# Implantation des bases des polynômes en Sage

Créé en 2005, le logiciel Sage [S+11] a pris une place importante au sein des systèmes de calcul formel pour la recherche en mathématique et plus particulièrement en combinatoire. Il a pour but de proposer une alternative libre et gratuite aux logiciels traditionnels tels que Maple ou Magma. Sous licence GPL, il s'ajoute à la grande famille des logiciels libres développés dans la philosophie du système d'exploitation Linux. Le développement libre n'est pas nouveau en mathématiques et Sage se propose aussi de réunir au sein d'un même logiciel les outils des différentes communautés. On y trouve entre autres des programmes tels que GAP et Symmetrica. Son modèle de développement est décentralisé, basé sur la communauté. Chaque utilisateur peut devenir développeur et proposer des modifications, corrections ou ajouts au logiciel. Ces propositions sont envoyées sous formes de *patchs* et validées par d'autres développeurs avant d'être incorporées à une nouvelle version du logiciel. Dans le domaine du calcul formel traditionnel, Sage a atteint un niveau comparable aux logiciels classiques. Il a été ajouté récemment à la liste des logiciels de calculs formels admis au concours de l'agrégation. On pourra lire [CCD+13] pour se familiariser avec ses principales fonctionnalités.

Le modèle de développement de Sage trouve un intérêt particulier en combinatoire algébrique. La recherche en combinatoire est souvent basée sur une exploration préliminaire par le calcul informatique. Les outils nécessaires font appel à de nombreuses branches des mathématiques : algèbre linéaire, théorie des groupes, théorie des représentations... Le développement d'une base commune par les différentes communautés est donc un atout majeur. Avant même l'arrivée de Sage, un projet *Mupad-Combinat* avait été mis en place au sein du logiciel Mupad. Cependant, les possibilités de développement ont été limitées par les contraintes liées au logiciel, en particulier car il était propriétaire. En 2008, la communauté *combinat* décide de rejoindre Sage devenant alors *Sage-combinat* [SCc08]. Concrètement, Sage-combinat est un ensemble de patchs qui visent à améliorer l'intégration de la combinatoire dans Sage. Sage-combinat est aussi et surtout une communauté

de chercheurs / développeurs qui créent, testent et intègrent ces patchs.

Il existait déjà dans Sage une implantation classique des polynômes multivariés en tant qu'expressions formelles en plusieurs variables. Cette implantation ne répond pas aux besoins de calculs sur des bases multiples telles que Schubert, Grothendieck ou les polynômes clés. En effet, on veut pouvoir travailler formellement sur les éléments de ces bases, leur appliquer des opérateurs, les convertir d'une base à l'autre et les développer sur la base des monômes seulement si nécessaire. Pour ce faire, nous considérons un polynôme dans une base quelconque comme une somme formelle de vecteurs. L'interprétation de ce vecteur est donné par la base : sur les monômes le vecteur est un exposant, sur les autres bases c'est ce que nous avons décrit dans le paragraphe 3.3. Cela nous permet aussi de travailler en un nombre quelconque de variables, les variables n'étant plus données chacune comme des éléments formels mais seulement par la taille du vecteur. Cette approche n'est pas nouvelle, on la trouve en particulier au sein du logiciel ACE développé sous Maple [Vei96] ou en partie dans le logiciel Symmetrica. Cependant, il n'existait pas encore de version en Sage, et donc accessible à tous, d'une telle implantation. Ce faisant, nous nous inscrivons ici au sein d'un projet plus important de développement de la combinatoire dans Sage. Notre implantation a fait l'objet de plusieurs présentation et d'une publication [Pon11].

Outre les aspects philosophiques qui nous ont fait choisir un logiciel libre plutôt que propriétaire comme Maple, le choix de Sage est aussi d'ordre technique. Sage est développé en python et utilise une architecture orientée objet. Toute notre implantation est basée sur cette architecture, utilisant de façon avancée l'héritage et la classification. En particulier, nous utilisons le modèle de développement en *Catégories / Parents / Éléments* mis en place par Nicolas Thiéry [NT09]. Par ailleurs, nous avons pu profiter des implantations déjà existantes en Sage comme les groupes de Coxeter ou les modules libres.

Dans le paragraphe 5.1, nous commençons par présenter les fonctionnalités de base du logiciel : création d'un polynôme, application d'un opérateur, changement de base, etc. Le paragraphe 5.2 nous sert à expliquer l'architecture globale du projet et nos choix d'implantation. Enfin, paragraphe 5.3, nous donnons des exemples d'applications avancées, et en particulier, les calculs du chapitre 4.

## 5.1 Fonctionnalités, exemples d'utilisation

### 5.1.1 Installation et distribution du logiciel

Le logiciel est disponible en tant que patch additionnel à Sage [Pon10] . On peut le télécharger sur sa page dédiée : `http://trac.sagemath.org/sage_trac/ticket/6629` et l'installer. Les étapes sont les suivantes :

1. Installer Sage.
2. Télécharger le patch contenant le programme.

3. Effectuer une copie de votre version de sage avec la commande `sage -clone polynomials`.
4. Lancer sage puis sur la line de commande sage, taper `hg_sage.apply("nom_du_patch.patch")`.
5. Quitter sage (quit) puis lancer la commande `sage -br`.

Le patch est en cours de révision. Il doit être approuvé par d'autres utilisateurs / développeurs et sera ensuite intégré au logiciel Sage. Il n'y aura alors plus d'installation spécifique et les fonctionnalités seront présentes sur l'installation par défaut de Sage.

Il est aussi possible de bénéficier du patch en installant la suite complète des patchs de *sage-combinat* par la commande `sage - combinat install`. Ces patchs sont expérimentaux et n'ont pas encore fait l'objet d'un processus de validation.

### 5.1.2 Création d'un polynôme, application d'opérateurs

On crée d'abord l'objet de base représentant l'algèbre des polynômes.

```
sage: A = AbstractPolynomialRing(QQ)
sage: A
The abstract ring of multivariate polynomials on x over Rational
   Field
```

On définit ensuite la base des monômes qui va nous servir à créer des éléments.

```
sage: m = A.monomial_basis(); m
The ring of multivariate polynomials on x over Rational Field on
   the monomial basis
```

On peut alors créer un polynôme à partir de `m`.

```
sage: pol = m[1,1,2] + m[2,3]; pol
x[1, 1, 2] + x[2, 3, 0]
```

L'élément `x[1,1,2]` signifie $x^{(1,1,2)} = x_1^1 x_2^1 x_3^2$ comme on peut le voir en transformant le polynôme en expression symbolique.

```
sage: pol.to_expr()
x1^2*x2^3 + x1*x2*x3^2
```

Il n'est pas nécessaire de préciser à l'avance le nombre de variables : il sera calculé en fonction de la taille du vecteur. L'objet polynôme connaît son nombre de variable à travers son objet *parent*.

```
sage: pol.parent()
The ring of multivariate polynomials on x over Rational Field with
    3 variables on the monomial basis
sage: pol = pol.change_nb_variables(4)
sage: pol
x[1, 1, 2, 0] + x[2, 3, 0, 0]
sage: pol.parent()
```

```
The ring of multivariate polynomials on x over Rational Field with
    4 variables on the monomial basis
```

Un polynôme est toujours vu comme une somme formelle de vecteurs. Il ne peut pas être factorisé. Si l'on multiplie deux polynômes, le résultat sera toujours donné sous forme développée.

```
sage: pol * pol
x[2, 2, 4, 0] + 2*x[3, 4, 2, 0] + x[4, 6, 0, 0]
```

On peut à présent appliquer des opérateurs de différences divisées à un polynôme. Par exemple, on trouve ci-dessous le calcul (3.12).

```
sage: pol = m[5,1]; pol
x[5, 1]
sage: pol.divided_difference(1)
x[1, 4] + x[2, 3] + x[3, 2] + x[4, 1]
sage: pol.isobaric_divided_difference(1)
x[1, 5] + x[2, 4] + x[3, 3] + x[4, 2] + x[5, 1]
sage: pol.hat_isobaric_divided_difference(1)
x[1, 5] + x[2, 4] + x[3, 3] + x[4, 2]
```

Même si le polynôme n'est défini que sur deux variables, on peut appliquer une différence divisée qui fait intervenir la troisième variable.

```
sage: pol.isobaric_divided_difference(2)
x[5, 1, 0] + x[5, 0, 1]
```

Certaines méthodes permettent d'appliquer une série d'opérations.

```
sage: pol = m[1,4,2] + m[5,5,1]
sage: pol.apply_reduced_word([1,2])
-x[1, 2, 2] + x[3, 1, 1]
sage: pol.apply_reduced_word([1,2],method="pi")
-x[2, 2, 3] - x[2, 3, 2] - x[3, 2, 2] + x[5, 1, 5] + x[5, 2, 4] +
   x[5, 3, 3] + x[5, 4, 2] + x[5, 5, 1]
sage: pol.apply_reduced_word([1,2],method="hatpi")
-x[1, 2, 4] - x[1, 3, 3] - x[2, 2, 3]
```

Ici on a appliqué au polynôme les opérateurs respectifs $\partial_1\partial_2$, $\pi_1\pi_2$ et $\hat{\pi}_1\hat{\pi}_2$. On peut par exemple vérifier la relation de tresse.

```
sage: pol.apply_reduced_word([1,2,1])
x[1, 1, 2] + x[1, 2, 1] + x[2, 1, 1]
sage: pol.apply_reduced_word([2,1,2])
x[1, 1, 2] + x[1, 2, 1] + x[2, 1, 1]
```

Il est aussi possible de mélanger différents types d'opérateurs. Voici la commande pour appliquer $\partial_2$ puis $\pi_1$.

```
sage: pol.apply_composed_morphism([("d",2),("pi",1)])
x[1, 5, 4] - x[2, 2, 2] + x[2, 4, 4] + x[2, 5, 3] + x[3, 3, 4] + x
    [3, 4, 3] + x[3, 5, 2] + x[4, 2, 4] + x[4, 3, 3] + x[4, 4, 2]
    + x[4, 5, 1] + x[5, 1, 4] + x[5, 2, 3] + x[5, 3, 2] + x[5, 4,
    1]
```

Enfin, on peut appliquer des opérateurs de type $B$, $C$ ou $D$ que nous avons vus paragraphe 3.1.3.

```
pol.divided_difference(1,"B")
x[-5, 5, 1] + x[-4, 5, 1] + x[-3, 5, 1] + x[-2, 5, 1] + x[-1, 4,
    2] + x[-1, 5, 1] + x[1, 5, 1] + x[2, 5, 1] + x[3, 5, 1] + x[4,
     5, 1] + x[0, 4, 2] + x[0, 5, 1]
sage: pol.divided_difference(1,"C")
x[-4, 5, 1] + x[-2, 5, 1] + x[2, 5, 1] + x[4, 5, 1] + x[0, 4, 2] +
     x[0, 5, 1]
sage: pol.divided_difference(2,"D")
x[-4, -5, 1] + x[-3, -4, 1] + x[-3, -1, 2] + x[-2, -3, 1] + x[-2,
    0, 2] + x[-1, -2, 1] + x[-1, 1, 2] + x[1, 3, 2] + x[1, 0, 1] +
     x[2, 1, 1] + x[3, 2, 1] + x[4, 3, 1] + x[5, 4, 1] + x[0, -1,
    1] + x[0, 2, 2]
```

Quelque soit le nombre de variables du polynôme, les opérateurs de types $B$, $C$ et $D$ sont toujours respectivement $\partial_i^B$, $\partial_i^C$ et $\partial_i^D$ définis en (3.34), (3.34) e (3.34), même quand $i \neq n$. On peut mélanger des opérateurs de types différents, par exemple, ici $\partial_1 \pi_1^B \pi_1^C$.

```
sage: pol.apply_composed_morphism([("d",1),("pi",1,"B"),("pi",2,"C
    ")])
-x[-3, -1, 2] - x[-3, 1, 2] - x[-2, -2, 2] - x[-2, -1, 2] - x[-2,
    1, 2] - x[-2, 2, 2] - x[-2, 0, 2] - x[-1, -3, 2] - x[-1, -2,
    2] - 2*x[-1, -1, 2] - 2*x[-1, 1, 2] - x[-1, 2, 2] - x[-1, 3,
    2] - x[-1, 0, 2] - x[1, -3, 2] - x[1, -2, 2] - 2*x[1, -1, 2] -
     2*x[1, 1, 2] - x[1, 2, 2] - x[1, 3, 2] - x[1, 0, 2] - x[2,
    -2, 2] - x[2, -1, 2] - x[2, 1, 2] - x[2, 2, 2] - x[2, 0, 2] -
    x[3, -1, 2] - x[3, 1, 2] - x[0, -3, 2] - x[0, -2, 2] - 2*x[0,
    -1, 2] - 2*x[0, 1, 2] - x[0, 2, 2] - x[0, 3, 2] - x[0, 0, 2]
```

Cependant, il est possible d'indexer les monômes directement par des éléments de l'espace ambiant du système de racines d'un groupe de Coxeter. Dans ce cas, le type du groupe est contenu dans l'objet polynôme et les opérateurs utilisés sont ceux du groupe.

```
sage: mb = A.ambient_space_basis("B")
sage: mb
The ring of multivariate polynomials on x over Rational Field on
    the Ambient space basis of type B
sage: polb = mb[1,4,2] + mb[5,5,1]
sage: polb.group_type()
'B'
sage: polb.divided_difference(1)
-x(1, 3, 2) - x(2, 2, 2) - x(3, 1, 2)
sage: polb.divided_difference(3)
x(1, 4, 0) + x(1, 4, -2) + x(1, 4, -1) + x(1, 4, 1) + x(5, 5, 0) +
     x(5, 5, -1)
```

Ici, la première opération réalisée est la différence divisée classique $\partial_1$ et la seconde est $\partial_3^B$. En effet comme on l'a vu paragraphe 3.1.3, en type $B$, seule la racine simple $r_n$ est différente des racines de type $A$.

Il est aussi possible de définir soi-même un opérateur agissant sur les exposants.

```
sage: def affine(self,key): return self.divided_difference_on_
    basis(key) - self.si_on_basis(key)
 sage: m.add_operator("a",affine)
sage: pol
x[1, 4, 2] + x[5, 5, 1]
sage: pol.apply_morphism(1,method="a")
-x[1, 3, 2] - x[2, 2, 2] - x[3, 1, 2] - x[4, 1, 2] - x[5, 5, 1]
sage: pol.divided_difference(1) - pol.si(1)
-x[1, 3, 2] - x[2, 2, 2] - x[3, 1, 2] - x[4, 1, 2] - x[5, 5, 1]
```

Ici, on a défini la méthode `affine` comme la différence entre la différence divisée et l'opérateur $s_i$ qui échange deux variables. En l'ajoutant à la base $m$, on a créé une famille d'opérateurs $a_i = \partial_i - s_i$. On peut par exemple vérifier que ce nouvel opérateur vérifie les relations de tresses.

```
sage: pol.apply_reduced_word([1,2,1],method="a") == pol.apply_
    reduced_word([2,1,2],method="a")
True
```

### 5.1.3 Schubert, Grothendieck, polynômes clés et autres bases

On a déjà montré différentes bases dans les exemples précédents. Les monômes peuvent être indexés soit par des vecteurs, soit par des éléments de l'espace ambiant d'un système de racine.

```
sage: A = AbstractPolynomialRing(QQ)
sage: m = A.monomial_basis(); m
The ring of multivariate polynomials on x over Rational Field on
    the monomial basis
sage: ma = A.ambient_space_basis("A"); ma
The ring of multivariate polynomials on x over Rational Field on
    the Ambient space basis of type A
sage: mb = A.ambient_space_basis("B"); mb
The ring of multivariate polynomials on x over Rational Field on
    the Ambient space basis of type B
sage: mc = A.ambient_space_basis("C"); mc
The ring of multivariate polynomials on x over Rational Field on
    the Ambient space basis of type C
sage: md = A.ambient_space_basis("D"); md
The ring of multivariate polynomials on x over Rational Field on
    the Ambient space basis of type D
sage: pol = m[2,2,3]; pol
x[2, 2, 3]
sage: mb(pol)
x(2, 2, 3)
```

Les autres bases sont définies à partir de l'algorithme qui développe un vecteur en somme de monômes. Voici par exemple les polynômes de Schubert.

```
sage: Schub = A.schubert_basis_on_vectors()
sage: Schub
The ring of multivariate polynomials on x over Rational Field on
   the Schubert basis of type A (indexed by vectors)
sage: pol = Schub[2,1,2] + Schub[1,3,2]
sage: pol
Y(1, 3, 2) + Y(2, 1, 2)
sage: pol.expand()
x(1, 3, 2) + x(2, 1, 2) + x(2, 2, 1) + x(2, 2, 2) + x(2, 3, 1) + x
   (3, 1, 1) + x(3, 1, 2) + x(3, 2, 1)
```

La méthode `expand` utilise l'algorithme que nous avons vu au paragraphe 3.3.1. À partir du développement d'un polynôme de Schubert, des méthodes préexistantes en Sage permettent d'obtenir le morphisme inverse d'une somme de monômes vers un polynôme de Schubert.

```
sage: Schub(m[2,4,1] + m[5,5,2])
Y(2, 4, 1) - Y(3, 3, 1) - Y(4, 2, 1) + Y(5, 5, 2)
sage: pol^2
Y(2, 6, 4) + 2*Y(3, 4, 4) + 2*Y(3, 5, 3) + Y(3, 5, 4) + Y(3, 6, 3)
    + Y(4, 2, 4) + Y(4, 3, 3) + 2*Y(4, 4, 3) + Y(4, 4, 4) + 2*Y
   (4, 5, 2) + Y(4, 5, 3) + Y(5, 2, 3) + 2*Y(5, 2, 4) + 2*Y(5, 3,
    3) + Y(5, 5, 2) + Y(6, 2, 2) + 2*Y(6, 2, 3)
```

Pour effectuer la multiplication, le programme développe les deux polynômes dans la base des monômes et transforme ensuite le résultat en polynôme de Schubert. On peut aussi appliquer les opérateurs de différences divisées directement aux polynômes de Schubert.

```
sage: pol
Y(1, 3, 2) + Y(2, 1, 2)
sage: pol.divided_difference(1)
Y(1, 1, 2)
sage: pol.isobaric_divided_difference(1)
Y(1, 2, 2) + Y(1, 3, 1) + Y(1, 3, 2)
sage: pol.hat_isobaric_divided_difference(1)
Y(1, 2, 2) + Y(1, 3, 1) - Y(2, 1, 2)
```

Par défaut, le programme applique l'opérateur sur le polynôme développé avant de l'exprimer dans la base des Schubert. Cependant, si une méthode a été implantée directement pour les polynômes de Schubert, c'est elle qui sera utilisée. Ici, c'est le cas de la différence divisée $\partial_i$.

Pour les polynômes de Grothendieck simples, on a implanté deux bases : celle où les variables $y_i$ sont spécialisées à 1 et où les exposants des $x$ sont négatifs et celle où un changement de variable a été appliqué pour obtenir des exposants positifs (cf. paragraphe 3.3.2).

```
sage: Grothp = A.grothendieck_positive_basis_on_vectors(); Grothp
The ring of multivariate polynomials on x over Rational Field on
   the Grothendieck basis of type A, with positive exposants (
   indexed by vectors)
sage: Grothn = A.grothendieck_negative_basis_on_vectors(); Grothn
```

```
The ring of multivariate polynomials on x over Rational Field on
    the Grothendieck basis of type A with negative exposants (
    indexed by vectors)
sage: pol1 = Grothp[1,3,2] + Grothp[2,1,2]; pol1
G(1, 3, 2) + G(2, 1, 2)
sage: pol2 = Grothn[1,3,2] + Grothn[2,1,2]; pol2
G(1, 3, 2) + G(2, 1, 2)
sage: pol1.expand()
x(1, 3, 2) + x(2, 1, 2) + x(2, 2, 1) + x(2, 3, 1) - 2*x(2, 3, 2) +
     x(3, 1, 1) - x(3, 2, 2) - x(3, 3, 1) + x(3, 3, 2)
sage: pol2.expand()
2*x(0, 0, 0) + x(-3, -3, -2) - x(-3, -3, -1) + x(-3, -2, 0) - 2*x
    (-3, -2, -2) + x(-3, -2, -1) - x(-3, -1, 0) + x(-3, -1, -2) +
    x(-2, 0, 0) + x(-2, -3, 0) - x(-2, -3, -2) - 5*x(-2, -2, 0) +
    5*x(-2, -2, -1) + 3*x(-2, -1, 0) + 2*x(-2, -1, -2) - 5*x(-2,
    -1, -1) - x(-2, 0, -2) - 3*x(-1, 0, 0) - x(-1, -3, 0) + x(-1,
    -3, -1) + 4*x(-1, -2, 0) + 3*x(-1, -2, -2) - 7*x(-1, -2, -1) -
     4*x(-1, -1, -2) + 4*x(-1, -1, -1) + x(-1, 0, -2) + 2*x(-1, 0,
     -1) - x(0, -2, -2) + x(0, -2, -1) - 2*x(0, -1, 0) + x(0, -1,
    -2) + x(0, -1, -1) - 2*x(0, 0, -1)
sage: polexp = pol1.expand()
sage: polexp.subs_var([(i, 1 - A.var(i)^(-1)) for i in xrange(1,4)
    ]) == pol2
True
```

Le changement de base de la base des monômes vers les polynômes de Grothendieck n'est définie que pour la base en exposants positifs.

```
sage: Grothp(m[2,4,1] + m[5,5,2])
G(2, 4, 1) - G(3, 3, 1) + G(3, 4, 1) - G(4, 2, 1) + G(4, 4, 1) + G
    (5, 5, 2)
sage: pol1^2
G(2, 6, 4) + 2*G(3, 4, 4) + 2*G(3, 5, 3) - G(3, 5, 4) + G(3, 6, 3)
     - 2*G(3, 6, 4) + G(4, 2, 4) + G(4, 3, 3) - G(4, 3, 4) + 2*G
    (4, 4, 3) - G(4, 4, 4) + 2*G(4, 5, 2) - 3*G(4, 5, 3) - G(4, 6,
     3) + G(4, 6, 4) + G(5, 2, 3) + G(5, 2, 4) + G(5, 3, 3) - 3*G
    (5, 3, 4) - 2*G(5, 4, 3) + 2*G(5, 4, 4) - G(5, 5, 2) + 2*G(5,
    5, 3) - G(5, 5, 4) - G(5, 6, 2) + 2*G(5, 6, 3) - G(5, 6, 4) +
    G(6, 2, 2) - G(6, 2, 4) - 3*G(6, 3, 3) + 3*G(6, 3, 4) - 2*G(6,
     4, 2) + 4*G(6, 4, 3) - 2*G(6, 4, 4) + 2*G(6, 5, 2) - 4*G(6,
    5, 3) + 2*G(6, 5, 4)
```

Les polynômes clés sont les seuls qui soient implantés pour les types $B$, $C$ et $D$ selon les définitions données au paragraphe 3.3.3. On peut vérifier les calculs donnés en exemple dans cette section.

```
sage: K = A.demazure_basis_on_vectors(); K
The ring of multivariate polynomials on x over Rational Field on
    the Demazure basis of type A (indexed by vectors)
sage: Kb = A.demazure_basis_on_vectors("B"); Kb
The ring of multivariate polynomials on x over Rational Field on
    the Demazure basis of type B (indexed by vectors)
sage: Kc = A.demazure_basis_on_vectors("C"); Kc
```

```
The ring of multivariate polynomials on x over Rational Field on
   the Demazure basis of type C (indexed by vectors)
sage: Kd = A.demazure_basis_on_vectors("C"); Kd
The ring of multivariate polynomials on x over Rational Field on
   the Demazure basis of type C (indexed by vectors)
sage: K[1,0,1].expand()
x(1, 1, 0) + x(1, 0, 1)
sage: Kb[2,-1,1].expand()
x(2, 0, 0) + x(2, -1, 1) + x(2, 1, 0) + x(2, 1, -1) + x(2, 1, 1) +
    x(2, 0, 1)
sage: Kc[2,-1,1].expand()
x(2, 0, 0) + x(2, -1, 1) + x(2, 1, -1) + x(2, 1, 1)
sage: Kd[2,-2,1].expand()
x(2, -2, 1) + x(2, -1, 0) + x(2, -1, 2) + 2*x(2, 1, 0) + x(2, 1,
   -2) + x(2, 1, 2) + x(2, 2, -1) + x(2, 2, 1) + x(2, 0, -1) + 2*
   x(2, 0, 1)
```

A chaque fois, le polynôme est développé dans la base `ma`, `mb`, `mc` ou `md`.

```
sage: Kb[2,-1,1].expand().parent()
The ring of multivariate polynomials on x over Rational Field with
    3 variables on the Ambient space basis of type B
sage: Kc[2,-1,1].expand().parent()
The ring of multivariate polynomials on x over Rational Field with
    3 variables on the Ambient space basis of type C
```

Les changements de bases des polynômes de Schubert, Grothendieck ou des polynômes clés se font non seulement vers la base monomiale mais aussi entre bases. Le passage par la base monomiale est fait de façon implicite par le programme.

```
sage: K(Schub[1,0,1])
K(1, 0, 1) + K(2, 0, 0)
sage: Schub(Grothp[1,4,2])
Y(1, 4, 2) - 2*Y(2, 4, 2) - Y(3, 4, 1) + Y(3, 4, 2)
```

Enfin, il est possible de définir sa propre base. En effet, une base est entièrement donnée par l'algorithme de développement d'un élément. On définit donc simplement cette fonction. Voici par exemple une copie des polynômes de Schubert.

```
sage: def schubert_on_basis(v, basis, call_back):
...       for i in xrange(len(v)-1):
...           if(v[i]<v[i+1]):
...               v[i], v[i+1] = v[i+1] + 1, v[i]
...               return call_back(v).divided_difference(i+1)
...       return basis(v)
```

Le paramètre `v` est le vecteur indexant l'élément, `basis` est la base sur laquelle on est en train de développer et `call_back` est l'appel récursif de la méthode.

```
sage: myBasis = A.linear_basis_on_vectors("A","MySchub","Y",
   schubert_on_basis)
sage: pol = myBasis[2,1,2] + myBasis[1,3,2]
```

```
sage: pol.expand()
x(1, 3, 2) + x(2, 1, 2) + x(2, 2, 1) + x(2, 2, 2) + x(2, 3, 1) + x
    (3, 1, 1) + x(3, 1, 2) + x(3, 2, 1)
sage: Schub(pol)
Y(1, 3, 2) + Y(2, 1, 2)
```

### 5.1.4 Polynômes doubles

Un polynôme en deux ensembles de variables $x$ et $y$ est un polynôme en $x$ dont les coefficients sont des polynômes en $y$. Nous avons défini une nouvelle classe pour en faciliter l'utilisation

```
sage: D = DoubleAbstractPolynomialRing(QQ); D
The abstract ring of multivariate polynomials on x over The
    abstract ring of multivariate polynomials on y over Rational
    Field
sage: D.an_element()
y[0]*x[0, 0, 0] + 2*y[0]*x[1, 0, 0] + y[0]*x[1, 2, 3] + 3*y[0]*x
    [2, 0, 0]
sage: mx = D.monomial_basis(); mx
The ring of multivariate polynomials on x over The abstract ring
    of multivariate polynomials on y over Rational Field on the
    monomial basis
sage: my = D.coeffs_ring().monomial_basis(); my
The ring of multivariate polynomials on y over Rational Field on
    the monomial basis
sage: pol = my[1,2] * mx[2,1,3] + my[1,2] * mx[1,4,2] + my[3,1] *
    mx[2,1,3]
sage: pol
(y[1,2])*x[1, 4, 2] + (y[1,2]+y[3,1])*x[2, 1, 3]
```

On peut facilement échanger le rôle de $x$ et $y$.

```
sage: pol.swap_coeffs_elements()
(x[1,4,2]+x[2,1,3])*y[1, 2] + (x[2,1,3])*y[3, 1]
sage: pol.swap_coeffs_elements().parent()
The ring of multivariate polynomials on y over The abstract ring
    of multivariate polynomials on x over Rational Field with 2
    variables on the monomial basis
```

On peut changer la base aussi bien des variables $x$ que $y$.

```
sage: Schubx = D.schubert_basis_on_vectors()
sage: Schuby = D.coeffs_ring().schubert_basis_on_vectors()
sage: pol = my[1,2] * mx[2,1,3] + my[1,2] * mx[1,4,2] + my[3,1] *
    mx[2,1,3]
sage: pol
(y[1,2])*x[1, 4, 2] + (y[1,2]+y[3,1])*x[2, 1, 3]
sage: Schubx(pol)
(y[1,2])*Yx(1, 4, 2) + (y[1,2]+y[3,1])*Yx(2, 1, 3) + (-y[1,2]-y
    [3,1])*Yx(2, 2, 2) + (-y[1,2]-y[3,1])*Yx(2, 3, 1) + (-y[1,2])*
    Yx(2, 3, 2) + (-y[1,2])*Yx(2, 4, 1) + (-y[1,2]-y[3,1])*Yx(3,
    1, 2) + (y[1,2]+y[3,1])*Yx(3, 2, 1) + (-y[1,2]-y[3,1])*Yx(4,
    1, 1) + (-y[1,2])*Yx(4, 1, 2) + (y[1,2])*Yx(4, 2, 1)
```

```
sage: pol.change_coeffs_bases(Schuby)
(Yy(1,2)-Yy(2,1))*x[1, 4, 2] + (Yy(1,2)-Yy(2,1)+Yy(3,1))*x[2, 1,
    3]
```

Enfin, on peut utiliser les bases de Schubert et Grothendieck doubles que nous avons définies dans les paragraphes 3.3.1 et 3.3.2.

```
sage: DSchub = D.double_schubert_basis_on_vectors(); DSchub
The ring of multivariate polynomials on x over The abstract ring
    of multivariate polynomials on y over Rational Field on the
    Double Schubert basis of type A (indexed by vectors)
sage: pol = DSchub[1,2,1]; pol
y[0]*YY(1, 2, 1)
sage: pol.expand()
(y(3,1,0)+y(3,0,1))*x(0, 0, 0) + (-y(2,1,0)-y(2,0,1)-y(3,0,0))*x
    (1, 0, 0) + (y(1,1,0)+y(1,0,1)+2*y(2,0,0))*x(1, 1, 0) + (-2*y
    (1,0,0)-y(0,1,0)-y(0,0,1))*x(1, 1, 1) + (-y[1])*x(1, 2, 0) + y
    [0]*x(1, 2, 1) + (y(1,1,0)+y(1,0,1)+y(2,0,0))*x(1, 0, 1) + y
    [2]*x(2, 0, 0) + (-y[1])*x(2, 1, 0) + y[0]*x(2, 1, 1) + (-y
    [1])*x(2, 0, 1) + (-y(2,1,0)-y(2,0,1)-y(3,0,0))*x(0, 1, 0) + (
    y(1,1,0)+y(1,0,1)+y(2,0,0))*x(0, 1, 1) + y[2]*x(0, 2, 0) + (-y
    [1])*x(0, 2, 1) + (-y(2,1,0)-y(2,0,1))*x(0, 0, 1)
sage: pol = my[1,1] * mx[1,2,1]; pol
(y[1,1])*x[1, 2, 1]
sage: DSchub(pol)
(y(2,3,1))*YY(0, 0, 0) + (-y(2,2,1))*YY(1, 0, 0) + (y(1,1,1))*YY
    (1, 1, 1) + (y[1,1])*YY(1, 2, 1) + (-y(2,2,0))*YY(1, 0, 1) +
    (-y[1,1])*YY(2, 1, 1) + (-y[2,1])*YY(2, 0, 1) + (y(2,2,1))*YY
    (0, 1, 0) + (y(2,1,1)+y(2,2,0))*YY(0, 1, 1) + (y[2,2])*YY(0,
    2, 0) + (y[2,1])*YY(0, 2, 1) + (y(2,3,0))*YY(0, 0, 1)
sage: DGroth = D.double_grothendieck_basis_on_vectors(); DGroth
The ring of multivariate polynomials on x over The abstract ring
    of multivariate polynomials on y over Rational Field on the
    Double Grothendieck basis of type A (indexed by vectors)
sage: pol = DGroth[1,2,1]; pol
y[0]*GG(1, 2, 1)
sage: pol.expand()
y[0]*x(0, 0, 0) + (-y(2,1,1))*x(-2, -2, 0) + (y(3,1,1))*x(-2, -2,
    -1) + (y(1,1,1))*x(-2, -1, 0) + (-y(2,1,1))*x(-2, -1, -1) + (-
    y[1])*x(-1, 0, 0) + (y(1,1,1))*x(-1, -2, 0) + (-y(2,1,1))*x
    (-1, -2, -1) + (y(2,0,0)-y(0,1,1))*x(-1, -1, 0) + (y(1,1,1)-y
    (3,0,0))*x(-1, -1, -1) + y[2]*x(-1, 0, -1) + (-y[1])*x(0, -1,
    0) + y[2]*x(0, -1, -1) + (-y[1])*x(0, 0, -1)
```

## 5.2 Architecture du logiciel

### 5.2.1 Catégorie / Parent / Élément

Une notion de base en programmation objet est celle *d'héritage*. Si plusieurs objets utilisent une même méthode, on peut les faire hériter d'un objet commun et implanter cette méthode dans ce parent. C'est ce qu'on appelle la factorisation

du code. Elle évite de répéter plusieurs fois le même algorithme. Quand les objets représentent des notions mathématiques, la notion d'héritage n'est plus suffisante. En effet, non seulement on veut pouvoir donner des méthodes communes à tous les polynômes mais on veut pouvoir travailler sur l'ensemble des polynômes en tant qu'objet. C'est à ce niveau qu'interviennent les notions *d'éléments* et de *parents*.

```
sage: from sage.structure.element import Element
sage: from sage.structure.parent import Parent
sage: A = AbstractPolynomialRing(QQ)
sage: pol = A.an_element()
sage: pol
x[0, 0, 0] + 2*x[1, 0, 0] + x[1, 2, 3] + 3*x[2, 0, 0]
sage: isinstance(pol,Element)
True
sage: pol.parent()
The ring of multivariate polynomials on x over Rational Field with
    3 variables on the monomial basis
sage: isinstance(pol.parent(),Parent)
True
```

Concrètement, `Element` et `Parent` sont des objets de Sage. Lorsqu'on définit la classe des `Polynomes`, c'est-à-dire l'objet représentant l'ensemble des polynômes, celui-ci hérite de `Parent`. Les éléments de type `Polynome` (sans "s") héritent de `Element`. Le lien qui unit les classes `Polynomes` et `Polynome` n'est pas un lien d'héritage. Cependant, on peut définir des méthodes communes à tous les polynômes au sein de la classe `Polynomes` : elles seront ajoutées dynamiquement aux éléments.

```
sage: type(pol)
sage.combinat.multivariate_polynomials.monomial.
   FiniteMonomialBasis_with_category.element_class
sage: type(pol.parent())
sage.combinat.multivariate_polynomials.monomial.
   FiniteMonomialBasis_with_category
```

Ici, la classe `Polynomes` est `FiniteMonomialBasis` et les éléments polynômes sont de type `FiniteMonomialBasis.element_class`. On remarque que le nom de la classe est en fait `FiniteMonomialBasis_with_category`, cette classe a été créée dynamiquement pour permettre au parent de récupérer les méthodes de sa *catégorie*.

En effet, il arrive que des algorithmes soient communs à plusieurs types de parents qui vérifient des propriétés mathématiques communes. Par exemple, un module libre dont la base est indexée par des objets combinatoires est considéré dans Sage comme un `Parent`. Si ce module est en fait une algèbre, il suffit de définir le produit sur les éléments de la base. Ainsi, dans la catégorie `Algèbre` on implémente un algorithme général qui s'appliquera à tous les parents utilisant la catégorie. Un parent possède en général de très nombreuses catégories.

```
sage: pol.parent().categories()
[The category of bases of The abstract ring of multivariate
   polynomials on x over Rational Field with 3 variables where
```

```
   algebra tower is The ring of multivariate polynomials on x
   over Rational Field on the monomial basis,
 Category of realizations of The abstract ring of multivariate
    polynomials on x over Rational Field with 3 variables,
 Category of realizations of sets,
 Category of graded algebras with basis over Rational Field,
 Category of graded modules with basis over Rational Field,
 Category of graded algebras over Rational Field,
 Category of graded modules over Rational Field,
 Category of algebras with basis over Rational Field,
 Category of modules with basis over Rational Field,
 Category of algebras over Rational Field,
 Category of rings,
 Category of rngs,
 Category of vector spaces over Rational Field,
 Category of modules over Rational Field,
 Category of bimodules over Rational Field on the left and
    Rational Field on the right,
 Category of left modules over Rational Field,
 Category of right modules over Rational Field,
 Category of commutative additive groups,
 Category of semirings,
 Category of commutative additive monoids,
 Category of commutative additive semigroups,
 Category of additive magmas,
 Category of monoids,
 Category of semigroups,
 Category of magmas,
 Category of sets,
 Category of sets with partial maps,
 Category of objects]
```

### 5.2.2 Multibases, multivariables

Un des objectifs du programme est de pouvoir travailler avec plusieurs bases. On définit pour cela un parent abstrait qui ne possédera pas directement d'éléments mais des *réalisations*, c'est-à-dire d'autre parents qui représenteront les différentes bases (cf. figure 5.1).

Les changements de bases sont des objets `Morphisme` qui possèdent une méthode retournant un objet de la base 2 à partir d'un objet de la base 1. On crée ces morphismes au moment de la création des bases et on les enregistre comme des *conversions de type*. Tout objet parent possède une méthode `call`. On appelle la méthode `call` du parent avec en argument un objet d'un autre type. La méthode vérifie alors s'il existe une conversion possible entre les deux types : elle utilise pour cela le graphe créé par les conversions enregistrées.

```
sage: A = AbstractPolynomialRing(QQ)
sage: pol = A.an_element(); pol
x[0, 0, 0] + 2*x[1, 0, 0] + x[1, 2, 3] + 3*x[2, 0, 0]
sage: Schub = A.schubert_basis_on_vectors()
```

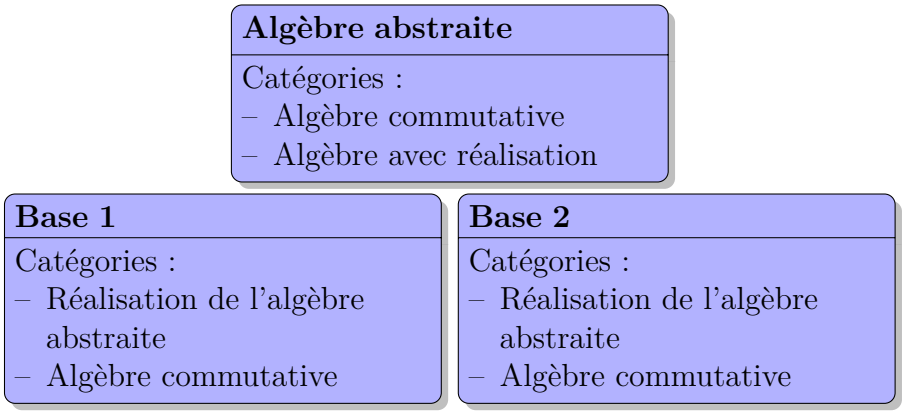


FIGURE 5.1 – Structure d'algèbre abstraite avec plusieurs bases

```
sage: Schub(pol)
Y(0, 0, 0) + 2*Y(1, 0, 0) + Y(1, 2, 3) - Y(1, 3, 2) + 3*Y(2, 0, 0)
    - Y(2, 1, 3) + Y(2, 3, 1) + Y(3, 1, 2) - Y(3, 2, 1) + Y(4, 1,
    1)
sage: Schub.has_coerce_map_from(pol.parent())
True
```

Cette architecture multi-bases est classique dans Sage et utilisée par de nombreuses implantations algébriques. Cependant, dans le cas des polynômes, nous devons gérer une difficulté supplémentaire : la gestion du multivarié. En effet, un des besoins du programme est de pouvoir travailler en un nombre quelconque de variables sans le préciser *à priori*. Pour cela, on utilise à nouveau le système de conversion et des algèbres abstraites.

```
sage: A = AbstractPolynomialRing(QQ); A
The abstract ring of multivariate polynomials on x over Rational
   Field
sage: m = A.monomial_basis(); m
The ring of multivariate polynomials on x over Rational Field on
   the monomial basis
sage: m3 = m.finite_basis(3); m3
The ring of multivariate polynomials on x over Rational Field with
    3 variables on the monomial basis
sage: F3 = A.finite_polynomial_ring(3); F3
The abstract ring of multivariate polynomials on x over Rational
   Field with 3 variables
sage: pol = A.an_element(); pol
x[0, 0, 0] + 2*x[1, 0, 0] + x[1, 2, 3] + 3*x[2, 0, 0]
sage: pol.parent() == m3
True
sage: pol.parent() == F3.monomial_basis()
True
```

Les réalisations directes de l'algèbre abstraite des polynômes sont aussi des algèbres abstraites. Pour une réalisation concrète, il faut la donnée d'une *base* et d'un *nombre de variables*. Chaque parent concret est créé à la volée par son parent

abstrait au moment de la création du polynôme. On crée aussi des conversions entre les bases concrètes en différents nombres de variables.

```
sage: A = AbstractPolynomialRing(QQ)
sage: m = A.monomial_basis()
sage: m3 = m.finite_basis(3)
sage: m4 = m.finite_basis(4)
sage: m3.has_coerce_map_from(m4)
False
sage: m4.has_coerce_map_from(m3)
True
```

Sur cet exemple, on peut lire qu'il existe une conversion automatique des polynômes en 3 variables vers les polynômes en 4 variables mais pas l'inverse. On a illustré cette structure dans la figure 5.2.

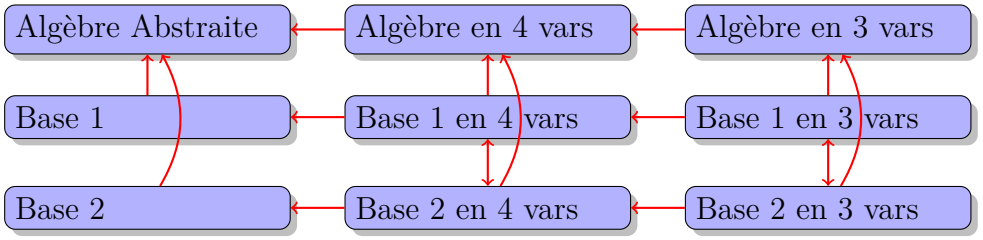


FIGURE 5.2 – Structure d'algèbre abstraite avec plusieurs bases et plusieurs variables. Les conversions sont représentées par des flèches rouges.

Cette structure interne est transparente. Les conversions automatiques permettent une utilisation fluide du programme.

Notre implantation comporte nativement les bases des polynômes décrites dans le chapitre 3. Cependant, il est tout à fait possible de rajouter des bases. En effet, en dehors des bases monomiales, toutes les bases sont définies sur le même principe. Elles héritent de la classe `LinearBasisOnVectors` et leurs éléments sont indexés par les éléments de l'espace ambiant du système de racine d'un groupe de Coxeter (qu'on peut tout simplement considérer comme des vecteurs). Seule est définie la fonction qui à un vecteur associe son développement en monôme. À partir de cette fonction, on crée le morphisme qui envoie les éléments de la nouvelle base sur les monômes. En créant un élément de la classe `LinearBasisOnVectors`, on crée une nouvelle base dotée de toute la structure décrite précédemment. On en a donné un exemple paragraphe 5.1.3. Il est possible de spécifier que le changement de base vers les monômes est triangulaire et dans ce cas, l'inversion se fera automatiquement et le morphisme inverse sera ajouté comme conversion. Malheureusement, la façon dont sont implantés les morphismes en Sage ne permet pas encore d'inverser les matrices graduées non triangulaires car Sage ne précalcule pas la matrice.

Grâce au graphe de conversion de Sage, bien qu'on ne définisse que la conversion d'une base donnée vers les monômes, toutes les conversions entre bases se font automatiquement. Pour créer une nouvelle base, l'utilisateur n'a donc qu'à écrire

le code entièrement spécifique à sa base et pourra utiliser l'ensemble du système mis en place.

### 5.2.3 Opérateurs : python, un langage dynamique

Les objets fondamentaux dans tous les changements de bases sur lesquels nous travaillons sont les opérateurs de différences divisées que nous avons définis au paragraphe 3.1. Un opérateur se définit par une fonction qui prend en paramètre un vecteur et un entier $i$ et qui lui associe une somme de monômes. A partir de ces fonctions nous créons des objets `Morphisme` de Sage qu'on peut appliquer à un polynôme. Le fait d'utiliser un objet `Morphisme` plutôt qu'une simple méthode permet d'utiliser des fonctions préexistantes comme la multiplication des morphismes et la création à partir d'une méthode sur la base.

La seule donnée dont nous avons besoin est donc la fonction définissant l'opérateur. Pour des raisons techniques, cette fonction est définie au sein d'une classe interne à chaque base `_divided_difference_wrapper`. Une instance de cette classe contient le paramètre $i$, la base dans laquelle vit le polynôme et éventuellement d'autres paramètres nécessaires à la différence divisées (par exemple, le type). On détaille ici le processus sur un exemple.

```
sage: A = AbstractPolynomialRing(QQ)
sage: m = A.monomial_basis()
sage: m3 = m.finite_basis(3)
sage: w = m._divided_difference_wrapper(m3,2)
sage: pol = m[2,1,4]; pol
x[2, 1, 4]
sage: key = list(pol)[0][0]; key
[2, 1, 4]
sage: w.divided_difference_on_basis(key)
-x[2, 1, 3] - x[2, 2, 2] - x[2, 3, 1]
sage: w.isobaric_divided_difference_on_basis(key)
-x[2, 2, 3] - x[2, 3, 2]
```

Pour créer `w`, on a donné en paramètre le parent du polynôme `m3` et `i=2`. L'objet `w` contient les méthodes sur la base utilisées pour créer les morphismes.

En fait, l'ensemble des méthodes est défini uniquement pour la base des monômes indexés par des éléments de l'espace ambiant d'un système de racine. Pour calculer la différence divisée, on utilise les opérations déjà existantes sur les groupes de Coxeter en se basant sur les formules décrites paragraphe 3.1.3. Cette méthode permet un algorithme qui ne dépend pas du type.

```
@cached_method
def divided_difference_on_basis(self,key):
        i = self._i
        keys = self._module.basis().keys()
        n = key.scalar(keys.simple_coroot(i))
        if n >= 0:
                return self._module.sum_of_monomials((keys(key-(j)
                   *keys.simple_root(i)-keys.basis()[i-1]) for j
```

```
                in xrange(n)))
        else:
                return -self.divided_difference_on_basis(keys.
                    simple_reflection(i)(key))
```

On remarque que l'on utilise ici le mot clé `cached_method`, c'est ce qui correspond à l'option `remember` de Maple. Elle permet de ne pas effectuer plusieurs fois le même calcul.

Les autres bases possèdent aussi une classe `_divided_difference_wrapper` mais font le plus souvent appel aux méthodes de la base monomiale de l'espace ambiant. Pour créer un opérateur, on vérifie d'abord si la méthode correspondante existe au sein de `_divided_difference_wrapper` et si oui on l'utilise, si non, on effectue une conversion du polynôme pour pouvoir utiliser la différence divisée par défaut. Voilà par exemple la méthode spécifique de la différence divisée des polynômes de Schubert.

```
@cached_method
def divided_difference_on_basis(self,key):
        i = self._i
        if(key[i-1] > key[i]):
                key2 = [key[j] for j in xrange(self._module.nb_
                    variables())]
                key2[i-1], key2[i] = key2[i], key2[i-1]-1
                return self._module(key2)
        return self._module.zero()
```

Dans le cas d'un polynôme de Schubert, la différence divisée correspond par définition à l'échange de deux composantes du vecteur. Il n'est donc pas nécessaire de développer le polynôme pour l'appliquer.

Comme nous l'avons indiqué dans le titre de cette section, Python est un langage dynamique. Cela signifie que l'on peut rajouter des méthodes "à la volée" lors de l'exécution du programme. C'est ce qui permet d'ajouter des opérateurs avec la méthode `add_operator` dont nous avons donné un exemple paragraphe 5.1.2. Le code se résume la fonction suivante.

```
def add_operator(self, name, method):
        setattr(self._divided_difference_wrapper, name +"_on_basis
            ", method)
```

La méthode prend en paramètre un nom qui sera celui à partir duquel on appellera l'opérateur et une méthode du type de celles que nous avons présentées plus haut. On ajoute simplement un nouvel attribut à la classe interne de la base.

## 5.3 Applications avancées

### 5.3.1 Degrés projectifs des variétés de Schubert

Comme nous l'avons vu au paragraphe 3.3.1, le produit sur les polynômes de Schubert s'interprète géométriquement en tant que produit dans l'anneau de

cohomologie de la variété de drapeau. Dans [Vei96], Veigneau calcule avec le logiciel ACE les degrés projectifs des variétés de Schubert. On peut effectuer un calcul similaire avec notre implantation.

Le degré projectif $d(X)$ d'une sous-variété $X \subset \mathbb{P}^M$ de codimension $k$ est le nombre d'intersections entre $X$ et un hyperplan générique de dimension $k$. Soit $\sigma \in \mathfrak{S}_n$ et $X_\sigma$ une cellule de Schubert de la variété de drapeau $\mathcal{F}_n = \mathcal{F}(\mathbb{C}^n)$ plongé dans l'espace projectif $\mathbb{P}^M$ par le plongement de Plücker (avec $M = 2^N - 1$ où $N = \frac{n(n-1)}{2}$ est la dimension de $\mathcal{F}_n$). Le degré projectif de la variété $X_\sigma$, $d(X_\sigma)$ est donné par un calcul sur les polynômes de Schubert. Soit $v$, le code de Lehmer de $\sigma$ et $h = (n-1)x_1 + (n-2)x_2 + \ldots + x_{n-1}$ la classe d'une section hyperplane. On a que $d(X_\sigma)$ est le coefficient de $Y_{[n-1,n-2,\ldots,0]}$ dans $Y_v h^{N-\ell(\sigma)}$ [Las82]. Il est donné par la fonction suivante.

```
def proj_deg(perm):
    n = len(perm)
    d = n*(n-1)/2 - perm.length()

    # we create the polynomial ring and the bases
    A = AbstractPolynomialRing(QQ)
    Schub = A.schubert_basis_on_vectors()

    # we compute the product
    h = sum( [(n-i) * A.var(i) for i in xrange(1,n)])
    res = Schub( h**d * Schub(perm.to_lehmer_code()))

    # we return the coefficient
    key = [n-i for i in xrange(1,n+1)]
    return res[key]
```

Par exemple, si $\sigma = 2143$, le degré projectif de la variété de Schubert $X_\sigma$ est 78.

```
sage: p = Permutation([2,1,4,3])
sage: proj_deg(p)
78
```

On peut aussi effectuer le calcul directement et lire le résultat sur le polynôme.

```
sage: A = AbstractPolynomialRing(QQ)
sage: Schub = A.schubert_basis_on_vectors()
sage: m = A.monomial_basis()
sage: pol = Schub( (3*m[1] + 2*m[0,1] + m[0,0,1])^4 * Schub
   [1,0,1,0])
sage: pol
8*Y(1, 1, 4, 0) + 23*Y(1, 2, 3, 0) + 24*Y(1, 3, 2, 0) + 39*Y(1, 4,
    1, 0) + 15*Y(1, 5, 0, 0) + Y(1, 0, 5, 0) + 48*Y(2, 1, 3, 0) +
    101*Y(2, 2, 2, 0) + 117*Y(2, 3, 1, 0) + 84*Y(2, 4, 0, 0) +
   12*Y(2, 0, 4, 0) + 173*Y(3, 1, 2, 0) + 78*Y(3, 2, 1, 0) + 147*
   Y(3, 3, 0, 0) + 53*Y(3, 0, 3, 0) + 283*Y(4, 1, 1, 0) + 171*Y
   (4, 2, 0, 0) + 96*Y(4, 0, 2, 0) + 93*Y(5, 1, 0, 0) + 176*Y(5,
   0, 1, 0) + 80*Y(6, 0, 0, 0)
sage: pol[3,2,1,0]
78
```

On peut utiliser la fonction pour calculer tous les degrés projectifs pour toutes les permutations de taille 4.

```
sage: degrees = {}
sage: for perm in Permutations(4):
....:     degrees[perm] = proj_deg(perm)
....:
sage: degrees
{[2, 1, 4, 3]: 78, [1, 3, 4, 2]: 48, [3, 2, 4, 1]: 3, [3, 1, 2,
   4]: 48, [4, 2, 1, 3]: 3, [1, 4, 2, 3]: 46, [3, 2, 1, 4]: 16,
   [4, 1, 3, 2]: 3, [2, 3, 4, 1]: 6, [3, 4, 2, 1]: 1, [1, 2, 3,
   4]: 720, [1, 3, 2, 4]: 280, [2, 4, 3, 1]: 3, [2, 3, 1, 4]: 46,
    [3, 4, 1, 2]: 2, [4, 2, 3, 1]: 1, [1, 4, 3, 2]: 16, [4, 1, 2,
    3]: 6, [2, 4, 1, 3]: 12, [4, 3, 1, 2]: 1, [4, 3, 2, 1]: 1,
   [3, 1, 4, 2]: 14, [2, 1, 3, 4]: 220, [1, 2, 4, 3]: 220}
```

### 5.3.2 Déterminants de fonctions de Schur

Les polynômes de Schubert Grassmannien sont ceux indexés par un vecteur $v$ tel que $v_1 \leq v_2 \leq \cdots \leq v_n$. On a vu dans le chapitre 3 que ce sont des polynômes symétriques et qu'ils correspondent aux fonctions de Schur dans le cas des Schubert simples. Plus précisément, la matrice de transition entre les polynômes de Schubert grassmanniens doubles et les fonctions de Schur est unitriangulaire.

```
sage: A = AbstractPolynomialRing(QQ)
sage: Schub =  A.schubert_basis_on_vectors()
sage: pol = Schub[1,2]
sage: pol.expand()
x(1, 2) + x(2, 1)
sage:
sage: D = DoubleAbstractPolynomialRing(QQ)
sage: DSChub = D.double_schubert_basis_on_vectors()
sage: pol = DSchub[1,2]
sage: pol
y[0]*YY(1, 2)
sage: Schub = D.schubert_basis_on_vectors()
sage: Schub(pol)
y[0]*Yx(1, 2) + (-y(2,1,0)-y(2,0,1))*Yx(0, 0) + (-y(1,0,0)-y
   (0,1,0)-y(0,0,1))*Yx(1, 1) + (y(1,1,0)+y(1,0,1)+y(2,0,0))*Yx
   (0, 1) + (-y[1])*Yx(0, 2)
```

Cela nous permet de calculer des déterminants de fonctions de Schur en les remplaçant par des polynômes de Schubert et en spécialisant arbitrairement les variables $y$. Par exemple, on peut calculer

$$|s_\mu(A)|_{\mu\subseteq \begin{smallmatrix}\square\\\square\end{smallmatrix}}, \tag{5.1}$$

où $A \in [\{x_1, x_2\}, \{x_1, x_3\}, \{x_2, x_3\}]$ et prouver que l'on obtient $\prod_{j>i}(x_j - x_i)$. Tout d'abord, remplaçons $s_\mu$ par

$$|Y_u(A, y)|_{u=00,01,11} \tag{5.2}$$

et spécialisons en $y_1 = x_1$, $y_2 = x_2$. Dans ce cas, le déterminant devient

$$\begin{vmatrix} 1 & 1 & 1 \\ 0 & x_3 - x_2 & x_3 - x_1 \\ 0 & 0 & (x_3 - x_1)(x_2 - x_1) \end{vmatrix} \tag{5.3}$$

et nous donne le résultat. La fonction suivante calcule cette matrice.

```
def compute_matrix(variables, alphabets, indices):

    n = len(indices)

    #Initial definitions
    D = DoubleAbstractPolynomialRing(QQ)
    DSchub = D.double_schubert_basis_on_vectors()

    result_matrix = []

    for u in indices:
        line = []

        #the expansion on the double schubert will allow us to
           compute the result
        pu = DSchub(u).expand()

        for a in alphabets:
                #we apply our polynomial on alphabets and
                   specialize the y
            coeff = pu.to_expr(alphabet = a, alphabety = variables
               )
            line.append(coeff)
        result_matrix.append(line)

    return Matrix(result_matrix)
```

On peut vérifier le résultat de (5.3).

```
sage: var('x1,x2,x3')
(x1, x2, x3)
sage: variables = (x1,x2,x3)
sage: alphabets = [[x1,x2],[x1,x3],[x2,x3]]
sage: indices =  [[0,0],[0,1],[1,1]]
sage: res = compute_matrix(variables, alphabets, indices)
sage: res
[1                          1                          1]
[0                   -x2 + x3                   -x1 + x3]
[0                          0 x1^2 - x1*x2 - x1*x3 + x2*x3]
sage: det = res.determinant()
sage: det
-(x2 - x3)*(x1^2 - x1*x2 - x1*x3 + x2*x3)
sage: factor(det)
-(x2 - x3)*(x1 - x3)*(x1 - x2)
```

### 5.3.3 Produit des polynômes de Grothendieck

Notre implantation peut être utilisée pour effectuer les calculs nécessaires au chapitre 4. Elle permet de vérifier le théorème 4.0.4 sur des exemples.

Tout d'abord, vérifions sur un exemple le théorème 4.0.2. On a besoin d'effectuer un produit de polynôme et donc de convertir depuis la base des monômes vers la base Grothendieck. Dans notre implantation, cela ne peut se faire que sur les polynômes de Grothendieck simples. On calcule $G_\sigma(1-x_1)(1-x_2)(1-x_3)(1-x_4)$ (le changement de variable remplace $x_i^{-1}$ par $1-x_i$). Par ailleurs, on travaille dans un espace quotienté par un idéal et on supprime donc les termes inutiles.

```
def apply_operators(perm, k):
    perm = Permutation(perm)
    code = perm.to_lehmer_code()

    A = AbstractPolynomialRing(QQ)
    Groth = A.grothendieck_positive_basis_on_vectors()
    pol= Groth( code )

    factor = prod( [A.one() - A.var(i+1) for i in xrange(k)] )

    pol =  pol*factor

    #suppression des termes inutiles
    res = pol.parent().zero()
    for (k,c) in pol:
        m = pol.nb_variables() - 1
        for i in xrange(pol.nb_variables()):
            if(k[i]>m): break
            m = m-1
        else:
            res+= pol.parent().term(k,c)

    return res
```

Pour $\sigma = 136254$, on obtient

```
sage: perm = Permutation([1,3,6,2,5,4])
sage: res1 = apply_operators(perm, 4)
sage: res1
-G(0, 1, 3, 1, 1, 0) - G(0, 1, 3, 2, 0, 0) + G(0, 1, 3, 2, 1, 0) +
    G(0, 1, 3, 0, 1, 0) + G(0, 2, 3, 1, 1, 0) + G(0, 2, 3, 2, 0,
   0) - G(0, 2, 3, 2, 1, 0) - G(0, 2, 3, 0, 1, 0) - G(0, 3, 3, 0,
    0, 0) + G(0, 3, 3, 1, 0, 0) - G(0, 3, 3, 1, 1, 0) - G(0, 3,
   3, 2, 0, 0) + G(0, 3, 3, 2, 1, 0) + G(0, 3, 3, 0, 1, 0)
```

Soit en terme de permutations

```
from sage.combinat import permutation
sage: perms = [ permutation.from_lehmer_code(key) for (key,c) in
   res1.elements()]
sage: perms
[[1, 5, 6, 3, 4, 2],
```

```
 [1, 4, 6, 5, 3, 2],
 [1, 3, 6, 2, 5, 4],
 [1, 5, 6, 2, 4, 3],
 [1, 5, 6, 3, 2, 4],
 [1, 5, 6, 4, 3, 2],
 [1, 3, 6, 5, 2, 4],
 [1, 3, 6, 4, 5, 2],
 [1, 4, 6, 5, 2, 3],
 [1, 4, 6, 2, 5, 3],
 [1, 5, 6, 2, 3, 4],
 [1, 5, 6, 4, 2, 3],
 [1, 4, 6, 3, 5, 2],
 [1, 3, 6, 5, 4, 2]]
```

On peut vérifier que ce sont bien les mêmes qui apparaissent dans le calcul suivant où l'on utilise le résultat du théorème 4.0.2.

```
sage: A = AbstractPolynomialRing(QQ)
sage: K = A.demazure_basis_on_vectors()
sage: pol = K[6,5,4,3,2,1]
sage: res2 = pol.apply_reduced_word([3,4,5,2,3,4],method="hatpi")
sage: res2 = res2.apply_reduced_word([3,2,1,2],method="pi")
sage: res2
K(1, 3, 6, 2, 5, 4) - K(1, 3, 6, 4, 5, 2) - K(1, 3, 6, 5, 2, 4) +
   K(1, 3, 6, 5, 4, 2) - K(1, 4, 6, 2, 5, 3) + K(1, 4, 6, 3, 5,
   2) + K(1, 4, 6, 5, 2, 3) - K(1, 4, 6, 5, 3, 2) - K(1, 5, 6, 2,
    3, 4) + K(1, 5, 6, 2, 4, 3) + K(1, 5, 6, 3, 2, 4) - K(1, 5,
   6, 3, 4, 2) - K(1, 5, 6, 4, 2, 3) + K(1, 5, 6, 4, 3, 2)
```

On généralise ce second calcul dans une fonction.

```
def product_permutations(perm, k):

    #Initial definitions
    A = AbstractPolynomialRing(QQ)
    K = A.demazure_basis_on_vectors()
    n = len(perm)

    # permutations
    omega = Permutation([n-i for i in xrange(n)])
    zeta = [perm[i] for i in xrange(k)]
    zeta.sort()
    zeta.reverse()
    zeta2 = [perm[i] for i in xrange(k,n)]
    zeta2.sort()
    zeta2.reverse()
    zeta.extend(zeta2)
    zeta = Permutation(zeta)

    #paths
    hatpis = (zeta * omega).reduced_word()
    pis = (perm * zeta.inverse()).reduced_word()

    #computation
```

```
    pol = K(list(omega))
    if(len(hatpis)!=0):
        pol = pol.apply_reduced_word(hatpis, method="hatpi")
    if(len(pis)!=0):
        pol = pol.apply_reduced_word(pis, method="pi")

    # perm set from polynomial
    perms = [Permutation(key) for (key,c) in pol.elements()]
    return perms
```

Sur l'exemple, cela donne

```
sage: perm = Permutation([1,3,6,2,5,4])
sage: perms = product_permutations(perm, 4)
sage: perms
[[1, 5, 6, 2, 4, 3],
 [1, 4, 6, 3, 5, 2],
 [1, 5, 6, 4, 3, 2],
 [1, 3, 6, 5, 4, 2],
 [1, 4, 6, 5, 2, 3],
 [1, 4, 6, 2, 5, 3],
 [1, 5, 6, 3, 2, 4],
 [1, 5, 6, 2, 3, 4],
 [1, 5, 6, 4, 2, 3],
 [1, 4, 6, 5, 3, 2],
 [1, 3, 6, 4, 5, 2],
 [1, 3, 6, 2, 5, 4],
 [1, 5, 6, 3, 4, 2],
 [1, 3, 6, 5, 2, 4]]
```

L'application des opérateurs est une opération formelle sur les vecteurs. Dans notre implantation, elle est directement définie au niveau des polynômes clés : aucun changement de base n'est nécessaire. Ce second calcul est donc beaucoup plus rapide que le premier : on passe de 10 secondes à un dixième de seconde.

```
sage: time( apply_operators(perm, 4))
CPU times: user 10.60 s, sys: 0.02 s, total: 10.62 s
Wall time: 10.73 s
-G(0, 1, 3, 1, 1, 0) - G(0, 1, 3, 2, 0, 0) + G(0, 1, 3, 2, 1, 0) +
    G(0, 1, 3, 0, 1, 0) + G(0, 2, 3, 1, 1, 0) + G(0, 2, 3, 2, 0,
   0) - G(0, 2, 3, 2, 1, 0) - G(0, 2, 3, 0, 1, 0) - G(0, 3, 3, 0,
    0, 0) + G(0, 3, 3, 1, 0, 0) - G(0, 3, 3, 1, 1, 0) - G(0, 3,
   3, 2, 0, 0) + G(0, 3, 3, 2, 1, 0) + G(0, 3, 3, 0, 1, 0)
sage: time(product_permutations(perm,4))
CPU times: user 0.09 s, sys: 0.00 s, total: 0.09 s
Wall time: 0.09 s
[[1, 5, 6, 2, 4, 3],
 [1, 4, 6, 3, 5, 2],
 [1, 5, 6, 4, 3, 2],
 [1, 3, 6, 5, 4, 2],
 [1, 4, 6, 5, 2, 3],
 [1, 4, 6, 2, 5, 3],
 [1, 5, 6, 3, 2, 4],
 [1, 5, 6, 2, 3, 4],
```

```
 [1, 5, 6, 4, 2, 3],
 [1, 4, 6, 5, 3, 2],
 [1, 3, 6, 4, 5, 2],
 [1, 3, 6, 2, 5, 4],
 [1, 5, 6, 3, 4, 2],
 [1, 3, 6, 5, 2, 4]]
```

À présent, vérifions le résultat du théorème 4.0.4. La fonction suivante génère l'intervalle de Bruhat entre deux permutations.

```
def bruhat_interval(p1, p2, res = None):
    if res is None:
        res = []
    if(p1.bruhat_lequal(p2)):
        res.append(p1)
        if(p1.length() < p2.length()):
            succs = p1.bruhat_succ()
            for p in succs:
                if p not in res:
                    bruhat_interval(p,p2,res)
        return res
    else:
        return None
```

On teste que l'ensemble obtenu par l'application des opérateurs est bien un intervalle de l'ordre de Bruhat.

```
def is_result_interval(perm, k):
    perms = product_permutations(perm, k)
    #get 1 max permutation
    pmax = perm
    for p in perms:
        if(pmax.bruhat_lequal(p)):
            pmax = p

    interval = bruhat_interval(perm,pmax)
    return set(interval) == set(perms)
```

Sur l'exemple précédent, cela donne :

```
sage: perm = Permutation([1,3,6,2,5,4])
sage: is_result_interval(perm,4)
True
```

On peut tester sur l'ensemble des permutations de taille 4 et 5.

```
def test_all_perms(size):
    for p in Permutations(size):
        for k in xrange(1,size):
            if not is_result_interval(p,k):
                print p,k
                return False
    return True
sage: test_all_perms(4)
True
```

```
sage: test_all_perms(5)
True
```