arXiv: cs.MS/0512073, 2005
LEEDS–MATH–PURE–2005–29

Volume 00, Number 0, Pages 000–000

SCHWERDTFEGER–FILLMORE–SPRINGER–CNOPS CONSTRUCTION IMPLEMENTED IN GiNaC

VLADIMIR V. KISIL

Dedicated to the memory of Dennis Ritchie

Abstract. This is an implementation of the Schwerdtfeger–Fillmore–Springer–Cnops construction (SFSCc) based on the Clifford algebra capacities [14] of the GiNaC computer algebra system. SFSCc linearises the linear-fraction action of the Möbius group. This turns to be very useful in several theoretical and applied fields including engineering. The package is realised as a C++ library and there are several Python wrapper of it, which can be used in interactive mode.

The core of this realisation of SFSCc 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. Calculations can be done either in vector or paravector formalism.

This library is a backbone of many results published in [18], which serve as illustrations of its usage. It can be ported (with various level of required changes) to other CAS with Clifford algebras capabilities.

There is an ISO image of a Live Debian DVD attached to this paper at arXiv and the Google drive (an updated version).

The software is distributed under GNU GPLv3, see Appendix F.

Contents

Date: 18th August 2016 (v2.3).
2010 *Mathematics Subject Classification.* Primary 51B25; Secondary 51N25, 68U05, 11E88, 68W30.
On leave from Odessa University.

1. Introduction

The usage of computer algebra system (CAS) in Clifford Algebra research has an established history with the famous "Green book" [6] already accompanied by a floppy disk with a `REDUCE` package. This tradition is very much alive, see for example the recent books [10, 17, 19] accompanied by a software CD/DVD. Numerous new packages are developed by various research teams across the world to work with Clifford algebras generally or address specific tasks, see on-line proceedings of the recent IKM-2006 conference [9].

Along this lines the present paper presents an implementation of the Schwerdtfeger–Fillmore–Springer–Cnops construction[1] (SFSCc) along with illustrations of its usage. SFSCc [5, § 4.1; 7; 13, § 4.2; 18; 19, § 4.2; 25, § 18; 27, § 1.1] 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 [2], which were described in [14]. The code is written using `noweb` literate programming tool [26]

The core of this realisation of SFSCc is done for an arbitrary dimension of $\mathbb{R}^n$ with a metric given by an arbitrary bilinear form. Corresponding calculation can be done using both vector or paravector formalims in Clifford algebras, see § E.1.5. Results of calculations are largely independent from used formalism with some notable exceptions: determinants of SFSC matrices and Möbius maps defined by those matrices, see Rems. 2.1, and 2.2.

Remark 1.1. Paravector formalism shall not work with GiNaC prior v.1.7.1. Earlier versions of GiNaC will result in errors of this type:

```
get_clifford_comp(): expression is not a Clifford vector to the given units
```

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` [12] or `Asymptote` [11] packages. This software is the backbone of many results published in [17–19] and we use its application to [18] for the demonstration purpose.

There is a Python wrapper [21] for this library. It 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`. The live DVD supplied with book [19] is based on the library presented in this paper and its Python wrapper.

This library is now a part of MoebInv project (`http://moebinv.sourceforge.net/`) [20]. Please look there for latest updates, source and binary distributions. ISO images of live DVD may be referred there as well. We do not plan to use arXiv for these purposes anymore.

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

The software is distributed under GNU GPLv3, see Appendix F and [8].

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

$$k\mathbf{x}^2 - 2\langle \mathbf{l}, \mathbf{x}\rangle + m = 0, \qquad \text{where } k, m \in \mathbb{R},\ \mathbf{l} \in \mathbb{R}^n. \tag{2.1}$$

The class **cycle** correspondingly has member variables k, l, m to describe the equation (2.1) and the Clifford algebra *unit* to describe the metric of surrounding space. The plenty of methods are supplied for various tasks within SFSCc.

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

[1]In the case of circles this technique was already spectacularly developed by H. Schwerdtfeger in 1960-ies, see [27]. Unfortunately, that beautiful book was not known to the present author until he accomplished his own works [16, 18, 19].

2.1. **Constructors of cycle.** Here is various constructors for the **cycle**s. The first one takes values of k, **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 [14].

3a ⟨cycle class constructors 3a⟩≡ (60b) 3b ▷

public:
cycle(**const ex** & *k*, **const ex** & *l*, **const ex** & *m*,
  const ex & *metr* = -(**new** *tensdelta*)→*setflag*(*status_flags*::*dynallocated*));

Defines:
`cycle`, used in chunks 4–9, 12a, 13a, 15–20, 22–26, 28e, 32–35, 53a, 60, 61b, 64–76, 78–80, 82–87, 89a, 91a, 93–95, and 97b.
`k`, used in chunks 3e, 4b, 8e, 9f, 14, 15, 19d, 22–25, 31c, 36, 50–52, 55a, 60, 62b, 64, 65a, 71–75, 78–80, 82–84, 89, 90, 93b, and 100a.
`l`, used in chunks 3, 4, 9b, 14, 15, 22e, 23b, 25–28, 31c, 50–52, 55–57, 60, 62b, 64, 65b, 67a, 71–75, 78a, 79c, 82–84, 88c, and 93b.
`m`, used in chunks 4, 14, 15, 22e, 23b, 25b, 26e, 28a, 50a, 51d, 56e, 57a, 60, 62–65, 71–75, 78a, 80a, 82–84, 89, 90, 93b, and 104e.
`metr`, used in chunks 3, 5f, 9, 65–70, 78c, 79a, 88, and 89a.
Uses `ex` 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c.

Constructor for a **cycle** (2.1) with $k = 1$ and given l defined by the condition that square of its “radius” (which is $\det C$, see [18, Defn. 5.1]) is *r_squared*. If a non-zero e is provided, then it is used to calculate $C.det(e)$, otherwise the default value is $C.det(metr)$. Note that for the default value of the *metr* the value of l coincides with the centre of this **cycle**.

3b ⟨cycle class constructors 3a⟩+≡ (60b) ◁3a 3c ▷

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 4–9, 12a, 13a, 15–20, 22–26, 28e, 32–35, 53a, 60, 61b, 64–76, 78–80, 82–87, 89a, 91a, 93–95, and 97b.
Uses `ex` 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, `l` 3a, and `metr` 3a.

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.

3c ⟨cycle class constructors 3a⟩+≡ (60b) ◁3b 3d ▷

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

Defines:
`cycle`, used in chunks 4–9, 12a, 13a, 15–20, 22–26, 28e, 32–35, 53a, 60, 61b, 64–76, 78–80, 82–87, 89a, 91a, 93–95, and 97b.
Uses `ex` 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c and `metr` 3a.

To any cycle SFSCc associates a matrix, which is of the form (2.2) [18, (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 [18, (3.2)].

The dimensionality of the point space may not be correctly guessed from the matrix if both vector and paravector formalisms are allowed (cf. § E.1.5), i.e. the absence of the *dirac_ONE* may come either from the vector formalism or mean $l.oplus(0)\equiv 0$ in paravector formalim. Thus, the the correct non-zero value of the dimensionality (the last parameter) shall be supplied whenever possible.

3d ⟨cycle class constructors 3a⟩+≡ (60b) ◁3c

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

Defines:
`cycle`, used in chunks 4–9, 12a, 13a, 15–20, 22–26, 28e, 32–35, 53a, 60, 61b, 64–76, 78–80, 82–87, 89a, 91a, 93–95, and 97b.
Uses `ex` 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, `matrix` 11d 16b 16c, and `metr` 3a.

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

3e ⟨accessing the data of a cycle 3e⟩≡ (60b) 4a ▷

public:
virtual inline ex *get_dim*() **const** { **return** *ex_to*<**varidx**>(*l.op*(1)).*get_dim*(); }
virtual ex *get_metric*() **const**;
virtual ex *get_metric*(**const ex** &*i0*, **const ex** &*i1*) **const**;
virtual inline ex *get_k*() **const** { **return** *k*; }

Defines:
`get_dim`, used in chunks 18e, 64, 65, 68–70, 74–76, 78–88, 90, 91a, 104b, 108c, and 109b.
`get_k`, used in chunks 18a, 20b, 30a, 31f, 35a, 67b, 73, 74c, 76, 89a, 95a, 96b, 100, and 101.
`get_metric`, used in chunks 68c, 70, 79, 82b, 84a, 90, and 94a.
Uses `ex` 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, `k` 3a, `l` 3a, `op` 4b, and `varidx` 14a 15a 15b.

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

4a ⟨accessing the data of a cycle 3e⟩+≡ (60b) ◁3e 4b▷

```
inline ex get_l() const { return l; }
inline ex get_l(const ex & i) const
{ return (l.is_zero()?0:l.subs(l.op(1) ≡ i, subs_options::no_pattern)); }
inline ex get_m() const {return m;}
inline ex get_unit() const {return unit;}
```

Defines:
get_l, used in chunks 9f, 18a, 30a, 31f, 35a, 67b, 73–76, 79c, 80a, 87c, 89, 90, 95, 96b, 100, and 101.
get_m, used in chunks 35a, 67b, 73, 74, 76, 89a, 96b, and 101.
get_unit, used in chunks 35a and 89a.
Uses ex 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, is_zero 4b, l 3a, m 3a, op 4b, and subs 4b.

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.

4b ⟨accessing the data of a cycle 3e⟩+≡ (60b) ◁4a 4c▷

```
size_t nops() const {return 4;}
ex op(size_t i) const;
ex & let_op(size_t i);
bool is_equal(const basic & other, bool projectively = true, bool ignore_unit = false) 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());}
inline cycle expand() const { return cycle(k.expand(), l.expand(), m.expand(), unit);}
```

Defines:
expand, used in chunks 31d, 62b, and 108a.
is_equal, used in chunks 16f, 19, 20f, 22–25, 28b, 32–35, 73b, and 104c.
is_zero, used in chunks 4a, 12a, 16–18, 20–23, 25–27, 29g, 31, 65–67, 69b, 73–76, 78–80, 84–87, 89–93, 95, 96b, 100, and 105–107.
let_op, used in chunks 63a, 71b, and 105b.
nops, used in chunks 63a, 65b, 68a, 71, 80a, 84a, 93a, 95a, 104a, 106a, and 108a.
normal, used in chunks 6b, 11d, 12a, 16–23, 25–36, 50, 52, 59d, 62b, 67c, 73b, 74a, 78a, 85–87, 90, 91a, 96c, 107b, and 108a.
op, used in chunks 3e, 4a, 17–19, 21–23, 25c, 26a, 29, 30c, 35c, 36, 50–54, 63a, 65d, 67–72, 75, 76, 78–87, 90, 91e, 93–96, 98d, 100, 104–106, 108, and 109.
subs, used in chunks 4a, 11d, 12a, 16–19, 21–24, 26–29, 31–36, 50–52, 54–59, 61c, 63a, 68c, 70d, 72a, 78–84, 86c, 93b, 96c, 97a, 104e, 108, and 109.
Uses bool 16a, cycle 3a 3a 3b 3b 3c 3d 5a 5a 5a 5a 5a 72c 72d 72d 73a 75d 75d 77a 77a 77a 77a 77a, ex 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, k 3a, l 3a, and m 3a.

We also provide a method $the\_same\_as()$ which return a $GiNaC$::**lst** of identities (i.e. $GiNaC$::**relationals**), which defines that two cycles are given by the same point of the projective space $\mathbb{P}^3$.

4c ⟨accessing the data of a cycle 3e⟩+≡ (60b) ◁4b

```
ex the_same_as(const basic & other) const;
```

Uses ex 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c.

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.

4d ⟨Linear operation as cycle methods 4d⟩≡ (60b)

```
virtual cycle add(const cycle & rh) const;
virtual cycle sub(const cycle & rh) const;
virtual cycle exmul(const ex & rh) const;
virtual cycle div(const ex & rh) const;
```

Defines:
add, used in chunks 76, 77, and 108a.
div, used in chunks 76 and 77.
exmul, used in chunks 76 and 77.
sub, used in chunks 76 and 77.
Uses cycle 3a 3a 3b 3b 3c 3d 5a 5a 5a 5a 5a 72c 72d 72d 73a 75d 75d 77a 77a 77a 77a 77a and ex 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c.

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

5a ⟨Linear operation on cycles 5a⟩≡ (60b) 5b▷
```
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 4–9, 12a, 13a, 15–20, 22–26, 28e, 32–35, 53a, 60, 61b, 64–76, 78–80, 82–87, 89a, 91a, 93–95, and 97b.
`operator*`, used in chunks 5b, 62d, and 77.
`operator+`, used in chunks 62d and 77.
`operator-`, used in chunks 62d and 77.
`operator/`, used in chunks 62d and 77.
Uses `ex` 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c.

We also define a product of two cycles through their matrix representation (2.2).

5b ⟨Linear operation on cycles 5a⟩+≡ (60b) ◁5a
```
const ex operator*(const cycle & lh, const cycle & rh);
```
Defines:
`ex`, used in chunks 3–11, 14c, 16–32, 34–36, 53b, 59–63, 65a, 67–76, 78–80, 82–93, 96c, 97a, and 103–109.
Uses `cycle` 3a 3a 3b 3b 3c 3d 5a 5a 5a 5a 5a 72c 72d 72d 73a 75d 75d 77a 77a 77a 77a 77a and `operator*` 5a.

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 the cycle has non-zero determinant, then it is scaled to have new determinant equal D, with 1 as the default value. The last parameter *fix_paravector*=**true** ensures that the result of normalisation is independent from the used formalism, see Rem. 2.1.

5c ⟨specific methods of the class cycle 5c⟩≡ (60b) 5d▷
```
public:
cycle normalize_det(const ex & e = 0,
                    const ex & sign = (new tensdelta)→setflag(status_flags::dynallocated),
                    const ex & D = 1, bool fix_paravector = true) const;
```
Defines:
`normalize_det`, used in chunks 5d, 61c, and 78b.
Uses `bool` 16a, `cycle` 3a 3a 3b 3b 3c 3d 5a 5a 5a 5a 5a 72c 72d 72d 73a 75d 75d 77a 77a 77a 77a 77a, and `ex` 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c.

The square $\langle C,C\rangle$ of the norm of a cycle C is twice its determinant $\det C$, we provide a method to normalise the norm as well.

5d ⟨specific methods of the class cycle 5c⟩+≡ (60b) ◁5c 5e▷
```
inline cycle normalize_norm(const ex & e = 0,
                    const ex & sign = (new tensdelta)→setflag(status_flags::dynallocated),
                    const ex & N = 1, bool fix_paravector = true) const
{return normalize_det(e, sign, N*numeric(1,2), fix_paravector);}
```
Defines:
`normalize_norm`, used in chunk 61c.
Uses `bool` 16a, `cycle` 3a 3a 3b 3b 3c 3d 5a 5a 5a 5a 5a 72c 72d 72d 73a 75d 75d 77a 77a 77a 77a 77a, `ex` 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, `normalize_det` 5c, and `numeric` 14a 57d.

The next normalization acts as follows: 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*.

5e ⟨specific methods of the class cycle 5c⟩+≡ (60b) ◁5d 5f▷
```
cycle normalize(const ex & k_new = numeric(1), const ex & e = 0) const;
```
Defines:
`normalize`, used in chunks 24a, 25e, 36, 55d, 56b, 61c, 78, 84b, 94a, and 96c.
Uses `cycle` 3a 3a 3b 3b 3c 3d 5a 5a 5a 5a 5a 72c 72d 72d 73a 75d 75d 77a 77a 77a 77a 77a, `ex` 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, and `numeric` 14a 57d.

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

5f ⟨specific methods of the class cycle 5c⟩+≡ (60b) ◁5e 6a▷
```
virtual ex center(const ex & metr = 0, bool return_matrix = false) const;
```
Defines:
`center`, used in chunks 17d, 19a, 21–23, 25c, 26b, 30, 36, 50b, 52, 53, 78c, 79a, and 94a.
Uses `bool` 16a, `ex` 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, and `metr` 3a.

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

6a ⟨specific methods of the class cycle 5c⟩+≡ (60b) ◁5f 6b▷

virtual ex *val*(**const ex** & *y*, **const ex** & *x* = 1) **const**;

Defines:
`val`, used in chunks 6b, 12a, 16d, 20g, 22a, 23a, 26, 31e, 54a, 83c, 84a, 96, 97b, and 101.
Uses `ex` 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c.

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.

6b ⟨specific methods of the class cycle 5c⟩+≡ (60b) ◁6a 6c▷

inline ex *passing*(**const ex** & *y*) **const** {**return** *val*(*y*).*numer*().*normal*() ≡ 0;}

Defines:
`passing`, used in chunks 11c, 16d, 17a, 20, 21a, 23a, 25b, 26e, 28a, 30c, 31c, 33a, 36, 55a, and 93b.
Uses `ex` 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, `normal` 4b, and `val` 6a.

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:

C2 = *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.

6c ⟨specific methods of the class cycle 5c⟩+≡ (60b) ◁6b 6d▷

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

Defines:
`subject_to`, used in chunks 11c, 16d, 20, 21a, 25, 26e, 28a, 30c, 31c, 36, 55a, 62b, 67a, and 80a.
Uses `cycle` 3a 3a 3b 3b 3c 3d 5a 5a 5a 5a 5a 72c 72d 72d 73a 75d 75d 77a 77a 77a 77a 77a and `ex` 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c.

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

$$\begin{pmatrix} \mathbf{l}_i\sigma^i_j\tilde{e}^j & m \\ k & -\mathbf{l}_i\sigma^i_j\tilde{e}^j \end{pmatrix} \quad \text{from the cycle } k\mathbf{x}^2 - 2\langle \mathbf{l}, \mathbf{x}\rangle + m = 0. \tag{2.2}$$

Note, that the Clifford unit $\tilde{e}$ has an arbitrary metric unrelated to the initial metric stored in the *unit* member variable. If the last parameter set to **true** then in paravector formalism a Clifford conjugation of the matrix will be return. The parameter does not make any effect in the vector formalism. This is required by several methods, e.g. **cycle**::*cycle_similarity*().

6d ⟨specific methods of the class cycle 5c⟩+≡ (60b) ◁6c 6e▷

virtual matrix *to_matrix*(**const ex** & *e* = 0,
const ex & *sign* = (**new** *tensdelta*)→*setflag*(*status_flags*::*dynallocated*),
bool *conjugate* = **false**) **const**;

Defines:
`to_matrix`, used in chunks 82–85 and 87.
Uses `bool` 16a, `ex` 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, and `matrix` 11d 16b 16c.

The next method returns the value of determinant of the matrix (2.2) corresponding to the **cycle**. It has explicit geometric meaning, see [18, § 5.1]. Before calculation the cycle is normalised by the condition *k*≡*k_norm*, if *k_norm* is zero then no normalisation is done.

6e ⟨specific methods of the class cycle 5c⟩+≡ (60b) ◁6d 6f▷

virtual ex *det*(**const ex** & *e* = 0,
const ex & *sign* = (**new** *tensdelta*)→*setflag*(*status_flags*::*dynallocated*),
const ex & *k_norm* = 0, **bool** *fix_paravector* = **false**) **const**;

Defines:
`det`, used in chunks 6f, 9e, 17, 18f, 78b, 86b, and 90a.
Uses `bool` 16a and `ex` 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c.

Remark 2.1. It shall be noted, that the determinant has opposite signs in vector and paravector formalisms. This can be fixed by the last Boolean parameter *fix_paravector*, which ensure that the sign will be the same as in vector formalism.

The determinant of a k-normalised cycle can be treated as the square of its radius

6f ⟨specific methods of the class cycle 5c⟩+≡ (60b) ◁6e 7a▷

virtual inline ex *radius_sq*(**const ex** & *e* = 0,
const ex & *sign* = (**new** *tensdelta*)→*setflag*(*status_flags*::*dynallocated*)) **const**
{ **return** *this*→*det*(*e*, *sign*, **numeric**(1), **true**);}

Defines:
`radius_sq`, used in chunks 21e, 26f, 28b, 30–36, 67a, 78a, and 94a.
Uses `det` 6e 84b, `ex` 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, and `numeric` 14a 57d.

The matrix (2.2) 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 or matrix supplied in C.

7a ⟨specific methods of the class cycle 5c⟩+≡ (60b) ◁6f 7b▷

virtual ex *mul*(**const ex** & C, **const ex** & $e = 0$,
const ex & *sign* = (**new** *tensdelta*)→*setflag*(*status_flags*::*dynallocated*),
const ex & *sign1* =0) **const**;

Defines:
mul, used in chunks 77, 84–87, 91a, and 106a.
Uses ex 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c.

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 $\mathrm{SL}_2(\mathbb{R})$ group the next two methods make a proper conversion of M into Clifford-valued form.

7b ⟨specific methods of the class cycle 5c⟩+≡ (60b) ◁7a 7c▷

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 ex & *sign_inv* = (**new** *tensdelta*)→*setflag*(*status_flags*::*dynallocated*)) **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 ex & *sign_inv* = (**new** *tensdelta*)→*setflag*(*status_flags*::*dynallocated*)) **const**;

Defines:
sl2_similarity, used in chunks 12a, 16–18, 23c, 33b, 86, 90, and 91.
Uses bool 16a, cycle 3a 3a 3b 3b 3c 3d 5a 5a 5a 5a 5a 72c 72d 72d 73a 75d 75d 77a 77a 77a 77a 77a, and ex 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c.

If M is a generic 2×2-matrix of another sort then it is used in the similarity in the unchanged form by the next method.

7c ⟨specific methods of the class cycle 5c⟩+≡ (60b) ◁7b 7d▷

virtual cycle *matrix_similarity*(**const ex** & M, **const ex** & $e = 0$,
const ex & *sign* = (**new** *tensdelta*)→*setflag*(*status_flags*::*dynallocated*),
bool *not_inverse*=**true**,
const ex & *sign_inv* = (**new** *tensdelta*)→*setflag*(*status_flags*::*dynallocated*)) **const**;

Defines:
matrix_similarity, used in chunks 7d, 57e, and 85.
Uses bool 16a, cycle 3a 3a 3b 3b 3c 3d 5a 5a 5a 5a 5a 72c 72d 72d 73a 75d 75d 77a 77a 77a 77a 77a, and ex 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c.

The 2×2-matrix $M = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$ can be also defined by the collection of its elements.

7d ⟨specific methods of the class cycle 5c⟩+≡ (60b) ◁7c 7e▷

virtual 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 ex & *sign_inv* = (**new** *tensdelta*)→*setflag*(*status_flags*::*dynallocated*)) **const**;

Uses bool 16a, cycle 3a 3a 3b 3b 3c 3d 5a 5a 5a 5a 5a 72c 72d 72d 73a 75d 75d 77a 77a 77a 77a 77a, ex 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, and matrix_similarity 7c.

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 [18, § 4.2].

7e ⟨specific methods of the class cycle 5c⟩+≡ (60b) ◁7d 8a▷

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

Defines:
cycle_similarity, used in chunks 18f, 22e, 24a, 25e, 34a, 36, 55d, 57, and 87.
Uses cycle 3a 3a 3b 3b 3c 3d 5a 5a 5a 5a 5a 72c 72d 72d 73a 75d 75d 77a 77a 77a 77a 77a and ex 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c.

A cycle in the matrix form (2.2) naturally defines a Möbius transformations of the points:

$$\begin{pmatrix} \mathbf{l}_i\sigma^i_j\tilde{e}^j & m \\ k & -\mathbf{l}_i\sigma^i_j\tilde{e}^j \end{pmatrix} : \mathbf{x} \mapsto \frac{\mathbf{l}_i\sigma^i_j\tilde{e}^j\mathbf{x}+m}{k\mathbf{x}-\mathbf{l}_i\sigma^i_j\tilde{e}^j} \tag{2.3}$$

The following methods realised this transformations.

8a ⟨specific methods of the class cycle 5c⟩+≡ (60b) ◁7e 8b▷

virtual ex *moebius_map*(**const ex** & *P*, **const ex** & *e* = 0,
const ex & *sign* = (**new** *tensdelta*)→*setflag*(*status_flags*::*dynallocated*)) **const**;

Defines:
`moebius_map`, used in chunks 19–23, 26c, and 36.
Uses `ex` 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c.

Remark 2.2. The result depends on either vector or paravector formalism is used. In two dimensions, the second component received the opposed sign in paravector formalism: for example, **lst**{*u*,*v*} and **lst**{*u*,-*v*}.

For two matrices C_1 and C_2 obtained from cycles the expression

$$\langle C_1, C_2\rangle = -\Re\,\mathrm{tr}\,(C_1C_2) \tag{2.4}$$

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

8b ⟨specific methods of the class cycle 5c⟩+≡ (60b) ◁8a 8c▷

virtual ex *cycle_product*(**const cycle** & *C*, **const ex** & *e* = 0,
const ex & *sign* = (**new** *tensdelta*)→*setflag*(*status_flags*::*dynallocated*)) **const**;

Defines:
`cycle_product`, used in chunks 8c and 21a.
Uses `cycle` 3a 3a 3b 3b 3c 3d 5a 5a 5a 5a 5a 72c 72d 72d 73a 75d 75d 77a 77a 77a 77a 77a and `ex` 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c.

The inner product (2.4) defines an orthogonality relation $\langle C_1, C_2\rangle \equiv 0$ in the space of cycles which returned by the method *is_orthogonal*().

8c ⟨specific methods of the class cycle 5c⟩+≡ (60b) ◁8b 8d▷

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

Defines:
`is_orthogonal`, used in chunks 19, 20, 34c, and 36.
Uses `cycle` 3a 3a 3b 3b 3c 3d 5a 5a 5a 5a 5a 72c 72d 72d 73a 75d 75d 77a 77a 77a 77a 77a, `cycle_product` 8b 84c, and `ex` 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c.

In many cases we need a higher order orthogonal relation between cycles— so called f-orthogonality, see [18, § 4.3], which is given by the relation:

$$\Re\,\mathrm{tr}(C^s_{\breve\sigma}\tilde{C}^s_{\breve\sigma}C^s_{\breve\sigma}R^s_{\breve\sigma}) = 0.$$

8d ⟨specific methods of the class cycle 5c⟩+≡ (60b) ◁8c 8e▷

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

Defines:
`is_f_orthogonal`, used in chunks 24, 25, 34–36, and 87c.
Uses `cycle` 3a 3a 3b 3b 3c 3d 5a 5a 5a 5a 5a 72c 72d 72d 73a 75d 75d 77a 77a 77a 77a 77a and `ex` 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c.

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

8e ⟨specific methods of the class cycle 5c⟩+≡ (60b) ◁8d

inline ex *is_linear*() **const** {**return** (*k* ≡ 0);}
inline ex *is_normalized*() **const** {**return** (*k* ≡ 1);}

Defines:
`is_linear`, used in chunks 21a, 25c, and 36.
`is_normalized`, used in chunk 30c.
Uses `ex` 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c and `k` 3a.

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.

9a ⟨constructors of the class cycle2D 9a⟩≡ (61b) 9b▷
```
public:
    cycle2D(const ex & k1, const ex & l1, const ex & m1,
            const ex & metr = -unit_matrix(2));
```
Defines:
cycle2D, used in chunks 9, 10c, 16–23, 25, 26e, 28a, 30–33, 35a, 36, 50a, 51d, 53, 55–57, 61, 62, 64, 66d, 88–91, 93b, 94a, 96, 100, and 102b.
Uses ex 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c and metr 3a.

Constructor for the **cycle2D** from l and square of its radius.

9b ⟨constructors of the class cycle2D 9a⟩+≡ (61b) ◁9a 9c▷
```
cycle2D(const lst & l, const ex & metr = -unit_matrix(2), const ex & r_squared =0,
  const ex & e =0, const ex & sign = unit_matrix(2));
```
Defines:
cycle2D, used in chunks 9, 10c, 16–23, 25, 26e, 28a, 30–33, 35a, 36, 50a, 51d, 53, 55–57, 61, 62, 64, 66d, 88–91, 93b, 94a, 96, 100, and 102b.
Uses ex 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, l 3a, and metr 3a.

Construction of **cycle2D** from its SFSCc matrix, dimensionality is not supplied because its is known to be 2.

9c ⟨constructors of the class cycle2D 9a⟩+≡ (61b) ◁9b 9d▷
```
cycle2D(const matrix & M, const ex & metr, const ex & e = 0, const ex & sign = 0);
```
Uses cycle2D 9a 9b 15c 15c 15d 15d 54b 55c 61b 62d 62d 62d 62d 62d 77b 77b 77b 77b 77b 89b 89b 89b 89b 92c, ex 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, matrix 11d 16b 16c, and metr 3a.

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.

9d ⟨constructors of the class cycle2D 9a⟩+≡ (61b) ◁9c
```
cycle2D(const cycle & C, const ex & metr = 0);
```
Uses cycle 3a 3a 3b 3b 3c 3d 5a 5a 5a 5a 5a 72c 72d 72d 73a 75d 75d 77a 77a 77a 77a 77a, cycle2D 9a 9b 15c 15c 15d 15d 54b 55c 61b 62d 62d 62d 62d 62d 77b 77b 77b 77b 77b 89b 89b 89b 89b 92c, ex 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, and metr 3a.

The realisation of 2D cycles through matrices with hypercomplex numbers [15,17,19] lead to some important differences with this library using the Clifford algebras. One of them: the determinant of a matrix change sign. The next method return the determinant as it will be calculated on those hypercomplex matrices.

9e ⟨methods specific for class cycle2D 9e⟩≡ (61b) 9f▷
```
public:
virtual inline ex hdet(const ex & e = 0,
  const ex & sign = (new tensdelta)→setflag(status_flags::dynallocated),
  const ex & k_norm = 0) const
{return -det(e, sign, k_norm, true);}
```
Defines:
hdet, used in chunk 22e.
Uses det 6e 84b and ex 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c.

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 [18].

9f ⟨methods specific for class cycle2D 9e⟩+≡ (61b) ◁9e 9g▷
```
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
```
Defines:
focal_length, used in chunks 17d and 33a.
focus, used in chunks 17d, 25d, 26b, 31, 33–35, 52–54, and 90a.
Uses bool 16a, cycle 3a 3a 3b 3b 3c 3d 5a 5a 5a 5a 5a 72c 72d 72d 73a 75d 75d 77a 77a 77a 77a 77a, ex 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, get_l 4a, and k 3a.

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

9g ⟨methods specific for class cycle2D 9e⟩+≡ (61b) ◁9f 10a▷
```
lst roots(const ex & y = 0, bool first = true) const;
```
Defines:
roots, used in chunks 21–23, 25c, 26a, 36, 50b, 52–54, 90, 91e, 94a, 95c, and 98d.
Uses bool 16a and ex 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c.

The next methods is a generalisation of the previous one: it returns intersection points with the line $ax+b$.

10a ⟨methods specific for class cycle2D 9e⟩+≡ (61b) ◁9g 10b▷
```
lst line_intersect(const ex & a, const ex & b) const;
```
Defines:
`line_intersect`, used in chunk 90c.
Uses `ex` 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c.

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. If you do not want imaginary cycles to be shown use the value `"invisible"` for *imaginary_options*.

10b ⟨methods specific for class cycle2D 9e⟩+≡ (61b) ◁10a 10c▷
```
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 string more_options = "",
        bool with_header = true, int points_per_arc = 0, bool asymptote = false,
        const string picture = "", bool only_path=false, bool is_continuation=false,
    const string imaginary_options="withcolor .9*green withpen pencircle scaled 4pt") const;
```
Defines:
`metapost_draw`, used in chunks 11, 92c, 100, and 102b.
Uses `bool` 16a, `ex` 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, and `string` 14a 59d 59d 107b 107c.

Besides inherited **cycle**::*sl2_similarity*() (see § E.1.4), there are further methods for two dimensional cycles to make similarity with complex, dual and double numbers. Real and imaginary parts need to be supplied as two separate matrices. In the first method only two matrices *M1* and *M2* are mandatory, if the rest is not supplied, the method *sl2_similarity*(**const ex** & *M*, **const ex** & *e*,...) will correctly handle this situation.

10c ⟨methods specific for class cycle2D 9e⟩+≡ (61b) ◁10b 11a▷
```
cycle2D sl2_similarity(const ex & M1, const ex & M2, const ex & e,
    const ex & sign,
    bool not_inverse=true,
    const ex & sign_inv = (new tensdelta)->setflag(status_flags::dynallocated)) const;
cycle2D sl2_similarity(const ex & a1, const ex & b1, const ex & c1, const ex & d1,
    const ex & a2, const ex & b2, const ex & c2, const ex & d2,
    const ex & e = 0,
    const ex & sign = (new tensdelta)->setflag(status_flags::dynallocated),
    bool not_inverse=true,
    const ex & sign_inv = (new tensdelta)->setflag(status_flags::dynallocated)) const;
```
Defines:
`sl2_similarity`, used in chunks 12a, 16–18, 23c, 33b, 86, 90, and 91.
Uses `bool` 16a, `cycle2D` 9a 9b 15c 15c 15d 15d 54b 55c 61b 62d 62d 62d 62d 62d 77b 77b 77b 77b 77b 89b 89b 89b 89b 92c, and `ex` 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c.

The similar method provides a drawing output for `Asymptote` [11] with the same meaning of parameters. However, format of *more_options* and *imaginary_options* should be adjusted correspondingly. Currently *asy_draw*() is realised as a wrapper around *metapost_draw*() but this may be changed.

11a ⟨methods specific for class cycle2D 9e⟩+≡ (61b) ◁10c 11b▷

```
inline void asy_draw(ostream & ost, const string picture,
                     const ex & xmin = -5, const ex & xmax = 5,
                     const ex & ymin = -5, const ex & ymax = 5, const lst & color = lst{},
                     const string more_options = "", bool with_header = true,
                     int points_per_arc = 0, const string imaginary_options="rgb(0,.9,0)+4pt") const
{metapost_draw(ost, xmin, xmax, ymin, ymax, color, more_options, with_header,
               points_per_arc, true, picture, false, false, imaginary_options); }

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 string more_options = "",
                     bool with_header = true, int points_per_arc = 0,
                     const string imaginary_options="rgb(0,.9,0)+4pt") const
{metapost_draw(ost, xmin, xmax, ymin, ymax, color, more_options, with_header,
               points_per_arc, true, "", false, false, imaginary_options); }
```

Defines:
`asy_draw`, used in chunks 50d and 53–58.
Uses `bool` 16a, `ex` 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, `metapost_draw` 10b, and `string` 14a 59d 59d 107b 107c.

Finally, we have a similar method which does not issue drawing command, instead it writes a definition for a (array of) path, which may be manipulated later.

11b ⟨methods specific for class cycle2D 9e⟩+≡ (61b) ◁11a

```
inline void asy_path(ostream & ost = std::cout,
                     const ex & xmin = -5, const ex & xmax = 5,
                     const ex & ymin = -5, const ex & ymax = 5,
                     int points_per_arc = 0, bool is_continuation = false) const
{metapost_draw(ost, xmin, xmax, ymin, ymax, lst{}, "", false,
               points_per_arc, true, "", true, is_continuation); }
```

Defines:
`asy_path`, never used.
Uses `bool` 16a, `ex` 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, and `metapost_draw` 10b.

2.7. **An Example: Möbius Invariance of cycles.** A quick illustration of the library usage is the symbolic calculation which proves the Lem. 3.1 from [16]: We check that a Möbius transformation $g \in \mathrm{SL}_2(\mathbb{R})$ acts on cycles by similarity $g : C \to gCg^{-1}$. We use the following predefined objects:

```
cycle2D C(k,lstl,n,m,e);
ex W=lstu,v;
```

Firstly we define a **cycle2D** *C2* by the condition between k, l and m in the generic **cycle2D** C that C passes through some point W.

11c ⟨Moebius transformation of cycles 11c⟩≡ (12b) 12a▷

```
C2 = Cv.subject_to(lst{Cv.passing(W)});
```

Uses `passing` 6b and `subject_to` 6c.

The point gW is defined to be the Möbius transform of W by an arbitrary g.

11d ⟨Moebius transforms of W 11d⟩≡ (16c)

```
const matrix gW=ex_to<matrix>(clifford_moebius_map(sl2_clifford(a, b, c, d, ev), W, ev).subs(sl2_relation1,
  subs_options::algebraic | subs_options::no_pattern).normal());
```

Defines:
`matrix`, used in chunks 3d, 6d, 9c, 14b, 18f, 23b, 25d, 31e, 34e, 35c, 57e, 60a, 65–70, 79c, 81–88, 90, 91, 108, and 109.
Uses `normal` 4b and `subs` 4b.

Finally we verify that the new cycle gCg^{-1} passes through P. This proves Lem. 3.1 from [18].

12a ⟨Moebius transformation of cycles 11c⟩+≡ (12b) ◁11c 16d▷

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

Uses cycle 3a 3a 3b 3b 3c 3d 5a 5a 5a 5a 5a 72c 72d 72d 73a 75d 75d 77a 77a 77a 77a 77a, is_zero 4b, normal 4b, sl2_similarity 7b 10c 61d 62a, subs 4b, and val 6a.

3. Demonstration through example

We illustrate the library usage by the complete program which was used for computer-assisted proofs in the paper [18]. The numerous cross-references between these two papers are active hyperlinks. It is recommended to obtain PDF files for both of them from http://arXiv.org and put into the same local directory. In this case clicking on a reference in a PDF reader will automatically transfer to the appropriate place (even in the other paper).

3.1. Outline of the *main*().

The *main*() procedure does several things:

(i) Makes symbolic calculations related to Möbius invariance;

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

```
⟨Moebius transformation of cycles 11c⟩
⟨K-orbit invariance 16f⟩
⟨Check Moebius transformations of zero cycles 17c⟩
⟨Check transformations of zero cycles by conjugation 18d⟩
cout << endl;
```

(ii) Calculates properties of orthogonality conditions and corresponding inversion in cycles;

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

```
⟨Orthogonality conditions 19c⟩
⟨Two points and orthogonality 20a⟩
⟨One point and orthogonality 20c⟩
⟨Orthogonal line 21a⟩
⟨Inversion in cycle 21e⟩
⟨Reflection in cycle 22e⟩
⟨Yaglom inversion 23b⟩
cout << endl;
```

(iii) Calculates properties of f-orthogonality conditions and second type of inversion;

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

```
⟨Focal orthogonality conditions 23c⟩
⟨One point and f-orthogonality 25b⟩
⟨f-orthogonal line 25c⟩
⟨f-inversion in cycle 25e⟩
cout << endl;
```

(iv) Calculates various length formulae;

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

```
⟨Distances from cycles 26e⟩
⟨Lengths from centre 30c⟩
⟨Lengths from focus 30e⟩
⟨Infinitesimal cycle 32a⟩
cout << endl;
```

(v) Generates `Asymptote` output of the for illustrations.

Since we aiming into two targets simultaneously—validate our software and use it for mathematical proofs—there are many double checks and superfluous calculations. In particular, all checks are done twice: for vector and paravector formalism (see also Rem. 1.1 for required GiNaC version). The positive aspect of this—a better illustration of the library usage.

3.1.1. *The program 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*>.

13a ⟨* 13a⟩≡ 13b ▷
```
⟨license 110⟩
#include <fstream>
#include <cycle.h>

#define par_matr diag_matrix(lst{-1, 0})
#define hyp_matr diag_matrix(lst{-1, 1})
using namespace MoebInv;
using namespace std;
using namespace GiNaC;
```

Defines:
hyp_matr, used in chunk 55c.
par_matr, used in chunks 53–55.
Uses cycle 3a 3a 3b 3b 3c 3d 5a 5a 5a 5a 5a 72c 72d 72d 73a 75d 75d 77a 77a 77a 77a 77a and MoebInv 58e.

We try to make the output more readable both in simple text and LaTeX modes.

13b ⟨* 13a⟩+≡ ◁ 13a 13c ▷
```
#define math_string << (output_latex?"$":"")
//$ (this is to balance dollar signs for LaTeX highlights in Xemacs)
#define wspaces (output_latex?"\\quad ":"  ")
```

Defines:
math_string, used in chunks 17, 19–21, 24, 25, and 27–35.
wspaces, used in chunks 17, 19, and 21–35.

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

13c ⟨* 13a⟩+≡ ◁ 13b 13d ▷
```
⟨Declaration of variables 14a⟩
⟨Subroutines definitions 16e⟩
int main(){
 cout << boolalpha;

 if (output_latex) cout << latex;
```

Defines:
main, never used.

If paravector calculations are not possible the corresponding warning is printed.

13d ⟨* 13a⟩+≡ ◁ 13c 13e ▷
```
#if GINAC_VERSION_ATLEAST(1,7,1)
#else
cerr << "GiNaC version is not sufficiently large to handle paravector calculations." << endl
   << "All false results for paravectors shall be ignored!" << endl;
#endif
```

Uses GINAC_VERSION_ATLEAST 59a 59a and paravector 63a 63c 103c 103c 103c 104d 104d 105a.

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

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

Uses catch 37a 37b.

We end up with drawing illustration to our paper [18].

13f ⟨* 13a⟩+≡ ◁ 13e
```
  ⟨Draw Asymptote pictures 36⟩
}
```

3.1.2. *Declaration of variables.* First we declare all variables from the standard GiNaC classes here.

14a ⟨Declaration of variables 14a⟩≡ (13c) 14b ▷

```
const string 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"),
  k("k"), l("L","l"), m("m"), n("n"), // Cycles parameters
  k1("k1","\\tilde{k}"), l1("l1","\\tilde{l}"), m1("m1","\\tilde{m}"), n1("n1","\\tilde{n}"),
  u("u"), v("v"), u1("u1"), v1("v1"), // Coordinates of points in ℝ^2
  epsilon("eps", "\\epsilon"); // The "infinitesimal" number

const varidx nu2(symbol("nu", "\\nu"), 2), mu2(symbol("mu", "\\mu"), 2);
```

Defines:
`numeric`, used in chunks 5, 6f, 15, 26e, 28a, 29b, 50–52, 54a, 55d, 58a, 59d, 61c, 64c, 66–70, 74–76, 78–80, 84–86, 90–101, 103, and 105–107.
`realsymbol`, used in chunk 96b.
`string`, used in chunks 10b, 11a, 16f, 18a, and 92c.
`varidx`, used in chunks 3e, 36, 65, 66b, 68–70, 75a, 76, 79, 80b, 82–84, 90, 94a, and 104b.
Uses `k` 3a, `l` 3a, `m` 3a, `points` 103a, `u` 100a, and `v` 100a.

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

14b ⟨Declaration of variables 14a⟩+≡ (13c) ◁ 14a 14c ▷

```
const realsymbol sign("si", "\\sigma"), sign1("si1", "\\breve{\\sigma}"), //Signs of e_1^2 of ĕ_1^2
                 sign2("si2", "\\sigma_2"), sign3("si3", "\\sigma_3"),
                 sign4("si4", "\\mathring{\\sigma}"),
                 s("s"), s1("s1", "s_1"), s2("s2", "s_2");
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:
`realsymbol`, used in chunk 96b.
`si`, used in chunks 22e, 28, 29, 36, 50–52, 56, and 57.
`si1`, used in chunks 28, 29, 36, 50–52, and 56.
Uses `jump_fnct` 59d, `l` 3a, and `matrix` 11d 16b 16c.

Here are several expressions which will keep results of calculations.

14c ⟨Declaration of variables 14a⟩+≡ (13c) ◁ 14b 14d ▷

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

Uses `ex` 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, `points` 103a, `u` 100a, and `v` 100a.

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

14d ⟨Declaration of variables 14a⟩+≡ (13c) ◁ 14c 15a ▷

```
const ex M = diag_matrix(lst{-1, sign}), // Metrics of point spaces
   ev = clifford_unit(mu2, M, 0), // Clifford algebra generators in the point space
   M1 = diag_matrix(lst{-1, sign1}), // Metrics of cycles spaces
   evs = clifford_unit(nu2, M1, 1), // Clifford algebra generators in the sphere space
   evh = clifford_unit(nu2, S2, 1), // Clifford algebra generators with Heviside function
   ev4 = clifford_unit(nu2, diag_matrix(lst{-1, sign4}), 2);
```

Defines:
`ex`, used in chunks 3–11, 14c, 16–32, 34–36, 53b, 59–63, 65a, 67–76, 78–80, 82–93, 96c, 97a, and 103–109.

Here we define clifford units for paravector formalism.

15a ⟨Declaration of variables 14a⟩+≡ (13c) ◁14d 15b▷

```
#if GINAC_VERSION_ATLEAST(1,7,1)
const varidx nu1(symbol("nu", "\\nu"), 1), mu1(symbol("mu", "\\mu"), 1);
const ex ep = clifford_unit(mu1, diag_matrix(lst{sign}), 0), // Clifford algebra generators in the point space
    eps = clifford_unit(nu1, diag_matrix(lst{sign1}), 1), // Clifford algebra generators in the sphere space
   eph = clifford_unit(nu1, diag_matrix(lst{jump_fnct(sign2)}), 1), // Clifford algebra generators in the sphere space
    ep4 = clifford_unit(nu1, diag_matrix(lst{sign4}), 2);
```

Defines:
ex, used in chunks 3–11, 14c, 16–32, 34–36, 53b, 59–63, 65a, 67–76, 78–80, 82–93, 96c, 97a, and 103–109.
varidx, used in chunks 3e, 36, 65, 66b, 68–70, 75a, 76, 79, 80b, 82–84, 90, 94a, and 104b.
Uses GINAC_VERSION_ATLEAST 59a 59a and jump_fnct 59d.

If GiNaC version is not sufficient to run paravector formalism, we simply copy values for vector formalism.

15b ⟨Declaration of variables 14a⟩+≡ (13c) ◁15a 15c▷

```
#else
const varidx nu1=nu2, mu1=mu2;
const ex ep = ev,
    eps = evs,
    eph = evh,
    ep4 = ev4;
#endif
```

Defines:
ex, used in chunks 3–11, 14c, 16–32, 34–36, 53b, 59–63, 65a, 67–76, 78–80, 82–93, 96c, 97a, and 103–109.
varidx, used in chunks 3e, 36, 65, 66b, 68–70, 75a, 76, 79, 80b, 82–84, 90, 94a, and 104b.

Now we define instances of **cycle2D** class. Some of them (like *real_line* or generic cycles *C* and *C1*) are constants. First they are done for vector formalism.

15c ⟨Declaration of variables 14a⟩+≡ (13c) ◁15b 15d▷

```
cycle2D C2, C3, C4, C5, C6, C7, C8, C9, C10, C11;

const cycle2D real_linev(0, lst{0, numeric(1)}, 0, ev), // the real line
        Cv(k, lst{l, n}, m, ev), Cv1(k1, lst{l1, n1}, m1, ev); // two generic cycles
const cycle2D Zvinf(0, lst{0, 0}, 1, ev), // the zero-radius cycle at infinity
        Zv(lst{u, v}, ev), Zv1(lst{u, v}, ev, 0, evs), // two generic cycles of zero-radius
        Zv2(lst{u, v}, ev, 0, evs, S2);
```

Defines:
cycle2D, used in chunks 9, 10c, 16–23, 25, 26e, 28a, 30–33, 35a, 36, 50a, 51d, 53, 55–57, 61, 62, 64, 66d, 88–91, 93b, 94a, 96, 100, and 102b.
Uses cycle 3a 3a 3b 3b 3c 3d 5a 5a 5a 5a 5a 72c 72d 72d 73a 75d 75d 77a 77a 77a 77a 77a, k 3a, l 3a, m 3a, numeric 14a 57d, u 100a, and v 100a.

And now—for paravector formalism.

15d ⟨Declaration of variables 14a⟩+≡ (13c) ◁15c 15e▷

```
const cycle2D real_linep(0, lst{0, numeric(1)}, 0, ep), // the real line
        Cp(k, lst{l, n}, m, ep), Cp1(k1, lst{l1, n1}, m1, ep); // two generic cycles
const cycle2D Zpinf(0, lst{0, 0}, 1, ep), // the zero-radius cycle at infinity
        Zp(lst{u, v}, ep), Zp1(lst{u, v}, ep, 0, eps), // two generic cycles of zero-radius
        Zp2(lst{u, v}, ep, 0, eps, S2);
```

Defines:
cycle2D, used in chunks 9, 10c, 16–23, 25, 26e, 28a, 30–33, 35a, 36, 50a, 51d, 53, 55–57, 61, 62, 64, 66d, 88–91, 93b, 94a, 96, 100, and 102b.
Uses cycle 3a 3a 3b 3b 3c 3d 5a 5a 5a 5a 5a 72c 72d 72d 73a 75d 75d 77a 77a 77a 77a 77a, k 3a, l 3a, m 3a, numeric 14a 57d, u 100a, and v 100a.

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

15e ⟨Declaration of variables 14a⟩+≡ (13c) ◁15d 16a▷

```
lst eqns, eqns1,
    vars=lst{k1, l1, m1, n1},
    solns, solns1, // Solutions of linear systems
      sign_val;
```

Here are **relational**s and lists of **relational**s which will be used for automatic simplifications in calculations. They are based on properties of $\mathrm{SL}_2(\mathbb{R})$ and values of the parameters.

16a ⟨Declaration of variables 14a⟩+≡ (13c) ◁15e 16b▷

```
const ex sl2_relation = (c*b ≡ a*d-1), sl2_relation1 = (a ≡ (1+b*c)÷d); // since ad − bc ≡ 1
 const lst signs_cube = lst{pow(sign, 3) ≡ sign, pow(sign1, 3) ≡ sign1}; // s_i^3 ≡ s_i since s_i = −1, 0, 1

const int debug = 0;
const bool output_latex = true;
```

Defines:
bool, used in chunks 4–7, 9–11, 17a, 18e, 29g, 60–62, 67c, 73b, 75b, 78, 82a, 84–86, 90–92, 94a, 100a, 108, and 109.
debug, used in chunks 20b, 21c, 25, 26e, 28a, and 29g.
ex, used in chunks 3–11, 14c, 16–32, 34–36, 53b, 59–63, 65a, 67–76, 78–80, 82–93, 96c, 97a, and 103–109.

Two generic points on the plain are defined as constant vectors (2×1matrices).

16b ⟨Declaration of variables 14a⟩+≡ (13c) ◁16a 16c▷

```
const matrix W(2,1, lst{u, v}), W1(2,1, lst{u1, v1}),
    Wbar(2,1, lst{u, -v}); // Needed for paravector formalism
```

Defines:
matrix, used in chunks 3d, 6d, 9c, 14b, 18f, 23b, 25d, 31e, 34e, 35c, 57e, 60a, 65–70, 79c, 81–88, 90, 91, 108, and 109.
Uses paravector 63a 63c 103c 103c 103c 104d 104d 105a, u 100a, and v 100a.

We will also frequently use their Möbius transforms.

16c ⟨Declaration of variables 14a⟩+≡ (13c) ◁16b 30b▷

```
const matrix gW1=ex_to<matrix>(clifford_moebius_map(sl2_clifford(a, b, c, d, ev), W1, ev).subs(sl2_relation1,
  subs_options::algebraic | subs_options::no_pattern).normal());
⟨Moebius transforms of W 11d⟩
```

Defines:
matrix, used in chunks 3d, 6d, 9c, 14b, 18f, 23b, 25d, 31e, 34e, 35c, 57e, 60a, 65–70, 79c, 81–88, 90, 91, 108, and 109.
Uses normal 4b and subs 4b.

We make rhe same check as in § 2.7 now for paravectors.

16d ⟨Moebius transformation of cycles 11c⟩+≡ (12b) ◁12a

```
    C2 = Cp.subject_to(lst{Cp.passing(W)});
    cout << "Conjugation of a cycle comes through Moebius transformation for paravectors: "
      << C2.sl2_similarity(a, b, c, d, eps, S2, true, S2).val(gW).subs(sl2_relation1,
                                   subs_options::algebraic | subs_options::no_pattern).normal().is_zero()
      << endl << endl;
```

Uses cycle 3a 3a 3b 3b 3c 3d 5a 5a 5a 5a 5a 72c 72d 72d 73a 75d 75d 77a 77a 77a 77a 77a, is_zero 4b, normal 4b, passing 6b, sl2_similarity 7b 10c 61d 62a, subject_to 6c, subs 4b, and val 6a.

We repeat some calculations several times for various values of parameters, such calculations are gathered here as subroutines.

16e ⟨Subroutines definitions 16e⟩≡ (13c)

```
    ⟨Parabolic Cayley transform of cycles 35a⟩
    ⟨Check conformal property 28c⟩
    ⟨Print perpendicular 30a⟩
    ⟨Focal length checks 31c⟩
    ⟨Infinitesimal cycle calculations 32c⟩
```

3.2. Möbius Transformation and Conjugation of Cycles.

3.2.1. *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 [18, Lem. 2.2] 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 $C2$. First for vector form.

16f ⟨K-orbit invariance 16f⟩≡ (12b) 17a▷

```
auto K_inv = [](string S, const ex & e) {
cycle2D C2 = cycle2D(1, lst{0, (pow(t,-1)-sign*t)÷2}, 1, e);
cout << "A K-orbit is preserved " << S << C2.sl2_similarity(cos(x), sin(x), -sin(x), cos(x), e).is_equal(C2)
```

Uses cycle2D 9a 9b 15c 15c 15d 15d 54b 55c 61b 62d 62d 62d 62d 62d 77b 77b 77b 77b 77b 89b 89b 89b 89b 92c,
ex 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, is_equal 4b, sl2_similarity 7b 10c 61d 62a, and string 14a 59d 59d 107b 107c.

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

17a ⟨K-orbit invariance 16f⟩+≡ (12b) ◁16f 17b▷

```
≪ ", and passing (0, t): " ≪ (bool)ex_to<relational>(C2.passing(lst{0, t})) ≪ endl; };
```

Uses bool 16a and passing 6b.

Now we do the check both for vectors and paravectors.

17b ⟨K-orbit invariance 16f⟩+≡ (12b) ◁17a

```
K_inv("for vectors: ", ev);
K_inv("for paravectors: ", ep);
```

3.2.2. *Transformation of Zero-Radius Cycles.* Firstly, we check some basic information about the zero-radius cycles. This mainly done to verify our library.

17c ⟨Check Moebius transformations of zero cycles 17c⟩≡ (12b) 17d▷

```
cout ≪ wspaces ≪ "Determinant of zero-radius Z1 cycle in metric e is for vector: "
  math_string ≪ canonicalize_clifford(Zv1.det(ev, S2)) math_string ≪ endl;
 cout ≪ wspaces ≪ "The opposite value for paravector: "
    ≪ canonicalize_clifford(Zv1.det(ev, S2)+Zp1.det(ep, S2)).normal().is_zero() ≪ endl;
```

Uses cycle 3a 3a 3b 3b 3c 3d 5a 5a 5a 5a 5a 72c 72d 72d 73a 75d 75d 77a 77a 77a 77a 77a, det 6e 84b, is_zero 4b, math_string 13b, normal 4b, paravector 63a 63c 103c 103c 103c 104d 104d 105a, and wspaces 13b.

17d ⟨Check Moebius transformations of zero cycles 17c⟩+≡ (12b) ◁17c 17e▷

```
cout ≪ wspaces ≪ "Focus of zero-radius cycle is (vector): " math_string
   ≪ Zv1.focus(ev) math_string ≪ endl;
cout ≪ wspaces ≪ "The same value for paravector: "
   ≪ (Zv1.focus(ev,true)-Zp1.focus(ep,true)).evalm().is_zero() ≪ endl;
cout ≪ wspaces ≪ "Centre of zero-radius cycle is (vector): " math_string
   ≪ Zv1.center(ev) math_string ≪ endl;
cout ≪ wspaces ≪ "The same value for paravector: "
   ≪ (Zv1.center(ev,true)-Zp1.center(ep,true)).evalm().is_zero() ≪ endl;
cout ≪ wspaces ≪ "Focal length of zero-radius cycle is (vector): " math_string
   ≪ Zv1.focal_length() math_string ≪ endl;
cout ≪ wspaces ≪ "The same value for paravector: "
   ≪ (Zv1.center(ev,true)-Zp1.center(ep,true)).evalm().is_zero() ≪ endl;
```

Uses center 5f, cycle 3a 3a 3b 3b 3c 3d 5a 5a 5a 5a 5a 72c 72d 72d 73a 75d 75d 77a 77a 77a 77a 77a, focal_length 9f, focus 9f, is_zero 4b, math_string 13b, paravector 63a 63c 103c 103c 103c 104d 104d 105a, and wspaces 13b.

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.

17e ⟨Check Moebius transformations of zero cycles 17c⟩+≡ (12b) ◁17d 17f▷

```
auto Z_rad_tr=[](const cycle2D & Z1, const ex & e, const ex & es)
   {return canonicalize_clifford(Z1.sl2_similarity(a, b, c, d, e, S2).det(es, S2)).subs(sl2_relation1,
                                                  subs_options::algebraic | subs_options::no_pattern); };

cout ≪ "Image of the zero-radius cycle under Moebius transform has zero radius vector: "
   ≪ Z_rad_tr(Zv1,ev,evs).is_zero()
   ≪ " and paravector: " ≪ Z_rad_tr(Zp1,ep,eps).is_zero() ≪ endl;
```

Uses cycle 3a 3a 3b 3b 3c 3d 5a 5a 5a 5a 5a 72c 72d 72d 73a 75d 75d 77a 77a 77a 77a 77a, cycle2D 9a 9b 15c 15c 15d 15d 54b 55c 61b 62d 62d 62d 62d 62d 77b 77b 77b 77b 77b 89b 89b 89b 89b 92c, det 6e 84b, ex 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, is_zero 4b, paravector 63a 63c 103c 103c 103c 104d 104d 105a, sl2_similarity 7b 10c 61d 62a, and subs 4b.

We calculate the Möbius transformation of the centre of Z

17f ⟨Check Moebius transformations of zero cycles 17c⟩+≡ (12b) ◁17e 18a▷

```
u2 = gW.op(0);
v2 = gW.op(1);
```

Uses op 4b.

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

18a ⟨Check Moebius transformations of zero cycles 17c⟩+≡ (12b) ◁17f 18b▷
```
auto Z_center=[](string S, const cycle2D & Z, const ex & e) {
    C2 = Z.sl2_similarity(a, b, c, d, e);
    K = C2.get_k();
    L0 = C2.get_l(0);
    L1 = C2.get_l(1).normal();
```

Uses `cycle2D` 9a 9b 15c 15c 15d 15d 54b 55c 61b 62d 62d 62d 62d 62d 77b 77b 77b 77b 77b 89b 89b 89b 89b 92c, `ex` 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, `get_k` 3e, `get_l` 4a, `normal` 4b, `sl2_similarity` 7b 10c 61d 62a, and `string` 14a 59d 59d 107b 107c.

And we finally check that gW coincides with the centre of the transformed cycle *C2*. This proves [18, Lem. 3.1].

18b ⟨Check Moebius transformations of zero cycles 17c⟩+≡ (12b) ◁18a 18c▷
```
  cout <<"The centre of the Moebius transformed zero-radius cycle for " << S
   << 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` 3a 3a 3b 3b 3c 3d 5a 5a 5a 5a 5a 72c 72d 72d 73a 75d 75d 77a 77a 77a 77a 77a and `subs` 4b.

Now its called for vectors and paravectors.

18c ⟨Check Moebius transformations of zero cycles 17c⟩+≡ (12b) ◁18b
```
  Z_center("vector: ", Zv, ev);
  Z_center("paravector: ", Zp, ep);
```

Uses `paravector` 63a 63c 103c 103c 103c 104d 104d 105a.

3.2.3. *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 $C_2 = CZC$.

18d ⟨Check transformations of zero cycles by conjugation 18d⟩≡ (12b) 18f▷
```
auto Z_conjugated=[](const cycle2D & Z, const cycle2D & C, const ex & e) {
    ⟨Check either vector formalism is used 18e⟩
```

Uses `cycle2D` 9a 9b 15c 15c 15d 15d 54b 55c 61b 62d 62d 62d 62d 62d 77b 77b 77b 77b 77b 89b 89b 89b 89b 92c and `ex` 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c.

On a number of occasions we will need to check either vector or paravector formalism is used.

18e ⟨Check either vector formalism is used 18e⟩≡ (18d 20–23 25 26e 30c)
```
    bool is_vector = (ex_to<idx>(e.op(1)).get_dim() ≡ 2);
```

Uses `bool` 16a, `get_dim` 3e, and `op` 4b.

The rest of the check for cycle conjuagation.

18f ⟨Check transformations of zero cycles by conjugation 18d⟩+≡ (12b) ◁18d 19a▷
```
    matrix S1=ex_to<matrix>(diag_matrix(lst{1, s1})),  S2=ex_to<matrix>(diag_matrix(lst{1, s2}));
    lst square_sub=lst{pow(s1,2)≡1, pow(s2,2)≡1};
    cycle2D Zn = Z.cycle_similarity(C, e, S1, S2, pow(S1,-1).evalm());
    cout << "Image of the zero-radius cycle under cycle similarity has zero radius for "
    << (is_vector? "" : "para") << "vector: " << canonicalize_clifford(Zn.det(e, S1)).subs(square_sub,
                                         subs_options::algebraic | subs_options::no_pattern).normal().is_zero()
    << endl;
```

Uses `cycle` 3a 3a 3b 3b 3c 3d 5a 5a 5a 5a 5a 72c 72d 72d 73a 75d 75d 77a 77a 77a 77a 77a, `cycle2D` 9a 9b 15c 15c 15d 15d 54b 55c 61b 62d 62d 62d 62d 62d 77b 77b 77b 77b 77b 89b 89b 89b 89b 92c, `cycle_similarity` 7e, `det` 6e 84b, `is_zero` 4b, `matrix` 11d 16b 16c, `normal` 4b, and `subs` 4b.

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 [18, Lem. 4.4]

19a ⟨Check transformations of zero cycles by conjugation 18d⟩+≡ (12b) ◁18f 19b▷
```
lst Pc=ex_to<lst>(Zn.center(diag_matrix(lst{-1,-s2*s1})));
if (is_vector)
    P=C.moebius_map(Z.center(diag_matrix(lst{-1,-s2÷s1})));
else
    P=C.moebius_map(Z.center(diag_matrix(lst{-1,s2÷s1})));

cout <<"The centre of the conjugated zero-radius cycle coinsides with Moebius trans for "
    << (is_vector? "" : "para") << "vector: " << equality((P.op(0)-Pc.op(0)).normal().subs(square_sub,
                                                          subs_options::algebraic))
    << ", " << equality((P.op(1)-Pc.op(1)).normal().subs(square_sub,subs_options::algebraic))
    << endl; };
```

Uses center 5f, cycle 3a 3a 3b 3b 3c 3d 5a 5a 5a 5a 5a 72c 72d 72d 73a 75d 75d 77a 77a 77a 77a 77a, moebius_map 8a 87b, normal 4b, op 4b, and subs 4b.

Finally checks are called in vector and paravector cases.

19b ⟨Check transformations of zero cycles by conjugation 18d⟩+≡ (12b) ◁19a
```
Z_conjugated(Zv, Cv, ev);
Z_conjugated(Zp, Cp, ep);
```

3.3. Orthogonality of Cycles.

3.3.1. *Various orthogonality conditions.* We calculate orthogonality condition between two **cycle2D**s by the identity $\Re\operatorname{tr}(C_1C_2) = 0$. The expression are stored in variables, which will be used later in our calculations.
Here is the orthogonality of two generic **cycle2D**s. . .

19c ⟨Orthogonality conditions 19c⟩≡ (12c) 19d▷
```
cout << wspaces << "The orthogonality in vectors is: " math_string
 << (ex)Cv.is_orthogonal(Cv1, evs, S2) math_string << endl
 << "for paravectors is the same: "
 << Cv.is_orthogonal(Cv1, evs, S2).is_equal(Cp.is_orthogonal(Cp1, eps, S2)) << endl;
```

Uses ex 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, is_equal 4b, is_orthogonal 8c, math_string 13b, and wspaces 13b.

. . . and then its reduction to orthogonality of two straight lines.

19d ⟨Orthogonality conditions 19c⟩+≡ (12c) ◁19c 19e▷
```
cout << wspaces << "The orthogonality of two lines is: " math_string
 << (ex)Cv.subs(k ≡ 0).is_orthogonal(Cv1.subs(k1 ≡ 0), evs, S2) math_string << endl;
```

Uses ex 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, is_orthogonal 8c, k 3a, math_string 13b, subs 4b, and wspaces 13b.

Here is the orthogonality of a generic **cycle2D** to a zero-radius **cycle2D**. This reduces to concurrence of the centre the zero-radius and generic cycle.

19e ⟨Orthogonality conditions 19c⟩+≡ (12c) ◁19d 19f▷
```
cout << wspaces << "The orthogonality to z-r-cycle is: " math_string
 << (ex)Cv.is_orthogonal(Zv, evs)  math_string << endl
    << "for paravectors is the same: " <<
    Cv.is_orthogonal(Zv, evs).is_equal(Cp.is_orthogonal(Zp, eps)) << endl;
```

Uses cycle 3a 3a 3b 3b 3c 3d 5a 5a 5a 5a 5a 72c 72d 72d 73a 75d 75d 77a 77a 77a 77a 77a, ex 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, is_equal 4b, is_orthogonal 8c, math_string 13b, and wspaces 13b.

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

19f ⟨Orthogonality conditions 19c⟩+≡ (12c) ◁19e
```
cout << wspaces << "The orthogonality of two z-r-cycle is: " math_string
 << (ex)cycle2D(lst{u1, v1}, ev, 0, S2).is_orthogonal(Zv, evs) math_string << endl
    << "for paravectors is the same: "
 << cycle2D(lst{u1, v1}, ev, 0, S2).is_orthogonal(Zv, evs).is_equal(
                                   cycle2D(lst{u1, v1}, ep, 0, S2).is_orthogonal(Zp, eps)) << endl;
```

Uses cycle 3a 3a 3b 3b 3c 3d 5a 5a 5a 5a 5a 72c 72d 72d 73a 75d 75d 77a 77a 77a 77a 77a, cycle2D 9a 9b 15c 15c 15d 15d 54b 55c 61b 62d 62d 62d 62d 62d 77b 77b 77b 77b 77b 89b 89b 89b 89b 92c, ex 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, is_equal 4b, is_orthogonal 8c, math_string 13b, and wspaces 13b.

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.

20a ⟨Two points and orthogonality 20a⟩≡ (12c) 20b ▷

```
C2 = Cv1.subject_to(lst{Cv1.passing(W),
        Cv1.passing(W1),
        Cv1.is_orthogonal(Cv, evs)}, vars);
```

Uses `is_orthogonal` 8c, `passing` 6b, and `subject_to` 6c.

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

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

20b ⟨Two points and orthogonality 20a⟩+≡ (12c) ◁ 20a

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

Uses `debug` 16a, `get_k` 3e, and `math_string` 13b.

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*.

20c ⟨One point and orthogonality 20c⟩≡ (12c) 20e ▷

```
auto Ortho_inv=[](const cycle2D & C, const cycle2D & C1, const ex & e, const ex & es) {
    ⟨Check either vector formalism is used 18e⟩
    C2 = C1.subject_to(lst{C1.passing(W),
            C1.is_orthogonal(C, es)});
```

Uses `cycle2D` 9a 9b 15c 15c 15d 15d 54b 55c 61b 62d 62d 62d 62d 62d 77b 77b 77b 77b 77b 89b 89b 89b 89b 92c, `ex` 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, `is_orthogonal` 8c, `passing` 6b, and `subject_to` 6c.

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

20e ⟨One point and orthogonality 20c⟩+≡ (12c) ◁ 20c 20f ▷

```
    P = C.moebius_map(is_vector? W : Wbar, e, -M1);

    C3 = C1.subject_to(lst{C1.passing(P),
            C1.passing(W),
            C1.is_orthogonal(C, es)});
```

Uses `is_orthogonal` 8c, `moebius_map` 8a 87b, `passing` 6b, and `subject_to` 6c.

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

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

20f ⟨One point and orthogonality 20c⟩+≡ (12c) ◁ 20e 20g ▷

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

Uses `is_equal` 4b.

(ii) that cycle *C2* passes through the inversion P as well.

20g ⟨One point and orthogonality 20c⟩+≡ (12c) ◁ 20f 20h ▷

```
     << "Orthogonal cycle passes through the transformed point "
     << (is_vector? "" : "para") << "vector: "
     << C2.val(P).normal().is_zero() << endl << endl;
  };
```

Uses `cycle` 3a 3a 3b 3b 3c 3d 5a 5a 5a 5a 5a 72c 72d 72d 73a 75d 75d 77a 77a 77a 77a 77a, `is_zero` 4b, `normal` 4b, and `val` 6a.

Finally we make both checks.

20h ⟨One point and orthogonality 20c⟩+≡ (12c) ◁ 20g

```
  Ortho_inv(Cv, Cv1, ev, evs);
  Ortho_inv(Cp, Cp1, ep, eps);
```

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*.

21a ⟨Orthogonal line 21a⟩≡ (12c) 21b▷

```
auto Ortho_line=[](const cycle2D & C, const cycle2D & C1, const ex & e, const ex & es) {
   ⟨Check either vector formalism is used 18e⟩
   C4 = C1.subject_to(lst{C1.passing(W), C1.passing(P), C1.is_linear()});
 cout << "For " << (is_vector? "" : "para") << "vectors" << endl
  << wspaces << "Line through point and its inverse is orthogonal: " << C4.cycle_product(C, es).is_zero()
   << endl;
```

Uses `cycle2D` 9a 9b 15c 15c 15d 15d 54b 55c 61b 62d 62d 62d 62d 62d 77b 77b 77b 77b 77b 89b 89b 89b 89b 92c, `cycle_product` 8b 84c, `ex` 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, `is_linear` 8e, `is_zero` 4b, `passing` 6b, `subject_to` 6c, and `wspaces` 13b.

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.

21b ⟨Orthogonal line 21a⟩+≡ (12c) ◁21a 21c▷

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

Uses `center` 5f, `math_string` 13b, `normal` 4b, `op` 4b, `roots` 9g, and `wspaces` 13b.

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*).

21c ⟨Orthogonal line 21a⟩+≡ (12c) ◁21b 21d▷

```
 if (is_vector)
   P1 = C.moebius_map(lst{u3+u, v3+v}, e, -M1);
 else
   P1 = C.moebius_map(lst{u3+u, -v3-v}, e, -M1);
 cout << wspaces << "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 << wspaces << "Conjugated vector to (u, v) is: " math_string
      << "(" << (P1.op(0)-u3).normal() << ", "
      << (P1.op(1)-v3).normal() << ")" math_string << endl; };
```

Uses `debug` 16a, `is_zero` 4b, `math_string` 13b, `moebius_map` 8a 87b, `normal` 4b, `op` 4b, `u` 100a, `v` 100a, and `wspaces` 13b.

Finally we make both checks.

21d ⟨Orthogonal line 21a⟩+≡ (12c) ◁21c

```
 Ortho_line(Cv, Cv1, ev, evs);
 Ortho_line(Cp, Cp1, ep, eps);
```

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

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

21e ⟨Inversion in cycle 21e⟩≡ (12c) 22a▷

```
auto Ghost_cycle=[](const cycle2D & C, const cycle2D & C1, const ex & e, const ex & es) {
   ⟨Check either vector formalism is used 18e⟩
   C5 = cycle2D(lst{u3, -v3*jump_fnct(sign)}, e, C.radius_sq(e, M1)).subs(signs_cube,
                                          subs_options::algebraic | subs_options::no_pattern);
```

Uses `cycle2D` 9a 9b 15c 15c 15d 15d 54b 55c 61b 62d 62d 62d 62d 62d 77b 77b 77b 77b 77b 89b 89b 89b 89b 92c, `ex` 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, `jump_fnct` 59d, `radius_sq` 6f, and `subs` 4b.

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

22a ⟨Inversion in cycle 21e⟩+≡ (12c) ◁21e 22b▷

```
cout << "For " << (is_vector? "" : "para") << "vectors" << endl
    << wspaces << "Ghost cycle has common roots with C : "
    << (C5.val(lst{C.roots().op(0), 0}).normal().is_zero()
       ∧ C5.val(lst{C.roots().op(1), 0}).normal().is_zero()) << endl
    << wspaces << "$\\chi(\\sigma)$-centre of ghist cycle is equal to "
    "$\\breve{\\sigma}$-centre of C: "
    << (C5.center(diag_matrix(lst{-1,jump_fnct(sign)}), true)-C.center(es, true)).normal().is_zero()
    << endl;
```

Uses center 5f, cycle 3a 3a 3b 3b 3c 3d 5a 5a 5a 5a 5a 72c 72d 72d 73a 75d 75d 77a 77a 77a 77a 77a, is_zero 4b, jump_fnct 59d, normal 4b, op 4b, roots 9g, val 6a, and wspaces 13b.

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

22b ⟨Inversion in cycle 21e⟩+≡ (12c) ◁22a 22c▷

```
P1 = C5.moebius_map(is_vector? W : Wbar, e, diag_matrix(lst{1, -jump_fnct(sign)}));
P = C.moebius_map(is_vector? W : Wbar, e, -M1);
```

Uses jump_fnct 59d and moebius_map 8a 87b.

The final check: *P1* (inversion in *C5* in terms of *sign*) coincides with *P*—the inversion in *C* in terms of *sign1*, see chunk 20d.

22c ⟨Inversion in cycle 21e⟩+≡ (12c) ◁22b 22d▷

```
cout << wspaces << "Inversion in (C-ghost, sign) coincides with inversion in (C, sign1): "
    << (P1-P).subs(signs_cube, subs_options::algebraic | subs_options::no_pattern).normal().is_zero()
    << endl; };
```

Uses is_zero 4b, normal 4b, subs 4b, and wspaces 13b.

Finally we make both checks.

22d ⟨Inversion in cycle 21e⟩+≡ (12c) ◁22c

```
Ghost_cycle(Cv, Cv1, ev, evs);
Ghost_cycle(Cp, Cp1, ep, eps);
```

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 [18, Lem. 4.5]. The cycle *C9* is defined through the value *C.det*(), to make this working for both vector and aparvector formalism we need to set the parameter *fix_paravector* = **true** or employ *C.hdet*() method, which set this automatically.

22e ⟨Reflection in cycle 22e⟩≡ (12c) 23a▷

```
for (si=-1; si<2; si+=2) {
   auto Inv_RL=[](const cycle2D & C, const cycle2D & C1, const cycle2D & real_line,
                  const ex & e, const ex & es) {
      ⟨Check either vector formalism is used 18e⟩
      C9 = cycle2D(k, lst{l, n+si*sqrt(C.hdet(es)*sign1)},m,es);
      cout << "For " << (is_vector? "" : "para") << "vectors" << endl
      << wspaces << "Inversion to the real line (with " << (si≡-1? "-" : "+") << " sign): " << endl
      << wspaces << "Conjugation of the real line is the cycle C: "
      << real_line.cycle_similarity(C9, es).subs(pow(sign1,2)≡1, subs_options::algebraic).is_equal(C) << endl
      << wspaces << "Conjugation of the cycle C is the real line: "
      << C.cycle_similarity(C9, es).subs(pow(sign1,2)≡1, subs_options::algebraic).is_equal(real_line) << endl
```

Uses cycle 3a 3a 3b 3b 3c 3d 5a 5a 5a 5a 5a 72c 72d 72d 73a 75d 75d 77a 77a 77a 77a 77a, cycle2D 9a 9b 15c 15c 15d 15d 54b 55c 61b 62d 62d 62d 62d 62d 77b 77b 77b 77b 77b 89b 89b 89b 89b 92c, cycle_similarity 7e, ex 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, hdet 9e, is_equal 4b, k 3a, l 3a, m 3a, si 14b, subs 4b, and wspaces 13b.

We also check two additional properties which caracterises the inversion cycle *C9* in term of common roots of C [18, Lem. 2] and C passing through *C9* centre [18, Lem. 3].

23a ⟨Reflection in cycle 22e⟩+≡ (12c) ◁22e

```
    << wspaces << "Inversion cycle has common roots with C: "
    << (C9.val(lst{C.roots().op(0), 0}).numer().normal().is_zero()
      ∧ C9.val(lst{C.roots().op(1), 0}).numer().normal().is_zero()) << endl
    << wspaces << "C passing the centre of inversion cycle: "
    << cycle2D(C, es).val(C9.center()).numer().subs(sign1≡sign, subs_options::no_pattern).normal()
    .subs(pow(sign,2)≡1, subs_options::algebraic | subs_options::no_pattern).is_zero() << endl; };
  Inv_RL(Cv, Cv1, real_linev, ev, evs);
  Inv_RL(Cp, Cp1, real_linep, ep, eps);
}
```

Uses center 5f, cycle 3a 3a 3b 3b 3c 3d 5a 5a 5a 5a 5a 72c 72d 72d 73a 75d 75d 77a 77a 77a 77a 77a, cycle2D 9a 9b 15c 15c 15d 15d 54b 55c 61b 62d 62d 62d 62d 62d 77b 77b 77b 77b 77b 89b 89b 89b 89b 92c, is_zero 4b, normal 4b, op 4b, passing 6b, roots 9g, subs 4b, val 6a, and wspaces 13b.

3.3.6. *Yaglom inversion of the second kind.* In the book [30, § 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 [16, Prop.4.5].

23b ⟨Yaglom inversion 23b⟩≡ (12c)

```
  auto Yaglom_inv=[](const cycle2D & real_line, const ex & e) {
    ⟨Check either vector formalism is used 18e⟩
    cout << "For " << (is_vector? "" : "para") << "vectors "
    << "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(is_vector? W : Wbar))).subs(sign≡0)
     -matrix(2,1,lst{u, 2*(k*pow(u-l,2)+m)-v})).normal().is_zero() << endl; };

  Yaglom_inv(real_linev, ev);
  Yaglom_inv(real_linep, ep);
```

Uses cycle2D 9a 9b 15c 15c 15d 15d 54b 55c 61b 62d 62d 62d 62d 62d 77b 77b 77b 77b 77b 89b 89b 89b 89b 92c, ex 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, is_zero 4b, k 3a, l 3a, m 3a, matrix 11d 16b 16c, moebius_map 8a 87b, normal 4b, subs 4b, u 100a, and v 100a.

3.4. **Focal Orthogonality.**

We study now the focal orthogonality condition (f-orthogonality), [18, § 4.3].

3.4.1. *Expressions for f-orthogonality.* One more simple consistency check: the *real_line* is invariant under all Möbius transformations.

23c ⟨Focal orthogonality conditions 23c⟩≡ (12d) 24a▷

```
  auto Focal_orth_cond=[](const cycle2D & real_line, const ex & e) {
    ⟨Check either vector formalism is used 18e⟩
    cout << "For " << (is_vector? "" : "para") << "vectors"
    << wspaces << "The real line is Moebius invariant: "
    << real_line.is_equal(real_line.sl2_similarity(a, b, c, d, e)) << endl; };
  Focal_orth_cond(real_linev,evs);
  Focal_orth_cond(real_linep,eps);
```

Uses cycle2D 9a 9b 15c 15c 15d 15d 54b 55c 61b 62d 62d 62d 62d 62d 77b 77b 77b 77b 77b 89b 89b 89b 89b 92c, ex 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, is_equal 4b, sl2_similarity 7b 10c 61d 62a, and wspaces 13b.

Formulae for focal orthogonality:

24a ⟨Focal orthogonality conditions 23c⟩+≡ (12d) ◁23c 24b▷
```
cout << "Reflection in the real line (vector): "
        math_string << Zv.cycle_similarity(real_linev, evs).normalize()
        math_string << endl
  << wspaces << "for paravector is the same: "
  << Zv.cycle_similarity(real_linev, evs).is_equal(Zp.cycle_similarity(real_linep, eps),true,true) << endl;

cout << "Reflection of the real line in cycle C (vectors): " << endl
  math_string << real_linev.cycle_similarity(Cv, evs, S2, S3) math_string << endl
    << wspaces << "for paravectors is the same: "
   << real_linev.cycle_similarity(Cv, evs, S2, S3).is_equal(real_linep.cycle_similarity(Cp, eps, S2, S3),true,true)
    << endl;
```

Uses cycle 3a 3a 3b 3b 3c 3d 5a 5a 5a 5a 5a 72c 72d 72d 73a 75d 75d 77a 77a 77a 77a 77a, cycle_similarity 7e, is_equal 4b, math_string 13b, normalize 5e, paravector 63a 63c 103c 103c 103c 104d 104d 105a, and wspaces 13b.

The focal orthogonality condition between two different cycles is calculated by the identity [18, § 4.3]

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

Here is f-orthogonality of two generic **cycle2D**s...

24b ⟨Focal orthogonality conditions 23c⟩+≡ (12d) ◁24a 24c▷
```
cout << "The f-orthogonality is (vectors): " math_string
  << (ex)Cv.is_f_orthogonal(Cv1, evs, S2) math_string << endl
  << wspaces << "for paravectors is the same: "
  << Cv.is_f_orthogonal(Cv1, evs, S2).is_equal(Cp.is_f_orthogonal(Cp1, eps, S2)) << endl;
```

Uses ex 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, is_equal 4b, is_f_orthogonal 8d, math_string 13b, and wspaces 13b.

...and its reduction to the straight lines case.

24c ⟨Focal orthogonality conditions 23c⟩+≡ (12d) ◁24b 24d▷
```
cout << wspaces << "The f-orthogonality of two lines is (vectors): " math_string
  << (ex)Cv.subs(k ≡ 0).is_f_orthogonal(Cv1.subs(k1≡0), evs, S2) math_string << endl
  << wspaces << "for paravectors is the same: "
  << Cv.subs(k ≡ 0).is_f_orthogonal(Cv1.subs(k1≡0), evs, S2).is_equal(
                          Cp.subs(k ≡ 0).is_f_orthogonal(Cp1.subs(k1≡0), eps, S2)) << endl;
```

Uses ex 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, is_equal 4b, is_f_orthogonal 8d, k 3a, math_string 13b, subs 4b, and wspaces 13b.

Here is f-orthogonality of a generic **cycle2D** to a zero-radius **cycle2D**.

24d ⟨Focal orthogonality conditions 23c⟩+≡ (12d) ◁24c 24e▷
```
cout << wspaces << "The f-orthogonality to z-r-cycle is first way (vectors): " << endl
  math_string << (ex)Cv.is_f_orthogonal(Zv1, evs, S2) math_string << endl
  << wspaces << "for paravectors is the same: "
  << Cv.is_f_orthogonal(Zv1, evs, S2).is_equal(Cp.is_f_orthogonal(Zp1, eps, S2)) << endl;
```

Uses cycle 3a 3a 3b 3b 3c 3d 5a 5a 5a 5a 5a 72c 72d 72d 73a 75d 75d 77a 77a 77a 77a 77a, ex 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, is_equal 4b, is_f_orthogonal 8d, math_string 13b, and wspaces 13b.

Since f-orthogonality is not symmetric [18, § 4.3], we calculate separately f-orthogonality of a zero-radius **cycle2D** to a generic **cycle2D**.

24e ⟨Focal orthogonality conditions 23c⟩+≡ (12d) ◁24d 25a▷
```
cout << wspaces << "The f-orthogonality to z-r-cycle in second way (vectors): " << endl
  math_string << (ex)Zv1.is_f_orthogonal(Cv, evs, S2) math_string << endl
  << wspaces << "for paravectors is the same: "
  << Zv1.is_f_orthogonal(Cv, evs, S2).is_equal(Zp1.is_f_orthogonal(Cp, eps, S2)) << endl;
```

Uses cycle 3a 3a 3b 3b 3c 3d 5a 5a 5a 5a 5a 72c 72d 72d 73a 75d 75d 77a 77a 77a 77a 77a, ex 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, is_equal 4b, is_f_orthogonal 8d, math_string 13b, and wspaces 13b.

Here is f-orthogonality of two zero-radius **cycle2D**s.

25a ⟨Focal orthogonality conditions 23c⟩+≡ (12d) ◁24e

```
//C9 = cycle2D(lst{u1, v1}, e);
cout << wspaces << "The f-orthogonality of two z-r-cycle is (vectors): " << endl
 math_string << (ex)Zv1.is_f_orthogonal(cycle2D(lst{u1, v1}, ev), evs, S2) math_string << endl
<< wspaces << "for paravectors is the same: "
<< Zv1.is_f_orthogonal(cycle2D(lst{u1, v1}, ev), evs, S2).is_equal(
                          Zp1.is_f_orthogonal(cycle2D(lst{u1, v1}, ep), eps, S2)) << endl;
```

Uses cycle 3a 3a 3b 3b 3c 3d 5a 5a 5a 5a 5a 72c 72d 72d 73a 75d 75d 77a 77a 77a 77a 77a, cycle2D 9a 9b 15c 15c 15d 15d 54b 55c 61b 62d 62d 62d 62d 62d 77b 77b 77b 77b 77b 89b 89b 89b 89b 92c, ex 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, is_equal 4b, is_f_orthogonal 8d, math_string 13b, and wspaces 13b.

3.4.2. *Properies of f-orthogonality.* Find the parameters of cycle passing through a point and f-orthogonal to the given one

25b ⟨One point and f-orthogonality 25b⟩≡ (12d)

```
cycle2D Cv6 = Cv1.subject_to(lst{Cv1.passing(W), Cv.is_f_orthogonal(Cv1, evs)}),
   Cp6 = Cp1.subject_to(lst{Cp1.passing(W), Cp.is_f_orthogonal(Cp1, eps)});
 if (debug > 1)
    cout << "Cycle f-orthogonal to (k, (l, n), m) is (vectors): " << endl
       math_string << C6 math_string << endl
        << wspaces << "for paravectors is the same: "
        << Cv6.is_equal(Cp6,true, true);
```

Uses cycle2D 9a 9b 15c 15c 15d 15d 54b 55c 61b 62d 62d 62d 62d 62d 77b 77b 77b 77b 77b 89b 89b 89b 89b 92c, debug 16a, is_equal 4b, is_f_orthogonal 8d, k 3a, l 3a, m 3a, math_string 13b, passing 6b, subject_to 6c, and wspaces 13b.

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

25c ⟨f-orthogonal line 25c⟩≡ (12d) 25d▷

```
auto Focal_orth_line=[](const cycle2D & C6, const cycle2D & C, const ex & e) {
   ⟨Check either vector formalism is used 18e⟩
   C7 = C6.subject_to(lst{C6.is_linear()});
   u4 = C.center().op(0);
   v4 = C7.roots(u4, false).op(0).normal();
```

Uses center 5f, cycle2D 9a 9b 15c 15c 15d 15d 54b 55c 61b 62d 62d 62d 62d 62d 77b 77b 77b 77b 77b 89b 89b 89b 89b 92c, ex 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, is_linear 8e, normal 4b, op 4b, roots 9g, and subject_to 6c.

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

25d ⟨f-orthogonal line 25c⟩+≡ (12d) ◁25c

```
cout << wspaces << "For " << (is_vector? "" : "para")
   << "vectors 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; };

 Focal_orth_line(Cv6, Cv, ev);
 Focal_orth_line(Cp6, Cp, ep);
```

Uses focus 9f, is_zero 4b, matrix 11d 16b 16c, normal 4b, and wspaces 13b.

3.4.3. *Inversion from the f-orthogonality.* We express f-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.

25e ⟨f-inversion in cycle 25e⟩≡ (12d) 26a▷

```
auto Focal_inversion=[](const cycle2D & C, const cycle2D & C6, const cycle2D & real_line,
                 const ex & e, const ex & es) {
   ⟨Check either vector formalism is used 18e⟩
   C8 = real_line.cycle_similarity(C, es, diag_matrix(lst{1, sign1}),
                       diag_matrix(lst{1, jump_fnct(sign)}), diag_matrix(lst{1, sign1})).normalize(n*k);
   if (debug > 1)
      cout << "f-ghost cycleis : " math_string << C8 math_string << endl;
```

Uses cycle2D 9a 9b 15c 15c 15d 15d 54b 55c 61b 62d 62d 62d 62d 62d 77b 77b 77b 77b 77b 89b 89b 89b 89b 92c, cycle_similarity 7e, debug 16a, ex 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, jump_fnct 59d, k 3a, math_string 13b, and normalize 5e.

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

26a ⟨f-inversion in cycle 25e⟩+≡ (12d) ◁25e 26b▷
```
cout << "For " << (is_vector? "" : "para") << "vectors" << endl;
cout << wspaces << "f-ghost cycle has common roots with C: "
   << (C8.val(lst{C.roots().op(0), 0}).numer().normal().is_zero()
     && C8.val(lst{C.roots().op(1), 0}).numer().normal().is_zero()) << endl;
```

Uses cycle 3a 3a 3b 3b 3c 3d 5a 5a 5a 5a 5a 72c 72d 72d 73a 75d 75d 77a 77a 77a 77a 77a, is_zero 4b, normal 4b, op 4b, roots 9g, val 6a, and wspaces 13b.

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

26b ⟨f-inversion in cycle 25e⟩+≡ (12d) ◁26a 26c▷
```
cout << wspaces << "$\\chi(\\sigma)$-center of f-ghost cycle coincides "
"with $\\breve{\\sigma}$-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 center 5f, cycle 3a 3a 3b 3b 3c 3d 5a 5a 5a 5a 5a 72c 72d 72d 73a 75d 75d 77a 77a 77a 77a 77a, focus 9f, jump_fnct 59d, normal 4b, and wspaces 13b.

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

26c ⟨f-inversion in cycle 25e⟩+≡ (12d) ◁26b 26d▷
```
P1 = C8.moebius_map(is_vector? W : Wbar, e, diag_matrix(lst{1, -jump_fnct(sign)}))
.subs(signs_cube, subs_options::algebraic | subs_options::no_pattern).normal();
cout << wspaces << "f-inversion in C coincides with inversion in f-ghost cycle: "
<< C6.val(P1).normal().subs(signs_cube, subs_options::algebraic | subs_options::no_pattern).normal().is_zero()
<< endl; };
```

Uses cycle 3a 3a 3b 3b 3c 3d 5a 5a 5a 5a 5a 72c 72d 72d 73a 75d 75d 77a 77a 77a 77a 77a, is_zero 4b, jump_fnct 59d, moebius_map 8a 87b, normal 4b, subs 4b, val 6a, and wspaces 13b.

Finally, we do the check for both formalisms.

26d ⟨f-inversion in cycle 25e⟩+≡ (12d) ◁26c
```
Focal_inversion(Cv,Cv6,real_linev,ev,evs);
Focal_inversion(Cp,Cp6,real_linep,ep,eps);
```

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 [16, Defn. 5.2]. Thus we first construct a generic *cycle2d C10* passing through two points (u, v) and (u', v').

26e ⟨Distances from cycles 26e⟩≡ (12e) 26f▷
```
auto Distance1=[](const cycle2D & C, const ex & e, const ex & es) {
   ⟨Check either vector formalism is used 18e⟩
   cycle2D 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 > 0) cout << wspaces << "C10 is:    " << C10 << endl;
```

Uses cycle2D 9a 9b 15c 15c 15d 15d 54b 55c 61b 62d 62d 62d 62d 62d 77b 77b 77b 77b 77b 89b 89b 89b 89b 92c, debug 16a, ex 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, l 3a, m 3a, numeric 14a 57d, passing 6b, subject_to 6c, and wspaces 13b.

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 $D'_l = 0$.

26f ⟨Distances from cycles 26e⟩+≡ (12e) ◁26e 27a▷
```
ex D = 4*C10.radius_sq(es);
Len_c = D.subs(lsolve(lst{D.diff(l) == 0}, lst{l})).normal();
```

Uses ex 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, l 3a, normal 4b, radius_sq 6f, and subs 4b.

Now we check that *Len_c* is equal to [18, Lem. 5.2]

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

27a ⟨Distances from cycles 26e⟩+≡ (12e) ◁26f 27b▷

```
cout << "For " << (is_vector? "" : "para") << "vectors" << endl;
cout << wspaces << "Distance between (u,v) and (u\',v\') in elliptic and hyperbolic spaces is "
  << endl;

if (output_latex) {
 ex dist = (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));
 cout << "\\(\\displaystyle " << dist << "\\): " << (Len_c-dist).normal().is_zero() << endl;
} else
 cout << 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 ex 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, is_zero 4b, normal 4b, u 100a, v 100a, and wspaces 13b.

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.

27b ⟨Distances from cycles 26e⟩+≡ (12e) ◁27a 27c▷

```
check_conformality(Len_c, -1);
C11 = C10.subs(lsolve(lst{D.diff(l) ≡ 0}, lst{l}));
print_perpendicular(C11);
```

Uses check_conformality 28c, l 3a, print_perpendicular 30a, and subs 4b.

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.

27c ⟨Distances from cycles 26e⟩+≡ (12e) ◁27b 27d▷

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

Uses l 3a, math_string 13b, normal 4b, subs 4b, u 100a, and wspaces 13b.

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

27d ⟨Distances from cycles 26e⟩+≡ (12e) ◁27c 27e▷

```
check_conformality(Len_c, 0);
C11 = C10.subs(lst{sign ≡0, l ≡ (u+u1)*half});
print_perpendicular(C11); };
```

Uses check_conformality 28c, l 3a, print_perpendicular 30a, subs 4b, and u 100a.

Now we are checking this in both formalisms.

27e ⟨Distances from cycles 26e⟩+≡ (12e) ◁27d 28a▷

```
Distance1(Cv, ev, evs);
Distance1(Cp, ep, eps);
```

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.

28a ⟨Distances from cycles 26e⟩+≡ (12e) ◁27e 28b▷

```
auto Distance2=[](const cycle2D & C, const ex & e, const ex & es) {
    cycle2D 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 << wspaces << C10 << endl;
```

Uses `cycle2D` 9a 9b 15c 15c 15d 15d 54b 55c 61b 62d 62d 62d 62d 62d 77b 77b 77b 77b 77b 89b 89b 89b 89b 92c, `debug` 16a, `ex` 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, `l` 3a, `m` 3a, `numeric` 14a 57d, `passing` 6b, `subject_to` 6c, `v` 100a, and `wspaces` 13b.

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

28b ⟨Distances from cycles 26e⟩+≡ (12e) ◁28a

```
    ex D = 4*C10.radius_sq(es);
    return D.subs(lsolve(lst{D.diff(n) == 0}, lst{n})).normal(); };

ex Dv=Distance2(Cv, ev, evs);
    cout << "For vectors distance between (u,v) and (u\',v\') "
        << "(value at critical point): " << endl
        << wspaces math_string << Dv  math_string
        << endl << endl
        << wspaces << " for paravector is the same: "
        <<Dv.is_equal(Distance2(Cp, ep, eps)) << endl;
```

Uses `ex` 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, `is_equal` 4b, `math_string` 13b, `normal` 4b, `paravector` 63a 63c 103c 103c 103c 104d 104d 105a, `radius_sq` 6f, `subs` 4b, `u` 100a, `v` 100a, and `wspaces` 13b.

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 few minutes on my computer. The rest is calculated within twenty seconds.

28c ⟨Check conformal property 28c⟩≡ (16e) 28d▷

```
void check_conformality(const ex & Len_c, int si = 3) {
 ⟨Evaluate the fraction 29e⟩
```

Defines:
`check_conformality`, used in chunks 27, 30d, and 31e.
Uses `ex` 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c and `si` 14b.

Several times we fork for two cases: the first one if the check is done for all signs combinations simultaneously.

28d ⟨Check conformal property 28c⟩+≡ (16e) ◁28c 28e▷

```
if (si > 2)
 cout << wspaces << "This distance/length is conformal:" ;
```

Uses `si` 14b and `wspaces` 13b.

The second case is we output coresponding results for different metric signs.

28e ⟨Check conformal property 28c⟩+≡ (16e) ◁28d 28f▷

```
else
 cout << wspaces << "Conformity in a cycle space with metric:   E      P      H " << endl;
```

Uses `cycle` 3a 3a 3b 3b 3c 3d 5a 5a 5a 5a 5a 72c 72d 72d 73a 75d 75d 77a 77a 77a 77a 77a and `wspaces` 13b.

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). The first loop is for point space metric sign.

28f ⟨Check conformal property 28c⟩+≡ (16e) ◁28e 29a▷
```
do {
 if (si > 1)
  si1 = 2;
 else {
  cout << wspaces << "Point space is " << eph_case(si) << ": ";
  si1 = -1;
 }
```

Uses `si` 14b, `si1` 14b, and `wspaces` 13b.

The second loop is for cycle space metric sign.

29a ⟨Check conformal property 28c⟩+≡ (16e) ◁28f 29b▷
```
 do {
  if (si < 2)
```

Uses `si` 14b.

However the substitution of signs is not done for dummy loops.

29b ⟨Check conformal property 28c⟩+≡ (16e) ◁29a 29c▷
```
 Len_cD = Len_fD.subs(lst{sign ≡ numeric(si), sign1 ≡ numeric(si1)},
          subs_options::algebraic | subs_options::no_pattern).normal();
```

Uses `normal` 4b, `numeric` 14a 57d, `si` 14b, `si1` 14b, and `subs` 4b.

But even for dummy loops we make a check the conformity.

29c ⟨Check conformal property 28c⟩+≡ (16e) ◁29b 29d▷
```
   ⟨Find the limit 29f⟩
   ⟨Check independence 29g⟩
```

and then finalise all loops.

29d ⟨Check conformal property 28c⟩+≡ (16e) ◁29c
```
       si1++;
    } while (si1 < 2);
    cout << endl;
    si+=2;
 } while (si < 2);
 }
```

Uses `si` 14b and `si1` 14b.

To this end we consider the ratio of distances between (u, v) and $(u + tx, v + ty)$ and between their images gW and $gW1$ under the generic Möbius transform.

29e ⟨Evaluate the fraction 29e⟩≡ (28c)
```
 ex Len_cD= ((Len_c.subs(lst{u ≡ gW.op(0), v≡gW.op(1), u1 ≡ gW1.op(0),
                 v1≡gW1.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)).normal();
 ex Len_fD = Len_cD;
```

Uses `ex` 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, `normal` 4b, `op` 4b, `subs` 4b, `u` 100a, and `v` 100a.

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

29f ⟨Find the limit 29f⟩≡ (29c)
```
 if (Len_cD.has(t))
  Len_cD = Len_cD.series(t≡0,1).op(0).normal();
```

Uses `normal` 4b and `op` 4b.

The limit of this ratio for $t \to 0$ should be independent from (x, y) (see [18, Defn. 5.4]).

29g ⟨Check independence 29g⟩≡ (29c)

```
bool is_conformal = ¬(Len_cD.is_zero() ∨ Len_cD.has(t)
    ∨ Len_cD.has(x) ∨ Len_cD.has(y));
cout ≪ "  " ≪ is_conformal;
if (debug > 0 ∨ (¬is_conformal ∧ (si > 2))) {
   cout ≪ ". The factor is: " ≪ endl ≪ wspaces math_string ≪ Len_cD.normal() math_string ;

}
```

Uses `bool` 16a, `debug` 16a, `is_zero` 4b, `math_string` 13b, `normal` 4b, `si` 14b, and `wspaces` 13b.

3.5.3. *Calculation of Perpendiculars.* Lengths define corresponding perpendicular conditions in terms of shortest routes, see [18, Defn. 5.5].

30a ⟨Print perpendicular 30a⟩≡ (16e)

```
void print_perpendicular(const cycle2D & C) {
 cout ≪ wspaces ≪ "Perpendicular to ((u,v); (u\',v\')) is: "
  math_string ≪ (C.get_l(1)+sign∗C.get_k()∗v1).normal() math_string ≪ "; "
  math_string ≪ (C.get_l(0)-C.get_k()∗u1).normal() math_string ≪ endl ≪ endl;
}
```

Defines:
`print_perpendicular`, used in chunks 27 and 30d.
Uses `cycle2D` 9a 9b 15c 15c 15d 15d 54b 55c 61b 62d 62d 62d 62d 62d 77b 77b 77b 77b 77b 89b 89b 89b 89b 92c, `get_k` 3e, `get_l` 4a, `math_string` 13b, `normal` 4b, `u` 100a, `v` 100a, and `wspaces` 13b.

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 [18, Defn. 5.3].
Firstly we need some more imaginary units, to accommodate different types of centres (foci).

30b ⟨Declaration of variables 14a⟩+≡ (13c) ◁16c 31g▷

```
ex sign5=sign4;
```

Uses `ex` 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c.

Then we build a **cycle2D** *C11* which passes through (u', v') and has its centre at (u, v).

30c ⟨Lengths from centre 30c⟩≡ (12e) 30d▷

```
auto Length_checks=[](const cycle2D & C, const ex & e, const ex & es, const ex & e4) {
   ⟨Check either vector formalism is used 18e⟩
   sign5=sign4;
   C11 = C.subject_to(lst{C.passing(W1), C.is_normalized()});
   C11 = C11.subject_to(lst{C11.center().op(0) ≡ u, C11.center(e4).op(1)≡ v});
```

Uses `center` 5f, `cycle2D` 9a 9b 15c 15c 15d 15d 54b 55c 61b 62d 62d 62d 62d 62d 77b 77b 77b 77b 77b 89b 89b 89b 89b 92c, `ex` 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, `is_normalized` 8e, `op` 4b, `passing` 6b, `subject_to` 6c, `u` 100a, and `v` 100a.

Then the distance is radius the *C11*, see [18, Lem. 1]. We check conformity and calculate the perpendicular at the end.

30d ⟨Lengths from centre 30c⟩+≡ (12e) ◁30c

```
   Len_c = C11.radius_sq(es).normal();
   cout ≪ "For " ≪ (is_vector? "" : "para") ≪ "vectors" ≪ endl;
   cout ≪ wspaces ≪ "Length from *center* between (u,v) and "
   math_string ≪ "(u^\\prime,v^\\prime)" math_string ≪ ":" ≪ endl  ≪ wspaces
   math_string ≪ Len_c math_string ≪ endl ;
   check_conformality(Len_c);
   print_perpendicular(C11);
```

Uses `center` 5f, `check_conformality` 28c, `math_string` 13b, `normal` 4b, `print_perpendicular` 30a, `radius_sq` 6f, `u` 100a, `v` 100a, and `wspaces` 13b.

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)* ≡ *v* by hand-made value for the parameter *n*.

There are two suitable values of *n* which correspond upward and downward parabolas, which are expressed by plus or minus before the square root. After the value of length was found we master a simpler expression for it which utilises the focal length *p* of the parabola.

30e ⟨Lengths from focus 30e⟩≡ (12e) 30f▷

```
focal_length_check(sign5*(-(v1-v)+sqrt(sign5*pow((u1-u), 2)+pow((v1-v), 2) -sign5*sign*pow(v1, 2))), C, e, es);
```

Uses focal_length_check 31c, u 100a, and v 100a.

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

30f ⟨Lengths from focus 30e⟩+≡ (12e) ◁30e 31a▷

```
focal_length_check(sign5*(-(v1-v)-sqrt(sign5*pow(u1-u, 2)+pow((v1-v), 2) -sign5*sign*pow(v1, 2))), C, e, es);
```

Uses focal_length_check 31c, u 100a, and v 100a.

We need to verify separately the case of *sign5*=0, in this case *p* has a rational value.

31a ⟨Lengths from focus 30e⟩+≡ (12e) ◁30f 31b▷

```
cout << "Shall be 'false' for conformality below" << endl;
sign5=0;
focal_length_check((pow(u1-u,2)-sign*pow(v1,2))÷(v1-v)÷2, C, e, es); };
```

Uses focal_length_check 31c, u 100a, and v 100a.

Finally, we do the check for both formalisms.

31b ⟨Lengths from focus 30e⟩+≡ (12e) ◁31a

```
Length_checks(Cv,ev,evs,ev4);
Length_checks(Cp,ep,eps,ep4);
```

Again to avoid non-linearity of equation, we first construct a desired cycle.

31c ⟨Focal length checks 31c⟩≡ (16e) 31d▷

```
void focal_length_check(const ex & p, const cycle2D & C, const ex e, const ex es) {
    cout << "Length from *focus* check for " math_string << "p = " << p math_string << endl;
    cycle2D C11 = C.subject_to(lst{C.passing(W1), k≡1, l ≡ u, n ≡ p});
```

Defines:
focal_length_check, used in chunks 30 and 31a.
Uses cycle2D 9a 9b 15c 15c 15d 15d 54b 55c 61b 62d 62d 62d 62d 62d 77b 77b 77b 77b 77b 89b 89b 89b 89b 92c, ex 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, focus 9f, k 3a, l 3a, math_string 13b, passing 6b, subject_to 6c, and u 100a.

And now we verify that the length is equal to $(1-\sigma_1)p^2 - 2vp$, see [18, Lem. 2].

31d ⟨Focal length checks 31c⟩+≡ (16e) ◁31c 31e▷

```
    ex Len_c = C11.radius_sq(es).subs(pow(sign4,2)≡1,subs_options::algebraic | subs_options::no_pattern).normal();
   cout << wspaces << "Length between (u,v) and (u\', v\') is equal to "
    << (output_latex ? "\\((\\mathring(\\sigma)-\\breve{\\sigma})p^2-2vp\\): ":"(s4-s1)*p^2-2vp: ")
    << (Len_c - ((sign5-sign1)*pow(p, 2) - 2*v*p)).subs(signs_cube, subs_options::algebraic | subs_options::no_pattern)
     .expand().subs(pow(sign4,2)≡1,subs_options::algebraic | subs_options::no_pattern).normal().is_zero()
      << endl;
```

Uses ex 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, expand 4b, is_zero 4b, normal 4b, radius_sq 6f, subs 4b, u 100a, v 100a, and wspaces 13b.

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

31e ⟨Focal length checks 31c⟩+≡ (16e) ◁31d 31f▷

```
    cout << wspaces << "checks: C11 passes through (u\', v\'): " << C11.val(W1).normal().is_zero()
  << "; C11 focus is at (u, v): "
  << (C11.focus(diag_matrix(lst{-1,sign5}),true).subs(pow(sign4,2)≡1,subs_options::algebraic)-matrix(2,1,lst{u,v}))
                                                  .evalm().normal().is_zero_matrix() << endl;
  check_conformality(Len_c);
```

Uses check_conformality 28c, focus 9f, is_zero 4b, matrix 11d 16b 16c, normal 4b, subs 4b, u 100a, v 100a, val 6a, and wspaces 13b.

We finally verify that focal perpendiculars are multiples of the vector $(\sigma v'+p, u-u')$, see [18, E-it:focal-perpendicularity].

31f ⟨Focal length checks 31c⟩+≡ (16e) ◁31e

```
cout << wspaces << "Perpendicular to ((u,v); (u\',v\')) is "
    << (output_latex ? "\\((\\sigma v\'+p, u-u\')\\): " : "(s*v\'+p, u-u\'): ")
    << ((C11.get_l(1)+sign*C11.get_k()*v1-(sign*v1+p)).normal().is_zero()
       && (C11.get_l(0)-C11.get_k()*u1-(u-u1)).normal().is_zero())
    << endl << endl;
}
```

Uses get_k 3e, get_l 4a, is_zero 4b, normal 4b, u 100a, v 100a, and wspaces 13b.

3.6. Infinitesimal Cycles.

The final bit of our calculation is related with the infinitesimal radius cycles, see [18, § 6.1]. Some additional parameters.

31g ⟨Declaration of variables 14a⟩+≡ (13c) ◁30b 34e▷

```
possymbol vp("vp","v_p"); //the positive instance of symbol v
ex displ; //displacement of the focus
```

Uses ex 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, focus 9f, and v 100a.

3.6.1. *Basic properties of infinitesimal cycles.* @We define an infinitesimal cycle *C10* such that its squared radius (det) is an infinitesimal number ε^2 and focus is at (u,v). This defined by the cycle $(1, u_0, n, u_0^2+2nv_0-\mathring{\sigma}n^2)$ where n satisfies to the equation

$$(\mathring{\sigma}-\breve{\sigma})n^2-2v_0n+\varepsilon^2=0. \tag{3.1}$$

Only one root of the quadratic case produces a cycle with an infinitesimal focal length, and we consider it here:

32a ⟨Infinitesimal cycle 32a⟩≡ (12e) 32b▷

```
infinitesimal_calculations(n==(vp-sqrt(pow(vp,2)+pow(epsilon,2)*(sign4-sign1)))/(sign4-sign1),
                           Cv,ev,evs,ev4,Cp,ep,eps,ep4);
//infinitesimal_calculations(n==(vp-abs(pow(pow(vp,2)-pow(epsilon,2)*(sign4-sign1),half)))/(sign4-sign1),
// C,e,es,e4,is_vector);
```

Defines:
infinitesimal_calculations, used in chunk 32b.

The second expression for an infinitesimal cycle for the case $\mathring{\sigma}=\breve{\sigma}$ is given by the substitution $n=-\frac{\varepsilon^2}{2v}$, which the root of (3.1) in this case.

32b ⟨Infinitesimal cycle 32a⟩+≡ (12e) ◁32a

```
infinitesimal_calculations(lst{n==pow(epsilon,2)/2/vp, sign4==sign1},Cv,ev,evs,ev4,Cp,ep,eps,ep4);
```

Uses infinitesimal_calculations 32a 32c.

We organise the infinitesimal cycles check as a separate subroutine and start it from several local variables definition.

32c ⟨Infinitesimal cycle calculations 32c⟩≡ (16e) 32d▷

```
void infinitesimal_calculations(const ex & nval, const cycle2D C, const ex e, const ex es, const ex e4,
                                const cycle2D Cn, const ex en, const ex ens, const ex en4) {
    exmap smap;
    smap[v]=vp;
```

Defines:
infinitesimal_calculations, used in chunk 32b.
Uses cycle2D 9a 9b 15c 15c 15d 15d 54b 55c 61b 62d 62d 62d 62d 62d 77b 77b 77b 77b 77b 89b 89b 89b 89b 92c, ex 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, and v 100a.

32d ⟨Infinitesimal cycle calculations 32c⟩+≡ (16e) ◁32c 33a▷

```
cycle2D C10 = cycle2D(1, lst{u, n}, pow(u,2)-pow(n,2)*sign1-pow(epsilon,2), e).subs(nval),
  Cn10 = cycle2D(1, lst{u, n}, pow(u,2)-pow(n,2)*sign1-pow(epsilon,2), en).subs(nval);
cout << wspaces << "Inf cycle is: " math_string << C10 math_string << endl;
cout << wspaces << "For paravector is the same: " << C10.is_equal(Cn10,true,true) << endl;
cout << wspaces << "Square of radius of the infinitesimal cycle is: "
   math_string << C10.radius_sq(es).subs(signs_cube, subs_options::algebraic
                    | subs_options::no_pattern).normal() math_string << endl
  << wspaces << "For paravector is the same: " << C10.radius_sq(es).subs(signs_cube, subs_options::algebraic
                    | subs_options::no_pattern).normal()
   .is_equal(Cn10.radius_sq(es).subs(signs_cube, subs_options::algebraic
                    | subs_options::no_pattern).normal()) << endl;
```

Uses cycle 3a 3a 3b 3b 3c 3d 5a 5a 5a 5a 5a 72c 72d 72d 73a 75d 75d 77a 77a 77a 77a 77a, cycle2D 9a 9b 15c 15c 15d 15d 54b 55c 61b 62d 62d 62d 62d 62d 77b 77b 77b 77b 77b 89b 89b 89b 89b 92c, is_equal 4b, math_string 13b, normal 4b, paravector 63a 63c 103c 103c 103c 104d 104d 105a, radius_sq 6f, subs 4b, u 100a, and wspaces 13b.

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

33a ⟨Infinitesimal cycle calculations 32c⟩+≡ (16e) ◁32d 33b▷

```
cout << wspaces << "Focus of infinitesimal cycle is: " math_string
  << C10.focus(e4).subs(nval) math_string << endl
  << wspaces << "For paravector is the same: "
  << C10.focus(e4).subs(nval).is_equal(Cn10.focus(en4).subs(nval)) << endl
  << wspaces << "Focal length is: " math_string
  << C10.focal_length().series(epsilon==0,3).normal() math_string << endl
  << wspaces << "For paravector is the same: "
  << C10.focal_length().series(epsilon==0,3).normal().is_equal(
                          Cn10.focal_length().series(epsilon==0,3).normal())
  << endl;

cout << wspaces << "Infinitesimal cycle (vector) passing points" math_string
   << "(u+" << epsilon*x << ", vp+"
   << lsolve(C10.subs(sign==0).passing(lst{u+epsilon*x,vp+y}),y).series(epsilon==0,3).normal()
   << "), " math_string << endl;

cout << wspaces << "Infinitesimal cycle (paravector) passing points" math_string
   << "(u+" << epsilon*x << ", vp+"
   << lsolve(Cn10.subs(sign==0).passing(lst{u+epsilon*x,vp+y}),y).series(epsilon==0,3).normal()
   << "), " math_string << endl;
```

Uses cycle 3a 3a 3b 3b 3c 3d 5a 5a 5a 5a 5a 72c 72d 72d 73a 75d 75d 77a 77a 77a 77a 77a, focal_length 9f, focus 9f, is_equal 4b, math_string 13b, normal 4b, paravector 63a 63c 103c 103c 103c 104d 104d 105a, passing 6b, points 103a, subs 4b, u 100a, and wspaces 13b.

3.6.2. *Möbius transformations of infinitesimal cycles.* Now we check that transformation of an infinitesimal cycle is an infinitesimal cycle again…

33b ⟨Infinitesimal cycle calculations 32c⟩+≡ (16e) ◁33a 34a▷

```
cycle2D C11=C10.sl2_similarity(a, b, c, d, es),
  Cn11=Cn10.sl2_similarity(a, b, c, d, ens);
cout ≪ wspaces ≪ "Image under SL2(R) of infinitesimal cycle has radius squared: " ≪ endl
    math_string ≪ C11.radius_sq(es).subs(sl2_relation1,
                     subs_options::algebraic | subs_options::no_pattern).subs(signs_cube,
                        subs_options::algebraic | subs_options::no_pattern)
    .series(epsilon≡0,3).normal()
    math_string ≪ endl
     ≪ wspaces ≪ "For paravector is the same: "
     ≪ C11.radius_sq(es).subs(sl2_relation1,
                 subs_options::algebraic | subs_options::no_pattern).subs(signs_cube,
                        subs_options::algebraic | subs_options::no_pattern)
    .series(epsilon≡0,3).normal().is_equal(Cn11.radius_sq(ens).subs(sl2_relation1,
                     subs_options::algebraic | subs_options::no_pattern).subs(signs_cube,
                            subs_options::algebraic | subs_options::no_pattern)
                 .series(epsilon≡0,3).normal()) ≪ endl;
```

Uses `cycle` 3a 3a 3b 3b 3c 3d 5a 5a 5a 5a 5a 72c 72d 72d 73a 75d 75d 77a 77a 77a 77a 77a, `cycle2D` 9a 9b 15c 15c 15d 15d 54b 55c 61b 62d 62d 62d 62d 62d 77b 77b 77b 77b 77b 89b 89b 89b 89b 92c, `is_equal` 4b, `math_string` 13b, `normal` 4b, `paravector` 63a 63c 103c 103c 103c 104d 104d 105a, `radius_sq` 6f, `sl2_similarity` 7b 10c 61d 62a, `subs` 4b, and `wspaces` 13b.

…cycle similarity is under the test…

34a ⟨Infinitesimal cycle calculations 32c⟩+≡ (16e) ◁33b 34b▷

```
cout ≪ wspaces ≪ "Image under cycle similarity of infinitesimal cycle has radius squared: "
   ≪ endl
   math_string ≪ C10.cycle_similarity(C, es).radius_sq(es).subs(signs_cube, subs_options::algebraic
                 | subs_options::no_pattern).series(epsilon≡0,3).normal() math_string ≪ endl
   ≪ wspaces ≪ "For paravector is the same: "
   ≪ C10.cycle_similarity(C, es).radius_sq(es).subs(signs_cube, subs_options::algebraic
                 | subs_options::no_pattern).series(epsilon≡0,3).normal()
   .is_equal(Cn10.cycle_similarity(Cn, es).radius_sq(ens).subs(signs_cube, subs_options::algebraic
                 | subs_options::no_pattern).series(epsilon≡0,3).normal())
   ≪ endl;
```

Uses `cycle` 3a 3a 3b 3b 3c 3d 5a 5a 5a 5a 5a 72c 72d 72d 73a 75d 75d 77a 77a 77a 77a 77a, `cycle_similarity` 7e, `is_equal` 4b, `math_string` 13b, `normal` 4b, `paravector` 63a 63c 103c 103c 103c 104d 104d 105a, `radius_sq` 6f, `subs` 4b, and `wspaces` 13b.

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

34b ⟨Infinitesimal cycle calculations 32c⟩+≡ (16e) ◁34a 34c▷

```
ex displ = (C11.focus(e4, true).subs(nval) - gW.subs(smap, subs_options::no_pattern)).evalm();
cout ≪ wspaces ≪ "Focus of the transormed cycle is from transformation of focus by: "
   math_string ≪ displ.subs(sl2_relation, subs_options::algebraic
        | subs_options::no_pattern).subs(lst{sign≡0,a≡(1+b*c)÷d}).series(epsilon≡0,2).normal()
   math_string ≪ endl;
```

Uses `cycle` 3a 3a 3b 3b 3c 3d 5a 5a 5a 5a 5a 72c 72d 72d 73a 75d 75d 77a 77a 77a 77a 77a, `ex` 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, `focus` 9f, `math_string` 13b, `normal` 4b, `subs` 4b, and `wspaces` 13b.

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

34c ⟨Infinitesimal cycle calculations 32c⟩+≡ (16e) ◁34b 34d▷

```
cout ≪ wspaces ≪ "Orthogonality (leading term) to infinitesimal cycle is:" ≪ endl ≪ wspaces
          math_string ≪ ex(C.is_orthogonal(C10, es)).series(epsilon≡0,1).normal() math_string ≪ endl;
```

Uses `cycle` 3a 3a 3b 3b 3c 3d 5a 5a 5a 5a 5a 72c 72d 72d 73a 75d 75d 77a 77a 77a 77a 77a, `ex` 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, `is_orthogonal` 8c, `math_string` 13b, `normal` 4b, and `wspaces` 13b.

And the both expressions for the f-orthogonality (see § 3.4) conditions with the infinitesimal radius cycle. The second relation verifies the Lem. 6.4 from [18].

34d ⟨Infinitesimal cycle calculations 32c⟩+≡ (16e) ◁34c 35b▷

```
cout << wspaces << "f-orthogonality of other cycle to infinitesimal:" << endl << wspaces
math_string << C.is_f_orthogonal(C10, es).series(epsilon≡0,1).normal() math_string << endl
    << "f-orthogonality of infinitesimal cycle to other:" << endl << wspaces
math_string << C10.is_f_orthogonal(C, es).series(epsilon≡0,3).normal() math_string << endl;
```

Uses cycle 3a 3a 3b 3b 3c 3d 5a 5a 5a 5a 5a 72c 72d 72d 73a 75d 75d 77a 77a 77a 77a 77a, is_f_orthogonal 8d, math_string 13b, normal 4b, and wspaces 13b.

3.6.4. *Cayley transform of infinitesimal cycles.* Here is two matrices which defines the Cayley transform and its inverses:

34e ⟨Declaration of variables 14a⟩+≡ (13c) ◁31g

```
const matrix TCv(2,2, lst{dirac_ONE(), -ev.subs(mu2≡1), sign1*ev.subs(mu2≡1), dirac_ONE()}),
  TCp(2,2, lst{dirac_ONE(), -ep.subs(mu1≡0), sign1*ep.subs(mu1≡0), dirac_ONE()});
// the inverse is TCI(2,2, lst{dirac_ONE(), e.subs(mu==1), -sign1*e.subs(mu==1), dirac_ONE()});
```

Uses matrix 11d 16b 16c and subs 4b.

We conclude with calculations of the parabolic Cayley transform [18, § 8.3] on infinitesimal radius cycles. The parabolic Cayley transform on cycles is defined by the following transformation.

35a ⟨Parabolic Cayley transform of cycles 35a⟩≡ (16e)

```
cycle2D cayley_parab(const cycle2D & C, const ex & sign = -1)
{
   return cycle2D(C.get_k()-2*sign*C.get_l(1), C.get_l(), C.get_m()-2*C.get_l(1), C.get_unit());
}
```

Uses cycle2D 9a 9b 15c 15c 15d 15d 54b 55c 61b 62d 62d 62d 62d 62d 77b 77b 77b 77b 77b 89b 89b 89b 89b 92c, ex 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, get_k 3e, get_l 4a, get_m 4a, and get_unit 4a.

The image of an infinitesimal cycle is another infinitesimal radius cycle…

35b ⟨Infinitesimal cycle calculations 32c⟩+≡ (16e) ◁34d 35c▷

```
C11 = cayley_parab(C10, sign1);
cout << wspaces << "Det of Cayley-transformed infinitesimal cycle: "
   math_string << C11.radius_sq(es).subs(lst{sign ≡ 0},
                    subs_options::algebraic | subs_options::no_pattern).series(epsilon≡0,3).normal()
   math_string << endl;
```

Uses cycle 3a 3a 3b 3b 3c 3d 5a 5a 5a 5a 5a 72c 72d 72d 73a 75d 75d 77a 77a 77a 77a 77a, math_string 13b, normal 4b, radius_sq 6f, subs 4b, and wspaces 13b.

. . . with its focus mapped by the Cayley transform.

35c ⟨Infinitesimal cycle calculations 32c⟩+≡ (16e) ◁35b 35d▷

```
displ = (C11.focus(e4, true).subs(nval)
        - clifford_moebius_map(TCv, matrix(2,1,lst{u,vp}), e)).evalm().normal();
    ex displn = (C11.focus(e4, true).subs(nval)
        - clifford_moebius_map(TCp, matrix(2,1,lst{u,vp}), en)).evalm().normal();
  cout ≪ wspaces ≪ "Focus of the Cayley-transformed infinitesimal cycle displaced by: " math_string;
 try{
    cout ≪ displ.subs(lst{sign ≡ 0},
             subs_options::algebraic | subs_options::no_pattern).series(epsilon≡0, 2).normal();
 } catch (exception &p) {
    cout ≪ "(" ≪ displ.op(0).subs(lst{sign ≡ 0},
             subs_options::algebraic | subs_options::no_pattern).series(epsilon≡0, 2).normal()
       ≪ ", " ≪ displ.op(1).subs(lst{sign ≡ 0},
             subs_options::algebraic | subs_options::no_pattern).series(epsilon≡0, 2).normal()
       ≪ ")";
 }
 cout math_string ≪ endl
 ≪ wspaces ≪ "For paravector is the same: " ≪ displ.is_equal(displn) ≪ endl;
```

Uses `catch` 37a 37b, `cycle` 3a 3a 3b 3b 3c 3d 5a 5a 5a 5a 5a 72c 72d 72d 73a 75d 75d 77a 77a 77a 77a 77a, `ex` 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, `focus` 9f, `is_equal` 4b, `math_string` 13b, `matrix` 11d 16b 16c, `normal` 4b, `op` 4b, `paravector` 63a 63c 103c 103c 103c 104d 104d 105a, `subs` 4b, `u` 100a, and `wspaces` 13b.

f-orthogonality of

35d ⟨Infinitesimal cycle calculations 32c⟩+≡ (16e) ◁35c

```
 cout ≪ wspaces ≪ "f-orthogonality of Cayley transforms of infinitesimal cycle to other:" ≪ endl ≪ wspaces
    math_string ≪ C11.is_f_orthogonal(cayley_parab(C,sign1), es).series(epsilon≡0,3).normal()
    math_string ≪ endl ≪ endl;
 }
```

Uses `cycle` 3a 3a 3b 3b 3c 3d 5a 5a 5a 5a 5a 72c 72d 72d 73a 75d 75d 77a 77a 77a 77a 77a, `is_f_orthogonal` 8d, `math_string` 13b, `normal` 4b, and `wspaces` 13b.

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 [18], see also Fig. 3. 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.

If GiNaC version allows, we produce all pictures twice: in vector and paravector formalism.

36 ⟨Draw Asymptote pictures 36⟩≡ (13f) 37a▷

```
#if GINAC_VERSION_ATLEAST(1,7,1)
for (int is_vector=0; is_vector<2;++is_vector) {
#else
for (int is_vector=1; is_vector<2;++is_vector) {
#endif
   cycle2D C, C1, Z, Z1, real_line, Zinf;
   varidx mu;
   ex e, es;
   ofstream asymptote;
   relational mu_subs;
   if (is_vector==1) {
      C=Cv; C1=Cv1; Z=Zv; Z1=Zv1;
      real_line=real_linev; Zinf=Zvinf;
      e=ev; es=evs;
      asymptote=ofstream("parab-ortho1-v.asy");
      mu=mu2;
      mu_subs=(mu==1);
   } else {
      C=Cp; C1=Cp1; Z=Zp; Z1=Zp1;
      real_line=real_linep;  Zinf=Zpinf;
      e=ep; es=eps;
      asymptote=ofstream("parab-ortho1-p.asy");
      mu=mu1;
      mu_subs=(mu==0);
   }

   P = C.moebius_map(is_vector==1? W : Wbar, e, -M1);
   P1 = C.moebius_map(is_vector==1? lst{u3+u, v3+v} : lst{u3+u, -v3-v}, e, -M1);

   C2 = C1.subject_to(lst{C1.passing(W), C1.is_orthogonal(C, es)});
   C4 = C1.subject_to(lst{C1.passing(W), C1.passing(P), C1.is_linear()});
   u3 = C.center().op(0);
   v3 = C4.roots(u3, false).op(0).normal();
   C5 = cycle2D(lst{u3, -v3*jump_fnct(sign)}, e, C.radius_sq(e, M1)).subs(signs_cube,
            subs_options::algebraic | subs_options::no_pattern);
   C6 = C1.subject_to(lst{C1.passing(W), C.is_f_orthogonal(C1, eps)});
   C7 = C6.subject_to(lst{C6.is_linear()});
   C8 = real_line.cycle_similarity(C, es, diag_matrix(lst{1, sign1}), diag_matrix(lst{1, jump_fnct(sign)}),
                           diag_matrix(lst{1, sign1})).normalize(n*k);


   asymptote << setprecision(2);
   for (si = -1; si < 2; si++) {
      for (si1 = -1; si1 < 2; si1++) {
         sign_val = lst{sign == si, sign1 == si1};
```

Uses center 5f, cycle2D 9a 9b 15c 15c 15d 15d 54b 55c 61b 62d 62d 62d 62d 62d 77b 77b 77b 77b 77b 89b 89b 89b 89b 92c, cycle_similarity 7e, ex 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, GINAC_VERSION_ATLEAST 59a 59a, is_f_orthogonal 8d, is_linear 8e, is_orthogonal 8c, jump_fnct 59d, k 3a, moebius_map 8a 87b, normal 4b, normalize 5e, op 4b, passing 6b, radius_sq 6f, roots 9g, si 14b, si1 14b, subject_to 6c, subs 4b, u 100a, v 100a, and varidx 14a 15a 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 D.

37a ⟨Draw Asymptote pictures 36⟩+≡ (13f) ◁36 37b▷
```
try {
    {⟨Drawing first orthogonality 50a⟩}
    {⟨Drawing focal orthogonality 51d⟩}
} catch (exception &p) {
    cerr << "*****       Got a problem with drawing " << p.what() << endl;
}
}
}
```

Defines:
`catch`, used in chunks 13e, 35c, 66b, 68b, 69a, 79a, and 108a.

We finish the code with generation of some additional pictures for the paper [18].

37b ⟨Draw Asymptote pictures 36⟩+≡ (13f) ◁37a
```
try {
    ⟨Extra pictures from Asymptote 52c⟩
} catch (exception &p) {
    cerr << "*****       Got a problem with extra drawing " << p.what() << endl;
}
asymptote.close();
}
```

Defines:
`catch`, used in chunks 13e, 35c, 66b, 68b, 69a, 79a, and 108a.

References

[1] Arpad and G. Kovacs. *UNetbootin—create bootable Live USB drives for Linux*, 2011. URL: http://unetbootin.sourceforge.net/. ↑40

[2] C. Bauer, A. Frink, R. Kreckel, and J. Vollinga. *GiNaC is Not a CAS*, 2001. URL: http://www.ginac.de/. ↑2, 39, 40

[3] F. Bellard. *QEMU—a generic and open source machine emulator and virtualizer*, 2011. URL: http://qemu.org/. ↑40

[4] J. Brandmeyer. *PyGiNaC—a Python interface to the C++ symbolic math library GiNaC*, 2004. URL: http://sourceforge.net/projects/pyginac/. ↑40

[5] J. Cnops. *An introduction to Dirac operators on manifolds*. Progress in Mathematical Physics, vol. 24. Birkhäuser Boston Inc., Boston, MA, 2002. ↑2, 84

[6] R. Delanghe, F. Sommen, and V. Souček. *Clifford algebra and spinor-valued functions. A function theory for the Dirac operator*. Mathematics and its Applications, vol. 53. Kluwer Academic Publishers Group, Dordrecht, 1992. Related REDUCE software by F. Brackx and D. Constales, With 1 IBM-PC floppy disk (3.5 inch). ↑2

[7] J. P. Fillmore and A. Springer. Möbius groups over general fields using Clifford algebras associated with spheres. *Internat. J. Theoret. Phys.*, **29** (3):225–246, 1990. ↑2

[8] GNU. *General Public License (GPL)*. Free Software Foundation, Inc., Boston, USA, version 3, 2007. URL: http://www.gnu.org/licenses/gpl.html. ↑2, 39, 110

[9] K. Gürlebeck and R. Schmiedel (eds.) *Proceedings of "the international conference on the applications of computer science and mathematics in architecture and civil engineering (ikm)"*, 2006. URL: http://euklid.bauing.uni-weimar.de/index.php?lang=en&what=papers. ↑2

[10] K. Gürlebeck, K. Habetha, and W. Sprößig. *Funktionentheorie in der Ebene und im Raum*. Grundstudium Mathematik. [Basic Study of Mathematics]. Birkhäuser Verlag, Basel, 2006. With 1 CD-ROM (Windows and UNIX). ↑2

[11] A. Hammerlindl, J. Bowman, and T. Prince. *Asymptote—powerful descriptive vector graphics language for technical drawing, inspired by MetaPost*, 2004. URL: http://asymptote.sourceforge.net/. ↑2, 11, 39

[12] J. D. Hobby. *MetaPost: A MetaFont like system with postscript output*. URL: http://www.tug.org/metapost.html. ↑2

[13] A. A. Kirillov. *A tale of two fractals*. Springer, New York, 2013. Draft: http://www.math.upenn.edu/~kirillov/MATH480-F07/tf.pdf. ↑2

[14] V. V. Kisil. An example of Clifford algebras calculations with GiNaC. *Adv. Appl. Clifford Algebr.*, **15** (2):239–269, 2005. E-print: arXiv:cs.MS/0410044, On-line. Zbl1099.65521. ↑1, 2, 3

[15] V. V. Kisil. Erlangen program at large–0: Starting with the group $\mathrm{SL}_2(\mathbf{R})$. *Notices Amer. Math. Soc.*, **54** (11):1458–1465, 2007. E-print: arXiv:math/0607387, On-line. Zbl1137.22006. ↑9

[16] V. V. Kisil. Schwerdtfeger–Fillmore-Springer-Cnops construction implemented in GiNaC. *Adv. Appl. Clifford Algebr.*, **17** (1):59–70, 2007. On-line. Updated full text, source files, and live ISO image: E-print: arXiv:cs.MS/0512073. Project page: http://moebinv.sourceforge.net/. Zbl05134765. ↑2, 11, 23, 26, 39, 40, 55

[17] V. V. Kisil. *Erlangen program at large*, 2010. On-line lecture notes: http://www.maths.leeds.ac.uk/~kisilv/courses/sl2_pgcourse.html. ↑2, 9

[18] V. V. Kisil. Erlangen program at large–1: Geometry of invariants. *SIGMA, Symmetry Integrability Geom. Methods Appl.*, **6** (076):45, 2010. E-print: arXiv:math.CV/0512416. MR2011i:30044. Zbl1218.30136. ↑1, 2, 3, 6, 7, 8, 9, 12, 13, 16, 18, 19, 22, 23, 24, 27, 29, 30, 31, 34, 35, 36, 37, 79, 82, 87

[19] V. V. Kisil. *Geometry of Möbius transformations: Elliptic, parabolic and hyperbolic actions of* $\mathrm{SL}_2(\mathbf{R})$. Imperial College Press, London, 2012. Includes a live DVD. Zbl1254.30001. ↑2, 9

[20] V. V. Kisil. Ensembles of cycles programmed in GiNaC, 2014. E-print: arXiv:1512.02960. Project page: http://moebinv.sourceforge.net/. ↑2, 39, 59

[21] V. V. Kisil and D. Seidel. *Python wrapper for cycle library based on pyGiNaC*, 2006. URL: http://maths.leeds.ac.uk/~kisilv/pycycle.html. ↑2

[22] *VirtualBox—powerful x86 and AMD64/Intel64 virtualization product*. Oracle, 2011. URL: http://www.virtualbox.org. ↑40

[23] *Open virtual machine—the open source implementation of VMware Tools*. OVMTP, 2011. URL: http://open-vm-tools.sourceforge.net/. ↑40

[24] F. Pérez and B. E. Granger. IPython: a system for interactive scientific computing. *Computing in Science and Engineering*, **9** (3):21–29, May 2007. URL: http://ipython.org. ↑41

[25] I. R. Porteous. *Clifford algebras and the classical groups*. Cambridge Studies in Advanced Mathematics, vol. 50. Cambridge University Press, Cambridge, 1995. ↑2

[26] N. Ramsey. Literate programming simplified. *IEEE Software*, **11** (5):97–105, 1994. Noweb — A Simple, Extensible Tool for Literate Programming. URL: http://www.eecs.harvard.edu/~nr/noweb/. ↑2

[27] H. Schwerdtfeger. *Geometry of complex numbers: Circle geometry, Moebius transformation, non-Euclidean geometry*. Dover Books on Advanced Mathematics. Dover Publications Inc., New York, 1979. A corrected reprinting of the 1962 edition. ↑2

[28] O. Skavhaug and O. Certik. *swiGiNaC—a Python interface to GiNaC, built with SWIG*, 2010. URL: http://swiginac.berlios.de/. ↑40

[29] *Debian—the universal operating system*. Software in the Public Interest, Inc., 1997. URL: http://www.debian.org/. ↑39

[30] I. M. Yaglom. *A simple non-Euclidean geometry and its physical basis*. Heidelberg Science Library. Springer-Verlag, New York, 1979. Translated from the Russian by Abe Shenitzer, with the editorial assistance of Basil Gordon. ↑23

Appendix A. How to Use the Software

This is information about Open Source Software project Moebinv [20], see its Webpage[2] for updates.

The enclosed DVD (ISO image) with software is derived from several open-source projects, notably Debian GNU–Linux [29], GiNaC library of symbolic calculations [2], Asymptote [11] and many others. Thus, our work is distributed under the *GNU General Public License (GPL) 3.0* [8].

You can download an ISO image of a Live GNU–Linux DVD with our CAS from several locations. The initial (now outdated) version was posted through the Data Conservancy Project arXiv.org associated to paper [16]. A newer version of ISO is now included as an auxiliary file to the same paper, see the subdirectory:

http://arxiv.org/src/cs/0512073v11/anc

Also, an updated versions (v2.5) of the ISO image is uploaded to Google Drive:

https://drive.google.com/file/d/0BzfWNH9hAT3VM3BLYUlLU012bFU
https://drive.google.com/file/d/0BzfWNH9hAT3VM2luWGo3d3VZSms

In this Appendix, we only briefly outline how to start using the enclosed DVD or ISO image. As soon as the DVD is running or the ISO image is mounted as a virtual file system, further help may be obtained on the computer screen. We also describe how to run most of the software on the disk on computers without a DVD drive at the end of Sections A.1, A.2.1 and A.2.2.

A.1. Viewing Colour Graphics.

The easiest part is to view colour illustrations on your computer. There are not many hardware and software demands for this task—your computer should have a DVD drive and be able to render HTML pages. The last task can be done by any web browser. If these requirements are satisfied, perform the following steps:

1. Insert the DVD disk into the drive of your computer.
2. Mount the disk, if required by your OS.
3. Open the contents of the DVD in a file browser.
4. Open the file `index.html` from the top-level folder of the DVD in a web browser, which may be done simply by clicking on its icon.
5. Click in the browser on the link View book illustrations.

If your computer does not have a DVD drive (e.g. is a netbook), but you can gain brief access to a computer with a drive, then you can copy the top-level folder `doc` from the enclosed DVD to a portable medium, say a memory stick. Illustrations (and other documentation) can be accessed by opening the `index.html` file from this folder.

In a similar way, the reader can access ISO images of bootable disks, software sources and other supplementary information described below.

A.2. Installation of CAS.

There are three major possibilities of using the enclosed CAS:

A. To boot your computer from the DVD itself.
B. To run it in a Linux emulator.
C. *Advanced*: recompile it from the enclosed sources for your platform.

Method A is straightforward and can bring some performance enhancement. However, it requires hardware compatibility; in particular, you must have the so-called i386 architecture. Method B will run on a much wider set of hardware and you can use CAS from the comfort of your standard desktop. However, this may require an additional third-party programme to be installed.

A.2.1. *Booting from the DVD Disk.*

WARNING: it is a general principle, that running a software within an emulator is more secure than to boot your computer in another OS. Thus we recommend using the method described in Section A.2.2.

It is difficult to give an exact list of hardware requirements for DVD booting, but your computer must be based on the i386 architecture. If you are ready to have a try, follow these steps:

1. Insert the DVD disk into the drive of your computer.
2. Switch off or reboot the computer.
3. Depending on your configuration, the computer may itself attempt to boot from the DVD instead of its hard drive. In this case you can proceed to step 5.
4. If booting from the DVD does not happen, then you need to reboot again and bring up the "boot menu", which allows you to chose the boot device manually. This menu is usually prompted by a "magic key" pressed just after the computer is powered on—see your computer documentation. In the boot menu, chose the CD/DVD drive.
5. You will be presented with the screen shown on the left in Fig. 1. Simply press Enter to chose the "Live (486)" or "Live (686-pae)" (for more advanced processors) to boot. To run 686-pae kernel in an emulator, e.g. VirtualBox, you may need to allow "PAE option" in settings.
6. If the DVD booted well on your computer you will be presented with the GUI screen shown on the right in Fig. 1. Congratulations, you can proceed to Section A.3.

[2] http://moebinv.sourceforge.net/

If the DVD boots but the graphic X server did not start for any reason and you have the text command prompt only, you can still use most of the CAS. This is described in the last paragraph of Section A.3.

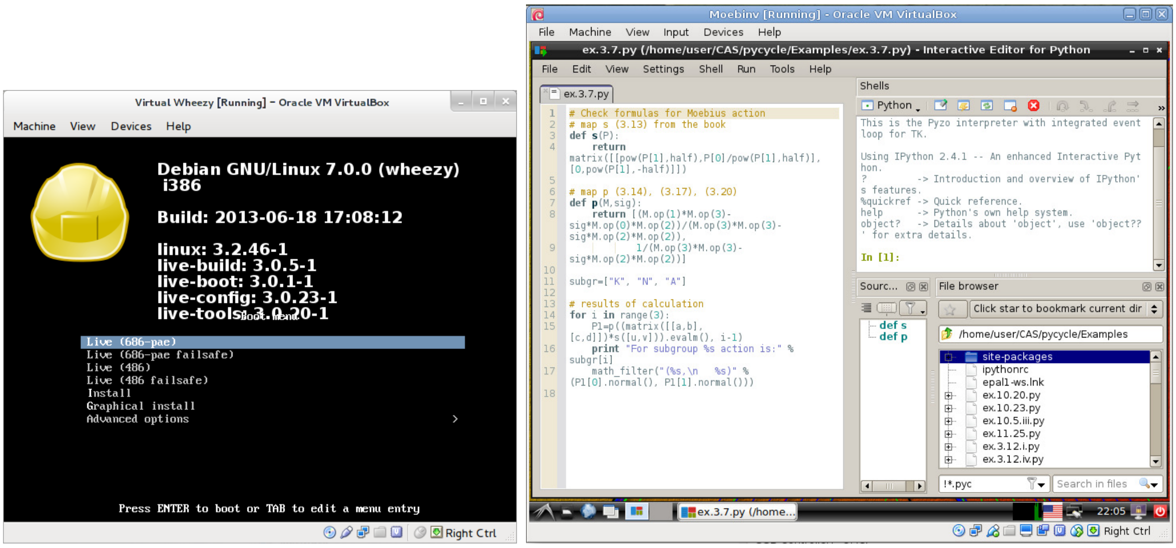

FIGURE 1. Initial screens of software start up. First, DVD boot menu; second, IDE screen after the booting.

If your computer does not have a DVD drive you may still boot the CAS on your computer from a spare USB stick of at least 1Gb capacity. For this, use `UNetbootin` [1] or a similar tool to put an ISO image of a boot disk on the memory stick. The ISO image(s) is located at the top-level folder `iso-images` of the DVD and the file `README` in this folder describes them. You can access this folder as described in Section A.1.

A.2.2. *Running a Linux Emulator.* You can also use the enclosed CAS on a wide range of hardware running various operating systems, e.g. Linux, Windows, Mac OS, etc. To this end you need to install a so-called *virtual machine*, which can emulate i386 architecture. I would recommend VirtualBox [22]—a free, open-source program which works well on many existing platforms. There are many alternatives (including open-source), for example: Qemu [3], Open Virtual Machine [23] and some others.

Here, we outline the procedure for VirtualBox—for other emulators you may need to make some adjustments. To use VirtualBox, follow these steps:

1. Insert the DVD disk in your computer.
2. Open the `index.html` file from the top directory of the DVD disk and follow the link "Installing VirtualBox". This is a detailed guide with all screenshots. Below we list only the principal steps from this guide.
3. Go to the web site of VirtualBox [22] and proceed to the download page for your platform.
4. Install VirtualBox on your computer and launch it.
5. Create a new virtual machine. Use either the entire DVD or the enclosed ISO images for the virtual DVD drive. If you are using the ISO images, you may wish to copy them first to your hard drive for better performance and silence. See the file `README` in the top-level folder `iso-images` for a description of the image(s).
6. Since a computer emulation is rather resource-demanding, it is better to close all other applications on slower computers (e.g. with a RAM less than 1Gb).
7. Start the newly-created machine. You will need to proceed through steps 5–6 from the previous subsections, as if the DVD is booting on your real computer. As soon as the machine presents the GUI, shown on the right in Fig. 1, you are ready to use the software.

If you succeeded in this you may proceed to Section A.3. Some tips to improve your experience with emulations are described in the detailed electronic manual.

A.2.3. *Recompiling the CAS on Your OS.* The core of our software is a `C++` library which is based on GiNaC [2]—see its web page for up-to-date information. The latter can be compiled and installed on both Linux and Windows. Subsequently, our library can also be compiled on these computers from the provided sources. Then, the library can be used in your `C++` programmes. See the top-level folder `src` on the DVD and the documentation therein. Also, the library source code (files `cycle.h` and `cycle.cpp`) is produced in the current directory if you pass the TEX file of the paper [16] through LATEX.

Our interactive tool is based on pyGiNaC [4]—a `Python` binding for GiNaC. This may work on many flavours of Linux as well. Please note that, in order to use pyGiNaC with the recent GiNaC, you need to apply my patches to the official version. The DVD contains the whole pyGiNaC source tree which is already patched and is ready to use.

There is also a possibility to use our library interactively with `swiGiNaC` [28], which is another `Python` binding for GiNaC and is included in many Linux distributions. The complete sources for binding our library to `swiGiNaC` are in the corresponding folder of the enclosed DVD. However, `swiGiNaC` does not implement full functionality of our library.

A.3. **Using the CAS and Computer Exercises.** Once you have booted to the GUI with the open CAS window as described in Subsections A.2.1 or A.2.2, a window with `Pyzo` (an integrated development environment—IDE)) shall start. The left frame is an editor for your code, some exercises from the book will appear there. Top right frame is a `IPython` shell, where your code will be executed. Bottom left frame presents the files tree.

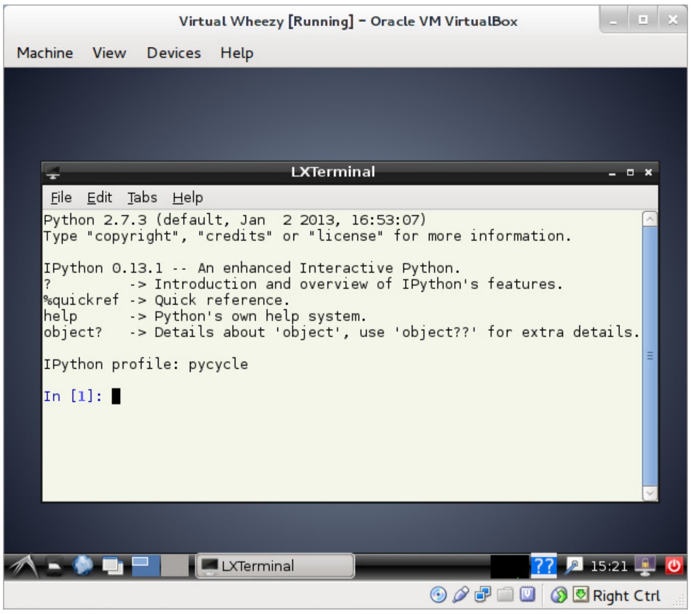

FIGURE 2. IPython shell.

`Pyzo` has a modern graphical user interface (GUI) and a detailed help system, thus we do not need to describe its work here. On the other hand, if a user wish to work with `IPython` shell alone (see Fig. 2), he may start the shall from

Main Menu→Accessories→CAS moebinv (ipython).

The presentation below will be given in terms of `IPython` shell, an interactions with `Pyzo` is even more intuitive.

Initially, you may need to configure your keyboard (if it is not a US layout). To install, for example, a Portuguese keyboard, you may type the following command at the `IPython` prompt (e.g. the top right frame of `Pyzo`):

```
In [2]: !change-xkbd pt
```

The keyboard will be switched and the corresponding national flag displayed at the bottom-left corner of the window. For another keyboard you need to use the international two-letter country code instead of `pt` in the above command. The first exclamation mark tells that the interpreter needs to pass this command to the shell.

A.3.1. *Warming Up.* The first few lines at the top of the CAS windows suggest several commands to receive a quick introduction or some help on the `IPython` interpreter [24]. Our CAS was loaded with many predefined objects—see Section A.5. Let us see what `C` is, for example:

```
In [3]: print C
------> print(C)
[cycle2D object]

In [4]: print C.string()
------> print(C.string())
(k, [L,n],m)
```

Thus, `C` is a two-dimensional cycle defined with the quadruple (k, l, n, m). Its determinant is:

```
In [5]: print C.hdet()
------> print(C.hdet())
k*m-L**2+si*n**2
```

Here, `si` stands for σ—the signature of the point space metric. Thus, the answer reads $km - l^2 + \sigma n^2$—the determinant of the SFSCc matrix of `C`. Note, that terms of the expression can appear in a different order: GiNaC does not have a predefined sorting preference in output.

As an exercise, the reader may now follow the proof of Theorem 4.13, remembering that the point `P` and cycle `C` are already defined. In fact, all statements and exercises marked by the symbol on the margins are already present on the DVD. For example, to access the proof of Theorem 4.13, type the following at the prompt:

```
In [6]: %ed ex.4.13.py
```

Here, the *special* `%ed` instructs the external editor `jed` to visit the file `ex.4.13.py`. This file is a `Python` script containing the same lines as the proof of Theorem 4.13 in the book. The editor `jed` may be manipulated from its menu and has command keystrokes compatible with GNU Emacs. For example, to exit the editor, press `Ctrl-X Ctrl-C`. After that, the interactive shell executes the visited file and outputs:

```
In [6]: %ed ex.4.13.py
Editing... done. Executing edited code...
Conjugated cycle passes the Moebius image of P: True
```

Thus, our statement is proven.

For any other CAS-assisted statement or exercise you can also visit the corresponding solution using its number next to the symbol 🖱 in the margin. For example, for Exercise 6.22, open file `ex.6.22.py`. However, the next mouse sign marks the item 6.24.i, thus you need to visit file `ex.6.24.i.py` in this case. These files are located on a read-only file system, so to modify them you need to save them first with a new name (`Ctrl-X Ctrl-W`), exit the editor, and then use `%ed` special to edit the freshly-saved file.

A.3.2. *Drawing Cycles.* You can visualise cycles instantly. First, we open an `Asymptote` instance and define a picture size:

```
In [7]: A=asy()
Asymptote session is open.  Available methods are:
    help(), size(int), draw(str), fill(str), clip(str), ...

In [8]: A.size(100)
```

Then, we define a cycle with centre $(0,1)$ and σ-radius 2:

```
In [9]: Cn=cycle2D([0,1],e,2)

In [10]: print Cn.string()
------> print(Cn.string())
(1, [0,1],-2-si)
```

This cycle depends on a variable `sign` and it must be substituted with a numeric value before a visualisation becomes possible:

```
In [11]: A.send(cycle2D(Cn.subs(sign==-1)).asy_string())

In [12]: A.send(cycle2D(Cn.subs(sign==0)).asy_string())

In [13]: A.send(cycle2D(Cn.subs(sign==1)).asy_string())

In [14]: A.shipout("cycles")

In [15]: del(A)
```

By now, a separate window will have opened with cycle `Cn` drawn triply as a circle, parabola and hyperbola. The image is also saved in the Encapsulated Postscript (EPS) file `cycles.eps` in the current directory.

Note that you do not need to retype inputs `12` and `13` from scratch. Up/down arrows scroll the input history, so you can simply edit the value of `sign` in the input line `11`. Also, since you are in Linux, the `Tab` key will do a completion for you whenever possible.

The interactive shell evaluates and remember all expressions, so it may sometime be useful to restart it. It can be closed by `Ctrl-D` and started from the Main Menu (the bottom-left corner of the screen) using Accessories → CAS pycyle. In the same menu folder, there are two items which open documentation about the library in PDF and HTML formats.

A.3.3. *Further Usage.* There are several batch checks which can be performed with CAS. Open a terminal window from Main Menu → Accessories → LXTerminal. Type at the command prompt:

```
$ cd ~/CAS/pycycle/
$ ./run-pyGiNaC.sh test_pycycle.py
```

A comprehensive test of the library will be performed and the end of the output will look like this:

```
True: sl2_clifford_list:  (0)
True: sl2_clifford_matrix:  (0)
True: jump_fnct (-1)

Finished. The total number of errors is 0
```

Under normal circumstances, the reported total number of errors will, of course, be zero. You can also run all exercises from this book in a batch. From a new terminal window, type:

```
$ cd ~/CAS/pycycle/Examples/
$ ./check_all_exercises.sh
```

Exercises will be performed one by one with their numbers reported. Numerous graphical windows will be opened to show pencils of cycles. These windows can be closed by pressing the `q` key for each of them. This batch file suppresses all output from the exercises, except those containing the `False` string. Under normal circumstances, these are only Exercises 7.14.i and 7.14.ii.

You may also access the CAS from a command line. This may be required if the graphic X server failed to start for any reason. From the command prompt, type the following:

```
$ cd ~/CAS/pycycle/Examples/
$ ./run-pyGiNaC.sh
```

The full capacity of the CAS is also accessible from the command prompt, except for the preview of drawn cycles in a graphical window. However, EPS files can still be created with `Asymptote`—see `shipout()` method.

A.4. **Library for Cycles.** Our `C++` library defines the class `cycle` to manipulate cycles of arbitrary dimension in a symbolic manner. The derived class `cycle2D` is tailored to manipulate two-dimensional cycles. For the purpose of the book, we briefly list here some methods for `cycle2D` in the `pyGiNaC` binding form only.

constructors: There are two main forms of `cycle2D` constructors:

```
C=cycle2D(k,[l,n],m,e) # Cycle defined by a quadruple
Cr=([u,v],e,r) # Cycle with center at [u,v] and radius r2
```

In both cases, we use a metric defined by a Clifford unit `e`.

operations: Cycles can be added (`+`), subtracted (`-`) or multiplied by a scalar (method `exmul()`). A simplification is done by `normal()` and substitution by `subs()`. Coefficients of cycles can be normalised by the methods `normalize()` (k-normalisation), `normalize_det()` and `normalize_norm()`.

evaluations: For a given cycle, we can make the following evaluations: `hdet()`—determinant of its (hypercomplex) SFSCc matrix, `radius_sq()`—square of the radius, `val()`—value of a cycle at a point, which is the power of the point to the cycle.

similarities: There are the following methods for building cycle similarities: `sl2_similarity()`, `matrix_similarity()` and `cycle_similarity()` with an element of $\mathrm{SL}_2(\mathbb{R})$, a matrix or another cycle, respectively.

checks: There are several checks for cycles, which return `GiNaC` relations. The latter may be converted to Boolean values if no variables are presented within them. The checks for a single cycle are: `is_linear()`, `is_normalized()` and `passing()`, the latter requires a parameter (point). For two cycles, they are `is_orthogonal()` and `is_f_orthogonal()`.

specialisation: Having a cycle defined through several variables, we may try to specialise it to satisfy some further conditions. If these conditions are *linear* with respect to the cycle's variables, this can be achieved through the very useful method `subject_to()`. For example, for the above defined cycle `C`, we can find

```
C2=C.subject_to([C.passing([u,v]), C.is_orthogonal(C1)])
```

where `C2` will be a generic cycle passing the point `[u,v]` and orthogonal to `C1`. See the proof of Theorem 4.13 for an application.

specific: There are the following methods specific to two dimensions: `focus()`, `focal_length()`—evaluation of a cycle's focus and focal length and `roots()`—finding intersection points with a vertical or horizontal line. For a generic line, use method `line_intersect()` instead.

drawing: For visualisation through `Asymptote`, you can use various methods: `asy_draw()`, `asy_path()` and `asy_string()`. They allow you to define the bounding box, colour and style of the cycle's drawing. See the examples or full documentation for details of usage.

Further information can be obtained from electronic documentation on the enclosed DVD, an inspection of the test file `CAS/pycycle/test_pycycle.py` and solutions of the exercises.

A.5. **Predefined Objects at Initialisation.** For convenience, we predefine many `GiNaC` objects which may be helpful. Here is a brief indication of the most-used:

`realsymbol`.: `a, b, c, d`: elements of $\mathrm{SL}_2(\mathbb{R})$ matrix.
`u, v, u1, v1, u2, v2`: coordinates of points.
`r, r1, r2`: radii.
`k, l, n, m, k1, l1, n1, m1`: components of cycles.
`sign, sign1, sign2, sign3, sign4`: signatures of various metrics.
`s, s1, s2, s3`: s parameters of SFSCc matrices.
`x, y, t`: spare to use.

`varidx`.: `mu, nu, rho, tau`: two-dimensional (in vector formalism) or one-dimensional indexes for Clifford units.

`matrix`.: `M, M1, M2, M3`: diagonal 2×2 matrices with entries -1 and i-th `sign` on their diagonal.
`sign_mat, sign_mat1, sign_mat2`: similar matrices with i-th `s` instead of `sign`.

`clifford_unit`.: `e, es, er, et`: Clifford units with metrics derived from matrices `M, M1, M2, M3`, respectively.

`cycle2D`.: The following cycles are predefined:

```
C=cycle2D(k,[l,n],m,e)     # A generic cycle
C1=cycle2D(k1,[l1,n1],m1,e)# Another generic cycle
Cr=([u,v],e,r2) # Cycle with centre at [u,v] and radius r2
Cu=cycle2D(1,[0,0],1,e)    # Unit cycle
real_line=cycle2D(0,[0,1],0,e)
Z=cycle2D([u,v], e)        # Zero radius cycles at [u,v]
Z1=cycle2D([u1,v1], e)     # Zero radius cycles at [u1,v1]
Zinf=cycle2D(0,[0,0],1,e)  # Zero radius cycles at infinity
```

The solutions of the exercises make heavy use of these objects. Their exact definition can be found in the file `CAS/pycycle/init_cycle.py` from the home directory.

Appendix B. Textual output of the program

Conjugation of a cycle comes through Moebius transformation for vectors: true
Conjugation of a cycle comes through Moebius transformation for paravectors: true
A K-orbit is preserved for vectors: true, and passing (0, t): true
A K-orbit is preserved for paravectors: true, and passing (0, t): true
  Determinant of zero-radius Z1 cycle in metric e is for vector: $-\sigma v^2 + v^2\breve{\sigma}$
  The opposite value for paravector: true
  Focus of zero-radius cycle is (vector): $u, \frac{1}{2}\sigma v - \frac{1}{2}v\breve{\sigma}$
  The same value for paravector: true
  Centre of zero-radius cycle is (vector): $u, -\sigma v$
  The same value for paravector: true
  Focal length of zero-radius cycle is (vector): $\frac{1}{2}v$
  The same value for paravector: true
Image of the zero-radius cycle under Moebius transform has zero radius vector: true and paravector: true
The centre of the Moebius transformed zero-radius cycle for vector: -equal-, -equal-
The centre of the Moebius transformed zero-radius cycle for paravector: -equal-, -equal-
Image of the zero-radius cycle under cycle similarity has zero radius for vector: true
The centre of the conjugated zero-radius cycle coinsides with Moebius trans for vector: -equal-, -equal-
Image of the zero-radius cycle under cycle similarity has zero radius for paravector: true
The centre of the conjugated zero-radius cycle coinsides with Moebius trans for paravector: -equal-, -equal-
  The orthogonality in vectors is: $\tilde{m}k + 2n\tilde{n}\breve{\sigma} + \tilde{k}m - 2\tilde{l}l == 0$
for paravectors is the same: true
  The orthogonality of two lines is: $2n\tilde{n}\breve{\sigma} - 2\tilde{l}l == 0$
  The orthogonality to z-r-cycle is: $-2ul + u^2k + m + 2nv\breve{\sigma} - \sigma v^2 k == 0$
for paravectors is the same: true
  The orthogonality of two z-r-cycle is: $-\sigma v^2 - \chi(\sigma_2)v1^2 - u1^2 - 2uu1 + 2vv1\breve{\sigma} + u^2 == 0$
for paravectors is the same: true
Both orthogonal cycles (through one point and through its inverse) are the same for vector: true
Orthogonal cycle passes through the transformed point vector: true
Both orthogonal cycles (through one point and through its inverse) are the same for paravector: true
Orthogonal cycle passes through the transformed point paravector: true
For vectors
  Line through point and its inverse is orthogonal: true
  All lines come through the point $(\frac{l}{k}, -\frac{n\breve{\sigma}}{k})$
  Conjugated vector is parallel to (u,v): true
For paravectors
  Line through point and its inverse is orthogonal: true
  All lines come through the point $(\frac{l}{k}, -\frac{n\breve{\sigma}}{k})$
  Conjugated vector is parallel to (u,v): true
For vectors
  Ghost cycle has common roots with C : true
  $\chi(\sigma)$-centre of ghist cycle is equal to $\breve{\sigma}$-centre of C: true
  Inversion in (C-ghost, sign) coincides with inversion in (C, sign1): true
For paravectors
  Ghost cycle has common roots with C : true
  $\chi(\sigma)$-centre of ghist cycle is equal to $\breve{\sigma}$-centre of C: true
  Inversion in (C-ghost, sign) coincides with inversion in (C, sign1): true
For vectors
  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
For paravectors
  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
For vectors
  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
For paravectors
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
For vectors Yaglom inversion of the second kind is three reflections in the cycles: true
For paravectors Yaglom inversion of the second kind is three reflections in the cycles: true
For vectors The real line is Moebius invariant: true
For paravectors The real line is Moebius invariant: true
Reflection in the real line (vector): $(1, \begin{pmatrix} u & -v \end{pmatrix}_{symbol4262}, -\sigma v^2 + u^2)$
for paravector is the same: true
Reflection of the real line in cycle C (vectors):
$(2n\chi(\sigma_2)\chi(\sigma_3)k\breve{\sigma}, \begin{pmatrix} 2n\chi(\sigma_2)\chi(\sigma_3)l\breve{\sigma} & -\chi(\sigma_2)km + n^2\chi(\sigma_2)\breve{\sigma} + \chi(\sigma_2)l^2 \end{pmatrix}_{symbol4443}, 2n\chi(\sigma_2)\chi(\sigma_3)m\breve{\sigma})$
for paravectors is the same: true
The f-orthogonality is (vectors): $\chi(\sigma_2)\tilde{n}l^2 + n\chi(\sigma_2)\tilde{k}m - 2n\chi(\sigma_2)\tilde{l}l + n^2\chi(\sigma_2)\tilde{n}\breve{\sigma} - \chi(\sigma_2)km\tilde{n} + n\chi(\sigma_2)\tilde{m}k == 0$
for paravectors is the same: true
The f-orthogonality of two lines is (vectors): $\chi(\sigma_2)\tilde{n}l^2 - 2n\chi(\sigma_2)\tilde{l}l + n^2\chi(\sigma_2)\tilde{n}\breve{\sigma} == 0$
for paravectors is the same: true
The f-orthogonality to z-r-cycle is first way (vectors):
$nu^2\chi(\sigma_2)k + n^2\chi(\sigma_2)v\breve{\sigma} - n\chi(\sigma_2)v^2k\breve{\sigma} - 2nu\chi(\sigma_2)l + \chi(\sigma_2)vl^2 + n\chi(\sigma_2)m - \chi(\sigma_2)vkm == 0$
for paravectors is the same: true
The f-orthogonality to z-r-cycle in second way (vectors):
$\chi(\sigma_2)vm + 2n\chi(\sigma_2)v^2\breve{\sigma} - \chi(\sigma_2)v^3k\breve{\sigma} + u^2\chi(\sigma_2)vk - 2u\chi(\sigma_2)vl == 0$
for paravectors is the same: true
The f-orthogonality of two z-r-cycle is (vectors):
$2\chi(\sigma_2)v^2v1\breve{\sigma} - 2u\chi(\sigma_2)u1v - \sigma\chi(\sigma_2)vv1^2 - \chi(\sigma_2)v^3\breve{\sigma} + \chi(\sigma_2)u1^2v + u^2\chi(\sigma_2)v == 0$
for paravectors is the same: true
For vectors all lines come through the focus related $\breve{e}$: true
For paravectors all lines come through the focus related $\breve{e}$: true
For vectors
f-ghost cycle has common roots with C: true
$\chi(\sigma)$-center of f-ghost cycle coincides with $\breve{\sigma}$-focus of C : true
f-inversion in C coincides with inversion in f-ghost cycle: true
For paravectors
f-ghost cycle has common roots with C: true
$\chi(\sigma)$-center of f-ghost cycle coincides with $\breve{\sigma}$-focus of C : true
f-inversion in C coincides with inversion in f-ghost cycle: true
For vectors
Distance between (u,v) and (u',v') in elliptic and hyperbolic spaces is
$\dfrac{(4vv1(-1+\sigma\breve{\sigma}) + \breve{\sigma}(\sigma(v-v1)^2 - (u-u1)^2))(\sigma(v-v1)^2 - (u-u1)^2)}{(u-u1)^2\breve{\sigma} - (v-v1)^2}$: true
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: $\frac{1}{2}\frac{\sigma v1^3 - 2uu1v - 2\sigma u1^2v1\breve{\sigma} + u1^2v1 + 3\sigma v^2v1 - 2uu1v1 - \sigma v^3 + u1^2v + u^2v1 - 3\sigma vv1^2 - 2u^2\sigma v1\breve{\sigma} + u^2v + 4u\sigma u1v1\breve{\sigma}}{2uu1\breve{\sigma} + v1^2 - 2vv1 + v^2 - u1^2\breve{\sigma} - u^2\breve{\sigma}}$
$\frac{1}{2}\frac{\sigma u1v1^2\breve{\sigma} + 2uv1^2 + u1^3\breve{\sigma} + u\sigma v^2\breve{\sigma} - u^3\breve{\sigma} - \sigma u1v^2\breve{\sigma} - 2uvv1 - 2u1v1^2 + 3u^2u1\breve{\sigma} - u\sigma v1^2\breve{\sigma} - 3uu1^2\breve{\sigma} + 2u1vv1}{2uu1\breve{\sigma} + v1^2 - 2vv1 + v^2 - u1^2\breve{\sigma} - u^2\breve{\sigma}}$
Value at the middle point (parabolic point space):
$u1^2 - 2uu1 + u^2$
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: $\sigma v1$; $\frac{1}{2}u - \frac{1}{2}u1$
For paravectors
Distance between (u,v) and (u',v') in elliptic and hyperbolic spaces is
$\dfrac{(4vv1(-1+\sigma\breve{\sigma}) + \breve{\sigma}(\sigma(v-v1)^2 - (u-u1)^2))(\sigma(v-v1)^2 - (u-u1)^2)}{(u-u1)^2\breve{\sigma} - (v-v1)^2}$: true
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: $\frac{1}{2}\frac{\sigma v1^3-2uu1v-2\sigma u1^2v1\breve{\sigma}+u1^2v1+3\sigma v^2v1-2uu1v1-\sigma v^3+u1^2v+u^2v1-3\sigma vv1^2-2u^2\sigma v1\breve{\sigma}+u^2v+4u\sigma u1v1\breve{\sigma}}{2uu1\breve{\sigma}+v1^2-2vv1+v^2-u1^2\breve{\sigma}-u^2\breve{\sigma}}$

$\frac{1}{2}\frac{\sigma u1v1^2\breve{\sigma}+2uv1^2+u1^3\breve{\sigma}+u\sigma v^2\breve{\sigma}-u^3\breve{\sigma}-\sigma u1v^2\breve{\sigma}-2uvv1-2u1v1^2+3u^2u1\breve{\sigma}-u\sigma v1^2\breve{\sigma}-3uu1^2\breve{\sigma}+2u1vv1}{2uu1\breve{\sigma}+v1^2-2vv1+v^2-u1^2\breve{\sigma}-u^2\breve{\sigma}}$

Value at the middle point (parabolic point space):

$u1^2-2uu1+u^2$

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: $\sigma v1$; $\frac{1}{2}u-\frac{1}{2}u1$

For vectors distance between (u,v) and (u',v') (value at critical point):

$-\frac{2uu1\breve{\sigma}-4v^2-u1^2\breve{\sigma}+4\sigma v^2\breve{\sigma}-u^2\breve{\sigma}}{\breve{\sigma}}$

for paravector is the same: true

For vectors

Length from *center* between (u,v) and (u',v'):

$\frac{u1^2\mathring{\sigma}^2-2uu1\mathring{\sigma}^2-\sigma v1^2\mathring{\sigma}^2-v^2\breve{\sigma}+u^2\mathring{\sigma}^2+2vv1\mathring{\sigma}}{\mathring{\sigma}^2}$

This distance/length is conformal: true

Perpendicular to ((u,v); (u',v')) is: $\frac{\sigma v1\mathring{\sigma}-v}{\mathring{\sigma}}$; $u-u1$

Length from *focus* check for $p=(\sqrt{(u-u1)^2\mathring{\sigma}+(v-v1)^2-\sigma v1^2\mathring{\sigma}}+v-v1)\mathring{\sigma}$

Length between (u,v) and (u', v') is equal to $(\mathring{(\sigma)}-\breve{\sigma})p^2-2vp$: true

checks: C11 passes through (u', v'): true; C11 focus is at (u, v): true

This distance/length is conformal: true

Perpendicular to ((u,v); (u',v')) is $(\sigma v'+p,u-u')$: true

Length from *focus* check for $p=-\mathring{\sigma}(\sqrt{(u-u1)^2\mathring{\sigma}+(v-v1)^2-\sigma v1^2\mathring{\sigma}}-v+v1)$

Length between (u,v) and (u', v') is equal to $(\mathring{(\sigma)}-\breve{\sigma})p^2-2vp$: true

checks: C11 passes through (u', v'): true; C11 focus is at (u, v): true

This distance/length is conformal: true

Perpendicular to ((u,v); (u',v')) is $(\sigma v'+p,u-u')$: true

Shall be 'false' for conformality below

Length from *focus* check for $p=\frac{1}{2}\frac{\sigma v1^2-(u-u1)^2}{v-v1}$

Length between (u,v) and (u', v') is equal to $(\mathring{(\sigma)}-\breve{\sigma})p^2-2vp$: true

checks: C11 passes through (u', v'): true; C11 focus is at (u, v): true

This distance/length is conformal: false. The factor is:

$\frac{y^2}{(yd^2+\sigma yc^2v^2+u^2yc^2+2uycd-2uc^2vx-2cvdx)^2}$

Perpendicular to ((u,v); (u',v')) is $(\sigma v'+p,u-u')$: true

For paravectors

Length from *center* between (u,v) and (u',v'):

$\frac{u1^2\mathring{\sigma}^2-2uu1\mathring{\sigma}^2-\sigma v1^2\mathring{\sigma}^2-v^2\breve{\sigma}+u^2\mathring{\sigma}^2+2vv1\mathring{\sigma}}{\mathring{\sigma}^2}$

This distance/length is conformal: true

Perpendicular to ((u,v); (u',v')) is: $\frac{\sigma v1\mathring{\sigma}-v}{\mathring{\sigma}}$; $u-u1$

Length from *focus* check for $p=(\sqrt{(u-u1)^2\mathring{\sigma}+(v-v1)^2-\sigma v1^2\mathring{\sigma}}+v-v1)\mathring{\sigma}$

Length between (u,v) and (u', v') is equal to $(\mathring{(\sigma)}-\breve{\sigma})p^2-2vp$: true

checks: C11 passes through (u', v'): true; C11 focus is at (u, v): true

This distance/length is conformal: true

Perpendicular to ((u,v); (u',v')) is $(\sigma v'+p,u-u')$: true

Length from *focus* check for $p=-\mathring{\sigma}(\sqrt{(u-u1)^2\mathring{\sigma}+(v-v1)^2-\sigma v1^2\mathring{\sigma}}-v+v1)$

Length between (u,v) and (u', v') is equal to $(\mathring{(\sigma)}-\breve{\sigma})p^2-2vp$: true

checks: C11 passes through (u', v'): true; C11 focus is at (u, v): true

This distance/length is conformal: true

Perpendicular to ((u,v); (u',v')) is $(\sigma v'+p,u-u')$: true

Shall be 'false' for conformality below

Length from *focus* check for $p=\frac{1}{2}\frac{\sigma v1^2-(u-u1)^2}{v-v1}$

Length between (u,v) and (u', v') is equal to $(\mathring{(\sigma)}-\breve{\sigma})p^2-2vp$: true

checks: C11 passes through (u', v'): true; C11 focus is at (u, v): true

This distance/length is conformal: false. The factor is:

$\frac{y^2}{(yd^2+\sigma yc^2v^2+u^2yc^2+2uycd-2uc^2vx-2cvdx)^2}$

Perpendicular to ((u,v); (u',v')) is $(\sigma v'+p,u-u')$: true

Inf cycle is: $(1,\begin{pmatrix} u & \frac{v_p}{\mathring{\sigma}-\breve{\sigma}}-\frac{\sqrt{v_p^2+\epsilon^2\mathring{\sigma}-\epsilon^2\breve{\sigma}}}{\mathring{\sigma}-\breve{\sigma}}\end{pmatrix}^{symbol6306},-\frac{\breve{\sigma}(\sqrt{v_p^2+\epsilon^2(\mathring{\sigma}-\breve{\sigma})}-v_p)^2}{(\mathring{\sigma}-\breve{\sigma})^2}-\epsilon^2+u^2)$

For paravector is the same: true
Square of radius of the infinitesimal cycle is: ϵ^2
For paravector is the same: true
Focus of infinitesimal cycle is: u, v_p
For paravector is the same: true
Focal length is: $(-\frac{1}{4}\frac{1}{v_p})\epsilon^2 + \mathcal{O}(\epsilon^3)$
For paravector is the same: true
Infinitesimal cycle (vector) passing points$(u + \epsilon x, vp + (-x^2 v_p) + (-\frac{1}{4}\frac{\mathring{\sigma}x^2 - x^2\breve{\sigma} - \mathring{\sigma}}{v_p})\epsilon^2 + \mathcal{O}(\epsilon^3))$,
Infinitesimal cycle (paravector) passing points$(u + \epsilon x, vp + (-x^2 v_p) + (-\frac{1}{4}\frac{\mathring{\sigma}x^2 - x^2\breve{\sigma} - \mathring{\sigma}}{v_p})\epsilon^2 + \mathcal{O}(\epsilon^3))$,
Image under SL2(R) of infinitesimal cycle has radius squared:
$(-\frac{4\mathring{\sigma}\breve{\sigma} - \breve{\sigma}^2 - 6\mathring{\sigma}^2\breve{\sigma}^2 - \mathring{\sigma}^4 + 4\mathring{\sigma}^3\breve{\sigma}}{(2ucd\breve{\sigma}^2 + u^2c^2\breve{\sigma}^2 - 2d^2\mathring{\sigma}\breve{\sigma} + u^2c^2\mathring{\sigma}^2 + 2ucd\mathring{\sigma}^2 - 4ucd\mathring{\sigma}\breve{\sigma} - 2u^2c^2\mathring{\sigma}\breve{\sigma} + d^2\breve{\sigma}^2 + d^2\mathring{\sigma}^2)^2})\epsilon^2 + \mathcal{O}(\epsilon^3)$
For paravector is the same: true
Image under cycle similarity of infinitesimal cycle has radius squared:
$(\frac{n^4\breve{\sigma}^2 + 8km\mathring{\sigma}^3l^2\breve{\sigma} + n^4\mathring{\sigma}^4\breve{\sigma}^2 - 2n^2\mathring{\sigma}^4l^2\breve{\sigma} - 4n^4\mathring{\sigma}\breve{\sigma} - 8n^2km\mathring{\sigma}\breve{\sigma}^2 + 6k^2m^2\mathring{\sigma}^2\breve{\sigma}^2 - 2kml^2\breve{\sigma}^2 - 12n^2\mathring{\sigma}^2l^2\breve{\sigma} + 6\mathring{\sigma}^2l^4\breve{\sigma}^2 - 4k^2m^2\mathring{\sigma}^3\breve{\sigma} + 2n^2km\mathring{\sigma}^4\breve{\sigma} + 8km\mathring{\sigma}l^2\breve{\sigma} + k^2m^2\breve{\sigma}^2 + l}{(2n^2\mathring{\sigma}\breve{\sigma}^2 + 4uk\mathring{\sigma}l\breve{\sigma} - 2uk\mathring{\sigma}^2l - 2\mathring{\sigma}l^2\breve{\sigma} - n^2\mathring{\sigma}^2\breve{\sigma} - 2}$
$\mathcal{O}(\epsilon^3)$
For paravector is the same: true
Focus of the transormed cycle is from transformation of focus by: $(\begin{pmatrix} 0 \\ 0 \end{pmatrix}) + (\begin{pmatrix} 0 \\ 0 \end{pmatrix})\epsilon + \mathcal{O}(\epsilon^2)$
Orthogonality (leading term) to infinitesimal cycle is:
$(-2ul + u^2k + m == 0) + \mathcal{O}(\epsilon)$
f-orthogonality of other cycle to infinitesimal:
$(-2nul + nu^2k + nm == 0) + \mathcal{O}(\epsilon)$
f-orthogonality of infinitesimal cycle to other:
$(0 == 0) + (0 == 0)\epsilon + (\frac{1}{2}(\frac{2ul + 2nv_p - u^2k - m}{v_p} == 0))\epsilon^2 + \mathcal{O}(\epsilon^3)$
Det of Cayley-transformed infinitesimal cycle: $(-\frac{1 + u^2\breve{\sigma} - v_p}{v_p})\epsilon^2 + \mathcal{O}(\epsilon^3)$
Focus of the Cayley-transformed infinitesimal cycle displaced by: $(\mathcal{O}(\epsilon^2), \mathcal{O}(\epsilon^2))$
For paravector is the same: true
f-orthogonality of Cayley transforms of infinitesimal cycle to other:
$(0 == 0) + (0 == 0)\epsilon + (\frac{1}{2}(\frac{2ul + 2nv_p - u^2k - m}{v_p} == 0))\epsilon^2 + \mathcal{O}(\epsilon^3)$
Inf cycle is: $(1, \begin{pmatrix} u & \frac{1}{2}\frac{\epsilon^2}{v_p} \end{pmatrix}^{symbol17875}, -\frac{1}{4}\frac{\epsilon^4\breve{\sigma}}{v_p^2} - \epsilon^2 + u^2)$
For paravector is the same: true
Square of radius of the infinitesimal cycle is: ϵ^2
For paravector is the same: true
Focus of infinitesimal cycle is: $u, -v_p$
For paravector is the same: true
Focal length is: $(\frac{1}{4}\frac{1}{v_p})\epsilon^2$
For paravector is the same: true
Infinitesimal cycle (vector) passing points$(u + \epsilon x, vp + (x^2 v_p - 2v_p) + (-\frac{1}{4}\frac{\breve{\sigma}}{v_p})\epsilon^2)$,
Infinitesimal cycle (paravector) passing points$(u + \epsilon x, vp + (x^2 v_p - 2v_p) + (-\frac{1}{4}\frac{\breve{\sigma}}{v_p})\epsilon^2)$,
Image under SL2(R) of infinitesimal cycle has radius squared:
$(\frac{1}{(u^2c^2 + 2ucd + d^2)^2})\epsilon^2 + \mathcal{O}(\epsilon^3)$
For paravector is the same: true
Image under cycle similarity of infinitesimal cycle has radius squared:
$(\frac{n^4\breve{\sigma}^2 + k^2m^2 - 2kml^2 + l^4 - 2n^2l^2\breve{\sigma} + 2n^2km\breve{\sigma}}{(u^2k^2 + l^2 - 2ukl - n^2\breve{\sigma})^2})\epsilon^2 + \mathcal{O}(\epsilon^3)$
For paravector is the same: true
Focus of the transormed cycle is from transformation of focus by: $(\begin{pmatrix} 0 \\ -2\frac{v_p}{u^2c^2 + 2ucd + d^2} \end{pmatrix}) + (\begin{pmatrix} 0 \\ 0 \end{pmatrix})\epsilon + \mathcal{O}(\epsilon^2)$
Orthogonality (leading term) to infinitesimal cycle is:
$(-2ul + u^2k + m == 0) + \mathcal{O}(\epsilon)$
f-orthogonality of other cycle to infinitesimal:
$(-2nul + nu^2k + nm == 0) + \mathcal{O}(\epsilon)$
f-orthogonality of infinitesimal cycle to other:
$(0 == 0) + (0 == 0)\epsilon + (\frac{1}{2}(-\frac{2ul - 2nv_p - u^2k - m}{v_p} == 0))\epsilon^2 + \mathcal{O}(\epsilon^3)$
Det of Cayley-transformed infinitesimal cycle: $(\frac{1 + u^2\breve{\sigma} + v_p}{v_p})\epsilon^2 + \mathcal{O}(\epsilon^3)$
Focus of the Cayley-transformed infinitesimal cycle displaced by: $(\begin{pmatrix} 0 \\ -2v_p \end{pmatrix}) + (\begin{pmatrix} 0 \\ 0 \end{pmatrix})\epsilon + \mathcal{O}(\epsilon^2)$
For paravector is the same: true

f-orthogonality of Cayley transforms of infinitesimal cycle to other:
$(0 == 0) + (0 == 0)\epsilon + (\frac{1}{2}(-\frac{2ul-2nv_p-u^2k-m}{v_p} == 0))\epsilon^2 + \mathcal{O}(\epsilon^3)$

Appendix C. Example of the produced graphics

An example of graphics generated by the program is given in Figure 3. This was produced by the part of program from the Section D.1.1.

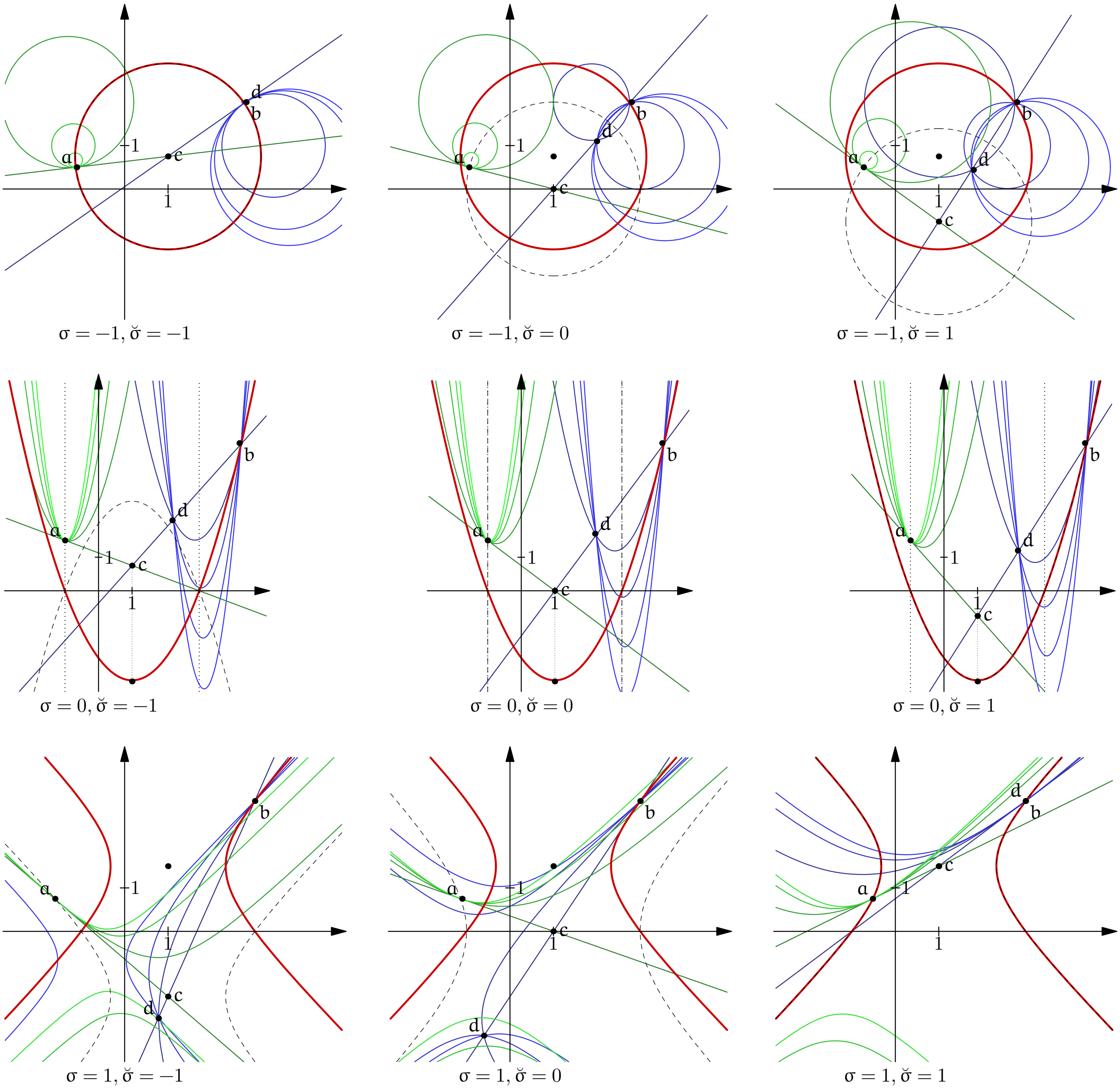

Figure 3. Orthogonality of the first kind in nine combinations.

Appendix D. Details of the Asymptote Drawing

D.1. Drawing Orthogonality Conditions.

D.1.1. *First Orthogonality Condition.* We define numeric values of all involved parameters first.

50a ⟨Drawing first orthogonality 50a⟩≡ (37a) 50b ▷

```
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), Cq =C2;
lst U, V;
```

Uses cycle2D 9a 9b 15c 15c 15d 15d 54b 55c 61b 62d 62d 62d 62d 62d 77b 77b 77b 77b 77b 89b 89b 89b 89b 92c, k 3a, l 3a, m 3a, numeric 14a 57d, si 14b, si1 14b, and subs 4b.

We use various initial data for various geometries.

50b ⟨Drawing first orthogonality 50a⟩+≡ (37a) ◁50a 50c ▷

```
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;
}
```

Uses center 5f, k 3a, l 3a, normal 4b, numeric 14a 57d, op 4b, points 103a, roots 9g, si 14b, and subs 4b.

Moebius transform of the first point.

50c ⟨Drawing first orthogonality 50a⟩+≡ (37a) ◁50b

```
U.append(P.op(0).subs(cycle_val).subs(lst{u ≡ U.op(0), v ≡ V.op(0)}).normal());
V.append(P.op(1).subs(cycle_val).subs(lst{u ≡ U.op(0), v ≡ V.op(0)}).normal());

asymptote ≪ endl ≪ "erase();" ≪ endl ≪ "size(175);" ≪ endl;
⟨Drawing orthogonal cycles 50d⟩
asymptote ≪ "shipout(\"first-ort-" ≪ eph_names[si+1] ≪ eph_names[si1+1] ≪ "\");" ≪ endl;
```

Uses normal 4b, op 4b, si 14b, si1 14b, subs 4b, u 100a, and v 100a.

We start drawing from cycles.

50d ⟨Drawing orthogonal cycles 50d⟩≡ (50c 52b) 51a ▷

```
for (int j = 0; j<2; j++)
   for (int i=0; i<(si≡1?4:5); i++)
      Cq.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 asy_draw 11a, numeric 14a 57d, op 4b, si 14b, subs 4b, u 100a, and v 100a.

To finish we add some additional drawing explaining the picture.

51a ⟨Drawing orthogonal cycles 50d⟩+≡ (50c 52b) ◁50d

```
asymptote ≪ "pair[] z={(" ≪ ex_to<numeric>(U.op(0).evalf()).to_double() ≪ ", "
 ≪ ex_to<numeric>(V.op(0).evalf()).to_double() ≪ ")";
 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() ≪ ")" ;


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 51c⟩
⟨Draw axes 51b⟩
```

Uses `numeric` 14a 57d, `op` 4b, and `si` 14b.

This chunk draws the standard coordinat axes.

51b ⟨Draw axes 51b⟩≡ (51a 53–57)

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

51c ⟨Put units 51c⟩≡ (51a 56d)

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

Uses `si` 14b and `si1` 14b.

D.1.2. *Focal Orthogonality Condition.* We draw some `Asymptote` pictures to illustrate the focal orthogonality relation. We define numeric values of all involved parameters first.

51d ⟨Drawing focal orthogonality 51d⟩≡ (37a) 52a▷

```
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), //sign3 == jump_fnct(-
si), //sign3 == (si > 0?numeric(-1):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), Cq =C6;
lst U, V;
```

Uses `cycle2D` 9a 9b 15c 15c 15d 15d 54b 55c 61b 62d 62d 62d 62d 62d 77b 77b 77b 77b 77b 89b 89b 89b 89b 92c, `jump_fnct` 59d, `k` 3a, `l` 3a, `m` 3a, `numeric` 14a 57d, `si` 14b, `si1` 14b, and `subs` 4b.

We use various initial data for various geometries.

52a ⟨Drawing focal orthogonality 51d⟩+≡ (37a) ◁51d 52b▷

```
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;
}
```

Uses `center` 5f, `focus` 9f, `k` 3a, `l` 3a, `normal` 4b, `numeric` 14a 57d, `op` 4b, `points` 103a, `roots` 9g, `si` 14b, and `subs` 4b.

Moebius transform of *P1*.

52b ⟨Drawing focal orthogonality 51d⟩+≡ (37a) ◁52a

```
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(175);" ≪ endl;
⟨Drawing orthogonal cycles 50d⟩
asymptote ≪ "shipout(\"sec-ort-" ≪ eph_names[si+1] ≪ eph_names[si1+1] ≪ "\");" ≪ endl;
```

Uses `normal` 4b, `op` 4b, `si` 14b, `si1` 14b, `subs` 4b, `u` 100a, and `v` 100a.

D.2. **Extra pictures from `Asymptote`.** We draw few more pictures in `Asymptote`.

52c ⟨Extra pictures from Asymptote 52c⟩≡ (37b)

```
numeric xmin(-5), xmax(5), ymin(-13,4), ymax = numeric(6);
⟨Three images of the same cycle 53a⟩
⟨Centres and foci of parabolas 53b⟩
⟨Zero-radius cycle implementations 54a⟩
⟨Parabolic diameters 54b⟩
⟨Distance as an extremum 55a⟩
⟨Infinitesimal cycles draw 55c⟩
⟨Cayley transform pictures 55d⟩
⟨Three inversions 56e⟩
⟨Hyperbolic inversion of a ball 57c⟩
```

Uses `numeric` 14a 57d.

D.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.

53a ⟨Three images of the same cycle 53a⟩≡ (52c)

```
asymptote << endl << "erase();" << endl << "size(250);" << 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) << ")); z.push(("
    << C2f.center().op(0) << ", " << C2f.center().op(1) << "));" << endl;
}
asymptote << "z.push((" << C1f.roots().op(0) << ", 0));  z.push((" << C1f.roots().op(1) << ", 0));" << 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 axes 51b⟩
asymptote << "shipout(\"same-cycle\");" << endl;
```

Uses asy_draw 11a, center 5f, cycle 3a 3a 3b 3b 3c 3d 5a 5a 5a 5a 5a 72c 72d 72d 73a 75d 75d 77a 77a 77a 77a 77a, cycle2D 9a 9b 15c 15c 15d 15d 54b 55c 61b 62d 62d 62d 62d 62d 77b 77b 77b 77b 77b 89b 89b 89b 89b 92c, op 4b, and roots 9g.

D.2.2. *Centres and foci of cycles.* We draw two parabolas and their centres with three type of foci.

53b ⟨Centres and foci of parabolas 53b⟩≡ (52c)

```
asymptote << endl << "erase();" << endl << "size(250);" << endl;
C1f = cycle2D(1, lst{-1.5, 2}, 3.75, par_matr);
C2f = cycle2D(1, lst{2, 2}, -3.5, par_matr);
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(-unit_matrix(2)).op(0) << ", " << C1f.center(-unit_matrix(2)).op(1)
 << "), (" << C2f.center(-unit_matrix(2)).op(0) << ", " << C2f.center(-unit_matrix(2)).op(1) << "), ";
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) << "), ("
    << F2.op(0) << ", " << F2.op(1) << ")" << (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 axes 51b⟩
asymptote << "shipout(\"parab-cent\");" << endl;
```

Uses asy_draw 11a, center 5f, cycle2D 9a 9b 15c 15c 15d 15d 54b 55c 61b 62d 62d 62d 62d 62d 77b 77b 77b 77b 77b 89b 89b 89b 89b 92c, ex 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, focus 9f, op 4b, and par_matr 13a.

D.2.3. *Zero-radius cycles.* Zero-radius cycles can look different in different EPH realisations, here is an illustration.

54a ⟨Zero-radius cycle implementations 54a⟩≡ (52c)
```
asymptote << endl << "erase();" << endl << "size(250);" << 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=lst{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()
                 << "), " << 0.4+0.4*i1 << "red+"
                 << .4-0.3*i2 << "green+"
                 << 0.6+0.3*i2 << "blue);" << endl;
      }
   }
   ⟨Draw axes 51b⟩
}
asymptote << "shipout(\"zero-cycles\");" << endl;
```

Uses asy_draw 11a, focus 9f, numeric 14a 57d, op 4b, subs 4b, u 100a, v 100a, and val 6a.

D.2.4. *Diameters of cycles.* The notion of diameter and related distance became strange in parabolic case.

54b ⟨Parabolic diameters 54b⟩≡ (52c)
```
asymptote << endl << "erase();" << endl << "size(250);" << endl;
C10 = cycle2D(1, lst{(-4-1)÷2.0, 0.5}, 4,par_matr);
C10.asy_draw(asymptote, xmin, xmax, ymin, ymax, lst{0.1, 0, 0.6});
asymptote << "pair[] z = {(" << C10.roots().op(0) << ", 0), (" << C10.roots().op(1) << ", 0)};" << endl;
cycle2D(1, lst{5÷2.0, 0.5}, 8,par_matr).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,par_matr);
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)); z.push((" << C10.roots().op(0) << ", 0));" << endl;
⟨Put labels on 22-23 54c⟩
⟨Draw axes 51b⟩
asymptote << "shipout(\"parab-diam\");" << endl;
```

Defines:
cycle2D, used in chunks 9, 10c, 16–23, 25, 26e, 28a, 30–33, 35a, 36, 50a, 51d, 53, 55–57, 61, 62, 64, 66d, 88–91, 93b, 94a, 96, 100, and 102b.

Uses asy_draw 11a, op 4b, par_matr 13a, and roots 9g.

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

54c ⟨Put labels on 22-23 54c⟩≡ (54b 55b)
```
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;
```

D.2.5. *Extremal property of the distance.* To illustrate the variational definition of the distance [16, Defn.5.2] we draw several cycles which passes two given points. The cycles with the extremal value of diameter is highlighted in bold.

55a ⟨Distance as an extremum 55a⟩≡ (52c) 55b▷

```
asymptote << endl << "erase();" << endl << "size(250);" << 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);
}
```

Uses asy_draw 11a, cycle2D 9a 9b 15c 15c 15d 15d 54b 55c 61b 62d 62d 62d 62d 62d 77b 77b 77b 77b 77b 89b 89b 89b 89b 92c, k 3a, l 3a, passing 6b, subject_to 6c, and subs 4b.

Put label on the picture.

55b ⟨Distance as an extremum 55a⟩+≡ (52c) ◁55a

```
asymptote << "pair[] z ={ (" << xmin+1 << ", " << ymax-5 << "),  (" << xmin+3 << ", "
                     << ymax-6.5 << "),  (" << xmax-4 << ", " << ymax-5 << "),  (" << xmax-1
                     << ", " << ymax-2 << ")};" << endl;
⟨Put labels on 22-23 54c⟩
asymptote << "  label(\"$d_e$\", .5z[0]+.5z[1], NE);" << endl
                     << "  label(\"$d_p$\", .5z[4]+.5z[5], S);" << endl;
⟨Draw axes 51b⟩
asymptote << "shipout(\"dist-extr\");" << endl;
```

D.2.6. *Infinitesimal cycles.* Here we draw a set of parabola with the same focus and the focal length tensing to zero.

55c ⟨Infinitesimal cycles draw 55c⟩≡ (52c)

```
asymptote << endl << "erase();" << endl << "size(250);" << endl;
for (int j=1; j < 5; j++) {
 cycle2D(lst{-2.5, 4.5}, -unit_matrix(2), 16.0*GiNaC::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}, hyp_matr, 25*GiNaC::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, GiNaC::pow(3,-j)}, 2*2+2.0*GiNaC::pow(3,-j)-GiNaC::pow(3,-2*j), par_matr)
   .asy_draw(asymptote, xmin, xmax, ymin, ymax, lst{1.0-0.17*j, 0, 0.17*j}, ".3", true, 7);
}
asymptote << " draw((2,1)--(2," << ymax << "), blue+1);" << endl;
cycle2D(lst{1, 1.25}, hyp_matr).asy_draw(asymptote, xmin, xmax, ymin, ymax÷3, lst{1, 0, 0}, "1");
asymptote << " dot((-2.5,4.5));" << endl
 << " dot((2,1));" << endl;
⟨Draw axes 51b⟩
asymptote << "shipout(\"infinites\");" << endl;
```

Defines:
cycle2D, used in chunks 9, 10c, 16–23, 25, 26e, 28a, 30–33, 35a, 36, 50a, 51d, 53, 55–57, 61, 62, 64, 66d, 88–91, 93b, 94a, 96, 100, and 102b.
Uses asy_draw 11a, hyp_matr 13a, and par_matr 13a.

D.2.7. *Pictures of the Cayley transform.* We draw now pictures of Cayley transform, which shows that the unit cycle *UC* may be obtained as a reflection of the real line into the cycle *C10f*.

55d ⟨Cayley transform pictures 55d⟩≡ (52c) 56a▷

```
xmin = -numeric(4,2); xmax=numeric(4,2); ymin=-numeric(7,2); ymax=numeric(3);
cycle2D C10f, UC;
C10f = cycle2D(1, lst{0, sign2}, sign, e);
UC=real_line.cycle_similarity(C10f, es).normalize();
```

Uses cycle2D 9a 9b 15c 15c 15d 15d 54b 55c 61b 62d 62d 62d 62d 62d 77b 77b 77b 77b 77b 89b 89b 89b 89b 92c, cycle_similarity 7e, normalize 5e, and numeric 14a 57d.

Now we run cycles over signatures of point and cycle spaces and sign of *sign2*.

56a ⟨Cayley transform pictures 55d⟩+≡ (52c) ◁55d 56b▷
```
for (si=-1; si<2; si++) {
 for (si1=-1; si1<2; si1++)
  if ((si ≡0 ) ∨ (si ≡ si1)) {
   asymptote ≪ endl ≪ "erase();" ≪ endl ≪ "size(250);" ≪ endl;
   for (int si2=-1; si2<2; si2=si2+2) {
    lst cycle_val = lst{sign ≡ si, sign1 ≡ si1, sign2≡si2};
```

Uses si 14b and si1 14b.

If point space is not parabolic, the unit cycle *UC* is the reflection of real line in *C10f* and we draw both of them.

56b ⟨Cayley transform pictures 55d⟩+≡ (52c) ◁56a 56c▷
```
    if (si ≠ 0 ) {
     ex_to<cycle2D>(UC.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);
     C10f.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 ? "1" : "Dotted "), true, 7);
```

Uses asy_draw 11a, cycle2D 9a 9b 15c 15c 15d 15d 54b 55c 61b 62d 62d 62d 62d 62d 77b 77b 77b 77b 77b 89b 89b 89b 89b 92c, normalize 5e, si 14b, si1 14b, and subs 4b.

In the parabolic space unit cycle obtained from the real line by *cayley_parab*() procedure.

56c ⟨Cayley transform pictures 55d⟩+≡ (52c) ◁56b 56d▷
```
    } else
     ex_to<cycle2D>(cayley_parab(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);
   }
```

Uses asy_draw 11a, cycle2D 9a 9b 15c 15c 15d 15d 54b 55c 61b 62d 62d 62d 62d 62d 77b 77b 77b 77b 77b 89b 89b 89b 89b 92c, and subs 4b.

The pictures are finished with standard stuff.

56d ⟨Cayley transform pictures 55d⟩+≡ (52c) ◁56c
```
   ⟨Put units 51c⟩
    ⟨Draw axes 51b⟩
   asymptote ≪ "shipout(\"cayley-"≪ eph_names[si+1] ≪ eph_names[si1+1]≪"\");" ≪ endl;
  }
 }
```

Uses si 14b and si1 14b.

D.2.8. *Three types of inversions.* We draw here pictures for three types of the inversions. First we make a rectangular grid.

56e ⟨Three inversions 56e⟩≡ (52c) 57a▷
```
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();" ≪ endl ≪ "size(250);" ≪ 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 axes 51b⟩
asymptote ≪ "shipout(\"pre-invers\");" ≪ endl;
```

Uses asy_draw 11a, cycle2D 9a 9b 15c 15c 15d 15d 54b 55c 61b 62d 62d 62d 62d 62d 77b 77b 77b 77b 77b 89b 89b 89b 89b 92c, l 3a, m 3a, and subs 4b.

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

57a ⟨Three inversions 56e⟩+≡ (52c) ◁56e 57b▷
```
  C4=C3.cycle_similarity(C2);
  for(int si=-1; si<2; si++) {
     asymptote << endl << "erase();" << endl << "size(250);" << 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 asy_draw 11a, cycle_similarity 7e, l 3a, m 3a, si 14b, and subs 4b.

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

57b ⟨Three inversions 56e⟩+≡ (52c) ◁57a
```
     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 axes 51b⟩
     asymptote << "shipout(\"inversion-" << eph_names[si+1] << "\");" << endl;
  }
```

Uses asy_draw 11a, cycle2D 9a 9b 15c 15c 15d 15d 54b 55c 61b 62d 62d 62d 62d 62d 77b 77b 77b 77b 77b 89b 89b 89b 89b 92c, cycle_similarity 7e, si 14b, and subs 4b.

D.2.9. *Drawing inversion of the hyperbolic ball.* A hyperbolic ball can be inverted without self-intersection. We produce here an illustration of this.

Firstly we define some parameters

57c ⟨Hyperbolic inversion of a ball 57c⟩≡ (52c) 57d▷
```
  const int frames=20, balls=10; // number of frames and balls
  const double r1=.1, r2=1, tmin=-3, tmax=3, // limits of balls' filling and inversions
     step2=(r2-r1)÷(balls-1); // steps between balls
```

Defines:
frames, used in chunk 58a.
r1, used in chunk 58b.

Then we open the file and put initialisation into it.

57d ⟨Hyperbolic inversion of a ball 57c⟩+≡ (52c) ◁57c 57e▷
```
  ofstream asymptote("ball-inv-d.asy");
  asymptote << setprecision(2);
  const numeric scale=2.5; //size of the picture
  asymptote << "scale = " << scale << ";" << endl;
```

Defines:
numeric, used in chunks 5, 6f, 15, 26e, 28a, 29b, 50–52, 54a, 55d, 58a, 59d, 61c, 64c, 66–70, 74–76, 78–80, 84–86, 90–101, 103, and 105–107.

We have one cycle which will inverted by the matrix T.

57e ⟨Hyperbolic inversion of a ball 57c⟩+≡ (52c) ◁57d 58a▷
```
     matrix T=matrix(2, 2, lst{dirac_ONE(), -t*e.subs(mu_subs), t*e.subs(mu_subs), dirac_ONE()});
     const cycle2D Hyp=cycle2D(lst{0,0},e, a).matrix_similarity(T);
```

Uses cycle2D 9a 9b 15c 15c 15d 15d 54b 55c 61b 62d 62d 62d 62d 62d 77b 77b 77b 77b 77b 89b 89b 89b 89b 92c, matrix 11d 16b 16c, matrix_similarity 7c, and subs 4b.

We run a cycle for different frames, the parameter t from the matrix T get specific values.

58a ⟨Hyperbolic inversion of a ball 57c⟩+≡ (52c) ◁57e 58b▷

```
for (int j=0; j≤2*frames ;j++ ) {
 double tval=(j≡0 ∧ j≡2*frames ? 0 :
    (j≡frames ? 10000000 :
     ex_to<numeric>((j<frames ? exp(tmin+j*(tmax-tmin)÷(frames-2)) :
                 -GiNaC::exp(tmin+(2*frames-j)*(tmax-tmin)÷(frames-2))).evalf()).to_double()));
```

Uses frames 57c and numeric 14a 57d.

Then we run a cycle over different hyperbolas filling up the ball. Two copies are drown for GIF and PDF images.

58b ⟨Hyperbolic inversion of a ball 57c⟩+≡ (52c) ◁58a 58c▷

```
 for (int i=0; i <balls; i++) {
    Hyp.subs(lst{sign≡1, a≡GiNaC::pow(r1+i*step2,2), t≡tval}).asy_draw(asymptote, "pa",
  -scale, scale, -scale, scale, lst{0.1+0.8*i÷balls, 0, 0.9-0.8*i÷balls});
    Hyp.subs(lst{sign≡1, a≡GiNaC::pow(r1+i*step2,2), t≡tval}).asy_draw(asymptote, "pb",
  -scale, scale, -scale, scale, lst{0.1+0.8*i÷balls, 0, 0.9-0.8*i÷balls});
 }
```

Uses asy_draw 11a, r1 57c, and subs 4b.

The boundary of the ball is drown in a highlighted way.

58c ⟨Hyperbolic inversion of a ball 57c⟩+≡ (52c) ◁58b 58d▷

```
 Hyp.subs(lst{sign≡1, a≡1, t≡tval}).asy_draw(asymptote, "pa",
  -scale, scale, -scale, scale, lst{1,0,0},"2pt");
 Hyp.subs(lst{sign≡1, a≡1, t≡tval}).asy_draw(asymptote, "pb",
  -scale, scale, -scale, scale, lst{1,0,0},"2pt");
 asymptote ≪ "newpic();" ≪ endl ≪ endl ;
}
```

Uses asy_draw 11a and subs 4b.

Finally we close the file.

58d ⟨Hyperbolic inversion of a ball 57c⟩+≡ (52c) ◁58c

```
asymptote.close();
```

Appendix E. 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.

E.1. **Cycle and cycle2D classes header files.**

E.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.

58e ⟨cycle.h 58e⟩≡ 59a▷

```
⟨license 110⟩
#include <stdexcept>
#include <ostream>
#include <sstream>

#include <ginac/ginac.h>

namespace MoebInv {
using namespace std;
using namespace GiNaC;
```

Defines:
MoebInv, used in chunks 13a, 59c, 64a, and 109c.

We may need to verify GiNaCversion, e.g. for paravector formalism (see Rem. 1.1 for required GiNaC version).

59a ⟨cycle.h 58e⟩+≡ ◁58e 59b▷

```
#define GINAC_VERSION_ATLEAST( major, minor, micro) \
   (GINACLIB_MAJOR_VERSION > major \
   ∨ (GINACLIB_MAJOR_VERSION ≡ major ∧ GINACLIB_MINOR_VERSION > minor) \
    ∨ (GINACLIB_MAJOR_VERSION ≡ major ∧ GINACLIB_MINOR_VERSION ≡ minor ∧ GINAC-
LIB_MICRO_VERSION ≥ micro))
```

Defines:
GINAC_VERSION_ATLEAST, used in chunks 13d, 15a, 36, 60c, 64c, 66c, 88, and 103c.

We define version number for our own library. For the change log see the file for companion library figure [20].

59b ⟨cycle.h 58e⟩+≡ ◁59a 59c▷

```
#define MOEBINV_MAJOR_VERSION 3
#define MOEBINV_MINOR_VERSION 0
```

Defines:
MOEBINV_MAJOR_VERSION, never used.
MOEBINV_MINOR_VERSION, never used.

The brief outline of the header file.

59c ⟨cycle.h 58e⟩+≡ ◁59b

```
⟨Auxiliary functions headers 59d⟩
⟨cycle class 60b⟩
⟨cycle2D class 61b⟩
⟨paravector class 63a⟩

} // namespace MoebInv
```

Uses MoebInv 58e.

E.1.2. *Some auxillary functions.* Here is the list of some auxiliary functions which are defined and used in the `cycle.h`. There are few additional functions we need.

59d ⟨Auxiliary functions headers 59d⟩≡ (59c) 60a▷

```
/** Check of equality of two expression and report the string */
const string equality(const ex & E);
inline const string equality(const ex & E1, const ex & E2) { return equality(E1-E2);}
inline const string 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 string eph_case(const numeric & sign);

/** Return even (real) part of a Clifford number */
ex scalar_part(const ex & e);

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

DECLARE_FUNCTION_1P(jump_fnct)
```

Defines:
jump_fnct, used in chunks 14b, 15a, 21, 22, 25, 26, 36, 51d, 79c, 90a, and 105–107.
string, used in chunks 10b, 11a, 16f, 18a, and 92c.
Uses ex 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, normal 4b, numeric 14a 57d, and subs 4b.

We often need a Clifford valued matrix which represent group of invertible matrices with real, complex or hypercomplex entries. The first two functions below produce a Clifford valued matrix from a real valued one. The last two functions produce a Clifford valued matrix from a pair of real matrix in a way which preserves multiplication of complex, dual or double numbers.

60a ⟨Auxiliary functions headers 59d⟩+≡ (59c) ◁59d

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

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 & M1, const ex & M2, const ex & e, bool not_inverse=true);

matrix sl2_clifford(const ex & a1, const ex & b1, const ex & c1, const ex & d1,
                    const ex & a2, const ex & b2, const ex & c2, const ex & d2,
                    const ex & e, bool not_inverse=true);
```

Uses bool 16a, ex 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, and matrix 11d 16b 16c.

E.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

60b ⟨cycle class 60b⟩≡ (59c)

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

 ⟨cycle class constructors 3a⟩
 ⟨service functions for class cycle 60c⟩
 ⟨accessing the data of a cycle 3e⟩
 ⟨specific methods of the class cycle 5c⟩
 ⟨Linear operation as cycle methods 4d⟩

protected:
 ex unit; // A Clifford unit to store the dimensionality and metric of the point space
 ex k;
 ex l;
 ex m;
};
 GINAC_DECLARE_UNARCHIVER(cycle);

 ⟨Linear operation on cycles 5a⟩
```

Uses cycle 3a 3a 3b 3b 3c 3d 5a 5a 5a 5a 5a 72c 72d 72d 73a 75d 75d 77a 77a 77a 77a 77a, ex 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, k 3a, l 3a, and m 3a.

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

60c ⟨service functions for class cycle 60c⟩≡ (60b) 60d▷

```
#if GINAC_VERSION_ATLEAST(1,5,0)
 void archive(archive_node &n) const;
 void read_archive(const archive_node &n, lst &sym_lst);
 return_type_t return_type_tinfo() const;
#endif
```

Uses GINAC_VERSION_ATLEAST 59a 59a.

Real and imaginary part of the representing vector.

60d ⟨service functions for class cycle 60c⟩+≡ (60b) ◁60c 61a▷

```
ex real_part() const;
ex imag_part() const;
inline ex evalf() const { return cycle(k.evalf(), l.evalf(), m.evalf(), unit);}
```

Uses cycle 3a 3a 3b 3b 3c 3d 5a 5a 5a 5a 5a 72c 72d 72d 73a 75d 75d 77a 77a 77a 77a 77a, ex 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, k 3a, l 3a, and m 3a.

Printing of cycles.

61a ⟨service functions for class cycle 60c⟩+≡ (60b) ◁60d

```
protected:
 void do_print(const print_dflt & c, unsigned level) const;
// void do_print_python(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;
```

E.1.4. *The derived class* **cycle2D** *for two dimensional cycles.* We derive a class **cycle2D** from **cycle** in order to add some more methods which only make sense in two dimensions.

61b ⟨cycle2D class 61b⟩≡ (59c)

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

 ⟨constructors of the class cycle2D 9a⟩
 ⟨methods specific for class cycle2D 9e⟩
 ⟨duplicated methods for class cycle2D 61c⟩
};
GINAC_DECLARE_UNARCHIVER(cycle2D);

 ⟨duplicated linear operation on cycle2D 62d⟩
```

Defines:
`cycle2D`, used in chunks 9, 10c, 16–23, 25, 26e, 28a, 30–33, 35a, 36, 50a, 51d, 53, 55–57, 61, 62, 64, 66d, 88–91, 93b, 94a, 96, 100, and 102b.
Uses `cycle` 3a 3a 3b 3b 3c 3d 5a 5a 5a 5a 5a 72c 72d 72d 73a 75d 75d 77a 77a 77a 77a 77a.

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

61c ⟨duplicated methods for class cycle2D 61c⟩≡ (61b) 61d▷

```
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 ex & sign = (new tensdelta)->setflag(status_flags::dynallocated),
                            const ex & D = 1, bool fix_paravector = true) const {
    return ex_to<cycle2D>(inherited::normalize_det(e, sign, D, fix_paravector)); }
inline cycle2D normalize_norm(const ex & e = 0,
                             const ex & sign = (new tensdelta)->setflag(status_flags::dynallocated),
                             const ex & N = 1, bool fix_paravector = true) const {
    return ex_to<cycle2D>(inherited::normalize_norm(e, sign, N, fix_paravector)); }
```

Uses `bool` 16a, `cycle2D` 9a 9b 15c 15c 15d 15d 54b 55c 61b 62d 62d 62d 62d 62d 77b 77b 77b 77b 77b 89b 89b 89b 89b 92c, `ex` 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, `normalize` 5e, `normalize_det` 5c, `normalize_norm` 5d, `numeric` 14a 57d, and `subs` 4b.

We duplicate the $\mathrm{SL}_2(\mathbb{R})$ similarity methods as well.

61d ⟨duplicated methods for class cycle2D 61c⟩+≡ (61b) ◁61c 62a▷

```
inline cycle2D 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 ex & sign_inv = (new tensdelta)->setflag(status_flags::dynallocated)) const {
       return ex_to<cycle2D>(inherited::sl2_similarity(a, b, c, d, e, sign, not_inverse, sign_inv)); }
```

Defines:
`sl2_similarity`, used in chunks 12a, 16–18, 23c, 33b, 86, 90, and 91.
Uses `bool` 16a, `cycle2D` 9a 9b 15c 15c 15d 15d 54b 55c 61b 62d 62d 62d 62d 62d 77b 77b 77b 77b 77b 89b 89b 89b 89b 92c, and `ex` 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c.

To separate calls with one or two matrices we provide various templates.

62a ⟨duplicated methods for class cycle2D 61c⟩+≡ (61b) ◁61d 62b▷
```
inline cycle2D sl2_similarity(const ex & M) const {
    return ex_to<cycle2D>(inherited::sl2_similarity(M)); }
cycle2D sl2_similarity(const ex & M, const ex & e) const;
cycle2D sl2_similarity(const ex & M, const ex & e, const ex & sign) const;
inline cycle2D sl2_similarity(const ex & M, const ex & e, const ex & sign, bool not_inverse,
    const ex & sign_inv = (new tensdelta)→setflag(status_flags::dynallocated)) const {
    return ex_to<cycle2D>(inherited::sl2_similarity(M, e, sign, not_inverse, sign_inv)); }
```
Defines:
sl2_similarity, used in chunks 12a, 16–18, 23c, 33b, 86, 90, and 91.
Uses bool 16a, cycle2D 9a 9b 15c 15c 15d 15d 54b 55c 61b 62d 62d 62d 62d 62d 77b 77b 77b 77b 77b 89b 89b 89b 89b 92c, and ex 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c.

Service methods in this class.

62b ⟨duplicated methods for class cycle2D 61c⟩+≡ (61b) ◁62a 62c▷
```
inline cycle2D normal() const { return cycle2D(k.normal(), l.normal(), m.normal(), unit.normal());}
inline cycle2D expand() const { return cycle2D(k.expand(), l.expand(), m.expand(), unit);}
inline ex evalf() const { return ex_to<cycle2D>(inherited::evalf());}
inline cycle2D subject_to(const ex & condition, const ex & vars = 0) const {
 return ex_to<cycle2D>(inherited::subject_to(condition, vars)); }

// cycle2D(const archive_node &n, lst &sym_lst);
 void archive(archive_node &n) const;
 // ex unarchive(const archive_node &n, lst &sym_lst);
 void read_archive(const archive_node &n, lst &sym_lst);
```

Uses cycle2D 9a 9b 15c 15c 15d 15d 54b 55c 61b 62d 62d 62d 62d 62d 77b 77b 77b 77b 77b 89b 89b 89b 89b 92c, ex 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, expand 4b, k 3a, l 3a, m 3a, normal 4b, and subject_to 6c.

Real and imaginary part of the representing vector.

62c ⟨duplicated methods for class cycle2D 61c⟩+≡ (61b) ◁62b
```
ex real_part() const;
ex imag_part() const;
```

Uses ex 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c.

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

62d ⟨duplicated linear operation on cycle2D 62d⟩≡ (61b)
```
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 9, 10c, 16–23, 25, 26e, 28a, 30–33, 35a, 36, 50a, 51d, 53, 55–57, 61, 62, 64, 66d, 88–91, 93b, 94a, 96, 100, and 102b.
ex, used in chunks 3–11, 14c, 16–32, 34–36, 53b, 59–63, 65a, 67–76, 78–80, 82–93, 96c, 97a, and 103–109.
Uses operator* 5a, operator+ 5a, operator- 5a, and operator/ 5a.

E.1.5. *Paravector class.* This is the definition of a technical class which wraps indexed objects to works as paravectors (see Rem. 1.1 for required GiNaC version). More precisely, for an n-tuple x_μ, $\mu = 0, \ldots, n-1$ the vector formalism associate the element $x_\mu e_\mu$ (Einstein summation notation) of the Clifford algebra $\mathcal{C}\ell(n)$. In the paravector formalism an n-tuple x_ν, $\nu = 0, \ldots, n-1$ is associated to the element $x_0 \cdot \mathbf{1} + x_{\nu-1}e_\nu$ of the Clifford algebra $\mathcal{C}\ell(n-1)$. Besides the smaller dimensionality the main advantage of the paravector formalism in two dimensions is commutativity of the Clifford algebras $\mathcal{C}\ell(1,0,0)$, $\mathcal{C}\ell(0,1,0)$ and $\mathcal{C}\ell(0,0,1)$ which are isomorphic to complex, dual and double numbers respectively.

GiNaC does not recognise dummy index summation in the expressions of the form $x_{\nu-1}e_\nu$. The present class *paravector* allows to wrap for GiNaC the paravector $x_0 \cdot \mathbf{1} + x_{\nu-1}e_\nu$ as $x_\mu \tilde{e}_\mu$ in the method *paravector::eval_indexed*(). Here is the formal part of its definition.

63a ⟨paravector class 63a⟩≡ (59c) 63b▷

```
class paravector : public basic
{
GINAC_DECLARE_REGISTERED_CLASS(paravector, basic)

public:
    paravector(const ex & b);
    void archive(archive_node &n) const;
    void read_archive(const archive_node &n, lst &sym_lst);
    return_type_t return_type_tinfo() const;
    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;
    size_t nops(size_t i) const {return 1;}
    ex op(size_t i) const;
    ex & let_op(size_t i);
    ex subs(const ex & e, unsigned options = 0) const;
    ex subs(const exmap & m, unsigned options = 0) const override;
```

Defines:
paravector, used in chunks 13d, 16–18, 24a, 28b, 32–35, 64b, 66b, 67c, 70, 83a, 104, and 105b.
Uses ex 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, let_op 4b, m 3a, nops 4b, op 4b, and subs 4b.

This is the only non-formal method in the class *paravector*, it evaluates if the shifted indexes $\mu \to \mu + 1$ leads to any particular evaluation.

63b ⟨paravector class 63a⟩+≡ (59c) ◁63a 63c▷

```
    ex eval_indexed(const basic & i) const;
```

Uses ex 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c.

Here is the only member of the class.

63c ⟨paravector class 63a⟩+≡ (59c) ◁63b

```
protected:
    ex vector;
};
GINAC_DECLARE_UNARCHIVER(paravector);
```

Defines:
paravector, used in chunks 13d, 16–18, 24a, 28b, 32–35, 64b, 66b, 67c, 70, 83a, 104, and 105b.
Uses ex 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c.

E.2. **Implementation of the cycle class.** We start from definitions of constructors in **cycle** class

64a ⟨cycle.cpp 64a⟩≡ 64b▷
```
⟨license 110⟩
#include <cycle.h>
namespace MoebInv {
using namespace std;
using namespace GiNaC;

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

Defines:
PRINT_CYCLE, used in chunk 75d.
Uses cycle 3a 3a 3b 3b 3c 3d 5a 5a 5a 5a 5a 72c 72d 72d 73a 75d 75d 77a 77a 77a 77a 77a, k 3a, l 3a, m 3a, and MoebInv 58e.

Macros for implementation of new classes

64b ⟨cycle.cpp 64a⟩+≡ ◁64a 64c▷
```
 GINAC_IMPLEMENT_REGISTERED_CLASS_OPT(cycle, basic,
        print_func<print_dflt>(&cycle::do_print).
                    //          print_func<print_python>(&cycle::do_print_python).
        print_func<print_latex>(&cycle::do_print_latex))

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

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

Uses cycle 3a 3a 3b 3b 3c 3d 5a 5a 5a 5a 5a 72c 72d 72d 73a 75d 75d 77a 77a 77a 77a 77a, cycle2D 9a 9b 15c 15c 15d 15d 54b 55c 61b 62d 62d 62d 62d 62d 77b 77b 77b 77b 77b 89b 89b 89b 89b 92c, and paravector 63a 63c 103c 103c 103c 104d 104d 105a.

tinfo is an important part of class definitions

64c ⟨cycle.cpp 64a⟩+≡ ◁64b 65a▷
```
#if GINAC_VERSION_ATLEAST(1,5,0)
return_type_t cycle::return_type_tinfo() const
{
   if (is_a<numeric>(get_dim()))
      switch (ex_to<numeric>(get_dim()).to_int()) {
      case 2:
         return make_return_type_t<cycle2D>();
      default:
      return make_return_type_t<cycle>();
      }
   else
      return make_return_type_t<cycle>();
}
#endif
cycle::cycle() : unit(), k(), l(), m()
{
#if GINAC_VERSION_ATLEAST(1,5,0)
#else
 tinfo_key = &cycle::tinfo_static;
#endif
}
```

Uses cycle 3a 3a 3b 3b 3c 3d 5a 5a 5a 5a 5a 72c 72d 72d 73a 75d 75d 77a 77a 77a 77a 77a, cycle2D 9a 9b 15c 15c 15d 15d 54b 55c 61b 62d 62d 62d 62d 62d 77b 77b 77b 77b 77b 89b 89b 89b 89b 92c, get_dim 3e, GINAC_VERSION_ATLEAST 59a 59a, k 3a, l 3a, m 3a, and numeric 14a 57d.

E.2.1. *Main constructor of cycle from all parameters given.* If all parameters of the cycle are given this constructor is used.

65a ⟨cycle.cpp 64a⟩+≡ ◁64c 65b▷

```
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` 3a 3a 3b 3b 3c 3d 5a 5a 5a 5a 5a 72c 72d 72d 73a 75d 75d 77a 77a 77a 77a 77a, `ex` 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, `k` 3a, `m` 3a, and `metr` 3a.

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

65b ⟨cycle.cpp 64a⟩+≡ ◁65a 65c▷

```
    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<varidx>(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` 3a 3a 3b 3b 3c 3d 5a 5a 5a 5a 5a 72c 72d 72d 73a 75d 75d 77a 77a 77a 77a 77a, `get_dim` 3e, `l` 3a, `matrix` 11d 16b 16c, `nops` 4b, and `varidx` 14a 15a 15b.

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

65c ⟨cycle.cpp 64a⟩+≡ ◁65b 65d▷

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

Uses `cycle` 3a 3a 3b 3b 3c 3d 5a 5a 5a 5a 5a 72c 72d 72d 73a 75d 75d 77a 77a 77a 77a 77a, `is_zero` 4b, and `matrix` 11d 16b 16c.

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

65d ⟨cycle.cpp 64a⟩+≡ ◁65c 66a▷

```
  if (is_a<clifford>(metr)) {
     if (D.is_zero())
        D = ex_to<varidx>(metr.op(1)).get_dim();
     unit =metr;
  } else {
     if (D.is_zero()) {
        if (is_a<indexed>(metr))
           D = ex_to<varidx>(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<varidx>(indices[0]).get_dim();
        }
     }
```

Uses `get_dim` 3e, `is_zero` 4b, `matrix` 11d 16b 16c, `metr` 3a, `op` 4b, and `varidx` 14a 15a 15b.

For metric of unknown type we throw an exception.

66a ⟨cycle.cpp 64a⟩+≡ ◁65d 66b▷
```
    if (D.is_zero())
       throw(std::invalid_argument("cycle::cycle(): the metric should be either tensor, "
                              "matrix, Clifford unit or indexed by two indices. "
                              "Otherwise supply the through the second parameter."));
```

Uses `cycle` 3a 3a 3b 3b 3c 3d 5a 5a 5a 5a 5a 72c 72d 72d 73a 75d 75d 77a 77a 77a 77a 77a, `is_zero` 4b, and `matrix` 11d 16b 16c.

Now we try to build the Clifford unit either for vector or paravector formalism.

66b ⟨cycle.cpp 64a⟩+≡ ◁66a 66c▷
```
    try {
       unit = clifford_unit(varidx((new symbol)→setflag(status_flags::dynallocated), D), metr);
    } catch (std::exception &p) {
       try {
          unit = clifford_unit(varidx((new symbol)→setflag(status_flags::dynallocated), D-1), metr);
       } catch (std::exception &p1) {
         throw(std::invalid_argument("cycle::cycle(): the metricis not suitable for both vector "
                                 "and paravector formalism"));
       }
    }
  }
```

Uses `catch` 37a 37b, `cycle` 3a 3a 3b 3b 3c 3d 5a 5a 5a 5a 5a 72c 72d 72d 73a 75d 75d 77a 77a 77a 77a 77a, `metr` 3a, `paravector` 63a 63c 103c 103c 103c 104d 104d 105a, and `varidx` 14a 15a 15b.

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

66c ⟨cycle.cpp 64a⟩+≡ ◁66b 67a▷
```
  #if GINAC_VERSION_ATLEAST(1,5,0)
  #else
   ⟨Set tinfo to dimension 66d⟩
  #endif
  }
```

Uses `GINAC_VERSION_ATLEAST` 59a 59a.

We set tinfo key for cycle according to its dimension

66d ⟨Set tinfo to dimension 66d⟩≡ (66c)
```
  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` 3a 3a 3b 3b 3c 3d 5a 5a 5a 5a 5a 72c 72d 72d 73a 75d 75d 77a 77a 77a 77a 77a, `cycle2D` 9a 9b 15c 15c 15d 15d 54b 55c 61b 62d 62d 62d 62d 62d 77b 77b 77b 77b 77b 89b 89b 89b 89b 92c, and `numeric` 14a 57d.

E.2.2. *Specific cycle constructors.* Constructor for cycle with the given determinant *r_squared*, e.g. zero-radius cycle by default.

67a ⟨cycle.cpp 64a⟩+≡ ◁66c 67b▷
```
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.radius_sq(e, sign) == r_squared}, lst{m_temp});
}
```

Uses cycle 3a 3a 3b 3b 3c 3d 5a 5a 5a 5a 5a 72c 72d 72d 73a 75d 75d 77a 77a 77a 77a 77a, ex 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, l 3a, metr 3a, numeric 14a 57d, radius_sq 6f, and subject_to 6c.

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

67b ⟨cycle.cpp 64a⟩+≡ ◁67a 67c▷
```
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 3a 3a 3b 3b 3c 3d 5a 5a 5a 5a 5a 72c 72d 72d 73a 75d 75d 77a 77a 77a 77a 77a, ex 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, get_k 3e, get_l 4a, get_m 4a, is_zero 4b, and metr 3a.

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

67c ⟨cycle.cpp 64a⟩+≡ ◁67b 67d▷
```
cycle::cycle(const matrix & M, const ex & metr, const ex & e, const ex & sign, const ex & dim)
{
   ⟨Create a Clifford unit 69b⟩
   ex M1=M;
   bool is_vector=(dim==0 || dim==D);
   ex Dsp=is_vector? D:dim;

   // Expensive checks, if this conditions are not satisfied,
   // corresponding errors will be generated later by the constructor
   /*
   if (is_vector &&
      not (M.rows() == 2 && M.cols() == 2 && (M.op(0)+M.op(3)).normal().is_zero()))
    throw(std::invalid_argument("cycle::cycle(): in vector formalism the second argument should be "
      "square 2x2 matrix with M(1,1)=-M(2,2)"));

   if (not is_vector &&
      not (M.rows() == 2 && M.cols() == 2 && (M.op(0)+clifford_bar(M.op(3))).normal().is_zero()))
    throw(std::invalid_argument("cycle::cycle(): in paravector formalism the second argument should"
      " be square 2x2 matrix with M(1,1)=-bar(M)(2,2)")); */
```

Uses bool 16a, cycle 3a 3a 3b 3b 3c 3d 5a 5a 5a 5a 5a 72c 72d 72d 73a 75d 75d 77a 77a 77a 77a 77a, ex 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, is_zero 4b, matrix 11d 16b 16c, metr 3a, normal 4b, op 4b, and paravector 63a 63c 103c 103c 103c 104d 104d 105a.

It may happen, that the scalar part extracted from matrix is equal to zero and we need to append it manually.

67d ⟨cycle.cpp 64a⟩+≡ ◁67c 68b▷
```
if (sign.is_zero()) {
   try {
      lst l0=ex_to<lst>(clifford_to_lst(M.op(0), e1));
      ⟨fixing the size of the list 68a⟩
```

Uses is_zero 4b and op 4b.

68a ⟨fixing the size of the list 68a⟩≡ (67–69)

```
if (l0.nops()<Dsp) {
    lst l1=lst{0};
    for (auto & x: l0)
        l1.append(x);
    l0=l1;
}
```

Uses nops 4b.

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

68b ⟨cycle.cpp 64a⟩+≡ ◁67d 68c▷

```
            (*this) = cycle(remove_dirac_ONE(M.op(2)), l0, (is_vector?1:-1)*remove_dirac_ONE(M.op(1)), metr);
        } catch  (std::exception &p) {
            lst l0=ex_to<lst>(clifford_to_lst(M.op(0)*clifford_inverse(M.op(2)), e1));
            ⟨fixing the size of the list 68a⟩
            (*this) = cycle(numeric(1), l0,
                        (is_vector?1:-1)*canonicalize_clifford(M.op(1)*clifford_inverse(M.op(2))), metr);
        }
    } else {
        varidx i0((new symbol)→setflag(status_flags::dynallocated), Dsp),
            i1((new symbol)→setflag(status_flags::dynallocated), Dsp, true);

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

Uses catch 37a 37b, cycle 3a 3a 3b 3b 3c 3d 5a 5a 5a 5a 5a 72c 72d 72d 73a 75d 75d 77a 77a 77a 77a 77a, ex 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, metr 3a, numeric 14a 57d, op 4b, and varidx 14a 15a 15b.

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

68c ⟨cycle.cpp 64a⟩+≡ ◁68b 68d▷

```
        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<varidx>(sign_m.op(1)).get_dim() ≡ Dsp)
                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<varidx>(ind[0]).get_dim() ≡ Dsp) ∧ (ex_to<varidx>(ind[1]).get_dim() ≡ Dsp))
                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"));
```

Uses cycle 3a 3a 3b 3b 3c 3d 5a 5a 5a 5a 5a 72c 72d 72d 73a 75d 75d 77a 77a 77a 77a 77a, get_dim 3e, get_metric 3e, op 4b, subs 4b, and varidx 14a 15a 15b.

The sign given as a matrix is oftenly used.

68d ⟨cycle.cpp 64a⟩+≡ ◁68c 69a▷

```
        } else if (is_a<matrix>(sign_m)) {
            if ((ex_to<matrix>(sign_m).cols() ≡ Dsp) ∧ (ex_to<matrix>(sign_m).rows() ≡ Dsp))
                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 3a 3a 3b 3b 3c 3d 5a 5a 5a 5a 5a 72c 72d 72d 73a 75d 75d 77a 77a 77a 77a 77a and matrix 11d 16b 16c.

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

69a ⟨cycle.cpp 64a⟩+≡ ◁68d 70a▷

```
    try {
        lst l0=ex_to<lst>(clifford_to_lst(M.op(0), e1));
        ⟨fixing the size of the list 68a⟩
        (*this) = cycle(remove_dirac_ONE(M.op(2)), indexed(matrix(1, ex_to<numeric>(Dsp).to_int(),
                                                 l0), i0.toggle_variance())*conv, (is_vector?1:-
  1)*remove_dirac_ONE(M.op(1)), metr);
    } catch  (std::exception &p) {
        lst l0=ex_to<lst>(clifford_to_lst(M.op(0)*clifford_inverse(M.op(2)), e1));
        ⟨fixing the size of the list 68a⟩
      (*this) = cycle(numeric(1), indexed(matrix(1, ex_to<numeric>(Dsp).to_int(), l0), i0.toggle_variance())*conv,
                  (is_vector?1:-1)*canonicalize_clifford(M.op(1)*clifford_inverse(M.op(2))), metr);
    }
  }
  }
```

Uses catch 37a 37b, cycle 3a 3a 3b 3b 3c 3d 5a 5a 5a 5a 5a 72c 72d 72d 73a 75d 75d 77a 77a 77a 77a 77a, matrix 11d 16b 16c, metr 3a, numeric 14a 57d, and op 4b.

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

69b ⟨Create a Clifford unit 69b⟩≡ (67c)

```
  ex e1, D=dim;
  if (e.is_zero()) {
    if (is_a<clifford>(metr)) {
      D=ex_to<varidx>(metr.op(1)).get_dim();
      e1=metr;
    } else {
      ex metr1;
      if (is_a<matrix>(metr)) {
        D = ex_to<matrix>(metr).cols();
        metr1 = metr;
      } else if (is_a<indexed>(metr)) {
        D = ex_to<varidx>(ex_to<indexed>(metr).get_indices()[0]).get_dim();
        metr1 = metr;
      } else
      throw(std::invalid_argument("cycle(): 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 {
    if (¬ is_a<clifford>(e))
      throw(std::invalid_argument("cycle(): if e is supplied, it shall be a Clifford unit"));
    e1 = e;
    D = ex_to<varidx>(e.op(1)).get_dim();
  }
```

Uses cycle 3a 3a 3b 3b 3c 3d 5a 5a 5a 5a 5a 72c 72d 72d 73a 75d 75d 77a 77a 77a 77a 77a, ex 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, get_dim 3e, is_zero 4b, matrix 11d 16b 16c, metr 3a, op 4b, and varidx 14a 15a 15b.

E.2.3. *Class* **cycle** *members access.* We append paravector formalism values to Clifford unit values.

70a ⟨cycle.cpp 64a⟩+≡ ◁69a 70b▷
```
ex expand_paravector_metric(const ex & unit) {
   int D=ex_to<numeric>(ex_to<idx>(unit.get_free_indices()[0]).get_dim()).to_int();
   matrix M=ex_to<matrix>(unit_matrix(D+1));
   M(0,0)=numeric(-1);
   for (int i=0; i<D; ++i)
      for (int j=0; j<D; ++j)
         M(i+1,j+1)=ex_to<clifford>(unit).get_metric(i,j);
   return indexed(M, varidx((new symbol)->setflag(status_flags::dynallocated), D+1),
                  varidx((new symbol)->setflag(status_flags::dynallocated), D+1));
}
```

Uses `ex` 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, `get_dim` 3e, `get_metric` 3e, `matrix` 11d 16b 16c, `numeric` 14a 57d, and `varidx` 14a 15a 15b.

70b ⟨cycle.cpp 64a⟩+≡ ◁70a 70c▷
```
ex cycle::get_metric() const {
   if (ex_to<idx>(unit.op(1)).get_dim() == get_dim())
      return ex_to<clifford>(unit).get_metric();
   else if (is_a<numeric>(get_dim())) {
      return expand_paravector_metric(unit);
   } else
    throw(std::runtime_error("cycle::get_metric(): cannot return metric for paravector formalism "
                         "with symbolic dimensions"));
}
```

Uses `cycle` 3a 3a 3b 3b 3c 3d 5a 5a 5a 5a 5a 72c 72d 72d 73a 75d 75d 77a 77a 77a 77a 77a, `ex` 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, `get_dim` 3e, `get_metric` 3e, `numeric` 14a 57d, `op` 4b, and `paravector` 63a 63c 103c 103c 103c 104d 104d 105a.

Similar procedure for specific indices.

70c ⟨cycle.cpp 64a⟩+≡ ◁70b 70d▷
```
ex cycle::get_metric(const ex &i0, const ex &i1) const {
   if (ex_to<idx>(unit.op(1)).get_dim() == get_dim())
      return ex_to<clifford>(unit).get_metric(i0, i1);
```

Uses `cycle` 3a 3a 3b 3b 3c 3d 5a 5a 5a 5a 5a 72c 72d 72d 73a 75d 75d 77a 77a 77a 77a 77a, `ex` 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, `get_dim` 3e, `get_metric` 3e, and `op` 4b.

We avoid calculations of unnecessary elements if only one value is requested.

70d ⟨cycle.cpp 64a⟩+≡ ◁70c 71a▷
```
   else if (is_a<idx>(i0) && ex_to<idx>(i0).is_numeric() &&
            is_a<idx>(i1) && ex_to<idx>(i1).is_numeric()) {
      int j0= ex_to<numeric>(ex_to<idx>(i0).get_value()).to_int(),
         j1= ex_to<numeric>(ex_to<idx>(i1).get_value()).to_int();
      if ( j0 > 0 && j1 > 0)
         return ex_to<clifford>(unit).get_metric(varidx(j0-1,get_dim()-1), varidx(j1-1,get_dim()-1));
      else if ( j0 == 0 && j1 == 0)
         return -numeric(1);
      else
         return 0;
   } else if (is_a<numeric>(get_dim())) {
      ex metr=expand_paravector_metric(unit);
      return metr.subs(lst{metr.op(1)==i0,metr.op(2)==i1});
   } else
    throw(std::runtime_error("cycle::get_metric(): cannot return metric for paravector formalism "
                         "with symbolic dimensions"));
 }
```

Uses `cycle` 3a 3a 3b 3b 3c 3d 5a 5a 5a 5a 5a 72c 72d 72d 73a 75d 75d 77a 77a 77a 77a 77a, `ex` 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, `get_dim` 3e, `get_metric` 3e, `metr` 3a, `numeric` 14a 57d, `op` 4b, `paravector` 63a 63c 103c 103c 103c 104d 104d 105a, `subs` 4b, and `varidx` 14a 15a 15b.

Class **cycle** has four operands.

71a ⟨cycle.cpp 64a⟩+≡ ◁70d 71b▷

```
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` 3a 3a 3b 3b 3c 3d 5a 5a 5a 5a 5a 72c 72d 72d 73a 75d 75d 77a 77a 77a 77a 77a, `ex` 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, `k` 3a, `l` 3a, `m` 3a, `nops` 4b, and `op` 4b.

Operands may be set through this method.

71b ⟨cycle.cpp 64a⟩+≡ ◁71a 72a▷

```
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::let_op(): requested operand out of the range (4)"));
 }
}
```

Uses `cycle` 3a 3a 3b 3b 3c 3d 5a 5a 5a 5a 5a 72c 72d 72d 73a 75d 75d 77a 77a 77a 77a 77a, `ex` 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, `k` 3a, `l` 3a, `let_op` 4b, `m` 3a, and `nops` 4b.

Substitutions works as usual in GiNaC.

72a ⟨cycle.cpp 64a⟩+≡ ◁71b 72b▷

```
cycle cycle::subs(const ex & e, unsigned options) const
{
    exmap em;
    if (e.info(info_flags::list)) {
        lst l = ex_to<lst>(e);
        for (const auto & i : l)
            em.insert(std::make_pair(i.op(0), i.op(1)));
    } else if (is_a<relational>(e))
        em.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 cycle(k.subs(em, options),l.subs(em, options),m.subs(em, options),unit.subs(em, options));
}
```

Uses `cycle` 3a 3a 3b 3b 3c 3d 5a 5a 5a 5a 5a 72c 72d 72d 73a 75d 75d 77a 77a 77a 77a 77a, `ex` 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, `k` 3a, `l` 3a, `m` 3a, `op` 4b, and `subs` 4b.

E.2.4. *Service methods for the GiNaC infrastructure.* Standard parts involving archiving, comparison and printing of the **cycle** class

72b ⟨cycle.cpp 64a⟩+≡ ◁72a 72c▷

Archiving routine.

72c ⟨cycle.cpp 64a⟩+≡ ◁72b 72d▷

```
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);
}
```

Defines:
`cycle`, used in chunks 4–9, 12a, 13a, 15–20, 22–26, 28e, 32–35, 53a, 60, 61b, 64–76, 78–80, 82–87, 89a, 91a, 93–95, and 97b.
Uses `k` 3a, `l` 3a, and `m` 3a.

Un-archiving routine.

72d ⟨cycle.cpp 64a⟩+≡ ◁72c 73a▷

```
void cycle::read_archive(const archive_node &n, lst &sym_lst)
{
    inherited::read_archive(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);
}
GINAC_BIND_UNARCHIVER(cycle);

//const char *cycle::get_class_name() { return "cycle"; }
```

Defines:
`cycle`, used in chunks 4–9, 12a, 13a, 15–20, 22–26, 28e, 32–35, 53a, 60, 61b, 64–76, 78–80, 82–87, 89a, 91a, 93–95, and 97b.
Uses `k` 3a, `l` 3a, and `m` 3a.

Comparison of **cycle**s.

73a ⟨cycle.cpp 64a⟩+≡ ◁72d 73b▷

```
int cycle::compare_same_type(const basic &other) const
{
    GINAC_ASSERT(is_a<cycle>(other));
    return inherited::compare_same_type(other);
÷*
  const cycle &o = static_cast<const cycle &>(other);
  if ((unit ≡ o.unit) ∧ (l*o.get_k() - o.get_l()*k).is_zero() ∧ (m*o.get_k() - o.get_m()*k).is_zero())
    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 4–9, 12a, 13a, 15–20, 22–26, 28e, 32–35, 53a, 60, 61b, 64–76, 78–80, 82–87, 89a, 91a, 93–95, and 97b.
Uses `get_k` 3e, `get_l` 4a, `get_m` 4a, `is_zero` 4b, `k` 3a, `l` 3a, and `m` 3a.

Equality of **cycle**s.

73b ⟨cycle.cpp 64a⟩+≡ ◁73a 74a▷

```
bool cycle::is_equal(const basic & other, bool projectively, bool ignore_unit) const
{
  if (not is_a<cycle>(other))
    return false;
  const cycle o = ex_to<cycle>(other);
  ex factor=0, ofactor=0;

  if (not (ignore_unit ∨ unit.is_equal(o.unit)))
    return false;

  if (projectively) {
    // Check that coefficients are scalar multiples of other
    if (not (m*o.get_k()-o.get_m()*k).normal().is_zero())
      return false;
    // Set up coefficients for proportionality
    if (get_k().normal().is_zero()) {
      factor=get_m();
      ofactor=o.get_m();
    } else {
      factor=get_k();
      ofactor=o.get_k();
    }

  } else
    // Check the exact equality of coefficients
    if (not ((get_k()-o.get_k()).normal().is_zero() ∧ (get_m()-o.get_m()).normal().is_zero()))
      return false;
```

Uses `bool` 16a, `cycle` 3a 3a 3b 3b 3c 3d 5a 5a 5a 5a 5a 72c 72d 72d 73a 75d 75d 77a 77a 77a 77a 77a, `ex` 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, `get_k` 3e, `get_m` 4a, `is_equal` 4b, `is_zero` 4b, `k` 3a, `m` 3a, and `normal` 4b.

Now we iterate through the coefficients of l.

74a ⟨cycle.cpp 64a⟩+≡ ◁73b 74b▷
```
if (is_a<numeric>(get_dim())) {
    int D = ex_to<numeric>(get_dim()).to_int();
    if (¬ (is_a<numeric>(o.get_dim()) ∧ D ≡ex_to<numeric>(o.get_dim()).to_int()))
        return false;

    for (int i=0; i<D; i++)
        if (projectively) {
            // search the the first non-zero coefficient
            if (factor.is_zero()) {
                factor=get_l(i);
                ofactor=o.get_l(i);
            } else
                if (¬ (get_l(i)∗ofactor-o.get_l(i)∗factor).normal().is_zero())
                    return false;
        } else
            if (¬ (get_l(i)-o.get_l(i)).normal().is_zero())
                return false;

    return true;
} else
    return (l∗ofactor-o.get_l()∗factor).normal().is_zero();
}
```

Uses get_dim 3e, get_l 4a, is_zero 4b, l 3a, normal 4b, and numeric 14a 57d.

We return a **lst** of equations, which describes the condition of the given **cycle** to be given by the same point of the projective space as *other*.

74b ⟨cycle.cpp 64a⟩+≡ ◁74a 74c▷
```
ex cycle::the_same_as(const basic & other) const
{
 if (¬ (is_a<cycle>(other) ∧ (get_dim() ≡ ex_to<cycle>(other).get_dim())))
    return lst{1≡0};
 ex f=1, f1=1;
 lst res;
```

Uses cycle 3a 3a 3b 3b 3c 3d 5a 5a 5a 5a 5a 72c 72d 72d 73a 75d 75d 77a 77a 77a 77a 77a, ex 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, and get_dim 3e.

If k is non-zero than we chose it as a normalizing factor.

74c ⟨cycle.cpp 64a⟩+≡ ◁74b 74d▷
```
 if (not k.is_zero()) {
  f = k;
  f1 = ex_to<cycle>(other).get_k();
  res.append(f1∗m ≡f∗ex_to<cycle>(other).get_m());
```

Uses cycle 3a 3a 3b 3b 3c 3d 5a 5a 5a 5a 5a 72c 72d 72d 73a 75d 75d 77a 77a 77a 77a 77a, get_k 3e, get_m 4a, is_zero 4b, k 3a, and m 3a.

Otherwise we try m for this.

74d ⟨cycle.cpp 64a⟩+≡ ◁74c 75a▷
```
 } else if (not m.is_zero()) {
  f = m;
  f1 = ex_to<cycle>(other).get_m();
 }
```

Uses cycle 3a 3a 3b 3b 3c 3d 5a 5a 5a 5a 5a 72c 72d 72d 73a 75d 75d 77a 77a 77a 77a 77a, get_m 4a, is_zero 4b, and m 3a.

And then we build equations equating coresopnding *l*s.

75a ⟨cycle.cpp 64a⟩+≡ ◁74d 75b▷
```
 if (ex_to<varidx>(unit.op(1)).is_numeric()) {
  int D = ex_to<numeric>(get_dim()).to_int();
  for (int i=0; i < D; ++i)
   res.append(f1*get_l(i)≡f*ex_to<cycle>(other).get_l(i));
 } else
  res.append(f1*l≡f*ex_to<cycle>(other).get_l());
 return res;
}
```

Uses cycle 3a 3a 3b 3b 3c 3d 5a 5a 5a 5a 5a 72c 72d 72d 73a 75d 75d 77a 77a 77a 77a 77a, get_dim 3e, get_l 4a, l 3a, numeric 14a 57d, op 4b, and varidx 14a 15a 15b.

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

75b ⟨cycle.cpp 64a⟩+≡ ◁75a 75c▷
```
bool cycle::is_zero() const
{
 return (k.is_zero() ∧ l.is_zero() ∧ m.is_zero());
}
```

Uses bool 16a, cycle 3a 3a 3b 3b 3c 3d 5a 5a 5a 5a 5a 72c 72d 72d 73a 75d 75d 77a 77a 77a 77a 77a, is_zero 4b, k 3a, l 3a, and m 3a.

Real and imaginary part of the representing vector.

75c ⟨cycle.cpp 64a⟩+≡ ◁75b 75d▷
```
ex cycle::real_part() const
{
    return cycle(k.real_part(),indexed(l.op(0).real_part(),l.op(1)),m.real_part(),unit);
}

ex cycle::imag_part() const
{
    return cycle(k.imag_part(),indexed(l.op(0).imag_part(),l.op(1)),m.imag_part(),unit);
}
```

Uses cycle 3a 3a 3b 3b 3c 3d 5a 5a 5a 5a 5a 72c 72d 72d 73a 75d 75d 77a 77a 77a 77a 77a, ex 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, k 3a, l 3a, m 3a, and op 4b.

Printing of **cycle**s.

75d ⟨cycle.cpp 64a⟩+≡ ◁75c 76▷
```
void cycle::do_print(const print_dflt & c, unsigned level) const
{
 PRINT_CYCLE
}

÷*void cycle::do_print_python(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 4–9, 12a, 13a, 15–20, 22–26, 28e, 32–35, 53a, 60, 61b, 64–76, 78–80, 82–87, 89a, 91a, 93–95, and 97b.
Uses PRINT_CYCLE 64a.

E.2.5. *Linear operation on cycles.* Here are linear operations on **cycle** defined as methods.

76 ⟨cycle.cpp 64a⟩+≡ ◁75d 77a▷

```
cycle cycle::add(const cycle & rh) const
{
   if (get_dim() ≠ rh.get_dim())
    throw(std::invalid_argument("cycle::add(): cannot add two cycles from diferent dimensions"));

   ex ln=indexed(((get_l().is_zero()?0:get_l().op(0))+(rh.get_l().is_zero()?0:rh.get_l().op(0))).evalm(),
              varidx((new symbol)→setflag(status_flags::dynallocated), get_dim()));
   return cycle(get_k()+rh.get_k(), ln, get_m()+rh.get_m(), unit);
}
cycle cycle::sub(const cycle & rh) const
{
   if (get_dim() ≠ rh.get_dim())
    throw(std::invalid_argument("cycle::add(): cannot subtract two cycles from diferent dimensions"));

   ex ln=indexed(((get_l().is_zero()?0:get_l().op(0))-(rh.get_l().is_zero()?0:rh.get_l().op(0))).evalm(),
              varidx((new symbol)→setflag(status_flags::dynallocated), get_dim()));
   return cycle(get_k()-rh.get_k(), ln, get_m()-rh.get_m(), unit);
}
cycle cycle::exmul(const ex & rh) const
{
   return cycle(get_k()*rh, indexed(get_l().is_zero() ? 0 : (get_l().op(0)*rh).evalm(),
                         varidx((new symbol)→setflag(status_flags::dynallocated), get_dim())),
           get_m()*rh, unit);
}
cycle cycle::div(const ex & rh) const
{
   return exmul(pow(rh, numeric(-1)));
}
```

Uses `add` 4d, `cycle` 3a 3a 3b 3b 3c 3d 5a 5a 5a 5a 5a 72c 72d 72d 73a 75d 75d 77a 77a 77a 77a 77a, `div` 4d, `ex` 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, `exmul` 4d, `get_dim` 3e, `get_k` 3e, `get_l` 4a, `get_m` 4a, `is_zero` 4b, `numeric` 14a 57d, `op` 4b, `sub` 4d, and `varidx` 14a 15a 15b.

The same linear structure is represented in operators overloading.

77a ⟨cycle.cpp 64a⟩+≡ ◁76 77b▷

```
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 4–9, 12a, 13a, 15–20, 22–26, 28e, 32–35, 53a, 60, 61b, 64–76, 78–80, 82–87, 89a, 91a, 93–95, and 97b.
`ex`, used in chunks 3–11, 14c, 16–32, 34–36, 53b, 59–63, 65a, 67–76, 78–80, 82–93, 96c, 97a, and 103–109.

Uses `add` 4d, `div` 4d, `exmul` 4d, `mul` 7a, `operator*` 5a, `operator+` 5a, `operator-` 5a, `operator/` 5a, and `sub` 4d.

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

77b ⟨cycle.cpp 64a⟩+≡ ◁77a 78a▷

```
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 9, 10c, 16–23, 25, 26e, 28a, 30–33, 35a, 36, 50a, 51d, 53, 55–57, 61, 62, 64, 66d, 88–91, 93b, 94a, 96, 100, and 102b.
`ex`, used in chunks 3–11, 14c, 16–32, 34–36, 53b, 59–63, 65a, 67–76, 78–80, 82–93, 96c, 97a, and 103–109.

Uses `add` 4d, `div` 4d, `exmul` 4d, `mul` 7a, `operator*` 5a, `operator+` 5a, `operator-` 5a, `operator/` 5a, and `sub` 4d.

E.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.

78a ⟨cycle.cpp 64a⟩+≡ ◁77b 78b▷

```
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(radius_sq(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;
                    break;
                }
        }
    }
    if (ratio.is_zero()) // No normalisation is possible
        return (*this);

  return cycle((k÷ratio).normal(), indexed((l.op(0)÷ratio).evalm().normal(), l.op(1)), (m÷ratio).normal(), unit);
}
```

Uses `cycle` 3a 3a 3b 3b 3c 3d 5a 5a 5a 5a 5a 72c 72d 72d 73a 75d 75d 77a 77a 77a 77a 77a, `ex` 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, `get_dim` 3e, `is_zero` 4b, `k` 3a, `l` 3a, `m` 3a, `normal` 4b, `normalize` 5e, `numeric` 14a 57d, `op` 4b, `radius_sq` 6f, and `subs` 4b.

The normalisation to determinant ±1. We try to avoid imaginary numbers, thus if $-d÷D$ is known to be nonnegative, then we use it for square root.

78b ⟨cycle.cpp 64a⟩+≡ ◁78a 78c▷

```
cycle cycle::normalize_det(const ex & e, const ex & sign, const ex & D, bool fix_paravector) const
{
    ex d = det(e, sign, 0, fix_paravector), k_new;
    if ((-d÷D).info(info_flags::nonnegative))
        k_new=k÷sqrt(-d÷D);
    else
        k_new=k÷sqrt(d÷D);

    return (d.is_zero()? *this: normalize(k_new, e));
}
```

Uses `bool` 16a, `cycle` 3a 3a 3b 3b 3c 3d 5a 5a 5a 5a 5a 72c 72d 72d 73a 75d 75d 77a 77a 77a 77a 77a, `det` 6e 84b, `ex` 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, `is_zero` 4b, `k` 3a, `normalize` 5e, and `normalize_det` 5c.

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

78c ⟨cycle.cpp 64a⟩+≡ ◁78b 79a▷

```
ex cycle::center(const ex & metr, bool return_matrix) const
{
    if (is_a<numeric>(get_dim())) {
        ex e1, M, D = get_dim();
        if (metr.is_zero())
            e1 = unit;
        else {
            if (is_a<clifford>(metr))
                e1=metr;
```

Uses `bool` 16a, `center` 5f, `cycle` 3a 3a 3b 3b 3c 3d 5a 5a 5a 5a 5a 72c 72d 72d 73a 75d 75d 77a 77a 77a 77a 77a, `ex` 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, `get_dim` 3e, `is_zero` 4b, `metr` 3a, and `numeric` 14a 57d.

otherwise we delegate to *clifford_unit* constructor to find the metric.

79a ⟨cycle.cpp 64a⟩+≡ ◁78c 79b▷

```
        else
            try {
                e1 = clifford_unit(varidx(0, D), metr);
            } catch (exception &p) {
                throw(std::invalid_argument("cycle::center(): supplied metric"
                                            " is not suitable for Clifford unit"));
            }
    }
```

Uses catch 37a 37b, center 5f, cycle 3a 3a 3b 3b 3c 3d 5a 5a 5a 5a 5a 72c 72d 72d 73a 75d 75d 77a 77a 77a 77a 77a, metr 3a, and varidx 14a 15a 15b.

Now we adjust for paravector formalism.

79b ⟨cycle.cpp 64a⟩+≡ ◁79a 79c▷

```
    if (D≡ex_to<idx>(e1.op(1)).get_dim())
        M=ex_to<clifford>(e1).get_metric();
    else
        M=expand_paravector_metric(e1);
    exvector f_ind=M.get_free_indices();
```

Uses get_dim 3e, get_metric 3e, and op 4b.

Finally, the centre is constructed for the cycle and given metric by the formula [18, Defn. 2.2]:

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

79c ⟨cycle.cpp 64a⟩+≡ ◁79b 80a▷

```
    lst c;
    for(int i=0; i<D; i++)
        if (k.is_zero())
            c.append(get_l(i));
        else
            //c.append(jump_fnct(-ex_to<clifford>(e1).get_metric(varidx(i, D), varidx(i, D)))*get_l(i)/k);
            c.append(-M.subs(lst{f_ind[0]≡i, f_ind[1]≡i})∗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 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, get_l 4a, get_metric 3e, is_zero 4b, jump_fnct 59d, k 3a, l 3a, matrix 11d 16b 16c, numeric 14a 57d, subs 4b, and varidx 14a 15a 15b.

E.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.

80a ⟨cycle.cpp 64a⟩+≡ ◁79c 80b▷
```
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` 3a 3a 3b 3b 3c 3d 5a 5a 5a 5a 5a 72c 72d 72d 73a 75d 75d 77a 77a 77a 77a 77a, `ex` 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, `get_dim` 3e, `get_l` 4a, `k` 3a, `m` 3a, `nops` 4b, `numeric` 14a 57d, `subject_to` 6c, and `subs` 4b.

An utility function, which creates an additional Clifford unit from various types of expressions. We need to know the default Clifford *unit* for this and the dimensionality D of a cycle.

80b ⟨cycle.cpp 64a⟩+≡ ◁80a 80c▷
```
ex make_clifford_unit(const ex & e, const ex & D, const ex & unit) {
    varidx i1((new symbol)→setflag(status_flags::dynallocated), D),
        i1s((new symbol)→setflag(status_flags::dynallocated), D-1);
```

Uses `ex` 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c and `varidx` 14a 15a 15b.

First, we process the supplied e to the standard form of the Clifford unit. In the next two cases it is always for vector formalism.

80c ⟨cycle.cpp 64a⟩+≡ ◁80b 81a▷
```
    if (e.is_zero()) {
        if (ex_to<idx>(unit.op(1)).get_dim()≡D)
            return unit.subs(unit.op(1) ≡ i1);
        else
            return unit.subs(unit.op(1) ≡ i1s);
```

Uses `get_dim` 3e, `is_zero` 4b, `op` 4b, and `subs` 4b.

We need to run through every possible type of the argument to see either vector or paravector formalism is used for it.

81a ⟨cycle.cpp 64a⟩+≡ ◁80c 81b▷
```
} else if (is_a<clifford>(e)) {
    if (ex_to<idx>(e.op(1)).get_dim()≡D)
        return e.subs(e.op(1) ≡ i1);
    else if (ex_to<idx>(e.op(1)).get_dim()≡D-1)
        return e.subs(e.op(1) ≡ i1s);
    else
        throw(std::invalid_argument("make_clifford_unit(): "
                                    "Clifford unit has unsuitable dimensionality"));
```

Uses get_dim 3e, op 4b, and subs 4b.

A similar type of obtaining dimensionality is used for indexed objects.

81b ⟨cycle.cpp 64a⟩+≡ ◁81a 81c▷
```
} else if (is_a<indexed>(e)) {
    if (ex_to<idx>(e.op(1)).get_dim()≡D)
        return clifford_unit(i1, e);
    else if (ex_to<idx>(e.op(1)).get_dim()≡D-1)
        return clifford_unit(i1s, e);
    else
        throw(std::invalid_argument("make_clifford_unit(): "
                                    "indexed object has unsuitable dimensionality"));
```

Uses get_dim 3e and op 4b.

The final pair of supported types.

81c ⟨cycle.cpp 64a⟩+≡ ◁81b 81d▷
```
} else if (is_a<tensor>(e)) {
    return clifford_unit(i1, e);
} else if (is_a<matrix>(e)) {
    int C=ex_to<matrix>(e).cols();
    if (C≡D)
        return clifford_unit(i1, e);
    else if (C≡D-1)
        return clifford_unit(i1s, e);
    else
        throw(std::invalid_argument("make_clifford_unit(): matrix has unsuitable size"));
```

Uses matrix 11d 16b 16c.

Other typeas are not supported.

81d ⟨cycle.cpp 64a⟩+≡ ◁81c 82a▷
```
} else
    throw(std::invalid_argument("make_clifford_unit(): expect a clifford number, matrix, tensor or "
                                "indexed as the first parameter"));
}
```

Uses matrix 11d 16b 16c.

E.2.8. *Conversion of the* **cycle** *to the matrix form.* This method is inverse to the constructor of the **cycle** from its matrix, see (2.2) and [18, § 3.1]. This can use either vector or paravector formalism.

82a ⟨cycle.cpp 64a⟩+≡ ◁81d 82b▷
```
matrix cycle::to_matrix(const ex & e, const ex & sign, bool conjugate) const
{
    ex conv, // Indexed object for convolution with l
        D = get_dim();

    ex es = make_clifford_unit(e, D, unit); // The Clifford unit to be used in the matrix
    ex one = dirac_ONE(ex_to<clifford>(es).get_representation_label());

    varidx i0((new symbol)→setflag(status_flags::dynallocated), D),
        i1((new symbol)→setflag(status_flags::dynallocated), D);
```

Uses `bool` 16a, `cycle` 3a 3a 3b 3b 3c 3d 5a 5a 5a 5a 5a 72c 72d 72d 73a 75d 75d 77a 77a 77a 77a 77a, `ex` 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, `get_dim` 3e, `l` 3a, `matrix` 11d 16b 16c, `to_matrix` 6d, and `varidx` 14a 15a 15b.

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

82b ⟨cycle.cpp 64a⟩+≡ ◁82a 82c▷
```
    ex sign_m = sign.evalm();

    if (is_a<tensor>(sign_m))
        conv = indexed(ex_to<tensor>(sign_m), i0, i1.toggle_variance());
    else if (is_a<clifford>(sign_m)) {
        if (ex_to<varidx>(sign_m.op(1)).get_dim() ≡ D)
            conv = ex_to<clifford>(sign_m).get_metric(i0, i1.toggle_variance());
        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<varidx>(ind[0]).get_dim() ≡ D) ∧ (ex_to<varidx>(ind[1]).get_dim() ≡ D))
            conv = sign_m.subs(lst{ind[0] ≡ i0, ind[1] ≡ i1.toggle_variance()});
        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.toggle_variance());
        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` 3a 3a 3b 3b 3c 3d 5a 5a 5a 5a 5a 72c 72d 72d 73a 75d 75d 77a 77a 77a 77a 77a, `ex` 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, `get_dim` 3e, `get_metric` 3e, `matrix` 11d 16b 16c, `op` 4b, `subs` 4b, `to_matrix` 6d, and `varidx` 14a 15a 15b.

When all components are ready the key element of the matrix can be build. If we use vector formalism the base element is simple. Finally, the matrix is constructed.

82c ⟨cycle.cpp 64a⟩+≡ ◁82b 83a▷
```
    if ( ex_to<idx>(es.op(1)).get_dim() ≡ D) {
        ex a00 = expand_dummy_sum(l.subs(ex_to<indexed>(l).get_indices()[0] ≡ i0.toggle_variance())
                             ∗ conv ∗ es.subs(es.op(1)≡i1));
        return matrix(2, 2, lst{a00, m ∗ one, k ∗ one, -a00});
```

Uses `ex` 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, `get_dim` 3e, `k` 3a, `l` 3a, `m` 3a, `matrix` 11d 16b 16c, `op` 4b, and `subs` 4b.

For a *paravector* formalism a bit more care is required.

83a ⟨cycle.cpp 64a⟩+≡ ◁82c 83b▷
```
    } else {
        ex lconv=simplify_indexed(l.subs(ex_to<indexed>(l).get_indices()[0] ≡ i0.toggle_variance()) * conv);
        if (is_a<indexed>(lconv)) {
            ex scalar_p = expand_dummy_sum(lconv.subs(ex_to<indexed>(lconv).get_indices()[0] ≡ 0)*one),
                vector_p = expand_dummy_sum(indexed(paravector(lconv.op(0)),
                                         ex_to<varidx>(es.op(1)).toggle_variance())* es);
         return matrix(2, 2, lst{scalar_p+ (conjugate?-1:1)*vector_p, -m * one, k * one, -scalar_p+(conjugate?-
  1:1)*vector_p});
```

Uses ex 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, k 3a, l 3a, m 3a, matrix 11d 16b 16c, op 4b, paravector 63a 63c 103c 103c 103c 104d 104d 105a, subs 4b, and varidx 14a 15a 15b.

This shall not happen.

83b ⟨cycle.cpp 64a⟩+≡ ◁83a 83c▷
```
        } else
            throw(std::runtime_error("cycle::to_matrix(): after convolution with sign the indexed "
                                 "object disappered"));
    }
  }
```

Uses cycle 3a 3a 3b 3b 3c 3d 5a 5a 5a 5a 5a 72c 72d 72d 73a 75d 75d 77a 77a 77a 77a 77a and to_matrix 6d.

E.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. Calculation of the value of the cycle on the homogeneous coordinates.

83c ⟨cycle.cpp 64a⟩+≡ ◁83b 84a▷
```
  ex cycle::val(const ex & y, const ex & x) 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 "
                                     "an indexed object with one varindex"));
```

Uses cycle 3a 3a 3b 3b 3c 3d 5a 5a 5a 5a 5a 72c 72d 72d 73a 75d 75d 77a 77a 77a 77a 77a, ex 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, get_dim 3e, val 6a, and varidx 14a 15a 15b.

Other cases are treated similarly.

84a ⟨cycle.cpp 64a⟩+≡ ◁83c 84b▷

```
} 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())
             - numeric(2)*x* l*y0.subs(i0 ≡ ex_to<varidx>(ex_to<indexed>(l).get_indices()[0]).toggle_variance())
                      +m*pow(x,2));
}
```

Uses `cycle` 3a 3a 3b 3b 3c 3d 5a 5a 5a 5a 5a 72c 72d 72d 73a 75d 75d 77a 77a 77a 77a 77a, `get_metric` 3e, `k` 3a, `l` 3a, `m` 3a, `matrix` 11d 16b 16c, `nops` 4b, `numeric` 14a 57d, `subs` 4b, `val` 6a, and `varidx` 14a 15a 15b.

E.2.10. *Matrix methods for* **cycle**. The method *det*() may be defined in several ways. An alternative to the present definition is *pseudodeterminant* [5, (4.9)]

```
ex cycle::det(const ex & e, const ex & sign)) const
{ex M = normalize().to_matrix(e, sign);
  return remove_dirac_ONE(M.op(0)*clifford_star(M.op(3))-M.op(1)*clifford_star(M.op(2))) ; }
```

However due to the structure of matrix this coincides with the usual determinant of the matrix.

84b ⟨cycle.cpp 64a⟩+≡ ◁84a 84c▷

```
ex cycle::det(const ex & e, const ex & sign, const ex & k_norm, bool fix_paravector) const
{
  ex es = make_clifford_unit(e, get_dim(), unit); // The Clifford unit to be used in the matrix
  return (fix_paravector ∧ (ex_to<idx>(es.op(1)).get_dim() ≠ get_dim())? -1 : 1)*
    remove_dirac_ONE((k_norm.is_zero()?*this:normalize(k_norm))
                     .to_matrix(es, sign).determinant());
}
```

Defines:
`det`, used in chunks 6f, 9e, 17, 18f, 78b, 86b, and 90a.

Uses `bool` 16a, `cycle` 3a 3a 3b 3b 3c 3d 5a 5a 5a 5a 5a 72c 72d 72d 73a 75d 75d 77a 77a 77a 77a 77a, `ex` 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, `get_dim` 3e, `is_zero` 4b, `matrix` 11d 16b 16c, `normalize` 5e, `op` 4b, and `to_matrix` 6d.

Similarly, we need to fix the value of the cycle product, so it sign will not depend on either vector or paravector formalism is used.

84c ⟨cycle.cpp 64a⟩+≡ ◁84b 85a▷

```
ex cycle::cycle_product(const cycle & C, const ex & e, const ex & sign) const {
  ex es = make_clifford_unit(e, get_dim(), unit); // The Clifford unit to be used in the matrix
  bool is_paravect = (ex_to<idx>(es.op(1)).get_dim() ≡ get_dim());
  return (is_paravect? 1 : -1)*
    scalar_part(ex_to<matrix>(mul(ex_to<cycle>(C).to_matrix(es, sign,true), es, sign)).trace());
}
```

Defines:
`cycle_product`, used in chunks 8c and 21a.

Uses `bool` 16a, `cycle` 3a 3a 3b 3b 3c 3d 5a 5a 5a 5a 5a 72c 72d 72d 73a 75d 75d 77a 77a 77a 77a 77a, `ex` 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, `get_dim` 3e, `matrix` 11d 16b 16c, `mul` 7a, `op` 4b, and `to_matrix` 6d.

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

85a ⟨cycle.cpp 64a⟩+≡ ◁84c 85b▷
```
ex cycle::mul(const ex & C, const ex & e, const ex & sign, const ex & sign1) const
{
   if (is_a<cycle>(C)) {
      return canonicalize_clifford(to_matrix(e, sign).mul(
            ex_to<cycle>(C).to_matrix(e.is_zero()?unit: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 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 3a 3a 3b 3b 3c 3d 5a 5a 5a 5a 5a 72c 72d 72d 73a 75d 75d 77a 77a 77a 77a 77a, ex 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, is_zero 4b, matrix 11d 16b 16c, mul 7a, and to_matrix 6d.

E.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: $M : C \mapsto MCM^{-1}$.

85b ⟨cycle.cpp 64a⟩+≡ ◁85a 85c▷
```
cycle cycle::matrix_similarity(const ex & M, const ex & e, const ex & sign, bool not_inverse,
                               const ex & sign_inv) const
{
   if (not (is_a<matrix>(M) ∧ ex_to<matrix>(M).rows()≡2 ∧ ex_to<matrix>(M).cols()≡2))
      throw(std::invalid_argument("cycle::matrix_similarity(): the first parameter sgould be "
                                  "a 2x2 matrix"));
   return matrix_similarity(M.op(0), M.op(1), M.op(2), M.op(3), e, sign, not_inverse, sign_inv);
}
```

Uses bool 16a, cycle 3a 3a 3b 3b 3c 3d 5a 5a 5a 5a 5a 72c 72d 72d 73a 75d 75d 77a 77a 77a 77a 77a, ex 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, matrix 11d 16b 16c, matrix_similarity 7c, and op 4b.

The same method works if the matrix is provided by its four elements.

85c ⟨cycle.cpp 64a⟩+≡ ◁85b 85d▷
```
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 ex & sign_inv) const
{
   ex es = make_clifford_unit(e, get_dim(), unit); // The Clifford unit to be used in the matrix
  matrix R=ex_to<matrix>(canonicalize_clifford(matrix(2,2,not_inverse?lst{a, b, c, d}:lst{clifford_star(d), -
clifford_star(b), -clifford_star(c), clifford_star(a)})
                                        .mul(ex_to<matrix>(mul(matrix(2,2,not_inverse?lst{clifford_star(d), -
clifford_star(b), -clifford_star(c), clifford_star(a)}:lst{a, b, c, d}), es, sign)))
                                        .evalm()).normal());
```

Uses bool 16a, cycle 3a 3a 3b 3b 3c 3d 5a 5a 5a 5a 5a 72c 72d 72d 73a 75d 75d 77a 77a 77a 77a 77a, ex 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, get_dim 3e, matrix 11d 16b 16c, matrix_similarity 7c, mul 7a, and normal 4b.

We do some anti-symmetrisation of the matrix before the call of **cycle**() constructor since matrix should posses it anyway but it may not be apparent to GiNaC.

85d ⟨cycle.cpp 64a⟩+≡ ◁85c 86a▷
```
         ÷∗        if (ex_to<idx>(es.op(1)).get_dim() ≡ get_dim())
              return cycle(matrix(2,2,lst{(R.op(0)-R.op(3))÷numeric(2),R.op(1),
                          R.op(2),(-R.op(0)+R.op(3))÷numeric(2)}), unit, es, sign_inv, get_dim());
           else
              return cycle(matrix(2,2,lst{(R.op(0)-clifford_bar(R.op(3)))÷numeric(2),R.op(1),R.op(2),
                          (-clifford_bar(R.op(0))+R.op(3))÷numeric(2)}), unit, es, sign_inv, get_dim());
           ∗÷
           return cycle(R, unit, es, sign_inv, get_dim());
}
```

Uses cycle 3a 3a 3b 3b 3c 3d 5a 5a 5a 5a 5a 72c 72d 72d 73a 75d 75d 77a 77a 77a 77a 77a, get_dim 3e, matrix 11d 16b 16c, numeric 14a 57d, and op 4b.

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

86a ⟨cycle.cpp 64a⟩+≡ ◁85d 86b▷
```
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 ex & sign_inv) const
{
//  ex sign_inv=is_a<matrix>(sign)?pow(sign,-1):sign;
   relational sl2_rel = (c*b ≡ (d*a-1));
```

Uses bool 16a, cycle 3a 3a 3b 3b 3c 3d 5a 5a 5a 5a 5a 72c 72d 72d 73a 75d 75d 77a 77a 77a 77a 77a, ex 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, matrix 11d 16b 16c, and sl2_similarity 7b 10c 61d 62a.

We check either the condition $ad - bc = 1$ can be used for substitution later.

86b ⟨cycle.cpp 64a⟩+≡ ◁86a 86c▷
```
   ex det=(a*d-b*c).eval();
   ex es=e.is_zero()?unit:e;
   if (is_a<numeric>(det) ∧ (ex_to<numeric>(det).evalf() ≠1))
      sl2_rel = (c*b≡c*b);
```

Uses det 6e 84b, ex 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, is_zero 4b, and numeric 14a 57d.

Evaluation of the matrix corresponding to the cycle.

86c ⟨cycle.cpp 64a⟩+≡ ◁86b 86d▷
```
   matrix R=ex_to<matrix>(canonicalize_clifford(
                  sl2_clifford(a, b, c, d, es, not_inverse)
                  .mul(ex_to<matrix>(mul(sl2_clifford(a, b, c, d, es, ¬not_inverse), es, sign_inv)))
                  .evalm().subs(sl2_rel, subs_options::algebraic | subs_options::no_pattern)).normal());
```

Uses matrix 11d 16b 16c, mul 7a, normal 4b, and subs 4b.

In vector formalism we make anti-symmetrisation of the matrix, and accordingly in para-vector.

86d ⟨cycle.cpp 64a⟩+≡ ◁86c 86e▷
```
   ÷*if (ex_to<idx>(es.op(1)).get_dim()≡get_dim())
    return cycle(matrix(2,2,lst{(R.op(0)-R.op(3))÷numeric(2),R.op(1),R.op(2),
    (-R.op(0)+R.op(3))÷numeric(2)}), unit, e, sign, get_dim());
    else
    return cycle(matrix(2,2,lst{(R.op(0)-clifford_bar(R).op(3))÷numeric(2),R.op(1),
    R.op(2),(-clifford_bar(R).op(0)+R.op(3))÷numeric(2)}), unit, e, sign, get_dim());*÷
      return cycle(R, unit, e, sign, get_dim());
}
```

Uses cycle 3a 3a 3b 3b 3c 3d 5a 5a 5a 5a 5a 72c 72d 72d 73a 75d 75d 77a 77a 77a 77a 77a, get_dim 3e, matrix 11d 16b 16c, numeric 14a 57d, and op 4b.

86e ⟨cycle.cpp 64a⟩+≡ ◁86d 87a▷
```
cycle cycle::sl2_similarity(const ex & M, const ex & e, const ex & sign, bool not_inverse,
                            const ex & sign_inv) 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, sign_inv);
   else
     throw(std::invalid_argument("sl2_similarity(): expect a list or matrix as the first parameter"));
}
```

Uses bool 16a, cycle 3a 3a 3b 3b 3c 3d 5a 5a 5a 5a 5a 72c 72d 72d 73a 75d 75d 77a 77a 77a 77a 77a, ex 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, matrix 11d 16b 16c, op 4b, and sl2_similarity 7b 10c 61d 62a.

cycle acts on other **cycle** by the similarity: $C : C_1 \mapsto CC_1C$, see [18, (4.8)]. If the metric e for similarity is not given, then we use the metric of C_1 for this.

87a ⟨cycle.cpp 64a⟩+≡ ◁86e 87b▷

```
cycle cycle::cycle_similarity(const cycle & C, const ex & e, const ex & sign, const ex & sign1,
                              const ex & sign_inv) const
{
// ex sign_inv=is_a<matrix>(sign)?pow(sign,-1):sign;
   ex es = make_clifford_unit(e, get_dim(), unit); // The Clifford unit to be used in the matrix
   if (ex_to<idx>(es.op(1)).get_dim() ≡ get_dim()) {// Vector formalism
      return cycle(ex_to<matrix>(canonicalize_clifford(C.mul(mul(C, es, sign,sign1.is_zero()?sign:sign1),
                                                   es, sign1.is_zero()?sign:sign1))),
                   unit, es, sign_inv, get_dim());
   } else { // Paravector formalism
      matrix M=ex_to<matrix>(to_matrix(es,sign,true)),
         M1=ex_to<matrix>(C.to_matrix(es,sign1.is_zero()?sign:sign1));
      return cycle(ex_to<matrix>(canonicalize_clifford((-M1*M*M1).evalm())),
                   unit, es, sign_inv, get_dim());
   }
}
```

Uses `cycle` 3a 3a 3b 3b 3c 3d 5a 5a 5a 5a 5a 72c 72d 72d 73a 75d 75d 77a 77a 77a 77a 77a, `cycle_similarity` 7e, `ex` 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, `get_dim` 3e, `is_zero` 4b, `matrix` 11d 16b 16c, `mul` 7a, `op` 4b, and `to_matrix` 6d.

Moebius map created by the cycle matrix.

87b ⟨cycle.cpp 64a⟩+≡ ◁87a 87c▷

```
ex cycle::moebius_map(const ex & P, const ex & e, const ex & sign) const {
   return clifford_moebius_map(to_matrix(e, sign), P, (e.is_zero()?unit:e));
}
```

Defines:
`moebius_map`, used in chunks 19–23, 26c, and 36.
Uses `cycle` 3a 3a 3b 3b 3c 3d 5a 5a 5a 5a 5a 72c 72d 72d 73a 75d 75d 77a 77a 77a 77a 77a, `ex` 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, `is_zero` 4b, and `to_matrix` 6d.

87c ⟨cycle.cpp 64a⟩+≡ ◁87b 88a▷

```
ex cycle::is_f_orthogonal(const cycle & C, const ex & e, const ex & sign, const ex & sign1,
                          const ex & sign_inv) const
{
   ex es=make_clifford_unit(e, get_dim(), unit);
   ex signc=sign1.is_zero()?sign:sign1;

   matrix M=ex_to<matrix>(to_matrix(es,sign,true)),
      M1=ex_to<matrix>(C.to_matrix(es,sign1.is_zero()?sign:sign1)),
      P= ex_to<matrix>(canonicalize_clifford((M*M1*M).evalm()));
   ÷* if (ex_to<idx>(es.op(1)).get_dim() ≡ get_dim()) { // Vector formalism
      P = ex_to<matrix>(canonicalize_clifford((M*M1*M).evalm()));
   } else { // Paravector formalism
      //    P = ex_to<matrix>(canonicalize_clifford((clifford_bar(M)*M1*clifford_bar(M)).evalm()));
      P = ex_to<matrix>(canonicalize_clifford(((M)*M1*(M)).evalm()));
      }*÷

   return (cycle(P, es, es, sign_inv, get_dim()).get_l(get_dim()-1).normal() ≡ 0);
// return (C.cycle_similarity(*this, e, sign, sign1).get_l(get_dim()-1).normal() == 0);
}
```

Uses `cycle` 3a 3a 3b 3b 3c 3d 5a 5a 5a 5a 5a 72c 72d 72d 73a 75d 75d 77a 77a 77a 77a 77a, `cycle_similarity` 7e, `ex` 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, `get_dim` 3e, `get_l` 4a, `is_f_orthogonal` 8d, `is_zero` 4b, `matrix` 11d 16b 16c, `normal` 4b, `op` 4b, and `to_matrix` 6d.

E.3. **Implementation of the cycle2D class.** The derived class **cycle2D** for two dimensional cycles. Here constructors, archiving, and comparison come first.

88a ⟨cycle.cpp 64a⟩+≡ ◁87c 88b▷
```
cycle2D::cycle2D() : inherited()
{
#if GINAC_VERSION_ATLEAST(1,5,0)
#else
 tinfo_key = &cycle2D::tinfo_static;
#endif
}
```

Uses cycle2D 9a 9b 15c 15c 15d 15d 54b 55c 61b 62d 62d 62d 62d 62d 77b 77b 77b 77b 77b 89b 89b 89b 89b 92c and GINAC_VERSION_ATLEAST 59a 59a.

88b ⟨cycle.cpp 64a⟩+≡ ◁88a 88c▷
```
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"));
#if GINAC_VERSION_ATLEAST(1,5,0)
#else
 tinfo_key = &cycle2D::tinfo_static;
#endif
}
```

Uses cycle2D 9a 9b 15c 15c 15d 15d 54b 55c 61b 62d 62d 62d 62d 62d 77b 77b 77b 77b 77b 89b 89b 89b 89b 92c, ex 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, get_dim 3e, GINAC_VERSION_ATLEAST 59a 59a, and metr 3a.

88c ⟨cycle.cpp 64a⟩+≡ ◁88b 88d▷
```
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"));
#if GINAC_VERSION_ATLEAST(1,5,0)
#else
 tinfo_key = &cycle2D::tinfo_static;
#endif
}
```

Uses cycle2D 9a 9b 15c 15c 15d 15d 54b 55c 61b 62d 62d 62d 62d 62d 77b 77b 77b 77b 77b 89b 89b 89b 89b 92c, ex 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, get_dim 3e, GINAC_VERSION_ATLEAST 59a 59a, l 3a, and metr 3a.

88d ⟨cycle.cpp 64a⟩+≡ ◁88c 89a▷
```
cycle2D::cycle2D(const matrix & M, const ex & metr, const ex & e, const ex & sign)
   : inherited(M, metr, e, sign, 2)
{
 if (get_dim() != 2)
  throw(std::invalid_argument("cycle2D::cycle2D(): class cycle2D is defined in two dimensions"));
#if GINAC_VERSION_ATLEAST(1,5,0)
#else
 tinfo_key = &cycle2D::tinfo_static;
#endif
}
```

Uses cycle2D 9a 9b 15c 15c 15d 15d 54b 55c 61b 62d 62d 62d 62d 62d 77b 77b 77b 77b 77b 89b 89b 89b 89b 92c, ex 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, get_dim 3e, GINAC_VERSION_ATLEAST 59a 59a, matrix 11d 16b 16c, and metr 3a.

89a ⟨cycle.cpp 64a⟩+≡ ◁88d 89b▷

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

Uses cycle 3a 3a 3b 3b 3c 3d 5a 5a 5a 5a 5a 72c 72d 72d 73a 75d 75d 77a 77a 77a 77a 77a, cycle2D 9a 9b 15c 15c 15d 15d 54b 55c 61b 62d 62d 62d 62d 62d 77b 77b 77b 77b 77b 89b 89b 89b 89b 92c, ex 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, get_k 3e, get_l 4a, get_m 4a, get_unit 4a, is_zero 4b, and metr 3a.

89b ⟨cycle.cpp 64a⟩+≡ ◁89a 89c▷

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

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

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

int cycle2D::compare_same_type(const basic &other) const
{
      GINAC_ASSERT(is_a<cycle2D>(other));
   return inherited::compare_same_type(other);
}

//const char *cycle2D::get_class_name() { return "cycle2D"; }
```

Defines:
cycle2D, used in chunks 9, 10c, 16–23, 25, 26e, 28a, 30–33, 35a, 36, 50a, 51d, 53, 55–57, 61, 62, 64, 66d, 88–91, 93b, 94a, 96, 100, and 102b.

Real and imaginary part of the representing vector.

89c ⟨cycle.cpp 64a⟩+≡ ◁89b 90a▷

```
ex cycle2D::real_part() const
{
   return cycle2D(k.real_part(),lst{get_l(0).real_part(),get_l(1).real_part()},m.real_part(),unit);
}

ex cycle2D::imag_part() const
{
   return cycle2D(k.imag_part(),lst{get_l(0).imag_part(),get_l(1).imag_part()},m.imag_part(),unit);
}
```

Uses cycle2D 9a 9b 15c 15c 15d 15d 54b 55c 61b 62d 62d 62d 62d 62d 77b 77b 77b 77b 77b 89b 89b 89b 89b 92c, ex 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, get_l 4a, k 3a, and m 3a.

E.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.

90a ⟨cycle.cpp 64a⟩+≡ ◁89c 90b▷

```
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), 0, true)÷(numeric(2)∗get_l(1)∗k)).normal()};
   return (return_matrix? (ex)matrix(2, 1, f) : (ex)f);
}
```

Uses `bool` 16a, `cycle2D` 9a 9b 15c 15c 15d 15d 54b 55c 61b 62d 62d 62d 62d 62d 77b 77b 77b 77b 77b 89b 89b 89b 89b 92c, `det` 6e 84b, `ex` 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, `focus` 9f, `get_l` 4a, `get_metric` 3e, `jump_fnct` 59d, `k` 3a, `matrix` 11d 16b 16c, `normal` 4b, `numeric` 14a 57d, and `varidx` 14a 15a 15b.

90b ⟨cycle.cpp 64a⟩+≡ ◁90a 90c▷

```
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 `bool` 16a, `cycle2D` 9a 9b 15c 15c 15d 15d 54b 55c 61b 62d 62d 62d 62d 62d 77b 77b 77b 77b 77b 89b 89b 89b 89b 92c, `ex` 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, `get_dim` 3e, `get_l` 4a, `get_metric` 3e, `is_zero` 4b, `k` 3a, `m` 3a, `numeric` 14a 57d, `op` 4b, `roots` 9g, and `varidx` 14a 15a 15b.

90c ⟨cycle.cpp 64a⟩+≡ ◁90b 90d▷

```
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-numeric(2)∗get_l(1)∗b+pm∗pow(b,2)).normal()).roots();
 }
```

Uses `cycle2D` 9a 9b 15c 15c 15d 15d 54b 55c 61b 62d 62d 62d 62d 62d 77b 77b 77b 77b 77b 89b 89b 89b 89b 92c, `ex` 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, `get_dim` 3e, `get_l` 4a, `get_metric` 3e, `k` 3a, `line_intersect` 10a, `m` 3a, `normal` 4b, `numeric` 14a 57d, `roots` 9g, and `varidx` 14a 15a 15b.

90d ⟨cycle.cpp 64a⟩+≡ ◁90c 91a▷

```
cycle2D cycle2D::sl2_similarity(const ex & M1, const ex & M2, const ex & e,
                              const ex & sign, bool not_inverse, const ex & sign_inv) const {
   if ((is_a<matrix>(M1) ∨ M1.info(info_flags::list)) ∧ (is_a<matrix>(M2) ∨ M2.info(info_flags::list)))
      return sl2_similarity(M1.op(0), M1.op(1), M1.op(2), M1.op(3),
                         M2.op(0), M2.op(1), M2.op(2), M2.op(3),e, sign, not_inverse, sign_inv);
   else
      throw(std::invalid_argument("cycle2D::sl2_similarity(): expect a lsts or matrices as "
                              "the first parameter"));
;
 }
```

Uses `bool` 16a, `cycle2D` 9a 9b 15c 15c 15d 15d 54b 55c 61b 62d 62d 62d 62d 62d 77b 77b 77b 77b 77b 89b 89b 89b 89b 92c, `ex` 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, `matrix` 11d 16b 16c, `op` 4b, and `sl2_similarity` 7b 10c 61d 62a.

91a ⟨cycle.cpp 64a⟩+≡ ◁90d 91b▷

```
cycle2D cycle2D::sl2_similarity(const ex & a1, const ex & b1, const ex & c1, const ex & d1,
                                const ex & a2, const ex & b2, const ex & c2, const ex & d2,
                                const ex & e, const ex & sign, bool not_inverse, const ex & sign_inv) const {
   ex es=e.is_zero()?unit:e;
   matrix R=ex_to<matrix>(canonicalize_clifford(
                              sl2_clifford(a1, b1, c1, d1, a2, b2, c2, d2, es, not_inverse)
                              .mul(ex_to<matrix>(mul(sl2_clifford(a1, b1, c1, d1,
                                                     a2, b2, c2, d2, es, !not_inverse), es, sign_inv)))
                              .evalm()).normal());
   return cycle(R, unit, e, sign, get_dim());
}
```

Uses bool 16a, cycle 3a 3a 3b 3b 3c 3d 5a 5a 5a 5a 5a 72c 72d 72d 73a 75d 75d 77a 77a 77a 77a 77a, cycle2D 9a 9b 15c 15c 15d 15d 54b 55c 61b 62d 62d 62d 62d 62d 77b 77b 77b 77b 77b 89b 89b 89b 89b 92c, ex 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, get_dim 3e, is_zero 4b, matrix 11d 16b 16c, mul 7a, normal 4b, and sl2_similarity 7b 10c 61d 62a.

This method try to guess either it was called for a single real matrix M and a Clifford unit e, or e supplies a second matrix.

91b ⟨cycle.cpp 64a⟩+≡ ◁91a 91c▷

```
cycle2D cycle2D::sl2_similarity(const ex & M, const ex & e) const {
   if (is_a<matrix>(e))
      return sl2_similarity(M, e, unit, (new tensdelta)->setflag(status_flags::dynallocated), true,
                            (new tensdelta)->setflag(status_flags::dynallocated));
   else
      return sl2_similarity(M, e, (new tensdelta)->setflag(status_flags::dynallocated), true,
                            (new tensdelta)->setflag(status_flags::dynallocated));
}
```

Uses cycle2D 9a 9b 15c 15c 15d 15d 54b 55c 61b 62d 62d 62d 62d 62d 77b 77b 77b 77b 77b 89b 89b 89b 89b 92c, ex 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, matrix 11d 16b 16c, and sl2_similarity 7b 10c 61d 62a.

91c ⟨cycle.cpp 64a⟩+≡ ◁91b 91d▷

```
cycle2D cycle2D::sl2_similarity(const ex & M, const ex & e, const ex & sign) const {
   if (is_a<matrix>(e))
      return sl2_similarity(M, e, sign, (new tensdelta)->setflag(status_flags::dynallocated), true,
                            (new tensdelta)->setflag(status_flags::dynallocated));
   else
      return sl2_similarity(M, e, sign, true, (new tensdelta)->setflag(status_flags::dynallocated));
}
```

Uses cycle2D 9a 9b 15c 15c 15d 15d 54b 55c 61b 62d 62d 62d 62d 62d 77b 77b 77b 77b 77b 89b 89b 89b 89b 92c, ex 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, matrix 11d 16b 16c, and sl2_similarity 7b 10c 61d 62a.

E.3.2. *Drawing* **cycle2D**. Some auxilliary functions used for drawing

91d ⟨cycle.cpp 64a⟩+≡ ◁91c 91e▷

```
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 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c and numeric 14a 57d.

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

91e ⟨cycle.cpp 64a⟩+≡ ◁91d 92a▷

```
#define DRAW_ARC(X, S)    u = X; \
   v = ex_to<numeric>(Cf.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 << "){" << du << "," << dv << "}"; \
   else          \
    ost << S <<  v << "," << u << "){" << (sign == 0? dv : -dv) << "," << (sign == 0? du : -du) << "}";
```

Defines:
DRAW_ARC, used in chunk 103b.
Uses du 100a, dv 100a, k_d 100a, numeric 14a 57d, op 4b, roots 9g, u 100a, v 100a, and zero_or_one 100a.

an auxillary function to find small numbers

92a ⟨cycle.cpp 64a⟩+≡ ◁91e 92b▷

```
bool is_almost_zero(const ex & x)
{
    if (is_a<numeric>(x))
        return (abs(ex_to<numeric>(x).to_double()) < 0.0000000001);
    else
        return x.is_zero();
}
```

Uses bool 16a, ex 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, is_zero 4b, and numeric 14a 57d.

an auxillary function to find almost numbers

92b ⟨cycle.cpp 64a⟩+≡ ◁92a 92c▷

```
bool is_almost_negative(const ex & x)
{
    if (is_a<numeric>(x))
        return (ex_to<numeric>(x.evalf()).to_double() < 0.0000000001);
    else
        return x.is_zero();
}
```

Uses bool 16a, ex 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, is_zero 4b, and numeric 14a 57d.

The main drawing routine for **cycle2D**.

92c ⟨cycle.cpp 64a⟩+≡ ◁92b 93a▷

```
void cycle2D::metapost_draw(ostream & ost, const ex & xmin, const ex & xmax,
                            const ex & ymin, const ex & ymax,
                            const lst & color, const string more_options, bool with_header,
                            int points_per_arc, bool asymptote, const string picture, bool only_path,
                            bool is_continuation, const string imaginary_options) const
{
 ostringstream draw_start, draw_options;
 string already_drawn =(is_continuation? "^^(" : "(" ); // Was any arc already drawn?
draw_start << "draw" << (asymptote ? "(" : "  ") << picture << (picture.size()==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 9, 10c, 16–23, 25, 26e, 28a, 30–33, 35a, 36, 50a, 51d, 53, 55–57, 61, 62, 64, 66d, 88–91, 93b, 94a, 96, 100, and 102b.

Uses bool 16a, ex 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, metapost_draw 10b, and string 14a 59d 59d 107b 107c.

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

93a ⟨cycle.cpp 64a⟩+≡ ◁92c 93b▷

```
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 ";
}
if (more_options ≠ "") {
   if (color.nops() ≡ 3)
      draw_options ≪ "+";
   else
      draw_options ≪ ",";
   draw_options ≪ more_options;
}
draw_options ≪ (asymptote ? ");" : ";") ≪ endl;
```

Uses `nops` 4b, `numeric` 14a 57d, and `op` 4b.

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

93b ⟨cycle.cpp 64a⟩+≡ ◁93a 94a▷

```
 if (with_header) {
  ost ≪ (asymptote ? "// Asymptote" : "% Metapost") ≪ " data in [" ≪ xmin ≪ ","
   ≪ xmax ≪ "]x[" ≪ ymin ≪ ","
     ≪ ymax ≪ "] for ";

 ostringstream equat;
 equat ≪ (ex)passing(lst{symbol("u"), symbol("v")});
 if (equat.str().length()< 256)
    ost ≪ equat.str();
 else
    ost ≪ " [approx.] " ≪ ex_to<cycle2D>(evalf()).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` 3a 3a 3b 3b 3c 3d 5a 5a 5a 5a 5a 72c 72d 72d 73a 75d 75d 77a 77a 77a 77a 77a, `cycle2D` 9a 9b 15c 15c 15d 15d 54b 55c 61b 62d 62d 62d 62d 62d 77b 77b 77b 77b 77b 89b 89b 89b 89b 92c, `ex` 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, `is_zero` 4b, `k` 3a, `l` 3a, `m` 3a, `op` 4b, `passing` 6b, `subs` 4b, `u` 100a, and `v` 100a.

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

94a ⟨cycle.cpp 64a⟩+≡ ◁93b 94b▷

```
cycle2D Cf=ex_to<cycle2D>(evalf()).normalize();
double xc = ex_to<numeric>(Cf.center().op(0)).to_double(),
   yc = ex_to<numeric>(Cf.center().op(1)).to_double(); // the center of cycle
double sign0 = ex_to<numeric>(-get_metric(varidx(0, 2), varidx(0, 2)).evalf()).to_double(),
sign1 = ex_to<numeric>(-get_metric(varidx(1, 2), varidx(1, 2)).evalf()).to_double(),
sign = sign0 * sign1;
double determinant = ex_to<numeric>(Cf.radius_sq()).to_double(),
   r=ex_to<numeric>(sqrt(abs(determinant))).to_double();
double epsilon=0.0000000001;
bool not_swapped = (sign>0 ∨ sign1≡0 ∨ ((sign <0) ∧ (determinant < epsilon)));
double signu = (not_swapped?sign0:sign1), signv = (not_swapped?sign1:sign0);
int iu = (not_swapped?0:1), iv = (not_swapped?1:0);
double umin = ex_to<numeric>((not_swapped ? xmin : ymin).evalf()).to_double(),
   umax = ex_to<numeric>((not_swapped ? xmax : ymax).evalf()).to_double(),
   vmin = ex_to<numeric>((not_swapped ? ymin: xmin).evalf()).to_double(),
   vmax = ex_to<numeric>((not_swapped ? ymax : xmax).evalf()).to_double(),
   uc = (not_swapped ? xc: yc), vc = (not_swapped ? yc : xc);
lst b_roots = ex_to<lst>(Cf.roots(vmin, not_swapped).evalf()),
   t_roots = ex_to<lst>(Cf.roots(vmax, not_swapped).evalf());
```

Uses `bool` 16a, `center` 5f, `cycle` 3a 3a 3b 3b 3c 3d 5a 5a 5a 5a 5a 72c 72d 72d 73a 75d 75d 77a 77a 77a 77a 77a, `cycle2D` 9a 9b 15c 15c 15d 15d 54b 55c 61b 62d 62d 62d 62d 62d 77b 77b 77b 77b 77b 89b 89b 89b 89b 92c, `get_metric` 3e, `normalize` 5e, `numeric` 14a 57d, `op` 4b, `radius_sq` 6f, `roots` 9g, and `varidx` 14a 15a 15b.

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**.

94b ⟨cycle.cpp 64a⟩+≡ ◁94a 103c▷

```
   ⟨Imaginary coefficients 96b⟩
   ⟨Draw a straight line 95a⟩
   ⟨Find intersection points with the boundary 96a⟩
   if (sign > 0) { // elliptic metric
      ⟨Draw a circle 97d⟩
         } else { // parabolic or hyperbolic  metric
      ⟨Draw a parabola or hyperbola 100a⟩
         }
ost ≪ endl;
ost.flags(keep_flags);
return;
}
```

If line is detected we identify its visible portion.

95a ⟨Draw a straight line 95a⟩≡ (94b) 95b ▷

```
if (b_roots.nops() ≠ 2) { // a linear object
    if (Cf.get_k().is_zero() ∧ Cf.get_l(0).is_zero() ∧ Cf.get_l(1).is_zero()) {
        if (with_header)
            ost ≪ " the zero-radius cycle at infinity" ≪ endl;
        return;
    }
    if (with_header)
        ost ≪ " (straight line)" ≪ endl;
    double u1, u2, v1, v2;
    if (b_roots.nops() ≡ 1){ // a "non-horisontal" line
        u1 = std::max(std::min(ex_to<numeric>(b_roots.op(0)).to_double(), umax), umin);
        u2 = std::min(std::max(ex_to<numeric>(t_roots.op(0)).to_double(), umin), umax);
    } else { // a "horisontal" line
        u1 = umin;
        u2 = umax;
    }
```

Uses cycle 3a 3a 3b 3b 3c 3d 5a 5a 5a 5a 5a 72c 72d 72d 73a 75d 75d 77a 77a 77a 77a 77a, get_k 3e, get_l 4a, is_zero 4b, nops 4b, numeric 14a 57d, and op 4b.

Vertical lines case.

95b ⟨Draw a straight line 95a⟩+≡ (94b) ◁95a 95c ▷

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

Uses get_l 4a, is_zero 4b, numeric 14a 57d, and op 4b.

Look for the visible portion of generic line.

95c ⟨Draw a straight line 95a⟩+≡ (94b) ◁95b 95d ▷

```
} else {
    v1 = ex_to<numeric>(Cf.roots(u1, ¬not_swapped).op(0)).to_double();
    v2 = ex_to<numeric>(Cf.roots(u2, ¬not_swapped).op(0)).to_double();
    if ((std::max(v1, v2)-vmax > epsilon) ∨ (std::min(v1, v2) -vmin < -epsilon )) {
        ost.flags(keep_flags);
        return; //out of scope
    }
}
```

Uses numeric 14a 57d, op 4b, and roots 9g.

Actual drawing of the line.

95d ⟨Draw a straight line 95a⟩+≡ (94b) ◁95c

```
    ost ≪ (only_path ? already_drawn : draw_start.str())
        ≪ (not_swapped? u1: v1) ≪ "," ≪ (not_swapped ? v1: u1)
        ≪ ")--(" ≪ (not_swapped ? u2: v2) ≪ "," ≪ (not_swapped ? v2: u2) ≪ ")"
        ≪ (only_path ? "" : draw_options.str());
already_drawn="^^(";
if (with_header)
    ost ≪ endl;
ost.flags(keep_flags);
return;
}
```

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

96a ⟨Find intersection points with the boundary 96a⟩≡ (94b)

```
double left[2] = {std::max(std::min(uc, umax), umin),
                  std::max(std::min(uc, umax), umin)},
    right[2] = {std::max(std::min(uc, umax), umin),
                std::max(std::min(uc, umax), umin)};

    if (ex_to<numeric>(b_roots.op(0).evalf()).is_real()) {
        if (ex_to<numeric>((b_roots.op(0)-b_roots.op(1)).evalf()).is_positive())
            b_roots = lst{b_roots.op(1), b_roots.op(0)}; // rearrange to have minimum value first
        left[0] = std::min(std::max(ex_to<numeric>(b_roots.op(0)).to_double(), umin), umax);
        right[0] = std::max(std::min(ex_to<numeric>(b_roots.op(1)).to_double(), umax), umin);
    }
    if (ex_to<numeric>(t_roots.op(0).evalf()).is_real()) {
        if (ex_to<numeric>((t_roots.op(0)-t_roots.op(1)).evalf()).is_positive())
            t_roots = lst{t_roots.op(1), t_roots.op(0)}; // rearrange to have minimum value first
        left[1] = std::min(std::max(ex_to<numeric>(t_roots.op(0)).to_double(), umin), umax);
        right[1] = std::max(std::min(ex_to<numeric>(t_roots.op(1)).to_double(), umax), umin);
    }
```

Defines:
`left`, used in chunks 98 and 101–103.
Uses `numeric` 14a 57d and `op` 4b.

If a **cycle2D** has complex coefficients it still may intersect the real plain in a couple of points. To find them we first solve the linear equation.

96b ⟨Imaginary coefficients 96b⟩≡ (94b) 96c ▷

```
    if (¬ (Cf.get_k().imag_part().is_zero() ∧ Cf.get_l(0).imag_part().is_zero()
           ∧ Cf.get_l(1).imag_part().is_zero() ∧ Cf.get_m().imag_part().is_zero())) {
        if (imaginary_options ≡ "invisible")
            return;
        realsymbol x1("x1"), y1("y1");
        cycle2D CI=ex_to<cycle2D>(Cf.imag_part());
        lst sol=ex_to<lst>(lsolve(lst{CI.val(lst{x1,y1})≡0}, lst{x1,y1}));
```

Uses `cycle2D` 9a 9b 15c 15c 15d 15d 54b 55c 61b 62d 62d 62d 62d 62d 77b 77b 77b 77b 77b 89b 89b 89b 89b 92c, `get_k` 3e, `get_l` 4a, `get_m` 4a, `is_zero` 4b, `realsymbol` 14a 14b, and `val` 6a.

Then we use the linear substitution to solve the quadratic equation.

96c ⟨Imaginary coefficients 96b⟩+≡ (94b) ◁96b 96d ▷

```
        CI=ex_to<cycle2D>(Cf.normalize().real_part());
        ex eq=(CI.val(lst{x1,y1}).subs(sol)).normal();
        ex t=(eq.has(x1)?x1:y1),  s=(eq.has(x1)?y1:x1);
        double A, B, C, D;
        A=ex_to<numeric>(eq.coeff(ex_to<symbol>(t),2)).to_double();
        B=ex_to<numeric>(eq.coeff(ex_to<symbol>(t),1)).to_double();
        C=ex_to<numeric>(eq.coeff(ex_to<symbol>(t),0)).to_double();
        D=B*B-4*A*C;
```

Uses `cycle2D` 9a 9b 15c 15c 15d 15d 54b 55c 61b 62d 62d 62d 62d 62d 77b 77b 77b 77b 77b 89b 89b 89b 89b 92c, `ex` 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, `normal` 4b, `normalize` 5e, `numeric` 14a 57d, `subs` 4b, and `val` 6a.

If the quadratic equation has real roots we draw respective points.

96d ⟨Imaginary coefficients 96b⟩+≡ (94b) ◁96c 97a ▷

```
        if (abs(A)<epsilon ∨ D≥0){
            if (with_header)
          ost ≪ endl ≪ "// imaginary coefficients, the intersection with the real plane is dots only";
```

Two roots are follow.

97a ⟨Imaginary coefficients 96b⟩+≡ (94b) ◁96d 97b▷

```
for(int i=-1; i<2; i+=2) {
    double t1;
    if (abs(A)<epsilon) {
        i=1; // No need for second pass
        if (abs(B)<epsilon)
            return; // trivial identity
        else
            t1=-C÷B;
    } else
        t1= ex_to<numeric>((-B+i*sqrt((numeric)D))÷2.0÷A).to_double();
    exmap em;
    em.insert(std::make_pair(t, t1));
    ex s1=s.subs(sol.subs(em));
    uc=ex_to<numeric>(eq.has(x1)? t1 : s1).to_double();
    vc=ex_to<numeric>(eq.has(x1)? s1 : t1).to_double();
```

Uses ex 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, numeric 14a 57d, and subs 4b.

After the double check, we reset the drawing style to the hard-coded style for imaginary objects.

97b ⟨Imaginary coefficients 96b⟩+≡ (94b) ◁97a 97c▷

```
    if (abs(ex_to<numeric>(Cf.val(lst{uc,vc}).evalf()).to_double()) < epsilon) {
        if (asymptote)
            draw_options.str(","+imaginary_options+");");
        else
            draw_options.str(" "+imaginary_options+";");
        ost ≪ endl;
        {⟨place a dot 99c⟩}
    } else {
        std::cerr ≪ "Calculation of dots in imaginary cycle is inaccurate" ≪ std::endl;
    }
}
```

Uses cycle 3a 3a 3b 3b 3c 3d 5a 5a 5a 5a 5a 72c 72d 72d 73a 75d 75d 77a 77a 77a 77a 77a, numeric 14a 57d, and val 6a.

If the quadratic equation does not have real roots we draw respective points.

97c ⟨Imaginary coefficients 96b⟩+≡ (94b) ◁97b

```
  } else
    if (with_header)
      ost ≪ endl ≪ "// imaginary coefficients, no intersection with the real plane" ≪ endl;
  ost ≪ endl;
  return;
}
```

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

97d ⟨Draw a circle 97d⟩≡ (94b) 98a▷

```
if (determinant > epsilon) {
 double u[2][4], v[2][4];
 if (with_header)
  ost ≪ " /circle of radius " ≪ r ≪ endl;
 if (uc+r < umin ∨ uc-r > umax ∨ vc+r< vmin ∨ vc-r > vmax ∨
    pow(std::max(umax-uc,uc-umin),2.0)+pow(std::max(vmax-vc,vc-vmin),2.0)<determinant) {
   if (with_header)
      ost ≪ "   // out of the window " ≪ endl;
 } else {
```

Uses u 100a and v 100a.

Depending from the y-position of the centre we draw different arcs. The first case is the centre is above the horizontal strip.

98a ⟨Draw a circle 97d⟩+≡ (94b) ◁97d 98b▷

```
if ( vc-vmax > epsilon) {
    u[0][2] = left[1]; u[0][3] = right[1];
    u[1][2] = left[0]; u[1][3] = right[0];
    u[0][0] = u[1][0] = uc;
    u[0][1] = u[1][1] = uc;
```

Uses `left` 96a and `u` 100a.

The case when the centre is in the the horizontal strip.

98b ⟨Draw a circle 97d⟩+≡ (94b) ◁98a 98c▷

```
} else if (vc-vmin > epsilon) {
    u[0][0] = left[1]; u[0][1] = right[1];
    u[0][2] = right[0]; u[0][3] = left[0];

    if (uc-r-umin > epsilon)
        u[1][0] = u[1][3] = uc-r;
    else
        u[1][0] = u[1][3] = umin;

    if (umax-uc-r > epsilon)
        u[1][1] = u[1][2] = uc+r;
    else
        u[1][1] = u[1][2] = umax;
```

Uses `left` 96a and `u` 100a.

Finally, the centre is below the horizontal strip.

98c ⟨Draw a circle 97d⟩+≡ (94b) ◁98b 98d▷

```
} else {
 u[0][0] = left[1]; u[0][1] = right[1];
 u[1][0] = left[0]; u[1][1] = right[0];
 u[0][2] = u[1][2] = uc;
 u[0][3] = u[1][3] = uc;
}
```

Uses `left` 96a and `u` 100a.

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

98d ⟨Draw a circle 97d⟩+≡ (94b) ◁98c 99a▷

```
lst y_roots;
for (int j=0; j<2; j++)
 for (int i=0; i<4; i++)
  if (abs(u[j][i]-uc) < epsilon) // Touch the horizontal boundary?
   v[j][i] = (i≡0 ∨ i ≡1? vc+r : vc-r);
  else if (abs(u[j][i]-uc-r)<epsilon ∨ abs(u[j][i]-uc+r)<epsilon) // Touch the vertical boundary?
   v[j][i] = vc;
  else {
   y_roots = Cf.roots(u[j][i], false);
   if (ex_to<numeric>(y_roots.op(0)).is_real()) { // does circle intersect the boundary?
    if (i<2)
       v[j][i] = std::min(ex_to<numeric>(std::max(y_roots.op(0), y_roots.op(1))).to_double(), vmax);
    else
       v[j][i] = std::max(ex_to<numeric>(std::min(y_roots.op(0), y_roots.op(1))).to_double(), vmin);
   } else
    v[j][i] = vc;
  }
```

Uses `numeric` 14a 57d, `op` 4b, `roots` 9g, `u` 100a, and `v` 100a.

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.

99a ⟨Draw a circle 97d⟩+≡ (94b) ◁98d 99b▷

```
for (int i=0; i<4; i++) {// actual drawing of four arcs
 int s = (i≡0 ∨ i ≡2? -1:1);
 if ((u[0][i] ≠ u[1][i]) ∨ (v[0][i] ≠ v[1][i])) {// do not draw empty arc
  ost ≪ "   " ≪ (only_path ? already_drawn : draw_start.str()) ≪ u[0][i] ≪","
     ≪ v[0][i] ≪ "){" ≪ s∗(v[0][i]-vc) ≪ "," ≪ s∗(uc-u[0][i])
     ≪ (asymptote ? "}::{" : "}...{")
     ≪ s∗(v[1][i]-vc) ≪ "," ≪ s∗(uc-u[1][i]) ≪ "}(" ≪ u[1][i] ≪"," ≪ v[1][i] ≪ ")"
     ≪ (only_path ? "" : draw_options.str());
  already_drawn="^^(";
 }
}
}
```

Uses u 100a and v 100a.

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

99b ⟨Draw a circle 97d⟩+≡ (94b) ◁99a 99e▷

```
} else if (is_almost_zero(determinant)) {
    if (with_header)
        ost ≪ " /circle of zero-radius" ≪ endl;
    ⟨place a dot 99c⟩
```

This code places a dot at the point (U, V).

99c ⟨place a dot 99c⟩≡ (97b 99b 102c) 99d▷

```
    double U=ex_to<numeric>(uc).to_double();
    double V=ex_to<numeric>(vc).to_double();
    if ((umin ≤U) ∧ (umax≥ U) ∧ (vmin≤ V) ∧ (vmax≥ V)) {
        ost ≪ (asymptote ? (only_path ? already_drawn : "dot(") : "draw " )
            ≪ picture ≪ (picture.size()≡0? "" : ",")
            ≪ (only_path ? "" : "(")
            ≪ uc ≪ "," ≪ vc ≪ ")" ≪ (only_path ? "" : draw_options.str());
    already_drawn="^^(";
```

Uses numeric 14a 57d.

99d ⟨place a dot 99c⟩+≡ (97b 99b 102c) ◁99c

```
    } else
        if (with_header)
            ost ≪ "// the vertex is out of range" ≪ endl;
```

99e ⟨Draw a circle 97d⟩+≡ (94b) ◁99b

```
} else
    if (with_header)
        ost ≪ " /circle of imaginary radius--not drawing" ≪ endl;
```

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

100a ⟨Draw a parabola or hyperbola 100a⟩≡ (94b)
```
double u, v, du, dv, k_d = ex_to<numeric>(Cf.get_k()).to_double(),
            lu = ex_to<numeric>(Cf.get_l(iu)).to_double(),
            lv = ex_to<numeric>(Cf.get_l(iv)).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 100b⟩
} else {
 ⟨Treating a hyperbola 102b⟩
}
```

Defines:
du, used in chunk 91e.
dv, used in chunk 91e.
k_d, used in chunks 91e, 101a, and 102d.
u, used in chunks 14–16, 21c, 23b, 27–33, 35c, 36, 50, 52b, 54a, 91e, 93b, and 97–99.
v, used in chunks 14–16, 21c, 23b, 27–32, 36, 50, 52b, 54a, 91e, 93b, and 97–99.
zero_or_one, used in chunks 91e, 102, and 103.
Uses bool 16a, get_k 3e, get_l 4a, k 3a, and numeric 14a 57d.

For parabolas degenerated into two parallel lines we draw them by the recursive call of this function **cycle2D**::*metapost_draw*().

100b ⟨Treating a parabola 100b⟩≡ (100a) 100c▷
```
if (sign0 ≡ 0 ∧ Cf.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), unit).metapost_draw(ost, xmin, xmax, ymin, ymax, color, more_options,
                                                 false, 0, asymptote, picture, only_path, is_continuation);
   cycle2D(0, lst{0, 1}, 2*b_roots.op(1), unit).metapost_draw(ost, xmin, xmax, ymin, ymax, color, more_options,
                                                 false, 0, asymptote, picture, only_path, true);
   if (with_header)
      ost ≪ endl;
   ost.flags(keep_flags);
   return;
```

Uses cycle2D 9a 9b 15c 15c 15d 15d 54b 55c 61b 62d 62d 62d 62d 62d 77b 77b 77b 77b 77b 89b 89b 89b 89b 92c, get_l 4a, is_zero 4b, metapost_draw 10b, and op 4b.

Two vertical lines are drawn here

100c ⟨Treating a parabola 100b⟩+≡ (100a) ◁100b 101a▷
```
} else if (sign1 ≡ 0 ∧ Cf.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), unit).metapost_draw(ost, xmin, xmax, ymin, ymax, color, more_options,
                                                 false, 0, asymptote, picture, only_path, is_continuation);
   cycle2D(0, lst{1, 0}, 2*b_roots.op(1), unit).metapost_draw(ost, xmin, xmax, ymin, ymax, color, more_options,
                                                 false, 0, asymptote, picture, only_path, true);
   if (with_header)
      ost ≪ endl;
   ost.flags(keep_flags);
   return;
}
```

Uses cycle2D 9a 9b 15c 15c 15d 15d 54b 55c 61b 62d 62d 62d 62d 62d 77b 77b 77b 77b 77b 89b 89b 89b 89b 92c, get_l 4a, is_zero 4b, metapost_draw 10b, and op 4b.

If a proper parabola is detected we rearrange intervals appropriately in order to draw pieces properly.

101a ⟨Treating a parabola 100b⟩+≡ (100a) ◁100c 101b▷

```
  if (with_header)
      ost ≪ " /parabola" ≪ endl;
   if (right[0]-left[0] > epsilon ∧ right[1]-left[1] > epsilon) {
      if (k_d∗(signu∗lv+signv∗lu) > 0) { //rearrange intervals
           double e = left[1]; left[1] = right[0]; right[0] = left[0]; left[0] =e;
      } else {
           double e = left[1]; left[1] = right[1]; right[1] = right[0]; right[0] =e;
      }
   }
```

Uses `k_d` 100a and `left` 96a.

Parabolas can be exactly represented by a cubic Bézier arc if the second and third control points correspondingly are:

$$\left(\frac{2}{3}x_0+\frac{1}{3}x_1, \frac{1}{n}\left(\frac{1}{6}x_0^2k+\frac{1}{3}x_0x_1k-\frac{2}{3}x_0l-\frac{1}{3}lx_1+\frac{1}{2}m\right)\right),$$
$$\left(\frac{1}{3}x_0+\frac{2}{3}x_1, \frac{1}{n}\left(\frac{1}{3}x_0kx_1-\frac{1}{3}x_0l-\frac{2}{3}lx_1+\frac{1}{6}kx_1^2+\frac{1}{2}m\right)\right).$$

101b ⟨Treating a parabola 100b⟩+≡ (100a) ◁101a 101c▷

```
  for (int i =0; i < 2; i++) {
      if (right[i]-left[i] > epsilon) { // a proper branch of a parabola
          double cp[8];
          if (not_swapped) {
              cp[0] = left[i];
              cp[1] = ex_to<numeric>(Cf.val(lst{cp[0],0})÷2.0÷Cf.get_l(1)).to_double();
              cp[6] = right[i];
              cp[7] = ex_to<numeric>(Cf.val(lst{cp[6],0})÷2.0÷Cf.get_l(1)).to_double();
              cp[2] = 2.0÷3.0∗cp[0]+1.0÷3.0∗cp[6];
              cp[3] = ex_to<numeric>((numeric(1,6)∗cp[0]∗cp[0]∗Cf.get_k() + 1.0÷3.0∗cp[0]∗cp[6]∗Cf.get_k()
                         - 2.0÷3.0∗cp[0]∗Cf.get_l(0)- 1.0÷3.0∗Cf.get_l(0)∗cp[6]+Cf.get_m()÷2.0)÷Cf.get_l(1)).to_double();
              cp[4] = 1.0÷3.0∗cp[0]+2.0÷3.0∗cp[6];
              cp[5] = ex_to<numeric>((1.0÷3.0∗cp[0]∗Cf.get_k()∗cp[6]-1.0÷3.0∗cp[0]∗Cf.get_l(0)
                                   -2.0÷3.0∗Cf.get_l(0)∗cp[6]+numeric(1,6)∗Cf.get_k()∗cp[6]∗cp[6]
                                   +Cf.get_m()÷2.0)÷Cf.get_l(1)).to_double();
```

Uses `get_k` 3e, `get_l` 4a, `get_m` 4a, `left` 96a, `numeric` 14a 57d, and `val` 6a.

The similar formulae for swapped drawing.

101c ⟨Treating a parabola 100b⟩+≡ (100a) ◁101b 102a▷

```
    } else {
              cp[1] = left[i];
              cp[0] = ex_to<numeric>(Cf.val(lst{0,cp[1]})÷2.0÷Cf.get_l(0)).to_double();
              cp[7] = right[i];
              cp[6] = ex_to<numeric>(Cf.val(lst{0,cp[7]})÷2.0÷Cf.get_l(0)).to_double();
              cp[3] = 2.0÷3.0∗cp[1]+1.0÷3.0∗cp[7];
              cp[2] = ex_to<numeric>((numeric(1,6)∗cp[1]∗cp[1]∗Cf.get_k() + 1.0÷3.0∗cp[1]∗cp[7]∗Cf.get_k()
                         - 2.0÷3.0∗cp[1]∗Cf.get_l(1)- 1.0÷3.0∗Cf.get_l(1)∗cp[7]+Cf.get_m()÷2.0)÷Cf.get_l(0)).to_double();
              cp[5] = 1.0÷3.0∗cp[1]+2.0÷3.0∗cp[7];
              cp[4] = ex_to<numeric>((1.0÷3.0∗cp[1]∗Cf.get_k()∗cp[7]-1.0÷3.0∗cp[1]∗Cf.get_l(1)
                                   -2.0÷3.0∗Cf.get_l(1)∗cp[7]+numeric(1,6)∗Cf.get_k()∗cp[7]∗cp[7]
                                   +Cf.get_m()÷2.0)÷Cf.get_l(0)).to_double();
    }
```

Uses `get_k` 3e, `get_l` 4a, `get_m` 4a, `left` 96a, `numeric` 14a 57d, and `val` 6a.

The actual drawing of the parabola arcs.

102a ⟨Treating a parabola 100b⟩+≡ (100a) ◁101c

```
    ost ≪ (only_path ? already_drawn : draw_start.str()) ≪ cp[0] ≪ "," ≪ cp[1] ≪ ") .. controls (";
  if (asymptote)
    ost ≪ cp[2] ≪ "," ≪ cp[3] ≪ ") and (" ≪ cp[4] ≪ "," ≪ cp[5] ≪ ") .. (";
  else
    ost ≪ "(" ≪ cp[2] ≪ "," ≪ cp[3] ≪ ")) and ((" ≪ cp[4] ≪ "," ≪ cp[5] ≪ ")) .. (";
  ost ≪ cp[6] ≪ "," ≪ cp[7] ≪ ")" ≪ (only_path ? "" : draw_options.str());
  already_drawn="^^(";
    }
  }
```

If a hyperbola degenerates into a light cone we draw it as two separate lines.

102b ⟨Treating a hyperbola 102b⟩≡ (100a) 102c▷

```
  if (abs(determinant)<epsilon) {
    if (with_header)
      ost ≪ " / a light cone at (" ≪ xc ≪ "," ≪ yc ≪")" ≪ endl;
    cycle2D(0, lst{1, 1}, 2*(uc+vc), unit).metapost_draw(ost, xmin, xmax, ymin, ymax, color, more_options,
                                   false, 0, asymptote, picture, only_path, is_continuation);
    cycle2D(0, lst{1, -1}, 2*(uc-vc), unit).metapost_draw(ost, xmin, xmax, ymin, ymax, color, more_options,
                                   false, 0, asymptote, picture, only_path, true);
```

Uses `cycle2D` 9a 9b 15c 15c 15d 15d 54b 55c 61b 62d 62d 62d 62d 62d 77b 77b 77b 77b 77b 89b 89b 89b 89b 92c and `metapost_draw` 10b.

We also put a dot to single out the light cone vertex.

102c ⟨Treating a hyperbola 102b⟩+≡ (100a) ◁102b 102d▷

```
    if (¬ only_path) {
      ⟨place a dot 99c⟩
      if (with_header)
        ost ≪ endl;
    }
    ost.flags(keep_flags);
    return;
```

Otherwise we rearrange the interwals for hyperbola branches.

102d ⟨Treating a hyperbola 102b⟩+≡ (100a) ◁102c 103a▷

```
  } else {
    if (with_header)
      ost ≪ " /hyperbola" ≪ endl;
    if (vmin-vc > epsilon) {
      double 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 (vc-vmax > epsilon) {
      double 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 `k_d` 100a, `left` 96a, and `zero_or_one` 100a.

Two arcs of the hyperbola are drown now

103a ⟨Treating a hyperbola 102b⟩+≡ (100a) ◁102d 103b▷
```
int points = (points_per_arc ≡ 0? 7 : points_per_arc);
for (int i =0; i < 2; i++) {
  double dir= ex_to<numeric>(csgn(signv*(2*zero_or_one-1))).to_double(); //direction of the tangent vectors
  //double dir = ((sign == 0? lv : signv*(2*zero_or_one-1))<0?-1:1); direction of the tangent vectors (second alternative)
   if (right[i]-left[i] > epsilon ) { // a proper branch of the hyperbola
```

Defines:
  points, used in chunks 14, 33a, 50b, 52a, and 103b.
Uses left 96a, numeric 14a 57d, and zero_or_one 100a.

Points for the spline are placed equally spaced in the hyperbolic angle parameter.

103b ⟨Treating a hyperbola 102b⟩+≡ (100a) ◁103a
```
        double f_left=ex_to<numeric>(asinh((left[i]-uc)÷r)).to_double(),
                  f_right=ex_to<numeric>(asinh((right[i]-uc)÷r)).to_double();
      DRAW_ARC(ex_to<numeric>(sinh(f_left)*r+uc).to_double(), (only_path ? already_drawn : draw_start.str()));
        for (int j=1; j<points; j++) {
        DRAW_ARC(ex_to<numeric>(sinh(f_left*(1.0-j÷(points-1.0))+f_right*j÷(points-1.0))*r+uc).to_double(),
                  (asymptote ? "::(" : "...(") );
        }
        ost ≪ (only_path ? "" : draw_options.str());
        already_drawn="^^(";
     }
     if (change_branch)
        zero_or_one = 1 - zero_or_one; // make a swap for the next branch of hyperbola
 }
```

Uses DRAW_ARC 91e, left 96a, numeric 14a 57d, points 103a, and zero_or_one 100a.

E.3.3. *Methods in paravector class.* Constructors and archivers.

103c ⟨cycle.cpp 64a⟩+≡ ◁94b 104a▷
```
paravector::paravector() : vector() {
#if GINAC_VERSION_ATLEAST(1,7,1)
#else
  std::cerr ≪ "GiNaC version is prior 1.7.1, the paravector formalism will not work properly!!!" ≪ std::endl
#endif
}

paravector::paravector(const ex & b) {
#if GINAC_VERSION_ATLEAST(1,7,1)
#else
  std::cerr ≪ "GiNaC version is prior 1.7.1, the paravector formalism will not work properly!!!" ≪ std::endl
#endif
   vector=ex_to<basic>(b);
}

void paravector::archive(archive_node &n) const {
   inherited::archive(n);
   n.add_ex("vector", vector);
}

void paravector::read_archive(const archive_node &n, lst &sym_lst) {
   inherited::read_archive(n, sym_lst);
   n.find_ex("vector", vector, sym_lst);
}
GINAC_BIND_UNARCHIVER(paravector);
```

Defines:
  paravector, used in chunks 13d, 16–18, 24a, 28b, 32–35, 64b, 66b, 67c, 70, 83a, 104, and 105b.
Uses ex 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c and GINAC_VERSION_ATLEAST 59a 59a.

This is the only non-trivial method in the class which motivate its existanse

104a ⟨cycle.cpp 64a⟩+≡ ◁103c 104b▷
```
  ex paravector::eval_indexed(const basic & i) const {
      GINAC_ASSERT(i.nops() ≡ 2 ∧ is_a<idx>(i.op(1)));

      idx mu;
```
Uses ex 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, nops 4b, op 4b, and paravector 63a 63c 103c 103c 103c 104d 104d 105a.

We build an index with the shifts index.

104b ⟨cycle.cpp 64a⟩+≡ ◁104a 104c▷
```
      if (is_a<varidx>(i.op(1))) {
          if (ex_to<varidx>(i.op(1)).is_contravariant()) {
              mu=varidx(ex_to<varidx>(i.op(1)).get_value()+1, ex_to<varidx>(i.op(1)).get_dim()+1,false);
          } else {
              mu=varidx(ex_to<varidx>(i.op(1)).get_value()+1, ex_to<varidx>(i.op(1)).get_dim()+1,true);
          }
      } else if(is_a<idx>(i.op(1)))
          mu=idx(ex_to<varidx>(i.op(1)).get_value()+1, ex_to<varidx>(i.op(1)).get_dim()+1);
      else
       throw(std::invalid_argument("paravector::eval_indexed(): second argument shall be an index"));
```
Uses get_dim 3e, op 4b, paravector 63a 63c 103c 103c 103c 104d 104d 105a, and varidx 14a 15a 15b.

Now we build the indexed object and check if a simplification occures.

104c ⟨cycle.cpp 64a⟩+≡ ◁104b 104d▷
```
      ex e=indexed(vector, mu);

      if (is_a<indexed>(e) ∧ e.op(1).is_equal(mu))
          return i.hold();
      else
          return e;
  }
```
Uses ex 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, is_equal 4b, and op 4b.

Paravectors are printed in the standard way.

104d ⟨cycle.cpp 64a⟩+≡ ◁104c 104e▷
```
  void paravector::do_print(const print_dflt & c, unsigned level) const {
      c.s ≪ vector;
  }

  void paravector::do_print_latex(const print_latex & c, unsigned level) const {
      c.s ≪ vector;
  }
```
Defines:
paravector, used in chunks 13d, 16–18, 24a, 28b, 32–35, 64b, 66b, 67c, 70, 83a, 104, and 105b.

Substitution method.

104e ⟨cycle.cpp 64a⟩+≡ ◁104d 105a▷
```
  ex paravector::subs(const ex & e, unsigned options) const {
      return paravector(vector.subs(e,options));
  }

  ex paravector::subs(const exmap & m, unsigned options) const {
      return paravector(vector.subs(m,options));
  }
```
Uses ex 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, m 3a, paravector 63a 63c 103c 103c 103c 104d 104d 105a, and subs 4b.

Some more service methods.

105a ⟨cycle.cpp 64a⟩+≡ ◁104e 105b▷
```
return_type_t paravector::return_type_tinfo() const {
    return make_return_type_t<paravector>();
}

int paravector::compare_same_type(const basic &other) const {
    GINAC_ASSERT(is_a<paravector>(other));
    return inherited::compare_same_type(other);
}
```

Defines:
`paravector`, used in chunks 13d, 16–18, 24a, 28b, 32–35, 64b, 66b, 67c, 70, 83a, 104, and 105b.

Finally, there are service methods to access the component of the *paravector*.

105b ⟨cycle.cpp 64a⟩+≡ ◁105a 105c▷
```
ex paravector::op(size_t i) const {
    GINAC_ASSERT(i≡0);
    return vector;
}

ex & paravector::let_op(size_t i) {
    GINAC_ASSERT(i≡0);
    return vector;
}
```

Uses `ex` 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, `let_op` 4b, `op` 4b, and `paravector` 63a 63c 103c 103c 103c 104d 104d 105a.

E.4. **Auxiliary functions implementation.** The auxillary functions defined as well.

E.4.1. *Heaviside function.* We define Heaviside function: $\chi(x) = 1$ for $x \geq 0$ and $\chi(x) = 0$ for $x < 0$.

105c ⟨cycle.cpp 64a⟩+≡ ◁105b 106a▷
```
//////////
// 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 3–11, 14c, 16–32, 34–36, 53b, 59–63, 65a, 67–76, 78–80, 82–93, 96c, 97a, and 103–109.
Uses `is_zero` 4b, `jump_fnct` 59d, and `numeric` 14a 57d.

106a ⟨cycle.cpp 64a⟩+≡ ◁105c 106b▷

```
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 3–11, 14c, 16–32, 34–36, 53b, 59–63, 65a, 67–76, 78–80, 82–93, 96c, 97a, and 103–109.
Uses `is_zero` 4b, `jump_fnct` 59d, `mul` 7a, `nops` 4b, `numeric` 14a 57d, and `op` 4b.

106b ⟨cycle.cpp 64a⟩+≡ ◁106a 106c▷

```
static ex jump_fnct_conjugate(const ex & arg) {
    return jump_fnct(arg);
}
```

Defines:
`ex`, used in chunks 3–11, 14c, 16–32, 34–36, 53b, 59–63, 65a, 67–76, 78–80, 82–93, 96c, 97a, and 103–109.
Uses `jump_fnct` 59d.

106c ⟨cycle.cpp 64a⟩+≡ ◁106b 106d▷

```
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 3–11, 14c, 16–32, 34–36, 53b, 59–63, 65a, 67–76, 78–80, 82–93, 96c, 97a, and 103–109.
Uses `jump_fnct` 59d and `numeric` 14a 57d.

106d ⟨cycle.cpp 64a⟩+≡ ◁106c 107a▷

```
static void jump_fnct_print_dflt_text(const ex & x, const print_context & c) {
    c.s ≪ "H("; x.print(c); c.s ≪ ")";
}

static void jump_fnct_print_latex(const ex & x, const print_context & c) {
    c.s ≪ "\\chi("; x.print(c); c.s ≪ ")";
}
```

Defines:
`jump_fnct_print_dflt_text`, used in chunk 107a.
`jump_fnct_print_latex`, used in chunk 107a.
Uses `ex` 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c.

All above methods are used to register the function now.

107a ⟨cycle.cpp 64a⟩+≡ ◁106d 107b▷
```
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).
        print_func<print_latex>(jump_fnct_print_latex).
        //derivative_func(2*delta).
        power_func(jump_fnct_power).
        conjugate_func(jump_fnct_conjugate));
```

Uses `jump_fnct` 59d, `jump_fnct_print_dflt_text` 106d, and `jump_fnct_print_latex` 106d.

This function prints if its parameter is zero in a prominent way.

107b ⟨cycle.cpp 64a⟩+≡ ◁107a 107c▷
```
const string equality(const ex & E) {
    if (E.normal().is_zero())
        return "-equal-";
    else
        return "DIFFERENT!!!";
}
```

Defines:
`string`, used in chunks 10b, 11a, 16f, 18a, and 92c.
Uses `ex` 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, `is_zero` 4b, and `normal` 4b.

This function decodes metric sign into human-readable form.

107c ⟨cycle.cpp 64a⟩+≡ ◁107b 108a▷
```
const string 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:
`string`, used in chunks 10b, 11a, 16f, 18a, and 92c.
Uses `is_zero` 4b and `numeric` 14a 57d.

We are trying find a scalar part of the given expression.

108a ⟨cycle.cpp 64a⟩+≡ ◁107c 108b▷

```
ex scalar_part(const ex & e) {
    ex given=canonicalize_clifford(e.expand()),
        out=0, term;
    if (is_a<add>(given)){
        for (size_t i=0; i<given.nops(); i++) {
            try {
                term=remove_dirac_ONE(given.op(i));
            } catch (exception &p) {
                term=0;
            }
            out+=term;
        }
        return out.normal();
    } else{
        try {
            return remove_dirac_ONE(given);
        } catch (exception &p) {
            return 0;
        }
    }
}
```

Uses add 4d, catch 37a 37b, ex 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, expand 4b, nops 4b, normal 4b, and op 4b.

Elements of $SL_2(\mathbb{R})$ are transformed into appropriate "cliffordian" matrix. This is really a wrapper for the next function.

108b ⟨cycle.cpp 64a⟩+≡ ◁108a 108c▷

```
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 bool 16a, ex 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, matrix 11d 16b 16c, and op 4b.

A Clifford valued matrix from real values is constructed here.

108c ⟨cycle.cpp 64a⟩+≡ ◁108b 109a▷

```
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,
            one = dirac_ONE(ex_to<clifford>(e).get_representation_label());
        if (ex_to<idx>(e.op(1)).get_dim()≡2)
            e0 = e.subs(e.op(1) ≡ 0);
        else
            e0 = one;
        if (not_inverse)
            return matrix(2, 2,
                        lst{a * one, b * e0,
                            c * pow(e0, 3), d * one});
        else
            return matrix(2, 2,
                        lst{d * one, -b * e0,
                            -c * pow(e0, 3), a * one});
    } else
        throw(std::invalid_argument("sl2_clifford(): expect a clifford numeber as a parameter"));
}
```

Uses bool 16a, ex 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, get_dim 3e, matrix 11d 16b 16c, op 4b, and subs 4b.

This is really a wrapper for the next function.

109a ⟨cycle.cpp 64a⟩+≡ ◁108c 109b▷

```
matrix sl2_clifford(const ex & M1, const ex & M2, const ex & e, bool not_inverse) {
   if ((is_a<matrix>(M1) ∨ M1.info(info_flags::list)) ∧ (is_a<matrix>(M2) ∨ M2.info(info_flags::list)))
      return sl2_clifford(M1.op(0), M1.op(1), M1.op(2), M1.op(3), M2.op(0), M2.op(1), M2.op(2), M2.op(3),
                  e, not_inverse);
   else
    throw(std::invalid_argument("sl2_clifford(): expect a list or matrix as the first parameter"));
}
```

Uses `bool` 16a, `ex` 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, `matrix` 11d 16b 16c, and `op` 4b.

A Clifford valued matrix from real values is constructed here.

109b ⟨cycle.cpp 64a⟩+≡ ◁109a 109c▷

```
matrix sl2_clifford(const ex & a1, const ex & b1, const ex & c1, const ex & d1,
               const ex & a2, const ex & b2, const ex & c2, const ex & d2,
               const ex & e, bool not_inverse) {
   if (is_a<clifford>(e)) {
      ex one = dirac_ONE(ex_to<clifford>(e).get_representation_label());
      if (ex_to<idx>(e.op(1)).get_dim()≡2) {
         ex e0 = e.subs(e.op(1) ≡ 0);
         ex e1 = e.subs(e.op(1) ≡ 1);
         ex e01=e0*e1;
         if (not_inverse)
            return matrix(2, 2,
                     lst{a1*one+a2*e01, b1*e0+b2*e1,
                        -c1*e0+c2*e1, d1*one-d2*e01});
         else
            return matrix(2, 2,
                     lst{d1*one+d2*e01, -b1*e0-b2*e1,
                        c1*e0-c2*e1, a1*one-a2*e01});
```

Uses `bool` 16a, `ex` 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, `get_dim` 3e, `matrix` 11d 16b 16c, `op` 4b, and `subs` 4b.

Matrices for paravector formalism are obvious.

109c ⟨cycle.cpp 64a⟩+≡ ◁109b

```
      } else {
         ex e0 = e.subs(e.op(1) ≡ 0);
         if (not_inverse)
            return matrix(2, 2,
                     lst{a1*one+a2*e0, b1*one+b2*e0,
                        c1*one+c2*e0, d1*one+d2*e0});
         else
            return matrix(2, 2,
                     lst{d1*one+d2*e0, -b1*one-b2*e0,
                        -c1*one-c2*e0, a1*one+a2*e0});
      }
   } else
      throw(std::invalid_argument("sl2_clifford(): expect a clifford numeber as a parameter"));
}

} // namespace MoebInv
```

Uses `ex` 5b 14d 15a 15b 16a 62d 77a 77b 105c 106a 106b 106c, `matrix` 11d 16b 16c, `MoebInv` 58e, `op` 4b, and `subs` 4b.

Appendix F. License

This programme is distributed under GNU GPLv3 [8].

110 ⟨license 110⟩≡ (13a 58e 64a)

Appendix G. Index of Identifiers

School of Mathematics, University of Leeds, Leeds LS2 9JT, UK
E-mail address: kisilv@maths.leeds.ac.uk
URL: http://www.maths.leeds.ac.uk/~kisilv/