Projet d'initiation - Rapport

Affapy : Bibliothèque d'arithmétique affine en Python

Quentin Deschamps
Florian Guily

Tristan Michel
Ruxue Zeng

Supervisé par M.HILAIRE Thibault

Mai 2020

Table des matières

1 Présentation

1.1 Introduction

En mathématiques et en informatique, l'arithmétique d'intervalles consiste à manipuler des intervalles à la place de nombres réels. Ces intervalles peuvent ensuite être ajoutés, soustraits, multipliés ou divisés. Une application de cette arithmétique est de connaître l'intervalle d'une expression. Cependant, cette arithmétique n'est pas toujours précise. En effet, l'intervalle obtenu est correct mais n'est souvent pas le plus petit possible. Il peut même être beaucoup trop large au point d'être quelques fois inutile.

C'est pourquoi, l'arithmétique affine a été introduite. Celle-ci produit des intervalles beaucoup plus minces lors de longues chaînes de calculs. En arithmétique affine, les quantités sont définies comme des combinaisons affines de la forme :

$$x_0 + \sum_{i=1}^{n} x_i \epsilon_i$$

où x_0 est le centre de cette quantité, les $\{x_i\}_{1 \leq i \leq n}$ sont les déviations et les $\{\epsilon_i\}_{1 \leq i \leq n}$ des symboles de bruit compris dans l'intervalle $[-1, 1]$.

Dans ce projet, nous avons implémenté ces deux arithmétiques en Python sous la forme d'une librairie *open-source*, nommée **Affapy**. Celle-ci est multi-précision grâce à l'utilisation de la librairie *mpmath*[1]. Voici le lien du dépôt :

https://gitlab.lip6.fr/hilaire/affapy

Ce rapport décrit d'abord les objectifs de notre projet, ainsi que notre organisation. Vous trouvez alors le plan précis de cette réalisation. Ensuite, l'implémentation de ces deux arithmétiques est expliquée. Enfin, nous exposons un bilan final du projet.

Les explications concernant l'implémentation sont complétées par la documentation de notre librairie (en anglais) que vous trouverez sur ce lien :

https://affapy.readthedocs.io/en/latest/

1.2 Objectifs

Les objectifs de ce projet sont les suivants :

- Écrire une librairie Python permettant de manipuler des quantités affines
- Être capable d'effectuer les opérations suivantes dessus :
  - Opérations basiques : addition, soustraction, multiplication, division
  - Puissances et racine carrée
  - Fonctions exponentielle et logarithmique

1. http://mpmath.org/

  - Fonctions trigonométriques (sin, cos, tan, cotan)
  - Autres fonctions : valeur absolue, fonctions hyperboliques
- Gérer les erreurs d'arrondis de calculs et pouvoir gérer la précision
- Rendre la librairie *open-source*, documentée et utilisable pour tous

1.3 Organisation

Afin de travailler de façon efficace, nous avons mis en place un dépôt sur *GitLab*. Nous pouvons alors voir facilement les modifications effectuées et l'historique du projet.

Pour écrire notre code, nous utilisons des IDE [2], qui sont *PyCharm* et *Visual Studio Code*. Cela nous permet de coder plus vite et plus facilement qu'avec un simple éditeur de texte. Ces IDE nous permettent aussi de mieux gérer le dépôt *GitLab* en effectuant des commit rapidement et en voyant plus facilement l'historique des versions et modifications.

Pour communiquer et travailler à distance, nous utilisons *Discord* afin d'effectuer des appels de groupe. Cela nous permet aussi de contacter notre encadrant de projet.

1.4 Plan du projet

Tout d'abord, nous avons décidé du nom de notre librairie. Celle-ci s'appelle *Affapy*. Le dépôt est composé de plusieurs répertoires :

- **affapy** : ce répertoire contient les fichiers de code source de la librairie. Il comprend :
  - *ia.py* qui contient la classe **Interval**, permettant de gérer l'arithmétique d'intervalles
  - *aa.py* qui contient la classe **Affine**, permettant de gérer l'arithmétique affine
  - *error.py* qui contient la classe **affapyError**, permettant la gestion des erreurs sur les deux fichiers précédents
  - *precision.py* qui contient la classe **precision**, permettant de gérer la précision des calculs
- **doc** : ce répertoire contient les fichiers nécessaires pour générer la documentation *Sphinx* du projet. Il contient aussi un makefile pour générer le diagramme UML.
- **test** : ce répertoire contient les fichiers de tests pour les classes Interval et Affine. Ces tests permettent de valider les fonctions écrites. Il y a un fichier par classe : *test_ interval.py* et *test_ affine.py*.

2. Integrated development environment

- **examples** : ce répertoire contient des fichiers d'exemples pour montrer l'utilisation de la librairie et comparer les deux arithmétiques.
- **setup.py** : ce fichier permet d'installer la librairie. Il permet aussi de publier la librairie sur PyPi [3] pour rendre celle-ci installable par la commande :

pip3 install affapy

2 Réalisation

Cette partie explique l'implémentation de ces deux arithmétiques. Vous pouvez vous reporter à la documentation [4] de la librairie (en anglais) pour plus de détails. Celle-ci est réalisée à partir de l'outil *Sphinx* [5] et est hébergée par le site *Read the Docs* [6].

2.1 Arithmétique d'intervalles

L'arithmétique d'intervalles (IA pour *Interval Arithmetic*) permet de contrôler la précision des calculs numériques. Le principe consiste à remplacer un nombre réel par un intervalle le contenant, et à effectuer les calculs avec des intervalles :

$$x = [x_-, x_+]$$

Par exemple, l'addition et la soustraction sont définies comme cela :

$$x + y = [x_- + y_-, x_+ + y_+]$$
$$x - y = [x_- - y_-, x_+ - y_+]$$

À chaque calcul, l'intervalle obtenu contient toujours la solution exacte.

Cette arithmétique a deux avantages majeurs :

1. Elle peut tenir compte des incertitudes sur les données, dans le cas de valeurs expérimentales par exemple.
2. Elle permet de manipuler sur ordinateur, en utilisant le calcul flottant, des quantités qui ne sont pas exactement représentables. L'exemple le plus connu est le nombre π. Ce nombre peut être représenté par l'intervalle $[3.14, 3.15]$.

Pour implémenter l'IA, nous avons créé la classe **Interval**. Pour initialiser une instance de cette classe, il suffit de donner deux valeurs : les bornes inférieure et supérieure de l'intervalle. Un élément de type Interval a donc deux champs : **inf** et **sup**. De plus, nous avons fait en sorte que, lors de l'initialisation

3. https://pypi.org/
4. https://affapy.readthedocs.io/en/latest/
5. https://www.sphinx-doc.org/en/master/
6. https://readthedocs.org/

d'un intervalle, le champ inf correspond au minimum des deux valeurs données et le champ sup le maximum.

Un intervalle est caractérisé par trois critères :

— La largeur (**width**) : $x_+ - x_-$

— Le milieu (**mid**) : $\frac{x_- + x_+}{2}$

— Le rayon (**radius**) : $\frac{x_+ - x_-}{2}$

Cette classe contient de nombreuses méthodes pour surcharger les opérateurs entre deux instances de la classe Interval, ou entre un intervalle et un flottant par exemple :

— Addition ($+$), soustraction ($-$) : **add**, **sub**

— Opposé ($-$ unaire) : **neg**

— Multiplication ($*$), division ($/$) : **mul**, **truediv**

— Puissance ($**$) : **pow**

D'autres méthodes correspondent à des fonctions mathématiques appliquées à un intervalle :

— Valeur absolue : **abs**

— Racine carrée : **sqrt**

— Exponentielle : **exp**

— Logarithme : **log**

— Fonctions trigonométriques : **cos**, **sin**, **tan**, **cotan**

— Fonctions hyperboliques : **cosh**, **sinh**, **tanh**

Elle contient également des méthodes pour comparer deux intervalles :

— Égalité ($==$), non-égalité ($!=$) : **eq**, **ne**

— Supérieur ($>$), supérieur ou égal ($>=$) : **gt**, **ge**

— Inférieur ($<$), inférieur ou égal ($<=$) : **lt**, **le**

Vous trouverez les explications de ces différentes méthodes dans la documentation à la section *Interval Arithmetic (IA)*.

2.2 Arithmétique affine

L'arithmétique affine (AA) est un moyen différent de l'arithmétique d'intervalles pour représenter une quantité. Les objectifs de cette arithmétique sont les mêmes que pour l'IA. Elle a été développée afin de surmonter les problèmes d'explosions de l'IA dans certains cas et pour améliorer la précision du résultat. Cette méthode s'appuie sur un autre moyen de représentation d'une quantité x réelle : une forme affine, nommée $\hat{x}$.

$$\hat{x} = x_0 + x_1\epsilon_1 + x_2\epsilon_2 + ... + x_n\epsilon_n = x_0 + \sum_{i=1}^{n} x_i\epsilon_i$$

Les coefficients x_i, appelés *écarts partiels*, sont des nombres flottants finis et connus. Les coefficients ϵ_i, appelés *symboles de bruit*, sont des nombres réels compris dans l'intervalle $[-1, 1]$ dont on ne connaît pas la valeur exacte. Chacun de ces symboles représente une source d'incertitude. On appelle x_0 la *valeur centrale* de la forme affine $\hat{x}$.

Une forme affine est donc composée d'un centre et de "compartiments" construits à partir d'un produit entre un écart partiel et un symbole de bruit. Ce mode de représentation permet un meilleur contrôle sur l'estimation de la quantité calculée en la décomposant en plusieurs compartiments, facilitant ainsi sa manipulation et augmentant donc sa précision.

On observe que les ϵ_i, supposés tous différents, sont tous compris dans le même intervalle $[-1, 1]$. De ce fait, il est possible de les assimiler à des identifiants, ou clés, servant à différentier les x_i entre eux. C'est donc naturellement que nous nous sommes dirigés vers une implémentation de ces derniers sous forme de **dictionnaire**. En Python, un dictionnaire est un ensemble d'éléments non ordonnés, caractérisés chacun par une clé et une valeur. Ce dictionnaire constitue un premier attribut de la classe **Affine**. Ses clés correspondent aux ϵ_i et ses valeurs sont les x_i (connus).

Le deuxième attribut correspond au centre x_0 de la forme affine. Nous avons choisi de le différencier du dictionnaire par souci de simplicité pour l'écriture des méthodes. Enfin, le dernier attribut correspond à l'objet Interval associé à l'objet Affine. Le passage d'une forme affine vers un intervalle est simple :

$$x = [x_0 - r, x_0 + r], \text{ avec } r = \sum_{i=1}^{n} |x_i|$$

Où r correspond au rayon de la forme affine. On peut aussi passer d'un intervalle vers une forme affine. Notons l'intervalle $[x_-, x_+]$. On a la relation suivante :

$$\hat{x} = x_0 + x_k\epsilon_k, \text{ avec } x_0 = \frac{x_- + x_+}{2}, \text{ et } x_k = \frac{x_- - x_+}{2}$$

Grâce à ces relations, l'utilisateur a donc deux possibilités pour créer une instance de la classe Affine :

— Donner le centre x_0 et le dictionnaire des symboles de bruits x_i.

— Donner l'intervalle de la valeur, qui est ensuite converti en forme affine.

En pratique, il est plus facile et plus commun de donner l'intervalle. La première méthode d'initialisation est utile pour l'implémentation des méthodes de la classe Affine.

Les opérations d'addition et de soustraction se font ainsi :

$$\hat{x} \pm \hat{y} = (x_0 \pm y_0) + \sum_{i=1}^{n} (x_i \pm y_i)\epsilon_i$$

Comme pour la classe Interval, les mêmes opérateurs arithmétiques, logiques ainsi que les fonctions mathématiques sont définis par les méthodes de la classe Affine. Les détails de leur implémentation sont donnés dans la documentation, dans la section *Affine Arithmetic (AA)*.

2.3 Calcul de précision

L'objectif était de créer une librairie de calcul multiprécision, c'est-à-dire que l'utilisateur puisse gérer la précision des calculs. Pour cela, nous avons utilisé la librairie *mpmath*[7].

En effet, au lieu de faire les calculs avec des flottants (de type *float*), nous utilisons dans les classes Interval et Affine des flottants multiprécision (de type *mpf*) définis par *mpmath*. L'avantage des *mpf* est que l'utilisateur peut régler la précision grâce à deux paramètres :

— Le nombre de décimales avec l'attribut **mp.dps**

— Le nombre de bits utilisés avec l'attribut **mp.prec**

Pour changer la précision, il suffit de modifier l'un des deux paramètres. D'après la documentation de *mpmath*, on a la relation suivante :

$$prec = 3.33dps$$

Par exemple, il faut 333 bits pour encoder le nombre π avec 100 décimales.

De plus, nous utilisons les opérations arithmétiques de *mpmath* pour les méthodes des classes Interval et Affine afin de gérer les arrondis grâce à l'attribut **rounding**. Par exemple, pour la méthode **add** de la classe Interval, nous utilisons la fonction **fadd** de *mpmath*. Nous arrondissons vers le bas le valeur de la borne inférieure (*rounding="f"* pour *floor*) et vers le haut la valeur de la borne supérieure (*rounding="c"* pour *ceiling*). Concernant les x_i d'une forme affine, les arrondis sont en mode "*away from zero*", c'est-à-dire que le résultat d'une opération doit d'éloigner le plus possible de 0. Cela s'écrit *rounding="u"* pour *up*. Cela permet de toujours obtenir l'intervalle du résultat exact de l'opération inclus dans l'intervalle calculé.

Pour l'utilisation de la librairie *affapy*, nous avons créé la classe **precision** afin que l'utilisateur gère la précision des calculs sans importer la librairie *mpmath*. En effet, cette classe permet d'utiliser ce qu'on appelle des décorateurs de fonctions pour définir la précision pour une fonction donnée. Cela correspond à une ligne mise au dessus de la fonction qui modifie l'environnement dans lequel la fonction va être exécutée. Par exemple, pour notre classe, le décorateur **@precision(dps=30)** permet de régler une précision de 30 décimales pour les nombres utilisés. La classe peut aussi être utilisée avec le mot clé de Python

7. http://mpmath.org/

with. Cela fonctionne comme le décorateur mais pour une portion de code.

Plus de précisions sont apportées sur l'utilisation de cet outil dans la documentation, dans la section *Precision context*.

2.4 Tests unitaires

En programmation, le **test unitaire** est une procédure permettant de vérifier le bon fonctionnement d'une partie précise d'un programme. Pour ce projet, nous avons mis en place des tests unitaires pour vérifier chaque méthode des classes Interval et Affine. Vous les trouverez dans le répertoire **test**.

Nous écrivons ces tests avant d'écrire les méthodes ce qui nous aide à déterminer comment écrire nos fonctions, c'est-à-dire ce qu'elles doivent retourner pour les différents cas. Si tous les tests retournent un succès, cela signifie que la partie couverte par les tests fonctionne comme prévu. Sinon, nous pouvons corriger efficacement les erreurs en nous focalisant sur la fonction défaillante et en analysant les cas qui ne fonctionnent pas.

De plus, pour améliorer la fiabilité du code, nous avons mis en place ce qu'on appelle l'**intégration continue**. Cela consiste à vérifier que chaque modification effectuée sur le code source ne provoque pas une nouvelle erreur. Pour cela, les tests sont lancés automatiquement à chaque commit.

Pour écrire les tests unitaires, nous avons utilisé le module **unittest** de Python. On peut séparer les tests effectués pour la librairie *affapy* en deux types :

- **Tests d'égalité** : comparent si deux variables sont égales ou non. Ils sont utilisés pour les tests dont on peut déterminer facilement le résultat. Par exemple, des additions ou soustractions simples entre des instances d'**Interval** et d'**Affine**.
- **Tests d'inclusion** : vérifient si une variable est incluse dans l'autre. Ils sont utilisés pour les opérations plus compliquées, comme la division et les fonctions trigonométriques par exemple. Ces tests vérifient si le résultat attendu est inclus dans le résultat calculé. Cela contribue à la fiabilité des résultats. Par exemple, pour tester la fonction *inv* de **Affine**, nous testons si l'intervalle associé à l'inverse de la forme affine $x = 10 + 6\epsilon_1$, qui est $\left[\frac{1}{16}, \frac{1}{4}\right]$, est bien compris dans l'intervalle du résultat de $inv(x)$.

Nous utilisons le décorateur *precision* avec 50 décimales (*dps=50*) pour les différents tests afin de vérifier la validité des modes d'arrondi.

2.5 Exemples et comparaison des arithmétiques

Dans la documentation, la dernière section nommée *Examples* explique les différents exemples du répertoire **examples**. Ceux-ci montrent l'utilisation de

la librairie *affapy*.

Les exemples 1 à 5 sont inspirés de ceux de la librairie *libaffa*[8]. Les exemples 4 et 5 montrent notamment la différence de précision évoquée précédemment entre l'arithmétique affine et l'arithmétique d'intervalles.

Expliquons l'exemple 5. On considère la fonction suivante :

$$x \mapsto \frac{\sqrt{x^2 - x + \frac{1}{2}}}{\sqrt{x^2 + \frac{1}{2}}}$$

On va alors considérer cette fonction sur un intervalle $[a, b]$ dont on extrait une subdivision régulière composée de n sous-intervalles. En lançant le programme **example5.py**, on peut donner en argument la borne inférieure, la borne supérieure et le nombre de sous-intervalles. Par défaut, si rien aucun argument n'est donné, le programme considère la fonction sur l'intervalle $[0, 5]$ avec 60 sous-intervalles.

Le programme va ensuite évaluer la fonction sur chacun des sous-intervalles avec les deux arithmétiques. Le résultat correspond alors à un box. Voici le graphe généré par cet exemple :

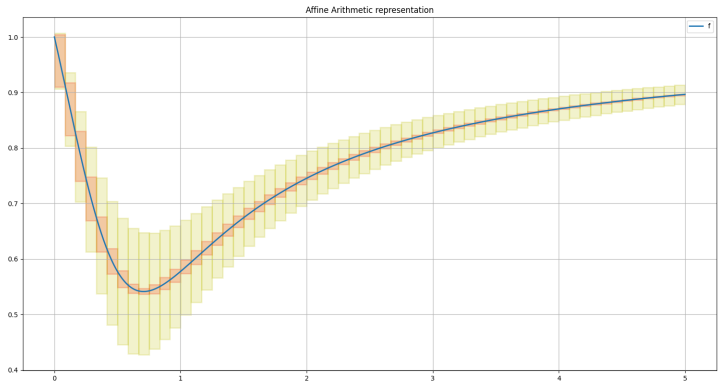


FIGURE 1 – Exemple 5

On peut voir :

— **En bleu** : la fonction

— **En jaune** : l'évaluation de la fonction par l'arithmétique d'intervalles

8. http://www.nongnu.org/libaffa/

— **En orange** : l'évaluation de la fonction par l'arithmétique affine

On voit clairement sur ce graphe que l'arithmétique affine est bien plus précise que l'arithmétique d'intervalles. L'exemple 4 fonctionne de la même manière pour une autre fonction.

3 Bilan

Pour conclure, les objectifs du projets sont atteints. La librairie *affapy* est utilisable et téléchargeable en utilisant *pip* avec la commande :

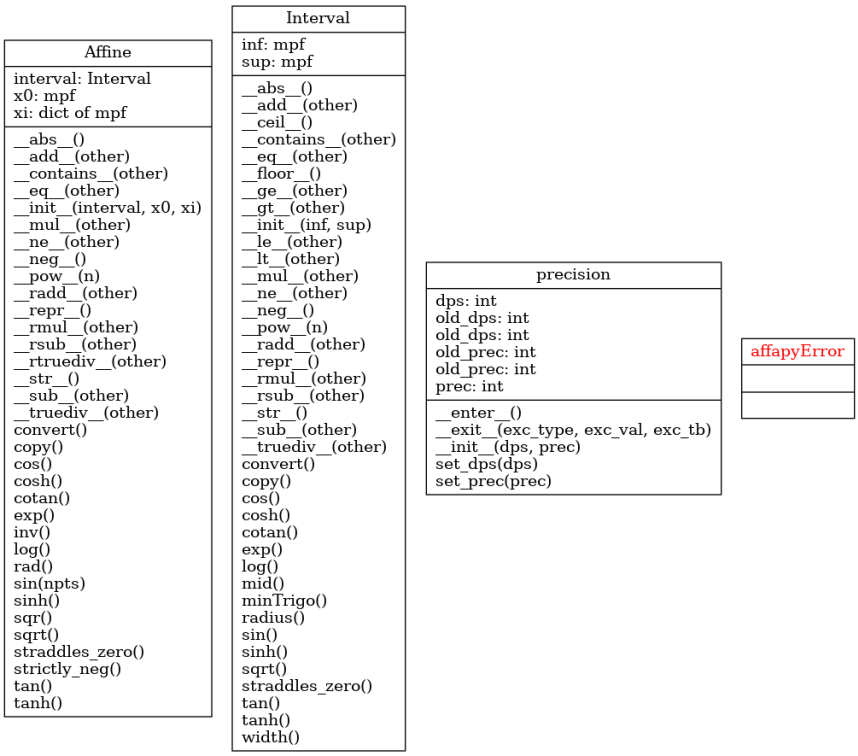
pip3 install affapy

La documentation réalisée avec *Sphinx* est complète et consultable sur ce lien : https://affapy.readthedocs.io/en/latest/index.html. Des exemples aident les utilisateurs à se familiariser avec la librairie.

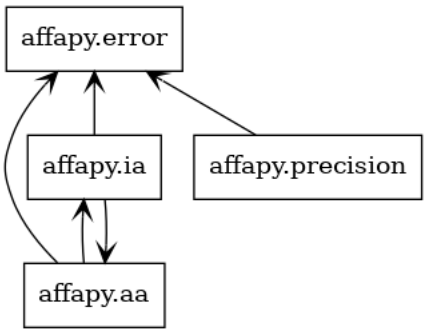
L'arithmétique affine étant un modèle en pleine évolution, il se pourrait que la librairie *affapy* évolue et propose de nouveaux modèles.

Annexes

Vous trouvez ci-dessous le diagramme des classes de la librairie *affapy* :



Voici ci-dessous les classes de la librairie *affapy* :



Références

[1] Ahmed Touhami, *Utilisation et Extension de l'Arithmétique Affine dans les Algorithmes Déterministes d'Optimisation Globale*, 2002

[2] J. Stolfi, *An Introduction to Affine Arithmetic*, 2003

[3] Jordan Ninin, Frédéric Messine, Pierre Hansen, *Combining interval and affine arithmetic with linear reformulation in deterministic global optimization*, 2003

Librairie *Libaffa*
Librairie *mpmath*