

Projet SFPN : Manipulation de suites P-récurrentes avec SageMath

Mathis Caristan & Aurélien Lamoureux

Sous la responsabilité de Marc Mezzarobba

Université Pierre & Marie Curie

29/05/2017

1 Introduction

2 Contenu du module

Les suites P-récurrentes sont des objets couramment utilisés en mathématiques et en sciences.

Problématique

- **La question se pose de comment représenter et manipuler informatiquement ces objets.**
- Les suites sont infinies.

Les suites P-récurrentes sont des objets couramment utilisés en mathématiques et en sciences.

Problématique

- **La question se pose de comment représenter et manipuler informatiquement ces objets.**
- Les suites sont infinies.

Solution

Il est nécessaire d'utiliser les propriétés mathématiques des suites P-récurrentes.

Définition formelle

Une suite P-récurrente sur un corps \mathbb{K} vérifie la propriété suivante :

$$\sum_{i=0}^k P_i(n) u_{n+i} = 0$$

où les P_i sont des polynômes en n , et k est l'ordre de la récurrence.

Une suite P-récurrente peut être représentée exactement avec sa relation de récurrence, et ses conditions initiales*

Exemples

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

$$\text{Factorielle : } (n+1)! - (n+1)(n!) = 0, \quad 0! = 1$$

TODO

SageMath, qu'est-ce que c'est ?

- Un logiciel de calcul formel
- Opensource
- Construit sur un ensemble d'outil pré-existant et Python
- Basé sur Python
- Doté d'une syntaxe spécifique pour la ligne de commande

Python ?

- C'est le langage sur lequel est basé Sage
- Python 2.7.9
- Les idiomes Sage sont transformés en Python pur
- Possibilité d'écrire des modules pour Sage en Python

- Implémente l'algèbre d'Ore
- Non intégrée au projet Sage, et développée par la communauté
- Contient une partie des outils nécessaires à la réalisation du projet
 - Définir une algèbre dans laquelle travailler

```
R.<n> = PolynomialRing(ZZ)
A.<Sn> = OreAlgebra(R)
```
 - Les fonctions `lclm` et `symmetric_product` pour $+/×$

```
annihilSum = annihil1.lclm(annihil2)
annihilProd = annihil1.symmetric_product(annihil2)
```
 - La fonction `forward_matrix_bsplitt` pour le calcul d'un terme

Problématique

Créer un module permettant la manipulation des suites p-récurives

Caractéristiques

- En Python
- Basé sur le modèle de programmation objet : une classe
- Surcharge d'opérateurs
- Des tests

Notre classe n'étend aucune classe pré-existante.

Objectifs principaux

- Un constructeur
- Les opérations $+$ et \times
- Une fonction pour calculer un élément

Objectifs du module

Objectifs principaux

- Un constructeur
- Les opérations $+$ et \times
- Une fonction pour calculer un élément

Objectifs importants

- Travailler dans différents anneaux
- Des suites constantes
- Une méthode qui teste si une suite est constante
- Les tests d'égalité/inégalité
- Un constructeur qui devine la récurrence

Objectifs du module

Objectifs principaux

- Un constructeur
- Les opérations $+$ et \times
- Une fonction pour calculer un élément

Objectifs importants

- Travailler dans différents anneaux
- Des suites constantes
- Une méthode qui teste si une suite est constante
- Les tests d'égalité/inégalité
- Un constructeur qui devine la récurrence

Objectifs secondaires

- Les opérateurs $<<$ et $>>$
- Un itérateur (infini)
- La division par une constante
- Un constructeur à partir d'une expression symbolique

Constructeur

C'est la méthode appelée par Python lors d'une instantiation de la classe.
C'est la première interaction de l'utilisateur avec le module.

Comportement par défaut

```
u = PRecSequence (conditions,annihilateur)
```

conditions est un dictionnaire
annihilateur est un objet du module OreAlgebra

Comportements secondaires

D'autres comportements sont possibles, grâce aux arguments optionnels de Python

- Création d'une suite constante
- "Guessing" a partir d'une liste d'éléments

Faut-il remplacer l'utilisation des arguments mots-clefs par le décorateur
`@classmethod` ?

Les valeurs dégénérées

Lorsque le polynôme dominant a des racines dans \mathbb{Z} :

$$(n-1)u_{n+1} - u_n = 0, u_0 = 1 \qquad u_0 = 1, u_1 = (-1)u_0 = -1, u_2 = ???$$

Les valeurs dégénérées

Lorsque le polynôme dominant a des racines dans \mathbb{Z} :

$$(n-1)u_{n+1} - u_n = 0, u_0 = 1 \quad u_0 = 1, u_1 = (-1)u_0 = -1, u_2 = ???$$

Les conditions initiales supplémentaires

Il est nécessaire de permettre de fixer des conditions supplémentaires.

Les valeurs dégénérées

Lorsque le polynôme dominant a des racines dans \mathbb{Z} :

$$(n-1)u_{n+1} - u_n = 0, u_0 = 1 \qquad u_0 = 1, u_1 = (-1)u_0 = -1, u_2 = ???$$

Les conditions initiales supplémentaires

Il est nécessaire de permettre de fixer des conditions supplémentaires.

1^{re} idée : Obliger l'utilisateur à renseigner les valeurs dégénérées. Obliger l'utilisateur à saisir *toutes* les valeurs jusqu'à la dernière racine.

Les valeurs dégénérées

Lorsque le polynôme dominant a des racines dans \mathbb{Z} :

$$(n-1)u_{n+1} - u_n = 0, u_0 = 1 \quad u_0 = 1, u_1 = (-1)u_0 = -1, u_2 = ???$$

Les conditions initiales supplémentaires

Il est nécessaire de permettre de fixer des conditions supplémentaires.

1^{re} idée : Obliger l'utilisateur à renseigner les valeurs dégénérées. Obliger l'utilisateur à saisir *toutes* les valeurs jusqu'à la dernière racine.

2^e idée : Lever des exceptions

Comment traiter les conditions initiales supplémentaires ?

Les valeurs n'influent pas le calcul

$$u_{n+2} - u_{n+1} - u_n = 0, u_0 = 0, u_1 = 1, u_5 = 6$$

Les valeurs de la suite sont (0, 1, 1, 2, 3, 6, 8, 13, 21...)

Les valeurs influent sur les termes suivants

$$u_{n+2} - u_{n+1} - u_n = 0, u_0 = 0, u_1 = 1, u_5 = 6$$

Les valeurs de la suite sont (0, 1, 1, 2, 3, 6, 9, 15, 24...)

Accéder à un élément de la suite

Il faut surcharger l'opérateur `__getitem__`, qui permet en Python d'accéder à l'élément n : `fibonacci[n]` ou `fibonacci[n1 : n2]`.

Première méthode : `to_list`

La classe de notre annihilateur propose `to_list`, qui calcule récursivement tous les termes jusqu'à n .

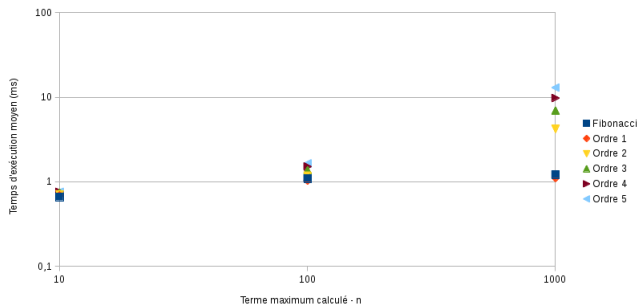
- Facile à implémenter
- Mais lent

Deuxième méthode : `forward_matrix_bsplint`

Utiliser l'algèbre linéaire pour calculer directement le terme n

- Théoriquement plus rapide
- Souffre d'un temps d'amorce

Nous avons comparé les temps d'exécution pour les deux méthodes.



Rapport du temps avec `to_list` sur le temps avec `forward_matrix_bsplitted`.

La seconde méthode est déjà plus efficace pour des valeurs de l'ordre de 100.

l'addition

- Trouver un operateur qui annule les deux termes
- Trouver de nouvelle conditions initials

Surcharger la multiplication

- La multiplication \times correspond à l'opérateur `__mul__` en Python
- Utilisation de `symmetric_product`

Utiliser les facteurs

$$\text{const : } C_{n+1} - C_n = 0, \quad E_0 = 2,$$

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

$$\text{Leur Somme : } S_{n+3} - 2 * S_{n+2} + S_n = 0$$

$$S_0 = 2, S_1 = 3, S_2 = 3$$

l'annihilateur annule les deux suites

1, 1, 2, 3, 5, 8, 13, 21, 34,...

2, 2, 2, 2, 2, 2, 2, 2, ..

Addition et multiplication - Procurer des conditions supplémentaires

Termes dégénérés dans les sommes/produits

Le coefficient dominant de certains produits ou sommes ont des racines dans \mathbb{Z} .

Utiliser les facteurs

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

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

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

D_4 est dégénéré, mais E_4 et F_4 existent.

TODO