|^*_{c^T H^{-1} c}}$. Pour cela, on effectue quatres étapes. La première(l.4) calcule `Hc` $= H^{-1} * c$. La deuxième(l.5) étape calcule `pre_nc` $= c *$ `Hc`. La troisième(l.13) appel une fonction intermédiaire : `set_nc`(l.9-12) calculant la racine carrée de `pre_nc`. Et enfin la dernière étape(l.15) calcule $\frac{\gamma}{\text{nc}}$. Cette fonction étant un peu complexe, un certain nombre d'**Assert** ont été rajoutés afin de spécifier ce que chaque étape effectue. Le calcul de la racine carrée a été placé dans une fonction à part afin de donner une étape de preuve intermédiaire. Cette étape permet de prouver de manière séparée que `pre_nc` est positif.

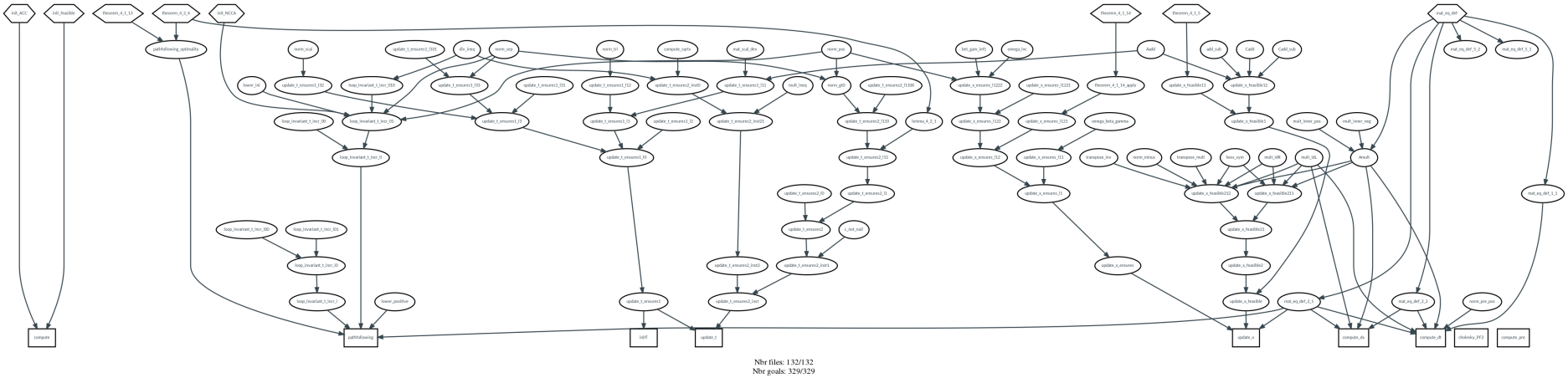
FIGURE 6.10 – Rajout d'**Assert** à une fonction afin de la prouver

été séparée en fonction intermédiaire à cause du calcul de la racine carrée, on peut voir le résultat dans la figure 6.10.

Une fois les contrats prouvés, nous avons suivi la procédure présentée dans la sous-section 6.1.1 : on a construit petit à petit des lemmes intermédiaires jusqu'à ce que chacun soit prouvable. On peut trouver dans la figure 6.11 l'arbre de dépendance de tous les lemmes et contrats. Nous ne rentrerons pas dans tous les détails de ces lemmes intermédiaires, mais mentionnerons la principale difficulté liée au solveur SMT : l'instanciation.

Instanciation dans les solveurs SMT Le point faible des solveurs SMT, en ce qui nous concerne, est l'instanciation. En effet, nous générons un grand nombre de lemmes et axiomes. Ces lemmes et axiomes utilisent des quantificateurs que les solveurs SMT doivent alors instancier. Ceci est difficile et pour les aider les solveurs SMT ou les logiciels en amont utilisent des déclencheurs. Le contexte d'un solveur SMT est composé d'un certain nombre d'hypothèses, qui peuvent être quantifié universellement par des variables. Le solveur SMT adjoint des déclencheurs à chaque hypothèse possédant un quantificateur universelle. Les déclencheurs sont des sous-formules de l'hypothèse faisant intervenir les variables quantifiées universellement. Le solveur SMT instanciera alors une hypothèse quantifié s'il arrive à unifier les déclencheurs de l'hypothèse avec d'autres hypothèses présente dans son domaine.

Les déclencheurs sont en général identifiés automatiquement par le solveur SMT, mais le résultat n'est pas toujours satisfaisant. Afin de résoudre ce problème, Alt-Ergo et Frama-C permettent à l'utilisateur de spécifier des déclencheurs manuellement. Nous avons utilisé cette fonctionnalité, mais cela n'a pas toujours suffi. Certains lemmes, comme celui présenté dans la figure 6.12, sont restés improuvables alors qu'une preuve Coq très simple existe :



Les rectangles en bas représentent les contrats de fonctions. Nous n'avons pas affiché les contrats des fonctions effectuant les opérations bas niveau par souci de place. Ils dépendent majoritairement de lemmes d'extensionalités. Deux fonctions ne sont pas prouvées, la fonction calculant la décomposition de Cholesky et celle calculant le gradient et la hessienne. On retrouve en haut, dans des losanges, les théorèmes que nous avons admis. Les ovales représentent les lemmes intermédiaires. Certains se prouvent sans lemmes supplémentaires, cela veut dire qu'ils reposent sur les axiomes définis dans les axiomatiques que nous générons, dans les lemmes générés par Frama-C ou ceux encodés directement dans les théories des solveurs SMT.

FIGURE 6.11 – Arbre complet de dépendances entre les lemmes

```
axiom Q_update_t_ensures1_l11:
  forall A,b,c,x : A_LMat.
  forall at_0,dt_0 : real .
  L_normI(A, b,
    L_mat_add(L_mat_add(L_grad(A, b, x), L_mat_scal(c, at_0)),
      L_mat_scal(c, dt_0)), x)
    = L_normI(A, b, L_mat_add(L_grad(A, b, x), L_mat_scal(c, at_0 + dt_0)),
        x)

axiom Q_update_t_ensures1_l12:
  forall A,b,c,x : A_LMat.
  forall at_0,dt_0 : real.
  L_normI(A, b,
    L_mat_add(L_mat_add(L_grad(A, b, x), L_mat_scal(c, at_0)),
      L_mat_scal(c, dt_0)), x)
    <= (L_normI(A, b, L_mat_scal(c, dt_0), x)
        + L_normI(A, b, L_mat_add(L_grad(A, b, x), L_mat_scal(c, at_0)), x))

goal lemma_update_t_ensures1_l1:
  forall r_1,r : real.
  forall L_3,L_2,L_1,L : A_LMat.
  let a = L_grad(L_3, L_2, L) : A_LMat in
  L_normI(L_3, L_2, L_mat_add(a, L_mat_scal(L_1, r + r_1)), L)
    <= (L_normI(L_3, L_2, L_mat_scal(L_1, r), L)
        + L_normI(L_3, L_2, L_mat_add(a, L_mat_scal(L_1, r_1)), L))
```

Ce fichier Alt-Ergo est issu d'un but généré pour le lemme intermédiaire de la première postcondition de `update_t`. Il est très simple à prouver, il suffit de réécrire le premier axiome dans le but puis d'appliquer le second. Alt-Ergo n'a jamais pu le prouver à part en instanciant manuellement le premier axiome. Plusieurs déclencheurs ont été essayés, mais aucun n'a permis au but d'être prouvé par Alt-Ergo.

FIGURE 6.12 – Lemme ne pouvant être prouvé par Alt-Ergo

```
simpl;intros.
rewrite <- Q_update_t_ensures1_l11;auto.
apply Q_update_t_ensures1_l12;auto.
```

On utilise le lemme `Q_update_t_ensures1_l11` pour développer `L_mat_scal(c, at_0 + dt_0))` puis on applique le lemme `Q_update_t_ensures1_l12`.

Afin de pallier ce problème nous avons mis en place un mécanisme permettant de transmettre une preuve Coq à Frama-C. On a ensuite adjoint, à chaque lemme posant problème, une preuve Coq dans le code pySil. Cette solution reste peu satisfaisante pour le moment, car un changement dans Frama-C ou pySil peut entrainer un changement du but Coq qui invaliderait la preuve. Pour rendre cela plus robuste, il faudrait soit réussir à trouver les bons déclencheurs, soit écrire des preuves Coq plus robustes en elles-mêmes ou même des scripts de preuve comme proposée dans Wang, Jobredeaux, Herencia-Zapana et al., 2013.

6.5 Expérimentations sur des problèmes de tailles croissantes

Nous avons dans un premier temps travaillé, sur un exemple simple de taille 5×2. Cependant, notre méthode est purement générique et nous pouvons donc tester le comportement de la preuve quand la taille des matrices augmente. La taille des matrices ne changera pas la prouvabilité, ce qui était prouvé en taille 5×2 sera prouvable pour les tailles supérieurs. Le problème vient, par contre, du temps qui sera nécessaire. Celui-ci est imprévisible théoriquement à cause du passage par des solveurs SMT, c'est pourquoi nous allons donc effectuer la preuve pour des problèmes de taille croissante afin de voir comment le temps de preuve évolue avec la taille des problèmes.

6.5.1 Générer des problèmes de tailles croissantes

Afin d'expérimenter notre méthode, nous avons besoin de problème de taille croissante. En effet, nous souhaitons tester notre preuve sur un grand nombre de problèmes et maitriser la taille de ces problèmes afin de la faire croitre. Nous avons donc décidé de générer automatiquement ces problèmes à partir de la donnée de sa taille. Il fallait cependant que le problème ait un intérieur non vide et borné. Pour cela, nous n'avons pas généré des matrices A, b, et c totalement au hasard.

Générer un problème d'intérieur non vide

Nous avons imposé que toutes les composantes de b soient strictement positives afin que le point 0 appartienne à l'ensemble faisable. Mathématiquement, cela n'apporte pas de perte de généralité : on peut toujours translater l'ensemble. Ceci nous garantira que l'intérieur de l'ensemble faisable sera toujours non vide. En effet, il existe un voisinage de zéro inclus dans chacune des contraintes et l'on peut en prendre l'intersection. Cette intersection sera toujours un voisinage de zéro et sera incluse dans toutes les contraintes donc dans l'intérieur de l'ensemble faisable.

Générer un problème borné

Afin de garantir que notre problème serait borné, nous avons rajouté des contraintes formant un hypercube de grande taille. Ainsi chaque problème possède aux moins $2 * N$ contraintes qui sont fixées. Ainsi, les problèmes générés ne sont pas totalement aléatoires, mais cela permet d'avoir facilement un grand nombre de problèmes avec, quand même, une assez grande variabilité.

Bilan

Ces contraintes étant fixées, on peut rajouter autant de lignes à A qu'on le souhaite, avec des valeurs purement aléatoires. De même pour b à condition que ce soient des réels strictement positifs. c peut être généré comme on le souhaite du moment qu'il est non nul. Ces contraintes sont mathématiquement peu restrictives, car on peut retrouver tous les problèmes qui étaient naturellement bornés et non vides par translation et homothétie. Ceci reste vrai dans notre cas, car nous n'abordons pas la problématique flottante. Si nous venions à la traiter, il faudrait trouver une méthode de génération plus subtile afin d'explorer la diversité des flottants.

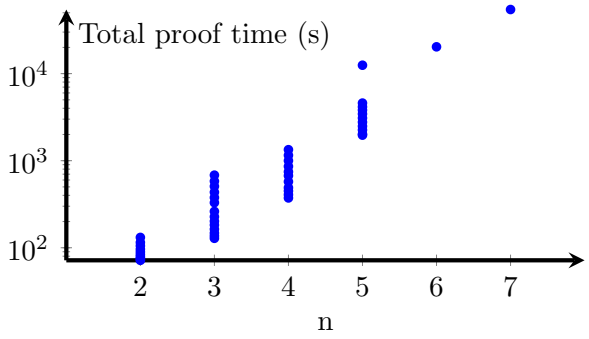
En pratique, nous avons généré des problèmes pour les tailles suivantes : $(n, m) \in [2, 4] \times [3 \times n, 5 \times n[$ avec des timeouts de 10 secondes.

6.5.2 Temps totaux pour des problèmes de tailles croissantes

Nous avons effectué ces expérimentations sur un ordinateur portable classique, équipé d'un processeur de 2.6GHz et 8Go de RAM. Le processeur possédait 4 coeurs, mais aucun outil utilisé ne pouvant en tirer parti, nous avons lancé chaque expérimentation sur un seul coeur(cela nous a cependant permis de lancer plusieurs expérimentations en même temps).

Comme représenté dans la figure 6.13, le temps total dépend fortement de la taille du problème d'entrée. Cela était attendu en particulier à cause de l'explosion du nombre de fichiers dû à la politique d'un fichier par opération atomique.

On a pu remarquer lors des expérimentations que hormis sa taille le problème avait peu d'incidence sur les temps de preuve. Par exemple, pour la taille $(3, 9)$ le temps de preuve se situait dans l'intervalle $[124s, 127s]$ avec une moyenne de $126s$, et une variance de 0.44. Il s'avère qu'en effet, les annotations dépendent assez peu des valeurs et le code non plus. La principale différence provient des précalculs. Par exemple, une matrice creuse généra moins de



Chaque point correspond au temps total qui a été nécessaire pour effectuer la preuve. Pour les petites tailles, la preuve s'effectue rapidement et donc chaque point représente une centaine d'exécutions. Pour les tailles plus grandes, certains buts ne passent pas à l'échelle et donc il a fallu allouer plus de temps à Alt-Ergo afin qu'ils soient prouvés. Ainsi, chaque point ne représente plus qu'une seule instance.

FIGURE 6.13 – Temps total de preuve pour des problèmes de taille $m \times n$ croissante

calcul. Ceci entrainera une difficulté accrue pour les prouveurs SMT qui devront en quelque sorte deviner la présence de zéros. Nous avons cependant fait en sorte, aux sous-sections précédentes, de réduire la taille des buts ce qui rend l'impact des matrices creuses sur la preuve moins important.

Notre méthode génère un très grand nombre de buts et fichiers. Nous avons pu identifier l'évolution du nombre de fichiers : nb_f et du nombre de buts : nb_g en fonction de la taille (m, n) des matrices.

$$\begin{cases} nb_f &= 57 + 6 \times n + m + 3 \times n^2 \\ nb_g &= 129 + 18 \times n + 3 \times m + 9 \times n^2 \end{cases}$$

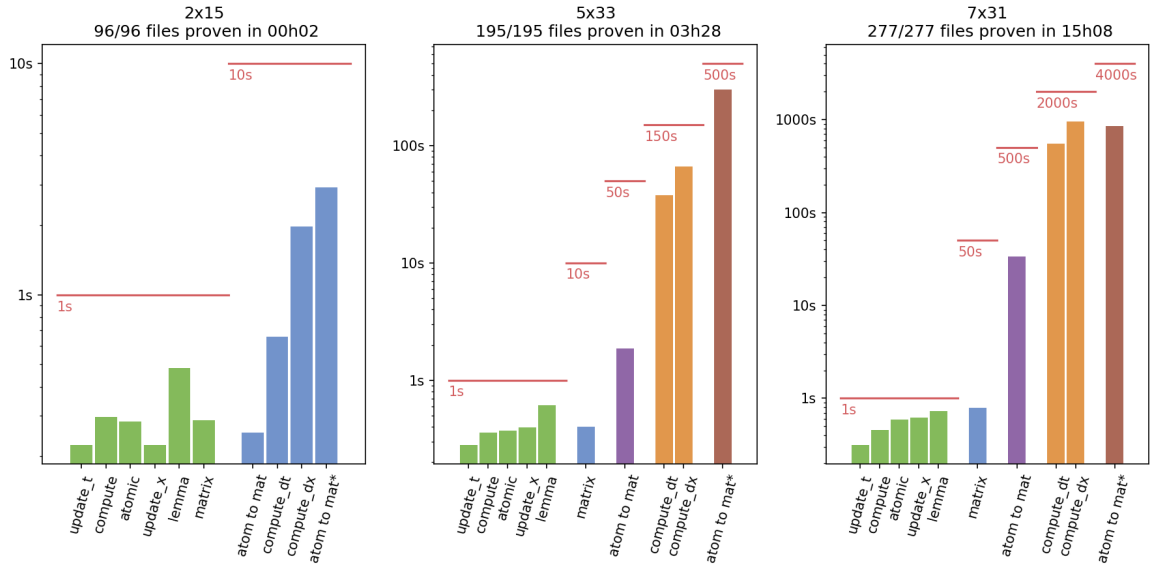
On peut voir, dans les deux cas, que les formules sont linéaires en m mais quadratique en n. Le temps de preuve devrait donc suivre ces courbes tant que l'on considère comme négligeable l'impact de la taille des matrices sur le temps de preuve. Ceci devient rapidement incorrect pour les buts faisant intervenir du code, celui-ci augmentant aussi avec la taille des matrices.

6.5.3 Identification des buts les plus difficiles

Nous avons mesuré le temps requis pour prouver les buts de chaque fichier ce qui nous a permis d'identifier les fichiers associés à des preuves plus difficiles par rapport aux buts qui passeront correctement à l'échelle. La figure 6.14 présente ces résultats. Nous avons rassemblé en groupes les fichiers en fonction de leur capacité à passer à l'échelle. On peut identifier les groupes suivants : *lemma* qui rassemble tous les lemmes ACSL, *atomic* qui rassemble toutes les opérations atomiques des opérations matricielles, *matrix* qui rassemble les contrats matriciels et *atom to mat* qui rassemble les contrats intermédiaires entre opération atomique et opération matricielle. Un fichier du dernier groupe a aussi été isolé : *atom to mat** car il présentait une nette différence de temps de preuve. Ce dernier fichier correspond au calcul du gradient et de la hessienne.

Les derniers groupes correspondent à des fichiers uniques contenant les contrats des opérations associées *update_dt*, *update_dx*, *compute_dt*, *compute_dx* and *compute*.

Ainsi, comme attendu, certaines propriétés sont difficiles à prouver. Les problèmes de passage à l'échelle sont particulièrement présents pour *compute_dt*, *compute_dx* et les groupes



Chaque groupe est décrit par son temps moyen et le timeout utilisé dans Alt-Ergo(la ligne rouge). Nous allons prendre l'exemple de la taille 5×33, pour lire ce schéma.Nous avons produit pour cette taille 195 buts, tous prouvés. Il faut moins de 1 seconde pour les buts des catégories *lemma*, *atomic*, *update_t*, *update_x* et *compute* ; moins de 10 secondes pour la catégorie *matrix* et 50s pour la catégorie *atom to mat*. *compute_dt* et *compute_dx* ont été prouvés avec un timeout de 150s. Le fichier isolé *atom to mat** a nécessité un timeout de 500s pour que tous ces buts soient prouvés. On peut noter que le temps de preuve moyen pour le groupe *compute_dt* est d'environ 40s ainsi seul un sous-ensemble des buts (2 parmi 6) a vraiment nécessité 150s pour être prouvé.

FIGURE 6.14 – Temps de preuve par groupe de buts

*set_* et *set_in* clusters. Cependant, les buts associés sont mathématiquement assez simples. Les principaux problèmes viennent plus de la capacité de Frama-C à les encoder correctement et la puissance des prouveurs SMT. En particulier, dans notre cas, comme nous avons toutes nos variables déclarées globales, les aliasings possibles sont très limités et un modèle mémoire [2] plus adéquat simplifierait grandement certains buts.

Conclusion

On a dans ce chapitre obtenu une implémentation C d'algorithme de points intérieurs, prouvables de manière automatique. Nous avons pour cela utilisé une méthode semi-systématique pour nous aider. Cela nous a permis d'enrichir l'algorithme, mais surtout le compilateur qui est maintenant plus robuste vis-à-vis de la preuve des opérations matricielles. Enfin, nous avons pu tester le code sur des exemples de tailles croissantes afin d'étudier le passage à l'échelle. Nous avons donc un outil permettant de générer un algorithme de points intérieurs spécifique à un problème donné. Ce code peut être ensuite automatiquement prouvé. Ceci permettra donc d'avoir un code de points intérieurs certifié qui pourrait être embarqué avec un niveau de confiance élevé. Il reste cependant un certain nombre d'aspects, qui peuvent limiter cette confiance ou l'utilisation d'un tel algorithme. Nous allons donc, dans le chapitre suivant, discuter de ces problèmes et éventuellement donner des pistes de solutions.

Les résultats de ce chapitre ont fait l'objet d'une soumission : Davy et al., 2018.

2. un modèle mémoire est la manière de représenter les variables et pointeurs lors du calcul de plus faible précondition. Voir sous-section 2.3.3

Bibliographie

Davy, Guillaume, Eric Feron, Pierre-Loic Garoche et Didier Henrion (nov. 2018). "Formal verification of an interior point algorithm instanciation". In : *LPAR-22* (cf. p. 6, 139, 164).

Nesterov, Yurii (2004). *Introductory lectures on convex optimization : a basic course*. Applied optimization. Boston, Dordrecht, London : Kluwer Academic Publ. (cf. p. 79, 85-87, 92, 93, 120-122).

Nesterov, Yurii et Arkadi Nemirovski (1994). *Interior-point Polynomial Algorithms in Convex Programming*. T. 13. Studies in Applied Mathematics. Society for Industrial et Applied Mathematics (cf. p. 85, 120).

Wang, Timothy, Romain Jobredeaux, Heber Herencia-Zapana et al. (2013). "From Design to Implementation : an Automated, Credible Autocoding Chain for Control Systems". In : *CoRR* abs/1307.2641. arXiv : 1307.2641 (cf. p. 4, 5, 103, 135).

Troisième partie

Bilan et perspectives

Chapitre 7

Discussion

Sommaire

Dans ce chapitre, nous allons étudier les limitations et possibles améliorations de ce travail. Dans une première section, nous reviendrons sur les moyens à notre disposition pour borner le temps d'exécution de notre algorithme. C'est une étape importante pour que ce travail soit utilisé dans un cadre embarqué critique. Nous discuterons ensuite de l'optimisation du code généré utile aussi dans le cadre de l'embarqué critique. La troisième section reviendra sur les possibles extensions de ce travail à d'autres algorithmes. La section 4 se concentrera à dégager quelques pistes de travail pour tenir compte des erreurs flottantes. Enfin, la dernière section énumérera de manière complète les éléments qu'il faut vérifier par des méthodes tierces afin d'avoir la garantie de correction absolue du code généré.

7.1 Calcul du temps d'exécution dans le pire des cas

Le code final que nous générons ne possède que des boucles bornées. On peut donc envisager un calcul de son temps d'exécution. D'un point de vue pratique cela passerait par l'utilisation d'outil comme OTAWA(Ballabriga et al., 2010) ou CHRONOS(Li et al., 2007).

D'un point de vue théorique tous les enjeux du calcul du temps d'exécution reposent sur la borne du nombre d'itérations de Newton de notre algorithme. Cette borne est définie dans l'équation (4.29) qui est, pour rappel :

$$n_{max} = \lceil 1 + \frac{\ln(\delta) + \ln(\|c\|^*_{x_{AC}}) - \ln(a) - \ln(\epsilon)}{\ln(q)} \rceil. \tag{7.1}$$

Dans cette équation δ, a et q ne dépendent que de la taille du problème. Ainsi, la partie critique de cette équation est $\|c\|^*_{x_{AC}}$ dont il faut trouver un majorant.

Si A et b sont connus, on peut calculer hors-ligne le centre analytique x_{AC}, la hessienne en ce point et donc calculer $\|c\|^*_{x_{AC}}$. Par exemple, si c varie et que sa norme euclidienne est toujours inférieure à M, on pourra borner $\|c\|^*_{x_{AC}}$ par λM avec $\lambda = \sup_{||z||=1} \|z\|^*_{x_{AC}}$.

En revanche, si A et b varient, il faut borner $\|c\|^*_{x_{AC}}$. En effet, $\|c\|^*_{x_{AC}}$ dépend du centre analytique, mais aussi de l'inverse de la hessienne en ce point qui dépendent tous deux de A et de b. Dans le cadre du MPC on peut se contenter de faire varier b et laisser A invariant. En effet, la partie paramétrique d'un MPC se situe souvent dans b. On peut se référer à la sous-section 4.3.1 pour plus de précisions.

Ainsi, on veut obtenir un majorant de $f(b) = \|c\|^*_{x_{AC}} = c^T H(x_{AC}(b), b)^{-1} c$ quand b varie dans un ensemble donné E. Géométriquement, les variations de b se traduisent par une translation des contraintes, leur direction ne dépendant que de A. Notre intuition est que si b varie dans un ensemble E, on puisse borner $f(b)$ par $f(b_0)$ avec b_0 un majorant de E. C'est-à-dire qu'on prend un ensemble faisable qui inclut tous les autres ensembles faisables. Nous n'avons cependant pas encore réussi à démontrer cette conjecture ou à l'infirmer.

7.2 Optimisation du code généré

Nous avons décidé de générer un code de points intérieurs ainsi que sa preuve par opposition à prouver un code générique déjà implémenté. Ce choix a eu deux origines, la première était de pouvoir instrumenter la preuve à plus haut niveau et la rendre plus facilement réutilisable et la seconde, qui nous intéresse dans cette partie, était de pouvoir rendre spécifique notre code aux problèmes à résoudre.

Nous avons pu effectuer quelques optimisations simples allant dans cette direction. En particulier, nous utilisons le fait que A et b soient connus en grande partie. Ceci permet à pySil de précalculer automatiquement les expressions mathématiques entièrement définies. Par exemple, on peut voir dans la figure 7.1 que le code spécifique généré par pySil sera

```
A = Mat(2, 2).fromArray([[5, 1],
                         [0, 3]], "A")
b = Mat(2, 1).fromArray([[1],
                         [2]], "b")
x = Mat(2, 1).Var('x')
y = Mat(2, 1).Var('y')
y %= A * x + b
```

(a) Code pySil effectuant un calcul affine

```
y[0,0]%=A[0,0]*x[0,0]+A[0,1]*x[1,0]+b[0,0]
y[1,0]%=A[1,0]*x[0,0]+A[1,1]*x[1,0]+b[1,0]
```

(b) Code naïf générique

```
f_y[0] = 5*f_x[0]+f_x[1]+1;
f_y[1] = 3*f_x[1]+2;
```

(c) Code généré par pySil

On peut voir à travers cet exemple deux manières de compiler un code pySil matriciel. Le premier ne prend pas en considération les valeurs de A et b. Au contraire, le second les utilise ce qui permet de simplifier le code généré. Notre compilateur tient toujours compte des valeurs constantes qui lui sont fournies pour simplifier toutes les expressions qu'il génère.

FIGURE 7.1 – Optimisation du code de calcul matriciel généré.

plus rapide par rapport à une implémentation générique naïve. L'impact sur la preuve de ces optimisations est pour l'instant limité à une possible augmentation du temps de preuve.

Cette optimisation reste très basique et pour améliorer sensiblement le temps de calcul on pourrait se référer à ce qui est utilisé par CVXGEN(MATTINGLEY et BOYD, 2012) qui permet justement de générer du code d'optimisation spécifique à un problème donné. L'enjeu dans notre cas n'est pas tant d'optimiser le code que de maintenir la preuve lorsque le code change. Nous avons cependant bâti une preuve très modulaire. En particulier, nous avons séparé de manière claire les résultats matriciels et mathématiques des problèmes d'implémentation de bas niveau. Ceci devrait permettre d'absorber les changements simples de code sans interaction supplémentaire et de rendre minimes les modifications à apporter aux annotations pour des changements plus importants. Ceci reste cependant théorique et une mise en pratique serait nécessaire pour tester concrètement la robustesse de notre approche vis-à-vis des modifications de code.

7.3 Extensions à d'autres algorithmes

Ce travail s'est concentré sur la résolution de problèmes LP. Cependant, nous avons fait le choix d'un algorithme de points intérieurs ce qui permet d'envisager sa généralisation à des problèmes SDP définis dans la sous-section 2.2.2. L'algorithme restera sensiblement identique, seule la fonction barrière changera et donc le calcul du gradient et de la hessienne. Une fonction de coût fonctionnant pour un problème SDP est

$$\log(\det(B - \sum_{i=0}^{m} x_i A)) \text{ avec } A_0, ..., A_m, B \in \mathcal{S}^n \text{ et } x \in \mathbb{R}^n \tag{7.2}$$

Cela nécessitera quelques ajustements dans les axiomatiques, mais la preuve devrait rester

très proche de la preuve actuelle.

On peut aussi envisager l'utilisation d'un autre algorithme que l'algorithme primal actuel. En effet, les algorithmes les plus efficaces actuellement sont les algorithmes primaux duaux. Ceux-ci sont cependant plus complexes et nécessiteraient de plus grandes modifications dans le code et la preuve. Cependant, la preuve repose sur les mêmes résultats et donc on devrait pouvoir exprimer les algorithmes les plus efficaces sans difficulté et il en va de même de leur preuve. La partie qui nécessiterait le plus de travail est la transformation de la boucle **`while`** en boucle **`for`**. En effet, il sera nécessaire d'extraire des preuves de convergence de ces algorithmes une borne concrète sur le nombre d'itérations. Cette borne risque cependant d'être très conservative comparée aux résultats pratiques, mais c'est le prix à payer pour avoir la garantie que le programme tourne dans un temps borné.

D'un point de vue plus général, pySil pourrait être utilisé pour tout projet faisant intervenir de la génération de code formellement prouvé. Il est particulièrement adapté pour des algorithmes reposant sur du calcul matriciel. De plus, ce travail offre une bonne marche à suivre complète pour des algorithmes reposant sur des itérations de Newton.

7.4 Calcul flottant

7.4.1 Présentation générale

Dans ce travail, nous avons décidé de laisser de côté l'aspect calcul flottant. Nous avons donc supposé les variables du programme comme réelles pour effectuer la preuve. Pour un travail plus complet, il conviendrait de revenir à une sémantique flottante correspondant aux systèmes sur lesquels le programme doit s'exécuter.

Nous allons, dans cette section, à défaut de proposer des solutions complètes, mettre en évidence les limites qu'apporte un calcul avec les flottants via une analyse empirique de notre code comme cela a pu être mené dans Févotte et Lathuilière, 2017.

Pour cela, nous allons prendre un problème linéaire simple à deux variables et cinq contraintes données dans la figure 7.2

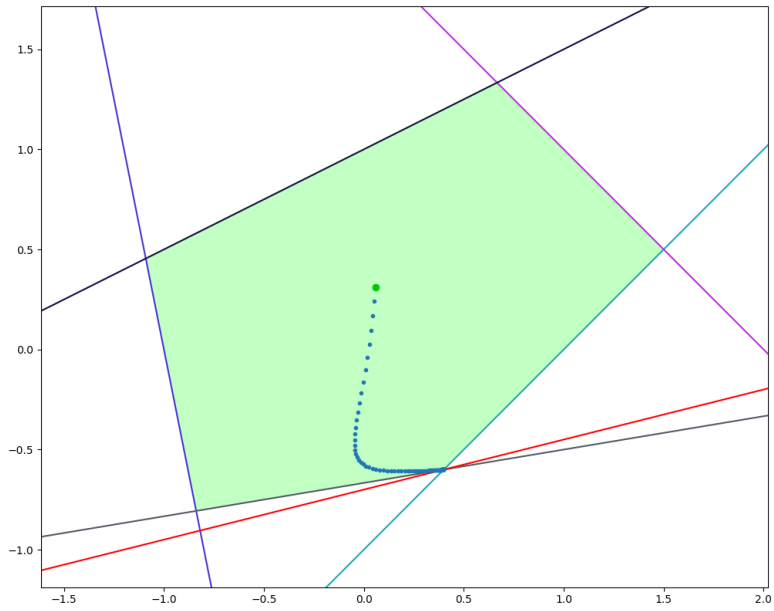
La solution théorique est facilement calculable et vaut `[[0.4], [-0.6]]`

Nous générons le code pour $\epsilon = 10^{-2}$ ce qui nous permet d'obtenir le résultat suivant : `[[0.3999949], [-0.6000003]]`, la figure 7.3 représente les différentes itérations de l'algorithme qui semble se comporter conformément aux attentes. Si on calcule la différence sur la fonction de coût, on obtient $-0.3999949 \times 1 - 0.6000003 \times 4 + 1 \times 0.4 + 4 \times 0.6 = -2.7999961 + 2.8 = 0.0000039 << 0.01$.

Tout semble s'être correctement déroulé, mais que se passe-t-il si on diminue ϵ et quelle confiance peut-on avoir dans ce résultat ?

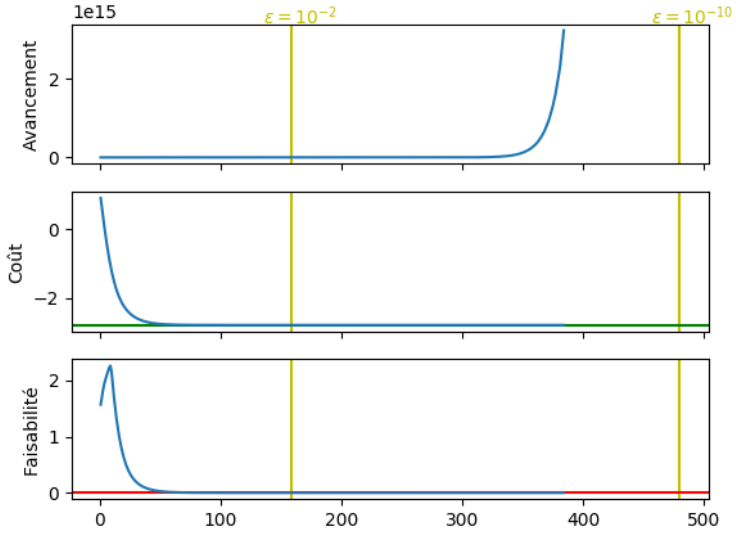
```
1 A = [[ 3, -3],
2      [ 1,  1],
3      [-5, -1],
4      [-1,  2],
5      [ 1, -6]]
6 b = [[3], [2], [5], [2], [4]]
7 c = [[-1],[ 4]]
```

Figure 7.2 – Problème linéaire de dimension 2(repris de la section 2.2)



En vert, on retrouve l'ensemble faisable. Chaque contrainte est représentée par une couleur différente : cyan, violet, bleu, noir, gris (ordre des lignes de (A, b)). En rouge, on a représenté la fonction de coût appliquée à l'optimum. Le point vert est le centre analytique et les points bleus représentent les différentes itérations de l'algorithme (formant le chemin central).

FIGURE 7.3 – Itérés de l'algorithme primal pour le problème de la figure 7.2



En abscisse on a représenté le numéro de l'itération. Les droites jaunes représentent la borne sur le nombre d'itérations pour deux ϵ donnés. La droite verte représente le coût optimal et la droite rouge représente la valeur au-delà de laquelle notre point n'est plus faisable. Des valeurs sont théoriquement générées jusqu'à la seconde droite jaune. Cependant, nous n'affichons pas les points non représentables(`nan`).

FIGURE 7.4 – Évolution de l'avancement t, du coût et de la faisabilité pour le problème 7.2

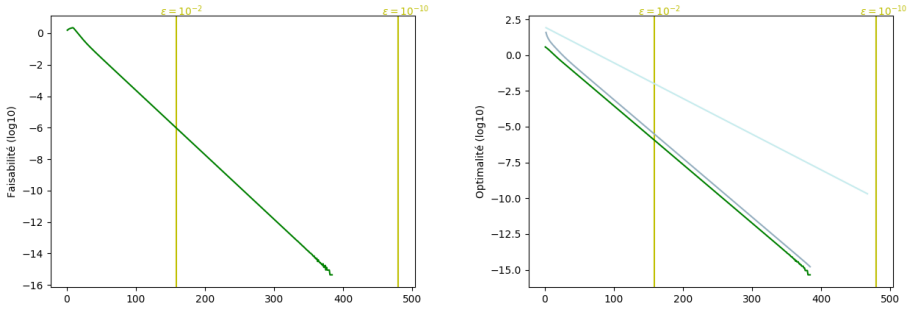
7.4.2 Évolution de la faisabilité et de l'optimalité

Si on lance l'algorithme avec $\epsilon = 10^{-10}$ on obtient le résultat `[[nan], [nan]]`. On a donc obtenu une erreur de calcul. Comme nous avons un algorithme prouvé, cela ne peut venir que d'une seule source : les calculs flottants. Nous allons essayer de voir dans cette sous-partie ce qui a pu se passer exactement. Pour cela, nous avons fait afficher à notre code la valeur des différentes variables du programme pour chaque itération et nous allons tracer ses valeurs afin de détecter à partir de quand l'algorithme commence à produire des erreurs.

En optimisation, deux grandes propriétés nous intéressent, l'optimalité et la faisabilité. Elles s'appliquent pour un point donné et nous allons donc associer pour chaque point une valeur représentant ces propriétés. $\mathcal{O} : \mathbb{R}^n \to \mathbb{R}$ représente l'optimalité et se définit par $\mathcal{O}(x) = \langle c, x\rangle - \langle c, x^*\rangle$ avec x^* la solution du problème. Plus elle est basse, plus notre point est optimal. La faisabilité est représentée par $\mathcal{F} : \mathbb{R}^n \to \mathbb{R}$ définit par $\mathcal{F}(x) = \min\limits_{0\leq i<n} b_i - A_i x$.

La figure 7.4, représente l'évolution de notre algorithme sur l'exemple 7.2. Sur cette figure on peut déjà remarquer que notre algorithme arrête de générer des nombres représentables à partir de la 384e itération. Les autres valeurs semblent se comporter correctement, t augmente exponentiellement, le coût décroit fortement pour se rapprocher de l'optimal et la faisabilité décroit elle aussi ce qui est normal vu que le point optimal est toujours sur la bordure (donc à la limite de la faisabilité).

Afin de mieux visualiser ce qui se passe, nous allons tracer les mêmes courbes en échelle logarithmique. On peut trouver l'évolution de la faisabilité dans la figure 7.5. Dans les deux cas, on observe une droite décroissante donc une exponentielle décroissante. Ceci confirme ce qui était attendu. Cependant, sur les dernières itérations, des variations autour de la droite montrent l'apparition d'erreurs de calcul.



En vert, à gauche on retrouve $\mathcal{F}$ et à droite $\mathcal{O}$ en échelle logarithmique. Les droites jaunes représentent toujours le nombre d'itérations. La courbe bleu foncé représente la borne sur l'optimalité que l'on a dans le cadre de la boucle while et la courbe bleu clair celle dans le cas de la boucle for.

FIGURE 7.5 – Évolution de l'optimalité $\mathcal{O}$ et de la faisabilité $\mathcal{F}$

7.4.3 Évolution de la condition de centrage approchée

Si on regarde les valeurs des variables du programme juste avant que l'algorithme génère des erreurs flottantes, on observe que la 5e contrainte est violée. En effet, on a $b_4 - A_4 x = 0$ pour le x fautif. La première intuition serait que $b_4 - A_4 x$ est trop proche de zéro et aurait donc été arrondi à zéro. Cependant, un rapide calcul en précision arbitraire montre que le calcul était bien correct. Or on sait que si un point est faisable (ce qui est le cas du point précédent) et qu'il vérifie la condition de centrage approché alors le nouveau point est faisable (cf. théorème 4.2.4). Ainsi nécessairement, la condition de centrage approché est violée. Nous avons donc tracé dans la figure 7.6 l'évolution de la condition de centrage approché. On remarque alors qu'elle est violée depuis plus de 50 itérations.

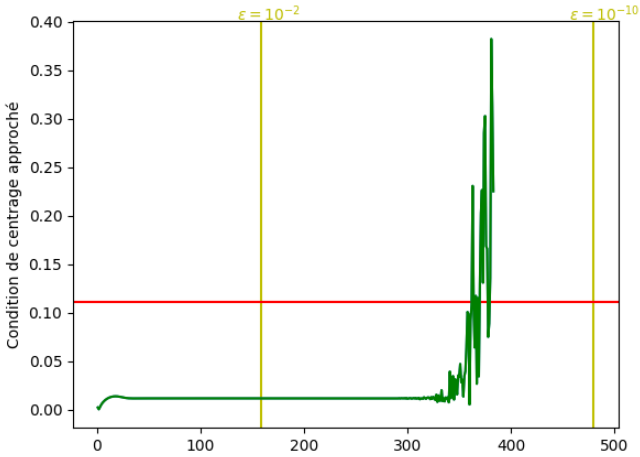
7.4.4 Évolution de l'erreur pour chaque étape de l'algorithme

Afin de localiser la source de ce problème, nous avons tracé l'erreur commise par chaque étape de l'algorithme dans la figure 7.7. Nous avons pu obtenir ces résultats en exécutant pour chaque itération les différentes étapes en précision flottante arbitraire grâce à la librairie sympy(MEURER et al., 2017). Pour chaque étape on a récupéré les valeurs des variables avant son exécution, on a effectué le même calcul en précision arbitraire puis on a calculé la différence relative entre les deux valeurs.

On voit que le calcul de t et de x se passe correctement, en revanche les calculs de la hessienne et du gradient commettent une erreur toujours plus grande à chaque itération. Ainsi, il est logique de finir par arriver à des erreurs significatives. On peut en revanche s'étonner que cette erreur ne se répercute pas sur le calcul final de x mais on peut supposer que l'erreur sur le gradient et la hessienne s'annulent lors du calcul du nouveau x.

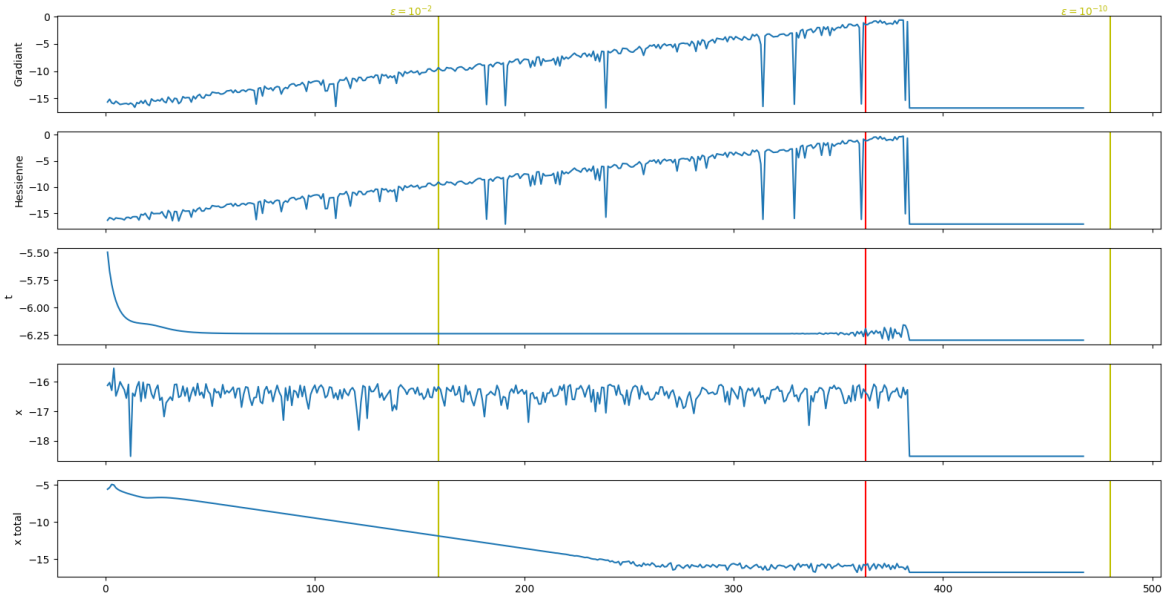
7.4.5 Évolution de l'erreur sur $b - A \cdot x$

Enfin, on a tracé dans la figure 7.8, l'erreur commise sur le calcul de $b - A \cdot x$ pour chaque composante, ainsi que l'erreur commise lors du passage à l'inverse. Elles sont toutes négligeables sauf pour le calcul de $b - A \cdot x$ de la première et la dernière contrainte. Ces deux contraintes correspondent justement aux deux contraintes qui seront violées si on atteint l'optimal. En effet, le fait que ces deux contraintes soient violées à l'optimal signifie que les composantes correspondantes dans $b_k - A_k \cdot x$ doivent tendre vers zéro le long de l'algorithme.



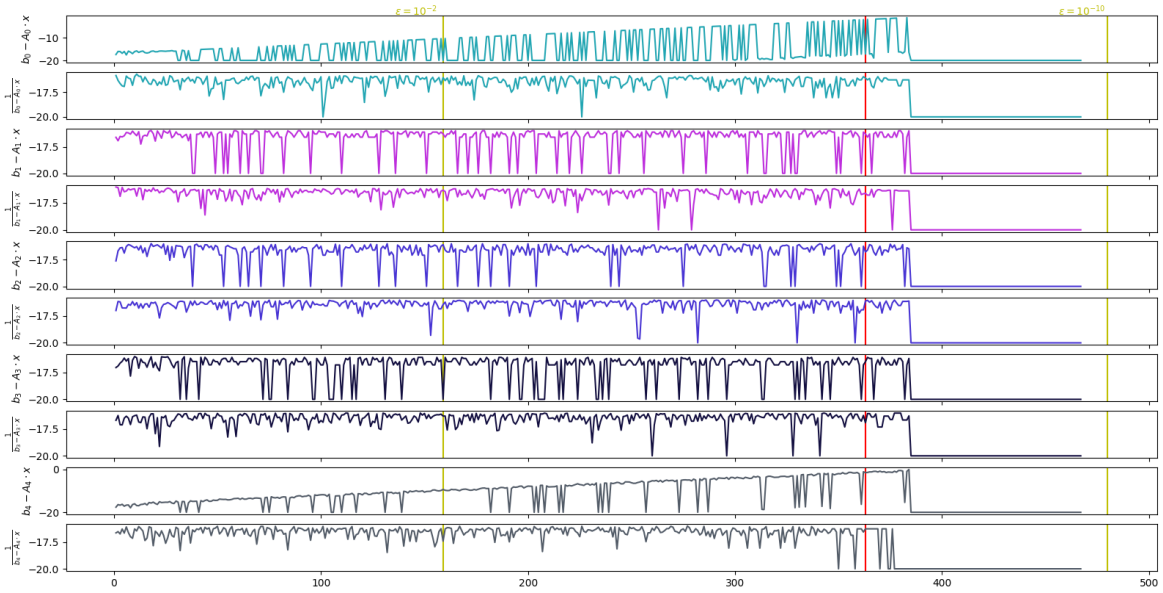
La courbe verte représente la quantité $\left\|\tilde{f}'(x,t)\right\|_x^*$, la droite rouge représente β. Les droites jaunes représentent la borne sur le nombre d'itérations pour deux ϵ données. Tant que la courbe verte est en dessous de la droite rouge, la condition de centrage approché est validée.

FIGURE 7.6 – Évolution de la condition de centrage approché



Chaque courbe représente l'erreur commise par l'étape correspondante de la boucle. Elle est affichée en logarithme base 10 (cela permet de voir directement à quelle décimale l'erreur apparaît). La dernière courbe représente l'erreur commise sur x par la boucle entière. La droite rouge représente l'itération où la condition de centrage approchée est violée.

FIGURE 7.7 – Erreur commuse par les différentes étapes de la boucle



La première courbe représente l'erreur commise sur la première composante du calcul de $b - A \cdot x$. La seconde l'erreur commise sur l'inversion de cette première composante. Puis il en va de même pour les deux suivantes avec la seconde composante et ainsi de suite. Les courbes sont toujours en logarithme base 10.

FIGURE 7.8 – Erreur sur $b - A \cdot x$

Ainsi, on a une somme de nombres flottants valant zéro. On ne peut pas atteindre de cette manière un nombre inférieur à 10^{-16}. Afin de remédier à ce problème il faudrait effectuer le calcul du gradient et de la hessienne sans effectuer directement le calcul de $b_k - A_k \cdot x$. Ce qui reste à faire pour un travail ultérieur.

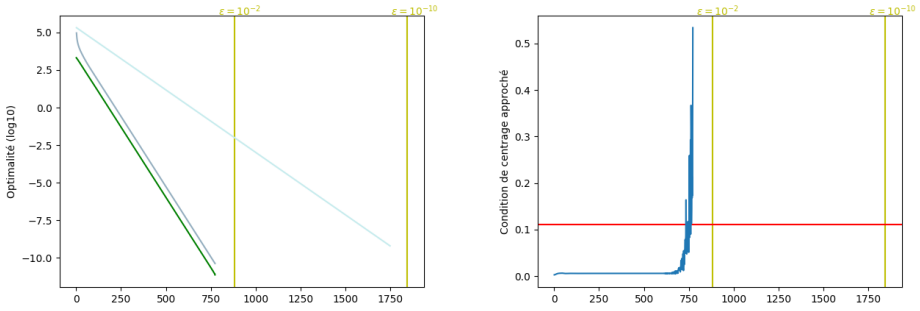
On a donc pu isoler la source du problème, rien ne garantit qu'en le corrigeant d'autres problèmes n'apparaîtront pas, mais ça serait la première étape pour diminuer les erreurs dues aux flottants.

Pour notre exemple de petite dimension, les erreurs interviennent quand on peut encore garantir une précision de 6 chiffres significatifs avec une précision réelle de 10. En revanche, si on augmente les dimensions et le nombre de contraintes, les erreurs vont apparaître plus tôt. Par exemple, la figure 7.9 montre l'évolution de la condition de centrage ainsi que l'optimalité pour un problème plus gros.

On voit dans cet exemple de plus grande dimension que les problèmes flottants arrivent avant même que l'on puisse garantir une précision de deux chiffres après la virgule. La précision réelle est elle cependant d'une dizaine de chiffres significatifs et cette précision peut être atteinte avec l'algorithme contenant la boucle while au lieu de la boucle for.

7.4.6 Bilan

On a pu voir dans cette section que les problématiques flottantes étaient primordiales. L'algorithme peut se retrouver à calculer avec des nombres non représentables rendant non exploitable son résultat. L'algorithme actuel subit les conséquences du modèle flottant même pour des problèmes de taille raisonnable. La taille à partir de laquelle les problèmes apparaissent pourrait être repoussée en trouvant une meilleure fonction `lower` pour borner le nombre d'itérations. Quoi qu'il en soit, nous n'avons aucune garantie que dans certains cas les erreurs flottantes puissent arriver beaucoup plus tôt. Ainsi, pour avoir un algorithme com-



On retrouve le graphique de la faisabilité à gauche et celui de la condition de centrage approché à droite, mais pour un problème de taille 50×10.

FIGURE 7.9 – Faisabilité et condition de centrage approché pour un problème plus grand

plètement certifié il faudrait reprendre le travail de preuve, mais avec un modèle flottant. On pourrait alors utiliser différente méthode comme ce qui a pu être fait dans GOUBAULT et PUTOT, 2006, BOLDO et MARCHÉ, 2011 ou GOODLOE et al., 2013. Il faudrait alors exprimer formellement à partir de quelle précision ϵ on ne peut plus garantir un résultat pour un problème donné et montrer que si on vérifie cette contrainte alors l'algorithme est correct. Cela représente un certain travail et ce manuscrit peut être vu comme la première étape vers celui-ci.

7.5 À qui doit-on faire confiance ?

Dans cette thèse nous avons développé un nouveau langage PYSIL, un compilateur pour celui-ci et nous avons écrit un algorithme de points intérieurs ainsi que sa spécification dans ce langage. Le compilateur permet de générer un code C annoté en ACSL. Nous avons ensuite prouvé ces annotations en utilisant Frama-C. Ainsi on peut dire qu'à la fin nous avons un code C prouvé correct. Cette appellation peut être considérée comme abusive, car elle repose sur un certain nombre d'hypothèses, comme la correction de certains logiciels utilisés ou la validité de certaines annotations. C'est pourquoi, dans cette section, nous allons lister tout ce à quoi nous devons faire confiance pour pouvoir ensuite affirmer la correction du code final. Nous nous plaçons dans la situation où le code a été prouvé avec un modèle flottant. Si ce n'est pas le cas, c'est une source d'erreur, la principale d'ailleurs. Nous verrons dans une première sous-section tous les logiciels dans lesquels nous devons avoir confiance, dans la deuxième nous énumèrerons quelles annotations doivent être vérifiées correctes par un moyen tiers et enfin, dans une troisième sous-section, nous verrons quelles parties du code ne sont pas encore prouvées.

7.5.1 Logiciel

Compilation C

Nous produisons et vérifions le code C, mais ce n'est pas ce code qui sera embarqué, mais sa version compilée. Le compilateur effectuant cette tâche peut lui aussi contenir des erreurs comme le rappelle YANG et al., 2011. Il est donc important de faire confiance aussi à cette étape de compilation. On peut utiliser, par exemple, CompCert présenté dans LEROY, 2009a qui a été formellement prouvé en Coq.

Frama-C

Frama-C est le logiciel qui parcourt le code C/ACSL généré. Grâce à son plug-in WP, il effectue un calcul de plus faible précondition sur le code et génère une formule logique qui peut ensuite être prouvée par un solveur SMT ou un assistant de preuve. Toute erreur lors du parcours du code, du calcul de plus faible précondition ou de la génération de la formule logique peut entrainer la validation d'un code pourtant incorrect. Il est donc primordial de faire confiance à Frama-C.

Alt-Ergo

Une fois la formule logique générée par Frama-C il est nécessaire de la prouver. Pour cela nous avons décidé d'utiliser majoritairement Alt-Ergo. Nous avons quelques buts validés par d'autres moyens comme Coq ou d'autres solveurs SMT comme Vampire ou Z3. Cependant il doit être possible à terme de faire valider tous les buts par Alt-Ergo. On en voit ici l'intérêt : il suffit de s'assurer de sa correction à lui. Dans le cas d'une utilisation de plusieurs solveurs, il faudrait faire confiance à tous. De plus, utiliser d'autres solveurs que Coq ou Alt-Ergo implique de passer par Why3 et donc impose de lui faire confiance aussi.

Bilan

Pour s'assurer de la correction d'un logiciel, on peut compter sur la relecture, le test, l'utilisation à grande échelle [1] ou la preuve formelle. Les deux premières solutions sont déjà utilisées, mais apportent des garanties relatives, une erreur se produisant rarement peut facilement leur échapper. La troisième est encore loin d'être atteinte bien que l'utilisation de Frama-C dans un contexte industriel existe Duprat et al., 2016. On peut aussi noter l'utilisation de CAVEAT, un ancêtre de Frama-C, ainsi que de AltErgo par Airbus pour l'A380. Ceci a d'ailleurs entrainé leur qualification aéronautique. Ainsi ces méthodes apportent des garanties limitées, mais existent partiellement. Il reste la dernière solution, la preuve formelle comme elle a pu être pratiquée dans ce travail. Elle n'a pas encore été pratiquée sur ces logiciels. Cela n'est pas chose aisée, car ils sont très complexes et les moyens sont limités, car il ne faut pas réaugmenter le nombre de logiciels à prouver.

Une autre solution que la vérification du logiciel lui-même est la production par le logiciel d'un certificat pouvant être vérifié par un programme tiers ensuite. Cette solution pourrait être particulièrement utile pour Alt-Ergo qui est parfois capable de fournir une preuve.

7.5.2 Annotation

Au-delà des logiciels, nous devons faire confiance à la spécification que l'on donne au programme que l'on souhaite prouver. En l'occurrence, nous générons cette spécification, on pourrait faire confiance à notre générateur, mais il est beaucoup plus simple et sûr de faire confiance directement à la spécification générée. On parle là de la spécification de la fonction principale de notre code. C'est à dire de son contrat. Ce contrat utilise l'axiomatique des matrices ainsi que celui d'optimisation. Ces axiomatiques sont composées de définitions d'opérateurs. Il faut, pour chaque opérateur, vérifier que sa définition correspond à celle attendue. En plus de ceci on doit aussi s'assurer que les théorèmes qui ont été admis sont bien correctement écrits.

1. Par exemple GCC est réputé fiable de par sa très grande utilisation dans de nombreux domaines différents.

```
/*@ requires mat_gt(b, mat_mult(A, MatVar((double*) pathfollowing_X,
        2, 1)));
  @ requires acc(A, b, c, 0, MatVar((double*)pathfollowing_X, 2, 1),
        BETA);
  @ ensures mat_gt(b, mat_mult(A, MatVar((double*)pathfollowing_X,
        2, 1)));
  @ ensures dot(MatVar((double*)pathfollowing_X, 2, 1), c)
        - sol(A, b, c) < EPSILON;*/
void pathfollowing();
```

FIGURE 7.10 – Contrat ACSL de la fonction principal pour un problème de taille 2×1.

Le contrat principal

Le contrat principal est celui de la fonction `pathfollowing`. On peut trouver sa version PYSIL dans la figure 4.10 ainsi que sa transcription en ACSL dans la figure 7.10. Ce contrat suppose que la fonction est appelée avec un point proche du centre analytique et dans l'ensemble. Le contrat, par ailleurs, garantit que le point final est faisable et proche à ϵ de l'optimum.

La correction de ce contrat nécessite de s'accorder sur la définition des opérateurs matriciels impliqués ainsi que la définition de l'opérateur `sol`. Ceux-ci sont définis respectivement dans l'axiomatique `matrix` et `optim`.

L'axiomatique `matrix`

C'est l'axiomatique qui définit toutes les opérations sur les matrices. On y définit un type pour toutes les matrices : `LMat`. Les opérateurs permettant de décrire un élément de son type sont `getM`, `getN` et `mat_get` renvoyant respectivement le nombre de lignes, le nombre de colonnes et la valeur de l'élément à une position donnée. On a ensuite un certain nombre d'opérateurs permettant de créer une matrice, `identity`, `unit`, `MatCst` et `Matvar`. Puis toutes les opérations internes ou semi-internes : la transposition, l'inversion, la multiplication interne et par un scalaire, l'addition et le produit scalaire. On retrouve aussi quelques prédicats : `isInversible`, `isNull`, l'égalité et l'inégalité. En général, chaque opérateur est défini par trois axiomes, un pour `getM`, `getN` et pour `mat_get`. Il faut s'assurer que les opérateurs intervenant dans le contrat de base ou dans tout énoncé que l'on suppose correct sont définis correctement. De plus, on doit s'assurer que les axiomes n'apportent pas d'inconsistance.

L'axiomatique `optim`

C'est l'axiomatique qui définit les opérateurs liés à l'optimisation, on retrouve `sol` qui représente la solution d'un problème d'optimisation linéaire donné. On retrouve des prédicats spécifiant qu'un problème linéaire donné est correct. On retrouve ensuite les opérateurs de gradient et de hessienne, et des définitions directes des normes locales et de la condition de centrage approché. Une axiomatique séparée est aussi générée et contient la définition de l'opérateur `lower`.

Les théorèmes

On a admis un certain nombre de théorèmes listés dans la sous-section 6.1.1. Toute la preuve du code repose sur ces quelques théorèmes, il faut donc aussi s'assurer de leur correction.

Bilan

On a donc une certaine quantité d'annotations ACSL dont la correction doit être vérifiée. Il restera toujours quoiqu'il arrive une partie que seul un humain pourra vérifier, c'est celle décrivant ce que l'on attend de l'algorithme. Le seul moyen d'éviter cette partie est de la placer dans un programme plus grand qui intégrerait la fonction, mais ce programme doit nécessairement avoir lui-même un contrat principal, qu'il faut vérifier, donc la partie humaine ne peut pas être exclue.

Pour les autres annotations, en particulier les axiomatiques, on peut effectuer une relecture manuelle ou bien les prouver avec des théories plus générales. En l'occurrence, remonter vers des définition et théorème plus généraux en ACSL représenterait un travail considérable. Il semble plus adéquat de transcrire la librairie ACSL dans un assistant de preuve puis de prouver au sein de cet assistant la correction de la librairie. Cela peut se faire par plongement de la théorie ACSL dans une théorie des matrices préexistante. On peut faire de même pour l'axiomatique définissant l'optimisation. Cependant, il n'est pas certain de trouver, au sein de l'assistant de preuve, une librairie d'optimisation préexistante. Il faudrait alors l'écrire et la prouver afin d'effectuer le plongement.

Il faut noter que cela n'enlève en rien la nécessité pour la personne s'assurant de la correction du contrat principal de vérifier au sein des théories que les opérateurs utilisés dans le contrat correspondent à sa propre compréhension de ce que font les opérateurs.

Enfin, concernant les théorèmes, ils nécessitent des résultats d'analyse et d'algèbre conséquents que l'on peut espérer trouver dans des assistants de preuve comme Coq Boldo, Lelay et Melquiond, 2015. Ainsi, si les librairies ont été plongées dans Coq on pourra aussi plonger les théorèmes et faire leurs preuves dans l'assistant.

7.5.3 Code

Afin de simplifier le travail, nous avons dû supposer quelques portions de code comme correctes ou repousser leur preuve à des travaux ultérieurs.

Calcul du gradient et de la hessienne

Nous avons décidé de laisser le calcul du gradient et de la hessienne de côté, car dans un premier temps il n'était pas évident de savoir quel formalisme conviendrait pour leur axiomatisation. A posteriori, le mieux serait de les définir par leur expression finale et non comme des dérivées. On renvoie, ainsi, les problématiques de dérivations dans la preuve des théorèmes admis. On pourra alors prouver cette partie du code.

Décomposition de Cholesky

Nous utilisons la décomposition de Cholesky pour résoudre deux équations linéaires. Nous avons décidé de ne pas du tout nous intéresser à cette partie du code. Prouver un algorithme de Cholesky est un travail complexe et intéressant par lui même. Son implémentation, en particulier dans le cadre de l'arithmétique flottante, a déjà pu être étudiée dans Roux, 2016 et Martin-Dorel et Roux, 2017.

Calcul du centre analytique, nombre d'itérations

Pour une utilisation embarquée, le calcul du centre analytique doit se faire en ligne et il conviendrait donc de vérifier aussi cet algorithme. Celui-ci est de même nature que l'algorithme principal. On possède donc déjà tous les outils nécessaires à sa preuve et elle ne devrait donc pas demander un travail trop important.

Cela pose aussi une limitation sur le contrôle du nombre d'itérations. En effet, pour borner le nombre d'itérations on a besoin du centre analytique qui sera calculé en ligne et non à la compilation. Ceci rend impossible le calcul du temps d'exécution à la compilation. Il faudrait donc trouver un moyen de borner, si nécessaire largement, le temps d'exécution de manière indépendante du centre analytique. Cette problématique est débattue dans la section 7.1.

Modèle flottant

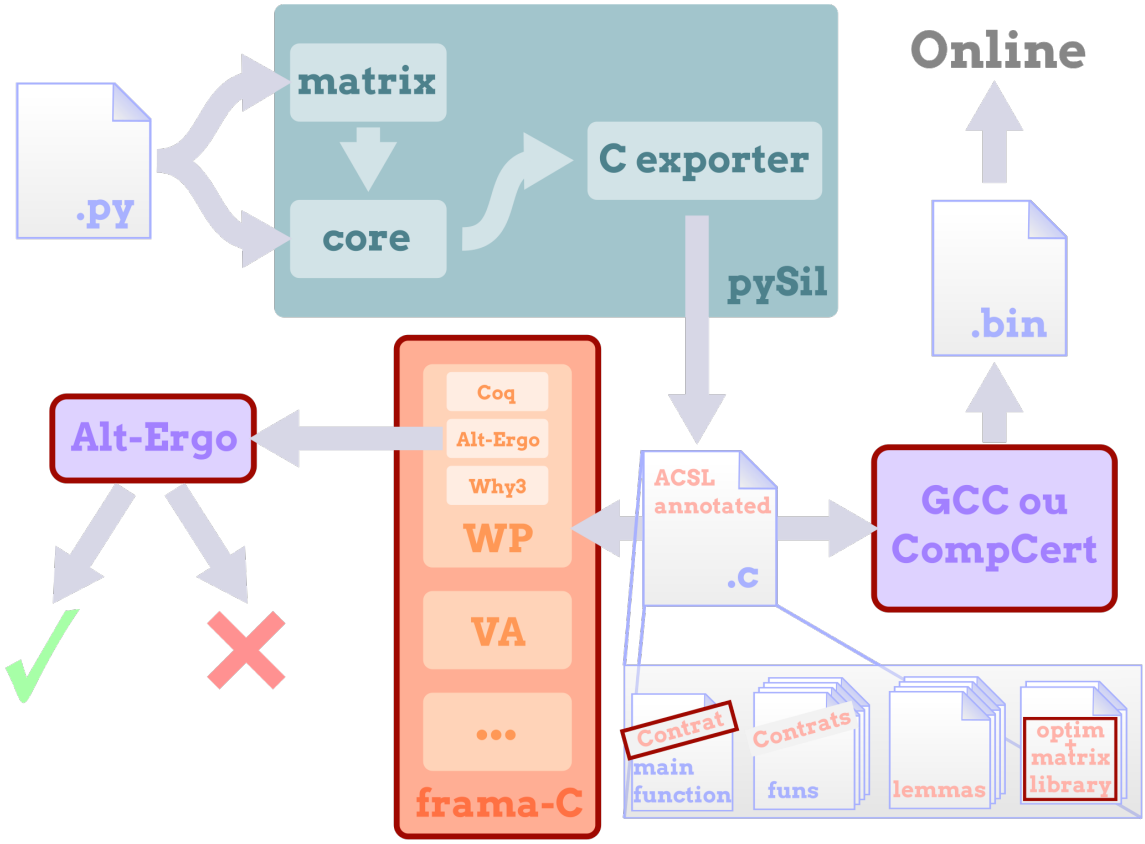
On a pu voir dans la section 7.4 que nous avions laissé de côté les aspects calculs flottants. Il faudrait reprendre le travail en utilisant un modèle flottant dans Frama-C à la place du modèle Réel comme ce qui a pu être fait dans Boldo et Marché, 2011 ou Goodloe et al., 2013. On pourrait aussi analyser l'algorithme comme ce qui est présenté dans Goubault et Putot, 2006 afin d'avoir une garantie mathématique qui pourrait ensuite être transposé vers Frama-c.

Bilan

On a vu dans cette partie que certaines portions de code nécessitant encore d'être prouvées. Elles ne possèdent pas de difficulté particulière, uniquement du temps serait nécessaire. Ainsi, ces limitations ne sont que temporaires et dans un produit final n'apparaîtraient pas. Pour le calcul flottant, cela nécessite en revanche un travail plus important qui reste à développer.

7.5.4 Conclusion

On a donc vu dans cette partie sur quelles hypothèses repose notre travail. Ces hypothèses impliquent qu'un humain vérifie leur correction à la main. On pourrait automatiser cette tâche pour bon nombre d'entre elles moyennant un travail plus ou moins conséquent. En revanche, deux hypothèses resteront toujours hors de portée d'une quelconque automatisation. La première est matérialisée dans le contrat de la fonction principale qui est l'expression informatique de l'attente que l'on a sur le code. L'autre hypothèse est la correction d'un logiciel de vérification. Afin de vérifier automatiquement un logiciel, il faut un autre logiciel, ainsi au moins un logiciel devra être vérifié par un autre moyen. Ce logiciel est généralement un assistant de preuve dont le coeur est assez minimal pour être vérifié à la main. On peut remarquer qu'on n'a pas besoin de faire confiance au compilateur pySil ou même au code écrit en pySil. Ceci est une volonté de réduire autant que possible les hypothèses effectuées et de ne pas devoir faire confiance à un logiciel de plus. On peut retrouver dans la figure 7.11, l'ensemble de la chaine de génération avec les hypothèses mises en évidence.



On a entouré en bordeaux les parties auxquels on doit faire confiance.

FIGURE 7.11 – Confiance dans la chaîne de génération et de preuve

Bibliographie

Ballabriga, Clément, Hugues Cassé, Christine Rochange et Pascal Sainrat (oct. 2010). "OTAWA : An Open Toolbox for Adaptive WCET Analysis". In : *8th IFIP WG 10.2 International Workshop on Software Technologies for Embedded and Ubiquitous Systems (SEUS)*. Sous la dir. de Sang Lyul Min ; Robert Pettit ; Peter Puschner ; Theo Ungerer. T. LNCS-6399. Software Technologies for Embedded and Ubiquitous Systems. Waidhofen/Ybbs, Austria : Springer, p. 35-46 (cf. p. 146).

Boldo, Sylvie, Catherine Lelay et Guillaume Melquiond (2015). "Coquelicot : A User-Friendly Library of Real Analysis for Coq". In : *Mathematics in Computer Science* 9.1, p. 41-62 (cf. p. 157).

Boldo, Sylvie et Claude Marché (2011). "Formal verification of numerical programs : from C annotated programs to mechanical proofs". In : *Mathematics in Computer Science* 5, p. 377-393 (cf. p. 154, 158).

Duprat, Stéphane et al. (jan. 2016). "Spreading Static Analysis with Frama-C in Industrial Contexts". In : *8th European Congress on Embedded Real Time Software and Systems (ERTS 2016)*. TOULOUSE, France (cf. p. 155).

Févotte, François et Bruno Lathuilière (2017). "Studying the Numerical Quality of an Industrial Computing Code : A Case Study on Code_aster". In : *Numerical Software Verification*. Sous la dir. d'Alessandro Abate et Sylvie Boldo. Cham : Springer International Publishing, p. 61-80 (cf. p. 148).

Goodloe, Alwyn E., César Muñoz, Florent Kirchner et Loïc Correnson (2013). "Verification of Numerical Programs : From Real Numbers to Floating Point Numbers". In : *NASA Formal Methods*. Sous la dir. de Guillaume Brat, Neha Rungta et Arnaud Venet. Berlin, Heidelberg : Springer Berlin Heidelberg, p. 441-446 (cf. p. 154, 158).

Goubault, Eric et Sylvie Putot (2006). "Static Analysis of Numerical Algorithms". In : *Static Analysis*. Sous la dir. de Kwangkeun Yi. Berlin, Heidelberg : Springer Berlin Heidelberg, p. 18-34 (cf. p. 154, 158).

Leroy, Xavier (2009a). "A formally verified compiler back-end". In : *Journal of Automated Reasoning* 43.4, p. 363-446 (cf. p. 154).

Li, Xianfeng, Yun Liang, Tulika Mitra et Abhik Roychoudhury (2007). "Chronos : A timing analyzer for embedded software". In : *Science of Computer Programming* 69.1. Special issue on Experimental Software and Toolkits, p. 56-67 (cf. p. 146).

Martin-Dorel, Érik et Pierre Roux (2017). "A reflexive tactic for polynomial positivity using numerical solvers and floating-point computations". In : *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs, CPP 2017, Paris, France, January 16-17, 2017*. Sous la dir. d'Yves Bertot et Viktor Vafeiadis. ACM, p. 90-99 (cf. p. 157).

Mattingley, Jacob et Stephen Boyd (mar. 2012). "CVXGEN : a code generator for embedded convex optimization". In : *Optimization and Engineering* 13.1, p. 1-27 (cf. p. 4, 16, 147).

Meurer, Aaron et al. (jan. 2017). "SymPy : symbolic computing in Python". In : *PeerJ Computer Science* 3, e103 (cf. p. 48, 151).

Roux, Pierre (2016). "Formal Proofs of Rounding Error Bounds - With Application to an Automatic Positive Definiteness Check". In : *J. Autom. Reasoning* 57.2, p. 135-156 (cf. p. 157).

Yang, Xuejun, Yang Chen, Eric Eide et John Regehr (juin 2011). "Finding and Understanding Bugs in C Compilers". In : *SIGPLAN Not.* 46.6, p. 283-294 (cf. p. 154).

Chapitre 8

Conclusion

Sommaire

8.1 Résumé des contributions

Ce travail a permis de développer dans le chapitre 3 un nouveau langage nommé pySil. Ce langage permet d'exprimer de manière simple et lisible un algorithme impératif avec calcul matriciel. De plus, il permet aussi de l'annoter avec sa spécification sous forme de triplet de Hoare. Nous avons présenté une sémantique pour ce langage. Elle nous a permis de prouver que l'extension matricielle pouvait être replongée dans le cœur impératif du langage.

Par la suite, au chapitre 5, nous avons développé un générateur de pySil vers C/ACSL pour ce langage, nous générons à la fois le code C correspondant à l'algorithme, mais aussi ses annotations en ACSL. Chaque calcul matriciel est donc généré automatiquement, il est de plus annoté par l'équation matricielle correspondante. Cette annotation est ensuite automatiquement prouvée par Frama-C/Alt-Ergo.

Au chapitre 4, nous avons utilisé le langage pySil pour écrire et annoter un algorithme de points intérieurs primal pour l'optimisation linéaire. Nous avons aussi expliqué dans ce chapitre comment l'utiliser pour un problème de commande prédictive.

Dans le chapitre 6, nous avons développé des moyens permettant de prouver toutes les obligations de preuve générées par Frama-C sur le code C généré à partir de l'algorithme écrit au chapitre 4. Pour cela, nous avons dû écrire un grand nombre de lemmes en pySil et fragmenter le code originel en plusieurs fonctions n'effectuant qu'une seule opération facilement spécifiable. Nous avons finalement prouvé tous les contrats et buts principaux. De plus, l'utilisation de solveur SMT pour la preuve des obligations Frama-C permet de lui donner une assez grande robustesse.

Si on trace le bilan de tout ceci, nous avons créé un cadre logiciel permettant la génération d'un code certifiable par preuve formelle résolvant des problèmes d'optimisation. Nous générons un code C sans allocations dynamiques, sans dépendances et sans boucles while. Ainsi, il peut être très facilement embarquable dans un système critique. On peut noter que les garanties que nous apportons sur le code ne reposent pas sur notre code, mais sur des programmes tiers bien plus généralistes et déjà utilisés donc plus facilement certifiables.

Ainsi, vérifier l'implémentation d'un algorithme d'optimisation complexe est faisable. Les premiers résultats de cette thèse en particulier du chapitre 4 ont pu être publié dans Cohen et al., 2017, Feron et al., 2017. Nous avons aussi un article accepté à LPAR-22 Davy et al., 2018 qui reprend les résultats du chapitre 6.

8.2 Limitations

Au chapitre 7, nous avons extrait un certain nombre de limitations à notre travail que nous résumons ici.

Le temps d'exécution de notre programme est assez prohibitif comparé à ce qui se fait aujourd'hui. Cela est dû à deux raisons. La première est que nous avons choisi un algorithme plus simple à implémenter, mais qui est moins efficace. La seconde raison est que nous calculons à l'avance le nombre d'itérations. Pour obtenir ce nombre, nous effectuons des surapproximations ce qui entraine une borne plus importante que le nombre d'itérations réellement nécessaires. Par ailleurs, même avec ces surapproximations, nous n'avons pas trouvé de méthode systématique pour calculer cette borne, dans le cas de la paramétrisation.

Nous utilisons un modèle réel pour la preuve du code généré. Cependant, le code s'exécute avec un modèle de nombres flottants. Ainsi, nous avons un décalage entre le code que nous prouvons et le code réellement exécuté. L'utilisation de nombres flottants de grandes précisions permet en général de repousser la plupart des problèmes. Cependant, cela reste une faille au niveau de la certification.

La preuve de correction que nous apportons sur le code repose sur un certain nombre d'hypothèses que nous pouvons rassembler en trois groupes. Le premier est celui des logiciels,

un bug dans Frama-C, Alt-Ergo ou Coq pourrait déclarer notre code valide sans qu'il soit effectivement correct. Le code C est ensuite lui-même compilé apportant une nouvelle source potentielle d'erreurs. Le second groupe est celui des annotations. Le contrat principal de l'algorithme doit être vérifié par l'utilisateur, car c'est elle qui représente ce qu'on attend de lui. Dans les annotations, on retrouve aussi la librairie de matrice, d'optimisation et un certain nombre de théorèmes d'optimisation admis. Toutes ces annotations sont aussi des sources potentielles d'erreur. Enfin, le troisième groupe est celui du code : nous n'avons pas abordé la preuve de l'algorithme de Cholesky ainsi que celle de la partie purement calculatoire du gradient et de la hessienne.

8.3 Perspectives

Nous pouvons, dans un premier temps, extraire de chacune de nos limitations une perspective visant à lui trouver une solution.

Concernant le temps d'exécution, on pourrait mener une analyse mathématique un peu plus poussée de la borne sur le temps d'exécution afin d'essayer d'améliorer la borne ou à défaut prouver qu'elle est optimale. Par ailleurs, travailler sur la borne serait aussi l'occasion de trouver une façon systématique de la calculer.

Pour la problématique des nombres flottants, deux grands axes se distinguent. Le premier, empirique, consisterait à chercher les zones du programme sources de potentielles erreurs flottantes afin de les corriger. Le second serait de passer à une sémantique de nombre flottant dans la preuve. Ceci est une simple option dans Frama-C, mais cela nécessiterait de retravailler une bonne partie de la preuve.

Afin de renforcer la confiance dans la preuve, on pourrait prouver notre implémentation de Cholesky et le calcul du gradient. Un travail plus complexe, consistant à prouver formellement la correction des axiomatiques utilisées et les théorèmes admis, pourrait être entrepris, en Coq par exemple.

Dans un second temps, nous avons effectué certains choix de conception pour des buts précis que nous n'avons pas toujours pu pleinement exploiter. Ce sont donc des perspectives naturelles à notre travail.

La première concerne l'optimisation du code généré. Nous n'utilisons finalement que de manière très limitée les informations très précises sur le problème d'optimisation que nous avons à la compilation. Nous avons développé un système de génération de code très modulaire à cette fin. Nous pourrions donc facilement implémenter des optimisations de code en particulier sur le calcul matriciel.

La seconde concerne l'algorithme utilisé. Afin de partir sur une base simple, nous avons implémenté un algorithme primal pour des problèmes d'optimisations linéaires. Cet algorithme moyennant une modification légère du code et de la preuve, permettrait tout à fait de fonctionner sur des problèmes SDP. Par ailleurs, l'algorithme utilisé n'est pas le plus efficace, mais il a un fonctionnement proche des meilleurs algorithmes. On pourrait donc reprendre le travail du chapitre 4 et 6 sur un nouvel algorithme plus efficace. La proximité entre les deux algorithmes rendrait la tâche beaucoup plus simple, une partie du travail ayant déjà été fait.

La troisième perspective est un retour au contexte initial du contrôle. On pourrait utiliser ce travail sur un exemple concret de commande prédictive. On pourrait alors placer ce code au sein d'un programme plus général entièrement prouvé. Cela donnerait l'occasion de voir comment se combinent les preuves des différents composants logiciels.

Dans un troisième temps, nous souhaitons rendre les outils présentés dans cette thèse plus accessibles. Quelques parties de ce travail pourrait être rendu indépendantes et réutilisées. C'est le cas de l'axiomatique sur les matrices générée en ACSL que l'on pourrait extraire

afin d'être rendue publique et directement utilisable. PYSIL dans sa globalité, moyennant un travail de documentation pourrait être réutilisé dans d'autre contexte impliquant de la génération, de la preuve et des algorithmes numériques.

Bibliographie

Cohen, Raphael, Guillaume Davy, Eric Feron et Pierre-Loic Garoche (2017). "Formal Verification for Embedded Implementation of Convex Optimization Algorithms". In : *IFAC-PapersOnLine* 50.1, p. 5867-5874 (cf. p. 6, 94, 164).

Davy, Guillaume, Eric Feron, Pierre-Loic Garoche et Didier Henrion (nov. 2018). "Formal verification of an interior point algorithm instanciation". In : *LPAR-22* (cf. p. 6, 139, 164).

Feron, Eric M., Raphael P. Cohen, Guillaume Davy et Pierre-Loic Garoche (jan. 2017). "Validation of Convex Optimization Algorithms and Credible Implementation for Model Predictive Control". In : *AIAA Scitex* (cf. p. 6, 94, 164).

Bibliographie globale

Abrial, J.-R. (1996). *The B-book : Assigning Programs to Meanings*. New York, NY, USA : Cambridge University Press (cf. p. 45).

Açikmese, Behçet, John M. III Carson et Lars Blackmore (2013). "Lossless Convexification of Nonconvex Control Bound and Pointing Constraints of the Soft Landing Optimal Control Problem". In : *IEEE Trans. Contr. Sys. Techn.* 21.6, p. 2104-2113 (cf. p. 76).

Appel, Andrew W (2011). "Verified software toolchain". In : *European Symposium on Programming*. Springer, p. 1-17 (cf. p. 46).

Ballabriga, Clément, Hugues Cassé, Christine Rochange et Pascal Sainrat (oct. 2010). "OTAWA : An Open Toolbox for Adaptive WCET Analysis". In : *8th IFIP WG 10.2 International Workshop on Software Technologies for Embedded and Ubiquitous Systems (SEUS)*. Sous la dir. de Sang Lyul Min ; Robert Pettit ; Peter Puschner ; Theo Ungerer. T. LNCS-6399. Software Technologies for Embedded and Ubiquitous Systems. Waidhofen/Ybbs, Austria : Springer, p. 35-46 (cf. p. 146).

Baudin, Patrick, François Bobot, Loïc Correnson et Zaynah Dargaye (p. d.). *Frama-C/WP*. https://frama-c.com/download/frama-c-wp-manual.pdf (cf. p. 33).

Bemporad, Alberto, Francesco Borrelli et Manfred Morari (déc. 2002). "Model predictive control based on linear programming - the explicit solution". In : *IEEE Transactions on Automatic Control* 47.12, p. 1974-1985 (cf. p. 4, 5, 76).

Bensalem, Saddek, Yassine Lakhnech et Hassen Saidi (1996). "Powerful techniques for the automatic generation of invariants". In : *Computer Aided Verification*. Sous la dir. de Rajeev Alur et Thomas A. Henzinger. Berlin, Heidelberg : Springer Berlin Heidelberg, p. 323-335 (cf. p. 28).

Blackmore, Lars (2016). In : *IEEE Control Systems Magazine* 36.6, p. 24-26 (cf. p. 3, 76).

Boldo, Sylvie, Catherine Lelay et Guillaume Melquiond (2015). "Coquelicot : A User-Friendly Library of Real Analysis for Coq". In : *Mathematics in Computer Science* 9.1, p. 41-62 (cf. p. 157).

Boldo, Sylvie et Claude Marché (2011). "Formal verification of numerical programs : from C annotated programs to mechanical proofs". In : *Mathematics in Computer Science* 5, p. 377-393 (cf. p. 154, 158).

Boyd, Stephen et Lieven Vandenberghe (2004). *Convex Optimization*. Cambridge University Press (cf. p. 17).

Cohen, Raphael, Guillaume Davy, Eric Feron et Pierre-Loic Garoche (2017). "Formal Verification for Embedded Implementation of Convex Optimization Algorithms". In : *IFAC-PapersOnLine* 50.1, p. 5867-5874 (cf. p. 6, 94, 164).

Dantzig, George B. (1990). "A History of Scientific Computing". In : sous la dir. de Stephen G. Nash. New York, NY, USA : ACM, p. 141-151 (cf. p. 19).

Davy, Guillaume, Eric Feron, Pierre-Loic Garoche et Didier Henrion (nov. 2018). "Formal verification of an interior point algorithm instanciation". In : *LPAR-22* (cf. p. 6, 139, 164).

Dijkstra, Edsger W. (août 1975). "Guarded Commands, Nondeterminacy and Formal Derivation of Programs". In : *Commun. ACM* 18.8, p. 453-457 (cf. p. 27).

Duprat, Stéphane et al. (jan. 2016). "Spreading Static Analysis with Frama-C in Industrial Contexts". In : *8th European Congress on Embedded Real Time Software and Systems (ERTS 2016)*. TOULOUSE, France (cf. p. 155).

Feron, Eric M., Raphael P. Cohen, Guillaume Davy et Pierre-Loic Garoche (jan. 2017). "Validation of Convex Optimization Algorithms and Credible Implementation for Model Predictive Control". In : *AIAA Scitex* (cf. p. 6, 94, 164).

Févotte, François et Bruno Lathuilière (2017). "Studying the Numerical Quality of an Industrial Computing Code : A Case Study on Code_aster". In : *Numerical Software Verification*. Sous la dir. d'Alessandro Abate et Sylvie Boldo. Cham : Springer International Publishing, p. 61-80 (cf. p. 148).

Filliâtre, Jean-Christophe et Andrei Paskevich (2013). "Why3 : Where Programs Meet Provers". In : *Proceedings of the 22Nd European Conference on Programming Languages and Systems*. ESOP'13. Rome, Italy : Springer-Verlag, p. 125-128 (cf. p. 45).

Gearhart, Jared Lee et al. (oct. 2013). "Comparison of open-source linear programming solvers". In : *Technical report, Sandia National Laboratories* (cf. p. 78).

Goodloe, Alwyn E., César Muñoz, Florent Kirchner et Loïc Correnson (2013). "Verification of Numerical Programs : From Real Numbers to Floating Point Numbers". In : *NASA Formal Methods*. Sous la dir. de Guillaume Brat, Neha Rungta et Arnaud Venet. Berlin, Heidelberg : Springer Berlin Heidelberg, p. 441-446 (cf. p. 154, 158).

Goubault, Eric et Sylvie Putot (2006). "Static Analysis of Numerical Algorithms". In : *Static Analysis*. Sous la dir. de Kwangkeun Yi. Berlin, Heidelberg : Springer Berlin Heidelberg, p. 18-34 (cf. p. 154, 158).

Herencia-Zapana, Heber et al. (2012). "PVS Linear Algebra Libraries for Verification of Control Software Algorithms in C/ACSL". In : *NASA Formal Methods - 4th International Symposium, NFM 2012, Norfolk, VA, USA, April 3-5, 2012. Proceedings*. Sous la dir. d'Alwyn Goodloe et Suzette Person. T. 7226. Lecture Notes in Computer Science. Springer, p. 147-161 (cf. p. 107).

Khachiyan, Leonid (1979). "A polynomial algorithm in linear programming". In : *Akademiia Nauk SSSR. Doklady* 244, p. 1093-1096 (cf. p. 78).

Leino, Rustan (juin 2008). "This is Boogie 2". In : Microsoft Research (cf. p. 45).

Leroy, Xavier (2009a). "A formally verified compiler back-end". In : *Journal of Automated Reasoning* 43.4, p. 363-446 (cf. p. 154).

Leroy, Xavier (2009b). "Formal verification of a realistic compiler". In : *Communications of the ACM* 52.7, p. 107-115 (cf. p. 99).

Li, Xianfeng, Yun Liang, Tulika Mitra et Abhik Roychoudhury (2007). "Chronos : A timing analyzer for embedded software". In : *Science of Computer Programming* 69.1. Special issue on Experimental Software and Toolkits, p. 56-67 (cf. p. 146).

Martin-Dorel, Érik et Pierre Roux (2017). "A reflexive tactic for polynomial positivity using numerical solvers and floating-point computations". In : *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs, CPP 2017, Paris, France, January 16-17, 2017*. Sous la dir. d'Yves Bertot et Viktor Vafeiadis. ACM, p. 90-99 (cf. p. 157).

Mattingley, Jacob et Stephen Boyd (2009). "Automatic code generation for real-time convex optimization". In : *Convex Optimization in Signal Processing and Communications*. Sous la dir. de Daniel P. Palomar et Yonina C.Editors Eldar. Cambridge University Press, p. 1-41 (cf. p. 76).

Mattingley, Jacob et Stephen Boyd (mar. 2012). "CVXGEN : a code generator for embedded convex optimization". In : *Optimization and Engineering* 13.1, p. 1-27 (cf. p. 4, 16, 147).

Mehrotra, Sanjay (1991). "On finding a vertex solution using interior point methods". In : *Linear Algebra and its Applications* 152, p. 233-253 (cf. p. 22).

MEURER, Aaron et al. (jan. 2017). "SymPy : symbolic computing in Python". In : *PeerJ Computer Science* 3, e103 (cf. p. 48, 151).

NESTEROV, Yurii (2004). *Introductory lectures on convex optimization : a basic course*. Applied optimization. Boston, Dordrecht, London : Kluwer Academic Publ. (cf. p. 79, 85-87, 92, 93, 120-122).

NESTEROV, Yurii et Arkadi NEMIROVSKI (1994). *Interior-point Polynomial Algorithms in Convex Programming*. T. 13. Studies in Applied Mathematics. Society for Industrial et Applied Mathematics (cf. p. 85, 120).

OLIPHANT, Travis E (2006). *A guide to NumPy*. T. 1 (cf. p. 48).

REIF, Wolfgang (1992). "The KIV system : Systematic construction of verified software". In : *International Conference on Automated Deduction*. Springer, p. 753-757 (cf. p. 46).

REYNOLDS, John C. (2002). "Separation Logic : A Logic for Shared Mutable Data Structures". In : *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*. LICS '02. Washington, DC, USA : IEEE Computer Society, p. 55-74 (cf. p. 33).

RIEU-HELFT, Raphaël, Claude MARCHÉ et Guillaume MELQUIOND (juil. 2017). "How to Get an Efficient yet Verified Arbitrary-Precision Integer Library". In : *9th Working Conference on Verified Software : Theories, Tools, and Experiments*. T. 10712. Lecture Notes in Computer Science. Heidelberg, Germany, p. 84-101 (cf. p. 45).

ROCKAFELLAR, Ralph Tyrell (2015). *Convex analysis*. Princeton University Press (cf. p. 22).

ROUX, Pierre (2016). "Formal Proofs of Rounding Error Bounds - With Application to an Automatic Positive Definiteness Check". In : *J. Autom. Reasoning* 57.2, p. 135-156 (cf. p. 157).

SEWELL, Thomas Arthur Leck, Magnus O. MYREEN et Gerwin KLEIN (juin 2013). "Translation Validation for a Verified OS Kernel". In : *SIGPLAN Not.* 48.6, p. 471-482 (cf. p. 99).

SPIELMAN, Daniel A et Shang-Hua TENG (2004). "Smoothed analysis of algorithms : Why the simplex algorithm usually takes polynomial time". In : *Journal of the ACM (JACM)* 51.3, p. 385-463 (cf. p. 78).

WANG, Timothy, Romain JOBREDEAUX, Heber HERENCIA-ZAPANA et al. (2013). "From Design to Implementation : an Automated, Credible Autocoding Chain for Control Systems". In : *CoRR* abs/1307.2641. arXiv : 1307.2641 (cf. p. 4, 5, 103, 135).

WANG, Timothy, Romain JOBREDEAUX, Marc PANTEL et al. (déc. 2016). "Credible autocoding of convex optimization algorithms". In : *Optimization and Engineering* 17.4, p. 781-812 (cf. p. 4, 5).

YANG, Xuejun, Yang CHEN, Eric EIDE et John REGEHR (juin 2011). "Finding and Understanding Bugs in C Compilers". In : *SIGPLAN Not.* 46.6, p. 283-294 (cf. p. 154).

YPMA, Tjalling J. (déc. 1995). "Historical Development of the Newton-Raphson Method". In : *SIAM Rev.* 37.4, p. 531-551 (cf. p. 20).

À *Supaéro*, le *Lundi premier décembre 2018*

Génération de codes et d'annotations prouvables d'algorithme de points intérieurs à destination de systèmes embarqués critiques.

Dans l'industrie, l'utilisation de l'optimisation est omniprésente. Elle consiste à calculer la meilleure solution tout en satisfaisant un certain nombre de contraintes. Cependant, ce calcul est complexe, long et pas toujours fiable. C'est pourquoi cette tâche est longtemps restée cantonnée aux étapes de conception, ce qui laissait le temps de faire les calculs puis de vérifier que la solution était correcte et si besoin refaire les calculs.

Ces dernières années, grâce à la puissance toujours grandissante des ordinateurs, l'industrie a commencé à intégrer des calculs d'optimisation au coeur des systèmes. C'est-à-dire que des calculs d'optimisation sont effectués en permanence au sein du système, parfois des dizaines de fois par seconde. Par conséquent, il est impossible de s'assurer a posteriori de la correction d'une solution ou de relancer un calcul. C'est pourquoi il est primordial de vérifier que le programme d'optimisation est parfaitement correct et exempt de bogue.

L'objectif de cette thèse a été de développer outils et méthodes pour répondre à ce besoin. Pour ce faire, nous avons utilisé la théorie de la preuve formelle qui consiste à considérer un programme comme un objet mathématique. Cet objet prend des informations en entrée et produit un résultat. On peut alors, sous certaines conditions sur les entrées, prouver que le résultat satisfait nos exigences. Notre travail a consisté à choisir un programme d'optimisation puis à prouver formellement que le résultat de ce programme est correct.

Vérification Hoare Optimisation Génération Preuve

Generation of codes and provable annotations of interior-point algorithms for critical embedded systems..

In the industry, the use of optimization is ubiquitous. Optimization consists of calculating the best solution subject to a number of constraints. However, this calculation is complex, long and not always reliable. This is why this task has long been confined to the design stages, which allowed time to do the computation and then check that the solution is correct and if necessary redo the computation.

In recent years, thanks to the ever-increasing power of computers, the industry has begun to integrate optimization computation at the heart of the systems. That is to say that optimization computation is carried out continuously within the system, sometimes dozens of times per second. Therefore, it is impossible to check a posteriori the solution or restart a calculation. That is why it is important to check that the program optimization is perfectly correct and bug-free.

The objective of this thesis was to develop tools and methods to meet this need. To do this we have used the theory of formal proof that is to consider a program as a mathematical object. This object takes input data and produces a result. We can then, under certain conditions on the inputs, prove that the result meets our requirements. Our job was to choose an optimization program and formally prove that the result of this program is correct.

SMT Verification Hoare Optimization Generation Proof SMT

Auteur :	Guillaume Davy
Directeur de thèse :	Didier Henrion
Co-Directeur :	Pierre-Loïc Garoche
Discipline :	Informatique et Télécommunications
Laboratoire :	SAE-ONERA MOIS
	Onera - Centre Toulouse
	2 Avenue Edouard Belin
	31000 Toulouse