Projet SFPN

Manipulation de suites P-récursives avec SageMath

Auteurs :
Mathis Caristan
Aurélien Lamoureux

Encadrant :
Marc Mezzarobba

Résumé

Ce rapport présente le travail que nous avons effectué au cours de ce projet. Nous présentons dans un premier temps ce que sont les suites P-récursives, et les motivations des travaux autour de ce domaine. Puis nous présentons l'outil SageMath et la bibliothèque OreAlgebra. Enfin, nous détaillons les choix et détails de l'implémentation que nous avons réalisé, avant de discuter des limites de celle-ci et des possibles améliorations.

Table des matières

1 Introduction

1.1 Suites p-récursives & Algèbre d'Ore

Les suites sont beaucoup utilisées en mathématiques et dans différents domaines scientifiques, et on cherche, comme souvent en informatique, à en avoir une représentation exacte. De plus, il est généralement important que cette représentation soit également efficace pour la manipulation mathématique de ces suites.

On s'intéresse ici en particulier aux suites dites p-récursives. Une suite $(u_n)_{n\in\mathbb{N}}$ sur un corps $\mathbb{K}$ est dite p-récursive si elle est solution d'une équation de la forme :

$$\sum_{i=0}^{k} p_i(n) u_{n+i} = 0 \tag{1}$$

où, les p_i sont des polynômes en n. Il est importante de noter que contrairement à des suites arbitraires, les suites p-récursives, bien que comportant un nombre potentiellement infini de termes, peuvent être représentées exactement simplement avec la relation de récurrence, et les conditions initiales. Des exemples communs de suites p-récursives sont par exemple la suite de Fibonacci, ou la fonction factorielle :

$$\begin{aligned} \text{Fibonacci} : F_{n+2} - F_{n+1} - F_n = 0, &\qquad F_0 = 0, F_1 = 1 \\ \text{Factorielle} : (n+1)! - (n+1)(n!) = 0, &\qquad 0! = 1 \end{aligned}$$

De plus, les suites p-récursives forment un anneau (Donner les détails ?). Dès lors, il semble pertinent de réaliser une implémentation utilisant ces propriétés mathématiques afin de manipuler et utiliser les suites p-récursives.
TODO : ore_algebra what & why
Intro ore... Pour le présent projet, nous n'avons besoin que des notions traitant des suites p-récursives.

1.2 Python & Sage

Sage est un outil de calcul formel libre. Il a été créé notamment pour proposer une alternative *opensource* aux logiciels existants comme Mathematica, Matlab, Maple ... Contrairement à ces logiciels, Sage s'appuie sur des outils et librairies déjà existants comme NumPy, SciPy, matplotlib, FLINT et d'autres... L'utilisation de ces outils est unifiée et uniformisée au travers d'un langage basé sur Python. Ce langage présente une syntaxe qui diffère légérement de celle de Python. Ainsi, Sage est doté d'un "pré-analyseur" ou "pré-parseur", qui transforme les idiomes Sage en pur Python. Bien qu'il existe également d'autres méthodes, on ne s'est intéressé qu'à l'écriture d'un module en Python au cours du projet. En particulier, Python 2, puisque Sage n'est pas compatible avec Python 3 (bien que des efforts soient faits en ce sens).

Bien que Sage fournisse de nombreuses librairies mathématiques, il n'inclut pas encore officiellement de librairie pour l'algèbre d'Ore. Nous avons eu donc recours à une bibliothèque en cours de développement par la communauté qui implémente l'algèbre d'Ore. Nous présentons certains outils de cette bibliothèque dans la section 1.3, tandis que ref propose une présentation plus générale.

1.3 La bibliotèque OreAlgebra

Nous avons évoqué le fait queSage dispose d'une syntaxe propre, qui s'appuie sur celle de Python. C'est celle-ci qui est utilisée dans l'article de documentation du module, et que nous reprendrons ici. Cela permettra également de présenter brièvement les éléments de syntaxe basiques, spécifiques à Sage.
TODO code extracts + explications. Pres ring/fields/pols/orealgebra

2 Méthodologie

La première tâche à laquelle nous nous sommes attelés a été une recherche bibliographique, pour comprendre le sujet (les suites p-récursives), et nos outils (Sage, Python et la bibliothèqe OreAlgebra). Les résultats de cette démarche sont présentés dans la section 1.
Puis nous avons commencé à discuter de l'implémentation. Bien que Sage dispose de sa propre syntaxe, il est d'usage d'écrire les modules en « Python pur ». La syntaxe spécifique de Sage est surtout du sucre syntaxique pour l'interface en ligne de commande. De plus, le pré-parseur de Sage n'est pas d'une robustesse à toutes épreuves, et son utilisation peut engendrer des résultats non voulus, et imprévisibles. Enfin, cela présente également l'intérêt de produire du code réutilisable dans un cadre plus large. Pour ces différentes raisons, le langage de notre implémentation a évidemment été Python [1].

2.1 Objectifs & Étapes

Le module que nous avons écrit est principalement constitué de la défintion d'une clase. Initialement, nous avons convenu d'une hierarchie d'objectifs que nous souhaitions voir atteints par cette classe. Les objectifs prioritaires qu'il nous semblait impératif d'atteindre sont les suivants :

- ✓ Un **constructeur** permettant à l'utilisateur de créer un objet de notre classe, en spécifiant des conditions initiales et un annhilateur.
- ✓ La **surcharge de l'opérateur** `__getitem__`, pour permettre à l'utilisateur d'accéder à un élément de la suite.

1. Plus exactement, Python 2.7.9

- ✓ La **surcharge des opérations** $+$ **et** $\times$ pour additioner et multiplier des instances de la classe entre elles.

Les démarches pour l'accomplissement de ces objectifs sont présentées dans leur section respective : 3.1, 3.2 et 3.3.

Puis, une liste d'objectifs importants parmi laquelle nous souhaitions en implémenter le plus possible :

- ✓ Faire en sorte que le code fonctionne dans plusieurs anneaux, notamment $\mathbb{Q}, \mathbb{R}, \mathbb{C}$ et les corps finis $\mathbb{F}_p$(initialement, on se concentre sur $\mathbb{Z}$).
- ✓ Un constructeur produisant une suite à partir d'une constante, permettant à terme de gérer des opérations du type *suite+constante*.
- ✗ Une méthode pour tester si une suite est constante.
- ✗ La surcharge des opérateurs de comparaison $=$ et $\neq$.
- ✗ Une méthode cherchant un annihilateur d'ordre inférieur produisant la même suite si il en existe un.
- ✗ Un constructeur, qui fabrique l'opérateur de récurrence à partir des premiers termes de la suite uniquement.

Enfin, nous avons établis des objectifs secondaires. Ceux-ci étaient essentiellement des objectifs dont l'intérêt était limité. Certains ont été implémentés quand ils ne requérait pas trop de réflexion :

- ✗ La surcharge des opérateurs de décallage << et >> (dont la sémantique doit être clarifiée).
- ✓ Un itérateur infini.
- ✗ La division d'une suite par une constante.
- ✗ Un constructeur qui fabrique une suite correspondant à une expression Sage.
- ✗ Un moyen de calculer des suites du type $u(3n+2)$ à partir de $u(n)$.

3 Implémentations

Nous présentons ici les détails des implémentations de certaines des fonctionnalités de la classe, et des erreurs et problèmes que nous avons rencontrés pendant la phase de développement.

3.1 Constructeur : `_init_`

Nous avons eu plusieurs changement de directions en ce qui concerne le constructeur. Le principal problème pour cette fonction, a été de déterminer les choix que nous faisions (et allions imposer à l'utilisateur) concernant les conditions initiales. Cette problématique est liée à la possibilité pour une suite d'avoir des valeurs dégénérées, et arrive lorsque le terme dominant de la récurrence a une racine ou plus

dans $\mathbb{Z}$. Prenons par exemple la suite définie par $(n-2)u_{n+1} - u_n = 0, u_0 = 1$. Nous pouvons sans problème calculer les deux termes suivants de la suite, $u_1 = \frac{-1}{2}, u_2 = \frac{1}{2}$, mais la relation de récurrence ne permet pas de calculer u_3, et donc tous les termes suivants. Une solution simple est de permettre à l'utilisateur de fixer des conditions initiales supplémentaires. Dans notre exemple, ajouter la condition $u_3 = 3$ permet de contourner la racine, et calculer toutes les valeurs suivantes, $u_4 = 3, u_5 = \frac{3}{2} \dots$

Suite à ce raisonnement, nous avions pris la décision de vérifier le terme dominant de l'annihilateur, et d'imposer à l'utilisateur de spécifier des valeurs supplémentaires pour toutes les racines de celui-ci. De plus, toutes les valeurs dans $[i_{min}, r_{max}]$ devaient être renseignées, où i_{min} est l'indice de début de la suite, et r_{max} la plus grande racine dans $\mathbb{Z}$. Cette méthode nous est cependant rapidement apparue comme étant problématique, en particulier dans le cas de grandes racines. Il est en effet contre-productif de demander à l'utilisateur toutes les valeurs d'une suite, jusqu'à la plus grande racine, si celle-ci est par exemple de l'ordre de 10^3. De même, dans le cas où il n'a besoin que de termes dont l'indice est petit devant r_{max}, l'obliger à donner les valeurs jusqu'à cette racine n'est pas logique. Nous sommes donc revenus sur cette décision, et avons choisi à la place de simplement lever une exception lorsque l'utilisateur demande le calcul d'une valeur dégénérée pour laquelle il n'a pas saisi de condition initiale supplémentaire.

Autoriser les conditions initiales supplémentaires entraîne cependant un second problème. Celui de savoir comment traiter ces valeurs vis-à-vis du calcul des éléments suivants. Deux approches ont été envisagées, et il n'est pas évident laquelle est la plus intéressante dans un cadre de calcul scientifique. La première, et sans doute la plus simple d'un point de vue de l'implémentation, consiste à ne pas considérer ces valeurs pour les calculs, mais uniquement pour les valeurs que prend la suite. La seconde au contraire, nous amène à considérer ces valeurs également du point de vue calculatoire. Considérons l'exemple suivant : $u_{n+2} - u_{n+1} - u_n = 0, u_0 = 0, u_1 = 1, u_5 = 6$. Dans les deux cas, les 6 premiers termes sont $(0, 1, 1, 2, 3, \mathit{6})$. Mais dans le premier cas, les termes suivants sont les termes usuels $(8, 13, 21...)$ car la valeur exceptionnelle n'est pas prise en compte pour le calcul des termes suivants. Tandis que dans le second cas, les termes suivants sont $(9, 15, 24...)$. L'idéal semblerait de laisser le choix à l'utilisateur quel mode il souhaite utiliser au moment de l'instanciation de la suite, par le biais d'un argument optionnel.

3.2 Cacul d'un élément de la suite : `__getitem__` et `to_list`

Pour obtenir un élément d'une suite, il nous semblait logique de surcharger l'opérateur `__getitem__`, qui permet d'obtenir un élément de la manière suivante : `u[42]`. En plus de ce choix, nous avons également surchargé la fonction `to_list(n)`, qui nous permet d'obtenir une liste des éléments de la suite, jusqu'à n. Nous avons utilisé plusieurs implémentations successives pour ces fonctions. Après être passés par

l'innévitable « méthode brouillone », ou le code était duppliqué, et aucune des deux fonctions n'avaient une identité propre, nous avons finalement opté pour faire en sorte que `to_list` appelle `__getitem__`. Une fois ce point éclairci, il nous restait à savoir comment implémenter le calcul même. Notre premier choix a été d'utiliser la fonction `to_list` des objets OREALGEBRA. En effet, pout un annihilateur et des conditions initiales données, celle-ci génère tous les éléments jusqu'à la valeur désirée.

Cependant, nous avons remarqué que cette méthode présentait un défaut important : son temps de calcul. Ceci est dû au fait que pour calculer un élément n, cette fonction calcule tous les éléments dans l'intervalle $[0, n]$. Or ceci est complètement inneficace dans le cas où on ne veut que l'élément n. Pour palier à ce problème, notre encadrant nous a suggéré d'utiliser la fonction `forward_matrix_bsplit`. Explications fctmt ou boite noire. Cette fonction permet de calculer directement un élément de la suite, sans calculer tous ses prédécesseurs [2].

Enfin, la dernière implémentation que nous ayons réalisé pour ce calcul est la suivante. Nous suspections que la fonction `forward_matrix_bsplit` était moins efficace que la fonction `to_list` pour des valeurs basses de n. Ainsi, dans l'idée d'optimiser au mieux la fonction, nous avons comparé les exécutions de ces deux fonctions en faisant varier deux paramètres. D'une part nous avons fait varier n, et d'autre part l'ordre des suites utilisées. Les résultats sont présentés dans la figure 1. Ces résultats ont été obtenus en prennant la moyenne du temps d'exécution sur 10 exécutions avec des annihilateurs générés aléatoirement (et ne créant pas de termes dégénérés) et d'ordre fixé. On note que dès le calcul d'éléments de l'ordre de 100, la méthode `forward_matrix_bsplit` est déjà plus efficace. Notons également que l'influence de l'ordre de la récurrence semble augmenter pour des valeurs plus élevées de n. Ces résultats nous ont permis de déterminer empiriquement une valeur à partir de laquelle nous passons d'une méthode à l'autre. Nous avons fixé cette valeur a 100.

Le dernier point que nous avons implémenté concernant ces fonctions, est la possibilité d'utiliser les slices de Python. Les slices, sont des objets qui peuvent être paramètre de `__getitem__`. Pour une liste, ils permettent d'indiquer qu'on souhaite obtenir une « tranche » de la liste, par exemple, les éléments 8 à 13, éventuellement avec un pas. Dans notre cas, nous avons simplement adapté notre fonction, afin qu'elle calcule le premier élément de la tranche avec la méthode optimale, et les éléments suivants avec la méthode `to_list` de l'annhilateur.

2. Plus précisément, cette fonction calcule k éléments, où k est l'ordre de la récurrence de la suite.

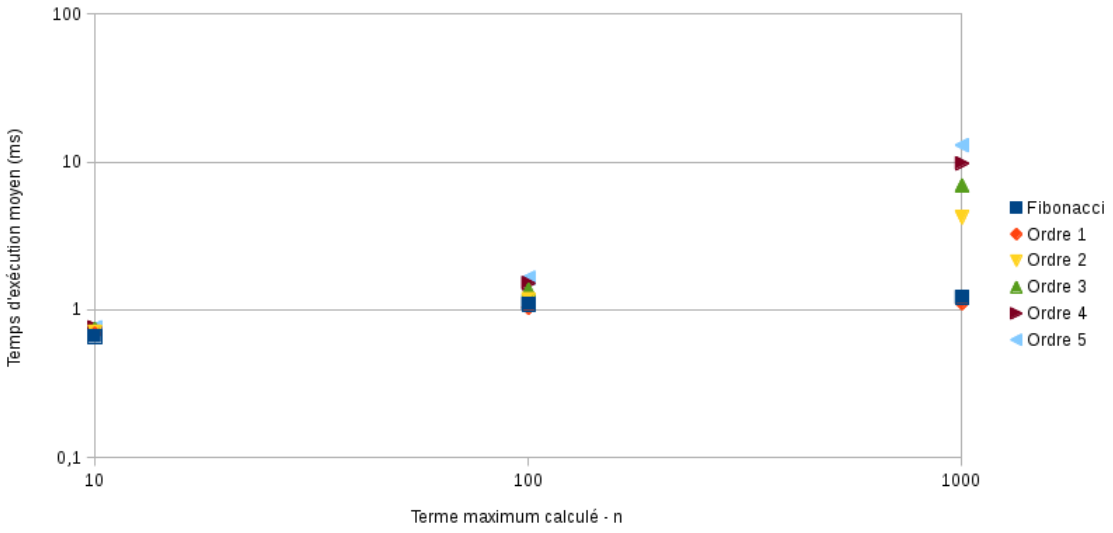

FIGURE 1 – Rapport du temps de calcul avec `to_list` sur le temps de calcul avec `forward_matrix_bsplit`, pour différentes valeurs de n, et différents ordres de récurrence. On utilise une échelle logarithmique.

3.3 Structure d'anneaux : `_add_` et `_mul_`

Nous avons déjà évoqué le fait que les suites P-récursives forment un anneau, l'implémentation des fonctions d'addition et de multiplication était donc une étape importante de l'avancée du projet. L'annihilateur de la somme de deux suites est definie comme le plus petit multiple commun à gauche (`lclm`) des deux annihilateurs des suite sommées. La fonction qui calcule le `lclm` de deux suites est fournie par la bibliothèque OREALGEBRA. Si la nouvelle suite est d'odre n alors ses conditons initales seront la somme des n premiers termes des deux suites. Pour cela nous utilisons la fonction `to_list` que nous avons écrite et récuperons les n premiers éléments des suites et les sommons. Par exemple si l'on ajoute la suite de `Fibonacci` et celle de `Tribonacci` :

$$\begin{aligned}
\text{Fibonacci} &: F_{n+2} - F_{n+1} - F_n = 0, \qquad F_0 = 0, F_1 = 1 \\
\text{Tribonacci} &: T_{n+3} - T_{n+2} - T_{n+1} - T_n = 0, \qquad T_0 = 0, T_1 = 1, T_2 = 1 \\
\text{Leur Sommme} &: S_{n+5} - 2S_{n+4} - S_{n+3} + S_{n+2} + 2S_{n+1} + S_n = 0, \\
&\qquad S_0 = 0, S_1 = 2, S_2 = 2, S_3 = 4, S_4 = 7
\end{aligned}$$

Cependant la suite nouvellement formée n'est pas toujours aussi facile à traiter,

prenons la cas de la somme des suites des `Entier consecutif` et `Fibonacci`

$$\begin{aligned}
\text{Ent. Consec. : } & nE_{n+1} - (n+1)E_n = 0, \qquad E_0 = 0, N_1 = 1 \\
\text{Fibonacci : } & F_{n+2} - F_{n+1} - F_n = 0, \qquad F_0 = 0, F_1 = 1 \\
\text{Leur Sommme : } & (n-1)D_{n+3} + (-2n+1)D_{n+2} + D_{n+1} + nD_n = 0, \\
& D_0 = 0, D_1 = 2, D_2 = 3
\end{aligned}$$

Cette suite est d'ordre 3, on devrais donc sommer les 3 premiers termes de E et F (respectivement $[0, 1, 2]$ et $[0, 1, 1]$) ce qui donnerai comme conditions initial pour D : $[0, 2, 3]$. Si l'on essaie de dérouler la récurrence pour calculer le prochain terme, nous obtenons $D_3 = 5$, mais nous remarquons que le terme D_4 est dégénéré (cf section 3.1), et ne peut être calculé. En revanche,F_4 et E_4 existent, nous avons donc décidé que lorsque l'annihilateur de la somme de deux suites est dans un cas similaire à celui-ci, nous calculons la somme des éléments dégénérés et les rajoutons dans les conditions initales.

En ce qui concerne la multiplication, le principe est exactement le même sauf que la multiplication de deux opérateur se calcule grâce à la fonction `symmetric_product`, qui est également fournie par OREALGEBRA.

3.4 Autres objectifs

Conclusion

Dans l'ensemble, nous avons réussi à développer une classe répondant aux objectifs principaux que nous nous étions fixés. Celle-ci permet une manipulation basique des suites P-récursives et propose également des fonctionnalités secondaires, telle que la possibilité de travailler dans différents ensembles ($\mathbb{Z}, \mathbb{Q}, \mathbb{R}$, nous n'avons pas testé $\mathbb{C}$ et $\mathbb{F}_p$), ou bien la manipulation de constantes comme une suite récursive.
Les principales difficultés que nous avons rencontrées, ont été pour la plupart liées à des problèmes de définitions, ou de paradigme. Pour les contourner, il a fallu se mettre à la place de l'utilisateur du module, et comprendre ce qu'il en attend lorsqu'il choisit de l'utiliser. Certains dilemmes n'ont pas de solution unique, et dépendent du cas spécifique auquel est confronté l'utilisateur. Pour cette raison, nous nous sommes efforcés de rester généraliste dans la manière dont nous traitions les problèmes, et de rendre les traitements spécifiques optionnels.