Séminaire Lotharingien de Combinatoire **66** (2011), Article B66z

MULTIVARIATE POLYNOMIALS IN SAGE

VIVIANE PONS

Abstract. We have developed a patch implementing multivariate polynomials seen as a multi-base algebra. The patch is to be released into the software Sage and can already be found within the Sage-Combinat distribution. One can use our patch to define a polynomial in a set of indexed variables and expand it into a linear basis of the multivariate polynomials. So far, we have the Schubert polynomials, the Key polynomials of types A, B, C, or D, the Grothendieck polynomials and the non-symmetric Macdonald polynomials. One can also use a double set of variables and work with specific double-linear bases like the double Schubert polynomials or double Grothendieck polynomials. Our implementation is based on a definition of the basis using *divided difference* operators and one can also define new bases using these operators.

Introduction

Multivariate polynomials and their bases appear in many combinatorial problems and one often needs to define a polynomial as a formal sum of elements that live in a specified basis. The usual implementation of multivariate polynomials is done as a tensor product of polynomials in one variable. But one can not consider the variables all together: all bases have to be defined by a product of bases of polynomials in one variable. It appeared to us that a clear and handy implementation of multivariate polynomials from a combinatorial point of view would be useful not only to our work but to the community.

Our approach is based on *divided difference* operators and their interpretation in terms of linear bases of the multivariate polynomials algebra as explained by Lascoux [2]. We define simple operators of types A, B, C, and D in Section 2.1 and use them to create the *divided difference* operators in Section 2.2. We then explain in Section 2.3 how these operators allow us to define linear bases of the multivariate polynomials.

Our software has been implemented in Sage, we explain our choice in Section 1. The full description of the implemented features with code examples can be found in Section 3. The development process is not finished at the time of submission of this paper and what the program now needs most is to be tested by many users so that bugs can be reported and suggestions be made on how to improve the features.

1. Choosing Sage

Sage is a mathematics software created in 2005. It is free and open source which is the main reason why we have chosen it. It is developed by many researchers around the world and one can join the very lively community who works specifically on combinatorics within the Sage-Combinat project. The Sage-Combinat community has been created in 2001 and was previously known as Mupad-Combinat. It has moved to Sage in 2008 keeping its main purpose: developing specific tools for algebraic combinatorics and sharing the programs among researchers.

From the beginning, we wanted not only to develop in Sage but to be part of the main project by adding our implementation to the software. The program is still in test mode within the Sage-combinat distribution (see the installation process in Section 3.1).

Working in Sage also allowed us to use previous work and structures developed by the community. We worked a lot with Weyl Groups which had already been implemented in Sage. We also used the standard implementation of multi-base algebras as a base for our own work.

And finally, we hope that being part of Sage will help our software to spread within the community and thus become useful to the largest possible community.

2. Multi-base polynomials

2.1. Type A, B, C, D operators. At first, we need to define simple operations on vectors of integers. Let $v \in \mathbb{Z}^n$, we have the following operators corresponding to the root system of respective types A, B, C, D:

$$vs_i = (\ldots, v_{i+1}, v_i, \ldots) \qquad \text{for } 1 \leq i < n \tag{1}$$

$$vs_i^B = vs_i^C = (\ldots, -v_i, \ldots) \qquad \text{for } 1 \leq i \leq n \tag{2}$$

$$vs_i^D = (\ldots, -v_i, -v_{i-1}, \ldots) \qquad \text{for } 2 \leq i \leq n \tag{3}$$

The group generated by $s_1, \ldots, s_{n-1}$ (respectively $s_1, \ldots, s_{n-1}, s_n^B$, and $s_1, \ldots,$ s_{n-1}, s_n^D) is the Weyl group of type A (respectively B or C, D). The operators satisfy the *braid relations*:

$$s_i s_{i+1} s_i = s_{i+1} s_i s_{i+1} \qquad \text{and } s_i s_j = s_j s_i,\ |i-j| \neq 1 \tag{4}$$

$$s_{n-1} s_n^B s_{n-1} s_n^B = s_n^B s_{n-1} s_n^B s_{n-1} \qquad \text{and } s_i s_n^B = s_n^B s_i,\ i \leq n-2 \tag{5}$$

$$s_{n-2} s_n^D s_{n-2} = s_n^D s_{n-2} s_n^D \qquad \text{and } s_i s_n^D = s_n^D s_i,\ i \neq n-2 \tag{6}$$

The orbit of the vector $[1, 2, \ldots, n]$ consists of all permutations of $1, 2, \ldots, n$, i.e., S_n, for type A, all signed permutations for type B, C, and all signed permutations with an even number of minus signs for type D. The elements of the different groups can be denoted by these objects. In the same way, elements of these groups can be seen as a product of operators s_i, called a *decomposition*. When the product is of minimal length, it is called a *reduced decomposition*.

2.2. Action on polynomials. We now have a natural action of the Weyl groups on polynomials. Indeed, let $x = (x_1, x_2, \ldots, x_n)$ be a set of variables and for $v \in \mathbb{Z}^n$, let x^v stand for the monomial

$$x_1^{v_1} x_2^{v_2} \ldots x_n^{v_n}. \tag{7}$$

A polynomial in the variables $x_1, \ldots, x_n$ can therefore be seen as a formal sum of vectors and the action of the operator s_i becomes an action on polynomials:

$$x^v s_i = x^{v s_i}. \tag{8}$$

From these simple operators s_i, we can now define the *divided difference* operators. For type A, we have:

$$f\partial_i := \frac{f - f^{s_i}}{x_i - x_{i+1}} \tag{9}$$

$$f\pi_i := \frac{x_i f - x_{i+1} f^{s_i}}{x_i - x_{i+1}} = f.(x_i\partial_i) \tag{10}$$

$$f\hat{\pi}_i := \frac{(f - f^{s_i})x_{i+1}}{x_i - x_{i+1}} = f.(\pi_i - 1) \tag{11}$$

$$fT_i := f.(\pi_i(t_1 + t_2) - s_i t_2) \tag{12}$$

for $1 \leq i \leq n-1$. ∂_i is the *Newton divided difference*, π_i and $\hat{\pi}_i$ are the *isobaric divided differences* and T_i is the generator of the *Hecke algebra* $\mathcal{H}_2$. As the s_i satisfy the braid relations, all the above operators do.

If $v_i > v_{i+1}$, the *Newton divided difference* ∂_i can be seen as an operator decrementing the vector degree and summing over all the intermediate monomials between $x^{(\dots,v_i-1,v_{i+1},\dots)}$ and $x^{(\dots,v_{i+1},v_i-1,\dots)}$. For example,

$$x^{(4,1)}\partial_1 = x^{(3,1)} + x^{(2,2)} + x^{(1,3)}. \tag{13}$$

When $v_i < v_{i+1}$, one just needs to switch v_i and v_{i+1}, multiply by -1 and do the previous operation. When $v_i = v_{i+1}$, the result is 0. This description can be used to give a more general, type-free definition. We can see the vectors indexing the monomials as elements of the ambient space of the root system of type A_{n-1}. The above ∂_i operation can then be seen as a formal sum of vectors, adding factors of the simple root $(\dots, -1, 1 \dots)$ to the original vector. The sign of $v_i - v_{i+1}$ is given by the scalar product between the vector and the i^{th} simple coroot of the ambient space. This definition is equivalent to the previous one. We can use it to define our divided differences in types B, C, and D, using the root systems of respective types B_n, C_n, and D_n. Compared to type A, we add the n^{th} simple root and coroot whose definition depends on the type and create the n^{th} divided difference operator:

$$\partial_n^B = \frac{1 - s_n^B}{x_n^{\frac{1}{2}} - x_n^{-\frac{1}{2}}} \tag{14}$$

$$\partial_n^C = \frac{1 - s_n^C}{x_n - x_n^{-1}} \tag{15}$$

$$\partial_n^D = \frac{1 - s_n^D}{x_{n-1}^{-1} - x_n} \tag{16}$$

The same construction can be done with the isobaric divided differences π and $\hat{\pi}$.

2.3. **Linear bases.** We can now use these operators to define linear bases of the ring of multivariate polynomials. Let $(x_1, x_2, \dots, x_n)$ and $(y_1, y_2, \dots, y_n)$ be two sets of variables and λ a partition of length n, i.e., $\lambda_1 \geq \lambda_2 \geq \ldots \geq \lambda_n$. We then define *dominant*

Schubert polynomials (respectively *Grothendieck polynomials* and *Key polynomials*) by

$$Y_\lambda := \prod_{i=1}^{n} \prod_{j=1}^{\lambda_i} (x_i - y_j), \tag{17}$$

$$G_\lambda := \prod_{i=1}^{n} \prod_{j=1}^{\lambda_i} (1 - y_j x_i^{-1}), \tag{18}$$

$$K_\lambda = \hat{K}_\lambda := x^\lambda. \tag{19}$$

We define Schubert polynomials to be all the non-zero images of the dominant Schubert polynomials under products of ∂_i and Grothendieck polynomials to be all the images of the dominant Grothendieck polynomials under products of π_i. Similarly, the two types of Key polynomials are defined by taking all the images under products of π_i or of $\hat{\pi}_i$ respectively. Since the operators satisfy relations, we cannot index the polynomials by the choice of the starting point and the sequence of operators used. Rather, we use weight vectors $v \in \mathbb{N}^n$, the recursive definition being

$$Y_{\dots,v_{i+1},v_i-1,\dots} = Y_v \partial_i \tag{20}$$

$$G_{\dots,v_{i+1},v_i-1,\dots} = G_v \pi_i \tag{21}$$

$$K_v \pi_i = K_{vs_i} \tag{22}$$

$$\hat{K}_v \hat{\pi}_i = \hat{K}_{vs_i}, \tag{23}$$

the inal vectors v satisfying $v_i > v_{i+1}$.

As the operators satisfy braids relations, the order one chooses to apply the recursive rule on a vector does not change the result. There are dominant polynomials in the images of a dominant polynomial in the Schubert and Grothendieck case; therefore, one has to check consistency, but this is easy. These families constitute triangular bases of the polynomials in $(x_1, \dots, x_n)$. And one can easily express an arbitrary polynomial in these bases by inverting a triangular matrix. When working with a single set of variables, one can specialize the y_i's to 0 (respectively to 1) and obtain simple Schubert polynomials (respectively Grothendieck polynomials), so that dominant polynomials become

$$Y_\lambda = x^\lambda, \tag{24}$$

$$G_\lambda = \prod_{i=1}^{n} (1 - x_i^{-1})^{\lambda_i}. \tag{25}$$

To work with positive exponents on the Grothendieck basis, one can also set $x_i = 1 - x_i^{-1}$. Both versions of the bases are available in our implementation.

Using the same method, we can also define non-symmetric Macdonald polynomials. In this case, there will be only one generator polynomial, i.e., $M_{0,0\dots,0} = 1$, and we will use both the T_i operator and a raising operator to increase the polynomial degree. The recursive rule is due to Sahi and Knop [1] and one can find its description in Lascoux [2].

One can also define type B, C, and D Key polynomials using the operators defined from the Weyl group of the given types as explained in Section 2.2. The corresponding families of polynomials will then be indexed by vectors in $\mathbb{Z}^n$ instead of $\mathbb{N}^n$ and become bases of the Laurent polynomials of $(x_1, \ldots, x_n)$, i.e., with both positive and negative exponents.

3. Software description

3.1. Installation process. Our software has been developed as a part of the Sage project. Nevertheless, as it is still in test mode, at the publication time of this paper, it is not yet available on the main Sage distribution, but one can use it within the Sage-Combinat distribution.

Step 1

If Sage is not already installed on a computer, please follow the instructions on the Sage website [6] to get the installation process corresponding to your system.

To get the latest version of our program, make sure you have the last version of Sage installed. You can upgrade your Sage version by running the following command inside your sage directory:

```
./sage -upgrade
```

The examples below work with Sage 4.7 and later versions.

Step 2

Install Sage-Combinat [5] by running the following command inside the sage directory:

```
./sage -combinat install
```

If you encounter problems, you find more information by visiting the Sage-Combinat website [5].

3.2. Define a polynomial. Sage programming is object-oriented. The object containing the main software methods is called *AbstractPolynomialRing*. One first needs to create this object:

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

Here, A represents the abstract algebra. To create an actual polynomial, we need a concrete basis.

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

m is a concrete basis and we shall use it to create polynomials. Both of the syntaxes presented below can be used.

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

$x[1,1,2]$ means $x^{(1,1,2)} = x_1^1 x_2^1 x_3^2$. One does not have to declare beforehand how many variables are to be used, it will be computed from the size of the vectors. To know on how many variables a polynomial is defined, one can look at its parent. It can also be changed if needed.

```
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
```

Now we have a polynomial object to work with. A polynomial will always be seen as a formal sum of vectors: it cannot be factorized. If two polynomials are multiplied, the result will always be expanded as a sum.

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

3.3. **Apply operators.** We can now apply the *divided differences* operators defined in Section 2.2.

```
sage: pol = m[1,1,2] + m[2,3]; pol
x[1, 1, 2] + x[2, 3, 0]
sage: pol.divided_difference(2)
-x[1, 1, 1] + x[2, 1, 1] + x[2, 2, 0] + x[2, 0, 2]
sage: pol.divided_difference_isobar(2)
x[2, 1, 2] + x[2, 2, 1] + x[2, 3, 0] + x[2, 0, 3]
```

By default, the operator type is A, but we can also apply B, C, and D operators.

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

We have seen in Section 2.2 that type B, C, and D operators were defined from the root systems of types B_n, C_n, and D_n, by the addition of the n^{th} simple root and coroot. For each type, only one new operator was created which was the n^{th} divided difference. But on the above example, applying the second divided difference on a polynomial in three variables gives a different result for types B, C, and D than for type A. Even though groups have been used to give a definition of our operators, we have extended it to obtain a definition only depending on the polynomial. From the root systems of type B_n, C_n, and D_n, we had

$$\partial_n^B = \frac{1 - s_n^B}{x_n^{\frac{1}{2}} - x_n^{-\frac{1}{2}}} \tag{26}$$

$$\partial_n^C = \frac{1 - s_n^C}{x_n - x_n^{-1}} \tag{27}$$

$$\partial_n^D = \frac{1 - s_n^D}{x_{n-1}^{-1} - x_n} \tag{28}$$

So we just set

$$\partial_i^B = \frac{1 - s_i^B}{x_i^{\frac{1}{2}} - x_i^{-\frac{1}{2}}} \tag{29}$$

$$\partial_i^C = \frac{1 - s_i^C}{x_i - x_i^{-1}} \tag{30}$$

$$\partial_i^D = \frac{1 - s_i^D}{x_{i-1}^{-1} - x_i} \tag{31}$$

with $1 \leq i \leq n$ for types B and C, and $2 \leq i \leq n$ for type D. These definitions allow us to study these operators for themselves and not relatively to their groups. As an example, this allows us to study the relations between ∂_i^B and ∂_{i+1}^B. It does not make sense from a group point of view but it remains an interesting question.

Of course, one may want to use the group based operators only and not the generalized ones. This is possible by using a different basis. The basis we have been using so far is called the *Monomial basis* and is not related to any group. One can use a basis which is directly related to a root system called the *Ambient space 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: pol = ma[1,1,2] + ma[2,3]
sage: pol
x(1, 1, 2) + x(2, 3, 0)
sage: pol.parent()
The ring of multivariate polynomials on x over Rational Field
   with 3 variables on the Ambient space basis of type A
sage: pol.divided_difference(2)
-x(1, 1, 1) + x(2, 1, 1) + x(2, 2, 0) + x(2, 0, 2)
```

Note that *Ambient Space basis* is very close to the *Monomial basis* but the polynomial contains its type within its parent. It is related to a root system and only operators defined by this root system can be applied.

```
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: pol = mb[1,1,2] + mb[2,3]
```

```
sage: pol.divided_difference(2)
-x(1, 1, 1) + x(2, 1, 1) + x(2, 2, 0) + x(2, 0, 2)
sage: pol.divided_difference(3)
x(1, 1, 0) + x(1, 1, -2) + x(1, 1, -1) + x(1, 1, 1)
```

Conversions between the *Monomial basis* and the *Ambient space bases* can be carried out easily:

```
sage: pol = m[1,1,2] + m[2,3]; pol
x[1, 1, 2] + x[2, 3, 0]
sage: pol.parent()
The ring of multivariate polynomials on x over Rational Field
   with 3 variables on the monomial basis
sage: pol = ma(pol); pol
x(1, 1, 2) + x(2, 3, 0)
sage: pol.parent()
The ring of multivariate polynomials on x over Rational Field
   with 3 variables on the Ambient space basis of type A
sage: pol = mb(pol); pol
x(1, 1, 2) + x(2, 3, 0)
sage: pol.parent()
The ring of multivariate polynomials on x over Rational Field
   with 3 variables on the Ambient space basis of type B
```

Even though the objects seem similar, one must always be careful with which basis one is working as it will impact the result of the operations as soon as operators are used.

3.4. **Working with multi-bases.** We have already seen that our polynomials could be expressed on a different basis depending on which operations we wanted to make. But the *Monomial basis* as well as the *Ambient space bases* are just different versions of polynomials seen as sums of monomials. It is also possible to work with the linear bases we have defined in Section 2.3. Here is an example of the Schubert basis:

```
sage: A = AbstractPolynomialRing(QQ)
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)
```

It can be used to create a Schubert polynomial and convert it to the monomial basis.

```
sage: pol = Schub[1,2,2] + Schub[3,4]; pol
Y(1, 2, 2) + Y(3, 4, 0)
sage: pol.expand()
x(1, 2, 2) + x(2, 1, 2) + x(2, 2, 1) + x(3, 4, 0) + x(4, 3, 0)
sage: m(pol)
x[1, 2, 2] + x[2, 1, 2] + x[2, 2, 1] + x[3, 4, 0] + x[4, 3, 0]
sage: Schub(m[1,2,4] + m[2,3])
Y(1, 2, 4) - Y(1, 3, 3) - Y(1, 4, 2) - Y(2, 1, 4) + Y(2, 3, 0) +
    Y(2, 3, 2) + Y(2, 4, 1) + Y(3, 1, 3) - Y(3, 2, 0) - Y(3, 2,
   2) - Y(4, 2, 1) + Y(5, 1, 1)
```

One can multiply Schubert polynomials together and the result will be given in the same basis. However the program is converting the two polynomials into the monomial basis to multiply them and then convert the result back into the Schubert basis.

```
sage: pol1 = Schub[1,2,2] + Schub[3,4]
sage: pol2 = Schub[3,1,2]
sage: pol1 * pol2
Y(4, 3, 4) + Y(5, 2, 4) + Y(6, 5, 2) + Y(6, 6, 1) + Y(7, 4, 2) +
    Y(7, 5, 1)
```

We have other bases implemented. Below is an example of the Key polynomials. One can convert directly from Schubert to Key polynomials without using the monomial basis manually (it is automatically done by the program):

```
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: pol = K[2,1,4] + K[3,5,1];pol
K(2, 1, 4) + K(3, 5, 1)
sage: pol.expand()
x(2, 1, 4) + x(2, 2, 3) + x(2, 3, 2) + x(2, 4, 1) + x(3, 1, 3) +
    x(3, 2, 2) + x(3, 3, 1) + x(3, 5, 1) + x(4, 1, 2) + x(4, 2,
   1) + x(4, 4, 1) + x(5, 3, 1)
sage: Schub(pol)
Y(2, 1, 4) + Y(3, 5, 1) - Y(5, 1, 1)
sage: K(m[1,2,4] + m[2,3])
K(1, 2, 4) - K(1, 3, 3) - K(1, 4, 2) - K(2, 1, 4) + K(2, 3, 0) +
    K(2, 3, 2) + K(2, 4, 1) + K(3, 1, 3) - K(3, 2, 0) - K(3, 2,
   2) + K(4, 1, 2) - K(4, 2, 1)
sage: Khat = A.demazure_hat_basis_on_vectors()
sage: pol = Khat[2,1,4] + Khat[3,5,1];pol
^K(2, 1, 4) + ^K(3, 5, 1)
sage: pol.expand()
x(2, 1, 4) + x(2, 2, 3) + x(2, 3, 2) + x(3, 1, 3) + x(3, 2, 2) +
    x(3, 5, 1) + x(4, 4, 1)
sage: Schub(pol)
Y(2, 1, 4) - Y(2, 4, 1) + Y(3, 5, 1) - Y(4, 1, 2) + Y(4, 2, 1) -
    Y(5, 1, 1) - Y(5, 3, 1)
sage: Khat(m[1,2,4] + m[2,3])
^K(1, 2, 4) - ^K(1, 3, 3) + ^K(2, 3, 0) + ^K(2, 3, 2)
```

The key polynomials are also defined in type B, C, and D.

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

```
x(1, 2, 0) + x(1, 2, -2) + x(1, 2, -1) + x(1, 2, 1) + x(1, 2, 2)
    + x(2, 1, 0) + x(2, 1, -2) + x(2, 1, -1) + x(2, 1, 1) + x(2,
    1, 2) + x(2, 2, 0) + x(2, 2, -1) + x(2, 2, 1)
sage: pol = m[-2,1,1] + m[1,-1,1]; pol
x[-2, 1, 1] + x[1, -1, 1]
sage: K(pol)
K(0, 0, 0) + K(-2, 1, 1) - K(-1, 1, 1) - K(-1, 1, 2) - K(-1, 0,
   1) - 2*K(1, 0, 0) - K(1, -2, 1) + K(1, -1, 0) + 2*K(1, -1, 1)
    + K(1, -1, 2) + K(1, 1, 0) - K(1, 1, -1) - 2*K(1, 0, 1) + K
   (0, 1, 1) + K(0, 0, 1)
```

Back in type A, we also have the simple Grothendieck basis. We have two versions of it related by a change of variable explained in Section 2.3 to avoid using negative exponents.

```
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 exponents (
   indexed by vectors)
sage: pol = Grothn[1,2] + Grothn[2,2]; pol
G(1, 2) + G(2, 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 exponents (
   indexed by vectors)
sage: pol.expand()
2*x(0, 0) + x(-2, 0) - x(-2, -1) - 3*x(-1, 0) - x(-1, -2) + 4*x
   (-1, -1) + x(0, -2) - 3*x(0, -1)
sage: pol = Grothp[1,2] + Grothp[2,2]; pol
G(1, 2) + G(2, 2)
sage: pol.expand()
x(1, 2) + x(2, 1)
sage: pol.expand().subs_var([(i,1-A.var(i)^(-1)) for i in xrange
   (1,3)])
2*x(0, 0) + x(-2, 0) - x(-2, -1) - 3*x(-1, 0) - x(-1, -2) + 4*x
   (-1, -1) + x(0, -2) - 3*x(0, -1)
```

The last basis we have implemented are the non-symmetric Macdonald polynomials. In order to use it, one has to define a polynomial ring on a bigger field than $\mathbb{Q}$ to use variables in the coefficients.

```
sage: var('t1 t2 q')
(t1, t2, q)
sage: K.<t1,t2,q> = QQ[]
sage: K = K.fraction_field()
sage: A = AbstractPolynomialRing(K);A
```

```
The abstract ring of multivariate polynomials on x over Fraction
    Field of Multivariate Polynomial Ring in t1, t2, q over
   Rational Field
sage: Mac = A.macdonald_basis_on_vectors()
sage: pol = Mac[1,2]; pol
M(1, 2)
sage: pol.expand()
t2^3*x(0, 0) + t2^2/q*x(1, 0) + ((t2*q+t2)/q^2)*x(1, 1) + 1/q^2*
   x(1, 2) + ((t2^2*q+t2^2)/q)*x(0, 1) + t2/q*x(0, 2)
sage: m = A.monomial_basis()
sage: Mac( m[1,1] )
(-t1*t2)*M(0, 0) + M(1, 0) + q*M(1, 1) + ((t1*t2*q^2+t2^2*q-t1*t
   2-t2^2)/(-t1*q-t2))*M(0, 1)
```

3.5. **Define a new basis.** All our bases are defined with divided differences acting recursively on sums of monomials. If one needs to work with a new basis where objects are indexed by vectors, the only thing to be implemented is the rule converting one vector to a sum of monomials. The inverse conversion is automatically done if the basis is triangular. One can define one's own conversion function and so create a new basis. Below is an example to recreate the Schubert polynomials.

```
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)
```

Above is a definition of a recursive function that transforms a Schubert element into a sum of monomials. The principle is easy: we take the Schubert vector as an argument (v) and test if we find an index i such that $v_i < v_{i+1}$. If we do, we compute the Schubert polynomial where v_i and v_{i+1} are switched and apply a divided difference operator. If v is antidominant, then the result is x^v that we get with *basis*(v). The three parameters of this function are the vector v corresponding to our Schubert element, a *basis* parameter which is the monomial basis we convert to, and *call_back* that we use to call back our function (this ensures getting a cached method, i.e., things are not calculated twice). To create a new basis, one needs to write a function that takes these three arguments and returns the polynomial associated with the vector v. It can directly be written into the sage command line as above or in the notebook, or in a file attached to your session. The function will not directly be called by the user, it will be passed to a method and the program will send the right values to *basis* and *call_back*. If more arguments are needed, one just adds them this way:

```
sage: def qt_schubert_on_basis(v, basis, call_back, q=1, t=1):
...     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 q*1/t*call_back(v).divided_difference(i
   +1)
...     return basis(v)
```

Now that we have the function, we will pass it to our algebra to create a new basis:

```
sage: A = AbstractPolynomialRing(QQ)
sage: myBasis = A.linear_basis_on_vectors("A","MySchub","Y",
   schubert_on_basis)
sage: pol = myBasis[2,1,3];pol
Y(2, 1, 3)
sage: pol.expand()
x(2, 1, 3) + x(2, 2, 2) + x(2, 3, 1) + x(3, 1, 2) + x(3, 2, 1) +
    x(4, 1, 1)
sage: myBasis(A.an_element())
Y(1, 2, 3) - Y(1, 3, 2) - Y(2, 1, 3) + Y(2, 3, 1) + Y(3, 1, 2) -
    Y(3, 2, 1) + Y(4, 1, 1)
```

This is a copy of the Schubert basis, and it works the same way as the previous bases we have seen in Section 3.4. Below is an example with a parametrized function:

```
sage: var('q t')
(q, t)
sage: K.<q,t> = QQ[]
sage: K = K.fraction_field()
sage: A = AbstractPolynomialRing(K)
sage: qtSchubertBasis = A.linear_basis_on_vectors("A","qtSchub
   ","YQ",qt_schubert_on_basis,(("q",q),("t",t)) )
sage: pol = qtSchubertBasis[1,2,3]; pol
YQ(1, 2, 3)
sage: pol.expand()
q^3/t^3*x(1, 2, 3) + q^3/t^3*x(1, 3, 2) + q^3/t^3*x(2, 1, 3) +
   2*q^3/t^3*x(2, 2, 2) + q^3/t^3*x(2, 3, 1) + q^3/t^3*x(3, 1,
   2) + q^3/t^3*x(3, 2, 1)
```

The extra parameters are sent by a tuple of couples *(parameter name, parameter value)* to the main algebra that will create the new basis.

3.6. **Double set of variables.** Our program also contains another algebra to work with a double set of variables.

```
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
```

One can see that the double algebra is the algebra of multivariate polynomials in the x variables with the multivariate polynomials in the y variables as coefficients.

```
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]
```

One can specify which bases to use in the x variables and which bases to use in the y variables.

```
sage: Dx = D
sage: Dy = D.base_ring()
sage: Schubx = Dx.schubert_basis_on_vectors()
sage: Schuby = Dy.schubert_basis_on_vectors()
sage: pol = Schuby[2,1,3] * Schubx[1,1,2]
sage: pol
(Yy(2,1,3))*Yx(1, 1, 2)
```

The *expand* function or all direct conversions are done in the x variables.

```
sage: pol.expand()
(Yy(2,1,3))*x(1, 1, 2) + (Yy(2,1,3))*x(1, 2, 1) + (Yy(2,1,3))*x
   (2, 1, 1)
sage: mx = Dx.monomial_basis()
sage: mx(pol)
(Yy(2,1,3))*x[1, 1, 2] + (Yy(2,1,3))*x[1, 2, 1] + (Yy(2,1,3))*x
   [2, 1, 1]
```

But, of course, one can also easily change the basis for the y variables.

```
sage: my = Dy.monomial_basis()
sage: pol.change_coeffs_bases(my)
(y[2,1,3]+y[2,2,2]+y[2,3,1]+y[3,1,2]+y[3,2,1]+y[4,1,1])*Yx(1, 1,
    2)
sage: pol = mx(pol); pol
(Yy(2,1,3))*x[1, 1, 2] + (Yy(2,1,3))*x[1, 2, 1] + (Yy(2,1,3))*x
   [2, 1, 1]
sage: pol.change_coeffs_bases(my)
(y[2,1,3]+y[2,2,2]+y[2,3,1]+y[3,1,2]+y[3,2,1]+y[4,1,1])*x[1, 1,
   2] + (y[2,1,3]+y[2,2,2]+y[2,3,1]+y[3,1,2]+y[3,2,1]+y[4,1,1])*
   x[1, 2, 1] + (y[2,1,3]+y[2,2,2]+y[2,3,1]+y[3,1,2]+y[3,2,1]+y
   [4,1,1])*x[2, 1, 1]
```

One can also change the role of variables between the main ones and the coefficients.

```
sage: pol
(Yy(2,1,3))*x[1, 1, 2] + (Yy(2,1,3))*x[1, 2, 1] + (Yy(2,1,3))*x
   [2, 1, 1]
sage: pol.swap_coeffs_elements()
(x[1,1,2]+x[1,2,1]+x[2,1,1])*Yy(2, 1, 3)
```

So we have seen that we can use our previous bases on a double set of variables. But we also have specific bases that only work with a double set of variables. Let us see the double Schubert polynomials and double Grothendieck polynomials, the way they were defined in Section 2.3.

```
sage: DoubleSchub = D.double_schubert_basis_on_vectors();
   DoubleSchub
```

```
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 = DoubleSchub[1,2]; pol
y[0]*YY(1, 2)
sage: pol.expand()
(-y(2,1,0)-y(2,0,1))*x(0, 0) + (y(1,1,0)+y(1,0,1)+y(2,0,0))*x(1,
    0) + (-2*y(1,0,0)-y(0,1,0)-y(0,0,1))*x(1, 1) + y[0]*x(1, 2)
   + (-y[1])*x(2, 0) + y[0]*x(2, 1) + (y(1,1,0)+y(1,0,1)+y
   (2,0,0))*x(0, 1) + (-y[1])*x(0, 2)
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]; pol
y[0]*GG(1, 2)
sage: pol.expand()
y[0]*x(0, 0) + (-y(2,1,1))*x(-2, -2) + (y(1,1,1))*x(-2, -1) + (-
   y[1])*x(-1, 0) + (y(1,1,1))*x(-1, -2) + (y(2,0,0)-y(0,1,1))*x
   (-1, -1) + (-y[1])*x(0, -1)
```

4. Some advanced applications

4.1. **Projective degrees of Schubert varieties.** In his Ph.D. thesis, Veigneau [7] presents an application of the software ACE by computing the projective degree of Schubert varieties. We can implement this same function with our patch on Sage-Combinat. The projective degree $d(X)$ of a sub-variety $X \subset \mathbb{P}^M$ of codimension k is the number of intersections between X and a generic hyperplane of dimension k. For σ a permutation of size n and X_σ a Schubert sub-variety of the flag variety $\mathcal{F}(\mathbb{C}^n)$ embedded in $\mathbb{P}^M$ by the Plücker embedding (with $M = 2^N - 1$ where $N = \frac{n(n-1)}{2}$ is the dimension of $\mathcal{F}(\mathbb{C}^n)$), $d(X_\sigma)$ is a coefficient in a product in the Schubert basis. More precisely, the first Chern class of the tautologic invertible vectorial fiber of $\mathbb{P}^M$ is $h = (n-1)x_1 + (n-2)x_2 + \ldots + x_{n-1}$ and $d(X_\sigma)$ is the coefficient of $Y_{n-1,n-2,\ldots,0}$ in $h^{N-\ell(\sigma)}Y_v$, where Y_v is the Schubert polynomial indexed by v, the Lehmer code of σ [3]. The following function computes these degrees:

```
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 look for the right coefficient
    for (key, coeff) in res:
        if ([key[i] for i in xrange(n)] == [n-i for i in xrange
           (1,n+1)]):
            return coeff

    return 0
```

One can also compute the product and directly read the result:

```
sage: A = AbstractPolynomialRing(QQ)
sage: m = A.monomial_basis()
sage: Schub =  A.schubert_basis_on_vectors()
sage: Schub( (3*m[1] + 2*m[0,1] + m[0,0,1])^4 * Schub[1,0,1,0])
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: proj_deg(Permutation([2,1,4,3]))
78
```

We can use our function to compute the degree for all permutations of size 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}
```

4.2. **Determinants of Schur functions.** Grassmannian Schubert polynomials are the Schubert polynomials indexed by vectors v such that $v_1 \leq v_2 \leq \ldots \leq v_n$. They are symmetric functions in $x_1, \ldots, x_n$. In a single set of variables (i.e., specializing y to 0), Grassmannian Schubert polynomials are equal to Schur functions. More precisely, the transition matrix between double Grassmannian Schubert polynomials and Schur functions is unitriangular.

```
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)
```

This allows us to compute determinants of Schur functions by replacing these by Schubert polynomials and specializing arbitrarily the y variables. For example, we can compute

$$|s_\mu(A)|_{\mu\subseteq 11}, \tag{32}$$

where $A \in [\{x_1, x_2\}, \{x_1, x_3\}, \{x_2, x_3\}]$, and prove that it is equal to $\prod_{j>i}(x_j - x_i)$. First, we replace s_μ by

$$|Y_u(A, y)|_{u=00,01,11} \tag{33}$$

and specialize $y_1 = x_1$, $y_2 = x_2$, in which case the determinant becomes

$$\begin{vmatrix} 1 & 1 & 1 \\ 0 & x_3 - x_2 & x_3 - x_1 \\ 0 & 0 & (x_3 - x_1)(x_2 - x_1) \end{vmatrix} \tag{34}$$

and gives the result. The following function computes the above Matrix:

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

    n = len(indices)

    #Initial definitions
    K = PolynomialRing(QQ,[var(v) for v in variables])
    K = K.fraction_field()
    D = DoubleAbstractPolynomialRing(K)
    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 alphabet:
                #we apply our polynomial on alphabets and
                   specialize the y (this should be improved on
                   further versions)
            pol = pu.subs_var( [(i,K(a[i])) for i in xrange(len(
               a))])
            pol = pol.swap_coeffs_elements()
            pol = pol.subs_var( [(i,K(variables[i])) for i in
               xrange(pol.nb_variables()) ] )
            if(pol ==0):
                coeff = 0
            else:
                coeff = list(list(pol)[0][1])[0][1]
            line.append(coeff)
        result_matrix.append(line)

    return Matrix(K,result_matrix)
```

In Sage:

```
sage: variables = ("x1", "x2", "x3")
sage: alphabet = [["x1","x2"],["x1","x3"],["x2","x3"]]
sage: indices = [[0,0],[0,1],[1,1]]
sage:
sage: res = compute_matrix(variables, alphabet, 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
-x1^2*x2 + x1*x2^2 + x1^2*x3 - x2^2*x3 - x1*x3^2 + x2*x3^2
sage: factor(det)
(x2 - x3) * (-x1 + x2) * (x1 - x3)
```

References

[1] F. Knop and S. Sahi. Difference equations and symmetric polynomials defined by their zeros. *International Mathematics Research Notices*, 1996(10):473–486, 1996.

[2] A. Lascoux. Schubert and Macdonald polynomials, a parallel. Electronically available at http://igm.univ-mlv.fr/%7Eal/ARTICLES/Dummies.pdf.

[3] A. Lascoux. Classes de Chern des variétés de drapeaux. *C. R. Acad. Sci. Paris Sér. I Math.*, 295(5):393–398, 1982.

[4] I. G. Macdonald. Affine Hecke algebras and orthogonal polynomials. *Astérisque* No. 237, Exp. No. 797, 4, 189–207, 1996. Séminaire Bourbaki, Vol. 1994/95.

[5] The Sage-Combinat community. Sage-Combinat: enhancing Sage as a toolbox for computer exploration in algebraic combinatorics, 2008. http://combinat.sagemath.org.

[6] W. A. Stein et al. *Sage Mathematics Software (Version 4.7)*. The Sage Development Team, 2011. http://www.sagemath.org.

[7] S. Veigneau. *Calcul symbolique et calcul distribué en combinatoire algébrique*. PhD thesis, Université de Marne-la-Vallée, 1996.

Institut Gaspard Monge, Université Paris-Est, Marne-la-Vallée, France