Projet SFPN: Manipulation de suites P-récursives 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-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 :

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

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.

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

De plus, les suites p-récursives forment un aneau (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 et 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", qui transforme les idiomes Sage en pur Python. Ainsi, il est possible d'écrire des librairies pour 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.

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 bibliotèque 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-récursives), et nos outils (Sage, Python et la bibliothèqe 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 Python [1].

2.1 Module Python

La base du module a été d'écrire une classe Python (init. n'étend aucun classes). Cette classe devait notamment permettre d'utiliser la représentation basée sur la relation de récurrence, et des conditions initiales. Immédiatement après, nous avons surchargé l'opérateur `__getitem__` pour accéder au n-ième terme de la suite. Initialement, nous calculions tous les termes de la suite, jusqu'à celui voulu, que nous

1. Plus exactement, Python 2.7.9

renvoyions, mais cette méthode est très inefficace. Nous avons donc résolu d'utiliser la fonction `forward_matrix` du module ORE_ALGEBRA à la place (exemple et comparaison complex avec Fibo ?)
calculer ts les elts vs calculer que le bon élement, exemple de différence tps exec sur 100000 !)
Par la suite, nous avons également surcharger les opérateurs d'addition, soustraction et multiplication, en accord avec les lois de l'agèbre d'Ore.

2.2 Objectifs & Étapes

Initialement, nous avons convenu d'une hierarchie d'objectifs que nous souhaitions atteindre pour ce projet. 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 des opérations** $+$ **et** $\times$ pour additioner 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 $mathbbF_p$(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

3.1 Constructeur : `__init__`

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 », 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. 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 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

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

FIGURE 1 – TODO

éléments suivants avec la méthode `to_list` de l'annhilateur.

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

Comme les suites P-récucives munis des lois de l'addition et de la multiplication forme un anneaux, l'implementation des fonctions d'ajout et de multiplication de ces suites nous a parus evident. L'operateur de la somme de deux suites est definie comme le plus petit multiple commun (`lclm`) des deux operateurs des suite que l'on ajoute.la fonction qui calcul le `lclm` de deux suites est fournie par la bibliothèque OREALGEBRA. Si la nouvelle suite est d'odre n alors ces conditons initals serons la somme des n premiers termes des deux suites. Pour cela nous utilison la fonction `ToList` que nous avons crée et recuperons les n premiers elements des suites et les sommons. Par exemple si l'on ajoute la suite de `Fibonacci` et celle de `Tribonacci`

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

Cependant la suite nouvellement formé n'est pas toujours aussi facile à traiter, prenons la cas de la somme des suites des `Entier consecutif` et `Fibonacci`

Ent. Consec. : $n * E_{n+1} - (n+1)E_n = 0, \qquad E_0 = 0, N_1 = 1$
Fibonacci : $F_{n+2} - F_{n+1} - F_n = 0, \qquad F_0 = 0, F_1 = 1$
Leur Sommme : $(n-1) * D_{n+3} + (-2 * n + 1) * D_{n+2} + D_{n+1} n * D_n = 0,$
$D_0 = 0, D_1 = 2, D_2 = 3$

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 recurence 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 peux 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 calcultons la somme des éléments dégénératifs et les rajoutons dans les conditions initals.

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 fournie par OREALGEBRA.

Conclusion

Résultats

Perspectives