Rapport de stage de L3

Epsilon, ou l'arithmétique d'intervalles au service de la certification de programmes numériques

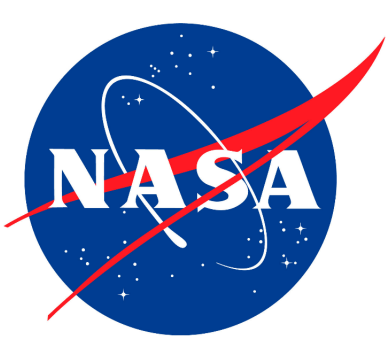
Maxime Legrand

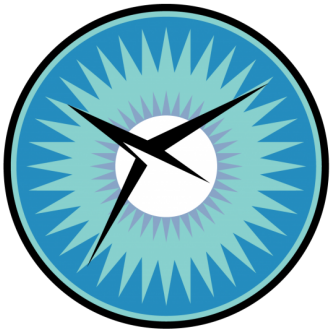
École Normale Supérieure, Paris

Encadré par :

César Muñoz
Alwyn E. Goodloe

NASA Langley Research Center, Hampton





29 août 2014

Table des matières

Introduction

Le stage concerné par ce rapport a eu lieu sur le campus du *NASA Langley Research Center* (LaRC) du 9 juin au 15 août 2014. En accord avec la politique de sécurité du gouvernement américain, les non-titulaires de la carte verte qui y travaillent ont leur bureau dans un espace séparé régi par une organisation à but non lucratif, le *National Institute of Aerospace* (NIA). C'est donc dans l'un des immeubles de cet organisme que s'est déroulé mon stage, en compagnie d'employés et d'autres étudiants venus du monde entier.

Je pouvais rentrer dans les bâtiments du LaRC grâce à un badge m'y autorisant à condition d'être à tout moment accompagné par un citoyen américain habilité. J'ai ainsi pu rendre visite à toute l'équipe du *Langley Formal Methods* (même si quelques uns sont basés au NIA) et discuter avec chacun de ses membres. J'ai également pu assister à un grand nombre de *talks*, qu'ils soient tenus au NIA, au LaRC ou encore au *Air and Space Center*, basé dans le centre-ville de Hampton.

Mes encadrants venaient presque tous les jours dans les locaux du NIA pour discuter des nouveaux objectifs et tester les avancées produites depuis la veille. Une grande force de ce stage était qu'il n'y avait pas d'objectifs immuables à atteindre, mais plutôt des recherches à faire sur différents fronts pour améliorer notre compréhension des enjeux et problématiques et deviner quelle direction pouvait être la plus prometteuse.

C'était donc un véritable travail de recherche avec beaucoup d'autonomie et d'échanges à double sens avec les membres de l'équipe, avec pour objectif principal de quitter l'équipe en ayant contribué à l'augmentation de la compréhension générale et des outils disponibles.

Mon travail principal, et celui que je vais relater dans ce rapport, consistait en la reprise et l'amélioration d'un outil développé au préalable par un autre français, Loïc Correnson, rattaché au CEA et parmi les principaux contributeurs de WP sous Frama-C, et en l'amélioration de la prise en main et de la compréhension de la théorie sous-jacente pour l'équipe l'utilisant. J'ai également présenté un exposé dont j'ai laissé les transparents, rédigés de façon aussi pédagogique et auto-suffisante que possible, accompagnés d'un rapport technique et censés permettre aux nouveaux arrivants et stagiaires d'apprivoiser la théorie et rentrer dans le code au plus vite.

Je commencerai ce rapport par un exposé du problème général et sa fuite progressive vers la nécessité d'un outil d'analyse statique efficace et d'une sûreté irréprochable sur du code de programmes numériques. Je présenterai ensuite le fonctionnement d'Epsilon tel qu'il est après le travail de Loïc Correnson et le mien, tout en tentant d'expliquer mes choix stratégiques sur la version révisée. Enfin, je traiterai la façon dont est concrètement codé et implanté l'outil Epsilon, ainsi que les grandes lignes de sa certification avec l'outil PVS, et donnerai des exemples d'utilisation en pratique.

I. Problème général

I.1 Il était une fois, deux avions

Un des principaux objectifs de l'organisation hôte est aujourd'hui de développer un grand nombre de routines, accompagnées d'outils adaptés, destinées à accompagner les pilotes dans leur participation au trafic aérien. Ainsi, ils seraient plus enclins à gérer des situations de crise en plein vol sans directive extérieure fiable (contrôle aérien, parfaite communication entre les avions, ...). De nombreuses situations sont ainsi traitées : décollage, vol et atterrissage dans des situations de communication variées.

Le problème que nous traitons ici concerne la phase de vol avec absence totale de communication avec l'extérieur. Le pilote connaît sa position, sa direction et sa vitesse, ainsi que celles des appareils environnants, mais ne peut pas établir de stratégie coordonnée avec les autres pilotes.

Les contraintes de sécurité que doit à tout prix respecter chaque pilote sont simples :

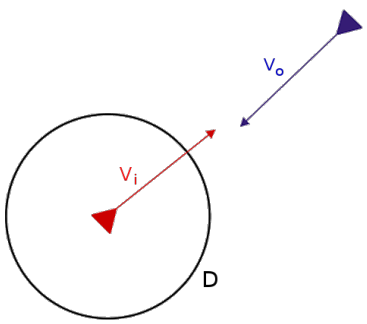
1. Ne pas pénétrer dans la zone de sécurité d'un autre avion
2. Ne pas laisser un autre avion pénétrer dans sa zone de sécurité

ou la zone de sécurité est définie par les règles de trafic aérien. Usuellement, c'est un cylindre centré en l'appareil concerné de rayon 5 nautiques et de hauteur dépendant de la situation.

Évidemment, si les zones de sécurité ont les mêmes dimensions pour les deux avions, les règles 1. et 2. sont équivalentes. De plus, si chaque pilote s'engage au préalable à respecter sa part de l'accord, il suffit de n'en garder qu'une.

Pour simplifier l'étude et parce que c'est suffisant pour introduire notre outil, nous ne nous intéresserons qu'à une vision en deux dimensions du problème, dans laquelle tous les appareils seraient sur un même plan horizontal. Nous considérerons aussi dans les exemples à venir que la distance de séparation horizontale (à savoir le diamètre de notre cylindre) est fixée à $D = 5$ nautiques.

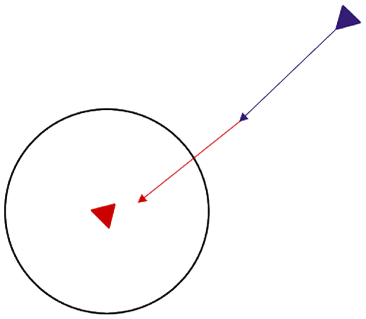
Si deux avions se rapprochent l'un de l'autre, on se trouve alors dans la situation illustrée par le schéma suivant :



Pour les notations, on utilisera à partir de maintenant les indices o et i pour désigner respectivement l'avion dans lequel est embarqué notre outil (*ownship*) et celui avec lequel on veut éviter un conflit (*traffic aircraft*). v_o et v_i, dans le schéma ci-dessus, sont donc leur vitesse au sol.

On notera d'autre part s_o et s_i leur position initiale (quand la situation est présenté à l'algorithme), $s = s_o - s_i$ la position relative de notre avion et v_o' le nouveau vecteur vitesse que l'on cherche pour éviter le conflit. Ces notations permettront de clarifier les schémas et portions de code à venir.

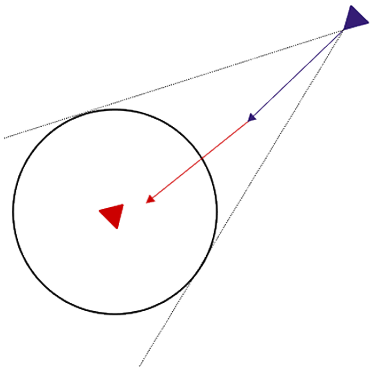
Étant donné que l'on veut ne peut influer que sur notre propre comportement, une situation "canonique" plus en phase avec nos capacités et revenant à la situation précédente serait celle où l'avion de trafic est immobile, et où l'opposé de son vecteur vitesse est ajouté au nôtre :



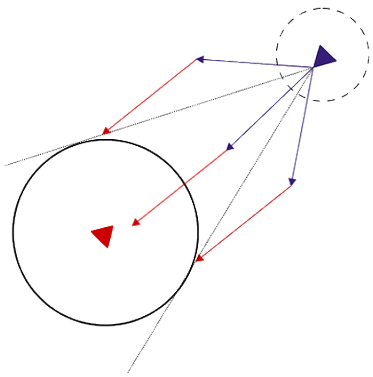
L'objectif est donc ici d'éviter un obstacle immobile en ne maîtrisant pas totalement sa vitesse (le vecteur $-v_i$ n'est pas modifiable). Évidemment, tout cela ne fonctionne que si l'on considère que la vitesse de l'avion de trafic va rester constante, mais si l'on règle la situation dans le cas alors :

— le pilote aura des informations en temps réel qu'il verra évoluer et pourra réagir en conséquence, par exemple en "prévoyant large" ;
— il sera facile, par une approximation de l'accélération de l'avion de trafic de calculer une "zone dangereuse" autour de la direction limite aisément communicable au pilote.

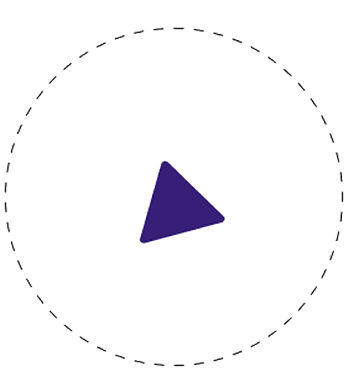
La question est alors : quelles directions suivre ? On s'aperçoit vite que dans cette situation particulière, c'est les tangentes à la zone de sécurité passant par notre appareil qui vont jouer le rôle de limites.

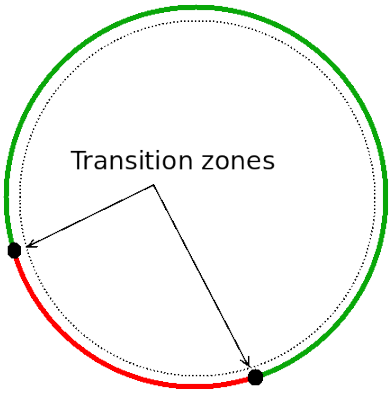


Mais si l'on veut vraiment se diriger en dehors de ces tangentes avec la contrainte $-v_i$, on se retrouve dans la situation suivante, intuitive une fois qu'elle est représentée :



Une fois ces vecteurs limites calculés, on peut communiquer au pilote les angles à emprunter (en vert) pour éviter la pénétration dans la zone de sécurité (événement que nous pourrons appeler de façon barbare “perte de séparation” pour coller au terme anglais *separation loss*). On indique également de façon binaire en rouge les angles à éviter à tout prix, car d'autres couleurs pourront venir nuancer le vert (par exemple une zone “dangereuse” orange).





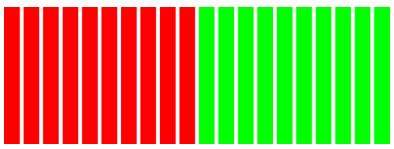
Les zones de transition indiquées sont ainsi les éléments essentiels de l'information qui sera communiquée au pilote, et forment également le cœur de notre travail.

I.2 Problèmes liés à l'évaluation flottante

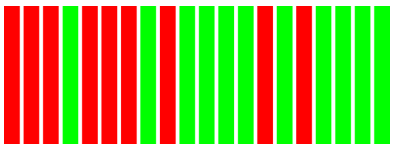
Tout comme les tangentes à la zone de sécurité, nos zones de transition sont une information précise (ce sont théoriquement des points) et essentielle dans le fonctionnement de notre outil, qui est lui-même qualifiable de *critique* dans le sens où des vies humaines reposent sur lui.

Sans reprendre l'habituel laïus sur le vol inaugural raté d'Ariane 5, ou encore les déboires d'un missile Patriot pendant la première guerre du Golfe, on sait que ce n'est pas toujours prudent de laisser ce genre de situation à une machine sans un fort contrôle de la correction de ce qu'il se passe.

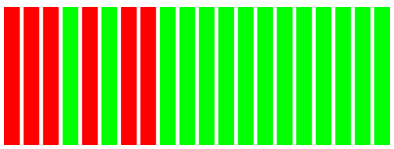
En l'occurrence, au niveau de nos zones de transition, il y a une différence notable entre la situation idéale où tous les calculs sont effectués dans le monde des réels et des fonctions mathématiques exactes, et la situation véritable où valeurs et fonctions sont formatées pour être reconnues par l'ordinateur. Ici, par exemple, l'inexactitude des fonctions et des valeurs peut amener à un brouillage couplé à des erreurs (comme un décalage), impossibles à repérer au seul vu du résultat.



Cas idéal

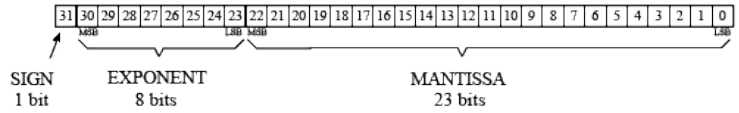


Cas effectif



Cas effectif dangereux

Ce phénomène se comprend aisément au regard du fonctionnement actuel de l'arithmétique flottante, sous les standards de la norme IEEE-754. On rappelle que les flottants sont un sous-ensemble fini des réels, dont les éléments sont représentables par un nombre fini de bits, déterminé par les caractéristiques de la machine concernée.



Example 1

0 00000111 11000000000000000000000

\+ 7 0.75

$$+1.75 \times 2^{(7-127)} = +1.316554 \times 10^{-36}$$

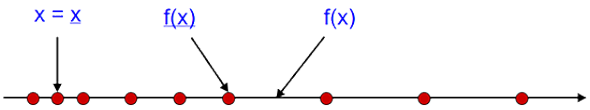
Example 2

1 10000001 01100000000000000000000

\- 129 0.375

$$-1.375 \times 2^{(129-127)} = -5.500000$$

En pratique, si l'on veut calculer numériquement l'image d'un réel x par une fonction réelle f, on approche ce réel à l'aide d'un flottant $\underline{x}$ puis on lui applique une procédure censée implanter la fonction mathématique. On obtient alors alors un flottant généralement assez proche du véritable $f(x)$, voire le plus proche dans le cas où $x = \underline{x}$ et que la procédure implant f est correctement arrondie.



Ce cas suffit en général, mais si l'on itère la fonction de nombreuses fois (ex. : suites), ou que l'on compose plusieurs fonctions complexes (ex. : commandes de vol) sans stratégie d'arrondi, cette erreur peut engendrer des résultats très éloignés de la réalité. Une suite récurrente censée converger vers 0 peut ainsi se retrouver à diverger vers $+\infty$ ou un missile opter pour une trajectoire bien éloignée de celle initialement prévue. Pour un outil embarqué critique tel que celui présenté précédemment, ne pas considérer ces phénomènes, ni agir en conséquence, est aujourd'hui hors de question. D'où la nécessité d'utiliser des méthodes formelles pour définir un sous-ensemble, d'une sûreté totale mais d'une taille convenable, des cas sans danger d'utilisation.

I.3 Présentation des outils

Frama-C est une plate-forme offrant de nombreux outils permettant l'analyse statique de code C. Elle a été choisie par la LFM pour la certification de certains de ses logiciels embarqués, notamment grâce à sa grande modularité (par l'utilisation de plug-in) et la possibilité d'apports personnels (*cf.* but du stage). Une description que je trouve élégante peut être trouvée sur le site de l'INRIA :

> *Frama-C fait collaborer de nombreuses techniques d'analyse statique, de la vérification de règles de codage à l'analyse par interprétation abstraite et la vérification de spécifications fonctionnelles grâce aux principes de l'approche déductive de la vérification de programmes. Les domaines visés sont le logiciel embarqué critique, les implémentations de bibliothèques cryptographiques, et la sécurité.*

WP est un plug-in de Frama-C destiné à trouver, comme son nom l'indique, des "plus faibles pré-conditions" (*weakest preconditions*), c'est-à-dire des conditions suffisantes les plus faibles possible avant d'entrer dans une procédure pour que soient respectées à sa sortie des post-conditions définies par l'utilisateur. On communique avec WP en annotant la procédure (un programme C) à analyser avec des spécifications écrites en langage ACSL.

Epsilon est une fonctionnalité de WP permettant d'évaluer la perte de précision due aux arrondis successifs en arithmétique flottante. Son but est, à terme, de gérer suffisamment bien les approximations flottantes à l'aide d'une solide arithmétique d'intervalles pour permettre à WP de fonctionner de façon correcte sur les programmes numériques en prenant en compte ces approximations sans avoir à s'en préoccuper personnellement. L'objectif est alors de faire fonctionner un solveur SMT sur une population de contraintes sous forme d'appartenances à des intervalles qu'il pourra sans risque considérer comme réels et idéaux.

Le module est écrit directement dans le code de WP, et est accompagné d'un autre petit programme, PrecisionLoss, se chargeant de faire le lien avec le plug-in. Ce lien s'effectue majoritaire par des redéfinitions de variables et des réponses aux appels de routine de Frama-C.

II. Fonctionnement d'Epsilon

II.1 Arithmétique d'intervalles et arithmétique flottante

L'approche consistant à s'éloigner autant que possible des points de transition afin de "faire le maximum" pour éviter la collision est bien entendu inacceptable car on veut pouvoir optimiser la trajectoire dans le domaine sécurisé. Ce serait de plus complètement ingérable si l'on était au milieu de plusieurs avions, au point que l'on ne serait même plus maître de sa trajectoire globale.

Pour bénéficier d'une confiance totale en les informations fournies par notre logiciel, nous allons donc faire appel à l'arithmétique d'intervalles, qui a pour avantage d'avoir un sûreté facilement prouvable (*cf.* III.3) et une précision adaptable selon la complexité de l'analyseur et le temps de calcul que l'on est prêt à fournir.

On rappelle en premier lieu qu'elle consiste à la base au fait de borner des valeurs dans des intervalles (quand on travaille avec des réels, on commence en général par borner une valeur a avec l'intervalle $[a, a]$) avant de leur appliquer des opérations de transition élémentaires correspondant aux manipulations effectuées sur les valeurs. Pour les opérations de bases, on a par exemple, si $a \in [x_a, y_a]$ et $b \in [x_b, y_b]$ sont des réels :

- $a + b \in [x_a + y_a, x_b + y_b]$
- $-a \in [-y_a, -x_a]$
- $a * b \in [\min(x_a x_b, x_a y_b, y_a x_b, y_a y_b), \max(x_a x_b, x_a y_b, y_a x_b, y_a y_b)]$
- $\frac{1}{a} \in [1/y_a, 1/x_a]$ si $x_a > 0$ ou $y_a < 0$ (indéfini sinon).
- $\sqrt{a} \in [\sqrt{x_a}, \sqrt{y_a}]$ si $x_a \geq 0$ (indéfini sinon)

L'arithmétique d'intervalles, comme nous le verrons dans III.2, a de plus le mérite pour notre usage d'être particulièrement adaptée à l'établissement de bornes d'erreur en arithmétique flottante. Par exemple, si a est une valeur réelle initialisée à la valeur flottante la plus proche dans un domaine de $\mathbb{R}$ où les flottants sont espacés d'une distance d_a, on peut initialiser son intervalle de sécurité à $[a - \frac{d_a}{2}, a + \frac{d_a}{2}]$. Si on fait de même pour b et qu'on admet que l'addition est implantée avec une erreur absolue maximale e, alors on aura :

$$\underline{\underline{a + b}} \in [a + b - \frac{d_a}{2} - \frac{d_b}{2} - e, a + b + \frac{d_a}{2} + \frac{d_b}{2} + e].$$

C'est ici un exemple simplifié, mais on se rend bien compte que si l'on implante correctement des opérations de base et des opérateurs de composition, on sera en capacité de lancer des analyses statiques performantes sur de grandes plages de valeurs interférant entre elles, de la même manière qu'avec l'arithmétique d'intervalles classique sur des réels.

II.2 Théorie derrière Epsilon

Le principe fondamental d'Epsilon est d'essayer de manipuler les valeurs qu'on lui donne de la façon demandée, en gardant assez d'informations sur elles pour pouvoir "reconstituer" les *vraies* valeurs, réelles, correspondant à ce qu'elles devraient être avec un fonctionnement idéal (en calcul formel plutôt que numérique). Plus précisément, l'outil Epsilon anticipe ce qui lui sera demandé – par WP ou lors de sa preuve formelle avec PVS – concernant ces vraies valeurs,

et tâche de propager ces informations de la manière la plus précise possible, afin de minimiser la marge de sécurité (pour laisser le maximum de liberté au pilote) et faciliter la preuve grâce à des informations précises et exploitables.

Formellement, pour une valeur x, nous travaillons donc avec un couple.

$$x = (x_a, x_m)$$

où x_a est la valeur dont dispose réellement le programmeur (*actual value*) et x_m est l'idéal, correspondant à ce qu'elle devrait être (*model value*).

$$x = (x_a, x_m) \in \mathbb{F} \times \mathbb{R}$$

où $\mathbb{F}$ représente l'ensemble des flottants et $\mathbb{R}$ celui des réels.

On peut considérer ce produit cartésien comme un *anneau* si l'on considère les opérations usuelles $+$, $\times$ supposées respectivement implantées en flottants par les procédures $\oplus$, $\otimes$ de la façon suivante :

$$\begin{array}{rcl} x + y & = & (x_a \oplus y_a, x_m + y_m) \\ x \times y & = & (x_a \otimes y_a, x_m \times y_m) \end{array}$$

Dans les faits, l'utilisateur n'aura accès qu'à la projection sur la première coordonnée de ces données. Pour répondre aux attentes d'un WP modifié pour calculer des pré-conditions en prenant en considération les approximations flottantes, ou encore celles de PVS pour pouvoir certifier notre code, nous engageons certaines données dans le cheminement d'une valeur x :

— un intervalle (ou, nous le verrons, une réunion de deux intervalles centrée en 0) dans lequel elle se trouve (*range*) ;
— son erreur absolue $\delta(x) = |x_a - x_m|$;
— son erreur relative $\varepsilon(x) = |\frac{x_a - x_m}{x_m}|$.

Essentiellement, la première information ne sert qu'à calculer plus facilement une sur-approximation raisonnable des deux autres, ou de trouver l'une en fonction de l'autre quand les deux ne sont ne sont pas sur un pied d'égalité face à l'implantation (tous les exemples sont dans III.2).

En exemple simple (l'outil traite une multitude de cas particuliers pour gagner en précision), on a ainsi :

$$\left\{ \begin{array}{lcl} x_m & \in & [lb, ub] \\ \delta(x) & \leq & D \end{array} \right. \quad \Longrightarrow \quad \varepsilon(x) \leq \frac{\delta(x)}{a}.$$

II.3 Représentation des valeurs et fonctionnement

La représentation des valeurs dans l'outil Epsilon était un véritable *big deal*, car elle devait être en accord avec un certain nombre de principes et influençait grandement l'écriture de l'outil. Dans la structure fonctionnelle qui m'a servi de modèle, Loïc Correnson avait introduit presque toutes les variables et fonctions nécessaires pour le raccord avec WP et le futur bon fonctionnement de celui-ci dans sa version améliorée. Pour pouvoir écrire son code en un temps très court, et le rendre lisible par son successeur (et votre humble serviteur), il avait opté pour une simplicité efficace. Les structures de données devaient être élégamment prenables en mains, et les procédures de manipulation – bien que non optimisées – courtes et compréhensibles.

Mon tout premier travail a été de parcourir ce code pour en comprendre les moindres détails. Quelques jours après, je faisais ma première présentation à l'équipe pour le leur expliquer (et leur rappeler les principes de son fonctionnement et de sa sortie) et définir avec eux les premiers enjeux de ma contribution (situation un peu cocasse, mais finalement très efficace).

Ma première et principale contribution a alors été de réécrire entièrement le programme en implantant une nouvelle représentation des valeurs, accompagnée de toutes les procédures de manipulation nécessaires, cette fois-ci optimisées. Le tout devait être modulaire et paramétrable afin de configurer aisément la précision, en sacrifiant si nécessaire des capacités de calcul.

Pour l'expliquer, commençons par présenter les principaux points qui influençaient le choix du type de représentation des valeurs :

1. Être sûre.
2. Être relativement simple (au sens de facilement prouvable) à manipuler.
3. Être proche de l'écriture canonique des flottants.
4. Être aussi précis que possible.

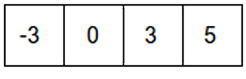
Le point 1. est une condition *sine qua non*, tandis que les autres dépendent surtout de l'utilisation précise que l'on veut en faire. Ainsi, le point 2. dépend du temps que l'on veut passer à certifier notre outil tandis que le 3. influe directement sur le 2. et sur la compréhensibilité du code par les instances humaines de vérification. Enfin, le point 4. correspond à notre objectif de précision sûre.

Dans sa première implantation Epsilon utilisait comme représentation des valeurs les puissances de deux. Que ce soit une borne de notre *range*, ou un majorant de δ ou ε, chaque valeur était ainsi une simple puissance de deux. Évidemment, cette représentation respecte parfaitement les points 1., 2. et 3., mais pas vraiment le dernier. En cela, elle respectait à la lettre les conditions à remplir lors de sa création fin 2013.

En fait, elle est quand même moins grossière qu'elle n'y paraît pour des procédures courtes utilisant des délibérément petites données (comme des différences de grandes valeurs proches) du même ordre de grandeur, et ceci grâce à la progression géométrique des $2^n, n \in \mathbb{Z}$. Cependant, imaginons que l'on additionne une très petite valeur v à une grande valeur $V \in [2^m, 2^M]$, alors $v + V$ deviendrait majorée par 2^{M+1}, soit le double d'une valeur potentiellement très grosse, pour un changement ridicule. Il suffit alors d'imaginer que l'on ait besoin d'itérer ce genre de calcul pour que ça devienne vite insupportable.

C'est pourtant le cas pour un calcul de trajectoire instant par instant en fonction de l'accélération et des positions précédentes possibles dans une situation où l'on serait coupé de l'information GPS. Or, c'est une des situations que cherche à traiter l'équipe : pouvoir s'assurer d'arriver relativement près de la destination intermédiaire suivante en évaluant la trajectoire par un ciel couvert et sans information de repérage extérieure.

À la place, l'équipe a accepté une idée que j'avais eue un peu avant, et qui avait été très bien accueillie par Loïc Correnson, de considérer les valeurs comme des sommes finies de puissances de deux, représentées par la liste des exposants dont la taille serait paramétrable. En effet, cette représentation respecte aussi bien que la précédente les points 1. et 3., et n'ajoute qu'une difficulté passagère au travers du 2. En revanche, le gain sur le point 4. est énorme dès l'ajout du second exposant (l'exemple présenté ci-dessus ne fonctionnerait soudainement que beaucoup moins bien) et n'a de limite que la capacité de calcul qu'on est prêt à investir, sans être inutilement gourmande. C'est dans les faits une notation simplifiée des flottants, avec une écriture en base deux, mais modulaire par l'utilisation de nombres décimaux. Elle est en outre très bien adaptée aux opérations effectuées dans le module (qui sont loin d'être toutes présentées ici). Ainsi, on a, par exemple :



$= 2^{-3} + 2^0 + 2^3 + 2^5 = 0.125 + 1 + 8 + 32 = 41.125$

III. Implantation

Le principal objectif de cette partie est de donner une vision claire et concrète de ce qui se passe à haut niveau dans le module Epsilon du plug-in WP. Précédée des points théoriques des passages précédents, la lecture de cette partie devrait suffire au lecteur pour se persuader qu'il serait lui-même capable d'implanter, prendre en mains et améliorer le module.

Des différentes implantations procédurales, nous ne traiterons que les fonctions qui ont l'avantage d'être très représentatives même par une approche haut niveau. La technicité des autres procédures (en particulier les nombreux cas de compositions ou d'améliorations des erreurs) demandent une plus ample plongée dans le code que ne le permet ce court rapport.

III.1 Implantation des valeurs

Comme indiqué précédemment, la forme générique des valeurs que l'on manipule est la suivante :

$$\left\{\begin{array}{ll} \text{signe :} & \texttt{sign} \text{ (+, - ou Any)} \\ & \\ \text{erreur relative } (\varepsilon) : & \texttt{rel} \\ \text{erreur absolue } (\delta) : & \texttt{abs} \\ & \\ \text{borne inférieure :} & \texttt{lb} \\ \text{borne supérieure :} & \texttt{ub} \end{array}\right\}$$

Comme on l'avait laissé entendre précédemment, les plages de valeurs de sont pas exactement des intervalles dans tous les cas. L'idée était de ne pas perdre trop d'information sur une valeur x étant donné $|x|$ ou x^2. Ainsi, si $|x| \in (+)[lb, ub]$ on pourra dire que $x \in (\text{Any})[lb, ub]$ qui est bien plus précis que $[-ub, ub]$ en plus d'avoir le grand avantage de ne pas inclure 0 dans le cas général, ce qui est décisif pour des opérations telles que la multiplication ou la division.

Avec la représentation des valeurs adoptée plus haut, on introduit naturellement un système de types donnant la signature suivante :

$$\left\{\begin{array}{ll} \text{signe :} & \texttt{sign (type : sign)} \\ & \\ \text{erreur relative :} & \texttt{rel (type : int list)} \\ \text{erreur absolue :} & \texttt{abs (type : int list)} \\ & \\ \text{borne inférieure :} & \texttt{lb (type : int list)} \\ \text{borne supérieure :} & \texttt{ub (type : int list)} \end{array}\right\}$$

III.2 Implantation des fonctions

On imagine désormais avoir deux valeurs a et b sur lesquelles on veut appliquer nos fonctions de base ($+,\ -,\ \times,\ /,\ \sqrt{\cdot}$). On adopte les notations suivantes :

$$a = \left\{ \begin{array}{ll} \text{signe :} & \texttt{sign}(a) \\ \\ \text{erreur relative :} & \texttt{rel}(a) \\ \text{erreur absolue :} & \texttt{abs}(a) \\ \\ \text{borne inférieure :} & \texttt{lb}(a) \\ \text{borne supérieure :} & \texttt{ub}(a) \end{array} \right\} \quad b = \left\{ \begin{array}{ll} \text{signe :} & \texttt{sign}(b) \\ \\ \text{erreur relative :} & \texttt{rel}(b) \\ \text{erreur absolue :} & \texttt{abs}(b) \\ \\ \text{borne inférieure :} & \texttt{lb}(b) \\ \text{borne supérieure :} & \texttt{ub}(b) \end{array} \right\}$$

Ceci étant fait, il ne nous reste plus qu'à appliquer les principe de la norme IEEE-754 sur l'arithmétique flottante dans les machines modernes. On ne s'intéressera ici qu'aux machines 32 et 64 bits.

On rappelle que si l'on a la plage de valeurs et une des deux erreurs, alors on peut obtenir une sur-approximation de l'autre par division par lb ou multiplication par ub. On n'en calculera donc qu'une sur deux pour chaque fonction et on appellera *rel* et *abs* les fonctions pour passer de l'une à l'autre. Le module inclut également une fonctionnalité d'amélioration des erreurs dans les cas où on dispose d'informations supplémentaires.

D'autre part, l'espacement des flottants est tel que l'erreur relative pour des fonctions correctement arrondies (comme c'est le cas pour toutes celles présentées ici) ne dépasse pas 2^{-f+1} où f est le nombre de bits consacrés à l'exposant. On note *lop* (pour *loss of precision*) cette erreur et on a alors :

— $lop = 0$ pour le modèle Réel
— $lop = 2^{-23}$ pour le modèle F32
— $lop = 2^{-57}$ pour le modèle F64

Une fois encore, on peut obtenir l'équivalent absolu de cette valeur grâce aux bornes des valeurs considérées. On notera alors $\text{LOP} = abs(lop)$.

Une dernière variable qui interviendra est *sd* (pour *smallest decimal*) qui représente la plus petite valeur de digit que l'on s'autorise dans nos opérations sur les listes. C'est en particulier sur elle que repose l'équilibre de notre outil entre précision et performance.

Addition

Opération sur les listes L'algorithme est très naturel, étant donné que nous gardons nos listes ordonnées de façon croissante.

On commence par implanter l'addition d'un digit d à une liste l. Si la liste est vite l'addition renvoie $[d]$. Sinon $l = h :: t$ et on a trois cas :

— si $d < h$, alors on renvoie $d :: l$
— si $d = h$, alors appelle récursivement la fonction avec $d + 1$ et t
— si $d > h$, alors on appelle récursivement la fonction avec d et t pour obtenir l' et on renvoie $h :: l'$.

L'addition de deux listes $\oplus$ revient ensuite à additionner un à un tous les digits de l'un à l'autre.

On ne détaillera pas les implantations de toutes les fonctions (car elles sont souvent naturelles et dans le code) mais celle-ci avait le mérite d'être très simple et très rapide, et de concrétiser la façon dont on manipule nos valeurs.

Erreur absolue Si $a_a \in [a_m - \delta_a, a_m + \delta_a]$ et $b_a \in [b_m - \delta_b, b_m + \delta_b]$, alors $a_a + b_a \in [a_m + b_m - (\delta_a + \delta_b), a_m + b_m + \delta_a + \delta_b]$. D'où $\delta_{a+b} = \delta_a + \delta_b$ dans l'idéal, et donc $\delta_{a+b} = \delta_a + \delta_b + \text{LOP}$ en pratique.

Signe et bornes Pour calculer le signe, on s'intéresse à plusieurs cas particuliers. Si a et b ont le même signe s, alors $a+b$ sera de signe s et il suffit d'additionner leurs bornes grâce à notre implantation sur les listes. S'ils ne sont pas de même signe et que la borne inférieure de celui de signe s est plus grande que la supérieure de l'autre, alors $a+b$ sera de signe s. Sinon, leur signe est Any. Dans les deux cas, on soustrait intelligemment leurs bornes de façon naturelle. On obtient ainsi $\texttt{sign}(a+b)$, $\texttt{lb}(a+b)$ et $\texttt{ub}(a+b)$.

Résultat Après addition de a et b on a alors :

$$a+b=\left\{\begin{array}{ll} \text{signe :} & \texttt{sign}(a+b) \\ & \\ \text{erreur relative :} & rel(\texttt{abs}(a)) \\ \text{erreur absolue :} & \texttt{abs}(a)\oplus\texttt{abs}(b)\oplus \text{LOP} \\ & \\ \text{borne inférieure :} & \texttt{lb}(a+b) \\ \text{borne supérieure :} & \texttt{ub}(a+b) \end{array}\right\}$$

Soustraction

Principe $a-b=a+(-b)$. Il nous suffit donc pour avoir la soustraction d'implanter l'opposé, ce qui est immédiat.

Résultat $-a = a$ `with` signe = $\texttt{-sign}(a)$.

Multiplication

Principe La multiplication sur les listes $\otimes$ n'est pas bien plus difficile à implanter.

On utilise cette fois l'erreur relative bien plus en phase avec cette opération. On fait deux suppositions dont les négations sont prises en mains de façon naturelle par Epsilon :

— a et b sont positifs (il suffit de changer le signe dans l'enregistrement final) ;

— $\varepsilon_a < 1$ et $\varepsilon_b < 1$ (sinon la valeur, peut être négative ou nulle, ce qui lève une exception de nullité pour la multiplication, d'infini pour la division et qui ramène à 0 avec contrôle du signe pour la racine carrée).

Alors on a $a(1\pm\varepsilon_a)b(1\pm\varepsilon) \in [ab(1-(\varepsilon_a+\varepsilon_b-\varepsilon_a\varepsilon_b)), ab(1+(\varepsilon_a+\varepsilon_b+\varepsilon_a\varepsilon_b))]$. D'où $\varepsilon_{a\times b}=\varepsilon_a+\varepsilon_b+\varepsilon_a\varepsilon_b$ dans l'idéal et $\varepsilon_{a\times b}=\varepsilon_a+\varepsilon_b+\varepsilon_a\varepsilon_b+lop$ dans les faits.

Le signe est quand à lui le produit des signes et les bornes le produit des bornes.

Résultat Après multiplication de a et b on obtient alors :

$$a\times b=\left\{\begin{array}{ll} \text{signe :} & \texttt{sign}(a)\times\texttt{sign}(b) \\ & \\ \text{erreur relative :} & \texttt{rel}(a)\oplus\texttt{rel}(b)\oplus\texttt{rel}(a)\otimes\texttt{rel}(b) \\ \text{erreur absolue :} & abs(\texttt{rel}(a\times b)) \\ & \\ \text{borne inférieure :} & \texttt{lb}(a)\times\texttt{lb}(a) \\ \text{borne supérieure :} & \texttt{ub}(a)\times\texttt{ub}(b) \end{array}\right\}$$

Division

Principe On commence par implanter la division binaire $\oslash$ sur les listes. J'ai en fait écrit deux fonctions $\lfloor\oslash\rfloor$ et $\lceil\oslash\rceil$ prenant en argument supplémentaire sd pour calculer respectivement les arrondis inférieur et supérieur les plus proches avec sd pour plus petite décimale (leur code est classique).

Le signe est le produit des signes, les bornes le quotient des bornes et $\frac{1\pm\varepsilon_a}{1\pm\varepsilon_b} \in [\frac{1-\varepsilon_a}{1+\varepsilon_b}, \frac{1+\varepsilon_a}{1-\varepsilon_b}] = [1-\frac{\varepsilon_a+\varepsilon_b}{1+\varepsilon_b}, 1+\frac{\varepsilon_a+\varepsilon_b}{1-\varepsilon_b}] \subset [1-\frac{\varepsilon_a+\varepsilon_b}{1-\varepsilon_b}, 1+\frac{\varepsilon_a+\varepsilon_b}{1-\varepsilon_b}]$.

D'où $\varepsilon_{\frac{a}{b}} = \frac{\varepsilon_a+\varepsilon_b}{1-\varepsilon_b} + lop$.

Résultat Après division de a et b on obtient alors :

$$\frac{a}{b} = \left\{ \begin{array}{ll} \text{signe :} & \texttt{sign}(a) \times \texttt{sign}(b) \\ \\ \text{erreur relative :} & (\texttt{rel}(a) \oplus \texttt{rel}(b)) \lceil \oslash \rceil (1 \ominus \texttt{rel}(b)) \\ \text{erreur absolue :} & abs(\texttt{rel}(\frac{a}{b})) \\ \\ \text{borne inférieure :} & \texttt{lb}(a) \lfloor \oslash \rfloor \texttt{ub}(a) \\ \text{borne supérieure :} & \texttt{ub}(a) \lceil \oslash \rceil \texttt{lb}(b) \end{array} \right\}$$

Racine carrée

Opération sur les listes Je n'ai pas la place de donner ici l'algorithme de racine carrée sur les listes, mais ce n'est pas bien grave car il est très classique et trouvable dans la littérature. Il s'agit en fait de l'algorithme de calcul bit à bit, qui n'est pas optimal mais présente de nombreux avantages parmi lesquels :

— (relativement) facile à implanter ;
— particulièrement adapté à l'écriture binaire ;
— chaque bit calculé est définitif, ce qui renforce le premier point et nous permet de créer une bonne synergie avec le reste du programme.

En bref, c'est un algorithme itératif utilisant des suites et tirant bénéfice du fait que

$$\begin{array}{rl} (a_1 + a_2 + ... + a_k)^2 = & a_1^2 \\ & + 2a_1a_2 + a_2^2 \\ & + 2(a_1 + a_2)a_3 + a_3^2 \\ & + ... \\ & + 2(\sum_{i=1}^{k-1} a_i)a_k + a_k^2 \end{array}$$

De même que pour la division, on en produit deux versions $\lfloor\sqrt{}\rfloor$ et $\lceil\sqrt{}\rceil$. La racine carrée ne prend et ne renvoie que des valeurs positives. Et les bornes de la racine carrée sont la racine carrée des bornes.

On cherche enfin à obtenir son erreur relative.

$$\begin{array}{rll} \sqrt{[1-\varepsilon_a, 1+\varepsilon_a]} & = 1 - (1 - \sqrt{[1-\varepsilon_a, 1+\varepsilon_a]}) & \leq 1 - (1 - \sqrt{1-\varepsilon_a}) \\ & = 1 + (\sqrt{[1-\varepsilon_a, 1+\varepsilon_a]} - 1) & \leq 1 + (\sqrt{1+\varepsilon_a} - 1) \\ & & \leq 1 + (1 - \sqrt{1-\varepsilon_a}) \end{array} .$$

D'où $\sqrt{a}_a \in \sqrt{a}_m \times [1-\varepsilon_{\sqrt{a}}, 1+\varepsilon_{\sqrt{a}}]$ avec

$$\varepsilon_{\sqrt{a}} = 1 - \sqrt{1-\varepsilon_a}$$

Résultat Après application de la racine on obtient alors :

$$\sqrt{a} = \left\{ \begin{array}{ll} \text{signe :} & + \\ \\ \text{erreur relative :} & 1 \ominus \lfloor\sqrt{}\rfloor(1 \ominus \texttt{rel}(a)) \\ \text{erreur absolue :} & abs(\texttt{rel}(\frac{a}{b})) \\ \\ \text{borne inférieure :} & \lfloor\sqrt{}\rfloor \texttt{lb}(a) \\ \text{borne supérieure :} & \lceil\sqrt{}\rceil \texttt{ub}(b) \end{array} \right\}$$

III.3 Vérification

La vérification du module est, à l'heure où est écrit cet article, en cours de procédure. Pour l'instant, la multiplication a été formellement vérifiée, et les autres fonctions sont en bonne passe de l'être à leur tour sur le modèle de cette dernière. Quant aux autres procédures, elles sont peu à peu définies dans l'environnement de PVS et servent concrètement à l'optimisation des fonctions. Parvenir à certifier toutes les fonctions mène donc à certifier la correction de ces procédures (mais pas leur efficacité dont il faut se persuader soi-même).

La vérification est une étape importante, notamment pour se conformer aux normes de sécurité de l'organisme hôte. Contrairement à ce que certains peuvent penser (je les vois sourire d'ici), il n'y a aucun paradoxe du "jusqu'où doit-on aller ?" dans le sens où :

— PVS est un outil éprouvé ayant déjà la confiance des instances de vérification

— Epsilon est quelque peu différent dans le sens où il manipule formellement flottants et réels, et qu'il sert un plug-in (WP) ne faisant pas que de la vérification, mais aussi du calcul de pré-conditions. Ne se certifiant pas lui-même, il est important d'être sûr que ces préconditions mèneront à la situation de sécurité demandée.

Ce travail a été dans la pratique fait par un expert de PVS nouveau venu dans la LFM, Mariano Moscato, avec mon aide pour la compréhension du code et l'élaboration de stratégies de vérification.

III.4 Utilisation et sortie

L'utilisation ne peut bien sûr se faire qu'avec la version expérimentale de WP dans lequel nous avons injecté notre module. Il suffit alors de faire appel à lui à l'aide de la commande

```
frama-c -pp-annot -wp -wp-lop [+options] myfile.c
```

Le fichier myfile.c doit ainsi contenir le code de la fonction que l'on veut vérifier, annoté selon les spécifications qu'on veut lui imposer. Les annotations sont faites en langage ACSL (il existe des centaines de pages de documentation sur son utilisation) avec en particulier les fonctions *\actual* et *\model* pour parler de la valeur effective et de la valeur idéale. Généralement, la procédure d'implantation est la fonction elle-même, et elle est comparée à la fonction idéale que l'on définit en ACSL sous des balises *axiomatic* (je n'ai pas trouvé mieux).

J'ai ainsi pu écrire en C, annoter en ACSL et fait fonctionner sous la version modifiée de Frama-C plusieurs fonctions utiles pour le problème présenté en début de rapport parmi lesquelles :

— `tangent_line` pour calculer les tangentes à la zone de sûreté ;

— `trk_line` pour calculer les directions à prendre si l'on garde la même vitesse pour que le vecteur somme suive ces tangentes ;

— `trk_circle` donnant les directions calculées par `trk_line` en prenant en main des cas critiques (perte de séparation) ;

— des débuts sur d'autres fonctions : `trk_circle`, `gs_line`, `gs_circle`, `gs_bands`...

Les résultats sont pour l'instant assez prometteurs. En fait, pour l'instant, la sortie se contente de l'affichage des intervalles de présence et des erreurs des différentes variables utilisées dans le programme, résultant d'un point fixe sur leur optimisation. Désormais, WP peut donc manipuler les variables comme des réels, et laisser à Epsilon la charge des préconditions liées aux flottants.

La prochaine étape, normalement finale dans le développement, sera de modifier WP pour prendre en compte les remarques d'Epsilon pour ensuite effectuer du calcul de pré-conditions en donnant à un solveur SMT des informations qu'il pourra utiliser en les considérant comme réelles et idéales.

Conclusion

Ce stage fût je pense très positif dans le sens où les encadrants ont semblé agréablement surpris par le nombre et la diversité des avancées produites, et qu'il était difficile de faire plus car le reste était destiné à être fait conjointement avec leur contact au CEA.

J'ai également établi un grand nombre de contacts et échangé avec beaucoup de membres de l'équipe du *Langley Formal Methods*, notamment au travers de discussions très intéressantes et de *talks* présentés par chacun (le mien s'est déroulé après une grosse moitié du stage). Tout cela en pays étranger dès la L3, ce que j'ai personnellement trouvé très riche et formateur.

Il m'a été proposé d'y retourner à Noël et à l'Été prochain, mais je pense que ce sera incompatible avec le programme que je me réserve pour l'année à venir. J'ai ainsi gardé en tête en écrivant ce rapport qu'il se devait d'être particulièrement utile à mon successeur pour rentrer au plus vite dans le vif du sujet et se sentir à l'aise avec la sémantique résidant derrière le code. Le grand nombre de tests – et les commentaires les accompagnant – laissés sur place devraient alors lui permettre de rapidement prendre en mains les outils utilisés par l'équipe.

J'aimerais pour finir remercier tous ceux qui ont permis un si bon déroulement de ce stage, à commencer par Loïc que j'ai pu rencontrer avant de partir en guise de préparation et avec qui j'ai pu continuer à discuter par mail des améliorations possibles. Puis Alwyn et César pour leur encadrement très humain, Natasha pour son soutien et sa conversation très riche, Andrew pour son aide en toutes circonstances et Heber pour avoir été mon *senseï* en ACSL.