Orbits of the mapping class groups

Computing very large orbits

Kay Magaard and Sergey Shpectorov

University of Birmingham

The 3rd de Brún Workshop, Galway
30 November–10 December 2009

Overview

Overview

Overview

Overview

Overview

Hurwitz loci

By a curve X we mean an irreducible closed complex curve.

Hurwitz loci

By a curve X we mean an irreducible closed complex curve. Topologically, X is an orientable compact surface, i.e. a torus with $g \geq 0$ holes. The parameter g is the genus of X.

Hurwitz loci

By a curve X we mean an irreducible closed complex curve. Topologically, X is an orientable compact surface, i.e. a torus with $g \geq 0$ holes. The parameter g is the genus of X.

Curves of genus g taken up to the natural equivalence are parameterized by the points of the moduli space $\mathcal{M}_g$.

Hurwitz loci

By a curve X we mean an irreducible closed complex curve. Topologically, X is an orientable compact surface, i.e. a torus with $g \geq 0$ holes. The parameter g is the genus of X.

Curves of genus g taken up to the natural equivalence are parameterized by the points of the moduli space $\mathcal{M}_g$. For a fixed finite group G, let $\mathcal{M}_{g,G}$ be the subset of $\mathcal{M}_g$ consisting of all points that correspond to the curves X with $\mathrm{Aut}(X) \geq H \cong G$.

Hurwitz loci

By a curve X we mean an irreducible closed complex curve. Topologically, X is an orientable compact surface, i.e. a torus with $g \geq 0$ holes. The parameter g is the genus of X.

Curves of genus g taken up to the natural equivalence are parameterized by the points of the moduli space $\mathcal{M}_g$. For a fixed finite group G, let $\mathcal{M}_{g,G}$ be the subset of $\mathcal{M}_g$ consisting of all points that correspond to the curves X with $\mathrm{Aut}(X) \geq H \cong G$. (When $g \geq 2$, $|G| \leq 84(g-1)$.)

Hurwitz loci

By a curve X we mean an irreducible closed complex curve. Topologically, X is an orientable compact surface, i.e. a torus with $g \geq 0$ holes. The parameter g is the genus of X.

Curves of genus g taken up to the natural equivalence are parameterized by the points of the moduli space $\mathcal{M}_g$. For a fixed finite group G, let $\mathcal{M}_{g,G}$ be the subset of $\mathcal{M}_g$ consisting of all points that correspond to the curves X with $\mathrm{Aut}(X) \geq H \cong G$. (When $g \geq 2$, $|G| \leq 84(g-1)$.) The set $\mathcal{M}_{g,G}$ is an algebraic set in $\mathcal{M}_g$.

Hurwitz loci

By a curve X we mean an irreducible closed complex curve. Topologically, X is an orientable compact surface, i.e. a torus with $g \geq 0$ holes. The parameter g is the genus of X.

Curves of genus g taken up to the natural equivalence are parameterized by the points of the moduli space $\mathcal{M}_g$. For a fixed finite group G, let $\mathcal{M}_{g,G}$ be the subset of $\mathcal{M}_g$ consisting of all points that correspond to the curves X with $\mathrm{Aut}(X) \geq H \cong G$. (When $g \geq 2$, $|G| \leq 84(g-1)$.) The set $\mathcal{M}_{g,G}$ is an algebraic set in $\mathcal{M}_g$. Its irreducible components are called the Hurwitz loci corresponding to g and G.

Main objective

Main objective

- Study Hurwitz loci in low genus cases by hand and via computational means.

Main objective

- Study Hurwitz loci in low genus cases by hand and via computational means.
- Also, develop those computational means.

Signature and type

Let X be a curve of genus $g \geq 2$ and $G \leq \mathrm{Aut}(X)$.

Signature and type

Let X be a curve of genus $g \geq 2$ and $G \leq \mathrm{Aut}(X)$. The quotient $Y = X/G$ is naturally a curve; its genus g_0 is called the orbit genus.

Signature and type

Let X be a curve of genus $g \geq 2$ and $G \leq \mathrm{Aut}(X)$. The quotient $Y = X/G$ is naturally a curve; its genus g_0 is called the orbit genus.

The natural mapping $\pi : X \to Y$ is a ramified covering with a finite number of branch points $y_1, \ldots, y_r$.

Signature and type

Let X be a curve of genus $g \geq 2$ and $G \leq \mathrm{Aut}(X)$. The quotient $Y = X/G$ is naturally a curve; its genus g_0 is called the orbit genus.

The natural mapping $\pi : X \to Y$ is a ramified covering with a finite number of branch points $y_1, \ldots, y_r$. Let $\pi(x_i) = y_i$ and set $m_i = |G_{x_i}|$.

Signature and type

Let X be a curve of genus $g \geq 2$ and $G \leq \mathrm{Aut}(X)$. The quotient $Y = X/G$ is naturally a curve; its genus g_0 is called the orbit genus.

The natural mapping $\pi : X \to Y$ is a ramified covering with a finite number of branch points $y_1, \ldots, y_r$. Let $\pi(x_i) = y_i$ and set $m_i = |G_{x_i}|$. Then $g_0 - (m_1, \ldots, m_r)$ is called the signature of the action of G on X.

Signature and type

Let X be a curve of genus $g \geq 2$ and $G \leq \mathrm{Aut}(X)$. The quotient $Y = X/G$ is naturally a curve; its genus g_0 is called the orbit genus.

The natural mapping $\pi : X \to Y$ is a ramified covering with a finite number of branch points $y_1, \ldots, y_r$. Let $\pi(x_i) = y_i$ and set $m_i = |G_{x_i}|$. Then $g_0 - (m_1, \ldots, m_r)$ is called the signature of the action of G on X.

Every inertia subgroup G_{x_i} has a distinguished generator g_i. Let $C_i = \{g_i^G\}$.

Signature and type

Let X be a curve of genus $g \geq 2$ and $G \leq \mathrm{Aut}(X)$. The quotient $Y = X/G$ is naturally a curve; its genus g_0 is called the orbit genus.

The natural mapping $\pi : X \to Y$ is a ramified covering with a finite number of branch points $y_1, \ldots, y_r$. Let $\pi(x_i) = y_i$ and set $m_i = |G_{x_i}|$. Then $g_0 - (m_1, \ldots, m_r)$ is called the signature of the action of G on X.

Every inertia subgroup G_{x_i} has a distinguished generator g_i. Let $C_i = \{g_i^G\}$. Then $(C_1, \ldots, C_r)$ is the ramification type of the action.

Signature and type

Let X be a curve of genus $g \geq 2$ and $G \leq \mathrm{Aut}(X)$. The quotient $Y = X/G$ is naturally a curve; its genus g_0 is called the orbit genus.

The natural mapping $\pi : X \to Y$ is a ramified covering with a finite number of branch points $y_1, \ldots, y_r$. Let $\pi(x_i) = y_i$ and set $m_i = |G_{x_i}|$. Then $g_0 - (m_1, \ldots, m_r)$ is called the signature of the action of G on X.

Every inertia subgroup G_{x_i} has a distinguished generator g_i. Let $C_i = \{g_i^G\}$. Then $(C_1, \ldots, C_r)$ is the ramification type of the action.

Signature and type stay constant on each Hurwitz locus, so they are used to distinguish Hurwitz loci.

Fundamental group

Let $\hat{Y} = Y \setminus \{y_1, \ldots, y_r\}$ and $\hat{X} = \pi^{-1}(\hat{Y})$.

Fundamental group

Let $\hat{Y} = Y \setminus \{y_1, \ldots, y_r\}$ and $\hat{X} = \pi^{-1}(\hat{Y})$. Then π induces a normal covering $\hat{X} \to \hat{Y}$, for which G is the group of deck transformations.

Fundamental group

Let $\hat{Y} = Y \setminus \{y_1, \ldots, y_r\}$ and $\hat{X} = \pi^{-1}(\hat{Y})$. Then π induces a normal covering $\hat{X} \to \hat{Y}$, for which G is the group of deck transformations.

Pick base points $x \in \hat{X}$ and $y \in \hat{Y}$, so that $\pi(x) = y$.

Fundamental group

Let $\hat{Y} = Y \setminus \{y_1, \ldots, y_r\}$ and $\hat{X} = \pi^{-1}(\hat{Y})$. Then π induces a normal covering $\hat{X} \to \hat{Y}$, for which G is the group of deck transformations.

Pick base points $x \in \hat{X}$ and $y \in \hat{Y}$, so that $\pi(x) = y$. The factor group $\pi_1(\hat{Y}, y)/\pi^*(\pi_1(\hat{X}, x))$ acts regularly on the fiber $\pi^{-1}(y)$ and this action commutes with the action of G, so $G \cong \pi_1(\hat{Y}, y)/\pi^*(\pi_1(\hat{X}, x))$.

Fundamental group

Let $\hat{Y} = Y \setminus \{y_1, \ldots, y_r\}$ and $\hat{X} = \pi^{-1}(\hat{Y})$. Then π induces a normal covering $\hat{X} \to \hat{Y}$, for which G is the group of deck transformations.

Pick base points $x \in \hat{X}$ and $y \in \hat{Y}$, so that $\pi(x) = y$. The factor group $\pi_1(\hat{Y}, y)/\pi^*(\pi_1(\hat{X}, x))$ acts regularly on the fiber $\pi^{-1}(y)$ and this action commutes with the action of G, so $G \cong \pi_1(\hat{Y}, y)/\pi^*(\pi_1(\hat{X}, x))$.

Thus, we have a natural surjection $\psi : \pi_1(\hat{Y}, y) \to G$ with kernel $\pi^*(\pi_1(\hat{X}, x))$.

Fundamental group

Let $\hat{Y} = Y \setminus \{y_1, \ldots, y_r\}$ and $\hat{X} = \pi^{-1}(\hat{Y})$. Then π induces a normal covering $\hat{X} \to \hat{Y}$, for which G is the group of deck transformations.

Pick base points $x \in \hat{X}$ and $y \in \hat{Y}$, so that $\pi(x) = y$. The factor group $\pi_1(\hat{Y}, y)/\pi^*(\pi_1(\hat{X}, x))$ acts regularly on the fiber $\pi^{-1}(y)$ and this action commutes with the action of G, so $G \cong \pi_1(\hat{Y}, y)/\pi^*(\pi_1(\hat{X}, x))$.

Thus, we have a natural surjection $\psi : \pi_1(\hat{Y}, y) \to G$ with kernel $\pi^*(\pi_1(\hat{X}, x))$. This ψ is defined up to an inner automorphism of G.

Fundamental group

Let $\hat{Y} = Y \setminus \{y_1, \ldots, y_r\}$ and $\hat{X} = \pi^{-1}(\hat{Y})$. Then π induces a normal covering $\hat{X} \to \hat{Y}$, for which G is the group of deck transformations.

Pick base points $x \in \hat{X}$ and $y \in \hat{Y}$, so that $\pi(x) = y$. The factor group $\pi_1(\hat{Y}, y)/\pi^*(\pi_1(\hat{X}, x))$ acts regularly on the fiber $\pi^{-1}(y)$ and this action commutes with the action of G, so $G \cong \pi_1(\hat{Y}, y)/\pi^*(\pi_1(\hat{X}, x))$.

Thus, we have a natural surjection $\psi : \pi_1(\hat{Y}, y) \to G$ with kernel $\pi^*(\pi_1(\hat{X}, x))$. This ψ is defined up to an inner automorphism of G.

Note that Y, $y_1, \ldots, y_r$, y, and ψ identify X and G.

Standard generators

Topologically, $\hat{Y}$ is specified by just two parameters: genus g_0 and the number of punctures r.

Standard generators

Topologically, $\hat{Y}$ is specified by just two parameters: genus g_0 and the number of punctures r. So we can write $\Gamma_{g_0,r} = \pi_1(\hat{Y}, y)$.

Standard generators

Topologically, $\hat{Y}$ is specified by just two parameters: genus g_0 and the number of punctures r. So we can write $\Gamma_{g_0,r} = \pi_1(\hat{Y}, y)$. This group is generated by $2g_0 + r$ elements $\alpha_1, \ldots, \alpha_{g_0}$, $\beta_1, \ldots, \beta_{g_0}$, and $\gamma_1, \ldots, \gamma_r$,

Standard generators

Topologically, $\hat{Y}$ is specified by just two parameters: genus g_0 and the number of punctures r. So we can write $\Gamma_{g_0,r} = \pi_1(\hat{Y}, y)$. This group is generated by $2g_0 + r$ elements $\alpha_1, \ldots, \alpha_{g_0}$, $\beta_1, \ldots, \beta_{g_0}$, and $\gamma_1, \ldots, \gamma_r$, subject to a single relation

$$(*) \quad [\alpha_1, \beta_1] \cdots [\alpha_{g_0}, \beta_{g_0}] \cdot \gamma_1 \cdots \gamma_r = 1.$$

Standard generators

Topologically, $\hat{Y}$ is specified by just two parameters: genus g_0 and the number of punctures r. So we can write $\Gamma_{g_0,r} = \pi_1(\hat{Y}, y)$. This group is generated by $2g_0 + r$ elements $\alpha_1, \ldots, \alpha_{g_0}$, $\beta_1, \ldots, \beta_{g_0}$, and $\gamma_1, \ldots, \gamma_r$, subject to a single relation

$$(*) \quad [\alpha_1, \beta_1] \cdots [\alpha_{g_0}, \beta_{g_0}] \cdot \gamma_1 \cdots \gamma_r = 1.$$

We will call elements as above standard generators, or else a standard tuple of $\Gamma_{g_0,r}$.

Standard generators

Topologically, $\hat{Y}$ is specified by just two parameters: genus g_0 and the number of punctures r. So we can write $\Gamma_{g_0,r} = \pi_1(\hat{Y}, y)$. This group is generated by $2g_0 + r$ elements $\alpha_1, \ldots, \alpha_{g_0}$, $\beta_1, \ldots, \beta_{g_0}$, and $\gamma_1, \ldots, \gamma_r$, subject to a single relation

$$(*) \quad [\alpha_1, \beta_1] \cdots [\alpha_{g_0}, \beta_{g_0}] \cdot \gamma_1 \cdots \gamma_r = 1.$$

We will call elements as above standard generators, or else a standard tuple of $\Gamma_{g_0,r}$.

Note that $\Gamma_{g_0,r}$ has infinitely many standard tuples, but also that homeomorphisms of $\hat{Y}$ fixing y act transitively on standard tuples.

Image in G

Pick a particular standard tuple in $\Gamma_{g_0,r}$ and let $a_1, \ldots, a_{g_0}$, $b_1, \ldots, b_{g_0}$, and $c_1, \ldots, c_r$ be its image in G.

Image in G

Pick a particular standard tuple in $\Gamma_{g_0,r}$ and let $a_1, \ldots, a_{g_0}$, $b_1, \ldots, b_{g_0}$, and $c_1, \ldots, c_r$ be its image in G.

Then:

Image in G

Pick a particular standard tuple in $\Gamma_{g_0,r}$ and let $a_1, \ldots, a_{g_0}$, $b_1, \ldots, b_{g_0}$, and $c_1, \ldots, c_r$ be its image in G.

Then:

- The elements $a_1, \ldots, a_{g_0}$, $b_1, \ldots, b_{g_0}$, and $c_1, \ldots, c_r$ generate G.

Image in G

Pick a particular standard tuple in $\Gamma_{g_0,r}$ and let $a_1, \ldots, a_{g_0}$, $b_1, \ldots, b_{g_0}$, and $c_1, \ldots, c_r$ be its image in G.

Then:

- The elements $a_1, \ldots, a_{g_0}$, $b_1, \ldots, b_{g_0}$, and $c_1, \ldots, c_r$ generate G.
- They satisfy relation $(*)$.

Image in G

Pick a particular standard tuple in $\Gamma_{g_0,r}$ and let $a_1, \ldots, a_{g_0}$, $b_1, \ldots, b_{g_0}$, and $c_1, \ldots, c_r$ be its image in G.

Then:

- The elements $a_1, \ldots, a_{g_0}$, $b_1, \ldots, b_{g_0}$, and $c_1, \ldots, c_r$ generate G.
- They satisfy relation $(*)$.
- Every c_i is contained in the conjugacy class C_i.

Image in G

Pick a particular standard tuple in $\Gamma_{g_0,r}$ and let $a_1, \ldots, a_{g_0}$, $b_1, \ldots, b_{g_0}$, and $c_1, \ldots, c_r$ be its image in G.

Then:

- The elements $a_1, \ldots, a_{g_0}$, $b_1, \ldots, b_{g_0}$, and $c_1, \ldots, c_r$ generate G.
- They satisfy relation $(*)$.
- Every c_i is contained in the conjugacy class C_i.

Also, the tuple $(a_1, \ldots, a_{g_0}, b_1, \ldots, b_{g_0}, c_1, \ldots, c_r)$ specifies ψ. Hence it specifies the Hurwitz locus of X and G

Image in G

Pick a particular standard tuple in $\Gamma_{g_0,r}$ and let $a_1, \ldots, a_{g_0}$, $b_1, \ldots, b_{g_0}$, and $c_1, \ldots, c_r$ be its image in G.

Then:

- The elements $a_1, \ldots, a_{g_0}$, $b_1, \ldots, b_{g_0}$, and $c_1, \ldots, c_r$ generate G.
- They satisfy relation $(*)$.
- Every c_i is contained in the conjugacy class C_i.

Also, the tuple $(a_1, \ldots, a_{g_0}, b_1, \ldots, b_{g_0}, c_1, \ldots, c_r)$ specifies ψ. Hence it specifies the Hurwitz locus of X and G (but it does depend on the choice of the standard tuple in $\Gamma_{g_0,r}$).

Mapping class group

Let $M_{g_0,r}$ be the mapping class group of $\hat{Y}$, that is, the group of all (orientation preserving) homeomorphisms of $\hat{Y}$, taken up to isotopy. That is, $M_{g_0,r} = \mathrm{Homeo}^+(\hat{Y})/\mathrm{Iso}(\hat{Y})$.

Mapping class group

Let $M_{g_0,r}$ be the mapping class group of $\hat{Y}$, that is, the group of all (orientation preserving) homeomorphisms of $\hat{Y}$, taken up to isotopy. That is, $M_{g_0,r} = \mathrm{Homeo}^+(\hat{Y})/\mathrm{Iso}(\hat{Y})$.

It can also be written as $\mathrm{Homeo}^+(\hat{Y})_y/\mathrm{Iso}(\hat{Y})_y$.

Mapping class group

Let $M_{g_0,r}$ be the mapping class group of $\hat{Y}$, that is, the group of all (orientation preserving) homeomorphisms of $\hat{Y}$, taken up to isotopy. That is, $M_{g_0,r} = \mathrm{Homeo}^+(\hat{Y})/\mathrm{Iso}(\hat{Y})$.

It can also be written as $\mathrm{Homeo}^+(\hat{Y})_y/\mathrm{Iso}(\hat{Y})_y$. Note that $\mathrm{Homeo}^+(\hat{Y})_y$ naturally acts on $\Gamma_{g_0,r} = \pi_1(\hat{Y}, y)$ and it induces a regular action on the set of standard tuples.

Mapping class group

Let $M_{g_0,r}$ be the mapping class group of $\hat{Y}$, that is, the group of all (orientation preserving) homeomorphisms of $\hat{Y}$, taken up to isotopy. That is, $M_{g_0,r} = \mathrm{Homeo}^+(\hat{Y})/\mathrm{Iso}(\hat{Y})$.

It can also be written as $\mathrm{Homeo}^+(\hat{Y})_y/\mathrm{Iso}(\hat{Y})_y$. Note that $\mathrm{Homeo}^+(\hat{Y})_y$ naturally acts on $\Gamma_{g_0,r} = \pi_1(\hat{Y}, y)$ and it induces a regular action on the set of standard tuples. Furthermore, $\mathrm{Iso}(\hat{Y})_y$ acts by inner automorphisms.

Mapping class group

Let $M_{g_0,r}$ be the mapping class group of $\hat{Y}$, that is, the group of all (orientation preserving) homeomorphisms of $\hat{Y}$, taken up to isotopy. That is, $M_{g_0,r} = \mathrm{Homeo}^+(\hat{Y})/\mathrm{Iso}(\hat{Y})$.

It can also be written as $\mathrm{Homeo}^+(\hat{Y})_y/\mathrm{Iso}(\hat{Y})_y$. Note that $\mathrm{Homeo}^+(\hat{Y})_y$ naturally acts on $\Gamma_{g_0,r} = \pi_1(\hat{Y}, y)$ and it induces a regular action on the set of standard tuples. Furthermore, $\mathrm{Iso}(\hat{Y})_y$ acts by inner automorphisms. Hence $M_{g_0,r}$ acts regularly on the set of standard tuples taken up to conjugation in $\Gamma_{g_0,r}$.

Mapping class group

Let $M_{g_0,r}$ be the mapping class group of $\hat{Y}$, that is, the group of all (orientation preserving) homeomorphisms of $\hat{Y}$, taken up to isotopy. That is, $M_{g_0,r} = \mathrm{Homeo}^+(\hat{Y})/\mathrm{Iso}(\hat{Y})$.

It can also be written as $\mathrm{Homeo}^+(\hat{Y})_y/\mathrm{Iso}(\hat{Y})_y$. Note that $\mathrm{Homeo}^+(\hat{Y})_y$ naturally acts on $\Gamma_{g_0,r} = \pi_1(\hat{Y}, y)$ and it induces a regular action on the set of standard tuples. Furthermore, $\mathrm{Iso}(\hat{Y})_y$ acts by inner automorphisms. Hence $M_{g_0,r}$ acts regularly on the set of standard tuples taken up to conjugation in $\Gamma_{g_0,r}$.

As always, there is a second (commuting) regular action of $\mathrm{Homeo}^+(\hat{Y})$ and of $M_{g_0,r}$.

Mapping class group

Let $M_{g_0,r}$ be the mapping class group of $\hat{Y}$, that is, the group of all (orientation preserving) homeomorphisms of $\hat{Y}$, taken up to isotopy. That is, $M_{g_0,r} = \mathrm{Homeo}^+(\hat{Y})/\mathrm{Iso}(\hat{Y})$.

It can also be written as $\mathrm{Homeo}^+(\hat{Y})_y/\mathrm{Iso}(\hat{Y})_y$. Note that $\mathrm{Homeo}^+(\hat{Y})_y$ naturally acts on $\Gamma_{g_0,r} = \pi_1(\hat{Y}, y)$ and it induces a regular action on the set of standard tuples. Furthermore, $\mathrm{Iso}(\hat{Y})_y$ acts by inner automorphisms. Hence $M_{g_0,r}$ acts regularly on the set of standard tuples taken up to conjugation in $\Gamma_{g_0,r}$.

As always, there is a second (commuting) regular action of $\mathrm{Homeo}^+(\hat{Y})$ and of $M_{g_0,r}$. Let's call this action dual.

Mapping class orbit

The natural action of the mapping class group $M_{g_0,r}$ on standard tuples in $\Gamma_{g_0,r}$ cannot be transferred to the factor group G

Mapping class orbit

The natural action of the mapping class group $M_{g_0,r}$ on standard tuples in $\Gamma_{g_0,r}$ cannot be transferred to the factor group G (because the kernel $\pi^*(\pi_1(\hat{X}, x)$ may not be invariant under it).

Mapping class orbit

The natural action of the mapping class group $M_{g_0,r}$ on standard tuples in $\Gamma_{g_0,r}$ cannot be transferred to the factor group G (because the kernel $\pi^*(\pi_1(\hat{X}, x)$ may not be invariant under it). Luckily, the dual action can!

Mapping class orbit

The natural action of the mapping class group $M_{g_0,r}$ on standard tuples in $\Gamma_{g_0,r}$ cannot be transferred to the factor group G (because the kernel $\pi^*(\pi_1(\hat{X}, x)$ may not be invariant under it). Luckily, the dual action can! Even better, the dual action is also easier to compute!

Mapping class orbit

The natural action of the mapping class group $M_{g_0,r}$ on standard tuples in $\Gamma_{g_0,r}$ cannot be transferred to the factor group G (because the kernel $\pi^*(\pi_1(\hat{X}, x)$ may not be invariant under it). Luckily, the dual action can! Even better, the dual action is also easier to compute!

The orbit of $(a_1, \ldots, a_{g_0}, b_1, \ldots, b_{g_0}, c_1, \ldots, c_r)$ under the dual action of $\mathrm{Homeo}^+(\hat{Y})_y$ (or of $M_{g_0,r}$, if tuples are taken up to conjugates) is called the mapping class orbit.

Mapping class orbit

The natural action of the mapping class group $M_{g_0,r}$ on standard tuples in $\Gamma_{g_0,r}$ cannot be transferred to the factor group G (because the kernel $\pi^*(\pi_1(\hat{X}, x)$ may not be invariant under it). Luckily, the dual action can! Even better, the dual action is also easier to compute!

The orbit of $(a_1, \ldots, a_{g_0}, b_1, \ldots, b_{g_0}, c_1, \ldots, c_r)$ under the dual action of $\mathrm{Homeo}^+(\hat{Y})_y$ (or of $M_{g_0,r}$, if tuples are taken up to conjugates) is called the mapping class orbit. It consists of the images under ψ of all standard tuples from $\Gamma_{g_0,r}$.

Mapping class orbit

The natural action of the mapping class group $M_{g_0,r}$ on standard tuples in $\Gamma_{g_0,r}$ cannot be transferred to the factor group G (because the kernel $\pi^*(\pi_1(\hat{X}, x)$ may not be invariant under it). Luckily, the dual action can! Even better, the dual action is also easier to compute!

The orbit of $(a_1, \ldots, a_{g_0}, b_1, \ldots, b_{g_0}, c_1, \ldots, c_r)$ under the dual action of $\mathrm{Homeo}^+(\hat{Y})_y$ (or of $M_{g_0,r}$, if tuples are taken up to conjugates) is called the mapping class orbit. It consists of the images under ψ of all standard tuples from $\Gamma_{g_0,r}$.

Most importantly, it uniquely identifies the Hurwitz locus.

Conclusion

To find the the Hurwitz loci in $\mathcal{M}_{g,G}$ we need:

- Determine all possible signatures $g_0 - (m_1, \ldots, m_r)$.

Conclusion

To find the the Hurwitz loci in $\mathcal{M}_{g,G}$ we need:

- Determine all possible signatures $g_0 - (m_1, \ldots, m_r)$.
- For each signature find all possible ramification types $\mathbf{C} = (C_1, \ldots, C_r)$ in G.

Conclusion

To find the the Hurwitz loci in $\mathcal{M}_{g,G}$ we need:

- Determine all possible signatures $g_0 - (m_1, \ldots, m_r)$.
- For each signature find all possible ramification types $\mathbf{C} = (C_1, \ldots, C_r)$ in G.
- For each type $\mathbf{C}$ determine all mapping class orbits on the generating tuples $a_1, \ldots, a_{g_0}, b_1, \ldots, b_{g_0}, c_1, \ldots, c_r$ in G that satisfy the relation $(*)$ and such that $c_i \in C_i$ for all i.

Conclusion

To find the the Hurwitz loci in $\mathcal{M}_{g,G}$ we need:

- Determine all possible signatures $g_0 - (m_1, \ldots, m_r)$.
- For each signature find all possible ramification types $\mathbf{C} = (C_1, \ldots, C_r)$ in G.
- For each type $\mathbf{C}$ determine all mapping class orbits on the generating tuples $a_1, \ldots, a_{g_0}, b_1, \ldots, b_{g_0}, c_1, \ldots, c_r$ in G that satisfy the relation $(*)$ and such that $c_i \in C_i$ for all i.

Note that the first two steps were done by Breuer for all $g \leq 48$ and all possible groups G.

Case of $g_0 = 0$

Our initial implementation (in cooperation with Völklein) was for the case where $g_0 = 0$, that is, $Y = \mathbb{P}^1$.

Case of $g_0 = 0$

Our initial implementation (in cooperation with Völklein) was for the case where $g_0 = 0$, that is, $Y = \mathbb{P}^1$. The reason for this restriction was that the general mapping class group action was not known at the time.

Case of $g_0 = 0$

Our initial implementation (in cooperation with Völklein) was for the case where $g_0 = 0$, that is, $Y = \mathbb{P}^1$. The reason for this restriction was that the general mapping class group action was not known at the time.

If $g_0 = 0$ then:

Case of $g_0 = 0$

Our initial implementation (in cooperation with Völklein) was for the case where $g_0 = 0$, that is, $Y = \mathbb{P}^1$. The reason for this restriction was that the general mapping class group action was not known at the time.

If $g_0 = 0$ then:

- The standard tuple simplifies to just $\gamma_1, \ldots, \gamma_r$.

Case of $g_0 = 0$

Our initial implementation (in cooperation with Völklein) was for the case where $g_0 = 0$, that is, $Y = \mathbb{P}^1$. The reason for this restriction was that the general mapping class group action was not known at the time.

If $g_0 = 0$ then:

- The standard tuple simplifies to just $\gamma_1, \ldots, \gamma_r$.
- The relation $(*)$ simplifies to $\gamma_1 \cdots \gamma_r = 1$.

Case of $g_0 = 0$

Our initial implementation (in cooperation with Völklein) was for the case where $g_0 = 0$, that is, $Y = \mathbb{P}^1$. The reason for this restriction was that the general mapping class group action was not known at the time.

If $g_0 = 0$ then:

- The standard tuple simplifies to just $\gamma_1, \ldots, \gamma_r$.
- The relation $(*)$ simplifies to $\gamma_1 \cdots \gamma_r = 1$.
- The mapping class group $M_{0,r}$ is simply the braid group B_r.

Case of $g_0 = 0$

Our initial implementation (in cooperation with Völklein) was for the case where $g_0 = 0$, that is, $Y = \mathbb{P}^1$. The reason for this restriction was that the general mapping class group action was not known at the time.

If $g_0 = 0$ then:

- The standard tuple simplifies to just $\gamma_1, \ldots, \gamma_r$.
- The relation $(*)$ simplifies to $\gamma_1 \cdots \gamma_r = 1$.
- The mapping class group $M_{0,r}$ is simply the braid group B_r.

The (dual) action of the standard generators (half-twists τ_i) of B_r on the tuple $(c_1, \ldots, c_r)$ is well-known:

Case of $g_0 = 0$

Our initial implementation (in cooperation with Völklein) was for the case where $g_0 = 0$, that is, $Y = \mathbb{P}^1$. The reason for this restriction was that the general mapping class group action was not known at the time.

If $g_0 = 0$ then:

- The standard tuple simplifies to just $\gamma_1, \ldots, \gamma_r$.
- The relation $(*)$ simplifies to $\gamma_1 \cdots \gamma_r = 1$.
- The mapping class group $M_{0,r}$ is simply the braid group B_r.

The (dual) action of the standard generators (half-twists τ_i) of B_r on the tuple $(c_1, \ldots, c_r)$ is well-known:

$$(c_1, \ldots, c_r)^{\tau_i} = (c_1, \ldots, c_{i-1}, c_{i+1}, c_i^{c_{i+1}}, c_{i+2}, \ldots, c_r).$$

A single braid orbit

The call

`BraidOrbit(c);`

computes the braid orbit of the tuple $c = (c_1, \ldots, c_r)$.

A single braid orbit

The call

`BraidOrbit(c);`

computes the braid orbit of the tuple $c = (c_1, \ldots, c_r)$. The group G is a global variable,

A single braid orbit

The call

`BraidOrbit(c);`

computes the braid orbit of the tuple $c = (c_1, \ldots, c_r)$. The group G is a global variable, it must be a permutation group.

A single braid orbit

The call

`BraidOrbit(c);`

computes the braid orbit of the tuple $c = (c_1, \ldots, c_r)$. The group G is a global variable, it must be a permutation group. Also global are r and the action of half-twists.

A single braid orbit

The call

`BraidOrbit(c);`

computes the braid orbit of the tuple $c = (c_1, \ldots, c_r)$. The group G is a global variable, it must be a permutation group. Also global are r and the action of half-twists. The action is given as a tuple of words in a free group F_r.

A single braid orbit

The call

`BraidOrbit(c);`

computes the braid orbit of the tuple $c = (c_1, \ldots, c_r)$. The group G is a global variable, it must be a permutation group. Also global are r and the action of half-twists. The action is given as a tuple of words in a free group F_r. It is enacted via the GAP function `MappedWord`.

A single braid orbit

The call

`BraidOrbit(c);`

computes the braid orbit of the tuple $c = (c_1, \ldots, c_r)$. The group G is a global variable, it must be a permutation group. Also global are r and the action of half-twists. The action is given as a tuple of words in a free group F_r. It is enacted via the GAP function `MappedWord`.

The basic algorithm is simple:

A single braid orbit

The call

`BraidOrbit(c);`

computes the braid orbit of the tuple $c = (c_1, \ldots, c_r)$. The group G is a global variable, it must be a permutation group. Also global are r and the action of half-twists. The action is given as a tuple of words in a free group F_r. It is enacted via the GAP function `MappedWord`.

The basic algorithm is simple: apply to c all generators of B_r to get new tuples; then act on the new tuples and so on, until you stop getting anything new.

Efficiency details

To control the size of the orbit we:

Efficiency details

To control the size of the orbit we: (a) consider tuples up to conjugacy;

Efficiency details

To control the size of the orbit we: (a) consider tuples up to conjugacy; and (b) act by pure braids (optional).

Efficiency details

To control the size of the orbit we: (a) consider tuples up to conjugacy; and (b) act by pure braids (optional).

To minimize access time to a tuple, we use hashing.

Efficiency details

To control the size of the orbit we: (a) consider tuples up to conjugacy; and (b) act by pure braids (optional).

To minimize access time to a tuple, we use hashing. The hash address must be invariant under conjugation, so we used orders of preselected random words in F_r.

Efficiency details

To control the size of the orbit we: (a) consider tuples up to conjugacy; and (b) act by pure braids (optional).

To minimize access time to a tuple, we use hashing. The hash address must be invariant under conjugation, so we used orders of preselected random words in F_r. This method has limitations.

Efficiency details

To control the size of the orbit we: (a) consider tuples up to conjugacy; and (b) act by pure braids (optional).

To minimize access time to a tuple, we use hashing. The hash address must be invariant under conjugation, so we used orders of preselected random words in F_r. This method has limitations.

Hulpke wrote for us a function minimizing a tuple.

Efficiency details

To control the size of the orbit we: (a) consider tuples up to conjugacy; and (b) act by pure braids (optional).

To minimize access time to a tuple, we use hashing. The hash address must be invariant under conjugation, so we used orders of preselected random words in F_r. This method has limitations.

Hulpke wrote for us a function minimizing a tuple. With this, the hash address is formed simply from the entries of the minimal tuple.

Efficiency details

To control the size of the orbit we: (a) consider tuples up to conjugacy; and (b) act by pure braids (optional).

To minimize access time to a tuple, we use hashing. The hash address must be invariant under conjugation, so we used orders of preselected random words in F_r. This method has limitations.

Hulpke wrote for us a function minimizing a tuple. With this, the hash address is formed simply from the entries of the minimal tuple.

A further minor improvement: when acting with pure braids, we can fix c_1 in all tuples and do conjugation only in $C(c_1)$.

All braid orbits

The call

`AllBraidOrbits(name, group, type, partition, str)`

computes all braid orbits of a given type.

All braid orbits

The call

$$\texttt{AllBraidOrbits}(\texttt{name}, \texttt{group}, \texttt{type}, \texttt{partition}, \texttt{str})$$

computes all braid orbits of a given type. The orbits are saved in a directory whose name is derived from the project name.

All braid orbits

The call

$$\texttt{AllBraidOrbits}(\texttt{name}, \texttt{group}, \texttt{type}, \texttt{partition}, \texttt{str})$$

computes all braid orbits of a given type. The orbits are saved in a directory whose name is derived from the project name.

Method: generate random tuples $(c_1, \ldots, c_r)$ in G, satisfying $(*)$ and such that $c_i \in C_i$ for all i;

All braid orbits

The call

$$\texttt{AllBraidOrbits}(\texttt{name}, \texttt{group}, \texttt{type}, \texttt{partition}, \texttt{str})$$

computes all braid orbits of a given type. The orbits are saved in a directory whose name is derived from the project name.

Method: generate random tuples $(c_1, \ldots, c_r)$ in G, satisfying $(*)$ and such that $c_i \in C_i$ for all i; then check each new tuple against the known orbits; if in neither,

All braid orbits

The call

$$\texttt{AllBraidOrbits}(\texttt{name}, \texttt{group}, \texttt{type}, \texttt{partition}, \texttt{str})$$

computes all braid orbits of a given type. The orbits are saved in a directory whose name is derived from the project name.

Method: generate random tuples $(c_1, \ldots, c_r)$ in G, satisfying $(*)$ and such that $c_i \in C_i$ for all i; then check each new tuple against the known orbits; if in neither, create the new orbit and add to the list.

All braid orbits

The call

$$\texttt{AllBraidOrbits}(\texttt{name}, \texttt{group}, \texttt{type}, \texttt{partition}, \texttt{str})$$

computes all braid orbits of a given type. The orbits are saved in a directory whose name is derived from the project name.

Method: generate random tuples $(c_1, \ldots, c_r)$ in G, satisfying $(*)$ and such that $c_i \in C_i$ for all i; then check each new tuple against the known orbits; if in neither, create the new orbit and add to the list.

When does it stop?

All braid orbits

The call

$$\texttt{AllBraidOrbits}(\texttt{name}, \texttt{group}, \texttt{type}, \texttt{partition}, \texttt{str})$$

computes all braid orbits of a given type. The orbits are saved in a directory whose name is derived from the project name.

Method: generate random tuples $(c_1, \ldots, c_r)$ in G, satisfying $(*)$ and such that $c_i \in C_i$ for all i; then check each new tuple against the known orbits; if in neither, create the new orbit and add to the list.

When does it stop? The total number of tuples of type $\mathbf{C}$ satisfying $(*)$ is a structure constant and it can be precomputed from the character table of G.

All braid orbits

The call

$$\texttt{AllBraidOrbits}(\texttt{name}, \texttt{group}, \texttt{type}, \texttt{partition}, \texttt{str})$$

computes all braid orbits of a given type. The orbits are saved in a directory whose name is derived from the project name.

Method: generate random tuples $(c_1, \ldots, c_r)$ in G, satisfying $(*)$ and such that $c_i \in C_i$ for all i; then check each new tuple against the known orbits; if in neither, create the new orbit and add to the list.

When does it stop? The total number of tuples of type **C** satisfying $(*)$ is a structure constant and it can be precomputed from the character table of G. The program produces all braid orbits (including the non-generating ones) until the structure constant is exhausted.

Matters of efficiency

The drawback of the random search is that the last small (non-generating!) orbit takes a looong time to show up.

Matters of efficiency

The drawback of the random search is that the last small (non-generating!) orbit takes a looong time to show up. Remedy: maintain the list of subgroups generated by tuples and use it to tilt the random search towards the smaller orbits.

Matters of efficiency

The drawback of the random search is that the last small (non-generating!) orbit takes a looong time to show up. Remedy: maintain the list of subgroups generated by tuples and use it to tilt the random search towards the smaller orbits.

Recent improvement (Völklein-Staszewski): precompute the number of generating tuples using inclusion-exclusion and the list of possible subgroups.

Matters of efficiency

The drawback of the random search is that the last small (non-generating!) orbit takes a looong time to show up. Remedy: maintain the list of subgroups generated by tuples and use it to tilt the random search towards the smaller orbits.

Recent improvement (Völklein-Staszewski): precompute the number of generating tuples using inclusion-exclusion and the list of possible subgroups. Requires automatic generation of the character tables for subgroups.

Matters of efficiency

The drawback of the random search is that the last small (non-generating!) orbit takes a looong time to show up. Remedy: maintain the list of subgroups generated by tuples and use it to tilt the random search towards the smaller orbits.

Recent improvement (Völklein-Staszewski): precompute the number of generating tuples using inclusion-exclusion and the list of possible subgroups. Requires automatic generation of the character tables for subgroups.

A minor performance enhancement: since the orbits are stored externally, we group random tuples in large batches and check each batch against each known orbit in turn. The tuples found to be in the orbit are immediately removed from the batch.

Matters of efficiency

The drawback of the random search is that the last small (non-generating!) orbit takes a looong time to show up. Remedy: maintain the list of subgroups generated by tuples and use it to tilt the random search towards the smaller orbits.

Recent improvement (Völklein-Staszewski): precompute the number of generating tuples using inclusion-exclusion and the list of possible subgroups. Requires automatic generation of the character tables for subgroups.

A minor performance enhancement: since the orbits are stored externally, we group random tuples in large batches and check each batch against each known orbit in turn. The tuples found to be in the orbit are immediately removed from the batch.

Alternative approach by Klüners does not use random search.

Applications of BRAID

(1) Hurwitz loci with "large" groups G.

Applications of BRAID

(1) Hurwitz loci with "large" groups G. The exact condition on the size of G guarantees via Riemann-Hurwitz formula that the orbit genus g_0 is zero. So BRAID can be used.

Applications of BRAID

(1) Hurwitz loci with "large" groups G. The exact condition on the size of G guarantees via Riemann-Hurwitz formula that the orbit genus g_0 is zero. So BRAID can be used. We did the computation up to $g = 10$ or so.

Applications of BRAID

(1) Hurwitz loci with "large" groups G. The exact condition on the size of G guarantees via Riemann-Hurwitz formula that the orbit genus g_0 is zero. So BRAID can be used. We did the computation up to $g = 10$ or so.

(2) Guralnick-Thompson Conjecture and the genus zero systems.

Applications of BRAID

(1) Hurwitz loci with "large" groups G. The exact condition on the size of G guarantees via Riemann-Hurwitz formula that the orbit genus g_0 is zero. So BRAID can be used. We did the computation up to $g = 10$ or so.

(2) Guralnick-Thompson Conjecture and the genus zero systems. Here g_0 is zero by definition, but g can be arbitrary big.

Applications of BRAID

(1) Hurwitz loci with "large" groups G. The exact condition on the size of G guarantees via Riemann-Hurwitz formula that the orbit genus g_0 is zero. So BRAID can be used. We did the computation up to $g = 10$ or so.

(2) Guralnick-Thompson Conjecture and the genus zero systems. Here g_0 is zero by definition, but g can be arbitrary big. Magaard with the help from his students (Badger, Wang) is trying to compile a complete list of exceptional genus zero systems.

Applications of BRAID

(1) Hurwitz loci with "large" groups G. The exact condition on the size of G guarantees via Riemann-Hurwitz formula that the orbit genus g_0 is zero. So BRAID can be used. We did the computation up to $g = 10$ or so.

(2) Guralnick-Thompson Conjecture and the genus zero systems. Here g_0 is zero by definition, but g can be arbitrary big. Magaard with the help from his students (Badger, Wang) is trying to compile a complete list of exceptional genus zero systems. Some of the orbits are huge and beyond what the current program can do.

Arbitrary g_0

To have a program that deals with the case of an arbitrary orbit genus g_0, we first of all need the action of the generators of $M_{g_0,r}$ on the tuples $a_1, \ldots, a_{g_0}, b_1, \ldots, b_{g_0}, c_1, \ldots, c_r$.

Arbitrary g_0

To have a program that deals with the case of an arbitrary orbit genus g_0, we first of all need the action of the generators of $M_{g_0,r}$ on the tuples $a_1, \ldots, a_{g_0}, b_1, \ldots, b_{g_0}, c_1, \ldots, c_r$. we found a suitable list of generators (half-twists and a bunch of Dehn twists in a paper by Labruere and Paris.

Arbitrary g_0

To have a program that deals with the case of an arbitrary orbit genus g_0, we first of all need the action of the generators of $M_{g_0,r}$ on the tuples $a_1, \ldots, a_{g_0}, b_1, \ldots, b_{g_0}, c_1, \ldots, c_r$. we found a suitable list of generators (half-twists and a bunch of Dehn twists in a paper by Labruere and Paris. After that we computed the action of these particular elements on the standard generators of $\pi_1(\hat{Y}, t)$.

Arbitrary g_0

To have a program that deals with the case of an arbitrary orbit genus g_0, we first of all need the action of the generators of $M_{g_0,r}$ on the tuples $a_1, \ldots, a_{g_0}, b_1, \ldots, b_{g_0}, c_1, \ldots, c_r$. we found a suitable list of generators (half-twists and a bunch of Dehn twists in a paper by Labruere and Paris. After that we computed the action of these particular elements on the standard generators of $\pi_1(\hat{Y}, t)$. We also did some obvious computer checks.

Arbitrary g_0

To have a program that deals with the case of an arbitrary orbit genus g_0, we first of all need the action of the generators of $M_{g_0,r}$ on the tuples $a_1, \ldots, a_{g_0}, b_1, \ldots, b_{g_0}, c_1, \ldots, c_r$. we found a suitable list of generators (half-twists and a bunch of Dehn twists in a paper by Labruere and Paris. After that we computed the action of these particular elements on the standard generators of $\pi_1(\hat{Y}, t)$. We also did some obvious computer checks.
The second necessary ingredient is the total number of tuples of the given type in G.

Arbitrary g_0

To have a program that deals with the case of an arbitrary orbit genus g_0, we first of all need the action of the generators of $M_{g_0,r}$ on the tuples $a_1, \ldots, a_{g_0}, b_1, \ldots, b_{g_0}, c_1, \ldots, c_r$. we found a suitable list of generators (half-twists and a bunch of Dehn twists in a paper by Labruere and Paris. After that we computed the action of these particular elements on the standard generators of $\pi_1(\hat{Y}, t)$. We also did some obvious computer checks.
The second necessary ingredient is the total number of tuples of the given type in G. This is not a structure constant!

Arbitrary g_0

To have a program that deals with the case of an arbitrary orbit genus g_0, we first of all need the action of the generators of $M_{g_0,r}$ on the tuples $a_1, \ldots, a_{g_0}, b_1, \ldots, b_{g_0}, c_1, \ldots, c_r$. we found a suitable list of generators (half-twists and a bunch of Dehn twists in a paper by Labruere and Paris. After that we computed the action of these particular elements on the standard generators of $\pi_1(\hat{Y}, t)$. We also did some obvious computer checks.
The second necessary ingredient is the total number of tuples of the given type in G. This is not a structure constant! Luckily, Guralnick in his PhD developed a formula for the number of ways a given element can be written as a product of g_0 commutators.

Arbitrary g_0

To have a program that deals with the case of an arbitrary orbit genus g_0, we first of all need the action of the generators of $M_{g_0,r}$ on the tuples $a_1, \ldots, a_{g_0}, b_1, \ldots, b_{g_0}, c_1, \ldots, c_r$. we found a suitable list of generators (half-twists and a bunch of Dehn twists in a paper by Labruere and Paris. After that we computed the action of these particular elements on the standard generators of $\pi_1(\hat{Y}, t)$. We also did some obvious computer checks.
The second necessary ingredient is the total number of tuples of the given type in G. This is not a structure constant! Luckily, Guralnick in his PhD developed a formula for the number of ways a given element can be written as a product of g_0 commutators.
The rest was a matter of correcting the old programs.

Arbitrary g_0

To have a program that deals with the case of an arbitrary orbit genus g_0, we first of all need the action of the generators of $M_{g_0,r}$ on the tuples $a_1, \ldots, a_{g_0}, b_1, \ldots, b_{g_0}, c_1, \ldots, c_r$. we found a suitable list of generators (half-twists and a bunch of Dehn twists in a paper by Labruere and Paris. After that we computed the action of these particular elements on the standard generators of $\pi_1(\hat{Y}, t)$. We also did some obvious computer checks.
The second necessary ingredient is the total number of tuples of the given type in G. This is not a structure constant! Luckily, Guralnick in his PhD developed a formula for the number of ways a given element can be written as a product of g_0 commutators.
The rest was a matter of correcting the old programs. The two new programs are called, correspondingly, `MappingClassOrbit` and `AllMappingClassOrbits`.

Arbitrary g_0

To have a program that deals with the case of an arbitrary orbit genus g_0, we first of all need the action of the generators of $M_{g_0,r}$ on the tuples $a_1, \ldots, a_{g_0}, b_1, \ldots, b_{g_0}, c_1, \ldots, c_r$. we found a suitable list of generators (half-twists and a bunch of Dehn twists in a paper by Labruere and Paris. After that we computed the action of these particular elements on the standard generators of $\pi_1(\hat{Y}, t)$. We also did some obvious computer checks.
The second necessary ingredient is the total number of tuples of the given type in G. This is not a structure constant! Luckily, Guralnick in his PhD developed a formula for the number of ways a given element can be written as a product of g_0 commutators.
The rest was a matter of correcting the old programs. The two new programs are called, correspondingly, `MappingClassOrbit` and `AllMappingClassOrbits`. They utilize the same ideas and approaches as before, and have the same limitations.

Inclusion of loci

The one notable addition to the package is the function `SubgroupTuple` that, given a standard tuple in a group G arising in its action on X, computes a similar tuple (with different g_0 and r) in a subgroup $H \leq G$, again acting on the same X.

Inclusion of loci

The one notable addition to the package is the function `SubgroupTuple` that, given a standard tuple in a group G arising in its action on X, computes a similar tuple (with different g_0 and r) in a subgroup $H \leq G$, again acting on the same X.

The core algorithm actually computes a standard tuple in a finite index subgroup of the infinite group $\pi_1(\hat{Y}, y)$.

Inclusion of loci

The one notable addition to the package is the function `SubgroupTuple` that, given a standard tuple in a group G arising in its action on X, computes a similar tuple (with different g_0 and r) in a subgroup $H \leq G$, again acting on the same X.

The core algorithm actually computes a standard tuple in a finite index subgroup of the infinite group $\pi_1(\hat{Y}, y)$.

Using this function we can determine, for each Hurwitz locus of G, the corresponding (usually, larger) Hurwitz locus of H.

Inclusion of loci

The one notable addition to the package is the function `SubgroupTuple` that, given a standard tuple in a group G arising in its action on X, computes a similar tuple (with different g_0 and r) in a subgroup $H \leq G$, again acting on the same X.

The core algorithm actually computes a standard tuple in a finite index subgroup of the infinite group $\pi_1(\hat{Y}, y)$.

Using this function we can determine, for each Hurwitz locus of G, the corresponding (usually, larger) Hurwitz locus of H. That is, we can compute the inclusion relation between Hurwitz loci of different groups.

Application

We computed the complete structure of Hurwitz loci for $g \leq 16$.

Application

We computed the complete structure of Hurwitz loci for $g \leq 16$.

Some of that was already known:

Application

We computed the complete structure of Hurwitz loci for $g \leq 16$.

Some of that was already known: Cases $g = 2$ and $g = 3$ were done completely.

Application

We computed the complete structure of Hurwitz loci for $g \leq 16$.

Some of that was already known: Cases $g = 2$ and $g = 3$ were done completely. For $g = 4$, there was a list of Hurwitz loci, but no information about inclusion.

Application

We computed the complete structure of Hurwitz loci for $g \leq 16$.

Some of that was already known: Cases $g = 2$ and $g = 3$ were done completely. For $g = 4$, there was a list of Hurwitz loci, but no information about inclusion. For $g = 5$ there was a list of maximal Hurwitz loci.

Application

We computed the complete structure of Hurwitz loci for $g \leq 16$.

Some of that was already known: Cases $g = 2$ and $g = 3$ were done completely. For $g = 4$, there was a list of Hurwitz loci, but no information about inclusion. For $g = 5$ there was a list of maximal Hurwitz loci.All of that was done "by hand".

Application

We computed the complete structure of Hurwitz loci for $g \leq 16$.

Some of that was already known: Cases $g = 2$ and $g = 3$ were done completely. For $g = 4$, there was a list of Hurwitz loci, but no information about inclusion. For $g = 5$ there was a list of maximal Hurwitz loci.All of that was done "by hand".

We checked our program against these results. We confirmed the results for $g = 2$,

Application

We computed the complete structure of Hurwitz loci for $g \leq 16$.

Some of that was already known: Cases $g = 2$ and $g = 3$ were done completely. For $g = 4$, there was a list of Hurwitz loci, but no information about inclusion. For $g = 5$ there was a list of maximal Hurwitz loci.All of that was done "by hand".

We checked our program against these results. We confirmed the results for $g = 2$, found a missing locus for $g = 3$ (confirmed by a hand computation),

Application

We computed the complete structure of Hurwitz loci for $g \leq 16$.

Some of that was already known: Cases $g = 2$ and $g = 3$ were done completely. For $g = 4$, there was a list of Hurwitz loci, but no information about inclusion. For $g = 5$ there was a list of maximal Hurwitz loci.All of that was done "by hand".

We checked our program against these results. We confirmed the results for $g = 2$, found a missing locus for $g = 3$ (confirmed by a hand computation), confirmed the list for $g = 4$,

Application

We computed the complete structure of Hurwitz loci for $g \leq 16$.

Some of that was already known: Cases $g = 2$ and $g = 3$ were done completely. For $g = 4$, there was a list of Hurwitz loci, but no information about inclusion. For $g = 5$ there was a list of maximal Hurwitz loci.All of that was done "by hand".

We checked our program against these results. We confirmed the results for $g = 2$, found a missing locus for $g = 3$ (confirmed by a hand computation), confirmed the list for $g = 4$, and found a locus for $g = 5$ that is listed as maximal, which in fact is not maximal.

Application

We computed the complete structure of Hurwitz loci for $g \leq 16$.

Some of that was already known: Cases $g = 2$ and $g = 3$ were done completely. For $g = 4$, there was a list of Hurwitz loci, but no information about inclusion. For $g = 5$ there was a list of maximal Hurwitz loci.All of that was done "by hand".

We checked our program against these results. We confirmed the results for $g = 2$, found a missing locus for $g = 3$ (confirmed by a hand computation), confirmed the list for $g = 4$, and found a locus for $g = 5$ that is listed as maximal, which in fact is not maximal.

This has to do with the so-called exceptional loci.

Application

We computed the complete structure of Hurwitz loci for $g \leq 16$.

Some of that was already known: Cases $g = 2$ and $g = 3$ were done completely. For $g = 4$, there was a list of Hurwitz loci, but no information about inclusion. For $g = 5$ there was a list of maximal Hurwitz loci.All of that was done "by hand".

We checked our program against these results. We confirmed the results for $g = 2$, found a missing locus for $g = 3$ (confirmed by a hand computation), confirmed the list for $g = 4$, and found a locus for $g = 5$ that is listed as maximal, which in fact is not maximal.

This has to do with the so-called exceptional loci. These are Hurwitz loci for a group G, which are also (full) Hurwitz loci for a larger group.

Application

We computed the complete structure of Hurwitz loci for $g \leq 16$.

Some of that was already known: Cases $g = 2$ and $g = 3$ were done completely. For $g = 4$, there was a list of Hurwitz loci, but no information about inclusion. For $g = 5$ there was a list of maximal Hurwitz loci.All of that was done "by hand".

We checked our program against these results. We confirmed the results for $g = 2$, found a missing locus for $g = 3$ (confirmed by a hand computation), confirmed the list for $g = 4$, and found a locus for $g = 5$ that is listed as maximal, which in fact is not maximal.

This has to do with the so-called exceptional loci. These are Hurwitz loci for a group G, which are also (full) Hurwitz loci for a larger group. In our search we found a great many examples of such loci.

Application

We computed the complete structure of Hurwitz loci for $g \leq 16$.

Some of that was already known: Cases $g = 2$ and $g = 3$ were done completely. For $g = 4$, there was a list of Hurwitz loci, but no information about inclusion. For $g = 5$ there was a list of maximal Hurwitz loci.All of that was done "by hand".

We checked our program against these results. We confirmed the results for $g = 2$, found a missing locus for $g = 3$ (confirmed by a hand computation), confirmed the list for $g = 4$, and found a locus for $g = 5$ that is listed as maximal, which in fact is not maximal.

This has to do with the so-called exceptional loci. These are Hurwitz loci for a group G, which are also (full) Hurwitz loci for a larger group. In our search we found a great many examples of such loci. In particular, the "non-maximal" locus above is exceptional.

Splitting the orbit

At the moment the program is known to have handled orbits of around half a million of conjugacy classes of tuples. However, the lengths increase rapidly with g. We need much-much longer orbits.

Splitting the orbit

At the moment the program is known to have handled orbits of around half a million of conjugacy classes of tuples. However, the lengths increase rapidly with g. We need much-much longer orbits.

The idea is to cut the curve $\hat{Y}$ across the middle, leaving about half of the punctures and about half of the holes on each side.

Splitting the orbit

At the moment the program is known to have handled orbits of around half a million of conjugacy classes of tuples. However, the lengths increase rapidly with g. We need much-much longer orbits.

The idea is to cut the curve $\hat{Y}$ across the middle, leaving about half of the punctures and about half of the holes on each side. This corresponds to splitting the standard tuple across the middle and introducing a new element "c" for the cut, which is treated as a new puncture on each side.

Splitting the orbit

At the moment the program is known to have handled orbits of around half a million of conjugacy classes of tuples. However, the lengths increase rapidly with g. We need much-much longer orbits.

The idea is to cut the curve $\hat{Y}$ across the middle, leaving about half of the punctures and about half of the holes on each side. This corresponds to splitting the standard tuple across the middle and introducing a new element "c" for the cut, which is treated as a new puncture on each side. The subgroup of the pure mapping class group $pM_{g_0,r}$, which stabilizes the isotopy class of the cut cycle, induces the full pure mapping class group on each side.

Splitting the orbit

At the moment the program is known to have handled orbits of around half a million of conjugacy classes of tuples. However, the lengths increase rapidly with g. We need much-much longer orbits.

The idea is to cut the curve $\hat{Y}$ across the middle, leaving about half of the punctures and about half of the holes on each side. This corresponds to splitting the standard tuple across the middle and introducing a new element "c" for the cut, which is treated as a new puncture on each side. The subgroup of the pure mapping class group $pM_{g_0,r}$, which stabilizes the isotopy class of the cut cycle, induces the full pure mapping class group on each side. So the big orbit splits into pieces, each piece being essentially a product of an orbit from the left side and an orbit from the right side.

Splitting the orbit

At the moment the program is known to have handled orbits of around half a million of conjugacy classes of tuples. However, the lengths increase rapidly with g. We need much-much longer orbits.

The idea is to cut the curve $\hat{Y}$ across the middle, leaving about half of the punctures and about half of the holes on each side. This corresponds to splitting the standard tuple across the middle and introducing a new element "c" for the cut, which is treated as a new puncture on each side. The subgroup of the pure mapping class group $pM_{g_0,r}$, which stabilizes the isotopy class of the cut cycle, induces the full pure mapping class group on each side. So the big orbit splits into pieces, each piece being essentially a product of an orbit from the left side and an orbit from the right side. We can expect those orbits to be on the order of the square root of the length of the big orbit.

Splitting the orbit

At the moment the program is known to have handled orbits of around half a million of conjugacy classes of tuples. However, the lengths increase rapidly with g. We need much-much longer orbits.

The idea is to cut the curve $\hat{Y}$ across the middle, leaving about half of the punctures and about half of the holes on each side. This corresponds to splitting the standard tuple across the middle and introducing a new element "c" for the cut, which is treated as a new puncture on each side. The subgroup of the pure mapping class group $pM_{g_0,r}$, which stabilizes the isotopy class of the cut cycle, induces the full pure mapping class group on each side. So the big orbit splits into pieces, each piece being essentially a product of an orbit from the left side and an orbit from the right side. We can expect those orbits to be on the order of the square root of the length of the big orbit. Of course, we can try to do this splitting repeatedly.

Splitting the orbit

At the moment the program is known to have handled orbits of around half a million of conjugacy classes of tuples. However, the lengths increase rapidly with g. We need much-much longer orbits.

The idea is to cut the curve $\hat{Y}$ across the middle, leaving about half of the punctures and about half of the holes on each side. This corresponds to splitting the standard tuple across the middle and introducing a new element "c" for the cut, which is treated as a new puncture on each side. The subgroup of the pure mapping class group $pM_{g_0,r}$, which stabilizes the isotopy class of the cut cycle, induces the full pure mapping class group on each side. So the big orbit splits into pieces, each piece being essentially a product of an orbit from the left side and an orbit from the right side. We can expect those orbits to be on the order of the square root of the length of the big orbit. Of course, we can try to do this splitting repeatedly.

Putting it back together

We can start by computing the orbits for the two sides and then pairing them up in a meaningful way.

Putting it back together

We can start by computing the orbits for the two sides and then pairing them up in a meaningful way. (Here we definitely need all orbits, not just the generating ones.)

Putting it back together

We can start by computing the orbits for the two sides and then pairing them up in a meaningful way. (Here we definitely need all orbits, not just the generating ones.)

Main question: How can we see which pieces are parts of the same big orbits?

Some obvious things we can do: Apply a missing generator of $M_{g_0,r}$ to a random tuple. Then locate the result within another piece. The two pieces are within the same big orbit.

Putting it back together

We can start by computing the orbits for the two sides and then pairing them up in a meaningful way. (Here we definitely need all orbits, not just the generating ones.)

Main question: How can we see which pieces are parts of the same big orbits?

Some obvious things we can do: Apply a missing generator of $M_{g_0,r}$ to a random tuple. Then locate the result within another piece. The two pieces are within the same big orbit. Do it repeatedly!

Putting it back together

We can start by computing the orbits for the two sides and then pairing them up in a meaningful way. (Here we definitely need all orbits, not just the generating ones.)

Main question: How can we see which pieces are parts of the same big orbits?

Some obvious things we can do: Apply a missing generator of $M_{g_0,r}$ to a random tuple. Then locate the result within another piece. The two pieces are within the same big orbit. Do it repeatedly! If the resulting graph is connected, great! we are done! There is only one orbit.

Putting it back together

We can start by computing the orbits for the two sides and then pairing them up in a meaningful way. (Here we definitely need all orbits, not just the generating ones.)

Main question: How can we see which pieces are parts of the same big orbits?

Some obvious things we can do: Apply a missing generator of $M_{g_0,r}$ to a random tuple. Then locate the result within another piece. The two pieces are within the same big orbit. Do it repeatedly! If the resulting graph is connected, great! we are done! There is only one orbit. What if the graph stubbornly stays disconnected?

Putting it back together

We can start by computing the orbits for the two sides and then pairing them up in a meaningful way. (Here we definitely need all orbits, not just the generating ones.)

Main question: How can we see which pieces are parts of the same big orbits?

Some obvious things we can do: Apply a missing generator of $M_{g_0,r}$ to a random tuple. Then locate the result within another piece. The two pieces are within the same big orbit. Do it repeatedly! If the resulting graph is connected, great! we are done! There is only one orbit. What if the graph stubbornly stays disconnected?

Another approach is to take two different partitions.

Putting it back together

We can start by computing the orbits for the two sides and then pairing them up in a meaningful way. (Here we definitely need all orbits, not just the generating ones.)

Main question: How can we see which pieces are parts of the same big orbits?

Some obvious things we can do: Apply a missing generator of $M_{g_0,r}$ to a random tuple. Then locate the result within another piece. The two pieces are within the same big orbit. Do it repeatedly! If the resulting graph is connected, great! we are done! There is only one orbit. What if the graph stubbornly stays disconnected?

Another approach is to take two different partitions. Then we can take random tuples from one partition and place them with the other one, getting a bipartite graph.

Putting it back together

We can start by computing the orbits for the two sides and then pairing them up in a meaningful way. (Here we definitely need all orbits, not just the generating ones.)

Main question: How can we see which pieces are parts of the same big orbits?

Some obvious things we can do: Apply a missing generator of $M_{g_0,r}$ to a random tuple. Then locate the result within another piece. The two pieces are within the same big orbit. Do it repeatedly! If the resulting graph is connected, great! we are done! There is only one orbit. What if the graph stubbornly stays disconnected?

Another approach is to take two different partitions. Then we can take random tuples from one partition and place them with the other one, getting a bipartite graph. Same questions apply.