

Outils de combinatoire analytique en Sage

Matthieu Dien, Marguerite Zamansky
encadrés par Antoine Genitrini et Frederic Peschanski

14 mai 2013

Table des matières

1	Combinatoire analytique	3
1.1	Quelques bases de combinatoire analytique	3
1.2	Opérateurs élémentaires	4
1.3	Spécifications	6
1.4	Fonctions génératrices multivariées	6
2	Sage	10
2.1	Introduction	10
2.1.1	Présentation	10
2.1.2	Architecture	11
2.2	Implémentation	12
2.2.1	LazyPowerSeries	12
2.2.2	Générateurs	12
2.2.3	Représentation des séries	13
2.3	Opérateurs	14
2.3.1	Addition	14
2.3.2	Multiplication	14
2.3.3	Séquence	14
2.3.4	Composition	16
2.3.5	Dérivation	17

Introduction

La combinatoire analytique est un domaine des mathématiques utilisée en informatique notamment pour calculer des complexités d'algorithmes, générer des données aléatoires ... Actuellement, il existe peu de logiciels permettant de travailler avec cet outil mathématique et la plupart sont propriétaires. C'est notamment le cas de Maple pour lequel il existe une librairie : Gfun. L'objectif de ce projet était donc d'étudier le portage de Gfun pour Sage, un logiciel libre de calcul formel, symbolique et numérique, puis d'en réaliser un prototype. Un autre objectif était notamment de généraliser le fonctionnement de Gfun qui ne peut travailler qu'avec une seule variable alors que la plupart des problèmes actuels sont de plusieurs variables.

Dans un premier temps nous reviendrons sur quelques rappels mathématiques de ce qu'est la combinatoire analytique et nous présenterons quelques exemples de problèmes à une variable et à plusieurs variables. Dans un second temps nous étudierons les différents moyens techniques nécessaires à la réalisation de l'implémentation puis son fonctionnement pour finir sur une description des algorithmes utilisés. Enfin, nous concluerons en présentant l'ensemble du travail accompli et celui qu'il reste à faire pour obtenir une version au moins aussi complète que Gfun.

Chapitre 1

Combinatoire analytique

1.1 Quelques bases de combinatoire analytique

La combinatoire est une branche des mathématiques qui s'intéresse à l'étude des structures finies ou dénombrables : des objets qui peuvent être créés par un nombre fini de règles. On peut s'intéresser aux propriétés de ces objets combinatoires, mais souvent on voudra les compter. Pour ça la méthode classique utilise des raisonnements par récurrence, souvent compliqués, parfois inefficaces. La combinatoire analytique a pour but de décrire de manière quantitative ces structures combinatoires en utilisant des outils analytiques. La pierre d'angle de cette théorie est la *fonction génératrice*. Nous commencerons par donner les définitions de quelques notions de base de la combinatoire analytique.

Définition Une classe combinatoire est un ensemble fini ou dénombrable sur lequel est défini une fonction taille qui vérifie les conditions suivantes :

- la taille d'un élément est un entier positif,
- il y a un nombre fini d'éléments de chaque taille.

Définition La suite de comptage d'une classe combinatoire \mathcal{A} est la suite $(A_n)_{n \geq 0}$ où A_n est le nombre d'objets de taille n dans \mathcal{A} .

Définition La série génératrice ordinaire d'une suite (A_n) est la série entière

$$A(z) = \sum_{n=0}^{\infty} A_n z^n.$$

La série génératrice ordinaire d'une classe combinatoire est la série génératrice ordinaire de sa suite de comptage. De manière équivalente, le série génératrice d'une classe combinatoire \mathcal{A} peut s'écrire sous la forme

$$A(z) = \sum_{a \in \mathcal{A}} z^{|a|}.$$

Nous attirons l'attention sur le fait qu'une série génératrice ordinaire est vue en tant que série formelle, et non comme une série entière, on peut la manipuler sans se soucier des problèmes de convergence.

Souvent, pour dénombrer les objets, on les décompose en éléments plus simples, mais du même type, pour obtenir une équation de récurrence vérifiée par les A_n . Cette équation peut-être plus ou moins simple à résoudre. L'approche proposée par la combinatoire analytique[?], repose sur la vision d'une classe combinatoire comme une construction à partir d'autres classes combinatoires. On définit les constructions admissibles, qui produisent une classe combinatoire.

Définition Soient, $\mathcal{A}, \mathcal{B}^1, \dots, \mathcal{B}^m$ des classes combinatoires, et Φ une construction,

$$\mathcal{A} = \Phi(\mathcal{B}^1, \dots, \mathcal{B}^m).$$

La construction Φ est admissible si et seulement si la suite de comptage (A_n) de \mathcal{A} ne dépend que des suites de comptages $(B_n^1), \dots, (B_n^m)$ de $\mathcal{B}^1, \dots, \mathcal{B}^m$.

Pour chaque construction admissible Φ , il existe un opérateur Ψ , agissant sur les séries génératrices ordinaires : si

$$\mathcal{A} = \Phi(\mathcal{B}^1, \dots, \mathcal{B}^m),$$

alors,

$$A(z) = \Psi(B^1(z), \dots, B^m(z)).$$

L'existence de cet opérateur va nous permettre d'utiliser la décomposition en classes combinatoires élémentaires pour obtenir les séries génératrices ordinaires de classes combinatoires plus compliquées.

1.2 Opérateurs élémentaires

Les constructions sur les classes combinatoires sont bâties à partir d'opérateurs élémentaires. Il n'y a pas d'ensemble canonique d'opérateurs élémentaires, et plus ils sont nombreux plus les classes combinatoires que l'on peut décrire sont complexes. Nous ne présentons ici que les trois opérateurs les plus simples : le produit cartésien, la somme et la séquence, car ce sont les trois sans lesquels on ne peut rien faire, et ce sont les trois que nous avons implémentés.

Premièrement, on se donne deux opérateurs d'arité zéro, \mathcal{E} et \mathcal{Z} , que l'on interprètera comme deux classes combinatoires particulières. \mathcal{E} sera interprété comme la classe combinatoire neutre, composée d'un unique élément de taille nulle. \mathcal{Z} sera interprété comme la classe combinatoire atomique, composée d'un unique élément de taille 1. Ces deux classes ont pour séries génératrices $E(z) = 1$ et $Z(z) = z$ respectivement.

Produit cartésien

La construction produit cartésien appliquée à deux classes combinatoires \mathcal{B} et \mathcal{C} donne la classe combinatoire

$$\mathcal{A} = \{\alpha = (\beta, \gamma) \mid \beta \in \mathcal{B}, \gamma \in \mathcal{C}\}$$

munie de la fonction taille définie par

$$|\alpha|_{\mathcal{A}} = |\beta|_{\mathcal{B}} + |\gamma|_{\mathcal{C}}.$$

On écrit alors $\mathcal{A} = \mathcal{B} \times \mathcal{C}$. En regardant toutes les possibilités de paires, on trouve que la suite de comptage de \mathcal{A} est donnée par le produit de convolution des suites de comptage de \mathcal{B} et \mathcal{C} :

$$A_n = \sum_{k=0}^n B_k C_{n-k}.$$

On remarque bien sûr que le produit cartésien est une construction admissible, et on reconnaît dans cette égalité le produit de deux séries entières. Ainsi la série génératrice de \mathcal{A} est

$$A(z) = B(z) \cdot C(z).$$

Somme combinatoire

La somme combinatoire de deux classes \mathcal{B} et \mathcal{C} est définie de manière à avoir les mêmes propriétés que la somme disjointe, mais sans avoir à imposer la disjonction des ensembles. Pour cela, on se donne deux objets neutres distincts, i.e. de taille nulle, \square et \diamond et on pose

$$\mathcal{B} + \mathcal{C} := (\{\square\} \times \mathcal{B}) \cup (\{\diamond\} \times \mathcal{C}).$$

Comme \square et \diamond sont distincts, le membre de droite est une union disjointe, pour toutes classes combinatoires \mathcal{B} et \mathcal{C} . Il y a autant d'objets de taille n dans $\{\square\} \times \mathcal{B}$ et $\{\diamond\} \times \mathcal{C}$ que dans \mathcal{B} et \mathcal{C} , et par conséquent, si $\mathcal{A} = \mathcal{B} + \mathcal{C}$, on a

$$A_n = B_n + C_n,$$

et ce quel que soit n , la somme combinatoire est donc une construction admissible. La série génératrice de \mathcal{A} est

$$A(z) = B(z) + C(z).$$

Séquence

Soit \mathcal{B} une classe combinatoire, on définit **SEQ**(\mathcal{B}) comme la somme combinatoire infinie

$$\mathbf{SEQ}(\mathcal{B}) := \mathcal{E} + \mathcal{B} + (\mathcal{B} \times \mathcal{B}) + (\mathcal{B} \times \mathcal{B} \times \mathcal{B}) + \dots$$

Ce qui est équivalent à voir la séquence comme l'ensemble des suites d'objets de \mathcal{B} :

$$\mathbf{SEQ}(\mathcal{B}) = \{(\beta_1, \dots, \beta_l) \mid \beta_i \in \mathcal{B}, l \geq 0\}.$$

En combinant les fonctions tailles associées au produit cartésien et à la somme combinatoire, on définit la taille dans **SEQ**(\mathcal{B}) :

$$|(\beta_1, \dots, \beta_l)|_{\mathbf{SEQ}(\mathcal{B})} = |\beta_1|_{\mathcal{B}} + \dots + |\beta_l|_{\mathcal{B}}$$

SEQ(\mathcal{B}) est une classe combinatoire si et seulement si, \mathcal{B} ne contient pas d'objet de taille nulle. En effet, s'il existe $\beta_0 \in \mathcal{B}$ de taille nulle, on peut créer une infinité de suites de β_0 , toutes de taille nulle, **SEQ**(\mathcal{B}) n'est donc pas une classe combinatoire.

Réciproquement, s'il n'existe pas d'objet de taille nulle dans \mathcal{B} , pour tout $n \geq 1$, il existe un nombre fini de décompositions en entiers de n , c'est à dire un nombre fini de suites d'entiers dont la somme est n , et donc un nombre fini d'antécédents de n par la fonction taille de **SEQ**(\mathcal{B}). Et la suite vide est le seul antécédent de 0. **SEQ**(\mathcal{B}) est donc une classe combinatoire, et la séquence est une construction admissible.

Pour calculer la série génératrice de $\mathcal{A} = \mathbf{SEQ}(\mathcal{B})$, on utilise les propriétés vues avec le produit cartésien et la somme combinatoire :

$$A(z) = 1 + B(z) + B(z) \cdot B(z) + B(z) \cdot B(z) \cdot B(z) \dots$$

$$A(z) = 1 + B(z) + B(z)^2 + B(z)^3 + \dots$$

$$A(z) = \frac{1}{1 - B(z)}$$

1.3 Spécifications

Définition Une spécification pour un n -uplet de classes combinatoires $(\mathcal{A}_1, \mathcal{A}_2, \dots, \mathcal{A}_n)$, est un système de n équations

$$\begin{cases} \mathcal{A}_1 = \Phi_1(\mathcal{A}_1, \mathcal{A}_2, \dots, \mathcal{A}_n) \\ \mathcal{A}_2 = \Phi_2(\mathcal{A}_1, \mathcal{A}_2, \dots, \mathcal{A}_n) \\ \vdots \\ \mathcal{A}_n = \Phi_n(\mathcal{A}_1, \mathcal{A}_2, \dots, \mathcal{A}_n) \end{cases}$$

où Φ_i est une construction admissible, utilisant les opérateurs du produit cartésien, de la somme combinatoire, ou de la séquence sur les classes $\mathcal{A}_1, \mathcal{A}_2, \dots, \mathcal{A}_n$, \mathcal{E} et \mathcal{Z} .

Exemple

Un arbre plan enraciné, ou arbre de Catalan, est composé soit d'une feuille, soit d'un nœud interne et d'un certain nombre de sous-arbres, mais la spécification d'un arbre enraciné dépendra du critère que l'on choisit pour la taille. Si l'on choisit que la taille d'un arbre est le nombre de ses feuilles, on obtient la spécification

$$\mathcal{A} = \mathcal{Z} + \text{SEQ}(\mathcal{A}).$$

Mais si on décide que la taille d'un arbre est le nombre de tous ces nœuds, internes et externes, on trouve la spécification

$$\mathcal{A} = \mathcal{Z} \times \text{SEQ}(\mathcal{A}).$$

Ces deux spécifications s'interprètent en deux équations fonctionnelles différentes :

pour le premier cas $A(z) = z + \frac{1}{1 - A(z)}$,

et $A(z) = \frac{z}{1 - A(z)}$ dans le second cas.

1.4 Fonctions génératrices multivariées

Jusque là, nous n'avons étudié les classes combinatoires que selon un seul critère, la taille des objets qu'elles contiennent, mais souvent, il est intéressant de pouvoir prendre en compte plusieurs critères. Par exemple dans les arbres de Catalan, on pourrait vouloir connaître le nombre d'arbre de taille 5 ayant 3 nœuds internes, ou encore prendre en compte aussi le nombre de nœuds binaires. Pour utiliser la méthode vue précédemment sur plusieurs paramètres, nous allons devoir définir précisément ces paramètres, et énoncer certaines propriétés qu'ils devront vérifier.

Définition Soit \mathcal{A} une classe combinatoire, un paramètre d -dimensionnel est une fonction

$$\chi : \mathcal{A} \mapsto \mathbb{N}^d.$$

La suite de comptage de \mathcal{A} selon la fonction taille et le paramètre χ est alors définie par :

$$A_{n,k_1,\dots,k_d} = \text{card} \{ \alpha \in \mathcal{A} \text{ tel que } |\alpha|_{\mathcal{A}} = n, \chi_1(\alpha) = k_1, \dots, \chi_d(\alpha) = k_d \}$$

On note \mathcal{A}_χ une classe combinatoire \mathcal{A} dotée d'un paramètre χ .

Pour simplifier l'écriture et la lecture on introduit la notation suivante, pour \mathbf{x} un vecteurs de d variables et \mathbf{k} un vecteur de \mathbb{N}^d ,

$$\mathbf{x}^{\mathbf{k}} = x_1^{k_1} x_2^{k_2} \cdots x_d^{k_d}.$$

On peut maintenant définir la fonction génératrice de cette nouvelle suite de comptage.

Définition Soit $A_{n,\mathbf{k}}$ une suite sur plusieurs index, où \mathbf{k} est un vecteur de \mathbb{N}^d , la série génératrice ordinaire de cette suite est

$$A(z, \mathbf{u}) = \sum_{n,\mathbf{k}} A_{n,\mathbf{k}} z^n \mathbf{u}^{\mathbf{k}}.$$

Soient \mathcal{A} une classe combinatoire et χ un paramètre, la série génératrice ordinaire de (\mathcal{A}, χ) , est comme auparavant la série génératrice de la suite de comptage associée :

$$A(z, \mathbf{u}) = \sum_{\alpha \in \mathcal{A}} z^{|\alpha|} \mathbf{u}^{\chi(\alpha)}.$$

Avec ces définitions tellement semblables à celles vues tout au début, on aimerait pouvoir construire les classes combinatoires dotées de paramètres par la même méthode qu'avec les classes combinatoires simples. Les spécifications fonctionneront de même, les trois constructions élémentaires que nous avons définies s'appliqueront de la même façon, mais il faut s'assurer que les paramètres sont compatibles entre eux, comme il fallait s'assurer quand on formait la classe \mathcal{A} à partir des classes \mathcal{B} et \mathcal{C} que la suite de comptage A_n ne dépendait que de B_n et C_n .

Définition Soient \mathcal{A}_χ , \mathcal{B}_ξ et \mathcal{C}_ζ des classes combinatoires dotées de paramètres tous trois de même dimension,

Produit cartésien Si $\mathcal{A} = \mathcal{B} \times \mathcal{C}$, χ est hérité de ξ et ζ si et seulement si $\chi(\beta, \gamma) = \xi(\beta) + \zeta(\gamma)$.

Somme combinatoire Si $\mathcal{A} = \mathcal{B} + \mathcal{C}$, χ est hérité de ξ et ζ si et seulement si

$$\chi(\omega) = \begin{cases} \xi(\omega) & \text{si } \omega \in \mathcal{B} \\ \zeta(\omega) & \text{si } \omega \in \mathcal{C} \end{cases}.$$

Séquence Si $\mathcal{A} = \text{SEQ}(\mathcal{B})$, χ est hérité de ξ si et seulement si

$$\chi(\beta_1, \dots, \beta_i) = \xi(\beta_1) + \xi(\beta_2) + \cdots + \xi(\beta_i).$$

Il apparaît donc que l'héritage est une extension de la condition que devait vérifier la taille pour qu'une construction soit admissible. On voit dans ce qui précède que la taille ne joue plus de rôle particulier, et on peut voir la taille comme une composante du paramètre dont une classe combinatoire est dotée. Ainsi, la série génératrice d'une classe \mathcal{A}_χ devient simplement

$$A(\mathbf{z}) = \sum_{\alpha \in \mathcal{A}} \mathbf{z}^{\chi(\alpha)}.$$

Les règles d'héritages pour les paramètres étant similaires aux règles de dépendance des tailles, les opérateurs élémentaires sont toujours valides, et les opérateurs agissant sur les séries génératrices sont inchangés, à

ceci près qu'ils s'appliquent maintenant sur des séries génératrices multivariées.

$$\text{Produit cartésien : } \mathcal{A} = \mathcal{B} \times \mathcal{C} \Rightarrow A(\mathbf{z}) = B(\mathbf{z}) \cdot C(\mathbf{z})$$








$$\text{Somme combinatoire : } \mathcal{A} = \mathcal{B} + \mathcal{C} \Rightarrow A(\mathbf{z}) = B(\mathbf{z}) + C(\mathbf{z})$$

$$\text{Séquence : } \mathcal{A} = \text{SEQ}(\mathcal{B}) \Rightarrow A(\mathbf{z}) = \frac{1}{1 - B(\mathbf{z})}$$

Exemple : les arbres de Catalan

Revenons à l'exemple des arbres de Catalan. Maintenant, nous savons prendre en compte plusieurs paramètres. En plus de la taille, en nombre de nœuds, nous nous intéressons également à la longueur de cheminement cumulée, c'est à dire, la somme des profondeurs de tous les nœuds de l'arbre.

Un arbre est toujours composé d'un nœud duquel pendent un certain nombre de sous-arbres (ce nombre est possiblement nul, l'arbre est alors réduit à une feuille).

longueur de cheminement :	0	1	2	3	4
taille 1			\emptyset		\emptyset
taille 2	\emptyset	\emptyset		\emptyset	
taille 3	\emptyset	\emptyset	\emptyset		\emptyset
taille 4	\emptyset	\emptyset	\emptyset	\emptyset	

Notons C la classe combinatoire des arbres de Catalan, et λ le paramètre qui représente la longueur de cheminement d'un arbre $\tau \in C$. On voit que la longueur de cheminement de τ est la somme des tailles de tous les sous-arbres de τ . Et de là, c'est aussi la longueur de cheminement des sous-arbres pendant à la racine de τ , plus la taille de ces sous-arbres. Soit V , l'ensemble de ces sous-arbres,

$$\lambda(\tau) = \sum_{v \in V} \lambda(v) + |v|.$$

Introduisons maintenant le paramètre $\mu(\tau) = |\tau| + \lambda(\tau)$.

On obtient ainsi la spécification

$$C_{n,\lambda} = \text{SEQ}(C_{n,\mu}) \times \mathcal{Z}.$$

L'interprétation en équation fonctionnelle n'est pas évidente, à cause des deux paramètres λ et μ . Il faut une petite manipulation. Nous avons d'ailleurs choisi la notation où l'on distingue la taille de l'objet, n , du paramètre pour plus de facilité dans cette manipulation. On a vu que

$$C_{n,\mu}(u, z) = \sum_{\tau \in C} z^{|\tau|} u^{\mu(\tau)},$$

et donc




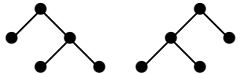
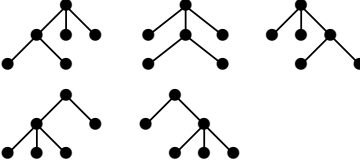
$$C_{n,\mu}(u, z) = \sum_{\tau \in C} z^{|\tau|} u^{|\tau|} u^{\lambda(\tau)} = C_{n,\lambda}(zu, u).$$

Et enfin, on a l'équation fonctionnelle

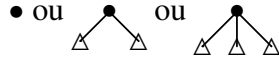
$$C_{n,\lambda}(z, u) = \frac{z}{1 - C_{n,\lambda}(zu, u)}.$$

Un autre exemple : les arbres binaires-ternaires

Un arbre binaire-ternaire est un arbre dont les nœuds internes ont deux ou trois fils :

taille 1	
taille 2	\emptyset
taille 3	
taille 4	
taille 5	
taille 6	

Un arbre binaire ternaire est soit une feuille, soit un nœud binaire avec deux arbres pendants, soit un nœud ternaire avec trois arbres pendants :



En appliquant ce qu'on n'a vu précédemment, on voit qu'un arbre binaire-ternaire est la somme de trois classes combinatoires : la classe composée des feuilles, la classe composée des nœuds binaires et des sous arbres pendants à ces nœuds et la classe composée des nœuds ternaires et des sous arbres. Ces deux dernières classes sont elles-mêmes le produit cartésien de classes : le produit des nœuds binaires et des paires d'arbres binaires-ternaires, et le produit des nœuds ternaires et des triplets d'arbres binaires-ternaires. On obtient ainsi la spécification suivante

$$\mathcal{BT} = \mathcal{Z} + \mathcal{Z} \times \mathcal{BT}^2 + \mathcal{Z} \times \mathcal{BT}^3,$$

qui s'interprète en l'équation fonctionnelle

$$BT(z, u, v) = z + uBT(z, u, v)^2 + vBT(z, u, v)^3,$$

où l'on note

$$\begin{cases} z : \text{le nombre de feuilles} \\ u : \text{le nombre de nœuds binaires} \\ v : \text{le nombre de nœuds ternaires.} \end{cases}$$

On pourrait aussi vouloir prendre en compte le nombre total de nœuds, w , on obtient alors l'équation fonctionnelle

$$BT(z, u, v, w) = zw + uwBT(z, u, v, w)^2 + vwBT(z, u, v, w)^3.$$

Chapitre 2

Sage

2.1 Introduction

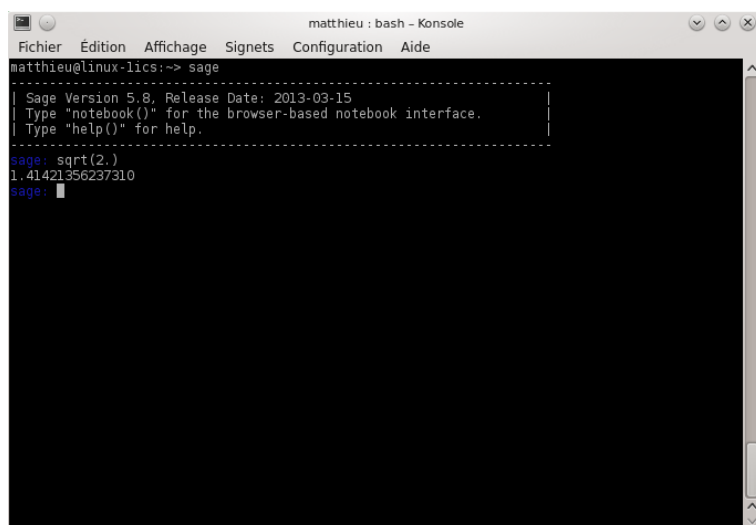
2.1.1 Présentation

Sage est un logiciel libre (sous licence GPL) de calcul formel, symbolique et numérique [S⁺13]. Il est principalement écrit en C/C++ et Python/Cython/Pyrex. Le but du projet Sage est de fournir une alternative aux solutions de calculs propriétaires comme Maple, Mathematica, Matlab ...

Historiquement Sage est un regroupement de différents projets libres autour d'une interface Python unifiée, pour faciliter les traitements entre ces *briques de base* (Singular, Maxima, GP/PARI, GAP). Mais de plus en plus le projet se tourne vers le développement de ses propres paquets pour acquérir son indépendance, et donc améliorer la cohérence de l'architecture, le calendrier de release et faciliter ainsi le développement de nouvelles fonctionnalités et la maintenance d'anciennes.

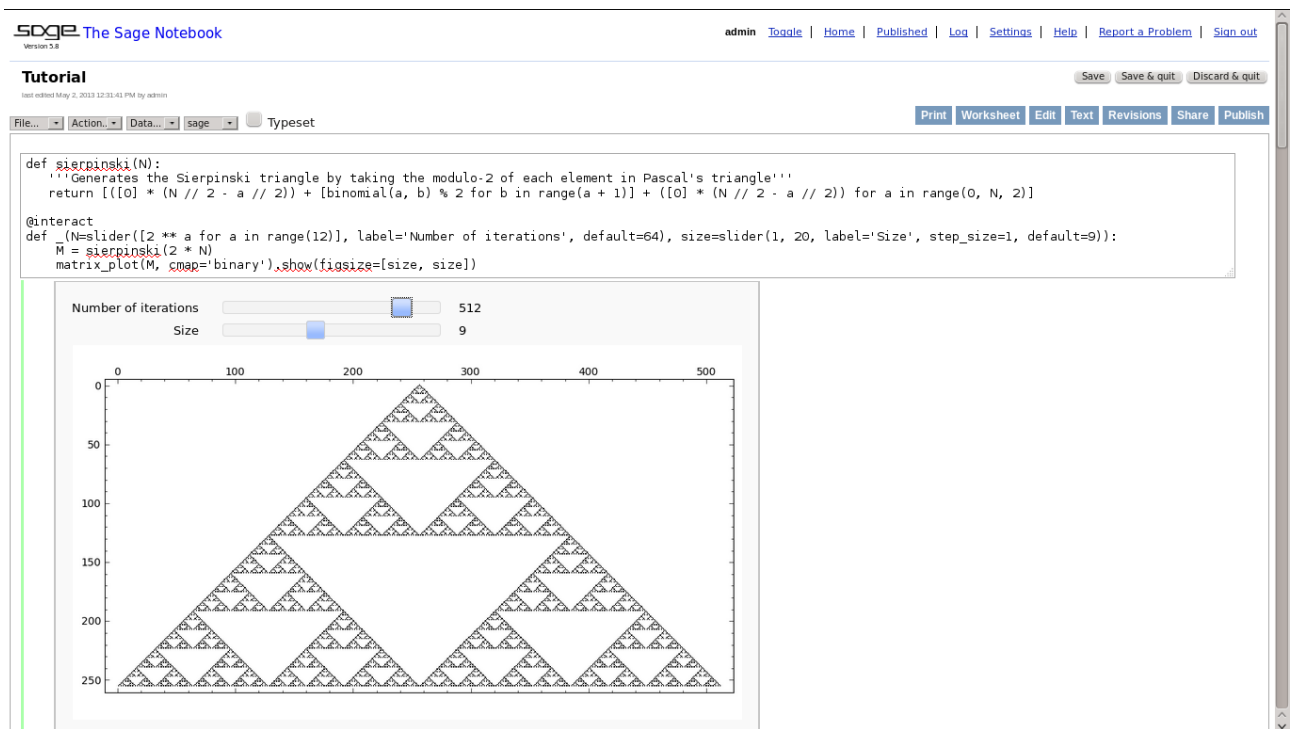
Dans Sage, deux interfaces sont disponibles :

- L'interface ligne de commande. Elle se présente comme un top-level python classique :



```
matthieu@linux-lics:~$ sage
-----
| Sage Version 5.8, Release Date: 2013-03-15
| Type "notebook()" for the browser-based notebook interface.
| Type "help()" for help.
-----
sage: sqrt(2)
1.41421356237310
sage: 
```

- L'interface bloc-note (ou notebook). Elle se présente sous la forme d'une interface embarquée dans un navigateur web, comme ceci :



Le langage permettant de manipuler les outils Sage est un Python avec quelques sucres syntaxiques supplémentaires permettant une manipulation plus aisée des concepts mathématiques utilisés. Par exemple, pour déclarer l'anneau des polynômes de variables x, y et z , à coefficients dans \mathbb{Q} : $R.<x, y, z> = \text{PolynomialRing}(\mathbb{Q})$

2.1.2 Architecture

Sage ayant pour objectif de couvrir l'ensemble des besoins en calcul pour tous les domaines des mathématiques, la taille du code est donc assez conséquente (un peu plus de 11 milliards de lignes de code sans compter les paquets supplémentaires). Cela a été une première difficulté dans la réalisation de ce projet.

Le code de Sage s'organise autour d'une hiérarchie de dossiers représentant les différents domaines mathématiques : *combinat* pour la combinatoire, *algebras* pour l'algèbre, *games* pour la théorie des jeux, *probability*, *graphs*, etc.

De plus, la fonctionnalité d'aide de Sage permet d'afficher le fichier source d'un objet Sage et son emplacement :

```
PolynomialRing?      #affichera la documentation
PolynomialRing??     #affichera le fichier source et son emplacement
```

Enfin, toute la documentation et les tests de Sage sont contenus directement dans le code source grâce au mécanisme des *docstring* Python : la documentation d'une fonction est écrite après son prototype dans le fichier source, par exemple

```
def une_fonction():
    """
    Ceci est la docstring de une_fonction
    une_fonction renvoie 1

    Test test directement dans la fonction comme dans Sage::

        sage : une_fonction()
        1
    """

    return 1
```

Cela permet une plus grande facilité pour la compréhension du code et le maintien à jour du code comme de la documentation et des tests.

2.2 Implémentation

2.2.1 LazyPowerSeries

Sage contenait déjà une implémentation des séries génératrices monovariées (basée sur le travail fait sur Aldor [HR06]) sous la forme d'une classe Python. L'implémentation se présente donc sous la forme de deux classes `FormalMultivariatePowerSeriesRing` et

`FormalMultivariatePowerSeries` héritant de `LazyPowerSeriesRing` et `LazyPowerSeries`. Les opérations supportées par les séries existantes sont l'addition, la multiplication, la composition, la dérivation et l'intégration (la primitive en fait). Pour nos séries nous supportons l'ensemble de ces opérations, exceptée la primitivation, et nous y avons ajouté la séquence.

Les classes `FormalMultivariatePowerSeriesRing` et `LazyPowerSeriesRing` sont nécessaires à Sage et permettent de garantir certaines fonctionnalités communes à tous les anneaux notamment les sucres syntaxiques vus précédemment.

De plus, comme son nom l'indique, l'implémentation existante était paresseuse, ce qui est nécessaire car les objets manipulés sont des séries donc potentiellement non finies (et c'est généralement le cas). Nous avons évidemment repris ce style de programmation. Cette *paresse* est obtenu en utilisant le concept de générateur python, que nous présenterons avant de présenter les opérations implémentées.

2.2.2 Générateurs

Les générateurs Python [SPH01], sont un mécanisme permettant de créer des objets itérables. Ils sont déclarés sous forme de fonction classique contenant le mot clé `yield`. A chaque appel de la méthode `next` associé à ce générateur, le corps de la *fonction* est exécuté jusqu'à rencontrer un `yield`, et l'argument donné à `yield` est retourné par `next` et l'exécution du corps se stoppe. Au prochain appel de `next`, l'exécution du corps reprend jusqu'au prochain `yield`.

Par exemple, si nous voulions itérer sur l'ensemble des entiers naturels :

```

def integers_definition():
    i = 0
    while True :
        yield i
        i += 1

integers = integers_definition()

while True :
    n = integers.next()
    if n % 2 == 0:
        print ("%d est pair"%n)
    else :
        print ("%d est impair"%n)

# ou

for n in integers_definition():
    if n % 2 == 0:
        print ("%d est pair"%n)
    else :
        print ("%d est impair"%n)

```

L'avantage des générateurs est qu'ils permettent d'effectuer les calculs seulement quand cela est nécessaire, c'est ce qui permet d'obtenir la *paresse* : chaque coefficient de la série ne sera calculé que quand il sera nécessaire.

2.2.3 Représentation des séries

Dans l'implémentation existante, les séries monovariées sont représentées par des *stream*. Un *stream* peut être vu comme une liste infinie où les éléments stockés sont les éléments dont on a explicitement demandé la valeur, qui ont été calculés. Les coefficients de la série sont rangés dans l'ordre croissant de l'ordre de leur monôme associé.

Par exemple, si nous définissons F comme étant la série génératrice monovariée dont les coefficients sont les nombres de Fibonacci (définis par $f_0 = 1$, $f_1 = 1$, $f_{n+2} = f_{n+1} + f_n$), alors nous pouvons illustrer sa représentation dans le tableau suivant :

entrée	sortie	mémoire
F	Uninitialized lazy power series	$[]$
$F.coefficients(2); F$	$1+x+O(x^2)$	$[1, 1]$
$F.coefficients(5); F$	$1 + x + 2x^2 + 3x^3 + 5x^4 + O(x^5)$	$[1, 1, 2, 3, 5]$

On voit alors comment sont manipulées les séries : le générateur permet le calcul des coefficients tandis que la *stream* permet le stockage des coefficients et la demande de nouveaux coefficients. C'est ce fonctionnement que nous avons gardé dans le cas des séries multivariées.

Dans le cas des séries génératrices multivariées, le problème est comment ranger nos coefficients dans un *stream* ? La première idée est de garder un ordre sur les monômes de plusieurs variables en prenant la somme

de l'ordre de chaque variable : $x^{11}y < x^4y^2z^8$ ($11 + 1 < 2 + 4 + 8$). Ainsi chaque case du *stream* contiendra une liste de coefficients de même ordre. Il faut aussi garder l'association entre chaque coefficient et son monôme, on stockera donc des couples coefficient/monôme. Les monômes seront représentés par des listes d'entiers contenant les puissances de chaque variable (dans un ordre fixé), par exemple : $x^i y^j z^k \approx [i, j, k]$.

Ainsi sur l'exemple des arbres binaires-ternaires de la première partie ($BT(z, u, v) = z + uBT(z, u, v)^2 + vBT(z, u, v)^3$) on a le tableau suivant :

entrée	sortie	mémoire
F	Uninitialized formal multivariate power series	[]
F.coefficients(2); F	$z + \dots$	[[], [(1, [1, 0, 0])]]
F.coefficients(5); F	$z + z^2 * u + z^3 * v + 2 * z^3 * u^2 + 5 * z^4 * u * v + \dots$	[[], [(1, [1, 0, 0])], [], [(1, [2, 1, 0])], [(1, [3, 0, 1])], [(2, [3, 2, 0])], [(5, [4, 1, 1])]]

On remarquera qu'une liste vide correspond à un terme nul. Maintenant la représentation de nos séries et le concept de générateur expliqués passons aux détails du mécanisme.

2.3 Opérateurs

Chaque opérateur correspond à deux méthodes de la classe `FormalMultivariatePowerSeries`. Une première méthode qui est en fait un générateur et qui calculera les coefficients de la série obtenue. Une deuxième méthode qui à partir du générateur crée une nouvelle instance pour la série calculée, cette méthode est totalement héritée de `LazyPowerSeries`.

Les opérateurs ont été implémentés de façon naïve, ce qui n'a pas empêché des difficultés sur la plupart d'entre eux.

2.3.1 Addition

Pour l'addition, l'algorithme est simple et correspond plus ou moins à une concaténation de deux listes en faisant attention aux termes qui ont le même monôme :

2.3.2 Multiplication

La multiplication est similaire à l'addition sauf que la construction de `new_list` n'est plus une simple concaténation mais un produit de convolution :

2.3.3 Séquence

Pour l'opérateur **SEQ** il y a deux façons de le concevoir :

- comme l'opérateur pseudo-inverse $\mathbf{SEQ}(F) \longrightarrow \frac{1}{1-F}$

Input : two series A and B

```
n ← 0
while True do
  new_list ← A.stream[n].copy() foreach (coeff,monom) in B.stream[n] do
    foreach (coeff',monom') in new_list do
      if monom == monom' then
        if coeff+coeff' == 0 then
          delete (coeff',monom') of new_list
          break
        else
          replace (coeff',monom') by (coeff'+coeff,monom') in new_list
          break
        end
      end
    end
    append (coeff,monom) to new_list
  end
  end
  n ← n + 1 yield new_list
end
```

Algorithme 1: add two series

Input : two series A and B

```
n ← 0
while True do
  for k ← 0 to n do
    new_list ← []
    foreach (coeff,monom) in A.stream[k] do
      foreach (coeff',monom') in B.stream[n-k] do
        append (coeff * coeff', monom + monom') to new_list
      end
    end
  end
  foreach (coeff,monom) and (coeff',monom') in new_list such that monom == monom' do
    if coeff+coeff' == 0 then
      delete (coeff',monom') and (coeff,monom) of new_list
    else
      replace (coeff',monom') by (coeff'+coeff,monom') and delete (coeff,monom) in new_list
    end
  end
  n ← n + 1
  yield new_list
end
```

Algorithme 2: multiply two series

- comme la fonction récursive $\text{SEQ}(F) \rightarrow 1 + F \cdot \text{SEQ}(F)$

La deuxième manière de voir **SEQ** semble bien sûr plus évidente à implémenter car plus simple et surtout plus stable numériquement. On ajoute d'ailleurs un champ `_pows` à notre classe, qui est un *stream* de `FormalMultivariatePowerSeries` et qui a pour générateur une simple fonction anonyme. Nous n'avons donc pas à recalculer les puissances de notre série à chaque itération, ce qui peut aussi servir pour la composition.

Input : one serie A with first term equal to 0

```

n ← 0
while True do
  new_list ← []
  for k ← 0 to n do
    | append A._pows[k].stream[n] to new_list
  end
  foreach (coeff, monom) and (coeff', monom') in new_list such that monom == monom' do
    | if coeff+coeff' == 0 then
    |   | delete (coeff', monom') and (coeff, monom) of new_list
    | else
    |   | replace (coeff', monom') by (coeff'+coeff, monom') and delete (coeff, monom) in new_list
    | end
  end
  n ← n + 1
  yield new_list
end

```

Algorithme 3: SEQ of a serie

2.3.4 Composition

La composition reprend en fait les opérations décrites précédemment. On construit une nouvelle série en remplaçant simplement la variable voulue par la série à composer :

Input : one series A and list L of series/variable of length the number of variable
every series of the list must have the first term equal to 0

```

n ← 0
new_serie ← first_term(A)
while True do
  foreach (coeff, powers) in A.coefficient(n) do
    | new_serie ← new_serie + coeff · Lpowers
  end
  n ← n + 1
  yield new_serie.coefficient(n)
end

```

Algorithme 4: composition of series

Précisons que la méthode `coefficient` renvoie la liste des coefficients en n^{ime} position et que L^{powers} est implémentée avec la méthode `map`.

2.3.5 Dérivation

La dérivation quant à elle est assez simple, il suffit de multiplier chaque coefficient par la puissance de la variable selon laquelle on dérive et de déplacer ce couple un rang avant :

Input : one serie A and p the position of a variable

```
n ← 1
while True do
  new_list ← []
  foreach (coeff, powers) in A._stream[n] do
    powers[p] ← powers[p] - 1
    if powers[p] ≥ 0 then
      | append (coeff * (powers[p] + 1), powers) to new_list
    end
  end
  n ← n + 1
  yield new_list
end
```

Algorithme 5: derivate one serie

Conclusion

Pour ce projet, nous avons certes dû étudier et maîtriser un nouvel outil mathématique, mais le principal défi a été l'implémentation de cet outil pour l'intégrer au logiciel Sage. Il nous a fallu d'une part trouver une représentation utilisable et efficace des objets manipulés, et pour ce faire, nous avons utilisé des concepts de programmation nouveaux pour nous, et des traits propres au langage Python. D'autre part, il fallait s'intégrer à un projet de grande ampleur, le développement de Sage, se familiariser avec les normes et les respecter pour pouvoir faire fonctionner les outils que nous avons créés. Au cours de ce travail de lecture de code, nous avons d'ailleurs repéré un bug dans un paquet existant, une erreur dans le calcul de l'ordre d'une série. Nous avons corrigé ce bug et soumis un patch. Ces difficultés une fois surmontées, il est très satisfaisant de voir notre travail utilisable directement dans Sage, et nous avons soumis notre paquet pour qu'il puisse être ajouté à une prochaine version de Sage, et que d'autres personnes puissent l'utiliser. Il y a bien sûr encore du travail, d'autres opérateurs, d'autres fonctionnalités à ajouter pour augmenter cette nouvelle librairie `multivariate_series`, et éventuellement proposer une alternative à `Gfun`. Notre travail permet des calculs sur les séries que ne permet pas `Gfun`, mais n'a pas encore intégré les calculs sur les coefficients disponibles dans `Gfun`. Il faut aussi continuer de développer Sage, en augmentant ou en créant d'autres paquets, pour que ce logiciel puisse être utilisé par le plus de scientifiques possible.

Bibliographie

- [HR06] Ralf Hemmecke and Martin Rubey. Aldor-Combinat : An Implementation of Combinatorial Species, 2006. Available at <http://www.risc.uni-linz.ac.at/people/hemmecke/aldor/combinat/>.
- [S⁺13] W. A. Stein et al. *Sage Mathematics Software (Version 5.8)*. The Sage Development Team, 2013. <http://www.sagemath.org>.
- [SPH01] Neil Schemenauer, Tim Peters, and Lie Hetland. Simple Generators, 2001. <http://www.python.org/dev/peps/pep-0255/>.