17th International Conference on the Application of Computer
Science and Mathematics in Architecture and Civil Engineering
K. Gürlebeck and C. Könke (eds.)
Weimar, Germany, 12–14 July 2006

FILLMORE–SPRINGER–CNOPS CONSTRUCTION IMPLEMENTED IN GINAC

Vladimir V. Kisil

School of Mathematics
University of Leeds
Leeds LS2 9JT
UK
email: kisilv@maths.leeds.ac.uk
Web: http://maths.leeds.ac.uk/~kisilv/

Keywords: Mathematical software, CAS, Clifford algebras, elliptic, parabolic, hyperbolic

Abstract. *This is an implementation of the Fillmore–Springer–Cnops construction (FSCc) [2] based on the Clifford algebra capacities [8] of the* GiNaC *computer algebra system. FSCc linearises the linear-fraction action of the Möbius group. This turns to be very useful in several theoretical and applied fields including engineering.*

The core of this realisation of FSCc is done for an arbitrary dimension, while a subclass for two dimensional cycles add some 2D-specific routines including a visualisation to PostScript files through the `MetaPost` *or* `Asymptote` *software.*

This library is a backbone of many result published in [7], which serve as illustrations of its usage. It can be ported (with various level of required changes) to other CAS with Clifford algebras capabilities.

CONTENTS

1 INTRODUCTION

The usage of Computer Algebra Systems (CAS) for study Clifford algebras has a long history, see for example [3]. Our recent study of geometrical properties in elliptic, parabolic and hyperbolic spaces [7] continues this tradition through an extensive use of computer-assisted proofs and automatically generated graphics. It is our hope that the used technique may be of interest for other researchers as well.

This paper presents an implementation of the Fillmore–Springer–Cnops construction (FSCc) along with illustrations of its usage. FSCc [2, 11] linearises the linear-fraction action of the Möbius group in $\mathbb{R}^n$. This has clear advantages in several theoretical and applied fields including engineering. Our implementation is based on the Clifford algebra capacities of the GiNaC computer algebra system [1], which were described in [8].

The core of this realisation of FSCc is done for an arbitrary dimension of $\mathbb{R}^n$ with a metric given by an arbitrary bilinear form. We also present a subclass for two dimensional cycles (i.e. circles, parabolas and hyperbolas), which add some 2D specific routines including a visualisation to PostScript files through the `MetaPost` [6] or `Asymptote` [5] packages. This software is the backbone of many results published in [7] and we use the full version of its application to [7] for the demonstration purpose in this paper. Actually the present paper and [7] make a companion reading for a fuller understanding of either of them.

There is a Python wrapper [10] for this library which is based on `BoostPython` and `pyGiNaC` packages. The wrapper allows to use all functions and methods from the library in Python scripts or Python interactive shell. The drawing of object from *cycle2d* may be instantly seen in the interactive mode through the `Asymptote`.

It can be ported (with various level of required changes) to other CAS with Clifford algebras capabilities similar to GiNaC.

2 USER INTERFACE TO CLASSES CYCLE AND CYCLE2D

The **cycle class** describes loci of points $\mathbf{x} \in \mathbb{R}^n$ defined by a quadratic equation

$$k\mathbf{x}^2 - 2\langle \mathbf{l}, \mathbf{x} \rangle + m = 0, \qquad \text{where } k, m \in \mathbb{R},\ \mathbf{l} \in \mathbb{R}^n. \tag{1}$$

The class **cycle** correspondingly has member variables *k*, *l*, *m* to describe the equation (1) and the Clifford algebra *unit* to describe the metric of surrounding space. The plenty of methods are supplied for various tasks within FSCc.

We also define a subclass **cycle2D** which has more methods specific to two dimensional environment.

2.1 Constructors of cycle

Here is various constructors for the **cycle**s. The first one takes values of k, $\mathbf{l}$, m as well as *metric* supplied directly. Note that l is admitted either in form of a **lst**, **matrix** or **indexed** objects from GiNaC. Similarly *metric* can be given by an object from either **tensor**, **indexed**, **matrix** or **clifford** classes exactly in the same way as metric is provided for a *clifford_unit*() constructors [8].

4 ⟨cycle class constructors 4⟩≡ (57a) 5a▷

```
public:
cycle(const ex & k1, const ex & l1, const ex & m1,
  const ex & metr = -(new tensdelta)→setflag(status_flags::dynallocated));
```

Defines:
`cycle`, used in chunks 5–11, 13e, 16c, 18–22, 24, 26, 31a, 34–36, 47, 57–67, 69, 70, 73–82, and 86.
Uses `ex` 6d 16b 58c 71 72 96 97a 97b 98a.

Constructor for a **cycle** (1) with $k = 1$ and given *l* defined by the condition that square of its "radius" (which is $\det C$, see [7, Defn. 5.1]) is *r_squared*. Note that for the default value of the *metric* the value of *l* coincides with the centre of this **cycle**.

5a ⟨cycle class constructors 4⟩+≡ (57a) ◁4 5b▷

```
cycle(const lst & l, const ex & metr = -(new tensdelta)→setflag(status_flags::dynallocated),
  const ex & r_squared = 0, const ex & e = 0,
  const ex & sign = (new tensdelta)→setflag(status_flags::dynallocated));
```

Defines:
cycle, used in chunks 5–11, 13e, 16c, 18–22, 24, 26, 31a, 34–36, 47, 57–67, 69, 70, 73–82, and 86.
Uses ex 6d 16b 58c 71 72 96 97a 97b 98a and lst 17b.

If we want to have a cycle identical to to a given one *C* up to a space metric which should be replaced by a new one *metr*, we can use the next constructor.

5b ⟨cycle class constructors 4⟩+≡ (57a) ◁5a 5c▷

```
cycle(const cycle & C, const ex & metr);
```

Uses cycle 4 5a 6c 6c 6c 6c 6c 68a 68b 69c 69c 71 71 71 71 71 and ex 6d 16b 58c 71 72 96 97a 97b 98a.

To any cycle FSCc associates a matrix, which is of the form (2) [7, (3.2)]. The following constructor make a **cycle** from its matrix representation, i.e. it is the realisation of the inverse of the map Q [7, (3.2)].

5c ⟨cycle class constructors 4⟩+≡ (57a) ◁5b

```
cycle(const matrix & M, const ex & metr, const ex & e = 0, const ex & sign = 0);
```

Uses cycle 4 5a 6c 6c 6c 6c 6c 68a 68b 69c 69c 71 71 71 71 71, ex 6d 16b 58c 71 72 96 97a 97b 98a, and matrix 15b 16a.

2.2 Accessing parameters of a cycle

The following set of methods *get_∗*() provide a reading access to the various data in the class.

5d ⟨accessing the data of a cycle 5d⟩≡ (57a) 5e▷

```
public:
 inline ex get_dim() const { return ex_to<idx>(unit.op(1)).get_dim(); }
 inline ex get_metric() const { return ex_to<clifford>(unit).get_metric(); }
 inline ex get_metric(const ex &i0, const ex &i1) const
          { return ex_to<clifford>(unit).get_metric(i0, i1); }
 inline ex get_k() const { return k; }
```

Uses ex 6d 16b 58c 71 72 96 97a 97b 98a.

The member *l* can be obtained as the whole by the call *get_l*(), or its individual component is read, for example, by *get_l*(1).

5e ⟨accessing the data of a cycle 5d⟩+≡ (57a) ◁5d 6a▷

```
inline ex get_l() const { return l; }
inline ex get_l(const ex & i) const { return l.subs(l.op(1) ≡ i, subs_options::no_pattern); }
inline ex get_m() const {return m;}
inline ex get_unit() const {return unit;}
```

Uses ex 6d 16b 58c 71 72 96 97a 97b 98a.

Methods *nops*(), *op*(), *let_op*(), *is_equal*(), *subs*() are standard for expression in GiNaC and described in the GiNaC tutorial. The first three methods are rarely called by a user. In many cases the method *subs*() may replaced by more suitable *subject_to*() 2.4.

6a ⟨accessing the data of a cycle 5d⟩+≡ (57a) ◁5e
```
size_t nops() const {return 4;}
ex op(size_t i) const;
ex & let_op(size_t i);
bool is_equal(const basic & other) const;
bool is_zero() const;
cycle subs(const ex & e, unsigned options = 0) const;
inline cycle normal() const
        { return cycle(k.normal(), l.normal(), m.normal(), unit.normal());}
```

Uses cycle 4 5a 6c 6c 6c 6c 6c 68a 68b 69c 69c 71 71 71 71 71 and ex 6d 16b 58c 71 72 96 97a 97b 98a.

2.3 Linear Operations on Cycles

Cycles are represented by a points in a projective vector space, thus we wish to have a full set of linear operation on them. The metric is inherited from the first **cycle** object. First we define it as an methods of the **cycle** class.

6b ⟨Linear operation as cycle methods 6b⟩≡ (57a)
```
cycle add(const cycle & rh) const;
cycle sub(const cycle & rh) const;
cycle exmul(const ex & rh) const;
cycle div(const ex & rh) const;
```

Uses cycle 4 5a 6c 6c 6c 6c 6c 68a 68b 69c 69c 71 71 71 71 71 and ex 6d 16b 58c 71 72 96 97a 97b 98a.

After that we overload standard binary operations for **cycle**.

6c ⟨Linear operation on cycles 6c⟩≡ (57a) 6d▷
```
const cycle operator+(const cycle & lh, const cycle & rh);
const cycle operator-(const cycle & lh, const cycle & rh);
const cycle operator*(const cycle & lh, const ex & rh);
const cycle operator*(const ex & lh, const cycle & rh);
const cycle operator÷(const cycle & lh, const ex & rh);
```

Defines:
cycle, used in chunks 5–11, 13e, 16c, 18–22, 24, 26, 31a, 34–36, 47, 57–67, 69, 70, 73–82, and 86.
Uses ex 6d 16b 58c 71 72 96 97a 97b 98a.

We also define a product of two cycles through their matrix representation (2).

6d ⟨Linear operation on cycles 6c⟩+≡ (57a) ◁6c
```
const ex operator*(const cycle & lh, const cycle & rh);
```

Defines:
ex, used in chunks 4–12, 14a, 15c, 17a, 28b, 33b, 35, 48, 55–59, 62, 63, 65–68, 70, 73–89, 92b, 93a, 95a, and 98–100.
Uses cycle 4 5a 6c 6c 6c 6c 6c 68a 68b 69c 69c 71 71 71 71 71.

2.4 Geometric methods in cycle

We start from some general methods which deal with **cycle**. The next method is needed to get rid of the homogeneous ambiguity in the projective space of cycles. If *k_new*=0 the **cycle** is normalised such that its det becomes 1. Otherwise the first non-zero coefficient among *k*, *m*, l_0, l_1, ... is set to *k_new*.

7a ⟨specific methods of the class cycle 7a⟩≡ (57a) 7b▷

```
public:
 cycle normalize(const ex & k_new = numeric(1), const ex & e = 0) const;
 cycle normalize_det(const ex & e = 0) const;
```

Uses cycle 4 5a 6c 6c 6c 6c 6c 68a 68b 69c 69c 71 71 71 71 71, ex 6d 16b 58c 71 72 96 97a 97b 98a, and numeric 15a.

The method *center*() returns a list of components of the cycle centre or the corresponding vector (*D*x1 matrix) if the dimension is not symbolic. The metric, if not supplied is taken from the cycle.

7b ⟨specific methods of the class cycle 7a⟩+≡ (57a) ◁7a 7c▷

```
 ex center(const ex & metr = 0, bool return_matrix = false) const; // center of the cycle
```

Uses cycle 4 5a 6c 6c 6c 6c 6c 68a 68b 69c 69c 71 71 71 71 71 and ex 6d 16b 58c 71 72 96 97a 97b 98a.

The next method returns the value of the expression $k\mathbf{x}^2 - 2\langle \mathbf{l}, \mathbf{x}\rangle + m$ for the given cycle and point $\mathbf{x}$. Obviously it should be 0 if $\mathbf{x}$ belongs to the cycle.

7c ⟨specific methods of the class cycle 7a⟩+≡ (57a) ◁7b 7d▷

```
 ex val(const ex & y) const;
```

Uses ex 6d 16b 58c 71 72 96 97a 97b 98a.

Then method *passing*() returns a **relational** defined by the identity $k\mathbf{x}^2 - 2\langle \mathbf{l}, \mathbf{x}\rangle + m \equiv 0$, i.e this relational describes incidence of point to a cycle.

7d ⟨specific methods of the class cycle 7a⟩+≡ (57a) ◁7c 7e▷

```
inline relational passing(const ex & y) const {return val(y).numer()==0; }
```

Uses ex 6d 16b 58c 71 72 96 97a 97b 98a and relational 17b.

We oftenly need to consider a cycle which satisfies some additional conditions, this can be done by the following method *subject_to*. Its typical application looks like:

C1 = *C.subject_to*(**lst**(*C.passing*(*P*), *C.is_orthogonal*(*C1*)));

The second parameters *vars* specifies which components of the **cycle** are considered as unknown. Its default value represents all of them which are symbols.

7e ⟨specific methods of the class cycle 7a⟩+≡ (57a) ◁7d 8a▷

```
 cycle subject_to(const ex & condition, const ex & vars = 0) const;
```

Uses cycle 4 5a 6c 6c 6c 6c 6c 68a 68b 69c 69c 71 71 71 71 71 and ex 6d 16b 58c 71 72 96 97a 97b 98a.

2.5 Methods representing FSCc

There is a set of specific methods which represent mathematical side of FSCc. The next method is the main gateway to the FSCc, it generates the 2×2 matrix

$$\begin{pmatrix} \mathrm{l}_i\sigma^i_j\tilde{e}^j & m \\ k & -\mathrm{l}_i\sigma^i_j\tilde{e}^j \end{pmatrix} \qquad \text{from the cycle } k\mathbf{x}^2 - 2\langle \mathbf{l}, \mathbf{x}\rangle + m = 0. \tag{2}$$

Note, that the Clifford unit $\tilde{e}$ has an arbitrary metric unrelated to the initial metric stored in the *unit* member variable.

8a ⟨specific methods of the class cycle 7a⟩+≡ (57a) ◁7e 8b▷
matrix *to_matrix*(**const ex** & *e* = 0,
const ex & *sign* = (**new** *tensdelta*)→*setflag*(*status_flags*::*dynallocated*)) **const**;

Uses ex 6d 16b 58c 71 72 96 97a 97b 98a and matrix 15b 16a.

The next method returns the value of determinant of the matrix (2) corresponding to the **cycle**. It has explicit geometric meaning, see [7, § 5.1]. Before calculation the cycle is normalised by the condition *k*≡*k_norm*, if *k_norm* is zero then no normalisation is done.

8b ⟨specific methods of the class cycle 7a⟩+≡ (57a) ◁8a 8c▷
ex *det*(**const ex** & *e* = 0,
const ex & *sign* = (**new** *tensdelta*)→*setflag*(*status_flags*::*dynallocated*),
const ex & *k_norm* = **numeric**(1)) **const**;

Uses ex 6d 16b 58c 71 72 96 97a 97b 98a and numeric 15a.

The matrix (2) corresponding to a cycle may be multiplied by another matrix, which in turn may be either generated by another cycle or be of a different origin. The next methods multiplies a cycle by another cycle of matrix supplied in *C*.

8c ⟨specific methods of the class cycle 7a⟩+≡ (57a) ◁8b 8d▷
matrix *mul*(**const ex** & *C*, **const ex** & *e* = 0,
const ex & *sign* = (**new** *tensdelta*)→*setflag*(*status_flags*::*dynallocated*),
const ex & *sign1* =0) **const**;

Uses ex 6d 16b 58c 71 72 96 97a 97b 98a and matrix 15b 16a.

Having a matrix C which represents a cycle and another matrix M we can consider a similar matrix $M^{-1}CM$. The later matrix will correspond to a cycle as well, which may be obtained by the following three methods. In the case then M belongs to the $SL_2(\mathbb{R})$ group the next two methods make a proper conversion of M into Clifford-valued form.

8d ⟨specific methods of the class cycle 7a⟩+≡ (57a) ◁8c 9a▷
cycle *sl2_similarity*(**const ex** & *a*, **const ex** & *b*, **const ex** & *c*, **const ex** & *d*, **const ex** & *e* = 0,
const ex & *sign* = (**new** *tensdelta*)→*setflag*(*status_flags*::*dynallocated*),
bool *not_inverse*=**true**) **const**;
cycle *sl2_similarity*(**const ex** & *M*, **const ex** & *e* = 0,
const ex & *sign* = (**new** *tensdelta*)→*setflag*(*status_flags*::*dynallocated*),
bool *not_inverse*=**true**) **const**;

Uses cycle 4 5a 6c 6c 6c 6c 6c 68a 68b 69c 69c 71 71 71 71 71 and ex 6d 16b 58c 71 72 96 97a 97b 98a.

If M is a generic matrix of another sort then ii is used in the similarity in the unchanged form by the next method.

9a ⟨specific methods of the class cycle 7a⟩+≡ (57a) ◁8d 9b▷

```
cycle matrix_similarity(const ex & a, const ex & b, const ex & c,
                const ex & d, const ex & e = 0,
                const ex & sign = (new tensdelta)→setflag(status_flags::dynallocated),
                bool not_inverse=true) const;
```

Uses cycle 4 5a 6c 6c 6c 6c 6c 68a 68b 69c 69c 71 71 71 71 71 and ex 6d 16b 58c 71 72 96 97a 97b 98a.

Finally, we have a method for reflection of a cycle in another cycle C, which is given by the similarity of the representing matrices: CC_1C, see [7, § 4.2].

9b ⟨specific methods of the class cycle 7a⟩+≡ (57a) ◁9a 9c▷

```
cycle cycle_similarity(const cycle & C, const ex & e = 0,
                const ex & sign = (new tensdelta)→setflag(status_flags::dynallocated),
                const ex & sign1 = 0) const;
```

Uses cycle 4 5a 6c 6c 6c 6c 6c 68a 68b 69c 69c 71 71 71 71 71 and ex 6d 16b 58c 71 72 96 97a 97b 98a.

A cycle in the matrix form (2) naturally defines a Möbius transformations of the points:

$$\begin{pmatrix} l_i\sigma^i_j\tilde{e}^j & m \\ k & -l_i\sigma^i_j\tilde{e}^j \end{pmatrix} : \mathbf{x} \mapsto \frac{l_i\sigma^i_j\tilde{e}^j\mathbf{x}+m}{k\mathbf{x}-l_i\sigma^i_j\tilde{e}^j} \tag{3}$$

The following methods realised this transformations.

9c ⟨specific methods of the class cycle 7a⟩+≡ (57a) ◁9b 9d▷

```
inline ex moebius_map(const ex & P, const ex & e = 0,
        const ex & sign = (new tensdelta)→setflag(status_flags::dynallocated)) const
 {return clifford_moebius_map(to_matrix(e, sign), P, (e.is_zero()?get_metric():e));}
```

Uses ex 6d 16b 58c 71 72 96 97a 97b 98a.

For two matrices C_1 and C_2 obtained from cycles the expression

$$\langle C_1, C_2\rangle = \Re\, tr\,(C_1C_2) \tag{4}$$

naturally defines an inner product in the space of cycles. The follwong methods realised it.

9d ⟨specific methods of the class cycle 7a⟩+≡ (57a) ◁9c 10a▷

```
inline ex inner_product(const cycle & C, const ex & e = 0,
        const ex & sign = (new tensdelta)→setflag(status_flags::dynallocated)) const
 {return scalar_part(mul(C, e, sign).trace());}
```

Uses cycle 4 5a 6c 6c 6c 6c 6c 68a 68b 69c 69c 71 71 71 71 71 and ex 6d 16b 58c 71 72 96 97a 97b 98a.

The inner product (4) defines an orthogonality relation $\langle C_1, C_2\rangle \equiv 0$ in the space of cycles which returned by the method *is_orthogonal*().

10a ⟨specific methods of the class cycle 7a⟩+≡ (57a) ◁9d 10b▷

```
inline relational is_orthogonal(const cycle & C, const ex & e = 0,
  const ex & sign = (new tensdelta)→setflag(status_flags::dynallocated)) const
   {return (inner_product(C, e, sign) ≡ 0);}
```

Uses cycle 4 5a 6c 6c 6c 6c 6c 68a 68b 69c 69c 71 71 71 71 71, ex 6d 16b 58c 71 72 96 97a 97b 98a, and relational 17b.

The remaining to methods check if a cycle is a liner object and if it is normalised to $k = 1$.

10b ⟨specific methods of the class cycle 7a⟩+≡ (57a) ◁10a

```
inline relational is_linear() const {return (k ≡ 0);}
inline relational is_normalized() const {return (k ≡ 1);}
```

Uses relational 17b.

2.6 Two dimensional cycles

Two dimensional cycle **cycle2D** is a derived class of **cycle**. We need to add only very few specific methods for two dimensions, notably for the visualisation.
This a specialisation of the constructors from **cycle** class to **cycle2D**. Here is the main constructor.

10c ⟨constructors of the class cycle2D 10c⟩≡ (58a) 10d▷

```
public:
 cycle2D(const ex & k1, const ex & l1, const ex & m1,
          const ex & metr = diag_matrix(lst(-1, -1)));
```

Uses cycle2D 10d 16c 16c 50a 52 58c 58c 58c 58c 58c 72 72 72 72 72 82c 82c 85a, ex 6d 16b 58c 71 72 96 97a 97b 98a, and lst 17b.

Constructor for the cycle from *l* and square of its radius.

10d ⟨constructors of the class cycle2D 10c⟩+≡ (58a) ◁10c 10e▷

```
cycle2D(const lst & l, const ex & metr = diag_matrix(lst(-1, -1)), const ex & r_squared =0,
          const ex & e =0, const ex & sign = diag_matrix(lst(1, 1)));
```

Defines:
cycle2D, used in chunks 10, 14a, 18b, 21b, 23–26, 28a, 29e, 33–35, 42, 45, 47, 48, 51, 53, 54, 58, 59a, 62a, 81–83, 91b, 92a, and 94b.
Uses ex 6d 16b 58c 71 72 96 97a 97b 98a and lst 17b.

Make a two dimensional cycle out of a general one, if the dimensionality of the space permits. The metric of point space can be replaced as well if a valid *metr* is supplied.

10e ⟨constructors of the class cycle2D 10c⟩+≡ (58a) ◁10d

```
cycle2D(const cycle & C, const ex & metr = 0);
```

Uses cycle 4 5a 6c 6c 6c 6c 6c 68a 68b 69c 69c 71 71 71 71 71, cycle2D 10d 16c 16c 50a 52 58c 58c 58c 58c 58c 72 72 72 72 72 82c 82c 85a, and ex 6d 16b 58c 71 72 96 97a 97b 98a.

The method *focus*() returns list of the focus coordinates and the focal length is provided by *focal_length*(). This turns to be meaningful not only for parabolas, see [7].

11a ⟨methods specific for class cycle2D 11a⟩≡ (58a) 11b▷

```
public:
 ex focus(const ex & e = diag_matrix(lst(-1, 1)), bool return_matrix = false) const;
 inline ex focal_length() const {return (get_l(1)÷2÷k);} // focal length of the cycle
```

Uses `cycle` 4 5a 6c 6c 6c 6c 6c 68a 68b 69c 69c 71 71 71 71 71, `ex` 6d 16b 58c 71 72 96 97a 97b 98a, and `lst` 17b.

The methods *roots*() returns values of u (if *first* **true**) such that $k(u^2+ey^2)-2l_1u-2l_2y+m=0$, i.e. solves a quadratic equations. If *first* = **false** then values of v satisfying to $k(y^2+ev^2)-2l_1y-2l_2v+m=0$ are returned.

11b ⟨methods specific for class cycle2D 11a⟩+≡ (58a) ◁11a 11c▷

```
 lst roots(const ex & y = 0, bool first = true) const; //roots of a cycle
 lst line_intersect(const ex & a, const ex & b) const; // intersection points with the line ax+b
```

Uses `cycle` 4 5a 6c 6c 6c 6c 6c 68a 68b 69c 69c 71 71 71 71 71, `ex` 6d 16b 58c 71 72 96 97a 97b 98a, `lst` 17b, and `points` 95b.

The method *metapost_draw*() outputs to the stream *ost* `MetaPost` comands to draw parts of two the **cycle2D** within the rectangle with the lower left vertex (*xmin*, *ymin*) and upper right (*xmax*, *ymax*). The colour of drawing is specified by *color* (the default is black) and any additional `MetaPost`options can be provided in the string *more_options*. By default each set of the drawing commands is preceded a comment line giving description of the cycle, this can be suppressed by setting *with_header* = **false**. The default number of points per arc is reasonable in most cases, however user can override this with supplying a value to *points_per_arc*. The last parameter is for internal use.

11c ⟨methods specific for class cycle2D 11a⟩+≡ (58a) ◁11b 12a▷

```
 void metapost_draw(ostream & ost, const ex & xmin = -5, const ex & xmax = 5,
          const ex & ymin = -5, const ex & ymax = 5, const lst & color = lst(),
          const char * more_options = "", bool with_header = true,
          int points_per_arc = 0, bool asymptote = false, char * picture = "") const;
```

Uses `ex` 6d 16b 58c 71 72 96 97a 97b 98a and `lst` 17b.

The similar method provides a drawing output for Asymptote [5] with the same meaning of parameters. However format of *more_options* should be adjusted correspondingly. Currently *asy_draw*() is realised as a wrapper around *metapost_draw*() but this may be changed.

12a ⟨methods specific for class cycle2D 11a⟩+≡ (58a) ◁11c

```
inline void asy_draw(ostream & ost, char * picture,
                     const ex & xmin = -5, const ex & xmax = 5,
                     const ex & ymin = -5, const ex & ymax = 5, const lst & color = lst(),
                     const char * more_options = "", bool with_header = true,
                     int points_per_arc = 0) const
{ metapost_draw(ost, xmin, xmax, ymin, ymax, color, more_options,
with_header, points_per_arc, true, picture); }

inline void asy_draw(ostream & ost = std::cout,
     const ex & xmin = -5, const ex & xmax = 5,
     const ex & ymin = -5, const ex & ymax = 5, const lst & color = lst(),
       const char * more_options = "", bool with_header = true,
       int points_per_arc = 0) const
{ metapost_draw(ost, xmin, xmax, ymin, ymax, color, more_options,
with_header, points_per_arc, true); }
```

Uses ex 6d 16b 58c 71 72 96 97a 97b 98a and lst 17b.

3 DEMONSTRATION THROUGH EXAMPLE

3.1 Outline of the *main*()

The *main*() procedure does several things:

1. Makes symbolic calculations related to Möbius invariance;

12b ⟨List of symbolic calculations 12b⟩≡ (14c) 13a▷

```
⟨Moebius transformation of cycles 17c⟩
⟨K-orbit invariance 18b⟩
⟨Check Moebius transformations of zero cycles 18d⟩
⟨Check transformations of zero cycles by conjugation 20a⟩
cout << endl;
```

2. Calculates properties of orthogonality conditions and corresponding inversion in cycles;

13a ⟨List of symbolic calculations 12b⟩+≡ (14c) ◁12b 13b▷
  ⟨Orthogonality conditions 20c⟩
  ⟨Two points and orthogonality 21c⟩
  ⟨One point and orthogonality 21e⟩
  ⟨Orthogonal line 22e⟩
  ⟨Inversion in cycle 23b⟩
  ⟨Reflection in cycle 24b⟩
  ⟨Yaglom inversion 25a⟩
  cout ≪ *endl*;

3. Calculates properties of s-orthogonality conditions and second type of inversion;

13b ⟨List of symbolic calculations 12b⟩+≡ (14c) ◁13a 13c▷
  ⟨Second orthogonality conditions 25b⟩
  ⟨One point and s-orthogonality 26d⟩
  ⟨s-Orthogonal line 26e⟩
  ⟨s-Inversion in cycle 27b⟩
  cout ≪ *endl*;

4. Calculates various length formulae;

13c ⟨List of symbolic calculations 12b⟩+≡ (14c) ◁13b 13d▷
  ⟨Distances from cycles 28a⟩
  ⟨Lengths from centre 32a⟩
  ⟨Lengths from focus 32c⟩
  ⟨Infinitesimal cycle calculations 34b⟩
  cout ≪ *endl*;

5. Generates `Asymptote` output of the for illustrations.

13d ⟨List of symbolic calculations 12b⟩+≡ (14c) ◁13c

Since we aiming into two targets simultaneously—validate our software and use it for mathematical proofs—there are many double checks and superfluous calculations. The positive aspect of this—a better illustration of the library usage.

3.1.1 Program's outline

Here is the main entry into the program and its outline. We start from some inclusions, note that GiNaC is included through <**cycle**.*h*>.

13e ⟨* 13e⟩≡ 14a▷
```
#include <cycle.h>
#include <fstream>
```

Uses `cycle` 4 5a 6c 6c 6c 6c 6c 68a 68b 69c 69c 71 71 71 71 71.

This function will be used for the parabolic Cayley transform in subsection 3.6

14a ⟨* 13e⟩+≡ ◁13e 14b▷
```
cycle2D parab_tr(const cycle2D & C, const ex & sign = 1)
{
 return cycle2D(C.get_k()-sign*C.get_l(1), C.get_l(), C.get_m()-C.get_l(1), C.get_unit());
}
```

Uses cycle2D 10d 16c 16c 50a 52 58c 58c 58c 58c 58c 72 72 72 72 72 82c 82c 85a and ex 6d 16b 58c 71 72 96 97a 97b 98a.

The structure of the program is transparent. We declare all variables.

14b ⟨* 13e⟩+≡ ◁14a 14c▷
```
int main(){
 ⟨Declaration of variables 15a⟩
 cout << boolalpha;
```

Defines:
main, never used.

Then we make all symbolic calculations listed above. The exception catcher helps to identify the possible problems.

14c ⟨* 13e⟩+≡ ◁14b 14d▷
```
try {
  ⟨List of symbolic calculations 12b⟩
 } catch (exception &p) {
 cerr << "*****        Got problem1: " << p.what() << endl;
 }
```

Defines:
catch, used in chunk 63b.

We end up with drawing illustration to our paper [7].

14d ⟨* 13e⟩+≡ ◁14c
```
 ⟨Draw Asymptote pictures 36b⟩
 }
```

3.1.2 Declaration of variables

First we declare all variables from the standard GiNaC classes here.

15a ⟨Declaration of variables 15a⟩≡ (14b) 15b ▷

```
const char* eph_names="eph";
const numeric half(1,2);

const realsymbol a("a"), b("b"), c("c"), d("d"), x("x"), y("y"), z("z"), t("t"),
// Cycles parameters
        k("k"), l("L"), m("m"), k1("k1"), l1("l1"), m1("m1"), n("n"), n1("n1"),
        u("u"), v("v"), u1("u'"), v1("v'"), // Coordinates of points in ℝ^2
        epsilon("eps", "\\epsilon"); // The "infinitesimal" number

const varidx nu(symbol("nu", "\\nu"), 2), mu(symbol("mu", "\\mu"), 2);
```

Defines:
eph_names, used in chunks 42, 45, 53, and 54c.
numeric, used in chunks 7a, 8b, 16c, 28a, 29e, 31b, 33b, 34a, 42, 43, 45, 46, 49, 53, 56, 58b, 62, 63b, 65a, 69a, 70, 73–75, 83–89, 91–93, and 95–99.
realsymbol, never used.
varidx, used in chunks 60, 61, 63a, 65b, 74, 76, 78, 83, and 86b.
Uses points 95b, u 91a, and v 91a.

We need a plenty of symbols which will hold various parameters like e_1^2, $\breve{e}_1^2$, s for the FSCc.

15b ⟨Declaration of variables 15a⟩+≡ (14b) ◁ 15a 15c ▷

```
//Signs of e_1^2 of ĕ_1^2
const realsymbol sign("s", "\\sigma"), sign1("s1", "\\breve{\\sigma}"),
        sign2("s2", "\\sigma_2"), sign3("s3", "\\sigma_3"), sign4("s4", "\\sigma_4");
int si, si1; // Values of e_1^2 and ĕ_1^2 for substitutions

const matrix S2(2, 2, lst(1, 0, 0, jump_fnct(sign2))),
        S3(2, 2, lst(1, 0, 0, jump_fnct(sign3))),
        S4(2, 2, lst(1, 0, 0, jump_fnct(sign4))); //Signs of l in the matrix representations of cycles
```

Defines:
matrix, used in chunks 5c, 8, 25a, 27a, 56, 60, 61, 63–65, 74, 76–81, 83a, and 100.
realsymbol, never used.
si, used in chunks 24b, 29, 31b, 36b, 42–45, 53, and 54.
si1, used in chunks 31b, 36b, 42, 44, 45, and 53.
Uses jump_fnct 56 and lst 17b.

Here are several expressions which will keep results of calculations.

15c ⟨Declaration of variables 15a⟩+≡ (14b) ◁ 15b 16a ▷

```
ex u2, v2, // Coordinates of the Moebius transform of (u, v)
        u3, v3, u4, v4, u5, v5,
        P, P1, P2, P3, // points on the plain
        K, L0, L1, // Parameters of cycles
        Len_c, Len_cD, Len_f, Len_fD; // Expressions of Lengths
```

Uses ex 6d 16b 58c 71 72 96 97a 97b 98a, points 95b, u 91a, and v 91a.

Two generic points on the plain are defined as constant vectors (2x1 matrices).

16a ⟨Declaration of variables 15a⟩+≡ (14b) ◁15c 16b▷
```
const matrix W(2,1, lst(u, v)), W1(2,1, lst(u1, v1));
```

Defines:
matrix, used in chunks 5c, 8, 25a, 27a, 56, 60, 61, 63–65, 74, 76–81, 83a, and 100.
Uses lst 17b, u 91a, and v 91a.

Next we define metrics (through Clifford units) for the space of points (*M*, *e*) and space of spheres (*M1*, *es*).

16b ⟨Declaration of variables 15a⟩+≡ (14b) ◁16a 16c▷
```
const ex M = diag_matrix(lst(-1, sign)), // Metrics of point spaces
        e = clifford_unit(mu, M, 0), // Clifford algebra generators in the point space
        M1 = diag_matrix(lst(-1, sign1)), // Metrics of cycles spaces
        es = clifford_unit(nu, M1, 1); // Clifford algebra generators in the sphere space
```

Defines:
ex, used in chunks 4–12, 14a, 15c, 17a, 28b, 33b, 35, 48, 55–59, 62, 63, 65–68, 70, 73–89, 92b, 93a, 95a, and 98–100.
Uses lst 17b.

Now we define instances of **cycle2D** class. Some of them (like *real_line* or generic cycles *C* and *C1*) are constants.

16c ⟨Declaration of variables 15a⟩+≡ (14b) ◁16b 16d▷
```
cycle2D C2, C3, C4, C5, C6, C7, C8, C9, C10, C11;

const cycle2D real_line(0, lst(0, numeric(1)), 0, e), // the real line
        C(k, lst(l, n), m, e), C1(k1, lst(l1, n1), m1, e); // two generic cycles
const cycle2D Zinf(0, lst(0, 0), 1, e), // the zero-radius cycle at infinity
        Z(lst(u, v), e), Z1(lst(u, v), e, 0, es), // two generic cycles of zero-radius
        Z2(lst(u, v), e, 0, es, S2);
```

Defines:
cycle2D, used in chunks 10, 14a, 18b, 21b, 23–26, 28a, 29e, 33–35, 42, 45, 47, 48, 51, 53, 54, 58, 59a, 62a, 81–83, 91b, 92a, and 94b.
Uses cycle 4 5a 6c 6c 6c 6c 6c 68a 68b 69c 69c 71 71 71 71 71, lst 17b, numeric 15a, u 91a, and v 91a.

For solution of various systems of linear equations we need the followings **lst**s.

16d ⟨Declaration of variables 15a⟩+≡ (14b) ◁16c 17a▷
```
lst eqns, eqns1,
        vars=lst(k1, l1, m1, n1),
        solns, solns1, // Solutions of linear systems
        sign_val;
```

Uses lst 17b.

These are expression for various orthogonality conditions between cycles which we be defined and used later.

17a ⟨Declaration of variables 15a⟩+≡ (14b) ◁16d 17b▷

ex *ortho*, *orthoz*, *orthotz*, // Different orthogonality relations of the first kind
// Different orthogonality relations of the second kind
ortho_second, *orthoz_second_a*, *orthoz_second_b*, *orthotz_second*;

Uses `ex` 6d 16b 58c 71 72 96 97a 97b 98a.

Here are **relational**s and lists of **relational**s which will be used for automatic simplifications in calculations. They are based on properties of $SL_2(\mathbb{R})$ and values of the parameters.

17b ⟨Declaration of variables 15a⟩+≡ (14b) ◁17a

// since $ad - bc \equiv 1$
const relational *sl2_relation* = ($c{*}b \equiv a{*}d$-1), *sl2_relation1* = ($a \equiv (1{+}b{*}c){\div}d$);
// $s_i^3 \equiv s\_i$ since $s_i = -1, 0, 1$
const lst *signs_cube* = **lst**(*pow*(*sign*, 3) ≡ *sign*, *pow*(*sign1*, 3) ≡ *sign1*);

int *debug* = 0;

Defines:
`debug`, used in chunks 21d, 23a, 26–30, and 32d.
`lst`, used in chunks 5a, 10–12, 15–18, 20–36, 42, 43a, 45, 47–54, 60a, 62b, 64, 67, 68a, 74b, 75, 77, 78, 80a, 82, 83, 85, 86, 88c, 89b, 91–94, and 100a.
`relational`, used in chunks 7d, 10, and 67a.

3.2 Möbius Transformation and Conjugation of Cycles

3.2.1 Möbius Invariance of cycles

We check now that a Möbius transformation g SL acts on cycles by similarity $g : C \to gCg^{-1}$. Firstly we define a **cycle2D** *C2* by the condition between *k*, *l* and *m* in the generic **cycle2D** *C* that *C* goes through some point (u, v).

17c ⟨Moebius transformation of cycles 17c⟩≡ (12b) 17d▷

C2 = *C*.*subject_to*(**lst**(*C*.*passing*(*W*)));

Uses `lst` 17b.

Then we define the point *P* to be the Möbius transfrom of (u, v) by an arbitrary g.

17d ⟨Moebius transformation of cycles 17c⟩+≡ (12b) ◁17c 18a▷

P = *clifford_moebius_map*(*sl2_clifford*(*a*, *b*, *c*, *d*, *e*), *W*, *e*).*subs*(*sl2_relation1*,
subs_options::*algebraic* | *subs_options*::*no_pattern*);

Finally we verify that the new cycle gCg^{-1} goes through *P*. This proves [7, Lem. 3.2].

18a ⟨Moebius transformation of cycles 17c⟩+≡ (12b) ◁17d

```
cout << "Conjugation of a cycle comes through "
        "Moebius transformation: "
  << C2.sl2_similarity(a, b, c, d, es, S2).val(P).subs(sl2_relation1,
        subs_options::algebraic | subs_options::no_pattern).normal().is_zero()
  << endl << endl;
```

Uses cycle 4 5a 6c 6c 6c 6c 6c 68a 68b 69c 69c 71 71 71 71 71.

3.2.2 Transformations of K-orbits

As a simple check we verify that cycles given by the equation $(u^2-\sigma v^2)-2v\frac{t^{-1}-\sigma t}{2}+1=0$, see [7, Lem. 2.4] are K-invariant, i.e. are K-orbits. To this end we make a similarity of a cycle *C2* of this from with a matrix from K and check that the result coincides with *C2*.

18b ⟨K-orbit invariance 18b⟩≡ (12b) 18c▷

```
C2 = cycle2D(1, lst(0, (pow(t,-1)-sign*t)÷2), 1, e);
cout << "A K-orbit is preserved: "
        << C2.sl2_similarity(cos(x), sin(x), -sin(x), cos(x), e).is_equal(C2)
```

Uses cycle2D 10d 16c 16c 50a 52 58c 58c 58c 58c 58c 72 72 72 72 72 82c 82c 85a and lst 17b.

We also check that *C2* passing the point $(0,t)$.

18c ⟨K-orbit invariance 18b⟩+≡ (12b) ◁18b

```
  << ", and  passing (0, t): " << (bool)C2.passing(lst(0, t)) << endl;
```

Uses lst 17b.

3.2.3 Transformation of Zero-Radius Cycles

Firstly, we check some basic information about the zero-radius cycles. This mainly done to verify our library.

18d ⟨Check Moebius transformations of zero cycles 18d⟩≡ (12b) 19a▷

```
cout << " Determinant of zero-radius Z1 cycle in metric e is "
  << canonicalize_clifford(Z1.det(e, S2)) << endl
  << " Focus of zero-radius cycle is " << Z1.focus(e) << endl
  << " Centre of zero-radius cycle is " << Z1.center(e) << endl
  << " Focal length of zero-radius cycle is " << Z1.focal_length() << endl;
```

Uses cycle 4 5a 6c 6c 6c 6c 6c 68a 68b 69c 69c 71 71 71 71 71.

This chunk checks that Möbius transformation of a zero-radius cycle is a zero-radius cycle with centre obtained from the first one by the same Möbius transformation.

19a ⟨Check Moebius transformations of zero cycles 18d⟩+≡ (12b) ◁18d 19b▷
```
C2 = Z1.sl2_similarity(a, b, c, d, e, S2);
cout << "Image of the zero-radius cycle under "
        "Moebius transform has radius: "
 << canonicalize_clifford(C2.det(es, S2)).subs(sl2_relation1,
         subs_options::algebraic | subs_options::no_pattern) << endl;
```

Uses cycle 4 5a 6c 6c 6c 6c 6c 68a 68b 69c 69c 71 71 71 71 71.

Here we find parameters of the transformed zero-radius cycle $C_2 = gZg^{-1}$.

19b ⟨Check Moebius transformations of zero cycles 18d⟩+≡ (12b) ◁19a 19c▷
```
C2 = Z.sl2_similarity(a, b, c, d, e, S2);
K = C2.get_k();
L0 = C2.get_l(0);
L1 = C2.get_l(1);
```

Now we calculate the Möbius transformation of the centre of Z

19c ⟨Check Moebius transformations of zero cycles 18d⟩+≡ (12b) ◁19b 19d▷
```
P = clifford_moebius_map(sl2_clifford(a, b, c, d, e), W, e);
u2 = P.op(0).subs(sl2_relation, subs_options::algebraic | subs_options::no_pattern);
v2 = P.op(1).subs(sl2_relation, subs_options::algebraic | subs_options::no_pattern);
```

And we finally check that P coincides with the centre of the transformed cycle $C2$. This proves [7, Lem. 3.11].

19d ⟨Check Moebius transformations of zero cycles 18d⟩+≡ (12b) ◁19c
```
cout <<"The centre of the Moebius transformed "
      "zero-radius cycle is: "
 << equality((u2*K-L0).subs(sl2_relation, subs_options::algebraic
                          | subs_options::no_pattern)) << ", "
 << equality((v2*K-L1).subs(sl2_relation, subs_options::algebraic
                          | subs_options::no_pattern)) <<endl ;
```

Uses cycle 4 5a 6c 6c 6c 6c 6c 68a 68b 69c 69c 71 71 71 71 71 and equality 56 99a.

3.2.4 Cycles conjugation

This chunk checks that transformation of a zero-radius cycle by conjugation with a cycle is a zero-radius cycle with centre obtained from the first one by the same transformation.

Firstly we calculate parameters of $C_2 = CZC$.

20a ⟨Check transformations of zero cycles by conjugation 20a⟩≡ (12b) 20b ▷

```
C2 = Z.cycle_similarity(C, e, S2, S3);
cout << "Image of the zero-radius cycle under cycle"
        " similarity has radius: "
 << canonicalize_clifford(C2.det(e, S2)).subs(sl2_relation1,
        subs_options::algebraic | subs_options::no_pattern).normal() << endl;

K = C2.get_k();
L0 = C2.get_l(0);
L1 = C2.get_l(1);
```

Uses cycle 4 5a 6c 6c 6c 6c 6c 68a 68b 69c 69c 71 71 71 71 71.

Then we check that it coincides with transformation point P which is calculated in agreement with above used matrices $S2$ and $S3$. This proves the result [7, Lem. 4.10]

20b ⟨Check transformations of zero cycles by conjugation 20a⟩+≡ (12b) ◁ 20a

```
P = C.moebius_map(W, e, S2.mul(S3));
u2 = P.op(0);
v2 = P.op(1);

cout <<"The centre of the conjugated zero-radius cycle"
      " coincides with Moebius tr: "
 << equality(u2*K-L0) << ", " << equality(v2*K-L1) <<endl ;
```

Uses cycle 4 5a 6c 6c 6c 6c 6c 68a 68b 69c 69c 71 71 71 71 71 and equality 56 99a.

3.3 Orthogonality of Cycles

3.3.1 Various orthogonality conditions

We calculate orthogonality condition between two **cycle2D**s by the identity $\Re\, tr(C_1 C_2) = 0$. The expression are stored in variables, which will be used later in our calculations.

ortho is the orthogonality of two generic **cycle2D**s.

20c ⟨Orthogonality conditions 20c⟩≡ (13a) 21a ▷

```
ortho = C.is_orthogonal(C1, es, S2);
cout << " The orthogonality is " << ortho << endl
    << " The orthogonality of two lines is "
    << ortho.subs(lst(k == 0, k1 == 0)) << endl;
```

Uses lst 17b.

orthoz is the orthogonality of a generic **cycle2D** to a zero-radius **cycle2D**.

21a ⟨Orthogonality conditions 20c⟩+≡ (13a) ◁20c 21b▷

```
orthoz = C.is_orthogonal(Z, es);
cout << " The orthogonality to z-r-cycle is " << orthoz << endl;
```

Uses cycle 4 5a 6c 6c 6c 6c 6c 68a 68b 69c 69c 71 71 71 71 71.

orthotz is the orthogonality of two zero-radius **cycle2D**s.

21b ⟨Orthogonality conditions 20c⟩+≡ (13a) ◁21a

```
C2 = cycle2D(lst(u1, v1), e, 0, S2);
orthotz = C2.is_orthogonal(Z1, es);
cout << " The orthogonality of two z-r-cycle is " << orthotz << endl;
```

Uses cycle 4 5a 6c 6c 6c 6c 6c 68a 68b 69c 69c 71 71 71 71 71, cycle2D 10d 16c 16c 50a 52 58c 58c 58c 58c 58c 72 72 72 72 72 82c 82c 85a, and lst 17b.

This chunk finds the parameters of a cycle *C2* passing through two points (u, v), (u_1, v_1) and orthogonal to the given cycle *C*. This gives three linear equations with four variables which are consistent in a generic position.

21c ⟨Two points and orthogonality 21c⟩≡ (13a) 21d▷

```
C2 = C1.subject_to(lst(C1.passing(W),
        C1.passing(W1),
        C1.is_orthogonal(C, es)), vars);
```

Uses lst 17b.

To find the singularity condition of the above solution we analyse the denominator of *k*, which calculated to be:

$$k = \frac{-2(u'(\sigma_1 n + vk) - vl + (-kv' - \sigma_1 n)u + lv')n_1}{-u'^2 l + u'^2 uk + \sigma l v'^2 - u'u^2 k + u'v^2 \sigma k + u'm - u\sigma k v'^2 + u^2 l - v^2 \sigma l - um}.$$

21d ⟨Two points and orthogonality 21c⟩+≡ (13a) ◁21c

```
if (debug > 0)
cout << "Cycle through two point is possible and "
    "unique if denominator is not zero: " << endl
    << C2.get_k() << endl << endl;
```

Uses debug 17b.

3.3.2 Orthogonality and Inversion

Now we check that any orthogonal cycle comes through the inverse of any its point. To this end we calculate a generic cycle *C2* passing through a point (u, v) and orthogonal to a cycle *C*.

21e ⟨One point and orthogonality 21e⟩≡ (13a) 22b▷

```
C2 = C1.subject_to(lst(C1.passing(W),
        C1.is_orthogonal(C, es)));
```

Uses lst 17b.

Then we calculate another cycle *C3* with an additional condition that it passing through the Möbius transform P of (u, v).

22b ⟨One point and orthogonality 21e⟩+≡ (13a) ◁21e 22c▷

```
P = C.moebius_map(W, e, -M1);
C3 = C1.subject_to(lst(C1.passing(P),
        C1.passing(W),
        C1.is_orthogonal(C, es)));
```

Uses `lst` 17b.

Then we check twice in different ways the same mathematical statement:

1. that both cycles *C2* and *C3* are identical, i.e. the addition of inverse point does not put more restrictions;

   22c ⟨One point and orthogonality 21e⟩+≡ (13a) ◁22b 22d▷

   ```
   cout << "Both orthogonal cycles (through one point and"
           " through its inverse) are the same: "
     << C2.is_equal(C3) << endl
   ```

2. that cycle *C2* goes through the Möbius transform *P* as well.

   22d ⟨One point and orthogonality 21e⟩+≡ (13a) ◁22c

   ```
     << "Orthogonal cycle goes through the transformed point: "
     << C2.val(P).normal().is_zero() << endl << endl;
   ```

   Uses `cycle` 4 5a 6c 6c 6c 6c 6c 68a 68b 69c 69c 71 71 71 71 71.

3.3.3 Orthogonal Lines

This chunk checks that the straight line *C4* passing through a point (u, v) and its inverse *P* in the cycle *C* is orthogonal to the initial cycle *C*.

22e ⟨Orthogonal line 22e⟩≡ (13a) 22f▷

```
C4 = C1.subject_to(lst(C1.passing(W),
        C1.passing(P),
        C1.is_linear()));
cout << "Line through point and its inverse is orthogonal: "
        << C4.inner_product(C, es).is_zero() << endl;
```

Uses `lst` 17b.

We also calculate that all such lines intersect in a single point (u_3, v_3), which is independent from (u, v). This point will be understood as centre of the cycle *C5* in § 3.3.4.

22f ⟨Orthogonal line 22e⟩+≡ (13a) ◁22e 23a▷

```
u3 = C.center().op(0);
v3 = C4.roots(u3, false).op(0).normal();
cout << "All lines come through the point ("
        << u3 << ", " << v3 << ")" << endl;
```

The double check is done next: we calculate the inverse *P1* of a vector (*u3*+*u*, *v3*+*v*) and check that *P1*-(*u3*, *v3*) is collinear to (*u*, *v*).

23a ⟨Orthogonal line 22e⟩+≡ (13a) ◁22f

```
P1 = C.moebius_map(lst(u3+u, v3+v), e, -M1);
cout << "Conjugated vector is parallel to (u,v): "
 << ((P1.op(0)-u3)*v-(P1.op(1)-v3)*u).normal().is_zero() << endl;
if (debug > 1)
 cout << "Conjugated vector to (u, v) is: (" << (P1.op(0)-u3).normal()
     << ", " << (P1.op(1)-v3).normal() << ")" << endl;
```

Uses debug 17b, lst 17b, u 91a, and v 91a.

3.3.4 The Ghost Cycle

We build now the cycle *C5* which defines inversion. We build it from two conditions:

1. *C5* has its centre in the point (*u3*, *v3*) which is the intersection of all orthogonal lines (see § 3.3.3).
2. The determinant of *C5* with delta-sign is equal to determinant of *C* with signs defined by *M1*.

23b ⟨Inversion in cycle 23b⟩≡ (13a) 23c ▷

```
C5 = cycle2D(lst(u3, -v3*jump_fnct(sign)), e, C.det(e, M1)).subs(signs_cube,
        subs_options::algebraic | subs_options::no_pattern);
```

Uses cycle2D 10d 16c 16c 50a 52 58c 58c 58c 58c 58c 72 72 72 72 72 82c 82c 85a, jump_fnct 56, and lst 17b.

As a consequence we find out that *C5* has the same roots as *C*.

23c ⟨Inversion in cycle 23b⟩+≡ (13a) ◁23b 23d ▷

```
cout << "C5 has common roots with C : "
   << (C5.val(lst(C.roots().op(0), 0)).is_zero()
     ∧ C5.val(lst(C.roots().op(1), 0)).is_zero()) << endl
   << "H(s)-centre of C5 is equal to s1-centre of C: "
   << (C5.center(diag_matrix(lst(-1,jump_fnct(sign))), true)
     -C.center(es, true)).normal().is_zero() <<  endl;
```

Uses jump_fnct 56 and lst 17b.

Finally we calculate point *P1* which is the inverse of (u_3, v_3) in *C5*.

23d ⟨Inversion in cycle 23b⟩+≡ (13a) ◁23c 24a ▷

```
P1 = C5.moebius_map(W, e, diag_matrix(lst(1, -jump_fnct(sign))));
```

Uses jump_fnct 56 and lst 17b.

The final check: *P* (inversion in *C5* in terms of *sign*) coincides with *P*—the inversion in *C* in terms of *sign1*, see chunk 22a.

24a ⟨Inversion in cycle 23b⟩+≡ (13a) ◁23d
```
cout ≪ "Inversion in (C5, sign)  coincides with"
        " inversion in (C, sign1): "
 ≪ (P1-P).subs(signs_cube, subs_options::algebraic
                | subs_options::no_pattern).normal().is_zero()
 ≪ endl;
```

3.3.5 The real line and reflection in cycles

We check that conjugation $C_1 \mathbb{R} C_1$ maps the *real_line* to the cycle *C* and wise verse for the properly chosen *C1*, see [7, Lem. 4.14].

24b ⟨Reflection in cycle 24b⟩≡ (13a) 24c▷
```
for (si=-1; si<2; si+=2) {
 C9 = cycle2D(k*sign1, lst(l*sign1, n*sign1
     +si*sqrt(-C.det(es,(new tensdelta)→setflag(status_flags::dynallocated), k)*sign1)),
            m*sign1,es);
 cout ≪ "Inversion to the real line (with " ≪ (si≡-1? "-" : "+")
  ≪ " sign): " ≪ endl
  ≪ "   Conjugation of the real line is the cycle C: "
  ≪ real_line.cycle_similarity(C9, es).is_equal(C) ≪ endl
  ≪ "   Conjugation of the cycle C is the real line: "
  ≪ C.cycle_similarity(C9, es).is_equal(real_line) ≪ endl
```

Uses `cycle` 4 5a 6c 6c 6c 6c 6c 68a 68b 69c 69c 71 71 71 71 71, `cycle2D` 10d 16c 16c 50a 52 58c 58c 58c 58c 58c 72 72 72 72 72 82c 82c 85a, `lst` 17b, and `si` 15b.

We also check two additional properties which caracterises the inversion cycle *C9* in term of common roots of *C* [7, Lem. 4.14.ii] and *C* passing through *C9* centre [7, Lem. 4.14.iii].

24c ⟨Reflection in cycle 24b⟩+≡ (13a) ◁24b
```
  ≪ "   Inversion cycle has common roots with C: "
  ≪ (C9.val(lst(C.roots().op(0), 0)).is_zero() ∧ C9.val(lst(C.roots().op(1), 0)).is_zero())
  ≪ endl
  ≪ "   C passing the centre of inversion cycle: "
  ≪ cycle2D(C, es).val(C9.center()).subs(pow(sign1,2)≡1, subs_options::algebraic
     | subs_options::no_pattern).subs(sign1≡sign).normal().is_zero() ≪ endl;
}
```

Uses `cycle` 4 5a 6c 6c 6c 6c 6c 68a 68b 69c 69c 71 71 71 71 71, `cycle2D` 10d 16c 16c 50a 52 58c 58c 58c 58c 58c 72 72 72 72 72 82c 82c 85a, and `lst` 17b.

3.3.6 Yaglom inversion of the second kind

In the book [13, § 10] the inversion of second kind related to a parabola $v = k(u-l)^2 + m$ is defined by the map:

$$(u, v) \mapsto (u, 2(k(u-l)^2 + m) - v).$$

We shows here that this is a composition of three inversions in two parabolas and the real line, see [9, Prop.4.15].

25a ⟨Yaglom inversion 25a⟩≡ (13a)

```
cout << "Yaglom inversion of the second kind is three"
        " reflections in the cycles: "
 << (real_line.moebius_map(cycle2D(lst(l, 0), e, -m÷k).moebius_map(cycle2D(lst(l, 2*m),
                                e, -m÷k).moebius_map(W))).subs(sign≡0)
  -matrix(2,1,lst(u, 2*(k*pow(u-l,2)+m)-v))).normal().is_zero() << endl;
```

Uses `cycle2D` 10d 16c 16c 50a 52 58c 58c 58c 58c 58c 72 72 72 72 72 82c 82c 85a, `lst` 17b, `matrix` 15b 16a, `u` 91a, and `v` 91a.

3.4 Orthogonality of the Second Kind

We study now the orthogonality condition of the second kind (s-orthogonality), [7, § 4.3].

3.4.1 Expressions for s-orthogonality

One more simple consistency check: the *real_line* is invariant under all Möbius transformations.

25b ⟨Second orthogonality conditions 25b⟩≡ (13b) 25c ▷

```
cout << "The real line is Moebius invariant: "
        << real_line.is_equal(real_line.sl2_similarity(a, b, c, d, es)) << endl
    << "Reflection in the real line: "
    << Z.cycle_similarity(real_line, es).normalize() << endl;
```

The orthogonality condition of the second kind between two different cycles is calculated by the identity [7, § 4.3]

$$\Re\, tr \langle C_1 C_2 C_1, \mathbb{R} \rangle = 0.$$

ortho_second is s-orthogonality of two generic **cycle2D**s.

25c ⟨Second orthogonality conditions 25b⟩+≡ (13b) ◁ 25b 26a ▷

```
ortho_second = C1.cycle_similarity(C, es, S2).get_l(1).normal();
cout << " The s-orthogonality is " << ortho_second << " = 0" << endl
    << " The s-orthogonality of two lines is "
    << ortho_second.subs(lst(k ≡ 0, k1 ≡ 0)) << endl;
```

Uses `lst` 17b.

ortho_second_a is s-orthogonality of a generic **cycle2D** to a zero-radius **cycle2D**.

26a ⟨Second orthogonality conditions 25b⟩+≡ (13b) ◁25c 26b▷

```
orthoz_second_a = C.cycle_similarity(Z1, es, S2).get_l(1).normal();
cout << " The s-orthogonality to z-r-cycle is first way: "
        << orthoz_second_a << " = 0" << endl;
```

Uses cycle 4 5a 6c 6c 6c 6c 6c 68a 68b 69c 69c 71 71 71 71 71.

Since s-orthogonality is not symmetric [7, § 4.3], *ortho_second_b* is s-orthogonality of a zero-radius **cycle2D** to a generic **cycle2D**.

26b ⟨Second orthogonality conditions 25b⟩+≡ (13b) ◁26a 26c▷

```
orthoz_second_b = Z1.cycle_similarity(C, es, S2).get_l(1).normal();
cout << " The s-orthogonality to z-r-cycle is second way: "
        << orthoz_second_b << " = 0" << endl;
```

Uses cycle 4 5a 6c 6c 6c 6c 6c 68a 68b 69c 69c 71 71 71 71 71.

ortho_second is s-orthogonality of two zero-radius **cycle2D**s.

26c ⟨Second orthogonality conditions 25b⟩+≡ (13b) ◁26b

```
C9 = cycle2D(lst(u1, v1), e);
orthotz_second = C9.cycle_similarity(Z1, es, S2).get_l(1).normal();
cout << " The s-orthogonality of two z-r-cycle is "
        << orthotz_second << " = 0" << endl;
```

Uses cycle 4 5a 6c 6c 6c 6c 6c 68a 68b 69c 69c 71 71 71 71 71, cycle2D 10d 16c 16c 50a 52 58c 58c 58c 58c 58c 72 72 72 72 72 82c 82c 85a, and lst 17b.

3.4.2 Properies of s-orthogonality

Find the parameters of cycle passing through a point and s-orthogonal to the given one

26d ⟨One point and s-orthogonality 26d⟩≡ (13b)

```
C6 = C1.subject_to(lst(C1.passing(W), ortho_second ≡ 0));
if (debug > 1)
 cout << "Cycle s-orthogonal to (k, (l, n), m) is: " << endl
      << C6 << endl;
```

Uses debug 17b and lst 17b.

Check the orthogonality of the line through a point to the cycle.

26e ⟨s-Orthogonal line 26e⟩≡ (13b) 27a▷

```
C7 = C6.subject_to(lst(C6.is_linear()));
u4 = C.center().op(0);
v4 = C7.roots(u4, false).op(0).normal();
```

Uses lst 17b.

All orthogonal lines come through the same point, which the focus of the cycle *C* with respect to metric (-1, -*sign1*).

27a ⟨s-Orthogonal line 26e⟩+≡ (13b) ◁26e

```
cout << "All lines come through the focus related \\breve{e}: "
  << (C.focus(diag_matrix(lst(-1, -sign1)), true)-matrix(2, 1,
                     lst(u4, v4))).normal().is_zero() << endl;
```

Uses lst 17b and matrix 15b 16a.

3.4.3 Inversion from the s-orthogonality

We express s-orthogonality to a cycle *C* through the usual orthogonality to another cycle *C8*. This cycle is the reflection of the real line in *C*, see 3.3.5.

27b ⟨s-Inversion in cycle 27b⟩≡ (13b) 27c▷

```
C8 = real_line.cycle_similarity(C, es, diag_matrix(lst(1, sign1)), diag_matrix(lst(1,
                     jump_fnct(sign)))).normalize(n*k);
if (debug > 1)
 cout << "C8 is : " << C8 << endl;
```

Uses debug 17b, jump_fnct 56, and lst 17b.

We check that *C8* has common roots with *C*.

27c ⟨s-Inversion in cycle 27b⟩+≡ (13b) ◁27b 27d▷

```
cout << "C8 has common roots with C : "
        << (C8.val(lst(C.roots().op(0), 0)).is_zero()
             ∧ C8.val(lst(C.roots().op(1), 0)).is_zero()) << endl;
```

Uses lst 17b.

This chunk checks that centre of *C8* coincides with focus of *C*.

27d ⟨s-Inversion in cycle 27b⟩+≡ (13b) ◁27c 27e▷

```
cout << "H(s)-center of C8  coincides with s1-focus of C : "
  << (C8.center(diag_matrix(lst(-1,jump_fnct(sign))), true)
   -C.focus(diag_matrix(lst(-1, -sign1)), true)).evalm().normal().is_zero_matrix()
 << endl;
```

Uses jump_fnct 56 and lst 17b.

Finally we check that s-inversion in *C* defined through s-orthogonality coincides with inversion in *C8*.

27e ⟨s-Inversion in cycle 27b⟩+≡ (13b) ◁27d

```
P1 = C8.moebius_map(W, e, diag_matrix(lst(1, -jump_fnct(sign)))).subs(signs_cube,
        subs_options::algebraic | subs_options::no_pattern).normal();
cout << "s-Inversion in C coincides with inversion in C8 : "
  << C6.val(P1).normal().subs(signs_cube, subs_options::algebraic
                     | subs_options::no_pattern).normal().is_zero()
  << endl;
```

Uses jump_fnct 56 and lst 17b.

3.5 Distances and Lengths

3.5.1 Distances between points

We calculate several distances from the cycles.

The distance is given by the extremal value of diameters for all possible cycles passing through the both points [9, Defn. 5.2]. Thus we first construct a generic *cycle2d C10* passing through two points (u, v) and (u', v').

28a ⟨Distances from cycles 28a⟩≡ (13c) 28b▷

```
C10 = cycle2D(numeric(1), lst(l, n), m, e);
C10 = C10.subject_to(lst(C10.passing(W),
     C10.passing(W1)), lst(m, n, l));
if (debug > 1)
 cout << "   C10 is:      " << C10 << endl;
```

Uses cycle2D 10d 16c 16c 50a 52 58c 58c 58c 58c 58c 72 72 72 72 72 82c 82c 85a, debug 17b, lst 17b, and numeric 15a.

Then we calculate the square of its radius as the value of the determinant D. The point l of extremum *Len_c* is calculated from the condition $D'_l = 0$.

28b ⟨Distances from cycles 28a⟩+≡ (13c) ◁28a 28c▷

```
ex D = 4*C10.det(es);
Len_c = D.subs(lsolve(lst(D.diff(l) ≡ 0), lst(l))).normal();
```

Uses ex 6d 16b 58c 71 72 96 97a 97b 98a and lst 17b.

Now we check that *Len_c* is equal to [7, Lem. 5.5]

$$d^2(y, y') = \frac{\breve{\sigma}((u-u')^2 - \sigma(v-v')^2) + 4(1-\sigma\breve{\sigma})vv'}{(u-u')^2\breve{\sigma} - (v-v')^2}((u-u')^2 - \sigma(v-v')^2),$$

28c ⟨Distances from cycles 28a⟩+≡ (13c) ◁28b 29a▷

```
cout << "Distance between (u,v) and (u',v') in elliptic "
        "and hyperbolic spaces is "
 << endl << endl
 << "   s1*((u-u')^2-s*(v-v')^2)+4*(1-s*s1)*v*v')*"
        "((u-u')^2-s*(v-v')^2)"
 << endl
 << "   ------------------------------------------------------"
        "----------------         :  "
 << (Len_c-(sign1*(pow(u-u1,2)-sign*pow(v-v1,2))
     +4*(1-sign*sign1)*v*v1)*(pow(u-u1,2)
  -sign*pow(v-v1,2))÷(pow(u-u1,2)*sign1-pow(v-v1,2))).normal().is_zero() << endl
 <<"                  (u-u')^2*s1-(v-v')^2" << endl << endl;
```

Uses u 91a and v 91a.

We verify now the conformal property for this distance. To this end we need images *P2* and *P3* of (u, v) and (u', v') under the Möbius transformations.

29a ⟨Distances from cycles 28a⟩+≡ (13c) ◁28c 29b▷

```
P2 = clifford_moebius_map(sl2_clifford(a, b, c, d, e), W, e).subs(sl2_relation1,
 subs_options::algebraic | subs_options::no_pattern).normal();
P3 = clifford_moebius_map(sl2_clifford(a, b, c, d, e), lst(u+t*x, v+t*y), e).subs(sl2_relation1,
 subs_options::algebraic | subs_options::no_pattern).normal();
```

Uses `lst` 17b, `u` 91a, and `v` 91a.

Conformity is verified in the same chunk (see § 3.5.2) for this and all subsequent distances and lengths. Value *si* = -1 initiates conformality checks only in elliptic and hyperbolic point spaces.

29b ⟨Distances from cycles 28a⟩+≡ (13c) ◁29a 29c▷

```
si = -1;
⟨Check conformal property for various signs 31a⟩
C11 = C10.subs(lsolve(lst(D.diff(l) ≡ 0), lst(l)));
⟨Print perpendicular 31d⟩
```

Uses `lst` 17b and `si` 15b.

In parabolic space the extremal value is attained in the point $\frac{1}{2}(u + u1)$, since it separates upward-branched parabolas from down-branched.

29c ⟨Distances from cycles 28a⟩+≡ (13c) ◁29b 29d▷

```
Len_c = D.subs(lst(sign ≡0, l ≡ (u+u1)*half)).normal();
cout << "Value at the middle point (parabolic point space):" << endl
     << "          " << Len_c << endl;
```

Uses `lst` 17b and `u` 91a.

Value *si* = 0 initiates conformality checks only in the parabolic point space.

29d ⟨Distances from cycles 28a⟩+≡ (13c) ◁29c 29e▷

```
si = 0;
⟨Check conformal property for various signs 31a⟩
C11 = C10.subs(lst(sign ≡0, l ≡ (u+u1)*half));
⟨Print perpendicular 31d⟩
```

Uses `lst` 17b, `si` 15b, and `u` 91a.

We need to check the case $v = v'$ separately, since it is not covered by the above chunk. This is done almost identically to the previous case, with replacement of l by n, since the value of l is now fixed.

29e ⟨Distances from cycles 28a⟩+≡ (13c) ◁29d 30a▷

```
C10 = cycle2D(numeric(1), lst(l, n), m, e);
C10 = C10.subject_to(lst(C10.passing(W),
 C10.passing(lst(u1, v))));
if (debug > 1)
  cout << "     " << C10 << endl;
```

Uses `cycle2D` 10d 16c 16c 50a 52 58c 58c 58c 58c 58c 72 72 72 72 72 82c 82c 85a, `debug` 17b, `lst` 17b, `numeric` 15a, and `v` 91a.

This time the extremal point *n* is found from the condition $D'_n = 0$.

30a ⟨Distances from cycles 28a⟩+≡ (13c) ◁29e
```
D = 4*C10.det(es);
Len_c = D.subs(lsolve(lst(D.diff(n) == 0), lst(n))).normal();
cout << "Distance between (u,v) and (u',v): " << endl
    << "   Value at critical point:" << endl << "        "
    << Len_c << endl << endl;
```

Uses lst 17b, u 91a, and v 91a.

3.5.2 Check of the conformal property

We check conformal property of all distances and lengths. This is most time-consuming portion of the program and it took about five minutes on my computer. The rest is calculated within twenty seconds.

30b ⟨Check conformal property 30b⟩≡ (32b 33d)
```
⟨Evaluate the fraction 30c⟩
⟨Find the limit 30d⟩
cout << " This distance/length is conformal:";
⟨Check independence 30e⟩
cout << endl;
```

To this end we consider the ratio of distances between (u, v) and $(u + tx, v + ty)$ and between their images *P2* and *P3* under the generic Moebius transform."

30c ⟨Evaluate the fraction 30c⟩≡ (30b 31a)
```
Len_cD= (Len_c.subs(lst(u == P2.op(0), v==P2.op(1), u1 == P3.op(0),
  v1==P3.op(1)), subs_options::algebraic | subs_options::no_pattern)
  /Len_c.subs(lst(u1==u+t*x, v1==v+t*y), subs_options::algebraic | subs_options::no_pattern));
```

Uses lst 17b, u 91a, and v 91a.

If *Len_cD* has the variable *t*, we take the limit $t \to 0$ using the power series expansions.

30d ⟨Find the limit 30d⟩≡ (30b 31c)
```
if (Len_cD.has(t))
  Len_cD = Len_cD.series(t==0,1).op(0).normal();
```

The limit of this ratio for $t \to 0$ should be independent from (x, y) (see [7, Defn. 5.9]).

30e ⟨Check independence 30e⟩≡ (30b 31c)
```
cout << "  " << !(Len_cD.is_zero() || Len_cD.has(t) || Len_cD.has(x) || Len_cD.has(y));
if (debug > 0)
  cout << ". The factor is: " << endl << "   " << Len_cD;
```

Uses debug 17b.

This is a similar check to the previous one, we start from the same fraction.

31a ⟨Check conformal property for various signs 31a⟩≡ (29) 31b▷

```
⟨Evaluate the fraction 30c⟩
Len_fD = Len_cD;
cout << "  Conformity in a cycle space with metric:   "
     "E      P      H  " << endl;
```

Uses cycle 4 5a 6c 6c 6c 6c 6c 68a 68b 69c 69c 71 71 71 71 71.

However we make the substitution of all possible combinations of *sign* and *sign1* (an initial value of *si* should be set before in order to separate parabolic case from others) . . .

31b ⟨Check conformal property for various signs 31a⟩+≡ (29) ◁31a 31c▷

```
for (; si < 2; si+=2) {
 cout << "     Point space is " << eph_case(si) << ": ";
 for (si1 = -1; si1 < 2; si1++) {
  Len_cD = Len_fD.subs(lst(sign ≡ numeric(si), sign1 ≡ numeric(si1)),
       subs_options::algebraic | subs_options::no_pattern).normal();
```

Uses eph_case 56 99b, lst 17b, numeric 15a, si 15b, and si1 15b.

. . . and only then check the conformity (in order to keep the complexity of the expression under control)

31c ⟨Check conformal property for various signs 31a⟩+≡ (29) ◁31b

```
  ⟨Find the limit 30d⟩
  ⟨Check independence 30e⟩
 }
 cout << endl;
}
```

3.5.3 Calculation of Perpendiculars

Lengths define corresponding perpendicular conditions in terms of shortest routes, see [7, Defn. 5.12].

31d ⟨Print perpendicular 31d⟩≡ (29 32b 33d)

```
cout << " Perpendicular to ((u,v); (u′,v′)) is: "
 << (C11.get_l(1)+sign∗C11.get_k()∗v1).normal() << ";  "
 << (C11.get_l(0)-C11.get_k()∗u1).normal() << endl<<endl;
```

Uses u 91a and v 91a.

3.5.4 Length of intervals from centre

We calculate the lengths derived from the cycle with a *centre* at one point and passing through the second, see [7, Defn. 5.3].
We build a **cycle2D** *C11* which passes through (u', v') and has its centre at (u, v).

32a ⟨Lengths from centre 32a⟩≡ (13c) 32b▷

```
C11 = C.subject_to(lst(C.passing(W1), C.is_normalized()));
C11 = C11.subject_to(lst(C11.center().op(0) == u, C11.center().op(1)== v));
```

Uses lst 17b, u 91a, and v 91a.

Then the distance is radius the *C11*, see [7, Lem. 5.7.i]. We check conformity and calculate the perpendicular at the end.

32b ⟨Lengths from centre 32a⟩+≡ (13c) ◁32a

```
Len_c = C11.det(es).normal();
cout << "Length from *center* between (u,v) and (u1,v1):" << endl
     << "     " << Len_c << endl ;
⟨Check conformal property 30b⟩
⟨Print perpendicular 31d⟩
```

Uses u 91a and v 91a.

3.5.5 Length of intervals from focus

We calculate the length derived from the cycle with a *focus* at one point. To use the linear solver in GiNaC we need to replace the condition $C10.focus().op(1) \equiv v$ by hand-made value for the parameter n.

32c ⟨Lengths from focus 32c⟩≡ (13c) 33a▷

```
C10 = C.subject_to(lst(C.passing(W1), C.is_normalized()));
C11 = C10.subject_to(lst(C10.focus().op(0) == u,
    n == v-v1+sqrt(pow((v-v1), 2) +pow((u-u1), 2)-sign*pow(v1, 2))));
⟨Lengths from focus debug output 32d⟩
```

Uses lst 17b, u 91a, and v 91a.

The obtained expression are very cumbersome thus we output them only for the research and debugging purposes.

32d ⟨Lengths from focus debug output 32d⟩≡ (32c 33a)

```
if (debug > 0)
    cout << "Lengths from focus between (u,v) and (u1,v1):" << endl
         << "   " << C11.det(es).normal() << endl
         << "  (Check: C11 focus should be at (u, v) and is at ("
         << C11.focus().op(0).normal()
         << ", " << C11.focus().op(1).normal() << "))" << endl << endl;
```

Uses debug 17b, u 91a, and v 91a.

There are two suitable values of n which correspond upward and downward parabolas, which are expressed by plus or minus before the square root. This chunk check the same calculation for the second value.

33a ⟨Lengths from focus 32c⟩+≡ (13c) ◁32c 33b▷

```
C11 = C10.subject_to(lst(C10.focus().op(0) ≡ u,
    n ≡ v-v1-sqrt(pow((v-v1), 2) +pow((u-u1), 2)-sign*pow(v1, 2))));
⟨Lengths from focus debug output 32d⟩
```

Uses lst 17b, u 91a, and v 91a.

After the value of length was found in two previous chunks we master as simpler expression for it which utilises the focal length p of the parabola. Again to avoid non-linearity of equation, we first construct a desired cycle.

33b ⟨Lengths from focus 32c⟩+≡ (13c) ◁33a 34a▷

```
ex p = -(v1-v)+sqrt(pow((v1-v), 2) +pow((u1-u), 2)-sign*pow(v1, 2));
C11 = cycle2D(numeric(1), lst(u, p), 2*p*v1-pow(u1, 2)+2*u*u1+sign*pow(v1, 2), e);
⟨Make length output and checks 33c⟩
```

Uses cycle2D 10d 16c 16c 50a 52 58c 58c 58c 58c 58c 72 72 72 72 72 82c 82c 85a, ex 6d 16b 58c 71 72 96 97a 97b 98a, lst 17b, numeric 15a, u 91a, and v 91a.

And now we verify that the length is equal to $(1 - \sigma_1)p^2 - 2vp$, see [7, Lem. 5.7.ii].

33c ⟨Make length output and checks 33c⟩≡ (33b 34a) 33d▷

```
Len_c = C11.det(es).normal();
cout << "Length from *focus* between (u,v) and (u1,v1)"
      " is equal to (1-s1)p^2-2vp : "
    << (Len_c - (1-sign1)*pow(p, 2) + 2*v*p).subs(signs_cube,
        subs_options::algebraic | subs_options::no_pattern).expand().normal().is_zero()
    << endl;
```

Uses u 91a and v 91a.

and we check all requested properties for *C11*: it passes (*u1*,*v1*) and has focus at (*u*, *v*).

33d ⟨Make length output and checks 33c⟩+≡ (33b 34a) ◁33c

```
cout << "  checks: C11 goes through (u1, v1): "
    << C11.val(W1).normal().is_zero()
    << "; C11 focus is at (u, v): " << (C11.focus().op(0).normal()-u).is_zero()
    << endl;
⟨Check conformal property 30b⟩
⟨Print perpendicular 31d⟩
```

Uses u 91a and v 91a.

This chunk is similar to the previous one but checks the second parabola (the minus sign before the square root).

34a ⟨Lengths from focus 32c⟩+≡ (13c) ◁33b

```
p = -(v1-v)-sqrt(pow((v1-v), 2) +pow((u1-u), 2)-sign*pow(v1, 2));
C11 = cycle2D(numeric(1), lst(u, p), 2*p*v1-pow(u1,2)+2*u*u1+sign*pow(v1, 2), e);
⟨Make length output and checks 33c⟩
```

Uses cycle2D 10d 16c 16c 50a 52 58c 58c 58c 58c 58c 72 72 72 72 72 82c 82c 85a, lst 17b, numeric 15a, u 91a, and v 91a.

3.6 Infinitesimal Cycles

The final bit of our calculation is related with the infinitesimal radius cycles, see [7, § 5.3].

3.6.1 Basic properties of infinitesimal cycles

We define such a cycle *C10* bellow and check it radius (det) is an infinitesimal number, i.e. is *Order*(*eps*).

34b ⟨Infinitesimal cycle calculations 34b⟩≡ (13c) 34c ▷

```
C10 = cycle2D(1, lst(u, epsilon), pow(u,2)+2*epsilon*v-pow(epsilon,2), e);
cout << "Square of radius of the infinitesimal cycle is: "
 <<C10.det(es).subs(signs_cube, subs_options::algebraic
     | subs_options::no_pattern).series(epsilon≡0,1) << endl;
```

Uses cycle 4 5a 6c 6c 6c 6c 6c 68a 68b 69c 69c 71 71 71 71 71, cycle2D 10d 16c 16c 50a 52 58c 58c 58c 58c 58c 72 72 72 72 72 82c 82c 85a, lst 17b, u 91a, and v 91a.

Then we verify that in parabolic space it focus is in the point (u, v) and the focal length is an infinitesimal.

34c ⟨Infinitesimal cycle calculations 34b⟩+≡ (13c) ◁34b 35a ▷

```
cout << "Focus of infinitesimal cycle is: " <<C10.focus() << endl
 << "Focal length is: " <<C10.focal_length().series(epsilon≡0,1) << endl;

cout << "Infinitesimal cycle passing points (u+sqrt(eps)*x, v+"
 << lsolve(C10.subs(sign≡0).passing(lst(u+sqrt(epsilon)*x,v+y)),y).series(epsilon≡0,1)
 << "), " << endl;
```

Uses cycle 4 5a 6c 6c 6c 6c 6c 68a 68b 69c 69c 71 71 71 71 71, lst 17b, points 95b, u 91a, and v 91a.

3.6.2 Möbius transformations of infinitesimal cycles

Now we check that transformation of an infinitesimal cycle is an infinitesimal cycle again…

35a ⟨Infinitesimal cycle calculations 34b⟩+≡ (13c) ◁34c 35b▷

```
cout << "Image under SL2(R) of infinitesimal cycle "
        "has radius squared: "
  << C10.sl2_similarity(a, b, c, d, es).det(es).subs(signs_cube, subs_options::algebraic
            | subs_options::no_pattern).series(epsilon==0,1) << endl
  << "Image under cycle similarity of infinitesimal "
     "cycle has radius squared: "
  << C10.cycle_similarity(C, es).det(es).subs(signs_cube, subs_options::algebraic
            | subs_options::no_pattern).series(epsilon==0,1) << endl;
```

Uses cycle 4 5a 6c 6c 6c 6c 6c 68a 68b 69c 69c 71 71 71 71 71.

…and focus of the transformed cycle is (up to infinitesimals) obtained from the focus of initial cycle by the same transformation.

35b ⟨Infinitesimal cycle calculations 34b⟩+≡ (13c) ◁35a 35c▷

```
ex D = (ex_to<cycle2D>(C10.sl2_similarity(a, b, c, d, es)).focus(diag_matrix(lst(-1,1)), true)
  - clifford_moebius_map(sl2_clifford(a, b, c, d, es), W, es)).evalm();
cout << "Focus of the transormed cycle is from transformation"
        " of focus by: "
  << D.subs(sl2_relation, subs_options::algebraic
      | subs_options::no_pattern).subs(lst(sign1==0)).series(epsilon==0,1).normal() << endl;
```

Uses cycle 4 5a 6c 6c 6c 6c 6c 68a 68b 69c 69c 71 71 71 71 71, cycle2D 10d 16c 16c 50a 52 58c 58c 58c 58c 58c 72 72 72 72 72 82c 82c 85a, ex 6d 16b 58c 71 72 96 97a 97b 98a, and lst 17b.

3.6.3 Orthogonality with infinitesimal cycles

We also find expressions for the orthogonality (see § 3.3) with the infinitesimal radius cycle.

35c ⟨Infinitesimal cycle calculations 34b⟩+≡ (13c) ◁35b 35d▷

```
cout << "Orthogonality (leading term) to infinitesimal cycle is:"
     << endl << "         "
     << ex(C.is_orthogonal(C10, es)).series(epsilon==0,1) << endl;
```

Uses cycle 4 5a 6c 6c 6c 6c 6c 68a 68b 69c 69c 71 71 71 71 71 and ex 6d 16b 58c 71 72 96 97a 97b 98a.

And the both expressions for the s-orthogonality (see § 3.4) conditions with the infinitesimal radius cycle. The second relation verifies the Lem. 5.21 from [7].

35d ⟨Infinitesimal cycle calculations 34b⟩+≡ (13c) ◁35c 36a▷

```
cout << "s-Orthogonality of other cycle to infinitesimal:"
     << endl << "         "
     <<(C10.cycle_similarity(C, es).get_l(1)/n==0).series(epsilon==0,1).normal() << endl
     << "s-Orthogonality of infinitesimal cycle to other:"
     << endl << "         "
     << (C.cycle_similarity(C10, es).get_l(1)==0).series(epsilon==0,2).normal() << endl;
```

Uses cycle 4 5a 6c 6c 6c 6c 6c 68a 68b 69c 69c 71 71 71 71 71.

3.6.4 Cayley transform of infinitesimal cycles

We conclude with calculations of the parabolic Cayley transform [7, § 7.3] on infinitesimal radius cycles. The image is another infinitesimal radius cycle with its focus mapped by the Cayley transform.

36a ⟨Infinitesimal cycle calculations 34b⟩+≡ (13c) ◁35d

```
cout << "Det of Cayley-transformed infinitesimal cycle: "
 << parab_tr(C10).det().subs(lst(sign == 0),
     subs_options::algebraic | subs_options::no_pattern).series(epsilon==0,1) << endl
    << "Focus of the Cayley-transformed infinitesimal cycle: "
    << parab_tr(C10).focus().op(1).subs(lst(sign == 0),
   subs_options::algebraic | subs_options::no_pattern).series(epsilon==0,1) << endl;
```

Uses cycle 4 5a 6c 6c 6c 6c 6c 68a 68b 69c 69c 71 71 71 71 71 and lst 17b.

3.7 Drawing the Asymptote output

Although we use every possibility above to make double and cross checks one may still wish to see "by his own eyes" that the all calculations are correct. This may be done as follows.

We draw some Asymptote pictures which are included in [7], see also Fig. 1. We start from illustration of the both orthogonality relations, see § 3.3 and 3.4. They are done for nine ($= 3 \times 3$) possible combinations of metrics (elliptic, parabolic and hyperbolic) for the space of points and space of cycles.

36b ⟨Draw Asymptote pictures 36b⟩≡ (14d) 37a▷

```
ofstream asymptote("parab-ortho1.asy");
asymptote << setprecision(2);
for (si = -1; si < 2; si++) {
 for (si1 = -1; si1 < 2; si1++) {
  sign_val = lst(sign == si, sign1 == si1);
```

Uses lst 17b, si 15b, and si1 15b.

For each of those combinations we produce pictures from the set of data which is almost identical. This help to see the influence of *sign* and *sign1* parameters with constant other ones. All those graphics are mainly application of *asy_draw*() method (see § 2.6 mixed with some Asymptote drawing instructions. Since this is rather technical issue we put it separately in Appendix C.

37a ⟨Draw Asymptote pictures 36b⟩+≡ (14d) ◁36b 37b▷

```
try {
 {⟨Drawing first orthogonality 42⟩}
 {⟨Drawing second orthogonality 45⟩}
} catch (exception &p) {
 cerr ≪ "*****          Got problem2: " ≪ p.what() ≪ endl;
}
}
}
```

Defines:
catch, used in chunk 63b.

We finish the code with generation of some additional pictures for the paper [7].

37b ⟨Draw Asymptote pictures 36b⟩+≡ (14d) ◁37a

```
try {
 ⟨Extra pictures from Asymptote 46⟩
} catch (exception &p) {
 cerr ≪ "*****          Got problem3: " ≪ p.what() ≪ endl;
}
asymptote.close();
```

Defines:
catch, used in chunk 63b.

4 HOW TO GET THE CODE

1. Get the LaTeX source of this paper [9] from the arXiv.org.
2. Run the source through LaTeX. Five new files will be created in the current directory.
   (a) noweb [12] sources.
   (b) Header file cycle.h of the library.
   (c) C++ source cycle.cpp of the library.
   (d) C++ source parab-ortho1.cpp of the example.
   (e) Asymptote source of the graphics.
3. Use it on your own risk under the GNU General Public License [4].

A TEXTUAL OUTPUT OF THE PROGRAM

Here is the complete textual output of the program (with *debug*=0):

```
Conjugation of a cycle comes through Moebius
                transformation: true

A K-orbit is preserved: true, and  passing (0, t): true
 Determinant of zero-radius Z1 cycle in metric e is s1*v^2-s*v^2
 Focus of zero-radius cycle is {u,1/2*(-s1+s)*v}
 Centre of zero-radius cycle is {u,-s*v}
 Focal length of zero-radius cycle is 1/2*v
Image of the zero-radius cycle under Moebius transform
                has radius: 0
The centre of the Moebius transformed zero-radius cycle
                is: _equal_, _equal_
Image of the zero-radius cycle under cycle similarity
                has radius: 0
The centre of the conjugated zero-radius cycle coincides
        with Moebius tr: _equal_, _equal_

 The orthogonality is -2*l1*L+m1*k+m*k1+2*n1*s1*n==0
 The orthogonality of two lines is -2*l1*L+2*n1*s1*n==0
 The orthogonality to z-r-cycle is
                k*u^2+m+2*s1*n*v-2*L*u-k*s*v^2==0
 The orthogonality of two z-r-cycle is
        -s1*v^2+2*v'*s1*v-u'^2+u^2-v'^2*H(s2)-2*u'*u==0
Both orthogonal cycles (through one point and
                through its inverse) are the same: true
Orthogonal cycle goes through the transformed point: true

Line through point and its inverse is orthogonal: true
All lines come through the point (L*k^(-1), -k^(-1)*s1*n)
Conjugated vector is parallel to (u,v): true
C5 has common roots with C : true
H(s)-centre of C5 is equal to s1-centre of C: true
Inversion in (C5, sign)  coincides with inversion
                in (C, sign1): true
Inversion to the real line (with - sign):
   Conjugation of the real line is the cycle C: true
   Conjugation of the cycle C is the real line: true
   Inversion cycle has common roots with C: true
   C passing the centre of inversion cycle: true
Inversion to the real line (with + sign):
   Conjugation of the real line is the cycle C: true
   Conjugation of the cycle C is the real line: true
   Inversion cycle has common roots with C: true
   C passing the centre of inversion cycle: true
Yaglom inversion of the second kind is three
                reflections in the cycles: true
```

```
The real line is Moebius invariant: true
Reflection in the real line: (1, [[u,-v]].symbol2126, u^2-s*v^2)
 The s-orthogonality is (H(s2)*n1*L^2-m*H(s2)*n1*k+m1*H(s2)*k*n
       -2*H(s2)*l1*L*n+m*H(s2)*k1*n+H(s2)*n1*s1*n^2)*H(s2) = 0
 The s-orthogonality of two lines is (H(s2)*n1*L^2-2*H(s2)*l1*L*n
       +H(s2)*n1*s1*n^2)*H(s2)
 The s-orthogonality to z-r-cycle is first way: H(s2)*(-2*H(s2)*L*v*u
       -H(s2)*k*s1*v^3+2*H(s2)*s1*n*v^2+m*H(s2)*v+H(s2)*k*v*u^2) = 0
 The s-orthogonality to z-r-cycle is second way:
       H(s2)*(H(s2)*L^2*v+m*H(s2)*n-H(s2)*k*s1*n*v^2-2*H(s2)*L*n*u
       -m*H(s2)*k*v+H(s2)*k*n*u^2+H(s2)*s1*n^2*v) = 0
 The s-orthogonality of two z-r-cycle is
       H(s2)*(-H(s2)*s1*v^3+H(s2)*u'^2*v+H(s2)*v*u^2-2*H(s2)*u'*v*u
       -v'^2*H(s2)*s*v+2*v'*H(s2)*s1*v^2) = 0
All lines come through the focus related \breve{e}: true
C8 has common roots with C : true
H(s)-center of C8  coincides with s1-focus of C : true
s-Inversion in C coincides with inversion in C8 : true

Distance between (u,v) and (u',v') in elliptic and hyperbolic spaces is

s1*((u-u')^2-s*(v-v')^2)+4*(1-s*s1)*v*v')*((u-u')^2-s*(v-v')^2)
--------------------------------------------------------------- : true
             (u-u')^2*s1-(v-v')^2

  Conformity in a cycle space with metric:   E      P      H
    Point space is Elliptic case (sign = -1):  true false false
    Point space is Hyperbolic case (sign = 1):  false false true
 Perpendicular to ((u,v); (u',v')) is:
(-4*s1*u'*u+2*s1*u^2-2*v'^2+4*v'*v+2*s1*u'^2-2*v^2)^(-1)
*(-v'^3*s+3*v'^2*s*v-u'^2*v+2*v'*s1*u'^2*s-v'*u'^2
-3*v'*s*v^2-4*v'*s1*u'*s*u
-v'*u^2+2*u'*v*u+2*v'*s1*s*u^2+2*v'*u'*u+s*v^3-v*u^2);
(-4*s1*u'*u+2*s1*u^2-2*v'^2+4*v'*v+2*s1*u'^2-2*v^2)^(-1)
*(-2*v'^2*u-v'^2*s1*u'*s+v'^2*s1*s*u+s1*u^3-3*s1*u'*u^2+s1*u'*s*v^2
-s1*u'^3-s1*s*v^2*u+2*v'^2*u'-2*v'*u'*v+2*v'*v*u+3*s1*u'^2*u)

Value at the middle point (parabolic point space):
     u'^2+u^2-2*u'*u
  Conformity in a cycle space with metric:   E      P      H
    Point space is Parabolic case (sign = 0):  true true true
 Perpendicular to ((u,v); (u',v')) is: v'*s; -1/2*u'+1/2*u

Distance between (u,v) and (u',v):
   Value at critical point:
     (-2*s1*u'*u-4*s1*s*v^2+s1*u^2+s1*u'^2+4*v^2)*s1^(-1)

Length from *center* between (u,v) and (u1,v1):
   (-v'^2*s^3+u'^2*s^2-s1*v^2-2*u'*s^2*u+s^2*u^2+2*v'*s*v)*s^(-2)
 This distance/length is conformal: true
```

```
 Perpendicular to ((u,v); (u',v')) is: (v'*s^2-v)*s^(-1); -u'+u

Length from *focus* between (u,v) and (u1,v1) is
                equal to (1-s1)p^2-2vp : true
  checks: C11 goes through (u1, v1): true; C11 focus
                is at (u, v): true
 This distance/length is conformal: true
 Perpendicular to ((u,v); (u',v')) is:
-v'+sqrt(u'^2-v'^2*s+v'^2+u^2-2*v'*v-2*u'*u+v^2)+v'*s+v; -u'+u

Length from *focus* between (u,v) and (u1,v1) is
                equal to (1-s1)p^2-2vp : true
  checks: C11 goes through (u1, v1): true; C11 focus is
                at (u, v): true
 This distance/length is conformal: true
 Perpendicular to ((u,v); (u',v')) is:
-v'-sqrt(u'^2-v'^2*s+v'^2+u^2-2*v'*v-2*u'*u+v^2)+v'*s+v; -u'+u

Square of radius of the infinitesimal cycle is: Order(eps)
Focus of infinitesimal cycle is: {u,v}
Focal length is: Order(eps)
Infinitesimal cycle passing points
        (u+sqrt(eps)*x, v+(1/2*x^2)+Order(eps)),
Image under SL2(R) of infinitesimal cycle has radius squared:
        Order(eps)
Image under cycle similarity of infinitesimal cycle has radius
        squared:  Order(eps)
Focus of the transormed cycle is from transformation of
        focus by: ([[0],[0]])+Order(eps)
Orthogonality (leading term) to infinitesimal cycle is:
     (k*u^2+m-2*L*u==0)+Order(eps)
s-Orthogonality of other cycle to infinitesimal:
     (k*u^2+m-2*L*u==0)+Order(eps)
s-Orthogonality of infinitesimal cycle to other:
     (0==0)+(k*u^2+m-2*n*v-2*L*u==0)*eps+Order(eps^2)
Det of Cayley-transformed infinitesimal cycle: Order(eps)
Focus of the Cayley-transformed infinitesimal cycle:
        (-1/2-1/2*u^2+v)+Order(eps)
```

B EXAMPLE OF THE PRODUCED GRAPHICS

An example of graphics generated by the program is given in Figure 1.

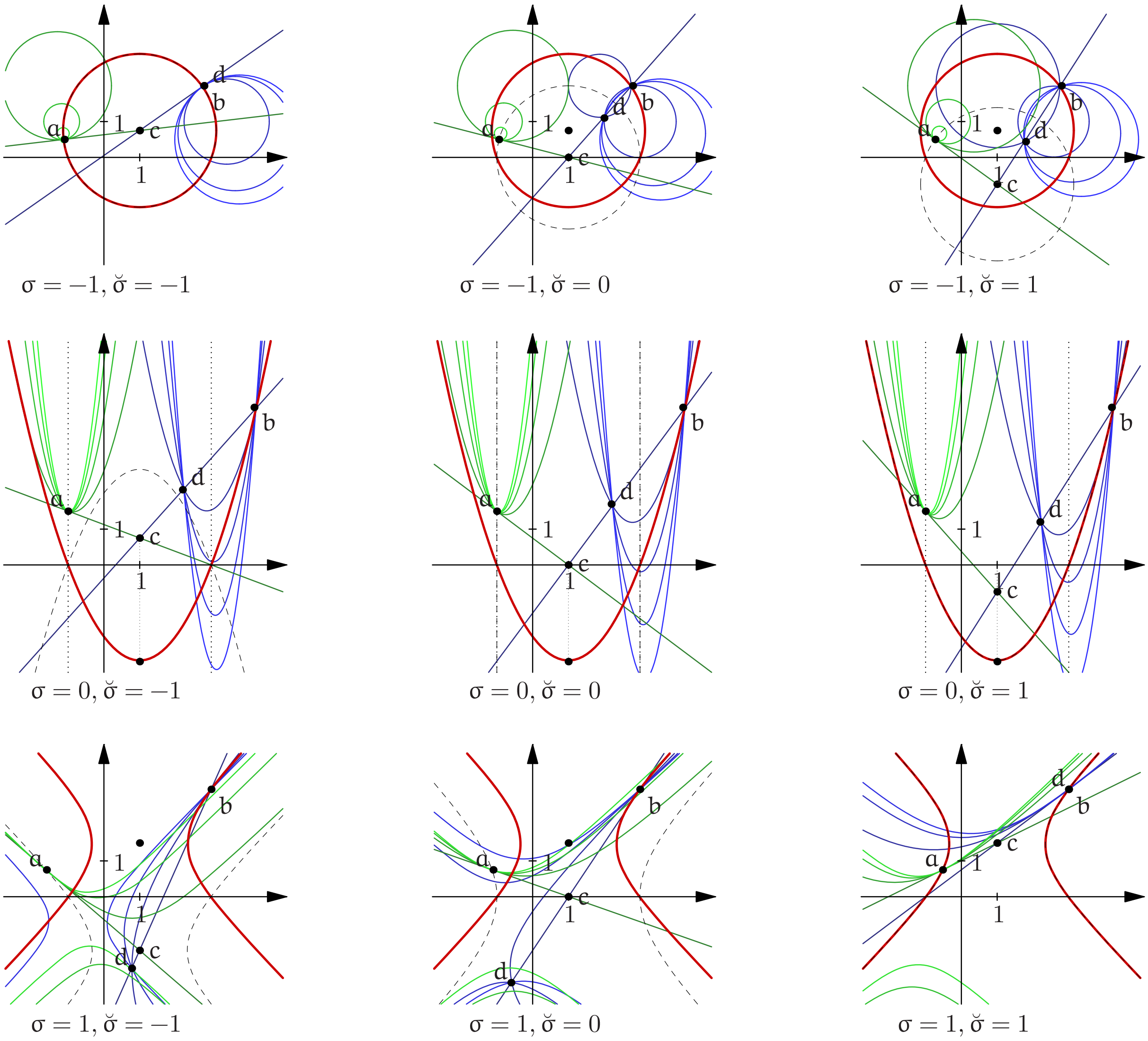

Figure 1: Orthogonality of the first kind in nine combinations.

C DETAILS OF THE ASYMPTOTE DRAWING

C.1 Drawing Orthogonality Conditions

C.1.1 First Orthogonality Condition

We define numeric values of all involved parameters first.

42 ⟨Drawing first orthogonality 42⟩≡ (37a)

```
numeric xmin(-11,4), xmax(5), ymin(-3), ymax = (si ≡ 0?numeric(25, 4): 4);
lst cycle_val = lst(sign ≡ numeric(si), sign1 ≡ numeric(si1),
                    k ≡ numeric(2,3), l ≡ numeric(2,3),
                    n ≡ (si ≡ 1?numeric(-1):numeric(1,2)), m ≡numeric(-2));
cycle2D Cf = C.subs(cycle_val), Cg = C5.subs(cycle_val), Cp =C2;
lst U, V;
switch (si) {
case -1: // points b, a, center, c, d
 U = numeric(11,4), Cg.roots(half).op(0), Cf.center().op(0).subs(cycle_val),
     (l÷k).subs(cycle_val);
 V = Cf.roots(U.op(0), false).op(1), half, Cf.center().op(1).subs(cycle_val),
  C4.roots(l÷k, false).op(0).normal().subs(cycle_val);
 break;
case 0:
 U = numeric(17,4), Cg.roots().op(0), Cf.center().op(0).subs(cycle_val),
     (l÷k).subs(cycle_val);
 V = Cf.roots(U.op(0), false).op(0), numeric(3,2),
     Cf.roots(l÷k, false).op(0).subs(cycle_val),
  C4.roots(l÷k, false).op(0).normal().subs(cycle_val);
 break;
case 1:
 U = numeric(12,4), Cg.roots(numeric(3,4)).op(0),
     Cf.center().op(0).subs(cycle_val), (l÷k).subs(cycle_val);
 V = Cf.roots(U.op(0), false).op(0), numeric(3,4),
     Cf.center().op(1).subs(cycle_val),
  C4.roots(l÷k, false).op(0).normal().subs(cycle_val);
 break;
}
U.append(P.op(0).subs(cycle_val).subs(lst(u ≡ U.op(0),
            v ≡ V.op(0))).normal()); // Moebius transform of the first point
V.append(P.op(1).subs(cycle_val).subs(lst(u ≡ U.op(0), v ≡ V.op(0))).normal());

asymptote ≪ endl ≪ "erase();" ≪ endl;
⟨Drawing orthogonal cycles 43a⟩
asymptote ≪ "shipout(\"first-ort-" ≪ eph_names[si+1]
         ≪ eph_names[si1+1] ≪ "\");" ≪ endl;
```

Uses cycle2D 10d 16c 16c 50a 52 58c 58c 58c 58c 58c 72 72 72 72 72 82c 82c 85a, eph_names 15a, lst 17b, numeric 15a, points 95b, si 15b, si1 15b, u 91a, and v 91a.

We start drawing from cycles.

43a ⟨Drawing orthogonal cycles 43a⟩≡ (42 45) 43b ▷

```
for (int j = 0; j<2; j++)
 for (int i=0; i<(si≡1?4:5); i++)
  Cp.subs(lst(k1 ≡ (si ≡ 0? numeric(3*i,2): numeric(i, 4)), n1 ≡ half, u ≡ U.op(j),
    v ≡ V.op(j))).subs(cycle_val).asy_draw(asymptote, xmin, xmax, ymin, ymax,
         lst(0.2, 0.2+j*(0.3+i÷8.0), 0.2+(1-j)*(0.3+i÷8.0)));

Cf.asy_draw(asymptote, xmin, xmax, ymin, ymax, lst(0.8, 0, 0), "+1");
Cg.asy_draw(asymptote, xmin, xmax, ymin, ymax, lst(0, 0, 0), "+0.3+dashed");
if (si ≡ 0)
 C5.subs(lst(sign ≡0, sign1≡0)).subs(cycle_val).asy_draw(asymptote, xmin, xmax, ymin, ymax,
                                                         lst(0, 0, 0), "+dotted");
```

Uses lst 17b, numeric 15a, si 15b, u 91a, and v 91a.

To finish we add some additional drawing explaining the picture.

43b ⟨Drawing orthogonal cycles 43a⟩+≡ (42 45) ◁43a

```
asymptote ≪ "pair[]  z={(" ≪ ex_to<numeric>(U.op(0).evalf()).to_double() ≪ ",  "
 ≪ ex_to<numeric>(V.op(0).evalf()).to_double() ≪ ")*u";
for (int j = 1; j<5; j++)
 asymptote ≪ ",  (" ≪ ex_to<numeric>(U.op(j).evalf()).to_double() ≪ ",  "
   ≪ ex_to<numeric>(V.op(j).evalf()).to_double() ≪ ")*u";

asymptote ≪ "};" ≪ endl  ≪ "   dot(z);" ≪ endl
 ≪ (si ≡ 0? "   draw((z[2].x,0)--z[2],  0.3+dotted);" : "") ≪ endl
 ≪ (si ≡ 0? "   draw((z[3].x,0)--z[3],  0.3+dotted);" : "") ≪ endl
 ≪ "   label(\"$a$\",  z[1],  NW);" ≪ endl
  ≪ "   label(\"$b$\",  z[0],  SE);" ≪ endl
 ≪ "   label(\"$c$\",  z[3],  E);" ≪ endl
  ≪ "   label" ≪ "(\"$d$\",  z[4],  " ≪ (si ≡1?"NW);":"NE);") ≪ endl;

⟨Put units 44⟩
⟨Draw axises 43c⟩
```

Uses numeric 15a, si 15b, and u 91a.

This chunk draws the standard coordinat axises.

43c ⟨Draw axises 43c⟩≡ (43b 47–54)

```
asymptote ≪ "   draw_axises((" ≪ xmin.to_double() ≪ ",  " ≪ ymin.to_double()
 ≪ "),  (  " ≪ xmax.to_double() ≪ ",  " ≪ ymax.to_double() ≪ "));" ≪ endl;
```

44 ⟨Put units 44⟩≡ (43b 53)

```
asymptote << "  label(\"$\\sigma=" << si << ", \\breve{\\sigma}=" << si1
  << "$\", (0, " << ymin.to_double() << ")*u, S);" << endl
  << "draw((1,-0.1)*u--(1,0.1)*u);" << endl
  << "draw((-0.1,1)*u--(0.1,1)*u);" << endl
  << "label(\"$1$\", (1,0)*u, S);" << endl
  << "label(\"$1$\", (0,1)*u, E);" << endl;
```

Uses si 15b, si1 15b, and u 91a.

C.1.2 Second Orthogonality Condition

We draw some `Asymptote` pictures to illustrate the second orthogonality relation. We define numeric values of all involved parameters first.

45 ⟨Drawing second orthogonality 45⟩≡ (37a)

```
numeric xmin(-11,4), xmax(5), ymin(-13,4), ymax = (si ≡ 0?numeric(6): numeric(15,4));
lst cycle_val = lst(sign ≡ numeric(si), sign1 ≡ numeric(si1), sign2 ≡ numeric(1),
    k ≡ numeric(2,3), l ≡ numeric(2,3), n ≡ (si ≡ 1?numeric(-4,3):half),
                              m ≡(si ≡ 1?numeric(-9,3):numeric(-2)));
cycle2D Cf = C.subs(cycle_val), Cg = C8.subs(cycle_val), Cp =C6;
lst U, V;
switch (si) {
case -1: // points b, a, center, c, d
 U = numeric(11,4), Cg.roots(half).op(0), Cf.focus().op(0).subs(cycle_val), (l÷k).subs(cycle_val);
 V = Cf.roots(U.op(0), false).op(1), half, Cf.focus().op(1).subs(cycle_val),
  C7.roots(l÷k, false).op(0).normal().subs(cycle_val);
 break;
case 0:
 U = numeric(4), Cf.roots().op(0), Cf.focus().op(0).subs(cycle_val), (l÷k).subs(cycle_val);
 V = Cf.roots(U.op(0), false).op(0), numeric(3,2), Cf.focus().op(0).subs(cycle_val),
  C7.roots(l÷k, false).op(0).normal().subs(cycle_val);
 break;
case 1:
 U = Cf.roots(numeric(1)).op(1), Cg.roots(numeric(6, 4)).op(1),
  Cf.focus().op(0).subs(cycle_val), (l÷k).subs(cycle_val);
 V = numeric(1), numeric(6, 4), Cf.focus().op(1).subs(cycle_val),
  C7.roots(l÷k, false).op(0).normal().subs(cycle_val);
 break;
 }
U.append(P1.op(0).subs(cycle_val).subs(lst(u ≡ U.op(0),
                     v ≡ V.op(0))).normal()); // Moebius transform of U.op(0)
V.append(P1.op(1).subs(cycle_val).subs(lst(u ≡ U.op(0), v ≡ V.op(0))).normal());

asymptote ≪ endl ≪ "erase();" //<< endl << "size(250);"
 ≪ endl;
⟨Drawing orthogonal cycles 43a⟩
asymptote ≪ "shipout(\"sec-ort-" ≪ eph_names[si+1] ≪ eph_names[si1+1]
          ≪ "\");" ≪ endl;
```

Uses `cycle2D` 10d 16c 16c 50a 52 58c 58c 58c 58c 58c 72 72 72 72 72 82c 82c 85a, `eph_names` 15a, `lst` 17b, `numeric` 15a, `points` 95b, `si` 15b, `si1` 15b, `u` 91a, and `v` 91a.

C.2 Extra pictures from **Asymptote**

We draw few more pictures in Asymptote.

46 ⟨Extra pictures from Asymptote 46⟩≡ (37b)
numeric *xmin*(-5), *xmax*(5), *ymin*(-13,4), *ymax* = **numeric**(6);
⟨Three images of the same cycle 47⟩
⟨Centres and foci of parabolas 48⟩
⟨Zero-radius cycle implementations 49⟩
⟨Parabolic diameters 50a⟩
⟨Distance as an extremum 51⟩
⟨Infinitesimal cycles draw 52⟩
⟨Cayley transform pictures 53⟩
⟨Three inversions 54a⟩

Uses numeric 15a.

C.2.1 Different implementations of the same cycle

A cycle represented by a four numbers $(k, l, n, m$ looks different in three spaces with different metrics.

47 ⟨Three images of the same cycle 47⟩≡ (46)

```
asymptote << endl << "erase();" << endl;
cycle2D C1f, C2f;
asymptote << "pair[]  z;";
for (int j = -1; j<2; j++) {
 C1f = cycle2D(1, lst(-2.5, 1), 3.75, diag_matrix(lst(-1, j)));
 C2f = cycle2D(1, lst(2.75, 3), 14.0625, diag_matrix(lst(-1, j)));
 C1f.asy_draw(asymptote, xmin, xmax, ymin, ymax, lst(0, 1.0-0.4*(j+1), 0.4*(j+1)),
              "+.75", true, 7);
 C2f.asy_draw(asymptote, xmin, xmax, ymin, ymax, lst(0, 1.0-0.4*(j+1), 0.4*(j+1)),
              "+.75", true, 7);
 asymptote << "z.push((" << C1f.center().op(0) << ",  " << C1f.center().op(1)
           << ")*u); z.push(("
           << C2f.center().op(0) << ",  " << C2f.center().op(1) << ")*u);" << endl;
}
asymptote << "z.push((" << C1f.roots().op(0) << ", 0)*u);  z.push(("
  << C1f.roots().op(1) << ", 0)*u);" << endl
  << " dot(z);" << endl
  << "  for (int j = 0; j<2; ++j) {"
  << "    label(\"$c_e$\", z[j], E);" << endl
  << "    label(\"$c_p$\", z[j+2], SE);" << endl
  << "    label(\"$c_h$\", z[j+4], E);" << endl
  << "    label((j==0?\"$r_0$\":\"$r_1$\"), "
             "z[j+6], (j==0? SW: SE));" << endl
  << "    draw(z[j]--z[j+4], .3+dashed);" << endl
  << "  }" << endl;
⟨Draw axises 43c⟩
asymptote << "shipout(\"same-cycle\");" << endl;
```

Uses cycle 4 5a 6c 6c 6c 6c 6c 68a 68b 69c 69c 71 71 71 71 71, cycle2D 10d 16c 16c 50a 52 58c 58c 58c 58c 58c 72 72 72 72 72 82c 82c 85a, lst 17b, and u 91a.

C.2.2 Centres and foci of cycles

We draw two parabolas and their centres with three type of foci.

48 ⟨Centres and foci of parabolas 48⟩≡ (46)

```
asymptote << endl << "erase();" << endl;
C1f = cycle2D(1, lst(-1.5, 2), 3.75, diag_matrix(lst(-1, 0)));
C2f = cycle2D(1, lst(2, 2), -3.5, diag_matrix(lst(-1, 0)));
C1f.asy_draw(asymptote, xmin, xmax, ymin, ymax, lst(0, 1.0-0.4, 0.4), "+.75", true, 7);
C2f.asy_draw(asymptote, xmin, xmax, ymin, ymax, lst(0, 1.0-0.4, 0.4), "+.75", true, 7);

asymptote << "pair[] z= {(" << C1f.center(diag_matrix(lst(-1,-1))).op(0) << ", "
    << C1f.center(diag_matrix(lst(-1,-1))).op(1) << ")*u,     ("
    << C2f.center(diag_matrix(lst(-1,-1))).op(0) << ", "
    << C2f.center(diag_matrix(lst(-1,-1))).op(1) << ")*u, ";
for (int j = -1; j<2; j++) {
 ex MS = diag_matrix(lst(-1, j));
 lst F1 = ex_to<lst>(C1f.focus(MS)), F2 = ex_to<lst>(C2f.focus(MS));
 asymptote << "     (" << F1.op(0) << ", " << F1.op(1) << ")*u, ("
   << F2.op(0) << ", " << F2.op(1) << ")*u" << (j==1? "};" : ", ") << endl;
}
asymptote << " dot (z);" << endl
 << " draw(z[0]--z[1], dashed);" << endl;

asymptote << "for (int j=1; j<3; ++j) {" << endl
 << "   label(\"$c_e$\", z[j-1], N);" << endl
 << "   label(\"$f_e$\", z[j+1], E);" << endl
 << "   label(\"$f_p$\", z[j+3], E);" << endl
 << "   label(\"$f_h$\", z[j+5], E);" << endl
 << " draw(z[j+1]--z[j+5], dotted+0.5);" << endl
 << "}" << endl;
⟨Draw axises 43c⟩
asymptote << "shipout(\"parab-cent\");" << endl;
```

Uses cycle2D 10d 16c 16c 50a 52 58c 58c 58c 58c 58c 72 72 72 72 72 82c 82c 85a, ex 6d 16b 58c 71 72 96 97a 97b 98a, lst 17b, and u 91a.

C.2.3 Zer-radius cycles

Zero-radius cycles can look different in different EPH realisations, here is an illustration.

49 ⟨Zero-radius cycle implementations 49⟩≡ (46)

```
asymptote << endl << "erase();" << endl
 << "pair[] z;" << endl;
{
 numeric xmin(-5), xmax(15), ymin(-5), ymax(5);
 for (int i1=-1; i1<2; i1++) {
  for(int i2=-1; i2<2; i2++) {
   lst val(sign==i1, sign1==i2, u==6*i1+4, v==1.7);
   Z1.subs(val).asy_draw(asymptote, xmin, xmax, ymin, ymax, lst(0.5+0.4*i1, .5-0.3*i2, 0.5+0.3*i2),
                                    "", true, 7);
   asymptote << "dot((" << ex_to<numeric>(Z1.focus(e).op(0).subs(val)).to_double()
       << ", "<< ex_to<numeric>(Z1.focus(e).op(1).subs(val)).to_double()
       << ")*u, " << 0.4+0.4*i1 << "red+"
       << .4-0.3*i2 << "green+"
       << 0.6+0.3*i2 << "blue);" << endl;
  }
 }
 ⟨Draw axises 43c⟩
}
asymptote << "shipout(\"zero-cycles\");" << endl;
```

Uses `lst` 17b, `numeric` 15a, `u` 91a, and `v` 91a.

C.2.4 Diameters of cycles

The notion of diameter and related distance became strange in parabolic case.

50a ⟨Parabolic diameters 50a⟩≡ (46)

```
asymptote << endl << "erase();" << endl;
C10 = cycle2D(1, lst((-4-1)÷2.0, 0.5), 4,diag_matrix(lst(-1, 0)));
C10.asy_draw(asymptote, xmin, xmax, ymin, ymax, lst(0.1, 0, 0.6));
asymptote << "pair[] z = {(" << C10.roots().op(0) << ", 0)*u, ("
        << C10.roots().op(1) << ", 0)*u};" << endl;
cycle2D(1, lst(5÷2.0, 0.5), 8,diag_matrix(lst(-1, 0))).asy_draw(asymptote,
        xmin, xmax, ymin, ymax, lst(0.1, 0.6, 0), "", true, 7);
C10 =cycle2D(-1, lst(-5÷2.0, 0.5), 8-5.0*5÷2.0,diag_matrix(lst(-1, 0)));
C10.asy_draw(asymptote, xmin, xmax, ymin, ymax, lst(0.1, 0.6, 0),
"+dashed ", true, 7);
asymptote << "z.push((" << C10.roots().op(1) << ", 0)*u); z.push(("
        << C10.roots().op(0) << ", 0)*u);" << endl;
⟨Put labels on 22-23 50b⟩
⟨Draw axises 43c⟩
asymptote << "shipout(\"parab-diam\");" << endl;
```

Defines:
cycle2D, used in chunks 10, 14a, 18b, 21b, 23–26, 28a, 29e, 33–35, 42, 45, 47, 48, 51, 53, 54, 58, 59a, 62a, 81–83, 91b, 92a, and 94b.
Uses lst 17b and u 91a.

Here is the common part of drawing points and labels on the figures 22-23.

50b ⟨Put labels on 22-23 50b⟩≡ (50a 51)

```
asymptote << "z.push((z[2].x,0)); z.push((z[3].x,0));" << endl
 << " dot(z);" << endl
 << " draw(z[2]--z[3], black+.3);" << endl
 << " draw(z[0]--z[1], black+1.2);" << endl
 << " draw(z[4]--z[5], black+1.2);" << endl
 << "  label(\"$z_1$\", z[0], NW);" << endl
 << "  label(\"$z_2$\", z[1], SE);" << endl
 << "  label(\"$z_3$\", z[2], SW);" << endl
 << "  label(\"$z_4$\", z[3], SE);" << endl;
```

C.2.5 Extramal property of the distance

To illustrate the variational definition of the distance [9, Defn.5.2] we draw several cycles which passes two given points. The cycles with the extremal value of diameter is highlighted in bold.

51 ⟨Distance as an extremum 51⟩≡ (46)

```
asymptote << endl << "erase();" << endl;
for (int j=-2; j < 3; j++) {
 ex_to<cycle2D>(C.subject_to(lst(C.passing(lst(xmin+1, ymax-5)),
                                 C.passing(lst(xmin+3, ymax-6.5)), k == 1,
      l == xmin+2+0.5*j)).subs(sign == -1)).asy_draw(asymptote,
         xmin, xmax, ymin, ymax,
            lst(0, 0.4*abs(j), 1.0-0.4*abs(j)), (j == 0 ? "+1" : "+.3"));
 ex_to<cycle2D>(C.subject_to(lst(C.passing(lst(xmax-4, ymax-5)),
         C.passing(lst(xmax-1, ymax-2)), k == 1,
      l == xmax-2.5-0.2*(j+2))).subs(sign == 0)).asy_draw(asymptote,
                          xmin, xmax, ymin, ymax,
             lst(0.2*(j+2), 0, 1.0-0.2*(j+2)), (j == -2 ? "+1" : "+.3"), true, 7);
}
asymptote << "pair[] z ={ (" << xmin+1 << ", " << ymax-5 << ")*u,  ("
 << xmin+3 << ", "
 << ymax-6.5 << ")*u,  (" << xmax-4 << ", " << ymax-5 << ")*u,  (" << xmax-1
 << ", " << ymax-2 << ")*u};" << endl;
⟨Put labels on 22-23 50b⟩
asymptote << "  label(\"$d_e$\", .5z[0]+.5z[1], NE);" << endl
  << "  label(\"$d_p$\", .5z[4]+.5z[5], S);" << endl;
⟨Draw axises 43c⟩
asymptote << "shipout(\"dist-extr\");" << endl;
```

Uses cycle2D 10d 16c 16c 50a 52 58c 58c 58c 58c 58c 72 72 72 72 72 82c 82c 85a, lst 17b, and u 91a.

C.2.6 Infinitesimal cycles

Here we draw a set of parabola with the same focus and the focal length tensing to zero.

52 ⟨Infinitesimal cycles draw 52⟩≡ (46)

```
asymptote << endl << "erase();"        << endl;
for (int j=1; j < 5; j++) {
 cycle2D(lst(-2.5, 4.5), diag_matrix(lst(-1,-1)), 16.0*pow(2, -2*j)).asy_draw(asymptote,
                         xmin, xmax, ymin, ymax,
                         lst(0, 0.2*abs(j), 1.0-0.2*abs(j)), "+.3");
 cycle2D(lst(1, 1.25), diag_matrix(lst(-1,1)), 25*pow(1.8, -2*j)).asy_draw(asymptote,
                         xmin, xmax, ymin, ymax/3,
                         lst(0.2*abs(j), 1.0-0.2*abs(j), 0), "+.3", true, 5+j);
 cycle2D(1, lst(2, pow(3,-j)), 2*2+2.0*pow(3,-j)-pow(3,-2*j), diag_matrix(lst(-1,0)))
     .asy_draw(asymptote, xmin,
               xmax, ymin, ymax, lst(1.0-0.17*j, 0, 0.17*j), "+.3", true, 7);
}
asymptote << " draw((2,1)*u--(2," << ymax << ")*u, blue+1);" << endl;
cycle2D(lst(1, 1.25), diag_matrix(lst(-1,1))).asy_draw(asymptote, xmin, xmax,
               ymin, ymax/3, lst(1, 0, 0), "+1");
asymptote << " dot((-2.5,4.5)*u);" << endl
  << " dot((2,1)*u);" << endl;
⟨Draw axises 43c⟩
asymptote << "shipout(\"infinites\");" << endl;
```

Defines:
cycle2D, used in chunks 10, 14a, 18b, 21b, 23–26, 28a, 29e, 33–35, 42, 45, 47, 48, 51, 53, 54, 58, 59a, 62a, 81–83, 91b, 92a, and 94b.
Uses lst 17b and u 91a.

C.2.7 Pictures of the Cayley transform

53 ⟨Cayley transform pictures 53⟩≡ (46)

```
xmin = -numeric(4,2); xmax=numeric(4,2); ymin=-numeric(3); ymax=numeric(7,2);
cycle2D C10f, C11f;
C10f = cycle2D(1, lst(0, sign2), sign, e);
for (si=-1; si<2; si++) {
 for (si1=-1; si1<2; si1++)
  if ((si ≡0 ) ∨ (si ≡ si1)) {
   asymptote ≪ endl ≪ "erase();" ≪ endl;
   for (int si2=-1; si2<2; si2=si2+2) {
    lst cycle_val = lst(sign ≡ si, sign1 ≡ si1, sign2≡si2);
    if (si ≠ 0 ) {
     C11f = C10f.subs(cycle_val,
          subs_options::algebraic | subs_options::no_pattern).normalize();
    }
    ex_to<cycle2D>((si?real_line.cycle_similarity(C11f, es).normalize().subs(cycle_val,
              subs_options::algebraic
                  | subs_options::no_pattern):parab_tr(real_line,sign1).subs(cycle_val,
                        subs_options::algebraic
                            | subs_options::no_pattern))).asy_draw(asymptote, xmin, xmax, ymin, ymax,
                                       lst(0, 0, 0.7), "+1.5", true, 7);
    if (si ≠ 0 ) {
     C11f.subs(cycle_val,
         subs_options::algebraic | subs_options::no_pattern).normalize().asy_draw(asymptote,
                                  xmin, xmax, ymin, ymax,
                            lst(0, 0.7, 0), (si2 ≡si1 ? "+.5" : "+.5+dotted "), true, 7);
    }
   }
   ⟨Put units 44⟩
     ⟨Draw axises 43c⟩
   asymptote ≪ "shipout(\"cayley-"≪ eph_names[si+1] ≪ eph_names[si1+1]
        ≪"\");" ≪ endl;
  }
}
```

Uses `cycle2D` 10d 16c 16c 50a 52 58c 58c 58c 58c 58c 72 72 72 72 72 82c 82c 85a, `eph_names` 15a, `lst` 17b, `numeric` 15a, `si` 15b, and `si1` 15b.

C.2.8 Three types of inversions

We draw here pictures for three types of the inversions. First we make a rectangular grid.

54a ⟨Three inversions 54a⟩≡ (46) 54b▷

```
xmin=-2; xmax=2; ymin=-2; ymax=2;
C2=cycle2D(lst(0,(1-abs(sign))÷2),e, 1);
C3=cycle2D(0,lst(l,n),m,e);
asymptote ≪ endl ≪ "erase(); u=1cm;" ≪ endl;
for(double i=-4; i≤4; i+=.4) {
 C3.subs(lst(sign≡-1, l≡0, n≡1, m≡i)).asy_draw(
  asymptote, xmin, xmax, ymin, ymax, lst(0.5, .75, 0.5),"+0.25pt", true, 7);
 C3.subs(lst(sign≡-1, l≡1, n≡0, m≡i)).asy_draw(
  asymptote, xmin, xmax, ymin, ymax, lst(0.5, .5, 0.75),"+0.25pt", true, 7);
}
C2.subs(sign≡-1).asy_draw(asymptote, xmin, xmax, ymin, ymax, lst(1,0,0),"+.75pt",
                                           true, 7);
⟨Draw axises 43c⟩
asymptote ≪ "shipout(\"pre-invers\");" ≪ endl;
```

Uses cycle2D 10d 16c 16c 50a 52 58c 58c 58c 58c 58c 72 72 72 72 72 82c 82c 85a, lst 17b, and u 91a.

Now we define inversions of the grid lines in the unit cycle and draw them for three different metrics.

54b ⟨Three inversions 54a⟩+≡ (46) ◁54a 54c▷

```
C4=C3.cycle_similarity(C2);
for(int si=-1; si<2; si++) {
 asymptote ≪ endl ≪ "erase();" ≪ endl;
 for(double i=-4; i≤4; i+=.4) {
  C4.subs(lst(sign≡si, l≡0, n≡1, m≡i)).asy_draw(
   asymptote, xmin, xmax, ymin, ymax, lst(0.5, .75, 0.5),"+0.25pt", true, 9);
  C4.subs(lst(sign≡si, l≡1, n≡0, m≡i)).asy_draw(
   asymptote, xmin, xmax, ymin, ymax, lst(0.5, .5, 0.75),"+0.25pt", true, 9);
 }
 C2.subs(sign≡si).asy_draw(asymptote, xmin, xmax, ymin, ymax, lst(1,0,0),"+.75pt",
                                           true, 7);
```

Uses lst 17b and si 15b.

We conclude by drawing the image of the cycle at infinity *Zinf*.

54c ⟨Three inversions 54a⟩+≡ (46) ◁54b

```
 ex_to<cycle2D>(Zinf.cycle_similarity(C2)).subs(sign≡si).asy_draw(
  asymptote, xmin, xmax, ymin, ymax, lst(0,0,1), (si≡-1? "+3pt": "+.75pt"));
 ⟨Draw axises 43c⟩
 asymptote ≪ "shipout(\"inversion-" ≪ eph_names[si+1] ≪ "\");" ≪ endl;
}
```

Uses cycle2D 10d 16c 16c 50a 52 58c 58c 58c 58c 58c 72 72 72 72 72 82c 82c 85a, eph_names 15a, lst 17b, and si 15b.

D THE IMPLEMENTATION THE CLASSES CYCLE AND CYCLE2D

This is the main file providing implementation the Classes **cycle** and **cycle2D**. It is not well documented yet.

D.1 Cycle and cycle2D classes header files

D.1.1 Cycle header file

This the header file describing the classes **cycle** and *cycle2d*. We start from the general inclusions and definitions and then defining those two classes.

55 ⟨cycle.h 55⟩≡

```
#include <stdexcept>
#include <ostream>
#include <sstream>
//#include <cmath>
 using namespace std;

#include <ginac/ginac.h>
 using namespace GiNaC;

#define CYCLELIB_MAJOR_VERSION 1
#define CYCLELIB_MINOR_VERSION 0
//const ex INFINITY;
⟨Auxiliary functions headers 56⟩
⟨cycle class 57a⟩
⟨cycle2D class 58a⟩
```

Defines:
  CYCLELIB_MAJOR_VERSION, never used.
  CYCLELIB_MINOR_VERSION, never used.
Uses ex 6d 16b 58c 71 72 96 97a 97b 98a.

D.1.2 Some auxillary functions

Here is the list of some auxiliary functions which are defined and used in the `cycle.h`.

56 ⟨Auxiliary functions headers 56⟩≡ (55)

```
/** Check of equality of two expression and report the string */
DECLARE_FUNCTION_1P(jump_fnct)

const char *equality(const ex & E);
inline const char *equality(const ex & E1, const ex & E2) { return equality(E1-E2); }
inline const char *equality(const ex & E, const ex & solns1, const ex & solns2)
{ ex e = E; return equality(e.subs(solns1), e.subs(solns2)); }

/** Return the string describing the case (elliptic, parabolic or hyperbolic) */
const char *eph_case(const numeric & sign);

/** Return even (real) part of a Clifford number */
inline ex scalar_part(const ex & e)
{ return remove_dirac_ONE(normal(canonicalize_clifford(e
                    + clifford_bar(e))))÷numeric(2); }

/** Return odd part of a Clifford number */
inline ex clifford_part(const ex & e)
{ return normal(canonicalize_clifford(e - clifford_bar(e)))÷numeric(2); }

/** Produces a Clifford matrix form of element of SL2 */
matrix sl2_clifford(const ex & a, const ex & b, const ex & c, const ex & d,
                    const ex & e, bool not_inverse=true);

matrix sl2_clifford(const ex & M, const ex & e, bool not_inverse=true);
```

Defines:
`eph_case`, used in chunk 31b.
`equality`, used in chunks 19d and 20b.
`jump_fnct`, used in chunks 15b, 23, 27, 83a, and 96–98.
Uses `ex` 6d 16b 58c 71 72 96 97a 97b 98a, `matrix` 15b 16a, and `numeric` 15a.

D.1.3 Members and methods in class cycle

The class **cycle** is derived from class **basic** in GiNaC according to the general guidelines given in the GiNaC tutorial. is defined through the general s

57a ⟨cycle class 57a⟩≡ (55)

```
/* * The class holding cycles kx^2-2<l,x>+m=0 */
class cycle : public basic
{
 GINAC_DECLARE_REGISTERED_CLASS(cycle, basic)

 ⟨cycle class constructors 4⟩
 ⟨service functions for class cycle 57b⟩
 ⟨accessing the data of a cycle 5d⟩
 ⟨specific methods of the class cycle 7a⟩
 ⟨Linear operation as cycle methods 6b⟩

protected:
 ex unit; // A Clifford unit to store the dimensionality and metric of the point space
 ex k;
 ex l;
 ex m;
};
 ⟨Linear operation on cycles 6c⟩
```

Uses cycle 4 5a 6c 6c 6c 6c 6c 68a 68b 69c 69c 71 71 71 71 71 and ex 6d 16b 58c 71 72 96 97a 97b 98a.

This is a set of the service functions which is required that a **cycle** is properly archived or printed to a stream.

57b ⟨service functions for class cycle 57b⟩≡ (57a)

```
// internal constructors
//cycle(const ex & k1, const ex & l1, const ex & m1, const ex & metr, const exvector & v,
// bool discardable = false);
//cycle(const ex & k1, const ex & l1, const ex & m1, const ex & metr, std::auto_ptr<exvector> vp);

protected:
 void do_print(const print_dflt & c, unsigned level) const;
 void do_print_dflt(const print_dflt & c, unsigned level) const;
 void do_print_latex(const print_latex & c, unsigned level) const;
```

Uses cycle 4 5a 6c 6c 6c 6c 6c 68a 68b 69c 69c 71 71 71 71 71, ex 6d 16b 58c 71 72 96 97a 97b 98a, and v 91a.

D.1.4 The derived class cycle2D for two dimensional cycles

We derive a derived class **cycle2D** from **cycle** in order to add some more methods which only make sense in two dimensions.

58a ⟨cycle2D class 58a⟩≡ (55)

```
class cycle2D : public cycle
{
 GINAC_DECLARE_REGISTERED_CLASS(cycle2D, cycle)

 ⟨constructors of the class cycle2D 10c⟩
 ⟨methods specific for class cycle2D 11a⟩
 ⟨duplicated methods for class cycle2D 58b⟩
};
 ⟨duplicated linear operation on cycle2D 58c⟩
```

Uses cycle 4 5a 6c 6c 6c 6c 6c 68a 68b 69c 69c 71 71 71 71 71 and cycle2D 10d 16c 16c 50a 52 58c 58c 58c 58c 58c 72 72 72 72 72 82c 82c 85a.

The general framework developed in the **cycle** class have some duplicates for two dimensions.

58b ⟨duplicated methods for class cycle2D 58b⟩≡ (58a)

```
inline cycle2D subs(const ex & e, unsigned options = 0) const {
 return ex_to<cycle2D>(inherited::subs(e, options)); }
inline cycle2D normalize(const ex & k_new = numeric(1), const ex & e = 0) const {
 return ex_to<cycle2D>(inherited::normalize(k_new, e)); }
inline cycle2D normalize_det(const ex & e = 0) const {
 return ex_to<cycle2D>(inherited::normalize_det(e)); }
inline cycle2D normal() const { return cycle2D(k.normal(), l.normal(), m.normal(),
unit.normal());}
```

Uses cycle2D 10d 16c 16c 50a 52 58c 58c 58c 58c 58c 72 72 72 72 72 82c 82c 85a, ex 6d 16b 58c 71 72 96 97a 97b 98a, and numeric 15a.

We also specialise for the derived class **cycle2D** all operations defined in § 2.3

58c ⟨duplicated linear operation on cycle2D 58c⟩≡ (58a)

```
const cycle2D operator+(const cycle2D & lh, const cycle2D & rh);
const cycle2D operator-(const cycle2D & lh, const cycle2D & rh);
const cycle2D operator*(const cycle2D & lh, const ex & rh);
const cycle2D operator*(const ex & lh, const cycle2D & rh);
const cycle2D operator÷(const cycle2D & lh, const ex & rh);
const ex operator*(const cycle2D & lh, const cycle2D & rh);
```

Defines:
 cycle2D, used in chunks 10, 14a, 18b, 21b, 23–26, 28a, 29e, 33–35, 42, 45, 47, 48, 51, 53, 54, 58, 59a, 62a, 81–83, 91b, 92a, and 94b.
 ex, used in chunks 4–12, 14a, 15c, 17a, 28b, 33b, 35, 48, 55–59, 62, 63, 65–68, 70, 73–89, 92b, 93a, 95a, and 98–100.

D.2 Implementation of the cycle class

We start from definitions of constructors in **cycle** class

59a ⟨cycle.cpp 59a⟩≡ 59b ▷

```
#include <cycle.h>
#define PRINT_CYCLE c.s << "("; \
 k.print(c, level); \
 c.s << ", "; \
 l.print(c, level); \
 c.s << ", "; \
 m.print(c, level); \
 c.s << ")";

 GINAC_IMPLEMENT_REGISTERED_CLASS_OPT(cycle, basic,
     print_func<print_dflt>(&cycle::do_print).
     print_func<print_latex>(&cycle::do_print_latex))

 GINAC_IMPLEMENT_REGISTERED_CLASS(cycle2D, cycle)
//,    print_func<print_dflt>(&cycle2D::do_print)

cycle::cycle() : unit(), k(), l(), m()
{
 tinfo_key = &cycle::tinfo_static;
}
```

Defines:
PRINT_CYCLE, used in chunk 69c.
Uses cycle 4 5a 6c 6c 6c 6c 6c 68a 68b 69c 69c 71 71 71 71 71 and cycle2D 10d 16c 16c 50a 52 58c 58c 58c 58c 58c 72 72 72 72 72 82c 82c 85a.

D.2.1 Main constructor of cycle from all parameters given

If all parameters of the cycle are given this constructor is used.

59b ⟨cycle.cpp 59a⟩+≡ ◁ 59a 60a ▷

```
cycle::cycle(const ex & k1, const ex & l1, const ex & m1, const ex & metr) // Main constructor
 : k(k1), m(m1)
{
 ex D, metric;
```

Uses cycle 4 5a 6c 6c 6c 6c 6c 68a 68b 69c 69c 71 71 71 71 71 and ex 6d 16b 58c 71 72 96 97a 97b 98a.

The first portion of the code processes various form of presentation for *l*.

60a ⟨cycle.cpp 59a⟩+≡ ◁59b 60b▷

```
if (is_a<indexed>(l1.simplify_indexed())) {
 l = ex_to<indexed>(l1.simplify_indexed());
 if (ex_to<indexed>(l).get_indices().size() ≡ 1) {
  D = ex_to<idx>(ex_to<indexed>(l).get_indices()[0]).get_dim();
 } else
  throw(std::invalid_argument("cycle::cycle(): the second parameter "
                              "should be an indexed object"
     "with one varindex"));
} else if (is_a<matrix>(l1) ∧ (min(ex_to<matrix>(l1).rows(),
                                   ex_to<matrix>(l1).cols()) ≡1)) {
 D = max(ex_to<matrix>(l1).rows(), ex_to<matrix>(l1).cols());
 l = indexed(l1, varidx((new symbol)→setflag(status_flags::dynallocated), D));
} else if (l1.info(info_flags::list) ∧ (l1.nops() > 0)) {
 D = l1.nops();
 l = indexed(matrix(1, l1.nops(), ex_to<lst>(l1)),
          varidx((new symbol)→setflag(status_flags::dynallocated), D));
```

Uses `cycle` 4 5a 6c 6c 6c 6c 6c 68a 68b 69c 69c 71 71 71 71 71, `lst` 17b, `matrix` 15b 16a, and `varidx` 15a.

If *l1* is zero we will try to get missing information from the metrix in the next chunk, otherwise throw an exception.

60b ⟨cycle.cpp 59a⟩+≡ ◁60a 61a▷

```
} else if (not l1.is_zero())
throw(std::invalid_argument("cycle::cycle(): the second parameter"
                            " should be an indexed object, "
                            "matrix or list"));
```

Uses `cycle` 4 5a 6c 6c 6c 6c 6c 68a 68b 69c 69c 71 71 71 71 71 and `matrix` 15b 16a.

Now we process the metric parameter, in case *l1* did not provide information on the dimensionality we try to get it here.

61a ⟨cycle.cpp 59a⟩+≡ ◁60b 61b▷
```
if (is_a<clifford>(metr)) {
  if (D.is_zero())
   D = ex_to<idx>(metr.op(1)).get_dim();
  unit = clifford_unit(varidx(0, D), ex_to<clifford>(metr).get_metric());
} else {
  if (D.is_zero()) {
   if (is_a<indexed>(metr))
    D = ex_to<idx>(metr.op(1)).get_dim();
   else if (is_a<matrix>(metr))
    D = ex_to<matrix>(metr).rows();
   else {
    exvector indices = metr.get_free_indices();
    if (indices.size() ≡ 2)
     D = ex_to<idx>(indices[0]).get_dim();
   }
  }
```

Uses `matrix` 15b 16a and `varidx` 15a.

For metric of unknown type we throw an exception.

61b ⟨cycle.cpp 59a⟩+≡ ◁61a 61c▷
```
  if (D.is_zero())
   throw(std::invalid_argument("cycle::cycle(): the metric should be"
          " either tensor, "
          "matrix, Clifford unit or indexed by to indices. "
          "Otherwise supply the through the second parameter."));

  unit = clifford_unit(varidx(0, D), metr);
}
```
Uses `cycle` 4 5a 6c 6c 6c 6c 6c 68a 68b 69c 69c 71 71 71 71 71, `matrix` 15b 16a, and `varidx` 15a.

Now we come back to the case *l1* is zero and try to resolve it with new info on *D*.

61c ⟨cycle.cpp 59a⟩+≡ ◁61b 62b▷
```
if (l1.is_zero()) {
  if (not D.is_zero())
   l = indexed(0, varidx((new symbol)→setflag(status_flags::dynallocated), D));
  else
   throw(std::invalid_argument("cycle::cycle(): the second argument is zero"
              " and the metric "
              "does not tell the dimensionality of space"));
  }
 ⟨Set tinfo to dimension 62a⟩
}
```

Uses `cycle` 4 5a 6c 6c 6c 6c 6c 68a 68b 69c 69c 71 71 71 71 71 and `varidx` 15a.

We set tinfo key for cycle according to its dimension

62a ⟨Set tinfo to dimension 62a⟩≡ (61c)
```
if (is_a<numeric>(D))
 switch (ex_to<numeric>(D).to_int()) {
 case 2:
  tinfo_key = &cycle2D::tinfo_static;
  break;
 default:
  tinfo_key = &cycle::tinfo_static;
  break;
 }
else
 tinfo_key = &cycle::tinfo_static;
```

Uses cycle 4 5a 6c 6c 6c 6c 6c 68a 68b 69c 69c 71 71 71 71 71, cycle2D 10d 16c 16c 50a 52 58c 58c 58c 58c 58c 72 72 72 72 72 82c 82c 85a, and numeric 15a.

D.2.2 Specific cycle constructors

Constructor for cycle with the given determinant *r_squared*, e.g. zero-radius cycle by default.

62b ⟨cycle.cpp 59a⟩+≡ ◁61c 62c▷
```
cycle::cycle(const lst & l, const ex & metr, const ex & r_squared, const ex & e,
                    const ex & sign)
{
 symbol m_temp;
 cycle C(numeric(1), l, m_temp, metr);
 (*this) = C.subject_to(lst(C.det(e, sign) ≡ r_squared), lst(m_temp));
}
```

Uses cycle 4 5a 6c 6c 6c 6c 6c 68a 68b 69c 69c 71 71 71 71 71, ex 6d 16b 58c 71 72 96 97a 97b 98a, lst 17b, and numeric 15a.

This is the constructor of a cycle identical to the given one with replaced metric in the point space.

62c ⟨cycle.cpp 59a⟩+≡ ◁62b 63a▷
```
cycle::cycle(const cycle & C, const ex & metr)
{
 (*this) = metr.is_zero()? C : cycle(C.get_k(), C.get_l(), C.get_m(), metr);
}
```

Uses cycle 4 5a 6c 6c 6c 6c 6c 68a 68b 69c 69c 71 71 71 71 71 and ex 6d 16b 58c 71 72 96 97a 97b 98a.

Constructor of a cycle from a matrix representations. First we check that matrix is in a proper form.

63a ⟨cycle.cpp 59a⟩+≡ ◁62c 63b▷

```
cycle::cycle(const matrix & M, const ex & metr, const ex & e, const ex & sign)
{
 if (not (M.rows() ≡ 2 ∧ M.cols() ≡ 2 ∧ (M.op(0)+M.op(3)).normal().is_zero()))
  throw(std::invalid_argument("cycle::cycle(): the second argument should "
                              "be square 2x2 matrix with M(1,1)=-M(2,2)"));

 ⟨Create a Clifford unit 65b⟩
 varidx i0((new symbol)→setflag(status_flags::dynallocated), D),
 i1((new symbol)→setflag(status_flags::dynallocated), D, true);
```

Uses cycle 4 5a 6c 6c 6c 6c 6c 68a 68b 69c 69c 71 71 71 71 71, ex 6d 16b 58c 71 72 96 97a 97b 98a, matrix 15b 16a, and varidx 15a.

There are different options for *sign*, which should be checked. First we verify is it zero and use the default value in this case.

63b ⟨cycle.cpp 59a⟩+≡ ◁63a 64▷

```
if (sign.is_zero()) {
 try {
  (∗this) = cycle(remove_dirac_ONE(M.op(2)), clifford_to_lst(M.op(0), e1),
                  remove_dirac_ONE(M.op(1)), metr);
 } catch (std::exception &p) {
  (∗this) = cycle(numeric(1), clifford_to_lst(M.op(0)∗clifford_inverse(M.op(2)), e1),
     canonicalize_clifford(M.op(1)∗clifford_inverse(M.op(2))), metr);
 }
} else {
 ex sign_m, conv;
 sign_m = sign.evalm();
```

Uses catch 14c 37a 37b 65a, cycle 4 5a 6c 6c 6c 6c 6c 68a 68b 69c 69c 71 71 71 71 71, ex 6d 16b 58c 71 72 96 97a 97b 98a, and numeric 15a.

If *sign* is not zero we process different types which can supply it.

64 ⟨cycle.cpp 59a⟩+≡ ◁63b 65a▷

```
if (is_a<tensor>(sign_m))
 conv = indexed(ex_to<tensor>(sign_m), i0, i1);
else if (is_a<clifford>(sign_m)) {
 if (ex_to<idx>(sign_m.op(1)).get_dim() ≡ D)
  conv = ex_to<clifford>(sign_m).get_metric(i0, i1);
 else
  throw(std::invalid_argument("cycle::cycle(): the sign should be "
         "a Clifford unit with "
         "the dimensionality matching to the second parameter"));
} else if (is_a<indexed>(sign_m)) {
 exvector ind = ex_to<indexed>(sign_m).get_indices();
 if ((ind.size() ≡ 2) ∧ (ex_to<idx>(ind[0]).get_dim() ≡ D)
      ∧ (ex_to<idx>(ind[1]).get_dim() ≡ D))
  conv = sign_m.subs(lst(ind[0] ≡ i0, ind[1] ≡ i1));
 else
  throw(std::invalid_argument("cycle::cycle(): the sign should be an "
                    "indexed object with two "
            "indices and their dimensionality matching "
            "to the second parameter"));
} else if (is_a<matrix>(sign_m)) {
 if ((ex_to<matrix>(sign_m).cols() ≡ D) ∧ (ex_to<matrix>(sign_m).rows() ≡ D))
  conv = indexed(ex_to<matrix>(sign_m), i0, i1);
 else
  throw(std::invalid_argument("cycle::cycle(): the sign should be "
                       "a square matrix with the "
      "dimensionality matching to the second parameter"));
} else
 throw(std::invalid_argument("cycle::cycle(): the sign should be"
                    " either tensor, indexed, matrix "
                    "or Clifford unit"));
```

Uses `cycle` 4 5a 6c 6c 6c 6c 6c 68a 68b 69c 69c 71 71 71 71 71, `lst` 17b, and `matrix` 15b 16a.

Then all blocks of the matrix are used to construct the cycle in main constructor.

65a ⟨cycle.cpp 59a⟩+≡ ◁64 66a▷

```
try {
 (*this) = cycle(remove_dirac_ONE(M.op(2)), indexed(matrix(1,
                              ex_to<numeric>(D).to_int(),
               clifford_to_lst(M.op(0), e1)), i0.toggle_variance())*conv,
                         remove_dirac_ONE(M.op(1)), metr);
} catch  (std::exception &p) {
 (*this) = cycle(numeric(1), indexed(matrix(1, ex_to<numeric>(D).to_int(),
                                                clifford_to_lst(M.op(0)
                  *clifford_inverse(M.op(2)), e1)), i0.toggle_variance())*conv,
    canonicalize_clifford(M.op(1)*clifford_inverse(M.op(2))), metr);
}
}
}
```

Defines:
`catch`, used in chunk 63b.
Uses `cycle` 4 5a 6c 6c 6c 6c 6c 68a 68b 69c 69c 71 71 71 71 71, `matrix` 15b 16a, and `numeric` 15a.

We need the proper Clifford unit to decompose M(0,0) element into vector for *l*.

65b ⟨Create a Clifford unit 65b⟩≡ (63a)

```
ex e1, D;
if (e.is_zero()) {
 ex metr1;
 if (is_a<matrix>(metr)) {
  D = ex_to<matrix>(metr).cols();
  metr1 = metr;
 } else if (is_a<clifford>(metr)) {
  D = ex_to<idx>(metr.op(1)).get_dim();
  metr1 = ex_to<clifford>(metr).get_metric();
 } else if (is_a<indexed>(metr)) {
  D = ex_to<idx>(ex_to<indexed>(metr).get_indices()[0]).get_dim();
  metr1 = metr;
 } else
  throw(std::invalid_argument("Could not determine the dimensionality "
                               "of point space "
     "from the supplied metric or Clifford unit"));

 e1 = clifford_unit(varidx((new symbol)->setflag(status_flags::dynallocated), D), metr1);
} else {
 e1 = e;
 D = ex_to<idx>(e.op(1)).get_dim();
}
```

Uses `ex` 6d 16b 58c 71 72 96 97a 97b 98a, `matrix` 15b 16a, and `varidx` 15a.

D.2.3 Class cycle members access

Class **cycle** has four operands.

66a ⟨cycle.cpp 59a⟩+≡ ◁65a 66b▷

```
ex cycle::op(size_t i) const
{
 GINAC_ASSERT(i<nops());

 switch (i) {
 case 0:
  return k;
 case 1:
  return l;
 case 2:
  return m;
 case 3:
  return unit;
 default:
  throw(std::invalid_argument("cycle::op(): requested operand out of "
                              "the range (4)"));
 }
}
```

Uses cycle 4 5a 6c 6c 6c 6c 6c 68a 68b 69c 69c 71 71 71 71 71 and ex 6d 16b 58c 71 72 96 97a 97b 98a.

Operands may be set through this method.

66b ⟨cycle.cpp 59a⟩+≡ ◁66a 67a▷

```
ex & cycle::let_op(size_t i)
{
 GINAC_ASSERT(i<nops());

 ensure_if_modifiable();
 switch (i) {
 case 0:
  return k;
 case 1:
  return l;
 case 2:
  return m;
 case 3:
  return unit;
 default:
  throw(std::invalid_argument("cycle::op(): requested operand out of "
                              "the range (4)"));
 }
}
```

Uses cycle 4 5a 6c 6c 6c 6c 6c 68a 68b 69c 69c 71 71 71 71 71 and ex 6d 16b 58c 71 72 96 97a 97b 98a.

Substitutions works as usual in GiNaC.

67a ⟨cycle.cpp 59a⟩+≡ ◁66b 67b▷

```
cycle cycle::subs(const ex & e, unsigned options) const
{
 exmap m;
 if (e.info(info_flags::list)) {
  lst l = ex_to<lst>(e);
  for (lst::const_iterator i = l.begin(); i != l.end(); ++i)
   m.insert(std::make_pair(i->op(0), i->op(1)));
 } else if (is_a<relational>(e)) {
  m.insert(std::make_pair(e.op(0), e.op(1)));
 } else
  throw(std::invalid_argument("cycle::subs(): the parameter should be a "
                              "relational or a lst"));

 return ex_to<cycle>(inherited::subs(m, options));
}
```

Uses cycle 4 5a 6c 6c 6c 6c 6c 68a 68b 69c 69c 71 71 71 71 71, ex 6d 16b 58c 71 72 96 97a 97b 98a, lst 17b, and relational 17b.

D.2.4 Service methods for the GiNaC infrastructure

Standard parts involving archiving, comparison and printing of the **cycle** class

67b ⟨cycle.cpp 59a⟩+≡ ◁67a 68a▷

```
cycle::cycle(const archive_node &n, lst &sym_lst) : inherited(n, sym_lst)
{
 n.find_ex("k-param", k, sym_lst);
 n.find_ex("l-param", l, sym_lst);
 n.find_ex("m-param", m, sym_lst);
 n.find_ex("unit", unit, sym_lst);
}
```

Uses cycle 4 5a 6c 6c 6c 6c 6c 68a 68b 69c 69c 71 71 71 71 71 and lst 17b.

Archiving routine.

68a ⟨cycle.cpp 59a⟩+≡ ◁67b 68b▷

```
void cycle::archive(archive_node &n) const
{
 inherited::archive(n);
 n.add_ex("k-param", k);
 n.add_ex("l-param", l);
 n.add_ex("m-param", m);
 n.add_ex("unit", unit);
}

ex cycle::unarchive(const archive_node &n, lst &sym_lst)
{
 return (new cycle(n, sym_lst))→setflag(status_flags::dynallocated);
}
```

Defines:
cycle, used in chunks 5–11, 13e, 16c, 18–22, 24, 26, 31a, 34–36, 47, 57–67, 69, 70, 73–82, and 86.
Uses ex 6d 16b 58c 71 72 96 97a 97b 98a and lst 17b.

Comparison of **cycle**s.

68b ⟨cycle.cpp 59a⟩+≡ ◁68a 69a▷

```
int cycle::compare_same_type(const basic &other) const
{
 const cycle &o = static_cast<const cycle &>(other);
 if ((unit ≡ o.unit) ∧ (l∗o.get_k() ≡ o.get_l()∗k) ∧ (m∗o.get_k() ≡ o.get_m()∗k))
  return 0;
 else if ((unit < o.unit)
   ∨ (l∗o.get_k() < o.get_l()∗k) ∨ (m∗o.get_k() < o.get_m()∗k))
  return -1;
 else
  return 1;
}
```

Defines:
cycle, used in chunks 5–11, 13e, 16c, 18–22, 24, 26, 31a, 34–36, 47, 57–67, 69, 70, 73–82, and 86.

Equality of **cycle**s.

69a ⟨cycle.cpp 59a⟩+≡ ◁68b 69b▷

```
bool cycle::is_equal(const basic & other) const
{
 if (not is_a<cycle>(other))
  return false;

 const cycle o = ex_to<cycle>(other);
 if (¬ (unit.is_equal(o.unit) ∧ (m∗o.get_k()-o.get_m()∗k).normal().is_zero()))
  return false;

 if (is_a<numeric>(get_dim())) {
  int D = ex_to<numeric>(get_dim()).to_int();
  for (int i=0; i<D; i++)
   if (¬ (get_l(i)∗o.get_k()-o.get_l(i)∗k).normal().is_zero())
    return false;
  return true;
 } else
  return (l∗o.get_k()).normal().is_equal((o.get_l()∗k).normal());
}
```

Uses `cycle` 4 5a 6c 6c 6c 6c 6c 68a 68b 69c 69c 71 71 71 71 71 and `numeric` 15a.

A **cycle** is zero if and only if its all components are zero

69b ⟨cycle.cpp 59a⟩+≡ ◁69a 69c▷

```
bool cycle::is_zero() const
{
 return (k.is_zero() ∧ l.is_zero() ∧ m.is_zero());
}
```

Uses `cycle` 4 5a 6c 6c 6c 6c 6c 68a 68b 69c 69c 71 71 71 71 71.

Printing of **cycle**s.

69c ⟨cycle.cpp 59a⟩+≡ ◁69b 70▷

```
void cycle::do_print(const print_dflt & c, unsigned level) const
{
 PRINT_CYCLE
}

void cycle::do_print_latex(const print_latex & c, unsigned level) const
{
 PRINT_CYCLE
}
```

Defines:
`cycle`, used in chunks 5–11, 13e, 16c, 18–22, 24, 26, 31a, 34–36, 47, 57–67, 69, 70, 73–82, and 86.
Uses `PRINT_CYCLE` 59a.

D.2.5 Linear operation on cycles

Here are linear operations on **cycle** defined as methods.

70 ⟨cycle.cpp 59a⟩+≡ ◁69c 71▷

```
cycle cycle::add(const cycle & rh) const
{
 return cycle(get_k()+rh.get_k(), get_l()+rh.get_l(get_l().op(1)),
    get_m()+rh.get_m(), get_metric());
}
cycle cycle::sub(const cycle & rh) const
{
 return cycle(get_k()-rh.get_k(), get_l()-rh.get_l(get_l().op(1)),
    get_m()-rh.get_m(), get_metric());
}
cycle cycle::exmul(const ex & rh) const
{
 return cycle(get_k()*rh, indexed((get_l().op(0)*rh).evalm(), get_l().op(1)),
    get_m()*rh, get_metric());
}
cycle cycle::div(const ex & rh) const
{
 return exmul(pow(rh, numeric(-1)));
}
```

Uses `cycle` 4 5a 6c 6c 6c 6c 6c 68a 68b 69c 69c 71 71 71 71 71, `ex` 6d 16b 58c 71 72 96 97a 97b 98a, and `numeric` 15a.

The same linear structure is represented in operators overloading.

71 ⟨cycle.cpp 59a⟩+≡ ◁70 72▷

```
const cycle operator+(const cycle & lh, const cycle & rh)
{
 return lh.add(rh);
}
const cycle operator-(const cycle & lh, const cycle & rh)
{
 return lh.sub(rh);
}
const cycle operator*(const cycle & lh, const ex & rh)
{
 return lh.exmul(rh);
}
const cycle operator*(const ex & lh, const cycle & rh)
{
 return rh.exmul(lh);
}
const cycle operator÷(const cycle & lh, const ex & rh)
{
 return lh.div(rh);
}
const ex operator*(const cycle & lh, const cycle & rh)
{
 return lh.mul(rh);
}
```

Defines:
`cycle`, used in chunks 5–11, 13e, 16c, 18–22, 24, 26, 31a, 34–36, 47, 57–67, 69, 70, 73–82, and 86.
`ex`, used in chunks 4–12, 14a, 15c, 17a, 28b, 33b, 35, 48, 55–59, 62, 63, 65–68, 70, 73–89, 92b, 93a, 95a, and 98–100.

We make a specialisation of these operation for **cycle2D** class as well.

72 ⟨cycle.cpp 59a⟩+≡ ◁71 73a▷

```
const cycle2D operator+(const cycle2D & lh, const cycle2D & rh)
{
 return ex_to<cycle2D>(lh.add(rh));
}
const cycle2D operator-(const cycle2D & lh, const cycle2D & rh)
{
 return ex_to<cycle2D>(lh.sub(rh));
}
const cycle2D operator*(const cycle2D & lh, const ex & rh)
{
 return ex_to<cycle2D>(lh.exmul(rh));
}
const cycle2D operator*(const ex & lh, const cycle2D & rh)
{
 return ex_to<cycle2D>(rh.exmul(lh));
}
const cycle2D operator÷(const cycle2D & lh, const ex & rh)
{
 return ex_to<cycle2D>(lh.div(rh));
}
const ex operator*(const cycle2D & lh, const cycle2D & rh)
{
 return ex_to<cycle2D>(lh.mul(rh));
}
```

Defines:

cycle2D, used in chunks 10, 14a, 18b, 21b, 23–26, 28a, 29e, 33–35, 42, 45, 47, 48, 51, 53, 54, 58, 59a, 62a, 81–83, 91b, 92a, and 94b.

ex, used in chunks 4–12, 14a, 15c, 17a, 28b, 33b, 35, 48, 55–59, 62, 63, 65–68, 70, 73–89, 92b, 93a, 95a, and 98–100.

D.2.6 Specific methods for cycle

We oftenly need to normalise cycles to get rid of ambiguity in their definition. This is typically by prescribing a value to k.

73a ⟨cycle.cpp 59a⟩+≡ ◁72 73b▷

```
cycle cycle::normalize(const ex & k_new, const ex & e) const
{
 ex ratio = 0;
 if (k_new.is_zero()) // Make the determinant equal 1
  ratio = sqrt(det(e));
 else { // First non-zero coefficient among k, m, l_0, l_1, ... is set to k_new
  if (¬k.is_zero())
   ratio = k÷k_new;
  else if (¬m.is_zero())
   ratio = m÷k_new;
  else {
   int D = ex_to<numeric>(get_dim()).to_int();
   for (int i=0; i<D; i++)
    if (¬l.subs(l.op(1) ≡ i).is_zero()) {
     ratio = l.subs(l.op(1) ≡ i)÷k_new;
     i = D;
    }
  }
 }
 if (ratio.is_zero()) // No normalisation is possible
  return (*this);

 if (ratio.is_zero())
  return (*this);
 return cycle(k÷ratio, indexed((l.op(0)÷ratio).evalm().normal(), l.op(1)), (m÷ratio).normal(),
                        get_metric());
}
```

Uses `cycle` 4 5a 6c 6c 6c 6c 6c 68a 68b 69c 69c 71 71 71 71 71, `ex` 6d 16b 58c 71 72 96 97a 97b 98a, and `numeric` 15a.

The normalisation to unit determinant

73b ⟨cycle.cpp 59a⟩+≡ ◁73a 74a▷

```
cycle cycle::normalize_det(const ex & e) const
{
 ex d = det(e);
 return (d.is_zero()? *this: normalize(k÷sqrt(d), e));
}
```

Uses `cycle` 4 5a 6c 6c 6c 6c 6c 68a 68b 69c 69c 71 71 71 71 71 and `ex` 6d 16b 58c 71 72 96 97a 97b 98a.

This methods returns a centre of the **cycle** depending from the provided metric.

74a ⟨cycle.cpp 59a⟩+≡ ◁73b 74b▷

```
ex cycle::center(const ex & metr, bool return_matrix) const
{
 if (is_a<numeric>(get_dim())) {
  ex e1, D = get_dim();
  if (metr.is_zero())
   e1 = unit;
  else {
   if (is_a<clifford>(metr)) {
    e1 = clifford_unit(varidx(0, D), ex_to<clifford>(metr).get_metric());
   } else if (is_a<indexed>(metr) ∨ is_a<matrix>(metr) ∨ is_a<tensor>(metr)) {
    e1 = clifford_unit(varidx(0, D), metr);
   } else
    throw(std::invalid_argument("cycle::center(): the metric should be "
                                "either tensor, indexed, "
                                "matrix or Clifford unit"));
  }
```

Uses cycle 4 5a 6c 6c 6c 6c 6c 68a 68b 69c 69c 71 71 71 71 71, ex 6d 16b 58c 71 72 96 97a 97b 98a, matrix 15b 16a, numeric 15a, and varidx 15a.

Finally, the centre is constructed for the cycle and given metric by the formula [7, Defn. 2.10]:

$$\left(-e_0^2\frac{l_0}{k}, -e_1^2\frac{l_1}{k}, \ldots, -e_{D-1}^2\frac{l_{D-1}}{k}\right)$$

74b ⟨cycle.cpp 59a⟩+≡ ◁74a 75▷

```
  lst c;
  for(int i=0; i<D; i++)
   if (k.is_zero())
    c.append(get_l(i));
   else
    c.append(-ex_to<clifford>(e1).get_metric(varidx(i, D), varidx(i, D))*get_l(i)÷k);
  return (return_matrix? (ex)matrix(ex_to<numeric>(D).to_int(), 1, c) : (ex)c);
 } else {
  return l÷k;
 }
}
```

Uses ex 6d 16b 58c 71 72 96 97a 97b 98a, lst 17b, matrix 15b 16a, numeric 15a, and varidx 15a.

D.2.7 Build cycle with given properties

We oftenly need **cycle**s with prescribed properties, e.g. when converting of **cycle**s to normalised form or matrix. This routine takes a system of linear equations with the **cycle** parameters and try to resolve it. The list of unknown parameters is either supplied or build automatically in a way suitable for most applications.

75 ⟨cycle.cpp 59a⟩+≡ ◁74b 76▷

```
cycle cycle::subject_to(const ex & condition, const ex & vars) const
{
 lst vars1;
 if (vars.info(info_flags::list) ∧(vars.nops() ≠ 0))
  vars1 = ex_to<lst>(vars);
 else if (is_a<symbol>(vars))
  vars1 = lst(vars);
 else if ((vars ≡ 0) ∨ (vars.nops() ≡ 0)) {
  if (is_a<symbol>(m))
   vars1.append(m);
  if (is_a<numeric>(get_dim()))
   for (int i = 0; i < ex_to<numeric>(get_dim()).to_double(); i++)
    if (is_a<symbol>(get_l(i)))
     vars1.append(get_l(i));
  if (is_a<symbol>(k))
   vars1.append(k);
  if (vars1.nops() ≡ 0)
   throw(std::invalid_argument("cycle::subject_to(): could not construct "
                               "the default list of parameters"));
 } else
  throw(std::invalid_argument("cycle::subject_to(): second parameter "
                              "should be a list of symbols"
                              " or a single symbol"));

 return subs(lsolve(condition.info(info_flags::relation_equal)? lst(condition) : condition,
      vars1), subs_options::algebraic | subs_options::no_pattern);
}
```

Uses cycle 4 5a 6c 6c 6c 6c 6c 68a 68b 69c 69c 71 71 71 71 71, ex 6d 16b 58c 71 72 96 97a 97b 98a, lst 17b, and numeric 15a.

D.2.8 Conversion of the cycle to the matrix form

This method is inverse to the constructor of the **cycle** from its matrix, see (2) and [7, § 3.1]. First, we process the supplied *e* to the standard form of the Clifford unit.

76 ⟨cycle.cpp 59a⟩+≡ ◁75 77a▷

```
matrix cycle::to_matrix(const ex & e, const ex & sign) const
{
 ex one, es, conv, D = get_dim();
 varidx i0((new symbol)→setflag(status_flags::dynallocated), D),
  i1((new symbol)→setflag(status_flags::dynallocated), D, true);
 if (e.is_zero()) {
  one = dirac_ONE();
  es = clifford_unit(i1.toggle_variance(), get_metric());
 } else if (is_a<clifford>(e)) {
  one = dirac_ONE(ex_to<clifford>(e).get_representation_label());
  es = e.subs(e.op(1) ≡ i1.toggle_variance());
 } else if (is_a<tensor>(e) ∨ is_a<indexed>(e) ∨ is_a<matrix>(e)) {
  one = dirac_ONE();
  es = clifford_unit(i1.toggle_variance(), e);
 } else
  throw(std::invalid_argument("cycle::to_matrix(): expect a "
                              "clifford number, matrix, tensor or "
                              "indexed as the first parameter"));
```

Uses cycle 4 5a 6c 6c 6c 6c 6c 68a 68b 69c 69c 71 71 71 71 71, ex 6d 16b 58c 71 72 96 97a 97b 98a, matrix 15b 16a, and varidx 15a.

Then we work out the sign, which should be used.

77a ⟨cycle.cpp 59a⟩+≡ ◁76 77b▷

```
ex sign_m = sign.evalm();

if (is_a<tensor>(sign_m))
 conv = indexed(ex_to<tensor>(sign_m), i0, i1);
else if (is_a<clifford>(sign_m)) {
 if (ex_to<idx>(sign_m.op(1)).get_dim() == D)
  conv = ex_to<clifford>(sign_m).get_metric(i0, i1);
 else
  throw(std::invalid_argument("cycle::to_matrix(): the sign "
                              "should be a Clifford unit with the "
     "dimensionality matching to the second parameter"));
} else if (is_a<indexed>(sign_m)) {
 exvector ind = ex_to<indexed>(sign_m).get_indices();
 if ((ind.size() == 2) && (ex_to<idx>(ind[0]).get_dim() == D)
     && (ex_to<idx>(ind[1]).get_dim() == D))
  conv = sign_m.subs(lst(ind[0] == i0, ind[1] == i1));
 else
  throw(std::invalid_argument("cycle::to_matrix(): the sign should "
                              "be an indexed object with two "
     "indices and their dimensionality matching "
     "to the second parameter"));
} else if (is_a<matrix>(sign_m)) {
 if ((ex_to<matrix>(sign_m).cols() == D)
     && (ex_to<matrix>(sign_m).rows() == D))
  conv = indexed(ex_to<matrix>(sign_m), i0, i1);
 else
  throw(std::invalid_argument("cycle::to_matrix(): the sign should "
                              "be a square matrix with the "
     "dimensionality matching to the second parameter"));
} else
 throw(std::invalid_argument("cycle::to_matrix(): the sign should "
                             "be either tensor, indexed, "
    "matrix or Clifford unit"));
```

Uses cycle 4 5a 6c 6c 6c 6c 6c 68a 68b 69c 69c 71 71 71 71 71, ex 6d 16b 58c 71 72 96 97a 97b 98a, lst 17b, and matrix 15b 16a.

When all components are ready the matrix is build in few lines.

77b ⟨cycle.cpp 59a⟩+≡ ◁77a 78▷

```
ex a00 = expand_dummy_sum(l.subs(ex_to<indexed>(l).get_indices()[0]
                                == i0.toggle_variance()) * conv * es);

return matrix(2, 2, lst(a00, m * one, k * one, -a00));
}
```

Uses ex 6d 16b 58c 71 72 96 97a 97b 98a, lst 17b, and matrix 15b 16a.

D.2.9 Calculation of a value of cycle at a point

This is used in the construction of a relational **cycle**::*passing* describing incidence of a point to cycle.

78 ⟨cycle.cpp 59a⟩+≡ ◁77b 79a▷

```
ex cycle::val(const ex & y) const
{
 ex y0, D = get_dim();
 varidx i0, i1;
 if (is_a<indexed>(y)) {
  i0 = ex_to<varidx>(ex_to<indexed>(y).get_indices()[0]);
  if ((ex_to<indexed>(y).get_indices().size() ≡ 1) ∧ (i0.get_dim() ≡ D)) {
   y0 = ex_to<indexed>(y);
   i1 = varidx((new symbol)→setflag(status_flags::dynallocated), D);
  } else
   throw(std::invalid_argument("cycle::val(): the second parameter "
                          "should be a indexed object with "
                          "one varindex"));
 } else if (y.info(info_flags::list) ∧ (y.nops() ≡ D)) {
  i0 = varidx((new symbol)→setflag(status_flags::dynallocated), D);
  i1 = varidx((new symbol)→setflag(status_flags::dynallocated), D);
  y0 = indexed(matrix(1, y.nops(), ex_to<lst>(y)), i0);
 } else if (is_a<matrix>(y) ∧ (min(ex_to<matrix>(y).rows(),
                              ex_to<matrix>(y).cols()) ≡1)
    ∧ (D ≡ max(ex_to<matrix>(y).rows(), ex_to<matrix>(y).cols()))) {
  i0 = varidx((new symbol)→setflag(status_flags::dynallocated), D);
  i1 = varidx((new symbol)→setflag(status_flags::dynallocated), D);
  y0 = indexed(y, i0);
 } else
  throw(std::invalid_argument("cycle::val(): the second parameter "
                             "should be a indexed object, "
     "matrix or list"));

 return expand_dummy_sum(-k*y0*y0.subs(i0 ≡ i1)*get_metric(i0.toggle_variance(),
                                                  i1.toggle_variance())
    - 2* l*y0.subs(i0 ≡ ex_to<varidx>(ex_to<indexed>(l).get_indices()[0])
                    .toggle_variance()) +m);
}
```

Uses `cycle` 4 5a 6c 6c 6c 6c 6c 68a 68b 69c 69c 71 71 71 71 71, `ex` 6d 16b 58c 71 72 96 97a 97b 98a, `lst` 17b, `matrix` 15b 16a, and `varidx` 15a.

D.2.10 Matrix methods for cycle

The method *det*() may be defined in several ways. An alternative to the present definition is

```
ex cycle::det(const ex & e = 0,

 const ex & sign = (new tensdelta)→setflag(status_flags::dynallocated)) const
{ex M = normalize().to_matrix(e, sign);

return remove_dirac_ONE(M.op(0)∗M.op(3)-M.op(1)∗M.op(2)) ; }
```

79a ⟨cycle.cpp 59a⟩+≡ ◁78 79b▷

```
ex cycle::det(const ex & e, const ex & sign, const ex & k_norm) const
{
 return remove_dirac_ONE((k_norm.is_zero()?∗this:normalize(k_norm))
                               .to_matrix(e, sign).determinant());
}
```

Uses cycle 4 5a 6c 6c 6c 6c 6c 68a 68b 69c 69c 71 71 71 71 71 and ex 6d 16b 58c 71 72 96 97a 97b 98a.

Multiplication of cycles in the matrix representations and their similarity with respect to elements of $SL_2(\mathbb{R})$ and other cycles.

79b ⟨cycle.cpp 59a⟩+≡ ◁79a 80a▷

```
matrix cycle::mul(const ex & C, const ex & e, const ex & sign, const ex & sign1) const
{
 if (is_a<cycle>(C))
  return ex_to<matrix>(canonicalize_clifford(to_matrix(e, sign)
                              .mul(ex_to<cycle>(C).to_matrix(e,
                                    sign1.is_zero()?sign:sign1))));
 else if (is_a<matrix>(C) ∧ (ex_to<matrix>(C).rows() ≡ 2)
         ∧ (ex_to<matrix>(C).cols() ≡ 2))
  return ex_to<matrix>(canonicalize_clifford(to_matrix(e, sign).mul(ex_to<matrix>(C))));
 else
  throw(std::invalid_argument("cycle::mul(): cannot multiply a cycle"
                              " by anything but a cycle "
                              "or 2x2 matrix"));
}
```

Uses cycle 4 5a 6c 6c 6c 6c 6c 68a 68b 69c 69c 71 71 71 71 71, ex 6d 16b 58c 71 72 96 97a 97b 98a, and matrix 15b 16a.

D.2.11 Actions of cycle as matrix

cycle in the matrix form can act on other objects, or matrices can acts on **cycle**. Any 2×2-matrix acts on a **cycle** by the similarity: $M : C \mapsto MCM^{-1}$.

80a ⟨cycle.cpp 59a⟩+≡ ◁79b 80b▷

```
cycle cycle::matrix_similarity(const ex & a, const ex & b, const ex & c,
                               const ex & d, const ex & e,
                               const ex & sign, bool not_inverse) const
{
 return cycle(ex_to<matrix>(canonicalize_clifford(matrix(2,2,lst(a, b, c, d))
                      .mul(mul(matrix(2,2,lst(d, -b, -c, a)), e, sign))
                         .evalm()).normal()), get_metric(), e, sign);
}
```

Uses cycle 4 5a 6c 6c 6c 6c 6c 68a 68b 69c 69c 71 71 71 71 71, ex 6d 16b 58c 71 72 96 97a 97b 98a, lst 17b, and matrix 15b 16a.

For elements of $SL_2(\mathbb{R})$ we have a specific method which make the proper “cliffordization” of the matrix first.

80b ⟨cycle.cpp 59a⟩+≡ ◁80a 80c▷

```
cycle cycle::sl2_similarity(const ex & a, const ex & b, const ex & c,
                            const ex & d, const ex & e, const ex & sign,
                            bool not_inverse) const
{
 return cycle(ex_to<matrix>(canonicalize_clifford(
      sl2_clifford(a, b, c, d, e.is_zero()?unit:e, not_inverse)
         .mul(mul(sl2_clifford(a, b, c, d, e.is_zero()?unit:e, ¬not_inverse), e, sign))
      .evalm().subs(c*b ≡ (d*a-1), subs_options::algebraic
             | subs_options::no_pattern)).normal()), get_metric(), e, sign);
}
```

Uses cycle 4 5a 6c 6c 6c 6c 6c 68a 68b 69c 69c 71 71 71 71 71, ex 6d 16b 58c 71 72 96 97a 97b 98a, and matrix 15b 16a.

80c ⟨cycle.cpp 59a⟩+≡ ◁80b 81a▷

```
cycle cycle::sl2_similarity(const ex & M, const ex & e, const ex & sign,
                            bool not_inverse) const
{
 if (is_a<matrix>(M) ∨ M.info(info_flags::list))
  return sl2_similarity(M.op(0), M.op(1), M.op(2), M.op(3), e, sign, not_inverse);
 else
  throw(std::invalid_argument("sl2_clifford(): expect a list or matrix"
                         " as the first parameter"));
}
```

Uses cycle 4 5a 6c 6c 6c 6c 6c 68a 68b 69c 69c 71 71 71 71 71, ex 6d 16b 58c 71 72 96 97a 97b 98a, and matrix 15b 16a.

cycle acts on other **cycle** by the similarity: $C : C_1 \mapsto CC_1C$, see [7, (4.8)].

81a ⟨cycle.cpp 59a⟩+≡ ◁80c 81b▷

```
cycle cycle::cycle_similarity(const cycle & C, const ex & e,
                              const ex & sign, const ex & sign1) const
{
 return cycle(ex_to<matrix>(canonicalize_clifford(C.mul(mul(C, e, sign,
                       sign1.is_zero()?sign:sign1), e,
                   sign1.is_zero()?sign:sign1))), get_metric(), e, sign);
}
```

Uses cycle 4 5a 6c 6c 6c 6c 6c 68a 68b 69c 69c 71 71 71 71 71, ex 6d 16b 58c 71 72 96 97a 97b 98a, and matrix 15b 16a.

D.3 Implementation of the cycle2D class

The derived class **cycle2D** for two dimensional cycles. Here constructors, archiving, and comparison come first.

81b ⟨cycle.cpp 59a⟩+≡ ◁81a 81c▷

```
cycle2D::cycle2D() : inherited()
{
 tinfo_key = &cycle2D::tinfo_static;
}
```

Uses cycle2D 10d 16c 16c 50a 52 58c 58c 58c 58c 58c 72 72 72 72 72 82c 82c 85a.

81c ⟨cycle.cpp 59a⟩+≡ ◁81b 82a▷

```
cycle2D::cycle2D(const ex & k1, const ex & l1, const ex & m1, const ex & metr)
 : inherited(k1, l1, m1, metr)
{
 if (get_dim() ≠ 2)
  throw(std::invalid_argument("cycle2D::cycle2D(): class cycle2D is "
                              "defined in two dimensions"));
 tinfo_key = &cycle2D::tinfo_static;
}
```

Uses cycle2D 10d 16c 16c 50a 52 58c 58c 58c 58c 58c 72 72 72 72 72 82c 82c 85a and ex 6d 16b 58c 71 72 96 97a 97b 98a.

82a ⟨cycle.cpp 59a⟩+≡ ◁81c 82b▷

```
cycle2D::cycle2D(const lst & l, const ex & r_squared, const ex & metr, const ex & e,
                 const ex & sign)
 : inherited(l, r_squared, metr, e, sign)
{
 if (get_dim() ≠ 2)
  throw(std::invalid_argument("cycle2D::cycle2D(): class cycle2D is "
                              "defined in two dimensions"));
 tinfo_key = &cycle2D::tinfo_static;
}
```

Uses cycle2D 10d 16c 16c 50a 52 58c 58c 58c 58c 58c 72 72 72 72 72 82c 82c 85a, ex 6d 16b 58c 71 72 96 97a 97b 98a, and lst 17b.

82b ⟨cycle.cpp 59a⟩+≡ ◁82a 82c▷

```
cycle2D::cycle2D(const cycle & C, const ex & metr)
{
 (*this) = cycle2D(C.get_k(), C.get_l(), C.get_m(), (metr.is_zero()? C.get_metric(): metr));
}
```

Uses cycle 4 5a 6c 6c 6c 6c 6c 68a 68b 69c 69c 71 71 71 71 71, cycle2D 10d 16c 16c 50a 52 58c 58c 58c 58c 58c 72 72 72 72 72 82c 82c 85a, and ex 6d 16b 58c 71 72 96 97a 97b 98a.

82c ⟨cycle.cpp 59a⟩+≡ ◁82b 83a▷

```
void cycle2D::archive(archive_node &n) const
{
 inherited::archive(n);
}

cycle2D::cycle2D(const archive_node &n, lst &sym_lst) : inherited(n, sym_lst) {; }

ex cycle2D::unarchive(const archive_node &n, lst &sym_lst)
{
 return (new cycle2D(n, sym_lst))→setflag(status_flags::dynallocated);
}

int cycle2D::compare_same_type(const basic &other) const
{
 return inherited::compare_same_type(other);
}
```

Defines:
cycle2D, used in chunks 10, 14a, 18b, 21b, 23–26, 28a, 29e, 33–35, 42, 45, 47, 48, 51, 53, 54, 58, 59a, 62a, 81–83, 91b, 92a, and 94b.
Uses ex 6d 16b 58c 71 72 96 97a 97b 98a and lst 17b.

D.3.1 The member functions of the derived class cycle2D

The standard definition of the focus for a parabola is

$$\left(\frac{l}{k}, \frac{m}{2n} - \frac{l^2}{2nk} + \frac{n}{2k}\right).$$

We calculate focus of a cycle based on its determinant in the corresponding metric.

83a ⟨cycle.cpp 59a⟩+≡ ◁82c 83b▷

```
ex cycle2D::focus(const ex & e, bool return_matrix) const
{
 lst f=lst(jump_fnct(-get_metric(varidx(0, 2), varidx(0, 2)))*get_l(0)/k,
     (-det(e, (new tensdelta)->setflag(status_flags::dynallocated), k)/(2*get_l(1)*k)).normal());
 return (return_matrix ? (ex)matrix(2, 1, f) : (ex)f);
}
```

Uses cycle2D 10d 16c 16c 50a 52 58c 58c 58c 58c 58c 72 72 72 72 72 82c 82c 85a, ex 6d 16b 58c 71 72 96 97a 97b 98a, jump_fnct 56, lst 17b, matrix 15b 16a, and varidx 15a.

83b ⟨cycle.cpp 59a⟩+≡ ◁83a 83c▷

```
lst cycle2D::roots(const ex & y, bool first) const
{
 ex D = get_dim();
 lst k_sign = lst(-k*get_metric(varidx(0, D), varidx(0, D)), -k*get_metric(varidx(1, D),
                                                                 varidx(1, D)));
 int i0 = (first?0:1), i1 = (first?1:0);
 ex c = k_sign.op(i1)*pow(y, 2) - numeric(2)*get_l(i1)*y+m;
 if (k_sign.op(i0).is_zero())
  return (get_l(i0).is_zero() ? lst() : lst(c/get_l(i0)/numeric(2)));
 else {
  ex disc = sqrt(pow(get_l(i0), 2) - k_sign.op(i0)*c);
  return lst((get_l(i0)-disc)/k_sign.op(i0), (get_l(i0)+disc)/k_sign.op(i0));
 }
}
```

Uses cycle2D 10d 16c 16c 50a 52 58c 58c 58c 58c 58c 72 72 72 72 72 82c 82c 85a, ex 6d 16b 58c 71 72 96 97a 97b 98a, lst 17b, numeric 15a, and varidx 15a.

83c ⟨cycle.cpp 59a⟩+≡ ◁83b 84a▷

```
lst cycle2D::line_intersect(const ex & a, const ex & b) const
{
 ex D = get_dim();
 ex pm = -k*get_metric(varidx(1, D), varidx(1, D));
 return cycle2D(k*(numeric(1)+pm*pow(a,2)).normal(),
                lst((get_l(0)+get_l(1)*a-pm*a*b).normal(), 0),
                (m-2*get_l(1)*b+pm*pow(b,2)).normal()).roots();
}
```

Uses cycle2D 10d 16c 16c 50a 52 58c 58c 58c 58c 58c 72 72 72 72 72 82c 82c 85a, ex 6d 16b 58c 71 72 96 97a 97b 98a, lst 17b, numeric 15a, and varidx 15a.

D.3.2 Drawing cycle2D

Some auxilliary functions used for drawing

84a ⟨cycle.cpp 59a⟩+≡ ◁83c 84b▷

```
inline ex max(const ex &a, const ex &b)
{return ex_to<numeric>((a-b).evalf()).is_positive()?a:b;}
inline ex min(const ex &a, const ex &b)
{return ex_to<numeric>((a-b).evalf()).is_positive()?b:a;}
```

Uses ex 6d 16b 58c 71 72 96 97a 97b 98a and numeric 15a.

The most complicated member function in the class **cycle2D**

84b ⟨cycle.cpp 59a⟩+≡ ◁84a 85a▷

```
#define PAIR(X, Y)  ex_to<numeric>((X).evalf()).to_double() \
              << "," <<            \
  ex_to<numeric>((Y).evalf()).to_double()
#define DRAW_ARC(X, S) \
   u = ex_to<numeric>((X).evalf()).to_double(); \
   v = ex_to<numeric>(roots(X, ¬not_swapped).op(zero_or_one)\
                     .evalf()).to_double();              \
   du = dir*(-k_d*signv*v+lv);   \
   dv = dir*(k_d*signu*u-lu);    \
   if (not_swapped)        \
    ost << S <<  u << "," << v << ")*u{" << du << "," << dv << "}"; \
   else          \
    ost << S <<  v << "," << u << ")*u{" << (sign ≡ 0? dv : -dv) \
          << "," << (sign ≡ 0? du : -du) << "}";
```

Defines:
DRAW_ARC, used in chunk 95b.
PAIR, used in chunks 88b, 90, and 94a.
Uses du 91a, dv 91a, k_d 91a, numeric 15a, u 91a, v 91a, and zero_or_one 91a.

The main drawing routine for **cycle2D**.

85a ⟨cycle.cpp 59a⟩+≡ ◁84b 85b▷

```
void cycle2D::metapost_draw(ostream & ost, const ex & xmin, const ex & xmax,
                 const ex & ymin, const ex & ymax,
                 const lst & color, const char * more_options, bool with_header,
                 int points_per_arc, bool asymptote, char * picture) const
{
 ex D = get_dim();
 ostringstream draw_start, draw_options;
 draw_start << "\tdraw" << (asymptote ? "(" : "  ")
            << picture << (int(*picture)==0? "" : ",") << "(";
 ios_base::fmtflags keep_flags = ost.flags(); // Keep stream's flags to be restored on the exit
 draw_options.flags(keep_flags); // Synchronise flags between the streams
 draw_options.precision(ost.precision()); // Synchronise flags between the streams
```

Defines:
cycle2D, used in chunks 10, 14a, 18b, 21b, 23–26, 28a, 29e, 33–35, 42, 45, 47, 48, 51, 53, 54, 58, 59a, 62a, 81–83, 91b, 92a, and 94b.
Uses ex 6d 16b 58c 71 72 96 97a 97b 98a and lst 17b.

Each drawing command is concluded by options containing color, etc. They are formatted differently for Asymptote and MetaPost.

85b ⟨cycle.cpp 59a⟩+≡ ◁85a 86a▷

```
 ost << fixed;
 draw_options << fixed;
 if (color.nops() == 3)
  if (asymptote)
   draw_options << ",rgb("
      << ex_to<numeric>(color.op(0)).to_double() << ","
      << ex_to<numeric>(color.op(1)).to_double() <<","
      << ex_to<numeric>(color.op(2)).to_double() << ")";
  else
   draw_options << showpos << " withcolor "
      << ex_to<numeric>(color.op(0)).to_double() << "*red"
      << ex_to<numeric>(color.op(1)).to_double() <<"*green"
      << ex_to<numeric>(color.op(2)).to_double() << "*blue ";

 draw_options << more_options << (asymptote ? ");" : ";") << endl;
```

Uses numeric 15a.

A drawing command can be also preceded by a human-readable comment describing the cycle to be drawn.

86a ⟨cycle.cpp 59a⟩+≡ ◁85b 86b▷

```
if (with_header)
 ost << (asymptote ? "\t// Asymptote" : "\t% Metapost") << " data in ["
     << xmin << "," << xmax << "]x[" << ymin << ","
     << ymax << "] for " << (ex)passing(lst(symbol("u"), symbol("v")));

if (k.is_zero() ∧ l.subs(l.op(1) ≡ 0).is_zero() ∧ l.subs(l.op(1) ≡ 1).is_zero() ∧ m.is_zero()) {
 ost << " zero cycle, (whole plane) " << endl;
 ost.flags(keep_flags);
 return;
}
```

Uses cycle 4 5a 6c 6c 6c 6c 6c 68a 68b 69c 69c 71 71 71 71 71, ex 6d 16b 58c 71 72 96 97a 97b 98a, lst 17b, u 91a, and v 91a.

There are several parameters which control the output. Their values depend from either we draw **cycle** in the original coordinates or swap the u and v

86b ⟨cycle.cpp 59a⟩+≡ ◁86a 87a▷

```
ex xc = center().op(0), yc = center().op(1); // the center of cycle
double sign0 = ex_to<numeric>(-get_metric(varidx(0, D), varidx(0, D)).evalf()).to_double(),
 sign1 = ex_to<numeric>(-get_metric(varidx(1, D), varidx(1, D)).evalf()).to_double(),
 sign = sign0 ∗ sign1;
numeric determinant = ex_to<numeric>(det().evalf());
bool not_swapped = (sign>0 ∨ sign1≡0 ∨ ((sign <0) ∧ ¬determinant.is_positive()));
double signu = (not_swapped?sign0:sign1), signv = (not_swapped?sign1:sign0);
int iu = (not_swapped?0:1), iv = (not_swapped?1:0);
ex umin = (not_swapped ? xmin : ymin), umax = (not_swapped ? xmax : ymax),
 vmin = (not_swapped ? ymin: xmin), vmax = (not_swapped ? ymax : xmax),
 uc = (not_swapped ? xc: yc), vc = (not_swapped ? yc : xc);
lst b_roots = roots(vmin, not_swapped), t_roots = roots(vmax, not_swapped);
```

Uses cycle 4 5a 6c 6c 6c 6c 6c 68a 68b 69c 69c 71 71 71 71 71, ex 6d 16b 58c 71 72 96 97a 97b 98a, lst 17b, numeric 15a, and varidx 15a.

Here is the outline of the rest of the method. It effectively splits into several cases depending from the space metric and degeneracy of **cycle2D**.

87a ⟨cycle.cpp 59a⟩+≡ ◁86b 96▷

```
⟨Draw a straight line 87b⟩
⟨Find intersection points with the boundary 88c⟩
if (sign > 0) { // elliptic metric
 ⟨Draw a circle 89a⟩
} else { // parabolic or hyperbolic  metric
 ⟨Draw a parabola or hyperbola 91a⟩
}
ost ≪ endl;
ost.flags(keep_flags);
return;
}
```

If line is detected we identify its visible portion.

87b ⟨Draw a straight line 87b⟩≡ (87a) 87c▷

```
if (b_roots.nops() ≠ 2) { // a linear object
 if (with_header)
  ost ≪ "  (straight  line)" ≪ endl;
 ex u1, u2, v1, v2;
 if (b_roots.nops() ≡ 1){ // a "non-horisontal" line
  u1 = max(min(b_roots.op(0), umax), umin);
  u2 = min(max(t_roots.op(0), umin), umax);
 } else { // a "horisontal" line
  u1 = umin;
  u2 = umax;
 }
```

Uses ex 6d 16b 58c 71 72 96 97a 97b 98a.

Vertical lines case.

87c ⟨Draw a straight line 87b⟩+≡ (87a) ◁87b 88a▷

```
 if (get_l(iv).is_zero()) { // a vertical line
  if (ex_to<numeric>((b_roots.op(0)- umin).evalf()).is_positive()
   ∧ ex_to<numeric>((umax-b_roots.op(0)).evalf()).is_positive()) {
   v1 =  vmin;
   v2 =  vmax;
  } else { // out of scope
   ost.flags(keep_flags);
   return;
  }
```

Uses numeric 15a.

Look for the visible portion of generic line.

88a ⟨Draw a straight line 87b⟩+≡ (87a) ◁87c 88b▷
```
} else {
 v1 = roots(u1, ¬not_swapped).op(0);
 v2 = roots(u2, ¬not_swapped).op(0);
 if ((max(v1, v2) > vmax) ∨ (min(v1, v2) < vmin)) {
  ost.flags(keep_flags);
  return; //out of scope
 }
}
```

Actual drawing of the line.

88b ⟨Draw a straight line 87b⟩+≡ (87a) ◁88a
```

ost ≪ draw_start.str() ≪ PAIR(not_swapped ? u1: v1, not_swapped ? v1: u1)
 ≪ ")*u--(" ≪ PAIR(not_swapped ? u2: v2, not_swapped ? v2: u2) ≪ ")*u"
 ≪ draw_options.str();
if (with_header)
 ost ≪ endl;
ost.flags(keep_flags);
return;
}
```

Uses PAIR 84b and u 91a.

Make initially this intervals (left[i], right[i]) irrelevant for drawing by default, if necessary, it will be redefined letter on.

88c ⟨Find intersection points with the boundary 88c⟩≡ (87a)
```
ex left[2] = {max(min(get_l(iu)÷k, umax), umin),
   max(min(get_l(iu)÷k, umax), umin)},
    right[2] = left;
// rearrange to have minimum value first
if (¬ex_to<numeric>((k∗signu).evalf()).is_positive()) {
 b_roots = lst(b_roots.op(1), b_roots.op(0));
 t_roots = lst(t_roots.op(1), t_roots.op(0));
}
if (ex_to<numeric>(b_roots.op(0).evalf()).is_real()) {
 left[0] = min(max(b_roots.op(0), umin), umax);
 right[0] = max(min(b_roots.op(1), umax), umin);
}
if (ex_to<numeric>(t_roots.op(0).evalf()).is_real()) {
 left[1] = min(max(t_roots.op(0), umin), umax);
 right[1] = max(min(t_roots.op(1), umax), umin);
}
```

Uses ex 6d 16b 58c 71 72 96 97a 97b 98a, lst 17b, and numeric 15a.

We start from the most involved case of a circle with a positive radius. Two this end we calculate coordinates $x[2][4]$ and $y[2][4]$ of endpoints for up to four arcs making the circle. The x-components of intersection points with vertical boundaries are rearranged appropriately.

89a ⟨Draw a circle 89a⟩≡ (87a) 89b ▷

```
if (determinant.is_positive()) {
 ex r = sqrt(det()), x[2][4], y[2][4];
 if (with_header)
  ost << " /circle of radius " << r << endl;
 x[0][0] = left[1]; x[0][1] = right[1]; x[0][2] = right[0]; x[0][3] = left[0];

 if (ex_to<numeric>((xc-r-xmin).evalf()).is_positive())
  x[1][0] = x[1][3] = xc-r;
 else
  x[1][0] = x[1][3] = xmin;

 if (ex_to<numeric>((xmax-xc-r).evalf()).is_positive())
  x[1][1] = x[1][2] = xc+r;
 else
  x[1][1] = x[1][2] = xmax;
```

Uses ex 6d 16b 58c 71 72 96 97a 97b 98a and numeric 15a.

We calculate now the y-components of the endpoints corresponding to x-components found before..

89b ⟨Draw a circle 89a⟩+≡ (87a) ◁ 89a 90a ▷

```
 lst y_roots;
 for (int j=0; j<2; j++)
  for (int i=0; i<4; i++)
   if ((x[j][i]-xc).is_zero()) // Touch the horizontal boundary?
    y[j][i] = (i==0 || i ==1? yc+r : yc-r);
   else if ((x[j][i]-xc-r).is_zero() || (x[j][i]-xc+r).is_zero()) // Touch the vertical boundary?
    y[j][i] = yc;
   else {
    y_roots = roots(x[j][i], false);
    if (ex_to<numeric>(y_roots.op(0).evalf()).is_real()) { // does circle intersect the boundary?
     if (i<2)
      y[j][i] = min(max(y_roots.op(0), y_roots.op(1)), ymax);
     else
      y[j][i] = max(min(y_roots.op(0), y_roots.op(1)), ymin);
    } else
     y[j][i] = yc;
   }
```

Uses lst 17b and numeric 15a.

Now we drawing up to four arcs which make the visible part of the circle. Each arc is defined through its two endpoints and tangent vector in them.

90a ⟨Draw a circle 89a⟩+≡ (87a) ◁89b 90b▷

```
for (int i=0; i<4; i++) {// actual drawing of four arcs
 int s = (i≡0 ∨ i ≡2? -1:1);
 ost ≪ "\t" ≪ draw_start.str() ≪ PAIR(x[0][i], y[0][i]) ≪ ")*u{"
        ≪ PAIR(s∗(y[0][i]-yc), s∗(xc-x[0][i]))
        ≪ (asymptote ? "}::{" : "}...{")
        ≪ PAIR(s∗(y[1][i]-yc), s∗(xc-x[1][i])) ≪ "}(" ≪ PAIR(x[1][i], y[1][i]) ≪ ")*u"
        ≪ draw_options.str();
 }
```

Uses PAIR 84b and u 91a.

Finally, for zero-radius circles we draw a point and do not draw anything for circles with an imaginary radius.

90b ⟨Draw a circle 89a⟩+≡ (87a) ◁90a

```
} else if (det().is_zero()) {
 if (with_header)
  ost ≪ " /circle of zero-radius" ≪ endl;
 ost ≪ (asymptote ? "\tdot(" : "\tdraw ") ≪ picture
       ≪ (int(∗picture)≡0? "" : ",") ≪ "("
       ≪ PAIR(xc, yc) ≪ ")*u" ≪ draw_options.str();
} else
 if (with_header)
  ost ≪ " /circle of imaginary radius--not drawing" ≪ endl;
```

Uses PAIR 84b and u 91a.

First we look if the parabola or hyperbola are degenerates into two lines, then treat two types of cycles separately.

91a ⟨Draw a parabola or hyperbola 91a⟩≡ (87a)

```
double u, v, du, dv, k_d = ex_to<numeric>(k.evalf()).to_double(),
        lu = ex_to<numeric>(get_l(iu).evalf()).to_double(),
        lv = ex_to<numeric>(get_l(iv).evalf()).to_double();

bool change_branch = (sign ≠ 0); // either to do a swap of branches
int zero_or_one = (sign ≡ 0 ∨ k_d*signv > 0 ? 0 : 1); // for parabola and positive k take first

if (sign ≡ 0) {
 ⟨Treating a parabola 91b⟩
} else {
 ⟨Treating a hyperbola 94b⟩
}
```

Defines:
du, used in chunk 84b.
dv, used in chunk 84b.
k_d, used in chunks 84b, 92b, and 95a.
u, used in chunks 15, 16, 23a, 25a, 28–34, 42–45, 47–52, 54a, 84b, 86a, 88b, 90, and 94a.
v, used in chunks 15, 16, 23a, 25a, 28–34, 42, 43a, 45, 49, 57b, 84b, and 86a.
zero_or_one, used in chunks 84b and 95.
Uses numeric 15a.

For parabolas degenerated into two parallel lines we draw them by the recursive call of this function
cycle2D::*metapost_draw*().

91b ⟨Treating a parabola 91b⟩≡ (91a) 92a ▷

```
if (sign0 ≡ 0 ∧ get_l(0).is_zero()) {
 if (with_header)
  ost << " /parabola degenerated into two horizontal lines" << endl;
 cycle2D(0, lst(0, 1), 2*b_roots.op(0), get_metric()).metapost_draw(ost, xmin, xmax,
                                   ymin, ymax, color, more_options,
                                        false, 0, asymptote, picture);
 cycle2D(0, lst(0, 1), 2*b_roots.op(1), get_metric()).metapost_draw(ost, xmin, xmax,
                                   ymin, ymax, color, more_options,
                                       false, 0, asymptote, picture);
 if (with_header)
  ost << endl;
 ost.flags(keep_flags);
 return;
```

Uses cycle2D 10d 16c 16c 50a 52 58c 58c 58c 58c 58c 72 72 72 72 72 82c 82c 85a and lst 17b.

Two vertical lines are drawn here

92a ⟨Treating a parabola 91b⟩+≡ (91a) ◁91b 92b▷

```
} else if (sign1 ≡ 0 ∧ get_l(1).is_zero()) {
if (with_header)
 ost ≪ "  /parabola degenerated into two vertical lines" ≪ endl;
cycle2D(0, lst(1, 0), 2∗b_roots.op(0), get_metric()).metapost_draw(ost, xmin, xmax,
                    ymin, ymax, color, more_options,
                 false, 0, asymptote, picture);
cycle2D(0, lst(1, 0), 2∗b_roots.op(1), get_metric()).metapost_draw(ost, xmin, xmax,
                    ymin, ymax, color, more_options,
                 false, 0, asymptote, picture);
if (with_header)
 ost ≪ endl;
ost.flags(keep_flags);
return;
}
```

Uses cycle2D 10d 16c 16c 50a 52 58c 58c 58c 58c 58c 72 72 72 72 72 82c 82c 85a and lst 17b.

If a proper parabola is detected we rearrange intervals appropriately in order to draw pieces properly.

92b ⟨Treating a parabola 91b⟩+≡ (91a) ◁92a 93a▷

```
if (with_header)
 ost ≪ "  /parabola" ≪ endl;
if (ex_to<numeric>((right[0]-left[0]).evalf()).is_positive()
    ∧ ex_to<numeric>((right[1]-left[1]).evalf()).is_positive())
 if (k_d∗signu > 0) { //rearrange intervals
  ex e = left[1]; left[1] = right[0]; right[0] = left[0]; left[0] =e;
 } else {
  ex e = left[1]; left[1] = right[1]; right[1] = right[0]; right[0] =e;
 }
```

Uses ex 6d 16b 58c 71 72 96 97a 97b 98a, k_d 91a, and numeric 15a.

Parabolas can be exactly represented by a cubic Bézier arc if the second and third control points correspondingly are:

$$\left(\frac{2}{3}x_0 + \frac{1}{3}x_1, \frac{1}{n}\left(\frac{1}{6}x_0^2 k + \frac{1}{3}x_0 x_1 k - \frac{2}{3}x_0 l - \frac{1}{3}l x_1 + \frac{1}{2}m\right)\right),$$
$$\left(\frac{1}{3}x_0 + \frac{2}{3}x_1, \frac{1}{n}\left(\frac{1}{3}x_0 k x_1 - \frac{1}{3}x_0 l - \frac{2}{3}l x_1 + \frac{1}{6}k x_1^2 + \frac{1}{2}m\right)\right).$$

93a ⟨Treating a parabola 91b⟩+≡ (91a) ◁92b 93b▷

```
  for (int i =0; i < 2; i++) {
   if (ex_to<numeric>((right[i]-left[i]).evalf()).is_positive()) { // a proper branch of a parabola

    ex cp[8];
    if (not_swapped) {
     cp[0] = left[i];
     cp[1] = val(lst(cp[0],0))÷2÷get_l(1);
     cp[6] = right[i];
     cp[7] = val(lst(cp[6],0))÷2÷get_l(1);
     cp[2] = numeric(2,3)*cp[0]+numeric(1,3)*cp[6];
     cp[3] = (numeric(1,6)*cp[0]*cp[0]*k + numeric(1,3)*cp[0]*cp[6]*k
       - numeric(2,3)*cp[0]*get_l(0)- numeric(1,3)*get_l(0)*cp[6]+m÷2)÷get_l(1);
     cp[4] = numeric(1,3)*cp[0]+numeric(2,3)*cp[6];
     cp[5] = (numeric(1,3)*cp[0]*k*cp[6]-numeric(1,3)*cp[0]*get_l(0)
       -numeric(2,3)*get_l(0)*cp[6]+numeric(1,6)*k*cp[6]*cp[6]+m÷2)÷get_l(1);
```

Uses `ex` 6d 16b 58c 71 72 96 97a 97b 98a, `lst` 17b, and `numeric` 15a.

The similar formulae for swapped drawing.

93b ⟨Treating a parabola 91b⟩+≡ (91a) ◁93a 94a▷

```
    } else {
     cp[1] = left[i];
     cp[0] = val(lst(0,cp[1]))÷2÷get_l(0);
     cp[7] = right[i];
     cp[6] = val(lst(0,cp[7]))÷2÷get_l(0);
     cp[3] = numeric(2,3)*cp[1]+numeric(1,3)*cp[7];
     cp[2] = (numeric(1,6)*cp[1]*cp[1]*k + numeric(1,3)*cp[1]*cp[7]*k
       - numeric(2,3)*cp[1]*get_l(1)- numeric(1,3)*get_l(1)*cp[7]+m÷2)÷get_l(0);
     cp[5] = numeric(1,3)*cp[1]+numeric(2,3)*cp[7];
     cp[4] = (numeric(1,3)*cp[1]*k*cp[7]-numeric(1,3)*cp[1]*get_l(1)
       -numeric(2,3)*get_l(1)*cp[7]+numeric(1,6)*k*cp[7]*cp[7]+m÷2)÷get_l(0);
    }
```

Uses `lst` 17b and `numeric` 15a.

The actual drawing of the parabola arcs.

94a ⟨Treating a parabola 91b⟩+≡ (91a) ◁93b

```
ost ≪ draw_start.str() ≪ PAIR(cp[0], cp[1]) ≪ ")*u .. controls (";
if (asymptote)
      ost ≪ PAIR(cp[2], cp[3]) ≪ ")*u and ("
            ≪ PAIR(cp[4], cp[5]) ≪ ")*u .. (";
else
      ost ≪ "(" ≪ PAIR(cp[2], cp[3]) ≪ ")*u) and (("
            ≪ PAIR(cp[4], cp[5]) ≪ ")*u) .. (";
ost ≪ PAIR(cp[6], cp[7]) ≪ ")*u" ≪ draw_options.str();
}
}
```

Uses `PAIR` 84b and `u` 91a.

If a hyperbola degenerates into a light cone we draw it as two separate lines.

94b ⟨Treating a hyperbola 94b⟩≡ (91a) 95a▷

```
if (determinant.is_zero()) {
 if (with_header)
  ost ≪ " / a light cone at (" ≪ xc ≪ "," ≪ yc ≪ ")" ≪ endl;
 cycle2D(0, lst(1, 1), 2*(xc+yc), get_metric()).metapost_draw(ost, xmin, xmax,
                    ymin, ymax, color, more_options,
               false, 0, asymptote, picture);
 cycle2D(0, lst(1, -1), 2*(xc-yc), get_metric()).metapost_draw(ost, xmin, xmax,
                    ymin, ymax, color, more_options,
               false, 0, asymptote, picture);
 if (with_header)
  ost ≪ endl;
 ost.flags(keep_flags);
 return;
```

Uses `cycle2D` 10d 16c 16c 50a 52 58c 58c 58c 58c 58c 72 72 72 72 72 82c 82c 85a and `lst` 17b.

Otherwise we rearrange the interwals for hyperbola branches.

95a ⟨Treating a hyperbola 94b⟩+≡ (91a) ◁94b 95b▷

```
} else {
if (with_header)
 ost << "  /hyperbola" << endl;
if (ex_to<numeric>((vmin-vc).evalf()).is_positive()) {
 ex e = left[1]; left[1] = right[0]; right[0] = left[0]; left[0] =e;
 change_branch = false;
 zero_or_one = (k_d*signv > 0 ? 1 : 0);
 }
if (ex_to<numeric>((vc-vmax).evalf()).is_positive()) {
 ex e = left[1]; left[1] = right[1]; right[1] = right[0]; right[0] =e;
 change_branch = false;
 zero_or_one = (k_d*signv > 0 ? 0 : 1);
 }
}
```

Uses ex 6d 16b 58c 71 72 96 97a 97b 98a, k_d 91a, numeric 15a, and zero_or_one 91a.

Two arcs of the hyperbola are drown now

95b ⟨Treating a hyperbola 94b⟩+≡ (91a) ◁95a

```
int points = (points_per_arc ≡ 0? 5 : points_per_arc);
for (int i =0; i < 2; i++) {
 //direction of the tangent vectors
 double dir = ex_to<numeric>(csgn(signv*(2*zero_or_one-1))).to_double();
 if (ex_to<numeric>((right[i]-left[i]).evalf()).is_positive()) { // a proper branch of the hyperbola
  DRAW_ARC(left[i], draw_start.str());
  for (int j=1; j<points; j++) {
   DRAW_ARC(left[i]*(1.0-j÷(points-1.0))+right[i]*j÷(points-1.0),
             (asymptote ? "::(" : "...(") );
  }
  ost << draw_options.str();
 }
 if (change_branch)
  zero_or_one = 1 - zero_or_one; // make a swap for the next branch of hyperbola
}
```

Defines:
points, used in chunks 11b, 15, 34c, 42, and 45.
Uses DRAW_ARC 84b, numeric 15a, and zero_or_one 91a.

D.4 Auxiliary functions implementation

The auxillary functions defined as well.

D.4.1 Heaviside function

We define Heaviside function: $\chi(x) = 1$ for $x \geq 0$ and $\chi(x) = 0$ for $x < 0$.

96 ⟨cycle.cpp 59a⟩+≡ ◁87a 97a▷

```
//////////
// Jump function
//////////

static ex jump_fnct_evalf(const ex & arg)
{
 if (is_exactly_a<numeric>(arg)) {
  if ((ex_to<numeric>(arg).is_real() ∧ ex_to<numeric>(arg).is_positive())
        ∨ ex_to<numeric>(arg).is_zero())
   return numeric(1);
  else
   return numeric(-1);
 }

 return jump_fnct(arg).hold();
}
```

Defines:
ex, used in chunks 4–12, 14a, 15c, 17a, 28b, 33b, 35, 48, 55–59, 62, 63, 65–68, 70, 73–89, 92b, 93a, 95a, and 98–100.
Uses jump_fnct 56 and numeric 15a.

97a ⟨cycle.cpp 59a⟩+≡ ◁96 97b▷

```
static ex jump_fnct_eval(const ex & arg)
{
 if (is_exactly_a<numeric>(arg)) {
  if ((ex_to<numeric>(arg).is_real() ∧ ex_to<numeric>(arg).is_positive())
         ∨ ex_to<numeric>(arg).is_zero())
   return numeric(1);
  else
   return numeric(-1);
 } else if (is_exactly_a<mul>(arg) ∧
     is_exactly_a<numeric>(arg.op(arg.nops()-1))) {
  numeric oc = ex_to<numeric>(arg.op(arg.nops()-1));
  if (oc.is_real()) {
   if (oc > 0)
    // jump_fnct(42*x) -> jump_fnct(x)
    return jump_fnct(arg÷oc).hold();
   else
    // jump_fnct(-42*x) -> -jump_fnct(x)
    return -jump_fnct(arg÷oc).hold();
  }
 }
 return jump_fnct(arg).hold();
}
```

Defines:
ex, used in chunks 4–12, 14a, 15c, 17a, 28b, 33b, 35, 48, 55–59, 62, 63, 65–68, 70, 73–89, 92b, 93a, 95a, and 98–100.
Uses jump_fnct 56 and numeric 15a.

97b ⟨cycle.cpp 59a⟩+≡ ◁97a 98a▷

```
static ex jump_fnct_conjugate(const ex & arg)
{
 return jump_fnct(arg);
}
```

Defines:
ex, used in chunks 4–12, 14a, 15c, 17a, 28b, 33b, 35, 48, 55–59, 62, 63, 65–68, 70, 73–89, 92b, 93a, 95a, and 98–100.
Uses jump_fnct 56.

98a ⟨cycle.cpp 59a⟩+≡ ◁97b 98b▷

```
static ex jump_fnct_power(const ex & arg, const ex & exp)
{
 if (is_a<numeric>(exp) ∧ ex_to<numeric>(exp).is_integer()) {
  if (ex_to<numeric>(exp).is_even())
   return numeric(1);
  else
   return jump_fnct(arg);
 }
 if (is_a<numeric>(exp) ∧ ex_to<numeric>(-exp).is_positive())
  return ex_to<basic>(pow(jump_fnct(arg), -exp)).hold();
 return ex_to<basic>(pow(jump_fnct(arg), exp)).hold();
}
```

Defines:
ex, used in chunks 4–12, 14a, 15c, 17a, 28b, 33b, 35, 48, 55–59, 62, 63, 65–68, 70, 73–89, 92b, 93a, 95a, and 98–100.
Uses jump_fnct 56 and numeric 15a.

98b ⟨cycle.cpp 59a⟩+≡ ◁98a 98c▷

```
static void jump_fnct_print_dflt_text(const ex & x, const print_context & c)
{
 c.s ≪ "H("; x.print(c); c.s ≪ ")";
}
```

Defines:
jump_fnct_print_dflt_text, used in chunk 98c.
Uses ex 6d 16b 58c 71 72 96 97a 97b 98a.

All above methods are used to register the function now.

98c ⟨cycle.cpp 59a⟩+≡ ◁98b 99a▷

```
REGISTER_FUNCTION(jump_fnct, eval_func(jump_fnct_eval).
      evalf_func(jump_fnct_evalf).
      latex_name("\\chi").
      //text_name("H").
      print_func<print_dflt>(jump_fnct_print_dflt_text).
      //derivative_func(2*delta).
      power_func(jump_fnct_power).
      conjugate_func(jump_fnct_conjugate));
```

Uses jump_fnct 56 and jump_fnct_print_dflt_text 98b.

This function prints if its parameter is zero in a prominent way.

99a ⟨cycle.cpp 59a⟩+≡ ◁98c 99b▷

```
const char *equality(const ex & E)
{
 if (normal(E).is_zero())
  return "_equal_";
 else
  return "DIFFERENT!!!";
}
```

Defines:
equality, used in chunks 19d and 20b.
Uses ex 6d 16b 58c 71 72 96 97a 97b 98a.

This function decodes metric sign into human-readable form.

99b ⟨cycle.cpp 59a⟩+≡ ◁99a 100a▷

```
const char *eph_case(const numeric & sign)
{
 if (numeric(sign-(-1)).is_zero())
  return "Elliptic case (sign = -1)";
 if (numeric(sign).is_zero())
  return "Parabolic case (sign = 0)";
 if (numeric(sign-1).is_zero())
  return "Hyperbolic case (sign = 1)";
 return "Unknown case!!!!";
}
```

Defines:
eph_case, used in chunk 31b.
Uses numeric 15a.

Elements of $SL_2(\mathbb{R})$ are transformed into appropriate "cliffordian" matrix.

100a ⟨cycle.cpp 59a⟩+≡ ◁99b 100b▷

```
matrix sl2_clifford(const ex & a, const ex & b, const ex & c, const ex & d, const ex & e,
                    bool not_inverse)
{
 if (is_a<clifford>(e)) {
  ex e0 = e.subs(e.op(1) ≡ 0);
  ex one = dirac_ONE(ex_to<clifford>(e).get_representation_label());
  if (not_inverse)
   return matrix(2, 2,
       lst(a * one, b * pow(e0, 3),
        c * e0, d * one));
  else
   return matrix(2, 2,
       lst(d * one, -b * pow(e0, 3),
        -c * e0, a * one));
 } else
  throw(std::invalid_argument("sl2_clifford(): expect a clifford numeber"
                       " as a parameter"));
}
```

Uses ex 6d 16b 58c 71 72 96 97a 97b 98a, lst 17b, and matrix 15b 16a.

100b ⟨cycle.cpp 59a⟩+≡ ◁100a

```
matrix sl2_clifford(const ex & M, const ex & e, bool not_inverse)
{
 if (is_a<matrix>(M) ∨ M.info(info_flags::list))
  return sl2_clifford(M.op(0), M.op(1), M.op(2), M.op(3), e, not_inverse);
 else
  throw(std::invalid_argument("sl2_clifford(): expect a list or matrix "
                       "as the first parameter"));
}
```

Uses ex 6d 16b 58c 71 72 96 97a 97b 98a and matrix 15b 16a.

REFERENCES

[1] Christian Bauer, Alexander Frink, Richard Kreckel, and Jens Vollinga. GiNaC is Not a CAS. http://www.ginac.de/. 4

[2] Jan Cnops. *An introduction to Dirac operators on manifolds*, volume 24 of *Progress in Mathematical Physics*. Birkhäuser Boston Inc., Boston, MA, 2002. 1, 4

[3] R. Delanghe, F. Sommen, and V. Souček. *Clifford Algebra and Spinor-Valued Functions*, volume 53 of *Mathematics and its Applications*. Kluwer Academic Publishers Group, Dordrecht, 1992. A function theory for the Dirac operator, Related REDUCE software by F. Brackx and D. Constales, With 1 IBM-PC floppy disk (3.5 inch). 3

[4] Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA. *GNU General Public License*, second edition, 1991. http://www.gnu.org/licenses/gpl.html. 37

[5] Andy Hammerlindl, John Bowman, and Tom Prince. Asymptote—powerful descriptive vector graphics language for technical drawing, inspired by MetaPost. http://asymptote.sourceforge.net/. 4, 12

[6] John D. Hobby. MetaPost: A MetaFont like system with postscript output. http://www.tug.org/metapost.html. 4

[7] Vladimir V. Kisil. Elliptic, parabolic and hyperbolic analytic function theory–1: Geometry of invariants. 2005. E-print: arXiv:math.CV/0512416. preprint LEEDS–MATH–PURE–2005–28. (To appear). 1, 3, 4, 5, 8, 9, 11, 14, 18, 19, 20, 24, 25, 26, 28, 30, 31, 32, 33, 34, 35, 36, 37, 74, 76, 81

[8] Vladimir V. Kisil. An example of clifford algebras calculations with GiNaC. *Adv. in Appl. Clifford Algebras*, 15(2):239–269, 2005. E-print: arXiv:cs.MS/0410044. 1, 4

[9] Vladimir V. Kisil. Fillmore–Springer–Cnops constructions implemented in GiNaC. 2005. E-print: arXiv:cs.MS/0512073. preprint LEEDS–MATH–PURE–2005–29. (To appear). 25, 28, 37, 51

[10] Vladimir V. Kisil and Daniel Seidel. Python wrapper for cycle library based on pyGiNaC. http://maths.leeds.ac.uk/~kisilv/pycycle.html, 2006. 4

[11] Ian R. Porteous. *Clifford algebras and the classical groups*, volume 50 of *Cambridge Studies in Advanced Mathematics*. Cambridge University Press, Cambridge, 1995. 4

[12] Norman Ramsey. Noweb — a simple, extensible tool for literate programming. http://www.eecs.harvard.edu/~nr/noweb/. 37

[13] I. M. Yaglom. *A simple non-Euclidean geometry and its physical basis*. Springer-Verlag, New York, 1979. An elementary account of Galilean geometry and the Galilean principle of relativity, Heidelberg Science Library, Translated from the Russian by Abe Shenitzer, With the editorial assistance of Basil Gordon. 25

E INDEX OF IDENTIFIERS