ORE POLYNOMIALS IN SAGE

MANUEL KAUERS *, MAXIMILIAN JAROSCHEK *, FREDRIK JOHANSSON *

Abstract. We present a Sage implementation of Ore algebras. The main features for the most common instances include basic arithmetic and actions; gcrd and lclm; D-finite closure properties; natural transformations between related algebras; guessing; desingularization; solvers for polynomials, rational functions and (generalized) power series. This paper is a tutorial on how to use the package.

1. Introduction

In computer algebra, objects are often described implicitly through equations they satisfy. For example, the exponential function $\exp(x)$ is uniquely specified by the linear differential equation $f'(x) - f(x) = 0$ and the initial value $f(0) = 1$. Likewise, the sequence F_n of Fibonacci numbers is uniquely determined by the linear recurrence $F_{n+2} - F_{n+1} - F_n = 0$ and the two initial values $F_0 = 0$, $F_1 = 1$. Especially for representing functions or sequences that cannot be expressed in "closed form", the differential or difference equations they may satisfy provide an attractive way to store them on the computer. The question is then how to calculate with objects which are given in this form.

Algorithms for Ore algebras provide a systematic answer to this question [3, 5]. Invented in the first half of the 20th century [15] with the objective of providing a unified theory for various kinds of linear operators, they have been used for many years in computer algebra systems, for example in the Maple packages OreTools [1], gfun [16] or mgfun [4], or in the Mathematica packages by Mallinger [14] and Koutschan [12, 11].

The purpose of this paper is to introduce an implementation of a collection of algorithms related to Ore algebras for the computer algebra system Sage [17]. It is addressed to first-time users who are already familiar with Sage, and with the theory of Ore algebras and its use for doing symbolic computation related to special functions. Readers unfamiliar with Sage are referred to [17], and readers unfamiliar with Ore algebras may wish to consult the recent tutorial [10] and the references given there for an introduction to the subject.

At the time of writing, the package we describe here is still under construction and has not yet been incorporated into the official Sage distribution. Readers who want to try it out are invited to download the current version from

`http://www.risc.jku.at/research/combinat/software/ore_algebra`

and are encouraged to send us bug reports or other comments. We hope that the community will find the code useful.

The following instructions show how to load the code and then create an Ore algebra A of linear differential operators and an Ore algebra B of recurrence operators. Observe the correct application of the respective commutation rules in both cases.

* Supported by the Austrian FWF grant Y464-N18.

```
sage: from ore_algebra import *
sage: R.<x> = PolynomialRing(ZZ); A.<Dx> = OreAlgebra(R)
sage: A
```

$$\mathbf{Z}[x]\langle \mathrm{Dx}\rangle$$

```
sage: A.random_element()
```

$$(2x-2)\,\mathrm{Dx}^2 + \left(-6x^2-2x-1\right)\mathrm{Dx} - 9x^2 - 21$$

```
sage: Dx*x
```

$$x\mathrm{Dx} + 1$$

```
sage: B.<Sx> = OreAlgebra(R)
sage: B
```

$$\mathbf{Z}[x]\langle \mathrm{Sx}\rangle$$

```
sage: Sx*x
```

$$(x+1)\,\mathrm{Sx}$$

More details on the construction of Ore algebras are given in the following section. The construction and manipulation of elements of Ore algebras is discussed in Section 3.

The package also supports Ore algebras with several generators already. However, so far we offer hardly more functionality than addition and multiplication for these. Much more functionality is available for algebras with one generator. Some of it is described in Section 4. We plan to add more code for the multivariate case in the future.

2. Ore Algebras

An Ore algebra is determined by a base ring and a finite number of generators. In the examples above, the base ring was $\mathbf{Z}[x]$, and the generators were Dx and Sx, respectively. If no other information is provided in the arguments, the `OreAlgebra` constructor chooses the nature of the generators according to their name: a generator called Dt represents the standard derivation d/dt acting on the generator t of the base ring, a generator called Sn represents the standard shift operator sending the generator n of the base ring to $n+1$.

For this way of generating algebras, generator names must be composed of one of the following single-letter prefixes followed by the name of a generator of the base ring.

Prefix	Name	Commutation rule
D	Standard derivation d/dx	$\mathrm{Dx}\,x = x\,\mathrm{Dx} + 1$
S	Standard shift $x \rightsquigarrow x+1$	$\mathrm{Sx}\,x = (x+1)\,\mathrm{Sx}$
T or Θ	Eulerian derivation $x\,d/dx$	$\mathrm{Tx}\,x = x\,\mathrm{Tx} + x$
F or Δ	Forward difference Δ_x	$\mathrm{Fx}\,x = (x+1)\mathrm{Fx} + 1$
Q	q-shift $x \rightsquigarrow q\,x$	$\mathrm{Qx}\,x = q\,x\,\mathrm{Qx}$
J	q-derivation ("Jackson derivation")	$\mathrm{Jx}\,x = q\,x\,\mathrm{Jx} + 1$
C	commutative generator	$\mathrm{Cx}\,x = x\,\mathrm{Cx}$

For the q-shift and the q-derivation, the base ring must contain an element q. The element playing the role of q can be specified as an optional argument.

```
sage: R.<x> = PolynomialRing(ZZ['q'])
sage: A.<Qx> = OreAlgebra(R)
sage: Qx*x
```

$$qx\mathrm{Qx}$$

```
sage: A.<Qx> = OreAlgebra(R, q=2)
sage: Qx*x
```

$$2x\mathrm{Qx}$$

In general, the commutation rules of a generator X of an Ore algebra A with base ring R are governed by two maps, $\sigma\colon R \to R$ and $\delta\colon R \to R$, where σ is a ring endomorphism (i.e., $\sigma(a+b) = \sigma(a) + \sigma(b)$ and $\sigma(ab) = \sigma(a)\sigma(b)$ for all $a, b \in R$) and δ is a skew-derivation for σ (i.e., $\delta(a+b) = \delta(a)+\delta(b)$ and $\delta(ab) = \delta(a)b+\sigma(a)\delta(b)$ for all $a, b \in R$). With two such maps being given, the generator X satisfies the commutation rule $Xa = \sigma(a)X + \delta(a)$ for every $a \in R$. If there is more than one generator, then each of them has its own pair of maps σ, δ. Different generators commute with each other; noncommutativity only takes place between generators and base ring elements.

It is possible to create an Ore algebra with user specified commutation rules. In this form, each generator must be declared by a tuple (X, σ, δ), where X is the name of the generator (a string), and σ and δ are dictionaries which contain the images of the generators of the base ring under the respective map. Here is how to specify an algebra of difference operators in this way:

```
sage: R.<x> = ZZ['x']
sage: A = OreAlgebra(R, ('X', {x:x+1}, {x:1}))
sage: X = A.gen()
sage: X*x
```

$$(x+1)\,X + 1$$

As another example, here is how to define an algebra of differential operators whose base ring is a differential field $K = \mathbb{Q}(x, y, z)$ where y represents $\exp(x)$ and z represents $\log(x)$:

```
sage: K = ZZ['x','y','z'].fraction_field()
sage: x,y,z = K.gens()
sage: A = OreAlgebra(K, ('D', {}, {x:1, y:y, z:1/x}))
sage: D = A.gen()
sage: D*x, D*y, D*z
```

$$\left(xD + 1, yD + y, zD + \frac{1}{x}\right)$$

In the dictionary specifying σ, omitted generators are understood to be mapped to themselves, so that `{}` in the definition of A in the example above is equivalent to `{x:x,y:y,z:z}`. In the dictionaries specifying δ, omitted generators are understood to be mapped to zero.

For Ore algebras with several generators, it is possible to mix specifications of generators via triples (X, σ, δ) with generators using the naming convention shortcuts as explained before. Continuing the previous example, here is a way to define an algebra A over K with two generators, a D that behaves like before, and in addition an Sx which acts like the standard shift on x and leaves the other generators fixed.

```
sage: A = OreAlgebra(K, ('D', {}, {x:1, y:y, z:1/x}), 'Sx')
sage: D, Sx = A.gens()
sage: D*x, Sx*x
```

$$(xD + 1, (x + 1)\mathrm{Sx})$$

```
sage: D*y, Sx*y
```

$$(yD + y, y\mathrm{Sx})$$

```
sage: D*z, Sx*z
```

$$\left(zD + \frac{1}{x}, z\mathrm{Sx}\right)$$

Not every ring is suitable as base ring of an Ore algebra. Base rings must themselves be polynomial rings (univariate or multivariate), or fraction fields of polynomial rings. Their base rings in turn may be either $\mathbb{Z}$, $\mathbb{Q}$, a prime field $GF(p)$, or a number field $\mathbb{Q}(\alpha)$, or — recursively — some ring which itself would be suitable as base ring of an Ore algebra.

```
sage: ZZ['x'].fraction_field()['y','z'] ### OK
```

$$\mathrm{Frac}(\mathbf{Z}[x])[y, z]$$

```
sage: GF(1091)['x','y','z']['u'] ### OK
```

$$\mathbf{F}_{1091}[x, y, z][u]$$

```
sage: ( ZZ['x','y','z'].quotient_ring(x^2+y^2+z^2-1) )['u'] ### not OK
```

$$\mathbf{Z}[x, y, z]/\left(x^2 + y^2 + z^2 - 1\right)\mathbf{Z}[x, y, z][u]$$

```
sage: GF(9, 'a')['x'] ### not OK
```

$$\mathbf{F}_{3^2}[x]$$

Note that the maps σ and δ must leave all the elements of the base ring's base ring fixed. They may only have nontrivial images for the top level generators.

The constituents of an Ore algebra A can be accessed through the methods summarized in the following table. Further methods can be found in the documentation.

Method name	short description
`associated_commutative_algebra()`	returns a polynomial ring with the same base ring as A and whose generators are named like the generators of A
`base_ring()`	returns the base ring of A
`delta(i)`	returns a callable object representing the delta map associated to the ith generator (default: $i = 0$)
`gen(i)`	returns the ith generator (default: $i = 0$)
`sigma(i)`	returns a callable object representing the sigma map associated to the ith generator (default: $i = 0$)
`var(i)`	returns the name of the ith generator (default: $i = 0$)

Examples:

```
sage: R.<x> = ZZ['x']; A.<Dx> = OreAlgebra(R)
```

```
sage: A
```

$$\mathbf{Z}[x]\langle \mathrm{Dx} \rangle$$

```
sage: A.associated_commutative_algebra()
```

$$\mathbf{Z}[x][\mathrm{Dx}]$$

```
sage: A.base_ring()
```

$$\mathbf{Z}[x]$$

```
sage: A.gen()
```

$$\mathrm{Dx}$$

```
sage: s = A.sigma(); d = A.delta();
```

```
sage: s(x^5), d(x^5)
```

$$\left(x^5, 5x^4\right)$$

3. Ore Polynomials

Ore polynomials are elements of Ore algebras, i.e., Sage objects whose parent is an Ore algebra object as described in the previous section. They can be constructed by addition and multiplication from generators and elements of the base ring.

```
sage: R.<x> = ZZ['x']; A.<Dx> = OreAlgebra(R)
```

```
sage: (5*x^2+3*x-7)*Dx^2 + (3*x^2+8*x-1)*Dx + (9*x^2-3*x+8)
```

$$\left(5x^2+3x-7\right)\mathrm{Dx}^2+\left(3x^2+8x-1\right)\mathrm{Dx}+9x^2-3x+8$$

Alternatively, an Ore polynomial can be constructed from any piece of data that is also accepted by the constructor of the associated commutative algebra. The associated commutative algebra of an Ore algebra is the commutative polynomial ring with the same base ring as the Ore algebra and with generators that are named like the generators of the Ore algebra. In particular, it is possible to create an Ore polynomial from the corresponding commutative polynomial, from a coefficient list, or even from a string representation.

```
sage: R.<x> = ZZ['x']; A.<Dx> = OreAlgebra(R)
sage: Ac = A.associated_commutative_algebra()
sage: Ac
```

$$\mathbf{Z}[x][\mathrm{Dx}]$$

```
sage: A(Ac.random_element())
```

$$(x-3)\,\mathrm{Dx}^2+\left(x^2+x+445\right)\mathrm{Dx}-x^2+x+1$$

```
sage: A([5*x,7*x-3,3*x+1])
```

$$(3x+1)\,\mathrm{Dx}^2+(7x-3)\,\mathrm{Dx}+5x$$

```
sage: A("(5*x^2+3*x-7)*Dx^2 + (3*x^2+8*x-1)*Dx + (9*x^2-3*x+8)")
```

$$\left(5x^2+3x-7\right)\mathrm{Dx}^2+\left(3x^2+8x-1\right)\mathrm{Dx}+9x^2-3x+8$$

Ore polynomials can also be created from Ore polynomials that belong to other algebras, provided that such a conversion is meaningful.

```
sage: R.<x> = ZZ['x']; A.<Dx> = OreAlgebra(R)
sage: L = (5*x^2+3*x-7)*Dx^2 + (3*x^2+8*x-1)*Dx + (9*x^2-3*x+8)
sage: L.parent()
```

$$\mathbf{Z}[x]\langle\mathrm{Dx}\rangle$$

```
sage: B = OreAlgebra(QQ['x'], 'Dx')
sage: L = B(L)
sage: L.parent()
```

$$\mathbf{Q}[x]\langle\mathrm{Dx}\rangle$$

In accordance with the Sage coercion model, such conversions take place automatically (if possible) when operators from different algebras are added or multiplied. Note that the result of such an operation need not belong to either of the parents of the operands but may instead have a suitable "common extension" as parent.

```
sage: A = OreAlgebra(ZZ['t']['x'], 'Dx')
sage: B = OreAlgebra(QQ['x'].fraction_field(), 'Dx')
sage: L = A.random_element() + B.random_element()
sage: L.parent()
```

$$\mathrm{Frac}(\mathbf{Q}[t][x])\langle \mathrm{Dx} \rangle$$

4. Selected Methods

Besides basic arithmetic for Ore operators, the package provides a wide range of methods to create, manipulate and solve several different kinds of operators. Some of these methods are accessible in any Ore algebra while others are tied specifically to, e.g., recurrence operators or differential operators.

In this section, we give an overview of the functionality provided by the package. Because of space limitation, only some of the available methods can be discussed here. For further information, we refer to the documentation.

4.1. **Methods for General Algebras.** A univariate Ore algebra over a field is a left Euclidean domain, which means that it is possible to perform left division with remainder. Building upon this, the greatest common right divisor (GCRD) and the least common left multiple (LCLM) of two Ore polynomials can be computed. The package provides a number of methods to carry out these tasks.

Method name	short description
`A.quo_rem(B)`	returns the left quotient and the left remainder of A and B.
`A.gcrd(B)`	returns the greatest common right divisor of A and B.
`A.xgcrd(B)`	returns the greatest common right divisor of A and B and the according Bézout coefficients.
`A.lclm(B)`	returns the least common left multiple of A and B.
`A.xlclm(B)`	returns the least common left multiple L of A and B and the left quotients of L and A and of L and B.
`A.resultant(B)`	returns the resultant of A and B (see [13] for its definition and properties).

All these methods are also available for Ore operators living in univariate Ore algebras over a base ring R which does not necessarily have to be a field. The operators are then implicitly assumed to live in the respective Ore algebra over the quotient field K of R. The output will be the GCRD (LCLM, quotient, remainder) in the Ore algebra over K but not over R, in which these objects might not exist or might not be computable.

```
sage: A = OreAlgebra(ZZ['n'], 'Sn')
sage: G = A.random_element(2)
sage: L1, L2 = A.random_element(7), A.random_element(5)
sage: while L1.gcrd(L2) != 1: L2 = A.random_element(5)
sage: L1, L2 = L1*G, L2*G
sage: L1.gcrd(L2) == G.normalize()
```

True

```
sage: L3, S, T = L1.xgcrd(L2)
sage: S*L1 + T*L2 == L3
```

True

```
sage: LCLM = L1.lclm(L2)
sage: LCLM % L1 == LCLM % L2 == 0
```

True

```
sage: LCLM.order() == L1.order() + L2.order() - G.order()
```

True

The GCRD is only unique up to multiplication (from the left) by an element from the base ring. The method `normalize` called in line 6 of the listing above multiplies a given operator from the left by some element from the base ring such as to produce a canonical representative from the class of all the operators that can be obtained from each other by left multiplication of a base ring element. This facilitates the comparison of output.

The efficiency of computing the GCRD depends on the size of the coefficients of intermediate results, and there are different strategies to control this growth via so called polynomial remainder sequences (PRS). The default is the improved PRS described in [7], which will usually be the fastest choice. Other strategies can be selected by the option `prs`.

```
sage: A = OreAlgebra(ZZ['n'], 'Sn')
sage: L1, L2 = A.random_element(3), A.random_element(2)
sage: algs = ["improved", "classic", "monic", "subresultant"]
sage: [L1.gcrd(L2, prs=a) for a in algs]
```

$[1, 1, 1, 1]$

If L_1, L_2 are operators, then the solutions of their GCRD are precisely the common solutions of L_1 and L_2. The LCLM, on the other hand, is the minimal order operator whose solution space contains all the solutions of L_1 and all the solutions of L_1. Because of this property, GCRD and LCLM are useful tools for constructing operators with prescribed solutions. For example, here is how to construct a differential operator which has the solutions x^5 and $\exp(x)$, starting from the obvious operators annihilating x^5 and $\exp(x)$, respectively.

```
sage: R.<x> = ZZ[]; A.<Dx> = OreAlgebra(R)
sage: L = (Dx - 1).lclm(x*Dx - 5)
sage: L
```

$$\left(x^2 - 5x\right) \mathrm{Dx}^2 + \left(-x^2 + 20\right) \mathrm{Dx} + 5x - 20$$

```
sage: L(x^5)
```

$$0$$

```
sage: L(exp(x)).full_simplify()
```

$$0$$

Observe how in the last two lines we apply the operator L to other objects. Such applications are not defined for every algebra and in general have to be specified by the user through an optional argument:

```
sage: A.<Qqn> = OreAlgebra(ZZ['q']['qn'])
sage: var('q', 'n', 'x')
```

$$(q, n, x)$$

```
sage: (Qqn^2+Qqn+1)(q^n, action=lambda expr: expr.substitute(n=n+1))
```

$$q^n + q^{(n+2)} + q^{(n+1)}$$

```
sage: (Qqn^2+Qqn+1)(x, action=lambda expr: expr.substitute(x=q*x))
```

$$q^2x + qx + x$$

Thanks to the LCLM operation discussed above, we have the property that when f and g are two objects which are annihilated by some operators L_1, L_2 belonging to some Ore algebra A then this algebra contains also an operator which annihilates their sum $f + g$. In other words, the class of solutions of operators of A is closed under addition. It turns out that similar closure properties hold for other operations. The following table lists some of the corresponding methods. Methods for more special closure properties will appear further below.

Method name	short description
`lclm()`	computes an annihilating operator for $f + g$ from annihilating operators for f and g
`symmetric_product()`	computes an annihilating operator for fg from annihilating operators for f and g
`symmetric_power()`	computes an annihilating operator for f^n from an annihilating operator for f and a given positive integer n
`annihilator_of_associate()`	computes an annihilating operator for $M(f)$ from an annihilating operator for f and a given operator M
`annihilator_of_polynomial()`	computes an annihilating operator for the object $p(f, \partial f, \partial^2 f, \dots)$ from an annihilating operator for f and a given multivariate polynomial p.

As an example application, let us prove Cassini's identity for Fibonacci numbers:

$$F_{n+1}^2 - F_n F_{n+2} = (-1)^n.$$

The idea is to derive, using commands from the table above, a recurrence satisfied by the left hand side, and then show that this recurrence is also valid for the right hand side.

```
sage: A.<Sn> = OreAlgebra(ZZ['n'])
sage: fib = Sn^2 - Sn - 1
sage: R.<x0,x1,x2> = ZZ['n']['x0','x1','x2']
sage: fib.annihilator_of_polynomial(x1^2 - x0*x2)
```

$$\mathrm{Sn} + 1$$

As this operator obviously annihilates $(-1)^n$, the proof is complete after checking that the identity holds for $n = 0$. Another way of carrying out the same computation using the other commands would be as follows.

```
sage: A.<Sn> = OreAlgebra(ZZ['n'])
sage: fib = Sn^2 - Sn - 1
sage: L1 = fib.annihilator_of_associate(Sn).symmetric_power(2)
sage: L2 = fib.annihilator_of_associate(Sn^2).symmetric_product(fib)
sage: L1.lclm(L2)
```

$$\mathrm{Sn}^3 - 2\mathrm{Sn}^2 - 2\mathrm{Sn} + 1$$

Observe that the resulting operator again annihilates $(-1)^n$, but its order is higher than the operator obtained before, so we need to check more initial values to complete the proof. For larger computations, the command `annihilator_of_polynomial` would also consume less computation time than the step-by-step approach.

4.2. **Methods for Special Algebras.** For the elements of some of the most important algebras, additional methods have been implemented. The following table lists some of the additional methods available for differential operators, i.e., elements of an Ore algebra of the form $R[x]\langle Dx\rangle$ or $K(x)\langle Dx\rangle$.

Method name	short description
`to_S()`	converts to a recurrence operator for the Taylor series solutions at the origin
`to_F()`	converts to a difference operator for the Taylor series solutions at the origin
`to_T()`	rewrites in terms of the Euler derivative
`annihilator_of_integral()`	converts an annihilator for $f(x)$ to one for $\int f(x)dx$
`annihilator_of_composition()`	converts an annihilator for $f(x)$ to one for $f(a(x))$ where $a(x)$ is algebraic over the base ring
`desingularize()`	computes a left multiple of this operator with polynomial coefficients and lowest possible leading coefficient degree
`associate_solutions(p)`	applied to an operator P, this computes, if possible, an operator M and a rational function m such that $DM = p + mP$ (see [2] for further information)
`polynomial_solutions()`	computes the polynomial solutions of this operator
`rational_solutions()`	computes the rational function solutions of this operator
`power_series_solutions()`	computes power series solutions of this operator
`generalized_series_solutions()`	computes generalized series solutions of this operator

As an example application, we compute an annihilator for the error function $\frac{2}{\sqrt{\pi}} \int_0^x \exp(-t^2)dt$, starting from the differential equation for $\exp(x)$, and produce the recurrence for the Taylor series coefficients at the origin. Finally, we compute the series solutions of the differential operator at infinity.

```
sage: R.<x> = ZZ['x']; A.<Dx> = OreAlgebra(R, 'Dx')
sage: (Dx - 1).annihilator_of_composition(-x^2)
```

$$\mathrm{Dx} + 2x$$

```
sage: L = (Dx + 2*x).annihilator_of_integral()
sage: L
```

$$\mathrm{Dx}^2 + 2x\mathrm{Dx}$$

```
sage: L.to_S(OreAlgebra(ZZ['n'], 'Sn'))
```

$$\left(n^2 + 3n + 2\right)\mathrm{Sn}^2 + 2n$$

```
sage: L.power_series_solutions(10)
```

$$\left[x - \frac{1}{3}x^3 + \frac{1}{10}x^5 - \frac{1}{42}x^7 + \frac{1}{216}x^9 + O(x^{10}), 1 + O(x^{10})\right]$$

```
sage: L.annihilator_of_composition(1/x).generalized_series_solutions()
```

$$\left[\exp\left(-x^{-2}\right) \cdot x \cdot \left(1 - \frac{1}{2}x^2 + \frac{3}{4}x^4 + O(x^5)\right), 1 + O(x^5)\right]$$

The last output implies that the operator annihilating $\int_0^x \exp(-t^2)dt$ also admits a solution which behaves for $x \to \infty$ like $\frac{1}{x}\exp(-x^2)$.

The next example illustrates the methods for finding rational and polynomial solutions of an operator L. These methods accept as an optional parameter an inhomogeneous part consisting of a list (or tuple) of base ring elements, $(f_1, \ldots, f_r)$. They return as output a list of tuples $(g, c_1, \ldots, c_r)$ with $L(g) = c_1 f_1 + \cdots + c_r f_r$ where g is a polynomial or rational function and $c_1, \ldots, c_r$ are constants, i.e., elements of the base ring's base ring. The tuples form a vector space basis of the solution space.

In the example session below, we start from two polynomials p, q, then compute an operator L having p and q as solutions, and then recover p and q from L. Note that for consistency also the solutions of homogeneous equations are returned as tuples. At the end we give an example for solving an inhomogeneous equation.

```
sage: R.<x> = ZZ[]
sage: p = x^2 + 3*x + 8; q = x^3 - 7*x + 5
sage: A.<Dx> = OreAlgebra(R)
sage: L = (p*Dx - p.derivative()).lclm(q*Dx - q.derivative())
sage: L
```

$$\left(x^4 + 6x^3 + 31x^2 - 10x - 71\right)\mathrm{Dx}^2 + \left(-4x^3 - 18x^2 - 62x + 10\right)\mathrm{Dx} + 6x^2 + 18x + 14$$

```
sage: L.polynomial_solutions()
```

$$\left[\left(-3x^3 + 2x^2 + 27x + 1\right), \left(-8x^3 + 5x^2 + 71x\right)\right]$$

```
sage: M = (2*x+3)*Dx^2 + (4*x+5)*Dx + (6*x+7)
sage: sol = M.polynomial_solutions([1,x,x^2,x^3])
sage: sol
```

$$\left[(1,7,6,0,0)\,,(x,5,11,6,0)\,,\left(x^2,6,14,15,6\right)\right]$$

```
sage: map(lambda s: M(s[0]) == s[1]+s[2]*x+s[3]*x^2+s[4]*x^3, sol)
```

[True, True, True]

The functions `polynomial_solutions` and `rational_solutions` are not only defined for differential operators but also for recurrence operators, i.e., elements of an Ore algebra of the form $R[x]\langle Sx\rangle$ or $K(x)\langle Sx\rangle$. Some other methods defined for recurrence operators are listed in the following table.

Method name	short description
`to_D()`	converts annihilator for the coefficients in a power series to a differential operator for the sum
`to_F()`	converts shift operator to a difference operator
`to_T()`	converts to a differential operator in terms of the Euler derivative
`annihilator_of_sum()`	converts an annihilator for $f(n)$ to one for the sum $\sum_{k=0}^{n} f(k)$
`annihilator_of_composition()`	converts an annihilator for $f(n)$ to one for $f(\lfloor un+v\rfloor)$ where $u, v \in \mathbb{Q}$
`annihilator_of_interlacing()`	interlaces two or more sequences
`desingularize()`	computes a left multiple of this operator with polynomial coefficients and lowest possible leading coefficient degree
`associate_solutions(p)`	applied to an operator P, this computes, if possible, an operator M and a rational function m such that $(S-1)M = p+mP$ (see [2] for further information)
`polynomial_solutions()`	computes the polynomial solutions of this operator
`rational_solutions()`	computes the rational function solutions of this operator
`generalized_series_solutions()`	computes asymptotic expansions of sequences annihilated by the operator
`to_list()`	computes terms of a sequence annihilated by the operator

As an example application, we compute an annihilator for the sequence $c(n) = \sum_{k=0}^{n} 1/k!$:

```
sage: R.<n> = ZZ[]; A.<Sn> = OreAlgebra(R)
sage: inverse_factorials = (n + 1) * Sn - 1
sage: partial_sums = inverse_factorials.annihilator_of_sum()
sage: partial_sums
```

$$(n+2)\,\mathrm{Sn}^2 + (-n-3)\,\mathrm{Sn} + 1$$

The `to_list` method returns the first few values of a sequence, given the initial values:

```
sage: L = partial_sums.to_list([1, 2], 8)
```

```
sage: L
```

$$\left[1, 2, \frac{5}{2}, \frac{8}{3}, \frac{65}{24}, \frac{163}{60}, \frac{1957}{720}, \frac{685}{252}\right]$$

```
sage: N(L[7])
```

$$2.71825396825397$$

We compute the asymptotic expansion of the sequence of terms to estimate how many terms we need to approximate e to a given number of digits:

```
sage: digits = 10^5
```

```
sage: asymp = inverse_factorials.generalized_series_solutions(3)
```

```
sage: target = lambda x: log(abs(asymp[0](RR(x))), 10) + digits
```

```
sage: num_terms = ceil(find_root(target, 1, 10^6))
```

```
sage: num_terms
```

$$25207$$

In some cases, for example when the base ring is $\mathbb{Z}$ or $\mathbb{Z}[x]$, isolated values of a sequence can be computed asymptotically faster for large n than by listing all values, using the binary splitting technique. The `forward_matrix_bsplit` method, called with argument n, returns a matrix P and a polynomial Q such that P/Q multiplied by a column vector of initial values $c_0, c_1, \ldots$ yields $c_n, c_{n+1}, \ldots$. This way, computing 10^5 digits of e takes a fraction of a second:

```
sage: e_approx = N(e, 400000)
```

```
sage: P, Q = partial_sums.forward_matrix_bsplit(num_terms)
```

```
sage: u = Matrix([[e_approx], [e_approx]]) - P * Matrix([[1], [2]]) / Q
```

```
sage: u.change_ring(RealField(20))
```

$$\begin{pmatrix} 1.3053 \times 10^{-100009} \\ 5.1780 \times 10^{-100014} \end{pmatrix}$$

4.3. **Guessing.** Guessing is, in some sense, the reverse operation of `to_list` for recurrence operators, or of `power_series_solutions` for differential operators. It is of the most popular features of packages like gfun, and there are even some special purpose packages dedicated to this technique [9, 6]. The basic idea is simple. Given a finite array of numbers, thought of as the first terms of an infinite sequence, we want to know whether this sequence satisfies a recurrence. The algorithm behind a guessing engine searches for small equations matching the given data. Generically, no such equations exist, so if some are found, it is fair to "guess" that they are in fact valid equations for the whole infinite sequence.

We provide a guessing function which takes as input a list of terms and an Ore algebra, and returns as output an operator which matches the given data and which, in some measure, would be unlikely to exist for random data.

```
sage: data = [ 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 ]
sage: L = guess(data, OreAlgebra(ZZ['n'], 'Sn'))
sage: L
```

$$-\mathrm{Sn}^2 + \mathrm{Sn} + 1$$

```
sage: L(data)
```

$$[0, 0, 0, 0, 0, 0, 0, 0, 0]$$

```
sage: M = guess(data, OreAlgebra(ZZ['x'], 'Dx'))
sage: M
```

$$\left(-x^3 - x^2 + x\right) \mathrm{Dx} - x^2 - 1$$

```
sage: M(x/(1-x-x^2))
```

$$0$$

If an algebra of differential operators is supplied as second argument, the data is understood as the first few coefficients of a power series. The output operator is expected to have this power series as solution.

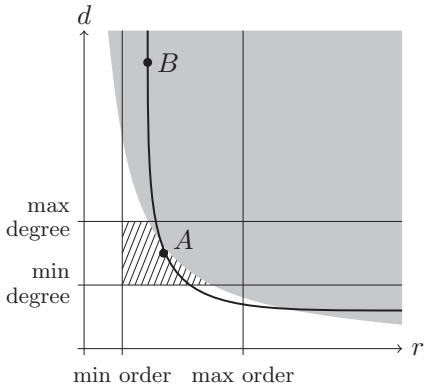
It can happen that the procedure is unable to find an operator matching the given data. In this case, an exception is raised. There are two possible explanations for such an event. Either the sequence in question does not satisfy any equations, or it does but the equations are so big that more data is needed to detect them.

Several options are available for customizing the search for relations. In order to explain them, we first need to give some details on the underlying algorithms. For simplicity of language, we restrict here to the case of recurrence operators. The situation for differential operators is very similar.

For the most typical situations, there are two important hyperbolas. One describes the region in the (r, d)-plane consisting of all points for which there exists an operator of order r and degree d truly satisfied by the sequence in question. (See [8] for an explanation why the boundary of this region is usually a hyperbola.) The second describes the region of all points (r, d) for which an operator of order r and degree d can be detected when N terms are provided as input. This region is determined by the requirement $(r + 1)(d + 2) < N$.

The method tests a sequence of points (r_1, d_1), (r_2, d_2), ... right below this second hyperbola. Success at a point (r_i, d_i) means that some evidence for an operator of order $\leq r_i$ and degree $\leq d_i$ has been found. This operator however is not explicitly computed. Instead, the method uses the partial information found about this operator to calculate an operator which with high probability is the minimal order operator satisfied by the sequence in question. This operator is usually more interesting than the one at (r_i, d_i), and its computation is usually more efficient.

Using the option `path`, the user can specify a list of points (r_i, d_i) which should be used instead of the standard path. By setting the options `min_degree`, `max_degree`, `min_order`, `max_order`, all points (r, d) of the path are discarded for which r or d is not within the specified bounds. These options can be used to accelerate the search in situations where the user has some knowledge (or intuition) about the size of the expected equations.



The figure on the right illustrates the typical situation for guessing problems that are not too small and not too artificial. The gray region indicates the area which is not accessible with the given amount of data. Only the points (r, d) below it can be tested for an operator of order r and degree d that fits to the given data. Let's assume that operators exist on and above the solid black hyperbola. The user will usually not know this curve in advance but may have some expectations about it and can restrict the search accordingly, for example to the dashed area shown in the figure. The method will detect the existence of an operator, say at point A, and construct from the information gained at this point an operator of minimal possible order, which may correspond to point B. This operator is returned as output. Note that the degree of the output may exceed the value of `max_degree`, and its order may be smaller than `min_order`:

```
sage: data = [(n+1)^10*2^n + 3^n for n in xrange(200)]

sage: L = guess(data, OreAlgebra(ZZ['n'],'Sn'), min_order=3, max_degree=5)

sage: L.order(), L.degree()
```

$$(2, 10)$$

In order to test a specific point (r, d), the data array must contain at least $(r + 1)(d + 2)$ terms. If it has more terms, the guess becomes more trustworthy, but also the computation time increases. By setting the option `ensure` to a positive integer e, the user can request that only such points (r, d) should be tested for which the data array contains at least e more terms than needed. This increases the reliability. By setting the option `cut` to a positive integer c, the user requests that for testing a point (r, d), the method should take into account at most c more terms than needed. If the data array contains more terms, superfluous ones are ignored in the interest of better performance. We must always have $0 \leq e \leq c \leq \infty$. The default setting is $e = 0$, $c = \infty$.

References

[1] Sergei A. Abramov, Ha Q. Le, and Ziming Li. OreTools: a computer algebra library for univariate ore polynomial rings. Technical Report CS-2003-12, University of Waterloo, 2003.

[2] Sergei A. Abramov and Mark van Hoeij. Integration of solutions of linear functional equations. *Integral transforms and Special Functions*, 9:3–12, 1999.

[3] Manuel Bronstein and Marko Petkovšek. An introduction to pseudo-linear algebra. *Theoretical Computer Science*, 157(1):3–33, 1996.

[4] Frédéric Chyzak. *Fonctions holonomes en calcul formel*. PhD thesis, INRIA Rocquencourt, 1998.

[5] Frédéric Chyzak and Bruno Salvy. Non-commutative elimination in Ore algebras proves multivariate identities. *Journal of Symbolic Computation*, 26:187–227, 1998.

[6] Waldemar Hebisch and Martin Rubey. Extended Rate, more GFUN. *Journal of Symbolic Computation*, 46(8):889–903, 2011.

[7] Maximilian Jaroschek. Improved polynomial remainder sequences for Ore polynomials. *Journal of Symbolic Computation*, 2013. to appear.

[8] Maximilian Jaroschek, Manuel Kauers, Shaoshi Chen, and Michael F. Singer. Desingularization explains order-degree curves for Ore operators. In Manuel Kauers, editor, *Proceedings of ISSAC'13*, 2013. to appear.

[9] Manuel Kauers. Guessing handbook. Technical Report 09-07, RISC-Linz, 2009.

[10] Manuel Kauers. The holonomic toolkit. In Johannes Bluemlein and Carsten Schneider, editors, *Computer Algebra in Quantum Field Theory: Integration, Summation and Special Functions*, Texts and Monographs in Symbolic Computation. Springer, 2013.

[11] Christoph Koutschan. *Advanced Applications of the Holonomic Systems Approach*. PhD thesis, RISC-Linz, Johannes Kepler Universität Linz, 2009.

[12] Christoph Koutschan. HolonomicFunctions (User's Guide). Technical Report 10-01, RISC Report Series, University of Linz, Austria, January 2010.

[13] Ziming Li. *A Subresultant Theory for Linear Differential, Linear Difference, and Ore Polynomials, with Applications*. PhD thesis, RISC-Linz, 1996.

[14] Christian Mallinger. Algorithmic manipulations and transformations of univariate holonomic functions and sequences. Master's thesis, J. Kepler University, Linz, August 1996.

[15] Øystein Ore. Theory of non-commutative polynomials. *Annals of Mathematics*, 34:480–508, 1933.

[16] Bruno Salvy and Paul Zimmermann. Gfun: a Maple package for the manipulation of generating and holonomic functions in one variable. *ACM Transactions on Mathematical Software*, 20(2):163–177, 1994.

[17] W. A. Stein et al. *Sage Mathematics Software (Version 5.9)*. The Sage Development Team, 2013. `http://www.sagemath.org`.

Manuel Kauers, Research Institute for Symbolic Computation (RISC), J. Kepler University Linz, Austria

E-mail address: `mkauers@risc.uni-linz.ac.at`

Maximilian Jaroschek, Research Institute for Symbolic Computation (RISC), J. Kepler University Linz, Austria

E-mail address: `mjarosch@risc.uni-linz.ac.at`

Fredrik Johansson, Research Institute for Symbolic Computation (RISC), J. Kepler University Linz, Austria

E-mail address: `fjohanss@risc.uni-linz.ac.at`