

• **Projet SFPN: Manipulation de suites P-récurives avec SageMath**

Encadré par Marc MEZZAROBBA

Mathis CARISTAN & Aurélien LAMOUREUX

22 mai 2017

• **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-récurives, 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	3
1.1	Suites p-récurrentes & Algèbre d'Ore	3
1.2	Python & Sage	3
1.3	La bibliothèque OREALGEBRA	4
2	Méthodologie	4
2.1	Objectifs & Étapes	4
3	Implémentations	5
3.1	Constructeur : <code>__init__</code>	5
3.2	Cacul d'un élément de la suite : <code>__getitem__</code> et <code>to_list</code>	6
3.3	Structure d'anneaux : <code>__add__</code> et <code>__mul__</code>	8
	Conclusion	9
	Résultats	9
	Perspectives	9

1 Introduction

1.1 Suites p-récurives & 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-récurives. Une suite $(u_n)_{n \in \mathbb{N}}$ sur un corps \mathbb{K} est dite p-récurive si elle est solution d'une équation de la forme :

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

où, les p_i sont des polynômes en n . Il est importante de noter que contrairement à des suites arbitraires, les suites p-récurives, 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-récurives sont par exemple la suite de Fibonacci, ou la fonction factorielle.

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

De plus, les suites p-récurives 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-récurives.

TODO : [ore_algebra what & why](#)

Intro ore... Pour le présent projet, nous n'avons besoin que des notions traitant des suites p-récurives.

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 bibliothèques 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 légèrement de celle de Python. Ainsi, Sage est doté d'un "pré-analyseur" ou "pré-parseur", qui transforme les idiomes Sage en pur Python. Ainsi, il est possible d'écrire des bibliothèques pour Sage en Python pur ou en "langage sage". Bien qu'il existe également d'autres méthodes, on ne s'est intéressé qu'à celles-ci au cours du projet. [ref doc sage ?](#)

Comme évoqué plus haut, Sage est basé sur Python, et c'est donc naturellement que nous avons choisi ce langage pour le 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. L'utilisation de cette bibliothèque est présentée dans 1.3 en ce qui concerne les outils qui nous sont utiles dans ce projet, et dans [ref](#) pour une présentation plus générale.

1.3 La bibliothèque OREALGEBRA

Remarquons enfin que Sage dispose d'une syntaxe propre, qui s'appuie sur celle de Python. C'est celle-ci qui est utilisée dans la documentation de la bibliothèque, 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-récurrentes), et nos outils (Sage, Python et la bibliothèque OREALGEBRA). Les résultats de cette démarche sont présentés dans la partie ??.

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 Python¹.

2.1 Objectifs & Étapes

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

1. Plus exactement, Python 2.7.9

- ✓ Un **constructeur** permettant à l'utilisateur de créer un objet de notre classe, en spécifiant des conditions initiales et un annihilateur.
- ✓ La **surcharge des opérations** $+$ et \times pour additionner et multiplier des instances de la classe entre elles.
- ✓ La **surcharge de l'opérateur** `__getitem__`, pour permettre à l'utilisateur d'accéder à un élément de la suite.

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 requéraient pas trop de réflexion :

- ✗ La surcharge des opérateurs de décalage $<<$ 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 faisons (et allons 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 le second terme de la suite, $u_1 = -1$, mais la relation de récurrence ne permet pas de calculer u_2 , 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 $u_2 = 3$ permet de contourner la racine, et calculer toutes les valeurs suivantes.

Suite à ce raisonnement, nous avons 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 vite 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 la valeur pour 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 $(0, 1, 1, 2, 3, 6)$. 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. Tandis que dans le second cas, les termes suivants sont $(9, 15, 24...)$.

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

Pour calculer 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 tous les éléments de la suite, jusqu'à n . Nous avons utilisé plusieurs implémentations successives pour ces fonctions. Après être passés par l'inévitable « méthode brouillone », où le code était dupliqué, et aucune des deux fonctions n'avaient une identité propre, nous avons finalement opté pour faire en

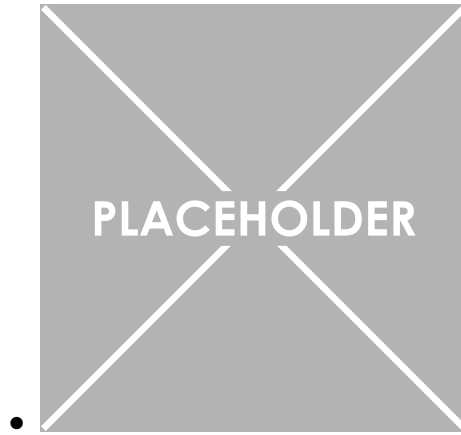


FIGURE 1 – TODO

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, pour 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 inefficace 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_bsplitt`. **Explications fctmt ou boîte noire**. Cette fonction permet de calculer directement un élément de la suite, sans calculer tous ses prédécesseurs².

Enfin, la dernière implémentation que nous avons réalisé pour ce calcul est la suivante. Nous suspectons que la fonction `forward_matrix_bsplitt` 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. **TODO commentaires**. Ces résultats nous ont permis de déterminer empiriquement une valeur à partir de laquelle nous passons d'une méthode à l'autre.

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

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

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.

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

Puisque les suites P-récursives, munies des lois de l'addition et de la multiplication, forment un anneau, l'implémentation des fonctions d'addition et de multiplication de ces suites nous a parue évident. L'opérateur de la somme de deux suites est définie comme le plus petit multiple commun à gauche (`lclm`) des deux opérateurs des suite que l'on ajoute. La fonction qui calcule le `lclm` de deux suites est fournie par la bibliothèque `OREALGEBRA`. Si la nouvelle suite est d'ordre n alors ces conditions initiales seront la somme des n premiers termes des deux suites. Pour cela nous utilisons la fonction `to_list` que nous avons écrite et récupérons les n premiers éléments des suites et les sommions. Par exemple si l'on ajoute la suite de `Fibonacci` et celle de `Tribonacci`

$$\text{Fibonacci} : F_{n+2} - F_{n+1} - F_n = 0, \quad F_0 = 0, F_1 = 1$$

$$\text{Tribonacci} : T_{n+3} - T_{n+2} - T_{n+1} - T_n = 0, \quad T_0 = 0, T_1 = 1, T_2 = 1$$

$$\begin{aligned} \text{Leur Somme} : S_{n+5} - 2 * S_{n+4} - S_{n+3} + S_{n+2} + 2 * S_{n+1} + S_n &= 0, \\ S_0 = 0, S_1 = 2, S_2 = 2, S_3 = 4, S_4 = 7 \end{aligned}$$

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`

$$\text{Ent. Consec.} : n * E_{n+1} - (n + 1)E_n = 0, \quad E_0 = 0, N_1 = 1$$

$$\text{Fibonacci} : F_{n+2} - F_{n+1} - F_n = 0, \quad F_0 = 0, F_1 = 1$$

$$\begin{aligned} \text{Leur Somme} : (n - 1) * D_{n+3} + (-2 * n + 1) * D_{n+2} + D_{n+1}n * D_n &= 0, \\ D_0 = 0, D_1 = 2, D_2 = 3 \end{aligned}$$

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 : $[0,2,3]$. Mais maintenant si l'on essaie de dérouler la recurrence pour calculer le prochain terme nous obtenont $D_3 = 5$ mais pour le terme D_4 nous obtenons une erreur car on a $0 * D_4 + 1 * D_3 + D_2 + 1 * D_1 = 0$ on ne peut calculer D_4 , cela ce produit a cause du $n-1$ devant D_{n+3} mais comme F_4 et E_4 existe, nous avons donc décidé que lorsque l'opérateur de la somme de deux suites est dans un cas similaire a celui-ci, nous calculons la somme des éléments dégénératifs et les rajoutons dans les conditions initiales.

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

Conclusion

Résultats

Perspectives