diffstrata – A SAGE PACKAGE FOR CALCULATIONS IN THE TAUTOLOGICAL RING OF THE MODULI SPACE OF ABELIAN DIFFERENTIALS

MATTEO COSTANTINI, MARTIN MÖLLER, AND JONATHAN ZACHHUBER

ABSTRACT. The boundary of the multi-scale differential compactification of strata of abelian differentials admits an explicit combinatorial description. However, even for low-dimensional strata, the complexity of the boundary requires use of a computer. We give a description of the algorithms implemented in the SageMath package diffstrata to enumerate the boundary components and perform intersection theory on this space. In particular, the package can compute the Euler characteristic of strata using the methods developed by the same authors in [CMZ20].

CONTENTS

1. INTRODUCTION

An important tool in studying the moduli space of (meromorphic) abelian differentials $\mathbb{P}\Omega\mathcal{M}_g(\mu)$ is its modular compactification, the space of multi-scale differentials $\Xi\overline{\mathcal{M}}_{g,n}(\mu)$ for μ an integer partition of $2g - 2$. The boundary is of a combinatorial nature, parametrised, for any μ, by finitely many labeled *level graphs* [BCGGM3]. However, already listing isomorphism classes of these graphs is a non-trivial task, and already for $g = 3$ the number of components becomes so large that even listing them is unfeasible by hand.

Moreover, in [CMZ20] the tautological ring of $\Xi\overline{\mathcal{M}}_{g,n}(\mu)$ is described and calculations therein may be expressed purely in terms of the combinatorics of the boundary.

Again, even the simplest calculations are extremely cumbersome to perform without the assistance of a computer.

The diffstrata package provides a framework for calculations in the tautological ring of $\Xi\overline{\mathcal{M}}_{g,n}(\mu)$. It is implemented in sage [SageMath] and is inspired by the package admcycles [DSZ20] for calculations in the tautological ring of $\overline{\mathcal{M}}_{g,n}$. However, due to the differences in the structure of the boundary, the implementation and interface of the two packages have only little in common. A key step in the evaluation process is performed by admcycles, though.

diffstrata may be used naively for basic inquiries about strata. The following example contains the complete code required to calculate the (orbifold) Euler characteristic of $\mathbb{P}\Omega\mathcal{M}_2(2)$ (using [CMZ20]):

```
sage: from admcycles.diffstrata import *
sage: X=Stratum((2,))
sage: X.euler_characteristic()
-1/40
```

We can also easily display information on the compactification of a stratum:

```
sage: from admcycles.diffstrata import *
sage: X=Stratum((2,2))
sage: X.info()
Stratum: (2, 2)
with residue conditions: []

Genus: [3]
Dimension: 6
Boundary Graphs (without horizontal edges):
Codimension 0: 1 graph
Codimension 1: 20 graphs
Codimension 2: 86 graphs
Codimension 3: 147 graphs
Codimension 4: 110 graphs
Codimension 5: 30 graphs
Total graphs: 394
```

Moreover, we can compute the Masur–Veech volume as an intersection number of ξ-powers and ψ-classes, cf. [CMSZ19]:

```
sage: from admcycles.diffstrata import *
sage: X=Stratum((1,1))
sage: (X.xi^2 * X.psi(1) * X.psi(2)).evaluate()
-1/720
sage: (X.xi^3 * X.psi(1)).evaluate()
-1/360
```

However, more extensive calculations necessarily require a deeper understanding of the syntax and the underlying objects of the package.

The most fundamental notion is that of an *(enhanced) profile*, encoding isomorphism classes of level graphs using tuples of integers (essentially writing them as products of divisors). It is explained in detail in Section 5 together with the algorithms used to list all boundary components. This notion is used by diffstrata,

together with an encoding of ψ-polynomials, to encode all additive generators of the tautological ring and all calculations performed involve (formal) sums of these.

Implementing the recursive structure of the boundary, even when considering holomorphic strata, disconnected meromorphic strata with residue conditions will appear as levels of level graphs. To encode these, we use *Generalised Strata*, as introduced in [CMZ20]. These may be thought of as container objects, where the graphs are stored and the calculations performed.

However, implementing any formula in the tautological ring will require some understanding of the subtleties surrounding enhanced profiles and the degeneration of level graphs, as well as extracting levels from graphs and working with these. In Section 10.1 we discuss, as an example, the implementation of the formula for the Euler characteristic using level-wise evaluations of top powers of the tautological class ξ, cf. [CMZ20].

Algorithmic aspects. Implementing the compactification $\Xi\overline{\mathcal{M}}_{g,n}(\mu)$ poses a series of challenges not encountered in the boundary of $\overline{\mathcal{M}}_{g,n}$. First, constructing all level graphs satisfying precisely the conditions of [BCGGM3, Def. 1.1] is a non-trivial problem. While generating a codimension-one degeneration of a stable graph inside $\overline{\mathcal{M}}_{g,n}$ is straight-forward (add an edge either as a loop or by splitting a vertex subject to the stability condition), for a *level graph* this is the more subtle problem of *splitting a level*.

To solve it, we must construct all divisors in any *generalised* stratum, i.e. meromorphic, disconnected and with residue conditions, as these appear as levels already inside low-genus holomorphic strata. Since all the calculations of the Chern classes of the logarithmic cotangent bundle and in particular of the Euler characteristic ([CMZ20]) and also the computation of Masur-Veech volumes ([CMSZ19]) happen in the tautological ring defined by clutching of *non-horizontal divisors*, the diffstrata package treats exclusively graphs without horizontal edges. See [CMZ20, Section 8] for other candidates of tautological rings that are potentially larger but might actually agree with the tautological ring used here.

We usually refer to two-level graphs for brevity as BICs (*bicoloured graphs*, as in [FP18]).

The combinatorics arising from the distribution of point orders, level structures, genera and edges leads to the simplest approach, namely considering all enhanced level structures on stable graphs in $\overline{\mathcal{M}}_{g,n}$ (e.g. using admcycles), being too slow even for low-dimensional holomorphic strata. Instead, we describe an algorithm for directly generating all BICs in a generalised stratum in detail in Section 5.2.

Moreover, this requires checking the Global Residue Condition (GRC), discovered in [BCGGM1], for generalised strata, i.e. implementing the $\mathfrak{R}$-GRC [CMZ20, §4]. For this, we refine the combinatorial criterion of [MUW17] to work for the $\mathfrak{R}$-GRC in Section 3.

Having generated all BICs, this allows us to construct any graph by recursively clutching BICs (one for each level-crossing). As a by-product, this yields discrete coordinates for the enhanced level graphs, as any graph splits uniquely into a product of distinct BICs: we therefore number the BICs of a stratum and associate to each level graph its *profile*, the tuple of indices of BICs appearing as levels of this graph. Unfortunately, this is not always injective: profiles may be reducible and we need an *enhanced profile* to refer to a graph uniquely, see Section 5.4 for details.

However, when multiplying two tautological classes, we require an implementation of the excess intersection formula for $\Xi\overline{\mathcal{M}}_{g,n}(\mu)$ [CMZ20, §8] and this requires a good understanding of the degeneration graph of the boundary components. Here again the notion of profile is key: it allows us to efficiently determine which graphs appear as degenerations and is essential in the computation of normal bundles and general intersections, see Section 7.

Finally, even in low-genus holomorphic strata, calculations involving top-classes of the tautological ring are only feasible because we use extensive caching. Again, our ability to replace graphs by enhanced profiles is essential, but already the higher-dimensional strata in genus three require more techniques to become manageable, see Section 9. Also, several computed values are written to files so that they can be easily reused between `sage` sessions and may be easily precomputed on another machine and imported into the current session, see Section 9.2.

Open questions. For the moduli space of (pointed) curves $\overline{\mathcal{M}}_{g,n}$ enumerating the boundary strata, also known as tropical curves, and providing a tight estimate for their growth rate has been discussed at various places ([Cha12], [MP11]), but we are aware of a complete solution only in genus zero ([McM14]). It seems interesting to address the analogous problem in $\Xi\overline{\mathcal{M}}_{g,n}(\mu)$, to count the number of boundary strata of at least provide tight estimates for their number. Some tables are given in Section 5 for the number of graphs without horizontal nodes. Such estimates would also be the basis for serious runtime analysis of the algorithms in `diffstrata`.[1]

Related to this problem is the question of determining the number of components of enhanced profiles for a given profile. There are obvious coarse upper bounds for this number, but in practice the number is relatively small in average (for fixed μ and codimension). Can this be proven? On the algorithmic side, it would be useful to design an algorithm that implements directly the degeneration of enhanced profiles. This could avoid the generation of the graphs altogether an lead to significant speed-up.

The alternating sum of the number of enhanced profiles appear to be zero for all holomorphic strata (we thank A. Neitzke for observing this during a talk!). Can this be proven? Moreover, the same appears to hold for the number of profiles, although the degeneration process of this is a main source of the complexity of the algorithm, see Section 5.4.

Installation. The package `diffstrata` is included with `admcycles` version 1.1 or greater. See [DSZ20] for a detailed guide to installing it.

From now on, all examples will assume that the line

```
sage: from admcycles.diffstrata import *
```

has been executed!

Structure. We begin by giving an overview of the package interface for the "casual user" in Section 2, before explaining the implementation in more detail. In Section 3 we start by reviewing some of the mathematical background. In Section 4, we explain the fundamental objects of `diffstrata` and how they relate to their mathematical counterparts. The algorithms to construct all non-horizontal level

---

[1]In practice, one runs out of memory before speed becomes a serious issue: Calculations in genus four take several hours but need several TB of RAM.

graphs as well as determining the degeneration graph of a stratum and computing isomorphisms of graphs are described in Section 5. Section 6 explains how `diffstrata` encodes and evaluates (using `admcycles`) tautological classes on strata and in Section 7 the implementation of the excess intersection formula is discussed. In Section 8 the subtleties around splitting graphs around a level and clutching are discussed and the recursive evaluation of ξ on levels is explained. Finally, in Section 9 we explain how and which computations are cached by `diffstrata` and how to import pre-computed values and end in Section 10 by illustrating how `diffstrata` calculates Euler characteristics and giving a few examples of cross-checks and tests.

List of Algorithms

2. Basic Interface

Before discussing the implementation details of `diffstrata`, we briefly revisit the examples of the introduction. The first step is always generating a stratum:

```
sage: X=Stratum((2,))
sage: print(X)
Stratum: (2,)
with residue conditions: []
```

Here we have defined a `Stratum` object. The argument is a Python `tuple` and may contain integers that sum to $2g - 2$ to define any meromorphic stratum (note the trailing `,` if there is only one entry!). The `print` statement displays information about the object and hints that this is in fact an instance of a `GeneralisedStratum` that can be disconnected and have residue conditions at the poles, see Section 4 for a more detailed discussion.

Creating a `Stratum` automatically performs a series of calculations. For example, all non-horizontal divisors (BICs) are generated and can now be accessed through `X`:

```
sage: X.bics
[EmbeddedLevelGraph(LG=LevelGraph([1, 1],[[1], [2, 3]],[(1, 3)],{1: 0, 2: 2,
     3: -2},[0, -1],True),dmp={2: (0, 0)},dlevels={0: 0, -1: -1}),
 EmbeddedLevelGraph(LG=LevelGraph([1, 0],[[1, 2], [3, 4, 5]],[(1, 4), (2, 5)
    ],{1: 0, 2: 0, 3: 2, 4: -2, 5: -2},[0, -1],True),dmp={3: (0, 0)},
    dlevels={0: 0, -1: -1})]
```

This illustrates how `diffstrata` represents level graphs internally. The multitude of decorations makes the classes `EmbeddedLevelGraph` and `LevelGraph` a bit unwieldy and there should be little reason to enter them by hand. But they do store the essential information, they are a backbone of `diffstrata`, and they appear frequently in the output. Details are described in Section 4.

For a single `EmbeddedLevelGraph`, e.g. an element of `X.bics`, we may use its `explain` method to produce a human-readable description of the graph:

```
sage: X.bics[0].explain()
LevelGraph embedded into stratum Stratum: (2,)
with residue conditions: []
 with:
On level 0:
* A vertex (number 0) of genus 1
On level 1:
* A vertex (number 1) of genus 1
The marked points are on level 1.
More precisely, we have:
* Marked point (0, 0) of order 2 on vertex 1 on level 1
Finally, we have one edge. More precisely:
* one edge between vertex 0 (on level 0) and vertex 1 (on level 1) with
     prong 1.
```

Instead of entering graphs by hand, we should always use (enhanced) profiles to refer to them inside `X`. For BICs, this is simply their index in `X.bics`. We can list all profiles of a given length:

```
sage: X.enhanced_profiles_of_length(2)
((((0, 1), 0),)
sage: X.enhanced_profiles_of_length(3)
()
```

Note that there are no profiles of length 3 even though

```
sage: X.dim()
3
```

The reason is that `diffstrata` ignores all graphs with horizontal edges in the boundary. We can also retrieve the `EmbeddedLevelGraph` from an (enhanced) profile:

```
sage: X.lookup_graph((0,1))
EmbeddedLevelGraph(LG=LevelGraph([1, 0, 0],[[1], [2, 3, 4], [5, 6, 7]],[(1,
    4), (2, 6), (3, 7)],{1: 0, 2: 0, 3: 0, 4: -2, 5: 2, 6: -2, 7: -2},[0,
    -1, -2],True),dmp={5: (0, 0)},dlevels={0: 0, -1: -1, -2: -2})
```

See Section 5 for details on profiles and graph generation.

The examples of the introduction also illustrated working in the tautological ring of X. We may inspect the individual classes. Using `print` gives a more readable output:

```
sage: print(X.psi(1))
Tautological class on Stratum: (2,)
with residue conditions: []

1 * Psi class 1 with exponent 1 on level 0 * Graph ((), 0) +

sage: X.psi(1)
ELGTautClass(X=GeneralisedStratum(sig_list=[Signature((2,))],res_cond=[]),
    psi_list=[(1, AdditiveGenerator(X=GeneralisedStratum(sig_list=[
    Signature((2,))],res_cond=[]),enh_profile=((), 0),leg_dict={1: 1}))])
```

This illustrates how `diffstrata` encodes elements of the tautological ring: a tautological class is represented by an `ELGTautClass`, which is in turn a sum of `AdditiveGenerators`. Each `AdditiveGenerator` corresponds to a ψ-monomial on a graph and thus carries the information of an enhanced profile and a `leg_dict` encoding the ψ-powers: every ψ-class is associated to a marked point of a level [CMZ20, Thm. 1.5], i.e. a leg of the graph. A ψ-monomial is thus encoded by a Python `dict` with entries of the form `l : n` where `l` is the number of a leg of the graph and `n` is the exponent of the ψ-class associated to this leg. For example, we saw above that `X.bics[0]` is the compact-type graph in the boundary of $\Omega\mathcal{M}_2(2)$. We see from the `LevelGraph` that the marked point is at leg `2` (cf. Section 4), the ψ-class at this point is therefore represented by the `leg_dict {2 : 1}`. We can enter this into `diffstrata` as follows:

```
sage: A = X.additive_generator(((0,), 0), {2 : 1})
sage: print(A)
Psi class 2 with exponent 1 on level 1 * Graph ((0,), 0)
```

Note that we had to use the *enhanced* profile `((0,), 0)` to refer to the graph. Details and more examples may be found in Section 6.

Tautological classes may be added and multiplied. We can check that the class A we defined agrees with the product of the ψ-class on the stratum with the class of the graph:

```
sage: A == X.psi(1) * X.additive_generator(((0,), 0))
True
```

Moreover, when squaring, e.g., the class of a graph, a normal bundle contribution appears:

```
sage: print(X.additive_generator(((0,), 0))^2)
```

```
Tautological class on Stratum: (2,)
with residue conditions: []

-1 * Psi class 1 with exponent 1 on level 0 * Graph ((0,), 0) +
-1 * Psi class 3 with exponent 1 on level 1 * Graph ((0,), 0) +
```

The multiplication process is described in detail in Section 7.

In the formulas for the Euler characteristic [CMZ20], the class $\xi = c_1(\mathcal{O}(-1))$ of the tautological bundle and its restriction $\xi_{B_\Gamma^{[i]}}$ to a level i of a graph Γ were key. For a stratum, the class ξ is easily accessible:

```
sage: print(X.xi)
Tautological class on Stratum: (2,)
with residue conditions: []

3 * Psi class 1 with exponent 1 on level 0 * Graph ((), 0) +
-1 * Graph ((0,), 0) +
-1 * Graph ((1,), 0) +
```

Moreover, it is not difficult to compute $\xi_{B_\Gamma^{[i]}}$ (here for the top-level of the compact-type graph):

```
sage: print(X.xi_at_level(0, ((0,),0)))
Tautological class on Stratum: (2,)
with residue conditions: []

1 * Psi class 1 with exponent 1 on level 0 * Graph ((0,), 0) +
```

More details and examples may be found in Example 7.2 and Remark 8.3.

We now describe these objects and the implementation in more detail.

3. GENERALISED STRATA

We begin by briefly recalling the notions from [BCGGM3] and [CMZ20] in the generality that we here require.

3.1. **Strata with residue conditions.** To obtain a recursive structure on the boundary of $\Xi\overline{\mathcal{M}}_{g,n}(\mu)$, recall the definition of *generalised stratum*, introduced in [CMZ20, §4] to cover the case of a level of an enhanced level graph. More precisely, we allow differentials on disconnected surfaces: denote by $\mu_i = (m_{i,1}, \ldots, m_{i,n_i}) \in \mathbb{Z}^{n_i}$ the type of a differential, i.e., we require that $\sum_{j=1}^{n_i} m_{i,j} = 2g_i - 2$ for some $g_i \in \mathbb{Z}$ and $i = 1, \ldots, k$. Then we define, for a tuple $\mathbf{g} = (g_1, \ldots, g_k)$ of genera and a tuple $\mathbf{n} = (n_1, \ldots, n_k)$ together with $\boldsymbol{\mu} = (\mu_1, \ldots, \mu_k)$, the disconnected stratum

$$\Omega\mathcal{M}_{\mathbf{g},\mathbf{n}}(\boldsymbol{\mu}) \,=\, \prod_{i=1}^{k} \Omega\mathcal{M}_{g_i,n_i}(\mu_i)\,.$$

Note that the projectivized stratum $\mathbb{P}\Omega\mathcal{M}_{\mathbf{g},\mathbf{n}}(\boldsymbol{\mu})$ is the quotient by the diagonal action of $\mathbb{C}^*$, not the quotient by the action of $(\mathbb{C}^*)^k$.

Moreover, we consider subspaces of these *cut out by residue conditions*. More precisely, denote by $H_p \subseteq \cup_{i=1}^{k}\{(i,1), \cdots (i,n_i)\}$ the subset of the marked points such that $m_{i,j} < -1$. Now consider vector spaces $\mathfrak{R}$ of the following special shape, modelled on the global residue condition from [BCGGM1]: for λ a partition of H_p,

with parts denoted by $\lambda^{(k)}$, and a subset $\lambda_{\mathfrak{R}}$ of the parts of λ, we define the $\mathbb{C}$-vector space

$$\mathfrak{R} := \left\{ r = (r_{i,j})_{(i,j) \in H_p} \in \mathbb{C}^{H_p} \quad \text{and} \quad \sum_{(i,j) \in \lambda^{(k)}} r_{i,j} = 0 \quad \text{for all} \quad \lambda^{(k)} \in \lambda_{\mathfrak{R}} \right\}.$$

We denote the subspace of surfaces with residues in $\mathfrak{R}$ by $\Omega\mathcal{M}_{\mathbf{g},\mathbf{n}}^{\mathfrak{R}}(\boldsymbol{\mu})$.

In [CMZ20, Prop. 4.2] a modular compactification $\mathbb{P}\Xi\overline{\mathcal{M}}_{\mathbf{g},\mathbf{n}}^{\mathfrak{R}}(\boldsymbol{\mu})$ of $\mathbb{P}\Omega\mathcal{M}_{\mathbf{g},\mathbf{n}}^{\mathfrak{R}}(\boldsymbol{\mu})$ is constructed in analogy to [BCGGM3]. Consequently, the boundary components are parametrised by *enhanced level graphs*. More precisely, a *level graph* is defined to be a stable graph together with a level function. Recall that a *stable graph* is a tuple $\Gamma = (V_i, H_i, E_i, g_i, v_i, \iota_i)_{i=1,\ldots,k}$ consisting of *vertices* V_i, a *genus map* $g_i \colon V_i \to \mathbb{Z}_{\geq 0}$, *legs* H_i that are associated to the vertices by a *vertex map* $v_i \colon H_i \to V_i$ and come with an involution $\iota_i \colon H_i \to H_i$, the two-cycles of which form the *edges* $E_i \subseteq H_i \times H_i$ while the fixed points (denoted H_i^m) are in bijection with the n_i marked points. The k graphs $\Gamma_i = (V_i, H_i, E_i, g_i, v_i, \iota_i)$ are required to be connected and satisfy the usual stability conditions. Moreover, we set $g = \sqcup g_i$, $v = \sqcup v_i$, $E = \sqcup E_i$, $H = \sqcup H_i$, and $V = \sqcup V_i$. Note that this data induces a unique bijection $o \colon \bigsqcup H_i^m \to \boldsymbol{\mu}$ associating to each marked point (i,j) the order $m_{i,j}$ of the differential.

A *level function* on the vertices is a map $\ell \colon V \to \mathbb{Z}$, which we normalise to take values in $\{0, -1, \ldots, -L\}$ and require that, for all edges $e \in E$, $\ell(v(e^+)) \geq \ell(v(e^-))$, where we write $e =: (e^+, e^-) \in H \times H$.

Moreover, an *enhancement* is a map $\kappa \colon E \to \mathbb{Z}_{\geq 0}$ such that $\kappa(e) = 0$ if and only if e is horizontal (i.e. $\ell(v(e^+)) = \ell(v(e^-))$), subject to the following stability condition: define the *degree* of a vertex v in V to be

$$\deg(v) = \sum_{h \in H^m, v(h)=v} o(h) + \sum_{e \in E, v(e^+)=v} (\kappa(e) - 1) - \sum_{e \in E, v(e^-)=v} (\kappa(e) + 1).$$

Then the enhancement is admissible, if $\deg(v) = 2g(v) - 2$ holds for every vertex v of Γ.

The *enhanced level graph* then consists of the triple (Γ, ℓ, κ). We denote the corresponding boundary component of $\mathbb{P}\Xi\overline{\mathcal{M}}_{\mathbf{g},\mathbf{n}}^{\mathfrak{R}}(\boldsymbol{\mu})$ by D_Γ.

Remark 3.1. Note that $\kappa(e)$ corresponds to the number of *prongs* at e and, for a non-horizontal edge, the associated differential has a zero of order $\kappa(e) - 1$ on the top component and a pole of order $-\kappa(e) - 1$ on the bottom component (for horizontal edges there is a simple pole on each component). This gives an extension of o to H.

Using this identification, the stability condition of [BCGGM3, §2] is simply the requirement that the orders of zeros and poles sum to $2g_{i,j} - 2$ on each vertex. See [BCGGM3, §2] and [CMZ20, §3.2] for details.

To determine, for a generalised stratum $\mathbb{P}\Xi\overline{\mathcal{M}}_{\mathbf{g},\mathbf{n}}^{\mathfrak{R}}(\boldsymbol{\mu})$, which enhanced level graphs give non-empty boundary components, we recall the $\mathfrak{R}$-GRC from [CMZ20, §4]: starting with an enhanced level graph Γ, we construct a new *auxiliary level graph* $\widetilde{\Gamma}$ by adding, for each $\lambda^{(k)} \in \lambda_{\mathfrak{R}}$, a new vertex $v_{\lambda^{(k)}}$ to Γ at level ∞ and converting a tuple $(i,j) \in \lambda^{(k)}$ into an edge from the marked point (i,j) to the vertex $v_{\lambda^{(k)}}$. We then say that Γ satisfies the $\mathfrak{R}$-*global residue condition ($\mathfrak{R}$-GRC)* if the tuple of residues at the legs in H_p belongs to $\mathfrak{R}$ and for every level $L < \infty$ of $\widetilde{\Gamma}$ and every connected component Y of the subgraph $\widetilde{\Gamma}_{>L}$ one of the following conditions holds.

(1) The component Y contains a marked point with a prescribed pole that is *not* in $\lambda_{\mathfrak{R}}$.

(2) The component Y contains a marked point with a prescribed pole $(i,j) \in H_p$ and there is an $r \in \mathfrak{R}$ with $r_{(i,j)} \neq 0$.

(3) Let $e_1, \ldots, e_b$ denote the set of edges where Y intersects $\widetilde{\Gamma}_{=L}$. Then

$$\sum_{j=1}^{b} \mathrm{Res}_{e_j^-} \, \eta_{v(e_j^-)} \; = \; 0 \,,$$

where $v(e_j^-) \in \widetilde{\Gamma}_{=L}$.

By [CMZ20, Prop. 4.2], the boundary components D_Γ of $\mathbb{P}\Xi\overline{\mathcal{M}}_{\mathbf{g,n}}^{\mathfrak{R}}(\boldsymbol{\mu})$ are parametrised by enhanced level graphs (Γ, ℓ, κ) satisfying the $\mathfrak{R}$-GRC.

For applications such as listing all graphs in the boundary of a stratum, it is convenient to have a purely graph-theoretic criterion in analogy to the one shown for the classical GRC in [MUW17]. The following proposition strips the tropical language away in the criterion [MUW17, Theorem 1] and generalises it to meromorphic strata with residue conditions.

Let (Γ, ℓ, κ) be an enhanced level graph.

We call a vertex v of Γ *inconvenient* if $g(v) = 0$, if it is not adjacent to any edge e with enhancement $\kappa(e) = 0$ and if it is adjacent to a leg with a *very high enhancement* in the following precise sense: denote by $\mathfrak{p}(v)$ the set of half-edges on the vertex v that are poles (in the sense of Remark 3.1). Then the condition is that there is a $p \in \mathfrak{p}(v)$ such that

$$o(p) > \sum_{p' \in \mathfrak{p}(v)} (o(p') - 1) - 1.$$

Proposition 3.2. The boundary stratum D_Γ of $\Omega\mathcal{M}_{\mathbf{g,n}}^{\mathfrak{R}}(\boldsymbol{\mu})$ associated with the enhanced level graph (Γ, ℓ, κ) is non-empty if and only if both of the following conditions are satisfied

(1) For every inconvenient vertex v of Γ there is
   (a) a simple cycle based at v that does not pass through any vertex of level smaller than v, or
   (b) the graph of levels $\geq \ell(v)$ deprived of the vertex v has two components, each of which has a marked pole in H_p whose residue is not constrained to zero for all elements of $\mathfrak{R}$.

(2) For every horizontal edge e of Γ there is
   (a) a simple cycle based through e that does not pass through any vertex of level smaller than $\ell(e)$, or
   (b) the graph of levels $\geq \ell(e)$ deprived of the edge e has two components, each of which has a marked pole in H_p whose residue is not constrained to zero for all elements of $\mathfrak{R}$.

Proof. Consider first the case that $\widetilde{\Gamma}$ is connected. We view $\widetilde{\Gamma}$ as an enhanced level graph of an auxiliary stratum $\widetilde{X}$ as follows: we add prongs to the new edges of $\widetilde{\Gamma}$ in accordance with the pole orders of the half-edges on Γ. For each vertex v at level ∞, we then extend the genus function by g_v setting $2g_v - 2$ as the sum of orders of the half-edges on v induced by the newly added prongs, possibly adding an extra simple zero to fix parity issues. Note that all new vertices are of positive genus, so

stability is not an issue. Therefore, for the stratum $\widetilde{X}$, we are now reduced to the situation of [MUW17, Theorem 1].

In a product of strata, clearly a graph is admissible if and only if each component is admissible. $\square$

See Section 4.2 for examples illustrating this criterion.

This criterion allows us to explicitly construct all graphs in a given stratum. In fact, we can construct all graphs with no horizontal edges recursively from the two-level graphs.

3.2. **Constructing Level Graphs.** Recall the undegeneration maps δ_i [CMZ20, §3.3], contracting all level crossing of an enhanced level graph Γ without horizontal edges except for the i-th level crossing, yielding a two-level graph. The component of Γ is contained in the product of the components of $\delta_i(\Gamma)$, which may be irreducible.

Definition 3.3. Let (Γ, ℓ, κ) be an enhanced level graph with L levels and without horizontal edges. We define the *profile* of Γ to be the tuple $(\delta_1(\Gamma), \ldots, \delta_L(\Gamma))$. An *enhanced profile* is a profile together with a choice of irreducible component.

Note that the association of a profile to a graph is not injective, see Example 5.6. It is primarily useful to encode efficiently how graphs degenerate. By definition, the information of an enhanced profile of Γ is equivalent to the data of Γ.

Generating all non-horizontal graphs inside a stratum is thus equivalent to listing all non-empty enhanced profiles. We do this recursively. For X a generalised stratum, denote by $\mathrm{BIC}(X)$ the (non-horizontal) two-level graphs in the boundary of X.

Definition 3.4. Let X be a generalised stratum. For each $\Gamma \in \mathrm{BIC}(X)$ we define:
   (1) the generalised strata $\Gamma^\top$ and $\Gamma^\perp$, the top and bottom levels of Γ;
   (2) a map $\beta_\Gamma^\top \colon \mathrm{BIC}(\Gamma^\top) \to \mathrm{BIC}(X)$ that associates to a graph $\Gamma' \in \mathrm{BIC}(\Gamma^\top)$ the graph $\delta_0(\Lambda) \in \mathrm{BIC}(X)$ where Λ is the graph obtained by clutching Γ' to $\Gamma^\perp$;
   (3) a map $\beta_\Gamma^\perp \colon \mathrm{BIC}(\Gamma^\perp) \to \mathrm{BIC}(X)$ that associates to a graph $\Gamma' \in \mathrm{BIC}(\Gamma^\perp)$ the graph $\delta_1(\Lambda) \in \mathrm{BIC}(X)$ where Λ is the graph obtained by clutching $\Gamma^\top$ to Γ'.

The following proposition is an immediate consequence of [CMZ20, Prop. 5.1].

Proposition 3.5. Let X be a generalised stratum and $\Gamma \in \mathrm{BIC}(X)$.
   (1) The images of $\beta_\Gamma^\top$ and $\beta_\Gamma^\perp$ are disjoint.
   (2) The profile (Γ', Γ) is non-empty if and only if Γ' is in the image of $\beta_\Gamma^\top$.
   (3) The profile (Γ, Γ') is non-empty if and only if Γ' is in the image of $\beta_\Gamma^\perp$.

As a consequence, we may define a partial order $\prec$ on $\mathrm{BIC}(X)$ by defining $\Gamma \prec \Gamma'$ if and only if (Γ', Γ) is non-empty.

Remark 3.6. The maps β are not necessarily injective. Indeed, whenever a profile (Γ_1, Γ_2) is reducible, i.e. contains at least two distinct enhanced level graphs $\Lambda_1 \neq \Lambda_2$, cutting the bottom level off these three-level graphs gives two distinct BICs $\tilde{\Lambda}_1, \tilde{\Lambda}_2 \in \mathrm{BIC}(\Gamma_2^\top)$ (the "cut" edges correspond to edges of Γ_2 and become marked points in $\Gamma_2^\top$ and $\Gamma_2^\perp$) with $\beta_{\Gamma_2}^\top(\tilde{\Lambda}_1) = \beta_{\Gamma_2}^\top(\tilde{\Lambda}_2) = \Gamma_1$. See also Example 5.22.

However, non-injectivity does not imply reducibility. Degenerating a level in different ways may give *isomorphic* graphs, see Example 8.1 and Figure 9.

This allows us to recursively compute all profiles.

Proposition 3.7. Let X be a generalised stratum.

(1) The enhanced level graphs in $\mathrm{BIC}(X)$ can be listed explicitly.
(2) All non-empty profiles in X can be constructed recursively from $\mathrm{BIC}(X)$.
(3) All graphs inside a profile can be constructed explicitly from the profile's components.

Proof. This is essentially the content of Section 5: the effective construction of BICs is explained in detail in Section 5.2, in particular Algorithm 5.7.

The non-empty profiles are constructed recursively: a non-empty profile $(\Gamma_1, \ldots, \Gamma_l)$ of length l may be extended to a non-empty profile $(\Gamma_0, \Gamma_1, \ldots, \Gamma_l)$ if and only if $\Gamma_1 \prec \Gamma_0$. Indeed, $\Gamma_1^\top$ is also the top level of any graph Λ in $(\Gamma_1, \ldots, \Gamma_l)$ and thus a preimage $(\beta_{\Gamma_1}^\top)^{-1}(\Gamma_0)$ exists in $\mathrm{BIC}(\Gamma_1^\top)$ and can be clutched to Λ to yield a graph in $(\Gamma_0, \Gamma_1, \ldots, \Gamma_l)$. Similarly, the profile may be extended to $(\Gamma_1, \ldots, \Gamma_l, \Gamma_{l+1})$ if and only if $\Gamma_{l+1} \prec \Gamma_l$.

The converse direction is simply squishing of a level.

To get all the graphs we follow the above procedure, noting that we might obtain several graphs in the same profile if β is non-injective (we clutch each preimage with each other graph). $\qquad\square$

The key observation is that BICs and three-level graphs are sufficient for constructing the entire stratum. In particular, all levels are seen by these.

Remark 3.8. Let X be a generalised stratum. Note that any level L appearing in any graph in X is one of the following three types:

(1) a top level of a BIC,
(2) a bottom level of a BIC, or
(3) a middle level of a three-level graph.

Indeed, given a graph Γ, level l of Γ remains unchanged by contracting any level crossing not adjacent to l. Contracting all non-adjacent levels results either in a BIC (if l is top or bottom level) or in a three-level graph around level l.

While the reducibility is recorded by the three-level graphs, determining the componentes of a profile is not straight-forward. Indeed, given two non-empty profiles (Γ_1, Γ_2) and (Γ_2, Γ_3) the profile $(\Gamma_1, \Gamma_2, \Gamma_3)$ is non-empty, but in the reducible case it is not clear how the reducibility and the individual graphs are related.

Remark 3.9. Determining the reducibility of a profile is a delicate issue and is discussed in detail in Section 5.4. In particular, each of the following may occur:

(1) a degeneration of an irreducible profile can be reducible (Example 5.6);
(2) a degeneration of a reducible profile can be irreducible (Example 5.22);
(3) A reducible profile implies a non-injectivity of one of the maps β.

The converse of the last statement is false in general, as the non-injectivity may stem from the presence of automorphisms.

4. Level Graphs and Embeddings

The `diffstrata` package uses three basic classes to model the objects appearing in the boundary of $\mathbb{P}\Xi\overline{\mathcal{M}}_{g,n}(\mu)$.

- **GeneralisedStratum** models the strata $\mathbb{P}\Xi\overline{\mathcal{M}}_{g,n}(\mu)$. All essential operations will be performed by an object of this type.
- **LevelGraph** is a low-level object that represents the actual underlying graph. While it is important for the underlying calculations, there should be little or no reason to construct an explicit **LevelGraph** directly. It was originally modelled on the class **stgraph** of **admcycles**.
- **EmbeddedLevelGraph** is essentially a wrapper to encode how a **LevelGraph** is embedded into a **GeneralisedStratum**. A **LevelGraph** will usually appear as an **EmbeddedLevelGraph** and should be accessed through its *profile*, see Section 5.

Note that every level of an **EmbeddedLevelGraph** is itself a **GeneralisedStratum** of lower dimension, hence the recursive structure.

A **GeneralisedStratum** is given by the following data:

- a list of **Signature** objects, giving the signatures of the (possibly) meromorphic strata appearing as factors, and
- optionally a list of residue conditions, signalling which residues add up to zero.

Consequently, the marked points of a stratum may be uniquely referred to by their coordinates inside the signature tuple.

Note that there is also the **Stratum** class, which is simply a frontend for the class **GeneralisedStratum** with a simpler syntax: It can only be used to create connected strata with no residue conditions:

```
sage: X=Stratum((2,))
sage: print(X)
Stratum: (2,)
with residue conditions: []
sage: isinstance(X, GeneralisedStratum)
True
```

Note that **tuple**s with only one entry must be terminated by a **,**!

Remark 4.1. A **GeneralisedStratum** must be provided with a list of **Signature** objects, a **tuple** will raise an error! A **Signature** object is initialised by a signature **tuple** and makes all its intrinsic properties easily accessible. It is easiest understood via the package documentation:

```
sage: Signature?
Init signature: Signature(sig)
Docstring:
   A signature of a stratum.

   Attributes:
       sig (tuple): signature tuple
       g (int): genus
       n (int): total number of points
       p (int): number of poles
       z (int): number of zeroes
       poles (tuple): tuple of pole orders
       zeroes (tuple): tuple of zero orders
       pole_ind (tuple): tuple of indices of poles
       zero_ind (tuple): tuple of indices of zeroes
```

```
    EXAMPLES

    sage: from admcycles.diffstrata.sig import Signature
    sage: sig=Signature((2,1,-1,0))
    sage: sig.g
    2
    sage: sig.n
    4
    sage: sig.poles
    (-1,)
    sage: sig.zeroes
    (2, 1)
    sage: sig.pole_ind
    (2,)
    sage: sig.zero_ind
    (0, 1)
    sage: sig.p
    1
    sage: sig.z
    2
Init docstring:
    Initialise signature

    Args:
        sig (tuple): signature tuple of integers adding up to 2g-2
```

4.1. **Points on graphs and strata.** Analogous to Section 3, the `diffstrata`
package uses zeros and poles of a differential in two different contexts: the *marked
points* of the stratum, as elements of $\boldsymbol{\mu}$, have associated *half-edges* of the graph.
To illustrate these, we need to first briefly explain how a `LevelGraph` encodes the
information of an enhanced level graph.

A point or *leg* of a `LevelGraph` is given by a positive integer. Each vertex of a
`LevelGraph` has a (possibly empty) list of legs associated to it and each edge is a
`tuple` of two legs: the command

```
sage: L=LevelGraph([1, 0],[[1, 2], [3, 4, 5]],[(1, 4), (2, 5)],{1: 0, 2: 0,
    3: 2, 4: -2, 5: -2},[0, -1])
```

encodes a graph L with two vertices, one of genus 1 (with internal name, the position
in the list, 0) and one of genus 0 (with internal name 1). The (nested) list of legs
is in the same order as the list of vertices: the vertex 0 has two legs, stored in the
list [1, 2] and the vertex 1 has three legs, stored in [3, 4, 5].

Furthermore, the graph has two edges, the first, (1, 4), connecting leg 1 on
vertex 0 with leg 4 on vertex 1, and the second, (2, 5), connecting leg 2 on
vertex 0 to leg 5 on vertex 1.

There are two more pieces of information needed to determine a `LevelGraph`: a
`dict`, associating to every leg the order of the differential at this point (e.g. leg 4
is a pole of order -2 and leg 3 is a zero of order 2; leg 1 is simply a marked point
(of order 0)) and a `list` of levels, [0, -1], indicating that vertex 0 is on level 0
and vertex 1 is on level -1.

FIGURE 1. The "banana" graph in the boundary of $\Omega\mathcal{M}_2(2)$.

This uniquely determines an enhanced level graph in the sense of Section 3. We see that L describes the "banana" graph in the boundary of $\Omega\mathcal{M}_2(2)$ depicted in Figure 1.

We can access all of this information about the `LevelGraph` from within `sage`:

```
sage: L=LevelGraph([1, 0],[[1, 2], [3, 4, 5]],[(1, 4), (2, 5)],{1: 0, 2: 0,
    3: 2, 4: -2, 5: -2},[0, -1])
sage: L.genus(0)
1
sage: L.legsatvertex(1)
[3, 4, 5]
sage: L.edges
[(1, 4), (2, 5)]
sage: L.vertex(1)  # vertex of leg
0
sage: L.levelofvertex(1)
-1
sage: L.orderatleg(3)
2
```

The orders are equivalent to the number of prong-matchings at the edges. The prongs are stored in a dictionary and their lcm can be calculated easily:

```
sage: L=LevelGraph([0, 0, 0],[[1, 2, 3], [4, 5, 6], [7, 8, 9]],[(3, 6), (4,
    8), (5, 9)],{1: 1, 2: -4, 3: 1, 4: 1, 5: 0, 6: -3, 7: 3, 8: -3, 9:
    -2},[0, -1, -2])
sage: L.prongs.items()
dict_items([((3, 6), 2), ((4, 8), 2), ((5, 9), 1)])
sage: L.prongs[(3,6)]
2
sage: lcm(L.prongs.values())
2
sage:
```

Remark 4.2. Note that, as mentioned above, `LevelGraph`s should not be entered "by hand", but instead `EmbeddedLevelGraph`s and profiles should be used.

A marked point as an element of $\boldsymbol{\mu}$ is considered by `diffstrata` as a point of the `GeneralisedStratum`. An `EmbeddedLevelGraph` is, essentially, a `LevelGraph` together with the information, which of its legs correspond to marked points of the stratum.
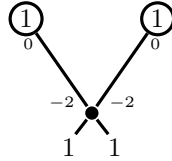
FIGURE 2. The "V"-shaped graph in the boundary of the stratum $\Omega\mathcal{M}_2(1,1)$.

This is encoded by a `dict` usually denoted `dmp` (dictionary of marked points) that identifies legs of the `LevelGraph` (integers) with points of the stratum (`tuple`s: index of component, index of point in signature, as in Section 3).

For technical reasons, the embedding also requires a dictionary of levels, `dlevels`.

In the above example, giving such an embedding is straight-forward:

```
sage: L=LevelGraph([1, 0],[[1, 2], [3, 4, 5]],[(1, 4), (2, 5)],{1: 0, 2: 0,
    3: 2, 4: -2, 5: -2},[0, -1])
sage: X=Stratum((2,))
sage: ELG=EmbeddedLevelGraph(X, L, dmp={3: (0,0)}, dlevels={0: 0, -1: -1})
```

Of course, our graph is isomorphic to one of the two BICs generated automatically by the stratum X:

```
sage: any(ELG.is_isomorphic(B) for B in X.bics)
True
sage: len([B for B in X.bics if ELG.is_isomorphic(B)])
1
```

Our `GeneralisedStratum` objects may have several connected components, as well as residue conditions, since we want to consider arbitrary levels of `LevelGraph`s as strata,

Consider the "V"-shaped graph in the boundary of the stratum $\Omega\mathcal{M}_2(1,1)$ depicted in Figure 2. In `sage`, we may extract the two levels to see examples of more complicated strata:

```
sage: X=Stratum((1,1))
sage: V=LevelGraph([1, 1, 0],[[1], [2], [3, 4, 5, 6]],[(1, 5), (2, 6)],{1:
    0, 2: 0, 3: 1, 4: 1, 5: -2, 6: -2},[0, 0, -1])
sage: ELV=EmbeddedLevelGraph(X, V, dmp={3: (0, 0), 4: (0, 1)}, dlevels={0:
    0, -1: -1})
sage: ELV.level(0)
LevelStratum(sig_list=[Signature((0,)), Signature((0,))],res_cond=[],
    leg_dict={1: (0, 0), 2: (1, 0)})
sage: ELV.level(1)
LevelStratum(sig_list=[Signature((1, 1, -2, -2))],res_cond=[[(0, 2)], [(0,
    3)]],leg_dict={3: (0, 0), 4: (0, 1), 5: (0, 2), 6: (0, 3)})
```

Using `print` gives more readable output:

```
sage: print(ELV.level(0))
Product of Strata:
Signature((0,))
Signature((0,))
```

```
with residue conditions:
dimension: 3
leg dictionary: {1: (0, 0), 2: (1, 0)}
leg orbits: [[(1, 0), (0, 0)]]

sage: print(ELV.level(1))
Stratum: Signature((1, 1, -2, -2))
with residue conditions: [(0, 2)] [(0, 3)]
dimension: 0
leg dictionary: {3: (0, 0), 4: (0, 1), 5: (0, 2), 6: (0, 3)}
leg orbits: [[(0, 0)], [(0, 1)], [(0, 3), (0, 2)]]
```

Observe that the extracted levels also remember the action of the automorphism group of the graph they were extracted from on their marked points.

The residue conditions are given by a nested list of points of the stratum. Marked poles whose residue adds up to 0 are contained in the same list.

To illustrate this, let us compare the lower level of the Banana graph of above:

```
sage: B=LevelGraph([1, 0],[[1, 2], [3, 4, 5]],[(1, 4), (2, 5)],{1: 0, 2: 0,
    3: 2, 4: -2, 5: -2},[0, -1])
sage: Y=Stratum((2,))
sage: ELB=EmbeddedLevelGraph(Y, B, dmp={3: (0,0)}, dlevels={0: 0, -1: -1})
sage: ELB.level(1)
LevelStratum(sig_list=[Signature((2, -2, -2))],res_cond=[[(0, 1), (0, 2)]],
    leg_dict={3: (0, 0), 4: (0, 1), 5: (0, 2)})
sage: print(ELB.level(1))
Stratum: Signature((2, -2, -2))
with residue conditions: [(0, 1), (0, 2)]
dimension: 0
leg dictionary: {3: (0, 0), 4: (0, 1), 5: (0, 2)}
leg orbits: [[(0, 0)], [(0, 1), (0, 2)]]
```

In this case, the stratum records that the poles share a residue condition.

Remark 4.3. Note that while, mathematically, levels are usually indexed with negative numbers starting at 0, in `diffstrata` it is often much less confusing to work with positive level numbers. To ease this translation, `LevelGraph`s come with a notion of "internal" versus "relative" level number:

```
sage: B.internal_level_number(1)
-1
sage: B.level_number(-1)
1
```

In particular, the internal level numbers might not even be consecutive, while the `level_number`s are guaranteed to run from 0 to the number of levels.

`EmbeddedLevelGraph`s also come with an `explain` method, intended to describe them in a more human-readable format. For the above examples:

```
sage: ELV.explain()
LevelGraph embedded into stratum Stratum: (1, 1)
with residue conditions: []
 with:
On level 0:
```
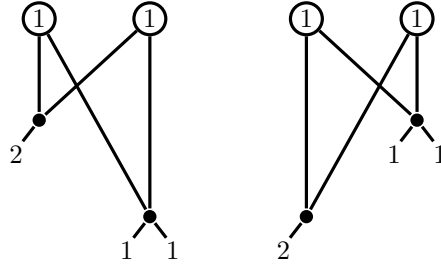
FIGURE 3. An illegal (left) and legal (right) graph in the boundary of $\Omega\mathcal{M}_3(2, 1, 1)$.

```
* A vertex (number 0) of genus 1
* A vertex (number 1) of genus 1
On level 1:
* A vertex (number 2) of genus 0
The marked points are on level 1.
More precisely, we have:
* Marked point (0, 0) of order 1 on vertex 2 on level 1
* Marked point (0, 1) of order 1 on vertex 2 on level 1
Finally, we have 2 edges. More precisely:
* one edge between vertex 0 (on level 0) and vertex 2 (on level 1) with
    prong 1.
* one edge between vertex 1 (on level 0) and vertex 2 (on level 1) with
    prong 1.
sage: ELB.explain()
LevelGraph embedded into stratum Stratum: (2,)
with residue conditions: []
 with:
On level 0:
* A vertex (number 0) of genus 1
On level 1:
* A vertex (number 1) of genus 0
The marked points are on level 1.
More precisely, we have:
* Marked point (0, 0) of order 2 on vertex 1 on level 1
Finally, we have 2 edges. More precisely:
* 2 edges between vertex 0 (on level 0) and vertex 1 (on level 1) with
    prongs 1 and 1.
```

4.2. **Checking the $\mathfrak{R}$-GRC: points "at level ∞".** Checking the $\mathfrak{R}$-GRC is split into two steps. First, for *any* LevelGraph, we may check the classical GRC:

```
sage: L=LevelGraph([1, 0],[[1, 2], [3, 4, 5]],[(1, 4), (2, 5)],{1: 0, 2: 0,
    3: 2, 4: -2, 5: -2},[0, -1])
sage: L.is_legal()
True
```

Example 4.4. Consider the graphs in the stratum $(2, 1, 1)$ that are depicted in Figure 3. Note that the left graph is illegal, as the bottom-left vertex is inconve-
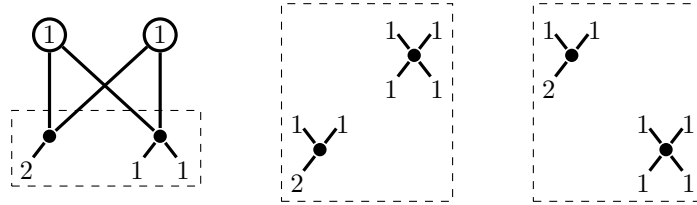
FIGURE 4. The "zigzag-graph" (left) in the boundary of the stratum $\Omega\mathcal{M}_3(2,1,1)$ and an legal (center) and illegal (right) degeneration of its bottom level.

nient as defined in Section 3.1: It is a stratum with signature $(2,-2,-2)$ and both residues are forced zero, as there is no cycle or pole in the graph above to rectify this. This problem does not occur in the right graph.

In diffstrata, we observe:

```
sage: L=LevelGraph([1, 1, 0, 0],[[1, 2], [3, 4], [5, 6, 7], [8, 9, 10,
    11]],[(1, 6), (3, 7), (4, 10), (2, 11)],{1: 0, 2: 0, 3: 0, 4: 0, 5: 2,
    6: -2, 7: -2, 8: 1, 9: 1, 10: -2, 11: -2},[0, 0, -1, -2])
sage: L.is_legal()
Vertex 2 is illegal!
False
sage: R=LevelGraph([1, 1, 0, 0],[[1, 2], [3, 4], [5, 6, 7], [8, 9, 10,
    11]],[(1, 6), (3, 7), (4, 10), (2, 11)],{1: 0, 2: 0, 3: 0, 4: 0, 5: 2,
    6: -2, 7: -2, 8: 1, 9: 1, 10: -2, 11: -2},[0, 0, -2, -1])
sage: R.is_legal()
True
```

Similarly, if we start with an EmbeddedLevelGraph, we may check the $\mathfrak{R}$-GRC.

Example 4.5. Consider the example from above as a degeneration of the bottom level of the "zigzag-graph", the BIC that is the common δ_1 of the two graphs of Example 4.4, as depicted in Figure 4. Note that there are residue conditions intertwining the simple zeros on the two components. These imply that the middle graph is legal, while the right one is not.

To check this in diffstrata, we begin by generating the stratum and entering the graph:

```
sage: X=Stratum((2,1,1))
sage: LG=LevelGraph([1, 1, 0, 0],[[1, 2], [3, 4], [5, 6, 7], [8, 9, 10,
    11]],[(1, 6), (3, 7), (4, 10), (2, 11)],{1: 0, 2: 0, 3: 0, 4: 0, 5: 2,
    6: -2, 7: -2, 8: 1, 9: 1, 10: -2, 11: -2},[0, 0, -1, -1])
sage: ELG=EmbeddedLevelGraph(X,LG,dmp={5: (0, 0), 8: (0, 1), 9: (0, 2)},
    dlevels={0: 0, -1: -1})
```

We may convince ourselves that the graph is as depicted in Figure 4:

```
sage: ELG.explain()
LevelGraph embedded into stratum Stratum: (2, 1, 1)
with residue conditions: []
 with:
On level 0:
```

```
* A vertex (number 0) of genus 1
* A vertex (number 1) of genus 1
On level 1:
* A vertex (number 2) of genus 0
* A vertex (number 3) of genus 0
The marked points are on level 1.
More precisely, we have:
* Marked point (0, 0) of order 2 on vertex 2 on level 1
* Marked point (0, 1) of order 1 on vertex 3 on level 1
* Marked point (0, 2) of order 1 on vertex 3 on level 1
Finally, we have 4 edges. More precisely:
* one edge between vertex 0 (on level 0) and vertex 2 (on level 1) with
    prong 1.
* one edge between vertex 1 (on level 0) and vertex 2 (on level 1) with
    prong 1.
* one edge between vertex 1 (on level 0) and vertex 3 (on level 1) with
    prong 1.
* one edge between vertex 0 (on level 0) and vertex 3 (on level 1) with
    prong 1.
```

We may extract the bottom level of this EmbeddedLevelGraph:

```
sage: L=ELG.level(1); L  # extract bottom level
LevelStratum(sig_list=[Signature((2, -2, -2)), Signature((1, 1, -2, -2))],
    res_cond=[[(0, 1), (1, 3)], [(0, 2), (1, 2)]],leg_dict={5: (0, 0), 6:
    (0, 1), 7: (0, 2), 8: (1, 0), 9: (1, 1), 10: (1, 2), 11: (1, 3)})
```

Embedding (into L!) the two degenerations depicted in Figure 4, we confirm that one is legal, while the other is not:

```
sage: M=EmbeddedLevelGraph(L, LevelGraph([0, 0],[[1, 2, 3], [4, 5, 6,
    7]],[],{1: 2, 2: -2, 3: -2, 4: 1, 5: 1, 6: -2, 7: -2},[-1, 0]),dmp={1:
    (0, 0), 2: (0, 1), 3: (0, 2), 4: (1, 0), 5: (1, 1), 6: (1, 2), 7: (1, 3)
    },dlevels={-1: -1, 0: 0})
sage: M.is_legal()
True
sage: R=EmbeddedLevelGraph(L, LevelGraph([0, 0],[[1, 2, 3], [4, 5, 6,
    7]],[],{1: 2, 2: -2, 3: -2, 4: 1, 5: 1, 6: -2, 7: -2},[0, -1]),dmp={1:
    (0, 0), 2: (0, 1), 3: (0, 2), 4: (1, 0), 5: (1, 1), 6: (1, 2), 7: (1, 3)
    },dlevels={-1: -1, 0: 0})
sage: R.is_legal()
False
```

4.3. **The Underlying Graph.** Each EmbeddedLevelGraph has an associated "underlying graph", which is a Sage Graph. It's main use is checking the graph-theoretic conditions for the $\mathfrak{R}$-GRC corresponding, essentially, to the auxiliary graph $\widetilde{\Gamma}$ constructed in the proof of Proposition 3.2. To encode all necessary data, it has a slightly more involved format, which we briefly explain.

The underlying graph is, formally, associated to the underlying LevelGraph. The vertices are formally tuples encoding the vertex number and its genus:

```
sage: LG=LevelGraph([1, 1, 0, 0],[[1, 2], [3, 4], [5, 6, 7], [8, 9, 10,
    11]],[(1, 6), (3, 7), (4, 10), (2, 11)],{1: 0, 2: 0, 3: 0, 4: 0, 5: 2,
    6: -2, 7: -2, 8: 1, 9: 1, 10: -2, 11: -2},[0, 0, -1, -1])
```

```
sage: LG.genus(0)
1
sage: LG.UG_vertex(0)
(0, 1, 'LG')
sage: LG.underlying_graph
Looped multi-graph on 4 vertices
sage: LG.underlying_graph.vertices()
[(0, 1, 'LG'), (1, 1, 'LG'), (2, 0, 'LG'), (3, 0, 'LG')]
sage: LG.underlying_graph.edges()
[((0, 1, 'LG'), (2, 0, 'LG'), (1, 6)), ((0, 1, 'LG'), (3, 0, 'LG'), (2, 11))
    , ((1, 1, 'LG'), (2, 0, 'LG'), (3, 7)), ((1, 1, 'LG'), (3, 0, 'LG'), (4,
    10))]
```

The entry `'LG'` indicates that this vertex comes from a vertex of the `LevelGraph`.

In the case of an `EmbeddedLevelGraph`, the underlying `LevelGraph` may be endowed with extra "points at ∞" to encode residue conditions. To illustrate this, we consider again the situation of Example 4.5.

```
sage: ELG=EmbeddedLevelGraph(X, LG, dmp={5: (0, 0), 8: (0, 1), 9: (0, 2)},
    dlevels={0: 0, -1: -1})
sage: X=Stratum((2,1,1))
sage: LG=LevelGraph([1, 1, 0, 0],[[1, 2], [3, 4], [5, 6, 7], [8, 9, 10,
    11]],[(1, 6), (3, 7), (4, 10), (2, 11)],{1: 0, 2: 0, 3: 0, 4: 0, 5: 2,
    6: -2, 7: -2, 8: 1, 9: 1, 10: -2, 11: -2},[0, 0, -1, -1])
sage: ELG=EmbeddedLevelGraph(X, LG, dmp={5: (0, 0), 8: (0, 1), 9: (0, 2)},
    dlevels={0: 0, -1: -1})
sage: L=ELG.level(1); print(L)
Product of Strata:
Signature((2, -2, -2))
Signature((1, 1, -2, -2))
with residue conditions: [(0, 1), (1, 3)] [(0, 2), (1, 2)]
dimension: 1
leg dictionary: {5: (0, 0), 6: (0, 1), 7: (0, 2), 8: (1, 0), 9: (1, 1), 10:
    (1, 2), 11: (1, 3)}
leg orbits: [[(0, 0)], [(0, 1), (0, 2)], [(1, 0)], [(1, 1)], [(1, 2), (1, 3)
    ]]

sage: L.bics[0]  # numbering of list items might differ
EmbeddedLevelGraph(LG=LevelGraph([0, 0, 0],[[1, 2, 3], [4, 5, 6], [7, 8,
    9]],[(6, 9)],{1: 2, 2: -2, 3: -2, 4: -2, 5: -2, 6: 2, 7: 1, 8: 1, 9:
    -4},[0, 0, -1],True),dmp={1: (0, 0), 2: (0, 1), 3: (0, 2), 4: (1, 2), 5:
     (1, 3), 7: (1, 0), 8: (1, 1)},dlevels={0: 0, -1: -1})
sage: L.bics[0].LG.underlying_graph.vertices()
[(0, 0, 'LG'), (0, 0, 'res'), (1, 0, 'LG'), (1, 0, 'res'), (2, 0, 'LG')]
sage: L.bics[0].LG.underlying_graph.edges()
[((0, 0, 'LG'), (0, 0, 'res'), 'res0edge2'), ((0, 0, 'LG'), (1, 0, 'res'), '
    res1edge3'), ((0, 0, 'res'), (1, 0, 'LG'), 'res0edge5'), ((1, 0, 'LG'),
     (1, 0, 'res'), 'res1edge4'), ((1, 0, 'LG'), (2, 0, 'LG'), (6, 9))]
```

Each residue condition corresponds to one vertex of the underlying graph, labelled with `'res'`. This vertex is connected to all poles sharing this residue condition.

Note that the residue conditions of a stratum are stored most transparently in the "smooth" (0-level) graph inside the stratum. In the above situation:

```
sage: L.smooth_LG
EmbeddedLevelGraph(LG=LevelGraph([0, 0],[[1, 2, 3], [4, 5, 6, 7]],[],{1: 2,
    2: -2, 3: -2, 4: 1, 5: 1, 6: -2, 7: -2},[0, 0],True),dmp={1: (0, 0), 2:
     (0, 1), 3: (0, 2), 4: (1, 0), 5: (1, 1), 6: (1, 2), 7: (1, 3)},dlevels
    ={0: 0})
sage: L.smooth_LG.LG.underlying_graph.vertices()
[(0, 0, 'LG'), (0, 0, 'res'), (1, 0, 'LG'), (1, 0, 'res')]
sage: L.smooth_LG.LG.underlying_graph.edges()
[((0, 0, 'LG'), (0, 0, 'res'), 'res0edge2'), ((0, 0, 'LG'), (1, 0, 'res'), '
    res1edge3'), ((0, 0, 'res'), (1, 0, 'LG'), 'res0edge7'), ((1, 0, 'LG'),
     (1, 0, 'res'), 'res1edge6')]
```

4.4. **Checking the $\mathfrak{R}$-GRC.** To check the Global Residue Condition, we use Proposition 3.2. This is checked on the level of vertices (and horizontal edges, although we do not care for them here).

More precisely, LevelGraph uses the following check for legality:

Algorithm 4.6 (Vertex Legality (GRC)). The method is_illegal_vertex checks the legality of a vertex v in the following steps:

Step 1: We first check if v is *inconvenient* in the sense of Section 3, that is $g_v = 0$, there are no simple poles on v and there exists a pole order that is greater than the difference of the sum of the pole orders and the number of poles, i.e., denoting by m_i the pole orders and by p the number of poles, there exists an m_i such that $m_i > \sum m_j - p - 1$.

If v is not inconvenient, v is legal.

Step 2: We next check for loops above v: If there are less than two edges going up from v, there can be no loop and v is illegal. Otherwise we build the subgraph of the underlying graph that consists of vertices above the level of v and generate a list of its connected components (this is easy, as this are sage Graphs).

Step 3: For each of these connected components, we check if there are two edges connecting v to this connected component. (there is a technical subtlety here: the subgraph described above does not contain v and hence non of the edges we are interested in. We also can't restrict to the edges of the LevelGraph as will become clear when describing the $\mathfrak{R}$-GRC below. Therefore, we actually work with two subgraphs, the abovegraph consisting of all edges with level $\geq$ the level of v and the subgraph yielding the connected components, which consists of the vertices above the level of v in the connected component of abovegraph containing v.) If this is the case, v is legal.

Step 4: In the case of a meromorphic stratum, v is legal if there are at least two "free" poles on the connected component of v. We thus count all free poles at v and in the connected components above.

Step 5: If all of this fails, v is illegal.

In the case that there are residue conditions, we have to slightly modify Step 3 above. More precisely, also EmbeddedLevelGraph includes an is_legal method that checks the $\mathfrak{R}$-GRC in a stratum with residue conditions.

Moreover, is_legal also checks if any of the levels of the graph are empty (i.e. calls the appropriate is_empty methods, described below).

Then a list of *all* poles (as points on the graph) contained in *any* residue condition passed to the enveloping stratum is compiled. In particular, *poles are not included in this list, if they are only constrained by the Residue Theorem on some component.* This list is used in Step 4 of the above algorithm: a pole is free if it is not contained in this list, i.e. not involved in any residue condition passed to the stratum.

Note that, as we are working with the underlying graph, the vertices "at ∞" are considered for all connectivity issues (once these have been set up correctly by the EmbeddedLevelGraph).

Remark 4.7. While most of diffstrata works only for graphs with no horizontal edges (in particular the notion of profile does not immediately extend) LevelGraphs may have horizontal edges and is_legal can be used to check legality of edges, cf. Proposition 3.2.

4.5. **Residue Matrices and Dimension.** To see how the residue conditions interact, it is often helpful to consider the matrix given by their linear equations. Moreover, for correctly degenerating residue conditions, it is important to distinguish residue conditions imposed "from the stratum" from those imposed on a particular graph by the Residue Theorem on each vertex.

Given a list of residue conditions and a GeneralisedStratum, one can transform these into a matrix using the matrix_from_res_conditions method:

```
sage: X=GeneralisedStratum([Signature((2,-2,-2)),Signature((2,-2,-2))])
sage: X.matrix_from_res_conditions([[(0,1),(0,2),(1,2)],[(0,1),(1,1)],[(1,1)
    ,(1,2)]])
[1 1 0 1]
[1 0 1 0]
[0 0 1 1]
```

The residue_matrix of a stratum is this matrix for the residue conditions of the stratum.

For calculating the dimension of a stratum, we need to consider *all* imposed residue conditions, i.e. also the conditions imposed by the Residue Theorem on each component. This can be done for *any* EmbeddedLevelGraph by using the method residue_matrix_from_RT, the combined matrix (with all residue conditions) is available via full_residue_matrix.

Example 4.8. We illustrate the difference by considering the bottom level of the V-graph, the Banana graph and the zigzag graph introduced above.

Again, we input the graphs a LevelGraphs and embed them into their corresponding strata:

```
sage: V=LevelGraph([1, 1, 0],[[1], [2], [3, 4, 5, 6]],[(1, 5), (2, 6)],{1:
    0, 2: 0, 3: 1, 4: 1, 5: -2, 6: -2},[0, 0, -1])
sage: X=Stratum((1,1))
sage: ELV=EmbeddedLevelGraph(X, V, dmp={3: (0, 0), 4: (0, 1)}, dlevels={0:
    0, -1: -1})
sage: B=LevelGraph([1, 0],[[1, 2], [3, 4, 5]],[(1, 4), (2, 5)],{1: 0, 2: 0,
    3: 2, 4: -2, 5: -2},[0, -1])
sage: Y=Stratum((2,))
sage: ELB=EmbeddedLevelGraph(Y, B, dmp={3: (0,0)}, dlevels={0: 0, -1: -1})
sage: Z=Stratum((2,1,1))
```

```
sage: LG=LevelGraph([1, 1, 0, 0],[[1, 2], [3, 4], [5, 6, 7], [8, 9, 10,
    11]],[(1, 6), (3, 7), (4, 10), (2, 11)],{1: 0, 2: 0, 3: 0, 4: 0, 5: 2,
    6: -2, 7: -2, 8: 1, 9: 1, 10: -2, 11: -2},[0, 0, -1, -1])
sage: ELZ=EmbeddedLevelGraph(Z, LG, dmp={5: (0, 0), 8: (0, 1), 9: (0, 2)},
    dlevels={0: 0, -1: -1})
```

We extract the bottom levels of each graph:

```
sage: V_bottom=ELV.level(1)
sage: B_bottom=ELB.level(1)
sage: Z_bottom=ELZ.level(1)
```

Inspecting these, we see the different residue conditions imposed on the poles:

```
sage: print(V_bottom)
Stratum: Signature((1, 1, -2, -2))
with residue conditions: [(0, 2)] [(0, 3)]
dimension: 0
leg dictionary: {3: (0, 0), 4: (0, 1), 5: (0, 2), 6: (0, 3)}
leg orbits: [[(0, 0)], [(0, 1)], [(0, 3), (0, 2)]]

sage: print(B_bottom)
Stratum: Signature((2, -2, -2))
with residue conditions: [(0, 1), (0, 2)]
dimension: 0
leg dictionary: {3: (0, 0), 4: (0, 1), 5: (0, 2)}
leg orbits: [[(0, 0)], [(0, 1), (0, 2)]]

sage: print(Z_bottom)
Product of Strata:
Signature((2, -2, -2))
Signature((1, 1, -2, -2))
with residue conditions: [(0, 1), (1, 3)] [(0, 2), (1, 2)]
dimension: 1
leg dictionary: {5: (0, 0), 6: (0, 1), 7: (0, 2), 8: (1, 0), 9: (1, 1), 10:
    (1, 2), 11: (1, 3)}
leg orbits: [[(0, 0)], [(0, 1), (0, 2)], [(1, 0)], [(1, 1)], [(1, 2), (1, 3)
    ]]
```

We can now calculate the residue matrices of the strata:

```
sage: V_bottom.residue_matrix()
[1 0]
[0 1]
sage: B_bottom.residue_matrix()
[1 1]
sage: Z_bottom.residue_matrix()
[1 0 0 1]
[0 1 1 0]
```

and compare these to the `full_residue_matrix` of the `smooth_LG` inside each stratum.

```
sage: B_bottom.smooth_LG.residue_matrix_from_RT
[1 1]
sage: V_bottom.smooth_LG.residue_matrix_from_RT
```

```
[1 1]
sage: Z_bottom.smooth_LG.residue_matrix_from_RT
[1 1 0 0]
[0 0 1 1]
sage: B_bottom.smooth_LG.full_residue_matrix
[1 1]
[1 1]
sage: V_bottom.smooth_LG.full_residue_matrix
[1 0]
[0 1]
[1 1]
sage: Z_bottom.smooth_LG.full_residue_matrix
[1 0 0 1]
[0 1 1 0]
[1 1 0 0]
[0 0 1 1]
```

The rank of the `full_residue_matrix` is the rank of the residue space of the stratum. It is vital for calculating the dimension of a stratum. Recall [CMZ20, Remark 4.1] that

$$\dim X = \sum_{i=1}^{k}(2g_i + n_i - 1) - \operatorname{rk}(M) - 1,$$

where k is the number of connected components and M is the full residue matrix. We use this to calculate the correct dimensions:

```
sage: B_bottom.dim()
0
sage: V_bottom.dim()
0
sage: Z_bottom.dim()
1
```

Example 4.9. We can check that the top and bottom level of every BIC in a stratum do indeed add up to one less than the dimension:

```
sage: X4=GeneralisedStratum([Signature((4,))])
sage: all(B.level(0).dim() + B.level(1).dim() == X4.dim() - 1 for B in X4.
    bics)
True
```

Remark 4.10. We can also check if the residue at a pole is forced 0 by the residue conditions (i.e. if the rank of the `full_residue_matrix` changes by adding this condition). In the case that a stratum has simple poles, we use this to determine if the stratum is empty (if a simple pole is forced 0):

```
sage: I=Stratum((1,-1))
sage: I.smooth_LG.residue_zero((0,1))
True
sage: I.is_empty()
True
```

4.6. **Level Extraction.** We have already seen level extraction in action and describe the process now in some more detail.

Note that, strictly speaking, an extracted level is a `LevelStratum`, which inherits from `GeneralisedStratum`.

Revisiting the V-graph from above:

```
sage: ELV.level(1)
LevelStratum(sig_list=[Signature((1, 1, -2, -2))],res_cond=[[(0, 2)], [(0,
    3)]],leg_dict={3: (0, 0), 4: (0, 1), 5: (0, 2), 6: (0, 3)})
sage: type(ELV.level(1))
<class 'admcycles.diffstrata.levelstratum.LevelStratum'>
sage: isinstance(ELV.level(1), GeneralisedStratum)
True
```

The main extra piece of information it holds is `leg_dict`, a dictionary mapping the legs of the `LevelGraph` we extracted the level of (in this case V) to the marked points of the `LevelStratum`. Also, the orbits of the automorphism group are recorded:

```
sage: ELV
EmbeddedLevelGraph(LG=LevelGraph([1, 1, 0],[[1], [2], [3, 4, 5, 6]],[(1, 5),
     (2, 6)],{1: 0, 2: 0, 3: 1, 4: 1, 5: -2, 6: -2},[0, 0, -1],True),dmp={3:
     (0, 0), 4: (0, 1)},dlevels={0: 0, -1: -1})
sage: print(ELV.level(1))
Stratum: Signature((1, 1, -2, -2))
with residue conditions: [(0, 2)] [(0, 3)]
dimension: 0
leg dictionary: {3: (0, 0), 4: (0, 1), 5: (0, 2), 6: (0, 3)}
leg orbits: [[(0, 0)], [(0, 1)], [(0, 3), (0, 2)]]
```

The main part of the extraction happens again on the level of the `LevelGraph`, more precisely in the method `stratum_from_level`.

Algorithm 4.11 (Level Extraction)**.** The method `stratum_from_level` is implemented as follows:

> **Step 1:** Extraction of the relevant data from the graph: vertices at the current level as well as the legs and their orders. Here, we must be careful with the level numbering, cf. Remark 4.3. As we sort the marked points into their new signatures, we create `leg_dict`.
> **Step 2:** Adding the residue conditions is implemented analogous to the $\mathfrak{R}$-GRC check: the poles on the level are sorted by the connected component of the underlying graph they are attached to. Recall that the underlying graph includes the "level at ∞". Again, components containing free poles must be ignored. Moreover, the poles need to translated into their respective strata points for storing the residue conditions.

When a level is extracted from an `EmbeddedLevelGraph` (as is usually the case), first the free poles determined by the residue conditions of the "big" graph, as described above and then `stratum_from_level` is called on the underlying `LevelGraph` with this information.

The resulting `LevelStratum` is then equipped with the automorphism orbits of the legs and cached.

Remark 4.12. In the case of a BIC, i.e. an `EmbeddedLevelGraph` with exactly two levels, the levels are also stored in the `top` and `bot` attributes:

```
sage: ELV.is_bic()
True
sage: ELV.top == ELV.level(0)
True
sage: ELV.bot == ELV.level(1)
True
```

5. Generating all Level Graphs

In this section, we give a detailed description of the explicit construction of Proposition 3.7.

Generating all level graphs (as `EmbeddedLevelGraph`s) inside a generalised stratum is done recursively: first, we generate all two-level graphs (BICs) and then we clutch them together in all possible ways. This gives the degeneration graph as a by-product.

We can inspect this data for any stratum:

```
sage: X=Stratum((4,))
sage: X.info()
Stratum: (4,)
with residue conditions: []

Genus: [3]
Dimension: 5
Boundary Graphs (without horizontal edges):
Codimension 0: 1 graph
Codimension 1: 8 graphs
Codimension 2: 19 graphs
Codimension 3: 16 graphs
Codimension 4: 4 graphs
Total graphs: 48
```

Note that the number of graphs increases quickly:

```
sage: X=Stratum((1,1,1,1))
sage: X.info()
Stratum: (1, 1, 1, 1)
with residue conditions: []

Genus: [3]
Dimension: 8
Boundary Graphs (without horizontal edges):
Codimension 0: 1 graph
Codimension 1: 102 graphs
Codimension 2: 1100 graphs
Codimension 3: 4222 graphs
Codimension 4: 7531 graphs
Codimension 5: 6708 graphs
Codimension 6: 2856 graphs
Codimension 7: 456 graphs
Total graphs: 22976
```

Remark 5.1. For genus 4, strata having several million boundary components appear. Already the generation of the strata requires significant computational and memory power.

The key observation is that, by Proposition 3.5, each three-level graph arises by either clutching a top level of a BIC to a BIC of its bottom level or a BIC of the top level to the bottom level. On the other hand, each three-level graph is contained in the intersection of two (different) BICs of the Stratum.

More explicitly, any three-level graph is contained in a profile $(\beta_\Gamma^\top(\Gamma'), \Gamma)$ for some $\Gamma \in \mathrm{BIC}(X)$ and $\Gamma' \in \mathrm{BIC}(\Gamma^\top)$ (or equivalently $(\Lambda, \beta_\Lambda^\perp(\Lambda'))$ for some $\Lambda \in \mathrm{BIC}(X)$ and $\Lambda' \in \mathrm{BIC}(\Lambda^\perp)$).

We can thereby express the clutching of a product of BICs in the top and bottom components of a BIC in our stratum as a product of BICs of our stratum. Hence the procedure is recursive.

Therefore, to generate all graphs, we need to generate only the BICs together with, for each BIC, top and bottom components and the two maps.

More precisely, the degeneration graph of a `GeneralisedStratum` X is determined by the following information:

- The set $\mathrm{BIC}(X)$ corresponding to the `list X.bics`.
- For each $\Gamma \in \mathrm{BIC}(X)$, its top and bottom component $\Gamma^\top$ and $\Gamma^\perp$ (each a `GeneralisedStratum` together with a dictionary mapping Stratum points to `LevelGraph` points). If Γ corresponds to an `EmbeddedLevelGraph` B, $\Gamma^\top$ is given by `B.top` and $\Gamma^\perp$ is given by `B.bot`.
- For each $\Gamma \in \mathrm{BIC}(X)$, the sets $\mathrm{BIC}(\Gamma^\top)$ and $\mathrm{BIC}(\Gamma^\perp)$ together with the maps $\beta_\Gamma^\top$ and $\beta_\Gamma^\perp$. The maps β are given as dictionaries of indices of `list`s: if Γ corresponds to `X.bics[i]`, then
  - `top_to_bic(i)` is a `dict` mapping indices of `X.bics[i].top.bics` to indices of `X.bics` such that `X.DG.top_to_bic(i)[j]` corresponds to $\beta_\Gamma^\top(\Gamma')$ if Γ' corresponds to `X.bics[i].top.bics[j]` and
  - `bot_to_bic(i)` is a `dict` mapping indices of `X.bics[i].bot.bics` to indices of `X.bics` such that `X.DG.bot_to_bic(i)[j]` corresponds to $\beta_\Gamma^\perp(\Gamma')$ if Γ' corresponds to `X.bics[i].bot.bics[j]`.

See Example 5.8 for details.

Remark 5.2. For caching purposes, it is essential for `diffstrata` to refer to a BIC by its *index* in the `list X.bics`.

Correspondingly, a profile is encoded as a `tuple` of `int`s, an enhanced profile is encoded as a nested `tuple`, `(p, i)`, where `p` is a profile and `i` is an `int` determining the component. As described above, the maps β are also given by `dict`s mapping indices to indices.

Note, however, that starting with Sage 9, the numbering of profiles can (and will!) change between every `sage` session!

We can now calculate the degeneration graph.

Algorithm 5.3 (Degeneration Graph)**.**

  Step 1: Calculate all BICs in a `GeneralisedStratum`.
  Step 2: Separate these into top and bottom component.

Step 3: Calculate all BICs in every top and bottom component.

Step 4: Calculate `top_to_bic` and `bot_to_bic` for each BIC in the Stratum (as dictionaries mapping an index of `bics` of `top`/`bot` to an index of `bics` of the stratum).

In particular, this yields a recursive algorithm for the `EmbeddedLevelGraph` of an arbitrary product of BICs in the stratum as follows:

Algorithm 5.4 (Building a Graph from BICs)**.**

   INPUT: Product of BICs.

   OUTPUT: `EmbeddedLevelGraph`.

   Step 1: Choose a BIC B from the product (e.g. the first).

   Step 2: Find the preimages of the other BICs in the product under `top_to_bic` and `bot_to_bic` of B.

   This gives (possibly multiple) products of BICs in the top and bottom stratum of B.

   Step 3: Apply to product in top an bottom to get two `EmbeddedLevelGraphs`.

   Step 4: Return the clutching of the top and bottom graph.

Moreover, we can generate the "lookup list", consisting of the non-empty profiles in each stratum.

Before we discuss the details of the implementation, we give an overview of the relevant interface and some examples.

5.1. **Interface and Examples.** Let X be a `GeneralisedStratum`. To access an `EmbeddedLevelGraph` inside X, we need to provide an enhanced profile, that is

- a `tuple` of indices of `X.bics`, the profile, and
- an `int` as a choice of component, in case this profile is reducible.

The relevant methods are

- `X.lookup` with argument a profile `tuple` and returns a `list` of all the `EmbeddedLevelGraph`s matching this profile (even if the profile is irreducible, this returns a `list` of length 1!).
- `X.lookup_graph` with argument a profile `tuple` and optional argument a choice of connected component to be provided as an `int` index of `X.lookup`, which defaults to `0`.
- `X.lookup_list` is a nested `list`, where `X.lookup_list[codim]` is a `list` of profiles of length `codim` (i.e. the corresponding level graph is of codimension `codim`).
- `X.enhanced_profiles_of_length` with argument an `int` returns a list of all enhanced profiles of this length.

Example 5.5. Let us consider the stratum $\Omega\mathcal{M}_2(2)$. The degeneration graph consists of the three graphs depicted in Figure 5

In `diffstrata`, the situation is as follows. We first generate the stratum and may then inspect the BICs:

```
sage: X=Stratum((2,))
sage: X.bics  # order might differ
[EmbeddedLevelGraph(LG=LevelGraph([1, 0],[[1, 2], [3, 4, 5]],[(1, 4), (2, 5)
    ],{1: 0, 2: 0, 3: 2, 4: -2, 5: -2},[0, -1],True),dmp={3: (0, 0)},
    dlevels={0: 0, -1: -1}),
```

FIGURE 5. The degeneration graph of $\Omega\mathcal{M}_2(2)$.

```
EmbeddedLevelGraph(LG=LevelGraph([1, 1],[[1], [2, 3]],[(1, 3)],{1: 0, 2: 2,
    3: -2},[0, -1],True),dmp={2: (0, 0)},dlevels={0: 0, -1: -1})]
```

As these are `EmbeddedLevelGraph`s, we may use the `explain` method to relate them to Figure 5:

```
sage: X.bics[0].explain()
LevelGraph embedded into stratum Stratum: (2,)
with residue conditions: []
 with:
On level 0:
* A vertex (number 0) of genus 1
On level 1:
* A vertex (number 1) of genus 0
The marked points are on level 1.
More precisely, we have:
* Marked point (0, 0) of order 2 on vertex 1 on level 1
Finally, we have 2 edges. More precisely:
* 2 edges between vertex 0 (on level 0) and vertex 1 (on level 1) with
    prongs 1 and 1.
sage: X.bics[1].explain()
LevelGraph embedded into stratum Stratum: (2,)
with residue conditions: []
 with:
On level 0:
* A vertex (number 0) of genus 1
On level 1:
* A vertex (number 1) of genus 1
The marked points are on level 1.
More precisely, we have:
* Marked point (0, 0) of order 2 on vertex 1 on level 1
Finally, we have one edge. More precisely:
* one edge between vertex 0 (on level 0) and vertex 1 (on level 1) with
    prong 1.
```

The profiles are stored in the `lookup_list` of `X`:

```
sage: X.lookup_list
[[()], [(0,), (1,)], [(1, 0)]]
```

We can inspect the graph in a profile:

```
sage: X.lookup_graph((1,0))    # this might be (0,1) !!
EmbeddedLevelGraph(LG=LevelGraph([1, 0, 0],[[1], [2, 3, 4], [5, 6, 7]],[(1,
    4), (2, 6), (3, 7)],{1: 0, 2: 0, 3: 0, 4: -2, 5: 2, 6: -2, 7: -2},[0,
    -1, -2],True),dmp={5: (0, 0)},dlevels={0: 0, -1: -1, -2: -2})
sage: X.lookup((1,0))   # list
[EmbeddedLevelGraph(LG=LevelGraph([1, 0, 0],[[1], [2, 3, 4], [5, 6, 7]],[(1,
    4), (2, 6), (3, 7)],{1: 0, 2: 0, 3: 0, 4: -2, 5: 2, 6: -2, 7: -2},[0,
    -1, -2],True),dmp={5: (0, 0)},dlevels={0: 0, -1: -1, -2: -2})]
```

Again, the `explain` method relates this graph to Figure 5:

```
sage: X.lookup_graph((1,0)).explain()
LevelGraph embedded into stratum Stratum: (2,)
with residue conditions: []
 with:
On level 0:
* A vertex (number 0) of genus 1
On level 1:
* A vertex (number 1) of genus 0
On level 2:
* A vertex (number 2) of genus 0
The marked points are on level 2.
More precisely, we have:
* Marked point (0, 0) of order 2 on vertex 2 on level 2
Finally, we have 3 edges. More precisely:
* one edge between vertex 0 (on level 0) and vertex 1 (on level 1) with
    prong 1.
* 2 edges between vertex 1 (on level 1) and vertex 2 (on level 2) with
    prongs 1 and 1.
```

Note that we should always keep Remark 5.2 in mind when working with (enhanced) profiles! We convince ourselves, that the profile works as described, i.e. are compatible with the maps δ_i:

```
sage: G=X.lookup_graph((1,0))
sage: G.delta(1)
EmbeddedLevelGraph(LG=LevelGraph([1, 1],[[1], [4, 5]],[(1, 4)],{1: 0, 4: -2,
    5: 2},[0, -1],True),dmp={5: (0, 0)},dlevels={0: 0, -1: -1})
sage: G.delta(1).is_isomorphic(X.bics[1])
True
sage: G.delta(2)
EmbeddedLevelGraph(LG=LevelGraph([1, 0],[[2, 3], [5, 6, 7]],[(2, 6), (3, 7)
    ],{2: 0, 3: 0, 5: 2, 6: -2, 7: -2},[0, -2],True),dmp={5: (0, 0)},
    dlevels={0: 0, -2: -1})
sage: G.delta(2).is_isomorphic(X.bics[0])
True
```

We can also list the enhanced profiles and convince ourselves that all profiles are irreducible in this case.

FIGURE 6. An example of a reducible profile: the two non-isomorphic three-level graphs G and H have the same two-level graphs as undegenerations.

```
sage: X.enhanced_profiles_of_length(0)
(((), 0),)
sage: X.enhanced_profiles_of_length(1)
(((0,), 0), ((1,), 0))
sage: X.enhanced_profiles_of_length(2)
(((1, 0), 0),)
```

Note that the empty profile always corresponds to the 0-level graph, `smooth_LG`:

```
sage: X.lookup_graph(()) == X.smooth_LG
True
```

While the profile gives a good first approximation of an `EmbeddedLevelGraph`, there can be distinct level graphs sharing the same profile.

Example 5.6. Consider the following level graphs in the boundary of $\Omega\mathcal{M}_3(4)$ depicted in Figure 6.

In light of Remark 5.2, it is always a good idea to use intrinsic properties of the graphs to safely retrieve their profile:

```
sage: X=GeneralisedStratum([Signature((4,))])
sage: left_graph = [ep for ep in X.enhanced_profiles_of_length(2)
....:     if set(X.lookup_graph(*ep).LG.prongs.values()) == set([1,3]) \
....:         and len(X.lookup_graph(*ep).LG.edges) == 4 \
....:         and not X.lookup_graph(*ep).LG.has_long_edge()]
sage: assert len(left_graph) == 1
sage: left_graph = left_graph[0]
sage: right_graph = [ep for ep in X.enhanced_profiles_of_length(2)
....:     if set(X.lookup_graph(*ep).LG.prongs.values()) == set([1,3]) \
....:         and len(X.lookup_graph(*ep).LG.edges) == 3 \
....:         and X.lookup_graph(*ep).LG.has_long_edge()]
sage: assert len(right_graph) == 1
sage: right_graph = right_graph[0]
```

We check that the undegenerations agree as claimed:

```
sage: assert right_graph[0] == left_graph[0]  # compare profiles
```

```
sage: assert X.lookup_graph(*right_graph).delta(1).is_isomorphic(X.bics[
    right_graph[0][0]])   # check profiles are compatible with delta as
    claimed
sage: assert X.lookup_graph(*right_graph).delta(2).is_isomorphic(X.bics[
    right_graph[0][1]])
sage: assert X.lookup_graph(*left_graph).delta(1).is_isomorphic(X.bics[
    left_graph[0][0]])
sage: assert X.lookup_graph(*left_graph).delta(2).is_isomorphic(X.bics[
    left_graph[0][1]])
```

We confirm that the found graphs correspond to Figure 6:

```
sage: X.lookup_graph(*left_graph).explain()
LevelGraph embedded into stratum Stratum: (4,)
with residue conditions: []
 with:
On level 0:
* A vertex (number 0) of genus 1
On level 1:
* A vertex (number 1) of genus 0
On level 2:
* A vertex (number 2) of genus 0
The marked points are on level 2.
More precisely, we have:
* Marked point (0, 0) of order 4 on vertex 2 on level 2
Finally, we have 4 edges. More precisely:
* 2 edges between vertex 0 (on level 0) and vertex 1 (on level 1) with
    prongs 1 and 1.
* 2 edges between vertex 1 (on level 1) and vertex 2 (on level 2) with
    prongs 3 and 1.
sage: X.lookup_graph(*right_graph).explain()
LevelGraph embedded into stratum Stratum: (4,)
with residue conditions: []
 with:
On level 0:
* A vertex (number 0) of genus 1
On level 1:
* A vertex (number 1) of genus 1
On level 2:
* A vertex (number 2) of genus 0
The marked points are on level 2.
More precisely, we have:
* Marked point (0, 0) of order 4 on vertex 2 on level 2
Finally, we have 3 edges. More precisely:
* one edge between vertex 0 (on level 0) and vertex 1 (on level 1) with
    prong 1.
* one edge between vertex 1 (on level 1) and vertex 2 (on level 2) with
    prong 3.
* one edge between vertex 0 (on level 0) and vertex 2 (on level 2) with
    prong 1.
```

One can (and should) inspect the enhanced profile stored in `right_graph` and `left_graph` but, as mentioned above, the actual numbers will differ between `sage` sessions, while the above code should always retrieve the same graphs.

For the record, in $\Omega\mathcal{M}_3(4)$, among the 19 three-level graphs, 15 have an irreducible profile, while there are two pairs sharing a profile:

```
sage: len(X.enhanced_profiles_of_length(2))
19
sage: len([p for p in X.lookup_list[2] if len(X.lookup(p)) == 1])
15
sage: len([p for p in X.lookup_list[2] if len(X.lookup(p)) == 2])
2
```

We end this section with a brief summary of the various methods around undegeneration. They come, broadly speaking, in two groups: the first are methods of `EmbeddedLevelGraph` and thus use the explicit graphs; the second are methods of `GeneralisedStratum` and deal exclusively with (enhanced) profiles.

In `EmbeddedLevelGraph`:

- `squish_vertical` returns the `EmbeddedLevelGraph` where the supplied level-crossing has been squished.
- `delta` returns the BIC (as an `EmbeddedLevelGraph`) where all but the supplied level-crossings have been squished.

In `GeneralisedStratum`:

- `squish` may be applied to an *enhanced profile* and performs the vertical squish in the language of enhanced profiles.
- `is_degeneration` may be applied to two *enhanced profiles* and answers if the first is a degeneration of the second.
- `lies_over` may be applied to two BICs `i` and `j` (as indices of `bics`, i.e. `int`s) and answers if `(i, j)` is a non-empty profile.
- `merge_profiles` takes two *profiles* (i.e. `tuple`s of indices of `bics`) and merges them with respect to the ordering of `lies_over`.
- `common_degenerations` may be applied to a pair of *enhanced profiles* and returns a `list` of *enhanced profiles* that occur as common degenerations.
- `codim_one_degenerations` may be applied to an *enhanced profile* and returns a `list` of *enhanced profiles* corresponding to all degenerations with exactly one more level.
- `codim_one_common_undegenerations` may be applied to three *enhanced profiles* and gives a list of codimension one degenerations of the third, that are also degenerations of the other two *enhanced profiles* provided.
- `minimal_common_undegeneration` given two *enhanced profiles* return the *enhanced profile* of the graph of minimal dimension that is a common degeneration of both.

More details can be found in the `docstring`s of the methods.

We now explain the relevant steps of the implementation in more detail.

5.2. **Generating all BICs.** The generation of all BICs is again accomplished by `diffstrata` in several steps: first, a list of all combinatorially possible 2-level graphs is generated for each component of the stratum, then the GRC is checked for each graph and these are then combined to give all possible BICs for a `GeneralisedStratum`. Finally, they are sorted by isomorphism class and checked against the $\mathfrak{R}$-GRC.

For a `GeneralisedStratum` X, the BICs are automatically generated (as a list of `EmbeddedLevelGraphs`) and accessed through `X.bics`. Note that in Sage 9 the numbering of this list changes with every `sage` session. The BICs are generated by `GeneralisedStratum.gen_bics()`.

```
sage: X=GeneralisedStratum([Signature((0,0))])
sage: X.bics
[EmbeddedLevelGraph(LG=LevelGraph([1, 0],[[1], [2, 3, 4]],[(1, 4)],{1: 0, 2:
    0, 3: 0, 4: -2},[0, -1],True),dmp={2: (0, 0), 3: (0, 1)},dlevels={0: 0,
    -1: -1})]
sage: X=GeneralisedStratum([Signature((2,))])
sage: X.bics # order will change!
[EmbeddedLevelGraph(LG=LevelGraph([1, 0],[[1, 2], [3, 4, 5]],[(1, 4), (2, 5)
    ],{1: 0, 2: 0, 3: 2, 4: -2, 5: -2},[0, -1],True),dmp={3: (0, 0)},
    dlevels={0: 0, -1: -1}),
 EmbeddedLevelGraph(LG=LevelGraph([1, 1],[[1], [2, 3]],[(1, 3)],{1: 0, 2: 2,
    3: -2},[0, -1],True),dmp={2: (0, 0)},dlevels={0: 0, -1: -1})]
sage: assert X.bics == X.gen_bic()
```

Combinatorial enumeration. The combinatorial enumeration occurs inside the `bic` module of `diffstrata`. More precisely, the raw list of `LevelGraphs` is generated by `bic.bic_alt_noiso` (which needs to be provided a `Signature` or signature tuple as an argument).

```
sage: from admcycles.diffstrata.bic import bic_alt_noiso
sage: bic_alt_noiso((1,1))
[LevelGraph([1, 1],[[3], [1, 2, 4]],[(3, 4)],{1: 1, 2: 1, 4: -2, 3: 0},[0,
    -1],True),
 LevelGraph([2, 0],[[3], [1, 2, 4]],[(3, 4)],{1: 1, 2: 1, 4: -4, 3: 2},[0,
    -1],True),
 LevelGraph([1, 1, 0],[[3], [4], [1, 2, 5, 6]],[(3, 5), (4, 6)],{1: 1, 2: 1,
    5: -2, 6: -2, 3: 0, 4: 0},[0, 0, -1],True),
 LevelGraph([1, 1, 0],[[4], [3], [1, 2, 5, 6]],[(3, 5), (4, 6)],{1: 1, 2: 1,
    5: -2, 6: -2, 3: 0, 4: 0},[0, 0, -1],True),
 LevelGraph([1, 0],[[3, 4], [1, 2, 5, 6]],[(3, 5), (4, 6)],{1: 1, 2: 1, 5:
    -2, 6: -2, 3: 0, 4: 0},[0, -1],True)]
```

Note that in Sage 9 the ordering of these is no longer deterministic, i.e. the BICs will be produced in a different ordering in every `sage` session.

To describe the algorithm, we denote by z the number of zeros and by **z** the corresponding vector of these zeros. By analogy, we denote by p the number of poles, by **p** the vector of poles and by n the number of marked points (the marked points are all of order zero and thus do not need to be distinguished here). Furthermore, we denote the associated genus by g.

We recall some elementary bounds: We denote the maximal sum of genera of vertices on bottom level by `g_bot_max` and the minimal sum of genera of vertices on top level by `g_top_min`. As every top-level vertex v with $g_v = 0$ requires at least one pole, `g_bot_max` $= g - 1$ and `g_top_min` $= 1$ for holomorphic strata and correspondingly `g_bot_max` $= g$ and `g_top_min` $= 0$ for meromorphic strata.

We now summarise the actual algorithm. The steps correspond mostly to nested loops.

Algorithm 5.7 (BIC Generation)**.**

Step 1: We begin by iterating over `bot_comp_len`, the possible number of vertices on bottom level. As every bottom-level vertex needs at minimum either a zero or (if it's genus 0 and has only one edge going up) two marked points, we note that $z + n$ is an upper bound for `bot_comp_len`.

Step 2: Next, we distribute the zeros between upper and lower level by iterating over all 2-length partitions of **z** (and also include the case where all zeros are on bottom level), ensuring at each step that we have enough zeros to satisfy `bot_comp_len`.

Step 3: The zeros are distributed onto the bottom components. If there are no marked points, every component needs at least one zero, otherwise we are more flexible. We use the appropriate helper function, `_distribute_fully` (partitions composed with permutations) or `_distribute_points` (essentially a powerset, implemented via computing b-ary representations of numbers up to b^l and using this to place the points on the components).

Step 4: The genus is partitioned into the contribution from the top vertices, the bottom vertices and the graph. For this, we iterate over `total_g_bot` and `total_g_bot` (using the bounds described above).

Step 5: `total_g_bot` is distributed onto the bottom components. For this, we use the helper function `_distribute_part_ordered`, which is essentially a wrapper for partitions of a fixed length filled with zeros. At this point, we have added all zeros and will have to add (at least) double poles for the edges. Thus, if the orders on any vertex v sum up to less than $2g_v$, we can move on to the next iteration.

Step 6: We now distribute the poles **p**. This is again achieved via 2-length partitions. Note that every $g = 0$ vertex on top needs at least one pole (to compensate the edge(s) going down), so this gives an immediate check for the partitions.

Step 7: Next, we distribute the poles among the bottom components. As poles are optional, we use `_distribute_points`. At this point, we also save the difference of $2g_v - 2$ and the orders distributed to v, i.e. the "space" left for half-edges. We test that this is at least -2 on each component, so that there is potential for at least one edge going up for every vertex.

Step 8: *Now we consider the top level for the first time.* We iterate through the number of top-level vertices, `top_comp_len`, which is bounded by the sum of `total_g_bot` and the number of poles on top-level. As we know the number of vertices and the distribution of genus, the Euler formula determines the number of edges, `num_of_edges`. This gives a "global" check for the "spaces" left on the bottom components: they must sum up to $-2 \cdot$ `num_of_edges`.

Step 9: Similar to above, we now distribute genus, poles and zeros on top level: we use again `_distribute_part_ordered` and `_distribute_points`, as any zeros and poles are optional on top-level. We also run the obvious tests on the orders and record the spaces on top (there are much fewer constraints here, as the spaces may well be zero).

Step 10: We now place the half-edges. We again start on bottom-level, because the poles give stronger constraints. Moreover, the half-edge orders on bottom determine those on top. The distribution of the half-edges into the

spaces is accomplished by `_place_legs_on_bot`, which uses partitions to recursively place the poles, and `_place_legs_on_top`, which works similar but uses the orders distributed on bottom-level.

Step 11: To assert that the graph we have created is connected, we create a Sage `Graph` consisting of the edges we have placed. Note that every vertex is attached to an edge and we don't care for multi-edges when checking connectedness, so a very basic Sage `Graph` suffices here.

Step 12: The only thing missing are the marked points. These are distributed via `_distribute_points` among all the vertices. We now check for stability.

Step 13: In the final step, the legs are renumbered for consistency and the data is used to create a `LevelGraph`. At this point, we check the GRC (via `LevelGraph.is_legal()`). We also remove duplicates from the list of generated graphs.

While we do not claim that algorithm is optimal, it certainly runs in a reasonable time even for large strata:

```
sage: from admcycles.diffstrata.bic import bic_alt_noiso
sage: %
CPU times: user 678 ms, sys: 9.38 ms, total: 688 ms
Wall time: 691 ms
384
```

Uniting `LevelGraph`s. The BICs of a `GeneralisedStratum` are the products of the BICs inside each component, subject to the residue conditions. Moreover, if there are several connected components, we also need to include the smooth stratum on each level.

Therefore, `gen_bic` starts by running `bic_alt_noiso` on each connected component and generating the `dmp` for the embedding into the enveloping stratum. This is possible, because the BIC algorithm described above places the points on specific legs[2]: on each BIC, the i-th point of the signature is the point $i + 1$.

Then, potentially the `smooth_LG` of each component is added and we iterate over the product of these `EmbeddedLevelGraph`s. Building the "product graph" is now simply a renumbering issue and accomplished by `unite_embedded_graphs`.

The result is a list of valid `EmbeddedLevelGraph`s (`dmp` is now surjective), on which we can check the $\mathfrak{R}$-GRC via `is_legal`. Finally, we sort this list into isomorphism classes.

All BICs of a stratum are stored in the list `bics`, which is generated automatically on first call:

```
sage: P=GeneralisedStratum([Signature((0,0)),Signature((0,))])
sage: P.bics  # order might change
[EmbeddedLevelGraph(LG=LevelGraph([1, 0, 1],[[1], [2, 3, 4], [5]],[(1, 4)
    ],{1: 0, 2: 0, 3: 0, 4: -2, 5: 0},[0, -1, 0],True),dmp={2: (0, 0), 3:
    (0, 1), 5: (1, 0)},dlevels={0: 0, -1: -1}),
 EmbeddedLevelGraph(LG=LevelGraph([1, 0, 1],[[1], [2, 3, 4], [5]],[(1, 4)
    ],{1: 0, 2: 0, 3: 0, 4: -2, 5: 0},[0, -1, -1],True),dmp={2: (0, 0), 3:
    (0, 1), 5: (1, 0)},dlevels={0: 0, -1: -1}),
```

---

[2]This is only true for the `LevelGraph`s generated on a single component. In the final `EmbeddedLevelGraph`, the legs will be renumbered and this assumption is no longer valid!

```
EmbeddedLevelGraph(LG=LevelGraph([1, 1],[[1, 2], [3]],[],{1: 0, 2: 0, 3:
    0},[0, -1],True),dmp={1: (0, 0), 2: (0, 1), 3: (1, 0)},dlevels={0: 0, -1
    : -1}),
EmbeddedLevelGraph(LG=LevelGraph([1, 1],[[1, 2], [3]],[],{1: 0, 2: 0, 3:
    0},[-1, 0],True),dmp={1: (0, 0), 2: (0, 1), 3: (1, 0)},dlevels={-1: -1,
     0: 0})]
```

5.3. **Generating Profiles and Graphs.** We now make Algorithm 5.4 more explicit.

Listing all Profiles. We begin by constructing the `lookup_list` of all profiles in a `GeneralisedStratum`. The codimension 0 and 1 lists have already been treated: the first consists of the empty profile, the second of (the indices of) all BICs.

The key tool for working with profiles are the maps `top_to_bic` and `bot_to_bic` described above. These are implemented as dictionary objects.

More precisely, let `X` be a `GeneralisedStratum`. Then, for each index `i` of `X.bics`, `X.DG.top_to_bic(i)` is a `dict` mapping indices of `X.bics[i].top.bics` to indices of `X.bics`. Similarly, `X.DG.bot_to_bic(i)` is a `dict` mapping indices of `X.bics[i].bot.bics` to indices of `X.bics`.

Example 5.8. We illustrate this in $\Omega\mathcal{M}_2(2)$, cf. Example 5.5 and note Remark 5.2:

```
sage: X=Stratum((2,))
sage: X.bics  # order might differ
[EmbeddedLevelGraph(LG=LevelGraph([1, 0],[[1, 2], [3, 4, 5]],[(1, 4), (2, 5)
    ]],{1: 0, 2: 0, 3: 2, 4: -2, 5: -2},[0, -1],True),dmp={3: (0, 0)},
    dlevels={0: 0, -1: -1}),
 EmbeddedLevelGraph(LG=LevelGraph([1, 1],[[1], [2, 3]],[(1, 3)],{1: 0, 2: 2,
     3: -2},[0, -1],True),dmp={2: (0, 0)},dlevels={0: 0, -1: -1})]
sage: X.DG.top_to_bic(0)
{0: 1}
sage: X.DG.bot_to_bic(0)
{}
sage: X.DG.top_to_bic(1)
{}
sage: X.DG.bot_to_bic(1)
{0: 0}
sage: X.lookup_list[0]
[()]
sage: X.lookup_list[1]
[(0,), (1,)]
sage: X.lookup_list[2]
[(1, 0)]
```

For completeness, we list the divisors of the top and bottom strata:

```
sage: X.bics[0].top.bics
[EmbeddedLevelGraph(LG=LevelGraph([1, 0],[[1], [2, 3, 4]],[(1, 4)],{1: 0, 2:
     0, 3: 0, 4: -2},[0, -1],True),dmp={2: (0, 0), 3: (0, 1)},dlevels={0: 0,
     -1: -1})]
sage: X.bics[0].bot.bics
[]
sage: X.bics[1].top.bics
[]
```

```
sage: X.bics[1].bot.bics
[EmbeddedLevelGraph(LG=LevelGraph([0, 0],[[1, 2, 3], [4, 5, 6]],[(2, 5), (3,
      6)],{1: -2, 2: 0, 3: 0, 4: 2, 5: -2, 6: -2},[0, -1],True),dmp={1: (0,
      1), 4: (0, 0)},dlevels={0: 0, -1: -1})]
```

Note that these are of course not graphs in `X` (there are more marked points in `dmp`!) but instead the `top` and `bot` strata of the BICs:

```
sage: X.bics[0].top
LevelStratum(sig_list=[Signature((0, 0))],res_cond=[],leg_dict={1: (0, 0), 2
      : (0, 1)})
sage: X.bics[0].bot
LevelStratum(sig_list=[Signature((2, -2, -2))],res_cond=[[(0, 1), (0, 2)]],
      leg_dict={3: (0, 0), 4: (0, 1), 5: (0, 2)})
sage: X.bics[1].top
LevelStratum(sig_list=[Signature((0,))],res_cond=[],leg_dict={1: (0, 0)})
sage: X.bics[1].bot
LevelStratum(sig_list=[Signature((2, -2))],res_cond=[[(0, 1)]],leg_dict={2:
      (0, 0), 3: (0, 1)})
```

For examples of `top_to_bic` and `bot_to_bic` failing to be injective, see Example 5.22 and Example 8.1 below.

Remark 5.9. Similar to `top_to_bic` and `bot_to_bic` there is also the function `X.DG.middle_to_bic` describing degenerations of the middle levels of 3-level graphs. In light of Remark 3.8, these give all ways of recursively extending a profile.

While `middle_to_bic` is not needed for generating all `EmbeddedLevelGraphs` it is essential for pulling back classes from a level, cf. Section 8.

With the help of `top_to_bic` and `bot_to_bic`, it is not difficult to construct the `lookup_list` recursively:

Algorithm 5.10 (Profile Generation)**.**

    Step 1: Construct `lookup_list[0]` and `lookup_list[1]`.

    Step 2: Constructing profiles of length $l + 1 \geq 2$: for every profile $p = (p_1, \ldots, p_l)$ of length l, add any profile of the form $(p_0, \ldots, p_l)$, where p_0 lies in the `top_to_bic`-image of p_1. If $l > 1$, i.e. $p_1 \neq p_l$, we also add the profiles $(p_1, \ldots, p_{l+1})$, where p_{l+1} lies in the `bot_to_bic`-image of p_l.

    Step 3: Duplicates are removed and the `list` constructed in the previous step is appended to `lookup_list`.

    Step 4: Steps 2 and 3 are repeated until the list produced in Step 3 is empty.

Remark 5.11. A few notes on the validity of the above algorithm:

(1) The algorithm terminates, as there are no graphs with more levels than the dimension of the stratum. Indeed, each level increases the codimension of the corresponding boundary contribution by 1 and thus also the dimension of the levels must eventually decrease and they will stop having BICs.

(2) The entries p_i of a profile are distinct, cf. [CMZ20, §5]. Moreover, a BIC can appear only as a top *or* a bottom degeneration of another BIC. Hence p_0 and p_l cannot have been contained in the previous profile.

(3) In the case $l = 1$, the bottom-level degenerations (p_1, p_2) of p_1 are also obtained as top-level degenerations of p_2.

All that is needed in the above construction are the maps `top_to_bic` and `bot_to_bic`, which are constructed as follows (for `top_to_bic(i)`, `bot_to_bic(i)` uses the same algorithm, with the obvious changes).

Algorithm 5.12 (Construction of `top_to_bic` and `bot_to_bic`).

> **Step 1:** Loop through all BICs `X.bics[i].top`.
>   We denote `X.bics[i].top.bics[j]` by `Bt`.
> **Step 2:** Clutch `Bt` to `X.bics[i].bot` to obtain a (3-level) graph `G`.
> **Step 3:** Squish the bottom level of `G` to obtain the BIC `G.delta(1)`.
> **Step 4:** Go through the list `X.bics` and check which of these is isomorphic to `G.delta(1)`. The index of this graph (in `X.bics`) is stored as the image of `j` in `top_to_bic(i)`.

Remark 5.13. The implementation includes a small optimisation: Using the fact that a BIC can only appear as a top- *or* a bottom-level degeneration of another BIC, we remove the BICs used for `top_to_bic(i)` from the candidates for `bot_to_bic(i)` and vice versa.

Two operations are used in the above algorithm: Clutching and `delta`. Clutching and splitting graphs is a delicate issue and is described in more detail in Section 8.

We briefly describe the process of squishing graphs and the implementation of `delta`.

While we may apply `delta` to an `EmbeddedLevelGraph`, the actual work happens on the underlying `LevelGraph` (the `dmp` is not changed by applying `delta`).

On the `LevelGraph`, `delta` is implemented by consecutively contracting all levels, except for one. Because the graph has fewer levels on each iteration, it is easier to keep track of the numbering when iterating through the levels starting at bottom level.

Remark 5.14. Every `LevelGraph` object should be considered *immutable*. In particular, all these operations *do not* change the `LevelGraph`, but instead always return a new object.

Squishing a single level is achieved by `squish_vertical(i)`.

While level-crossings can be very non-trivial, squishing horizontal edges is straightforward. There are only two possibilities:

(1) The edge is a loop. In this case, remove the edge and increase the genus of the vertex by one.
(2) The edge connects two distinct vertices v_1 and v_2. In this case, remove the edge, add the genus of v_2 to the genus of v_1 and all the legs of v_2 to the legs of v_1 and remove v_2.

We can fall back to this for vertical squishing:

Algorithm 5.15 (Vertical Squishing).

> **Step 1:** Determine the set of vertices `vv` on the next lower level, as well as the edges, `ee`, passing from level `i` to the next lower level.
> **Step 2:** Make a copy of the information needed to generate the `LevelGraph` (i.e. genera, legs, edges, pole orders and levels).
> **Step 3:** Raise the level of every vertex in `vv` to `i` in the copied data.
> **Step 4:** Create a new `LevelGraph` from this data.
> **Step 5:** Squish each of the (now horizontal!) edges in `ee` on this graph.

Remark 5.16. This is not very efficient, because a new graph has to be created for every edge crossing the level. However, this implementation can still be found in the method `LevelGraph.squish_vertical_slow`. The actual `squish_vertical` avoids this, by reimplementing the book-keeping that is avoided by the recursive graph creation, creates the "correct" data in one go and thus creates only one new `LevelGraph`.

Example 5.17. We briefly illustrate squishing a 4-level graph and how this relates to the numbering of the profile. Note that we use `any` so that we make no assumptions about the irreducibility of the profiles.

```
sage: X=GeneralisedStratum([Signature((4,))])
sage: p = X.enhanced_profiles_of_length(4)[0][0]
sage: g = X.lookup_graph(p)
sage: assert any(g.squish_vertical(0).is_isomorphic(G)
                    for G in X.lookup(p[1:]))
sage: assert any(g.squish_vertical(1).is_isomorphic(G)
                    for G in X.lookup(p[:1]+p[2:]))
sage: assert any(g.squish_vertical(2).is_isomorphic(G)
                    for G in X.lookup(p[:2]+p[3:]))
sage: assert any(g.squish_vertical(3).is_isomorphic(G)
                    for G in X.lookup(p[:3]))
```

Remark 5.18. There are some subtleties regarding the level numbering.

(1) The arguments of `delta` and `squish_vertical` are shifted by one, that is `delta(i)` squishes all level passages except the one from $i-1$ to i, while `squish_vertical(i)` squishes the level passage from i to $i+1$.

(2) Note that the `squish_vertical` method exists both for `LevelGraph`s and `EmbeddedLevelGraph`s. Of course, the `EmbeddedLevelGraph` version should be used (the level dictionary needs to be adapted) and the argument is the *relative* level number (while the `LevelGraph` version requires the internal level number).

Building Graphs from Profiles. Now that we have generated all possible profiles, the question remains how to build the graph(s) associated to a profile. This is achieved by recursive clutching. The idea is to pick a BIC B from the profile and sort the remainder into profiles in $B^\top$ an $B^\perp$, build the graphs there and then clutch these together.

More precisely, the `lookup` method of a `GeneralisedStratum X` is implemented, for a profile $(p_0, \ldots, p_l)$, as follows:

Algorithm 5.19 (Graph lookup)**.**

Step 1: Denote by B the BIC with index p_0 in `X.bics`.
Step 2: Create a new nested `list bot_lists`.
Step 3: For every `j` in the (remaining) profile $(p_1, \ldots, p_l)$, find all occurrences of `j` in the values of `bot_to_bic`.
Step 4: For each occurrence of `j` in the values of `bot_to_bic`, append the corresponding key to (a copy of) each `list` in `bot_lists`.
Step 5: Replace `bot_lists` by this nested list and continue the loop at Step 3.

Step 6: Apply `B.bot.lookup` to each profile in `bot_lists` to obtain a `list` of `EmbeddedLevelGraph`s in `B.bot`.

Step 7: Clutch each of these graphs to the graph `B.top` to obtain a `list` of `EmbeddedLevelGraph`s in `X`.

Step 8: Return this `list`, sorted by isomorphism classes.

As mentioned above, clutching and splitting are subtle issues and will be addressed in detail in Section 8.

Remark 5.20. The "branching out" in Step 4 can cause reducible profiles (if `bot_to_bic` is not injective and the clutching also results in non-isomorphic graphs in `X`).

The "inverse" dictionaries of `top_to_bic` and `bot_to_bic` needed in Step 3 are also readily available: Revisiting Example 5.8, we see that

```
sage: X.DG.top_to_bic(0)
{0: 1}
sage: X.DG.top_to_bic_inv(0)
{1: [0]}
sage: X.DG.bot_to_bic(0)
{}
sage: X.DG.bot_to_bic_inv(0)
{}
```

Obviously, the values of `top_to_bic_inv` and `bot_to_bic_inv` have to be `list`s, as the dictionaries are not necessarily injective.

Now, `lookup_graph` simply returns elements of the `list` generated by `lookup` (by default the first).

Remark 5.21. Note that the `lookup` algorithm actually checks both `top_to_bic` and `bot_to_bic` and thus works for an arbitrary permutation of the profile (cf. [CMZ20, Prop. 5.1]).

As the ordering given by `delta` is important at many other places, for useful comparison and caching, it makes sense to restrict to the "ordered" profile.

For example (note that the profile values are "arbitrary", cf. Remark 5.2):

```
sage: X=Stratum((4,))
sage: X.lookup_list[2][0]
(2, 7)
sage: X.lookup((2,7))
[EmbeddedLevelGraph(LG=LevelGraph([1, 0, 0],[[1], [2, 3, 4, 5], [6, 7, 8,
    9]],[(1, 5), (2, 7), (3, 8), (4, 9)],{1: 0, 2: 0, 3: 0, 4: 0, 5: -2, 6:
    4, 7: -2, 8: -2, 9: -2},[0, -1, -2],True),dmp={6: (0, 0)},dlevels={0:
    0, -1: -1, -2: -2})]
sage: X.lookup((7,2))
[EmbeddedLevelGraph(LG=LevelGraph([1, 0, 0],[[1], [2, 3, 4, 5], [6, 7, 8,
    9]],[(3, 7), (4, 8), (5, 9), (1, 2)],{1: 0, 2: -2, 3: 0, 4: 0, 5: 0, 6:
    4, 7: -2, 8: -2, 9: -2},[0, -1, -2],True),dmp={6: (0, 0)},dlevels={0:
    0, -1: -1, -2: -2})]
sage: X.lookup((7,2))[0].is_isomorphic(X.lookup((2,7))[0])
True
```

However, `lookup_graph` does not permit this and requires the profile to be ordered:

```
sage: X.lookup_graph((2,7))
EmbeddedLevelGraph(LG=LevelGraph([1, 0, 0],[[1], [2, 3, 4, 5], [6, 7, 8,
    9]],[(1, 5), (2, 7), (3, 8), (4, 9)],{1: 0, 2: 0, 3: 0, 4: 0, 5: -2, 6:
     4, 7: -2, 8: -2, 9: -2},[0, -1, -2],True),dmp={6: (0, 0)},dlevels={0:
    0, -1: -1, -2: -2})
sage: X.lookup_graph((7,2))
```

5.4. Checking Degenerations.

Clearly, for a graph Γ' to be a degeneration of Γ, it is necessary for the profile of Γ to be an (ordered) subset of the profile of Γ'. However, if Γ is reducible, this criterion is no longer sufficient.

To check whether a graph associated to an enhanced profile is the degeneration of the graph associated to another enhanced profile, diffstrata provides the method is_degeneration:

```
sage: X=Stratum((2,))
sage: X.is_degeneration(((1,), 0), ((), 0))
True
sage: X.is_degeneration(((1,), 0), ((0,), 0))
False
sage: X.is_degeneration(((), 0), ((0,), 0))
False
```

Note that the arguments are enhanced profiles, i.e. tuples of tuples.

Before describing the implementation, we continue to examine some of the phenomena in $\Omega\mathcal{M}_3(4)$, the minimal stratum in genus 3, continuing Example 5.6.

Keeping Remark 5.2 in mind, we nonetheless work with concrete enhanced profiles to ease readability.

Example 5.22. We revisit the situation of Example 5.6. It is not difficult to find the BICs $\delta_1(G)$ and $\delta_2(G)$ in the list of (8) BICs in $\Omega\mathcal{M}_3(4)$. In our case, we see that $\delta_1(G)$ corresponds to BIC 1 and $\delta_2(G)$ corresponds to BIC 6 (note the values of the prongs!).

```
sage: X=Stratum((4,))
sage: len(X.bics)
8
sage: X.bics[1].explain()
LevelGraph embedded into stratum Stratum: (4,)
with residue conditions: []
 with:
On level 0:
* A vertex (number 0) of genus 1
On level 1:
* A vertex (number 1) of genus 1
The marked points are on level 1.
More precisely, we have:
* Marked point (0, 0) of order 4 on vertex 1 on level 1
Finally, we have 2 edges. More precisely:
* 2 edges between vertex 0 (on level 0) and vertex 1 (on level 1) with
    prongs 1 and 1.
sage: X.bics[6].explain()
LevelGraph embedded into stratum Stratum: (4,)
```

```
with residue conditions: []
 with:
On level 0:
* A vertex (number 0) of genus 2
On level 1:
* A vertex (number 1) of genus 0
The marked points are on level 1.
More precisely, we have:
* Marked point (0, 0) of order 4 on vertex 1 on level 1
Finally, we have 2 edges. More precisely:
* 2 edges between vertex 0 (on level 0) and vertex 1 (on level 1) with
    prongs 3 and 1.
```

Consequently, the profile (1, 6) is reducible and contains the graphs G and H. Here, H has enhanced profile ((1, 6), 0) and G is ((1, 6), 1) (note the genera!):

```
sage: X.lookup((1,6))
[EmbeddedLevelGraph(LG=LevelGraph([1, 1, 0],[[1, 2], [3, 4], [5, 6, 7]],[(2,
     4), (3, 6), (1, 7)],{1: 0, 2: 0, 3: 2, 4: -2, 5: 4, 6: -4, 7: -2},[0,
    -1, -2],True),dmp={5: (0, 0)},dlevels={0: 0, -1: -1, -2: -2}),
 EmbeddedLevelGraph(LG=LevelGraph([1, 0, 0],[[1, 2], [3, 4, 5, 6], [7, 8,
    9]],[(1, 5), (2, 6), (3, 8), (4, 9)],{1: 0, 2: 0, 3: 2, 4: 0, 5: -2, 6:
    -2, 7: 4, 8: -4, 9: -2},[0, -1, -2],True),dmp={7: (0, 0)},dlevels={0:
    0, -1: -1, -2: -2})]
```

We see the reducibility reflected in top_to_bic and bot_to_bic (illustrated best by their inverse dictionaries):

```
sage: X.DG.bot_to_bic_inv(1)
{3: [0, 2], 6: [1, 4, 7], 0: [3], 4: [5], 2: [6]}
sage: X.DG.top_to_bic_inv(6)
{1: [0, 4], 4: [1], 5: [2], 3: [3]}
```

The four-level graphs degenerating from this profile can be seen in Figure 7. Applying a compact-type degeneration on top level, the profile remains reducible:

```
sage: X.lookup((5,1,6))
[EmbeddedLevelGraph(LG=LevelGraph([1, 0, 1, 0],[[1], [2, 3, 4], [5, 6], [7,
    8, 9]],[(1, 4), (3, 6), (5, 8), (2, 9)],{1: 0, 2: 0, 3: 0, 4: -2, 5: 2,
     6: -2, 7: 4, 8: -4, 9: -2},[0, -1, -2, -3],True),dmp={7: (0, 0)},
    dlevels={0: 0, -1: -1, -2: -2, -3: -3}),
 EmbeddedLevelGraph(LG=LevelGraph([1, 0, 0, 0],[[1], [2, 3, 4], [5, 6, 7,
    8], [9, 10, 11]],[(1, 4), (2, 7), (3, 8), (5, 10), (6, 11)],{1: 0, 2: 0,
     3: 0, 4: -2, 5: 2, 6: 0, 7: -2, 8: -2, 9: 4, 10: -4, 11: -2},[0, -1,
    -2, -3],True),dmp={9: (0, 0)},dlevels={0: 0, -1: -1, -2: -2, -3: -3})]
```

We thus see the necessity of considering the *enhanced* profile when considering degenerations:

```
sage: X.is_degeneration(((5,1,6),0), ((1,6),0))
True
sage: X.is_degeneration(((5,1,6),0), ((1,6),1))
False
sage: X.is_degeneration(((5,1,6),1), ((1,6),0))
```

FIGURE 7. The four-level graphs degenerating from the *reducible*
profile (1, 6). From left to right: ((5,1, 6), 0), ((5, 1,
6), 1), ((1, 4, 6), 0), ((1, 3, 6), 0) and ((1, 3, 6), 1).
Note that (1, 4, 6) is *irreducible*.

```
False
sage: X.is_degeneration(((5,1,6),1), ((1,6),1))
True
```

Also, G admits a compact-type degeneration on middle level that is not possible in
H for stability reasons. The profile (1, 4, 6) is irreducible:

```
sage: X.lookup((1,4,6))
[EmbeddedLevelGraph(LG=LevelGraph([1, 0, 0, 0],[[1, 2], [3, 4, 5], [6, 7,
    8], [9, 10, 11]],[(1, 4), (2, 5), (3, 8), (6, 10), (7, 11)],{1: 0, 2: 0,
    3: 2, 4: -2, 5: -2, 6: 2, 7: 0, 8: -4, 9: 4, 10: -4, 11: -2},[0, -1,
    -2, -3],True),dmp={9: (0, 0)},dlevels={0: 0, -1: -1, -2: -2, -3: -3})]
sage: X.is_degeneration(((1,4,6),0), ((1,6),0))
False
sage: X.is_degeneration(((1,4,6),0), ((1,6),1))
True
```

The final two degenerations come from intersecting with BIC 3, the "triple banana"
(cf. Example 5.24 and Figure 8). Again, the profile remains reducible but note that
both graphs have long edges now.

```
sage: X.lookup((1,3,6))
[EmbeddedLevelGraph(LG=LevelGraph([1, 0, 0, 0],[[1, 2], [3, 4, 5], [6, 7,
    8], [9, 10, 11]],[(2, 5), (3, 7), (4, 8), (6, 10), (1, 11)],{1: 0, 2: 0,
    3: 0, 4: 0, 5: -2, 6: 2, 7: -2, 8: -2, 9: 4, 10: -4, 11: -2},[0, -1,
    -2, -3],True),dmp={9: (0, 0)},dlevels={0: 0, -1: -1, -2: -2, -3: -3}),
 EmbeddedLevelGraph(LG=LevelGraph([1, 0, 0, 0],[[1, 2], [3, 4, 5], [6, 7,
    8], [9, 10, 11]],[(2, 5), (1, 7), (4, 8), (6, 10), (3, 11)],{1: 0, 2: 0,
    3: 0, 4: 0, 5: -2, 6: 2, 7: -2, 8: -2, 9: 4, 10: -4, 11: -2},[0, -1,
    -2, -3],True),dmp={9: (0, 0)},dlevels={0: 0, -1: -1, -2: -2, -3: -3})]
sage: X.is_degeneration(((1,3,6),0), ((1,6),0))
True
sage: X.is_degeneration(((1,3,6),1), ((1,6),0))
False
```

FIGURE 8. From left to right: the "triple banana", BIC 3; the compact type BIC inside $\Omega\mathcal{M}_0(4, -2, -2, -2)$, the bottom level of the "triple banana"; the unique graph in the profile (3, 6); the unique graph in the profile (1, 3).

```
sage: X.is_degeneration(((1,3,6),1), ((1,6),1))
True
sage: X.is_degeneration(((1,3,6),0), ((1,6),1))
False
```

Note that the two graphs in the profile (1, 3, 6) cannot be distinguished by their levels.

Remark 5.23. We summarise the following observations from Example 5.22.

(1) Extending an irreducible profile can make it reducible.
(2) Extending a reducible profile can make it irreducible.
(3) A reducible profile implies a non-injectivity of `top_to_bic` and `bot_to_bic`.

The converse of the last statement is not true in general.

Example 5.24. We continue in the notation of Example 5.22. Consider the "triple banana", the BIC with a non-trivial S_3 action (cf. Figure 8). It is again not difficult to find in the `list` of BICs, here it is number 3:

```
sage: X.bics[3].explain()
LevelGraph embedded into stratum Stratum: (4,)
with residue conditions: []
 with:
On level 0:
* A vertex (number 0) of genus 1
On level 1:
* A vertex (number 1) of genus 0
The marked points are on level 1.
More precisely, we have:
* Marked point (0, 0) of order 4 on vertex 1 on level 1
Finally, we have 3 edges. More precisely:
* 3 edges between vertex 0 (on level 0) and vertex 1 (on level 1) with
    prongs 1, 1 and 1.
sage: len(X.bics[3].automorphisms)
6
```

The bottom level is the stratum $\Omega\mathcal{M}_0(4, -2, -2, -2)$, which is one-dimensional and contains three BICs of compact type, distinguished only by the numbering of their marked points (dmp!):

```
sage: B=X.bics[3].bot; print(B)
Stratum: Signature((4, -2, -2, -2))
with residue conditions: [(0, 1), (0, 2), (0, 3)]
dimension: 1
leg dictionary: {4: (0, 0), 5: (0, 1), 6: (0, 2), 7: (0, 3)}
leg orbits: [[(0, 0)], [(0, 1), (0, 3), (0, 2)]]

sage: B.bics
[EmbeddedLevelGraph(LG=LevelGraph([0, 0],[[1, 2, 3], [4, 5, 6]],[(3, 6)],{1:
    -2, 2: -2, 3: 2, 4: 4, 5: -2, 6: -4},[0, -1],True),dmp={1: (0, 1), 2:
    (0, 3), 4: (0, 0), 5: (0, 2)},dlevels={0: 0, -1: -1}),
 EmbeddedLevelGraph(LG=LevelGraph([0, 0],[[1, 2, 3], [4, 5, 6]],[(3, 6)],{1:
    -2, 2: -2, 3: 2, 4: 4, 5: -2, 6: -4},[0, -1],True),dmp={1: (0, 2), 2:
    (0, 3), 4: (0, 0), 5: (0, 1)},dlevels={0: 0, -1: -1}),
 EmbeddedLevelGraph(LG=LevelGraph([0, 0],[[1, 2, 3], [4, 5, 6]],[(3, 6)],{1:
    -2, 2: -2, 3: 2, 4: 4, 5: -2, 6: -4},[0, -1],True),dmp={1: (0, 1), 2:
    (0, 2), 4: (0, 0), 5: (0, 3)},dlevels={0: 0, -1: -1})]
```

Checking bot_to_bic, one might expect the profile (3, 6) to be reducible, but it turns out that, no matter how we glue the compact type graph into the bottom level, the resulting graphs are always isomorphic:

```
sage: X.DG.bot_to_bic_inv(3)
{6: [0, 1, 2]}
sage: X.lookup((3,6))
[EmbeddedLevelGraph(LG=LevelGraph([1, 0, 0],[[1, 2, 3], [4, 5, 6], [7, 8,
    9]],[(2, 5), (3, 6), (4, 8), (1, 9)],{1: 0, 2: 0, 3: 0, 4: 2, 5: -2, 6:
    -2, 7: 4, 8: -4, 9: -2},[0, -1, -2],True),dmp={7: (0, 0)},dlevels={0:
    0, -1: -1, -2: -2})]
```

Recall, however, that after intersecting with BIC 1, the profile (1, 3, 6) *is* reducible! In other words, gluing the three BICs of B into the bottom level of (1, 3) yields two non-isomorphic graphs, cf. Example 5.22 and Figure 7.

Example 5.25. The last point of Example 5.24 is worth emphasizing. Using the notation of above, we observe the following situation: the profile (1, 3) is irreducible, the profile (3, 6) is irreducible, but the profile (1, 3, 6) is reducible!

```
sage: X.lookup((1,3))
[EmbeddedLevelGraph(LG=LevelGraph([1, 0, 0],[[1, 2], [3, 4, 5], [6, 7, 8,
    9]],[(2, 5), (1, 7), (3, 8), (4, 9)],{1: 0, 2: 0, 3: 0, 4: 0, 5: -2, 6:
    4, 7: -2, 8: -2, 9: -2},[0, -1, -2],True),dmp={6: (0, 0)},dlevels={0:
    0, -1: -1, -2: -2})]
sage: X.lookup((3,6))
[EmbeddedLevelGraph(LG=LevelGraph([1, 0, 0],[[1, 2, 3], [4, 5, 6], [7, 8,
    9]],[(2, 5), (3, 6), (4, 8), (1, 9)],{1: 0, 2: 0, 3: 0, 4: 2, 5: -2, 6:
    -2, 7: 4, 8: -4, 9: -2},[0, -1, -2],True),dmp={7: (0, 0)},dlevels={0:
    0, -1: -1, -2: -2})]
sage: X.lookup((1,3,6))
```

```
[EmbeddedLevelGraph(LG=LevelGraph([1, 0, 0, 0],[[1, 2], [3, 4, 5], [6, 7,
    8], [9, 10, 11]],[(2, 5), (3, 7), (4, 8), (6, 10), (1, 11)],{1: 0, 2: 0,
    3: 0, 4: 0, 5: -2, 6: 2, 7: -2, 8: -2, 9: 4, 10: -4, 11: -2},[0, -1,
    -2, -3],True),dmp={9: (0, 0)},dlevels={0: 0, -1: -1, -2: -2, -3: -3}),
 EmbeddedLevelGraph(LG=LevelGraph([1, 0, 0, 0],[[1, 2], [3, 4, 5], [6, 7,
    8], [9, 10, 11]],[(2, 5), (1, 7), (4, 8), (6, 10), (3, 11)],{1: 0, 2: 0,
    3: 0, 4: 0, 5: -2, 6: 2, 7: -2, 8: -2, 9: 4, 10: -4, 11: -2},[0, -1,
    -2, -3],True),dmp={9: (0, 0)},dlevels={0: 0, -1: -1, -2: -2, -3: -3})]
```

Recall also Figure 7 and Figure 8.

Determining how and when a profile becomes reducible is still very mysterious. It is also not immediately obvious from `bot_to_bic` and `top_to_bic`:

```
sage: X.DG.bot_to_bic_inv(1)
{3: [0, 2], 6: [1, 4, 7], 0: [3], 4: [5], 2: [6]}
sage: X.DG.bot_to_bic_inv(3)
{6: [0, 1, 2]}
sage: X.DG.top_to_bic_inv(6)
{1: [0, 4], 4: [1], 5: [2], 3: [3]}
sage: X.DG.top_to_bic_inv(3)
{1: [0, 1, 2], 5: [3]}
```

Moreover, one should not expect the reducibility of a profile to be determined by the reducibility of the consecutive length-2 profiles appearing in it. In particular, while the product of the number of components of all length-2-profiles appearing as undegenerations gives an upper bound, this is very coarse and it is not clear how it can be improved (e.g. how to see the irreducibility of (1, 4, 6) in this example).

As these examples illustrate, working with enhanced profiles can be quite subtle and is not yet completely understood. Therefore, `diffstrata` has to occasionally work with the underlying graph. Most methods in `GeneralisedStratum` that give relationships between the underlying `LevelGraph`s of profiles start with `explicit`. The backbone of `is_degeneration` is `explicit_leg_maps`, which raises a `UserWarning` if no map is found, i.e. the enhanced profiles are not degenerations of each other. Using the notation from the above examples, we see e.g.

```
sage: X.lookup_graph((5,))
EmbeddedLevelGraph(LG=LevelGraph([1, 2],[[1], [2, 3]],[(1, 3)],{1: 0, 2: 4,
    3: -2},[0, -1],True),dmp={2: (0, 0)},dlevels={0: 0, -1: -1})
sage: X.lookup_graph((5,6))
EmbeddedLevelGraph(LG=LevelGraph([1, 1, 0],[[1], [2, 3, 4], [5, 6, 7]],[(1,
    4), (2, 6), (3, 7)],{1: 0, 2: 2, 3: 0, 4: -2, 5: 4, 6: -4, 7: -2},[0,
    -1, -2],True),dmp={5: (0, 0)},dlevels={0: 0, -1: -1, -2: -2})
sage: X.explicit_leg_maps(((5,),0), ((5,6),0))
[{1: 1, 2: 5, 3: 4}]
```

Of course, if there are automorphisms involved, there will be many leg maps:

```
sage: X.lookup_graph((1,))
EmbeddedLevelGraph(LG=LevelGraph([1, 1],[[1, 2], [3, 4, 5]],[(1, 4), (2, 5)
    ],{1: 0, 2: 0, 3: 4, 4: -2, 5: -2},[0, -1],True),dmp={3: (0, 0)},
    dlevels={0: 0, -1: -1})
sage: X.lookup_graph((1,3))
```

```
EmbeddedLevelGraph(LG=LevelGraph([1, 0, 0],[[1, 2], [3, 4, 5], [6, 7, 8,
    9]],[(2, 5), (1, 7), (3, 8), (4, 9)],{1: 0, 2: 0, 3: 0, 4: 0, 5: -2, 6:
     4, 7: -2, 8: -2, 9: -2},[0, -1, -2],True),dmp={6: (0, 0)},dlevels={0:
    0, -1: -1, -2: -2})
sage: X.explicit_leg_maps(((1,),0), ((1,3),0))
[{2: 1, 1: 2, 3: 6, 5: 7, 4: 5}, {2: 2, 1: 1, 3: 6, 5: 5, 4: 7}]
sage: len(X.lookup_graph((1,)).automorphisms)
2
sage: len(X.lookup_graph((1,3)).automorphisms)
2
```

In fact, `explicit_leg_maps` first squishes the larger graph at the appropriate places to land in the profile of the smaller graph. Recall from Section 5.3 that this removes legs and vertices but leaves the numbering of the remaining graph untouched! Then it goes through this profile and checks if any component is isomorphic to the squished graph. When one is found, all isomorphisms are returned. Now, `is_degeneration` simply returns a boolean if there exists a leg map.

Now, implementing the methods described in Section 5.1 from these building blocks is fairly straight-forward.

5.5. **Isomorphisms.** The final step in the discussion of graphs and their degenerations is determining, when two graphs are isomorphic. Obviously, an isomorphism of `EmbeddedLevelGraph`s must respect the embedding: recall, e.g., the three compact type BICs in the stratum $\Omega\mathcal{M}_0(4, -2, -2, -2)$ from Example 5.24. In that case, the underlying `LevelGraph`s were identical and the isomorphism classes depended only on the embedding.

On the other hand, an isomorphism of `EmbeddedLevelGraph`s can permute the marked points of a `LevelGraph` if this is "corrected" by the embedding `dmp`.

Definition 5.26. An *isomorphism* of `EmbeddedLevelGraph`s is a `tuple` of `dict`s, (`isom_vertices`, `isom_legs`), satisfying the following compatibility conditions:

(1) `isom_vertices` maps vertices to vertices (as indices of the corresponding `list`s), respecting the genera and levels.
(2) `isom_legs` maps half-edges to half-edges, respecting the edges, the pole-orders and the marked points of the stratum (via `dmp`).

Remark 5.27. The leg maps of Section 5.4 consisted only of the `isom_legs` component.

The methods regarding isomorphisms are quite straight-forward to use. However, as the isomorphism classes are already encoded in the enhanced profiles, this "higher-level" access should be preferred. The number of automorphisms is, obviously, still important.

Automorphisms are recorded in a `list`. We briefly illustrate that they respect the prongs:

```
sage: X=Stratum((4,))
sage: symmetric_banana=EmbeddedLevelGraph(X,LG=LevelGraph([2, 0],[[1, 2],
    [3, 4, 5]],[(1, 4), (2, 5)],{1: 1, 2: 1, 3: 4, 4: -3, 5: -3},[0, -1],
    True),dmp={3: (0, 0)},dlevels={0: 0, -1: -1})
sage: symmetric_banana.automorphisms
[({0: 0, 1: 1}, {2: 1, 1: 2, 3: 3, 5: 4, 4: 5}),
 ({0: 0, 1: 1}, {2: 2, 1: 1, 3: 3, 5: 5, 4: 4})]
```

```
sage: asymmetric_banana=EmbeddedLevelGraph(X,LG=LevelGraph([2, 0],[[1, 2],
    [3, 4, 5]],[(1, 4), (2, 5)],{1: 2, 2: 0, 3: 4, 4: -4, 5: -2},[0, -1],
    True),dmp={3: (0, 0)},dlevels={0: 0, -1: -1})
sage: asymmetric_banana.automorphisms
[({0: 0, 1: 1}, {2: 2, 1: 1, 3: 3, 5: 5, 4: 4})]
```

The cardinality of the automorphism group is seen easily using `len`:

```
sage: len(symmetric_banana.automorphisms)
2
sage: len(asymmetric_banana.automorphisms)
1
sage: set([len(B.automorphisms) for B in X.bics])
{1, 2, 6}
```

We can also simply check if two graphs are isomorphic:

```
sage: symmetric_banana.is_isomorphic(asymmetric_banana)
False
sage: symmetric_banana.is_isomorphic(symmetric_banana)
True
```

Note that the isomorphisms are generators (for checking, it is enough to construct the first). If we want to list all, we should convert them to a `list`:

```
sage: symmetric_banana.isomorphisms(asymmetric_banana)
<generator object EmbeddedLevelGraph.isomorphisms at 0x1cf5112a0>
sage: list(symmetric_banana.isomorphisms(asymmetric_banana))
[]
sage: list(symmetric_banana.isomorphisms(symmetric_banana))
[({0: 0, 1: 1}, {2: 1, 1: 2, 3: 3, 5: 4, 4: 5}),
 ({0: 0, 1: 1}, {2: 2, 1: 1, 3: 3, 5: 5, 4: 4})]
```

For the implementation, we decided to construct isomorphisms level by level. An isomorphism of `EmbeddedLevelGraph`s is then a set of compatible level isomorphisms. We iterate through the isomorphisms on each level and yield whenever we find compatible (i.e. respecting the edges) level isomorphisms for all levels. Note that we use `dlevels` for this, as these should be compatible.

An *isomorphism of levels* now consists of a map (`dict`) vertices to vertices and a map (`dict`) legs to legs, respecting the genus, the number of legs on every vertex, the order at every leg and the marked points of the stratum (via `dmp`).

These are constructed by the following algorithm:

Algorithm 5.28 (Level Isomorphisms)**.**

> **Step 1:** Extract the information about the current level from the `LevelGraph`. Note that we do not use `level` to avoid all the overhead (residue conditions, etc.).
>
> **Step 2:** If the number of vertices, legs, legs per vertex, or the genera do not match, we are done: there can be no isomorphism.
>
> **Step 3:** The same marked points have to be on this level. Also, this gives the first part of the maps, as their legs (and thus also their vertices) must be mapped to each other. Note that we must also ensure that the vertices are compatible (same genera, numbers of legs, orders) and the marked points are split among vertices the same way.

Step 4: For each genus g appearing, we map the vertices of genus g to the vertices of genus g on the target level. For this, we use a simple recursive algorithm, enumerating all legal maps.

Step 5: For each of these vertex maps, we construct all legal leg maps in a similar fashion (vertex by vertex).

Step 6: Finally, we take the product of all constructed level isomorphisms and check which of these are compatible with the edges.

This allows us to construct the degeneration graph of the isomorphism classes non-horizontal level graphs of any generalised stratum and reference the objects in a fairly efficient way.

6. Additive Generators, Tautological Classes and Evaluation

The purpose of the `diffstrata` package is to facilitate calculations in the tautological ring of strata. In light of [CMZ20, Thm. 1.5], any tautological class may be expressed as a formal sum of ψ-classes on graphs.

The `diffstrata` package uses two classes to model the situation. Denote by `X` a `GeneralisedStratum`.

- An `AdditiveGenerator` encodes a product of ψ-classes on (various levels of) an `EmbeddedLevelGraph` inside `X`.
- An `ELGTautClass` is a formal sum of `AdditiveGenerator`s, all on the same stratum `X`.

Any `ELGTautClass`es on `X` can be added, multiplied and evaluated (i.e. integrated against the class of `X`), although this will give 0 if the class is not of top degree. The evaluation works by breaking down the expression into ψ-products on meromorphic strata and using the `admcycles` package [DSZ20] to evaluate these, see Section 6.3 for details.

Moreover, the class $\xi = c_1(\mathcal{O}(-1))$ of `X` is encoded using Sauvaget's relation (cf. [Sau19, Thm. 6(1)], [CMZ20, Prop. 8.2] for the adaption to this setting) by `X.xi`:

```
sage: X=Stratum((2,))
sage: print(X.xi)
Tautological class on Stratum: (2,)
with residue conditions: []

3 * Psi class 1 with exponent 1 on level 0 * Graph ((), 0) +
-1 * Graph ((0,), 0) +
-1 * Graph ((1,), 0) +

sage: print(X.xi^X.dim())
Tautological class on Stratum: (2,)
with residue conditions: []

27 * Psi class 1 with exponent 3 on level 0 * Graph ((), 0) +
-9 * Psi class 1 with exponent 1 on level 0 * Psi class 2 with exponent 1 on
     level 1 * Graph ((0,), 0) +
-2 * Psi class 1 with exponent 1 on level 0 * Psi class 3 with exponent 1 on
     level 1 * Graph ((0,), 0) +
3 * Psi class 1 with exponent 1 on level 0 * Graph ((0, 1), 0) +
-1/2 * Psi class 1 with exponent 1 on level 0 * Psi class 2 with exponent 1
     on level 0 * Graph ((1,), 0) +
```

```
-1/4 * Psi class 1 with exponent 2 on level 0 * Graph ((1,), 0) +
-1/4 * Psi class 2 with exponent 2 on level 0 * Graph ((1,), 0) +

sage: (X.xi^X.dim()).evaluate()
-1/640
```

We now describe these classes in some more detail and explain the evaluation process.

6.1. **AdditiveGenerators.** An `AdditiveGenerator` requires three pieces of information:

- a `GeneralisedStratum`, X;
- an *enhanced profile* describing a level graph in X (cf. Section 5.1) and
- a `dict` encoding the powers of psi classes on this graph.

We explain the last item: the enhanced profile has an `EmbeddedLevelGraph` associated to it and each leg of the underlying `LevelGraph` is determined by a number (`int`). The `leg_dict` of an `AdditiveGenerator` is a `dict` with entries of the from `{l : e}`, where `l` is the number of a leg and `e` is the exponent of the ψ-class at that leg (more precisely: the pullback of the ψ-class on the level the leg is on).

Remark 6.1. For caching purposes, `AdditiveGenerator`s should never be created directly. Instead, the `additive_generator` method of `GeneralisedStratum` should be used:

```
sage: A = X.additive_generator(((0,), 0), {2 : 1})
sage: print(A)
Psi class 2 with exponent 1 on level 1 * Graph ((0,), 0)
```

Remark 6.2. `AdditiveGenerator`s should be considered immutable, i.e. once generated they should not be changed. Indeed, each `AdditiveGenerator` has a unique hash, consisting of the enhanced profile and the ψ-dictionary that is used for fast lookup and comparison.

6.2. **Tautological Classes.** An `ELGTautClass` consists of

- a `GeneralisedStratum`, X and
- a `list`, `psi_list`, of `tuple`s, each consisting of a coefficient (usually a rational number) and an `AdditiveGenerator` on X.

`ELGTautClass` has a `reduce` method that is called upon initialisation and after every operation. This method combines elements of `psi_list` having the same `AdditiveGenerator` and removes elements with coefficient zero or that vanish for dimension reasons.

`ELGTautClass`es associated to the same stratum can be added (the `psi_list`s are united and the `reduce` method is applied) and multiplied (see Section 7). Each of these operations yields again an `ELGTautClass`.

Remark 6.3. For summing `ELGTautClass`es, one may use the `sum` function. However, this calls `reduce` at every step of the summation. It is therefore often preferable to use `GeneralisedStratum`'s `ELGsum` method, which first concatenates all `psi_list`s and only calls `reduce` on the total list.

Any `AdditiveGenerator` may be converted to its associated `ELGTautClass` using its `as_taut` method. Any graph in X can be converted to its associated

ELGTautClass using `X.taut_from_graph`. Moreover, every `GeneralisedStratum`, X, comes with a set of "builtin" tautological classes:

- `X.ZERO` is the class with empty `psi_list`.
- `X.ONE` is the class that consists only of the zero-level graph `X.smooth_LG` representing smooth curves.
- `X.psi(l)` is the ψ-class associated to the marked point `l` of X. Note that `l` is a leg number of `X.smooth_LG`. This should only be used for connected strata.
- `X.xi` is the class of ξ in X using Sauvaget's relation (cf. [Sau19, Thm. 6(1)], [CMZ20, Prop. 8.2]). Note that this involves a choice of leg. By default, we choose the one resulting in the expression with the least number of summands. To choose a leg manually, use `X.xi_with_leg` and provide a stratum point (cf. Section 4.1).
- `X.exp_xi` is the class $\exp(\xi)$ on X.
- `X.c1_E` is the first Chern class of $\Omega^1(\log)$ on X (cf. [CMZ20, Thm. 1.1]).
- `X.chern_class(n)` gives the n-th Chern class of $\Omega^1(\log)$ on X (cf. [CMZ20, Thm. 9.10]).
- `X.calL()` (or more generally, `X.calL(ep, i)` for an enhanced profile `ep` associated to a graph Γ in X and a level i of Γ) gives the class $\mathcal{L}$ or more generally $\mathcal{L}_\Gamma^{[i]}$, see Section 7.3.

Note that we can multiply an `ELGTautClass` not only with any rational number, but also with any `sage` object that can be multiplied with a rational number. In particular, this allows us to use `sage` symbolic expressions:

```
sage: X=Stratum((2,))
sage: var('a', 'b')
(a, b)
sage: T=a*X.ONE + b*X.psi(1)
sage: print(T^2)
Tautological class on Stratum: (2,)
with residue conditions: []

a^2 * Graph ((), 0) +
2*a*b * Psi class 1 with exponent 1 on level 0 * Graph ((), 0) +
b^2 * Psi class 1 with exponent 2 on level 0 * Graph ((), 0) +

sage: (T^3).evaluate()
1/1920*b^3
```

6.3. **Evaluation.** Any `ELGTautClass` can be evaluated, `AdditiveGenerators` that are not top degree will evaluate to 0 without complaining:

```
sage: X=Stratum((2,))
sage: (X.psi(1)).evaluate()
0
sage: (X.psi(1)^3).evaluate()
1/1920
```

Note that `ELGTautClass` offers the check `is_equidimensional` and the methods `degree` and `list_by_degree`:

```
sage: X.xi.is_equidimensional()
True
sage: T=X.ONE + X.psi(1)
sage: T.is_equidimensional()
False
sage: print(T.degree(1))
Tautological class on Stratum: (2,)
with residue conditions: []

1 * Psi class 1 with exponent 1 on level 0 * Graph ((), 0) +

sage: print(T.degree(0))
Tautological class on Stratum: (2,)
with residue conditions: []

1 * Graph ((), 0) +
```

Observe that `list_by_degree` is a `list` of length `X.dim()` filled up with `X.ZERO`:

```
sage: T.list_by_degree()
[ELGTautClass(X=GeneralisedStratum(sig_list=[Signature((2,))],res_cond=[]),
    psi_list=[(1, AdditiveGenerator(X=GeneralisedStratum(sig_list=[
    Signature((2,))],res_cond=[]),enh_profile=((), 0),leg_dict={}))]),
 ELGTautClass(X=GeneralisedStratum(sig_list=[Signature((2,))],res_cond=[]),
    psi_list=[(1, AdditiveGenerator(X=GeneralisedStratum(sig_list=[
    Signature((2,))],res_cond=[]),enh_profile=((), 0),leg_dict={1: 1}))]),
 ELGTautClass(X=GeneralisedStratum(sig_list=[Signature((2,))],res_cond=[]),
    psi_list=[]),
 ELGTautClass(X=GeneralisedStratum(sig_list=[Signature((2,))],res_cond=[]),
    psi_list=[])]
```

The key to evaluating an `ELGTautClass` is to split it into pieces that can be evaluated by `admcycles` using the expression of the class of a stratum in the tautological ring of $\overline{\mathcal{M}}_{g,n}$ by Sauvaget [Sau19]. For this purpose, the `stack_factor` of an `AdditiveGenerator` is defined as follows: let `G` be the associated graph, then the `stack_factor` is the quotient of the product of the prongs of `G` and the product of `B.ell` for every `B` appearing in the profile of `G` and the number of automorphisms of `G`. For a BIC `B`, the number `B.ell` is the lcm of the prongs of `B`. By [CMZ20, Lemma 9.12] this factor is necessary to convert an integral over a boundary stratum D_Γ into the product of level-wise evaluations.

Algorithm 6.4 (Evaluation).

Step 1: Go through `psi_list` and take the sum of the evaluation of the `AdditiveGenerators` multiplied with their respective coefficients.

Step 2: Each `AdditiveGenerator` is evaluated by sorting the ψ-classes by level and taking the product of the evaluations of these ψ-polynomials on each level. This product is then multiplied with the `stack_factor`.

Step 3: If the residue space $\mathfrak{R}$ is empty, we evaluate as follows. If the level is disconnected, it evaluates to 0; if it is 0-dimensional, it evaluates to 1; otherwise we use the `Strataclass` function of `admcycles` to generate the tautological class of the stratum on $\overline{\mathcal{M}}_{g,n}$ and integrate this against the ψ-classes, using `admcycle`'s `evaluate`.

Step 4: If the level splits as a product (with residue conditions, i.e. on the level of the underlying graph, cf. Section 4.3), it evaluates to 0, since the fiber dimension to the product of moduli spaces is positive and the ψ-classes are pullbacks from there.

Otherwise, we create a new `GeneralisedStratum` with one residue condition removed. We repeat until this condition is non-trivial (or $\mathfrak{R} = \emptyset$) and evaluate the product of the (original) ψ-expression and the class cut out by this residue condition (using [CMZ20, Prop. 8.3]). The class cut out by a residue condition inside a `GeneralisedStratum` can be obtained by `res_stratum_class`.

Remark 6.5. We provide some more details on the above algorithm.
  (1) If a level of an `AdditiveGenerator` evaluates to 0, the evaluation returns immediately and lower levels are not evaluated.
  (2) The evaluations in Step 3 are based on the algorithms in [DSZ20]. In particular the `Strataclass` function is based on the description of fundamental classes of (non-generalised) strata conjectured in [FP18] and [Sch18] and proven recently in [BHPSS20] based on results from [HS19].

Example 6.6. We illustrate the calculation of a class cut out by a residue condition:

```
sage: X=GeneralisedStratum([Signature((23,5,-13,-17))])
sage: assert X.res_stratum_class([(0,2)]).evaluate() == 5
```

In fact, this stratum has three boundary points corresponding to graphs Γ_1, Γ_2 and Γ_3 that have respectively the marked points of order $(23, 5)$, $(23, -13)$ and $(23, -17)$ on lower level. By [CMZ20, Prop. 8.2] we can express ξ using the first ψ-class as $\int_X \xi = 24 - 29 - 11 - 7 = -23$. Now in [CMZ20, Prop. 8.3] the contributing boundary graphs are Γ_2 (because the point with order -13 is on lower level) and Γ_3 (because the zero residue at the point of order -13 on upper level is automatic), but not Γ_1. We find that the evaluation of the boundary stratum is $-(-23) - 11 - 7 = 5$, as claimed.

7. NORMAL BUNDLES, PULLBACK AND MULTIPLICATION

While adding two `AdditiveGenerator`s is straight-forward, expressing the intersection of two `AdditiveGenerator`s again as a sum of `AdditiveGenerator`s is subtle, in particular if the intersection is not transversal.

A first approximation is finding common degenerations of two graphs, but if these graphs have a common undegeneration, there is a normal bundle contribution. The precise answer is the excess intersection formula [CMZ20, Prop. 8.1].

Example 7.1. Consider the minimal stratum in genus two, $\Omega\mathcal{M}_2(2)$. The class of ξ can be expressed, by Sauvaget's relation, as a sum of `AdditiveGenerator`s (cf. Section 6).

```
sage: X=Stratum((2,))
sage: print(X.xi)
Tautological class on Stratum: (2,)
with residue conditions: []

3 * Psi class 1 with exponent 1 on level 0 * Graph ((), 0) +
-1 * Graph ((0,), 0) +
```

```
-1 * Graph ((1,), 0) +
```

As $\dim \Omega \mathcal{M}_2(2) = 3$, we can evaluate ξ^3. However, calculating this class requires several applications of the excess intersection formula described above:

```
sage: X.dim()
3
sage: print(X.xi^3)
Tautological class on Stratum: (2,)
with residue conditions: []

27 * Psi class 1 with exponent 3 on level 0 * Graph ((), 0) +
-9 * Psi class 1 with exponent 1 on level 0 * Psi class 2 with exponent 1 on
    level 1 * Graph ((0,), 0) +
-2 * Psi class 1 with exponent 1 on level 0 * Psi class 3 with exponent 1 on
    level 1 * Graph ((0,), 0) +
3 * Psi class 1 with exponent 1 on level 0 * Graph ((0, 1), 0) +
-1/2 * Psi class 1 with exponent 1 on level 0 * Psi class 2 with exponent 1
    on level 0 * Graph ((1,), 0) +
-1/4 * Psi class 1 with exponent 2 on level 0 * Graph ((1,), 0) +
-1/4 * Psi class 2 with exponent 2 on level 0 * Graph ((1,), 0) +
```

The resulting top degree class can now be evaluated (cf. Section 6.3) to give the expected result:

```
sage: (X.xi^3).evaluate()
-1/640
```

Before going into the details of the implementation, we illustrate the use and necessity of multiplication inside an ambient stratum. The standard operations `*` and `^` are performed in the Chow ring of the stratum as illustrated above. We can instead work in the Chow ring of any substratum D_Γ by specifying as `ambient` the enhanced profile of Γ.

Example 7.2. Consider the stratum $\Omega \mathcal{M}_2(1, 1)$ and the boundary divisor represented by a compact-type graph with a genus two component on top and a genus zero component on bottom level. By inspecting the `list` of BICs we find that it is BIC 3 (cf. Remark 5.2):

```
sage: Y=Stratum((1,1))
sage: Y.bics[3]
EmbeddedLevelGraph(LG=LevelGraph([2, 0],[[1], [2, 3, 4]],[(1, 4)],{1: 2, 2:
    1, 3: 1, 4: -4},[0, -1],True),dmp={2: (0, 0), 3: (0, 1)},dlevels={0: 0,
    -1: -1})
```

As the stratum `Y` is four-dimensional, we can multiply this graph with ξ^3 to obtain a top-degree class that we may evaluate to find a number:

```
sage: Y.dim()
4
sage: (Y.xi^3*Y.additive_generator(((3,),0))).evaluate()
-1/640
```

This matches the observation that the top level is the minimal stratum in genus two where ξ^3 evaluates to $-\frac{1}{640}$, the bottom level is a point and there are no prongs or automorphisms involved, compare [CMZ20, §4.3 and Lemma 9.12]:

```
sage: Y.bics[3].top
LevelStratum(sig_list=[Signature((2,))],res_cond=[],leg_dict={1: (0, 0)})
sage: (Y.bics[3].top.xi^3).evaluate()   # calculating on Y.bics[3].top
-1/640
```

We can perform the same calculation in Y using `xi_at_level`, see Remark 8.3. However, as the class $\xi_\Gamma^{[i]}$ correspoding to `xi_at_level` lives not on Y but on the BIC 3, performing the normal multiplication with `*` or `^` will not yield the correct result: we must use our BIC Γ as the ambient stratum, as we want to perform the multiplication in $\mathrm{CH}(D_\Gamma)$, i.e. *before* pushing forward to Y.

Indeed, while the evaluation of the cube of `xi_at_level` performed in Y is

```
sage: (Y.xi_at_level(0, ((3,),0))^3).evaluate()
0
```

we obtain the expected result when using the correct ambient stratum, the BIC 3:

```
sage: CT_xi_top = Y.xi_at_level(0, ((3,),0))
sage: (Y.intersection(Y.intersection(CT_xi_top, CT_xi_top, ((3,),0)),
    CT_xi_top, ((3,),0))).evaluate()
-1/640
```

In the special case of taking exponents of `xi_at_level`, we can use the method `xi_at_level_pow`, which computes the exponent with the correct ambient graph:

```
sage: (Y.xi_at_level_pow(0, ((3,),0), 3)).evaluate()
-1/640
```

Indeed, in this case we may even use the method `top_xi_at_level` which computes and evaluates the top-power of ξ:

```
sage: Y.top_xi_at_level(((3,),0), 0)
-1/640
```

In fact, whenever possible this method should be used (even for ξ on the whole stratum), as the results are cached and reused, see Section 9.

Remark 7.3. Note that by Sauvaget [Sau18, Prop. 3.3], for a connected holomorphic stratum X of genus g, $\int_X \xi^k = 0$ for $k \geq 2g$. In this case, nothing is computed and `X.ZERO` is returned immediately.

7.1. **Interface.** While the operations `*` and `^` should work intuitively, the syntax of the underlying methods is more involved. In particular, when enhanced profiles, `AdditiveGenerators` and `ELGTautClass`es appear is a subtle issue that we try to explain in the following summary.

In the following, `ambient` is always an *enhanced profile* corresponding to the ambient stratum in which the multiplication is taking place (see Section 7.2 for details). By default, it always corresponds to the full enveloping stratum, i.e. the empty enhanced profile `((), 0)`.

The following are methods of `GeneralisedStratum`:

- `intersection` calculates the product of two tautological classes inside `ambient`.

      Arguments: `ELGTautClass`, `ELGTautClass`, `ambient`
      Returns: `ELGTautClass`

- `intersection_AG` calculates the product of two additive generators inside `ambient`
  Arguments: `AdditiveGenerator`, `AdditiveGenerator`, `ambient`
  Returns: `ELGTautClass`
- `normal_bundle` calculates the normal bundle of a graph inside `ambient`.
  Arguments: enhanced profile, `ambient`
  Returns: `ELGTautClass`
- `cnb` calculates the common normal bundle of two graphs inside `ambient`.
  Arguments: Enhanced profile, enhanced profile, `ambient`
  Returns: `ELGTautClass`
- `gen_pullback` calculates the generalised pullback of an additive generator to a graph inside `ambient`.
  Arguments: `AdditiveGenerator`, enhanced profile, `ambient`
  Returns: `ELGTautClass`
- `gen_pullback_taut` calculates the generalised pullback of a tautological class to a graph inside `ambient`.
  Arguments: `ELGTautClass`, enhanced profile, `ambient`
  Returns: `ELTTautClass`

Moreover, the graphs needed are calculated by a series of methods that are straight-forwardly implemented using the observations of Section 5.4. In contrast to the methods described above, the arguments are always enhanced profiles and they return `list`s of enhanced profiles:

- `common_degenerations` finds the common degenerations of two graphs;
- `codim_one_degenerations` finds degenerations of an enhanced profile associated to a graph with one more level;
- `codim_one_common_undegenerations` finds degenerations of `ambient` with one more level that are additionally undegenerations of two given enhanced profiles;
- `minimal_common_undegeneration` finds the graph of minimal dimension that is undegeneration of two supplied graphs.

Finally, an expression of the form `A*B` is evaluated according to the following rules:

- if `A` and `B` are both `ELGTautClass`es, `intersection` is called (with empty `ambient`);
- if one of the two is an `AdditiveGenerator` and the other an `ELGTautClass`, they are multiplied using the `as_taut` method;
- otherwise, if one of the two is an `ELGTautClass`, scalar multiplication with the other is attempted. This can be used, e.g., for multiplication with symbolic variables, cf. Section 6.2.

Note that two `AdditiveGenerators` may only be multiplied (using `*`) if the underlying graphs are the same; otherwise `intersection_AG` should be used. Moreover, multiplying with `X.ONE` is the identity and with `0` or `X.ZERO` gives `X.ZERO`.

Using `^`, we may calculate powers of a `ELGTautClass` (with empty `ambient`). The method `pow` of `GeneralisedStratum` allows the specification of an `ambient`.

7.2. **Intersections.** Mathematically, the situation is the following: we implement the general version of the excess intersection formula [CMZ20, eq. (61)].

Let Λ_1 and Λ_2 be degenerations of a k-level graph Γ and denote the associated strata by D_{Λ_1}, D_{Λ_2} and D_Γ, as in the following diagram:

$$
\begin{array}{ccc}
D_\Pi & \xrightarrow{\ j_{\Pi,\Lambda_2}\ } & D_{\Lambda_2} \\
{\scriptstyle j_{\Pi,\Lambda_1}}\Big\downarrow & & \Big\downarrow{\scriptstyle j_{\Lambda_2,\Gamma}} \\
D_{\Lambda_1} & \xrightarrow{\ j_{\Lambda_1,\Gamma}\ } & D_\Gamma
\end{array}
$$

Furthermore, we define $\nu^\Pi_{(\Lambda_1 \cap \Lambda_2)/\Gamma}$ to be the product of the pullback to D_Π of the normal bundles $\mathcal{N}_{\Gamma',\Gamma}$ where Γ' ranges over all $k+1$-level graphs Γ' that are a degeneration of Γ and that are moreover common to Λ_1 and Λ_2, see Section 7.3. For any $\alpha \in \mathrm{CH}^\bullet(D_{\Lambda_2})$ we can then express its push-forward pulled back to Λ_1 as

$$
\mathrm{j}^*_{\Lambda_1,\Gamma}\mathrm{j}_{\Lambda_2,\Gamma*}\alpha \;=\; \sum_\Pi \mathrm{j}_{\Pi,\Lambda_1,*}\left(\nu^\Pi_{(\Lambda_1 \cap \Lambda_2)/\Gamma} \cdot \mathrm{j}^*_{\Pi,\Lambda_2}\alpha\right),
$$

where the sum ranges over all (Λ_1, Λ_2)-graphs Π. This corresponds to the product of α with the class of Λ_2 in $\mathrm{CH}(D_\Gamma)$. A level graph Π is a (Λ_1, Λ_2)-*graph* if there are undegeneration morphisms $\rho_i \colon \Pi \to \Lambda_i$, i.e. edge contraction morphisms with the property that there are subsets I and J of levels such that $\delta_I(\Pi) = \Lambda_1$ and $\delta_J(\Pi) = \Lambda_2$. For details, see [CMZ20, §8.1].

Remark 7.4. We briefly summarise the relationship to Section 7.1. The correspondence between the mathematical and the `diffstrata` objects is as follows:

- the enveloping (projective) stratum is a `GeneralisedStratum X`;
- the graphs Λ_1, Λ_2 correspond to enhanced profiles `ep_1` and `ep_2` in `X`;
- the graph Γ corresponds to the enhanced profile `ambient`;
- the class α corresponds to an `AdditiveGenerator A` with underlying graph `ep_1`.

Then we may perform the following calculations in `diffstrata`:

- `X.common_undegenerations(ep_1, ep_2)` gives the set of (Λ_1, Λ_2)-graphs;
- each ρ_i is given by `X.explicit_leg_maps`;
- $\mathrm{j}^*_{\Pi,\Lambda_2}\alpha$ is given by `X.gen_pullback(A, pi_i, ambient)` for `pi_i` an enhanced profile in `X.common_undegenerations(ep_1, ep_2)`;
- the individual normal bundles $\mathcal{N}_{\Gamma',\Gamma}$ can be calculated using `normal_bundle`;
- `X.cnb(ep_1, ep_2, ambient)` corresponds to the product of normal bundles appearing in $\nu_{(\Lambda_1 \cap \Lambda_2)/\Gamma}$ (not yet pulled back to $\Lambda_1 \cap \Lambda_2$, see Section 7.3);
- $\mathrm{j}^*_{\Lambda_1,\Gamma}\mathrm{j}_{\Lambda_2,\Gamma*}\alpha$ is given by setting `AG_2=X.additive_generator(ep_2)` and then performing `X.intersection_AG(A, AG_2, ambient)`.

Of course, the product of two `AdditiveGenerator`s on the same graph is simply the product of their ψ-polynomials (sum of the `leg_dict`s) and the product of two tautological classes is the sum of the products of their `AdditiveGenerator`s.

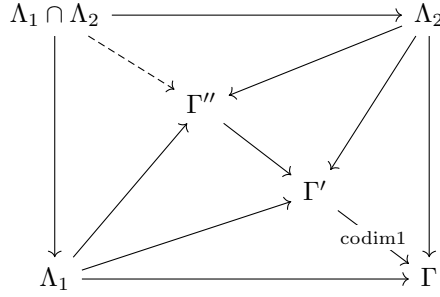Therefore, the excess intersection formula is given by a pullback as the base case (no common intersection) and recursive multiplication with normal bundles. Note that the codimension of `ambient` increases at each step and therefore the normal bundle becomes trivial after finitely many iterations.

7.3. Normal bundles. The key ingredient for the multiplication is the calculation of normal bundles.

The first Chern class of the normal bundle of a divisor is computed in [CMZ20, Thm 7.1]. In `diffstrata`, this can be computed using `normal_bundle`:

```
sage: X=Stratum((2,))
sage: X.normal_bundle(((0,),0)) == X.taut_from_graph((0,),0)^2
True
sage: X.normal_bundle(((1,),0)) == X.taut_from_graph((1,),0)^2
True
```

However, for the excess intersection formula, we need to compute $\nu^{\Pi}_{(\Lambda_1 \cap \Lambda_2)/\Gamma}$. The situation is summarised in the following diagram of the involved graphs:

where the arrows represent undegeneration maps. The graph Γ'' is the *minimal common undegeneration* of Λ_1 and Λ_2 (`X.minimal_common_undegeneration`), corresponding to the intersection of the profiles, while $\Lambda_1 \cap \Lambda_2$ corresponds to the union of the profiles (up to reducibility issues, cf. Section 5.4). The graph(s) Γ' are codimension one degenerations of Γ that are common undegenerations of Λ_1 and Λ_2 (`X.codim_one_common_undegenerations`).

For the excess intersection formula, we need the product of the normal bundles $\mathcal{N}_{\Gamma',\Gamma} = \mathcal{N}_{D_{\Gamma'}/D_\Gamma}$ where Γ' is a $k+1$-level degeneration of the k-level graph Γ. More precisely, the normal bundles $\mathcal{N}_{D_{\Gamma'}/D_\Gamma}$ are pulled back to $D_{\Gamma''}$ and the product is computed in $\mathrm{CH}(D_{\Gamma''})$. The normal bundle is computed in [CMZ20, Prop. 7.5]:

$$c_1(\mathcal{N}_{\Gamma',\Gamma}) \;=\; \frac{1}{\ell_{\delta_i(\Gamma')}}\big(-\xi_\Gamma^{\prime[i]} - c_1(\mathcal{L}_\Gamma^{\prime[i]}) + \xi_\Gamma^{\prime[i+1]}\big) \quad \text{in} \quad \mathrm{CH}^1(D_{\Gamma'}),$$

where

$$\mathcal{L}_{\Gamma'}^{[i]} \;=\; \mathcal{O}_{D_{\Gamma'}}\Big(\sum_{\Gamma' \overset{[i]}{\leadsto} \widehat{\Delta}} \ell_{\delta_{i+1}(\widehat{\Delta})} D_{\widehat{\Delta}}\Big),$$

where the sum runs over all graphs $\widehat{\Delta} \in \mathrm{LG}_{k+2}(\overline{B})$ that yield divisors in $D_{\Gamma'}'$ by splitting the i-th level. These are then pulled back to $D_{\Gamma''}$, see Section 7.4, and then they are multiplied (in $\mathrm{CH}(D_{\Gamma''})$, i.e. using `ambient` Γ'').

Remark 7.5. Observe that this recursive procedure terminates: indeed, the product in $\mathrm{CH}(D_\Gamma)$ has been transformed to a product in $\mathrm{CH}(D_{\Gamma''})$ which is of strictly smaller dimension. Moreover, if the dimension is small enough, transversality is ensured by dimension reasons.

In `diffstrata`, $\xi_\Gamma^{[i]}$ is given by `X.xi_at_level(i, ep)`, where `ep` is the enhanced profile corresponding to Γ in `X`, see Remark 8.3. The easiest way to produce $\mathcal{L}_\Gamma^{[i]}$ is by generating all the BICs inside the `GeneralisedStratum` at level i and glue

these into Γ (see Section 8 for details about gluing BICs into a level). The class of $\mathcal{L}_\Gamma^{[i]}$ is computed by the method calL.

Recall that $\ell_{\delta_i(\Gamma)}$ is the lcm of the prongs of the BIC $\delta_i(\Gamma)$. In diffstrata this is stored in the attribute ell of EmbeddedLevelGraph (if it is a BIC).

We summarise the normal bundle algorithm:

Algorithm 7.6 (Normal Bundle)**.**

> **Step 1:** Compute the minimal_common_undegeneration, min_com (corresponding to Γ'').
> **Step 2:** If min_com is ambient the intersection is transversal, we return 1.
> **Step 3:** Loop through ep in codim_one_common_undegenerations (corresponding to Γ').
> **Step 4:** Calculate the level i where ep and min_com differ.
> **Step 5:** Calculate the normal bundle as in [CMZ20, Eq. (58)] with the function xi_at_level for $\xi^{[i]}$ and glue_AG_at_level for $\mathcal{L}$ (see Section 8 for details).
> **Step 6:** Pull this normal bundle back to min_com, cf. Section 7.4.
> **Step 7:** Return the product of the normal bundles (one for each ep) inside min_com.

Remark 7.7. Note that in the case of a transversal intersection, the "dummy" value 1 is returned:

```
sage: X.cnb(((1,),0),((0,),0))
1
```

The reason for this inconsistency is that the normal bundle should correspond to the class ONE, but inside an ambient this would be ambient, which is not what we want. Therefore, this case must be handled separately!

Example 7.8. We continue in the setting of Example 7.2 in the boundary of $\Omega\mathcal{M}_2(1,1)$. Recall that Y.bics[3] was the compact-type graph with top-level an $\mathcal{M}_2(2)$. We may intersect this graph with the banana graph and the other compact-type graph in Y:

```
sage: Y.bics[3]
EmbeddedLevelGraph(LG=LevelGraph([2, 0],[[1], [2, 3, 4]],[(1, 4)],{1: 2, 2:
    1, 3: 1, 4: -4},[0, -1],True),dmp={2: (0, 0), 3: (0, 1)},dlevels={0: 0,
    -1: -1})
sage: Y.bics[1]
EmbeddedLevelGraph(LG=LevelGraph([1, 0],[[1, 2], [3, 4, 5, 6]],[(1, 5), (2,
    6)],{1: 0, 2: 0, 3: 1, 4: 1, 5: -2, 6: -2},[0, -1],True),dmp={3: (0, 0),
    4: (0, 1)},dlevels={0: 0, -1: -1})
sage: Y.bics[2]
EmbeddedLevelGraph(LG=LevelGraph([1, 1],[[1], [2, 3, 4]],[(1, 4)],{1: 0, 2:
    1, 3: 1, 4: -2},[0, -1],True),dmp={2: (0, 0), 3: (0, 1)},dlevels={0: 0,
    -1: -1})
```

Calculating the common normal bundle of these intersections we obtain:

```
sage: print(Y.cnb(((1,3),0),((2,3),0)))
Tautological class on Stratum: (1, 1)
with residue conditions: []
```

```
-1 * Psi class 1 with exponent 1 on level 0 * Graph ((3,), 0) +
```

As expected, this is the normal of the BIC 3, as the other two BICs intersect transversally:

```
sage: print(Y.normal_bundle(((3,),0)))
Tautological class on Stratum: (1, 1)
with residue conditions: []

-1 * Psi class 1 with exponent 1 on level 0 * Graph ((3,), 0) +
```

Calculating, e.g.

```
sage: print(Y.cnb(((1,3),0),((1,),0)))
Tautological class on Stratum: (1, 1)
with residue conditions: []

-1/2 * Psi class 2 with exponent 1 on level 0 * Graph ((1,), 0) +
-1/2 * Psi class 1 with exponent 1 on level 0 * Graph ((1,), 0) +
-1/2 * Psi class 5 with exponent 1 on level 1 * Graph ((1,), 0) +
-1/2 * Psi class 6 with exponent 1 on level 1 * Graph ((1,), 0) +
```

gives the normal bundle of the banana graph.

7.4. **Pulling back classes.** To calculate the pullback of an `AdditiveGenerator`, we consider first the base case and then the generalised case.

Let `ep` be the enhanced profile of a graph Λ in `X`, `A` an `AdditiveGenerator` on `ep` corresponding to a class α on D_Λ and `ep_deg` the enhanced profile of a degeneration Π of Λ, i.e. we obtain Λ by contracting some of the level-crossings of Π. Then there are finitely many contraction morphisms $\rho\colon \Pi \to \Lambda$ and each of these gives a well-defined pullback map of α. The pullback is the weighted sum of these and is given by `A.pull_back(ep_deg)`.

Example 7.9. Consider the minimal stratum in genus 2, $\Omega\mathcal{M}_2(2)$, with the notation of Example 5.5, see also Figure 5. Let `A` denote the ψ-class on the top-level of the compact-type divisor:

```
sage: X=Stratum((2,))
sage: X.bics
[EmbeddedLevelGraph(LG=LevelGraph([1, 0],[[1, 2], [3, 4, 5]],[(1, 4), (2, 5)
    ],{1: 0, 2: 0, 3: 2, 4: -2, 5: -2},[0, -1],True),dmp={3: (0, 0)},
    dlevels={0: 0, -1: -1}),
 EmbeddedLevelGraph(LG=LevelGraph([1, 1],[[1], [2, 3]],[(1, 3)],{1: 0, 2: 2,
    3: -2},[0, -1],True),dmp={2: (0, 0)},dlevels={0: 0, -1: -1})]
sage: A=X.additive_generator(((1,),0), {1:1})
sage: print(A)
Psi class 1 with exponent 1 on level 0 * Graph ((1,), 0)
```

We may pull this class back to the intersection with the banana graph `((1,0), 0)`:

```
sage: print(A.pull_back(((1,0),0)))
Tautological class on Stratum: (2,)
with residue conditions: []

1 * Psi class 1 with exponent 1 on level 0 * Graph ((1, 0), 0) +
```

Considering instead the class B of the ψ-class on bottom-level of the same divisor:

```
sage: B=X.additive_generator(((1,),0), {3:1}); print(B)
Psi class 3 with exponent 1 on level 1 * Graph ((1,), 0)
sage: print(B.pull_back(((1,0),0)))
Tautological class on Stratum: (2,)
with residue conditions: []
```

It vanishes for dimension reasons when pulled back to the intersection.

Finally, consider the class S of one of the top half-legs of the banana graph. Pulling it back to the intersection will also vanish for dimension reasons, but pulling it back to the banana graph itself illustrates the weighted sum: there are two graph morphisms (switching the edges).

```
sage: S=X.additive_generator(((0,),0), {1:1}); print(S)
Psi class 1 with exponent 1 on level 0 * Graph ((0,), 0)
sage: print(S.pull_back(((0,),0)))
Tautological class on Stratum: (2,)
with residue conditions: []

1/2 * Psi class 2 with exponent 1 on level 0 * Graph ((0,), 0) +
1/2 * Psi class 1 with exponent 1 on level 0 * Graph ((0,), 0) +
```

Notice that in all of the above cases an `ELGTautClass` is returned.

The implementation of `pull_back` is simply a matter of transforming the `leg_dict` of A via the map ρ to `ep_deg` and dividing by the number of these maps. The maps are given by `explicit_leg_maps`, cf. Section 5.4.

However, for the excess intersection formula, we require a more general notion of pullback. As above, let `ep` be the enhanced profile of a graph Λ in X, A an `AdditiveGenerator` on `ep` corresponding to a class α on D_Λ, but now we do not require the "target" graph Λ' to be a degeneration of Λ. Instead, we will pull α back to the intersection $\Lambda \cap \Lambda'$:

Note that this must include the normal bundle contribution of the minimal common undegeneration Γ'' of Λ and Λ' (in D_Γ).

More precisely, the algorithm to compute the pullback of α to $\Lambda \cap \Lambda'$ is:

Algorithm 7.10 (Pullback)**.**

Step 1: Compute the common normal bundle of α and Λ' in D_Γ (Section 7.3).

Step 2: If it is 1, i.e. the intersection is transversal, perform the pullback to each graph of $\Lambda \cap \Lambda'$ as described above.

Step 3: Otherwise, multiply (in $\mathrm{CH}(\Lambda \cap \Lambda')$) the pullback to each graph of $\Lambda \cap \Lambda'$ with the normal bundle, with ambient Γ''.

Remark 7.11. Observe that this recursive procedure terminates: indeed, as for the normal bundle calculation, the involved intersections are always performed in an `ambient` stratum of strictly lower dimension, cf. Remark 7.5.

Example 7.12. We continue in the setting of Example 7.2 and Example 7.8 in the boundary of $\Omega\mathcal{M}_2(1, 1)$. Recall that `Y.bics[3]` was the compact-type graph with top-level an $\mathcal{M}_2(2)$. The normal bundle of this graph is simply a ψ-class on top-level:

```
sage: N = Y.normal_bundle(((3,),0))
sage: print(N)
Tautological class on Stratum: (1, 1)
with residue conditions: []

-1 * Psi class 1 with exponent 1 on level 0 * Graph ((3,), 0) +
```

Here, the V-graph is BIC 0. It has empty intersection with the BIC 3. We may still perform the pullback and obtain the `ZERO` class:

```
sage: Y.bics[0]
EmbeddedLevelGraph(LG=LevelGraph([1, 1, 0],[[1], [2], [3, 4, 5, 6]],[(2, 5),
    (1, 6)],{1: 0, 2: 0, 3: 1, 4: 1, 5: -2, 6: -2},[0, 0, -1],True),dmp={3:
    (0, 0), 4: (0, 1)},dlevels={0: 0, -1: -1})
sage: print(Y.gen_pullback_taut(N, ((0,),0)))
Tautological class on Stratum: (1, 1)
with residue conditions: []
```

Note that we must use `gen_pullback_taut` if we want to pull back an `ELGTautClass` instead of an `AdditiveGenerator`!

Consider now the normal bundle `N2` of the BIC 2 (the other compact type graph). Pulling this back to the BIC 3, we obtain the normal bundle on the intersection:

```
sage: N2=Y.normal_bundle(((2,),0))
sage: print(Y.gen_pullback_taut(N2, ((3,),0)))
Tautological class on Stratum: (1, 1)
with residue conditions: []

-1 * Psi class 1 with exponent 1 on level 0 * Graph ((2, 3), 0) +
-1 * Psi class 3 with exponent 1 on level 1 * Graph ((2, 3), 0) +
```

However, this is `N2` pulled back to `(2, 3)` *inside* BIC 2:

```
sage: print(Y.gen_pullback_taut(N2, ((2,3),0)))
Tautological class on Stratum: (1, 1)
with residue conditions: []

2 * Psi class 1 with exponent 1 on level 0 * Psi class 3 with exponent 1 on
    level 1 * Graph ((2, 3), 0) +

sage: print(Y.gen_pullback_taut(N2, ((2,3),0), ((2,),0)))
Tautological class on Stratum: (1, 1)
with residue conditions: []

-1 * Psi class 1 with exponent 1 on level 0 * Graph ((2, 3), 0) +
-1 * Psi class 3 with exponent 1 on level 1 * Graph ((2, 3), 0) +
```
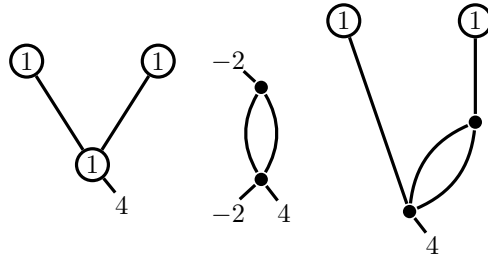
FIGURE 9. From left to right: the V-graph in $\Omega\mathcal{M}_3(4)$; the BICs `0` and `2` in the bottom level of V, distinguished only by the labelings of the marked points on top and bottom level; the unique graph in the profile `(0, 5)`.

The generalised pullback in `Y` is this pullback multiplied with `N2` inside the BIC `2`:

```
sage: print(Y.intersection(Y.gen_pullback_taut(N2, ((2,3),0), ((2,),0)),N2,
    ((2,),0)))
Tautological class on Stratum: (1, 1)
with residue conditions: []

2 * Psi class 1 with exponent 1 on level 0 * Psi class 3 with exponent 1 on
    level 1 * Graph ((2, 3), 0) +
```

Using `gen_pullback` and `cnb` it is not difficult to implement the multiplication of arbitrary tautological classes of a `GeneralisedStratum` using the excess intersection formula [CMZ20, Eq. (61)].

8. Clutching, Splitting and Gluing

Following the philosophy of trading geometric for combinatorial complexity, we wish to transform an expression of classes inside a stratum X into expressions on the levels of the graphs in X. Mathematically, this requires a well-behaved level-projection map, cf. [CMZ20, §4.2]. In practice, this requires us, given a graph Γ and a level L of Γ, to glue a graph Γ' of the generalised stratum L into Γ to obtain a graph Γ'' in X.

In light of Remark 3.8 this allows to reduce any calculation to the levels of two and three level graphs together with the combinatorial data of the degeneration graph of X.

For our purposes, it will suffice to consider the situation that Γ' is a BIC, resulting in a graph in X with one more level than Γ. However, the implementation of this is rather delicate. The main difficulty stems from a level occuring in several degenerations of the same graph but with different automorphism groups acting on the legs. These automorphisms will, in general, not respect the numbering of the BICs, so extra care must be taken when calculating the profile of Γ'' from that of Γ and Γ'.

Example 8.1. Consider the V-graph in the boundary of $\Omega\mathcal{M}_3(4)$, cf. Figure 9. Here it is BIC `0` (cf. Remark 5.2)

FIGURE 10. From left to right: the long V (1, 0); the two graphs in (1, 0, 5) (distinguished by one long edge versus two long edges).

```
sage: X=Stratum((4,))
sage: V=X.bics[0]    # index might change!
sage: V.explain()
LevelGraph embedded into stratum Stratum: (4,)
with residue conditions: []
 with:
On level 0:
* A vertex (number 0) of genus 1
* A vertex (number 1) of genus 1
On level 1:
* A vertex (number 2) of genus 1
The marked points are on level 1.
More precisely, we have:
* Marked point (0, 0) of order 4 on vertex 2 on level 1
Finally, we have 2 edges. More precisely:
* one edge between vertex 0 (on level 0) and vertex 2 (on level 1) with
    prong 1.
* one edge between vertex 1 (on level 0) and vertex 2 (on level 1) with
    prong 1.
```

In the following discussion, the underlying LevelGraph and the leg numbering will be important. Note that the bottom level of V has three BICs, each in the shape of a banana, distinguished only by the location of the marked points: for BIC 1, both are on top, while for 0 and 2 one is on top and one on bottom.

```
sage: V
EmbeddedLevelGraph(LG=LevelGraph([1, 1, 1],[[1], [2], [3, 4, 5]],[(1, 4),
    (2, 5)],{1: 0, 2: 0, 3: 4, 4: -2, 5: -2},[0, 0, -1],True),dmp={3: (0, 0)
    },dlevels={0: 0, -1: -1})
sage: V.bot.bics
[EmbeddedLevelGraph(LG=LevelGraph([0, 0],[[1, 2, 3], [4, 5, 6, 7]],[(2, 6),
    (3, 7)],{1: -2, 2: 0, 3: 0, 4: 4, 5: -2, 6: -2, 7: -2},[0, -1],True),
    dmp={1: (0, 1), 4: (0, 0), 5: (0, 2)},dlevels={0: 0, -1: -1}),
```

```
EmbeddedLevelGraph(LG=LevelGraph([0, 0],[[1, 2, 3, 4], [5, 6, 7]],[(3, 6),
    (4, 7)],{1: -2, 2: -2, 3: 1, 4: 1, 5: 4, 6: -3, 7: -3},[0, -1],True),
    dmp={1: (0, 1), 2: (0, 2), 5: (0, 0)},dlevels={0: 0, -1: -1}),
EmbeddedLevelGraph(LG=LevelGraph([0, 0],[[1, 2, 3], [4, 5, 6, 7]],[(2, 6),
    (3, 7)],{1: -2, 2: 0, 3: 0, 4: 4, 5: -2, 6: -2, 7: -2},[0, -1],True),
    dmp={1: (0, 2), 4: (0, 0), 5: (0, 1)},dlevels={0: 0, -1: -1})]
```

Gluing either of 0 or 2 into bottom level results in the same graph (cf. Figure 9), as they are exchanged by an automorphism of V. This corresponds to both being mapped to the same index by `bot_to_bic` and the corresponding profile (0, 5) being irreducible:

```
sage: X.DG.bot_to_bic(0)
{0: 5, 1: 7, 2: 5}
sage: X.lookup((0,5))
[EmbeddedLevelGraph(LG=LevelGraph([1, 0, 1, 0],[[1], [2, 3, 4], [5], [6, 7,
    8, 9]],[(1, 4), (2, 7), (3, 8), (5, 9)],{1: 0, 2: 0, 3: 0, 4: -2, 5: 0,
    6: 4, 7: -2, 8: -2, 9: -2},[0, -1, 0, -2],True),dmp={6: (0, 0)},
    dlevels={0: 0, -1: -1, -2: -2})]
```

However, the edges are distinguishable in this graph, so e.g. ψ-classes might behave differently!

If we do a top-level degeneration of V first, resulting in the "long" V graph (cf. Figure 10), however, the situation changes. The profile of the long V can be found by inspecting `top_to_bic`.

```
sage: X.DG.top_to_bic(0)
{0: 1, 1: 1}
sage: long_V=X.lookup_graph((1,0)); long_V
EmbeddedLevelGraph(LG=LevelGraph([1, 1, 1],[[1], [2], [3, 4, 5]],[(1, 4),
    (2, 5)],{1: 0, 2: 0, 3: 4, 4: -2, 5: -2},[0, -1, -2],True),dmp={3: (0,
    0)},dlevels={0: 0, -1: -1, -2: -2})
```

Indeed, even though the bottom level has not changed, gluing in BIC 0 or 1 now results in different graphs: the profile (1, 0, 5) is reducible (cf. Figure 9).

```
sage: X.lookup((1,0,5))
[EmbeddedLevelGraph(LG=LevelGraph([1, 1, 0, 0],[[1], [2], [3, 4, 5], [6, 7,
    8, 9]],[(1, 5), (3, 7), (4, 8), (2, 9)],{1: 0, 2: 0, 3: 0, 4: 0, 5: -2,
    6: 4, 7: -2, 8: -2, 9: -2},[0, -1, -2, -3],True),dmp={6: (0, 0)},
    dlevels={0: 0, -1: -1, -2: -2, -3: -3}),
 EmbeddedLevelGraph(LG=LevelGraph([1, 1, 0, 0],[[1], [2], [3, 4, 5], [6, 7,
    8, 9]],[(1, 5), (3, 7), (4, 8), (2, 9)],{1: 0, 2: 0, 3: 0, 4: 0, 5: -2,
    6: 4, 7: -2, 8: -2, 9: -2},[-1, 0, -2, -3],True),dmp={6: (0, 0)},
    dlevels={0: 0, -1: -1, -2: -2, -3: -3})]
```

Therefore, when gluing in a graph of the "reference" level (here `X.bics[0].bot`, the bottom level of the bottom BIC), we must keep track of the isomorphism used to identify `delta` of the graph with the "reference" BIC.

8.1. **Splitting Dictionaries.** To resolve the ambiguities described above, we always have to work with a fixed reference stratum. In accordance with Remark 3.8 this will be either a top or bottom level of a BIC or the middle level of a three-level graph. In the first two cases, this level can be found directly from the profile p: it is either `X.bics[p[0]].top` or `X.bics[p[-1]].bot`.

However, the length-two profile around an intermediate level could be reducible. The method `three_level_profile_for_level` retrieves the enhanced profile of the three-level graph around a given level of the graph associated to an enhanced profile.

To extract all the data we need from a graph to glue a BIC into one of its levels, we use *splitting dictionaries*. The easiest way to create a splitting dictionary is via `splitting_info_at_level`. We illustrate this for the graph `(5,0)` in `X=Stratum((4,))`, in the situation of Figure 9, see Example 8.1:

```
sage: d, leg_dict, L = X.splitting_info_at_level(((0,5),0), 1)
sage: d
{'X': GeneralisedStratum(sig_list=[Signature((4,))],res_cond=[]),
 'top': EmbeddedLevelGraph(LG=LevelGraph([1, 1],[[1], [5]],[],{1: 0, 5:
    0},[0, 0],True),dmp={5: (0, 0), 1: (1, 0)},dlevels={0: 0}),
 'bottom': EmbeddedLevelGraph(LG=LevelGraph([0],[[6, 7, 8, 9]],[],{6: 4, 7:
    -2, 8: -2, 9: -2},[-2],True),dmp={6: (0, 0), 9: (0, 3), 8: (0, 1), 7:
    (0, 2)},dlevels={-2: -2}),
 'middle': LevelStratum(sig_list=[Signature((0, 0, -2))],res_cond=[[(0, 2)
    ]],leg_dict={2: (0, 0), 3: (0, 1), 4: (0, 2)}),
 'emb_dict_top': {},
 'emb_dict_mid': {},
 'emb_dict_bot': {(0, 0): (0, 0)},
 'clutch_dict': {(1, 0): (0, 2)},
 'clutch_dict_lower': {(0, 1): (0, 2), (0, 0): (0, 1)},
 'clutch_dict_long': {(0, 0): (0, 3)}}
sage: leg_dict
{4: (0, 2), 2: (0, 1), 3: (0, 0)}
sage: L
LevelStratum(sig_list=[Signature((0, 0, -2))],res_cond=[[(0, 2)]],leg_dict
    ={2: (0, 0), 3: (0, 1), 4: (0, 2)})
```

As this illustrates, `splitting_info_at_level` actually returns a `tuple` consisting of a splitting dictionary, a dictionary `leg_dict` of legs and a `LevelStratum`. The `LevelStratum` is the standardised level in the sense of Remark 3.8 and will in general differ from the one returned by `level(l)` of the `EmbeddedLevelGraph` associated to the enhanced profile. Similarly, `leg_dict` should be used instead of `dmp` of `level(l)`.

The splitting dictionary uses keywords to encode the relevant data for performing a clutch. More precisely, for an enhanced profile and a level `l`, we obtain a `dict` containing information about the graphs and strata:

> `X:` the enveloping `GeneralisedStratum`;
>
> `top:` an `EmbeddedLevelGraph` inside the `top` component of the top BIC of the three-level graph around the level `l`;
>
> `bottom:` an `EmbeddedLevelGraph` inside the `bot` component of the bottom BIC of the three-level graph around the level `l`;
>
> `middle:` the `LevelStratum` corresponding to the middle level of the three-level graph around the level `l`;

as well as dictionaries describing the clutching. Note that the "removed" edges are now marked points of the strata, so the `dict`s are all in terms of stratum points (cf. Section 4.1). More precisely:

FIGURE 11. From left to right: the "double v"-graph, `11`; the "v"-graph, `23`; their intersection `(11, 23)` all in the boundary of $\Omega\mathcal{M}_3(2,1,1)$. Note that we omit the prongs whenever they are 1.

> **clutch_dict:** a `dict` mapping points of `top` stratum to points of `middle` stratum;
>
> **clutch_dict_lower:** a `dict` mapping points of `middle` stratum to points of `bottom` stratum;
>
> **clutch_dict_long:** a `dict` mapping points of `top` stratum to points of `bottom` stratum.

Finally, there are three `emb_dict`s, relating the marked points of `X` to the marked points in `top`, `middle` and `bottom`.

Using these, we can simply replace `top`, `middle` or `bottom` by degenerations (inside these strata) and use the information stored in the clutching dictionary to "safely" clutch these together.

Example 8.2. Consider the following situation in the boundary of $\Omega\mathcal{M}_3(2,1,1)$: We consider the intersection `G` of the asymmetric "double v"-graph and a "v"-graph depicted in Figure 11. Inspecting `bics`, we find that these are numbered `11` and `23` (cf. Remark 5.2). Consequently, `G` has profile `(11, 23)`.

```
sage: X=Stratum((2,1,1))
sage: G=X.lookup_graph((11, 23))
sage: X.lookup_graph((11,23)).explain()
LevelGraph embedded into stratum Stratum: (2, 1, 1)
with residue conditions: []
 with:
On level 0:
* A vertex (number 0) of genus 1
* A vertex (number 2) of genus 1
On level 1:
* A vertex (number 1) of genus 0
On level 2:
* A vertex (number 3) of genus 0
The marked points are on levels 1 and 2.
More precisely, we have:
* Marked point (0, 1) of order 1 on vertex 1 on level 1
* Marked point (0, 0) of order 2 on vertex 3 on level 2
* Marked point (0, 2) of order 1 on vertex 3 on level 2
Finally, we have 4 edges. More precisely:
```

```
* 2 edges between vertex 0 (on level 0) and vertex 1 (on level 1) with
    prongs 1 and 1.
* one edge between vertex 1 (on level 1) and vertex 3 (on level 2) with
    prong 2.
* one edge between vertex 2 (on level 0) and vertex 3 (on level 2) with
    prong 1.
```

However, comparing the top level of G to the top of the top BIC, 11, we find that the components have been switched:

```
sage: G.level(0)
LevelStratum(sig_list=[Signature((0, 0)), Signature((0,))],res_cond=[],
    leg_dict={1: (0, 0), 2: (0, 1), 7: (1, 0)})
sage: X.bics[11].top
LevelStratum(sig_list=[Signature((0,)), Signature((0, 0))],res_cond=[],
    leg_dict={1: (0, 0), 2: (1, 0), 3: (1, 1)})
```

The top-level degenerations of 11 are given by top_to_bic (cf. Section 5.3). How-ever, to successfully degenerate via clutching, we have to use the bics inside top of BIC 11. This stratum is part of the information given by splitting_info_at_level:

```
sage: X.DG.top_to_bic(11)
{0: 16, 1: 12, 2: 12, 3: 31}
sage: d, leg_dict, L = X.splitting_info_at_level(((11,23),0), 0)
sage: L
LevelStratum(sig_list=[Signature((0,)), Signature((0, 0))],res_cond=[],
    leg_dict={1: (0, 0), 2: (1, 0), 3: (1, 1)})
```

Note that it corresponds to X.bics[11].top, not G.level(0)!

To insert a BIC into the top level, we replace the appropriate entry of the clutching dictionary and feed this to clutch (using Python's ** operator):

```
sage: d
{'X': GeneralisedStratum(sig_list=[Signature((2, 1, 1))],res_cond=[]),
 'top': EmbeddedLevelGraph(LG=LevelGraph([1, 1],[[1, 2], [7]],[],{1: 0, 2:
    0, 7: 0},[0, 0],True),dmp={7: (0, 0), 2: (1, 0), 1: (1, 1)},dlevels={0:
     0}),
 'bottom': EmbeddedLevelGraph(LG=LevelGraph([0, 0],[[3, 4, 5, 6], [8, 9, 10,
    11]],[(4, 10)],{3: 1, 4: 1, 5: -2, 6: -2, 8: 2, 9: 1, 10: -3, 11:
    -2},[-1, -2],True),dmp={3: (0, 1), 8: (0, 0), 9: (0, 2), 11: (0, 3), 6:
     (0, 4), 5: (0, 5)},dlevels={-1: -1, -2: -2}),
 'clutch_dict': {(1, 1): (0, 5), (1, 0): (0, 4), (0, 0): (0, 3)},
 'emb_dict_top': {},
 'emb_dict_bot': {(0, 0): (0, 0), (0, 1): (0, 1), (0, 2): (0, 2)}}
sage: d['top']=L.bics[1]
sage: H=clutch(**d)
```

Note that using G.level(0).bics[1] would result in an error here, as swapping of the components invalidates the gluing data.

In accordance with top_to_bic, we find that the profile of H is in fact given by (12, 11, 23). However, this profile is not reducible:

```
sage: H.is_isomorphic(X.lookup_graph((12,11,23), 0))
False
sage: H.is_isomorphic(X.lookup_graph((12,11,23), 1))
```

```
True
```

Indeed, we obtain the other component by clutching in instead the other BIC of L that is mapped to 12 by `top_to_bic`:

```
sage: d['top']=L.bics[2]
sage: HH=clutch(**d)
sage: HH.is_isomorphic(X.lookup_graph((12,11,23),0))
True
```

Continuing with this graph, we see that we run into a similar problem when trying to degenerate level 1: the reference level around level 1 is the middle level of the three-level graph around this level. We find it using `three_level_profile_for_level` (indeed, the profile is irreducible and we need the "non-standard" component):

```
sage: X.three_level_profile_for_level(((12,11,23),0),1)
((12, 11), 1)
sage: X.lookup_graph((12,11),1).level(1)
LevelStratum(sig_list=[Signature((0,)), Signature((0, 0, -2))],res_cond
    =[[(1, 2)]],leg_dict={1: (0, 0), 3: (1, 0), 4: (1, 1), 5: (1, 2)})
```

Now, it turns out that this reference level is in fact the level 1 of HH but *not* of the reference graph associated to the enhanced profile `((12, 11, 23), 0)` (the fixed representative of the isomorphism class of graphs):

```
sage: HH.level(1)
LevelStratum(sig_list=[Signature((0,)), Signature((0, 0, -2))],res_cond
    =[[(1, 2)]],leg_dict={1: (0, 0), 3: (1, 0), 4: (1, 1), 5: (1, 2)})
sage: X.lookup_graph((12,11,23),0).level(1)
LevelStratum(sig_list=[Signature((0, 0, -2)), Signature((0,))],res_cond
    =[[(0, 2)]],leg_dict={2: (0, 0), 3: (0, 1), 4: (0, 2), 5: (1, 0)})
```

Again, the components have been switched. However, `splitting_info_at_level` gives the reference level (the middle level of `three_level_profile_for_level`) and this is the one we should use for clutching:

```
sage: d, leg_dict, L = X.splitting_info_at_level(((12,11,23),0), 1)
sage: L
LevelStratum(sig_list=[Signature((0,)), Signature((0, 0, -2))],res_cond
    =[[(1, 2)]],leg_dict={1: (0, 0), 3: (1, 0), 4: (1, 1), 5: (1, 2)})
```

In particular, this agrees with `middle_to_bic`:

```
sage: X.DG.middle_to_bic(((12,11),1))
{0: 16, 1: 31}
sage: d
{'X': GeneralisedStratum(sig_list=[Signature((2, 1, 1))],res_cond=[]),
 'top': EmbeddedLevelGraph(LG=LevelGraph([1],[[1]],[],{1: 0},[0],True),dmp
    ={1: (0, 0)},dlevels={0: 0}),
 'bottom': EmbeddedLevelGraph(LG=LevelGraph([0, 0],[[6, 7, 8, 9], [10, 11,
    12, 13]],[(7, 12)],{6: 1, 7: 1, 8: -2, 9: -2, 10: 2, 11: 1, 12: -3, 13:
     -2},[-2, -3],True),dmp={6: (0, 1), 10: (0, 0), 11: (0, 2), 13: (0, 3),
     9: (0, 4), 8: (0, 5)},dlevels={-2: -2, -3: -3}),
 'middle': LevelStratum(sig_list=[Signature((0,)), Signature((0, 0, -2))],
    res_cond=[[(1, 2)]],leg_dict={1: (0, 0), 3: (1, 0), 4: (1, 1), 5: (1, 2)
    }),
```

```
 'emb_dict_top': {},
 'emb_dict_mid': {},
 'emb_dict_bot': {(0, 0): (0, 0), (0, 1): (0, 1), (0, 2): (0, 2)},
 'clutch_dict': {(0, 0): (1, 2)},
 'clutch_dict_lower': {(1, 1): (0, 5), (1, 0): (0, 4), (0, 0): (0, 3)},
 'clutch_dict_long': {}}
sage: d['middle'] = L.bics[0]
sage: HHH=clutch(**d)
sage: HHH.is_isomorphic(X.lookup_graph((12,16,11,23)))
True
sage: d['middle'] = L.bics[1]
sage: HHH=clutch(**d)
sage: HHH.is_isomorphic(X.lookup_graph((12,31,11,23)))
True
sage: HHH.is_isomorphic(X.lookup_graph((12,16,11,23)))
False
```

Note that in the above example, the top strata differed in an obvious manner and clutching immediately raised an error. However, more subtle renumberings of BICs can occur:

```
sage: X.lookup_graph((15,33)).level(0)
LevelStratum(sig_list=[Signature((0, 2))],res_cond=[],leg_dict={1: (0, 0), 2
    : (0, 1)})
sage: X.bics[15].top
LevelStratum(sig_list=[Signature((2, 0))],res_cond=[],leg_dict={1: (0, 0), 2
    : (0, 1)})
sage: X.lookup_graph((15,33)).level(0).bics == X.bics[15].top.bics
False
sage: X.lookup_graph((15,33)).level(0).bics
[EmbeddedLevelGraph(LG=LevelGraph([1, 1],[[1], [2, 3, 4]],[(1, 4)],{1: 0, 2:
    2, 3: 0, 4: -2},[0, -1],True),dmp={2: (0, 1), 3: (0, 0)},dlevels={0: 0,
    -1: -1}),
 EmbeddedLevelGraph(LG=LevelGraph([1, 0],[[1, 2, 3], [4, 5, 6]],[(2, 5), (3,
    6)],{1: 0, 2: 0, 3: 0, 4: 2, 5: -2, 6: -2},[0, -1],True),dmp={1: (0, 0)
    , 4: (0, 1)},dlevels={0: 0, -1: -1}),
 EmbeddedLevelGraph(LG=LevelGraph([1, 1],[[1, 2], [3, 4]],[(2, 4)],{1: 0, 2:
    0, 3: 2, 4: -2},[0, -1],True),dmp={1: (0, 0), 3: (0, 1)},dlevels={0: 0,
    -1: -1}),
 EmbeddedLevelGraph(LG=LevelGraph([2, 0],[[1], [2, 3, 4]],[(1, 4)],{1: 2, 2:
    2, 3: 0, 4: -4},[0, -1],True),dmp={2: (0, 1), 3: (0, 0)},dlevels={0: 0,
    -1: -1}),
 EmbeddedLevelGraph(LG=LevelGraph([1, 0],[[1, 2], [3, 4, 5, 6]],[(1, 5), (2,
    6)],{1: 0, 2: 0, 3: 2, 4: 0, 5: -2, 6: -2},[0, -1],True),dmp={3: (0, 1)
    , 4: (0, 0)},dlevels={0: 0, -1: -1})]
sage: X.bics[15].top.bics
[EmbeddedLevelGraph(LG=LevelGraph([1, 0],[[1, 2, 3], [4, 5, 6]],[(2, 5), (3,
    6)],{1: 0, 2: 0, 3: 0, 4: 2, 5: -2, 6: -2},[0, -1],True),dmp={1: (0, 1)
    , 4: (0, 0)},dlevels={0: 0, -1: -1}),
 EmbeddedLevelGraph(LG=LevelGraph([1, 1],[[1], [2, 3, 4]],[(1, 4)],{1: 0, 2:
    2, 3: 0, 4: -2},[0, -1],True),dmp={2: (0, 0), 3: (0, 1)},dlevels={0: 0,
    -1: -1}),
```

```
EmbeddedLevelGraph(LG=LevelGraph([1, 0],[[1, 2], [3, 4, 5, 6]],[(1, 5), (2,
    6)],{1: 0, 2: 0, 3: 2, 4: 0, 5: -2, 6: -2},[0, -1],True),dmp={3: (0, 0)
    , 4: (0, 1)},dlevels={0: 0, -1: -1}),
EmbeddedLevelGraph(LG=LevelGraph([2, 0],[[1], [2, 3, 4]],[(1, 4)],{1: 2, 2:
    2, 3: 0, 4: -4},[0, -1],True),dmp={2: (0, 0), 3: (0, 1)},dlevels={0: 0,
    -1: -1}),
EmbeddedLevelGraph(LG=LevelGraph([1, 1],[[1, 2], [3, 4]],[(2, 4)],{1: 0, 2:
    0, 3: 2, 4: -2},[0, -1],True),dmp={1: (0, 1), 3: (0, 0)},dlevels={0: 0,
    -1: -1})]
```

In this case, the graphs clutch without error and the mistake is much harder to detect!

```
sage: d, leg_dict, L = X.splitting_info_at_level(((15,33),0), 0)
sage: d
{'X': GeneralisedStratum(sig_list=[Signature((2, 1, 1))],res_cond=[]),
 'top': EmbeddedLevelGraph(LG=LevelGraph([2],[[1, 2]],[],{1: 0, 2: 2},[0],
    True),dmp={1: (0, 1), 2: (0, 0)},dlevels={0: 0}),
 'bottom': EmbeddedLevelGraph(LG=LevelGraph([0, 0],[[3, 4, 5, 6], [7, 8,
    9]],[(5, 8)],{3: 1, 4: 1, 5: 0, 6: -4, 7: 2, 8: -2, 9: -2},[-1, -2],
    True),dmp={3: (0, 1), 4: (0, 2), 7: (0, 0), 9: (0, 4), 6: (0, 3)},
    dlevels={-1: -1, -2: -2}),
 'clutch_dict': {(0, 0): (0, 3), (0, 1): (0, 4)},
 'emb_dict_top': {},
 'emb_dict_bot': {(0, 0): (0, 0), (0, 1): (0, 1), (0, 2): (0, 2)}}
sage: L
LevelStratum(sig_list=[Signature((2, 0))],res_cond=[],leg_dict={1: (0, 0), 2
    : (0, 1)})
sage: X.DG.top_to_bic(15)
{0: 32, 1: 12, 2: 31, 3: 26, 4: 31}
sage: d['top'] = X.lookup_graph((15,33)).level(0).bics[0]
sage: clutch(**d).is_isomorphic(X.lookup_graph((32,15,33)))
False
sage: d['top'] = L.bics[0]
sage: clutch(**d).is_isomorphic(X.lookup_graph((32,15,33)))
True
```

8.2. **Pullback of Classes on a Level.** Since for our goals it is only necessary to pull back the classes ξ and $\mathcal{L}$ from a level, we restrict this discussion to the case of codimension-one classes. There are thus two cases to distinguish. Let G be a graph with enhanced profile ep and L be the standardised level stratum at level l of G. Then a codimension-one class in L is either

- a ψ-class on L or
- a BIC in L.

In the first case, the pullback will consist of ψ-classes on the graph G, in the second case, the pullback class will be a one-level degeneration of G. The pullback of ξ from level l is accomplished by xi_at_level.

Remark 8.3. The method xi_at_level(i,ep) corresponds to the class $\xi_\Gamma^{[i]}$ where Γ is the level graph associated to the enhanced profile ep, see [CMZ20, Prop. 4.7] for details. As the class $\xi_\Gamma^{[i]}$ is an element of $\mathrm{CH}(D_\Gamma)$, care must be taken when multiplying, cf. Example 7.2.

For example, for any Stratum `X`, multiplying any graph with ξ yields an equivalent class to pulling back ξ from top level. We can check this for one-dimensional graphs by evaluating:

```
sage: all((X.xi*X.taut_from_graph(*ep)).evaluate() == X.xi_at_level(0, ep).
    evaluate() for ep in X.enhanced_profiles_of_length(X.dim()-1))
True
```

Remark 8.4. Note that it is essential for `L` to be the standardised level in the sense of Remark 3.8 and obtained via `splitting_info_at_level` (not `G.level(l)`!), to obtain the correct class.

Using `splitting_info_at_level`, the pullback of a ψ-class is straight-forward, as we have access to `L.smooth_LG.dmp` and `leg_dict` to find the correct leg number on `G`.

Pulling back the class of a BIC is a slightly more involved, as we need to determine the enhanced profile of the one-level degeneration of `G`:

Algorithm 8.5 (Gluing in a BIC)**.**
> **Step 1:** Determine the new profile. This is given by the degeneration graph via `X.DG.top_to_bic`, `X.DG.bot_to_bic` or `X.DG.middle_to_bic`, depending on the location of `L`.
> **Step 2:** Determine the graph. For this, we replace `L` by the BIC in the splitting dictionary as illustrated in Example 8.2 and use `clutch` to build the `EmbeddedLevelGraph` in `X`.
> **Step 3:** We find the enhanced profile by locating the isomorphism class of the clutched graph inside the new profile.

Moreover, we need to weigh the pulled back class with the contribution from comparison of multiplicities in the level projections as given in [CMZ20, Prop. 4.7]. The correction factor is the product of the edge contribution and the automorphism contribution. The edge contribution is the quotient of the `ell` of the BIC of `X` that extended the profile and the `ell` of the BIC of `L` that was inserted. The automorphism factor is the quotient of the number of automorphisms of the glued graph and the product of the number of automorphisms of `G` and the BIC of `L`.

Leg choice. As the class of ξ is always implemented via Sauvaget's relation, it requires a choice of leg. By default, the one giving the shortest expression is chosen (i.e. the one appearing on bottom level for the fewest BICs in `L`). Using the optional argument `leg`, we may specify a *leg of* `G` that is to be used. `xi_at_level` will raise a `ValueError` if the leg is not found on level `l`.

Empty Profile. We may apply `xi_at_level` to the empty profile `((), 0)` and obtain ξ on `X`. Note, however, that while `xi_with_leg` requires a stratum point (cf. Section 4.1), the optional `leg` argument of `xi_at_level` requires a leg of `X.smooth_LG`:

```
sage: X=Stratum((2,-2))
sage: X.smooth_LG
EmbeddedLevelGraph(LG=LevelGraph([1],[[1, 2]],[],{1: 2, 2: -2},[0],True),dmp
    ={1: (0, 0), 2: (0, 1)},dlevels={0: 0})
sage: X.xi == X.xi_at_level(0, ((),0))
True
```

```
sage: X.xi_with_leg((0,0)) == X.xi_at_level(0, ((),0), leg=1)
True
sage: X.xi_with_leg((0,1)) == X.xi_at_level(0, ((),0), leg=2)
True
```

8.3. **Splitting Graphs.** Let `G` be the `EmbeddedLevelGraph` associated to the enhanced profile `enh_profile`, written as `(p, i)` inside the `GeneralisedStratum X`. To construct the clutching dictionaries used above, we must, given a level `l`, realise the subgraph of `G` above level `l` inside the `top` level of the BIC `p[l-1]`. Denote by `L` the standardised level `l` of `G`.

Once this has been accomplished, the splitting essentially reduces to the case of a BIC (if `l` is `0`, i.e. top level, or `len(p)`, i.e. bottom level) or that of a three-level graph split around the middle level.

The method `splitting_info_at_level` performs this distinction and serves as a wrapper for the low-level splitting functions, returning all the information needed for clutching.

Extracting subgraphs. The extraction of the subgraph is accomplished by the method `sub_graph_from_level` and yields a "true" subgraph, i.e. the names of the vertices and legs are the same as in `G`.

To get the embedding into the appropriate level, we have to fix an undegeneration to the appropriate BIC via `explicit_leg_maps`, cf. Section 5.4. Note that the marked points of `L` correspond to the marked points of `X` above (resp. below) level `l` on `G` and to the top (resp. bottom) legs of the edges that cross level `l`.

Using `sub_graph_from_level`, it suffices to retrieve the splitting dictionary of the appropriate BIC or three-level graph. The splitting dictionary of the level graph `G` is then obtained by replacing the appropriate strata by the subgraphs given by `sub_graph_from_level`.

Splitting BICs. Splitting BICs is implemented, for a graph with exactly two levels, in the method `split` of `EmbeddedLevelGraph`.

For a BIC, splitting happens in several steps:

Algorithm 8.6 (Splitting a BIC)**.**

> **Step 1:** Extract `top` and `bot` (via `level`).
> **Step 2:** Construct `emb_top` and `emb_bot` by combining `dmp` with the `leg_dicts` of `top` and `bot`.
> **Step 3:** Save the gluing information from the cut edges. Because this is a BIC, all edges are cut in this process.

This yields the splitting dictionary.

Splitting three-level graphs. A three-level graph is determined by its enhanced profile. Consequently, while the splitting of BICs is a method of `EmbeddedLevelGraph`, the corresponding method for three-level graphs, `doublesplit`, is a method of `GeneralisedStratum`.

This is important, because when splitting around a level, everything should be embedded into `top` and `bot` of the two BICs surrounding the level, the only strata we can control. In particular, we need to split into `top` and `bot` of the BICs `p[0]` and `p[1]` and not into `level(0)` and `level(2)` of the three-level graph, as these could differ by a non-trivial automorphism! See Example 8.2.

We therefore have to work with `explicit_leg_maps` to fix undegeneration maps to `p[0]` and `p[1]`. This gives a map from points in the three-level graphs to points in the BIC and we can compose this with the embedding of `top` and `bot` to construct the embedding maps for the splitting dictionary.

As this is a three-level graph, again all edges are cut in this process. In contrast to the BIC case, we need to distinguish edges from top to middle, middle to bottom and top to bottom (long edges) here.

All this is stored in the splitting dictionary and returned by `doublesplit`.

8.4. **Clutching.** The `clutch` method of the `stratatautring` module takes a splitting dictionary and produces from it an `EmbeddedLevelGraph`.

Note that the case clutching two graphs (e.g. degenerating top or bottom level) and clutching three graphs (e.g. degenerating an "interior" level) must be distinguished; one cannot perform clutch twice or recursively, as the intermediate graph is not embedded into the same stratum.

Note that clutching is performed on graphs, so the top, middle and bottom components are converted to their respective `smooth_LG` if one of them is stratum. The clutch is then performed in several steps:

Algorithm 8.7 (Clutching)**.**

      Step 1: Unite the vertices, renumbering the levels appropriately.

      Step 2: Unite the legs, renumbering appropriately. Here we have to keep track of the renumbering for the new `dmp` and to apply the clutching information. We also insert the renumbered (old) edges and use the clutching dictionaries to store the legs that are to be identified.

      Step 3: We create the new edges from the information gathered in Step 2.

      Step 4: We use this data to create an `EmbeddedLevelGraph` that we return.

Using Python's `**` operator, we can feed a splitting dictionary directly into `clutch`:

```
sage: X=GeneralisedStratum([Signature((1,1))])
sage: assert clutch(**X.bics[1].split()).is_isomorphic(X.bics[1])
sage: assert all(clutch(**B.split()).is_isomorphic(B) for B in X.bics)
```

Of course, because of the renumbering, we cannot assume the graphs to be the same after splitting and clutching. They are, however, isomorphic. The same works for three-level graphs and `doublesplit`, also for a more complicated stratum:

```
sage: X=GeneralisedStratum([Signature((1,1))])
sage: assert all(X.lookup_graph(*ep).is_isomorphic(clutch(**X.doublesplit(ep
    ))) for ep in X.enhanced_profiles_of_length(2))
sage: X=GeneralisedStratum([Signature((2,2,-2))])
sage: assert all(X.lookup_graph(*ep).is_isomorphic(clutch(**X.doublesplit(ep
    ))) for ep in X.enhanced_profiles_of_length(2))
```

In particular, there are three-level graphs with long edges in both these strata.

9. Caching

The boundary strata grow in size very quickly. Even for holomorphic strata in genus 3, class calculations would not be possible without extensive caching.

The downside of this is that `diffstrata` has quite an extensive memory footprint; there is surely still room for much optimisation.

9.1. **Graphs.** Working with explicit `LevelGraph`s is painfully slow. The main achievement of Section 5.3 was to associate to each `EmbeddedLevelGraph` an enhanced profile, i.e. a `tuple` (of `tuple`s) of integers for each isomorphism class. Ideally, we always refer to a graph by its enhanced profile.

Remark 9.1. Note that it is important to use `tuple`s and not `list`s for (enhanced) profiles, as `tuple`s are immutable and may thus be used as arguments of cached functions and as keys of dictionaries.

However, for certain degeneration questions as well as level extraction, we do need concrete representations of the graphs as well as explicit isomorphisms and leg maps. For this, we store, for each enhanced profile, a *reference graph*. More precisely, for a `GeneralisedStratum X` and a profile `p`, `X.lookup(p)` will, on first call, generate all graphs in the profile `p` (cf. Section 5.3) and stores this `list` in the dictionary `X._lookup` with key `p`. On subsequent calls, this dictionary is used, so that the clutching is only done once and the *same* `EmbeddedLevelGraph` is returned on every lookup. Moreover, for profiles of length 1, the entries of `bics` are used:

```
sage: G=X.bics[0]
sage: G is X.lookup_graph((0,),0)
True
sage: H=X.lookup_graph((1,0),0)
sage: H is X.lookup_graph((1,0),0)
True
```

Moreover, each `EmbeddedLevelGraph G` has a `list G._levels` that is filled with the extracted levels on first call of `G.level` (and these are subsequently reused). Therefore, also the individual levels of the graph associated to an enhanced profile will not be regenerated. Also, other intrinsic values, such as the number of automorphisms, are stored on first computation and then retrieved.

Finally, `AdditiveGenerator`s are considered immutable and are even hashable, i.e. using them, e.g., as keys in dictionaries works:

```
sage: a=X.additive_generator(((0,),0))
sage: {a : 1}
{AdditiveGenerator(X=GeneralisedStratum(sig_list=[Signature((2,))],res_cond
    =[]),enh_profile=((0,), 0),leg_dict={}): 1}
```

By contrast, `ELGTautClass`es are mutable (their `psi_list`s can and will be changed, for example by `reduce`) and therefore may *not* be used as keys.

This allows any method whose arguments consist *only* of enhanced profiles and `AdditiveGenerator`s to be cached (using `sage`'s `@cached_method` decorator). This is a key reason for splitting all methods in Section 7 into operations involving only these objects (instead of working only with `ELGTautClass`es of `EmbeddedLevelGraph`s).

Also, note that `AdditiveGenerator`s should always be created and used via `X.additive_generator` as this stores them in the `_AGs` dictionary of `X` and allows them to be reused (instead of being created newly on each call):

```
sage: a=X.additive_generator(((0,),0))
sage: a is X.additive_generator(((0,),0))
True
sage: a is AdditiveGenerator(X, ((0,),0))
False
```

This allows computations in strata of genus 3 and 4 in feasible time. However, the memory footprint is considerable: already in genus 3, the larger strata use about 20GB, while in genus 4 already more than a TB is required.

9.2. **Files and Values.** As described in Section 6.3, `diffstrata` uses the package `admcycles` to evaluate top-degree `ELGTautClass`es. As the computations of `admcycles` are also very involved, we cache every use and, in fact, (attempt to) write any computed value into a local file that is automatically (attempted to be) reused.

More precisely, given the signature `sig` of a stratum (note that `admcycles` works only with connected strata without residue conditions, cf. Section 6.3) and a ψ-polynomial `psis` (as a `dict` mapping points of the stratum to exponents), the method `adm_evaluate` uses `adm_key` to compute a key consisting of the signature and `psis` transformed into a `tuple`. To avoid needless recomputations, the signature is sorted and `psis` renumbered accordingly:

```
sage: from admcycles.diffstrata.levelstratum import adm_key
sage: adm_key((2,-2), {1: 2, 2: 1})
((-2, 2), ((1, 1), (2, 2)))
```

Next, we check if the cache dictionary exists and if not, we attempt to load it from the file `adm_evals.sobj` (see below for details). If the key exists in the cache, we return its value, otherwise we use `admcycles` to compute the value, store it in the cache and write this back into the file.

Similarly, whenever we evaluate a top-power of ξ, we save this to file, as the Euler characteristic can be computed purely from the degeneration graph and this information (cf. Section 10.1).

More precisely, for a `GeneralisedStratum` X, an enhanced profile and a level l, the method `X.top_xi_at_level` computes the evaluation of the top-power of $\xi_\Gamma^{[l]}$ on the graph Γ associated to the enhanced profile, i.e. `X.xi_at_level_pow` as in Example 7.2. These numbers are stored in a cache dictionary that is synchronised with the file `top_xis.sobj` as described above.

The ξ-cache uses `LevelStratum`'s method `dict_key` to compute a key, again with the aim of performing as few evaluations as necessary: the list of components is sorted, as is the signature of each component; then the residue conditions are renumbered appropriately and also sorted. Finally, everything is converted to a nested `tuple`.

Example 9.2. We illustrate the `dict_key`s in the stratum $\Omega\mathcal{M}_2(1,1)$. For the V-graph, the top stratum is disconnected and produces:

```
sage: print(VT)
Product of Strata:
Signature((0,))
Signature((0,))
with residue conditions:
dimension: 3
leg dictionary: {1: (0, 0), 2: (1, 0)}
leg orbits: [[(1, 0), (0, 0)]]

sage: VT.dict_key()
(((0,), (0,)), ())
```

The bottom stratum has residue conditions:

```
sage: print(VB)
Stratum: Signature((1, 1, -2, -2))
with residue conditions: [(0, 3)] [(0, 2)]
dimension: 0
leg dictionary: {3: (0, 0), 4: (0, 1), 5: (0, 2), 6: (0, 3)}
leg orbits: [[(0, 0)], [(0, 1)], [(0, 3), (0, 2)]]

sage: VB.dict_key()
(((-2, -2, 1, 1),), (((0, 0),), ((0, 1),)))
```

By contrast, the bottom level of the banana graph produces:

```
sage: print(BB)
Stratum: Signature((1, 1, -2, -2))
with residue conditions: [(0, 2), (0, 3)]
dimension: 1
leg dictionary: {3: (0, 0), 4: (0, 1), 5: (0, 2), 6: (0, 3)}
leg orbits: [[(0, 0)], [(0, 1)], [(0, 3), (0, 2)]]

sage: BB.dict_key()
(((-2, -2, 1, 1),), (((0, 0), (0, 1)),))
```

Importing and Exporting Values. As described above, any value computed with `adm_evaluate` is cached and synchronised with the file `adm_evals.sobj` (actually the `global` variable `FILENAME`, which is set to `adm_evals.sobj` by default). This is accomplished by the method `load_adm_evals`.

The `diffstrata` method `import_adm_evals` takes a filename as an argument and attempts to update the cache dictionary from `load_adm_evals` with the dictionary read from this file. The result is written immediately to `FILENAME`.

Similarly, the ξ-cache is synchronised with `XI_FILENAME`, which is `top_xis.sobj` by default, the loading is handled by `load_xis` and these can be imported using `import_top_xis`.

For file handling, `sage`'s methods `save` and `load` are used, i.e. the files are stored in the *current working directory*. However, `admcycles` versions *after* `v1.1` include the modules `adm_eval_cache` and `xi_cache` that automatically initialise the cache with a large number of values for low-genus strata. The caching is then controlled by the more versatile `cache` module and the files are stored in `DOT_SAGE`.

Example 9.3. Assume we have a computed only the top ξ of the minimal stratum in genus 3, our ξ-cache would look like this and we may export it:

```
sage: my_xi
{(((4,),), ()): 305/580608}
sage: save(my_xi, 'my_xi.sobj')
```

We may import this anywhere (and check):

```
sage: import_top_xis('my_xi.sobj')
sage: load_xis()
{(((4,),), ()): 305/580608}
```

Timing this calculation confirms that the cached value is being used:

```
sage: X=Stratum((4,))
sage: %
CPU times: user 14.9 ms, sys: 970 µs, total: 15.9 ms
Wall time: 15.5 ms
305/580608
```

Printing Values. As the generation of keys for strata and ψ-polynomials described above hinder the readability of the content of the cache, diffstrata includes the methods print_top_xis and print_adm_evals to print these values in a more human-readable format.

Example 9.4. Continuing Example 9.3 from above, assume we are importing a second file and want to confirm that our previous value was not deleted:

```
sage: import_top_xis('my_other_xi.sobj')
sage: print_top_xis()
Stratum           | Residue Conditions        | xi^dim
-----------------------------------------------------------------
(-2, -2, 1, 1)    | [[(0, 0)], [(0, 1)]]      | 1
(4,)              | ()                        | 305/580608
```

We can also pass the dict to be printed as an argument, e.g. to inspect a file before loading:

```
sage: print_top_xis(load('my_xi.sobj'))
Stratum           | Residue Conditions        | xi^dim
-----------------------------------------------------------------
(4,)              | ()                        | 305/580608
sage: print_top_xis(load('my_other_xi.sobj'))
Stratum           | Residue Conditions        | xi^dim
-----------------------------------------------------------------
(-2, -2, 1, 1)    | [[(0, 0)], [(0, 1)]]      | 1
```

Additionally, one may want to filter the content of the cache:

```
sage: xi_cache=load_xis()
sage: val_one = {k : v for k, v in xi_cache.items() if v == 1}
sage: print_top_xis(val_one)
Stratum           | Residue Conditions        | xi^dim
-----------------------------------------------------------------
(-2, -2, 1, 1)    | [[(0, 0)], [(0, 1)]]      | 1
```

To facilitate this, the method list_top_xis picks apart the key of the ξ-cache:

```
sage: for sigs, res_conds, value in list_top_xis():
....:     print('%
....:
((-2, -2, 1, 1),) (((0, 0),), ((0, 1),)) 1
((4,),) () 305/580608
```

10. Tests and Computations

We use this section to briefly illustrate how the described methods of diffstrata may be used to implement some of the key results and crosschecks of formulas in [CMZ20]

10.1. **Euler Characteristics.** We illustrate the methods described above to implement [CMZ20, Thm. 1.3] for computing the Euler characteristics of strata. Recall that the (orbifold) Euler characteristic of the moduli space $\Omega\mathcal{M}_{g,n}(\mu)$ is the dimension-weighted sum over all level graphs $\Gamma \in \mathrm{LG}_L(B)$ without horizontal nodes

$$\chi(B) \;=\; (-1)^d \sum_{L=0}^{d} \sum_{\Gamma \in \mathrm{LG}_L(B)} \ell_\Gamma N_\Gamma^\top \int_{D_\Gamma} \prod_{i=0}^{L} (\xi_\Gamma^{[i]})^{d_\Gamma^{[i]}}$$

of the product of the top power of the of the first Chern class $\xi_\Gamma^{[i]}$ of the tautological bundle at each level, where $d_\Gamma^{[i]}$ is the dimension of the projectivized moduli space at level i and where $d = \dim(B) = N - 1$. The equivalence of the above formula and the one stated in [CMZ20, Eq. (2)] follows from [CMZ20, Lemma 9.12].

   We may implement this using `diffstrata` for a stratum `X` as follows:

Algorithm 10.1 (Euler Characteristic).
   Step 1: Loop over `L` from `0` to `X.dim()`.
   Step 2: Loop over `ep` in `X.enhanced_profiles_of_length(L)`.
   Step 3: Writing the profile `ep=(p, enh)`, the number l_Γ is the product over `X.bics[p[i]].ell` for all `i` and $N_\Gamma^\top$ is `X.bics[p[0]].top.dim() + 1`.
   Step 4: Calculate the product using `top_xi_at_level` (this uses $d_\Gamma^{[i]}$ and returns a number). It's also cached, cf. Section 9.
   Step 5: Sum all these *numbers* together.

Example 10.2. The above algorithm is implemented by `euler_characteristic`. We run this with an empty cache:

```
sage: print_adm_evals()
Stratum            | Psis                            | eval
------------------------------------------------------------------
sage: print_top_xis()
Stratum            | Residue Conditions              | xi^dim
------------------------------------------------------------------
sage: X=Stratum((4,))
sage: %
CPU times: user 7.31 s, sys: 108 ms, total: 7.41 s
Wall time: 7.43 s
-55/504
```

Re-inspecting the cache, we see that it has been filled:

```
sage: print_top_xis()
Stratum            | Residue Conditions              | xi^dim
------------------------------------------------------------------
(-4, -2, 4)        | [(0, 0), (0, 1)]                | 1
(-4, 0, 2)         | [(0, 0)]                        | 1
(-4, 1, 1)         | [(0, 0)]                        | 1
(-4, 4)            | [(0, 0)]                        | -15/8
(-3, -3, 4)        | [(0, 0), (0, 1)]                | 1
(-2, -2, -2, 4)    | [(0, 0), (0, 1), (0, 2)]        | -4
(-2, -2, -2, 4)    | [[(0, 0), (0, 2)], [(0, 1)]]    | 1
(-2, -2, 0, 2)     | [(0, 0), (0, 1)]                | -2
(-2, -2, 1, 1)     | [[(0, 0)], [(0, 1)]]            | 1
(-2, -2, 1, 1)     | [(0, 0), (0, 1)]                | -1
```

```
(-2, -2, 2)          | [(0, 0), (0, 1)]            | 1
(-2, -2, 4)          | [[(0, 0)], [(0, 1)]]        | -11/12
(-2, -2, 4)          | [(0, 0), (0, 1)]            | 13/8
(-2, 0, 0)           | [(0, 0)]                    | 1
(-2, 0, 0, 0)        | [(0, 0)]                    | -1
(-2, 0, 2)           | [(0, 0)]                    | 1/8
(-2, 1, 1)           | [(0, 0)]                    | 0
(-2, 2)              | [(0, 0)]                    | -1/8
(-2, 4)              | [(0, 0)]                    | -23/1152
(0,)                 | ()                          | 1/24
[(0,), (-2, 0, 0)]   | [(1, 0)]                    | -1/24
[(0,), (0,)]         | ()                          | -1/576
[(0,), (0, 0)]       | ()                          | 0
(0, 0)               | ()                          | 0
(0, 0, 0)            | ()                          | 0
(0, 2)               | ()                          | 0
(1, 1)               | ()                          | 0
(2,)                 | ()                          | -1/640
(4,)                 | ()                          | 305/580608
sage: print_adm_evals()
Stratum              | Psis                        | eval
----------------------------------------------------------------
(0,)                 | {1: 1}                      | 1/24
(-2, 2)              | {1: 1}                      | 1/8
(-2, 0, 0, 0)        | {1: 1}                      | 1
(-2, -2, -2, 4)      | {1: 1}                      | 1
(-2, -2, 1, 1)       | {1: 1}                      | 1
(-4, 4)              | {1: 1}                      | 5/8
(-2, -2, 0, 2)       | {1: 1}                      | 1
(0, 0)               | {1: 1, 2: 1}                | 1/24
(-2, -2, 4)          | {1: 1, 2: 1}                | 11/12
(-2, 0, 2)           | {1: 1, 2: 1}                | 1/4
(-2, 1, 1)           | {1: 1, 2: 1}                | 1/6
(0, 0, 0)            | {1: 1, 2: 1, 3: 1}          | 1/12
(-2, 4)              | {1: 1, 2: 2}                | 73/1152
(0, 0, 0)            | {1: 1, 2: 2}                | 1/12
(1, 1)               | {1: 1, 2: 3}                | 1/720
(-2, -2, 4)          | {1: 1, 3: 1}                | 11/12
(0, 0)               | {1: 2}                      | 1/24
(-2, -2, 4)          | {1: 2}                      | 19/24
(-2, 0, 2)           | {1: 2}                      | 1/8
(-2, 1, 1)           | {1: 2}                      | 1/24
(-2, 4)              | {1: 2, 2: 1}                | 97/1152
(1, 1)               | {1: 2, 2: 2}                | 1/720
(2,)                 | {1: 3}                      | 1/1920
(0, 0, 0)            | {1: 3}                      | 1/24
(-2, 4)              | {1: 3}                      | 43/1152
(0, 2)               | {1: 4}                      | 11/1920
(1, 1)               | {1: 4}                      | 1/720
(4,)                 | {1: 5}                      | 13/580608
(-4, 4)              | {2: 1}                      | 5/8
(-2, 2)              | {2: 1}                      | 1/8
(-2, 0, 0, 0)        | {2: 1}                      | 1
```

```
(-2, 1, 1)           | {2: 1, 3: 1}              | 1/6
(-2, 0, 2)           | {2: 2}                   | 1/4
(-2, 1, 1)           | {2: 2}                   | 1/6
(-2, 4)              | {2: 3}                   | 19/1152
(-2, -2, 0, 2)       | {3: 1}                   | 1
(-2, -2, 1, 1)       | {3: 1}                   | 1
(-2, -2, 4)          | {3: 2}                   | 7/24
(-2, -2, -2, 4)      | {4: 1}                   | 1
```

Of course this affects any future calculations:

```
sage: %
CPU times: user 2.52 ms, sys: 5.39 ms, total: 7.91 ms
Wall time: 22.3 ms
-55/504
```

In fact, the cache was already used in the first calculation, as all levels appear already in the two and three-level graphs (see Remark 3.8).

Example 10.3. Note that calling `euler_characteristic` is simply a frontend for calling the method `euler_char_immediate_evaluation`. Calling this directly, we may use the `quiet=False` option to give extensive output:

```
sage: X=Stratum((2,))
sage: X.euler_char_immediate_evaluation(quiet=False)
Generating enhanced profiles of length 0...
Building all graphs in () (1/1)...
Going through 1 profiles of length 0...
1 / 1, ((), 0): Calculating xi at level 0 (cache) -1/640 Done.
Generating enhanced profiles of length 1...
Building all graphs in (0,) (1/2)...
Building all graphs in (1,) (2/2)...
Going through 2 profiles of length 1...
1 / 2, ((0,), 0): Calculating xi at level 0 (cache) 1/24 level 1 (cache)
    -1/8 Done.
2 / 2, ((1,), 0): Calculating xi at level 0 (cache) 0 Product 0. Done.
Generating enhanced profiles of length 2...
Building all graphs in (0, 1) (1/1)...
Going through 1 profiles of length 2...
1 / 1, ((0, 1), 0): Calculating xi at level 0 (cache) 1/24 level 1 (cache) 1
     level 2 (cache) 1 Done.
Generating enhanced profiles of length 3...
Going through 0 profiles of length 3...
-1/40
```

Example 10.4. Alternatively, we can calculate the Chern character of the logarithmic cotangent bundle using [CMZ20, Thm. 1.2] and use Newton's identity to calculate the Chern polynomial. Of course, this is a longer calculation and there is less caching (`top_xi_at_level` is not called), but it may be used to check the consistency of the formulas:

```
sage: X=Stratum((2,))
sage: X.top_chern_class().evaluate()
1/40
```

```
sage: X.euler_char()
-1/40
```

Note that `euler_char` is simply a frontend for various methods computing the Chern polynomial. More details may be found in the docstrings of the various methods.

10.2. **Crosschecks.** The module `tests` includes some more cross-checks and example computations using `diffstrata`. For example, `leg_tests` tests on each one-dimensional graph Γ of a stratum if the evaluation of the $\xi_\Gamma^{[i]}$ at that level is the same (for every leg!) as the product of Γ with ξ. Note that these expressions can be evaluated and the numbers compared.

The method `commutativity_check` runs an extensive commutativity check on a stratum, i.e. multiplying products of BICs in various orders to give top-level classes and check that these evaluate to the same number, testing the normal bundle and intersection formulas along the way.

Finally, the class `BananaSuite` tests the strata $\Omega\mathcal{M}_1(k, 1, -k-1)$ (cf. [CMZ20, §10.3]) implementing the D-notation introduced there and including a method to test [CMZ20, Prop. 10.2].

Example 10.5. We illustrate the tests on some small strata:

```
sage: leg_test((4,))
Graph ((3, 6, 7, 2), 0): xi evaluated: 1/48 (dim of Level 0: 1)
level: 0, leg: 1, xi ev: 1/48
Graph ((3, 6, 7, 2), 1): xi evaluated: 1/24 (dim of Level 0: 1)
level: 0, leg: 1, xi ev: 1/24
Graph ((3, 6, 5, 2), 0): xi evaluated: 1/48 (dim of Level 0: 1)
level: 0, leg: 1, xi ev: 1/48
Graph ((3, 6, 5, 4), 0): xi evaluated: 1/48 (dim of Level 0: 1)
level: 0, leg: 1, xi ev: 1/48
sage: commutativity_check((2,))
Starting IPs
(0, 0, 0)
0 0
Starting IPs
(0, 0, 1)
0 0
Starting IPs
(0, 1, 0)
0 1
Starting IPs
(0, 1, 1)
0 1
Starting IPs
(1, 0, 0)
1 0
Starting IPs
(1, 0, 1)
1 0
Starting IPs
(1, 1, 0)
1 1
```

```
Starting IPs
(1, 1, 1)
1 1
sage: B=BananaSuite(2)
sage: B.check()
D(1,1)^2 = -1, RHS = -1
D(1,2)^2 = -1, RHS = -1
D(5,1)^2 = -3/2, RHS = -3/2
True
```

References

[BCGGM1]   M. Bainbridge, D. Chen, Q. Gendron, S. Grushevsky, and M. Möller. "Compactification of strata of Abelian differentials". In: *Duke Math. J.* 167.12 (2018), pp. 2347–2416.

[BCGGM3]   M. Bainbridge, D. Chen, Q. Gendron, S. Grushevsky, and M. Möller. *The moduli space of multi-scale differentials.* Preprint. 2019. arXiv: 1910.13492.

[BHPSS20]   Y. Bae, D. Holmes, R. Pandharipande, J. Schmitt, and R Schwarz. *Pixtons formula and Abel-Jacobi theory on the Picard stack.* (2020). arXiv: 2004.08676.

[Cha12]   M. Chan. "Combinatorics of the tropical Torelli map". In: *Algebra Number Theory* 6.6 (2012), pp. 1133–1169.

[CMSZ19]   D. Chen, M. Möller, A. Sauvaget, and D. Zagier. *Masur-Veech volumes and intersection theory on moduli spaces of abelian differentials.* (2019). arXiv: 1901.01785. to appear in Invent. Math.

[CMZ20]   M. Costantini, M. Möller, and J. Zachhuber. *The Chern classes and the Euler characteristic of the moduli spaces of abelian differentials.* (2020). arXiv: 2006.xxxx.

[DSZ20]   V. Delecroix, J. Schmitt, and J. van Zelm. *admcycles – a Sage package for calculations in the tautological ring of the moduli space of stable curves.* (2020). arXiv: 2002.01709.

[FP18]   G. Farkas and R. Pandharipande. "The moduli space of twisted canonical divisors". In: *J. Inst. Math. Jussieu* 17.3 (2018), pp. 615–672.

[HS19]   D. Holmes and J. Schmitt. *Infinitesimal structure of the pluricanonical double ramification locus.* (2019). arXiv: 1909.11981.

[McM14]   C. McMullen. "Moduli spaces in genus zero and inversion of power series". In: *Enseign. Math.* 60.1-2 (2014), pp. 25–30.

[MP11]   S. Maggiolo and N. Pagani. "Generating stable modular graphs". In: *J. Symbolic Comput.* 46.10 (2011), pp. 1087–1097.

[MUW17]   M. Möller, M. Ulirsch, and A. Werner. "Realizability of tropical canonical divisors". In: (2017). arXiv: arXiv:1710.0640. to appear in: J. Eur. Math. Soc.

[SageMath]   The Sage Developers. *SageMath, the Sage Mathematics Software System (Version 9.0).* https://www.sagemath.org. 2020.

[Sau18]   A. Sauvaget. "Volumes and Siegel–Veech constants of $\mathcal{H}(2g-2)$ and Hodge integrals". In: *Geom. Funct. Anal.* 28.6 (2018), pp. 1756–1779.

[Sau19]   A. Sauvaget. "Cohomology classes of strata of differentials". In: *Geom. Topol.* 23.3 (2019), pp. 1085–1171.

[Sch18]   J. Schmitt. "Dimension theory of the moduli space of twisted k-differentials". In: *Doc. Math.* 23 (2018), pp. 871–894.

E-mail address: `costanti@math.uni-bonn.de`

Institut für Mathematik, Universität Bonn, Endenicher Allee 60, 53115 Bonn, Germany

E-mail address: `zachhuber@math.uni-frankfurt.de`

Institut für Mathematik, Goethe-Universität Frankfurt, Robert-Mayer-Str. 6–8, 60325 Frankfurt am Main, Germany

E-mail address: `moeller@math.uni-frankfurt.de`