The Origami Package

Computing Veech groups of origamis

Version 2.0.1

09.07.2024

Simon Ertl
Luca L. Junk
Pascal Kattler
Alexander Rogovskyy
Pascal Schumann
Andrea Thevis
Gabriela Weitze-Schmithüsen

Simon Ertl Email: s8siertl@stud.uni-saarland.de
Homepage: http://www.math.uni-sb.de/ag/weitze/
Address: AG Weitze-Schmithüsen
FR 6.1 Mathematik
Universität des Saarlandes
D-66041 Saarbrücken

Luca L. Junk Email: junk@math.uni-sb.de
Homepage: https://www.uni-saarland.de/lehrstuhl/weber-moritz/team/luca-junk.html
Address: Saarland University
Department of Mathematics
Postfach 15 11 50
66041 Saarbrücken
Germany

Pascal Kattler Email: kattler@math.uni-sb.de
Homepage: http://www.math.uni-sb.de/ag/weitze/
Address: AG Weitze-Schmithüsen
FR 6.1 Mathematik
Universität des Saarlandes
D-66041 Saarbrücken

Alexander Rogovskyy Email: s8alrogo@stud.uni-saarland.de
Homepage: http://www.math.uni-sb.de/ag/weitze/
Address: AG Weitze-Schmithüsen
FR 6.1 Mathematik
Universität des Saarlandes
D-66041 Saarbrücken

Pascal Schumann Email: s8pcschu@stud.uni-saarland.de
Homepage: http://www.math.uni-sb.de/ag/weitze/
Address: AG Weitze-Schmithüsen
FR 6.1 Mathematik
Universität des Saarlandes
D-66041 Saarbrücken

Andrea Thevis Email: thevis@math.uni-frankfurt.de
Homepage: https://www.uni-frankfurt.de/115635174/Dr__Andrea_Thevis
Address: FB 12 - Institut für Mathematik
Johann Wolfgang Goethe-Universität
Robert-Mayer-Str. 6-8
D-60325 Frankfurt am Main

Gabriela Weitze-Schmithüsen Email: weitze@math.uni-sb.de
Homepage: http://www.math.uni-sb.de/ag/weitze/

Address: AG Weitze-Schmithüsen
FR 6.1 Mathematik
Universität des Saarlandes
D-66041 Saarbrücken

Copyright

Acknowledgements

We thank Sergio Siccha for his support and valuable input at the beginning of this project. Furthermore, we would like to thank Vincent Delecroix and Samuel Lelièvre for fruitful discussions about algorithms implemented in the surface_dynamics package in sage. Some of the functionality of the surface_dynamics package can be used in this package as well. For this we use an interface between GAP and sage. We are grateful to Mohamed Barakat and Markus Pfeiffer for helping teaching us how to use these interfaces. Moreover, we thank Thomas Breuer for helpful ideas and comments on some algorithms implemented in this package.

This software package is part of the project I.8 'Algorithmic approaches to Teichmüller curves' (AG Weitze-Schmithüsen, Saarland University) supported by SFB-TRR 195 'Symbolic Tools in Mathematics and their Application' of the German Research Foundation (DFG).

Contents

Chapter 1

Introduction

1.1 Overview

This package provides methods for calculations with certain translation surfaces called origamis. An origami (also known as square-tiled surface) is a finite covering of a torus which is ramified at most over one point. It can be described in the following way from two permutations $\sigma_x, \sigma_y \in S_d$. We take d squares $Q_1, \dots, Q_d$ and glue the lower side of Q_i to the upper side of $Q_{\sigma_y(i)}$ and the right side of Q_i to the left side of $Q_{\sigma_x(i)}$. We require origamis to be connected and thus the group generated by σ_x and σ_y acts transitively on $\{1,\dots,d\}$. In this package we identify an origami with a pair of permutations, which acts transitively on $\{1,\dots,d\}$ up to simultaneous conjugation. This corresponds to renumbering the squares. By choosing a certain numbering in a canonical way one can achieve a canonical representative.

We are especially interested in the so-called Veech group of an origami. This is a finite-index subgroup of $SL_2(\mathbb{Z})$ which carries a lot of information about the geometric and dynamic properties of the underlying translation surface. For further information about origamis and translation surfaces in general see e.g. [Zmi11], [Sch05], [HL06], [Ran17] and [Zor06].

1.2 Notations

Since we are mainly interested in Veech groups of origamis, which are subgroups of $SL_2(\mathbb{Z})$ of finite index, we fix two generators of $SL_2(\mathbb{Z})$

$$S = \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix}$$

and

$$T = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}.$$

Furthermore we fix the free group F generated by $\tilde{S}$ and $\tilde{T}$. We consider the canonical epimorphism $\pi : F \to SL_2(\mathbb{Z})$ with $\pi(\tilde{S}) = S$ and $\pi(\tilde{T}) = T$.

Chapter 2

The functionality of this package

2.1 Basic construction of origamis

2.1.1 Origami

▷ `Origami(`*`sigma_x, sigma_y`*`)` (operation)

Returns: An origami.

This function constructs an `Origami` object from two given permutations σ_x and σ_y. A test is performed to check whether the surface described by the given permutations is connected, i.e. whether the group generated by the two permutations acts transitively.

Example

```
gap> O := Origami((1,2,3,4,5),(5,6));
Origami((1,2,3,4,5), (5,6), 6)
```

▷ `OrigamiNC(`*`sigma_x, sigma_y`*`)` (operation)

Returns: An origami.

This function constructs an `Origami` object from two given permutations σ_x and σ_y without checking whether or not the described surface is connected.

Example

```
gap> O := OrigamiNC((1,2,3,4,5),(5,6));
Origami((1,2,3,4,5), (5,6), 6)
```

▷ `HorizontalPerm(`*`O`*`)` (attribute)

Returns: A permutation.

Returns the permutation σ_x describing the horizontal gluing of the unit squares.

Example

```
gap> O := Origami((1,2,3,4,5),(5,6));
Origami((1,2,3,4,5), (5,6), 6)
gap> HorizontalPerm(O);
(1,2,3,4,5)
```

▷ `VerticalPerm(O)` (attribute)

Returns: A permutation.

Returns the permutation σ_y describing the vertical gluing of the unit squares.

Example
```
gap> O := Origami((1,2,3,4,5),(5,6));
Origami((1,2,3,4,5), (5,6), 6)
gap> VerticalPerm(O);
(5,6)
```

▷ `DegreeOrigami(O)` (attribute)

Returns: A positive integer.

Returns the degree of the origami `O`, i.e. the number of unit squares which are glued together. In terms of permutations, the degree corresponds to the largest moved point of σ_x and σ_y.

Example
```
gap> O := Origami((1,2,3,4,5),(5,6));
Origami((1,2,3,4,5), (5,6), 6)
gap> DegreeOrigami(O);
6
```

2.2 The $SL_2(\mathbb{Z})$-action

The group $SL_2(\mathbb{Z})$ acts on the set of all origamis via the following formulas:

$$S.O(\sigma_x,\sigma_y) = O(\sigma_y^{-1},\sigma_x), \qquad T.O(\sigma_x,\sigma_y) = O(\sigma_x,\sigma_y\sigma_x^{-1})$$

where S and T denote the standard generators

$$S = \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix}, \quad T = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}$$

of $SL_2(\mathbb{Z})$. The succeeding methods implement this action in GAP.

2.2.1 ActionOfS

▷ `ActionOfS(O)` (operation)

Returns: An origami.

For a given origami `O` this methods computes the origami $S.O$ as described above.

Example

```
gap> O:= Origami((1,6,4,7,5,3)(2,8), (1,4,5,3,8,2,6), 8);
Origami((1,6,4,7,5,3)(2,8), (1,4,5,3,8,2,6), 8)
gap> ActionOfS(O);
Origami((1,6,2,8,3,5,4), (1,6,4,7,5,3)(2,8), 8)
```

▷ `ActionOfT(O)` (operation)

Returns: An origami.

For a given origami `O` this methods computes the origami $T.O$ as described above.

Example

```
gap> O:= Origami((1,6,4,7,5,3)(2,8), (1,4,5,3,8,2,6), 8);
Origami((1,6,4,7,5,3)(2,8), (1,4,5,3,8,2,6), 8)
gap> ActionOfT(O);
Origami((1,6,4,7,5,3)(2,8), (1,6,3,2)(4,7), 8)
```

▷ `ActionOfSInv(O)` (operation)

Returns: An origami.

For a given origami `O` this methods computes the origami $S^{-1}.O$.

Example

```
gap> O:= Origami((1,6,4,7,5,3)(2,8), (1,4,5,3,8,2,6), 8);
Origami((1,6,4,7,5,3)(2,8), (1,4,5,3,8,2,6), 8)
gap> ActionOfSInv(O);
Origami((1,4,5,3,8,2,6), (1,3,5,7,4,6)(2,8), 8)
```

▷ `ActionOfTInv(O)` (operation)

Returns: An origami.

For a given origami `O` this methods computes the origami $T^{-1}.O$.

Example

```
gap> O:= Origami((1,6,4,7,5,3)(2,8), (1,4,5,3,8,2,6), 8);
Origami((1,6,4,7,5,3)(2,8), (1,4,5,3,8,2,6), 8)
gap> ActionOfTInv(O);
Origami((1,6,4,7,5,3)(2,8), (1,7,5)(2,4,3), 8)
```

▷ `ActionOfSL2(w, O)` (operation)

Returns: An origami.

For a given origami `O` and a word `w` in S and T (given as a string), this methods computes the origami $w^{-1}.O$.

Example

```
gap> O:= Origami((1,6,4,7,5,3)(2,8), (1,4,5,3,8,2,6), 8);
Origami((1,6,4,7,5,3)(2,8), (1,4,5,3,8,2,6), 8)
gap> ActionOfSL2("S*T", O);
Origami((1,6,2,8,3,5,4), (3,4,7)(5,8,6), 8)
```

2.3 Equivalence of origamis and normal forms

We consider two origamis to be equivalent (or isomorphic), if they are the same up to relabelling the squares. In terms of pairs of permutations, this corresponds to simultaneous conjugation. Equivalence can be checked by computing a certain normal form for origamis.

2.3.1 OrigamiNormalForm

▷ `OrigamiNormalForm(O)` (operation)

Returns: An origami.

Computes a normal form of a given origami. This normal form has the property that two origamis are equivalent if and only if their normal form are equal. Note that this method does not copy any previously computed attributes from the old origami to the new one.

Example

```
gap> O := Origami((1,3,2,4),(2,3));
Origami((1,3,2,4), (2,3), 4)
gap> OrigamiNormalForm(O);
Origami((1,2,3,4), (3,4), 4)
```

▷ `CopyOrigamiInNormalForm(O)` (operation)

Returns: An origami.

This method does the same as `OrigamiNormalForm` but *does copy* previously computed attributes.

▷ `OrigamisEquivalent(O1, O2)` (operation)

Returns: Two origamis.

Checks whether two given origamis are equivalent by comparing their normal forms.

Example

```
gap> P := Origami((1,2,3,4), (3,4), 4);
Origami((1,2,3,4), (3,4), 4)
gap> O := Origami((1,3,2,4),(2,3));
Origami((1,3,2,4), (2,3), 4)
gap> OrigamisEquivalent(O, P);
true
```

2.4 Computing attributes of origamis

2.4.1 Stratum

▷ `Stratum(O)` (attribute)

Returns: A (possibly empty) list of positive integers.

This function calculates the stratum of an origami, i.e. the list of the nonzero degrees of the singularities. For a singularity of cone angle $2k\pi$ the degree of the singularity is $k-1$.

Example

```
    gap> Stratum(Origami((1,6,4,7,5,3)(2,8), (1,4,5,3,8,2,6)));
    [ 1, 5 ]
```

▷ `Genus(O)` (attribute)

Returns: A non-negative integer.

Calculates the genus of an origami.

Example

```
  gap> Genus(Origami((1,2,3,4), (1,2)(3,4)));
  2
```

▷ `IndexOfMonodromyGroup(O)` (attribute)

Returns: A positive integer.

Calculates the index of the monodromy group of an origami, i.e. the index of the subgroup generated by σ_x and σ_y in S_d where d is the degree of the origami.

Example

```
  gap> IndexOfMonodromyGroup(Origami((1,3,6)(4,5),(2,4,3,5)(7,1)));
  1
```

▷ `VeechGroup(O)` (attribute)

Returns: A subgroup of $SL_2(\mathbb{Z})$.

The Veech group of an origami is a finite-index subgroup of $SL_2(\mathbb{Z})$. In GAP it is is represented as a `ModularSubgroup` from the ModularGroup package. It is determined by two permutations σ_S and σ_T describing how the generators S and T of $SL_2(\mathbb{Z})$ act on the cosets. See [Sch04] for a detailed description of the algorithm used to compute the Veech group.

Example

```
  gap> O := Origami((1,2,5)(3,4,6), (1,2)(5,6));
  Origami((1,2,5)(3,4,6), (1,2)(5,6), 6)
  gap> G := VeechGroup(O);
  <modular subgroup of index 10>
  gap> Display(G);
  ModularSubgroup(
  S : ( 1, 2)( 3, 6)( 4, 7)( 5, 8)( 9,10)
  T : ( 1, 3)( 2, 4, 5)( 6, 8,10, 9, 7)
  R : ( 1, 8, 7)( 2, 6)( 3, 4,10, 9, 5)
  J : ( 1, 5, 6)( 2, 3, 7)( 4, 9, 8) )
```

▷ `VeechGroupAndOrbit(`*`O`*`)` (operation)

Returns: A record.

Computes both the Veech group and the orbit under the $SL_2(\mathbb{Z})$ action of a given origami *`O`*. Also returns the matrices with map the origami *`O`* to the corresponding orbit elements. The returned record has the following format:

- *VeechGroup* : the Veech group as a `ModularSubgroup`
- *orbit* : the $SL_2(\mathbb{Z})$-orbit of *`O`* as a list of `Origamis`
- *matrices* : a list of matrices mapping *`O`* to the corresponding orbit elements

▷ `DeckGroup(`*`O`*`)` (attribute)

Returns: A group.

Computes the group of deck transformations of the origami *`O`* as a covering of the once punctured torus. Note that the deck transformations can be seen as permutations of the squares of the origami.

Example

```
gap> O := Origami((1,2,3,4,5), (), 5);
Origami((1,2,3,4,5), (), 5)
gap> DeckGroup(O);
Group([ (), (1,2,3,4,5), (1,3,5,2,4), (1,4,2,5,3), (1,5,4,3,2) ])
```

▷ `IsElementOfDeckGroup(`*`p, O`*`)` (operation)

Returns: A boolean.

Checks whether or not the permutation *`p`* defines a deck transformation of the origami *`O`*.

▷ `IsNormalOrigami(`*`O`*`)` (attribute)

Returns: A boolean.

Checks whether or not a given origami is normal. An origami is called normal if its deck group acts transitively on the fibres of the covering.

Example

```
gap> O := Origami((1,2,3,4,5), (), 5);
Origami((1,2,3,4,5), (), 5)
gap> IsNormalOrigami(O);
true
```

▷ `SymplecticBasisOfHomology(`*`O`*`)` (operation)

Returns: A list of words.

Computes a basis of the homology $H_1(O;\mathbb{Z})$ that is symplectic with respect to the intersection form. The basis elements are represented as words in x and y which define closed paths starting from the square 1.

Example

```
gap> O := Origami((1,2),(2,3));
Origami((1,2), (2,3), 3)
gap> SymplecticBasisOfHomology(O);
[ x*y^2*x^-1, x*y^-2*x^-2*y^2*x^-1, x*y^-2*x^-1*y^-1*x*y^4*x^-1, x*y^-3*x^-1*y*x^-1 ]
```

▷ `SpinStructure(O)` (attribute)

Returns: 0 or 1

Computes the parity of the spin structure of O.

Example

```
gap> O := Origami((1,2,3,4,5), (), 5);
Origami((1,2,3,4,5), (), 5)
gap> SpinStructure(O);
1
```

▷ `SumOfLyapunovExponents(O)` (function)

Returns: A positive Integer.

This function calculates the sum of the positive Lyapunov exponents of the Origami `O`.

Example

```
gap> O:=Origami((1,2,3,4,5,6,7,8,9),(1,3)(2,4,8)(5,7));
Origami((1,2,3,4,5,6,7,8,9), (1,3)(2,4,8)(5,7), 9)
gap> SumOfLyapunovExponents(O);
32/15
```

2.5 Normal Origamis

Normal origamis can equivalently be described by a finite group G and a pair x,y of generators of G. The elements of G correspond to squares and the horizontal (resp. vertical) gluing is determined by left multiplication with x (resp. y), i.e. if $g,h \in G$ then g is glued to h in the x-direction if $xg = h$ and analogously for the y-direction. The group G is then evidently the deck group of the described origami. Furthermore, every normal origami is of this form where G is its deck group. Two normal origamis (x,y,G), (x',y',G) given in such a form are equivalent if and only if there is an automorphism $\varphi \in \mathrm{Aut}(G)$ of G such that $\varphi(x) = x'$ and $\varphi(y) = y'$. We introduce a new GAP-category called `IsNormalStoredOrigami` which is a subcategory of the `Origami`-category for representing normal origamis which are given in terms of the above data. Since every `NormalStoredOrigami` is also an `Origami`, every method implemented for `Origamis` is also available for `NormalStoredOrigamis`.

2.5.1 NormalStoredOrigami

▷ `NormalStoredOrigami(x, y, G)` (operation)

Returns: A normal origami.

For a given finite group `G` and two generators `x, y` this method constructs a new normal origami in the way described above. It is given as a `NormalStoredOrigami` object. This constructor checks whether or not `G` is actually generated by `x` and `y`.

Example
```
gap> G := AlternatingGroup(8);
Alt( [ 1 .. 8 ] )
gap> gen := MinimalGeneratingSet(G);
[ (1,2,3,4,5,6,7), (6,7,8) ]
gap> x := gen[1];; y := gen[2];;
gap> O := NormalStoredOrigami(x, y, G);
Normal Origami( A_8.1^-1*A_8.2*A_8.1 , A_8.1^-1*A_8.2, Group( [ A_8.1, A_8.2 ]) )
```

▷ `NormalStoredOrigamiNC(x, y, G)` (operation)

Returns: A normal origami.

Does the same as the `NormalStoredOrigami` constructor above but does not check whether or not `G` is generated by `x` and `y`. ▷ `HorizontalElement(O)` (attribute)

Returns: A group element.

For a `NormalStoredOrigami`, this method returns the *x*-element which describes the gluing in the horizontal direction. ▷ `VerticalElement(O)` (attribute)

Returns: A group element.

For a `NormalStoredOrigami`, this method returns the *y*-element which describes the gluing in the vertical direction. ▷ `AsPermutationRepresentation(O)` (operation)

Returns: An origami.

Converts a `NormalStoredOrigami` into an `Origami`, i.e. computes the *x*- and *y*-permutations describing the gluing of the squares.

Example
```
gap> G := AlternatingGroup(4);
Alt( [ 1 .. 4 ] )
gap> gen := MinimalGeneratingSet(G);
[ (2,4,3), (1,3)(2,4) ]
gap> x := gen[1];; y := gen[2];;
gap> O := NormalStoredOrigami(x, y, G);
Normal Origami( f1 , f2, Group( [ f1, f2, f3 ] ) )
gap> AsPermutationRepresentation(O);
Origami((1,2,5)(3,6,9)(4,7,10)(8,11,12), (1,3)(2,7)(4,8)(5,12)(6,11)(9,10), 12)
```

▷ `AsPermutationRepresentation(O)` (operation)

Returns: An origami.

Converts a normal `Origami` into a `NormalStoredOrigami`.
▷ `AllNormalOrigamisFromGroup(G)` (operation)

Returns: A list of normal origamis.

For a given group *G*, this method computes all normal origamis (up to equivalence) which can be build from *G*.

Example

```
gap> G := AlternatingGroup(4);
Alt( [ 1 .. 4 ] )
gap> AllNormalOrigamisFromGroup(G);
[ Normal Origami( f3 , f1*f3, Group( [ f1, f2, f3 ] ) ),
Normal Origami( f1 , f3, Group( [ f1, f2, f3 ] ) ),
Normal Origami( f1 , f1*f3, Group( [ f1, f2, f3 ] ) ),
Normal Origami( f1 , f1^2*f3, Group( [ f1, f2, f3 ] ) ) ]
```

▷ `AllNormalOrigamisByDegree(d)` (operation)

Returns: A list of normal origamis.

Computes all normal origamis (up to equivalence) with degree *d*.

Example

```
gap> AllNormalOrigamisByDegree(5);
[ Normal Origami( <identity> of ... , f1, Group( [ f1 ] ) ),
Normal Origami( f1 , <identity> of ..., Group( [ f1 ] ) ),
Normal Origami( f1 , f1, Group( [ f1 ] ) ),
Normal Origami( f1 , f1^2, Group( [ f1 ] ) ),
Normal Origami( f1 , f1^3, Group( [ f1 ] ) ),
Normal Origami( f1 , f1^4, Group( [ f1 ] ) ) ]
```

2.6 Constructing examples of origamis

In this section we describe some methods for constructing explicit examples and families of origamis. More examples of some special constructions of families of orgamis are given in Chapter 7.

2.6.1 AllOrigamisByDegree

▷ `AllOrigamisByDegree(d)` (operation)

Returns: A list of origamis.

Returns a list of all origamis of degree *d* up to equivalence. ▷ `AllOrigamisInStratum(d, stratum)` (operation)

Returns: A list of origamis.

Returns a list of all origamis of degree *d* in the given stratum.

Example

```
gap> OrigamiListInStratum(4, [1,1]);
[ Origami((1,2), (1,3)(2,4), 4), Origami((1,2), (1,3,2,4), 4),
  Origami((1,2)(3,4), (2,3), 4), Origami((1,2)(3,4), (2,3,4), 4),
  Origami((1,2)(3,4), (1,2,3,4), 4), Origami((1,2,3), (2,3,4), 4),
  Origami((1,2,3), (2,4,3), 4), Origami((1,2,3), (1,2)(3,4), 4),
  Origami((1,2,3,4), (2,4), 4), Origami((1,2,3,4), (1,2)(3,4), 4) ]
```

▷ `QuasiRegularOrigami(G, H, r, u)` (operation)

Returns: An origami.

The function calculates an origami by a given group `G` and a subgroup of `G H`. `H` must not contain any normal subgroups of G and `r` and `u` are the generators of G. We identify the tiles of the origami with the right cosets of H in G. The right neighbour of the tile Hg is the tile Hgr and the upper neighbour of Hg is Hgu.

Example
```
    G:=SymmetricGroup(3);
    Sym( [ 1 .. 3 ] )
    gap> H:=Group(());
    Group(())
    gap>QuasiRegularOrigami(G,H,(1,2),(2,3));
    Origami((1,2)(3,5)(4,6), (1,3)(2,4)(5,6), 6)
```

▷ `QROFromGroup(G)` (operation)

Returns: A list of origamis.

Returns a list of all quasi-regular origamis which can be build from the group `G`.

▷ `QROFromOrder(d)` (operation)

Returns: A list of origamis.

Returns a list of all quasi-regular origamis of order `d`. ▷ `NormalformConjugators(O)` (operation)

Returns: A list of permutations.

Calculates all permutations which yield a normal form when conjugated to the `O`.

Example
```
  gap>origami:=Origami((1,2,3,4), (1,2)(3,4));
  Origami((1,2,3,4), (1,2)(3,4), 4)
  gap>NormalformConjugators(origami);
  [ (), (1,3,2), (1,3)(2,4), (1,2,4) ]
```

▷ `TranslationsOfOrigami(O)` (operation)

Returns: A list of permutations.

Computes all translations of the origami as permutations of the squares of `O`. This is equivalent to the permutations t which satisfy $t \circ \sigma_x \circ t^{-1} = \sigma_x$ and $t \circ \sigma_y \circ t^{-1} = \sigma_y$, where σ_x, σ_y are the permutations describing the horizontal and the vertical gluing of the origami, respectively.

Example
```
    gap> origami:=Origami((1,2,3,4), (1,2)(3,4));
   Origami((1,2,3,4), (1,2)(3,4), 4)
   gap> TranslationsOfOrigami(origami);
   [ (), (1,3)(2,4) ]
```

▷ `IsHyperelliptic(O)` (operation)

Returns: A boolean.

Tests whether an origami is hyperelliptic.

Example
```
gap> origami:=Origami((1,2,3,4), (1,2)(3,4));
Origami((1,2,3,4), (1,2)(3,4), 4)
gap> IsHyperelliptic(origami);
true
gap> origami:=Origami((1,2,3,4,5,6), (1,2)(3,4)(5,6));
Origami((1,2,3,4,5,6), (1,2)(3,4)(5,6), 6)
gap> IsHyperelliptic(origami);
false
```

▷ `PointReflectionsOfOrigami(O)` (operation)

Returns: A list of permutations.

Computes all point reflections of O, i.e. all affine homeomorphisms with derivative $-I$, where I is the identity matrix. This is equivalent to the permutations t which satisfy $t \circ \sigma_x \circ t^{-1} = \sigma_x^{-1}$ and $t \circ \sigma_y \circ t^{-1} = \sigma_y^{-1}$, where σ_x, σ_y are the permutations describing the horizontal and the vertical gluing of the origami, respectively.

Example
```
gap> o:=Origami((1,2,3,4),(1,2)(3,4));
Origami((1,2,3,4), (1,2)(3,4), 4)
gap> PointReflectionsOfOrigami(o);
[ (1,2)(3,4), (1,4)(2,3) ]
```

▷ `AutomorphismsOfOrigami(O)` (operation)

Returns: Two lists consisting of two entries: The first entry is a list of permutations and the second entry is 1 resp. -1.

Computes all automorphisms of the origami, i.e. the translations and the point reflections. More precisely it returns two lists. The first list contains the translations and is labelel with 1. The second list contains the point reflections and is labeled with -1.

Example
```
gap> o:=Origami((1,2,3,4),(1,2)(3,4));
Origami((1,2,3,4), (1,2)(3,4), 4)
gap> AutomorphismsOfOrigami(o);
[ [ [ (), (1,3)(2,4) ], 1 ], [ [ (1,2)(3,4), (1,4)(2,3) ], -1 ] ]
```

▷ `FixedPointsOfPointReflections(O, sigma)` (operation)

Returns: A list of fixed points. The fixed points are given as lists with three entries.

This function computes the fixed points of a point reflection σ of the origami O given as permutation of its squares. The function does not check whether σ is a point reflection, i.e. whether it satisfies $\sigma \circ \sigma_x \circ \sigma^{-1} = \sigma_x^{-1}$ and $\sigma \circ \sigma_y \circ \sigma^{-1} = \sigma_y^{-1}$. The fixed points are returned in a list with entries $[a,b,c]$, where a is the square in which the fixed point occurs and (b,c) are the local coordinates in the square a and for the lower left corners a is a representative of the connected lower left corners.

Example
```

gap> origami:=Origami((1,2,3,4),(1,2)(3,4));
Origami((1,2,3,4), (1,2)(3,4), 4)
gap> sigma:=PointReflectionsOfOrigami(origami)[1];
(1,2)(3,4)
```

```
gap> FixedPointsOfPointReflections(origami, sigma);
[ [ 2, 0, 0.5 ], [ 4, 0, 0.5 ], [ 1, 0.5, 0 ], [ 2, 0.5, 0 ], [ 3, 0.5, 0 ], [ 4, 0.5, 0 ]
```

▷ `FixedPointsOfTranslation(O, sigma)` (operation)

Returns: A list of fixed points. The fixed points are given as lists of vertices.

This function computes the fixed points of a translation σ of the origami O given as permutations of its squares. The function does not check whether σ is a translation, i.e. whether it satisfies $\sigma \circ \sigma_x \circ \sigma^{-1} = \sigma_x$ and $\sigma \circ \sigma_y \circ \sigma^{-1} = \sigma_y$. The fixed points of a translation are always vertices of the squares. Each fixed point is returned as list of vertices, more precisely as the labels of all squares such that the corresponding vertex is the fixed point.

Example
```
      gap> origami:=Origami((1,2,3,4),(1,2)(3,4));
Origami((1,2,3,4), (1,2)(3,4), 4)
gap> TranslationsOfOrigami(origami);
[ (), (1,3)(2,4) ]
gap> sigma:=TranslationsOfOrigami(origami)[2];
(1,3)(2,4)
gap> FixedPointsOfTranslation(origami, sigma);
[ [ 1, 3 ], [ 2, 4 ] ]
```

▷ `FixedPointsOfAutomorphism(O, sigma)` (operation)

Returns: A list of fixed points. The fixed points are presened in the format corresponding of the type, i.e. the format depends on whether the automorphism is a translation or a point reflection.

This function computes the fixed points of an automorphism σ of the origami O. If sigma is a point reflection, the fixed points are returned in a list with entries $[[a,b,c],d]$, where a is the square in which the fixed point occurs, (b,c) are the local coordinates in the square and d is -1. If σ is a translation it is returned as list of vertices labeled with 1.

Example
```
gap> O:=Origami((1,2,3,4),(1,2)(3,4));
Origami((1,2,3,4), (1,2)(3,4), 4)
gap> sigma:=(1,3)(2,4);
(1,3)(2,4)
gap> FixedPointsOfAutomorphism(O,sigma);
[ [ [ 1, 3 ], [ 2, 4 ] ], 1 ]
```

▷ `GenusOfQuotient(O)` (operation)

Returns: An integer.

For a given origami this function computes the genus of the quotient $O/Trans(O)$, where $Trans(O)$ is the translation group.

Example
```
gap> O:=Origami((1,2,3)(4,5,6),(3,4));
```

```
Origami((1,2,3)(4,5,6), (3,4), 6)
gap> GenusOfQuotient(O);
1
```

▷ `OrigamiQuotient(O)` (operation)

Returns: An origami

For a given origami this function computes the quotient origami $O/Trans(O)$, where $Trans(O)$ is the translation group.

Example

```
gap> O:=Origami((1,2,3)(4,5,6),(3,4));
Origami((1,2,3)(4,5,6), (3,4), 6)
gap> OrigamiQuotient(O);
Origami((1,2,3), (), 3)
```

Chapter 3

The homology of origamis and the action of the Veech group.

Important premise: We assume in this section, that all considered origamis have no nontrivial translations. This is equivalent to that the corresponding torus cover has no nontrivial deck transformations.

3.1 The non tautological part of homology

This section contains functions, which compute a basis of the homology $H = H_1(\mathscr{O}, \mathbb{Z})$ of an origami $\mathscr{O}$ with d squares and the action of an affine transformation f of $\mathscr{O}$ on H. The later is the map $f_*: H \to H$. We consider the cellular homology with the squares $Q_1, \dots Q_d$ of the origami as 2-cells. We call σ_i the bottom edge of the square Q_i and ξ_i the left edge of the square Q_i. Then we can represent an element of H as a formal linear combination of the $\sigma's$ and the $\xi's$. In GAP we represent elements of H as elements of $\mathbb{Z}^{2d}$ in the following way: σ_i is represented by e_i and ξ_i is represented by e_{i+d}, where e_i is the i-th. standard basis vector of $\mathbb{Z}^{2d}$. Formal sums of elements in H are represented by the appropriate sums in $\mathbb{Z}^{2d}$. One has to keep in mind that this description is not unique. More precisely the homology H is isomorphic to a subquotient of $\mathbb{Z}^{2d}$.

The homology has a splitting as sum of the *tautological part* H^T and the *non-tautological part* H^{NT} which is equivalent under the action of the affine group $\mathrm{Aff}(\mathscr{O})$ of the origami $\mathscr{O}$. I.e. $H = H^T \oplus H^{NT}$ with H^T and H^{NT} both invariant under the action of $\mathrm{Aff}(\mathscr{O})$. The sum is orthogonal with respect to the intersection form, the dimension of H^T is 2 and the dimension of H^{NT} is $2g-2$. In the part of the package described in this chapter we provide functions which compute the action of the affine group on the homology H and on the non-tautological part H^{NT} and thus obtain the *shadow Veech group*. For more information on this topic see for example [BKK+22].

3.2 Functions for the action on homology

3.2.1 HomologyOrigami

▷ `HomologyOrigami(O)` (operation)

Returns: A list of vectors in $\mathbb{Z}^{2d}$

This function computes a basis of the homology of *O*

Example

```
gap> O := Origami((1,2,3,4,5)(6,7), (1,6,8)(2,7));
```

```
        Origami((1,2,3,4,5)(6,7), (1,6,8)(2,7), 8)
        gap> HomologyOrigami(O);
        [ [ 0, 0, 1, 1, 1, -1, 0, 0, 1, 0, 0, -1, 0, 0, -1, 0 ],
        [ 0, 0, 0, 0, 0, 0, 1, 0, 0, -1, 0, 0, 0, 0, -2, 0 ],
        [ 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, -1, 0, 0, 0, 0, 0 ],
        [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0 ],
        [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0 ],
        [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1 ] ]
```

▷ `NonTautPartOfHomologyOrigami(`*O, H*`)` (operation)

Returns: A list of vectors in $\mathbb{Z}^{2d}$

This function computes a basis of the non-tautological part of the homology *H* of *O*.

Example
```
    gap> O := Origami((1,2,3,4,5)(6,7), (1,6,8)(2,7));
        Origami((1,2,3,4,5)(6,7), (1,6,8)(2,7), 8)
        gap> NonTautPartOfHomologyOrigami(O, HomologyOrigami( O ));
        [ [ 0, 0, 1, 1, 1, -1, 0, -2, 1, 0, 2, -1, 0, 0, -1, -1 ],
        [ 0, 0, 0, 0, 0, 0, 1, -1, 0, -1, 1, 0, 0, 0, -2, 2 ],
        [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, -1 ],
        [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, -1 ] ]
```

▷ `HomologyToString(`*H1*`)` (function)

Returns: a string

This function gives a string representation of the homology vector *H1* in the homology of an origami as a formal linear combination of the $\sigma_i's$ (resp. s) and the $\xi_i's$ (resp. z).

Example
```
    gap> O := Origami((1,2,3,4,5)(6,7), (1,6,8)(2,7));
        Origami((1,2,3,4,5)(6,7), (1,6,8)(2,7), 8)
        gap> H1 := HomologyOrigami(O)[1];
        [ 0, 0, 1, 1, 1, -1, 0, 0, 1, 0, 0, -1, 0, 0, -1, 0 ]
        gap> HomologyToString( H1 );
        "1s_3 + 1s_4 + 1s_5 - 1s_6 + 1z_1 - 1z_4 - 1z_7"
```

▷ `ActionOfMatrixOnHom(`*O, A*`)` (operation)

Returns: a matrix in $Sp_{2g}(\mathbb{Z})$

This function computes the action of an affine homeomorphism with derivative *A* on the homology of *O* w.r.t the basis of the homology computed with `HomologyOrigami`.

Example
```
        gap> O := Origami((1,2,3,4,5)(6,7), (1,6,8)(2,7));
        Origami((1,2,3,4,5)(6,7), (1,6,8)(2,7), 8)
        gap> ActionOfMatrixOnHom(O, [[1,10],[0,1]]);
        [ [ 1, -2, -2, 2, 0, 0 ], [ -5, -11, -2, 2, 5, 0 ],
        [ -5, -14, -3, 4, 5, 10 ], [ -10, -26, -6, 7, 10, 10 ],
        [ -10, -26, -6, 6, 11, 0 ], [ 0, 0, 0, 0, 0, 1 ] ]
```

▷ `ActionOfMatrixOnHom(O, A, H)` (operation)

Returns: a matrix in $Sp_{2g}(\mathbb{Z})$

In this method you can choose a basis `H` of the homology.

Example

```
gap> O := Origami((1,2,3,4,5)(6,7), (1,6,8)(2,7));
Origami((1,2,3,4,5)(6,7), (1,6,8)(2,7), 8)
        gap> H := [ [ 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ],
>[ 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0 ],
>[ 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0 ],
>[ 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 1 ],
>[ 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0 ],
>[ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0 ] ];
[ [ 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ],
[ 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0 ],
[ 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0 ],
[ 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 1 ],
[ 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0 ],
[ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0 ] ]
gap> ActionOfMatrixOnHom(O, [[1,10],[0,1]], H);
[ [ 1, 0, 0, 2, 2, 2 ], [ 0, 1, 0, 5, 5, 0 ], [ 0, 0, 1, 10, 0, 0 ],
[ 0, 0, 0, 1, 0, 0 ], [ 0, 0, 0, 0, 1, 0 ], [ 0, 0, 0, 0, 0, 1 ] ]
```

▷ `ActionOfMatrixOnNonTaut(O, A)` (operation)

Returns: a matrix in $Sp_{2g-2}(\mathbb{Z})$

This function computes the action of an affine homeomorphism with derivative `A` on the non-tautological part of the homology of `O` w.r.t the basis of the homology computed with `HomologyOrigami` and the basis of the non-tautological part of the homology is computed with `NonTautPartOfHomologyOrigami` .

Example

```
gap> O := Origami((1,2,3,4,5)(6,7), (1,6,8)(2,7));
Origami((1,2,3,4,5)(6,7), (1,6,8)(2,7), 8)
gap> ActionOfMatrixOnNonTaut(O, [[1,10],[0,1]]);
[ [ 5, 0, 2, 0 ], [ -1, -9, 2, 5 ], [ -8, 0, -3, 0 ], [ 2, -20, 6, 11 ] ]
```

▷ `ActionOfMatrixOnNonTaut(O, A, H)` (operation)

Returns: a matrix in $Sp_{2g-2}(\mathbb{Z})$

In this method you can choose in addition a basis `H` of the homology.

Example

```
gap> O := Origami((1,2,3,4,5)(6,7), (1,6,8)(2,7));
```

```
Origami((1,2,3,4,5)(6,7), (1,6,8)(2,7), 8)
gap> H := [ [ 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ],
>[ 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0 ],
>[ 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0 ],
>[ 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 1 ],
>[ 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0 ],
>[ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0 ] ];
[ [ 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ],
[ 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0 ],
[ 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0 ],
[ 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 1 ],
[ 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0 ],
[ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0 ] ]
gap> ActionOfMatrixOnNonTaut(O, [[1,10],[0,1]], H);
[ [ 1, 0, -4, -2 ], [ 0, 1, 5, 5 ], [ 0, 0, 1, 0 ], [ 0, 0, 0, 1 ] ]
```

▷ `ActionOfMatrixOnNonTaut(`*O, A, NT, H*`)` (operation)

Returns: a matrix in $Sp_{2g-2}(\mathbb{Z})$

In this method you can choose in addition a basis *H* of the homology as well as a basis *NT* of the non-tautological part of the homology.

Example

```
gap> O := Origami((1,2,3,4,5)(6,7), (1,6,8)(2,7));
Origami((1,2,3,4,5)(6,7), (1,6,8)(2,7), 8)
gap> H := [ [ 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ],
>[ 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0 ],
>[ 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0 ],
>[ 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 1 ],
>[ 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0 ],
>[ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0 ] ];
[ [ 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ],
[ 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0 ],
[ 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0 ],
[ 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 1 ],
[ 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0 ],
[ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0 ] ]
gap> ActionOfMatrixOnNonTaut(O, [[1,10],[0,1]],
>NonTautPartOfHomologyOrigami(O, H), H);
[ [ 1, 0, -4, -2 ], [ 0, 1, 5, 5 ], [ 0, 0, 1, 0 ], [ 0, 0, 0, 1 ] ]
```

▷ `ShadowVeechGroup(`*O*`)` (operation)

Returns: a matrix group $S \subseteq Sp_{2g-2}(\mathbb{Z})$

This function computes the image of the action of the Veech group on the non-tautological part of the homology of *O* . If A is in the Veech group of *O* and f is an affine transformation with derivative A then $f_{*|H^{NT}}: H^{NT} \to H^{NT}$ is in the Shadow Veech group of *O* , encoded as matrix. The basis of the homology is computed with `HomologyOrigami` and the basis of the non-tautological part is computed with `NonTautPartOfHomologyOrigami`.

Example

```
gap> O := Origami((1,2,3,4,5)(6,7), (1,6,8)(2,7));
Origami((1,2,3,4,5)(6,7), (1,6,8)(2,7), 8)
gap> ShadowVeechGroup(O);
#I  RRS defined 152 primary and 196 secondary subgroup generators
<matrix group with 152 generators>
```

▷ `ShadowVeechGroup(O, H)` (operation)

Returns: a matrix group $S \subseteq Sp_{2g-2}(\mathbb{Z})$

In this method you can choose in addition a basis *H* of the homology, where the basis of the non-tautological part is computed with `NonTautPartOfHomologyOrigami`.

Example

```
gap> O := Origami((1,2,3,4,5)(6,7), (1,6,8)(2,7));
Origami((1,2,3,4,5)(6,7), (1,6,8)(2,7), 8)
gap> H := [ [ 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ],
>[ 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0 ],
>[ 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0 ],
>[ 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 1 ],
>[ 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0 ],
>[ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0 ] ];
[ [ 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ],
[ 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0 ],
[ 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0 ],
[ 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 1 ],
[ 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0 ],
[ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0 ] ]
gap> ShadowVeechGroup(O, H);
#I  RRS defined 152 primary and 196 secondary subgroup generators
<matrix group with 152 generators>
```

Chapter 4

SageMath and the surface_dynamics package

The SageMath package *surface_dynamics* by Vincent Delecroix [VD] implements functions to study origamis in Sage. We provide a small interface to use these methods from GAP. An installation of Sage together with the *surface_dynamics* package and the GAP packages HomalgToCAS, IO_ForHomalg, IO and RingsForHomalg are needed.

4.1 SageMath functions

The following descriptions and examples are mainly taken from the manual of [VD].

4.1.1 VeechgroupBySage

▷ `VeechgroupBySage(O)` (function)

Returns: A subgroup of $SL_2(\mathbb{Z})$.

This function executes the SageMath method `veech_group` and returns its result as a GAP object.

▷ `IsHyperellipticBySage(O)` (function)

Returns: A boolean.

Checks whether or not the origami *O* is hyperelliptic. ▷ `IsPrimitiveBySage(O)` (function)

Returns: A boolean.

Checks whether or not the origami *O* is primitive. An origami is called primitive, if it does not cover another origami, or equivalently if the action of the monodromy group has no non-trivial block.

▷ `ReduceBySage(O)` (function)

Returns: An Origami.

Returns a reduced origami isomorphic (up to $SL(2,\mathbb{Q})$ action) to *O*.

Example
```
gap> ReduceBySage(Origami((1,2)(3,4), (1,3,5,6)(2,4) ));
Origami((1,2), (1,3), 6)
```

▷ `AbsolutePeriodGeneratorsBySage(O)` (function)

Returns: A list of lists of integers.

Returns a generating set of the absolute periods of *O*. To each curve on an origami, we can associate its holonomy (that is an element of $\mathbb{Z} \times \mathbb{Z}$). This function returns a generating set of the module generated by holonomies of closed curves.

Example
```
gap> AbsolutePeriodGeneratorsBySage(Origami((1,2,3,4)(5,6), (1,5)(2,6) ));
[ [ 2, 0 ], [ 2, 0 ], [ 0, 1 ], [ 0, 1 ] ]
```

▷ `LatticeOfAbsolutePeriodsBySage(O)` (function)

Returns: A list of lists of integers.

Returns (a,t,u) where $((a,0),(t,u))$ is a standard basis for the lattice of the absolute periods of *O*. The lattice of periods of an origami is the sublattice of $\mathbb{Z}^2$ generated by the holonomy vectors of its saddle connections. Any sublattice of $\mathbb{Z}^2$ has a standard basis consisting of a horizontal vector $(a,0)$ and a non horizontal vector (t,u), where a,t,u are integers satisfying $0 < t < a$ and $0 < t$.

Example
```
gap> LatticeOfAbsolutePeriodsBySage(Origami((1,2)(3,4), (2,3) ));
[ [ 2, 0 ], [ 0, 1 ] ]
```

▷ `OptimalDegreeBySage(O)` (function)

Returns: An integer.

Computes the optimal degree of *O*. The optimal degree of an origami is the degree of the map to the largest torus. Any origami $X \to T$ factors as $i \circ \pi_{opt}$ where i is an isogeny. The optimal degree is the degree of π_{opt}.

Example
```
gap> OptimalDegreeBySage(Origami((1,2)(3,4), (2,3) ));
2
```

▷ `PeriodGeneratorsBySage(O)` (function)

Returns: A list of lists of integer.

Return a list of periods that generate the lattice of periods.

Example
```
gap> PeriodGeneratorsBySage(Origami((1,3,6)(2,5,7)(4),(1,2,4,3,5,6,7) ));
[ [ 2, 0 ], [ 1, 0 ], [ 1, 0 ], [ 0, 2 ], [ -1, 2 ], [ 0, 1 ] ]
```

▷ `WidthsAndHeightsBySage(O)` (function)

Returns: A list of lists of integers.

Return the list of widths and heights of cylinders in *O*.

Example
```
gap> WidthsAndHeightsBySage(Origami((1,2)(3,4),(1,3,5,6)(2,4) ));
[ [ 1, 2 ], [ 2, 2 ] ]
```

▷ `SumOfLyapunovExponentsBySage(O)` (function)

Returns: A rational number.

Computes the sum of Lyapunov exponents for *O*.

Example

```
gap> SumOfLyapunovExponentsBySage(Origami((1,2)(3,4),(2,3) ));
3/2
```

▷ `IntermediateCoversBySage(`*O*`)` (function)

Returns the list of intermediate covers of this origami. Not yet implemented.

Chapter 5

The Origami Database

5.1 Setting up the database

To use the origami database, you need an instance of ArangoDB running on port 8529. After installing ArangoDB, you need to set up a database as follows:

- user: ’origami’
- password: none
- database: ’origami’

In this database, create three collections called *origami_representatives*, *origamis* and *veechgroups*. Please download the data from `https://origami.math.uni-sb.de/` and import the *json*-files into the corresponding collections. All of this can be done in the web interface of ArangoDB, which you can reach under `http://127.0.0.1:8529` (unless you have specified a different endpoint for your local ArangoDB-Server).

5.2 Using the origami database

5.2.1 ConnectToOrigamiDB (ConnectToOrigamiDB)

▷ `ConnectToOrigamiDB()` (function)

Returns: An `ArangoDatabase` object

Connects to the arango database ’origami’ at the endpoint ’http+tcp://127.0.0.1:8529’ with the user ’origami’.

5.2.2 InsertVeechGroupIntoDB (InsertVeechGroupIntoDB)

▷ `InsertVeechGroupIntoDB(G)` (operation)

Returns: An `ArangoDocument`

Takes a `ModularSubgroup` `G` and inserts it into the Veech group table. Only precomputed attributes are inserted. This function does not compute anything new from the group. No test is performed to check whether `G` already exists in the database. The resulting `ArangoDocument` corresponding to the new database entry is returned.

5.2.3 GetVeechGroupDBEntry (GetVeechGroupDBEntry)

▷ `GetVeechGroupDBEntry(G)` (operation)

Returns: An `ArangoDocument`

Takes a `ModularSubgroup` *G* and returns the corresponding database entry or `fail` if *G* is not in the database.

5.2.4 GetVeechGroupsFromDB (GetVeechGroupsFromDB)

▷ `GetVeechGroupsFromDB(constraints)` (operation)

Returns: A list of `ModularSubgroups`

Returns all Veech groups in the database subject to the argument *constraints* which is given in the form of a record.

Example

```
gap> GetVeechGroupsFromDB(rec(index := 30, genus := 0));
[list of modular subgroups]
gap> GetVeechGroupsFromDB(rec(congruence := true, level := 12));
[list of modular subgroups]
gap> GetVeechGroupsFromDB(rec(deficiency := 5));
[list of modular subgroups]
```

5.2.5 UpdateVeechGroupDBEntry (UpdateVeechGroupDBEntry)

▷ `UpdateVeechGroupDBEntry(G)` (operation)

Returns: An `ArangoDocument`

Updates the Veech group entry in the database with newly computed data and returns the corresponding `ArangoDocument`.

5.2.6 RemoveVeechGroupFromDB (RemoveVeechGroupFromDB)

▷ `RemoveVeechGroupFromDB(G)` (operation)

Returns: Nothing.

Removes the Veech group *G* from the database (if present). NOT YET IMPLEMENTED

5.2.7 InsertOrigamiRepresentativeIntoDB (InsertOrigamiRepresentativeIntoDB)

▷ `InsertOrigamiRepresentativeIntoDB(O)` (operation)

Returns: An `ArangoDocument`

Inserts the normal form of the origami *O* into the origami representative list. Inserts only precomputed data. If the Veech group of *O* does not already exist in the database, it is also inserted. This function assumes that no other element of the $SL_2(\mathbb{Z})$ orbit of *O* exists in the representative list, no test is performed to check this.

5.2.8 GetOrigamiOrbitRepresentativeDBEntry (GetOrigamiOrbitRepresentativeDBEntry)

▷ `GetOrigamiOrbitRepresentativeDBEntry(O)` (operation)

Returns: An `ArangoDocument`

Takes an Origami *O* and returns the corresponding database entry or `fail` if *O* is not in the database.

5.2.9 GetOrigamiOrbitRepresentativesFromDB (GetOrigamiOrbitRepresentativesFromDB)

▷ `GetOrigamiOrbitRepresentativesFromDB(constraints)` (operation)

Returns: A list of `Origamis`

Returns all origami orbit representatives in the database subject to the constraints given in the form of the record *constraints*.

Example
```
gap> GetOrigamiOrbitRepresentativesFromDB(rec(stratum := [4]));
[ Origami((1,4,3)(2,5), (1,5,3,4)(2,6), 6) ]
gap> GetOrigamiOrbitRepresentativesFromDB(rec(degree := 5));
[ Origami((1,3,5), (1,3)(2,4,5), 5) ]
```

5.2.10 GetAllOrigamiOrbitRepresentativesFromDB (GetAllOrigamiOrbitRepresentativesFromDB)

▷ `GetAllOrigamiOrbitRepresentativesFromDB()` (operation)

Returns: A list of `Origamis`

Returns all origami orbit representatives in the database. Shorthand for `GetOrigamiOrbitRepresentativesFromDB(rec());`

5.2.11 UpdateOrigamiOrbitRepresentativeDBEntry (UpdateOrigamiOrbitRepresentativeDBEntry)

▷ `UpdateOrigamiOrbitRepresentativeDBEntry(O)` (operation)

Returns: An `ArangoDocument`

Updates the origami orbit representative entry in the database with newly computed data and returns the corresponding `ArangoDocument`. If the origami orbit representative *O* does not exist in the database, it is inserted.

5.2.12 RemoveOrigamiOrbitRepresentativeFromDB (RemoveOrigamiOrbitRepresentativeFromDB)

▷ `RemoveOrigamiOrbitRepresentativeFromDB(O)` (operation)

Returns: Nothing.

Removes the origami orbit representative *O* from the database if present. NOT YET IMPLEMENTED!

5.2.13 InsertOrigamiWithOrbitRepresentativeIntoDB (InsertOrigamiWithOrbitRepresentativeIntoDB)

▷ `InsertOrigamiWithOrbitRepresentativeIntoDB(O, R)` (operation)

Returns: An `ArangoDocument`

Inserts the origami `O` into the origami list with orbit representative `R` which is inserted into the origami orbit representative list if it is not already in there. Returns the `ArangoDocument` corresponding to `O`.

5.2.14 InsertOrigamiIntoDB (InsertOrigamiIntoDB)

▷ `InsertOrigamiIntoDB(O)` (operation)

Returns: An `ArangoDocument`

Inserts an origami `O` into the database. If the Veech group and the orbit of `O` is known, this function checks if there is already a representative of the orbit of `O` in the database. If not, `O` is inserted as the representative of its orbit, if yes, `O` is only inserted into the origami list with a reference to the representative. If the Veech group of `O` is not known, it is inserted as its own representative. This might result in entries in the origami orbit representative list which are in the same orbit.

5.2.15 GetOrigamiDBEntry (GetOrigamiDBEntry)

▷ `GetOrigamiDBEntry(O)` (operation)

Returns: An `ArangoDocument`

Returns the `ArangoDocument` in the origami list corresponding to the origami `O` or fail if does not exist in the database.

5.2.16 GetOrigamiOrbit (GetOrigamiOrbit)

▷ `GetOrigamiOrbit(O)` (operation)

Returns: A list of `Origamis`

Returns all origamis in the database which are in the orbit of `O`.

5.2.17 UpdateRepresentativeOfOrigami (UpdateRepresentativeOfOrigami)

▷ `UpdateRepresentativeOfOrigami(O, R)` (operation)

Returns: Nothing.

Updates the orbit representative of `O` to be `R`.

5.2.18 RemoveOrigamiFromDB (RemoveOrigamiFromDB)

▷ `RemoveOrigamiFromDB(O)` (operation)

Returns: Nothing.

Removes the origami `O` from the database.

Chapter 6

Cyclic Torus Covers

Let T_n be the $n \times n$-Torus. Hence topologically it is torus with n^2 punctures. A *cyclic n-torus cover of degree d* is a normal covering $X \to T_n$ whose Deck-group is cyclic with d elements. We can obtain an origami from each cyclic n-torus cover by appending the map $p_n : T_n \to T_1$, which sends each square to the singular square of the trivial torus. We call such origamis *cyclic torus cover origamis*. The functions described in this chapter were programmed and used in the context of [Rog21] and provide a toolkit to work with these special class of origamis.

6.1 Monodromy vectors and bases of the fundamental group

A *cyclic n-torus cover of degree d* is determined by its monodromy map $m : \pi_1(T_n) \to \mathbb{Z}/d\mathbb{Z}$. Recall that the fundamental group $\pi_1(T_n)$ is a free group in $N = n^2 + 1$ generators. If we choose a basis of $\pi_1(T_n)$, every cyclic torus cover origami can be described as vector in $(\mathbb{Z}/d\mathbb{Z})^N$. We call this vector *monodromy vector with respect to this basis*. There are two bases (which we will call L and S) of the fundamental group of T_n which we will use. The base point is chosen as the midpoint of the lower left square.

The basis L consists of the full horizontal path to the right, the full vertical path upwards and loops around each of the $n^2 - 1$ corner points of the squares (numbered left to right and bottom up; the loop on the most upper right corner excluded) in this order.

The basis S consists of picking a maximal set of slits between squares such that T_n without these is still simply connected. The crossing of one such slit bottom up (for horizontal slits) and left to right (for vertical slits) is then an element of the basis.

Convertion between these two bases is performed with `BaseChangeLToS` (6.3.1)

See [Rog21] for more information regarding the construction of cyclic torus covers and the bases L and S.

6.2 General Cyclic Torus Cover functions

6.2.1 GeneralizedCyclicTorusCover

▷ `GeneralizedCyclicTorusCover(n, d, vslits, hslits)` (function)

Returns: a cyclic torus cover origami

A cyclic torus cover consists of d copies of the trivial origami T_n. Each of the n^2 fields of T_n gets assigned a label from 1 to n^2 row-wise from left to right and bottom up. Let f be a field with the label k in the i-th copy of the cyclic torus cover. Then f's right neighbour's label is determined by determining the usual right neighbour in T_n and its copy is $((i+\texttt{vslits[k]}) \bmod d)$. It works in the same way for the upper neighbour and `hslits`.

Example
```
gap> GeneralizedCyclicTorusCover(2, 2, [1,0,0,0], [0,0,0,0]);
Origami((1,6,5,2)(3,4)(7,8), (1,3)(2,4)(5,7)(6,8), 8)
```

6.2.2 CombOrigami

▷ `CombOrigami(n, x, y)` (function)

Returns: a comb origami, which is a cyclic torus cover of degree 2 specified by a single point $P = (x,y)$

A comb origami is a special cyclic torus cover of degree 2, specified by a single point P on T_n. The coordinates are given in the range $0,..,n-1^2$, where the point $(0,0)$ is located in the lower left corner. P must not be a 2-torsion point, that is, it must not be $(0,0)$, $(n/2,n/2)$, $(n/2,0)$ or $(0,n/2)$. The coordinates are considered modulo n. See [HS07] for more details.

Example
```
gap> CombOrigami(3, 1, 0);
Origami((1,2,3)(4,5,6,13,14,15)(7,8,9)(10,11,12)(16,17,18),
(1,4,7)(2,5,8,11,14,17)(3,6,9)(10,13,16)(12,15,18), 18)
```

6.2.3 SearchForCyclicTorusOrigamiWithVeechGroup

▷ `SearchForCyclicTorusOrigamiWithVeechGroup(n, p, H)` (function)

Returns: a monodromy vector in $(\mathbb{Z}/p\mathbb{Z})^{n^2+1}$ representing a cyclic torus cover origami with respect to the basis L that has H as its Veech group or `false` if no such vector is found.

p must be prime. H must be a congruence subgroup of level p and n must be ≥ 2.

Example
```
gap> S := [ [ 0, -1 ], [ 1, 0 ] ];; T := [ [ 1, 1 ], [ 0, 1 ] ];;
gap> H := ModularSubgroup([S^-2, T*S^-1, T^-1*S^-1]);;
gap> SearchForCyclicTorusOrigamiWithVeechGroup(4, 3, H);
[ 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 1, 0, 1, 0, 0, 1 ]
```

6.2.4 Cyclic torus cover origamis from monodromy vectors

▷ `CyclicTorusCoverOrigamiS(n, d, v)` (function)
▷ `CyclicTorusCoverOrigamiL(n, d, v)` (function)

Returns: a cyclic torus cover origami whose monodromy vector with respect to the basis S (respectively L) is v.

n must be ≥ 2, $d \geq 1$ and $v \in (\mathbb{Z}/d\mathbb{Z})^{n^2+1}$ a vector such that its elements generate $\mathbb{Z}/d\mathbb{Z}$.

Example
```
gap> CyclicTorusCoverOrigamiS(2,2,[1,0,1,0,0]);
Origami((1,2,5,6)(3,4)(7,8), (1,3,5,7)(2,4)(6,8), 8)
```

6.3 Matrices acting on the homology of the n-Torus

Given a homeomorphism $f : T_n \to T_n$, one can consider the induced linear map on the homology of T_n. If we furthermore choose two bases of the homology, we can consider this linear map as a matrix. This section computes some of these matrices.

6.3.1 BaseChangeLToS

▷ `BaseChangeLToS(n)` (function)

Returns: a matrix $M = D_{SL}$ representing a change of basis between the bases L and S.

Returns a matrix corresponding to the change of basis from L to S on the homology of T_n. The matrix has the following property: given any cyclic torus cover origami as a monodromy vector v with respect to the basis S, you may obtain the corresponding monodromy vector with respect to basis L using $v \cdot D_{SL}$.

Example
```
gap> Display(BaseChangeLToS(2));
[ [   0,   1,   1,  -1,   0 ],
  [   0,   0,  -1,   1,   0 ],
  [   1,   0,  -1,   0,   1 ],
  [   0,   0,   1,   0,  -1 ],
  [   0,   1,   0,   0,   1 ] ]
```

6.3.2 TranslationGroupOnHomologyOfTn

▷ `TranslationGroupOnHomologyOfTn(n)` (function)

Returns: the group of translations (as matrices) acting on monodromy vectors with respect to L.

This group of order n^2 is generated by the matrices representing the translation by one square to the right and the translation by one square up with respect to the basis L.

Example
```
gap> Order(TranslationGroupOnHomologyOfTn(3));
9
```

6.3.3 ActionOfTOnHomologyOfTn

▷ `ActionOfTOnHomologyOfTn(n)` (function)

Returns: a matrix representing the action of the generator T of $\mathrm{Sl}_2(\mathbb{Z})$ on the homology of T_n with respect to L.

The generator T (shearing to the right) of $\mathrm{Sl}_2(\mathbb{Z})$ can be viewed as an affine map on T_n if we assume that the lower left corner is fixed. This function returns the corresponding map on the homology as a matrix with respect to L.

Example
```
gap> Display(ActionOfTOnHomologyOfTn(2));
[ [   1,   1,   0,   0,   0 ],
  [   0,   1,   0,   0,   0 ],
  [   0,   0,   1,   0,  -1 ],
  [   0,   0,   0,   1,  -1 ],
  [   0,  -1,   0,   0,  -1 ] ]
```

6.3.4 ActionOfSOnHomologyOfTn

▷ `ActionOfSOnHomologyOfTn(n)` (function)

Returns: a matrix representing the action of the generator S of $\mathrm{Sl}_2(\mathbb{Z})$ on the homology of T_n with respect to L.

The generator S (rotation by $\pi/2$ counterclockwise) of $\mathrm{Sl}_2(\mathbb{Z})$ can be viewed as an affine map on T_n if we assume that the lower left corner is fixed. This function returns the corresponding map on the homology as a matrix with respect to L.

Example

```
gap> Display(ActionOfSOnHomologyOfTn(2));
[ [   0,  -1,   0,   0,   0 ],
  [   1,   0,   0,   0,   0 ],
  [  -1,   0,   1,   0,   0 ],
  [   0,   0,   0,   0,   1 ],
  [  -1,   0,   0,   1,   0 ] ]
```

6.3.5 ActionOfMatrixOnHomologyOfTn

▷ `ActionOfMatrixOnHomologyOfTn(n, A)` (function)

Returns: a matrix representing the action of $A \in \mathrm{Sl}_2(\mathbb{Z})$ on the homology of T_n with respect to L.

Any matrix $A \in \mathrm{Sl}_2(\mathbb{Z})$ can be viewed as an affine map on T_n if we assume that the lower left corner is fixed. A must be in $\mathrm{Sl}_2(\mathbb{Z})$. It is written as a word in the generators S and T of $\mathrm{Sl}_2(\mathbb{Z})$, then the corresponding word with the matrices calculated by `ActionOfTOnHomologyOfTn` (6.3.3) and `ActionOfSOnHomologyOfTn` (6.3.4) is taken and returned.

Example

```
gap> M := [ [ 0, -1 ], [ 1, 1 ] ];; # = S * T
gap> Display(ActionOfMatrixOnHomologyOfTn(2, M));
[ [   0,  -1,   0,   0,   0 ],
  [   1,   1,   0,   0,   0 ],
  [  -1,  -1,   1,   0,  -1 ],
  [   0,  -1,   0,   0,  -1 ],
  [  -1,  -1,   0,   1,  -1 ] ]
```

Chapter 7

More special Origamis

This section lists functions for the construction of some special families of origamis.

7.1 Functions for constructing special origamis

7.1.1 RandomOrigami

▷ `RandomOrigami(d)` (function)

Returns: An Origami

This function returns a random origami of degree d. It is usually used for testing. As it is randomised over its permutations, the probability distribution is not guaranteed to be uniform on the orbits.

7.1.2 XOrigami

▷ `XOrigami(n)` (function)

Returns: A special origami

This function returns special origamis, so called Xorigamis. Xorigamis have degree $2n$. The horizontal permutation is the $2n$-cycle $(1,\dots,2n)$ and the vertical permutation is of the form:

$$(1,2)(3,4)..(2n-1,2n)$$

Example

```
gap> XOrigami(2);
Origami((1,2,3,4), (1,2)(3,4), 4)
```

7.1.3 ElevatorOrigami

▷ `ElevatorOrigami(a, b, d)` (function)

Returns: A special origami

The elevator origami consists of d steps of height b and length a. The last step is connected to the first step.

Example

```
gap>  Elevator(2,0,3);
Origami((1,2)(3,5)(4,6), (1,3)(2,4)(5,6), 6)
```

7.1.4 StaircaseOrigami

▷ `StaircaseOrigami(a, b, d)` (function)

Returns: A special origami

The staircase origami consists of d steps of height b and length a.

Example
```
gap> Staircase(2,0,3);
Origami((1,2)(3,4)(5,6), (2,3)(4,5), 6)
```

References

[BKK^{+}22] Etienne Bonnafoux, Manuel Kany, Pascal Kattler, Carlos Matheus, Rogelio Niño, Manuel Sedano-Mendoza, Ferrán Valdez, and Gabriela Weitze-Schmithüsen. Arithmeticity of the Kontsevich–Zorich monodromies of certain families of square-tiled surfaces, arXiv:2206.06595 [math.DS], 2022. 18

[HL06] Pascal Hubert and Samuel Lelièvre. Prime arithmetic Teichmüller discs in $\mathscr{H}(2)$. *Israel J. Math.*, 2006. 4

[HS07] Frank Herrlich and Gabriela Schmithüsen. A comb of origami curves in the moduli space M_3 with three dimensional closure. *Geom. Dedicata*, 124:69–94, 2007. 31

[Ran17] Anja Randecker. Skript zur Vortragsreihe Unendliche Translationsflächen, 2017. 4

[Rog21] Alexander Rogovskyy. *Origamis, die zyklisch über dem NxN-Torus faktorisieren*. Bachelor's thesis, Universität des Saarlandes, 2021. 30

[Sch04] Gabriela Schmithüsen. An algorithm for finding the Veech group of an origami. *Experiment. Math.*, 2004. 9

[Sch05] Gabriela Schmithüsen. *Veech Groups of Origamis*. PhD thesis, University of Karlsruhe, 2005. 4

[VD] Samuel Lelièvre Vincent Delecroix, Charles Fougeron. surface_dynamics sagemath package version 0.4.0. `https://pypi.org/project/surface_dynamics/`. 23

[Zmi11] David Zmiaikou. *Origamis and permutation groups*. PhD thesis, University Paris-Sud, 2011. 4

[Zor06] Anton Zorich. Flat surfaces. In *Frontiers in number theory, physics, and geometry. I.* Springer, Berlin, 2006. 4

Index