Drinfeld modules in SageMath

David Ayotte(a), Xavier Caruso(b), Antoine Leudière(c), Joseph Musleh(d)

Abstract

We present the first implementation of Drinfeld modules fully integrated in the SageMath ecosystem. First features will be released with SageMath 10.0.

The pursuit of Class Field Theory has been a long-standing dream, once held by Kronecker himself. In 1854, he made a significant contribution to the field with the announcement of the Kronecker-Weber theorem, which states that every abelian number field can be generated by a cyclotomic extension of $\mathbb{Q}$. Similarly, extensions of imaginary quadratic number fields can be described using a so-called Hilbert class field [CLRS09]. Many important results of the field were conjectured by Hilbert and Kronecker. Some of them were only proven in the twentieth century, by mathematicians like Takagi, Artin, and Chevalley [CLRS09]. And to this day, the general quest for describing extensions of a number field remains elusive. But what if the quest was easier for function fields?

In 1974, Drinfeld introduced the now-known Drinfeld modules [Dri74], pursuing the ideas of Carlitz [Car35]. With Drinfeld modules, one can develop an explicit class field Theory for function fields: every Drinfeld module can be assigned a rank; cyclotomic function fields are generated by torsion spaces of rank 1 Drinfeld modules and j-invariants of rank 2 Drinfeld intervene in the construction of the function-field analogue of the Hilbert class field. Later developments saw Drinfeld modules being instrumental in Lafforgue's proof of some of Langlands conjectures for function fields [Laf02]. The analogue question for number fields is still out of reach.

In the recent years, purely algorithmic thesis [Car18] and papers [CGS20, MS19, MS23, LS22] have been published, emphasizing efficiency. The present implementation began as the need for a unified and tangible manipulation tool, which we hope will be useful to a large community. We made notable efforts to accompany the code with exemplary documentation and use pre-existing SageMath facilities wherever possible. Our three core principles were *reliability*, *user interface elegance*, and *integration*. The original ticket (see Github PR #35026) was opened in April 2022 and merged in March 2023. Many *pull requests* have since been proposed to enhance the capabilities of the original contribution and are under active development, fueling an ever-growing interest in Drinfeld modules.

Mathematical background. Before entering into the core of this presentation, we need to recall basic definitions related to Drinfeld modules. Let $\mathbb{F}_q$ be a finite field with q elements, let K be an extension of $\mathbb{F}_q$ and let $\overline{K}$ be an algebraic closure of K. Additionally, we equip K with a structure of $\mathbb{F}_q[T]$-*field*, meaning we give ourselves a morphism of $\mathbb{F}_q$-algebras $\gamma : \mathbb{F}_q[T] \to K$. We use the notation τ to denote the $\mathbb{F}_q$-linear endomorphism of $\overline{K}$ defined by $x \mapsto x^q$. We define the *ring of Ore polynomials* $K\{\tau\}$ as the ring whose elements are sum of the form $a_0 + a_1\tau + \cdots + a_n\tau^n$ where $n \in \mathbb{Z}_{\geqslant 0}$ and $a_i \in K$ for all $0 \leqslant i \leqslant n$. In $K\{\tau\}$, we have the identity $\tau a = a^q\tau$ whenever $a \in K$.

A *Drinfeld module* over K is a morphism of $\mathbb{F}_q$-algebras $\phi : \mathbb{F}_q[T] \to K\{\tau\}$ such that $\phi(T)$ has constant coefficient $\gamma(T)$ and nonzero degree in τ. We remark that $\phi(T)$ entirely determines ϕ; we often denote it simply by ϕ_T. The name *module* comes from the fact that ϕ endows $\overline{K}$ with an action of $\mathbb{F}_q[T]$, defined by $a \cdot x = \phi(a)(x)$ for all a in $\mathbb{F}_q[T]$ and x in $\overline{K}$.

Given two Drinfeld modules ϕ, ψ, a *morphism* $\phi \to \psi$ is an Ore polynomial $u \in K\{\tau\}$ such that $u\phi_T = \psi_T u$. An *isogeny* is a nonzero morphism.

1 Implementation choices

When we started this project, our objective was to provide a general well-documented package for Drinfeld modules, intended to the working mathematician in the domain. Our main concern was then to develop a software with an easy-to-use interface, implementing all the basics of the theory and not focussing on a particular application.

1.1 Why SageMath? We choose to implement our package in SageMath for several reasons. Firstly, SageMath is a mathematical computational tool built on top of the widely-used Python programming language. As a free and open source software, it benefits from contributions from mathematicians with different backgrounds. With its general design philosophy, it also accommodates a vast range of mathematical domains. SageMath thus appeals to a large audience, and then meets our idea of providing tools to all the community of mathematicians working with Drinfeld modules. Secondly, SageMath looks particularly adapted to our project since it already implements two important primitives for us, namely:

1. the ring of Ore polynomials, with many additional functionalities when the base ring is a finite field,
2. a framework for manipulating ring extensions (which is useful to us because we need to view K as a $\mathbb{F}_q[T]$-algebra).

After we made this choice, we prioritized careful integration within the ecosystem of SageMath. This forced us to be very rigourous and we benefited a lot from the feedback of the SageMath's core developers; in particular, throughout the develop-

(a)Concordia University
(b)CNRS, University of Bordeaux, INRIA
(c)University of Lorraine, INRIA, CNRS
(d)University of Waterloo

ment of the project, we were constantly very careful to the simplificity of the interface, the clarity and the completeness of the documentation and the unit tests. Concretely, each class, method or function is augmented with a doctest that has description, tests and examples. The entry point of the documentation is the docstring of `DrinfeldModule`, accessed in the SageMath console by running `DrinfeldModule?`. For specific methods, the `?` keyword is also used, *e.g.* `phi.rank?`. The documentation also appears in the SageMath Reference Manual [Dev].

Our library is completely open and as such, we encourage all mathematicians and computer scientists to improve it with any contribution that may interest them.

1.2 The base type of Drinfeld module objects. A first difficulty we encoutered at the very beginning of the project was that a Drinfeld module is *not* an actual module in the classical sense. In particular, a Drinfeld module has no underlying set and a morphism between Drinfeld modules is not a set-theoretical map. However, in the SageMath idiom, most objects are either sets with additional structures — a so-called `Parent` — or elements in such sets — an `Element`. This philosophy is referred to as the *parent/element framework*. It is often implicitely assumed in SageMath. For example, the default *Test Suite* of a parent checks that its category is a subcategory of **Sets**, the constructor of `Morphism` objects assume that the domain and codomain are both parents, *etc.* For Drinfeld modules, this raises many questions and we eventually had to make a difficult choice between the three following compromises:

1. Making Drinfeld modules elements (as they are *in fine* morphisms) and their set a parent (the so-called "homsets" in SageMath); this option offers a standard parent/element framework.
2. Implementing Drinfeld modules as parents without elements, following actually the implementation of elliptic curves[1]. This option makes the implementation of morphisms between Drinfeld module (and, more generally, of the category of Drinfeld modules) easier. Besides, making in some sense Drinfeld modules as function field analogues of elliptic curve, this option has a strong mathematical base.
3. Implementing Drinfeld modules as `CategoryObject`. This class does exist in SageMath and it is not expected to have elements. However, unfortunately, it is used only sporadically, it is currently incompatible with `Morphism` objects and it is no longer maintained (it is possibly intended to disappear eventually).

All these options have their benefits and drawbacks. We discussed all of them with the SageMath core developers (see Github PR #37313 and Github PR #34534). At some point, the third option looked to us the most mathematically appealing; however given that `CategoryObjects` are not fully supported, we decided to rule out this possibility. On the other hand, the first option seems more practical but we believed that it was too mathematically misleading; it would also require a workaround to make morphisms work. We then ultimately chose the second option.

[1]In SageMath, elliptic curves `E` are schemes, and `E.an_element()` return an element whose parent is not `E`, but the group `G` of points of `E`. In that case, `G` and `E` are distinct objects.

2 Overview of our package

Our package is publicly available on Github: https://github.com/xcaruso/sage/tree/drinfeld-modules. It is intended to be ultimately included in the standard distribution of SageMath. Actually, about half of the package will be released with SageMath 10.0, the other half is still under review; we hope that it will be approved soon by the SageMath community.

Alternatively, we offer the possibility to try our package online on the platform plm-binder. For this, please go to the URL https://caruso.perso.math.cnrs.fr/notebook/drinfeld-modules; after a few seconds, a Jupyter notebook will open with a full tutorial presenting the main functionalities of our package. Beyond reading the tutorial, plm-binder allows for editing the notebook, executing commands, creating new worksheets, *etc.* Be careful however that your modifications will not be stored after your session is closed; if you want to keep them, do not forget to download your notebooks!

2.1 Construction and basic properties. A Drinfeld module is a rather sophisticated mathematical object, whose definition already involves several nontrivial ingredients: a morphism $\gamma : \mathbb{F}_q[T] \to K$, the ring of Ore polynomials $K\{\tau\}$. In our package, we have tried as much as possible to minimize the number of lines for creating a Drinfeld module. In particular, in most cases, it is not needed to define explicitely γ and $K\{\tau\}$.

```
sage: K.<w> = GF(4)
sage: phi = DrinfeldModule(GF(2)['T'], [w, 0, 0, 1])
sage: phi
Drinfeld module defined by T |--> t^3 + w
```

Once a Drinfeld module is instantiated, we have access to a panel of methods for accessing its most important invariants, *e.g.* `phi.characteristic()`, `phi.rank()`, `phi.height()`, *etc.* It is also also possible to compute the value $\phi(a)$ by simply using the syntax `phi(a)`.

2.2 Morphisms and isogenies. Given that Drinfeld modules do not have elements, the morphisms between them are the main tools at our disposal for understanding their structure. Our package provides the method `hom` for easily constructing morphisms.

```
sage: t = phi.ore_variable()
sage: phi.hom(t + w)
Drinfeld Module morphism:
  From: Drinfeld module defined by T |--> t^3 + w
  To:   Drinfeld module defined by T |--> t^3 + t^2 +
        w*t + w
  Defn: t + w
```

We observe that the software has automatically determined the codomain. Once we have constructed a morphism f, many methods become available, *e.g.* `f.codomain()`, `f.is_isomorphism()`, *etc.* At the level of Drinfeld modules themselves, the method `is_isomorphic` allows for checking whether two Drinfeld modules are isomorphic. When K is finite, a very important morphism is the Frobenius endomorphism defined by the Ore polynomial $\tau^{[K:\mathbb{F}_q]}$ (see also §2.4). Our package provides the method `phi.frobenius_endomorphism()` for rapidly instantiating it.

Of course, addition and composition of morphisms are implemented, as well as inverse of isomorphisms. We observe in addition that any polynomial $P \in \mathbb{F}_q[T]$ defines an endomorphism of ϕ (corresponding to the Ore polynomial ϕ_P). In particular, the Hom spaces $\mathrm{Hom}(\phi, \psi)$ inherits a structure of left module over $\mathbb{F}_q[T]$, which is accessible in our package *via* the operator `*`. This simple syntax allows for writing down easily complex formulas.

Finally, in contrast to the case of elliptic curves, computing morphisms between Drinfeld modules defined over finite fields amounts to solving a linear system over $\mathbb{F}_q$. This leads to an efficient algorithm for finding isogenies [Wes22], which we implemented in our package.

```
sage: psi = DrinfeldModule(GF(2)['T'], [w, w+1, 1, 1])
sage: Hom(phi, psi).an_isogeny()
Drinfeld Module morphism:
  From: Drinfeld module defined by T |--> t^3 + w
  To:   Drinfeld module defined by T |--> t^3 + t^2 +
       (w + 1)*t + w
  Defn: t^2 + (w + 1)*t + 1
```

The command `Hom(phi, psi).basis(degree=d)` returns more generally an $\mathbb{F}_q$-basis of the vector space of morphisms between ϕ and ψ defined by an Ore polynomial of degree at most d.

2.3 *j*-invariants.

In the classical theory, it is well known that elliptic curves over an algebraically closed field are classified, up to isomorphisms, by their j-invariants [Sil09, Proposition 1.4]. Moreover, when working over a quadratic imaginary field R, the j-invariants of elliptic curves with complex multiplication by R provide an explicit description of abelian extensions of R [Sil94, Chap. II]. Similar results hold for Drinfeld modules: one can attach to any Drinfeld module ϕ of rank 2 a j-invariant which determines the isomorphism class of ϕ over an algebraic closure; besides, certain j-invariants play a pivotal role in the study of certain algebraic extensions of $\mathbb{F}_q(T)$ [Gek83, (4.4)], [Ham03, Theorem 6.9].

The j-invariant of a Drinfeld module of rank 2 is given by a simple closed formula: if $\phi_T = \gamma(T) + g_1(\phi)\tau + g_2(\phi)\tau^2$, then $j(\phi) := g_1(\phi)^{q+1}/g_2(\phi)$. This makes it easy to compute and our package provides a direct method for accessing it.

```
sage: phi = DrinfeldModule(GF(2)['T'], [w, w+1, w+2])
sage: phi.j_invariant()
w + 1
```

In the context of Drinfeld modules, it turns out that j-invariants are defined in any rank [Pot98]. A Drinfeld module of rank $r > 2$ does not have a single j-invariant but a complete family of j-invariants indexed by the integral points of a convex subset of $\mathbb{R}^r$. Fortunately, those j-invariants are still given by explicit closed formulas, making their computation possible. Our package provides methods (`basic_j_invariant_parameters`, `basic_j_invariants`, `jk_invariants`, *etc.*) for computing and manipulating those j-invariants in any rank. We refer to our tutorial on plm-binder for more details.

2.4 Norms and characteristic polynomials.

In the classical setting, morphisms (resp. endomorphisms) between elliptic curves have norms (resp. characteristic polynomials) which can be found by looking at the action on the Tate module [Lor96, §5]. Again, similar facts hold true in the Drinfeld setting [Gek91, Lem. 3.10]: there is a well-defined notion of Tate module of a Drinfeld module and morphisms between Drinfeld modules do induce linear transformations of the associated Tate modules. From this construction, one can define the *norm* of a general isogeny and the *characteristic polynomial* of an endomorphism. Unfortunately, computing in practice the Tate module is a hard task in general given that the latter usually lives in a quite large extension of K. Norms and characteristic polynomials have however alternative interpretations, which makes tangible the perspective of computing them efficiently. Concretely, algorithms for this task based on the notion of Anderson motives [And86] have been designed in [CL23]. We implemented them in our package; they are available through the methods `norm` and `charpoly`.

When K is finite, a distinguished endomorphism of a Drinfeld module ϕ is its Frobenius endomorphism. Its characteristic polynomial plays a prominent role in the theory; notably, it entirely determines the isogeny class of ϕ [Gek91, Th. 3.5]. In our package, we implemented three different algorithms for computing this invariant, namely:

- the `motive` algorithm, based on Anderson motives as already discussed above,
- the `crystalline` algorithm [MS23], based on the action of the Frobenius on the crystalline cohomology,
- the `CSA` algorithm [CL23], based on a reinterpretation of the characteristic polynomial of the Frobenius as a reduced norm in some central simple algebra.

Figure 1 (on page 4) compares the timings of our three algorithms[2] depending on the rank of the Drinfeld module and the degree of the extension $K/\mathbb{F}_q$ (with $q = 5$ in our example). We observe that the `CSA` algorithm performs better when the rank is large, whereas the `crystalline` algorithm is the best when $[K : \mathbb{F}_q]$ is large. The method `frobenius_charpoly`, which is the entry point for this task in our package, is tuned for choosing by itself the best available algorithm depending on the input; the user can nevertheless require the use of a specific algorithm by passing in the keyword `algorithm`.

As a byproduct of this computation, we implemented a method `is_isogenous` which checks whether two given Drinfeld modules are isogenous.

2.5 Exponential and logarithm.

A quite important perspective on Drinfeld modules is the analytic point of view. To explain it, let us go back to the case of elliptic curves: we know that an elliptic curve E over $\mathbb{C}$ is uniformized by a free $\mathbb{Z}$-submodule in $\mathbb{C}$ of rank 2, *i.e.* $E(\mathbb{C}) \cong \mathbb{C}/\Lambda$ as complex Lie groups [Sil09, VI §5]. In the Drinfeld setting, a similar result holds after replacing the field $\mathbb{C}$ by $\mathbb{C}_\infty$, the completion for the valuation associated to $\frac{1}{T}$ of an algebraic closure of $\mathbb{F}_q((\frac{1}{T}))$ [Gos96, Theorem 4.6.9]. In this situation, the uniformization is obtained *via* a $\mathbb{F}_q$-linear, surjective and nonconstant function $e_\phi : \mathbb{C}_\infty \longrightarrow \mathbb{C}_\infty$ called the *exponential* of the Drinfeld module ϕ. The exponential may be represented

[2] There is still some place for optimization, here. Indeed, the three algorithm rely eventually to the computation of the characteristic polynomial of an actual matrix with coefficients in $K[T]$. For this task, we just called the `charpoly` function of SageMath which, unfortunately, implements a slow generic algorithm with quartic complexity. Nevertheless, we believe that Figure 1 is meaningful in the sense that the comparison between timings are relevant.

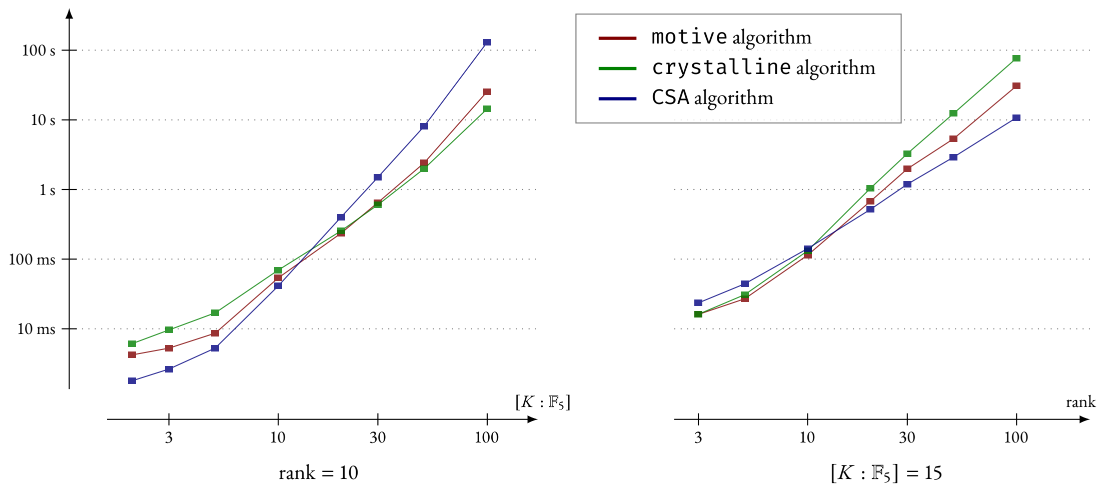

Figure 1: Timings for the computation of the characteristic polynomial of the Frobenius endomorphism
(CPU: Intel Core i5-8250U at 1.60GHz — OS: Ubuntu 22.04.1)

by a power series

$$e_\phi(z) = z + \sum_{i\geqslant 1} \alpha_i z^{q^i}$$

for $\alpha_i \in \mathbb{C}_\infty$ and $z \in \mathbb{C}_\infty$. The *logarithm* of ϕ, denoted $\log_\phi$ is the compositional inverse of the exponential. We refer the reader to chapter 4 of [Gos96] for more details. In our implementation, any Drinfeld module possesses the methods `exponential` and `logarithm` which compute power series approximation of e_ϕ and $\log_\phi$ respectively. The code computes the power series lazily, meaning that any coefficient is computed on demands and the user does not need to input any precision parameter.

Acknowledgements

We thank Pierre-Jean Spaenlehauer and Emmanuel Thomé for their guidance. David Ayotte was supported by FRQNT doctoral scholarship. This work also benefited from the financial support of the ANR projects CLap-CLap (ANR-18-CE40-0026-01) and PadLEfAn (ANR-22-CE40-0013).

References

[And86] Greg W. Anderson. *t*-Motives. *Duke Mathematical Journal*, 53(2):457 – 502, 1986.

[Car35] Leonard Carlitz. On certain functions connected with polynomials in a Galois field. *Duke Math. J.*, 1(2):137–168, 1935.

[Car18] Perlas Caranay. *Computing Isogeny Volcanoes of Rank Two Drinfeld Modules*. PhD thesis, University of Calgary, 2018.

[CGS20] Perlas Caranay, Mathew Greenberg, and Renate Scheidler. Computing modular polynomials and isogenies of rank two Drinfeld modules over finite fields. *Contemporary Mathematics*, 754:293–314, 2020.

[CL23] Xavier Caruso and Antoine Leudière. Computing norms and characteristic polynomials on general drinfeld modules. Manuscript in preparation., 2023.

[CLRS09] Keith Conrad, Franz Lemmermeyer, Peter J. Roquette, and Jean-Pierre Serre. History of class field theory. 2009.

[Dev] The SageMath Developers. *SageMath Reference Manual, Release 10.0*. To be released with SageMath Version 10.0.

[Dri74] Vladimir G. Drinfeld. Elliptic modules. *Mathematics of the Ussr-Sbornik*, 23(4):561–592, 1974.

[Gek83] Ernst-Ulrich Gekeler. Zur arithmetik von drinfel'd-moduln. *Math. Ann.*, 262(2):167–182, 1983.

[Gek91] Ernst-Ulrich Gekeler. On finite Drinfeld modules. *Journal of algebra*, 1(141):187–203, 1991.

[Gos96] David Goss. *Basic structures of function field arithmetic*, volume 35 of *Ergebnisse der Mathematik und ihrer Grenzgebiete (3) [Results in Mathematics and Related Areas (3)]*. Springer-Verlag, Berlin, 1996.

[Ham03] Yoshinori Hamahata. The values of *J*-invariants for Drinfeld modules. *Manuscripta Math.*, 112(1):93–108, 2003.

[Laf02] Laurent Lafforgue. Chtoucas de Drinfeld, formule des traces d'Arthur-Selberg et correspondance de Langlands. In *Proceedings of the International Congress of Mathematicians, Vol. I (Beijing, 2002)*, pages 383–400. Higher Ed. Press, Beijing, 2002.

[Lor96] Dino Lorenzini. *An Invitation to Arithmetic Geometry*. American Mathematical Society, 1996.

[LS22] Antoine Leudière and Pierre-Jean Spaenlehauer. Computing a group action from the class field theory of imaginary hyperelliptic function fields. 2022. arXiv:2203.06970.

[MS19] Yossef Musleh and Éric Schost. Computing the characteristic polynomial of a finite rank two Drinfeld module. In *Proceedings of the 2019 ACM on International Symposium on Symbolic and Algebraic Computation*, pages 307–314. ACM, 2019.

[MS23] Yossef Musleh and Éric Schost. Computing the characteristic polynomial of endomorphisms of a finite drinfeld module using crystalline cohomology. 2023. arXiv:2302.08611.

[Pot98] Igor Yu. Potemine. Minimal terminal **Q**-factorial models of Drinfeld coarse moduli schemes. *Math. Phys. Anal. Geom.*, 1(2):171–191, 1998.

[Sil94] Joseph H. Silverman. *Advanced topics in the arithmetic of elliptic curves*, volume 151 of *Graduate Texts in Mathematics*. Springer-Verlag, New York, 1994.

[Sil09] Joseph H. Silverman. *The arithmetic of elliptic curves*, volume 106 of *Graduate Texts in Mathematics*. Springer, Dordrecht, second edition, 2009.

[Wes22] Benjamin Wesolowski. Computing isogenies between finite drinfeld modules. Cryptology ePrint Archive, Paper 2022/438, 2022.

Tutorial: Drinfeld modules in SageMath

David Ayotte, Xavier Caruso, Antoine Leudière, Joseph Musleh

Try online at `https://caruso.perso.math.cnrs.fr/notebook/drinfeld-modules`

Contents

1 Construction and basic properties

1.1 Definitions

Let $\mathbb{F}_q$ denote a finite field with q elements. We consider a $\mathbb{F}_q$-algebra A and an A-algebra K. We denote by $\gamma : A \to K$ the defining morphism of K.

Let also $K\{\tau\}$ denote the ring of *skew* polynomials over K in the variable τ, that is the ring of usual polynomials with multiplication twisted by the rule

$$\tau a = a^q \tau, \quad \forall a \in K.$$

By definition, a A-Drinfeld module is a ring homomorphism $\phi : A \to K\{\tau\}$ such that

- for all $a \in A$, the constant coefficient of $\phi(a)$ is $\gamma(a)$,
- for all $a \in A$, $a \notin \mathbb{F}_q$, the skew polynomial $\phi(a)$ has positive degree.

We often write ϕ_a instead of $\phi(a)$.

In what follows, we will only consider the case where $A = \mathbb{F}_q[T]$ (our implementation is limited to this setting so far) and will simply say *Drinfeld module* instead of $\mathbb{F}_q[T]$-Drinfeld module. We remark that a Drinfeld module ϕ is entirely defined by the datum of ϕ_T.

1.2 Construction in SageMath

We first define $\mathbb{F}_q$, A and K.

```
[1]: Fq = GF(5)
     A.<T> = Fq[]
     K.<z> = Fq.extension(3)
```

We can now create a Drinfeld module using the constructor `DrinfeldModule`:

```
[2]: phi = DrinfeldModule(A, [z, 0, 1, 1])
     phi
```

```
[2]: Drinfeld module defined by T |--> t^3 + t^2 + z
```

The first argument of the constructor is the base ring `A`. The second argument can be either:

- the skew polynomial ϕ_T, or
- the list of coefficients of ϕ_T (as in the example above).

In both cases, one can recover the underlying skew polynomial ring using the method `ore_polring`:

```
[3]: phi.ore_polring()
```

```
[3]: Ore Polynomial Ring in t over Finite Field in z of size 5^3 over its base
     twisted by Frob
```

The image of a polynomial $a \in A$ can be obtained by simply calling `phi(a)`:

```
[4]: phi(T)
```

```
[4]: t^3 + t^2 + z
```

```
[5]: phi(T^2 + 1)
```

```
[5]: t^6 + 2*t^5 + t^4 + 2*z*t^3 + (3*z^2 + z + 1)*t^2 + z^2 + 1
```

```
[6]: phi(T^2 + 1) == phi(T)^2 + 1    # basic check: phi is a ring homomorphism
```

```
[6]: True
```

1.3 Basic properties

Many standard invariants attached to Drinfeld modules can be computed.

1.3.1 Characteristic

The *characteristic* of a Drinfeld module ϕ is, by definition, (a generator of) the kernel of γ.

```
[7]: phi.characteristic()
```

```
[7]: T^3 + 3*T + 3
```

1.3.2 Rank

The *rank* of a Drinfeld module ϕ is, by definition, the degree of the skew polynomial ϕ_T. More generally, it satisfies the relation:

$$\forall a \in \mathbb{F}_q[T], \quad \deg \phi_a = \operatorname{rank}(\phi) \cdot \deg a.$$

[8]:
```
phi.rank()
```

[8]:
```
3
```

[9]:
```
a = T^2 + 1
phi(a).degree() == phi.rank() * a.degree()
```

[9]:
```
True
```

1.3.3 Height

Let $\mathfrak{p}$ be the characteristic of ϕ. The height of ϕ is, by definition, the quotient of the τ-valuation of $\phi_\mathfrak{p}$ be the degree of $\mathfrak{p}$; it is always a positive integer.

Remark: The height is an important invariant because it is related to the rank of the $\mathfrak{p}$-torsion points of the Drinfeld module.

[10]:
```
phi.height()
```

[10]:
```
1
```

Having height 1 is the generic case. However, Drinfeld modules with higher heights also exist. Here is an example in rank 2:

[11]:
```
psi = DrinfeldModule(A, [z, 0, 1])
psi.height()
```

[11]:
```
2
```

1.4 Categories of Drinfeld modules

When a Drinfeld module is creates, SageMath automatically creates at the same time the category in which it lives. This category remembers the base rings A and K and the defining morphism $\gamma : A \to K$.

[12]:
```
C = phi.category()
C
```

[12]:
```
Category of Drinfeld modules over Finite Field in z of size 5^3 over its base
```

Having this category as an actual object in SageMath is important for dealing with morphisms between Drinfeld modules (as we shall see later on). Besides, several important invariants are defined at the level of the category:

[13]:
```
C.base()             # The field K, vue as an algebra over A
```

```
[13]: Finite Field in z of size 5^3 over its base
```

```
[14]: C.base_morphism()    # The defining morphism gamma
                           # also accessible by C.base().defining_morphism()
```

```
[14]: Ring morphism:
        From: Univariate Polynomial Ring in T over Finite Field of size 5
        To:   Finite Field in z of size 5^3 over its base
        Defn: T |--> z
```

```
[15]: C.ore_polring()      # The ring K{t}
```

```
[15]: Ore Polynomial Ring in t over Finite Field in z of size 5^3 over its base
      twisted by Frob
```

```
[16]: C.characteristic()
```

```
[16]: T^3 + 3*T + 3
```

2 Morphisms and isogenies

2.1 Definitions

Let $\phi, \psi : \mathbb{F}_q[T] \to K\{\tau\}$ be two Drinfeld modules in the same category.

By definition, a *morphism* $\phi \to \psi$ is the datum of $u \in K\{\tau\}$ such that $u\phi_T = \psi_T u$. (This relation implies more generally that $u\phi_a = \psi_a u$ for all $a \in \mathbb{F}_q[T]$.)

An *isogeny* is, by definition, a nonzero morphism. We observe that if an isogeny $\phi \to \psi$ exists, then ϕ and ψ must have the same rank.

2.2 Construction

[1]:
```
Fq = GF(5)
A.<T> = Fq[]
K.<z> = Fq.extension(3)
phi = DrinfeldModule(A, [z, 0, 1, z])
```

2.2.1 The constructor hom

The simplest way to create a morphism is to use the method `hom`, to which we can directly pass in the defining skew polynomial:

[2]:
```
t = phi.ore_variable()
f = phi.hom(t + 1)
f
```

[2]:
```
Drinfeld Module morphism:
  From: Drinfeld module defined by T |--> z*t^3 + t^2 + z
  To:   Drinfeld module defined by T |--> (2*z^2 + 4*z + 4)*t^3 + (3*z^2 +␣
 ↪2*z +
2)*t^2 + (2*z^2 + 3*z + 4)*t + z
  Defn: t + 1
```

On the example above, we observe that the codomain of the isogeny (which is not ϕ) was automatically determined:

[3]:
```
psi = f.codomain()
psi
```

[3]:
```
Drinfeld module defined by T |--> (2*z^2 + 4*z + 4)*t^3 + (3*z^2 + 2*z +␣
 ↪2)*t^2
+ (2*z^2 + 3*z + 4)*t + z
```

An important class of endomorphisms of a given Drinfeld module ϕ are scalar multiplications: they are endomorphisms corresponding to the skew polynomial ϕ_a for $a \in \mathbb{F}_q[T]$. Those endomorphisms can be simply instantiated as follows:

[4]:
```
g = phi.hom(T)
g
```

[4]:
```
Endomorphism of Drinfeld module defined by T |--> z*t^3 + t^2 + z
  Defn: z*t^3 + t^2 + z
```

2.2.2 Set of morphisms between two Drinfeld modules

When we know the domain ϕ and the codomain ψ, we can construct the *homset* $\mathrm{Hom}(\phi, \psi)$:

[5]:
```
H = Hom(phi, psi)
H
```

[5]:
```
Set of Drinfeld module morphisms from (gen) z*t^3 + t^2 + z to (gen) (2*z^2 +
4*z + 4)*t^3 + (3*z^2 + 2*z + 2)*t^2 + (2*z^2 + 3*z + 4)*t + z
```

As a shortcut, when $\phi = \psi$, we can use the `End` primitive:

[6]:
```
E = End(phi)
E
```

[6]:
```
Set of Drinfeld module morphisms from (gen) z*t^3 + t^2 + z to (gen) z*t^3 +␣
 ↪t^2
+ z
```

Notice that those homsets are the parents in which `f` and `g` (which we created earlier) naturally live:

[7]:
```
f.parent() is H
```

[7]:
```
True
```

[8]:
```
g.parent() is E
```

[8]:
```
True
```

Once the homset is defined, we can call it to create morphisms in it (as it is usual in SageMath). This provides an alternative to the `hom` constructor to define morphisms. For example:

[9]:
```
Frob = E(t^3)
Frob
```

[9]:
```
Endomorphism of Drinfeld module defined by T |--> z*t^3 + t^2 + z
  Defn: t^3
```

The latter is the Frobenius endomorphism. It is also available *via* the method `frobenius_endomorphism`:

[10]:
```
phi.frobenius_endomorphism()
```

[10]:
```
Endomorphism of Drinfeld module defined by T |--> z*t^3 + t^2 + z
  Defn: t^3
```

2.3 About the structure of Hom(ϕ, ψ) and End(ϕ)

First of all, of course, there is a composition law on the homsets: when the domain of f is equal to the codomain of g, one can compose f and g, producing a new morphism $f \circ g$. In the Drinfeld module setting, this operation simply corresponds to the multiplication of the underlying skew polynomials.

It is available in SageMath using the multiplication operator `*` or the exponentiation operator `^` (or `**`):

```
[11]: f * g
```

```
[11]: Drinfeld Module morphism:
        From: Drinfeld module defined by T |--> z*t^3 + t^2 + z
        To:   Drinfeld module defined by T |--> (2*z^2 + 4*z + 4)*t^3 + (3*z^2 +␣
       ↪2*z +
      2)*t^2 + (2*z^2 + 3*z + 4)*t + z
        Defn: (2*z^2 + 4*z + 4)*t^4 + (z + 1)*t^3 + t^2 + (2*z^2 + 4*z + 4)*t + z
```

```
[12]: Frob^5
```

```
[12]: Endomorphism of Drinfeld module defined by T |--> z*t^3 + t^2 + z
        Defn: t^15
```

We observe that the sum of two morphisms $\phi \to \psi$ is well-defined. Indeed if u and v satisfies the relation $\phi_T u = u\psi_T$ and $\phi_T v = v\psi_T$ then one immediately deduce that $\phi_T(u+v) = (u+v)\psi_T$. Therefore $\mathrm{Hom}(\phi, \psi)$ is naturally a commutative group for the addition.

Besides, given that elements of $\mathbb{F}_q[T]$ define endomorphisms of any Drinfeld module:

- the Hom space $\mathrm{Hom}(\phi, \psi)$ inherits a structure of left $\mathbb{F}_q[T]$-module (composing on the left by the endomorphism ψ_a of ψ),
- the End space $\mathrm{End}(\phi)$ inherits a structure of $\mathbb{F}_q[T]$-algebra.

Our implementation handles those quite transparently. For example:

```
[13]: T * f
```

```
[13]: Drinfeld Module morphism:
        From: Drinfeld module defined by T |--> z*t^3 + t^2 + z
        To:   Drinfeld module defined by T |--> (2*z^2 + 4*z + 4)*t^3 + (3*z^2 +␣
       ↪2*z +
      2)*t^2 + (2*z^2 + 3*z + 4)*t + z
        Defn: (2*z^2 + 4*z + 4)*t^4 + (z + 1)*t^3 + t^2 + (2*z^2 + 4*z + 4)*t + z
```

```
[14]: Frob + g
```

```
[14]: Endomorphism of Drinfeld module defined by T |--> z*t^3 + t^2 + z
        Defn: (z + 1)*t^3 + t^2 + z
```

```
[15]: Frob * g == g * Frob
```

```
[15]: True
```

```
[16]: (Frob + g)^5 == Frob^5 + g^5
```

```
[16]: True
```

2.4 Recognizing isomorphisms

It follows easily from the definition that a morphism $f : \phi \to \psi$ is an isomorphism if and only if the skew polynomial defining f is constant.

The method `is_isomorphism` allows for checking this fact.

```
[17]: g.is_isomorphism()
```

```
[17]: False
```

```
[18]: h = phi.hom(z)
      h
```

```
[18]: Drinfeld Module morphism:
        From: Drinfeld module defined by T |--> z*t^3 + t^2 + z
        To:   Drinfeld module defined by T |--> z*t^3 + (4*z^2 + z + 4)*t^2 + z
        Defn: z
```

```
[19]: h.is_isomorphism()
```

```
[19]: True
```

The method `inverse` computes the inverse of an isomorphism:

```
[20]: h.inverse()
```

```
[20]: Drinfeld Module morphism:
        From: Drinfeld module defined by T |--> z*t^3 + (4*z^2 + z + 4)*t^2 + z
        To:   Drinfeld module defined by T |--> z*t^3 + t^2 + z
        Defn: 3*z^2 + 4
```

It is also possible to check if two given Drinfeld modules are isomorphic using the method `is_isomorphic`:

```
[21]: phi2 = h.codomain()
      phi.is_isomorphic(phi2)
```

```
[21]: True
```

```
[22]: phi.is_isomorphic(psi)
```

```
[22]: False
```

In the last case, ϕ and ψ are isogenous (*via* f) but they are not isomorphic.

2.5 The case of finite fields

When K is a finite field, the subspace of $\mathrm{Hom}(\phi, \psi)$ consisting of morphisms defined by a skew polynomial of degree at most d (for some given positive integer d) is a finite dimensional $\mathbb{F}_q$-vector space.

The method `basis` on the homset returns a basis of this vector space:

```
[23]: H.basis(degree=5)
```

```
[23]: [Drinfeld Module morphism:
         From: Drinfeld module defined by T |--> z*t^3 + t^2 + z
         To:   Drinfeld module defined by T |--> (2*z^2 + 4*z + 4)*t^3 + (3*z^2 +␣
       ↪2*z
      + 2)*t^2 + (2*z^2 + 3*z + 4)*t + z
         Defn: t + 1,
       Drinfeld Module morphism:
         From: Drinfeld module defined by T |--> z*t^3 + t^2 + z
         To:   Drinfeld module defined by T |--> (2*z^2 + 4*z + 4)*t^3 + (3*z^2 +␣
       ↪2*z
      + 2)*t^2 + (2*z^2 + 3*z + 4)*t + z
         Defn: (2*z^2 + 4*z + 3)*t^4 + z*t^3 + t^2 + (2*z^2 + 4*z + 4)*t + z,
       Drinfeld Module morphism:
         From: Drinfeld module defined by T |--> z*t^3 + t^2 + z
         To:   Drinfeld module defined by T |--> (2*z^2 + 4*z + 4)*t^3 + (3*z^2 +␣
       ↪2*z
      + 2)*t^2 + (2*z^2 + 3*z + 4)*t + z
         Defn: (3*z^2 + 3*z + 1)*t^4 + (3*z^2 + 4*z)*t^3 + (3*z^2 + z + 1)*t^2 +␣
       ↪(2*z
      + 3)*t + z^2,
       Drinfeld Module morphism:
         From: Drinfeld module defined by T |--> z*t^3 + t^2 + z
         To:   Drinfeld module defined by T |--> (2*z^2 + 4*z + 4)*t^3 + (3*z^2 +␣
       ↪2*z
      + 2)*t^2 + (2*z^2 + 3*z + 4)*t + z
         Defn: t^4 + t^3]
```

Over finite fields, we also have one method for checking whether a Hom space is reduced to zero.

```
[24]: H.is_zero()
```

```
[24]: False
```

```
[25]: psi2 = DrinfeldModule(A, [z, 0, 1, z^2])
      H2 = Hom(phi, psi2)
      H2.is_zero()
```

```
[25]: True
```

Besides, when a Hom space is nonzero, we can use the method `an_isogeny` to produce a nonzero morphism in it:

```
[26]: H.an_isogeny()
```

```
[26]: Drinfeld Module morphism:
        From: Drinfeld module defined by T |--> z*t^3 + t^2 + z
        To:   Drinfeld module defined by T |--> (2*z^2 + 4*z + 4)*t^3 + (3*z^2 +␣
       ↪2*z +
      2)*t^2 + (2*z^2 + 3*z + 4)*t + z
```

```
Defn: t + 1
```

3 j-invariants

3.1 Definition

The j-invariants of a Drinfeld module form a family of elements which caracterizes the isomorphism class of the Drinfeld module over an algebraic closure.

More precisely, let $\phi : \mathbb{F}_q[T] \to K\{\tau\}$ be a Drinfeld module and write

$$\phi_T = g_0 + g_1\tau + \cdots + g_r\tau^r$$

where r denotes the rank of ϕ. Let $(k_1, \dots, k_n)$, $(d_1, \dots, d_n)$ be two n-tuple of integers such that $1 \leq k_1 < k_2 < \dots < k_n < r$ and $d_i > 0$ for all i. We consider an additional nonnegative integer d and assume the so-called *weight-0 condition*:

$$d_1(q^{k_1} - 1) + d_2(q^{k_2} - 1) + \cdots + d_n(q^{k_n} - 1) = d(q^r - 1).$$

To this datum, we associate the following quantity, called the $((k_1, \dots, k_n), (d_1, \dots, d_n, d))$-$j$-*invariant* of ϕ:

$$j_{k_1,\dots,k_n}^{d_1,\dots,d_n,d}(\phi) := \frac{1}{g_r^d} \prod_{i=1}^{n} g_{k_i}^{d_i}$$

A j-invariant is called *basic* when $\gcd(d_1, \dots, d_n, d) = 1$. There is only a finite number of basic j-invariants. Moreover, if $((k_1, \dots, k_n), (d_1, \dots, d_n, d))$ is any parameter, we can cook up a basic one simply by dividing the d_i's and d by their gcd. The j-invariant corresponding to the initial parameter is equal to the basic j-invariant raised to the power $\gcd(d_1, \dots, d_n, d)$.

A classical result, due to Potemine, asserts that two Drinfeld module ϕ and ψ are isomorphic over $\bar{K}$ if and only if their basic j-invariants are all equal.

3.2 Computation of j-invariants

[1]:
```
Fq = GF(5)
A.<T> = Fq[]
K.<z> = Fq.extension(3)
phi = DrinfeldModule(A, [z, 0, 1, z, z^2])
```

The first important thing to know is the (finite) list of parameters $((k_1, \dots, k_n), (d_1, \dots, d_n, d))$ leading to basic j-invariants. This list is accessible *via* the method `basic_j_invariant_parameters`.

[2]:
```
parameters = phi.basic_j_invariant_parameters()
parameters
```

[2]:
```
[((1, 2, 3), (26, 1, 4, 1)),
 ((1, 2, 3), (57, 1, 3, 1)),
 ((1, 2, 3), (88, 1, 2, 1)),
 ...
 ((1, 2, 3), (93, 26, 153, 32)),
 ((1, 2, 3), (155, 26, 151, 32)),
 ((1, 2, 3), (156, 26, 156, 33))]
```

For a Drinfeld module of rank 5, this list is quite long:

```
[3]: len(parameters)
```

```
[3]: 3402
```

However, when certain coefficients of ϕ_T vanish, some j-invariants will vanish as well as the latter are defined as products of the formers. In this case, it is useful to know the list of parameters leading to *nonzero* basic j-invariants. In order to get it, we can simply pass in the keyword `nonzero=True`.

```
[4]: nonzero_parameters = phi.basic_j_invariant_parameters(nonzero=True)
     nonzero_parameters
```

```
[4]: [((2, 3), (1, 30, 6)),
      ((2, 3), (6, 24, 5)),
      ((2, 3), (7, 54, 11)),
      ((2, 3), (8, 84, 17)),
      ((2, 3), (9, 114, 23)),
      ((2, 3), (10, 144, 29)),
      ((2, 3), (11, 18, 4)),
      ((2, 3), (13, 78, 16)),
      ((2, 3), (15, 138, 28)),
      ((2, 3), (16, 12, 3)),
      ((2, 3), (17, 42, 9)),
      ((2, 3), (19, 102, 21)),
      ((2, 3), (20, 132, 27)),
      ((2, 3), (21, 6, 2)),
      ((2, 3), (23, 66, 14)),
      ((2, 3), (25, 126, 26))]
```

Once we know a valid parameter, we can compute the j-invariant, using the method `j_invariant`:

```
[5]: parameter = nonzero_parameters[0]
     phi.j_invariant(parameter)
```

```
[5]: 4*z^2 + 2*z + 1
```

It is also possible to get the complete list of j-invariants by calling the method `basic_j_invariants`:

```
[6]: phi.basic_j_invariants(nonzero=True)
```

```
[6]: {((2, 3), (1, 30, 6)): 4*z^2 + 2*z + 1,
      ((2, 3), (6, 24, 5)): 4*z + 3,
      ((2, 3), (7, 54, 11)): 2*z,
      ((2, 3), (8, 84, 17)): 4*z^2 + 3*z + 1,
      ((2, 3), (9, 114, 23)): z^2 + 2*z + 1,
      ((2, 3), (10, 144, 29)): 2*z^2 + 2*z + 1,
      ((2, 3), (11, 18, 4)): 2*z + 3,
      ((2, 3), (13, 78, 16)): 3*z^2 + z,
      ((2, 3), (15, 138, 28)): z^2 + 3*z + 1,
      ((2, 3), (16, 12, 3)): 4*z^2 + 3*z + 4,
      ((2, 3), (17, 42, 9)): 3*z^2 + 3*z + 4,
      ((2, 3), (19, 102, 21)): 3*z^2 + 2*z + 4,
      ((2, 3), (20, 132, 27)): 2*z^2 + 2*z + 2,
      ((2, 3), (21, 6, 2)): z^2,
      ((2, 3), (23, 66, 14)): z^2 + 4*z + 1,
      ((2, 3), (25, 126, 26)): 2*z^2 + z + 1}
```

3.2.1 j_k-invariants

The j-invariants associated to the basic parameter

$$\left(k, \left(\frac{q^r - 1}{q^{\gcd(r,k)} - 1}, \frac{q^k - 1}{q^{\gcd(r,k)} - 1}\right)\right)$$

is known as j_k-invariants. In this case, the method `j_invariant` accepts the integer `k` as parameter as well.

[7]:
```
phi.j_invariant(3)
```

[7]:
```
3*z
```

[8]:
```
phi.j_invariant([[3], [156, 31]])
```

[8]:
```
3*z
```

3.3 j-invariants in small rank

3.3.1 The case of rank 1

[9]:
```
phi = DrinfeldModule(A, [z, 1])
phi
```

[9]:
```
Drinfeld module defined by T |--> t + z
```

In rank 1, there is no basic j-invariant.

[10]:
```
phi.basic_j_invariant_parameters()
```

[10]:
```
[]
```

It is then not relevant to compute j-invariant.

Besides, by Potemime's theorem, we conclude that all Drinfeld module of rank 1 with the same defining morphism $\gamma : \mathbb{F}_q[T] \to K$ are isomorphic over an algebraic closure $\bar{K}$ of K. It is not true however that they are isomorphic over K.

[11]:
```
psi = DrinfeldModule(A, [z, z])
phi.is_isomorphic(psi)
```

[11]:
```
False
```

```
[12]: phi.is_isomorphic(psi, absolutely=True)
```

```
[12]: True
```

3.3.2 The case of rank 2

```
[13]: phi = DrinfeldModule(A, [z, z^2, z^3])
      phi
```

```
[13]: Drinfeld module defined by T |--> (2*z + 2)*t^2 + z^2*t + z
```

For Drinfeld module of rank 2, there is only one basic j-invariant, namely $(1, (q + 1, 1))$

```
[14]: phi.basic_j_invariant_parameters()
```

```
[14]: [((1,), (6, 1))]
```

In this case (and only in this case!), it is allowed to call the method `j_invariant` without any parameter.

```
[15]: phi.j_invariant()
```

```
[15]: 4*z^2 + 4
```

4 Norms and characteristic polynomials

4.1 Definitions

4.1.1 The case of endomorphisms

Let $\phi : \mathbb{F}_q[T] \to K\{\tau\}$ be a Drinfeld module of characteristic $\mathfrak{p}$ and let $\mathfrak{q}$ be a maximal ideal of $\mathbb{F}_q[T]$ different from $\mathfrak{p}$.

For each positive integer n, one considers the set of $\mathbf{q}^n$-torsion points of ϕ defined by:

$$\phi[\mathfrak{q}^n] = \big\{x \in \bar{K} : \phi_a(x) = 0, \forall a \in \mathfrak{q}^n\big\}$$

where we agree that the variable τ in the skew polynomial ϕ_a acts as the Frobenius $x \mapsto x^q$. The space $\phi[\mathfrak{q}^n]$ inherits a structure of $\mathbb{F}_q[T]$-module by letting $a \in \mathbb{F}_q[T]$ acting as $x \mapsto \phi_a(x)$. By definition $\phi[\mathfrak{q}^n]$ is killed by $\mathfrak{q}^n$.

The Tate module of ϕ is defined by:

$$T_{\mathfrak{q}}(\phi) = \varprojlim_{n\to\infty} \phi[\mathfrak{q}^n]$$

it is a module over the ring $A_{\mathfrak{q}}$, defined as the completion of $\mathbb{F}_q[T]$ at the place $\mathfrak{q}$, *i.e.* $A_{\mathfrak{q}} = \varprojlim_{n\to\infty} \mathbb{F}_q[T]/\mathfrak{q}^n$. If r denotes the rank of ϕ, then it is a standard result that $T_{\mathfrak{q}}(\phi)$ is free of rank r over $A_{\mathfrak{q}}$.

Any endomorphism $f : \phi \to \phi$ induces a $A_{\mathfrak{q}}$-linear map $T_{\mathfrak{q}}(f) : T_{\mathfrak{q}}(\phi) \to T_{\mathfrak{q}}(\phi)$. By definition:

- the *norm* of f is the determinant of $T_{\mathfrak{q}}(f)$,
- the *characteristic polynomial* of f is the characteristic polynomial of $T_{\mathfrak{q}}(f)$.

One proves that they both do not depend on the choice of $\mathfrak{q}$ and that the former lies in $\mathbb{F}_q[T]$ while the latter is a polynomial with coefficients in $\mathbb{F}_q[T]$.

4.1.2 The case of general morphisms

We now consider a morphism $f : \phi \to \psi$ between two different Drinfeld modules ϕ and ψ. In this setting, the characteristic polynomial of f is no longer defined but the norm of f continues to make sense as an ideal (and not an actual element) of $\mathbb{F}_q[T]$.

In order to define it, we consider the application

$$\begin{array}{rccc} T(f): & \bar{K} & \to & \bar{K} \\ & x & \mapsto & u(x) \end{array}$$

where u is the skew polynomial defining f. We notice that $T(f)$ is $\mathbb{F}_q[T]$-linear if we endow the domain and the codomain with the structure of $\mathbb{F}_q[T]$-module coming from ϕ and ψ respectively. If f is nonzero, the cokernel of $T(f)$ is a finitely generated torsion $\mathbb{F}_q[T]$-module; therefore, it is of the form

$$\text{coker } T(f) \simeq \mathbb{F}_q[T]/\mathfrak{a}_1 \times \mathbb{F}_q[T]/\mathfrak{a}_2 \times \cdots \times \mathbb{F}_q[T]/\mathfrak{a}_m$$

for some ideals $\mathfrak{a}_1, \ldots, \mathfrak{a}_m$ of $\mathbb{F}_q[T]$. The *norm* of f is defined by $\nu(f) = \mathfrak{a}_1 \cdots \mathfrak{a}_m$ (product of ideals).

4.2 Computing norms and characteristic polynomials In SageMath

As we see, the definition of norms and characteristic polynomials involve intermediate complicated objects (namely, the Tate module $T_{\mathfrak{q}}(f)$ or the map $T(f)$) which themselves involve an algebraic closure of K and thus looks difficult to implement while keeping good performances.

However, our implementation provides facilities for computing efficiently norms and characteristic polynomials of arbitrary morphisms/endomorphisms. Those are based on alternative definitions of norms and characteristic polynomials coming from the theory of Anderson motives.

4.2.1 Endomorphisms

```
[1]: Fq = GF(5)
     A.<T> = Fq[]
     K.<z> = Fq.extension(3)
     phi = DrinfeldModule(A, [z, 0, 1, z])
```

We start our tour by computing the norm of the Frobenius endomorphism:

```
[2]: Frob = phi.frobenius_endomorphism()
     Frob.norm()
```

```
[2]: Principal ideal (T^3 + 3*T + 3) of Univariate Polynomial Ring in T over Finite
     Field of size 5
```

For consistency, the method `norm` always returns an ideal by default. This behavior can be overrided by passing in the argument `ideal=False`:

```
[3]: Frob.norm(ideal=False)
```

```
[3]: 3*T^3 + 4*T + 4
```

We observe that the norm of the Frobenius is $\mathbb{F}_q$-collinear to the characteristic, which is a standard fact:

```
[4]: phi.characteristic()
```

```
[4]: T^3 + 3*T + 3
```

```
[5]: 3 * phi.characteristic()
```

```
[5]: 3*T^3 + 4*T + 4
```

We can compute similarly the norm of the endomorphisms ϕ_a for some $a \in \mathbb{F}_q[T]$

```
[6]: f = phi.hom(T)
     f.norm()
```

```
[6]: Principal ideal (T^3) of Univariate Polynomial Ring in T over Finite Field of
     size 5
```

[7]:
```
g = phi.hom(T+1)
g.norm()
```

[7]:
```
Principal ideal (T^3 + 3*T^2 + 3*T + 1) of Univariate Polynomial Ring in T␣
 ↪over
Finite Field of size 5
```

In each case, we observe that the norm of ϕ_a is the ideal generated by a^3; it is actually a general fact that the norm of ϕ_a is $a^r \mathbb{F}_q[T]$ where r is the rank of the underlying Drinfeld module.

Similarly the characteric polynomial can be computed thanks to the method `charpoly` (or `characteristic_polynomial`):

[8]:
```
Frob.charpoly()
```

[8]:
```
X^3 + (T + 1)*X^2 + (2*T + 3)*X + 2*T^3 + T + 1
```

[9]:
```
f.charpoly()
```

[9]:
```
X^3 + 2*T*X^2 + 3*T^2*X + 4*T^3
```

[10]:
```
g.charpoly(var='Y')   # We can change the variable name
```

[10]:
```
Y^3 + (2*T + 2)*Y^2 + (3*T^2 + T + 3)*Y + 4*T^3 + 2*T^2 + 2*T + 4
```

Again, we can check that the characteristic polynomial of ϕ_a is $(X - a)^r$:

[11]:
```
g.charpoly().factor()
```

[11]:
```
(X + 4*T + 4)^3
```

4.2.2 General morphisms

For general morphisms, only the method `norm` is available (given that the characteristic polynomial is not defined in this generality).

[12]:
```
t = phi.ore_variable()
h = phi.hom(t + 1)
h
```

[12]:
```
Drinfeld Module morphism:
  From: Drinfeld module defined by T |--> z*t^3 + t^2 + z
  To:   Drinfeld module defined by T |--> (2*z^2 + 4*z + 4)*t^3 + (3*z^2 +␣
 ↪2*z +
2)*t^2 + (2*z^2 + 3*z + 4)*t + z
  Defn: t + 1
```

[13]:
```
h.norm()
```

[13]:
```
Principal ideal (T + 4) of Univariate Polynomial Ring in T over Finite Field␣
 ↪of
size 5
```

We verify, on examples, that the norm is multiplicative:

```
[14]: (h * Frob).norm() == h.norm() * Frob.norm()
```

```
[14]: True
```

```
[15]: (h * f).norm() == h.norm() * f.norm()
```

```
[15]: True
```

```
[16]: (Frob * g).norm() == Frob.norm() * g.norm()
```

```
[16]: True
```

4.3 The case of the Frobenius endomorphism

When K is a finite field, the characteristic polynomial of the Frobenius endomorphism is an invariant of primary importance. Our implementation provides a direct method for accessing it:

```
[17]: phi.frobenius_charpoly()
```

```
[17]: X^3 + (T + 1)*X^2 + (2*T + 3)*X + 2*T^3 + T + 1
```

Actually, several different algorithms are available for this computations. They are accessible by passing in the keyword `algorithm` which can be:

- `gekeler`: it tries to identify coefficients by writing that the characteristic polynomial annihilates the Frobenius endomorphism; this algorithm may fail is some cases;
- `motive`: it uses the action of the Frobenius on the Anderson motive;
- `crystalline`: it uses the action of the Frobenius on the crystalline cohomology;
- `CSA`: it exploits the structure of central simple algebra of Frac $K\{\tau\}$.

Of course, all those algorithms output the same answer but performances may vary. By default, the `crystalline` algorithm is chosen when the rank is smaller than the degree of the extension $K/\mathbb{F}_q$ whereas the `CSA` algorithm is chosen otherwise.

4.3.1 Isogeny test

An important result asserts that two Drinfeld modules ϕ and ψ are isogenous if and only if the characteristic polynomials of the Frobenius endomorphisms of ϕ and ψ agrees. This provides an efficient isogeny test, which is available in our package *via* the method `is_isogenous`.

```
[18]: psi = h.codomain()   # recall that h was an isogeny with domain phi
      phi.is_isogenous(psi)
```

```
[18]: True
```

```
[19]: phi2 = DrinfeldModule(A, [z, 0, 1, z^2])
      phi.is_isogenous(phi2)
```

```
[19]: False
```

Finally, we mention that, when an isogeny does exist, one can find it using the method an_isogeny on the homset.

[20]:
```
Hom(phi, psi).an_isogeny()
```

[20]:
```
Drinfeld Module morphism:
  From: Drinfeld module defined by T |--> z*t^3 + t^2 + z
  To:   Drinfeld module defined by T |--> (2*z^2 + 4*z + 4)*t^3 + (3*z^2 +␣
 ↪2*z +
2)*t^2 + (2*z^2 + 3*z + 4)*t + z
  Defn: t + 1
```

4.3.2 Timings

Below, we compare timings.

When $[K : \mathbb{F}_q]$ is large, the crystalline algorithm is the fastest:

[21]:
```
Fq = GF(5)
A.<T> = Fq[]
K.<z> = Fq.extension(50)
phi = DrinfeldModule(A, [z] + [K.random_element() for _ in range(5)] + [1])
```

[22]:
```
%time _ = phi.frobenius_charpoly(algorithm="motive")
```

```
CPU times: user 969 ms, sys: 5.7 ms, total: 975 ms
Wall time: 981 ms
```

[23]:
```
%time _ = phi.frobenius_charpoly(algorithm="crystalline")
```

```
CPU times: user 623 ms, sys: 2.98 ms, total: 626 ms
Wall time: 629 ms
```

[24]:
```
%time _ = phi.frobenius_charpoly(algorithm="CSA")
```

```
CPU times: user 3.92 s, sys: 14.6 ms, total: 3.93 s
Wall time: 3.96 s
```

On the contrary, when the rank is large, the CSA algorithm is the fastest:

[25]:
```
Fq = GF(5)
A.<T> = Fq[]
K.<z> = Fq.extension(5)
phi = DrinfeldModule(A, [z] + [K.random_element() for _ in range(50)] + [1])
```

[26]:
```
%time _ = phi.frobenius_charpoly(algorithm="motive")
```

```
CPU times: user 1.15 s, sys: 12.9 ms, total: 1.16 s
Wall time: 1.17 s
```

[27]:
```
%time _ = phi.frobenius_charpoly(algorithm="crystalline")
```

```
CPU times: user 2.08 s, sys: 13.9 ms, total: 2.1 s
Wall time: 2.11 s
```

```
[28]: %time _ = phi.frobenius_charpoly(algorithm="CSA")
```

```
CPU times: user 40 ms, sys: 1.97 ms, total: 42 ms
Wall time: 41.8 ms
```

5 Exponential and logarithm

5.1 Definitions

Consider the rational function field $K = \mathbb{F}_q(T)$ and let $K_\infty = \mathbb{F}_q((\frac{1}{T}))$ be the completion of this field at the place $\frac{1}{T}$. We define $\mathbb{C}_\infty$ to be the completion of an algebraic closure of K_∞. The field $\mathbb{C}_\infty$ plays a similar role to the field of complex numbers.

An important aspect of Drinfeld modules is their analytic point of view. Given any Drinfeld module $\phi : \mathbb{F}_q[T] \to \mathbb{C}_\infty\{\tau\}$ of rank r, one can show that there exists a $\mathbb{F}_q[T]$-lattice of rank r, *i.e.* a free $\mathbb{F}_q[T]$-submodule of $\mathbb{C}_\infty$ of rank r, denoted by Λ_ϕ, such that for any $z \in \mathbb{C}_\infty$ and $a \in \mathbb{F}_q[T]$

$$\phi_a(z) = z \prod_{\substack{\lambda \in a^{-1}\Lambda_\phi/\Lambda_\phi \\ \lambda \neq 0}} \left(1 - \frac{z}{e_\phi(\lambda)}\right)$$

where $e_\phi : \mathbb{C}_\infty \to \mathbb{C}_\infty$ is a $\mathbb{F}_q$-linear, surjective and nonconstant function. In particular, it may be written as a power series

$$e_\phi(z) = z + \sum_{i \geq 1} \alpha_i z^{q^i}.$$

This function is called the *exponential* of ϕ. The compositional inverse of e_ϕ exists and is named the *logarithm* of ϕ, denoted $\log_\phi$:

$$\log_\phi(z) = z + \sum_{i \geq 1} \beta_i z^{q^i}.$$

5.2 Explanations

We explain here how the code work. In short, given any Drinfeld module

$$\phi : T \mapsto g_0 + g_1\tau + \cdots + g_r\tau^r,$$

we first compute the logarithm using a recursive procedure and then we revert the obtained power series to get the exponential.

More precisely, we use the following functional equation:

$$e_\phi(az) = \phi_a(e_\phi(z)).$$

Applying $\log_\phi$ on both side of the functional equation we get

$$a \log_\phi(z) = \log_\phi(\phi_a(z)).$$

Then, one may compare the coefficients on both side of the above equation and obtain the recursive procedure:

$$a\beta_i = \sum_{n+m=i} \beta_n g_m^{q^n}.$$

Setting, $\beta_0 := 1$, this procedure yields an infinite sequence $(\beta_i)_{i\geq 0}$ corresponding to the logarithm.

Next, since $e_\phi \circ \log_\phi(z) = z$ by definition, we may now compute the coefficients of e_ϕ recursively:

$$\alpha_i = -\sum_{n=0}^{i-1} \alpha_i \beta_{i-n}^{q^n}.$$

The resulting sequence $(\alpha_i)_{i\geq 0}$ corresponds to the exponential of ϕ.

```
[1]: q = 4
     Fq = GF(q)
     A = Fq['T']
     K.<T> = Frac(A)
     phi = DrinfeldModule(A, [T, T+1, T^2 - T + 1])
```

We use the method `exponential` to compute a power series approximation of e_ϕ:

```
[2]: exp = phi.exponential()
     exp
```

```
[2]: z + ((1/(T^3+T^2+T))*z^4) + O(z^8)
```

To compute the logarithm, use the method `logarithm`:

```
[3]: log = phi.exponential()
     log
```

```
[3]: z + ((1/(T^3+T^2+T))*z^4) + O(z^8)
```

The returned power series is a *lazy power series*. This means that the user does not need to input any precision parameter and the code computes the coefficients on demands.

```
[4]: exp[q**5]
```

```
[4]: (T^2045 + T^2044 + ... + T^2 + 1)/(T^4827 + T^4825 + ... + T^342 + T^341)
```

```
[5]: log[q**5]
```

```
[5]: (T^2045 + T^2044 + ... + T^2 + 1)/(T^4827 + T^4825 + ... + T^342 + T^341)
```

The logarithm and exponential are the mutual compositional inverse to one another:

```
[6]: log.compose(exp)
```

```
[6]: z + O(z^8)
```

```
[7]: exp.compose(log)
```

```
[7]: z + O(z^8)
```

In order to obtain more coefficients, one can slice the series:

```
[8]: exp[0:q^2 + 1]   # coefficients from 0 to q^2
```

```
[8]: [0,
      1,
      0,
      0,
      1/(T^3 + T^2 + T),
      0,
      0,
      0,
      0,
      0,
      0,
      0,
      0,
      0,
      0,
      0,
      (T^14 + T^13 + T^12 + T^10 + T^9 + T^8 + T^6 + T^5 + T^4 + T + 1)/(T^28 +␣
      ↪T^24
     + T^20 + T^13 + T^9 + T^5)]
```

5.3 The case of the Carlitz module

Closed formulas for the coefficients of the exponential and logarithm are known when $\phi = \rho$, the Carlitz module:

$$\rho : T \mapsto T + \tau.$$

More precisely, for any $i > 0$ we define the following quantities:

- $[i] := T^{q^i} - T$,
- $D_0 := 1$ and $D_i := [i][i-1]^q \cdots [1]^{q^{i-1}}$,
- $L_0 := 1$ and $L_i := [i][i-1] \cdots [1]$.

Then, the exponential is given explicitely by:

$$e_\rho(z) = \sum_{i \geq 0} \frac{z^{q^i}}{D_i}$$

and the logarithm is given by

$$\log_\rho(z) = \sum_{i \geq 0} (-1)^i \frac{z^{q^i}}{L_i}.$$

```
[9]: rho = DrinfeldModule(A, [T, 1])  # Carlitz module
     exp = rho.exponential()
     log = rho.logarithm()
```

```
[10]: exp
```

[10]: z + ((1/(T^4+T))*z^4) + O(z^8)

[11]:
```
log
```

[11]: z + ((1/(T^4+T))*z^4) + O(z^8)

[12]:
```
def sq(i):
    """
    Return [i] = T^(q^i) - T.
    """
    return T^(q^i) - T

def D(i):
    """
    Return 1 if i = 0 and D_i = [i][i - 1]^q *... * [1]^(q^(i - 1)) otherwise.
    """
    if i == 0:
        return 1
    return prod(sq(n)**(q^(i - n)) for n in range(1, i + 1))

def L(i):
    """
    Return 1 if i = 0 and L_i = [i][i - 1] *...* [1] otherwise.
    """
    if i == 0:
        return 1
    return prod(sq(n) for n in range(1, i + 1))
```

We can check the expected values:

[13]:
```
assert exp[q]  == 1/D(1)
assert exp[q**2] == 1/D(2)
assert exp[q**3] == 1/D(3)
assert exp[q**4] == 1/D(4)
```

[14]:
```
assert log[q]  == -1/L(1)
assert log[q**2] == 1/L(2)
assert log[q**3] == -1/L(3)
assert log[q**4] == 1/L(4)
```