arXiv: 1512.02960, 2015
Volume 00, Number 0, Pages 000–000

AN EXTENSION OF MÖBIUS–LIE GEOMETRY WITH CONFORMAL ENSEMBLES OF CYCLES AND ITS IMPLEMENTATION IN A GiNaC LIBRARY

VLADIMIR V. KISIL

Abstract. We propose to consider ensembles of cycles (quadrics), which are interconnected through conformal-invariant geometric relations (e.g. "to be orthogonal", "to be tangent", etc.), as new objects in an extended Möbius–Lie geometry. It was recently demonstrated in several related papers, that such ensembles of cycles naturally parameterise many other conformally-invariant objects, e.g. loxodromes or continued fractions.

The paper describes a method, which reduces a collection of conformally invariant geometric relations to a system of linear equations, which may be accompanied by one fixed quadratic relation. To show its usefulness, the method is implemented as a C++ library. It operates with numeric and symbolic data of cycles in spaces of arbitrary dimensionality and metrics with any signatures. Numeric calculations can be done in exact or approximate arithmetic. In the two- and three-dimensional cases illustrations and animations can be produced. An interactive Python wrapper of the library is provided as well.

Contents

Date: October 7, 2018 (v3.1).
2010 *Mathematics Subject Classification.* Primary 51B25; Secondary 51N25, 51B10, 68U05, 11E88, 68W30.

List of Figures

1. Introduction

Lie sphere geometry [7, Ch. 3; 10] in the simplest planar setup unifies circles, lines and points—all together called *cycles* in this setup. Symmetries of Lie spheres geometry include (but are not limited to) fractional linear transformations (FLT) of the form:

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} : \; x \mapsto \frac{ax+b}{cx+d}, \qquad \text{where } \det \begin{pmatrix} a & b \\ c & d \end{pmatrix} \neq 0. \tag{1}$$

Following other sources, e.g. [55, § 9.2], we call (1) by FLT and reserve the name "Möbius maps" for the subgroup of FLT which fixes a particular cycle. For example, on the complex plane FLT are generated by elements of $SL_2(\mathbb{C})$ and Möbius maps fixing the real line are produced by $SL_2(\mathbb{R})$ [36, Ch. 1].

There is a natural set of FLT-invariant geometric relations between cycles (to be orthogonal, to be tangent, etc.) and the restriction of Lie sphere geometry to invariants of FLT is called *Möbius–Lie geometry*. Thus, an ensemble of cycles, structured by a set of such relations, will be mapped by FLT to another ensemble with the same structure.

It was shown recently that ensembles of cycles with certain FLT-invariant relations provide helpful parametrisations of new objects, e.g. points of the Poincaré extended space [42], loxodromes [44] or continued fractions [6, 41], see Example 3 below for further details. Thus, we propose *to extend Möbius–Lie geometry and consider ensembles of cycles as its new objects*, cf. formal Defn. 5. Naturally, "old" objects—cycles—are represented by simplest one-element ensembles without any relation. This paper provides conceptual foundations of such extension and demonstrates its practical implementation as a C++ library **figure**[1]. Interestingly, the development of this library shaped the general approach, which leads to specific realisations in [41, 42, 44].
More specifically, the library **figure** manipulates ensembles of cycles (quadrics) interrelated by certain FLT-invariant geometric conditions. The code is build on top of the previous library **cycle** [30, 31, 36], which manipulates individual cycles within the GiNaC [4] computer algebra system. Thinking an ensemble as a graph, one can say that the library **cycle** deals with individual vertices (cycles), while **figure** considers edges (relations between pairs of cycles) and the whole graph. Intuitively, an interaction with the library **figure** reminds compass-and-straightedge constructions, where new lines or circles are added to a drawing one-by-one through relations to already presented objects (the line through two points, the intersection point or the circle with given centre and a point). See Example 6 of such interactive construction from the Python wrapper, which provides an analytic proof of a simple geometric statement.

It is important that both libraries are capable to work in spaces of any dimensionality and metrics with an arbitrary signatures: Euclidean, Minkowski and even degenerate. Parameters of objects can be symbolic or numeric, the latter admit calculations with exact or approximate arithmetic. Drawing routines work with any (elliptic, parabolic or hyperbolic) metric in two dimensions and the euclidean metric in three dimensions.

The mathematical formalism employed in the library **cycle** is based on Clifford algebras, which are intimately connected to fundamental geometrical and physical objects [25, 26]. Thus, it is not surprising that Clifford algebras have been already used in various geometric algorithms for a long time, for example see [16, 27, 57] and further references there. Our package deals with cycles through Fillmore–Springer–Cnops construction (FSCc) which also has a long history, see [12, § 4.1; 17; 29, § 4.2; 34; 36, § 4.2; 54, § 1.1] and section 2.1 below. Compared to a plain analytical treatment [7, Ch. 3; 50, Ch. 2], FSCc is much more efficient and conceptually coherent in dealing with FLT-invariant properties of cycles. Correspondingly, the computer code based on FSCc is easy to write and maintain.

The paper outline is as follows. In Section 2 we sketch the mathematical theory (Möbius–Lie geometry) covered by the package of the previous library **cycle** [31] and the present library **figure**. We expose the subject with some references to its history since this can facilitate further development.

Sec. 3.1 describes the principal mathematical tool used by the library **figure**. It allows to reduce a collection of various linear and quadratic equations (expressing geometrical relations like orthogonality and tangency) to a set of linear equations and *at most one* quadratic relation (8). Notably, the quadratic relation is the same in all cases, which greatly simplifies its handling. This approach is the cornerstone of the library effectiveness both in symbolic and numerical computations. In Sec. 3.2 we present several examples of ensembles, which were already used in mathematical theories [41, 42, 44], then we describe how ensembles are encoded in the present library **figure** through the functional programming framework.

Sec. 4 outlines several typical usages of the package. An example of a new statement discovered and demonstrated by the package is given in Thm. 7. In Sec. 5 we list of some further tasks, which will extend capacities and usability of the package.

All coding-related material is enclosed as appendices. App. A contains examples of the library usage starting from the very simple ones. A systematic list of callable methods is given in Apps B–D. Any of Sec. 2 or Apps A–B can serve as an entry point for a reader with respective preferences and background. Actual code of the library is collected in Apps E–F.

[1] All described software is licensed under GNU GPLv3 [19].

2. Möbius–Lie Geometry and the **cycle** Library

We briefly outline mathematical formalism of the extend Möbius–Lie geometry, which is implemented in the present package. We do not aim to present the complete theory here, instead we provide a minimal description with a sufficient amount of references to further sources. The hierarchical structure of the theory naturally splits the package into two components: the routines handling individual cycles (the library **cycle** briefly reviewed in this section), which were already introduced elsewhere [31], and the new component implemented in this work, which handles families of interrelated cycles (the library **figure** introduced in the next section).

2.1. **Möbius–Lie geometry and FSC construction.** Möbius–Lie geometry in $\mathbb{R}^n$ starts from an observation that points can be treated as spheres of zero radius and planes are the limiting case of spheres with radii diverging to infinity. Oriented spheres, planes and points are called together *cycles*. Then, the second crucial step is to treat cycles not as subsets of $\mathbb{R}^n$ but rather as points of some projective space of higher dimensionality, see [8, Ch. 3; 10; 50; 54].

To distinguish two spaces we will call $\mathbb{R}^n$ as the *point space* and the higher dimension space, where cycles are represented by points—the *cycle space*. Next important observation is that geometrical relations between cycles as subsets of the point space can be expressed in term of some indefinite metric on the cycle space. Therefore, if an indefinite metric shall be considered anyway, there is no reason to be limited to spheres in Euclidean space $\mathbb{R}^n$ only. The same approach shall be adopted for quadrics in spaces $\mathbb{R}^{pqr}$ of an arbitrary signature $p+q+r=n$, including r nilpotent elements, cf. (2) below.

A useful addition to Möbius–Lie geometry is provided by the Fillmore–Springer–Cnops construction (FSCc) [12, § 4.1; 17; 29, § 4.2; 34; 36, § 4.2; 51, § 18; 54, § 1.1]. It is a correspondence between the cycles (as points of the cycle space) and certain 2×2-matrices defined in (4) below. The main advantages of FSCc are:

(i) The correspondence between cycles and matrices respects the projective structure of the cycle space.
(ii) The correspondence is FLT covariant.
(iii) The indefinite metric on the cycle space can be expressed through natural operations on the respective matrices.

The last observation is that for restricted groups of Möbius transformations the metric of the cycle space may not be completely determined by the metric of the point space, see [30; 34; 36, § 4.2] for an example in two-dimensional space.

FSCc is useful in consideration of the Poincaré extension of Möbius maps [42], loxodromes [44] and continued fractions [41]. In theoretical physics FSCc nicely describes conformal compactifications of various space-time models [24; 32; 36, § 8.1]. Regretfully, FSCc have not yet propagated back to the most fundamental case of complex numbers, cf. [55, § 9.2] or somewhat cumbersome techniques used in [7, Ch. 3]. Interestingly, even the founding fathers were not always strict followers of their own techniques, see [18].

We turn now to the explicit definitions.

2.2. **Clifford algebras, FLT transformations, and Cycles.** We describe here the mathematics behind the the first library called **cycle**, which implements fundamental geometrical relations between quadrics in the space $\mathbb{R}^{pqr}$ with the dimensionality $n=p+q+r$ and metric $x_1^2+\ldots+x_p^2-x_{p+1}^2-\ldots-x_{p+q}^2$. A version simplified for complex numbers only can be found in [41, 42, 44].

The Clifford algebra $\mathcal{C}\!\ell(p,q,r)$ is the associative unital algebra over $\mathbb{R}$ generated by the elements $e_1,\ldots,e_n$ satisfying the following relation:

$$e_ie_j=-e_je_i\,,\qquad \text{and}\qquad e_i^2=\begin{cases}-1, & \text{if } 1\le i\le p;\\ 1, & \text{if } p+1\le i\le p+q;\\ 0, & \text{if } p+q+1\le i\le p+q+r.\end{cases}\tag{2}$$

It is common [12, 14, 25, 26, 51] to consider mainly Clifford algebras $\mathcal{C}\!\ell(n)=\mathcal{C}\!\ell(n,0,0)$ of the Euclidean space or the algebra $\mathcal{C}\!\ell(p,q)=\mathcal{C}\!\ell(p,q,0)$ of the pseudo-Euclidean (Minkowski) spaces. However, Clifford algebras $\mathcal{C}\!\ell(p,q,r)$, $r>0$ with nilpotent generators $e_i^2=0$ correspond to interesting geometry [34, 36, 48, 58] and physics [20–22, 37, 38, 43] as well. Yet, the geometry with idempotent units in spaces with dimensionality $n>2$ is still not sufficiently elaborated.

An element of $\mathcal{C}\!\ell(p,q,r)$ having the form $x=x_1e_1+\ldots+x_ne_n$ can be associated with the vector $(x_1,\ldots,x_n)\in\mathbb{R}^{pqr}$. The *reversion* $a\mapsto a^*$ in $\mathcal{C}\!\ell(p,q,r)$ [12, (1.19(ii))] is defined on vectors by $x^*=x$ and extended to other elements by the relation $(ab)^*=b^*a^*$. Similarly the *conjugation* is defined on vectors by $\bar{x}=-x$ and the relation $\overline{ab}=\bar{b}\bar{a}$. We also use the notation $|a|^2=a\bar{a}$ for any product a of vectors. An important observation is that any non-zero $x\in\mathbb{R}^{n00}$ has a multiplicative inverse: $x^{-1}=\frac{\bar{x}}{|x|^2}$. For a 2×2-matrix $M=\begin{pmatrix}a & b\\ c & d\end{pmatrix}$ with Clifford entries we define, cf. [12, (4.7)]

$$\bar{M}=\begin{pmatrix}d^* & -b^*\\ -c^* & a^*\end{pmatrix}\qquad\text{and}\qquad M^*=\begin{pmatrix}\bar{d} & \bar{b}\\ \bar{c} & \bar{a}\end{pmatrix}.\tag{3}$$

Then $M\bar{M}=\delta I$ for the *pseudodeterminant* $\delta:=ad^*-bc^*$.

Quadrics in $\mathbb{R}^{pq}$—which we continue to call cycles—can be associated to 2×2 matrices through the FSC construction [12, (4.12); 17; 36, § 4.4]:

$$k\bar{x}x-l\bar{x}-x\bar{l}+m=0\quad\leftrightarrow\quad C=\begin{pmatrix}l & m\\ k & \bar{l}\end{pmatrix},\tag{4}$$

where $k,m\in\mathbb{R}$ and $l\in\mathbb{R}^{pq}$. For brevity we also encode a cycle by its coefficients (k,l,m). A justification of (4) is provided by the identity:

$$\begin{pmatrix}1 & \bar{x}\end{pmatrix}\begin{pmatrix}l & m\\ k & \bar{l}\end{pmatrix}\begin{pmatrix}x\\ 1\end{pmatrix}=kx\bar{x}-l\bar{x}-x\bar{l}+m,\qquad\text{since } \bar{x}=-x \text{ for } x\in\mathbb{R}^{pq}.$$

The identification is also FLT-covariant in the sense that the transformation (1) associated with the matrix $M=\begin{pmatrix}a & b\\ c & d\end{pmatrix}$ sends a cycle C to the cycle MCM^* [12, (4.16)]. We define the FLT-invariant inner product of cycles C_1 and

C_2 by the identity

$$\langle C_1, C_2 \rangle = \Re\,\mathrm{tr}(C_1 C_2)\,, \tag{5}$$

where $\Re$ denotes the scalar part of a Clifford number. This definition in term of matrices immediately implies that the inner product is FLT-invariant. The explicit expression in terms of components of cycles $C_1 = (k_1, l_1, m_1)$ and $C_2 = (k_2, l_2, m_2)$ is also useful sometimes:

$$\langle C_1, C_2 \rangle = l_1 l_2 + \bar{l}_1 \bar{l}_2 + m_1 k_2 + m_2 k_1\,. \tag{6}$$

As usual, the relation $\langle C_1, C_2 \rangle = 0$ is called the *orthogonality* of cycles C_1 and C_2. In most cases it corresponds to orthogonality of quadrics in the point space. More generally, most of FLT-invariant relations between quadrics may be expressed in terms FLT-invariant inner product (5). For the full description of methods on individual cycles, which are implemented in the library **cycle**, see the respective documentation [31].

Remark 1. Since cycles are elements of the projective space, the following *normalised cycle product*:

$$[C_1, C_2] := \frac{\langle C_1, C_2 \rangle}{\sqrt{\langle C_1, C_1 \rangle \langle C_2, C_2 \rangle}} \tag{7}$$

is more meaningful than the cycle product (5) itself. Note that, $[C_1, C_2]$ is defined only if neither C_1 nor C_2 is a zero-radius cycle (i.e. a point). Also, the normalised cycle product is $\mathrm{GL}_2(\mathbb{C})$-invariant in comparison to $\mathrm{SL}_2(\mathbb{C})$-invariance of (5).

We finish this brief review of the library **cycle** by pointing to its light version written in Asymptote language [23] and distributed together with the paper [44]. Although the light version mostly inherited API of the library **cycle**, there are some significant limitations caused by the absence of GiNaC support:

(i) there is no symbolic computations of any sort;
(ii) the light version works in two dimensions only;
(iii) only elliptic metrics in the point and cycle spaces are supported.

On the other hand, being integrated with Asymptote the light version simplifies production of illustrations, which are its main target.

3. Ensembles of Interrelated Cycles and the **figure** Library

The library **figure** has an ability to store and resolve the system of geometric relations between cycles. We explain below some mathematical foundations, which greatly simplify this task.

3.1. Connecting quadrics and cycles.

We need a vocabulary, which translates geometric properties of quadrics on the point space to corresponding relations in the cycle space. The key ingredient is the cycle product (5)–(6), which is linear in each cycles' parameters. However, certain conditions, e.g. tangency of cycles, involve polynomials of cycle products and thus are non-linear. For a successful algorithmic implementation, the following observation is important: *all non-linear conditions below can be linearised if the additional quadratic condition of normalisation type is imposed*:

$$\langle C, C \rangle = \pm 1. \tag{8}$$

This observation in the context of the Apollonius problem was already made in [18]. Conceptually the present work has a lot in common with the above mentioned paper of Fillmore and Springer, however a reader need to be warned that our implementation is totally different (and, interestingly, is more closer to another paper [17] of Fillmore and Springer).

Remark 2. Interestingly, the method of order reduction for algebraic equations is conceptually similar to the method of order reduction of differential equations used to build a geometric dynamics of quantum states in [1].

Here is the list of relations between cycles implemented in the current version of the library **figure**.

(i) A quadric is flat (i.e. is a hyperplane), that is, its equation is linear. Then, either of two equivalent conditions can be used:
  (a) k component of the cycle vector is zero;
  (b) the cycle is orthogonal $\langle C_1, C_\infty \rangle = 0$ to the "zero-radius cycle at infinity" $C_\infty = (0, 0, 1)$.
(ii) A quadric on the plane represents a line in Lobachevsky-type geometry if it is orthogonal $\left\langle C_1, C_{\mathbb{R}} \right\rangle = 0$ to the real line cycle $C_{\mathbb{R}}$. A similar condition is meaningful in higher dimensions as well.
(iii) A quadric C represents a point, that is, it has zero radius at given metric of the point space. Then, the determinant of the corresponding FSC matrix is zero or, equivalently, the cycle is self-orthogonal (isotropic): $\langle C, C \rangle = 0$. Naturally, such a cycle cannot be normalised to the form (8).
(iv) Two quadrics are orthogonal in the point space $\mathbb{R}^{pq}$. Then, the matrices representing cycles are orthogonal in the sense of the inner product (5).

(v) Two cycles C and $\tilde{C}$ are tangent. Then we have the following quadratic condition:

$$\left\langle C,\tilde{C}\right\rangle^2 = \langle C,C\rangle \left\langle \tilde{C},\tilde{C}\right\rangle \qquad \left(\text{ or } \left[C,\tilde{C}\right]=\pm 1\right). \tag{9}$$

With the assumption, that the cycle C is normalised by the condition (8), we may re-state this condition in the relation, which is linear to components of the cycle C:

$$\left\langle C,\tilde{C}\right\rangle = \pm\sqrt{\left\langle \tilde{C},\tilde{C}\right\rangle}. \tag{10}$$

Different signs here represent internal and outer touch.

(vi) Inversive distance θ of two (non-isotropic) cycles is defined by the formula:

$$\left\langle C,\tilde{C}\right\rangle = \theta\sqrt{\langle C,C\rangle}\sqrt{\left\langle \tilde{C},\tilde{C}\right\rangle} \tag{11}$$

In particular, the above discussed orthogonality corresponds to $\theta=0$ and the tangency to $\theta=\pm 1$. For intersecting spheres θ provides the cosine of the intersecting angle. For other metrics, the geometric interpretation of inversive distance shall be modified accordingly.

If we are looking for a cycle C with a given inversive distance θ to a given cycle $\tilde{C}$, then the normalisation (8) again turns the defining relation (11) into a linear with respect to parameters of the unknown cycle C.

(vii) A generalisation of Steiner power d of two cycles is defined as, cf. [18, § 1.1]:

$$d = \left\langle C,\tilde{C}\right\rangle + \sqrt{\langle C,C\rangle}\sqrt{\left\langle \tilde{C},\tilde{C}\right\rangle}, \tag{12}$$

where both cycles C and $\tilde{C}$ are k-normalised, that is the coefficient in front the quadratic term in (4) is 1. Geometrically, the generalised Steiner power for spheres provides the square of tangential distance. However, this relation is again non-linear for the cycle C.

If we replace C by the cycle $C_1 = \frac{1}{\sqrt{\langle C,C\rangle}}C$ satisfying (8), the identity (12) becomes:

$$d\cdot k = \left\langle C_1,\tilde{C}\right\rangle + \sqrt{\left\langle \tilde{C},\tilde{C}\right\rangle}, \tag{13}$$

where $k=\frac{1}{\sqrt{\langle C,C\rangle}}$ is the coefficient in front of the quadratic term of C_1. The last identity is linear in terms of the coefficients of C_1.

Summing up: if an unknown cycle is connected to already given cycles by any combination of the above relations, then all conditions can be expressed as *a system of linear equations for coefficients of the unknown cycle and at most one quadratic equation* (8).

3.2. **Figures as families of cycles—functional approach.** We start from some examples of ensembles of cycles, which conveniently describe FLT-invariant families of objects.

Example 3. (i) The Poincaré extension of Möbius transformations from the real line to the upper half-plane of complex numbers is described by a triple of cycles $\{C_1, C_2, C_3\}$ such that:
(a) C_1 and C_2 are orthogonal to the real line;
(b) $\langle C_1, C_2\rangle^2 \leq \langle C_1, C_1\rangle \langle C_2, C_2\rangle$;
(c) C_3 is orthogonal to any cycle in the triple including itself.
A modification [41] with ensembles of four cycles describes an extension from the real line to the upper half-plane of complex, dual or double numbers. The construction can be generalised to arbitrary dimensions [5].
(ii) A parametrisation of loxodromes is provided by a triple of cycles $\{C_1, C_2, C_3\}$ such that, cf. [44] and Fig. 1:
(a) C_1 is orthogonal to C_2 and C_3;
(b) $\langle C_2, C_3\rangle^2 \geq \langle C_2, C_2\rangle \langle C_3, C_3\rangle$.
Then, main invariant properties of Möbius–Lie geometry, e.g. tangency of loxodromes, can be expressed in terms of this parametrisation [44].
(iii) A continued fraction is described by an infinite ensemble of cycles (C_k) such that [6]:
(a) All C_k are touching the real line (i.e. are *horocycles*);
(b) (C_1) is a horizontal line passing through $(0, 1)$;
(c) C_{k+1} is tangent to C_k for all $k > 1$.
This setup was extended in [41] to several similar ensembles. The key analytic properties of continued fractions—their convergence—can be linked to asymptotic behaviour of such an infinite ensemble [6].
(iv) A remarkable relation exists between discrete integrable systems and Möbius geometry of finite configurations of cycles [9, 45–47, 53]. It comes from "reciprocal force diagrams" used in 19th-century statics, starting with J.C. Maxwell. It is demonstrated in that the geometric compatibility of reciprocal figures corresponds to the algebraic compatibility of linear systems defining these configurations. On the other hand, the algebraic compatibility of linear systems lies in the basis of integrable systems. In particular [45, 46], important integrability conditions encapsulate nothing but a fundamental theorem of ancient Greek geometry.
(v) An important example of an infinite ensemble is provided by the representation of an arbitrary wave as the envelope of a continuous family of spherical waves. A finite subset of spheres can be used as an approximation to the infinite family. Then, discrete snapshots of time evolution of sphere wave packets represent a FLT-covariant ensemble of cycles [3]. Further physical applications of FLT-invariant ensembles may be looked at [28].

One can easily note that the above parametrisations of some objects by ensembles of cycles are not necessary unique. Naturally, two ensembles parametrising the same object are again connected by FLT-invariant conditions. We presented only one example here, cf. [44].

Example 4. Two non-degenerate triples $\{C_1, C_2, C_3\}$ and $\{\tilde{C}_1, \tilde{C}_2, \tilde{C}_3\}$ parameterise the same loxodrome as in Ex. 3(ii) if and only if all the following conditions are satisfied:
(i) Pairs $\{C_2, C_3\}$ and $\{\tilde{C}_2, \tilde{C}_3\}$ span the same hyperbolic pencil. That is cycles $\tilde{C}_2$ and $\tilde{C}_3$ are linear combinations of C_2 and C_3 and vise versa.
(ii) Pairs $\{C_2, C_3\}$ and $\{\tilde{C}_2, \tilde{C}_3\}$ have the same normalised cycle product (7):

$$[C_2, C_3] = \left[\tilde{C}_2, \tilde{C}_3\right]. \tag{14}$$

(iii) The elliptic-hyperbolic identity holds:

$$\frac{\operatorname{arccosh}\left[C_j, \tilde{C}_j\right]}{\operatorname{arccosh}\,[C_2, C_3]} \equiv \frac{1}{2\pi} \arccos\left[C_1, \tilde{C}_1\right] \pmod 1, \tag{15}$$

where j is either 2 or 3.

Various triples of cycles parametrising the same loxodrome are animated on Fig. 1.

The respective equivalence relation for parametrisation of Poincaré extension from Ex. 3(i) is provided in [42, Prop. 12]. These examples suggest that one can expand the subject and applicability of Möbius–Lie geometry through the following formal definition.

Definition 5. Let X be a set, $R \subset X \times X$ be an oriented graph on X and f be a function on R with values in FLT-invariant relations from § 3.1. Then (R, f)-*ensemble* is a collection of cycles $\{C_j\}_{j\in X}$ such that

$$C_i \text{ and } C_j \text{ are in the relation } f(i, j) \text{ for all } (i, j) \in R.$$

For a fixed FLT-invariant equivalence relations $\sim$ on the set $\mathcal{E}$ of all (R, f)-ensembles, *the extended Möbius–Lie geometry* studies properties of cosets $\mathcal{E}/\sim$.

This definition can be suitably modified for
(i) ensembles with relations of more then two cycles; and/or
(ii) ensembles parametrised by continuous sets X, cf. wave envelopes in Ex. 3(v).

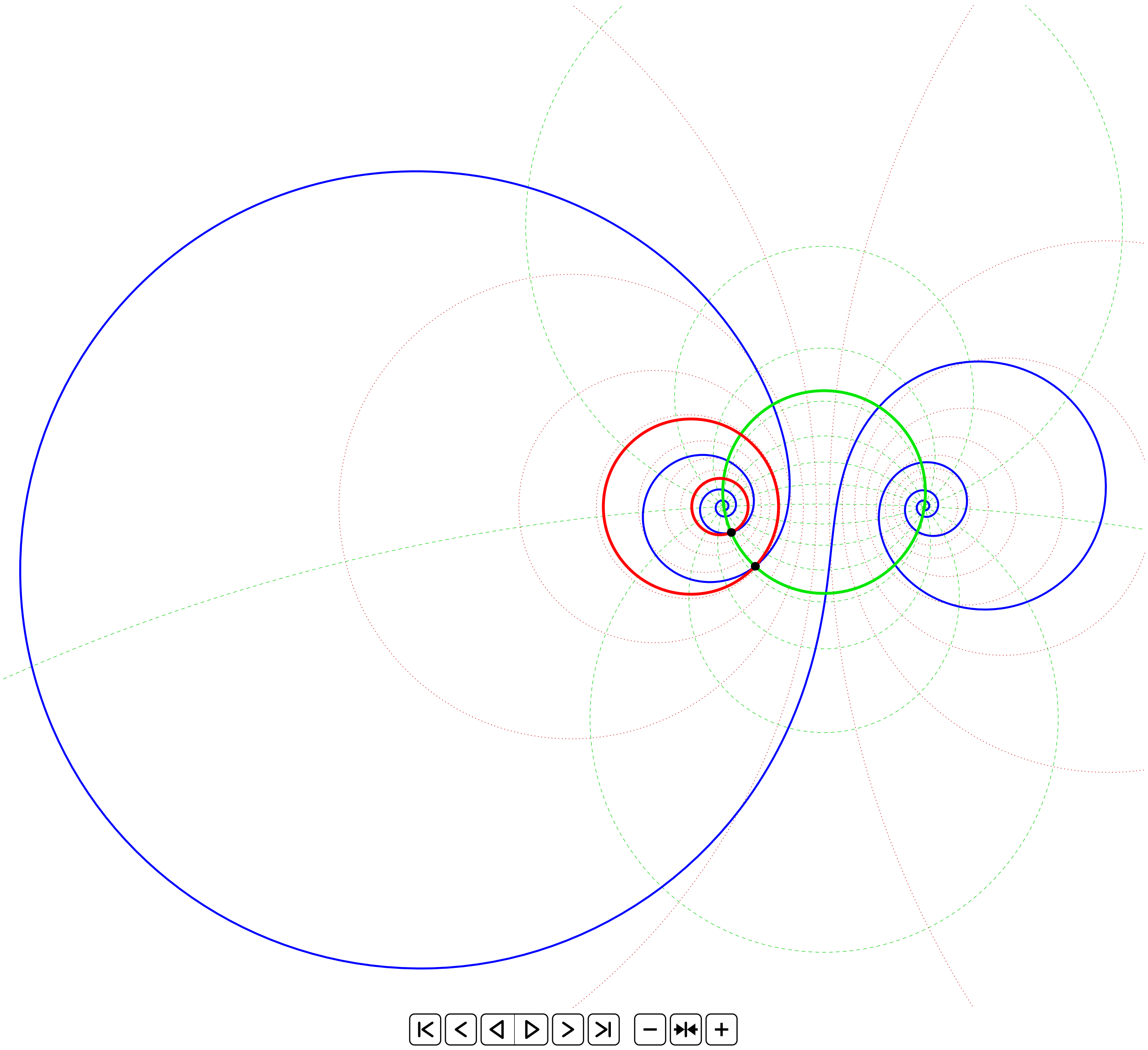

FIGURE 1. Animated graphics of equivalent three-cycle parametrisations of a loxodrome. The green cycle is C_1, two red circles are C_2 and C_3.

The above extension was developed along with the realisation the library **figure** within the *functional programming* framework. More specifically, an object from the **class figure** stores defining relations, which link new cycles to the previously introduced ones. This also may be treated as classical geometric compass-and-straightedge constructions, where new lines or circles are drawn through already existing elements. If requested, an explicit evaluation of cycles' parameters from this data may be attempted.

To avoid "chicken or the egg" dilemma all cycles are stored in a hierarchical structure of generations, numbered by integers. The basic principles are:

(i) Any explicitly defined cycle (i.e., a cycle which is not related to any previously known cycle) is placed into generation-0;

(ii) Any new cycle defined by relations to *previous* cycles from generations k_1, k_2, …, k_n is placed to the generation k calculated as:

$$k = \max(k_1, k_2, \ldots, k_n) + 1. \tag{16}$$

This rule does not forbid a cycle to have a relation to itself, e.g. isotropy (self-orthogonality) condition $\langle C, C\rangle = 0$, which specifies point-like cycles, cf. relation (iii) in § 3.1. In fact, this is the only allowed type of relations to cycles in the same (not even speaking about younger) generations.

There are the following alterations of the above rules:

(i) From the beginning, every figure has two pre-defined cycles: the real line (hyperplane) $C_{\mathbb{R}}$, and the zero radius cycle at infinity $C_\infty = (0, 0, 1)$. These cycles are required for relations (i) and (ii) from the previous subsection. As predefined cycles, $C_{\mathbb{R}}$ and C_∞ are placed in negative-numbered generations defined by the macros *REAL_LINE_GEN* and *INFINITY_GEN*.

(ii) If a point is added to generation-0 of a figure, then it is represented by a zero-radius cycle with its centre at the given point. Particular parameter of such cycle dependent on the used metric, thus this cycle is not considered as explicitly defined. Thereafter, the cycle shall have some parents at a negative-numbered generation defined by the macro *GHOST_GEN*.

A figure can be in two different modes: *freeze* or *unfreeze*, the second is default. In the *unfreeze* mode an addition of a new cycle by its relation prompts an evaluation of its parameters. If the evaluation was successful then the obtained parameters are stored and will be used in further calculations for all children of the cycle. Since many relations (see the previous Subsection) are connected to quadratic equation (8), the solutions may come in pairs. Furthermore, if the number or nature of conditions is not sufficient to define the cycle uniquely (up to natural quadratic multiplicity), then the cycle will depend on a number of free (symbolic) variable.

There is a macro-like tool, which is called **subfigure**. Such a **subfigure** is a **figure** itself, such that its inner hierarchy of generations and relations is not visible from the current **figure**. Instead, some cycles (of any generations) of the current **figure** are used as predefined cycles of generation-0 of **subfigure**. Then only one dependent cycle of **subfigure**, which is known as result, is returned back to the current **figure**. The generation of the result is calculated from generations of input cycles by the same formula (16).

There is a possibility to test certain conditions ("are two cycles orthogonal?") or measure certain quantities ("what is their intersection angle?") for already defined cycles. In particular, such methods can be used to prove geometrical statements according to the Cartesian programme, that is replacing the synthetic geometry by purely algebraic manipulations.

Example 6. As an elementary demonstration, let us prove that if a cycle r is orthogonal to a circle a at the point C of its contact with a tangent line l, then r is also orthogonal to the line l. To simplify setup we assume that a is the unit circle. Here is the Python code:

```
1  F=figure()
2  a=F.add_cycle(cycle2D(1,[0,0],-1),"a")
3  l=symbol("l")
4  C=symbol("C")
5  F.add_cycle_rel([is_tangent_i(a),is_orthogonal(F.get_infinity()),only_reals(l)],l)
6  F.add_cycle_rel([is_orthogonal(C),is_orthogonal(a),is_orthogonal(l),only_reals(C)],C)
7  r=F.add_cycle_rel([is_orthogonal(C),is_orthogonal(a)],"r")
8  Res=F.check_rel(l,r,"cycle_orthogonal")
9  for i in range(len(Res)):
10     print "Tangent and radius are orthogonal: %s" %\
11     bool(Res[i].subs(pow(cos(wild(0)),2)==1-pow(sin(wild(0)),2)).normal())
```

The first line creates an empty figure F with the default euclidean metric. The next line explicitly uses parameters $(1, 0, 0, -1)$ of a to add it to F. Lines 3–4 define symbols l and C, which are needed because cycles with these labels are defined in lines 5–6 through some relations to themselves and the cycle a. In both cases we want to have cycles with real coefficients only and C is additionally self-orthogonal (i.e. is a zero-radius). Also, l is orthogonal to infinity (i.e. is a line) and C is orthogonal to a and l (i.e. is their common point). The tangency condition for l and self-orthogonality of C are both quadratic relations. The former has two solutions each depending on one real parameter, thus line l has two instances. Correspondingly, the point of contact C and the orthogonal cycle r through C (defined in line 7) each have two instances as well. Finally, lines 8–11 verify that every instance of l is orthogonal to the respective instance of r, this is assisted by the trigonometric substitution $\cos^2(*) = 1 - \sin^2(*)$ used for parameters of l in line 11. The output predictably is:

```
Tangent and circle r are orthogonal: True
Tangent and circle r are orthogonal: True
```

An original statement proved by the library **figure** for the first time will be considered in the next Section.

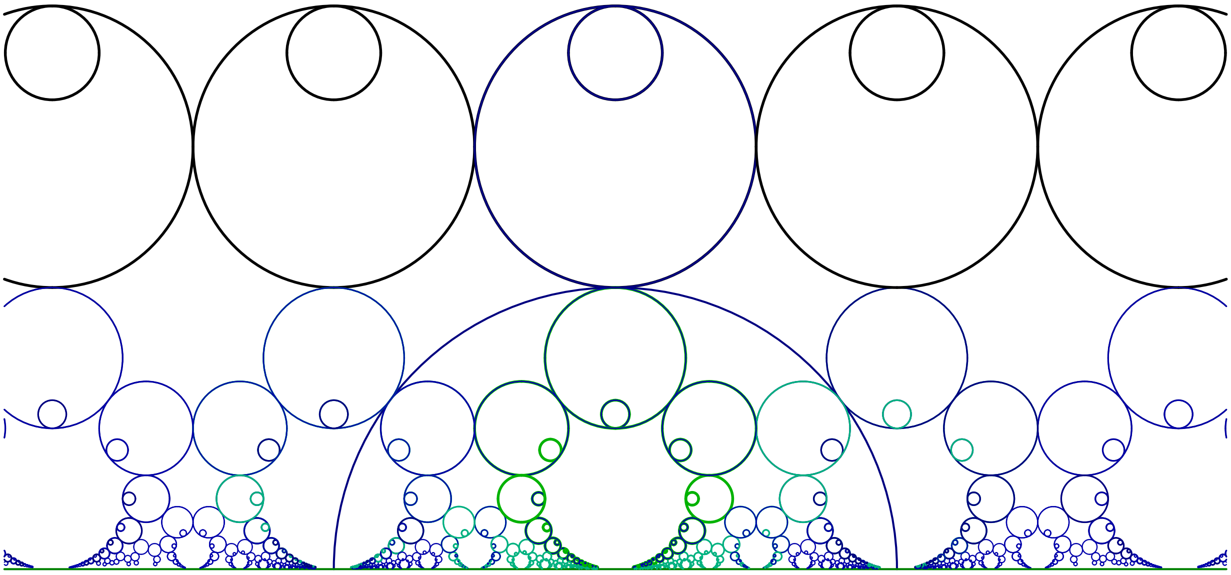

FIGURE 2. Action of the modular group on the upper half-plane.

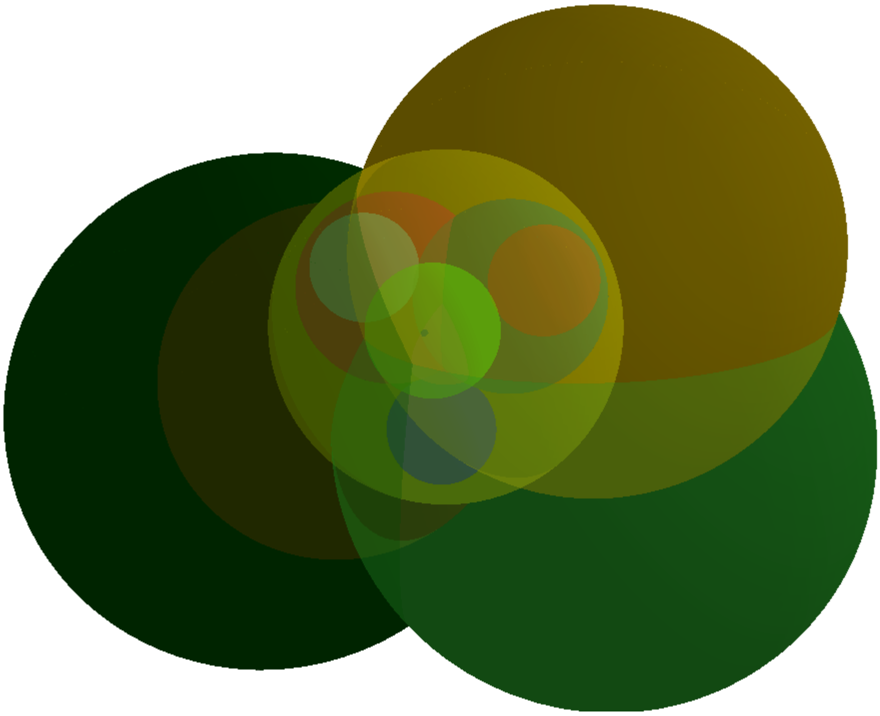

FIGURE 3. An example of Apollonius problem in three dimensions.

4. Mathematical Usage of the Library

The developed library **figure** has several different uses:

- It is easy to produce high-quality illustrations, which are fully-accurate in mathematical sence. The user is not responsible for evaluation of cycles' parameters, all computations are done by the library as soon as the figure is defined in terms of few geometrical relations. This is especially helpful for complicated images which may contain thousands of interrelated cycles. See Escher-like Fig. 2 which shows images of two circles under the modular group action [56, § 14.4], cf. A.3.
- The package can be used for computer experiments in Möbius–Lie geometry. There is a possibility to create an arrangement of cycles depending on one or several parameters. Then, for particular values of those parameters certain conditions, e.g. concurrency of cycles, may be numerically tested or graphically visualised. It is possible to create animations with gradual change of the parameters, which are especially convenient for illustrations, see Fig. 5 and [40].

- Since the library is based on the GiNaC system, which provides a symbolic computation engine, there is a possibility to make fully automatic proofs of various statements in Möbius–Lie geometry. Usage of computer-supported proofs in geometry is already an established practice [36, 49] and it is naturally to expect its further rapid growth.
- Last but not least, the combination of classical beauty of Lie sphere geometry and modern computer technologies is a useful pedagogical tool to widen interest in mathematics through visual and hands-on experience.

Computer experiments are especially valuable for Lie geometry of indefinite or nilpotent metrics since our intuition is not elaborated there in contrast to the Euclidean space [30, 33, 34]. Some advances in the two-dimensional space were achieved recently [36, 48], however further developments in higher dimensions are still awaiting their researchers.

As a non-trivial example of automated proof accomplished by the **figure** library for the first time, we present a FLT-invariant version of the classical nine-point theorem [13, § 1.8; 50, § I.1], cf. Fig. 4(a):

Theorem 7 (Nine-point cycle). *Let ABC be an arbitrary triangle with the orthocenter (the points of intersection of three altitudes) H, then the following nine points belongs to the same cycle, which may be a circle or a hyperbola:*

(i) *Foots of three altitudes, that is points of pair-wise intersections AB and CH, AC and BH, BC and AH.*
(ii) *Midpoints of sides AB, BC and CA.*
(iii) *Midpoints of intervals AH, BH and CH.*

There are many further interesting properties, e.g. nine-point circle is externally tangent to that triangle three excircles and internally tangent to its incircle as it seen from Fig. 4(a).

To adopt the statement for cycles geometry we need to find a FLT-invariant meaning of the midpoint A_m of an interval BC, because the equality of distances BA_m and A_mC is not FLT-invariant. The definition in cycles geometry can be done by either of the following equivalent relations:

- The midpoint A_m of an interval BC is defined by the cross-ratio $\frac{BA_m}{CA_m} : \frac{BI}{CI} = 1$, where I is the point at infinity.
- We construct the midpoint A_m of an interval BC as the intersection of the interval and the line orthogonal to BC and to the cycle, which uses BC as its diameter. The latter condition means that the cycle passes both points B and C and is orthogonal to the line BC.

Both procedures are meaningful if we replace the point at infinity I by an arbitrary fixed point N of the plane. In the second case all lines will be replaced by cycles passing through N, for example the line through B and C shall be replaced by a cycle through B, C and N. If we similarly replace "lines" by "cycles passing through N" in Thm. 7 it turns into a valid FLT-invariant version, cf. Fig. 4(b). Some additional properties, e.g. the tangency of the nine-points circle to the ex-/in-circles, are preserved in the new version as well. Furthermore, we can illustrate the connection between two versions of the theorem by an animation, where the infinity is transformed to a finite point N by a continuous one-parameter group of FLT, see. Fig. 5 and further examples at [40].

It is natural to test the nine-point theorem in the hyperbolic and the parabolic spaces. Fortunately, it is very easy under the given implementation: we only need to change the defining metric of the point space, this can be done for an already defined figure, see A.5. The corresponding figures Fig. 4(c) and (d) suggest that the hyperbolic version of the theorem is still true in the plain and even FLT-invariant forms. We shall clarify that the hyperbolic version of the Thm. 7 specialises the nine-point conic of a complete quadrilateral [11, 15]: in addition to the existence of this conic, our theorem specifies its type for this particular arrangement as equilateral hyperbola with the vertical axis of symmetry.

The computational power of the package is sufficient not only to hint that the new theorem is true but also to make a complete proof. To this end we define an ensemble of cycles with exactly same interrelations, but populate the generation-0 with points A, B and C with symbolic coordinates, that is, objects of the GiNaC **class realsymbol**. Thus, the entire figure defined from them will be completely general. Then, we may define the hyperbola passing through three bases of altitudes and check by the symbolic computations that this hyperbola passes another six "midpoints" as well, see A.6 .

In the parabolic space the nine-point Thm. 7 is not preserved in this manner. It is already observed [2, 33–36, 38, 42, 48], that the degeneracy of parabolic metric in the point space requires certain revision of traditional definitions. The parabolic variation of nine-point theorem may prompt some further considerations as well. An expanded discussion of various aspects of the nine-point construction shall be the subject of a separate paper.

5. To Do List

The library is still under active development. Along with continuous bug fixing there is an intention to extend both functionality and usability. Here are several nearest tasks planned so far:

- Expand class **subfigure** in a way suitable for encoding loxodromes and other objects of an extended Möbius–Lie geometry [42, 44].
- Add non-point transformations, extending the package to Lie sphere geometry.
- Add a method which will apply a FLT to the entire figure.
- Provide an effective parametrisation of solutions of a single quadratics condition.
- Expand drawing facilities in three dimensions to hyperboloids and paraboloids.

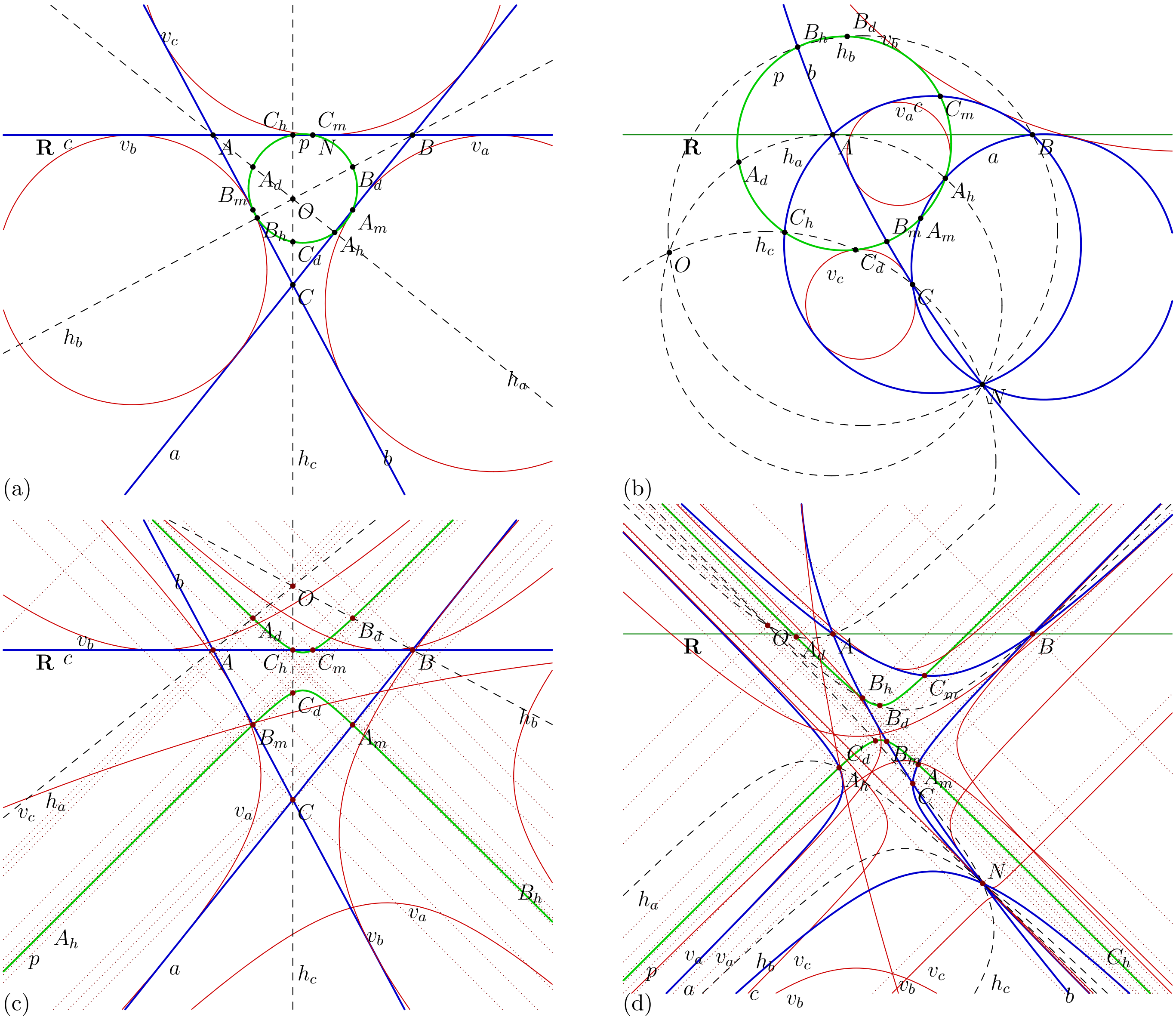

FIGURE 4. The illustration of the conformal nine-points theorem. The left column is the statement for a triangle with straight sides (the point N is at infinity), the right column is its conformal version (the point N is at the finite part). The first row show the elliptic point space, the second row—the hyperbolic point space. Thus, the top-left picture shows the traditional theorem, three other pictures—its different modifications.

- Maintain and improve the Graphical User Interface which makes the library accessible to users without programming skills.
- Investigate cloud computing options which can free a user from the burden of software installation.

Being an open-source project the library is open for contributions and suggestions of other developers and users.

Acknowledgement

I am grateful to Prof. Jay P. Fillmore for stimulating discussion, which enriched the library **figure**. Cameron Kumar wrote .

The University of Leeds provided a generous summer internship to work on Graphical User Interface to the library, which was initiated by Luke Hutton with skills and enthusiasm. Cameron Kumar wrote the initial version of a 3D cycle visualiser as a part of his BSc project at the University of Leeds.

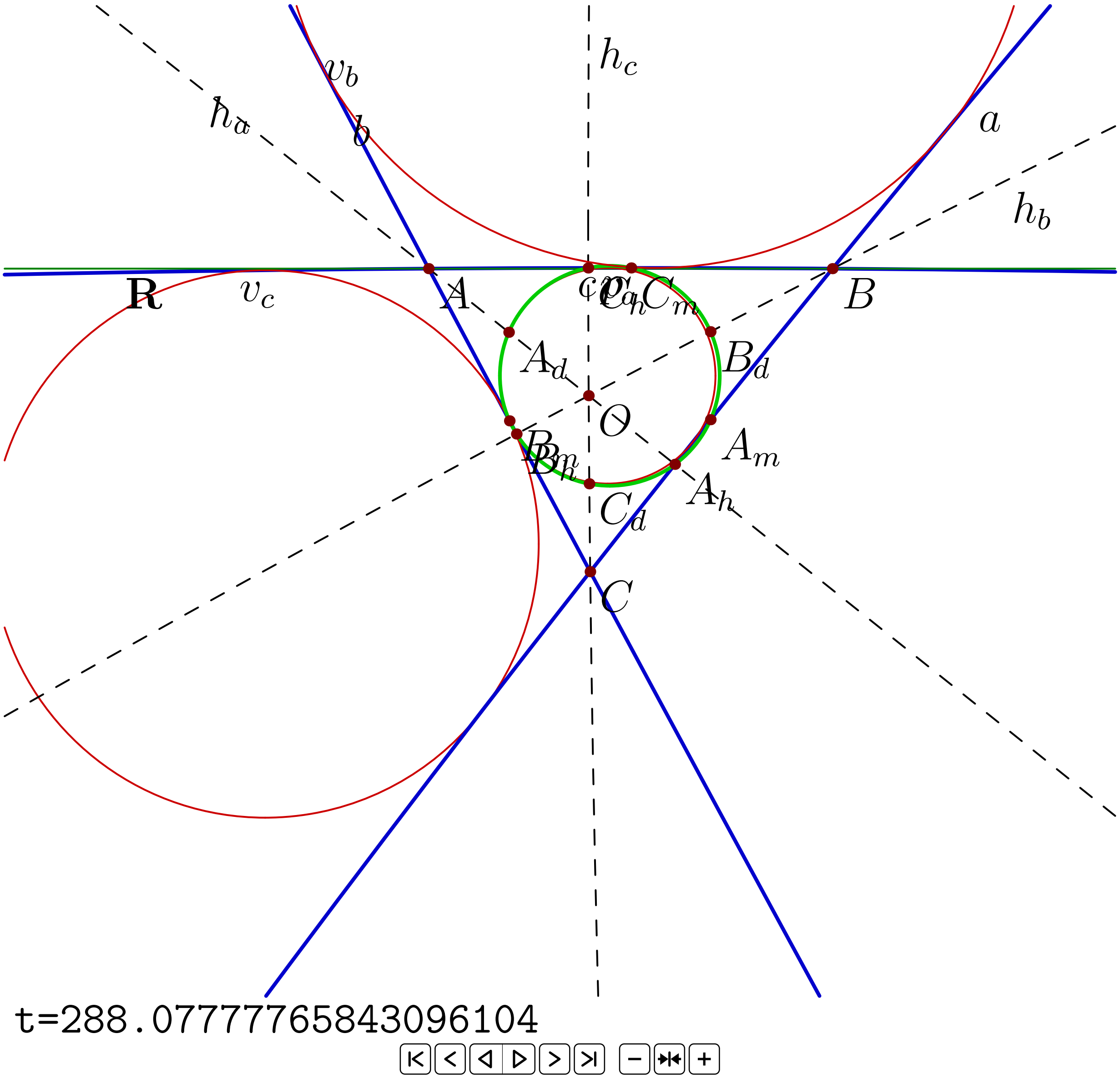

FIGURE 5. Animated transition between the classical and conformal versions of the nine-point theorem. Use control buttons to activate it. You may need Adobe Acrobat Reader for this feature.

REFERENCES

[1] F. Almalki and V. V. Kisil. Geometric dynamics of a harmonic oscillator, non-admissible mother wavelets and squeezed states:23, 2018. E-print: arXiv:1805.01399. ↑5
[2] D. E. Barrett and M. Bolt. Laguerre arc length from distance functions. *Asian J. Math.*, **14** (2):213–233, 2010. ↑11
[3] H. Bateman. *The mathematical analysis of electrical and optical wave-motion on the basis of Maxwell's equations*. Dover Publications, Inc., New York, 1955. ↑7
[4] C. Bauer, A. Frink, and R. Kreckel. Introduction to the GiNaC framework for symbolic computation within the C++ programming language. *J. Symbolic Computation*, **33** (1):1–12, 2002. Web: http://www.ginac.de/. E-print: arXiv:cs/0004015. ↑2
[5] A. F. Beardon. *The geometry of discrete groups*. Graduate Texts in Mathematics, vol. 91. Springer-Verlag, New York, 1995. Corrected reprint of the 1983 original. ↑7
[6] A. F. Beardon and I. Short. A geometric representation of continued fractions. *Amer. Math. Monthly*, **121** (5):391–402, 2014. ↑2, 7
[7] W. Benz. *Classical geometries in modern contexts. Geometry of real inner product spaces*. Birkhäuser Verlag, Basel, Second edition, 2007. ↑2, 4
[8] W. Benz. A fundamental theorem for dimension-free Möbius sphere geometries. *Aequationes Math.*, **76** (1–2):191–196, 2008. ↑4
[9] A. I. Bobenko and W. K. Schief. Circle complexes and the discrete CKP equation. *Int. Math. Res. Not. IMRN*, **5**:1504–1561, 2017. ↑7
[10] T. E. Cecil. *Lie sphere geometry: With applications to submanifolds*. Universitext. Springer, New York, Second, 2008. ↑2, 4
[11] Z. Cerin and G. M. Gianella. On improvements of the butterfly theorem. *Far East J. Math. Sci. (FJMS)*, **20** (1):69–85, 2006. ↑11
[12] J. Cnops. *An introduction to Dirac operators on manifolds*. Progress in Mathematical Physics, vol. 24. Birkhäuser Boston Inc., Boston, MA, 2002. ↑2, 4, 5
[13] H. S. M. Coxeter and S. L. Greitzer. *Geometry revisited*. Random House, New York, 1967. Zbl0166.16402. ↑11
[14] 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). ↑4
[15] M. DeVilliers. The nine-point conic: a rediscovery and proof by computer. *International Journal of Mathematical Education in Science and Technology*, **37** (1):7–14, 2006. ↑11

[16] L. Dorst, C. Doran, and J. Lasenby (eds.) *Applications of geometric algebra in computer science and engineering.* Birkhäuser Boston, Inc., Boston, MA, 2002. Papers from the conference (AGACSE 2001) held at Cambridge University, Cambridge, July 9–13, 2001. ↑2

[17] 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, 4, 5

[18] J. P. Fillmore and A. Springer. Determining circles and spheres satisfying conditions which generalize tangency, 2000. preprint, http://www.math.ucsd.edu/~fillmore/papers/2000LGalgorithm.pdf. ↑4, 5, 6, 28, 29, 87

[19] GNU. *General Public License (GPL).* Free Software Foundation, Inc., Boston, USA, version 3, 2007. URL: http://www.gnu.org/licenses/gpl.html. ↑2, 121

[20] N. A. Gromov. Possible quantum kinematics. II. Nonminimal case. *J. Math. Phys.*, **51** (8):083515, 12, 2010. ↑4

[21] N. A. Gromov and V. V. Kuratov. Possible quantum kinematics. *J. Math. Phys.*, **47** (1):013502, 9, 2006. ↑4

[22] N. A. Gromov. *Контракции Классических и Квантовых Групп. [contractions of classic and quanrum groups].* Moskva: Fizmatlit, 2012. ↑4

[23] 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/. ↑5, 23, 31, 36

[24] F. J. Herranz and M. Santander. Conformal compactification of spacetimes. *J. Phys. A*, **35** (31):6619–6629, 2002. E-print: arXiv:math-ph/0110019. ↑4

[25] D. Hestenes. *Space-time algebra.* Birkhäuser/Springer, Cham, Second, 2015. With a foreword by Anthony Lasenby. ↑2, 4

[26] D. Hestenes and G. Sobczyk. *Clifford algebra to geometric calculus. A unified language for mathematics and physics.* Fundamental Theories of Physics. D. Reidel Publishing Co., Dordrecht, 1984. ↑2, 4

[27] D. Hildenbrand. *Foundations of geometric algebra computing.* Geometry and Computing, vol. 8. Springer, Heidelberg, 2013. With a foreword by Alyn Rockwood. ↑2

[28] H. A. Kastrup. On the advancements of conformal transformations and their associated symmetries in geometry and theoretical physics. *Annalen der Physik*, **17** (9–10):631–690, 2008. E-print: arXiv:0808.2730. ↑7

[29] A. A. Kirillov. *A tale of two fractals.* Springer, New York, 2013. Draft: http://www.math.upenn.edu/~kirillov/MATH480-F07/tf.pdf. ↑2, 4

[30] 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. ↑2, 4, 11, 16, 23

[31] V. V. Kisil. Fillmore-Springer-Cnops construction implemented in GiNaC. *Adv. Appl. Clifford Algebr.*, **17** (1):59–70, 2007. On-line. A more recent version: E-print: arXiv:cs.MS/0512073. The latest documentation, source files, and live ISO image are at the project page: http://moebinv.sourceforge.net/. Zbl05134765. ↑2, 3, 5

[32] V. V. Kisil. Two-dimensional conformal models of space-time and their compactification. *J. Math. Phys.*, **48** (7):073506, 8, 2007. E-print: arXiv:math-ph/0611053. Zbl1144.81368. ↑4

[33] V. V. Kisil. Erlangen program at large—2: Inventing a wheel. The parabolic one. *Zb. Pr. Inst. Mat. NAN Ukr. (Proc. Math. Inst. Ukr. Ac. Sci.)*, **7** (2):89–98, 2010. E-print: arXiv:0707.4024. ↑11

[34] 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. ↑2, 4, 11, 19

[35] V. V. Kisil. Erlangen Programme at Large 3.2: Ladder operators in hypercomplex mechanics. *Acta Polytechnica*, **51** (4):44–53, 2011. E-print: arXiv:1103.1120. ↑11

[36] 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, 4, 11, 16, 23, 32, 34, 35, 38, 50, 87, 113

[37] V. V. Kisil. Is commutativity of observables the main feature, which separate classical mechanics from quantum?. *Известия Коми научного центра УрО РАН [Izvestiya Komi nauchnogo centra UrO RAN]*, **3** (11):4–9, 2012. E-print: arXiv:1204.1858. ↑4

[38] V. V. Kisil. Induced representations and hypercomplex numbers. *Adv. Appl. Clifford Algebras*, **23** (2):417–440, 2013. E-print: arXiv:0909.4464. Zbl1269.30052. ↑4, 11

[39] V. V. Kisil. An extension of Lie spheres geometry with conformal ensembles of cycles and its implementation in a GiNaC library, 2014. E-print: arXiv:1512.02960. Project page: http://moebinv.sourceforge.net/. ↑31

[40] V. V. Kisil. *MoebInv illustrations*, 2015. YouTube playlist. ↑10, 11

[41] V. V. Kisil. Remark on continued fractions, Möbius transformations and cycles. *Известия Коми научного центра УрО РАН [Izvestiya Komi nauchnogo centra UrO RAN]*, **25** (1):11–17, 2016. E-print: arXiv:1412.1457, on-line. ↑2, 4, 7

[42] V. V. Kisil. Poincaré extension of Möbius transformations. *Complex Variables and Elliptic Equations*, **62** (9):1221–1236, 2017. E-print: arXiv:1507.02257. ↑2, 4, 7, 11

[43] V. V. Kisil. Symmetry, geometry, and quantization with hypercomplex numbers. *Geometry, Integrability and Quantization*, **18**:11–76, 2017. E-print: arXiv:1611.05650. ↑4

[44] V. V. Kisil and J. Reid. Conformal parametrisation of loxodromes by triples of circles, 2018. E-print: arXiv:1802.01864. ↑2, 4, 5, 7, 11

[45] B. G. Konopelchenko and W. K. Schief. Menelaus' theorem, Clifford configurations and inversive geometry of the Schwarzian KP hierarchy. *J. Phys. A*, **35** (29):6125–6144, 2002. ↑7

[46] B. G. Konopelchenko and W. K. Schief. Reciprocal figures, graphical statics, and inversive geometry of the Schwarzian BKP hierarchy. *Stud. Appl. Math.*, **109** (2):89–124, 2002. ↑7

[47] B. G. Konopelchenko and W. K. Schief. Conformal geometry of the (discrete) Schwarzian Davey-Stewartson II hierarchy. *Glasg. Math. J.*, **47** (A):121–131, 2005. ↑7

[48] K. A. Mustafa. The groups of two by two matrices in double and dual numbers and associated Möbius transformations. *ArXiv e-prints: 1707.01349*, July 2017. E-print: arXiv:1707.01349. ↑4, 11

[49] P. Pech. *Selected topics in geometry with classical vs. computer proving.* World Scientific Publishing Co. Pte. Ltd., Hackensack, NJ, 2007. ↑11

[50] D. Pedoe. *Circles: A mathematical view.* MAA Spectrum. Mathematical Association of America, Washington, DC, 1995. Revised reprint of the 1979 edition, With a biographical appendix on Karl Feuerbach by Laura Guggenbuhl. ↑2, 4, 11

[51] I. R. Porteous. *Clifford algebras and the classical groups.* Cambridge Studies in Advanced Mathematics, vol. 50. Cambridge University Press, Cambridge, 1995. ↑4

[52] 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/. ↑31

[53] W. K. Schief and B. G. Konopelchenko. A novel generalization of Clifford's classical point-circle configuration. Geometric interpretation of the quaternionic discrete Schwarzian Kadomtsev-Petviashvili equation. *Proc. R. Soc. Lond. Ser. A Math. Phys. Eng. Sci.*, **465** (2104):1291–1308, 2009. ↑7

[54] 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, 4

[55] B. Simon. *Szegő's theorem and its descendants. Spectral theory for L^2 perturbations of orthogonal polynomials*. M. B. Porter Lectures. Princeton University Press, Princeton, NJ, 2011. ↑2, 4

[56] I. Stewart and D. Tall. *Algebraic number theory and Fermat's last theorem*. A K Peters, Ltd., Natick, MA, Third, 2002. ↑10, 18

[57] J. Vince. *Geometric algebra for computer graphics*. Springer-Verlag London, Ltd., London, 2008. ↑2

[58] 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. ↑4

Appendix A. Examples of Usage

This section presents several examples, which may be used for quick start. We begin with very elementary one, but almost all aspects of the library usage will be illustrated by the end of this section. See the beginning of Section B for installation advise. The collection of these programmes is also serving as a test suit for the library.

16a ⟨separating chunk 16a⟩≡ 16h ▷

A.1. Hello, Cycle!

Hello, Cycle! This is a minimalist example showing how to obtain a simple drawing of cycles in non-Euclidean geometry. Of course, we are starting from the library header file.

16b ⟨hello-cycle.cpp 16b⟩≡ 16d ▷

```
⟨license 121⟩
#include "figure.h"
⟨using all namespaces 16c⟩
int main(){
```

Defines:
`main`, used in chunk 88d.
Uses `figure` 16d 22e 32a 32c 38b 49a 50d 75a 80a 82b 82c 85a 86c 98c 99b 99d 100a 101a 103b 104a 105c 106c 106d 107a 109a 109c 110a.

To save keystrokes, we use the following **namespace**s.

16c ⟨using all namespaces 16c⟩≡ (16–18 20e 22c 27b 28a 30a)

```
using namespace std;
using namespace GiNaC;
using namespace MoebInv;
```

Defines:
`MoebInv`, used in chunks 42a and 52a.

We declare the figure F which will be constructed with the default elliptic metric in two dimensions.

16d ⟨hello-cycle.cpp 16b⟩+≡ ◁16b 16e▷

```
figure F;
```

Defines:
`figure`, used in chunks 16–18, 20e, 22, 27b, 28a, 30a, 36, 37, 45, 46c, 48, 50a, 52, 66, 75–84, 86, 87, 95d, 97–100, 106–108, 110, 111, and 117c.

Next we define a couple of points A and B. Every point is added to F by giving its explicit coordinates as a **lst** and a string, which will be used to label the point. The returned value is a GiNaC expression of **symbol** class, which will be used as a key of the respective point. All points are added to the zero generation.

16e ⟨hello-cycle.cpp 16b⟩+≡ ◁16d 16f▷

```
ex A=F.add_point(lst{-1,.5},"A");
ex B=F.add_point(lst{1,1.5},"B");
```

Defines:
`add_point`, used in chunks 17c and 23b.
Uses `ex` 41b 47e 47e 47e 53a.

Now we add a "line" in the Lobachevsky half-plane. It passes both points A and B and is orthogonal to the real line. The real line and the point at infinity were automatically added to F at its initialisation. The real line is accessible as *F.get_real_line*() method in **figure** class. A cycle passes a point if it is orthogonal to the cycle defined by this point. Thus, the line is defined through a list of three orthogonalities [30; 36, Defn. 6.1] (other pre-defined relations between cycles are listed in Section C). We also supply a string to label this cycle. The returned valued is a **symbol**, which is a key for this cycle.

16f ⟨hello-cycle.cpp 16b⟩+≡ ◁16e 16g▷

```
ex a=F.add_cycle_rel(lst{is_orthogonal(A),is_orthogonal(B),is_orthogonal(F.get_real_line())},"a");
```

Defines:
`add_cycle_rel`, used in chunks 17, 19–21, 23–25, 28–31, 83, 84a, 117, and 118.
`get_real_line`, used in chunk 17d.
Uses `ex` 41b 47e 47e 47e 53a and `is_orthogonal` 23c 38c.

Now, we draw our figure to a file. Its format (e.g. EPS, PDF, PNG, etc.) is determined by your default Asymptotesettings. This can be overwritten if a format is explicitly requested, see examples below. The output is shown at Figure 6.

16g ⟨hello-cycle.cpp 16b⟩+≡ ◁16f

```
F.asy_write(300,-3,3,-3,3,"lobachevsky-line");
return 0;
}
```

Defines:
`asy_write`, used in chunks 25, 26, and 29c.

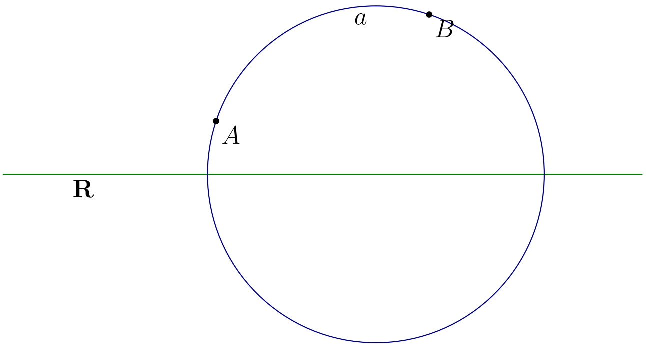

FIGURE 6. Lobachevky line.

16h ⟨separating chunk 16a⟩+≡ ◁16a 18a▷

A.2. **Animated cycle.** We use the similar construction to make an animation.

17a ⟨hello-cycle-anim.cpp 17a⟩≡ 17b▷
```
⟨license 121⟩
#include "figure.h"
⟨using all namespaces 16c⟩
int main(){
```

Defines:
main, used in chunk 88d.
Uses figure 16d 22e 32a 32c 38b 49a 50d 75a 80a 82b 82c 85a 86c 98c 99b 99d 100a 101a 103b 104a 105c 106c 106d 107a 109a 109c 110a.

It is preferable to *freeze* a figure with a symbolic parameter in order to avoid useless but expensive symbolic computations. It will be automatically *unfreeze* by *asy_animate* method below.

17b ⟨hello-cycle-anim.cpp 17a⟩+≡ ◁17a 17c▷
```
    figure F=figure().freeze();
    symbol t("t");
```
Defines:
freeze, used in chunk 26c.
unfreeze, used in chunk 104c.
Uses figure 16d 22e 32a 32c 38b 49a 50d 75a 80a 82b 82c 85a 86c 98c 99b 99d 100a 101a 103b 104a 105c 106c 106d 107a 109a 109c 110a.

This time the point A on the figure depends from the above parameter t and the point B is fixed as before.

17c ⟨hello-cycle-anim.cpp 17a⟩+≡ ◁17b 17d▷
```
    ex A=F.add_point(lst{-1*t,.5*t+.5},"A");
    ex B=F.add_point(lst{1,1.5},"B");
```

Uses add_point 16e 22g 32e 80b 80c and ex 41b 47e 47e 47e 53a.

The Lobachevsky line a is defined exactly as in the previous example but is implicitly (through A) depending on t now.

17d ⟨hello-cycle-anim.cpp 17a⟩+≡ ◁17c 17e▷
```
    ex a=F.add_cycle_rel(lst{is_orthogonal(A),is_orthogonal(B),is_orthogonal(F.get_real_line())},"a");
```

Uses add_cycle_rel 16f 23c 33a 83b, ex 41b 47e 47e 47e 53a, get_real_line 16f 49g, and is_orthogonal 23c 38c.

The new straight line b is defined as a cycle passing (orthogonal to) the point at infinity. It is accessible by *get_infinity* method.

17e ⟨hello-cycle-anim.cpp 17a⟩+≡ ◁17d 17f▷
```
    ex b=F.add_cycle_rel(lst{is_orthogonal(A),is_orthogonal(B),is_orthogonal(F.get_infinity())},"b");
```
Defines:
get_infinity, used in chunks 21a, 22a, and 29b.
Uses add_cycle_rel 16f 23c 33a 83b, ex 41b 47e 47e 47e 53a, and is_orthogonal 23c 38c.

Now we define the set of values for the parameter t which will be used for substitution into the figure.

17f ⟨hello-cycle-anim.cpp 17a⟩+≡ ◁17e 17g▷
```
    lst val;
    for (int i=0; i<40; ++i)
        val.append(t≡numeric(i+2,30));
```

Uses numeric 22d.

Finally animations in different formats are created similarly to the static picture from the previous example.

17g ⟨hello-cycle-anim.cpp 17a⟩+≡ ◁17f
```
    F.asy_animate(val,500,-2.2,3,-2,2,"lobachevsky-anim","mng");
    F.asy_animate(val,300,-2.2,3,-2,2,"lobachevsky-anim","pdf");
    return 0;
}
```
Defines:
asy_animate, used in chunk 26e.

The second command creates two files: `lobachevsky-anim.pdf` and `_lobachevsky-anim.pdf` (notice the underscore (`_`) in front of the file name, which makes the difference). The former is a stand-alone PDF file containing the desired animation. The latter may be embedded into another PDF document as shown on Fig. 7. To this end the LaTeX file need to have the command

```
\usepackage{animate}
```

in its preamble. To include the animation we use the command:

```
\animategraphics[controls]{50}{_lobachevsky-anim}{}{}
```

More options can be found in the documentation of `animate` package. Finally, the LaTeX file need to be compiled with the `pdf`LaTeX command.

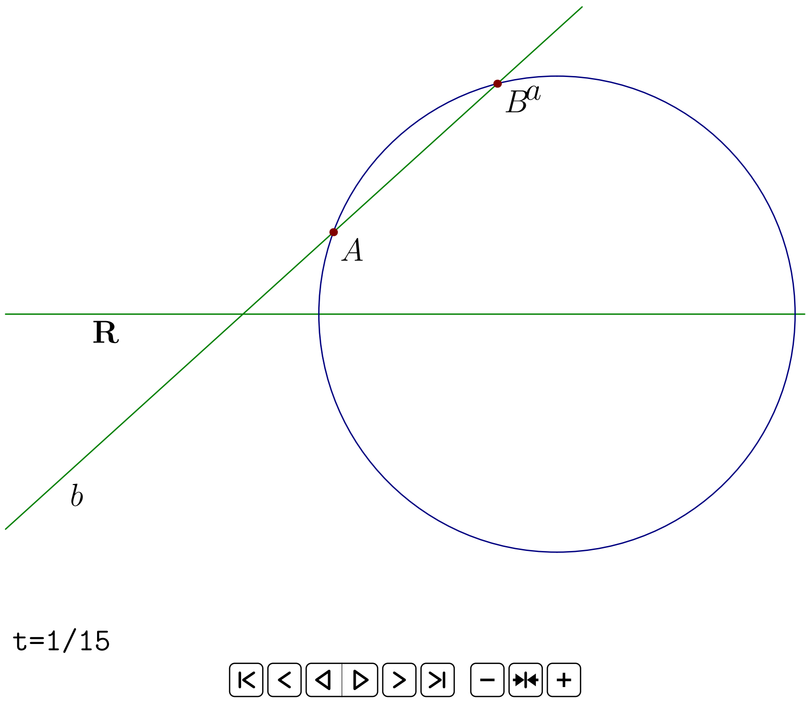

FIGURE 7. Animated Lobachevsky line: use the control buttons to run the animation. You may need Adobe Acrobat Reader for this feature.

18a ⟨separating chunk 16a⟩+≡ ◁16h 20d▷

A.3. **An illustration of the modular group action.** The library allows to build figures out of cycles which are obtained from each other by means of FLT. We are going to illustrate this by the action of the modular group $\mathrm{SL}_2(\mathbb{Z})$ on a single circle [56, § 14.4]. We repeatedly apply FLT $T = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}$ for translations and $S = \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix}$ for the inversion in the unit circle.

Here is the standard start of a programme with some additional variables being initialised.

18b ⟨modular-group.cpp 18b⟩≡ 19a ▷
```
⟨license 121⟩
#include "figure.h"
⟨using all namespaces 16c⟩
int main(){
   char buffer [50];
   int steps=3, trans=15;
   double epsilon=0.00001; // square of radius for a circle to be ignored
   figure F;
```

Defines:
main, used in chunk 88d.
Uses epsilon 53a and figure 16d 22e 32a 32c 38b 49a 50d 75a 80a 82b 82c 85a 86c 98c 99b 99d 100a 101a 103b 104a 105c 106c 106d 107a 109a 109c 110a.

We will use the metric associated to the figure, it can be extracted by *get_point_metric* method.

19a ⟨modular-group.cpp 18b⟩+≡ ◁ 18b 19b ▷
```
   ex e=F.get_point_metric();
```

Defines:
get_point_metric, used in chunk 76c.
Uses ex 41b 47e 47e 47e 53a.

Firstly, we add to the figure an initial cycle and, then, add new generations of its shifts and reflections.

19b ⟨modular-group.cpp 18b⟩+≡ ◁ 19a 19c ▷
```
   ex a=F.add_cycle(cycle2D(lst{0,numeric(3,2)},e,numeric(1,4)),"a");
   ex c=F.add_cycle(cycle2D(lst{0,numeric(11,6)},e,numeric(1,36)),"c");
   for (int i=0; i<steps;++i) {
```

Uses add_cycle 23a 32f 81d, ex 41b 47e 47e 47e 53a, and numeric 22d.

We want to shift all cycles in the previous generation. Their key are grasped by *get_all_keys* method.

19c ⟨modular-group.cpp 18b⟩+≡ ◁ 19b 19d ▷
```
      lst L=ex_to<lst>(F.get_all_keys(2*i,2*i));
      if (L.nops() == 0) {
         cout << "Terminate on iteration " << i << endl;
         break;
      }
```

Defines:
get_all_keys, used in chunks 20a, 105d, and 106d.
Uses nops 50a.

Each cycle with the collected key is shifted horizontally by an integer t in range $[-trans,trans]$. This done by *moebius_transform* relations and it is our responsibility to produce proper Clifford-valued entries to the matrix, see [34, § 2.1] for an advise.

19d ⟨modular-group.cpp 18b⟩+≡ ◁ 19c 19e ▷
```
      for (const auto& ck: L) {
         lst L1=ex_to<lst>(F.get_cycles(ck));
         for (auto x: L1) {
            for (int t=-trans; t<=trans;++t) {
               sprintf (buffer, "%s-%dt%d",ex_to<symbol>(ck).get_name().c_str(),i,t);
```

We shift initial cycles by zero in order to have their copies in the this generation.

19e ⟨modular-group.cpp 18b⟩+≡ ◁ 19d 19f ▷
```
               if ((t !=0 || i ==0)
```

To simplify the picture we are skipping circles whose radii would be smaller than the threshold.

19f ⟨modular-group.cpp 18b⟩+≡ ◁19e 19g▷

```
        ∧ ¬ ((ex_to<cycle>(x).det()-(pow(t,2)-1)∗epsilon).evalf()<0)){
        ex b=F.add_cycle_rel(moebius_transform(ck,true,
               lst{dirac_ONE(),t∗e.subs(e.op(1).op(0)≡0),0,dirac_ONE()}),buffer);
```

Defines:
moebius_transform, never used.
Uses add_cycle_rel 16f 23c 33a 83b, epsilon 53a, evalf 50a, ex 41b 47e 47e 47e 53a, op 50a, and subs 50a.

We want the colour of a cycle reflect its generation, smaller cycles also need to be drawn by a finer pen. This can be set for each cycle by *set_asy_style* method.

19g ⟨modular-group.cpp 18b⟩+≡ ◁19f 20a▷

```
        sprintf (buffer, "rgb(0,0,%.2f)+%.3f" ,1-1÷(i+1.),1÷(i+1.5));
        F.set_asy_style(b,buffer);
      }
    }
  }
}
```

Defines:
rgb, used in chunks 20, 24, 25, 28, 29b, 36, 37a, and 102b.
set_asy_style, used in chunks 20b, 23–25, 28, and 29b.

Similarly, we collect all key from the previous generation cycles to make their reflection in the unit circle.

20a ⟨modular-group.cpp 18b⟩+≡ ◁19g 20b▷

```
if (i<steps-1)
    L=ex_to<lst>(F.get_all_keys(2∗i+1,2∗i+1));
else
    L=lst{};
for (const auto& ck: L) {
    sprintf (buffer, "%ss",ex_to<symbol>(ck).get_name().c_str());
```

Uses get_all_keys 19c 33g 99a.

This time we keep things simple and are using *sl2_transform* relation, all Clifford algebra adjustments are taken by the library. The drawing style is setup accordingly.

20b ⟨modular-group.cpp 18b⟩+≡ ◁20a 20c▷

```
      ex b=F.add_cycle_rel(sl2_transform(ck,true,lst{0,-1,1,0}),buffer);
      sprintf (buffer, "rgb(0,0.7,%.2f)+%.3f" ,1-1÷(i+1.),1÷(i+1.5));
      F.set_asy_style(b,buffer);
    }
  }
```

Defines:
sl2_transform, never used.
Uses add_cycle_rel 16f 23c 33a 83b, ex 41b 47e 47e 47e 53a, rgb 19g 23d, and set_asy_style 19g 23d 37d.

Finally, we draw the picture. This time we do not want cycles label to appear, thus the last parameter *with_labels* of *asy_write* is **false**. We also want to reduce the size of Asymptote file and will not print headers of cycles, thus specifying *with_header*=**true**. The remaining parameters are explicitly assigned their default values.

20c ⟨modular-group.cpp 18b⟩+≡ ◁20b

```
  ex u=F.add_cycle(cycle2D(lst{0,0},e,numeric(1)),"u");
 F.asy_write(300,-2.17,2.17,0,2,"modular-group","pdf",default_asy,default_label,true,false,0,"rgb(0,.9,0)+4pt",true,fals
  return 0;
}
```

Defines:
asy_write, used in chunks 25, 26, and 29c.
Uses add_cycle 23a 32f 81d, ex 41b 47e 47e 47e 53a, numeric 22d, and rgb 19g 23d.

20d ⟨separating chunk 16a⟩+≡ ◁18a 22b▷

A.4. **Simple analysitcal demonstration.** The following example essentially repeats the code from Example 6. It will be better to start from a simpler case before we will consider more advanced usage in the next subsection. Also this example checks how cycle solver is handling cycles with free parameters if relations do not determine it uniquely.

The first line creates an empty figure F with the default euclidean metric.

20e ⟨figure-ortho-anlytic-proof.cpp 20e⟩≡ 20f▷
```
⟨license 121⟩
#include "figure.h"
⟨using all namespaces 16c⟩
int main(){
   figure F=figure();
```

Defines:
main, used in chunk 88d.
Uses figure 16d 22e 32a 32c 38b 49a 50d 75a 80a 82b 82c 85a 86c 98c 99b 99d 100a 101a 103b 104a 105c 106c 106d 107a 109a 109c 110a.

The next line explicitly uses parameters $(1, 0, 0, -1)$ of a to add it to F.

20f ⟨figure-ortho-anlytic-proof.cpp 20e⟩+≡ ◁20e 20g▷
```
   ex a=F.add_cycle(cycle2D(1,lst{0,0},-1),"a");
```

Uses add_cycle 23a 32f 81d and ex 41b 47e 47e 47e 53a.

Next lines define symbols l and C, which are needed because cycles with these labels are defined in next lines through some relations to themselves and the cycle a.

20g ⟨figure-ortho-anlytic-proof.cpp 20e⟩+≡ ◁20f 21a▷
```
   ex l=symbol("l");
   ex C=symbol("C");
```

Uses ex 41b 47e 47e 47e 53a and l 51c.

In both cases we want to have cycles with real coefficients only and C is additionally self-orthogonal (i.e. is a zero-radius). Also, l is orthogonal to infinity (i.e. is a line) and C is orthogonal to a and l (i.e. is their common point). The tangency condition for l and self-orthogonality of C are both quadratic relations. The former has two solutions each depending on one real parameter, thus line l has two instances.

21a ⟨figure-ortho-anlytic-proof.cpp 20e⟩+≡ ◁20g 21b▷
```
   F.add_cycle_rel(lst{is_tangent_i(a),is_orthogonal(F.get_infinity()),only_reals(l)},l);
```

Uses add_cycle_rel 16f 23c 33a 83b, get_infinity 17e 49g, is_orthogonal 23c 38c, is_tangent_i 25a 39d, l 51c, and only_reals 29b 30c 39b.

Correspondingly, the point of contact C...

21b ⟨figure-ortho-anlytic-proof.cpp 20e⟩+≡ ◁21a 21c▷
```
   F.add_cycle_rel(lst{is_orthogonal(C),is_orthogonal(a),is_orthogonal(l),only_reals(C)},C);
```

Uses add_cycle_rel 16f 23c 33a 83b, is_orthogonal 23c 38c, l 51c, and only_reals 29b 30c 39b.

...and the orthogonal cycle r through C (defined in line 7) each have two instances as well.

21c ⟨figure-ortho-anlytic-proof.cpp 20e⟩+≡ ◁21b 21d▷
```
   ex r=F.add_cycle_rel(lst{is_orthogonal(C),is_orthogonal(a)},"r");
```

Uses add_cycle_rel 16f 23c 33a 83b, ex 41b 47e 47e 47e 53a, and is_orthogonal 23c 38c.

Finally, we verify that every instance of l is orthogonal to the respective instance of r.

21d ⟨figure-ortho-anlytic-proof.cpp 20e⟩+≡ ◁21c 21e▷
```
   ex Res=F.check_rel(l, r, cycle_orthogonal);
```

Defines:
check_rel, used in chunks 22a and 25c.
Uses cycle_orthogonal 34b 113a, ex 41b 47e 47e 47e 53a, and l 51c.

This is assisted by the trigonometric substitution $\cos^2(*) = 1 - \sin^2(*)$ used for parameters of l.

21e ⟨figure-ortho-anlytic-proof.cpp 20e⟩+≡ ◁21d 21f▷
```
   for (size_t i=0; i< Res.nops(); ++i) {
      cout << "Tangent and radius are orthogonal: " << boolalpha
         << bool(ex_to<relational>(Res.op(i).subs(pow(cos(wild(0)),2)==1-pow(sin(wild(0)),2)).normal()))
         << endl;
   }
```

Uses nops 50a, op 50a, and subs 50a.

The output predictably is:

```
Tangent and circle r are orthogonal: true
Tangent and circle r are orthogonal: true
```

An additional check. We add a point $(1, 0)$ on c...

21f ⟨figure-ortho-anlytic-proof.cpp 20e⟩+≡ ◁21e 21g▷

```
ex B=F.add_cycle(cycle2D(lst{1,0}),"B");
```

Uses add_cycle 23a 32f 81d and ex 41b 47e 47e 47e 53a.

...and a generic cycle touching to c at B.

21g ⟨figure-ortho-anlytic-proof.cpp 20e⟩+≡ ◁21f 21h▷

```
ex b=symbol("b");
F.add_cycle_rel(lst{is_tangent(a),is_orthogonal(B),only_reals(b)}, b);
```

Uses add_cycle_rel 16f 23c 33a 83b, ex 41b 47e 47e 47e 53a, is_orthogonal 23c 38c, is_tangent 30c 39c, and only_reals 29b 30c 39b.

Add zero-radius cycles at the centres of a and b...

21h ⟨figure-ortho-anlytic-proof.cpp 20e⟩+≡ ◁21g 21i▷

```
ex Ca=F.add_cycle(cycle2D(ex_to<lst>(ex_to<cycle2D>(F.get_cycles(a).op(0)).center())),"Ca");
ex Cb=F.add_cycle(cycle2D(ex_to<lst>(ex_to<cycle2D>(F.get_cycles(b).op(0)).center())),"Cb");
```

Uses add_cycle 23a 32f 81d, ex 41b 47e 47e 47e 53a, and op 50a.

...and then a cycle passing two centres and the contact point.

21i ⟨figure-ortho-anlytic-proof.cpp 20e⟩+≡ ◁21h 22a▷

```
ex d=F.add_cycle_rel(lst{is_orthogonal(B), is_orthogonal(Ca), is_orthogonal(Cb)},"d");
```

Uses add_cycle_rel 16f 23c 33a 83b, ex 41b 47e 47e 47e 53a, and is_orthogonal 23c 38c.

Finally check that the cycle d is a line (passes the infinity).

22a ⟨figure-ortho-anlytic-proof.cpp 20e⟩+≡ ◁21i

```
Res = F.check_rel(d, F.get_infinity(), cycle_orthogonal);
for (size_t i=0; i< Res.nops(); ++i)
    cout << "Centres and the contact point are collinear: "
        << bool(ex_to<relational>(Res.op(i)))
        << endl;
}
```

Uses check_rel 21d 24g 34a 110c, cycle_orthogonal 34b 113a, get_infinity 17e 49g, nops 50a, and op 50a.

The output, as expected, is:

```
Centres and the contact point are collinear: true
```

22b ⟨separating chunk 16a⟩+≡ ◁20d 26g▷

A.5. **The nine-points theorem—conformal version.** Here we present further usage of the library by an aesthetically attractive example, see Section 4.

The start of our file is minimalistic, we definitely need to include the header of **figure** library.

22c ⟨nine-points-thm.cpp 22c⟩≡ (27a) 25a▷

```
⟨license 121⟩
#include "figure.h"
⟨using all namespaces 16c⟩
int main(){
    ⟨initial data for drawing 22d⟩
    ⟨build medioscribed cycle 22e⟩
```

Defines:
main, used in chunk 88d.

Uses figure 16d 22e 32a 32c 38b 49a 50d 75a 80a 82b 82c 85a 86c 98c 99b 99d 100a 101a 103b 104a 105c 106c 106d 107a 109a 109c 110a.

We define exact coordinates of points which will be used for our picture.

22d ⟨initial data for drawing 22d⟩≡ (22c)

numeric *x1*(-10,10), *y1*(0,1), *x2*(10,10), *y2*(0,1), *x3*(-1,5), *y3*(-3,2), *x4*(1,2), *y4*(-5,2);
int *sign*=-1;

Defines:
`numeric`, used in chunks 17f, 19b, 20c, 23b, 25, 26b, 28, 30b, 31c, 40, 47e, 55, 56a, 59d, 75, 77a, 78d, 81a, 85, 87b, 90e, 91a, 93, 97d, 101, 102, and 113–15.

We declare the figure F which will be constructed.

22e ⟨build medioscribed cycle 22e⟩≡ (22c 27b) 22f▷

figure *F*(**lst**{-1,*sign*});

Defines:
`figure`, used in chunks 16–18, 20e, 22, 27b, 28a, 30a, 36, 37, 45, 46c, 48, 50a, 52, 66, 75–84, 86, 87, 95d, 97–100, 106–108, 110, 111, and 117c.

We will need several "midpoints" in our constructions, the corresponding figure *midpoint_constructor* is readily available from the library.

22f ⟨build medioscribed cycle 22e⟩+≡ (22c 27b) ◁22e 22g▷

figure *SF*=*ex_to*<**figure**>(*midpoint_constructor*());

Uses `figure` 16d 22e 32a 32c 38b 49a 50d 75a 80a 82b 82c 85a 86c 98c 99b 99d 100a 101a 103b 104a 105c 106c 106d 107a 109a 109c 110a and `midpoint_constructor` 40e 117c.

Next we define vertices of the "triangle" A, B, C and the point N which will be an image if infinity. Every point is added to F by giving its explicit coordinates and a string, which will used to label it. The returned value is a GiNaC expression of **symbol** class which will be used as the key of a respective point. All points are added to the zero generation.

22g ⟨build medioscribed cycle 22e⟩+≡ (22c 27b) ◁22f 23a▷

ex *A*=*F.add_point*(**lst**{*x1*,*y1*},"A");
ex *B*=*F.add_point*(**lst**{*x2*, *y2*},"B");
ex *C*=*F.add_point*(**lst**{*x3*,*y3*},"C");

Defines:
`add_point`, used in chunks 17c and 23b.
Uses `ex` 41b 47e 47e 47e 53a.

There is the special point in the construction, which play the role of infinity. We first put this as cycle at infinity to make picture simple.

23a ⟨build medioscribed cycle 22e⟩+≡ (22c 27b) ◁22g 23b▷

ex *N*=*F.add_cycle*(**cycle_data**(0,**lst**{0,0},1),"N");

Defines:
`add_cycle`, used in chunks 19–21, 28b, 30b, 82a, and 117c.
`cycle_data`, used in chunks 26b, 46b, 52–54, 56–59, 63a, 68–71, 75, 81, 83–85, 87c, 89, 95–97, 113c, 117c, and 119a.
Uses `ex` 41b 47e 47e 47e 53a.

This is an alternative selection of point with N being at the centre of the triangle.

23b ⟨build medioscribed cycle 22e⟩+≡ (22c 27b) ◁23a 23c▷

//Fully symmetric data
// ex A=F.add_point(lst{-numeric(10,10},numeric(0,1)),"A")
// ex B=F.add_point(lst{numeric(10,10},numeric(0,1)),"B")
// ex C=F.add_point(lst{numeric(0,4},-numeric(1732050807,1000000000)),"C")
// ex N=F.add_point(lst{numeric(0,4},-numeric(577350269,1000000000)),"N")

Uses `add_point` 16e 22g 32e 80b 80c, `ex` 41b 47e 47e 47e 53a, and `numeric` 22d.

Now we add "sides" of the triangle, that is cycles passing two vertices and N. A cycle passes a point if it is orthogonal to the cycle defined by this point. Thus, each side is defined through a list of three orthogonalities [30; 36, Defn. 6.1]. We also supply a string to label this side. The returned valued is a **symbol** which is a key for this cycle.

23c ⟨build medioscribed cycle 22e⟩+≡ (22c 27b) ◁23b 23d▷

ex *a*=*F.add_cycle_rel*(**lst**{*is_orthogonal*(*B*),*is_orthogonal*(*C*),*is_orthogonal*(*N*)},"a");
ex *b*=*F.add_cycle_rel*(**lst**{*is_orthogonal*(*A*),*is_orthogonal*(*C*),*is_orthogonal*(*N*)},"b");
ex *c*=*F.add_cycle_rel*(**lst**{*is_orthogonal*(*A*),*is_orthogonal*(*B*),*is_orthogonal*(*N*)},"c");

Defines:
`add_cycle_rel`, used in chunks 17, 19–21, 23–25, 28–31, 83, 84a, 117, and 118.
`is_orthogonal`, used in chunks 16, 17, 21, 23, 24, 28d, 29b, 48a, and 113a.
Uses `ex` 41b 47e 47e 47e 53a.

We define the custom `Asymptote` [23] drawing style for sides of the triangle: the dark blue (*rgb* colour (0,0,0.8)) and line thickness 1pt.

23d ⟨build medioscribed cycle 22e⟩+≡ (22c 27b) ◁23c 23e▷

```
F.set_asy_style(a,"rgb(0,0,.8)+1");
F.set_asy_style(b,"rgb(0,0,.8)+1");
F.set_asy_style(c,"rgb(0,0,.8)+1");
```

Defines:
`rgb`, used in chunks 20, 24, 25, 28, 29b, 36, 37a, and 102b.
`set_asy_style`, used in chunks 20b, 23–25, 28, and 29b.

Now we drop "altitudes" in our triangle, that is again provided through three orthogonality relations. They will be draw as dashed lines.

23e ⟨build medioscribed cycle 22e⟩+≡ (22c 27b) ◁23d 23f▷

```
ex ha=F.add_cycle_rel(lst{is_orthogonal(A),is_orthogonal(N),is_orthogonal(a)},"h_a");
F.set_asy_style(ha,"dashed");
ex hb=F.add_cycle_rel(lst{is_orthogonal(B),is_orthogonal(N),is_orthogonal(b)},"h_b");
F.set_asy_style(hb,"dashed");
ex hc=F.add_cycle_rel(lst{is_orthogonal(C),is_orthogonal(N),is_orthogonal(c)},"h_c");
F.set_asy_style(hc,"dashed");
```

Uses `add_cycle_rel` 16f 23c 33a 83b, `ex` 41b 47e 47e 47e 53a, `is_orthogonal` 23c 38c, and `set_asy_style` 19g 23d 37d.

We need the base of altitude *ha*, which is the intersection points of the side *a* and respective altitude *ha*. A point can be can be characterised as a cycle which is orthogonal to itself [30; 36, Defn. 5.13]. To define a relation of a cycle to itself we first need to define a symbol *A1* and then add a cycle with the key *A1* and the relation *is_orthogonal* to *A1*. Finally, there are two such points: the base of altitude and *N*. To exclude the second one we add the relation *is_adifferent* ("almost different") to *N*.

23f ⟨build medioscribed cycle 22e⟩+≡ (22c 27b) ◁23e 24a▷

```
ex A1=symbol("A_h");
F.add_cycle_rel(lst{is_orthogonal(a),is_orthogonal(ha),is_orthogonal(A1),is_adifferent(N)},A1);
```

Defines:
`is_adifferent`, used in chunk 24.
Uses `add_cycle_rel` 16f 23c 33a 83b, `ex` 41b 47e 47e 47e 53a, and `is_orthogonal` 23c 38c.

Two other bases of altitude are defined in a similar manner.

24a ⟨build medioscribed cycle 22e⟩+≡ (22c 27b) ◁23f 24b▷

```
ex B1=symbol("B_h");
F.add_cycle_rel(lst{is_orthogonal(b),is_orthogonal(hb),is_adifferent(N),is_orthogonal(B1)},B1);
ex C1=symbol("C_h");
F.add_cycle_rel(lst{is_adifferent(N),is_orthogonal(c),is_orthogonal(hc),is_orthogonal(C1)},C1);
```

Uses `add_cycle_rel` 16f 23c 33a 83b, `ex` 41b 47e 47e 47e 53a, `is_adifferent` 23f 38f, and `is_orthogonal` 23c 38c.

We add the cycle passing all three bases of altitudes.

24b ⟨build medioscribed cycle 22e⟩+≡ (22c 27b) ◁24a 24c▷

```
ex p=F.add_cycle_rel(lst{is_orthogonal(A1),is_orthogonal(B1),is_orthogonal(C1)},"p");
F.set_asy_style(p,"rgb(0,.8,0)+1");
```

Uses `add_cycle_rel` 16f 23c 33a 83b, `ex` 41b 47e 47e 47e 53a, `is_orthogonal` 23c 38c, `rgb` 19g 23d, and `set_asy_style` 19g 23d 37d.

We build "midpoint" of the arc of *a* between *B* and *C*. To this end we use subfigure *SF* and supply the list of parameters *B*, *C* and *N* ("infinity") which are required by *SF*.

24c ⟨build medioscribed cycle 22e⟩+≡ (22c 27b) ◁24b 24d▷

```
ex A2=F.add_subfigure(SF,lst{B,C,N},"A_m");
```

Defines:
`add_subfigure`, used in chunks 24 and 84c.
Uses `ex` 41b 47e 47e 47e 53a.

Similarly we build other two "midpoints", they all will belong to the cycle *p*.

24d ⟨build medioscribed cycle 22e⟩+≡ (22c 27b) ◁24c 24e▷

```
ex B2=F.add_subfigure(SF,lst{C,A,N},"B_m");
ex C2=F.add_subfigure(SF,lst{A,B,N},"C_m");
```

Uses `add_subfigure` 24c 33b 84b and `ex` 41b 47e 47e 47e 53a.

O is the intersection point of altitudes ha and hb, again it is defined as a cycle with key O orthogonal to itself.

24e ⟨build medioscribed cycle 22e⟩+≡ (22c 27b) ◁24d 24f▷
```
ex O=symbol("O");
F.add_cycle_rel(lst{is_orthogonal(ha),is_orthogonal(hb),is_orthogonal(O),is_adifferent(N)},O);
```

Uses add_cycle_rel 16f 23c 33a 83b, ex 41b 47e 47e 47e 53a, is_adifferent 23f 38f, and is_orthogonal 23c 38c.

We build three more "midpoints" which belong to p as well.

24f ⟨build medioscribed cycle 22e⟩+≡ (22c 27b) ◁24e
```
ex A3=F.add_subfigure(SF,lst{O,A,N},"A_d");
ex B3=F.add_subfigure(SF,lst{B,O,N},"B_d");
ex C3=F.add_subfigure(SF,lst{C,O,N},"C_d");

⟨check the theorem 24g⟩
```

Uses add_subfigure 24c 33b 84b and ex 41b 47e 47e 47e 53a.

Now we want to check that the six additional points all belong to the build cycle p. The list of pre-defined conditions which may be checked is listed in Section B.4.

24g ⟨check the theorem 24g⟩≡ (24–26)
```
cout ≪ "Midpoint BC belongs to the cycle: " ≪ F.check_rel(p,A2,cycle_orthogonal) ≪ endl;
cout ≪ "Midpoint AC belongs to the cycle: " ≪ F.check_rel(p,B2,cycle_orthogonal) ≪ endl;
cout ≪ "Midpoint AB belongs to the cycle: " ≪ F.check_rel(p,C2,cycle_orthogonal) ≪ endl;
cout ≪ "Midpoint OA belongs to the cycle: " ≪ F.check_rel(p,A3,cycle_orthogonal) ≪ endl;
cout ≪ "Midpoint OB belongs to the cycle: " ≪ F.check_rel(p,B3,cycle_orthogonal) ≪ endl;
cout ≪ "Midpoint OC belongs to the cycle: " ≪ F.check_rel(p,C3,cycle_orthogonal) ≪ endl;
```
Defines:
check_rel, used in chunks 22a and 25c.
Uses cycle_orthogonal 34b 113a.

We inscribe the cycle va into the triangle through the relation *is_tangent_i* (that is "tangent from inside") and *is_tangent_* (that is "tangent from outside")to sides of the triangle. We also provide custom Asymptote drawing style: dar red colour and line thickness 0.5pt.

25a ⟨nine-points-thm.cpp 22c⟩+≡ (27a) ◁22c 25b▷
```
ex va=F.add_cycle_rel(lst{is_tangent_o(a),is_tangent_i(b),is_tangent_i(c)},"v_a");
F.set_asy_style(va,"rgb(0.8,0,0)+.5");
```
Defines:
is_tangent_i, used in chunks 21a, 25b, 29b, and 31b.
is_tangent_o, used in chunks 25b and 31b.
Uses add_cycle_rel 16f 23c 33a 83b, ex 41b 47e 47e 47e 53a, rgb 19g 23d, and set_asy_style 19g 23d 37d.

Similarly we define two other tangent cycles: touching two sides from inside and the third from outside (the relation *is_tangent_o*). We also define custom Asymptote styles for the new cycles.

25b ⟨nine-points-thm.cpp 22c⟩+≡ (27a) ◁25a 25d▷
```
ex vb=F.add_cycle_rel(lst{is_tangent_i(a),is_tangent_o(b),is_tangent_i(c)},"v_b");
F.set_asy_style(vb,"rgb(0.8,0,0)+.5");
ex vc=F.add_cycle_rel(lst{is_tangent_i(a),is_tangent_i(b),is_tangent_o(c)},"v_c");
F.set_asy_style(vc,"rgb(0.8,0,0)+.5");
⟨check that cycles are tangent 25c⟩
```

Uses add_cycle_rel 16f 23c 33a 83b, ex 41b 47e 47e 47e 53a, is_tangent_i 25a 39d, is_tangent_o 25a 39d, rgb 19g 23d, and set_asy_style 19g 23d 37d.

We also want to check the touching property between cycles:

25c ⟨check that cycles are tangent 25c⟩≡ (25 26)
```
cout ≪ "p and va are tangent: " ≪ F.check_rel(p,va,check_tangent).evalf() ≪ endl;
cout ≪ "p and vb are tangent: " ≪ F.check_rel(p,vb,check_tangent).evalf() ≪ endl;
cout ≪ "p and vc are tangent: " ≪ F.check_rel(p,vc,check_tangent).evalf() ≪ endl;
```

Uses check_rel 21d 24g 34a 110c, check_tangent 34d 113d, and evalf 50a.

Now, we draw our figure to the PDF and PNG files, it is shown at Figure 4.

25d ⟨nine-points-thm.cpp 22c⟩+≡ (27a) ◁25b 25e▷
```
F.asy_write(300,-3.1,2.4,-3.6,1.3,"nine-points-thm-plain", "pdf");
F.asy_write(600,-3.1,2.4,-3.6,1.3,"nine-points-thm-plain", "png");
```
Defines:
asy_write, used in chunks 25, 26, and 29c.

We also can modify a cycle at zero level by *move_point*. This time we restore the initial value of N as a debug check: this is a transition from a pre-defined **cycle** given above to a point (which is a calculated object due to the internal representation).

25e ⟨nine-points-thm.cpp 22c⟩+≡ (27a) ◁25d 25f▷

```
F.move_point(N,lst{numeric(1,2),-numeric(5,2)});
cerr << F << endl;
F.asy_draw(cout,cerr,"",-3.1,2.4,-3.6,1.3);

F.asy_write(300,-3.1,2.4,-3.6,1.3,"nine-points-thm", "pdf");
F.asy_write(600,-3.1,2.4,-3.6,1.3,"nine-points-thm", "png");
⟨check the theorem 24g⟩
⟨check that cycles are tangent 25c⟩
```

Defines:
move_point, used in chunks 25, 26, and 86a.
Uses asy_draw 36b 36b 101a, asy_write 16g 20c 25d 36c 36c 103b, and numeric 22d.

And now we use *move_point* to change coordinates of the point (without a change of its type).

25f ⟨nine-points-thm.cpp 22c⟩+≡ (27a) ◁25e 26a▷

```
F.move_point(N,lst{numeric(4,2),-numeric(5,2)});
F.asy_write(300,-3.1,2.4,-3.6,1.3,"nine-points-thm2");
⟨check the theorem 24g⟩
⟨check that cycles are tangent 25c⟩
```

Uses asy_write 16g 20c 25d 36c 36c 103b, move_point 25e 33c 85a, and numeric 22d.

Then, we move the cycle N to represent the point at infinity (0,**lst**{0,0},1), thus the drawing becomes the classical Nine Points Theorem in Euclidean geometry.

26a ⟨nine-points-thm.cpp 22c⟩+≡ (27a) ◁25f 26b▷

```
F.move_cycle(N, cycle_data(0,lst{0,0},1));
F.asy_write(300,-3.1,2.4,-3.6,1.3,"nine-points-thm1");
⟨check the theorem 24g⟩
⟨check that cycles are tangent 25c⟩
```

Defines:
cycle_data, used in chunks 26b, 46b, 52–54, 56–59, 63a, 68–71, 75, 81, 83–85, 87c, 89, 95–97, 113c, 117c, and 119a.
move_cycle, used in chunk 26b.
Uses asy_write 16g 20c 25d 36c 36c 103b.

We can draw the same figures in the hyperbolic metric as well. The checks show that the nine-point theorem is still valid!

26b ⟨nine-points-thm.cpp 22c⟩+≡ (27a) ◁26a 26c▷

```
F.move_cycle(N, cycle_data(0,lst{0,0},1));
F.set_metric(diag_matrix(lst{-1,1}));
F.asy_write(300,-3.1,2.4,-3.6,1.3,"nine-points-thm-plain-hyp");
⟨check the theorem 24g⟩
⟨check that cycles are tangent 25c⟩
F.move_point(N,lst{numeric(1,2),-numeric(5,2)});
F.asy_write(300,-3.1,2.4,-3.6,1.3,"nine-points-thm-hyp", "pdf");
F.asy_write(600,-3.1,2.4,-3.6,1.3,"nine-points-thm-hyp","png");
⟨check the theorem 24g⟩
⟨check that cycles are tangent 25c⟩
//F.set_metric(diag_matrix(lst{-1,0}));
//F.asy_write(300,-3.1,2.4,-3.6,1.3,"nine-points-thm-par", "pdf");
```

Uses asy_write 16g 20c 25d 36c 36c 103b, cycle_data 23a 26a 42c 43a 54c 54d 55a 56b 56c 56d, move_cycle 26a 33d 82c, move_point 25e 33c 85a, numeric 22d, and set_metric 32b 98c.

Finally, we produce an animation, which illustrate the transition from the traditional nine-point theorem to its conformal version. To this end we return to the elliptic metric and freeze the figure. This can be time-consuming and may be not performed by default.

26c ⟨nine-points-thm.cpp 22c⟩+≡ (27a) ◁26b 26d▷

```
 if (true) {
   F.set_metric(diag_matrix(lst{-1,-1}));
   F.freeze();
```

Uses freeze 17b 37b and set_metric 32b 98c.

We define a symbolic parameter t and make the point N depends on it.

26d ⟨nine-points-thm.cpp 22c⟩+≡ (27a) ◁26c 26e▷
```
realsymbol t("t");
F.move_point(N,lst{(1.0+t)÷2.0,-(5.0+t)÷2.0});
```

Uses move_point 25e 33c 85a and realsymbol 27c.

Then, the range of values *val* for the parameter t and then produce an animation based on these values. The resulting animation is presented on the Fig. 5.

26e ⟨nine-points-thm.cpp 22c⟩+≡ (27a) ◁26d 26f▷
```
lst val;
 int num=50;
for (int i=0;i≤num;++i)
    val.append(t≡exp(pow(2.2*(num-i)÷num,2.2))-1.0);
F.asy_animate(val,300,-3.1,2.4,-3.6,1.3,"nine-points-anim", "pdf");
}
```

Uses asy_animate 17g 37a 37a 104a.

We produce an illustration of *SF* in the canonical position. Everything is done now.

26f ⟨nine-points-thm.cpp 22c⟩+≡ (27a) ◁26e
```
  return 0;
}
```

26g ⟨separating chunk 16a⟩+≡ ◁22b 27f▷

27a ⟨* 27a⟩≡
⟨nine-points-thm.cpp 22c⟩

A.6. Proving the theorem: Symbolic computations.

27b ⟨nine-points-thm-symb.cpp 27b⟩≡ 27e▷
```
⟨license 121⟩
#include "figure.h"
⟨using all namespaces 16c⟩
int main(){
   cout ≪ "Prooving the theorem, this shall take a long time..."
        ≪ endl;
   ⟨initial data for proof 27c⟩
   ⟨build medioscribed cycle 22e⟩
```

Defines:
main, used in chunk 88d.
Uses figure 16d 22e 32a 32c 38b 49a 50d 75a 80a 82b 82c 85a 86c 98c 99b 99d 100a 101a 103b 104a 105c 106c 106d 107a 109a 109c 110a.

We define variables from **realsymbol** class to be used in symbolic computations.

27c ⟨initial data for proof 27c⟩≡ (27b) 27d▷
```
realsymbol x1("x1"), y1("y1"), x2("x2"), y2("y2"), x3("x3"), y3("y3"), x4("x4"), y4("y4");
```
Defines:
realsymbol, used in chunks 26d, 28d, 30c, 31b, 75, 78, 79, 91, 92, and 95a.

We also define the sign for the hyperbolic metric. The proof will work in the elliptic (conformal Euclidean) space as well, however we have synthetic poofs in this case. Symbolic computations in the hyperbolic space are mathematically sufficient for demonstration, but Figure 4 from the previous subsection is physiologically more convincing on the individual level. A synthetic proof for hyperbolic space would be interesting to obtain as well.

27d ⟨initial data for proof 27c⟩+≡ (27b) ◁27c
```
int sign=1;
```

We got the output, which make a full demonstration that the theorem holds in the hyperbolic space as well:

```
Midpoint BC belongs to the cycle {0==0}
Midpoint AC belongs to the cycle {0==0}
Midpoint AB belongs to the cycle {0==0}
Midpoint OA belongs to the cycle {0==0}
Midpoint OB belongs to the cycle {0==0}
Midpoint OC belongs to the cycle {0==0}
```

But be prepared, that that will take a long time (about 6 hours of CPU time of my slow PC).

27e ⟨nine-points-thm-symb.cpp 27b⟩+≡ ◁27b

```
    return 0;
  }
```

27f ⟨separating chunk 16a⟩+≡ ◁26g 29e▷

A.7. **Numerical relations.** To illustrate the usage of relations with numerical parameters we are solving the following problem from [18, Problem A]:

> Find the cycles having (all three conditions):
> - tangential distance 7 from the circle
> $$(u-7)^2+(v-1)^2=2^2;$$
> - angle $\arccos\frac{4}{5}$ with the circle
> $$(u-5)^2+(v-3)^2=5^2;$$
> - centres lying on the line
> $$\frac{5}{13}u+\frac{12}{13}v=0.$$

The statement of the problem uses orientation of cycles. Geometrically it is given by the inward or outward direction of the normal. In our library the orientation represented by the direction of the vector in the projective space, it changes to the opposite if the vector is multiplied by -1.
The start of of our code is similar to the previous one.

28a ⟨fillmore-springer-example.cpp 28a⟩≡ 28b▷

```
  ⟨license 121⟩
  #include "figure.h"
  ⟨using all namespaces 16c⟩
  int main(){
    ex sign=-numeric(1);
    varidx nu(symbol("nu", "\\nu"), 2);
    ex e = clifford_unit(nu, diag_matrix(lst{-numeric(1),sign}));
    figure F(e);
```

Defines:
  `main`, used in chunk 88d.
Uses `ex` 41b 47e 47e 47e 53a, `figure` 16d 22e 32a 32c 38b 49a 50d 75a 80a 82b 82c 85a 86c 98c 99b 99d 100a 101a 103b 104a 105c 106c 106d 107a 109a 109c 110a, and `numeric` 22d.

Now we define three circles given in the problem statement above.

28b ⟨fillmore-springer-example.cpp 28a⟩+≡ ◁28a 28c▷

```
    ex A=F.add_cycle(cycle(lst{numeric(7),numeric(1)},e,numeric(4)),"A");
    ex B=F.add_cycle(cycle(lst{numeric(5),numeric(3)},e,numeric(25)),"B");
    ex C=F.add_cycle(cycle(numeric(0),lst{numeric(5,13),numeric(12,13)},0,e),"C");
```

Uses `add_cycle` 23a 32f 81d, `ex` 41b 47e 47e 47e 53a, and `numeric` 22d.

All given data will be drawn in black inc.

28c ⟨fillmore-springer-example.cpp 28a⟩+≡ ◁28b 28d▷

```
    F.set_asy_style(A,"rgb(0,0,0)");
    F.set_asy_style(B,"rgb(0,0,0)");
    F.set_asy_style(C,"rgb(0,0,0)");
```

Uses `rgb` 19g 23d and `set_asy_style` 19g 23d 37d.

The solution D is a circle defined by the three above conditions. The solution will be drawn in red.

28d ⟨fillmore-springer-example.cpp 28a⟩+≡ ◁28c 29a▷

```
    realsymbol D("D"), T("T");
  F.add_cycle_rel(lst{tangential_distance(A,true,numeric(7)), // The tangential distance to A is 7
                      make_angle(B,true,numeric(4,5)), // The angle with B is arccos(4/5)
                      is_orthogonal(C), // If the centre is on C, then C and D are orthogonal
            is_real_cycle(D)}, D); // We require D be a real circle, as there are two imaginary solutions as well
    F.set_asy_style(D,"rgb(0.7,0,0)");
```

Defines:
is_real_cycle, used in chunk 31b.
make_angle, never used.
tangential_distance, never used.
Uses add_cycle_rel 16f 23c 33a 83b, is_orthogonal 23c 38c, numeric 22d, realsymbol 27c, rgb 19g 23d, and set_asy_style 19g 23d 37d.

The output tells parameters of two solutions:

```
  Solutions: {(-1.0, [0,0]~D, 0.99999999999999999734),
(-0.0069444444444444444444, [-0.089285714285714285705,0.037202380952380952383]~D, -1.0)}
```

Here, as in **cycle** library, the set of four numbers $(k, [l, n], m)$ represent the circle equation $k(u^2+v^2)-2lu-2nv+m = 0$.

29a ⟨fillmore-springer-example.cpp 28a⟩+≡ ◁28d 29b▷

```
    cout << "Solutions: "<< F.get_cycles(D).evalf() << endl;
```

Uses evalf 50a.

To visualise the tangential distances we may add the joint tangent lines to the figure. Some solutions are lines with imaginary coefficients, to avoid them we use *only_reals* condition. The tangents will be drawn in blue inc.

29b ⟨fillmore-springer-example.cpp 28a⟩+≡ ◁29a 29c▷

```
    F.add_cycle_rel(lst{is_tangent_i(D),is_tangent_i(A),is_orthogonal(F.get_infinity()),only_reals(T)},T);
    F.set_asy_style(T,"rgb(0,0,0.7)");
```

Defines:
only_reals, used in chunks 21 and 31b.
Uses add_cycle_rel 16f 23c 33a 83b, get_infinity 17e 49g, is_orthogonal 23c 38c, is_tangent_i 25a 39d, rgb 19g 23d, and set_asy_style 19g 23d 37d.

Finally we draw the picture, see Fig. 8, which shall be compared with [18, Fig. 1].

29c ⟨fillmore-springer-example.cpp 28a⟩+≡ ◁29b 29d▷

```
    F.asy_write(400,-4,20,-12.5,9,"fillmore-springer-example");
```

Uses asy_write 16g 20c 25d 36c 36c 103b.

Out of curiosity we may want to know that is square of tangents intervals which are separate circles A, D. The output is:

```
Sq. cross tangent distance: {41.000000000000000003,-7.571428571428571435}
```

Thus one solution does have such tangents with length $\sqrt{41}$, and for the second solution such tangents are imaginary since the square is negative. This happens because one solution D intersects A.

29d ⟨fillmore-springer-example.cpp 28a⟩+≡ ◁29c

```
    cout << "Sq. cross tangent distance: " << F.measure(D,A,sq_cross_t_distance_is).evalf() << endl;
   return 0;
  }
```

Defines:
measure, never used.
sq_cross_t_distance_is, never used.
Uses evalf 50a.

fillmore-springer-example.pdf

FIGURE 8. The illustration to Fillmore–Springer example, which may be compared with [18, Fig. 1].

29e ⟨separating chunk 16a⟩+≡ ◁27f 120▷

A.8. **Three-dimensional examples.** The most of the library functionality (except graphical methods) is literally preserved for quadrics in arbitrary dimensions. We demonstrate this on the following stereometric problem of Apollonius type, cf. [18, § 8]. Let four spheres of equal radii R have centres at four points $(1,1,1)$, $(-1,-1,1)$, $(-1,1,-1)$, $(1,-1,-1)$. These points are vertices of a regular tetrahedron and are every other vertices of a cube with the diagonal $2\sqrt{3}$.

There are two obvious spheres with the centre at the origin $(0,0,0)$ touching all four given spheres, they have radii $R+\sqrt{3}$ and $\sqrt{3}-R$. Are there any others?

We start from the standard initiation and define the metric of three dimensional Euclidean space.

29f ⟨3D-figure-example.cpp 29f⟩≡ 30c ▷
```
⟨3D-figure-example-common 30a⟩
```

The following two chunks are shared with the next example.

30a ⟨3D-figure-example-common 30a⟩≡ (29f 31a) 30b ▷
```
⟨license 121⟩
#include "figure.h"
⟨using all namespaces 16c⟩
int main(){
   ex e3D = clifford_unit(varidx(symbol("lam"), 3), diag_matrix(lst{-1,-1,-1})); // Metric for 3D space
   possymbol R("R");
   figure F(e3D);
```

Defines:
`main`, used in chunk 88d.

Uses `ex` 41b 47e 47e 47e 53a and `figure` 16d 22e 32a 32c 38b 49a 50d 75a 80a 82b 82c 85a 86c 98c 99b 99d 100a 101a 103b 104a 105c 106c 106d 107a 109a 109c 110a.

Then we put four given spheres to the figure. They are defined by their centres and square of radii.

30b ⟨3D-figure-example-common 30a⟩+≡ (29f 31a) ◁ 30a
```
   /* Numerical radii */
   ÷* ex P1=F.add_cycle(cycle(lst{1,1,1}, e3D, numeric(3,4)), "P1");
   ex P2=F.add_cycle(cycle(lst{-1,-1,1}, e3D, numeric(3,4)), "P2");
   ex P3=F.add_cycle(cycle(lst{1,-1,-1}, e3D, numeric(3,4)), "P3");
   ex P4=F.add_cycle(cycle(lst{-1,1,-1}, e3D, numeric(3,4)), "P4");
   *÷
   ex P1=F.add_cycle(cycle(lst{1,1,1}, e3D, pow(R,2)), "P1");
   ex P2=F.add_cycle(cycle(lst{-1,-1,1}, e3D, pow(R,2)), "P2");
   ex P3=F.add_cycle(cycle(lst{1,-1,-1}, e3D, pow(R,2)), "P3");
   ex P4=F.add_cycle(cycle(lst{-1,1,-1}, e3D, pow(R,2)), "P4");
```

Uses `add_cycle` 23a 32f 81d, `ex` 41b 47e 47e 47e 53a, and `numeric` 22d.

Then we introduce the unknown cycle by the four tangency conditions to given spheres. We also put two conditions to rule out non-geometric solutions: *is_real_cycle* checks that the radius is real, *only_reals* requires that all coefficients are real.

30c ⟨3D-figure-example.cpp 29f⟩+≡ ◁ 29f 30d ▷
```
   realsymbol N3("N3");
   F.add_cycle_rel(lst{is_tangent(P1), is_tangent(P2), is_tangent(P3),is_tangent(P4)
                  /* Tests below forbid all spheres with symbolic parameters */
                  //, only_reals(N3), is_real_cycle(N3)
         }, N3);
```

Defines:
`is_real_cycle`, used in chunk 31b.
`is_tangent`, used in chunk 21g.
`only_reals`, used in chunks 21 and 31b.

Uses `add_cycle_rel` 16f 23c 33a 83b and `realsymbol` 27c.

Then we output the solutions and their radii.

30d ⟨3D-figure-example.cpp 29f⟩+≡ ◁30c

```
    lst L=ex_to<lst>(F.get_cycles(N3));
    cout << L.nops() << " spheres found" << endl;
      for (auto x: L)
      cout << "Sphere: " << ex_to<cycle>(x).normal()
       << ", radius sq: " << (ex_to<cycle>(x).det()).normal()
       <<endl;
    return 0;
  }
```

Uses nops 50a.

For the numerical value $R = \frac{\sqrt{3}}{2}$, the program found 16 different solutions which satisfy to *is_real_cycle* and *only_reals* conditions. If we omit these conditions then additional 16 imaginary spheres will be producing (32 in total).

For the symbolic radii R again 32 different spheres are found. The condition *only_reals* leaves only two obvious spheres, discussed at the beginning of the subsection. This happens because for some value of R coefficient of other spheres may turn to be complex. Finally, if we use the condition *is_real_cycle*, then no sphere passes it—the square of its radius may become negative for some R.

For visualisation we can partially re-use the previous code.

31a ⟨3D-figure-visualise.cpp 31a⟩≡ 31b▷

```
  ⟨3D-figure-example-common 30a⟩
```

To simplify the structure we eliminate spheres which are different only up to the rotational symmetry of the initial set-up. To this end we explicitly specify inner or outer tangency for different spheres.

31b ⟨3D-figure-visualise.cpp 31a⟩+≡ ◁31a 31c▷

```
    realsymbol N0("N0"), N1("N1"), N2("N2"), N3("N3"),  N4("N4");
    F.add_cycle_rel(lst{is_tangent_o(P1), is_tangent_o(P2), is_tangent_o(P3),is_tangent_o(P4),
                is_real_cycle(N0), only_reals(N0)}, N0);
    F.add_cycle_rel(lst{is_tangent_o(P1), is_tangent_o(P2), is_tangent_o(P3),is_tangent_i(P4),
                is_real_cycle(N1), only_reals(N1)}, N1);
    F.add_cycle_rel(lst{is_tangent_o(P1), is_tangent_o(P2), is_tangent_i(P3),is_tangent_i(P4),
                is_real_cycle(N2), only_reals(N2)}, N2);
    F.add_cycle_rel(lst{is_tangent_o(P1), is_tangent_i(P2), is_tangent_i(P3),is_tangent_i(P4),
                is_real_cycle(N3), only_reals(N3)}, N3);
```

Uses add_cycle_rel 16f 23c 33a 83b, is_real_cycle 28d 30c 38g, is_tangent_i 25a 39d, is_tangent_o 25a 39d, only_reals 29b 30c 39b, and realsymbol 27c.

Now we save the arrangement for the numerical value $R^2 = \frac{3}{4}$.

31c ⟨3D-figure-visualise.cpp 31a⟩+≡ ◁31b

```
    F.subs(R≡sqrt(ex_to<numeric>(3))÷2).arrangement_write("appolonius");
    return 0;
  }
```

Defines:
  arrangement_write, never used.
Uses numeric 22d and subs 50a.

Now this arrangement can be visualised by loading the file `appolonius.txt` into the helper programme `cycle3D-visualiser`. A screenshot of such visualisation is shown on Fig. 3.

Appendix B. Public Methods in the **figure** class

This section lists all methods, which may be of interest to an end-user. An advanced user may find further advise in Appendix E, which outlines the library header file. Methods presented here are grouped by their purpose.

The source (interleaved with documentation in a `noweb` file) can be found at SourceForge project page [39] as Git tree. The code is written using `noweb` literate programming tool [52]. The code uses some C++11 features, e.g. regeps and *std::function*. Drawing procedures delegate to `Asymptote` [23].

The stable realises and full documentation are in Files section of the project. A release archive contain all C++ files extracted from the `noweb` source, thus only the standard C++ compiler is necessary to use them.

Furthermore, a live CD with the compiled library, examples and all required tools is distributed as an ISO image. You may find a link to the ISO image at the start of this Web page:

http://www.maths.leeds.ac.uk/~kisilv/courses/using_sw.html

It also explains how to use the live CD image either to boot your computer or inside a Virtual Machine.

B.1. **Creation and re-setting of figure, changing** *metric***.** Here are methods to initialise **figure** and manipulate its basic property.
This is the simplest constructor of an initial figure with the (point) metric *Mp*. By default, any figure contains the *real_line* and *infinity*. Parameter *M*, may be same as for definition of *clifford_unit* in GiNaC, that is, be represented by a square **matrix**, **clifford** or **indexed** class object. If the metric *Mp* is not provided, then the default elliptic metric in two dimensions is used, it is given by the matrix $\begin{pmatrix}-1 & 0\\ 0 & -1\end{pmatrix}$.

An advanced user may wish to specify a different metric for point and cycle spaces, see [36, § 4.2] for the discussion. By default, if the metric in the point space is $\begin{pmatrix}-1 & 0\\ 0 & \sigma\end{pmatrix}$ then the metric of cycle space is:

$$\begin{pmatrix}-1 & 0\\ 0 & -\chi(-\sigma)\end{pmatrix}, \qquad \text{where} \quad \chi(t)=\begin{cases}1, & t\geq 0;\\ -1, & t<0.\end{cases} \tag{17}$$

is the *Heaviside function* $\chi(\sigma)$ In other word, by default for elliptic and parabolic point space the cycle space has the same metric and for the parabolic point space the cycle space is elliptic. If a user want a different combination then the following constructor need to be used, see also *set_metric*() below

32a ⟨public methods in figure class 32a⟩≡ (49d) 32b ▷
figure(**const ex** & *Mp*, **const ex** & *Mc*=0);

Defines:
`figure`, used in chunks 16–18, 20e, 22, 27b, 28a, 30a, 36, 37, 45, 46c, 48, 50a, 52, 66, 75–84, 86, 87, 95d, 97–100, 106–108, 110, 111, and 117c.
Uses `ex` 41b 47e 47e 47e 53a.

The metrics set in the above constructor can be changed at any stage, and all cycles will be re-calculated in the figure accordingly. The parameter *Mp* can be the same type of object as in the first constructor **figure**(**const ex** &). The first form change the point space metric and derive respective cycle space metric as described above. In the second form both metrics are provided explicitly.

32b ⟨public methods in figure class 32a⟩+≡ (49d) ◁ 32a 32c ▷
void *set_metric*(**const ex** & *Mp*, **const ex** & *Mc*=0);

Defines:
`set_metric`, used in chunks 26 and 97c.
Uses `ex` 41b 47e 47e 47e 53a.

This constructor can be used to create a figure with the pre-defined collection *N* of cycles.

32c ⟨public methods in figure class 32a⟩+≡ (49d) ◁ 32b 32d ▷
figure(**const ex** & *Mp*, **const ex** & *Mc*, **const** *exhashmap*<**cycle_node**> & *N*);

Defines:
`figure`, used in chunks 16–18, 20e, 22, 27b, 28a, 30a, 36, 37, 45, 46c, 48, 50a, 52, 66, 75–84, 86, 87, 95d, 97–100, 106–108, 110, 111, and 117c.
Uses `cycle_node` 43c 45c 69c 70a 70b 70c 71a 71b 72b 72c 73a 74a 74d and `ex` 41b 47e 47e 47e 53a.

Remove all **cycle_node** from the figure. Only the *point_metric*, *cycle_metric*, *real_line* and *infinity* are preserved.

32d ⟨public methods in figure class 32a⟩+≡ (49d) ◁ 32c 32e ▷
void *reset_figure*();

Defines:
`reset_figure`, never used.

B.2. **Adding elements to figure.** This method add points to the figure. A point is represented as cycles with radius 0 (with respect to the cycle metric) and coordinates $x=(x_1,\ldots,x_n)$ of their centre (represented by a **lst** of the suitable length). The procedure returns a symbol, which can be used later to refer this point. In the first form parameters *name* and (optional) *TeXname* provide respective string to name this new symbol. In the second form the whole symbol *key* is provided (and it will be returned by the procedure).

32e ⟨public methods in figure class 32a⟩+≡ (49d) ◁ 32d 32f ▷
ex *add_point*(**const ex** & *x*, *string name*, *string TeXname*="");
ex *add_point*(**const ex** & *x*, **const ex** & *key*);

Defines:
`add_point`, used in chunks 17c and 23b.
`key`, used in chunks 32, 33, 37–40, 45e, 48–50, 80–86, 95–97, 99c, and 116c.
`name`, used in chunks 32, 33, 36, 37, 78d, 80–84, 97c, 100d, 103–105, and 112b.
`TeXname`, used in chunks 32, 33, 80b, 82–84, and 112b.
Uses `ex` 41b 47e 47e 47e 53a.

This method add a cycle at zero generation without parents. The returned value and parameters *name*, *TeXname* and *key* are as in the previous methods *add_point*().

32f ⟨public methods in figure class 32a⟩+≡ (49d) ◁32e 33a▷

```
ex add_cycle(const ex & C, string name, string TeXname="");
ex add_cycle(const ex & C, const ex & key);
```

Defines:
`add_cycle`, used in chunks 19–21, 28b, 30b, 82a, and 117c.
Uses `ex` 41b 47e 47e 47e 53a, `key` 32e, `name` 32e, and `TeXname` 32e.

Add a new node by a set *rel* of relations. The returned value and parameters *name*, *TeXname* and *key* are as in methods *add_point*().

33a ⟨public methods in figure class 32a⟩+≡ (49d) ◁32f 33b▷

```
ex add_cycle_rel(const lst & rel, string name, string TeXname="");
ex add_cycle_rel(const lst & rel, const ex & key);
ex add_cycle_rel(const ex & rel, string name, string TeXname="");
ex add_cycle_rel(const ex & rel, const ex & key);
```

Defines:
`add_cycle_rel`, used in chunks 17, 19–21, 23–25, 28–31, 83, 84a, 117, and 118.
Uses `ex` 41b 47e 47e 47e 53a, `key` 32e, `name` 32e, and `TeXname` 32e.

Add a new cycle as the result of certain subfigure F. The list L provides nodes from the present figure, which shall be substituted to the zero generation of F. See *midpoint_constructor*() for an example, how subfigure shall be defined, The returned value and parameters *name*, *TeXname* and *key* are as in methods *add_point*().

33b ⟨public methods in figure class 32a⟩+≡ (49d) ◁33a 33c▷

```
ex add_subfigure(const ex & F, const lst & L, string name, string TeXname="");
ex add_subfigure(const ex & F, const lst & L, const ex & key);
```

Defines:
`add_subfigure`, used in chunks 24 and 84c.
Uses `ex` 41b 47e 47e 47e 53a, `key` 32e, `name` 32e, and `TeXname` 32e.

B.3. **Modification, deletion and searches of nodes.** This method modifies a node created by *add_point*() by moving the centre to new coordinates $x = (x_1, \ldots, x_n)$ (represented by a **lst** of the suitable length).

33c ⟨public methods in figure class 32a⟩+≡ (49d) ◁33b 33d▷

```
void move_point(const ex & key, const ex & x);
```

Defines:
`move_point`, used in chunks 25, 26, and 86a.
Uses `ex` 41b 47e 47e 47e 53a and `key` 32e.

This method replaced a node referred by *key* with the value of a cycle C. This can be applied to a node without parents only.

33d ⟨public methods in figure class 32a⟩+≡ (49d) ◁33c 33e▷

```
void move_cycle(const ex & key, const ex & C);
```

Defines:
`move_cycle`, used in chunk 26b.
Uses `ex` 41b 47e 47e 47e 53a and `key` 32e.

Remove a node given *key* and all its children and grand children in all generations

33e ⟨public methods in figure class 32a⟩+≡ (49d) ◁33d 33f▷

```
void remove_cycle_node(const ex & key);
```

Defines:
`remove_cycle_node`, never used.
Uses `ex` 41b 47e 47e 47e 53a and `key` 32e.

Return the label for **cycle_node** with the first matching name. If the name is not found, the zero expression is returned.

33f ⟨public methods in figure class 32a⟩+≡ (49d) ◁33e 33g▷

```
ex get_cycle_key(string name) const;
```

Defines:
`get_cycle_key`, never used.
Uses `ex` 41b 47e 47e 47e 53a and `name` 32e.

Finally, we provide the methods to obtain the **lst** of keys for all nodes in generations between *mingen* and *maxgen* inclusively. The default value *GHOST_GEN* of *maxgen* removes the check of the upper bound. Thus, the call of the method with the default values produce the list of all key except the ghost generation. The second method orders keys from smaller to larger generations. The first method is faster on figures with many generation.

33g ⟨public methods in figure class 32a⟩+≡ (49d) ◁33f 34a▷

ex *get_all_keys*(**const int** *mingen* = *GHOST_GEN*+1, **const int** *maxgen* = *GHOST_GEN*) **const**;
ex *get_all_keys_sorted*(**const int** *mingen* = *GHOST_GEN*+1, **const int** *maxgen* = *GHOST_GEN*) **const**;

Defines:
`get_all_keys`, used in chunks 20a, 105d, and 106d.
`get_all_keys_sorted`, never used.
Uses `ex` 41b 47e 47e 47e 53a and `GHOST_GEN` 42b 42b.

B.4. **Check relations and measure parameters.** To prove theorems we need to measure (*measure*) some quantities or to check (*check_rel*) if two cycles from the figure are in a certain relation to each other, which were not explicitly defined by the construction.

B.4.1. *Checking relations.* A relation which may holds or not may be checked by the following method. It returns a **lst** of *GiNaC*::**relational**s, which present the relation between all pairs of cycles in the nodes with *key1* and *key2*. Typically two cycles are branching in the synchronous way. Thus it makes sense to compare only respective pairs, this is achieved with the default value *corresponds*=**true**.

34a ⟨public methods in figure class 32a⟩+≡ (49d) ◁33g 35a▷

ex *check_rel*(**const ex** & *key1*, **const ex** & *key2*, *PCR rel*, **bool** *use_cycle_metric*=**true**,
const ex & *parameter*=0, **bool** *corresponds*=**true**) **const**;

Defines:
`check_rel`, used in chunks 22a and 25c.
Uses `ex` 41b 47e 47e 47e 53a and `PCR` 45d.

The available cycles properties to check are as follows. Most of these properties are also behind the cycle relations described in C.

Orthogonality of cycles given by [36, § 6.1]:

$$\left\langle C, \tilde{C} \right\rangle = 0. \tag{18}$$

For circles it coincides with usual orthogonality, for other situations see [36, Ch. 6] for detailed analysis.

34b ⟨relations to check 34b⟩≡ (46e) 34c▷

ex *cycle_orthogonal*(**const ex** & *C1*, **const ex** & *C2*, **const ex** & *pr*=0);

Defines:
`cycle_orthogonal`, used in chunks 21d, 22a, 24g, 38c, 60–62, 64a, 81a, 117, and 118.
Uses `ex` 41b 47e 47e 47e 53a.

Focal orthogonality of cycles [36, § 6.6]:

$$\left\langle \tilde{C} C \tilde{C}, \mathbb{R} \right\rangle = 0. \tag{19}$$

34c ⟨relations to check 34b⟩+≡ (46e) ◁34b 34d▷

ex *cycle_f_orthogonal*(**const ex** & *C1*, **const ex** & *C2*, **const ex** & *pr*=0);

Defines:
`cycle_f_orthogonal`, used in chunks 38d, 61, 62a, and 64a.
Uses `ex` 41b 47e 47e 47e 53a.

Tangent condition between two cycles which shall be used for checks. This relation is not suitable for construction, use *is_tangent* and the likes from Section C for this.

34d ⟨relations to check 34b⟩+≡ (46e) ◁34c 34e▷

ex *check_tangent*(**const ex** & *C1*, **const ex** & *C2*, **const ex** & *pr*=0);

Defines:
`check_tangent`, used in chunk 25c.
Uses `ex` 41b 47e 47e 47e 53a.

Check two cycles are different.

34e ⟨relations to check 34b⟩+≡ (46e) ◁34d 34f▷

ex *cycle_different*(**const ex** & *C1*, **const ex** & *C2*, **const ex** & *pr*=0);

Defines:
`cycle_different`, used in chunks 38e, 61, 62a, 64a, and 81a.
Uses `ex` 41b 47e 47e 47e 53a.

Check two cycles are almost different, counting possible rounding errors.

34f ⟨relations to check 34b⟩+≡ (46e) ◁34e 34g▷

ex *cycle_adifferent*(**const ex** & *C1*, **const ex** & *C2*, **const ex** & *pr*=0);

Defines:
`cycle_adifferent`, used in chunks 38f, 61, 62a, 64a, and 118c.
Uses `ex` 41b 47e 47e 47e 53a.

Check that the cycle product with other cycle (or itself) is non-positive.

34g ⟨relations to check 34b⟩+≡ (46e) ◁34f 34h▷
```
ex product_sign(const ex & C1, const ex & C2, const ex & pr=1);
```
Defines:
`product_sign`, used in chunks 38g, 39a, 61, 62a, and 64a.
Uses `ex` 41b 47e 47e 47e 53a.

We may want to exclude cycles with imaginary coefficients, this condition check it.

34h ⟨relations to check 34b⟩+≡ (46e) ◁34g
```
ex coefficients_are_real(const ex & C1, const ex & C2, const ex & pr=1);
```
Defines:
`coefficients_are_real`, used in chunks 39b, 61, 62a, and 64a.
Uses `ex` 41b 47e 47e 47e 53a.

B.4.2. *Measuring quantites.* A quantity between two cycles may be measured by this method. Typically two cycles are branching in the synchronous way. Thus it makes sense to compare only respective pairs, this is achieved with the default value *corresponds*=**true**.

35a ⟨public methods in figure class 32a⟩+≡ (49d) ◁34a 35b▷
```
ex measure(const ex & key1, const ex & key2, PCR rel, bool use_cycle_metric=true,
        const ex & parameter=0, bool corresponds=true) const;
```
Defines:
`measure`, never used.
Uses `ex` 41b 47e 47e 47e 53a and `PCR` 45d.

B.5. **Accessing elements of the figure.** We can obtain *point_metric* and *cycle_metric* form a figure by the following methods.

35b ⟨public methods in figure class 32a⟩+≡ (49d) ◁35a 35c▷
```
inline ex get_point_metric() const { return point_metric; }
inline ex get_cycle_metric() const { return cycle_metric; }
```
Defines:
`get_cycle_metric`, used in chunk 76c.
`get_point_metric`, used in chunk 76c.
Uses `cycle_metric` 50f, `ex` 41b 47e 47e 47e 53a, and `point_metric` 50f.

Sometimes, we need to check the dimensionality of the figure, which is essentially the dimensionality of the metric.

35c ⟨public methods in figure class 32a⟩+≡ (49d) ◁35b 35d▷
```
inline ex get_dim() const { return ex_to<varidx>(point_metric.op(1)).get_dim(); }
```
Defines:
`get_dim()`, used in chunks 43a, 54, 55, 58, 59, 75–78, 80c, 81a, 85a, 87b, 97, 98, 101b, 105c, 115c, and 116d.
Uses `ex` 41b 47e 47e 47e 53a, `op` 50a, and `point_metric` 50f.

All **cycle** associated with a key *ck* can be obtained through the following method. The optional parameter tell which metric to use: either *point_metric* or *cycle_metric*. The method returns a list of cycles associated to the key *ck*.

35d ⟨public methods in figure class 32a⟩+≡ (49d) ◁35c 35e▷
```
inline ex get_cycles(const ex & ck, bool use_point_metric=true) const {
   return get_cycles(ck,use_point_metric?point_metric:cycle_metric);}
```
Defines:
`get_cycle`, used in chunks 19d, 21h, 29a, 30d, 43a, 44d, 57b, 63c, 69d, 97c, 101d, 105d, 106d, 110d, and 111d.
Uses `cycle_metric` 50f, `ex` 41b 47e 47e 47e 53a, and `point_metric` 50f.

In fact, we can use a similar method to get **cycle** with any permitted expression as a metric.

35e ⟨public methods in figure class 32a⟩+≡ (49d) ◁35d 35f▷
```
ex get_cycles(const ex & ck, const ex & metric) const;
```
Defines:
`get_cycle`, used in chunks 19d, 21h, 29a, 30d, 43a, 44d, 57b, 63c, 69d, 97c, 101d, 105d, 106d, 110d, and 111d.
Uses `ex` 41b 47e 47e 47e 53a.

The generation of the cycle associated to the key *ck* is provided by the method:

35f ⟨public methods in figure class 32a⟩+≡ (49d) ◁35e 35h▷
```
inline ex get_generation(const ex & ck) const {
   return ex_to<cycle_node>(get_cycle_node(ck)).get_generation();}
```
Defines:
`get_generation`, used in chunks 44e, 67d, 76d, 82–86, 98–101, 105d, and 107a.
Uses `cycle_node` 43c 45c 69c 70a 70b 70c 71a 71b 72b 72c 73a 74a 74d, `ex` 41b 47e 47e 47e 53a, and `get_cycle_node` 49e.

Sometimes we need to apply a function to all **cycle**s which compose the **figure**. Here we define the type for such a function.

35g ⟨defining types 35g⟩≡ (42a) 45d ▷
using *PEVAL* = *std*::*function*<**ex**(**const ex** &, **const ex** &)>;

Uses ex 41b 47e 47e 47e 53a.

This is the method to apply a function *func* to all particular **cycle**s which compose the **figure**. It returns a **lst** of **lst**s. Each sub-list has three elements: the returned value of *func*, the key of the respective **cycle_node** and the number of **cycle** in the respective node. The parameter *use_cycle_metric* tells which metric shall be used: either cycle space or point space, see [36, § 4.2].

35h ⟨public methods in figure class 32a⟩+≡ (49d) ◁ 35f 36b ▷
ex *apply*(*PEVAL* *func*, **bool** *use_cycle_metric*=**true**, **const ex** & *param* = 0) **const**;

Defines:
apply, never used.
Uses ex 41b 47e 47e 47e 53a.

B.6. **Drawing and printing.** There is a collections of methods which help to visualise a figure. We use Asymptote to produce PostScript, PDF, PNG or other files in two-dimensions and an interactive visualisation tool is available for three-dimensional figures.
The default behaviour of *asy_write*() is an attempt to display files produced by Asymptote. User can disable this visualisation.

36a ⟨additional functions header 36a⟩≡ (42a) 40e ▷
void *show_asy_on*();
void *show_asy_off*();

Defines:
show_asy_off, never used.
show_asy_on, never used.

B.6.1. *Two-dimensional graphics and animation.* The next method returns Asymptote [23] string which draws the entire figure. The drawing is controlled by two *style* and *lstring*. Initial parameters have the same meaning as in **cycle2D**::*asy_draw*(). Explicitly, the drawing is done within the rectangle with the lower left vertex (*xmin*, *ymin*) and upper right (*xmax*, *ymax*). The style of drawing is controlled by *default_asy* and *default_label*, see *asy_cycle_color*() and *label_pos*() for ideas. On complicated figures, see Fig. 2, we may not want cycles label to be printed at all, this can be controlled through *with_labels* parameter. By default the *real_line* is drawn and the comments in the file are presented, this can be amended through *with_realline* and *with_header* parameters respectively. 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 result is written to the stream *ost*.

36b ⟨public methods in figure class 32a⟩+≡ (49d) ◁ 35h 36c ▷
void *asy_draw*(*ostream* & *ost* =*std*::*cout*, *ostream* & *err*=*std*::*cerr*, **const** *string* *picture*="",
const ex & *xmin* = -5, **const ex** & *xmax* = 5,
const ex & *ymin* = -5, **const ex** & *ymax* = 5,
asy_style *style*=*default_asy*, *label_string* *lstring*=*default_label*,
bool *with_realline*=**true**, **bool** *with_header*=**true**,
int *points_per_arc* = 0, **const** *string* *imaginary_options*="rgb(0,.9,0)+4pt",
bool *with_labels*=**true**) **const**;

Defines:
asy_draw, used in chunks 25e and 102–104.
Uses asy_style 51d, ex 41b 47e 47e 47e 53a, label_string 51e, and rgb 19g 23d.

This method creates a temporary file with `Asymptote` commands to draw the figure, then calls the `Asymptote` to produce the graphic file, the temporary file is deleted afterwards. The parameters are the same as above in *asy_draw*(). The last parameter *rm_asy_file* tells if the `Asymptote` file shall be removed. User may keep it and fine-tune the result manually.

36c ⟨public methods in figure class 32a⟩+≡ (49d) ◁36b 37a▷

```
void asy_write(int size=300, const ex & xmin = -5, const ex & xmax = 5,
               const ex & ymin = -5, const ex & ymax = 5,
               string name="figure-view-tmp", string format="",
               asy_style style=default_asy, label_string lstring=default_label,
               bool with_realline=true, bool with_header=true,
               int points_per_arc=0, const string imaginary_options="rgb(0,.9,0)+4pt",
               bool rm_asy_file=true, bool with_labels=true) const;
```

Defines:
`asy_write`, used in chunks 25, 26, and 29c.

Uses `asy_style` 51d, `ex` 41b 47e 47e 47e 53a, `figure` 16d 22e 32a 32c 38b 49a 50d 75a 80a 82b 82c 85a 86c 98c 99b 99d 100a 101a 103b 104a 105c 106c 106d 107a 109a 109c 110a, `label_string` 51e, `name` 32e, and `rgb` 19g 23d.

This a method to produce an animation. The figure may depend from some parameters, for example of **symbol** class. The first argument *val* is a **lst**, which contains expressions for substitutions into the figure. That is, elements of *val* can be any expression suitable to use as the first parameter of *susb* method in GiNaC. For example, they may be **relational**s (e.g. $t{\equiv}1.5$) or **lst** of **relational**s (e.g. **lst**$\{t{\equiv}1.5,s{\equiv}2.1\}$. The method make the substitution the each element of **lst** into the figure and uses the resulting `Asymptote` drawings as a sequence of shots for the animations. The output *format* may be either predefined `"pdf"`, `"gif"`, `"mng"` or `"mp4"`, or user-specified `Asymptote` string.

The values of parameters can be put to the animation. The default bottom-left position is encoded as `"bl"` for *values_position*, other possible positions are `"br"` (bottom-right), `"tl"` (top-left) and `"tr"` (top-right). Any other string (e.g. the empty one) will preven the parameter values from printing.

The rest of parameters have the same meaning as in *asy_write*(). See the end of Sect. A.2 for further advise on animation embedded into PDF files.

37a ⟨public methods in figure class 32a⟩+≡ (49d) ◁36c 37b▷

```
void asy_animate(const ex & val,
                 int size=300, const ex & xmin = -5, const ex & xmax = 5,
                 const ex & ymin = -5, const ex & ymax = 5,
                 string name="figure-animatecf-tmp", string format="pdf",
                 asy_style style=default_asy, label_string lstring=default_label,
                 bool with_realline=true, bool with_header=true,
                 int points_per_arc = 0, const string imaginary_options="rgb(0,.9,0)+4pt",
                 const string values_position="bl", bool rm_asy_file=true,
                 bool with_labels=true) const;
```

Defines:
`asy_animate`, used in chunk 26e.

Uses `asy_style` 51d, `ex` 41b 47e 47e 47e 53a, `figure` 16d 22e 32a 32c 38b 49a 50d 75a 80a 82b 82c 85a 86c 98c 99b 99d 100a 101a 103b 104a 105c 106c 106d 107a 109a 109c 110a, `label_string` 51e, `name` 32e, and `rgb` 19g 23d.

Evaluation of **cycle** within a figure with symbolic entries may took a long time. To prevent this we may use *freeze* method, and then *unfreeze* after numeric substitution is done.

37b ⟨public methods in figure class 32a⟩+≡ (49d) ◁37a 37c▷

```
inline figure freeze() const {setflag(status_flags::expanded); return *this;}
inline figure unfreeze() const {clearflag(status_flags::expanded); return *this;}
```

Defines:
`freeze`, used in chunk 26c.
`unfreeze`, used in chunk 104c.

Uses `figure` 16d 22e 32a 32c 38b 49a 50d 75a 80a 82b 82c 85a 86c 98c 99b 99d 100a 101a 103b 104a 105c 106c 106d 107a 109a 109c 110a.

To speed-up evaluation of figures we may force float evaluation instead of exact arithmetic.

37c ⟨public methods in figure class 32a⟩+≡ (49d) ◁37b 37d▷

```
inline figure set_float_eval() {float_evaluation=true; return *this;}
inline figure set_exact_eval() {float_evaluation=false; return *this;}
```

Defines:
`set_exact_eval`, used in chunk 97b.
`set_float_eval`, used in chunk 97b.

Uses `figure` 16d 22e 32a 32c 38b 49a 50d 75a 80a 82b 82c 85a 86c 98c 99b 99d 100a 101a 103b 104a 105c 106c 106d 107a 109a 109c 110a and `float_evaluation` 51b.

These methods allow to specify or read an Asymptote drawing style for a particular node.

37d ⟨public methods in figure class 32a⟩+≡ (49d) ◁37c 37e▷

```
inline void set_asy_style(const ex & key, string opt) {nodes[key].set_asy_opt(opt);}
inline string get_asy_style(const ex & key) const {return ex_to<cycle_node>(get_cycle_node(key)).get_asy_opt();}
```

Defines:
get_asy_style, never used.
set_asy_style, used in chunks 20b, 23–25, 28, and 29b.
Uses cycle_node 43c 45c 69c 70a 70b 70c 71a 71b 72b 72c 73a 74a 74d, ex 41b 47e 47e 47e 53a, get_cycle_node 49e, key 32e, and nodes 51a.

B.6.2. *Three-dimensional visualisation.* In three dimensions a visualisation is possible with the help of an additional interactive programme `cycle3D-visualiser`. The following method produces a text file *name.txt* (the default suffix `".txt"` is added to *name* automatically). The file can be visualised by a helper programme. All cycles in generations starting from *first_gen* are represented by their centres, radii, generations and labels.

37e ⟨public methods in figure class 32a⟩+≡ (49d) ◁37d 38a▷

```
void arrangement_write(string name, int first_gen=0) const;
```

Defines:
arrangement_write, never used.
Uses name 32e.

The written file *filename* then can be loaded by texttt cycle3D-visualiser either through command line option or file choosing dialog. See documentations of the helper programme for available tools. In particular, it is possible to make screenshots similar to Fig. 3.

To print a figure F (of any dimensionality) as a list of nodes and relations between them it is enough to direct the figure to the stream:

$$cout \ll F \ll endl;$$

B.7. **Saving and openning.** We can write a figure to a file as a GiNaC archive (`*.gar` file) named *file_name* at a node *fig_name*.

38a ⟨public methods in figure class 32a⟩+≡ (49d) ◁37e 38b▷

```
void save(const char* file_name, const char* fig_name="myfig") const;
```

Defines:
save, used in chunks 80a and 104c.

This constructor reads a figure stored in a GiNaC archive (`*.gar` file) named *file_name* at a node *fig_name*.

38b ⟨public methods in figure class 32a⟩+≡ (49d) ◁38a 49f▷

```
figure(const char* file_name, string fig_name="myfig");
```

Defines:
figure, used in chunks 16–18, 20e, 22, 27b, 28a, 30a, 36, 37, 45, 46c, 48, 50a, 52, 66, 75–84, 86, 87, 95d, 97–100, 106–108, 110, 111, and 117c.

If a figure is created from a code, especially with sufficiently comments, then the code completely describes the figure. Moreover, such a code is probably the preferable archiving form of the figure. However, some figure can be also created from a Graphical User Interface by mouse clicks and stored as GiNaC gar-archives. In such cases it can be useful to write and store some human readable description of the figure, its author and license. Such information can be recorded, amended or read to/from the figure by the following methods:

49f ⟨public methods in figure class 32a⟩+≡ (49d) ◁38b 49g▷

```
inline void info_write(string whole_text) {info_text = whole_text;}
inline void info_append(string more_text) {info_text += more_text;}
inline string info_read() const {return info_text;}
```

Defines:
info_append, never used.
info_read, never used.
info_write, never used.
Uses info_text.

Appendix C. Public methods in **cycle_relation**

Nodes within figure are connected by sets of relations. There is some essential relations pre-defined in the library. Users can define their own relations as well.

The following relations between cycles are predefined. Orthogonality of cycles given by [36, § 6.1]:

$$\left\langle C, \tilde{C} \right\rangle = 0. \tag{20}$$

38c ⟨predefined cycle relations 38c⟩≡ (47d) 38d ▷

inline cycle_relation *is_orthogonal*(**const ex** & *key*, **bool** *cm*=**true**)
    {**return cycle_relation**(*key*, *cycle_orthogonal*, *cm*);}

Defines:
  `is_orthogonal`, used in chunks 16, 17, 21, 23, 24, 28d, 29b, 48a, and 113a.
Uses `cycle_orthogonal` 34b 113a, `cycle_relation` 40c 45e 46c 60d 61 62a 62b 64a 64b, `ex` 41b 47e 47e 47e 53a, and `key` 32e.

Focal orthogonality of cycles (19), see [36, § 6.6].

38d ⟨predefined cycle relations 38c⟩+≡ (47d) ◁ 38c 38e ▷

inline cycle_relation *is_f_orthogonal*(**const ex** & *key*, **bool** *cm*=**true**)
    {**return cycle_relation**(*key*, *cycle_f_orthogonal*, *cm*);}

Defines:
  `is_f_orthogonal`, used in chunk 113b.
Uses `cycle_f_orthogonal` 34c 113b, `cycle_relation` 40c 45e 46c 60d 61 62a 62b 64a 64b, `ex` 41b 47e 47e 47e 53a, and `key` 32e.

We may want a cycle to be different from another. For example, if we look for intersection of two lines we want to exclude the infinity, where they are intersected anyway. Then, we may add the condition *is_different*(*F.get_infinity*()).

38e ⟨predefined cycle relations 38c⟩+≡ (47d) ◁ 38d 38f ▷

inline cycle_relation *is_different*(**const ex** & *key*, **bool** *cm*=**true**)
    {**return cycle_relation**(*key*, *cycle_different*, *cm*);}

Defines:
  `is_different`, never used.
Uses `cycle_different` 34e 114c, `cycle_relation` 40c 45e 46c 60d 61 62a 62b 64a 64b, `ex` 41b 47e 47e 47e 53a, and `key` 32e.

Due to a possible rounding errors we include an approximate version of *is_different*.

38f ⟨predefined cycle relations 38c⟩+≡ (47d) ◁ 38e 38g ▷

inline cycle_relation *is_adifferent*(**const ex** & *key*, **bool** *cm*=**true**)
    {**return cycle_relation**(*key*, *cycle_adifferent*, *cm*);}

Defines:
  `is_adifferent`, used in chunk 24.
Uses `cycle_adifferent` 34f 113c, `cycle_relation` 40c 45e 46c 60d 61 62a 62b 64a 64b, `ex` 41b 47e 47e 47e 53a, and `key` 32e.

This relation check if a cycle is a non-positive vector, for circles this corresponds to real (non-imaginary) circles. By default we check this in the point space metric.

38g ⟨predefined cycle relations 38c⟩+≡ (47d) ◁ 38f 39a ▷

inline cycle_relation *is_real_cycle*(**const ex** & *key*, **bool** *cm*=**false**, **const ex** & *pr*=1)
 {**return cycle_relation**(*key*, *product_sign*, *cm*, *pr*);}

Defines:
  `is_real_cycle`, used in chunk 31b.
Uses `cycle_relation` 40c 45e 46c 60d 61 62a 62b 64a 64b, `ex` 41b 47e 47e 47e 53a, `key` 32e, and `product_sign` 34g 114d.

Effectively this is the same check but with a different name and other defaults. It may be used that both cycles are or are not separated by the light cone in the indefinite metric in space of cycles.

39a ⟨predefined cycle relations 38c⟩+≡ (47d) ◁ 38g 39b ▷

inline cycle_relation *product_nonpositive*(**const ex** & *key*, **bool** *cm*=**true**, **const ex** & *pr*=1)
 {**return cycle_relation**(*key*, *product_sign*, *cm*, *pr*);}

Defines:
  `product_nonpositive`, never used.
Uses `cycle_relation` 40c 45e 46c 60d 61 62a 62b 64a 64b, `ex` 41b 47e 47e 47e 53a, `key` 32e, and `product_sign` 34g 114d.

We may want to exclude cycles with imaginary coefficients, this condition check it.

39b ⟨predefined cycle relations 38c⟩+≡ (47d) ◁ 39a 39c ▷

inline cycle_relation *only_reals*(**const ex** & *key*, **bool** *cm*=**true**, **const ex** & *pr*=0)
 {**return cycle_relation**(*key*, *coefficients_are_real*, *cm*, *pr*);}

Defines:
  `only_reals`, used in chunks 21 and 31b.
Uses `coefficients_are_real` 34h 115c, `cycle_relation` 40c 45e 46c 60d 61 62a 62b 64a 64b, `ex` 41b 47e 47e 47e 53a, and `key` 32e.

This is tangency condition which shall be used to find tangent cycles.

39c ⟨predefined cycle relations 38c⟩+≡ (47d) ◁39b 39d▷

```
inline cycle_relation is_tangent(const ex & key, bool cm=true)
            {return cycle_relation(key, cycle_tangent, cm);}
```

Defines:
  `is_tangent`, used in chunk 21g.
Uses `cycle_relation` 40c 45e 46c 60d 61 62a 62b 64a 64b, `cycle_tangent` 46e 113e, `ex` 41b 47e 47e 47e 53a, and `key` 32e.

The split version for inner and outer tangent cycles.

39d ⟨predefined cycle relations 38c⟩+≡ (47d) ◁39c 39e▷

```
inline cycle_relation is_tangent_i(const ex & key, bool cm=true)
            {return cycle_relation(key, cycle_tangent_i, cm);}
inline cycle_relation is_tangent_o(const ex & key, bool cm=true)
            {return cycle_relation(key, cycle_tangent_o, cm);}
```

Defines:
  `is_tangent_i`, used in chunks 21a, 25b, 29b, and 31b.
  `is_tangent_o`, used in chunks 25b and 31b.
Uses `cycle_relation` 40c 45e 46c 60d 61 62a 62b 64a 64b, `cycle_tangent_i` 46e 114b, `cycle_tangent_o` 46e 114a, `ex` 41b 47e 47e 47e 53a, and `key` 32e.

The relation between cycles to "intersect with certain angle" (but the "intersection" may be imaginary). If cycles are intersecting indeed then the value of *pr* is the cosine of the angle.

39e ⟨predefined cycle relations 38c⟩+≡ (47d) ◁39d 39f▷

```
inline cycle_relation make_angle(const ex & key, bool cm=true, const ex & angle=0)
            {return cycle_relation(key, cycle_angle, cm, angle);}
```

Defines:
  `make_angle`, never used.
Uses `cycle_angle` 46e 114e, `cycle_relation` 40c 45e 46c 60d 61 62a 62b 64a 64b, `ex` 41b 47e 47e 47e 53a, and `key` 32e.

The next relation defines a generalisation of a Steiner power of a point for cycles.

39f ⟨predefined cycle relations 38c⟩+≡ (47d) ◁39e 39g▷

```
inline cycle_relation cycle_power(const ex & key, bool cm=true, const ex & cpower=0)
            {return cycle_relation(key, steiner_power, cm, cpower);}
```

Defines:
  `cycle_power`, never used.
Uses `cycle_relation` 40c 45e 46c 60d 61 62a 62b 64a 64b, `ex` 41b 47e 47e 47e 53a, `key` 32e, and `steiner_power` 46e 115a.

The next relation defines tangential distance between cycles.

39g ⟨predefined cycle relations 38c⟩+≡ (47d) ◁39f 39h▷

```
inline cycle_relation tangential_distance(const ex & key, bool cm=true, const ex & distance=0)
            {return cycle_relation(key, steiner_power, cm, pow(distance,2));}
```

Defines:
  `tangential_distance`, never used.
Uses `cycle_relation` 40c 45e 46c 60d 61 62a 62b 64a 64b, `ex` 41b 47e 47e 47e 53a, `key` 32e, and `steiner_power` 46e 115a.

The next relation defines cross-tangential distance between cycles.

39h ⟨predefined cycle relations 38c⟩+≡ (47d) ◁39g 40a▷

```
inline cycle_relation cross_t_distance(const ex & key, bool cm=true, const ex & distance=0)
            {return cycle_relation(key, cycle_cross_t_distance, cm, distance);}
```

Defines:
  `cross_t_distance`, never used.
Uses `cycle_cross_t_distance` 46e 115b, `cycle_relation` 40c 45e 46c 60d 61 62a 62b 64a 64b, `ex` 41b 47e 47e 47e 53a, and `key` 32e.

The next relation creates a cycle, which is a FLT of an existing cycle. The transformation is defined by a list of four entries which will make a 2×2 matrix. The default value corresponds to the identity map. User will need to use a proper Clifford algebra for the matrix to make this transformation works. In two dimensions the next method makes a relief.

40a ⟨predefined cycle relations 38c⟩+≡ (47d) ◁39h 40b▷

```
inline cycle_relation moebius_transform(const ex & key, bool cm=true,
                    const ex & matrix=lst{numeric(1),0,0,numeric(1)})
            {return cycle_relation(key, cycle_moebius, cm, matrix);}
```

Defines:
  `moebius_transform`, never used.
Uses `cycle_moebius` 47a 116b, `cycle_relation` 40c 45e 46c 60d 61 62a 62b 64a 64b, `ex` 41b 47e 47e 47e 53a, `key` 32e, and `numeric` 22d.

This is a simplified variant of the previous transformations for two dimension figures and transformations with real entries. The corresponding check will be carried out by the library. Then, the library will convert it into the proper Clifford valued matrix.

40b ⟨predefined cycle relations 38c⟩+≡ (47d) ◁40a

cycle_relation *sl2_transform*(**const ex** & *key*, **bool** *cm*=**true**,
const ex & **matrix**=**lst**{**numeric**(1),0,0,**numeric**(1)});

Defines:
`sl2_transform`, never used.
Uses `cycle_relation` 40c 45e 46c 60d 61 62a 62b 64a 64b, `ex` 41b 47e 47e 47e 53a, `key` 32e, and `numeric` 22d.

This is a constructor which creates a relation of the type *rel* to a node labelled by *key*. Boolean *cm* tells either to chose cycle metric or point metric for the relation. An additional parameter *p* can be supplied to the relation.

40c ⟨public methods for cycle relation 40c⟩≡ (46a)

cycle_relation(**const ex** & *key*, *PCR rel*, **bool** *cm*=**true**, **const ex** & *p*=0);

Defines:
`cycle_relation`, used in chunks 38–40, 43b, 45c, 46a, 52e, 60, 63a, 65, 70e, 71e, 73a, 81a, 83, 84a, 95e, 96d, and 116–18.
Uses `ex` 41b 47e 47e 47e 53a, `key` 32e, and `PCR` 45d.

There is also an additional method to define a joint relation to several parents by insertion of a **subfigure**, see *midpoint_constructor* below.

40d ⟨public methods for subfigure 40d⟩≡ (48b)

subfigure(**const ex** & *F*, **const ex** & *L*);

Defines:
`subfigure`, used in chunks 43d, 52e, 65, 67a, 70e, 71e, 84b, and 97.
Uses `ex` 41b 47e 47e 47e 53a.

Appendix D. Addtional utilities

Here is a procedure which returns a figure, which can be used to build a conformal version of the midpoint. The methods require three points, say *v1*, *v2* and *v3*. If *v3* is infinity, then the midpoint between *v1* and *v2* can be build using the orthogonality only. Put a cycle *v4* joining *v1*, *v2* and *v3*. Then construct a cycle *v5* with the diameter *v1–v2*, that is passing these points and orthogonal to *v4*. Then, put the cycle *v6* which passes *v3* and is orthogonal to *v4* and *v5*. The intersection *r* of *v6* and *v4* is the midpoint of *v1–v2*.

40e ⟨additional functions header 36a⟩+≡ (42a) ◁36a 40f▷

ex *midpoint_constructor*();

Defines:
`midpoint_constructor`, used in chunk 22f.
Uses `ex` 41b 47e 47e 47e 53a.

This utility make pair-wise comparison of cycles in the list *L* and deletes duplicates.

40f ⟨additional functions header 36a⟩+≡ (42a) ◁40e 40g▷

ex *unique_cycle*(**const ex** & *L*);

Defines:
`unique_cycle`, used in chunk 97a.
Uses `ex` 41b 47e 47e 47e 53a.

The debug output may be switched on and switched off by the following methods.

40g ⟨additional functions header 36a⟩+≡ (42a) ◁40f 41a▷

void *figure_debug_on*();
void *figure_debug_off*();
bool *figure_ask_debug_status*();

Defines:
`figure_ask_debug_status`, never used.
`figure_debug_off`, never used.
`figure_debug_on`, never used.

Solution of several quadratic equations in a sequence rapidly increases complexity of expression. We try to resolve this by some trigonometric or hyperbolic substitutions. Those expression in the turn need to be simplified as well in *evaluate_cycle*() for the condition *only_reals*. Later this variable will be assigned with a default list of trigonometric substitutions. User have a possibility to adjust this list in the run time.

41a ⟨additional functions header 36a⟩+≡ (42a) ◁40g

extern const ex *evaluation_assist*;

Defines:
`evaluation_assist`, used in chunks 89f and 91c.
Uses `ex` 41b 47e 47e 47e 53a.

Definition of the simplification rule.

41b ⟨figure library variables and constants 41b⟩≡ (52a) 47e ▷

```
const ex evaluation_assist = lst{power(cos(wild(0)),2)≡1-power(sin(wild(0)),2),
        power(cosh(wild(1)),2)≡1+power(sinh(wild(1)),2)};
```

Defines:
evaluation_assist, used in chunks 89f and 91c.
ex, used in chunks 16, 17, 19–25, 28, 30, 32–51, 53–70, 72, 74–92, 95d, 98–101, 103–105, and 107–119.

Appendix E. Figure Library Header File

Here is the header file of the library. Initially, an end-user does not need to know its structure much beyond the material presented in Sections B–C and illustrated in Section A. Here is some further topics which can be of interest as well:

- An intermediate end-user may wish to define his own **subfigure**s, see *midpoint_constructor* for a sample and Subsect. E.4.
- Furthermore, an advanced end-user may wish to define some additional **cycle_relation** to supplement already presented in Section C, in this case only knowledge of **cycle_relation** class is required, see Subsect. E.3.
- To adjust automatically created `Asymptote` graphics user may want to adjust the default styles, see Subsect. E.6.

41c ⟨figure.h 41c⟩≡ 41d ▷

```
⟨license 121⟩
#ifndef ____figure__
#define ____figure__
```

Defines:
____figure__, used in chunk 42a.

Some libraries we are using.

41d ⟨figure.h 41c⟩+≡ ◁ 41c 42a ▷

```
#include <iostream>
#include <cstdlib>
#include <cstdio>
#include <fstream>
#include <regex>
#include "cycle.h"

namespace MoebInv {
using namespace std;
using namespace GiNaC;
```

Defines:
MoebInv, used in chunks 42a and 52a.

The overview of the header file.

42a ⟨figure.h 41c⟩+≡ ◁ 41d

```
⟨figure define 42b⟩
⟨defining types 35g⟩
⟨cycle data header 42c⟩
⟨cycle node header 43b⟩
⟨cycle relations 45e⟩
⟨asy styles 51d⟩
⟨figure header 49a⟩
⟨subfigure header 48b⟩
⟨additional functions header 36a⟩
} // namespace MoebInv
#endif /* defined(____figure__) */
```

Uses ____figure__ 41c and MoebInv 16c 41d.

We use negative numbered generations to save the reference objects.

42b ⟨figure define 42b⟩≡ (42a)

```
#define REAL_LINE_GEN -1
#define INFINITY_GEN -2
#define GHOST_GEN -3
```

Defines:
`GHOST_GEN`, used in chunks 33g, 81a, 82d, 86e, 99a, 106d, and 107a.
`INFINITY_GEN`, used in chunks 75c, 76d, and 106d.
`REAL_LINE_GEN`, used in chunks 75d, 76d, 99b, and 101d.

E.1. **cycle_data class declaration.** The class to store explicit data of an individual **cycle**. An end-user does not need normally to know about it.

42c ⟨cycle data header 42c⟩≡ (42a) 42d ▷

```
class cycle_data : public basic
{
GINAC_DECLARE_REGISTERED_CLASS(cycle_data, basic)
```

Defines:
`cycle_data`, used in chunks 26b, 46b, 52–54, 56–59, 63a, 68–71, 75, 81, 83–85, 87c, 89, 95–97, 113c, 117c, and 119a.

The parameters of the stored **cycle**.

42d ⟨cycle data header 42c⟩+≡ (42a) ◁ 42c 43a ▷

```
protected:
    ex k_cd,
    l_cd,
    m_cd;
```

Uses `ex` 41b 47e 47e 47e 53a.

Public methods in the class. However, an end-user does not normally need them.

43a ⟨cycle data header 42c⟩+≡ (42a) ◁ 42d

```
public:
    cycle_data(const ex & C);
    cycle_data(const ex & k1, const ex l1, const ex &m1, bool normalize=false);
    ex make_cycle(const ex & metr) const;
    inline size_t nops() const { return 3; }
    ex op(size_t i) const;
    ex & let_op(size_t i);
    inline ex get_k() const { return k_cd; }
    inline ex get_l() const { return l_cd; }
    inline ex get_l(size_t i) const { return l_cd.op(0).op(i); }
    inline ex get_m() const { return m_cd;}
    inline long unsigned int get_dim() const { return l_cd.op(0).nops(); }
    void do_print(const print_dflt & con, unsigned level) const;
    void do_print_double(const print_dflt & con, unsigned level) const;
    void archive(archive_node &n) const;
    inline ex normalize() const {return cycle_data(k_cd,l_cd,m_cd,true);}
    ex num_normalize() const;
    void read_archive(const archive_node &n, lst &sym_lst);
    bool is_equal(const basic & other, bool projectively) const;
    bool is_almost_equal(const basic & other, bool projectively) const;
    cycle_data subs(const ex & e, unsigned options=0) const;
    ex subs(const exmap & em, unsigned options=0) const;
    inline bool has(const ex & x) const { return (k_cd.has(x) ∨ l_cd.has(x) ∨ m_cd.has(x)); }

protected:
    return_type_t return_type_tinfo() const;
};
GINAC_DECLARE_UNARCHIVER(cycle_data);
```

Defines:
`cycle_data`, used in chunks 26b, 46b, 52–54, 56–59, 63a, 68–71, 75, 81, 83–85, 87c, 89, 95–97, 113c, 117c, and 119a.
Uses `archive` 50a, `do_print_double` 49e, `ex` 41b 47e 47e 47e 53a, `get_dim()` 35c, `is_almost_equal` 117a, `nops` 50a, `op` 50a, `read_archive` 50a, and `subs` 50a.

E.2. **cycle_node class declaration.** Forward declaration.

43b ⟨cycle node header 43b⟩≡ (42a) 43c ▷
```
class cycle_relation;
```
Uses cycle_relation 40c 45e 46c 60d 61 62a 62b 64a 64b.

The class to store nodes containing data of particular cycles and relations between nodes. An end-user does not need normally to know about it.

43c ⟨cycle node header 43b⟩+≡ (42a) ◁ 43b 43d ▷
```
class cycle_node : public basic
{
GINAC_DECLARE_REGISTERED_CLASS(cycle_node, basic)
```
Defines:
cycle_node, used in chunks 32c, 35f, 37d, 44–46, 48, 49f, 51a, 52e, 63, 67–70, 72, 75, 76c, 79a, 81, 83c, 84b, 98, 99c, and 107–110.

Members of the class.

43d ⟨cycle node header 43b⟩+≡ (42a) ◁ 43c 44a ▷
```
protected:
    lst cycles; // List of cycle data entries
    int generation;
    lst children; // List of keys to cycle_nodes
    lst parents; // List of cycle_relations or a list containing a single subfigure
    string custom_asy; // Custom string for Asymptote
```
Uses subfigure 40d 48b 48d 66a 66b 66c 66d 66e.

Constructors in the class.

44a ⟨cycle node header 43b⟩+≡ (42a) ◁ 43d 44b ▷
```
public:
    cycle_node(const ex & C, int g=0);
    cycle_node(const ex & C, int g, const lst & par);
    cycle_node(const ex & C, int g, const lst & par, const lst & chil);
    cycle_node(const ex & C, int g, const lst & par, const lst & chil, string ca);
    cycle_node subs(const ex & e, unsigned options=0) const;
    void do_print_double(const print_dflt & con, unsigned level) const;
    ex subs(const exmap & m, unsigned options=0) const;
```
Uses cycle_node 43c 45c 69c 70a 70b 70c 71a 71b 72b 72c 73a 74a 74d, do_print_double 49e, ex 41b 47e 47e 47e 53a, m 51c, and subs 50a.

Add a chid **cycle_node** to the **cycle_node**.

44b ⟨cycle node header 43b⟩+≡ (42a) ◁ 44a 44c ▷
```
protected:
    inline void add_child(const ex & c) {children.append(c);}
```
Uses ex 41b 47e 47e 47e 53a.

Access **cycle** parameters.

44c ⟨cycle node header 43b⟩+≡ (42a) ◁ 44b 44d ▷
```
    inline ex get_cycles_data() const {return cycles;}
```
Uses ex 41b 47e 47e 47e 53a.

Return the **cycle** object for every **cycle_data** stored in *cycles*.

44d ⟨cycle node header 43b⟩+≡ (42a) ◁ 44c 44e ▷
```
    ex make_cycles(const ex & metr) const;
    inline ex get_cycle_data(int i) const {return cycles.op(i);}
```
Uses ex 41b 47e 47e 47e 53a and op 50a.

Return the generation number.

44e ⟨cycle node header 43b⟩+≡ (42a) ◁ 44d 44f ▷
```
    inline int get_generation() const {return generation;}
```
Uses get_generation 35f.

Return the children list

44f ⟨cycle node header 43b⟩+≡ (42a) ◁44e 44g▷
```
inline lst get_children() const {return children;}
```

Replace the current **cycle** with a new **cycle**.

44g ⟨cycle node header 43b⟩+≡ (42a) ◁44f 44h▷
```
void set_cycles(const ex & C);
```

Uses ex 41b 47e 47e 47e 53a.

Add one more **cycle** instance to list of *cycles*.

44h ⟨cycle node header 43b⟩+≡ (42a) ◁44g 44i▷
```
void append_cycle(const ex & C);
void append_cycle(const ex & k, const ex & l, const ex & m);
```

Uses ex 41b 47e 47e 47e 53a, k 51c, l 51c, and m 51c.

Return the parent list.

44i ⟨cycle node header 43b⟩+≡ (42a) ◁44h 44j▷
```
lst get_parents() const;
```

The method returns the list of all keys to parant cycles.

44j ⟨cycle node header 43b⟩+≡ (42a) ◁44i 45a▷
```
lst get_parent_keys() const ;
```

Remove a child of the **cycle_node**.

45a ⟨cycle node header 43b⟩+≡ (42a) ◁44j 45b▷
```
void remove_child(const ex & c);
```

Uses ex 41b 47e 47e 47e 53a.

Set or read `Asymptote` option for this particular node.

45b ⟨cycle node header 43b⟩+≡ (42a) ◁45a 45c▷
```
inline void set_asy_opt(const string opt)  {custom_asy=opt;}
inline string get_asy_opt() const {return custom_asy;}
```

Service functions including printout the mathematical expression.

45c ⟨cycle node header 43b⟩+≡ (42a) ◁45b
```
    inline size_t nops() const { return cycles.nops()+children.nops()+parents.nops(); }
    ex op(size_t i) const;
    ex & let_op(size_t i);
    void do_print(const print_dflt & con, unsigned level) const;
    void do_print_tree(const print_tree & con, unsigned level) const;
 protected:
    return_type_t return_type_tinfo() const;
    void archive(archive_node &n) const;
    void read_archive(const archive_node &n, lst &sym_lst);

 friend class cycle_relation;
 friend class figure;
 };
 GINAC_DECLARE_UNARCHIVER(cycle_node);
```

Defines:
  cycle_node, used in chunks 32c, 35f, 37d, 44–46, 48, 49f, 51a, 52e, 63, 67–70, 72, 75, 76c, 79a, 81, 83c, 84b, 98, 99c, and 107–110.

Uses archive 50a, cycle_relation 40c 45e 46c 60d 61 62a 62b 64a 64b, ex 41b 47e 47e 47e 53a, figure 16d 22e 32a 32c 38b 49a 50d 75a 80a 82b 82c 85a 86c 98c 99b 99d 100a 101a 103b 104a 105c 106c 106d 107a 109a 109c 110a, nops 50a, op 50a, and read_archive 50a.

E.3. **cycle_relation class declaration.** First, we define a type to hold cycle relations. That is a pointer to a functions with two arguments. See the definition of *cycle_orthogonal*, *cycle_different*, for samples.

45d ⟨defining types 35g⟩+≡ (42a) ◁35g

```
using PCR = std::function<ex(const ex &, const ex &, const ex &)>;
```

Defines:
PCR, used in chunks 34a, 35a, 40c, 45e, 46a, 60b, 110c, and 111c.
Uses ex 41b 47e 47e 47e 53a.

This class describes relations between **cycle_node**s. An advanced end-user may want to add some new relations similar to already provided in Section C. Note however, that archiving (saving) of user-defined relations cannot be done as they contain pointers to functions which are not portable.
Memebrs of the class.

45e ⟨cycle relations 45e⟩≡ (42a) 46a ▷

```
class cycle_relation : public basic
{
    GINAC_DECLARE_REGISTERED_CLASS(cycle_relation, basic)
protected:
    ex parkey; // A key to a parent cycle_node in figure
    PCR rel; // A pointer to function which produces the relation
    ex parameter; // The value, which is supplied to rel() as the third parameter
    bool use_cycle_metric; // If true uses the cycle space metric, otherwise the point space metric
```

Defines:
cycle_relation, used in chunks 38–40, 43b, 45c, 46a, 52e, 60, 63a, 65, 70e, 71e, 73a, 81a, 83, 84a, 95e, 96d, and 116–18.
Uses cycle_node 43c 45c 69c 70a 70b 70c 71a 71b 72b 72c 73a 74a 74d, ex 41b 47e 47e 47e 53a, figure 16d 22e 32a 32c 38b 49a 50d 75a 80a 82b 82c 85a 86c 98c 99b 99d 100a 101a 103b 104a 105c 106c 106d 107a 109a 109c 110a, key 32e, and PCR 45d.

Public methods in the class.

46a ⟨cycle relations 45e⟩+≡ (42a) ◁45e 46b ▷

```
public:
    ⟨public methods for cycle relation 40c⟩
    inline ex get_parkey() const {return parkey;}
    inline PCR get_PCR() const {return rel;}
    inline ex get_parameter() const {return parameter;}
    inline bool cycle_metric_inuse() const {return use_cycle_metric;}
    inline ex subs(const exmap & em, unsigned options=0) const
    {return cycle_relation(parkey, rel, use_cycle_metric, parameter.subs(em,options));}
```

Uses cycle_relation 40c 45e 46c 60d 61 62a 62b 64a 64b, ex 41b 47e 47e 47e 53a, PCR 45d, and subs 50a.

Protected methods in the class. The next method creates relation of *C1* to its parent. *C1* shall be in the **cycle_data** class.

46b ⟨cycle relations 45e⟩+≡ (42a) ◁46a 46c ▷

```
protected:
    ex rel_to_parent(const ex & C1, const ex & pmetric, const ex & cmetric,
                     const exhashmap<cycle_node> & N) const;
```

Uses cycle_node 43c 45c 69c 70a 70b 70c 71a 71b 72b 72c 73a 74a 74d and ex 41b 47e 47e 47e 53a.

Service methods in the class.

46c ⟨cycle relations 45e⟩+≡ (42a) ◁46b 46d ▷

```
    return_type_t return_type_tinfo() const;
    void do_print(const print_dflt & con, unsigned level) const;
    void do_print_tree(const print_tree & con, unsigned level) const;
```

(un)Archiving of **cycle_relation** is not universal. At present it only can handle relations declared in the header file *cycle_orthogonal*, *cycle_f_orthogonal*, *cycle_adifferent*, *cycle_different*, *cycle_tangent*, *cycle_power* etc. from Subsection C.

46d ⟨cycle relations 45e⟩+≡ (42a) ◁46c 46e▷
```
    void archive(archive_node &n) const;
    void read_archive(const archive_node &n, lst &sym_lst);

    inline size_t nops() const { return 2; }
    ex op(size_t i) const;
    ex & let_op(size_t i);

friend class cycle_node;
friend class figure;
};
GINAC_DECLARE_UNARCHIVER(cycle_relation);
```

Defines:
`cycle_relation`, used in chunks 38–40, 43b, 45c, 46a, 52e, 60, 63a, 65, 70e, 71e, 73a, 81a, 83, 84a, 95e, 96d, and 116–18.

Uses `archive` 50a, `cycle_node` 43c 45c 69c 70a 70b 70c 71a 71b 72b 72c 73a 74a 74d, `ex` 41b 47e 47e 47e 53a, `figure` 16d 22e 32a 32c 38b 49a 50d 75a 80a 82b 82c 85a 86c 98c 99b 99d 100a 101a 103b 104a 105c 106c 106d 107a 109a 109c 110a, `nops` 50a, `op` 50a, and `read_archive` 50a.

The following functions are used as *PCR* pointers for corresponding cycle relations.

46e ⟨cycle relations 45e⟩+≡ (42a) ◁46d 47a▷
⟨relations to check 34b⟩

The following procedures are used to construct relations but are impractical to check.

47a ⟨cycle relations 45e⟩+≡ (42a) ◁46e 47b▷
```
ex cycle_tangent(const ex & C1, const ex & C2, const ex & pr=0);
ex cycle_tangent_i(const ex & C1, const ex & C2, const ex & pr=0);
ex cycle_tangent_o(const ex & C1, const ex & C2, const ex & pr=0);
ex cycle_angle(const ex & C1, const ex & C2, const ex & pr);
ex steiner_power(const ex & C1, const ex & C2, const ex & pr);
ex cycle_cross_t_distance(const ex & C1, const ex & C2, const ex & pr);
```
Defines:
`cycle_angle`, used in chunks 39e, 61, 62a, and 64a.
`cycle_cross_t_distance`, used in chunks 39h, 61, 62a, and 64a.
`cycle_tangent`, used in chunks 39c, 61, 62a, and 64a.
`cycle_tangent_i`, used in chunks 39d, 61, 62a, and 64a.
`cycle_tangent_o`, used in chunks 39d, 61, 62a, and 64a.
`steiner_power`, used in chunks 39, 61, 62a, and 64a.

Uses `ex` 41b 47e 47e 47e 53a.

Fractional linear transformations.

47b ⟨cycle relations 45e⟩+≡ (42a) ◁47a 47c▷
```
ex cycle_moebius(const ex & C1, const ex & C2, const ex & pr);
ex cycle_sl2(const ex & C1, const ex & C2, const ex & pr);
```
Defines:
`cycle_moebius`, used in chunks 40a, 61, 62a, and 64a.
`cycle_sl2`, used in chunks 61, 62a, 64a, and 116c.

Uses `ex` 41b 47e 47e 47e 53a.

The next functions are used to measure certain quantities between cycles.

47c ⟨cycle relations 45e⟩+≡ (42a) ◁47b 47d▷
```
ex power_is(const ex & C1, const ex & C2, const ex & pr=1);
inline ex sq_t_distance_is(const ex & C1, const ex & C2, const ex & pr=1)
 {return power_is(C1,C2,1);}
inline ex sq_cross_t_distance_is(const ex & C1, const ex & C2, const ex & pr=-1)
 {return power_is(C1,C2,-1);}
ex angle_is(const ex & C1, const ex & C2, const ex & pr=0);
```
Defines:
`angle_is`, never used.
`power_is`, never used.
`sq_cross_t_distance_is`, never used.
`sq_t_distance_is`, never used.

Uses `ex` 41b 47e 47e 47e 53a.

We include the list of pre-defined metrics in two dimensions.

47d ⟨cycle relations 45e⟩+≡ (42a) ◁47c 48a▷
⟨predefined cycle relations 38c⟩

We explicitly define three types of metrics on a plane: elliptic, parabolic, hyperbolic.

48a ⟨cycle relations 45e⟩+≡ (42a) ◁47d ??▷

```
extern const ex metric_e, metric_p, metric_h;
```

Defines:
metric_e, used in chunk 48a.
metric_h, used in chunk 48a.
metric_p, used in chunk 48a.
Uses ex 41b 47e 47e 47e 53a.

The predefined metrics are based on diagonal matrices with different signatures.

47e ⟨figure library variables and constants 41b⟩+≡ (52a) ◁41b 52b▷

```
const ex metric_e = clifford_unit(varidx(symbol("i"), numeric(2)), indexed(diag_matrix(lst{-1,-1}), sy_symm(), varidx(symbol("j"), numeric(2)), varidx(symbol("k"), numeric(2))));
const ex metric_p = clifford_unit(varidx(symbol("i"), numeric(2)), indexed(diag_matrix(lst{-1,0}), sy_symm(), varidx(symbol("j"), numeric(2)), varidx(symbol("k"), numeric(2))));
const ex metric_h = clifford_unit(varidx(symbol("i"), numeric(2)), indexed(diag_matrix(lst{-1,1}), sy_symm(), varidx(symbol("j"), numeric(2)), varidx(symbol("k"), numeric(2))));
```

Defines:
ex, used in chunks 16, 17, 19–25, 28, 30, 32–51, 53–70, 72, 74–92, 95d, 98–101, 103–105, and 107–119.
metric_e, used in chunk 48a.
metric_h, used in chunk 48a.
metric_p, used in chunk 48a.
Uses k 51c and numeric 22d.

There is the list of pre-defined metrics in two dimensions cycle relations. Orthogonality of cycles of three types independent from a metric stored in the figure.

?? ⟨cycle relations 45e⟩+≡ (42a) ◁48a

```
inline ex cycle_orthogonal_e(const ex & C1, const ex & C2, const ex & pr=0) {
    return lst{(ex)lst{ex_to<cycle>(C1).is_orthogonal(ex_to<cycle>(C2), metric_e)}};}

inline ex cycle_orthogonal_p(const ex & C1, const ex & C2, const ex & pr=0) {
    return lst{(ex)lst{ex_to<cycle>(C1).is_orthogonal(ex_to<cycle>(C2), metric_p)}};}

inline ex cycle_orthogonal_h(const ex & C1, const ex & C2, const ex & pr=0) {
    return lst{(ex)lst{ex_to<cycle>(C1).is_orthogonal(ex_to<cycle>(C2), metric_h)}};}
```

Defines:
cycle_orthogonal_e, never used.
cycle_orthogonal_h, never used.
cycle_orthogonal_p, never used.
Uses ex 41b 47e 47e 47e 53a, is_orthogonal 23c 38c, metric_e 47d 47e, metric_h 47d 47e, and metric_p 47d 47e.

E.4. **subfigure class declaration. subfigure] class** *allows to encapsulate some common constructions.*

The library provides an important example [[*midpoint*. End-user may define his own **subfigure**s, they will not be handled as native ones, including (un)archiving.

In the essence **subfigure** is created from a **figure**, which were designed to be included in another figures.

48b ⟨subfigure header 48b⟩≡ (42a) 48c▷

```
class subfigure : public basic
{
    GINAC_DECLARE_REGISTERED_CLASS(subfigure, basic)
protected:
    ex subf; // A figure to be inserted
    lst parlist; // A list of key to a parent cycle_node in figure
public:
    ⟨public methods for subfigure 40d⟩
    inline ex subs(const exmap & em, unsigned options=0) const;
```

Defines:
subfigure, used in chunks 43d, 52e, 65, 67a, 70e, 71e, 84b, and 97.
Uses cycle_node 43c 45c 69c 70a 70b 70c 71a 71b 72b 72c 73a 74a 74d, ex 41b 47e 47e 47e 53a, figure 16d 22e 32a 32c 38b 49a 50d 75a 80a 82b 82c 85a 86c 98c 99b 99d 100a 101a 103b 104a 105c 106c 106d 107a 109a 109c 110a, key 32e, and subs 50a.

Some service methods.

48c ⟨subfigure header 48b⟩+≡ (42a) ◁48b 48d▷

```
  protected:
    inline ex get_parlist() const {return parlist;}
    inline ex get_subf() const {return subf;}
    return_type_t return_type_tinfo() const;
    void do_print(const print_dflt & con, unsigned level) const;
    void do_print_tree(const print_tree & con, unsigned level) const;
```

Uses `ex` 41b 47e 47e 47e 53a.

(un)Archiving of **cycle_relation** is not universal. At present it only can handle relations declared in the header file *p_orthogonal*, *p_f_orthogonal*, *p_adifferent*, *p_different* and *p_tangent* etc. from Subsection C.

48d ⟨subfigure header 48b⟩+≡ (42a) ◁48c

```
    void archive(archive_node &n) const;
    void read_archive(const archive_node &n, lst &sym_lst);

  friend class cycle_node;
  friend class figure;
  };
  GINAC_DECLARE_UNARCHIVER(subfigure);
```

Defines:
`subfigure`, used in chunks 43d, 52e, 65, 67a, 70e, 71e, 84b, and 97.
Uses `archive` 50a, `cycle_node` 43c 45c 69c 70a 70b 70c 71a 71b 72b 72c 73a 74a 74d, `figure` 16d 22e 32a 32c 38b 49a 50d 75a 80a 82b 82c 85a 86c 98c 99b 99d 100a 101a 103b 104a 105c 106c 106d 107a 109a 109c 110a, and `read_archive` 50a.

E.5. **figure class declaration.** The essential interface to **figure** class was already presented in Section B, here we keep the less-used elements. An advanced end-user may be interested in **figure** class members given in § E.5.1.
We define **figure** class as a children of GiNaC **basic**.

49a ⟨figure header 49a⟩≡ (42a) 49b▷

```
  class figure : public basic
  {
  GINAC_DECLARE_REGISTERED_CLASS(figure, basic)

  ⟨member of figure class 50e⟩
```

Defines:
`figure`, used in chunks 16–18, 20e, 22, 27b, 28a, 30a, 36, 37, 45, 46c, 48, 50a, 52, 66, 75–84, 86, 87, 95d, 97–100, 106–108, 110, 111, and 117c.

The method to update **cycle_node** with labelled by the *key*. Since the list of conditions may branches and has a variable length the method runs recursively with *level* parameterising the depth of nested calls.

49b ⟨figure header 49a⟩+≡ (42a) ◁49a 49c▷

```
  protected:
    ex update_cycle_node(const ex & key, const lst & eq_cond=lst{},
                         const lst & neq_cond=lst{}, lst res=lst{}, size_t level=0);
    void set_cycle(const ex & key, const ex & C);
```

Defines:
`set_cycle`, used in chunks 82c and 97c.
`update_cycle_node`, used in chunks 81c, 83d, 84b, 86a, 97a, 98a, and 100b.
Uses `ex` 41b 47e 47e 47e 53a and `key` 32e.

Evaluate a cycle through a list of conditions.

49c ⟨figure header 49a⟩+≡ (42a) ◁49b 49d▷

```
    ex evaluate_cycle(const ex & symbolic, const lst & cond) const;
```

Uses `evaluate_cycle` 87a and `ex` 41b 47e 47e 47e 53a.

We include here methods from Section B, which are of interest for an end-user.

49d ⟨figure header 49a⟩+≡ (42a) ◁49c 49e▷

```
  public:
    ⟨public methods in figure class 32a⟩
```

The following methods are public as well however may be less used.

49e ⟨figure header 49a⟩+≡ (42a) ◁49d 50b▷

inline ex *get_cycle_node*(**const ex** & *ck*) **const** {**return** *nodes.find*(*ck*)→*second*;}
void *do_print_double*(**const** *print_dflt* & *con*, **unsigned** *level*) **const**;

Defines:
`do_print_double`, used in chunks 43a, 44a, 55a, 71b, and 107a.
`get_cycle_node`, used in chunks 35f, 37d, and 106d.
Uses `ex` 41b 47e 47e 47e 53a and `nodes` 51a.

The method returning all nodes.

49g ⟨public methods in figure class 32a⟩+≡ (49d) ◁49f 49h▷

inline *exhashmap*<**cycle_node**> *get_nodes*() **const** {**return** *nodes*;}

Uses `cycle_node` 43c 45c 69c 70a 70b 70c 71a 71b 72b 72c 73a 74a 74d and `nodes` 51a.

Sometimes we need access to predefined *infinity* or the *real_line*, for example to specify a cycle relation to them.

49h ⟨public methods in figure class 32a⟩+≡ (49d) ◁49g 50a▷

inline ex *get_real_line*() **const** {**return** *real_line*;}
inline ex *get_infinity*() **const** {**return** *infinity*;}

Defines:
`get_infinity`, used in chunks 21a, 22a, and 29b.
`get_real_line`, used in chunk 17d.
Uses `ex` 41b 47e 47e 47e 53a, `infinity` 50e, and `real_line` 50e.

Return the maximal generation number of cycles in this figure.

50a ⟨public methods in figure class 32a⟩+≡ (49d) ◁49h ??▷

int *get_max_generation*() **const**;

Defines:
`get_max_generation`, used in chunk 99a.

Some standard GiNaC methods which are not very interesting for end-user, who is working within functional programming set-up.

?? ⟨public methods in figure class 32a⟩+≡ (49d) ◁50a

inline *size_t nops*() **const** {**return** 4+*nodes.size*();}
ex *op*(*size_t i*) **const**;
//ex & let_op(size_t i);
ex *evalf*(**int** *level*=0) **const**;
figure *subs*(**const ex** & *e*, **unsigned** *options*=0) **const**;
ex *subs*(**const** *exmap* & *m*, **unsigned** *options*=0) **const**;
void *archive*(*archive_node* &*n*) **const**;
void *read_archive*(**const** *archive_node* &*n*, **lst** &*sym_lst*);
bool *info*(**unsigned** *inf*) **const**;

Defines:
`archive`, used in chunks 43a, 45c, 46c, 48d, 56b, 61, 62a, 66, 73a, 79c, 80a, and 109a.
`evalf`, used in chunks 19f, 25c, 29, 53b, 55, 56a, 88a, 89f, 91e, 92b, 94b, 95c, 102a, 106a, 108c, 114d, and 115c.
`info`, used in chunks 72d, 81c, 83d, 84b, 86a, 91, 92, 96b, 98a, 100a, 108a, 110b, and 116c.
`nops`, used in chunks 19c, 21e, 22a, 30d, 43a, 45c, 46c, 56e, 57a, 65, 69, 70e, 73a, 76b, 77d, 80–82, 85–93, 95–98, 100a, 106, 107, 110d, 117b, and 119a.
`op`, used in chunks 19f, 21, 22a, 35c, 43–46, 54–56, 59d, 65a, 69a, 70e, 72d, 76–78, 81, 85c, 88–95, 97, 102, 106–108, 110e, 111c, and 116.
`read_archive`, used in chunks 43a, 45c, 46c, 48d, 56c, 62a, 66c, 74a, and 109c.
`subs`, used in chunks 19f, 21e, 31c, 43a, 44a, 46a, 48b, 54a, 59, 67a, 72, 81d, 89–91, 93–95, 104c, and 108.
Uses `ex` 41b 47e 47e 47e 53a, `figure` 16d 22e 32a 32c 38b 49a 50d 75a 80a 82b 82c 85a 86c 98c 99b 99d 100a 101a 103b 104a 105c 106c 106d 107a 109a 109c 110a, `m` 51c, and `nodes` 51a.

Printing and returning the objects list.

50b ⟨figure header 49a⟩+≡ (42a) ◁49e 50c▷

protected:
void *do_print*(**const** *print_dflt* & *con*, **unsigned** *level*) **const**;
return_type_t return_type_tinfo() **const**;

Update all cycles (with all children) in the given list.

50c ⟨figure header 49a⟩+≡ (42a) ◁50b 50d▷

void *update_node_lst*(**const ex** & *inlist*);

Defines:
`update_node_lst`, used in chunks 83a, 86, and 98b.
Uses `ex` 41b 47e 47e 47e 53a.

Update the entire figure.

50d ⟨figure header 49a⟩+≡ (42a) ◁50c

```
    figure update_cycles();
};
GINAC_DECLARE_UNARCHIVER(figure);
```

Defines:
`figure`, used in chunks 16–18, 20e, 22, 27b, 28a, 30a, 36, 37, 45, 46c, 48, 50a, 52, 66, 75–84, 86, 87, 95d, 97–100, 106–108, 110, 111, and 117c.
`update_cycles`, used in chunks 98d and 108b.

E.5.1. *Members of* **figure** *class.* A knowledge of **figure** class members may be useful for advanced users. The real line and infinity are two cycles which are present at any figure.

50e ⟨member of figure class 50e⟩≡ (49a) 50f▷

```
protected:
 ex real_line, // the key for the real line
     infinity; // the key for cycle at infinity
```

Defines:
`infinity`, used in chunks 49g, 75, 76d, 79, 81a, 107, and 109.
`real_line`, used in chunks 49g, 75–77, 79, 107, and 109.
Uses `ex` 41b 47e 47e 47e 53a and `key` 32e.

We define separate metrics for the point and cycle spaces, see [36, § 4.2].

50f ⟨member of figure class 50e⟩+≡ (49a) ◁50e 51a▷

```
    ex point_metric; // The metric of the point space encoded as a clifford_unit object
    ex cycle_metric; // The metric of the cycle space encoded as a clifford_unit object
```

Defines:
`cycle_metric`, used in chunks 35, 75–79, 95–97, 102a, and 107–111.
`point_metric`, used in chunks 35, 75–77, 79a, 95–97, 101d, and 107–111.
Uses `ex` 41b 47e 47e 47e 53a.

This is the *hashmap* of **cycle_node** which encode the relation in the figure.

51a ⟨member of figure class 50e⟩+≡ (49a) ◁50f 51b▷

```
    exhashmap<cycle_node> nodes; // List of cycle_node, exhashmap<cycle_node> object
```

Defines:
`nodes`, used in chunks 37d, 49, 50a, 75, 76c, 79b, 81–86, 95–101, and 107–111.
Uses `cycle_node` 43c 45c 69c 70a 70b 70c 71a 71b 72b 72c 73a 74a 74d.

The following variable controls either we are doing exact or float evaluations of cycles parameters.

51b ⟨member of figure class 50e⟩+≡ (49a) ◁51a 51c▷

```
    bool float_evaluation=false;
```

Defines:
`float_evaluation`, used in chunks 37c, 88a, 89f, 95c, 97b, and 109.

A string to record any information related to the figure. This library does not parse its content: it is primary intended for humans.

51c ⟨member of figure class 50e⟩+≡ (49a) ◁51b ??▷

```
    string info_text;
```

Defines:
`info_text`, never used.

These are symbols for internal calculations, they are out of the interest we do not count them in *nops*() methods.

?? ⟨member of figure class 50e⟩+≡ (49a) ◁51c

```
    ex k, m; // realsymbols for symbolic calculations
    lst l;
```

Defines:
`k`, used in chunks 19, 20, 35, 37d, 42–44, 47e, 49e, 53–60, 70b, 75, 79, 97d, and 106d.
`l`, used in chunks 20, 21, 42–44, 53–57, 59, 64a, 65d, 70b, 72d, 75, 78, 79, 84c, and 97d.
`m`, used in chunks 42–44, 50a, 53, 54, 56–59, 64a, 70b, 75, 79, 97d, and 108.
Uses `ex` 41b 47e 47e 47e 53a.

E.6. **Asymptote customization.** The library provides a possibility to fine-tune Asymptote output. We provide some default styles, a user may customise them according to existing needs.
We define a type for producing colouring scheme for Asymptote drawing.

51d ⟨asy styles 51d⟩≡ (42a) 51e ▷

```
using asy_style=std::function<string(const ex &, const ex &, lst &)>;
//typedef string (*asy_style)(const ex &, const ex &, lst &);
inline string no_color(const ex & label, const ex & C, lst & color) {color=lst{0,0,0}; return "";}
string asy_cycle_color(const ex & label, const ex & C, lst & color);
const asy_style default_asy=asy_cycle_color;
```

Defines:
asy_style, used in chunks 36, 37a, 101a, 103b, and 104a.
Uses asy_cycle_color 112a and ex 41b 47e 47e 47e 53a.

Similarly we produce a default labelling style.

51e ⟨asy styles 51d⟩+≡ (42a) ◁ 51d

```
using label_string=std::function<string(const ex &, const ex &, const string)>;
string label_pos(const ex & label, const ex & C, const string draw_str);
inline string no_label(const ex & label, const ex & C, const string draw_str) {return "";}
const label_string default_label=label_pos;
```

Defines:
label_string, used in chunks 36, 37a, 101a, 103b, and 104a.
Uses ex 41b 47e 47e 47e 53a and label_pos 112b.

Appendix F. Implementation of Classes

This is the outline of the code.

52a ⟨figure.cpp 52a⟩≡

```
⟨license 121⟩
#include <iostream>

#if __cplusplus >= 201703L
  #include <filesystem>
#endif

#include "figure.h"

namespace MoebInv {
using namespace std;
using namespace GiNaC;
⟨figure library variables and constants 41b⟩
⟨GiNaC declarations 52e⟩
⟨auxillary function 53a⟩
⟨add cycle relations 113a⟩
⟨cycle data class 53c⟩
⟨cycle relation class 60a⟩
⟨subfigure class 65c⟩
⟨cycle node class 67b⟩
⟨figure class 75a⟩
⟨addional functions 117a⟩
} // namespace MoebInv
```

Uses figure 16d 22e 32a 32c 38b 49a 50d 75a 80a 82b 82c 85a 86c 98c 99b 99d 100a 101a 103b 104a 105c 106c 106d 107a 109a 109c 110a and MoebInv 16c 41d.

52b ⟨figure library variables and constants 41b⟩+≡ (52a) ◁ 47e 52c ▷

```
unsigned do_not_update_subfigure = 0x0100;
```

Defines:
do_not_update_subfigure, used in chunks 67a and 108b.

This can de defined **false** to prevent some diagnostic output to *std*::*cerr*.

52c ⟨figure library variables and constants 41b⟩+≡ (52a) ◁52b 52d▷

bool *FIGURE_DEBUG*=**true**;

Defines:
`FIGURE_DEBUG`, used in chunks 71e, 79–83, 85, 86, 99c, 102d, 103a, 106d, 107a, and 119b.

This can de defined **false** to prevent some diagnostic output to *std*::*cerr*.

52d ⟨figure library variables and constants 41b⟩+≡ (52a) ◁52c

bool *show_asy_graphics*=**true**;

Defines:
`show_asy_graphics`, used in chunks 103d, 105b, and 119c.

We use GiNaC implementation macros for our classes.

52e ⟨GiNaC declarations 52e⟩≡ (52a)

GINAC_IMPLEMENT_REGISTERED_CLASS_OPT(**cycle_data**, **basic**,
print_func<*print_dflt*>(&**cycle_data**::*do_print*))

GINAC_IMPLEMENT_REGISTERED_CLASS_OPT(**cycle_relation**, **basic**,
print_func<*print_dflt*>(&**cycle_relation**::*do_print*).
print_func<*print_tree*>(&**cycle_relation**::*do_print_tree*))

GINAC_IMPLEMENT_REGISTERED_CLASS_OPT(**subfigure**, **basic**,
print_func<*print_dflt*>(&**subfigure**::*do_print*))

GINAC_IMPLEMENT_REGISTERED_CLASS_OPT(**cycle_node**, **basic**,
print_func<*print_dflt*>(&**cycle_node**::*do_print*).
print_func<*print_tree*>(&**cycle_relation**::*do_print_tree*))

GINAC_IMPLEMENT_REGISTERED_CLASS_OPT(**figure**, **basic**,
print_func<*print_dflt*>(&**figure**::*do_print*))

Uses `cycle_data` 23a 26a 42c 43a 54c 54d 55a 56b 56c 56d, `cycle_node` 43c 45c 69c 70a 70b 70c 71a 71b 72b 72c 73a 74a 74d, `cycle_relation` 40c 45e 46c 60d 61 62a 62b 64a 64b, `figure` 16d 22e 32a 32c 38b 49a 50d 75a 80a 82b 82c 85a 86c 98c 99b 99d 100a 101a 103b 104a 105c 106c 106d 107a 109a 109c 110a, and `subfigure` 40d 48b 48d 66a 66b 66c 66d 66e.

Exact solving of quadratic equations is not always practical, thus we relay on some rounding methods. If the outcome is not good for you increase the precision with *GiNaC*::*Digits*.

53a ⟨auxillary function 53a⟩≡ (52a) 53b▷

const ex *epsilon*=*GiNaC*::*pow*(10,-*Digits*÷2);

Defines:
`epsilon`, used in chunks 18b, 19f, 53b, and 114d.
`ex`, used in chunks 16, 17, 19–25, 28, 30, 32–51, 53–70, 72, 74–92, 95d, 98–101, 103–105, and 107–119.

an auxillary function to find small numbers

53b ⟨auxillary function 53a⟩+≡ (52a) ◁53a

bool *is_less_than_epsilon*(**const ex** & *x*)
{
  return (*x.is_zero*() ∨ *abs*(*x*).*evalf*() < *epsilon*) ;
}

Defines:
`is_less_than_epsilon`, used in chunks 58, 59, 89d, 91, 93, 94, 97a, 102a, 112a, 114d, 115c, and 117.
Uses `epsilon` 53a, `evalf` 50a, and `ex` 41b 47e 47e 47e 53a.

F.1. **Implementation of cycle_data class.** Constructors

53c ⟨cycle data class 53c⟩≡ (52a) 53d▷

cycle_data::**cycle_data**() : *k_cd*(), *l_cd*(), *m_cd*()
{
  ;
}

Uses `cycle_data` 23a 26a 42c 43a 54c 54d 55a 56b 56c 56d.

Constructors

53d ⟨cycle data class 53c⟩+≡ (52a) ◁53c 54a▷

```
cycle_data::cycle_data(const ex & C)
{
    if (is_a<cycle>(C)) {
        cycle C_new=ex_to<cycle>(C).normalize();
        ⟨cycle data class constructor common ??⟩
```

Uses cycle_data 23a 26a 42c 43a 54c 54d 55a 56b 56c 56d and ex 41b 47e 47e 47e 53a.

This part of the code will be recycled.

?? ⟨cycle data class constructor common ??⟩≡ (53d 54a)

```
        k_cd=C_new.get_k();
        l_cd=C_new.get_l();
        m_cd=C_new.get_m();
```

similarly we copy **cycle_data** object.

54a ⟨cycle data class 53c⟩+≡ (52a) ◁53d 54b▷

```
    } else if (is_a<cycle_data>(C)) {
        cycle_data C_new=ex_to<cycle_data>(C);
        ⟨cycle data class constructor common ??⟩
    } else
        throw(std::invalid_argument("cycle_data(): accept only cycle or cycle_data"
                                    " as the parameter"));
}
```

Uses cycle_data 23a 26a 42c 43a 54c 54d 55a 56b 56c 56d.

Constructors.

54b ⟨cycle data class 53c⟩+≡ (52a) ◁54a 54c▷

```
cycle_data::cycle_data(const ex & k1, const ex l1, const ex &m1, bool normalize)
{
    k_cd = k1;
    l_cd = l1;
    m_cd = m1;
    if (normalize) {
        ex ratio = 0;
        if (¬k_cd.is_zero()) // First non-zero coefficient among k_cd, m_cd, l_0, l_1, ... is set to 1
            ratio = k_cd;
        else if (¬m_cd.is_zero())
            ratio = m_cd;
        else {
            for (unsigned int i=0; i<get_dim(); i++)
                if (¬l_cd.subs(l_cd.op(1) ≡ i).is_zero()) {
                    ratio = l_cd.subs(l_cd.op(1) ≡ i);
                    break;
                }
        }

        if (¬ratio.is_zero()) {
            k_cd=(k_cd÷ratio).normal();
            l_cd=indexed((l_cd.op(0)÷ratio).evalm().normal(), l_cd.op(1));
            m_cd=(m_cd÷ratio).normal();
        }
    }
}
```

Uses cycle_data 23a 26a 42c 43a 54c 54d 55a 56b 56c 56d, ex 41b 47e 47e 47e 53a, get_dim() 35c, op 50a, and subs 50a.

54c ⟨cycle data class 53c⟩+≡ (52a) ◁54b 54d▷
```
return_type_t cycle_data::return_type_tinfo() const
{
    return make_return_type_t<cycle_data>();
}
```

Uses `cycle_data` 23a 26a 42c 43a 54c 54d 55a 56b 56c 56d.

54d ⟨cycle data class 53c⟩+≡ (52a) ◁54c 55a▷
```
int cycle_data::compare_same_type(const basic &other) const
{
        GINAC_ASSERT(is_a<cycle_data>(other));
        return inherited::compare_same_type(other);
}
```

Defines:
`cycle_data`, used in chunks 26b, 46b, 52–54, 56–59, 63a, 68–71, 75, 81, 83–85, 87c, 89, 95–97, 113c, 117c, and 119a.

Printing the cycle data

55a ⟨cycle data class 53c⟩+≡ (52a) ◁54d 55b▷
```
void cycle_data::do_print(const print_dflt & con, unsigned level) const
{
    con.s << "‘";
    this->k_cd.print(con, level);
    con.s << ", ";
    this->l_cd.print(con, level);
    con.s << ", ";
    this->m_cd.print(con, level);
    con.s << "’";
}
```

Defines:
`cycle_data`, used in chunks 26b, 46b, 52–54, 56–59, 63a, 68–71, 75, 81, 83–85, 87c, 89, 95–97, 113c, 117c, and 119a.

Printing the cycle data in the float mode if possible.

55b ⟨cycle data class 53c⟩+≡ (52a) ◁55a 55d▷
```
void cycle_data::do_print_double(const print_dflt & con, unsigned level) const
{
    if (¬ is_a<numeric>(get_dim())) {
        do_print(con, level);
    } else {
```

Defines:
`cycle_data`, used in chunks 26b, 46b, 52–54, 56–59, 63a, 68–71, 75, 81, 83–85, 87c, 89, 95–97, 113c, 117c, and 119a.
Uses `do_print_double` 49e, `get_dim()` 35c, and `numeric` 22d.

Check if conversion to double is possible and accurate.

55d ⟨cycle data class 53c⟩+≡ (52a) ◁55b 55e▷
```
        con.s << "(";
        if ((is_a<numeric>(k_cd) ∧ ¬ ex_to<numeric>(k_cd).is_crational())
            ∨ is_a<numeric>(k_cd.evalf())) {
            ex f=k_cd.evalf();
            ⟨common part of float output 55c⟩
```

Uses `evalf` 50a, `ex` 41b 47e 47e 47e 53a, and `numeric` 22d.

Here is the repeating part

55c ⟨common part of float output 55c⟩≡ (55 56)

```
        con.s ≪ ex_to<numeric>(f).to_double(); // only real part is converted
        if (¬ ex_to<numeric>(f).is_real()) {
            double b=ex_to<numeric>(f.imag_part()).to_double();
            if (b>0)
                con.s ≪ "+";
            con.s ≪ b ≪ "*I";
        }
```

Uses numeric 22d.

back to our routine.

55e ⟨cycle data class 53c⟩+≡ (52a) ◁55d 56a▷

```
    } else
        k_cd.print(con, level);
    con.s ≪ ", [[";
```

Run through all elements of the l vector.

56a ⟨cycle data class 53c⟩+≡ (52a) ◁55e 56b▷

```
    int D=ex_to<numeric>(get_dim()).to_int();
    for(int i=0; i< D; ++i) {
        if ((is_a<numeric>(l_cd.op(0).op(i)) ∧ ¬ ex_to<numeric>(l_cd.op(0).op(i)).is_crational())
            ∨ is_a<numeric>(l_cd.op(0).op(i).evalf())) {
            ex f=ex_to<numeric>(l_cd.op(0).op(i)).evalf();
            ⟨common part of float output 55c⟩
        } else
            l_cd.op(0).op(i).print(con, level);
        if (i<D-1)
            con.s ≪ ",";
    }
    con.s ≪ "]]";
    l_cd.op(1).print(con, level);
```

Uses evalf 50a, ex 41b 47e 47e 47e 53a, get_dim() 35c, numeric 22d, and op 50a.

Finishing with the m part.

56b ⟨cycle data class 53c⟩+≡ (52a) ◁56a 56c▷

```
    con.s ≪ ", ";
    if ((is_a<numeric>(m_cd) ∧ ¬ ex_to<numeric>(m_cd).is_crational())
        ∨ is_a<numeric>(m_cd.evalf())) {
        ex f=m_cd.evalf();
        ⟨common part of float output 55c⟩
    } else
        m_cd.print(con, level);
    con.s ≪ ")";
  }
}
```

Uses evalf 50a, ex 41b 47e 47e 47e 53a, and numeric 22d.

56c ⟨cycle data class 53c⟩+≡ (52a) ◁56b 56d▷

```
void cycle_data::archive(archive_node &n) const
{
    inherited::archive(n);
    n.add_ex("k-val", k_cd);
    n.add_ex("l-val", l_cd);
    n.add_ex("m-val", m_cd);
}
```

Defines:
cycle_data, used in chunks 26b, 46b, 52–54, 56–59, 63a, 68–71, 75, 81, 83–85, 87c, 89, 95–97, 113c, 117c, and 119a.
Uses archive 50a, k 51c, l 51c, and m 51c.

56d ⟨cycle data class 53c⟩+≡ (52a) ◁56c 56e▷
```
void cycle_data::read_archive(const archive_node &n, lst &sym_lst)
{
    inherited::read_archive(n, sym_lst);
    n.find_ex("k-val", k_cd, sym_lst);
    n.find_ex("l-val", l_cd, sym_lst);
    n.find_ex("m-val", m_cd, sym_lst);
}
```

Defines:
`cycle_data`, used in chunks 26b, 46b, 52–54, 56–59, 63a, 68–71, 75, 81, 83–85, 87c, 89, 95–97, 113c, 117c, and 119a.
Uses `k` 51c, `l` 51c, `m` 51c, and `read_archive` 50a.

56e ⟨cycle data class 53c⟩+≡ (52a) ◁56d 57a▷
```
GINAC_BIND_UNARCHIVER(cycle_data);
```

Defines:
`cycle_data`, used in chunks 26b, 46b, 52–54, 56–59, 63a, 68–71, 75, 81, 83–85, 87c, 89, 95–97, 113c, 117c, and 119a.

57a ⟨cycle data class 53c⟩+≡ (52a) ◁56e 57b▷
```
ex cycle_data::op(size_t i) const
{
 GINAC_ASSERT(i<nops());
    switch(i) {
    case 0:
        return k_cd;
    case 1:
        return l_cd;
    case 2:
        return m_cd;
    default:
        throw(std::invalid_argument("cycle_data::op(): requested operand out of the range (3)"));
    }
}
```

Uses `cycle_data` 23a 26a 42c 43a 54c 54d 55a 56b 56c 56d, `ex` 41b 47e 47e 47e 53a, `nops` 50a, and `op` 50a.

57b ⟨cycle data class 53c⟩+≡ (52a) ◁57a 57c▷
```
ex & cycle_data::let_op(size_t i)
{
    ensure_if_modifiable();
    GINAC_ASSERT(i<nops());
    switch(i) {
    case 0:
        return k_cd;
    case 1:
        return l_cd;
    case 2:
        return m_cd;
    default:
     throw(std::invalid_argument("cycle_data::let_op(): requested operand out of the range (3)"));
    }
}
```

Uses `cycle_data` 23a 26a 42c 43a 54c 54d 55a 56b 56c 56d, `ex` 41b 47e 47e 47e 53a, and `nops` 50a.

57c ⟨cycle data class 53c⟩+≡ (52a) ◁57b 58a▷
```
ex cycle_data::make_cycle(const ex & metr) const
{
    return cycle(k_cd, l_cd, m_cd, metr);
}
```

Uses `cycle_data` 23a 26a 42c 43a 54c 54d 55a 56b 56c 56d and `ex` 41b 47e 47e 47e 53a.

58a ⟨cycle data class 53c⟩+≡ (52a) ◁57c 58b▷

```
bool cycle_data::is_equal(const basic & other, bool projectively) const
{
    if (not is_a<cycle_data>(other))
        return false;
    const cycle_data o = ex_to<cycle_data>(other);
    ex factor=0, ofactor=0;

    if (projectively) {
        // Check that coefficients are scalar multiples of other
        if (not ((m_cd*o.get_k()-o.get_m()*k_cd).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 `cycle_data` 23a 26a 42c 43a 54c 54d 55a 56b 56c 56d and `ex` 41b 47e 47e 47e 53a.

Now we iterate through the coefficients of *l*.

58b ⟨cycle data class 53c⟩+≡ (52a) ◁58a 59a▷

```
    for (unsigned int i=0; i<get_dim(); 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;
}
```

Uses `get_dim()` 35c.

59a ⟨cycle data class 53c⟩+≡ (52a) ◁58b 59b▷

```
bool cycle_data::is_almost_equal(const basic & other, bool projectively) const
{
    if (not is_a<cycle_data>(other))
        return false;
    const cycle_data o = ex_to<cycle_data>(other);
    ex factor=0, ofactor=0;

    if (projectively) {
        // Check that coefficients are scalar multiples of other
        if (¬ (is_less_than_epsilon(m_cd*o.get_k()-o.get_m()*k_cd)))
            return false;
        // Set up coefficients for proportionality
        if (is_less_than_epsilon(get_k())) {
            factor=get_m();
            ofactor=o.get_m();
        } else {
            factor=get_k();
            ofactor=o.get_k();
        }

    } else
        // Check the exact equality of coefficients
        if (¬ (is_less_than_epsilon((get_k()-o.get_k()))
              ∧ is_less_than_epsilon(get_m()-o.get_m())))
            return false;
```

Uses `cycle_data` 23a 26a 42c 43a 54c 54d 55a 56b 56c 56d, `ex` 41b 47e 47e 47e 53a, `is_almost_equal` 117a, and `is_less_than_epsilon` 53b.

Now we iterate through the coefficients of l.

59b ⟨cycle data class 53c⟩+≡ (52a) ◁59a 59c▷

```
    for (unsigned int i=0; i<get_dim(); 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 (¬ is_less_than_epsilon(get_l(i)*ofactor-o.get_l(i)*factor))
                    return false;
        } else
            if (¬ is_less_than_epsilon(get_l(i)-o.get_l(i)))
                return false;

    return true;
}
```

Uses `get_dim()` 35c and `is_less_than_epsilon` 53b.

59c ⟨cycle data class 53c⟩+≡ (52a) ◁59b 59d▷

```
cycle_data cycle_data::subs(const ex & e, unsigned options) const
{
    return cycle_data(k_cd.subs(e,options),l_cd.subs(e,options),m_cd.subs(e,options),false);
}
```

Uses `cycle_data` 23a 26a 42c 43a 54c 54d 55a 56b 56c 56d, `ex` 41b 47e 47e 47e 53a, and `subs` 50a.

59d ⟨cycle data class 53c⟩+≡ (52a) ◁59c ??▷

```
ex cycle_data::subs(const exmap & em, unsigned options) const
{
    return cycle_data(k_cd.subs(em,options),l_cd.subs(em,options),m_cd.subs(em,options),false);
}
```

Uses `cycle_data` 23a 26a 42c 43a 54c 54d 55a 56b 56c 56d, `ex` 41b 47e 47e 47e 53a, and `subs` 50a.

?? ⟨cycle data class 53c⟩+≡ (52a) ◁59d

```
ex cycle_data::num_normalize() const
{
    if (¬ (is_a<numeric>(k_cd) ∧ is_a<numeric>(m_cd)
          ∧ is_a<numeric>(l_cd.op(0).op(0))  ∧ is_a<numeric>(l_cd.op(0).op(1))))
        return cycle_data(k_cd,l_cd,m_cd,true);

    numeric k1=ex_to<numeric>(k_cd),
        m1=ex_to<numeric>(m_cd);
    numeric r=max(abs(k1),abs(m1));
    for (unsigned int i=0; i<get_dim(); ++i)
        r=max(r,abs(ex_to<numeric>(l_cd.op(0).op(i))));

    if (is_less_than_epsilon(r))
        return cycle_data(k_cd,l_cd,m_cd,true);
    k1÷=r; k1=(is_less_than_epsilon(k1)?0:k1);
    m1÷=r; m1=(is_less_than_epsilon(m1)?0:m1);
    lst l1;
    for (unsigned int i=0; i<get_dim(); ++i) {
        numeric li= ex_to<numeric>(l_cd.op(0).op(i))÷r;
        l1.append(is_less_than_epsilon(li)?0:li);
    }

    return cycle_data(k1,indexed(matrix(1, get_dim(), l1), l_cd.op(1)),m1);
}
```

Uses `cycle_data` 23a 26a 42c 43a 54c 54d 55a 56b 56c 56d, `ex` 41b 47e 47e 47e 53a, `get_dim()` 35c, `is_less_than_epsilon` 53b, `numeric` 22d, and `op` 50a.

F.2. Implementation of cycle_relation class.

60a ⟨cycle relation class 60a⟩≡ (52a) 60b▷

```
cycle_relation::cycle_relation() : parkey(), parameter()
{
  rel = cycle_orthogonal;
  use_cycle_metric = true;
}
```

Uses `cycle_orthogonal` 34b 113a and `cycle_relation` 40c 45e 46c 60d 61 62a 62b 64a 64b.

60b ⟨cycle relation class 60a⟩+≡ (52a) ◁60a 60c▷

```
cycle_relation::cycle_relation(const ex & ck, PCR r, bool cm, const ex & p) {
    parkey = ck;
    rel = r;
    use_cycle_metric = cm;
    parameter=p;
}
```

Uses `cycle_relation` 40c 45e 46c 60d 61 62a 62b 64a 64b, `ex` 41b 47e 47e 47e 53a, and `PCR` 45d.

60c ⟨cycle relation class 60a⟩+≡ (52a) ◁60b 60d▷

```
return_type_t cycle_relation::return_type_tinfo() const
{
    return make_return_type_t<cycle_relation>();
}
```

Uses `cycle_relation` 40c 45e 46c 60d 61 62a 62b 64a 64b.

60d ⟨cycle relation class 60a⟩+≡ (52a) ◁60c 61▷

```
int cycle_relation::compare_same_type(const basic &other) const
{
       GINAC_ASSERT(is_a<cycle_relation>(other));
       return inherited::compare_same_type(other);
       ÷∗
  const cycle_relation &o = static_cast<const cycle_relation &>(other);
    if ((parkey ≡ o.parkey) ∧ (&rel ≡ &o.rel))
        return 0;
    else if ((parkey < o.parkey) ∨ (&rel < &o.rel))
        return -1;
    else
    return 1;∗÷
}
```

Defines:
`cycle_relation`, used in chunks 38–40, 43b, 45c, 46a, 52e, 60, 63a, 65, 70e, 71e, 73a, 81a, 83, 84a, 95e, 96d, and 116–18.

(un)Archiving of **cycle_relation** is not universal. At present it only can handle relations declared in the header file: *cycle_orthogonal*, *cycle_f_orthogonal*, *cycle_adifferent*, *cycle_different* and *cycle_tangent*.

61 ⟨cycle relation class 60a⟩+≡ (52a) ◁60d 62a▷

```
void cycle_relation::archive(archive_node &n) const
{
    inherited::archive(n);
    n.add_ex("cr-parkey", parkey);
    n.add_bool("use_cycle_metric", use_cycle_metric);
    n.add_ex("parameter", parameter);
    ex (*const* ptr)(const ex &, const ex &, const ex &)
       = rel.target<ex(*)(const ex&, const ex &,const ex&)>();
    if (ptr ∧ *ptr ≡ cycle_orthogonal)
       n.add_string("relation", "orthogonal");
    else if (ptr ∧ *ptr ≡ cycle_f_orthogonal)
       n.add_string("relation", "f_orthogonal");
    else if (ptr ∧ *ptr ≡ cycle_different)
       n.add_string("relation", "different");
    else if (ptr ∧ *ptr ≡ cycle_adifferent)
       n.add_string("relation", "adifferent");
    else if (ptr ∧ *ptr ≡ cycle_tangent)
       n.add_string("relation", "tangent");
    else if (ptr ∧ *ptr ≡ cycle_tangent_i)
       n.add_string("relation", "tangent_i");
    else if (ptr ∧ *ptr ≡ cycle_tangent_o)
       n.add_string("relation", "tangent_o");
    else if (ptr ∧ *ptr ≡ cycle_angle)
       n.add_string("relation", "angle");
    else if (ptr ∧ *ptr ≡ steiner_power)
       n.add_string("relation", "steiner_power");
    else if (ptr ∧ *ptr ≡ cycle_cross_t_distance)
       n.add_string("relation", "cross_distance");
    else if (ptr ∧ *ptr ≡ product_sign)
       n.add_string("relation", "product_sign");
    else if (ptr ∧ *ptr ≡ coefficients_are_real)
       n.add_string("relation", "are_real");
    else if (ptr ∧ *ptr ≡ cycle_moebius)
       n.add_string("relation", "moebius");
    else if (ptr ∧ *ptr ≡ cycle_sl2)
       n.add_string("relation", "sl2");
    else
      throw(std::invalid_argument("cycle_relation::archive(): archiving of this relation is not"
                                  " implemented"));
}
```

Defines:
`cycle_relation`, used in chunks 38–40, 43b, 45c, 46a, 52e, 60, 63a, 65, 70e, 71e, 73a, 81a, 83, 84a, 95e, 96d, and 116–18.

Uses `archive` 50a, `coefficients_are_real` 34h 115c, `cycle_adifferent` 34f 113c, `cycle_angle` 46e 114e, `cycle_cross_t_distance` 46e 115b, `cycle_different` 34e 114c, `cycle_f_orthogonal` 34c 113b, `cycle_moebius` 47a 116b, `cycle_orthogonal` 34b 113a, `cycle_sl2` 47a 116d, `cycle_tangent` 46e 113e, `cycle_tangent_i` 46e 114b, `cycle_tangent_o` 46e 114a, `ex` 41b 47e 47e 47e 53a, `product_sign` 34g 114d, and `steiner_power` 46e 115a.

62a ⟨cycle relation class 60a⟩+≡ (52a) ◁61 62b▷

```
void cycle_relation::read_archive(const archive_node &n, lst &sym_lst)
{
   ex e;
   inherited::read_archive(n, sym_lst);
   n.find_ex("cr-parkey", e, sym_lst);
   if (is_a<symbol>(e))
      parkey=e;
   else
      throw(std::invalid_argument("cycle_relation::read_archive(): read a non-symbol as"
                                  " a parkey from the archive"));
   n.find_ex("parameter", parameter, sym_lst);
   n.find_bool("use_cycle_metric", use_cycle_metric);
   string relation;
   n.find_string("relation", relation);
   if (relation ≡ "orthogonal")
      rel = cycle_orthogonal;
   else if (relation ≡ "f_orthogonal")
      rel = cycle_f_orthogonal;
   else if (relation ≡ "different")
      rel = cycle_different;
   else if (relation ≡ "adifferent")
      rel = cycle_adifferent;
   else if (relation ≡ "tangent")
      rel = cycle_tangent;
   else if (relation ≡ "tangent_i")
      rel = cycle_tangent_i;
   else if (relation ≡ "tangent_o")
      rel = cycle_tangent_o;
   else if (relation ≡ "angle")
      rel = cycle_angle;
   else if (relation ≡ "steiner_power")
      rel = steiner_power;
   else if (relation ≡ "cross_distance")
      rel = cycle_cross_t_distance;
   else if (relation ≡ "product_sign")
      rel = product_sign;
   else if (relation ≡ "are_real")
      rel = coefficients_are_real;
   else if (relation ≡ "moebius")
      rel = cycle_moebius;
   else if (relation ≡ "sl2")
      rel = cycle_sl2;
   else
      throw(std::invalid_argument("cycle_relation::read_archive(): archive contains unknown"
                                  " relation"));
}
```

Defines:
cycle_relation, used in chunks 38–40, 43b, 45c, 46a, 52e, 60, 63a, 65, 70e, 71e, 73a, 81a, 83, 84a, 95e, 96d, and 116–18.
Uses archive 50a, coefficients_are_real 34h 115c, cycle_adifferent 34f 113c, cycle_angle 46e 114e, cycle_cross_t_distance 46e 115b, cycle_different 34e 114c, cycle_f_orthogonal 34c 113b, cycle_moebius 47a 116b, cycle_orthogonal 34b 113a, cycle_sl2 47a 116d, cycle_tangent 46e 113e, cycle_tangent_i 46e 114b, cycle_tangent_o 46e 114a, ex 41b 47e 47e 47e 53a, product_sign 34g 114d, read_archive 50a, and steiner_power 46e 115a.

62b ⟨cycle relation class 60a⟩+≡ (52a) ◁62a 63a▷

```
GINAC_BIND_UNARCHIVER(cycle_relation);
```

Defines:
cycle_relation, used in chunks 38–40, 43b, 45c, 46a, 52e, 60, 63a, 65, 70e, 71e, 73a, 81a, 83, 84a, 95e, 96d, and 116–18.

63a ⟨cycle relation class 60a⟩+≡ (52a) ◁62b 63b▷

```
ex cycle_relation::rel_to_parent(const ex & C1, const ex & pmetric, const ex & cmetric,
                                 const exhashmap<cycle_node> & N) const
{
    GINAC_ASSERT(is_a<cycle_data>(C1));
```

Uses `cycle_data` 23a 26a 42c 43a 54c 54d 55a 56b 56c 56d, `cycle_node` 43c 45c 69c 70a 70b 70c 71a 71b 72b 72c 73a 74a 74d, `cycle_relation` 40c 45e 46c 60d 61 62a 62b 64a 64b, and `ex` 41b 47e 47e 47e 53a.

First we check if the required key exists in the cycles list. If there is no such key, we return the relation to the calling cycle itself.

63b ⟨cycle relation class 60a⟩+≡ (52a) ◁63a 63c▷

```
    exhashmap<cycle_node>::const_iterator cnode=N.find(parkey);
```

Uses `cycle_node` 43c 45c 69c 70a 70b 70c 71a 71b 72b 72c 73a 74a 74d.

Otherwise the list of equations is constructed for the found key.

63c ⟨cycle relation class 60a⟩+≡ (52a) ◁63b 64a▷

```
    lst res,
        cycles=ex_to<lst>(cnode->second.make_cycles(use_cycle_metric? cmetric : pmetric));

    for (const auto& it : cycles) {
        lst calc=ex_to<lst>(rel(ex_to<cycle_data>(C1).make_cycle(use_cycle_metric? cmetric : pmetric),
                              ex_to<cycle>(it), parameter));
        for (const auto& it1 : calc)
            res.append(it1);
    }
    return res;
}
```

Uses `cycle_data` 23a 26a 42c 43a 54c 54d 55a 56b 56c 56d.

64a ⟨cycle relation class 60a⟩+≡ (52a) ◁63c 64b▷

```
void cycle_relation::do_print(const print_dflt & con, unsigned level) const
{
    con.s << parkey << (use_cycle_metric? "|" : "/");
    ex (*const* ptr)(const ex &, const ex &, const ex &)
      = rel.target<ex(*)(const ex&, const ex &, const ex &)>();
    if (ptr && *ptr == cycle_orthogonal)
      con.s << "o";
    else if (ptr && *ptr == cycle_f_orthogonal)
      con.s << "f";
    else if (ptr && *ptr == cycle_different)
      con.s << "d";
    else if (ptr && *ptr == cycle_adifferent)
      con.s << "ad";
    else if (ptr && *ptr == cycle_tangent)
      con.s << "t";
    else if (ptr && *ptr == cycle_tangent_i)
      con.s << "ti";
    else if (ptr && *ptr == cycle_tangent_o)
      con.s << "to";
    else if (ptr && *ptr == steiner_power)
      con.s << "s";
    else if (ptr && *ptr == cycle_angle)
      con.s << "a";
    else if (ptr && *ptr == cycle_cross_t_distance)
      con.s << "c";
    else if (ptr && *ptr == product_sign)
      con.s << "p";
    else if (ptr && *ptr == coefficients_are_real)
      con.s << "r";
    else if (ptr && *ptr == cycle_moebius)
      con.s << "m";
    else if (ptr && *ptr == cycle_sl2)
      con.s << "l";
    else
      con.s << "u"; // unknown type of relations
    if (! parameter.is_zero())
      con.s << "[" << parameter << "]";
}
```

Defines:
  cycle_relation, used in chunks 38–40, 43b, 45c, 46a, 52e, 60, 63a, 65, 70e, 71e, 73a, 81a, 83, 84a, 95e, 96d, and 116–18.
Uses coefficients_are_real 34h 115c, cycle_adifferent 34f 113c, cycle_angle 46e 114e, cycle_cross_t_distance 46e 115b, cycle_different 34e 114c, cycle_f_orthogonal 34c 113b, cycle_moebius 47a 116b, cycle_orthogonal 34b 113a, cycle_sl2 47a 116d, cycle_tangent 46e 113e, cycle_tangent_i 46e 114b, cycle_tangent_o 46e 114a, ex 41b 47e 47e 47e 53a, l 51c, m 51c, product_sign 34g 114d, and steiner_power 46e 115a.

64b ⟨cycle relation class 60a⟩+≡ (52a) ◁64a 65a▷

```
void cycle_relation::do_print_tree(const print_tree & con, unsigned level) const
{
    // inherited::do_print_tree(con,level);
    parkey.print(con,level+con.delta_indent);
    // con.s << std::string(level+con.delta_indent, ' ') << (int)rel <<endl;
}
```

Defines:
  cycle_relation, used in chunks 38–40, 43b, 45c, 46a, 52e, 60, 63a, 65, 70e, 71e, 73a, 81a, 83, 84a, 95e, 96d, and 116–18.

65a ⟨cycle relation class 60a⟩+≡ (52a) ◁64b 65b▷
```
ex cycle_relation::op(size_t i) const
{
 GINAC_ASSERT(i<nops());
   switch(i) {
   case 0:
      return parkey;
   case 1:
      return parameter;
   default:
    throw(std::invalid_argument("cycle_relation::op(): requested operand out of the range (1)"));
   }
}
```

Uses `cycle_relation` 40c 45e 46c 60d 61 62a 62b 64a 64b, `ex` 41b 47e 47e 47e 53a, `nops` 50a, and `op` 50a.

65b ⟨cycle relation class 60a⟩+≡ (52a) ◁65a
```
ex & cycle_relation::let_op(size_t i)
{
   ensure_if_modifiable();
   GINAC_ASSERT(i<nops());
   switch(i) {
   case 0:
      return parkey;
   case 1:
      return parameter;
   default:
    throw(std::invalid_argument("cycle_relation::let_op(): requested operand out of the range (1)"));
   }
}
```

Uses `cycle_relation` 40c 45e 46c 60d 61 62a 62b 64a 64b, `ex` 41b 47e 47e 47e 53a, and `nops` 50a.

F.3. Implementation of subfigure class.

65c ⟨subfigure class 65c⟩≡ (52a) 65d▷
```
subfigure::subfigure() : inherited()
{
}
```

Uses `subfigure` 40d 48b 48d 66a 66b 66c 66d 66e.

65d ⟨subfigure class 65c⟩+≡ (52a) ◁65c 65e▷
```
subfigure::subfigure(const ex & F, const ex & l) {
   parlist = ex_to<lst>(l);
   subf = F;
}
```

Uses `ex` 41b 47e 47e 47e 53a, `l` 51c, and `subfigure` 40d 48b 48d 66a 66b 66c 66d 66e.

65e ⟨subfigure class 65c⟩+≡ (52a) ◁65d 66a▷
```
return_type_t subfigure::return_type_tinfo() const
{
   return make_return_type_t<subfigure>();
}
```

Uses `subfigure` 40d 48b 48d 66a 66b 66c 66d 66e.

66a ⟨subfigure class 65c⟩+≡ (52a) ◁65e 66b▷
```
int subfigure::compare_same_type(const basic &other) const
{
    GINAC_ASSERT(is_a<subfigure>(other));
    return inherited::compare_same_type(other);
}
```

Defines:
`subfigure`, used in chunks 43d, 52e, 65, 67a, 70e, 71e, 84b, and 97.

(un)Archiving of **subfigure** is not universal. At present it only can handle relations declared in the header file: *cycle_orthogonal* and *cycle_f_orthogonal*.

66b ⟨subfigure class 65c⟩+≡ (52a) ◁66a 66c▷
```
void subfigure::archive(archive_node &n) const
{
    inherited::archive(n);
    n.add_ex("parlist", ex_to<lst>(parlist));
    n.add_ex("subf", ex_to<figure>(subf));
}
```

Defines:
`subfigure`, used in chunks 43d, 52e, 65, 67a, 70e, 71e, 84b, and 97.
Uses `archive` 50a and `figure` 16d 22e 32a 32c 38b 49a 50d 75a 80a 82b 82c 85a 86c 98c 99b 99d 100a 101a 103b 104a 105c 106c 106d 107a 109a 109c 110a.

66c ⟨subfigure class 65c⟩+≡ (52a) ◁66b 66d▷
```
void subfigure::read_archive(const archive_node &n, lst &sym_lst)
{
    ex e;
    inherited::read_archive(n, sym_lst);
    n.find_ex("parlist", e, sym_lst);
    if (is_a<lst>(e))
        parlist=ex_to<lst>(e);
    else
        throw(std::invalid_argument("subfigure::read_archive(): read a non-lst as a parlist from"
                                    " the archive"));
    n.find_ex("subf", e, sym_lst);
    if (is_a<figure>(e))
        subf=ex_to<figure>(e);
    else
        throw(std::invalid_argument("subfigure::read_archive(): read a non-figure as a subf from"
                                    " the archive"));
}
```

Defines:
`subfigure`, used in chunks 43d, 52e, 65, 67a, 70e, 71e, 84b, and 97.
Uses `archive` 50a, `ex` 41b 47e 47e 47e 53a, `figure` 16d 22e 32a 32c 38b 49a 50d 75a 80a 82b 82c 85a 86c 98c 99b 99d 100a 101a 103b 104a 105c 106c 106d 107a 109a 109c 110a, and `read_archive` 50a.

66d ⟨subfigure class 65c⟩+≡ (52a) ◁66c 66e▷
```
GINAC_BIND_UNARCHIVER(subfigure);
```

Defines:
`subfigure`, used in chunks 43d, 52e, 65, 67a, 70e, 71e, 84b, and 97.

66e ⟨subfigure class 65c⟩+≡ (52a) ◁66d 67a▷
```
void subfigure::do_print(const print_dflt & con, unsigned level) const
{
    con.s << "subfig( " ;
        parlist.print(con, level+1);
        //      subf.print(con, level+1);
    con.s << ")" ;
}
```

Defines:
`subfigure`, used in chunks 43d, 52e, 65, 67a, 70e, 71e, 84b, and 97.

67a ⟨subfigure class 65c⟩+≡ (52a) ◁66e

```
inline ex subfigure::subs(const exmap & em, unsigned options) const {
    return subfigure(subf.subs(em,options | do_not_update_subfigure), parlist);
}
```

Uses `do_not_update_subfigure` 52b, `ex` 41b 47e 47e 47e 53a, `subfigure` 40d 48b 48d 66a 66b 66c 66d 66e, and `subs` 50a.

F.4. **Implementation of cycle_node class.** Default constructor.

67b ⟨cycle node class 67b⟩≡ (52a) 67c ▷

```
cycle_node::cycle_node()
{
    generation=0;
    custom_asy="";
}
```

Uses `cycle_node` 43c 45c 69c 70a 70b 70c 71a 71b 72b 72c 73a 74a 74d.

Create a **cycle_node** out of **cycle** or **cycle_node**.

67c ⟨cycle node class 67b⟩+≡ (52a) ◁67b 67e ▷

```
cycle_node::cycle_node(const ex & C, int g)
{
    custom_asy="";
    generation=g;
    ⟨set cycle data to the node 67d⟩
}
```

Uses `cycle_node` 43c 45c 69c 70a 70b 70c 71a 71b 72b 72c 73a 74a 74d and `ex` 41b 47e 47e 47e 53a.

We use this check to initialise or change cycle info of the node.

67d ⟨set cycle data to the node 67d⟩≡ (67c)

```
    if (is_a<cycle_node>(C)) {
        cycles=ex_to<lst>(ex_to<cycle_node>(C).get_cycles_data());
        generation = ex_to<cycle_node>(C).get_generation();
        children = ex_to<cycle_node>(C).get_children();
        parents = ex_to<cycle_node>(C).get_parents();
    } else
        ⟨check cycles are valid 68a⟩
```

Uses `cycle_node` 43c 45c 69c 70a 70b 70c 71a 71b 72b 72c 73a 74a 74d and `get_generation` 35f.

67e ⟨cycle node class 67b⟩+≡ (52a) ◁67c 68c ▷

```
cycle_node::cycle_node(const ex & C, int g, const lst & par)
{
    custom_asy="";
    generation=g;
    ⟨check cycles are valid 68a⟩
    ⟨check parents are valid 68b⟩
}
```

Uses `cycle_node` 43c 45c 69c 70a 70b 70c 71a 71b 72b 72c 73a 74a 74d and `ex` 41b 47e 47e 47e 53a.

68a ⟨check cycles are valid 68a⟩≡ (67 68 70a)
```
if (is_a<lst>(C)) {
   for (const auto& it : ex_to<lst>(C))
      if ( is_a<cycle_data>(it) ∨ is_a<cycle>(it))
         cycles.append(cycle_data(it));
      else
         throw(std::invalid_argument("cycle_node::cycle_node(): "
                                     "the parameter is list of something which is not"
                                     " cycle_data"));
} else if (is_a<cycle_data>(C)) {
   cycles = lst{C};
} else if (is_a<cycle>(C)) {
   cycles=lst{cycle_data(ex_to<cycle>(C).get_k(), ex_to<cycle>(C).get_l(),
                  ex_to<cycle>(C).get_m())};
} else
   throw(std::invalid_argument("cycle_node::cycle_node(): "
                               "the first parameters must be either cycle, cycle_data,"
                               " cycle_node or list of cycle_data"));
```

Uses `cycle_data` 23a 26a 42c 43a 54c 54d 55a 56b 56c 56d and `cycle_node` 43c 45c 69c 70a 70b 70c 71a 71b 72b 72c 73a 74a 74d.

68b ⟨check parents are valid 68b⟩≡ (67 68)
```
GINAC_ASSERT(is_a<lst>(par));
parents = ex_to<lst>(par);
```

68c ⟨cycle node class 67b⟩+≡ (52a) ◁67e 68d▷
```
cycle_node::cycle_node(const ex & C, int g, const lst & par, const lst & ch)
{
   generation=g;
   children=ch;
   custom_asy="";
   ⟨check cycles are valid 68a⟩
   ⟨check parents are valid 68b⟩
}
```

Uses `cycle_node` 43c 45c 69c 70a 70b 70c 71a 71b 72b 72c 73a 74a 74d and `ex` 41b 47e 47e 47e 53a.

68d ⟨cycle node class 67b⟩+≡ (52a) ◁68c 68e▷
```
cycle_node::cycle_node(const ex & C, int g, const lst & par, const lst & ch, string ca)
{
   generation=g;
   children=ch;
   custom_asy=ca;
   ⟨check cycles are valid 68a⟩
   ⟨check parents are valid 68b⟩
}
```

Uses `cycle_node` 43c 45c 69c 70a 70b 70c 71a 71b 72b 72c 73a 74a 74d and `ex` 41b 47e 47e 47e 53a.

68e ⟨cycle node class 67b⟩+≡ (52a) ◁68d 69a▷
```
return_type_t cycle_node::return_type_tinfo() const
{
   return make_return_type_t<cycle_node>();
}
```

Uses `cycle_node` 43c 45c 69c 70a 70b 70c 71a 71b 72b 72c 73a 74a 74d.

69a ⟨cycle node class 67b⟩+≡ (52a) ◁68e 69b▷
```
ex cycle_node::op(size_t i) const
{
    GINAC_ASSERT(i<nops());
    size_t ncyc=cycles.nops(), nchil=children.nops(), npar=parents.nops();
    if ( i < ncyc)
        return cycles.op(i);
    else if ( i < ncyc + nchil)
        return children.op(i-ncyc);
    else if ( i < ncyc + nchil + npar)
        return parents.op(i-ncyc-nchil);
    else
        throw(std::invalid_argument("cycle_node::op(): requested operand out of the range"));
}
```

Uses cycle_node 43c 45c 69c 70a 70b 70c 71a 71b 72b 72c 73a 74a 74d, ex 41b 47e 47e 47e 53a, nops 50a, and op 50a.

69b ⟨cycle node class 67b⟩+≡ (52a) ◁69a 69c▷
```
ex & cycle_node::let_op(size_t i)
{
    ensure_if_modifiable();
    GINAC_ASSERT(i<nops());
    size_t ncyc=cycles.nops(), nchil=children.nops(), npar=parents.nops();
    if ( i < ncyc)
        return cycles.let_op(i);
    else if ( i < ncyc + nchil)
        return children.let_op(i-ncyc);
    else if ( i < ncyc + nchil + npar)
        return parents.let_op(i-ncyc-nchil);
    else
        throw(std::invalid_argument("cycle_node::let_op(): requested operand out of the range"));
}
```

Uses cycle_node 43c 45c 69c 70a 70b 70c 71a 71b 72b 72c 73a 74a 74d, ex 41b 47e 47e 47e 53a, and nops 50a.

69c ⟨cycle node class 67b⟩+≡ (52a) ◁69b 69d▷
```
int cycle_node::compare_same_type(const basic &other) const
{
    GINAC_ASSERT(is_a<cycle_node>(other));
    return inherited::compare_same_type(other);
}
```

Defines:
cycle_node, used in chunks 32c, 35f, 37d, 44–46, 48, 49f, 51a, 52e, 63, 67–70, 72, 75, 76c, 79a, 81, 83c, 84b, 98, 99c, and 107–110.

If neither of parameters has multiply values we return a cycle.

69d ⟨cycle node class 67b⟩+≡ (52a) ◁69c 70a▷
```
ex cycle_node::make_cycles(const ex & metr) const
{
    lst res;
    for (const auto& it : cycles)
        res.append(ex_to<cycle_data>(it).make_cycle(metr));
    return res;
}
```

Uses cycle_data 23a 26a 42c 43a 54c 54d 55a 56b 56c 56d, cycle_node 43c 45c 69c 70a 70b 70c 71a 71b 72b 72c 73a 74a 74d, and ex 41b 47e 47e 47e 53a.

70a ⟨cycle node class 67b⟩+≡ (52a) ◁69d 70b▷

```
void cycle_node::set_cycles(const ex & C)
{
    cycles.remove_all();
    ⟨check cycles are valid 68a⟩
}
```

Defines:
cycle_node, used in chunks 32c, 35f, 37d, 44–46, 48, 49f, 51a, 52e, 63, 67–70, 72, 75, 76c, 79a, 81, 83c, 84b, 98, 99c, and 107–110.
Uses ex 41b 47e 47e 47e 53a.

70b ⟨cycle node class 67b⟩+≡ (52a) ◁70a 70c▷

```
void cycle_node::append_cycle(const ex & k, const ex & l, const ex & m)
{
    cycles.append(cycle_data(k,l,m));
}
```

Defines:
cycle_node, used in chunks 32c, 35f, 37d, 44–46, 48, 49f, 51a, 52e, 63, 67–70, 72, 75, 76c, 79a, 81, 83c, 84b, 98, 99c, and 107–110.
Uses cycle_data 23a 26a 42c 43a 54c 54d 55a 56b 56c 56d, ex 41b 47e 47e 47e 53a, k 51c, l 51c, and m 51c.

70c ⟨cycle node class 67b⟩+≡ (52a) ◁70b 70d▷

```
void cycle_node::append_cycle(const ex & C)
{
    if (is_a<cycle>(C))
        cycles.append(cycle_data(ex_to<cycle>(C).get_k(), ex_to<cycle>(C).get_l(),
                                 ex_to<cycle>(C).get_m()));
    else    if (is_a<cycle_data>(C))
        cycles.append(ex_to<cycle_data>(C));
    else
        throw(std::invalid_argument("cycle_node::append_cycle(const ex &): the parameter must be"
                                    " either cycle or cycle_data"));
}
```

Defines:
cycle_node, used in chunks 32c, 35f, 37d, 44–46, 48, 49f, 51a, 52e, 63, 67–70, 72, 75, 76c, 79a, 81, 83c, 84b, 98, 99c, and 107–110.
Uses cycle_data 23a 26a 42c 43a 54c 54d 55a 56b 56c 56d and ex 41b 47e 47e 47e 53a.

Return the list of parents—either *cycle_relations* or **subfigure**

70d ⟨cycle node class 67b⟩+≡ (52a) ◁70c 70e▷

```
lst cycle_node::get_parents() const
{
    return parents;
}
```

Uses cycle_node 43c 45c 69c 70a 70b 70c 71a 71b 72b 72c 73a 74a 74d.

The method returns the list of all keys to parant cycles.

70e ⟨cycle node class 67b⟩+≡ (52a) ◁70d 71a▷

```
lst cycle_node::get_parent_keys() const
{
    lst pkeys;
    if ( (parents.nops() ≡ 1) ∧ (is_a<subfigure>(parents.op(0)))) {
        pkeys=ex_to<lst>(ex_to<subfigure>(parents.op(0)).get_parlist());
    } else {
        for (const auto& it : parents)
            pkeys.append(ex_to<cycle_relation>(it).get_parkey());
    }
    return pkeys;
}
```

Uses cycle_node 43c 45c 69c 70a 70b 70c 71a 71b 72b 72c 73a 74a 74d, cycle_relation 40c 45e 46c 60d 61 62a 62b 64a 64b, nops 50a, op 50a, and subfigure 40d 48b 48d 66a 66b 66c 66d 66e.

Printing of a **cycle_node** has two almost identical form: accurate and float.

71a ⟨cycle node class 67b⟩+≡ (52a) ◁70e 71b▷
```
void cycle_node::do_print(const print_dflt & con, unsigned level) const
{
    ⟨start to print cycle node 71c⟩
        ex_to<cycle_data>(it).do_print(con, level);
        ⟨end to print cycle node 71d⟩
}
```

Defines:
`cycle_node`, used in chunks 32c, 35f, 37d, 44–46, 48, 49f, 51a, 52e, 63, 67–70, 72, 75, 76c, 79a, 81, 83c, 84b, 98, 99c, and 107–110.
Uses `cycle_data` 23a 26a 42c 43a 54c 54d 55a 56b 56c 56d.

And a similar one for the float printing

71b ⟨cycle node class 67b⟩+≡ (52a) ◁71a 72b▷
```
void cycle_node::do_print_double(const print_dflt & con, unsigned level) const
{
    ⟨start to print cycle node 71c⟩
        ex_to<cycle_data>(it).do_print_double(con, level);
        ⟨end to print cycle node 71d⟩
}
```

Defines:
`cycle_node`, used in chunks 32c, 35f, 37d, 44–46, 48, 49f, 51a, 52e, 63, 67–70, 72, 75, 76c, 79a, 81, 83c, 84b, 98, 99c, and 107–110.
Uses `cycle_data` 23a 26a 42c 43a 54c 54d 55a 56b 56c 56d and `do_print_double` 49e.

We output generation and all children, . . .

71c ⟨start to print cycle node 71c⟩≡ (71)
```
    con.s << '{';
    for (const auto& it : cycles) {
```

71d ⟨end to print cycle node 71d⟩≡ (71) 71e▷
```
        con.s << ", ";
    }
    con.s << generation << '}' << " --> (";
    // list all children
    for (lst::const_iterator it = children.begin(); it ≠ children.end();) {
        con.s << (*it);
        ++it;
        if (it ≠ children.end())
            con.s <<",";
    }
```

. . . then all parents.

71e ⟨end to print cycle node 71d⟩+≡ (71) ◁71d 72a▷
```
    con.s << ");  <-- (";
    if (generation > 0 ∨ FIGURE_DEBUG)
        for (lst::const_iterator it = parents.begin(); it ≠ parents.end();) {
            if (is_a<cycle_relation>(*it))
                ex_to<cycle_relation>(*it).do_print(con,level);
            else if (is_a<subfigure>(*it))
                ex_to<subfigure>(*it).do_print(con,level);
            ++it;
            if (it ≠ parents.end())
                con.s <<",";
        }
    con.s << ")";
```

Uses `cycle_relation` 40c 45e 46c 60d 61 62a 62b 64a 64b, `FIGURE_DEBUG` 52c, and `subfigure` 40d 48b 48d 66a 66b 66c 66d 66e.

Finally if the custom `Asymptote` style is not empty we print it as well.

72a ⟨end to print cycle node 71d⟩+≡ (71) ◁71e

```
if (custom_asy ≠ "")
    con.s≪ " /" ≪ custom_asy ≪ "/";
con.s ≪ endl;
```

72b ⟨cycle node class 67b⟩+≡ (52a) ◁71b 72c▷

```
void cycle_node::do_print_tree(const print_tree & con, unsigned level) const
{
    for (const auto& it : cycles)
        it.print(con, level);
    con.s ≪ std::string(level+con.delta_indent, ' ') ≪ "generation: "≪ generation ≪ endl;
    con.s ≪ std::string(level+con.delta_indent, ' ') ≪ "children" ≪endl;
    children.print(con,level+2*con.delta_indent);
    con.s ≪ std::string(level+con.delta_indent, ' ') ≪ "parents" ≪endl;
    parents.print(con,level+2*con.delta_indent);
    con.s ≪ std::string(level+con.delta_indent, ' ') ≪ "custom_asy: " ≪ custom_asy ≪endl;
}
```

Defines:
`cycle_node`, used in chunks 32c, 35f, 37d, 44–46, 48, 49f, 51a, 52e, 63, 67–70, 72, 75, 76c, 79a, 81, 83c, 84b, 98, 99c, and 107–110.

72c ⟨cycle node class 67b⟩+≡ (52a) ◁72b 72d▷

```
void cycle_node::remove_child(const ex & other)
{
    lst nchildren;
    for (const auto& it : children)
        if (it ≠ other)
            nchildren.append(it);
    children=nchildren;
}
```

Defines:
`cycle_node`, used in chunks 32c, 35f, 37d, 44–46, 48, 49f, 51a, 52e, 63, 67–70, 72, 75, 76c, 79a, 81, 83c, 84b, 98, 99c, and 107–110.
Uses `ex` 41b 47e 47e 47e 53a.

72d ⟨cycle node class 67b⟩+≡ (52a) ◁72c 72e▷

```
cycle_node cycle_node::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 ex_to<cycle_node>(subs(em, options));
}
```

Uses `cycle_node` 43c 45c 69c 70a 70b 70c 71a 71b 72b 72c 73a 74a 74d, `ex` 41b 47e 47e 47e 53a, `info` 50a, `l` 51c, `op` 50a, and `subs` 50a.

72e ⟨cycle node class 67b⟩+≡ (52a) ◁72d 73a▷

```
ex cycle_node::subs(const exmap & em, unsigned options) const
{
     return cycle_node(cycles.subs(em, options), generation, ex_to<lst>(parents.subs(em, options)), children, custom_asy);
}
```

Uses `cycle_node` 43c 45c 69c 70a 70b 70c 71a 71b 72b 72c 73a 74a 74d, `ex` 41b 47e 47e 47e 53a, and `subs` 50a.

73a ⟨cycle node class 67b⟩+≡ (52a) ◁72e 73b▷
```
void cycle_node::archive(archive_node &n) const
{
    inherited::archive(n);
    n.add_ex("cycles", cycles);
    n.add_unsigned("children_size", children.nops());
    if (children.nops()>0)
        for (const auto& it : children)
            n.add_ex("chil", it);

    n.add_unsigned("parent_size", parents.nops());
    if (parents.nops()>0) {
        n.add_bool("has_subfigure", false);
        for (const auto& it : parents)
            n.add_ex("par", ex_to<cycle_relation>(it));
    }
```

Defines:
cycle_node, used in chunks 32c, 35f, 37d, 44–46, 48, 49f, 51a, 52e, 63, 67–70, 72, 75, 76c, 79a, 81, 83c, 84b, 98, 99c, and 107–110.
Uses archive 50a, cycle_relation 40c 45e 46c 60d 61 62a 62b 64a 64b, and nops 50a.

storing the generation with its sign.

73b ⟨cycle node class 67b⟩+≡ (52a) ◁73a 73c▷
```
    bool neg_generation=(generation<0);
    n.add_bool("neg_generation", neg_generation);
    if (neg_generation)
        n.add_unsigned("abs_generation", -generation);
    else
        n.add_unsigned("abs_generation", generation);
```

saving the asymptote options

73c ⟨cycle node class 67b⟩+≡ (52a) ◁73b 74a▷
```
    n.add_string("custom_asy", custom_asy);
}
```

74a ⟨cycle node class 67b⟩+≡ (52a) ◁73c 74b▷
```
void cycle_node::read_archive(const archive_node &n, lst &sym_lst)
{
    inherited::read_archive(n, sym_lst);
    ex e;
    n.find_ex("cycles", e, sym_lst);
    cycles=ex_to<lst>(e);
    ex ch, par;
    unsigned int c_size;
    n.find_unsigned("children_size", c_size);

    if (c_size>0) {
        archive_node::archive_node_cit first = n.find_first("chil");
        archive_node::archive_node_cit last = n.find_last("chil");
        ++last;
        for (archive_node::archive_node_cit i=first; i != last; ++i) {
            ex e;
            n.find_ex_by_loc(i, e, sym_lst);
            children.append(e);
        }
    }

    unsigned int p_size;
    n.find_unsigned("parent_size", p_size);

    if (p_size>0) {
        archive_node::archive_node_cit first = n.find_first("par");
        archive_node::archive_node_cit last = n.find_last("par");
        ++last;
        for (archive_node::archive_node_cit i=first; i != last; ++i) {
            ex e;
            n.find_ex_by_loc(i, e, sym_lst);
            parents.append(e);
        }
    }
```

Defines:
`cycle_node`, used in chunks 32c, 35f, 37d, 44–46, 48, 49f, 51a, 52e, 63, 67–70, 72, 75, 76c, 79a, 81, 83c, 84b, 98, 99c, and 107–110.
Uses `ex` 41b 47e 47e 47e 53a and `read_archive` 50a.

restoring the generation with its sign

74b ⟨cycle node class 67b⟩+≡ (52a) ◁74a 74c▷
```
    bool neg_generation;
    n.find_bool("neg_generation", neg_generation);
    unsigned int abs_generation;
    n.find_unsigned("abs_generation", abs_generation);
    if (neg_generation)
        generation = -abs_generation;
    else
        generation = abs_generation;
```

restoring the asymptote options

74c ⟨cycle node class 67b⟩+≡ (52a) ◁74b 74d▷
```
    n.find_string("custom_asy", custom_asy);
}
```

74d ⟨cycle node class 67b⟩+≡ (52a) ◁74c
```
GINAC_BIND_UNARCHIVER(cycle_node);
```

Defines:
`cycle_node`, used in chunks 32c, 35f, 37d, 44–46, 48, 49f, 51a, 52e, 63, 67–70, 72, 75, 76c, 79a, 81, 83c, 84b, 98, 99c, and 107–110.

F.5. **Implementation of figure class.** Since this is the main class of the library, its implementation is most evolved.

F.5.1. **figure** *conctructors.* We create a figure with two initial objects: the cycle at infinity and the real line.

75a ⟨figure class 75a⟩≡ (52a) 75e ▷
```
figure::figure() : inherited(), k(realsymbol("k")), m(realsymbol("m")), l()
{
    l.append(realsymbol("l0"));
    l.append(realsymbol("l1"));
    infinity=symbol("infty","\\infty");
    real_line=symbol("R","\\mathbf{R}");
  point_metric = clifford_unit(varidx(real_line, 2), indexed(-(new tensdelta)->setflag(status_flags::dynallocated),
                                  sy_symm(), varidx(symbol("i"), 2), varidx(symbol("j"), 2)));
  cycle_metric = clifford_unit(varidx(real_line, 2), indexed(-(new tensdelta)->setflag(status_flags::dynallocated),
                                  sy_symm(), varidx(symbol("ic"), 2), varidx(symbol("jc"), 2)));
    ⟨set the infinity 75c⟩
    ⟨set the real line 75d⟩
}
```
Defines:
`figure`, used in chunks 16–18, 20e, 22, 27b, 28a, 30a, 36, 37, 45, 46c, 48, 50a, 52, 66, 75–84, 86, 87, 95d, 97–100, 106–108, 110, 111, and 117c.

Uses `cycle_metric` 50f, `infinity` 50e, `k` 51c, `l` 51c, `m` 51c, `point_metric` 50f, `real_line` 50e, and `realsymbol` 27c.

Dimension of the fiigure is taken and the respective vector is created.

75b ⟨initialise the dimension and vector 75b⟩≡ (75c 78f)
```
unsigned int dim=ex_to<numeric>(get_dim()).to_int();
lst l0;
for(unsigned int i=0; i<dim; ++i)
    l0.append(0);
```
Uses `get_dim()` 35c and `numeric` 22d.

75c ⟨set the infinity 75c⟩≡ (75a 78f 100a)
```
⟨initialise the dimension and vector 75b⟩
nodes[infinity] = cycle_node(cycle_data(numeric(0),indexed(matrix(1, dim, l0),
                                 varidx(infinity, dim)),numeric(1)),INFINITY_GEN);
```
Uses `cycle_data` 23a 26a 42c 43a 54c 54d 55a 56b 56c 56d, `cycle_node` 43c 45c 69c 70a 70b 70c 71a 71b 72b 72c 73a 74a 74d, `infinity` 50e, `INFINITY_GEN` 42b 42b, `nodes` 51a, and `numeric` 22d.

75d ⟨set the real line 75d⟩≡ (75a 78f 100a)
```
l0.remove_last();
l0.append(1);
nodes[real_line] = cycle_node(cycle_data(numeric(0),indexed(matrix(1, dim, l0),
                                 varidx(real_line, dim)),numeric(0)),REAL_LINE_GEN);
```
Uses `cycle_data` 23a 26a 42c 43a 54c 54d 55a 56b 56c 56d, `cycle_node` 43c 45c 69c 70a 70b 70c 71a 71b 72b 72c 73a 74a 74d, `nodes` 51a, `numeric` 22d, `real_line` 50e, and `REAL_LINE_GEN` 42b 42b.

This constructor may be called with several different inputs.

75e ⟨figure class 75a⟩+≡ (52a) ◁ 75a 76c ▷
```
figure::figure(const ex & Mp, const ex & Mc) : inherited(), k(realsymbol("k")), m(realsymbol("m")), l()
{
    infinity=symbol("infty","\\infty");
    real_line=symbol("R","\\mathbf{R}");
    bool inf_missing=true, R_missing=true;
    ⟨set point metric in figure 76a⟩
```
Uses `ex` 41b 47e 47e 47e 53a, `figure` 16d 22e 32a 32c 38b 49a 50d 75a 80a 82b 82c 85a 86c 98c 99b 99d 100a 101a 103b 104a 105c 106c 106d 107a 109a 109c 110a, `infinity` 50e, `k` 51c, `l` 51c, `m` 51c, `real_line` 50e, and `realsymbol` 27c.

Below are various parameters which can define a metric in the same way as it used to create a *cliffordunit* object in GiNaC. The point metric is indexed by the key of the real line and the cycle metric—by the key of the zero-radius cycle at infinity.

76a ⟨set point metric in figure 76a⟩≡ (75e 98c) 76b▷
```
if (is_a<clifford>(Mp)) {
   point_metric = clifford_unit(varidx(real_line,
                          ex_to<idx>(ex_to<clifford>(Mp).get_metric().op(1)).get_dim()),
                     ex_to<clifford>(Mp).get_metric());
 } else if (is_a<matrix>(Mp)) {
   ex D;
   if (ex_to<matrix>(Mp).rows() ≡ ex_to<matrix>(Mp).cols())
      D=ex_to<matrix>(Mp).rows();
   else
      throw(std::invalid_argument("figure::figure(const ex &, const ex &):"
                          " only square matrices are admitted as point metric"));
  point_metric = clifford_unit(varidx(real_line, D), indexed(Mp, sy_symm(), varidx(symbol("i"), D), varidx(symbol("j"
 } else if (is_a<indexed>(Mp)) {
   point_metric = clifford_unit(varidx(real_line, ex_to<idx>(Mp.op(1)).get_dim()), Mp);
```

Uses `ex` 41b 47e 47e 47e 53a, `figure` 16d 22e 32a 32c 38b 49a 50d 75a 80a 82b 82c 85a 86c 98c 99b 99d 100a 101a 103b 104a 105c 106c 106d 107a 109a 109c 110a, `get_dim()` 35c, `op` 50a, `point_metric` 50f, and `real_line` 50e.

If a **lst** is supplied we use as the signature of metric, entries *Mp* as the diagonal elements of the matrix.

76b ⟨set point metric in figure 76a⟩+≡ (75e 98c) ◁76a
```
   } else if (is_a<lst>(Mp)) {
    point_metric=clifford_unit(varidx(real_line, Mp.nops()), indexed(diag_matrix(ex_to<lst>(Mp)), sy_symm(),
                               varidx(symbol("i"), Mp.nops()), varidx(symbol("j"), Mp.nops())));
   }
```

Uses `nops` 50a, `point_metric` 50f, and `real_line` 50e.

If *Mp* is a figure we effectively copy it.

76c ⟨figure class 75a⟩+≡ (52a) ◁75e 76e▷
```
   else if (is_a<figure>(Mp)) {
      point_metric = ex_to<figure>(Mp).get_point_metric();
      cycle_metric = ex_to<figure>(Mp).get_cycle_metric();
      exhashmap<cycle_node> nnodes = ex_to<figure>(Mp).get_nodes();
      for (const auto& x: nnodes) {
         nodes[x.first]=x.second;
         ⟨identify infinity and real line 76d⟩
         }
```

Uses `cycle_metric` 50f, `cycle_node` 43c 45c 69c 70a 70b 70c 71a 71b 72b 72c 73a 74a 74d, `figure` 16d 22e 32a 32c 38b 49a 50d 75a 80a 82b 82c 85a 86c 98c 99b 99d 100a 101a 103b 104a 105c 106c 106d 107a 109a 109c 110a, `get_cycle_metric` 35b, `get_point_metric` 19a 35b, `nodes` 51a, and `point_metric` 50f.

We need to set *real_line* and *infinity* accordingly.

76d ⟨identify infinity and real line 76d⟩≡ (76c 79b)
```
         if (x.second.get_generation() ≡ REAL_LINE_GEN) {
            real_line = x.first;
            R_missing=false;
         }
         else if (x.second.get_generation() ≡ INFINITY_GEN) {
            infinity = x.first;
            inf_missing=false;
         }
```

Uses `get_generation` 35f, `infinity` 50e, `INFINITY_GEN` 42b 42b, `real_line` 50e, and `REAL_LINE_GEN` 42b 42b.

For an unknown type parameter we throw an exception.

76e ⟨figure class 75a⟩+≡ (52a) ◁76c 77e▷
```
} else
    throw(std::invalid_argument("figure::figure(const ex &, const ex &):"
                          " the first parameter shall be a figure, a lst, "
                          " a metric (can be either tensor, matrix,"
                          " Clifford unit or indexed by two indices) "));
⟨set cycle metric in figure 77a⟩
```

Uses `ex` 41b 47e 47e 47e 53a and `figure` 16d 22e 32a 32c 38b 49a 50d 75a 80a 82b 82c 85a 86c 98c 99b 99d 100a 101a 103b 104a 105c 106c 106d 107a 109a 109c 110a.

If a metric is not supplied or is zero then we clone the point space metric by the rule defined in equation (17). The cycle metric is indexed by the key of the zero-radius cycle at infinity. If the same index as for the point metric is used, then we have an issue with **figure**::*read_archive*(): for some mysterious reasons cycle metric is always a copy of the point metric.

77a ⟨set cycle metric in figure 77a⟩≡ (76e 98c) 77b▷
```
if (Mc.is_zero()) {
   ex D=get_dim();
   if (is_a<numeric>(D)) {
      lst l0;
      for(int i=0; i< ex_to<numeric>(D).to_int(); ++i)
         l0.append(-jump_fnct(-ex_to<clifford>(point_metric).get_metric(idx(i,D),idx(i,D))));
      cycle_metric = clifford_unit(varidx(infinity, D), indexed(diag_matrix(l0), sy_symm(),
                                  varidx(symbol("ic"), D), varidx(symbol("jc"), D)));
```

Uses `cycle_metric` 50f, `ex` 41b 47e 47e 47e 53a, `get_dim()` 35c, `infinity` 50e, `numeric` 22d, and `point_metric` 50f.

If dimensionality is not integer, then the point metric is copied.

77b ⟨set cycle metric in figure 77a⟩+≡ (76e 98c) ◁77a 77c▷
```
   } else
      cycle_metric = clifford_unit(varidx(infinity, D), indexed(point_metric.op(0), sy_symm(),
                                  varidx(symbol("ic"), D), varidx(symbol("jc"), D)));
```

Uses `cycle_metric` 50f, `infinity` 50e, `op` 50a, and `point_metric` 50f.

If the metric is supplied, we repeat the same procedure to set-up the metric of the cycle space as was done for point space.

77c ⟨set cycle metric in figure 77a⟩+≡ (76e 98c) ◁77b 77d▷
```
} else if (is_a<clifford>(Mc)) {
   cycle_metric = clifford_unit(varidx(infinity,
                     ex_to<idx>(ex_to<clifford>(Mc).get_metric().op(1)).get_dim()),
                  ex_to<clifford>(Mc).get_metric());
} else if (is_a<matrix>(Mc)) {
   if (ex_to<matrix>(Mp).rows() ≠ ex_to<matrix>(Mp).cols())
      throw(std::invalid_argument("figure::figure(const ex &, const ex &):"
                          " only square matrices are admitted as cycle metric"));

  cycle_metric = clifford_unit(varidx(infinity, get_dim()), indexed(Mc, sy_symm(), varidx(symbol("ic"),
                              get_dim()), varidx(symbol("jc"), get_dim())));
```

Uses `cycle_metric` 50f, `ex` 41b 47e 47e 47e 53a, `figure` 16d 22e 32a 32c 38b 49a 50d 75a 80a 82b 82c 85a 86c 98c 99b 99d 100a 101a 103b 104a 105c 106c 106d 107a 109a 109c 110a, `get_dim()` 35c, `infinity` 50e, and `op` 50a.

Other types of metric.

77d ⟨set cycle metric in figure 77a⟩+≡ (76e 98c) ◁77c
```
} else if (is_a<indexed>(Mc)) {
   cycle_metric = clifford_unit(varidx(infinity, ex_to<idx>(Mc.op(1)).get_dim()), Mc);
} else if (is_a<lst>(Mc)) {
  cycle_metric=clifford_unit(varidx(infinity, Mc.nops()), indexed(diag_matrix(ex_to<lst>(Mc)), sy_symm(),
                 varidx(symbol("ic"), Mc.nops()), varidx(symbol("jc"), Mc.nops())));
}
```

Uses `cycle_metric` 50f, `get_dim()` 35c, `infinity` 50e, `nops` 50a, and `op` 50a.

The error message

77e ⟨figure class 75a⟩+≡ (52a) ◁76e 78a▷
```
else
  throw(std::invalid_argument("figure::figure(const ex &, const ex &):"
                              " the second parameter"
                              " shall be omitted, equal to zero "
                              " or be a lst, a metric (can be either tensor, matrix,"
                              " Clifford unit or indexed by two indices)"));
```

Uses ex 41b 47e 47e 47e 53a and figure 16d 22e 32a 32c 38b 49a 50d 75a 80a 82b 82c 85a 86c 98c 99b 99d 100a 101a 103b 104a 105c 106c 106d 107a 109a 109c 110a.

Finally we check that point and cycle metrics have the same dimensionality.

78a ⟨figure class 75a⟩+≡ (52a) ◁77e 78b▷
```
if (¬ (get_dim()-ex_to<idx>(cycle_metric.op(1)).get_dim()).is_zero())
  throw(std::invalid_argument("figure::figure(const ex &, const ex &):"
                              "the point and cycle metrics shall have "
                              "the same dimensions"));
```

Uses cycle_metric 50f, ex 41b 47e 47e 47e 53a, figure 16d 22e 32a 32c 38b 49a 50d 75a 80a 82b 82c 85a 86c 98c 99b 99d 100a 101a 103b 104a 105c 106c 106d 107a 109a 109c 110a, get_dim() 35c, and op 50a.

We also check that *point_metric* and *cycle_metric* has the same dimensionality.

78b ⟨figure class 75a⟩+≡ (52a) ◁78a 78e▷
```
⟨check dimensionalities point and cycle metrics 78c⟩
⟨add symbols to match dimensionality 78d⟩
```

78c ⟨check dimensionalities point and cycle metrics 78c⟩≡ (78b 98c)
```
if (¬(get_dim()-ex_to<varidx>(cycle_metric.op(1)).get_dim()).is_zero())
  throw(std::invalid_argument("Metrics for point and cycle spaces have"
                              " different dimensionalities!"));
```

Uses cycle_metric 50f, get_dim() 35c, and op 50a.

We produce enough symbols to match dimensionality.

78d ⟨add symbols to match dimensionality 78d⟩≡ (78b 79b)
```
int D;
if (is_a<numeric>(get_dim())) {
  D=ex_to<numeric>(get_dim()).to_int();
  char name[6];
  for(int i=0; i<D; ++i) {
    sprintf(name, "l%d", i);
    l.append(realsymbol(name));
  }
}
```

Uses get_dim() 35c, l 51c, name 32e, numeric 22d, and realsymbol 27c.

Finally, we set-up two elements which present at any figure: the real line and infinity.

78e ⟨figure class 75a⟩+≡ (52a) ◁78b 79a▷
```
  ⟨setup real line and infinity 78f⟩
}
```

Finally, we supply nodes for the real line and the cycle at infinity.

78f ⟨setup real line and infinity 78f⟩≡ (78e 79b)
```
if (inf_missing) {
  ⟨set the infinity 75c⟩
}
if (R_missing) {
  ⟨initialise the dimension and vector 75b⟩
  ⟨set the real line 75d⟩
}
```

79a ⟨figure class 75a⟩+≡ (52a) ◁78e 79b▷

```
figure::figure(const ex & Mp, const ex & Mc, const exhashmap<cycle_node> & N):
        inherited(), k(realsymbol("k")), m(realsymbol("m")), l()
{
    infinity=symbol("infty","\\infty");
    real_line=symbol("R","\\mathbf{R}");
    bool inf_missing=true, R_missing=true;
    if (is_a<clifford>(Mp) ∧ is_a<clifford>(Mc)) {
        point_metric = Mp;
        cycle_metric = Mc;
    } else
     throw(std::invalid_argument("figure::figure(const ex &, const ex &, exhashmap<cycle_node>):"
                            " the point_metric and cycle_metric should be clifford_unit. "));
```

Uses cycle_metric 50f, cycle_node 43c 45c 69c 70a 70b 70c 71a 71b 72b 72c 73a 74a 74d, ex 41b 47e 47e 47e 53a, figure 16d 22e 32a 32c 38b 49a 50d 75a 80a 82b 82c 85a 86c 98c 99b 99d 100a 101a 103b 104a 105c 106c 106d 107a 109a 109c 110a, infinity 50e, k 51c, l 51c, m 51c, point_metric 50f, real_line 50e, and realsymbol 27c.

We coming nodes of cycle to the new figure.

79b ⟨figure class 75a⟩+≡ (52a) ◁79a 79c▷

```
    for (const auto& x: N) {
        nodes[x.first]=x.second;
        ⟨identify infinity and real line 76d⟩
    }
    ⟨add symbols to match dimensionality 78d⟩
    ⟨setup real line and infinity 78f⟩
}
```

Uses nodes 51a.

This constructor reads a figure from a file given by name.

79c ⟨figure class 75a⟩+≡ (52a) ◁79b 80a▷

```
figure::figure(const char* file_name, string fig_name) : inherited(), k(realsymbol("k")), m(realsymbol("m")), l()
{
    infinity=symbol("infty","\\infty");
    real_line=symbol("R","\\mathbf{R}");
    ⟨add gar extension 79d⟩
    GiNaC::archive A;
    std::ifstream ifs(fn.c_str(),  std::ifstream::in);

    ifs ≫ A;
    ∗this=ex_to<figure>(A.unarchive_ex(lst{infinity, real_line}, fig_name));
    string operation_name="read";
    ⟨write raw archive printout ??⟩
}
```

Uses archive 50a, figure 16d 22e 32a 32c 38b 49a 50d 75a 80a 82b 82c 85a 86c 98c 99b 99d 100a 101a 103b 104a 105c 106c 106d 107a 109a 109c 110a, infinity 50e, k 51c, l 51c, m 51c, real_line 50e, and realsymbol 27c.

We use c++17 features to process file names.

?? ⟨write raw archive printout ??⟩≡ (79c 80a)

```
#if __cplusplus >= 201703L
    if (FIGURE_DEBUG) {
        std::filesystem::path file_path=std::filesystem::path(fn.c_str()),
            file_name=std::filesystem::path("raw-"+operation_name+"-");
        file_name+=file_path.filename();
        ofstream out1(file_path.replace_filename(file_name));
        A.printraw(out1);
        out1.close();
        out1.flush();
    }
#endif
```

Uses FIGURE_DEBUG 52c.

.gar is the standard extension for GiNaC archive files.

79d ⟨add gar extension 79d⟩≡ (79c 80a)
```
string fn=file_name;
size_t found = fn.find(".gar");
if (found ≡ std::string::npos)
    fn=fn+".gar";

if (FIGURE_DEBUG)
    cerr ≪ "use filename: " ≪ fn ≪ endl;
```

Uses FIGURE_DEBUG 52c.

This method saves the figure to a file, which can be read by the above constructor.

80a ⟨figure class 75a⟩+≡ (52a) ◁79c 80b▷
```
void figure::save(const char* file_name, const char * fig_name) const
{
    ⟨add gar extension 79d⟩
    GiNaC::archive A;
    A.archive_ex(*this, fig_name);
    ofstream out(fn.c_str());
    out ≪ A;
    out.flush();
    out.close();
    string operation_name="save";
    ⟨write raw archive printout ??⟩
}
```

Defines:
figure, used in chunks 16–18, 20e, 22, 27b, 28a, 30a, 36, 37, 45, 46c, 48, 50a, 52, 66, 75–84, 86, 87, 95d, 97–100, 106–108, 110, 111, and 117c.
Uses archive 50a and save 38a 38a.

F.5.2. *Addition of new cycles.* This method is merely a wrapper for the second form below.

80b ⟨figure class 75a⟩+≡ (52a) ◁80a 80c▷
```
ex figure::add_point(const ex & x, string name, string TeXname)
{
    ⟨auto TeX name 84d⟩
    symbol key(name, TeXname_new);
    return add_point(x, key);
}
```
Defines:
add_point, used in chunks 17c and 23b.
Uses ex 41b 47e 47e 47e 53a, figure 16d 22e 32a 32c 38b 49a 50d 75a 80a 82b 82c 85a 86c 98c 99b 99d 100a 101a 103b 104a 105c 106c 106d 107a 109a 109c 110a, key 32e, name 32e, and TeXname 32e.

We start from check of parameters.

80c ⟨figure class 75a⟩+≡ (52a) ◁80b 81c▷
```
ex figure::add_point(const ex & x, const ex & key)
{
    if (not (is_a<lst>(x) and (x.nops() ≡ get_dim())))
        throw(std::invalid_argument("figure::add_point(const ex &, const ex &): "
                                    "coordinates of a point shall be a lst of the right lenght"));

    if (not is_a<symbol>(key))
        throw(std::invalid_argument("figure::add_point(const ex &, const ex &): the third"
                                    " argument need to be a point"));
⟨adding point with its parents 81a⟩
```
Defines:
add_point, used in chunks 17c and 23b.
Uses ex 41b 47e 47e 47e 53a, figure 16d 22e 32a 32c 38b 49a 50d 75a 80a 82b 82c 85a 86c 98c 99b 99d 100a 101a 103b 104a 105c 106c 106d 107a 109a 109c 110a, get_dim() 35c, key 32e, and nops 50a.

This part of the code is shared with *move_point*(). We create two ghost parents for a point, since the parameters of the cycle representing depend from the metric, thus it shall not be hard-coded into the node, see also Section 3.2.

81a ⟨adding point with its parents 81a⟩≡ (80c 86a) 81b ▷

```
int dim=x.nops();
lst l0, rels;
rels.append(cycle_relation(key,cycle_orthogonal,false));
rels.append(cycle_relation(infinity,cycle_different));

for(int i=0; i < dim; ++i)
    l0.append(numeric(0));

for(int i=0; i < dim; ++i) {
    l0[i]=numeric(1);
    char name[8];
    sprintf(name, "-(%d)", i);
    symbol mother(ex_to<symbol>(key).get_name()+name);
    nodes[mother]=cycle_node(cycle_data(numeric(0),indexed(matrix(1, dim, l0),
                                    varidx(mother, get_dim())),numeric(2)*x.op(i)),
                      GHOST_GEN, lst{}, lst{key});
    l0[i]=numeric(0);
    rels.append(cycle_relation(mother,cycle_orthogonal));
}
```

Uses cycle_data 23a 26a 42c 43a 54c 54d 55a 56b 56c 56d, cycle_different 34e 114c, cycle_node 43c 45c 69c 70a 70b 70c 71a 71b 72b 72c 73a 74a 74d, cycle_orthogonal 34b 113a, cycle_relation 40c 45e 46c 60d 61 62a 62b 64a 64b, get_dim() 35c, GHOST_GEN 42b 42b, infinity 50e, key 32e, name 32e, nodes 51a, nops 50a, numeric 22d, and op 50a.

We add relations to parents which define this point. All relations are given in *cycle_metric*, only self-orthogonality is given in terms of *point_metric*. This is done in sake of the parabolic point space.

81b ⟨adding point with its parents 81a⟩+≡ (80c 86a) ◁ 81a

```
nodes[key]=cycle_node(cycle_data(), 0, rels);
```

Uses cycle_data 23a 26a 42c 43a 54c 54d 55a 56b 56c 56d, cycle_node 43c 45c 69c 70a 70b 70c 71a 71b 72b 72c 73a 74a 74d, key 32e, and nodes 51a.

Now, cycle date shall be generated.

81c ⟨figure class 75a⟩+≡ (52a) ◁ 80c 81d ▷

```
    if (¬ info(status_flags::expanded))
        nodes[key].set_cycles(ex_to<lst>(update_cycle_node(key)));
    if (FIGURE_DEBUG)
        cerr << "Add the point: " << x << " as the cycle: " << nodes[key] << endl;
    return key;
}
```

Uses FIGURE_DEBUG 52c, info 50a, key 32e, nodes 51a, and update_cycle_node 49b 95d.

Add a cycle at zero level with a prescribed data.

81d ⟨figure class 75a⟩+≡ (52a) ◁ 81c 82a ▷

```
ex figure::add_cycle(const ex & C, const ex & key)
{
    ex lC=ex_to<cycle>(C).get_l();
    if (is_a<indexed>(lC))
        nodes[key]=cycle_node(C.subs(lC.op(1)≡key));
    else
        nodes[key]=cycle_node(C);
    if (FIGURE_DEBUG)
        cerr << "Add the cycle: " << nodes[key] << endl;
    return key;
}
```

Defines:
add_cycle, used in chunks 19–21, 28b, 30b, 82a, and 117c.

Uses cycle_node 43c 45c 69c 70a 70b 70c 71a 71b 72b 72c 73a 74a 74d, ex 41b 47e 47e 47e 53a, figure 16d 22e 32a 32c 38b 49a 50d 75a 80a 82b 82c 85a 86c 98c 99b 99d 100a 101a 103b 104a 105c 106c 106d 107a 109a 109c 110a, FIGURE_DEBUG 52c, key 32e, nodes 51a, op 50a, and subs 50a.

Add a cycle at zero level with a prescribed data.

82a ⟨figure class 75a⟩+≡ (52a) ◁81d 82b▷
```
ex figure::add_cycle(const ex & C, string name, string TeXname)
{
   ⟨auto TeX name 84d⟩
   symbol key(name, TeXname_new);
   return add_cycle(C, key);
 }
```

Uses add_cycle 23a 32f 81d, ex 41b 47e 47e 47e 53a, figure 16d 22e 32a 32c 38b 49a 50d 75a 80a 82b 82c 85a 86c 98c 99b 99d 100a 101a 103b 104a 105c 106c 106d 107a 109a 109c 110a, key 32e, name 32e, and TeXname 32e.

82b ⟨figure class 75a⟩+≡ (52a) ◁82a 82c▷
```
void figure::set_cycle(const ex & key, const ex & C)
{
   if (nodes.find(key) ≡ nodes.end())
      throw(std::invalid_argument("figure::set_cycle(): there is no node wi\
th the key given"));

   if (nodes[key].get_parents().nops() > 0)
      throw(std::invalid_argument("figure::set_cycle(): cannot modify data \
of a cycle with parents"));

   nodes[key].set_cycles(C);

   if (FIGURE_DEBUG)
      cerr ≪ "Replace the cycle: " ≪ nodes[key] ≪ endl;
}
```
Defines:
figure, used in chunks 16–18, 20e, 22, 27b, 28a, 30a, 36, 37, 45, 46c, 48, 50a, 52, 66, 75–84, 86, 87, 95d, 97–100, 106–108, 110, 111, and 117c.
set_cycle, used in chunks 82c and 97c.
Uses ex 41b 47e 47e 47e 53a, FIGURE_DEBUG 52c, key 32e, nodes 51a, and nops 50a.

82c ⟨figure class 75a⟩+≡ (52a) ◁82b 82d▷
```
void figure::move_cycle(const ex & key, const ex & C)
{
   if (nodes.find(key) ≡ nodes.end())
      throw(std::invalid_argument("figure::set_cycle(): there is no node with the key given"));

   if (nodes[key].get_generation() ≠ 0)
      throw(std::invalid_argument("figure::set_cycle(): cannot modify data of a cycle in"
                           " non-zero generation"));
```
Defines:
figure, used in chunks 16–18, 20e, 22, 27b, 28a, 30a, 36, 37, 45, 46c, 48, 50a, 52, 66, 75–84, 86, 87, 95d, 97–100, 106–108, 110, 111, and 117c.
move_cycle, used in chunk 26b.
Uses ex 41b 47e 47e 47e 53a, get_generation 35f, key 32e, nodes 51a, and set_cycle 49b 82b.

If we have at zero generation with parents, then they are ghost parents of the point, so shall be deleted. We cannot do this by *remove_cycle_node* since we do not want to remove all its grand childrens.

82d ⟨figure class 75a⟩+≡ (52a) ◁82c 83a▷
```
   if (nodes[key].get_parents().nops() > 0) {
      lst par=nodes[key].get_parent_keys();
      for(const auto& it : par)
         if (nodes[it].get_generation() ≡ GHOST_GEN)
            nodes.erase(it);
         else
            nodes[it].remove_child(key);
   }
   nodes[key].parents=lst{};
```

Uses get_generation 35f, GHOST_GEN 42b 42b, key 32e, nodes 51a, and nops 50a.

Now, the cycle may be set.

83a ⟨figure class 75a⟩+≡ (52a) ◁82d 83b▷

```
    nodes[key].set_cycles(C);
    update_node_lst(nodes[key].get_children());

    if (FIGURE_DEBUG)
        cerr << "Replace the cycle: " << nodes[key] << endl;
}
```

Uses `FIGURE_DEBUG` 52c, `key` 32e, `nodes` 51a, and `update_node_lst` 50c 100a.

A cycle can be added by a single **cycle_relation** or a **lst** of **cycle_relation**, but this is just a wrapper for a more general case below.

83b ⟨figure class 75a⟩+≡ (52a) ◁83a 83c▷

```
ex figure::add_cycle_rel(const ex & rel, const ex & key) {
    if (is_a<cycle_relation>(rel))
        return add_cycle_rel(lst{rel}, key);
    else
        throw(std::invalid_argument("figure::add_cycle_rel: a cycle shall be added "
                                    "by a single expression, which is a cycle_relation"));
}
```

Defines:
`add_cycle_rel`, used in chunks 17, 19–21, 23–25, 28–31, 83, 84a, 117, and 118.

Uses `cycle_relation` 40c 45e 46c 60d 61 62a 62b 64a 64b, `ex` 41b 47e 47e 47e 53a, `figure` 16d 22e 32a 32c 38b 49a 50d 75a 80a 82b 82c 85a 86c 98c 99b 99d 100a 101a 103b 104a 105c 106c 106d 107a 109a 109c 110a, and `key` 32e.

And now we add a cycle defined the list of relations. The generation of the new cycle is calculated by the rules described in Sec. 3.2.

83c ⟨figure class 75a⟩+≡ (52a) ◁83b 83d▷

```
ex figure::add_cycle_rel(const lst & rel, const ex & key)
{
    lst cond;
    int gen=0;

    for(const auto& it : rel) {
        if (ex_to<cycle_relation>(it).get_parkey() ≠ key)
            gen=max(gen, nodes[ex_to<cycle_relation>(it).get_parkey()].get_generation());
        nodes[ex_to<cycle_relation>(it).get_parkey()].add_child(key);
    }

    nodes[key]=cycle_node(cycle_data(),gen+1,rel);
```

Uses `add_cycle_rel` 16f 23c 33a 83b, `cycle_data` 23a 26a 42c 43a 54c 54d 55a 56b 56c 56d, `cycle_node` 43c 45c 69c 70a 70b 70c 71a 71b 72b 72c 73a 74a 74d, `cycle_relation` 40c 45e 46c 60d 61 62a 62b 64a 64b, `ex` 41b 47e 47e 47e 53a, `figure` 16d 22e 32a 32c 38b 49a 50d 75a 80a 82b 82c 85a 86c 98c 99b 99d 100a 101a 103b 104a 105c 106c 106d 107a 109a 109c 110a, `get_generation` 35f, `key` 32e, and `nodes` 51a.

83d ⟨figure class 75a⟩+≡ (52a) ◁83c 83e▷

```
    if (¬ info(status_flags::expanded))
        nodes[key].set_cycles(ex_to<lst>(update_cycle_node(key)));

    if (FIGURE_DEBUG)
        cerr << "Add the cycle: " << nodes[key] << endl;

    return key;
}
```

Uses `FIGURE_DEBUG` 52c, `info` 50a, `key` 32e, `nodes` 51a, and `update_cycle_node` 49b 95d.

This version automatically supply TEX label like c_{23} to symbols with names *c23*.

83e ⟨figure class 75a⟩+≡ (52a) ◁83d 84a▷
```
ex figure::add_cycle_rel(const lst & rel, string name, string TeXname)
{
    ⟨auto TeX name 84d⟩
    return add_cycle_rel(rel, symbol(name, TeXname_new));
}
```

Uses add_cycle_rel 16f 23c 33a 83b, ex 41b 47e 47e 47e 53a, figure 16d 22e 32a 32c 38b 49a 50d 75a 80a 82b 82c 85a 86c 98c 99b 99d 100a 101a 103b 104a 105c 106c 106d 107a 109a 109c 110a, name 32e, and TeXname 32e.

A similar method to add a cycle by a single relation.

84a ⟨figure class 75a⟩+≡ (52a) ◁83e 84b▷
```
ex figure::add_cycle_rel(const ex & rel, string name, string TeXname)
{
    if (is_a<cycle_relation>(rel)) {
        ⟨auto TeX name 84d⟩
        return add_cycle_rel(lst{rel}, symbol(name, TeXname_new));
    } else
        throw(std::invalid_argument("figure::add_cycle_rel: a cycle shall be added "
                                    "by a single expression, which is a cycle_relation"));
}
```

Uses add_cycle_rel 16f 23c 33a 83b, cycle_relation 40c 45e 46c 60d 61 62a 62b 64a 64b, ex 41b 47e 47e 47e 53a, figure 16d 22e 32a 32c 38b 49a 50d 75a 80a 82b 82c 85a 86c 98c 99b 99d 100a 101a 103b 104a 105c 106c 106d 107a 109a 109c 110a, name 32e, and TeXname 32e.

This method adds a **subfigure** as a single node. The generation of the new node is again calculated by the rules described in Sec. 3.2.

84b ⟨figure class 75a⟩+≡ (52a) ◁84a 84c▷
```
ex figure::add_subfigure(const ex & F, const lst & L, const ex & key)
{
    GINAC_ASSERT(is_a<figure>(F));
    int gen=0;

    for(const auto& it : L) {
        if (¬ it.is_equal(key))
            gen=max(gen, nodes[it].get_generation());
        nodes[it].add_child(key);
    }
    nodes[key]=cycle_node(cycle_data(),gen+1,lst{subfigure(F,L)});
    if (¬ info(status_flags::expanded))
        nodes[key].set_cycles(ex_to<lst>(update_cycle_node(key)));

    return key;
}
```
Defines:
add_subfigure, used in chunks 24 and 84c.

Uses cycle_data 23a 26a 42c 43a 54c 54d 55a 56b 56c 56d, cycle_node 43c 45c 69c 70a 70b 70c 71a 71b 72b 72c 73a 74a 74d, ex 41b 47e 47e 47e 53a, figure 16d 22e 32a 32c 38b 49a 50d 75a 80a 82b 82c 85a 86c 98c 99b 99d 100a 101a 103b 104a 105c 106c 106d 107a 109a 109c 110a, get_generation 35f, info 50a, key 32e, nodes 51a, subfigure 40d 48b 48d 66a 66b 66c 66d 66e, and update_cycle_node 49b 95d.

This is again a wrapper for the previous method with the newly defined symbol.

84c ⟨figure class 75a⟩+≡ (52a) ◁84b 85a▷
```
ex figure::add_subfigure(const ex & F, const lst & l, string name, string TeXname)
{
    ⟨auto TeX name 84d⟩
        return add_subfigure(F, l, symbol(name, TeXname_new));
}
```

Uses add_subfigure 24c 33b 84b, ex 41b 47e 47e 47e 53a, figure 16d 22e 32a 32c 38b 49a 50d 75a 80a 82b 82c 85a 86c 98c 99b 99d 100a 101a 103b 104a 105c 106c 106d 107a 109a 109c 110a, l 51c, name 32e, and TeXname 32e.

84d ⟨auto TeX name 84d⟩≡ (80b ? 82–84)

```
string TeXname_new;
std::regex e ("([[:alpha:]]+)([[:digit:]]+)");
std::regex e1 ("([[:alnum:]]+)_([[:alnum:]]+)");
if (TeXname ≡ "") {
    if (std::regex_match(name, e))
        TeXname_new=std::regex_replace (name,e,"$1_{$2}");
    else if (std::regex_match(name, e1))
        TeXname_new=std::regex_replace (name,e1,"$1_{$2}");
} else
    TeXname_new=TeXname;
```

Uses name 32e and TeXname 32e.

F.5.3. *Moving and removing cycles.* The method to change a zero-generation cycle to a point with given coordinates.

85a ⟨figure class 75a⟩+≡ (52a) ◁84c 85b▷

```
void figure::move_point(const ex & key, const ex & x)
{
    if (not (is_a<lst>(x) and (x.nops() ≡ get_dim())))
        throw(std::invalid_argument("figure::move_point(const ex &, const ex &): "
                                    "coordinates of a point shall be a lst of the right lenght"));

    if (nodes.find(key) ≡ nodes.end())
        throw(std::invalid_argument("figure::move_point(): there is no node with the key given"));

    if (nodes[key].get_generation() ≠ 0)
        throw(std::invalid_argument("figure::move_point(): cannot modify data of a cycle in"
                                    " non-zero generation"));

    if (FIGURE_DEBUG)
        cerr ≪ "A cycle is moved : " ≪ nodes[key] ≪ endl;
```

Defines:
figure, used in chunks 16–18, 20e, 22, 27b, 28a, 30a, 36, 37, 45, 46c, 48, 50a, 52, 66, 75–84, 86, 87, 95d, 97–100, 106–108, 110, 111, and 117c.
move_point, used in chunks 25, 26, and 86a.

Uses ex 41b 47e 47e 47e 53a, FIGURE_DEBUG 52c, get_dim() 35c, get_generation 35f, key 32e, nodes 51a, and nops 50a.

If number of parents was "dimension plus 2", so it was a proper point, we simply need to replace the ghost parents.

85b ⟨figure class 75a⟩+≡ (52a) ◁85a 85c▷

```
lst par=nodes[key].get_parent_keys();
unsigned int dim=x.nops();
lst l0;
for(unsigned int i=0; i<dim; ++i)
    l0.append(numeric(0));
```

Uses key 32e, nodes 51a, nops 50a, and numeric 22d.

We scan the name of parents to get number of components and substitute their new values.

85c ⟨figure class 75a⟩+≡ (52a) ◁85b 86a▷

```
char label[40];
sprintf(label, "%s-(%%d)",ex_to<symbol>(key).get_name().c_str());
if (par.nops() ≡ dim+2 ) {
    for(const auto& it : par) {
        unsigned int i=dim;
        int res=sscanf(ex_to<symbol>(it).get_name().c_str(), label, &i);
        if (res>0 and i<dim) {
            l0[i]=numeric(1);
            nodes[it].set_cycles(cycle_data(numeric(0),indexed(matrix(1, dim, l0),
                                            varidx(it, dim)), numeric(2)*x.op(i)));
            l0[i]=numeric(0);
        }
    }
```

Uses cycle_data 23a 26a 42c 43a 54c 54d 55a 56b 56c 56d, key 32e, nodes 51a, nops 50a, numeric 22d, and op 50a.

If the number of parents is zero, so it was a pre-defined cycle and we need to create ghost parents for it.

86a ⟨figure class 75a⟩+≡ (52a) ◁85c 86b▷
```
    } else if (par.nops() ≡ 0) {
        lst chil=nodes[key].get_children();
        ⟨adding point with its parents 81a⟩
        nodes[key].children=chil;
    } else
     throw(std::invalid_argument("figure::move_point(): strange number (neither 0 nor dim+2) of "
                                "parents, which zero-generation node shall have!"));

    if (info(status_flags::expanded))
        return;

    nodes[key].set_cycles(ex_to<lst>(update_cycle_node(key)));
    update_node_lst(nodes[key].get_children());
```

Uses figure 16d 22e 32a 32c 38b 49a 50d 75a 80a 82b 82c 85a 86c 98c 99b 99d 100a 101a 103b 104a 105c 106c 106d 107a 109a 109c 110a, info 50a, key 32e, move_point 25e 33c 85a, nodes 51a, nops 50a, update_cycle_node 49b 95d, and update_node_lst 50c 100a.

Then, to update all its children and grandchildren in all generations excluding this node itself.

86b ⟨figure class 75a⟩+≡ (52a) ◁86a 86c▷
```
     update_node_lst(nodes[key].get_children());
     if (FIGURE_DEBUG)
        cerr ≪ "Moved to: " ≪ x ≪ endl;
 }
```

Uses FIGURE_DEBUG 52c, key 32e, nodes 51a, and update_node_lst 50c 100a.

Afterwards, to remove all children (includes grand children, grand grand children…) of the **cycle_node**.

86c ⟨figure class 75a⟩+≡ (52a) ◁86b 86d▷
```
 void figure::remove_cycle_node(const ex & key)
 {
    lst branches=nodes[key].get_children();
    for (const auto& it : branches)
        remove_cycle_node(it);
```

Defines:
figure, used in chunks 16–18, 20e, 22, 27b, 28a, 30a, 36, 37, 45, 46c, 48, 50a, 52, 66, 75–84, 86, 87, 95d, 97–100, 106–108, 110, 111, and 117c.
remove_cycle_node, never used.

Uses ex 41b 47e 47e 47e 53a, key 32e, and nodes 51a.

Furthermore, to remove the **cycle_node** c from all its parents children lists.

86d ⟨figure class 75a⟩+≡ (52a) ◁86c 86e▷
```
    lst par = nodes[key].get_parent_keys();
    for (const auto& it : par) {
```

Uses key 32e and nodes 51a.

Parents of a point at gen-0 can be simply deleted as no other cycle need them and they are not of interest. For other parents we modify their *cildren* list.

86e ⟨figure class 75a⟩+≡ (52a) ◁86d 86f▷
```
        if (nodes[it].get_generation() ≡ GHOST_GEN)
            nodes.erase(it);
        else
            nodes[it].remove_child(key);
    }
```

Uses get_generation 35f, GHOST_GEN 42b 42b, key 32e, and nodes 51a.

Finally, remove the **cycle_node** from the figure.

86f ⟨figure class 75a⟩+≡ (52a) ◁86e 87a▷

```
    nodes.erase(key);
    if (FIGURE_DEBUG)
        cerr << "The cycle is removed: " << key << endl ;
}
```

Uses FIGURE_DEBUG 52c, key 32e, and nodes 51a.

F.5.4. *Evaluattion of cycles and figure updates.* This procedure can solve a system of linear conditions or a system with one quadratic equation. It was already observed in [18; 36, § 5.5], see Sec. 3.1, that n tangency-type conditions (each of them is quadratic) can be reduced to the single quadratic condition $\langle C, C\rangle = 1$ and n linear conditions like $\langle C, C^i\rangle = \lambda_i$.

87a ⟨figure class 75a⟩+≡ (52a) ◁86f 87b▷

```
ex figure::evaluate_cycle(const ex & symbolic, const lst & cond) const
{
    //cerr << boolalpha << "symbolic: "; symbolic.dbgprint();
    //cerr << "condit: "; cond.dbgprint();
    bool first_solution=true, // whetehr the first solution is suitable
        second_solution=false, // whetehr the second solution is suitable
        is_homogeneous=true; // indicates whether all conditions are linear
```

Defines:
evaluate_cycle, used in chunks 49c, 87d, and 96c.
Uses ex 41b 47e 47e 47e 53a and figure 16d 22e 32a 32c 38b 49a 50d 75a 80a 82b 82c 85a 86c 98c 99b 99d 100a 101a 103b 104a 105c 106c 106d 107a 109a 109c 110a.

This method can be applied to cycles with numerical dimensions.

87b ⟨figure class 75a⟩+≡ (52a) ◁87a 87c▷

```
    int D;
    if (is_a<numeric>(get_dim()))
        D=ex_to<numeric>(get_dim()).to_int();
    else
        throw logic_error("Could not resolve cycle relations if dimensionality is not numeric!");
```

Uses get_dim() 35c and numeric 22d.

Create the list of used symbols. The code is stolen from `cycle.nw`

87c ⟨figure class 75a⟩+≡ (52a) ◁87b 87d▷

```
    lst symbols, lin_cond, nonlin_cond;
    if (is_a<symbol>(ex_to<cycle_data>(symbolic).get_m()))
        symbols.append(ex_to<cycle_data>(symbolic).get_m());
    for (int i = 0; i < D; i++)
        if (is_a<symbol>(ex_to<cycle_data>(symbolic).get_l(i)))
            symbols.append(ex_to<cycle_data>(symbolic).get_l(i));
    if (is_a<symbol>(ex_to<cycle_data>(symbolic).get_k()))
        symbols.append(ex_to<cycle_data>(symbolic).get_k());
```

Uses cycle_data 23a 26a 42c 43a 54c 54d 55a 56b 56c 56d.

If no symbols are found we assume that the cycle is uniquely defined

87d ⟨figure class 75a⟩+≡ (52a) ◁87c 88a▷

```
    if (symbols.nops() == 0)
     throw(std::invalid_argument("figure::evaluate_cycle(): could not construct the default list of "
                            "parameters"));
    //cerr << "symbols: "; symbols.dbgprint();
```

Uses evaluate_cycle 87a, figure 16d 22e 32a 32c 38b 49a 50d 75a 80a 82b 82c 85a 86c 98c 99b 99d 100a 101a 103b 104a 105c 106c 106d 107a 109a 109c 110a, and nops 50a.

Build matrix representation from equation system. The code is stolen from `ginac/inifcns.cpp`.

88a ⟨figure class 75a⟩+≡ (52a) ◁87d 88b▷

```
lst rhs;
for (size_t r=0; r<cond.nops(); r++) {
    lst sys;
    ex eq = (cond.op(r).op(0)-cond.op(r).op(1)).expand(); // lhs-rhs==0
    if (float_evaluation)
        eq=eq.evalf();
    //cerr << "eq: "; eq.dbgprint();
    ex linpart = eq;
    for (size_t c=0; c<symbols.nops(); c++) {
        const ex co = eq.coeff(ex_to<symbol>(symbols.op(c)),1);
        linpart -= co*symbols.op(c);
        sys.append(co);
    }
    linpart = linpart.expand();
    //cerr << "sys: "; sys.dbgprint();
    //cerr << "linpart: "; linpart.dbgprint();
```

Uses `evalf` 50a, `ex` 41b 47e 47e 47e 53a, `float_evaluation` 51b, `nops` 50a, and `op` 50a.

test if system is linear and fill vars matrix

88b ⟨figure class 75a⟩+≡ (52a) ◁88a 88c▷

```
    bool is_linear=true;
    for (size_t i=0; i<symbols.nops(); i++)
        if (sys.has(symbols.op(i)) ∨ linpart.has(symbols.op(i)))
            is_linear = false;
    //cerr << "this equation linear? " << is_linear << endl;
```

Uses `nops` 50a and `op` 50a.

To avoid an expensive expansion we use the previous calculations to re-build the equation.

88c ⟨figure class 75a⟩+≡ (52a) ◁88b 88d▷

```
    if (is_linear) {
        lin_cond.append(sys);
        rhs.append(linpart);
        is_homogeneous &= linpart.normal().is_zero();
    } else
        nonlin_cond.append(cond.op(r));
}
//cerr << "lin_cond: "; lin_cond.dbgprint();
//cerr << "nonlin_cond: "; nonlin_cond.dbgprint();
```

Uses `op` 50a.

Solving the linear part, the code is again stolen from `ginac/inifcns.cpp`

88d ⟨figure class 75a⟩+≡ (52a) ◁88c 89a▷

```
lst subs_lst1, // The main list of substitutions of found solutions
    subs_lst2, // The second solution lists for quadratic equations
    free_vars; // List of free variables being parameters of the solution
if (lin_cond.nops()>0) {
    matrix solution;
    try {
        solution=ex_to<matrix>(lst_to_matrix(lin_cond)).solve(matrix(symbols.nops(),1,symbols),
                                                  matrix(rhs.nops(),1,rhs));
```

Uses `main` 16b 17a 18b 20e 22c 27b 28a 30a and `nops` 50a.

If the system is incompatible no cycle data is returned (probably singular matrix or otherwise overdetermined system, it is consistent to return an empty list)

89a ⟨figure class 75a⟩+≡ (52a) ◁88d 89b▷
```
    } catch (const std::runtime_error & e) {
        return lst{};
    }
    GINAC_ASSERT(solution.cols()==1);
    GINAC_ASSERT(solution.rows()==symbols.nops());
```

Uses nops 50a.

Now we sort out the result: free variables will be used for non-linear equation, resolved variables—for substitution.

89b ⟨figure class 75a⟩+≡ (52a) ◁89a 89c▷
```
    for (size_t i=0; i<symbols.nops(); i++)
        if (symbols.op(i)==solution(i,0))
            free_vars.append(symbols.op(i));
        else
            subs_lst1.append(symbols.op(i)==solution(i,0));
}
//cerr << "Lin system is homogeneous: " << is_homogeneous << endl;
```

Uses nops 50a and op 50a.

It is easy to solve a linear system, thus we immediate substitute the result.

89c ⟨figure class 75a⟩+≡ (52a) ◁89b 89d▷
```
cycle_data C_new, C1_new;
if (nonlin_cond.nops() == 0) {
    C_new = ex_to<cycle_data>(symbolic.subs(subs_lst1)).normalize();
    //cerr << "C_new: "; C_new.dbgprint();
```

Uses cycle_data 23a 26a 42c 43a 54c 54d 55a 56b 56c 56d, nops 50a, and subs 50a.

We check that the solution is not identical zero, which may happen for homogeneous conditions, for example. For this we prepare the respective norm of the cycle.

89d ⟨figure class 75a⟩+≡ (52a) ◁89c 89e▷
```
ex norm=pow(ex_to<cycle_data>(symbolic).get_k(),2)+pow(ex_to<cycle_data>(symbolic).get_m(),2);
for (int i = 0; i < D; i++)
    norm+=pow(ex_to<cycle_data>(symbolic).get_l(i),2);
first_solution &= ! is_less_than_epsilon(norm.subs(subs_lst1,
                                    subs_options::algebraic | subs_options::no_pattern));
```

Uses cycle_data 23a 26a 42c 43a 54c 54d 55a 56b 56c 56d, ex 41b 47e 47e 47e 53a, is_less_than_epsilon 53b, and subs 50a.

If some non-linear equations present and there are free variables, we sort out free and non-free variables.

89e ⟨figure class 75a⟩+≡ (52a) ◁89d 89f▷
```
} else if (free_vars.nops() > 0) {
    lst nonlin_cond_new;
    //cerr << "free_vars: "; free_vars.dbgprint();
    //cerr << "subs_lst1: "; subs_lst1.dbgprint();
```

Uses nops 50a.

Only one non-linear (quadratic) equation can be treated by this method, so we pick up the first from the list (hopefully other will be satisfied afterwards).

89f ⟨figure class 75a⟩+≡ (52a) ◁89e 90a▷
```
    ex quadratic_eq=nonlin_cond.op(0).subs(subs_lst1, subs_options::algebraic
                                    | subs_options::no_pattern);
    ex quadratic=(quadratic_eq.op(0)-quadratic_eq.op(1)).expand().normal()
        .subs(evaluation_assist,subs_options::algebraic).normal();
    if (float_evaluation)
        quadratic=quadratic.evalf();
    //cerr << "quadratic: "; quadratic.dbgprint();
```

Uses evalf 50a, evaluation_assist 41a 41b, ex 41b 47e 47e 47e 53a, float_evaluation 51b, op 50a, and subs 50a.

We reduce the list of free variables to only present in the quadratic.

90a ⟨figure class 75a⟩+≡ (52a) ◁89f 90b▷
```
lst quadratic_list;
for (size_t i=0; i < free_vars.nops(); ++i)
    if (quadratic.has(free_vars.op(i)))
        quadratic_list.append(free_vars.op(i));
free_vars=ex_to<lst>(quadratic_list);
//cerr << "free_vars which are present: "; free_vars.dbgprint();
```

Uses `nops` 50a and `op` 50a.

We check homogeneity of the quadratic equation.

90b ⟨figure class 75a⟩+≡ (52a) ◁90a 90c▷
```
if (is_homogeneous) {
    ex Q=quadratic;
    for (size_t i=1; i < free_vars.nops(); ++i)
        Q=Q.subs(free_vars.op(i)==free_vars.op(0));
    is_homogeneous &= (Q.degree(free_vars.op(0))==Q.ldegree(free_vars.op(0)));
}
//cerr << "Quadratic part is homogeneous: " << is_homogeneous << endl;
```

Uses `ex` 41b 47e 47e 47e 53a, `nops` 50a, `op` 50a, and `subs` 50a.

The equation may be linear for a particular free variable, we will search if it is.

90c ⟨figure class 75a⟩+≡ (52a) ◁90b 90d▷
```
bool is_quadratic=true;
exmap flat_var_em, var1_em, var2_em;
ex flat_var, var1, var2;
```

Uses `ex` 41b 47e 47e 47e 53a.

We now search if for some free variable the equation is linear

90d ⟨figure class 75a⟩+≡ (52a) ◁90c 90e▷
```
size_t i=0;
for (; i < free_vars.nops(); ++i) {
    //cerr << "degree: " << quadratic.degree(free_vars.op(i)) << endl;
    if (quadratic.degree(free_vars.op(i)) < 2) {
        is_quadratic=false;
        //cerr << "Equation is linear in "; free_vars.op(i).dbgprint();
        break;
    }
}
```

Uses `nops` 50a and `op` 50a.

If all equations are quadratic in any variable, we use homogenuity to reduce the last free variable.

90e ⟨figure class 75a⟩+≡ (52a) ◁90d 91a▷
```
if (is_quadratic) {
    if (is_homogeneous && free_vars.nops() > 1) {
        exmap erase_var;
        erase_var.insert(std::make_pair(free_vars.op(free_vars.nops()-1), numeric(1)));
        subs_lst1=ex_to<lst>(subs_lst1.subs(erase_var,
                                            subs_options::algebraic | subs_options::no_pattern));
        subs_lst1.append(free_vars.op(free_vars.nops()-1) == numeric(1));
        quadratic=quadratic.subs(free_vars.op(free_vars.nops()-1) == numeric(1));
        free_vars.remove_last();
        //cerr << "Quadratic reduced by homogenuity: "; quadratic.dbgprint();
    }
```

Uses `nops` 50a, `numeric` 22d, `op` 50a, and `subs` 50a.

and then proceed with solving of quadratic equation for each free variable attempting to find root-free presentation.

91a ⟨figure class 75a⟩+≡ (52a) ◁90e 91b▷
```
ex A, B, C, D, sqrtD;
for(i=0; i < free_vars.nops(); ++i) {
    A=quadratic.coeff(free_vars.op(i),2).normal();
    //cerr << "A: "; A.dbgprint();
    B=quadratic.coeff(free_vars.op(i),1);
    C=quadratic.coeff(free_vars.op(i),0);
    D=(pow(B,2)-numeric(4)*A*C).normal();
    sqrtD=sqrt(D);
    //cerr << "D: "; D.dbgprint();
```
Uses ex 41b 47e 47e 47e 53a, nops 50a, numeric 22d, and op 50a.

For the condition of real coefficients, we are checking whether another free variable survived in the discriminant of the quadratic equation.
TODO: this process need to be recursive for all free variables, not just for one as it is now.

91b ⟨figure class 75a⟩+≡ (52a) ◁91a 91c▷
```
    if (//need_reals &&
        free_vars.nops()>1) {
        int another=0;
        if (i==0)
            another=1;
```
Uses nops 50a.

If another free variable, denoted x here, presents in the discriminant $D = A_1x^2 + B_1x + C_1$, we try some hyperbolic or trigonometric substitutions.

91c ⟨figure class 75a⟩+≡ (52a) ◁91b 91d▷
```
        if (not is_less_than_epsilon(D) ∧ D.has(free_vars.op(another))) {
            ex A1=D.coeff(free_vars.op(another),2)
            .subs(evaluation_assist,subs_options::algebraic).normal(),
            B1=D.coeff(free_vars.op(another),1)
            .subs(evaluation_assist,subs_options::algebraic).normal(),
            C1=D.coeff(free_vars.op(another),0)
            .subs(evaluation_assist,subs_options::algebraic).normal(),
            D1=(pow(B1,2)-4*A1*C1).normal();
            //cerr << "Atempt to resolve square root for A1=" << A1;
            //cerr << ", B1=" << B1 << ", C1=" << C1 << ", D1=" << D1 << endl;
```
Uses evaluation_assist 41a 41b, ex 41b 47e 47e 47e 53a, is_less_than_epsilon 53b, op 50a, and subs 50a.

If the expression is linear, we make a substitution $D = B_1x + C_1 = y^2$, thus $x = (y^2 - C_1)/B_1$.

91d ⟨figure class 75a⟩+≡ (52a) ◁91c 91e▷
```
            if (is_less_than_epsilon(A1) ∧ not is_less_than_epsilon(B1)) {
                ex y=realsymbol(),
                x=(pow(y,2)-C1)÷B1;
                sqrtD=y;
                flat_var_em.insert(std::make_pair(free_vars.op(another), x));
                flat_var=(free_vars.op(another)≡x);
```
Uses ex 41b 47e 47e 47e 53a, is_less_than_epsilon 53b, op 50a, and realsymbol 27c.

If A_1 is positive, then the substitution depends on sign of the second discriminant $D_1 = B_1^2 - 4A_1C_1$

91e ⟨figure class 75a⟩+≡ (52a) ◁91d 92a▷
```
            } else if (A1.evalf().info(info_flags::positive)) {
```
Uses evalf 50a and info 50a.

Depending on the sign of D_1 and thus $C_1 - B_1^2/(4A_1)$ we are using either hyperbolic sine or cosine.

92a ⟨figure class 75a⟩+≡ (52a) ◁91e 92b▷
```
if (D1.info(info_flags::negative)) {
    ex y=realsymbol(),
    x=(sinh(y)*sqrt(-D1)-B1)÷2÷A1;
    sqrtD=sqrt(C1-pow(B1,2)÷4÷A1)*cosh(y);
    flat_var_em.insert(std::make_pair(free_vars.op(another), x));
    flat_var=(free_vars.op(another)≡x);
} else if (D1.info(info_flags::positive)) {
    ex y=realsymbol(),
    x=(cosh(y)*sqrt(D1)-B1)÷2÷A1;
    sqrtD=sqrt(pow(B1,2)÷4÷A1-C1)*sinh(y);
    flat_var_em.insert(std::make_pair(free_vars.op(another), x));
    flat_var=(free_vars.op(another)≡x);
}
```

Uses `ex` 41b 47e 47e 47e 53a, `info` 50a, `op` 50a, and `realsymbol` 27c.

If A_1 is negative and $C_1 - B_1^2/(4A_1) > 0$ we use the trigonometric substitution $(2A_1x + B_1)/\sqrt{4A_1C_1 - B_1^2} = \cos y$.

92b ⟨figure class 75a⟩+≡ (52a) ◁92a 92c▷
```
} else if (A1.evalf().info(info_flags::negative)) {
    if (D1.info(info_flags::negative)) {
        ex y=realsymbol(),
        x=(sin(y)*sqrt(-D1)-B1)÷2÷A1;
        sqrtD=sqrt(-C1+pow(B1,2)÷4÷A1)*cos(y);
        flat_var_em.insert(std::make_pair(free_vars.op(another), x));
        flat_var=(free_vars.op(another)≡x);
```

Uses `evalf` 50a, `ex` 41b 47e 47e 47e 53a, `info` 50a, `op` 50a, and `realsymbol` 27c.

If both are negative, we explicitly take out the imaginary part and use the above hyperbolic substitution with sinh.

92c ⟨figure class 75a⟩+≡ (52a) ◁92b 92d▷
```
    } else if (D1.info(info_flags::positive)) {
        ex y=realsymbol(),
        x=(sinh(y)*I*sqrt(D1)-B1)÷2÷A1;
        sqrtD=I*sqrt(C1-pow(B1,2)÷4÷A1)*cosh(y);
        flat_var_em.insert(std::make_pair(free_vars.op(another), x));
        flat_var=(free_vars.op(another)≡x);
    }
}
```

Uses `ex` 41b 47e 47e 47e 53a, `info` 50a, `op` 50a, and `realsymbol` 27c.

If a substitution was found we are staying with this solution.

92d ⟨figure class 75a⟩+≡ (52a) ◁92c 92e▷
```
            //cerr << "real_only sqrt(D): "; sqrtD.dbgprint();
            if (not (sqrtD-sqrt(D)).is_zero())
                break;
        }
    }
}
```

Put index back to the range if needed.

92e ⟨figure class 75a⟩+≡ (52a) ◁92d 93a▷
```
if (i ≡ free_vars.nops())
    --i;
```

Uses `nops` 50a.

Small perturbations of the zero determinant can create the unwanted imaginary entries, thus we treat it as exactly zero. Also negligibly small A corresponds to an effectively linear equation.

93a ⟨figure class 75a⟩+≡ (52a) ◁92e 93b▷

```
if (is_less_than_epsilon(D) ∨ ((¬ is_less_than_epsilon(B)) ∧ is_less_than_epsilon(A÷B))) {
    if (is_less_than_epsilon(D)) {
        //cerr << "zero determinant" << endl;
        var1=(-B÷numeric(2)÷A).subs(flat_var_em,subs_options::algebraic
                                    | subs_options::no_pattern).normal();
    } else {
        //cerr << "almost linear equation" << endl;
        var1=(-C÷B).subs(flat_var_em,subs_options::algebraic
                         | subs_options::no_pattern).normal();
    }
    var1_em.insert(std::make_pair(free_vars.op(i), var1));
    subs_lst1=ex_to<lst>(subs_lst1
                    .subs(var1_em,subs_options::algebraic | subs_options::no_pattern));
    subs_lst1=ex_to<lst>(subs_lst1.append(free_vars.op(i) ≡ var1)
                    .subs(flat_var_em,subs_options::algebraic | subs_options::no_pattern));
    if (flat_var.nops()>0)
        subs_lst1.append(flat_var);
    //cerr << "subs_lst1a: "; subs_lst1.dbgprint();
```

Uses `is_less_than_epsilon` 53b, `nops` 50a, `numeric` 22d, `op` 50a, and `subs` 50a.

For a non-zero discriminant we generate two solutions of the quadratic equation.

93b ⟨figure class 75a⟩+≡ (52a) ◁93a 93c▷

```
} else {
    second_solution=true;
    subs_lst2=subs_lst1;
    var1=((-B+sqrtD)÷numeric(2)÷A).subs(flat_var_em,subs_options::algebraic
                                        | subs_options::no_pattern).normal();
    var1_em.insert(std::make_pair(free_vars.op(i), var1));
    var2=((-B-sqrtD)÷numeric(2)÷A).subs(flat_var_em,subs_options::algebraic
                                        | subs_options::no_pattern).normal();
    var2_em.insert(std::make_pair(free_vars.op(i), var2));
    subs_lst1=ex_to<lst>(subs_lst1
                    .subs(var1_em,subs_options::algebraic | subs_options::no_pattern));
    subs_lst1=ex_to<lst>(subs_lst1.append(free_vars.op(i) ≡ var1)
                    .subs(flat_var_em,subs_options::algebraic | subs_options::no_pattern));
```

Uses `numeric` 22d, `op` 50a, and `subs` 50a.

Then we modify the second substitution list accordingly.

93c ⟨figure class 75a⟩+≡ (52a) ◁93b 93d▷

```
    subs_lst2=ex_to<lst>(subs_lst2
                    .subs(var2_em,subs_options::algebraic | subs_options::no_pattern));
    subs_lst2=ex_to<lst>(subs_lst2.append(free_vars.op(i) ≡ var2)
                    .subs(flat_var_em,subs_options::algebraic | subs_options::no_pattern));
```

Uses `op` 50a and `subs` 50a.

We need to add the values of *flat_var* which were assigned the numeric value.

93d ⟨figure class 75a⟩+≡ (52a) ◁93c 94a▷

```
    if (flat_var.nops()>0) {
        subs_lst1.append(flat_var);
        subs_lst2.append(flat_var);
    }
    //cerr << "subs_lst1b: "; subs_lst1.dbgprint();
    //cerr << "subs_lst2b: "; subs_lst2.dbgprint();
}
// end of the quadratic case
```

Uses `nops` 50a.

The non-linear equation is not quadratic in some variable, e.g. is $mk + 1 = 0$ then we are solving it as linear.

94a ⟨figure class 75a⟩+≡ (52a) ◁93d 94b▷

```
    } else {
        //cerr << "The equation is not quadratic in a single variable"<<endl;
        //cerr << "free_vars: "; free_vars.dbgprint();
        var1=-(quadratic.coeff(free_vars.op(i),0)÷quadratic.coeff(free_vars.op(i),1)).normal();
        var1_em.insert(std::make_pair(free_vars.op(i), var1));
        subs_lst1=ex_to<lst>(subs_lst1
                    .subs(var1_em,subs_options::algebraic | subs_options::no_pattern));
        subs_lst1.append(free_vars.op(i) ≡ var1);
        //cerr << "non-quadratic subs_lst1: "; subs_lst1.dbgprint();
    }
```

Uses op 50a and subs 50a.

Now we check that other non-linear conditions are satisfied by the found solutions.

94b ⟨figure class 75a⟩+≡ (52a) ◁94a 94c▷

```
        lst::const_iterator it1= nonlin_cond.begin();
        ++it1;
        //cerr << "Subs list: "; subs_lst1.dbgprint();
        lst subs_f1=ex_to<lst>(subs_lst1.evalf()), subs_f2;
        //cerr << "Subs list float: "; subs_f1.dbgprint();
        if(second_solution)
            subs_f2=ex_to<lst>(subs_lst2.evalf());
```

Uses evalf 50a.

Since CAS is not as perfect as one may wish, we checked obtained solutions in two ways: through float approximations and exact calculations. If either works then the solution is accepted.

94c ⟨figure class 75a⟩+≡ (52a) ◁94b 94d▷

```
        for (; it1 ≠ nonlin_cond.end(); ++it1) {
            first_solution &= (is_less_than_epsilon((it1→op(0)-it1→op(1)).subs(subs_f1,
                                        subs_options::algebraic | subs_options::no_pattern))
                    ∨ ((it1→op(0)-it1→op(1)).subs(subs_lst1,
                                subs_options::algebraic | subs_options::no_pattern)).normal().is_zero());
```

Uses is_less_than_epsilon 53b, op 50a, and subs 50a.

The same check for the second solution.

94d ⟨figure class 75a⟩+≡ (52a) ◁94c 94e▷

```
            if(second_solution)
                second_solution &= (is_less_than_epsilon((it1→op(0)-it1→op(1)).subs(subs_f2,
                                        subs_options::algebraic | subs_options::no_pattern))
                    ∨ ((it1→op(0)-it1→op(1)).subs(subs_lst2,
                                subs_options::algebraic | subs_options::no_pattern)).normal().is_zero());
        }
```

Uses is_less_than_epsilon 53b, op 50a, and subs 50a.

If a solution is good, then we use it to generate the respective cycle.

94e ⟨figure class 75a⟩+≡ (52a) ◁94d 95a▷

```
        if (first_solution)
            C_new=symbolic.subs(subs_lst1, subs_options::algebraic
                                | subs_options::no_pattern);

        //cerr << "C_new: "; C_new.dbgprint();
        if (second_solution)
            C1_new=symbolic.subs(subs_lst2, subs_options::algebraic
                                | subs_options::no_pattern);
        //cerr << "C1_new: "; C1_new.dbgprint();
    }
```

Uses subs 50a.

We check if any symbols survived after calculations...

95a ⟨figure class 75a⟩+≡ (52a) ◁94e 95b▷
```
lst repl;
if (ex_to<cycle_data>(C_new).has(ex_to<cycle_data>(symbolic).get_k()))
    repl.append(ex_to<cycle_data>(symbolic).get_k()≡realsymbol());
if (ex_to<cycle_data>(C_new).has(ex_to<cycle_data>(symbolic).get_m()))
    repl.append(ex_to<cycle_data>(symbolic).get_m()≡realsymbol());
if (ex_to<cycle_data>(C_new).has(ex_to<cycle_data>(symbolic).get_l().op(0).op(0)))
    repl.append(ex_to<cycle_data>(symbolic).get_l().op(0).op(0)≡realsymbol());
if (ex_to<cycle_data>(C_new).has(ex_to<cycle_data>(symbolic).get_l().op(0).op(1)))
    repl.append(ex_to<cycle_data>(symbolic).get_l().op(0).op(1)≡realsymbol());
```

Uses `cycle_data` 23a 26a 42c 43a 54c 54d 55a 56b 56c 56d, `op` 50a, and `realsymbol` 27c.

...and if they are, then we replace them for new one

95b ⟨figure class 75a⟩+≡ (52a) ◁95a 95c▷
```
if (repl.nops()>0) {
    if (first_solution)
        C_new=C_new.subs(repl);
    if (second_solution)
        C1_new=C1_new.subs(repl);
}

//cerr << endl;
```

Uses `nops` 50a and `subs` 50a.

Finally, every constructed cycle is added to the result.

95c ⟨figure class 75a⟩+≡ (52a) ◁95b 95d▷
```
    lst res;
    if (first_solution)
        res.append(float_evaluation? C_new.num_normalize().evalf():C_new.num_normalize());
    if (second_solution)
        res.append(float_evaluation? C1_new.num_normalize().evalf():C1_new.num_normalize());

    return res;
}
```

Uses `evalf` 50a and `float_evaluation` 51b.

This method runs recursively because we do not know in advance the number of conditions glued by and/or. Also, some relations (e.g. *moebius_trans* or **subfigure**) directly define the cycles, and for others we need to solve some equations.

95d ⟨figure class 75a⟩+≡ (52a) ◁95c 95e▷
```
ex figure::update_cycle_node(const ex & key, const lst & eq_cond, const lst & neq_cond, lst res, size_t level)
{
    //cerr << endl << "level: " << level << "; cycle: "; nodes[key].dbgprint();
    if (level ≡ 0) {// set the iniail symbolic cycle for calculations
        ⟨update node zero level 97b⟩
    }
```

Defines:
`update_cycle_node`, used in chunks 81c, 83d, 84b, 86a, 97a, 98a, and 100b.

Uses `ex` 41b 47e 47e 47e 53a, `figure` 16d 22e 32a 32c 38b 49a 50d 75a 80a 82b 82c 85a 86c 98c 99b 99d 100a 101a 103b 104a 105c 106c 106d 107a 109a 109c 110a, `key` 32e, and `nodes` 51a.

If we get here, then some equations need to be solved. We advance through the parents list to match the *level*.

95e ⟨figure class 75a⟩+≡ (52a) ◁95d 96a▷
```
lst par = nodes[key].get_parents();
lst::const_iterator it = par.begin();
std::advance(it,level);

lst new_cond=ex_to<lst>(ex_to<cycle_relation>(*it).rel_to_parent(nodes[key].get_cycles_data().op(0),
                                                                point_metric, cycle_metric, nodes));
```

Uses `cycle_metric` 50f, `cycle_relation` 40c 45e 46c 60d 61 62a 62b 64a 64b, `key` 32e, `nodes` 51a, `op` 50a, and `point_metric` 50f.

We need to go through the cycle at least once at every *level* and separate equations, which are used to calculate solutions, from inequalities, which will be only checked on the obtained solution.

96a ⟨figure class 75a⟩+≡ (52a) ◁95e 96b▷
```
  for (const auto& it1 : new_cond) {
    lst store_cond=neq_cond;
    lst use_cond=eq_cond;
    lst step_cond=ex_to<lst>(it1);
```

Iteration over the list of conditions

96b ⟨figure class 75a⟩+≡ (52a) ◁96a 96c▷
```
    for (const auto& it2 : step_cond)
      if ((is_a<relational>(it2) ∧ ex_to<relational>(it2).info(info_flags::relation_equal)))
        use_cond.append(it2);   // append the equation
      else if (is_a<cycle>(it2)) { // append a solution
        cycle Cnew=ex_to<cycle>(it2);
        res.append(cycle_data(Cnew.get_k(), Cnew.get_l().subs(Cnew.get_l().op(1)≡key),
                              Cnew.get_m()));
      } else
        store_cond.append(*it); // store the pointer to parents producing inequality
    //cerr << "use_cond: "; use_cond.dbgprint();
    //cerr << "store_cond: "; store_cond.dbgprint();
```

Uses `cycle_data` 23a 26a 42c 43a 54c 54d 55a 56b 56c 56d, `info` 50a, `key` 32e, `op` 50a, and `subs` 50a.

When all conditions are unwrapped and there are equations to solve, we call a solver. Solutions from *res* are copied there as well, then *res* is cleared.

96c ⟨figure class 75a⟩+≡ (52a) ◁96b 96d▷
```
    if(level ≡ par.nops()-1) { //if the last one in the parents list
      lst cnew;
      if (use_cond.nops()>0)
        cnew=ex_to<lst>(evaluate_cycle(nodes[key].get_cycle_data(0), use_cond));
      for (const auto& sol : res)
        cnew.append(sol);
      res=lst{};
```

Uses `evaluate_cycle` 87a, `key` 32e, `nodes` 51a, and `nops` 50a.

Now we check which of the obtained solutions satisfy to the restrictions in *store_cond*

96d ⟨figure class 75a⟩+≡ (52a) ◁96c 97a▷
```
      //cerr<< "Store cond: "; store_cond.dbgprint();
      //cerr<< "Use cond: "; use_cond.dbgprint();
      for (const auto& inew: cnew) {
        bool to_add=true;
        for (const auto& icon: store_cond) {
          lst suits=ex_to<lst>(ex_to<cycle_relation>(icon).rel_to_parent(inew,
                                                    point_metric, cycle_metric, nodes));
          //cerr<< "Suit: "; suits.dbgprint();
          for (const auto& is : suits)
            for (const auto& ic : is) {
```

Uses `cycle_metric` 50f, `cycle_relation` 40c 45e 46c 60d 61 62a 62b 64a 64b, `nodes` 51a, and `point_metric` 50f.

Two possibilities to check: either a **false** relational or a number close to zero.

97a ⟨figure class 75a⟩+≡ (52a) ◁96d 98a▷
```
                    if (is_a<relational>(ic)) {
                        if (¬(bool)ex_to<relational>(ic))
                            to_add=false;
                    } else if (is_less_than_epsilon(ic))
                        to_add=false;
                }
                if (¬ to_add)
                    break;
            }
            if (to_add)
                res.append(inew);
        }
    //cerr<< "Result: "; res.dbgprint();
    } else
        res=ex_to<lst>(update_cycle_node(key, use_cond, store_cond, res, level+1));
  }
  if (level ≡0)
      return unique_cycle(res);
  else
      return res;
}
```

Uses `is_less_than_epsilon` 53b, `key` 32e, `unique_cycle` 40f 119a, and `update_cycle_node` 49b 95d.

If the cycle is defined by by a **subfigure** all calculations are done within it.

97b ⟨update node zero level 97b⟩≡ (95d) 97c▷
```
  if ( nodes[key].get_parents().nops() ≡ 1 ∧ is_a<subfigure>(nodes[key].get_parents().op(0))) {
      figure F=ex_to<figure>(ex_to<basic>(ex_to<subfigure>(nodes[key].get_parents().op(0)).get_subf())
                      .clearflag(status_flags::expanded));
      F=float_evaluation? F.set_float_eval(): F.set_exact_eval();
```

Uses `figure` 16d 22e 32a 32c 38b 49a 50d 75a 80a 82b 82c 85a 86c 98c 99b 99d 100a 101a 103b 104a 105c 106c 106d 107a 109a 109c 110a, `float_evaluation` 51b, `key` 32e, `nodes` 51a, `nops` 50a, `op` 50a, `set_exact_eval` 37c, `set_float_eval` 37c, and `subfigure` 40d 48b 48d 66a 66b 66c 66d 66e.

We replace parameters of the **subfigure** by current parents and evaluate the result.

97c ⟨update node zero level 97b⟩+≡ (95d) ◁97b 97d▷
```
      lst parkeys=ex_to<lst>(ex_to<subfigure>(nodes[key].get_parents().op(0)).get_parlist());
      unsigned int var=0;
      char name[12];
      for (const auto& it : parkeys) {
          sprintf(name, "variable%03d", var);
          F.set_cycle(F.get_cycle_key(name), nodes[it].get_cycles_data());
          ++var;
      }
      F.set_metric(point_metric,cycle_metric); // this calls automatic figure re-calculation
      return F.get_cycles(F.get_cycle_key("result"));
```

Uses `cycle_metric` 50f, `figure` 16d 22e 32a 32c 38b 49a 50d 75a 80a 82b 82c 85a 86c 98c 99b 99d 100a 101a 103b 104a 105c 106c 106d 107a 109a 109c 110a, `get_cycle_key`, `key` 32e, `name` 32e, `nodes` 51a, `op` 50a, `point_metric` 50f, `set_cycle` 49b 82b, `set_metric` 32b 98c, and `subfigure` 40d 48b 48d 66a 66b 66c 66d 66e.

For a list of relations we simply set up a symbolic cycle and proceed with calculations in recursion.

97d ⟨update node zero level 97b⟩+≡ (95d) ◁97c
```
  } else
   nodes[key].set_cycles(cycle_data(k, indexed(matrix(1, ex_to<numeric>(get_dim()).to_int(), l), varidx(key, ex_to<nun
```

Uses `cycle_data` 23a 26a 42c 43a 54c 54d 55a 56b 56c 56d, `get_dim()` 35c, `k` 51c, `key` 32e, `l` 51c, `m` 51c, `nodes` 51a, and `numeric` 22d.

The figure is updated.

98a ⟨figure class 75a⟩+≡ (52a) ◁97a 98b▷
```
figure figure::update_cycles()
{
    if (info(status_flags::expanded))
        return *this;
    lst all_child;
    for (auto& x: nodes)
        if (ex_to<cycle_node>(x.second).get_generation() == 0) {
            if (ex_to<cycle_node>(x.second).get_parents().nops() > 0)
                nodes[x.first].set_cycles(ex_to<lst>(update_cycle_node(x.first)));
```
Defines:
`update_cycles`, used in chunks 98d and 108b.

Uses `cycle_node` 43c 45c 69c 70a 70b 70c 71a 71b 72b 72c 73a 74a 74d, `figure` 16d 22e 32a 32c 38b 49a 50d 75a 80a 82b 82c 85a 86c 98c 99b 99d 100a 101a 103b 104a 105c 106c 106d 107a 109a 109c 110a, `get_generation` 35f, `info` 50a, `nodes` 51a, `nops` 50a, and `update_cycle_node` 49b 95d.

We collect all children of the zero-generation cycles for subsequent update.

98b ⟨figure class 75a⟩+≡ (52a) ◁98a 98c▷
```
            lst ch=ex_to<cycle_node>(x.second).get_children();
            for (const auto& it1 : ch)
                all_child.append(it1);
        }
    all_child.sort();
    all_child.unique();
    update_node_lst(all_child);
    return *this;
}
```
Uses `cycle_node` 43c 45c 69c 70a 70b 70c 71a 71b 72b 72c 73a 74a 74d and `update_node_lst` 50c 100a.

F.5.5. *Additional methods.* Set the new metric for the figure, repeating the previous code from the constructor.

98c ⟨figure class 75a⟩+≡ (52a) ◁98b 98d▷
```
void figure::set_metric(const ex & Mp, const ex & Mc)
{
    ex D=get_dim();
    ⟨set point metric in figure 76a⟩
    ⟨set cycle metric in figure 77a⟩
    ⟨check dimensionalities point and cycle metrics 78c⟩
```
Defines:
`figure`, used in chunks 16–18, 20e, 22, 27b, 28a, 30a, 36, 37, 45, 46c, 48, 50a, 52, 66, 75–84, 86, 87, 95d, 97–100, 106–108, 110, 111, and 117c.
`set_metric`, used in chunks 26 and 97c.

Uses `ex` 41b 47e 47e 47e 53a and `get_dim()` 35c.

We check that the dimensionality of the new metric matches the old one.

98d ⟨figure class 75a⟩+≡ (52a) ◁98c 99a▷
```
    if (¬ (D-get_dim()).is_zero())
        throw(std::invalid_argument("New metric has a different dimensionality!"));
    update_cycles();
}
```
Uses `get_dim()` 35c and `update_cycles` 50d 98a.

The method collects all key for nodes with generations in the range [*intgen*,*maxgen*] inclusively.

99a ⟨figure class 75a⟩+≡ (52a) ◁98d 99b▷

```
ex figure::get_all_keys(const int mingen, const int maxgen) const {
    lst keys;
    for (const auto& x: nodes) {
        if (x.second.get_generation() ≥ mingen ∧
            (maxgen ≡ GHOST_GEN ∨ x.second.get_generation() ≤ maxgen))
            keys.append(x.first);
    }
    return keys;
}
```

Defines:
`get_all_keys`, used in chunks 20a, 105d, and 106d.

Uses `ex` 41b 47e 47e 47e 53a, `figure` 16d 22e 32a 32c 38b 49a 50d 75a 80a 82b 82c 85a 86c 98c 99b 99d 100a 101a 103b 104a 105c 106c 106d 107a 109a 109c 110a, `get_generation` 35f, `GHOST_GEN` 42b 42b, and `nodes` 51a.

The method also collects all key for nodes with generations in the range [*intgen*,*maxgen*] inclusively and sort them according to their generations from smaller to larger.

99b ⟨figure class 75a⟩+≡ (52a) ◁99a 99c▷

```
ex figure::get_all_keys_sorted(const int mingen, const int maxgen) const {
    lst keys;
    int mg=get_max_generation();
    if (maxgen ≠ GHOST_GEN ∧ maxgen < mg)
            mg =maxgen;
    for (int i=mingen; i ≤ mg; ++i)
        for (const auto& x: nodes) {
            if (x.second.get_generation() ≡ i)
                keys.append(x.first);
    }
    return keys;
}
```

Defines:
`get_all_keys_sorted`, never used.

Uses `ex` 41b 47e 47e 47e 53a, `figure` 16d 22e 32a 32c 38b 49a 50d 75a 80a 82b 82c 85a 86c 98c 99b 99d 100a 101a 103b 104a 105c 106c 106d 107a 109a 109c 110a, `get_generation` 35f, `get_max_generation` 49h 99b, `GHOST_GEN` 42b 42b, and `nodes` 51a.

Scanning for the biggest number generation.

99c ⟨figure class 75a⟩+≡ (52a) ◁99b 99d▷

```
int figure::get_max_generation() const {
    int max_gen = REAL_LINE_GEN;
    for (const auto& x: nodes)
        if (x.second.get_generation() > max_gen)
            max_gen = x.second.get_generation();
    return max_gen;
}
```

Defines:
`figure`, used in chunks 16–18, 20e, 22, 27b, 28a, 30a, 36, 37, 45, 46c, 48, 50a, 52, 66, 75–84, 86, 87, 95d, 97–100, 106–108, 110, 111, and 117c.
`get_max_generation`, used in chunk 99a.

Uses `get_generation` 35f, `nodes` 51a, and `REAL_LINE_GEN` 42b 42b.

Return the list of cycles stored in the node with *key*.

99d ⟨figure class 75a⟩+≡ (52a) ◁99c 100a▷
```
ex figure::get_cycles(const ex & key, const ex & metric) const
{
   exhashmap<cycle_node>::const_iterator  cnode=nodes.find(key);
   if (cnode ≡ nodes.end()) {
      if (FIGURE_DEBUG)
         cerr ≪ "There is no key " ≪ key ≪ " in the figure." ≪ endl;
      return lst{};
   } else
      return cnode→second.make_cycles(metric);
}
```
Defines:
`get_cycle`, used in chunks 19d, 21h, 29a, 30d, 43a, 44d, 57b, 63c, 69d, 97c, 101d, 105d, 106d, 110d, and 111d.
Uses `cycle_node` 43c 45c 69c 70a 70b 70c 71a 71b 72b 72c 73a 74a 74d, `ex` 41b 47e 47e 47e 53a, `figure` 16d 22e 32a 32c 38b 49a 50d 75a 80a 82b 82c 85a 86c 98c 99b 99d 100a 101a 103b 104a 105c 106c 106d 107a 109a 109c 110a, `FIGURE_DEBUG` 52c, `key` 32e, and `nodes` 51a.

Full reset of figure to the initial empty state.

100a ⟨figure class 75a⟩+≡ (52a) ◁99d 100b▷
```
void figure::reset_figure()
{
   nodes.clear();
   ⟨set the infinity 75c⟩
   ⟨set the real line 75d⟩
}
```
Defines:
`figure`, used in chunks 16–18, 20e, 22, 27b, 28a, 30a, 36, 37, 45, 46c, 48, 50a, 52, 66, 75–84, 86, 87, 95d, 97–100, 106–108, 110, 111, and 117c.
`reset_figure`, never used.
Uses `nodes` 51a.

Update nodes in the list and all their (grand)children subsequently.

100b ⟨figure class 75a⟩+≡ (52a) ◁100a 100c▷
```
void figure::update_node_lst(const ex & inlist)
{
   if (info(status_flags::expanded))
      return;

   lst intake=ex_to<lst>(inlist);
   while (intake.nops() ≠0) {
      int mingen=nodes[∗intake.begin()].get_generation();
      for (const auto& it : intake)
         mingen=min(mingen, nodes[it].get_generation());
      lst current, future;
      for (const auto& it : intake)
         if (nodes[it].get_generation() ≡ mingen)
            current.append(it);
         else
            future.append(it);
```
Defines:
`figure`, used in chunks 16–18, 20e, 22, 27b, 28a, 30a, 36, 37, 45, 46c, 48, 50a, 52, 66, 75–84, 86, 87, 95d, 97–100, 106–108, 110, 111, and 117c.
`update_node_lst`, used in chunks 83a, 86, and 98b.
Uses `ex` 41b 47e 47e 47e 53a, `get_generation` 35f, `info` 50a, `nodes` 51a, and `nops` 50a.

All nodes at the current list are updated.

100c ⟨figure class 75a⟩+≡ (52a) ◁100b 100d▷
```
      for (const auto& it : current) {
         nodes[it].set_cycles(ex_to<lst>(update_cycle_node(it)));
         lst nchild=nodes[it].get_children();
         for (const auto& it1 : nchild)
            future.append(it1);
      }
```
Uses `nodes` 51a and `update_cycle_node` 49b 95d.

Future list becomes new intake.

100d ⟨figure class 75a⟩+≡ (52a) ◁100c 101a▷
```
        intake=future;
        intake.sort();
        intake.unique();
    }
}
```

Find a symbolic key for a cycle labelled by a *name*.

101a ⟨figure class 75a⟩+≡ (52a) ◁100d 101c▷
```
ex figure::get_cycle_key(string name) const
{
    for (const auto& x: nodes)
        if (ex_to<symbol>(x.first).get_name() ≡ name)
            return x.first;

    return 0;
}
```
Defines:
get_cycle_key, never used.
Uses ex 41b 47e 47e 47e 53a, figure 16d 22e 32a 32c 38b 49a 50d 75a 80a 82b 82c 85a 86c 98c 99b 99d 100a 101a 103b 104a 105c 106c 106d 107a 109a 109c 110a, name 32e, and nodes 51a.

F.5.6. *Drawing methods.* Drawing the figure is possible only in two dimensions, thus we check this at the start.

101c ⟨figure class 75a⟩+≡ (52a) ◁101a 101d▷
```
void figure::asy_draw(ostream & ost, ostream & err, const string picture,
                    const ex & xmin, const ex & xmax, const ex & ymin, const ex & ymax,
                    asy_style style, label_string lstring, bool with_realline,
                    bool with_header, int points_per_arc, const string imaginary_options,
                    bool with_labels) const
{
    ⟨check that dimensionality is 2 101b⟩
```
Defines:
asy_draw, used in chunks 25e and 102–104.
figure, used in chunks 16–18, 20e, 22, 27b, 28a, 30a, 36, 37, 45, 46c, 48, 50a, 52, 66, 75–84, 86, 87, 95d, 97–100, 106–108, 110, 111, and 117c.
Uses asy_style 51d, ex 41b 47e 47e 47e 53a, and label_string 51e.

101b ⟨check that dimensionality is 2 101b⟩≡ (101c 104c 105a)
```
    if (¬ (get_dim()-2).is_zero())
        throw logic_error("Drawing is possible for two-dimensional figures only!");
```

Uses get_dim() 35c.

We will need to place different types of cycle into the different places of the Asymptote file.

101d ⟨figure class 75a⟩+≡ (52a) ◁101c 101e▷
```
    stringstream preamble_stream, main_stream, labels_stream;
    string dots;
    std::regex re("dot\\(");
```

Some bits will depend on the metric in the point space.

101e ⟨figure class 75a⟩+≡ (52a) ◁101d 102a▷

```
int point_metric_signature=ex_to<numeric>(ex_to<clifford>(point_metric).get_metric(idx(0,2),idx(0,2))
        *ex_to<clifford>(point_metric).get_metric(idx(1,2),idx(1,2)).eval()).to_int();

    for (const auto& x: nodes) {
        lst cycles=ex_to<lst>(x.second.make_cycles(point_metric));
        bool first_dot=true;

        for (const auto& it1 : cycles)
            try {
                if ( (x.second.get_generation() > REAL_LINE_GEN) ∨
                    ((x.second.get_generation() ≡ REAL_LINE_GEN) ∧ with_realline)) {
                    stringstream sstr;
                    if (with_header)
                        sstr ≪ "// label: " ≪ (x.first) ≪ endl;
```

Uses get_generation 35f, nodes 51a, numeric 22d, point_metric 50f, and REAL_LINE_GEN 42b 42b.

Produce the coulour and style for the cycle.

102a ⟨figure class 75a⟩+≡ (52a) ◁101e 102b▷

```
                lst colours=lst{0,0,0};
                string asy_opt;
                if (x.second.custom_asy≡"") {
                    asy_opt=style(x.first, (it1), colours);
                } else
                    asy_opt=x.second.custom_asy;
```

Zero-radius cycles are treated specially, its centre become known to Asymptote as a *pair*.

102b ⟨figure class 75a⟩+≡ (52a) ◁102a 102c▷

```
            if (is_less_than_epsilon(ex_to<cycle>(it1).det())) {
                double x1=ex_to<numeric>(ex_to<cycle>(it1).center(cycle_metric).op(0)
                            .evalf()).to_double(),
                    y1=ex_to<numeric>(ex_to<cycle>(it1).center(cycle_metric).op(1)
                            .evalf()).to_double();
                string var_name=regex_replace(ex_to<symbol>(x.first).get_name(), regex("[[:space:]]+"), "_");
                if (first_dot) {
                    preamble_stream ≪ "// label: " ≪ (x.first) ≪ endl
                            ≪ "pair[] " ≪ var_name ≪ "={";
                    first_dot = false;
                } else
                    preamble_stream ≪ ", ";

                preamble_stream ≪ "(" ≪ x1 ≪ "," ≪ y1 ≪ ")";
```

Uses cycle_metric 50f, evalf 50a, is_less_than_epsilon 53b, numeric 22d, and op 50a.

In the elliptic case we place the dot explicitly. . .

102c ⟨figure class 75a⟩+≡ (52a) ◁102b 102d▷

```
                if (point_metric_signature > 0
                    ∧ xmin ≤ x1 ∧ x1≤xmax ∧ ymin ≤ y1 ∧ y1≤ymax) {
                    sstr ≪ "dot(" ≪ var_name
                        ≪ (asy_opt≡""? "": ", ") ≪ asy_opt
                        ≪ ");" ≪ endl;
```

. . . , otherwise output is handled by the **cycle2D**::*draw_asy* method

102d ⟨figure class 75a⟩+≡ (52a) ◁102c 102e▷

```
                } else {
                    ex_to<cycle2D>(it1).asy_draw(sstr, picture, xmin, xmax,
                            ymin, ymax, colours, asy_opt, with_header, points_per_arc, imaginary_options);
```

Uses asy_draw 36b 36b 101a.

Since in parabolic spaces zero-radius cycles are detached from the their centres, which they denote we wish to have a hint on centres positions.

102e ⟨figure class 75a⟩+≡ (52a) ◁ 102d 102f ▷

```
        if (FIGURE_DEBUG ∧ point_metric_signature≡0
          ∧ xmin ≤ x1 ∧ x1≤xmax ∧ ymin ≤ y1 ∧ y1≤ymax)
          sstr ≪ "dot(" ≪ var_name ≪ ", black+3pt);" ≪ endl;
      }
```

Uses FIGURE_DEBUG 52c.

Drawing a generic cycle through **cycle2D**::*draw_asy* method

102f ⟨figure class 75a⟩+≡ (52a) ◁ 102e 103a ▷

```
    } else
      ex_to<cycle2D>(it1).asy_draw(sstr, picture, xmin, xmax,
                      ymin, ymax, colours, asy_opt, with_header, points_per_arc, imaginary_options);
```

Uses asy_draw 36b 36b 101a.

Dots and label will be drawn last to avoid over-painting.

103a ⟨figure class 75a⟩+≡ (52a) ◁ 102f 103b ▷

```
      if (std::regex_search(sstr.str(), re))
        dots+=sstr.str();
      else
        main_stream ≪ sstr.str();
```

Find the label position

103b ⟨figure class 75a⟩+≡ (52a) ◁ 103a 103c ▷

```
      if (with_labels)
        labels_stream ≪ lstring(x.first, (it1), sstr.str());
      }
    } catch (exception &p) {
      if (FIGURE_DEBUG)
        err ≪ "Failed to draw " ≪ x.first ≪": " ≪ x.second;
    }
```

Uses FIGURE_DEBUG 52c.

We do not forget to close the array of dots if any were printed.

103c ⟨figure class 75a⟩+≡ (52a) ◁ 103b 103d ▷

```
    if (¬ first_dot)
      preamble_stream ≪ "};" ≪ endl;
  }
  //cerr << "Dots: " << dots;
```

We record *info_text* as a comment to start the Asymptote file. We try to replace possible end-of-comment symbols.

103d ⟨figure class 75a⟩+≡ (52a) ◁ 103c 104a ▷

```
  ost  ≪ "/*" ≪ endl
    ≪ std::regex_replace(info_text, std::regex("\\*/"), "* /") ≪ endl
    ≪ "*/" ≪ endl;
```

Uses info_text.

If dots were output, we produce an auxiliary function, which labels an array of points.

104a ⟨figure class 75a⟩+≡ (52a) ◁ 103d 104b ▷

```
  if (preamble_stream.str() ≠ "")
    ost ≪ "// An auxiliary function" ≪ endl
      ≪ "void label(string L, pair[] P, pair D) {" ≪ endl
      ≪ "  for(pair k : P)" ≪ endl
      ≪ "    label(L, k, D);" ≪ endl
      ≪ "}" ≪ endl
      ≪ preamble_stream.str();
```

Uses k 51c.

Finally, we output the rest of drawings.

104b ⟨figure class 75a⟩+≡ (52a) ◁104a 104c▷

```
ost ≪ main_stream.str()
   ≪ dots
   ≪ labels_stream.str();
}
```

104c ⟨figure class 75a⟩+≡ (52a) ◁104b 104d▷

```
void figure::asy_write(int size, const ex & xmin, const ex & xmax, const ex & ymin, const ex & ymax,
                    string name, string format,
                    asy_style style, label_string lstring, bool with_realline,
                    bool with_header, int points_per_arc, const string imaginary_options,
                    bool rm_asy_file, bool with_labels) const
{
    ⟨check that dimensionality is 2 101b⟩
```

Defines:
`asy_write`, used in chunks 25, 26, and 29c.
`figure`, used in chunks 16–18, 20e, 22, 27b, 28a, 30a, 36, 37, 45, 46c, 48, 50a, 52, 66, 75–84, 86, 87, 95d, 97–100, 106–108, 110, 111, and 117c.

Uses `asy_style` 51d, `ex` 41b 47e 47e 47e 53a, `label_string` 51e, and `name` 32e.

Open the file.

104d ⟨figure class 75a⟩+≡ (52a) ◁104c 104e▷

```
string filename=name+".asy";
ofstream out(filename);
out ≪ "size(" ≪ size ≪ ");" ≪ endl;
asy_draw(out, cerr, "", xmin, xmax, ymin, ymax,
      style, lstring, with_realline, with_header, points_per_arc, imaginary_options, with_labels);
if (name ≡ "")
   out ≪ "shipout();" ≪ endl;
else
   out ≪ "shipout(\"" ≪ name ≪ "\");" ≪ endl;
out.flush();
out.close();
```

Uses `asy_draw` 36b 36b 101a and `name` 32e.

Preparation of `Asymptote` call.

104e ⟨figure class 75a⟩+≡ (52a) ◁104d 105a▷

```
char command[256];
strcpy(command, show_asy_graphics? "asy -V" : "asy");
if (format ≠ "") {
   strcat(command, " -f ");
   strcat(command, format.c_str());
}
strcat(command, " ");
strcat(command, name.c_str());
char * pcommand=command;
system(pcommand);
if (rm_asy_file)
   remove(filename.c_str());
}
```

Uses `name` 32e and `show_asy_graphics` 52d.

This method animates figures with parameters.

105a ⟨figure class 75a⟩+≡ (52a) ◁104e 105b▷

```
  void figure::asy_animate(const ex &val,
                   int size, const ex & xmin, const ex & xmax, const ex & ymin, const ex & ymax,
                   string name, string format, asy_style style, label_string lstring, bool with_realline,
                   bool with_header, int points_per_arc, const string imaginary_options,
                   const string values_position, bool rm_asy_file, bool with_labels) const
  {
     ⟨check that dimensionality is 2 101b⟩
     string filename=name+".asy";
     ofstream out(filename);
```

Defines:
  `asy_animate`, used in chunk 26e.
  `figure`, used in chunks 16–18, 20e, 22, 27b, 28a, 30a, 36, 37, 45, 46c, 48, 50a, 52, 66, 75–84, 86, 87, 95d, 97–100, 106–108, 110, 111, and 117c.

Uses `asy_style` 51d, `ex` 41b 47e 47e 47e 53a, `label_string` 51e, and `name` 32e.

Header of the file depends from format.

105b ⟨figure class 75a⟩+≡ (52a) ◁105a 105c▷

```
     if (format ≡ "pdf")
        out ≪ "settings.tex=\"pdflatex\";" ≪ endl
           ≪ "settings.embed=true;" ≪ endl
           ≪ "import animate;" ≪ endl
           ≪ "size(" ≪ size ≪ ");" ≪ endl
           ≪ "animation a=animation(\"" ≪ name ≪ "\");" ≪ endl;
     else
        out ≪ "import animate;" ≪ endl
           ≪ "size(" ≪ size ≪ ");" ≪ endl
           ≪ "animation a;" ≪ endl;
```

Uses `name` 32e.

For every element of *val* we perform the substitution and draw the corresponding picture.

105c ⟨figure class 75a⟩+≡ (52a) ◁105b 105d▷

```
     for (const auto& it : ex_to<lst>(val)) {
        out ≪ "save();" ≪ endl;
        unfreeze().subs(it).asy_draw(out, cerr, "", xmin, xmax, ymin, ymax,
                              style, lstring, with_realline, with_header, points_per_arc, imaginary_options, with_labels);
```

Uses `asy_draw` 36b 36b 101a, `save` 38a 38a, `subs` 50a, and `unfreeze` 17b 37b.

We prepare the value string for output.

105d ⟨figure class 75a⟩+≡ (52a) ◁105c 105e▷

```
        std::regex deq ("==");
        stringstream sstr;
        sstr ≪  (ex)it;
        string val_str=std::regex_replace(sstr.str(),deq,"=");
```

Uses `ex` 41b 47e 47e 47e 53a.

We put the value of parameters to the figure in accordance with *values_position*.

105e ⟨figure class 75a⟩+≡ (52a) ◁105d 106a▷

```
        if (values_position≡"bl")
           out ≪ "label(\"\\texttt{" ≪ val_str ≪ "}\",(" ≪ xmin ≪ "," ≪ ymin ≪ "), SE);";
        else if (values_position≡"br")
           out ≪ "label(\"\\texttt{" ≪ val_str ≪ "}\",(" ≪ xmax ≪ "," ≪ ymin ≪ "), SW);";
        else if (values_position≡"tl")
           out ≪ "label(\"\\texttt{" ≪ val_str ≪ "}\",(" ≪ xmin ≪ "," ≪ ymax ≪ "), NE);";
        else if (values_position≡"tr")
           out ≪ "label(\"\\texttt{" ≪ val_str ≪ "}\",(" ≪ xmax ≪ "," ≪ ymax ≪ "), NW);";

        out ≪ "a.add();" ≪ endl
           ≪ "restore();" ≪ endl;
     }
```

For output in PDF, GIF, MNG or MP4 format we supply default commands. User may do a custom command using *format* parameter.

106a ⟨figure class 75a⟩+≡ (52a) ◁105e 106b▷
```
if (format ≡ "pdf")
   out ≪ "label(a.pdf(\"controls\",delay=250,keep=!settings.inlinetex));" ≪ endl;
else if ((format ≡ "gif") ∨ (format ≡ "mp4") ∨ (format ≡ "mng"))
   out ≪ "a.movie(loops=10,delay=250);" ≪ endl;
else
   out ≪ format ≪ endl;
out.flush();
out.close();
```

Finally we run `Asymptote` to produce an animation.

106b ⟨figure class 75a⟩+≡ (52a) ◁106a 106c▷
```
   char command[256];
   strcpy(command, show_asy_graphics? "asy -V " : "asy ");
   if ((format ≡ "gif") ∨ (format ≡ "mp4") ∨ (format ≡ "mng")) {
      strcat(command, " -f ");
      strcat(command, format.c_str());
      strcat(command, " ");
   }
   strcat(command, name.c_str());
   char * pcommand=command;
   system(pcommand);
   if (rm_asy_file)
      remove(filename.c_str());
}
```

Uses `name` 32e and `show_asy_graphics` 52d.

All cycles in generations starting from *first_gen* (default value is 0) are dumped to a text file *name.txt*. Firstly, we check that the figure is three dimensional and then open the file.

106c ⟨figure class 75a⟩+≡ (52a) ◁106b 106d▷
```
void figure::arrangement_write(string name, int first_gen) const
{
   if (¬ (get_dim()-3).is_zero())
      throw(std::invalid_argument("figure::arrangement_write(): the figure is not in 3D!"));

   string filename=name+".txt";
   ofstream out(filename);
```

Defines:
`arrangement_write`, never used.
`figure`, used in chunks 16–18, 20e, 22, 27b, 28a, 30a, 36, 37, 45, 46c, 48, 50a, 52, 66, 75–84, 86, 87, 95d, 97–100, 106–108, 110, 111, and 117c.

Uses `get_dim()` 35c and `name` 32e.

We produce the iterator over all keys. This is a GiNaC **lst** thus we need iterations through its components.

106d ⟨figure class 75a⟩+≡ (52a) ◁106c 107a▷
```
   lst keys=ex_to<lst>(get_all_keys_sorted(first_gen));
   for (const auto& itk : keys) {
      ex gen=get_generation(itk);
      lst L=ex_to<lst>(get_cycles(itk));
```

Uses `ex` 41b 47e 47e 47e 53a, `get_all_keys_sorted`, and `get_generation` 35f.

This is again a GiNaC **lst**, thus we need iterations through its components again.

107a ⟨figure class 75a⟩+≡ (52a) ◁106d 107b▷
```
      for (const auto& it : L) {
         cycle C=ex_to<cycle>(it);
         ex center = C.center();
```

Uses `ex` 41b 47e 47e 47e 53a.

A line of text represents a cycle by three coordinates of its centre, radius, generation and label.

107b ⟨figure class 75a⟩+≡ (52a) ◁107a 107c▷
```
        out ≪ center.op(0).evalf() ≪ " " ≪ center.op(1).evalf() ≪ " " ≪ center.op(2).evalf()
          ≪ " " ≪ sqrt(C.radius_sq()).evalf()
          ≪ " " ≪ gen
          ≪ " " ≪ itk
          ≪ endl;
        }
    }
    out.flush();
    out.close();
}
```

Uses `evalf` 50a and `op` 50a.

F.5.7. *Service utilities.* Here is the minimal set of service procedures which is reuired by GiNaC for derived classes.

107c ⟨figure class 75a⟩+≡ (52a) ◁107b 108a▷
```
return_type_t figure::return_type_tinfo() const
{
   return make_return_type_t<figure>();
}
```

Uses `figure` 16d 22e 32a 32c 38b 49a 50d 75a 80a 82b 82c 85a 86c 98c 99b 99d 100a 101a 103b 104a 105c 106c 106d 107a 109a 109c 110a.

108a ⟨figure class 75a⟩+≡ (52a) ◁107c 108b▷
```
int figure::compare_same_type(const basic &other) const
{
     GINAC_ASSERT(is_a<figure>(other));
     return inherited::compare_same_type(other);
}
```

Defines:
`figure`, used in chunks 16–18, 20e, 22, 27b, 28a, 30a, 36, 37, 45, 46c, 48, 50a, 52, 66, 75–84, 86, 87, 95d, 97–100, 106–108, 110, 111, and 117c.

To print the figure means to print all its nodes.

108b ⟨figure class 75a⟩+≡ (52a) ◁108a 108c▷
```
void figure::do_print(const print_dflt & con, unsigned level) const {
   lst keys=ex_to<lst>(get_all_keys_sorted(FIGURE_DEBUG? GHOST_GEN:INFINITY_GEN));
   int N_cycle=0;

   for (const auto& ck: keys) {
      N_cycle += get_cycles(ck).nops();
      con.s ≪ ck ≪": " ≪ get_cycle_node(ck);
   }

   con.s ≪ "Altogether " ≪ N_cycle ≪ " cycles in "
        ≪ keys.nops() ≪ " cycle_nodes." ≪ endl;
}
```

Defines:
`figure`, used in chunks 16–18, 20e, 22, 27b, 28a, 30a, 36, 37, 45, 46c, 48, 50a, 52, 66, 75–84, 86, 87, 95d, 97–100, 106–108, 110, 111, and 117c.

Uses `FIGURE_DEBUG` 52c, `get_all_keys_sorted`, `get_cycle_node` 49e, `GHOST_GEN` 42b 42b, `INFINITY_GEN` 42b 42b, and `nops` 50a.

This is a variation of printing in the float form.

108c ⟨figure class 75a⟩+≡ (52a) ◁108b 109a▷

```
void figure::do_print_double(const print_dflt & con, unsigned level) const {
    for (const auto& x: nodes) {
        if (x.second.get_generation() > GHOST_GEN ∨ FIGURE_DEBUG) {
            con.s ≪ x.first ≪": ";
            ex_to<cycle_node>(x.second).do_print_double(con, level);
        }
    }
}
```

Defines:
figure, used in chunks 16–18, 20e, 22, 27b, 28a, 30a, 36, 37, 45, 46c, 48, 50a, 52, 66, 75–84, 86, 87, 95d, 97–100, 106–108, 110, 111, and 117c.

Uses cycle_node 43c 45c 69c 70a 70b 70c 71a 71b 72b 72c 73a 74a 74d, do_print_double 49e, FIGURE_DEBUG 52c, get_generation 35f, GHOST_GEN 42b 42b, and nodes 51a.

109a ⟨figure class 75a⟩+≡ (52a) ◁108c 109b▷

```
ex figure::op(size_t i) const
{
 GINAC_ASSERT(i<nops());
    switch(i) {
    case 0:
        return real_line;
    case 1:
        return infinity;
    case 2:
        return point_metric;
    case 3:
        return cycle_metric;
    default:
        exhashmap<cycle_node>::const_iterator it=nodes.begin();
        for (size_t n=4; n<i;++n)
            ++it;
        return it→second;
    }
}
```

Uses cycle_metric 50f, cycle_node 43c 45c 69c 70a 70b 70c 71a 71b 72b 72c 73a 74a 74d, ex 41b 47e 47e 47e 53a, figure 16d 22e 32a 32c 38b 49a 50d 75a 80a 82b 82c 85a 86c 98c 99b 99d 100a 101a 103b 104a 105c 106c 106d 107a 109a 109c 110a, infinity 50e, nodes 51a, nops 50a, op 50a, point_metric 50f, and real_line 50e.

109b ⟨figure class 75a⟩+≡ (52a) ◁109a 109c▷
```
÷∗ex & figure::let_op(size_t i)
{
    ensure_if_modifiable();
    GINAC_ASSERT(i<nops());
    switch(i) {
    case 0:
        return real_line;
    case 1:
        return infinity;
    case 2:
        return point_metric;
    case 3:
        return cycle_metric;
    default:
        exhashmap<cycle_node>::iterator it=nodes.begin();
        for (size_t n=4; n<i;++n)
            ++it;
        return nodes[it→first];
    }
}∗÷
```

Uses `cycle_metric` 50f, `cycle_node` 43c 45c 69c 70a 70b 70c 71a 71b 72b 72c 73a 74a 74d, `ex` 41b 47e 47e 47e 53a, `figure` 16d 22e 32a 32c 38b 49a 50d 75a 80a 82b 82c 85a 86c 98c 99b 99d 100a 101a 103b 104a 105c 106c 106d 107a 109a 109c 110a, `infinity` 50e, `nodes` 51a, `nops` 50a, `point_metric` 50f, and `real_line` 50e.

We need to make substitution in the form of *exmap*.

109c ⟨figure class 75a⟩+≡ (52a) ◁109b 109d▷
```
figure figure::subs(const ex & e, unsigned options) const
{
    exmap m;
    if (e.info(info_flags::list)) {
        lst sl = ex_to<lst>(e);
        for (const auto& i : sl)
            m.insert(std::make_pair(i.op(0), i.op(1)));
    } else if (is_a<relational>(e)) {
        m.insert(std::make_pair(e.op(0), e.op(1)));
    } else
     throw(std::invalid_argument("cycle::subs(): the parameter should be a relational or a lst"));

    return ex_to<figure>(subs(m, options));
}
```

Uses `ex` 41b 47e 47e 47e 53a, `figure` 16d 22e 32a 32c 38b 49a 50d 75a 80a 82b 82c 85a 86c 98c 99b 99d 100a 101a 103b 104a 105c 106c 106d 107a 109a 109c 110a, `info` 50a, `m` 51c, `op` 50a, and `subs` 50a.

109d ⟨figure class 75a⟩+≡ (52a) ◁109c 110a▷
```
ex figure::subs(const exmap & m, unsigned options) const
{
    exhashmap<cycle_node> snodes;
    for (const auto& x: nodes)
        snodes[x.first]=ex_to<cycle_node>(x.second.subs(m, options));

    if (options & do_not_update_subfigure)
        return figure(point_metric.subs(m, options), cycle_metric.subs(m, options), snodes);
    else
        return figure(point_metric.subs(m, options), cycle_metric.subs(m, options), snodes).update_cycles();
}
```

Uses `cycle_metric` 50f, `cycle_node` 43c 45c 69c 70a 70b 70c 71a 71b 72b 72c 73a 74a 74d, `do_not_update_subfigure` 52b, `ex` 41b 47e 47e 47e 53a, `figure` 16d 22e 32a 32c 38b 49a 50d 75a 80a 82b 82c 85a 86c 98c 99b 99d 100a 101a 103b 104a 105c 106c 106d 107a 109a 109c 110a, `m` 51c, `nodes` 51a, `point_metric` 50f, `subs` 50a, and `update_cycles` 50d 98a.

110a ⟨figure class 75a⟩+≡ (52a) ◁109d 110b▷
```
ex figure::evalf(int level) const
{
    exhashmap<cycle_node> snodes;
    for (const auto& x: nodes)
#if GINAC_VERSION_ATLEAST(1,7,0)
        snodes[x.first]=ex_to<cycle_node>(x.second.evalf());

    return figure(point_metric.evalf(), cycle_metric.evalf(), snodes);
#else
        snodes[x.first]=ex_to<cycle_node>(x.second.evalf(level));

    return figure(point_metric.evalf(level), cycle_metric.evalf(level), snodes);
#endif
}
```

Uses `cycle_metric` 50f, `cycle_node` 43c 45c 69c 70a 70b 70c 71a 71b 72b 72c 73a 74a 74d, `evalf` 50a, `ex` 41b 47e 47e 47e 53a, `figure` 16d 22e 32a 32c 38b 49a 50d 75a 80a 82b 82c 85a 86c 98c 99b 99d 100a 101a 103b 104a 105c 106c 106d 107a 109a 109c 110a, `nodes` 51a, and `point_metric` 50f.

F.5.8. *Archiving/Unarchiving utilities.*

110b ⟨figure class 75a⟩+≡ (52a) ◁110a 110c▷
```
void figure::archive(archive_node &an) const
{
    inherited::archive(an);
    an.add_ex("real_line", real_line);
    an.add_ex("infinity", infinity);
    an.add_ex("point_metric", point_metric);
    an.add_ex("cycle_metric", cycle_metric);
    an.add_bool("float_evaluation", float_evaluation);
```

Defines:
`figure`, used in chunks 16–18, 20e, 22, 27b, 28a, 30a, 36, 37, 45, 46c, 48, 50a, 52, 66, 75–84, 86, 87, 95d, 97–100, 106–108, 110, 111, and 117c.

Uses `archive` 50a, `cycle_metric` 50f, `float_evaluation` 51b, `infinity` 50e, `point_metric` 50f, and `real_line` 50e.

exhashmap class does not have an archiving facility, thus we store it as two correponding lists.

110c ⟨figure class 75a⟩+≡ (52a) ◁110b 111b▷
```
    lst keys, cnodes;
    for (const auto& x: nodes) {
        keys.append(x.first);
        cnodes.append(x.second);
    }
    an.add_ex("keys", keys);
    an.add_ex("cnodes", cnodes);
    an.add_string("info_text", info_text);
}
```

Uses `info_text` and `nodes` 51a.

The respective un-archiving function. For some unclear reasons if both point and cycle metrics are indexed by the same symbol, then the cyle metric becomes a copy of the point one.

111b ⟨figure class 75a⟩+≡ (52a) ◁110c 111c▷

```
void figure::read_archive(const archive_node &an, lst &sym_lst)
{
    inherited::read_archive(an, sym_lst);
    an.find_ex("point_metric", point_metric, sym_lst);
    an.find_ex("cycle_metric", cycle_metric, sym_lst);
    lst all_sym=sym_lst;
    ex keys, cnodes;
    an.find_ex("real_line", real_line, sym_lst);
    all_sym.append(real_line);
    an.find_ex("infinity", infinity, sym_lst);
    all_sym.append(infinity);
    an.find_bool("float_evaluation", float_evaluation);
```

Defines:
`figure`, used in chunks 16–18, 20e, 22, 27b, 28a, 30a, 36, 37, 45, 46c, 48, 50a, 52, 66, 75–84, 86, 87, 95d, 97–100, 106–108, 110, 111, and 117c.

Uses `cycle_metric` 50f, `ex` 41b 47e 47e 47e 53a, `float_evaluation` 51b, `infinity` 50e, `point_metric` 50f, `read_archive` 50a, and `real_line` 50e.

111c ⟨figure class 75a⟩+≡ (52a) ◁111b 111d▷

```
    //an.find_ex("keys", keys, all_sym);
    an.find_ex("keys", keys, sym_lst);
    for (const auto& it : ex_to<lst>(keys))
        all_sym.append(it);
    all_sym.sort();
    all_sym.unique();
    an.find_ex("cnodes", cnodes, all_sym);
    lst::const_iterator it1 = ex_to<lst>(cnodes).begin();
    nodes.clear();
    for (const auto& it : ex_to<lst>(keys)) {
        nodes[it]=ex_to<cycle_node>(*it1);
        ++it1;
    }
    an.find_string("info_text", info_text);
}
```

Uses `cycle_node` 43c 45c 69c 70a 70b 70c 71a 71b 72b 72c 73a 74a 74d, `info_text`, and `nodes` 51a.

111d ⟨figure class 75a⟩+≡ (52a) ◁111c 112a▷

```
GINAC_BIND_UNARCHIVER(figure);
```

Defines:
`figure`, used in chunks 16–18, 20e, 22, 27b, 28a, 30a, 36, 37, 45, 46c, 48, 50a, 52, 66, 75–84, 86, 87, 95d, 97–100, 106–108, 110, 111, and 117c.

112a ⟨figure class 75a⟩+≡ (52a) ◁111d 112b▷

```
bool figure::info(unsigned inf) const
{
    switch (inf) {
    case status_flags::expanded:
        return (inf & flags);
    }
    return inherited::info(inf);
}
```

Uses `figure` 16d 22e 32a 32c 38b 49a 50d 75a 80a 82b 82c 85a 86c 98c 99b 99d 100a 101a 103b 104a 105c 106c 106d 107a 109a 109c 110a and `info` 50a.

F.5.9. *Relations and measurements.* The method to check that two cycles are in a relation.

112b ⟨figure class 75a⟩+≡ (52a) ◁112a 112c▷
```
ex figure::check_rel(const ex & key1, const ex & key2, PCR rel, bool use_cycle_metric,
                     const ex & parameter, bool corresponds) const
{
    ⟨run through all cycles in two nodes correspondingly 110d⟩
    ⟨add checked relation 110e⟩
    ⟨run through all cycles in two nodes async 111a⟩
    ⟨add checked relation 110e⟩
```
Defines:
check_rel, used in chunks 22a and 25c.
Uses ex 41b 47e 47e 47e 53a, figure 16d 22e 32a 32c 38b 49a 50d 75a 80a 82b 82c 85a 86c 98c 99b 99d 100a 101a 103b 104a 105c 106c 106d 107a 109a 109c 110a, and PCR 45d.

This piece of code is common in *check_rel* and *measure*.

110d ⟨run through all cycles in two nodes correspondingly 110d⟩≡ (? 112b)
```
    lst res,
        cycles1=ex_to<lst>(ex_to<cycle_node>(nodes.find(key1)->second)
                          .make_cycles(use_cycle_metric? cycle_metric : point_metric)),
        cycles2=ex_to<lst>(ex_to<cycle_node>(nodes.find(key2)->second)
                          .make_cycles(use_cycle_metric? cycle_metric : point_metric));

    if (corresponds && cycles1.nops() == cycles2.nops()) {
        auto it2=cycles2.begin();
        for (const auto& it1 : cycles1) {
            lst calc=ex_to<lst>(rel(it1,*(it2++),parameter));
            for (const auto& itr : calc)
```
Uses cycle_metric 50f, cycle_node 43c 45c 69c 70a 70b 70c 71a 71b 72b 72c 73a 74a 74d, nodes 51a, nops 50a, and point_metric 50f.

We add corresponding relation. We wish to make output homogeneous despite of the fact that *rel* can be of different type: either returning **relational** or not.

110e ⟨add checked relation 110e⟩≡ (112b)
```
            {
                ex e=(itr.op(0)).normal();
                if (is_a<relational>(e))
                    res.append(e);
                else
                    res.append(e==0);
            }
```
Uses ex 41b 47e 47e 47e 53a and op 50a.

If cycles are treated asynchronously we run two independent loops.

111a ⟨run through all cycles in two nodes async 111a⟩≡ (? 112b)
```
     }
    } else {
        for (const auto& it1 : cycles1) {
            for (const auto& it2 : cycles2) {
                lst calc=ex_to<lst>(rel(it1,it2,parameter));
                for (const auto& itr : calc)
```
Simply finish the routine with the right number of brackets.

112c ⟨figure class 75a⟩+≡ (52a) ◁112b ??▷
```
            }
        }
    }
    return res;
}
```

The method to measure certain quantity, it essentially copies code from the previous method.

?? ⟨figure class 75a⟩+≡ (52a) ◁112c ??▷

```
ex figure::measure(const ex & key1, const ex & key2, PCR rel, bool use_cycle_metric,
                    const ex & parameter, bool corresponds) const
{
    ⟨run through all cycles in two nodes correspondingly 110d⟩
    res.append(itr.op(0));
    ⟨run through all cycles in two nodes async 111a⟩
    res.append(itr.op(0));
                }
            }
        }
    return res;
}
```

Defines:
measure, never used.
Uses ex 41b 47e 47e 47e 53a, figure 16d 22e 32a 32c 38b 49a 50d 75a 80a 82b 82c 85a 86c 98c 99b 99d 100a 101a 103b 104a 105c 106c 106d 107a 109a 109c 110a, op 50a, and PCR 45d.

We apply *func* to all cycles in the, figure one-by-one.

?? ⟨figure class 75a⟩+≡ (52a) ◁?? ??▷

```
ex figure::apply(PEVAL func, bool use_cycle_metric, const ex & param) const
{
    lst res;
    for (const auto& x: nodes) {
        int i=0;
        lst cycles=ex_to<lst>(x.second.make_cycles(use_cycle_metric? cycle_metric : point_metric));
        for (const auto& itc : cycles) {
            res.append(lst{func(itc, param), x.first, i});
            ++i;
        }
    }
    return res;
}
```

Defines:
apply, never used.
Uses cycle_metric 50f, ex 41b 47e 47e 47e 53a, figure 16d 22e 32a 32c 38b 49a 50d 75a 80a 82b 82c 85a 86c 98c 99b 99d 100a 101a 103b 104a 105c 106c 106d 107a 109a 109c 110a, nodes 51a, and point_metric 50f.

F.5.10. *Default Asymptote styles.* A simple Asymptote style. We produce different colours for points, lines and circles. No further options are specified.

?? ⟨figure class 75a⟩+≡ (52a) ◁?? ??▷

```
string asy_cycle_color(const ex & label, const ex & C, lst & color)
{
    string asy_options="";
    if (is_less_than_epsilon(ex_to<cycle>(C).det())) {// point
        color=lst{0.5,0,0};
        asy_options="dotted";
    } else if (is_less_than_epsilon(ex_to<cycle>(C).get_k())) // straight line
        color=lst{0,0.5,0};
    else  // a proper circle-hyperbola-parabola
        color=lst{0,0,0.5};

    return asy_options;
}
```

Defines:
asy_cycle_color, used in chunk 51d.
Uses ex 41b 47e 47e 47e 53a and is_less_than_epsilon 53b.

A style to place labels.

?? ⟨figure class 75a⟩+≡ (52a) ◁?? ??▷
```
string label_pos(const ex & label, const ex & C, const string draw_str) {
    stringstream sstr;
    sstr ≪ latex ≪ label;

    string name=ex_to<symbol>(label).get_name(), new_TeXname;

    if (sstr.str() ≡ name) {
        string TeXname;
        ⟨auto TeX name 84d⟩
        if (TeXname_new ≡"")
            new_TeXname=name;
        else
            new_TeXname=TeXname_new;
    } else
        new_TeXname=sstr.str();
```
Defines:
`label_pos`, used in chunk 51e.
Uses `ex` 41b 47e 47e 47e 53a, `name` 32e, and `TeXname` 32e.

We use *regex* to spot places for labels in the `Asymptote` output.

?? ⟨figure class 75a⟩+≡ (52a) ◁??
```
    std::regex draw("([.\\n\\r\\s]*)(draw)\\(([\\w]+,)?((?:\\(.+?\\)|\\{.+?\\}|[^-,0-9\\.])+),([.\\n\\r]*)");
    std::regex dot("([.\\n\\r\\s]*)(dot)\\(([\\w]*,)?((?:\\(.+?\\)|\\{.+?\\}|[^-,0-9\\.])+|[\\w]+),([.\\n\\r]
    std::regex e1("symbolLaTeXname");

    if (std::regex_search(draw_str, dot)) {
            string labelstr=std::regex_replace (draw_str, dot,
                            "label($3\"$symbolLaTeXname$\", $4, SE);\n",
                            std::regex_constants::format_no_copy);
            return std::regex_replace (labelstr, e1, new_TeXname);
    } else if (std::regex_search(draw_str, draw)) {
            string labelstr=std::regex_replace (draw_str, draw,
                            "label($3\"$symbolLaTeXname$\", point($4,0.1), SE);\n",
                            std::regex_constants::format_no_copy | std::regex_constants::format_first_only);
            return std::regex_replace (labelstr, e1, new_TeXname);
    } else
        return "";
}
```

F.6. **Functions defining cycle relations.** This is collection of linear cycle relations which do not require a parameter.

113a ⟨add cycle relations 113a⟩≡ (52a) 113b▷
```
ex cycle_orthogonal(const ex & C1, const ex & C2, const ex & pr)
{
    return lst{(ex)lst{ex_to<cycle>(C1).is_orthogonal(ex_to<cycle>(C2))}};
}
```
Defines:
`cycle_orthogonal`, used in chunks 21d, 22a, 24g, 38c, 60–62, 64a, 81a, 117, and 118.
Uses `ex` 41b 47e 47e 47e 53a and `is_orthogonal` 23c 38c.

113b ⟨add cycle relations 113a⟩+≡ (52a) ◁113a 113c▷
```
ex cycle_f_orthogonal(const ex & C1, const ex & C2, const ex & pr)
{
    return lst{(ex)lst{ex_to<cycle>(C1).is_f_orthogonal(ex_to<cycle>(C2))}};
}
```
Defines:
`cycle_f_orthogonal`, used in chunks 38d, 61, 62a, and 64a.
Uses `ex` 41b 47e 47e 47e 53a and `is_f_orthogonal` 38d.

113c ⟨add cycle relations 113a⟩+≡ (52a) ◁113b 113d▷
```
ex cycle_adifferent(const ex & C1, const ex & C2, const ex & pr)
{
   return lst{(ex)lst{cycle_data(C1).is_almost_equal(ex_to<basic>(cycle_data(C2)),true)? 0: 1}};
}
```
Defines:
cycle_adifferent, used in chunks 38f, 61, 62a, 64a, and 118c.
Uses cycle_data 23a 26a 42c 43a 54c 54d 55a 56b 56c 56d, ex 41b 47e 47e 47e 53a, and is_almost_equal 117a.

To *check* the tangential property we use the condition from [36, Ex. 5.26(i)]

$$(\langle C_1, C_2\rangle)^2 - \langle C_1, C_1\rangle \langle C_2, C_2\rangle = 0. \tag{21}$$

113d ⟨add cycle relations 113a⟩+≡ (52a) ◁113c 113e▷
```
ex check_tangent(const ex & C1, const ex & C2, const ex & pr)
{
   return lst{(ex)lst{pow(ex_to<cycle>(C1).cycle_product(ex_to<cycle>(C2)),2)
              -ex_to<cycle>(C1).cycle_product(ex_to<cycle>(C1))
              *ex_to<cycle>(C2).cycle_product(ex_to<cycle>(C2)) ≡ 0}};
}
```
Defines:
check_tangent, used in chunk 25c.
Uses ex 41b 47e 47e 47e 53a.

To *define* tangential property, theoretically we can use (21) as well. However, a system of several such quadratic conditions will be difficult to resolve. Thus, we use a single quadratic relations $\langle C_1, C_1\rangle = -1$ which allows to linearise the tangential property to a pair of identities: $\langle C_1, C_2\rangle \pm \sqrt{\langle C_2, C_2\rangle} = 0$.

113e ⟨add cycle relations 113a⟩+≡ (52a) ◁113d 114a▷
```
ex cycle_tangent(const ex & C1, const ex & C2, const ex & pr)
{
   return lst{lst{ex_to<cycle>(C1).cycle_product(ex_to<cycle>(C1))+numeric(1)≡0,
                   ex_to<cycle>(C1).cycle_product(ex_to<cycle>(C2))
                  -sqrt(abs(ex_to<cycle>(C2).cycle_product(ex_to<cycle>(C2))))≡0},
              lst{ex_to<cycle>(C1).cycle_product(ex_to<cycle>(C1))-numeric(1)≡0,
                   ex_to<cycle>(C1).cycle_product(ex_to<cycle>(C2))
                  -sqrt(abs(ex_to<cycle>(C2).cycle_product(ex_to<cycle>(C2))))≡0},
              lst{ex_to<cycle>(C1).cycle_product(ex_to<cycle>(C1))+numeric(1)≡0,
                   ex_to<cycle>(C1).cycle_product(ex_to<cycle>(C2))
                  +sqrt(abs(ex_to<cycle>(C2).cycle_product(ex_to<cycle>(C2))))≡0},
              lst{ex_to<cycle>(C1).cycle_product(ex_to<cycle>(C1))-numeric(1)≡0,
                   ex_to<cycle>(C1).cycle_product(ex_to<cycle>(C2))
                  +sqrt(abs(ex_to<cycle>(C2).cycle_product(ex_to<cycle>(C2))))≡0}};
}
```
Defines:
cycle_tangent, used in chunks 39c, 61, 62a, and 64a.
Uses ex 41b 47e 47e 47e 53a and numeric 22d.

114a ⟨add cycle relations 113a⟩+≡ (52a) ◁113e 114b▷
```
ex cycle_tangent_o(const ex & C1, const ex & C2, const ex & pr)
{
   return lst{lst{ex_to<cycle>(C1).cycle_product(ex_to<cycle>(C1))+numeric(1)≡0,
               ex_to<cycle>(C1).cycle_product(ex_to<cycle>(C2))
              -sqrt(abs(ex_to<cycle>(C2).cycle_product(ex_to<cycle>(C2))))≡0},
           lst{ex_to<cycle>(C1).cycle_product(ex_to<cycle>(C1))-numeric(1)≡0,
                   ex_to<cycle>(C1).cycle_product(ex_to<cycle>(C2))
                  -sqrt(abs(ex_to<cycle>(C2).cycle_product(ex_to<cycle>(C2))))≡0}};
}
```
Defines:
cycle_tangent_o, used in chunks 39d, 61, 62a, and 64a.
Uses ex 41b 47e 47e 47e 53a and numeric 22d.

114b ⟨add cycle relations 113a⟩+≡ (52a) ◁114a 114c▷

```
ex cycle_tangent_i(const ex & C1, const ex & C2, const ex & pr)
{
    return lst{lst{ex_to<cycle>(C1).cycle_product(ex_to<cycle>(C1))+numeric(1)≡0,
              ex_to<cycle>(C1).cycle_product(ex_to<cycle>(C2))
              +sqrt(abs(ex_to<cycle>(C2).cycle_product(ex_to<cycle>(C2))))≡0},
          lst{ex_to<cycle>(C1).cycle_product(ex_to<cycle>(C1))-numeric(1)≡0,
              ex_to<cycle>(C1).cycle_product(ex_to<cycle>(C2))
              +sqrt(abs(ex_to<cycle>(C2).cycle_product(ex_to<cycle>(C2))))≡0}};
}
```

Defines:
`cycle_tangent_i`, used in chunks 39d, 61, 62a, and 64a.
Uses `ex` 41b 47e 47e 47e 53a and `numeric` 22d.

114c ⟨add cycle relations 113a⟩+≡ (52a) ◁114b 114d▷

```
ex cycle_different(const ex & C1, const ex & C2, const ex & pr)
{
    return lst{(ex)lst{ex_to<cycle>(C1).is_equal(ex_to<basic>(C2), true)? 0: 1}};
}
```

Defines:
`cycle_different`, used in chunks 38e, 61, 62a, 64a, and 81a.
Uses `ex` 41b 47e 47e 47e 53a.

If the cycle product has imaginary part we return the false statement. For a real cycle product we check its sign.

114d ⟨add cycle relations 113a⟩+≡ (52a) ◁114c 114e▷

```
ex product_sign(const ex & C1, const ex & C2, const ex & pr)
{
    if (is_less_than_epsilon(ex_to<cycle>(C1).cycle_product(ex_to<cycle>(C1)).evalf().imag_part()))
        return lst{(ex)lst{pr*(ex_to<cycle>(C1).cycle_product(ex_to<cycle>(C1)).evalf().real_part() - ep-
silon) <0}};
    else
        return lst{(ex)lst{numeric(1) <0}};
}
```

Defines:
`product_sign`, used in chunks 38g, 39a, 61, 62a, and 64a.
Uses `epsilon` 53a, `evalf` 50a, `ex` 41b 47e 47e 47e 53a, `is_less_than_epsilon` 53b, and `numeric` 22d.

Now we define the relation between cycles to "intersect with certain angle" (but the "intersection" may be imaginary). If cycles are intersecting indeed then the value of *pr* is the cosine of the angle.

114e ⟨add cycle relations 113a⟩+≡ (52a) ◁114d 115a▷

```
ex cycle_angle(const ex & C1, const ex & C2, const ex & pr)
{
    return lst{lst{ex_to<cycle>(C1).cycle_product(ex_to<cycle>(C2).normalize_norm())-pr≡0,
              ex_to<cycle>(C1).cycle_product(ex_to<cycle>(C1))+numeric(1)≡0},
          lst{ex_to<cycle>(C1).cycle_product(ex_to<cycle>(C2).normalize_norm())-pr≡0,
              ex_to<cycle>(C1).cycle_product(ex_to<cycle>(C1))-numeric(1)≡0}};
}
```

Defines:
`cycle_angle`, used in chunks 39e, 61, 62a, and 64a.
Uses `ex` 41b 47e 47e 47e 53a and `numeric` 22d.

The next relation defines tangential distance between cycles.

115a ⟨add cycle relations 113a⟩+≡ (52a) ◁114e 115b▷

```
ex steiner_power(const ex & C1, const ex & C2, const ex & pr)
{
    cycle C=ex_to<cycle>(C2).normalize();
    return lst{lst{ex_to<cycle>(C1).cycle_product(C)+sqrt(abs(C.cycle_product(C)))
              -pr*ex_to<cycle>(C1).get_k()≡0,
              ex_to<cycle>(C1).cycle_product(ex_to<cycle>(C1))+numeric(1)≡0},
          lst{ex_to<cycle>(C1).cycle_product(C)+sqrt(abs(C.cycle_product(C)))
              -pr*ex_to<cycle>(C1).get_k()≡0,
              ex_to<cycle>(C1).cycle_product(ex_to<cycle>(C1))-numeric(1)≡0}};
}
```

Defines:
`steiner_power`, used in chunks 39, 61, 62a, and 64a.
Uses `ex` 41b 47e 47e 47e 53a and `numeric` 22d.

Cross tangential distance is different by a sign of one term.

115b ⟨add cycle relations 113a⟩+≡ (52a) ◁115a 115c▷
```
ex cycle_cross_t_distance(const ex & C1, const ex & C2, const ex & pr)
{
   cycle C=ex_to<cycle>(C2).normalize();
   return lst{lst{ex_to<cycle>(C1).cycle_product(C)-sqrt(abs(C.cycle_product(C)))
            -pow(pr,2)*ex_to<cycle>(C1).get_k()≡0,
            ex_to<cycle>(C1).cycle_product(ex_to<cycle>(C1))+numeric(1)≡0},
         lst{ex_to<cycle>(C1).cycle_product(C)-sqrt(abs(C.cycle_product(C)))
            -pow(pr,2)*ex_to<cycle>(C1).get_k()≡0,
            ex_to<cycle>(C1).cycle_product(ex_to<cycle>(C1))-numeric(1)≡0}};
}
```
Defines:
cycle_cross_t_distance, used in chunks 39h, 61, 62a, and 64a.
Uses ex 41b 47e 47e 47e 53a and numeric 22d.

Check that all coefficients of the first cycle are real.

115c ⟨add cycle relations 113a⟩+≡ (52a) ◁115b 115d▷
```
ex coefficients_are_real(const ex & C1, const ex & C2, const ex & pr)
{
   cycle C=ex_to<cycle>(ex_to<cycle>(C1.evalf()).imag_part());
   if (¬ (is_less_than_epsilon(C.get_k()) ∧ is_less_than_epsilon(C.get_m())))
      return lst{(ex)lst{0}};
   for (int i=0; i < ex_to<cycle>(C1).get_dim(); ++i)
      if (¬ is_less_than_epsilon(C.get_l(i)))
         return lst{(ex)lst{0}};

   return lst{(ex)lst{1}};
}
```
Defines:
coefficients_are_real, used in chunks 39b, 61, 62a, and 64a.
Uses evalf 50a, ex 41b 47e 47e 47e 53a, get_dim() 35c, and is_less_than_epsilon 53b.

F.6.1. *Measured quantities.* This function measures relative powers of two cycles, which turn to be their cycle product for norm-normalised vectors.

115d ⟨add cycle relations 113a⟩+≡ (52a) ◁115c 116a▷
```
ex angle_is(const ex & C1, const ex & C2, const ex & pr)
{
  return lst{(ex)lst{ex_to<cycle>(C1).normalize_norm().cycle_product(ex_to<cycle>(C2).normalize_norm())}};
}
```
Defines:
angle_is, never used.
Uses ex 41b 47e 47e 47e 53a.

This function measures relative powers of two cycles, which turn to be their cycle product for k-normalised vectors.

116a ⟨add cycle relations 113a⟩+≡ (52a) ◁115d 116b▷
```
ex power_is(const ex & C1, const ex & C2, const ex & pr)
{
   cycle Ca=ex_to<cycle>(C1).normalize(), Cb=ex_to<cycle>(C2).normalize();

   return lst{(ex)lst{Ca.cycle_product(Cb)+pr*sqrt(abs(Ca.cycle_product(Ca)*Cb.cycle_product(Cb)))}};
}
```
Defines:
power_is, never used.
Uses ex 41b 47e 47e 47e 53a.

116b ⟨add cycle relations 113a⟩+≡ (52a) ◁116a 116c▷
```
ex cycle_moebius(const ex & C1, const ex & C2, const ex & pr)
{
   return lst{(ex)lst{ex_to<cycle>(C2).matrix_similarity(pr.op(0),pr.op(1),pr.op(2),pr.op(3))}};
}
```
Defines:
cycle_moebius, used in chunks 40a, 61, 62a, and 64a.
Uses ex 41b 47e 47e 47e 53a and op 50a.

That relations works only for real matrices, thus we start from the relevant checks.

116c ⟨add cycle relations 113a⟩+≡ (52a) ◁116b 116d▷
```
cycle_relation sl2_transform(const ex & key, bool cm, const ex & matrix) {
   if (is_a<lst>(matrix) ∧ matrix.op(0).info(info_flags::real) ∧ matrix.op(1).info(info_flags::real)
      ∧ matrix.op(2).info(info_flags::real) ∧ matrix.op(3).info(info_flags::real))
         return cycle_relation(key, cycle_sl2, cm, matrix);
   else
      throw(std::invalid_argument("sl2_transform(): shall be applied only with a matrix having"
                           " real entries"));
}
```
Defines:
sl2_transform, never used.
Uses cycle_relation 40c 45e 46c 60d 61 62a 62b 64a 64b, cycle_sl2 47a 116d, ex 41b 47e 47e 47e 53a, info 50a, key 32e, and op 50a.

That relations works only in two dimensions, thus we start from the relevant checks.

116d ⟨add cycle relations 113a⟩+≡ (52a) ◁116c
```
ex cycle_sl2(const ex & C1, const ex & C2, const ex & pr)
{
   if (ex_to<cycle>(C2).get_dim() ≡ 2)
      return lst{(ex)lst{ex_to<cycle>(C2).sl2_similarity(pr.op(0),pr.op(1),pr.op(2),pr.op(3),
                                    ex_to<cycle>(C2).get_unit())}};
   else
      throw(std::invalid_argument("cycle_sl2(): shall be applied only in two dimensions"));
}
```
Defines:
cycle_sl2, used in chunks 61, 62a, 64a, and 116c.
Uses ex 41b 47e 47e 47e 53a, get_dim() 35c, and op 50a.

F.7. **Additional functions.** Equality of **cycle**s.

117a ⟨addional functions 117a⟩≡ (52a) 117b▷
```
bool is_almost_equal(const ex & A, const ex & B)
{
   if ((not is_a<cycle>(A)) ∨ (not is_a<cycle>(B)))
      return false;

   const cycle C1 = ex_to<cycle>(A),
      C2 = ex_to<cycle>(B);
   ex factor=0, ofactor=0;

   // Check that coefficients are scalar multiples of C2
   if (not is_less_than_epsilon((C1.get_m()*C2.get_k()-C2.get_m()*C1.get_k()).normal()))
      return false;
   // Set up coefficients for proportionality
   if (C1.get_k().normal().is_zero()) {
      factor=C1.get_m();
      ofactor=C2.get_m();
   } else {
      factor=C1.get_k();
      ofactor=C2.get_k();
   }
```
Defines:
is_almost_equal, used in chunks 43a, 58b, 113c, and 119a.
Uses ex 41b 47e 47e 47e 53a and is_less_than_epsilon 53b.

Now we iterate through the coefficients of l.

117b ⟨addional functions 117a⟩+≡ (52a) ◁117a 117c▷
```
    for (unsigned int i=0; i<C1.get_l().nops(); i++)
     // search the the first non-zero coefficient
     if (factor.is_zero()) {
        factor=C1.get_l(i);
        ofactor=C2.get_l(i);
     } else
        if (¬ is_less_than_epsilon((C1.get_l(i)*ofactor-C2.get_l(i)*factor).normal()))
           return false;
   return true;
 }
```

Uses `is_less_than_epsilon` 53b and `nops` 50a.

117c ⟨addional functions 117a⟩+≡ (52a) ◁117b 117d▷
```
 ex midpoint_constructor()
 {
   figure SF=ex_to<figure>((new figure)→setflag(status_flags::expanded));

   ex v1=SF.add_cycle(cycle_data(),"variable000");
   ex v2=SF.add_cycle(cycle_data(),"variable001");
   ex v3=SF.add_cycle(cycle_data(),"variable002");
```
Defines:
  `midpoint_constructor`, used in chunk 22f.
Uses `add_cycle` 23a 32f 81d, `cycle_data` 23a 26a 42c 43a 54c 54d 55a 56b 56c 56d, `ex` 41b 47e 47e 47e 53a, and `figure` 16d 22e 32a 32c 38b 49a 50d 75a 80a 82b 82c 85a 86c 98c 99b 99d 100a 101a 103b 104a 105c 106c 106d 107a 109a 109c 110a.

Join three point by an ”interval” cycle.

117d ⟨addional functions 117a⟩+≡ (52a) ◁117c 118a▷
```
   ex v4=SF.add_cycle_rel(lst{cycle_relation(v1,cycle_orthogonal),
         cycle_relation(v2,cycle_orthogonal),
         cycle_relation(v3,cycle_orthogonal)},
     "v4");
```

Uses `add_cycle_rel` 16f 23c 33a 83b, `cycle_orthogonal` 34b 113a, `cycle_relation` 40c 45e 46c 60d 61 62a 62b 64a 64b, and `ex` 41b 47e 47e 47e 53a.

A cycle ortogonal to the above interval.

118a ⟨addional functions 117a⟩+≡ (52a) ◁117d 118b▷
```
   ex v5=SF.add_cycle_rel(lst{cycle_relation(v1,cycle_orthogonal),
         cycle_relation(v2,cycle_orthogonal),
         cycle_relation(v4,cycle_orthogonal)},
     "v5");
```

Uses `add_cycle_rel` 16f 23c 33a 83b, `cycle_orthogonal` 34b 113a, `cycle_relation` 40c 45e 46c 60d 61 62a 62b 64a 64b, and `ex` 41b 47e 47e 47e 53a.

The perpendicular to the interval and the cycle passing the midpoint.

118b ⟨addional functions 117a⟩+≡ (52a) ◁118a 118c▷
```
   ex v6=SF.add_cycle_rel(lst{cycle_relation(v3,cycle_orthogonal),
         cycle_relation(v4,cycle_orthogonal),
         cycle_relation(v5,cycle_orthogonal)},
     "v6");
```

Uses `add_cycle_rel` 16f 23c 33a 83b, `cycle_orthogonal` 34b 113a, `cycle_relation` 40c 45e 46c 60d 61 62a 62b 64a 64b, and `ex` 41b 47e 47e 47e 53a.

The mid point as the intersection point.

118c ⟨addional functions 117a⟩+≡ (52a) ◁118b 119a▷

```
    ex r=symbol("result");
  SF.add_cycle_rel(lst{cycle_relation(v4,cycle_orthogonal),
          cycle_relation(v6,cycle_orthogonal),
          cycle_relation(r,cycle_orthogonal,false),
          cycle_relation(v3,cycle_adifferent)},
     r);

    return SF;
  }
```

Uses add_cycle_rel 16f 23c 33a 83b, cycle_adifferent 34f 113c, cycle_orthogonal 34b 113a, cycle_relation 40c 45e 46c 60d 61 62a 62b 64a 64b, and ex 41b 47e 47e 47e 53a.

This is an auxiliary function which removes duplicated cycles from a list L.

119a ⟨addional functions 117a⟩+≡ (52a) ◁118c 119b▷

```
  ex unique_cycle(const ex & L)
  {
    if(is_a<lst>(L) ∧ (L.nops() > 1) ) {
      lst res;
      lst::const_iterator it = ex_to<lst>(L).begin();
      if (is_a<cycle_data>(*it)) {
        res.append(*it);
        ++it;
        for (; it ≠ ex_to<lst>(L).end(); ++it) {
          bool is_new=true;
          if (¬ is_a<cycle_data>(*it))
            break; // a non-cycle detected, get out

          for (const auto& it1 : res)
            if (ex_to<cycle_data>(*it).is_almost_equal(ex_to<basic>(it1),true)
              ∨ ex_to<cycle_data>(*it).is_equal(ex_to<basic>(it1),true)) {
              is_new=false; // is a duplicate
              break;
            }
          if (is_new)
            res.append(*it);
        }
        if (it ≡ ex_to<lst>(L).end()) // all are processed, no non-cycle is detected
          return res;
      }
    }
    return L;
  }
```

Defines:
unique_cycle, used in chunk 97a.

Uses cycle_data 23a 26a 42c 43a 54c 54d 55a 56b 56c 56d, ex 41b 47e 47e 47e 53a, is_almost_equal 117a, and nops 50a.

The debug output may be switched on and switched off by the following methods.

119b ⟨addional functions 117a⟩+≡ (52a) ◁119a 119c▷

```
  void figure_debug_on() { FIGURE_DEBUG = true; }
  void figure_debug_off() { FIGURE_DEBUG = false; }
  bool figure_ask_debug_status() { return FIGURE_DEBUG; }
```

Defines:
figure_ask_debug_status, never used.
figure_debug_off, never used.
figure_debug_on, never used.

Uses FIGURE_DEBUG 52c.

Setting variable *show_asy_graphics* to switch `Asymptote` display on and off.

119c ⟨addional functions 117a⟩+≡ (52a) ◁119b

```
void show_asy_on() { show_asy_graphics=true; }
void show_asy_off() { show_asy_graphics=false; }
```

Defines:
`show_asy_off`, never used.
`show_asy_on`, never used.
Uses `show_asy_graphics` 52d.

Appendix G. Change Log

3.2: The following changes are committed:
- Add **figure**::*info_text* to record information for humans.
- Several bugs causing crashes fixed;
- Renamed several methods and members of different classes to avoid confusions and errors.
- Add method *get_all_keys_sorted*(), which sorts output from lower to higher generations. Method **figure**::*do_print*() uses it now for output.
- Better structure of the `Asymptote` output.
- Add **figure**::*get_max_generation*() method.
- Fix archiving/unarchiving of figure.
- **cycle_node** is archiving its custom `Asymptote` style.
- Minor improvements of code and documentation.
- Introduce *do_print_double*() for a more compact output of figures.

3.1: The following changes are committed:
- Updated cycle solver to handle homogeneous equations properly and produce root-free parametrisation in some cases.
- Theoretical aspects are revised in documentation.
- In cycles with numerous instances only corresponding cycles may be checked for a relation.
- Numerous other small improvements.

3.0: The following changes are committed:
- Functions *sl2_clifford*() and *sl2_similarity*() work for hypercomplex matrices as well.
- Cycle library is able to work both in vector and paravector formalisms.
- Add flag *ignore_unit* to **cycle**::*is_equal*().
- Add *with_label* parameter to **figure**::*asy_write*().
- Improved the example with modular group action.
- Numerous small improvements to code and documentations.

2.7: The following changes are committed:
- Container ([lst]) assignments are using curly brackets now.
- Some fixes for upcoming GiNaC 1.7.0.

2.6: The following changes are committed:
- Installation instructions are updated and tested.
- `PyGiNaC` (refreshed) is added as a subproject.

2.5: The following changes are committed:
- Documentation is updated.
- 3D visualiser is added as a subproject.
- Minor fixes and adjustments.

2.4: The following minor changes are committed:
- Embedded PDF animation can be produced.
- Numerous improvements to documentation.

2.3: The following minor changes are committed:
- The stereometric example is done with the symbolic parameter.
- A concise mathematical introduction is written.
- Re-shape code of figure library.
- Use both symbolic and float checks to analyse newly evaluated cycles.
- Some minor code improvements.

2.2: The following minor changes are committed:
- New cycle relations *moebius_transform* and *sl2_transform* are added.
- Example programme with modular group action is added.
- Method *add_cycle_rel* may take a single relation now.
- Numerous internal fixes.

2.1: The following minor changes are committed:
- The method **figure**::*get_all_keys*() is added
- Debug output may be switched on/off from the code.
- Improvements to documentation.
- Initialisation of cycles in Python wrapper are corrected.

2.0: The two-dimension restriction is removed from the **figure** library. This breaks APIs, thus the major version number is increased.

1.0: First official stable release with all essential functionality.

120 ⟨separating chunk 16a⟩+≡ ◁29e

Appendix H. License

This programme is distributed under GNU GPLv3 [19].

121 ⟨license 121⟩≡ (16–18 20e 22c 27b 28a 30a 41c 52a)

```
// The library for ensembles of interrelated cycles in non-Euclidean geometry
//
//  Copyright (C) 2014-2018 Vladimir V. Kisil <kisilv@maths.leeds.ac.uk>
//
//  This program is free software: you can redistribute it and/or modify
//  it under the terms of the GNU General Public License as published by
//  the Free Software Foundation, either version 3 of the License, or
//  (at your option) any later version.
//
//  This program is distributed in the hope that it will be useful,
//  but WITHOUT ANY WARRANTY; without even the implied warranty of
//  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
//  GNU General Public License for more details.
//
//  You should have received a copy of the GNU General Public License
//  along with this program.  If not, see <http://www.gnu.org/licenses/>.
```

Appendix I. Index of Identifiers

School of Mathematics, University of Leeds, Leeds LS2 9JT, England
Email address: kisilv@maths.leeds.ac.uk
URL: http://www.maths.leeds.ac.uk/~kisilv/