

AN EXAMPLE OF CLIFFORD ALGEBRAS CALCULATIONS WITH GiNaC

Vladimir V. Kisil*

School of Mathematics, University of Leeds, Leeds LS2 9JT, UK

e-mail: kisilv@maths.leeds.ac.uk

(Received: February 10, 2005, accepted March 2, 2005)

Abstract. This is an example of C++ code of Clifford algebra calculations with the GiNaC computer algebra system. This code makes both symbolic and numeric computations. It was used to produce illustrations for paper [14, 12].

Described features of GiNaC are already available at PyGiNaC [3] and due to course should propagate into other software like GNU Octave [7] and gTybalt [18] which use GiNaC library as their back-end.

1. Introduction

This example of Clifford algebras calculations uses GiNaC library [1], which includes a support for generic Clifford algebra starting from version 1.3.0. Both symbolic and numeric calculation are possible and can be blended with other functions of GiNaC. Described features of GiNaC are already available at PyGiNaC [3] and due to course should propagate into other software like GNU Octave [7] and gTybalt [18] which use GiNaC library as their back-end.

We bind our C++-code with documentation using noweb [16] within *the literate programming* concept [17]. Our program makes output of two types: some results are typed on screen for information only and the majority of calculated data are stored in files which are lately incorporate by MetaPost [11] to produce PostScript graphics for the paper [14, 12]. Since this code can be treated as software we are pleased to acknowledge that it is subject to GNU General Public License [10].

GiNaC allows to use a generic Clifford algebra, i.e. 2^n dimensional algebra with generators e_k satisfying the identities $e_i e_j + e_j e_i = B(i, j) + B(j, i)$ for some (*metric*) $B(i, j)$, which may be non-symmetric [8, 9] and contain symbolic entries. Such generators are created by the function

```
ex clifford_unit(const ex & mu, const ex & metr, unsigned char rl = 0,  
                bool anticommuting = false);
```

* On leave from Odessa University.

where mu should be **varidx** class object indexing the generators, an index mu with a **numeric** value may be of type idx as well. Parameter $metr$ defines the metric $B(i, j)$ and can be represented by a square **matrix**, **tensormetric** or **indexed** class object, optional parameter rl allows to distinguish different Clifford algebras (which will commute each other). The last optional parameter $anticommuting$ defines if the anticommuting assumption (i.e. $e_i e_j + e_j e_i = 0$) will be used for contraction of Clifford units. If the $metric$ is supplied by a **matrix** object, then the value of $anticommuting$ is calculated automatically and the supplied one will be ignored. One can overcome this by giving $metric$ through matrix wrapped into an **indexed** object.

Note that the call `clifford_unit(mu, minkmetric())` creates something very close to `dirac_gamma(mu)`, although `dirac_gamma` have more efficient simplification mechanism. The method `clifford::get_metric()` returns metric defining this Clifford number. The method `clifford::is_anticommuting()` returns the $anticommuting$ property of a unit.

If the matrix $B(i, j)$ is in fact symmetric you may prefer to create the Clifford algebra units with a call like that

```
ex e = clifford_unit(mu, indexed(B, sy-symm(), i, j));
```

since this may yield some further automatic simplifications. Again, for a metric defined through a **matrix** such a symmetry is detected automatically.

Individual generators of a Clifford algebra can be accessed in several ways. For example

```
{
  ...
  varidx nu(symbol("nu"), 4);
  realsymbol s("s");
  ex M = diag_matrix(lst(1, -1, 0, s));
  ex e = clifford_unit(nu, M);
  ex e0 = e.subs(nu ≡ 0);
  ex e1 = e.subs(nu ≡ 1);
  ex e2 = e.subs(nu ≡ 2);
  ex e3 = e.subs(nu ≡ 3);
  ...
}
```

will produce four generators of a Clifford algebra with properties $e_0^2 = 1$, $e_1^2 = -1$, $e_2^2 = 0$ and $e_3^2 = s$.

A similar effect can be achieved from the function

```
ex lst_to_clifford(const ex & v, const ex & mu, const ex & metr,
                  unsigned char rl = 0, bool anticommuting = false);
ex lst_to_clifford(const ex & v, const ex & e);
```

which converts a list or vector $v = (v_0, v_1, \dots, v_n)$ into the Clifford number $v_0e_0 + v_1e_1 + \dots + v_n e_n$ with e_k directly supplied in the second form of the procedure. In the first form the Clifford unit e_k is generated by *clifford_unit(mu, metr, rl, anticommuting)*. The previous code may be rewritten with help of *lst_to_clifford()* as follows

```
{
  ...
  varidx nu(symbol("nu"), 4);
  realsymbol s("s");
  ex M = diag.matrix(lst(1, -1, 0, s));
  ex e0 = lst_to_clifford(lst(1, 0, 0, 0), nu, M);
  ex e1 = lst_to_clifford(lst(0, 1, 0, 0), nu, M);
  ex e2 = lst_to_clifford(lst(0, 0, 1, 0), nu, M);
  ex e3 = lst_to_clifford(lst(0, 0, 0, 1), nu, M);
  ...
}
```

There is the inverse function

```
lst clifford_to_lst(const ex & e, const ex & c, bool algebraic=true);
```

which took an expression e and tries to find such a list $v = (v_0, v_1, \dots, v_n)$ that $e = v_0c_0 + v_1c_1 + \dots + v_n c_n$ with respect to given Clifford units c and none of v_k contains the Clifford units c (of course, this may be impossible). This function can use an *algebraic* method (default) or a symbolic one. In *algebraic* method v_k are calculated as $(ec_k + c_k e) / pow(c_k, 2)$. If $pow(c_k, 2)$ is zero or is not **numeric** for some k then the method will be automatically changed to symbolic. The same effect is obtained by the assignment (*algebraic=false*) in the procedure call.

There are several functions for (anti-)automorphisms of Clifford algebras:

```
ex clifford_prime(const ex & e)
inline ex clifford_star(const ex & e) { return e.conjugate(); }
inline ex clifford_bar(const ex & e) { return clifford_prime(e.conjugate()); }
```

The automorphism of a Clifford algebra *clifford_prime()* simply changes signs of all Clifford units in the expression. The reversion of a Clifford algebra *clifford_star()* coincides with *conjugate()* method and effectively reverses the order of Clifford units in any product. Finally the main anti-automorphism of a Clifford algebra *clifford_bar()* is the composition of two previous, i.e. makes the reversion and changes signs of all Clifford units in a product. Names for this functions corresponds to notations e' , e^* and \bar{e} used in Clifford algebra textbooks [2, 4, 5].

The function

```
ex clifford_norm(const ex & e);
```

calculates the norm of Clifford number from the expression $\|e\|^2 = e\bar{e}$. The inverse of a Clifford expression is returned by the function

```
ex clifford_inverse(const ex & e);
```

which calculates it as $e^{-1} = e/\|e\|^2$. If $\|e\| = 0$ then an exception is raised.

If a Clifford number happens to be a factor of *dirac_ONE*() then we can convert it to a “real” (non-Clifford) expression by the function

```
ex remove_dirac_ONE(const ex & e);
```

The function *canonicalize_clifford*() works for a generic Clifford algebra in a similar way as for Dirac gammas.

The last provided function is

```
ex clifford_moebius_map(const ex & a, const ex & b, const ex & c,
                      const ex & d, const ex & v, const ex & G,
                      unsigned char rl = 0, bool anticommuting = false);
ex clifford_moebius_map(const ex & M, const ex & v, const ex & G,
                      unsigned char rl = 0, bool anticommuting = false);
```

It takes a list or vector v and makes the Möbius (conformal or linear-fractional) transformation [4]

$$v \mapsto (av + b)(cv + d)^{-1} \quad \text{defined by the matrix } M = \begin{pmatrix} a & b \\ c & d \end{pmatrix}.$$

The matrix may be given in two different forms—as one entity or by its four elements. The last parameter G define the metric of the surrounding (pseudo-)Euclidean space. This can be an indexed object, tensormetric, matrix or a Clifford unit, in the later case the optional parameters rl and *anticommuting* are ignored even if supplied. The returned value of this function is a list of components of the resulting vector.

Finally the function

```
char clifford_max_label(const ex & e, bool ignore_ONE = false);
```

can detect a presence of Clifford objects in the expression e : if such objects are found it returns the maximal *representation_label* of them, otherwise -1. The optional parameter *ignore_ONE* indicates if *dirac_ONE* objects should be ignored during the search.

L^AT_EX output for Clifford units looks like `\clifford[1]{e}^{\{\nu\}}`, where 1 is the *representation_label* and ν is the index of the corresponding unit. This provides a flexible typesetting with a suitable definition of the `\clifford` command. For example, the definition

```
\newcommand{\clifford}[1] [] {}
```

typesets all Clifford units identically, while the alternative definition

```
\newcommand{\clifford}[2] [] {\ifcase #1 #2\or \tilde{#2}
\or \breve{#2} \fi}
```

prints units with *representation_label*=0 as e , with *representation_label*=1 as \tilde{e} and with *representation_label*=2 as \breve{e} .

2. Main procedure

Here is the main procedure, which has a very straightforward structure. This and next initialisation section is pretty standard. The first usage of GiNaC for Clifford algebras is in Section 4.

```
<*)≡
<Includes>
<Definitions>
<Global items>
int main (int argc, char**argv)
{
  <C++ variables declaration>
  <GiNaC variables declaration>
  <Pictures tuning>
  <Parabola parameters>
```

Now we run a cycle over the three possible type of metric in two dimensional space (i.e. *elliptic*, *parabolic* and *hyperbolic*). For each space we initialise the corresponding Clifford units, symbolically calculate various types of Möbius transforms as well as vector fields for three subgroups of $SL_2(\mathbb{R})$.

```
<*)+≡
for (metric = elliptic; metric ≤ hyperbolic; metric++) {
  cout << endl << endl << "Metric is: " << metric_name[metric] << "." << endl;
  <Initialise Clifford numbers>
  <Calculation of Moebius transformations>
  <Calculation of vector fields>
```

Then we run a cycle for three subgroups of $SL_2(\mathbb{R})$ (i.e. A, N, K). For all possible combinations of those with *metric* we

- build orbits of the subgroups and their transversal curves;
- two types of the Cayley transform images of all above curves;
- check some formulae in the paper;

```

(*)+=
    for (subgroup = subgroup_A; subgroup ≤ subgroup_K; subgroup++) {
        (Drawing arrows)
        (Building orbits)
        (Building transverses)
    }
}

```

Finally we draw eight frames which illustrates the continuous transformation of the future part of the light cone into the its past part [12, Figure 4].

```

(*)+=
    (Build future-past transition)
}

```

3. Auxiliary matter

We use GiNaC library and *sqrt()* function from *<cmath>* in one place.

```

(Includes)=
#include <ginac/ginac.h> // At least ver. 1.3.3!
#include <cmath>
using namespace std;
using namespace GiNaC;

```

3.1. DEFINES

Some constants are defined here for a better readability of the code.

```

(Definitions)=
// Defined constants
#define elliptic 0
#define parabolic 1
#define hyperbolic 2
#define subgroup_A 0
#define subgroup_N 1
#define subgroup_K 2
#define grey 0.6

```

Some macro definitions which we use to make more compact code. They initialise variables, open and close curve description in the `MetaPost` file. Here is initialisation of a new curve.

```

(Definitions)+≡
#define init_coord(X) upos[X] = 0;          \
    vpos[X] = 0; \
    udir[X] = 1; \
    vdir[X] = 0; \
    fprintf(fileout[X], "draw ")

```

This part is used to close a curve output in cases the end is reached or curves passes the infinity.

```

(Definitions)+≡
#define close_curve(X) fprintf(fileout[X], \
    "(a%+5.3f,b%+5.3f)*u withcolor %5.3f*%s;\n", \
    upos[X], vpos[X], color_grade, color_name[subgroup])
#define put_draw(X) fprintf(fileout[X], "\ndraw ")

```

Here is the common part, which is used for outputs a segment to files.

```

(Definitions)+≡
#define put_point(X) if (inversion) \
    fprintf(fileout[X], "(a%+5.3f,b%+5.3f)*u...", upos[X], vpos[X]); \
else if (direct) \
    fprintf(fileout[X], "(a%+5.3f,b%+5.3f)*u{(%+5.3f,%+5.3f)}...", upos[X], \
        vpos[X], udir[X], vdir[X]); \
else \
    fprintf(fileout[X], "(a%+5.3f,b%+5.3f)*u{(%+5.3f,%+5.3f)}...", upos[X], \
        vpos[X], trans_uf, trans_vf);

```

To restart curve we close the previous one and put new `draw` command.

```

(Definitions)+≡
#define renew_curve(Y)      close_curve(Y); \
    put_draw(Y)

```

We should make a rough check that the curve is still in the bounded area, if it cross infinity then such line should be discontinued and started from a new. Our bound are few times bigger that the real picture, the excellent cutting within the desired limits is done by `MetaPost` itself with the `clip currentpicture to ...;` command.

Besides some outer margins we put different types of bound depending from the nature of objects: sometimes it is limited to the upper half plane, sometimes to hyperbolic unit disk. The necessity of such checks in the hyperbolic case is explained in [12, § 2.5].

```

(Definitions)+≡
#define if_in_limits(X) if ( (abs(u_res.to_double()) <= ulim) \
  ^ (abs(v_res.to_double()) ≤ vlim) \
  ^ ((metric ≠ hyperbolic) \
    ∨ inversion \
    ∨ (¬cayley ^ (v_res.is_positive() ∨ v_res.is_zero())) \
    ∨ (cayley ^ ¬ex_to<numeric>(-pow(u_res,2)+pow(v_res,2)-1.001).is_positive())) { \
  upos[X] = u_res.to_double(); \
  vpos[X] = v_res.to_double(); \
  ex Vect = dV[subgroup][X].subs(1st(x ≡ u_res, y ≡ v_res)); \
  udir[X] = ex_to<numeric>(Vect.op(0)).to_double(); \
  vdir[X] = ex_to<numeric>(Vect.op(1)).to_double(); \
  put_point(X); \
} else { \
  renew_curve(X); \
}

```

Then a curve is going through infinity we catch the exception, close the corresponding `draw` statement of `MetaPost` and start a new one from the next point.

```

(Definitions)+≡
#define catch_handle(X) cerr <<"*** Got problem: " <<p.what() <<endl;\
  renew_curve(X)

```

Extracting of numerical values out of Moebius transformations

```

(Definitions)+≡
#define get_components u_res = ex_to<numeric>(res.op(0).evalf()); \
  v_res = ex_to<numeric>(res.op(1).evalf())

```

To make an accurate drawing we calculate the direction of a transverse line out of symbolic vector fields calculation done before.

```
(Definitions)+≡
#define transverse_dir(X)  if (!direct) { \
  trans_uf = ex_to<numeric>(trans_dir_sub[X].op(0).subs(a_node).evalf()).to_double(); \
  trans_vf = ex_to<numeric>(trans_dir_sub[X].op(1).subs(a_node).evalf()).to_double(); \
  if (trans_uf ≡ INFINITY) { trans_uf = 1; trans_vf = 0; } \
  else if (trans_uf ≡ -INFINITY) { trans_uf = -1; trans_vf = 0; } \
  else if (trans_vf ≡ INFINITY) { trans_uf = 0; trans_vf = 1; } \
  else if (trans_vf ≡ -INFINITY) { trans_uf = 0; trans_vf = -1; } \
  else if (abs(trans_uf)+abs(trans_vf) > 100) { \
    double r = sqrt(trans_uf * trans_uf + trans_vf * trans_vf); \
    trans_uf ÷= r; trans_vf ÷= r; \
  } \
}
```

Global variables which are used for the *openfile()* function below.

```
(Global items)≡
int subgroup, metric; // Subgroup iterator and sign of the metric of the space
numeric signum;
char sgroup[]="ANK", metric_name[]="eph"; // Names used for a readable output
```

Procedure *openfile()* is used for opening numerous data files with predefined template of name.

```
(Global items)+≡
FILE *openfile(const char *F) {
  char filename[]="cayley-t-k-e.d", templ[]="cayley-t-%.1s-%.1s.d";
  char *Sfilename=filename, *Stempl=templ;
  strcat(strcpy(Stempl, F), "-%.1s-%.1s.d");
  sprintf(Sfilename, Stempl, &sgroup[subgroup], &metric_name[metric]);
  return fopen(Sfilename, "w");
}
```

3.2. VARIABLES

First we define variables from the standard C++ classes.

```
(C++ variables declaration)≡
FILE *fileout[3]; /* files to pass results to \MetaPost\ */
static char *color_name[]={ "hyp", "par", "ell", "white"}, // subgroup colours
*formula[]={ "\nDistance to center is:", "\nDirectrice is:",
"\nDifference to foci is:"};
```

Some variables to keep track on logic.

```

(C++ variables declaration)+≡
  bool direct = true, // Is it orbit or transverse?
        cayley = false, // Is it the Cayley transform image?
        inversion = false; //Is it future-past inversion?

```

Here are variables for coordinates of point, vector, etc .

```

(C++ variables declaration)+≡
  float u, v, upos[3], vpos[3], udir[3], vdir[3], vval = 0,
        color_grade, focal_f[2] = {0, 0},
        trans_uf=1, trans_vf=0;

```

Then other variables of GiNaC types are defined as well. They are used for numeric and symbolic calculations

```

(CiNaC variables declaration)≡
  varidx nu(symbol("nu", "\\nu"), 2), mu(symbol("mu", "\\mu"), 2),
        psi(symbol("psi", "\\psi"),2), xi(symbol("xi", "\\xi"), 2);
  realsymbol x("x"), y("y"), t("t"), // for symbolic calculations
        a("a"), b("b"), c("c"), // parameters of the parabola  $v = au^2 + bu + c$ 
        tr_u("U"), tr_v("V"); // Vector ot the tranverse direction
  lst a_node, soln[2], a_trans;

```

```

(CiNaC variables declaration)+≡
  matrix M(2, 2), // The metric of the vector space
        C(2, 2), CI(2, 2), CI(2, 2), CII(2, 2), // Two versions of the Cayley transform
        T(2, 2), TI(2, 2), // The map from first to second Cayley transform
        Jacob[3][3]={matrix(2,2)}, // Jacobian of the Moebius transformation
        trans_dir[3][3]={matrix(2,1)}, // Components of transverse direction
        trans_dir_sub[3]={matrix(2,1)}; // Components of transverse direction

```

```

(CiNaC variables declaration)+≡
  numeric u_res, v_res, //coordinates of the Moebius transform
        up[3][2] = {0, 0, 0, 0, 0, 0}, // saved values of coordinates of the parabola
        vp[3][2] = {0, 0, 0, 0, 0, 0};
  const numeric half(1, 2);

```

```

(CiNaC variables declaration)+≡
  ex res, e, e0, e1,
        Moebius[3][5], dV[3][3],
  // indexed by [subgroup], [type](=direct, Cayley-operator, Cayley1-op, C-point, C1-p)
  ddV[3], Curv[3], // indexed by [type](=direct, Cayley-operator, Cayley1-op)
  focal, p,
  focal_l, focal_u, focal_v; // parameters of parabola given by  $v = au^2 + bu + c$ 

```

Here is the set of constants which allows to fine tune **MetaPost** output depending from the type of *subgroup* and *metric* used. The quality of pictures will significantly depend from the number of points chosen for iterations: too much lines will mess up the picture, very few makes it incomplete or insensitive to the singular regions. These numbers should be different for different combinations of *subgroup* and *metric*.

```

(Pictures tuning)≡
  const int vilimits[3][3] = {10, 20, 30, // indexed by [subgroup], [metric]
    10, 10, 19,
    10, 10, 10},
  fsteps[3][3] = {15, 15, 20, // indexed by [subgroup], [metric]
    15, 10, 20,
    12, 15, 15},
  float ulim = 25, vlim = 25,
  flimits[3][3] = {2.0, 2.0, 4.0, // indexed by [subgroup], [metric]
    10.0, 4.0, 4.0,
    0.5, 0.5, 0.5},
  vpoints[3][10] = {0, 1.0÷8, 1.0÷4, 1.0÷2, 1.0, 2.0, 3.0, 5.0, 8.0, 16.0, //[metric][point]
    0, 1.0÷8, 1.0÷4, 1.0÷2, 1, 2.0, 3.0, 6.0, 10.0, 20.0,
    0, 1.0÷8, 1.0÷4, 1.0÷2, 1.0, 2.0, 3.0, 5.0, 10.0, 100};

```

Initialise the set of coordinates for a cycle

```

(Initialisation of coordinates)≡
  init_coord(0);
  init_coord(1);
  put_draw(2);

```

4. Symbolic Clifford Algebra Calculations

This section finally starts to deal with Clifford algebras. We try to make all possible calculations symbolically at first instance and delay the numeric substitution to the latest stage. Besides a high accuracy of calculations this produces a faster code as well.

4.1. INITIALISATION OF CLIFFORD NUMBERS

We initialise Clifford numbers first.

```

<Initialise Clifford numbers>≡
  signum = numeric(metric-1); // the value of  $e_2^2$ 
  M = -1, 0, 0, signum;
  e = clifford_unit(mu, M);
  e0 = e.subs(mu ≡ 0);
  e1 = e.subs(mu ≡ 1);

```

There are alternative ways to define e , $e0$ and $e1$ which will work as well:

```

e = clifford_unit(mu, indexed(M, symmetric2(xi, psi)));
e0 = lst_to_clifford(lst(1.0, 0), mu, M);
e1 = lst_to_clifford(lst(0, 1.0), mu, M);

```

Now we define matrices used for definition of the alternative Cayley transforms [14, 12].

```

<Initialise Clifford numbers>+≡
  T = dirac_ONE(), e0, // Transformation to the alternative Cayley map
  e0, dirac_ONE(); // is given by  $\begin{pmatrix} 1 & e_1 \\ e_1 & 1 \end{pmatrix}$ 
  TI = dirac_ONE(), -e0, // The inverse of T
  -e0, dirac_ONE(); // is given by  $\begin{pmatrix} 1 & -e_1 \\ -e_1 & 1 \end{pmatrix}$ 

```

A form of matrix for the Cayley transform depends from the type of metric.

```

<Initialise Clifford numbers>+≡
  switch (metric) {
  case elliptic:
  case hyperbolic:
    C = dirac_ONE(), -e1, // First Cayley transform  $\begin{pmatrix} 1 & -e_2 \\ \sigma e_2 & 1 \end{pmatrix}$ 
    signum*e1, dirac_ONE();
    CI = dirac_ONE(), e1, // The inverse of C  $\begin{pmatrix} 1 & e_2 \\ -\sigma e_2 & 1 \end{pmatrix}$ 
    -signum*e1, dirac_ONE();
    C1 = C.mul(T); // Second Cayley transform
    C1I = TI.mul(CI); // The inverse of C1
  break;

```

In the parabolic case there are two different (elliptic and hyperbolic) types of the Cayley transform [14, 12].

```

<Initialise Clifford numbers>+≡
  case parabolic:
    C = dirac_ONE(), -e1*half, // First parabolic-elliptic Cayley transform
      -e1*half, dirac_ONE();
    CI = dirac_ONE(), e1*half, // The inverse of C
      e1*half, dirac_ONE();
    C1 = dirac_ONE(), -e1*half, // Second parabolic-hyperbolic Cayley transform
      e1*half, dirac_ONE();
    C1I = dirac_ONE(), e1*half, // The inverse of C1
      -e1*half, dirac_ONE();
  break;
}

```

4.2. MÖBIUS TRANSFORMATIONS

We calculate all Moebius transformations along the orbits as well as two their Cayley transforms only once in a symbolic way. Their usage will be made through the GiNaC substitution mechanism.

First, we define matrices Exp_A , Exp_N , Exp_K [15, VI.1] related to the Iwasawa decomposition of $SL_2(\mathbb{R})$ [15, § III.1].

```

<Calculation of Moebius transformations>≡
  matrix Exp_A(2, 2), Exp_N(2, 2), Exp_K(2, 2);
  Exp_A = exp(t) * dirac_ONE(), 0, // Matrix
    e^t  0
    0  e^-t  = exp  1  0
                    0 -1
    0, exp(-t) * dirac_ONE();
  Exp_N = dirac_ONE(), t * e0, // Matrix
    0  t
    0  0  = exp  0  1
                    0  0
    0, dirac_ONE();
  Exp_K = cos(t) * dirac_ONE(), sin(t) * e0, //
    cos t  -sin t
    sin t  cos t  = exp  0 -1
                    1  0
    sin(t) * e0, cos(t) * dirac_ONE();
  ex Exp[3] = {Exp_A, Exp_N, Exp_K};
  matrix E(2, 2);

```

Here we symbolically calculate the related Möbius transformations, as well as its two images under Cayley transform C and CI for operators and finally two Cayley images for points.

```

(Calculation of Moebius transformations)+≡
  for (subgroup = subgroup-A; subgroup ≤ subgroup-K; subgroup++) {
    try {
      Moebius[subgroup][0] = clifford_moebius_map(Exp[subgroup], lst(x, y), M);
      /* Cayley transforms of operators */
      Moebius[subgroup][1] = clifford_moebius_map(canonicalize_clifford(
        C.mul(ex_to<matrix>(Exp[subgroup]).mul(CI))), lst(x, y), M);
      Moebius[subgroup][2] = clifford_moebius_map(canonicalize_clifford(
        CI.mul(ex_to<matrix>(Exp[subgroup]).mul(CI))), lst(x, y), M);
      /* Cayley transforms of points */
      Moebius[subgroup][3] = clifford_moebius_map(C.mul(
        ex_to<matrix>(Exp[subgroup])), lst(x, y), M);
      Moebius[subgroup][4] = clifford_moebius_map(canonicalize_clifford(
        CI.mul(ex_to<matrix>(Exp[subgroup]))), lst(x, y), M);
    } catch (exception &p) {
      cerr << "*** Got problem in vector fields: " << p.what() << endl;
    }
  }

```

4.3. SYMBOLIC CALCULATIONS OF THE VECTOR FIELDS

We calculate symbolic expressions for the vector fields of three subgroups A , N and K , which are stated in [12, Lemmas 2.1 and 2.2] and shown on [12, Figures 1 and 2]. The formula for a derived representation $\rho(X)$ of a vector field X used here is [15, § VI.1]:

$$\rho(X) = \frac{d}{dt} \rho(e^{tX}) \Big|_{t=0}.$$

Results of calculations are directed to *stdout* and will be used for a better drawing of orbits, see 3.1.

We calculate *Jacobian* of the Möbius transformations in order to supply to *MetaPost* the tangents of the transverse lines.

```

(Calculation of vector fields)≡
try {
  cout << "Vect field \t Direct \t\t In Cayley \t\t In Cayley1" << endl;
  for (subgroup = subgroup_A; subgroup ≤ subgroup_K; subgroup++) {
    for (int cayley = 0; cayley < 3 ; cayley++) {
      lst Moeb = ex_to<lst>(Moebius[subgroup][cayley?cayley+2:0]);
      dV[subgroup][cayley] = Moebius[subgroup][cayley].diff(t).subs(t ≡ 0);
      Jacob[subgroup][cayley] = matrix(2,2, lst(Moeb.op(0).diff(x), Moeb.op(0).diff(y),
        Moeb.op(1).diff(x), Moeb.op(1).diff(y)));
      for (int i=0; i < 2; i++) // Transformation of a direction by a Jacobian
        trans_dir[subgroup][cayley] = Jacob[subgroup][cayley].mul(
          matrix(2, 1, lst(tr_u, tr_v)));
    }
    cout << " d" << sgroup[subgroup] << " is:\t" << dV[subgroup][0]
      << "\t" << dV[subgroup][1] << "\t " << dV[subgroup][2] << endl;
  }
}

```

We also calculate curvature of the orbits using the formula [6, § 5.1(20)]

$$k = \frac{|\ddot{x}\dot{y} - \dot{x}\ddot{y}|}{(\dot{x}^2 + \dot{y}^2)^{3/2}}$$

```

(Calculation of vector fields)+≡
for (int cayley = 0; cayley < 3 ; cayley++) {
  ddV[cayley] = Moebius[subgroup_K][cayley].diff(t, 2).subs(t ≡ 0);
  ex du = ex_to<lst>(dV[subgroup_K][cayley]).op(0);
  ex dv = ex_to<lst>(dV[subgroup_K][cayley]).op(1);
  ex ddu = ex_to<lst>(ddV[cayley]).op(0);
  ex ddv = ex_to<lst>(ddV[cayley]).op(1);
  Curv[cayley] = normal((ddu * dv - du * ddv) ÷ pow(du*du+dv*dv, 1.5));
}
cout << "Curvature of K-orbits: " << normal(Curv[0].subs(x ≡ 0)) << endl;
} catch (exception &p) {
  cerr << "*** Got problem in vector fields: " << p.what() << endl;
}

```

5. Numeric Calculations with Clifford Algebras

Numeric calculations are done in to fashions:

1. Through a substitution of numeric values to symbols in some previous symbolic results, subsection 5.1 and 5.5;
2. Direct calculations with numeric GiNaC classes, subsection 5.3.

The first example of substitution approach is the drawing of vector fields. Three vector fields are drawn by arrows into a `MetaPost` file.

```

(Drawing arrows)≡
fileout[0] = openfile("arrows"); // open MetaPost file
color_grade = grey;
for (int k = -10; k < 10; k++) // cycle over horizontal dir
  for (int j = 0; j < 11; j++) { // cycle over vertical dir
    u = k÷3.0; // Calculate coordinates of the point
    v = j÷3.0;
    u_res = ex.to<numeric>(dV[subgroup][0].subs(lst(x ≡ u, y ≡ v)).op(0));
    v_res = ex.to<numeric>(dV[subgroup][0].subs(lst(x ≡ u, y ≡ v)).op(1));
    fprintf(fileout[0],
      "myarrow ((a%+5.3f,b%+5.3f), (a%+5.3f%+5.3f*s,b%+5.3f%+5.3f*s))",
      u, v, u, u_res.to_double(), v, v_res.to_double());
    fprintf(fileout[0], " withcolor %5.3f*s;\n", color_grade, color_name[3]);
  }
fclose(fileout[0]);

```

5.1. NUMERIC CALCULATIONS OF ORBITS AND TRANSVERSES

For any of three possibility $e_2^2 = -1, 0, 1$ and three possible subgroups (A, N, K) orbits are constructed. First we open output `MetaPost` files.

```

(Building orbits)≡
direct = true;
fileout[0] = openfile("orbit");
fileout[1] = openfile("cayley"); // Cayley transform of the orbits
fileout[2] = openfile("cayl-a"); // Alternative Cayley transform of the orbits

```

This chunk runs iterations over the different orbits, which are initiated by the point vi .

```

⟨Building orbits⟩≡
  for (int vi = 0; vi < vilimits[subgroup][metric]; vi++) { // iterator over orbits
    color_grade = 1.2*vi÷vilimits[subgroup][metric];
    if (subgroup ≡ subgroup_K)
      cout << formula[metric] ;
    ⟨Initialisation of coordinates⟩
    ⟨Nodes iterations⟩
    ⟨Close all curves⟩
    ⟨Check parabolas⟩
  }
  ⟨Closing all files⟩

```

Each orbit is processed by iteration over the “time” parameter j on the orbit. For each node on an orbit an entry is put into appropriate MetaPost file, formulae from papers are numerically checked and all Cayley transforms are produced.

```

⟨Nodes iterations⟩≡
  for (int j = -fsteps[subgroup][metric]; j ≤ fsteps[subgroup][metric]; j++ ) {
    float f = flimits[subgroup][metric]*j÷fsteps[subgroup][metric]; // the angle of rotation
    ⟨Generating one entry⟩
    ⟨Check formulas in the paper⟩
    ⟨Producing Cayley transform of the orbit⟩
  }

```

Closing all files when finishing drawing orbits or transverses

```

⟨Closing all files⟩≡
  fclose(fileout[0]);
  fclose(fileout[1]);
  fclose(fileout[2]);

```

5.2. BUILDING OF TRANSVERSES

Construction of transverses to the orbits follows the same structure as for orbits themselves with the changed order of iterations over time parameter and orbit origins. We again start from opening of the corresponding files.

```

⟨Building transverses⟩≡
  direct = false;
  fileout[0] = openfile("orbit-t");
  fileout[1] = openfile("cayley-t"); // Cayley transform of transverses
  fileout[2] = openfile("cay1-a-t"); // Alternative Cayley transform of transverses
  color_grade = 1.2;
  ⟨Define transverse directions⟩;

```

Thus chunk performs iterations over the transverse lines.

```

⟨Building transverses⟩+≡
  for (int j = -fsteps[subgroup][metric]; j ≤ fsteps[subgroup][metric]; j++) {
    float f = flimits[subgroup][metric]*j÷fsteps[subgroup][metric]; // the angle of rotation
    ⟨Initialisation of coordinates⟩
    for (int vi = 0; vi < vilimits[subgroup][metric]; vi++) { // iterator over orbits
      vval = vpoints[metric][vi];
      ⟨Generating one entry⟩
      ⟨Producing Cayley transform of the orbit⟩
    }
    ⟨Close all curves⟩
  }
  ⟨Closing all files⟩

```

All MetaPost `draw` statements should be closed at the end of run.

```

⟨Close all curves⟩≡
  close_curve(0);
  close_curve(1);
  close_curve(2);

```

5.3. FUTURE-TO-PAST TRANSFORMATIONS

We finish our code by producing a set of frames of transformations from future to past of light cone. First we create a necessary *hyperbolic* setup and use *subgroup_K* to fill in parts of the light cone.

```

⟨Build future-past transition⟩≡
  const int curves = 15, nodes = 40, frames = 8;
  const float exp_scale = 1.3, node_scale = 4.0,
    rad[] = {1.0÷5, 1.0÷4, 1÷3.5, 1.0÷3, 1÷2.5, 1.0÷2,
             1÷1.5, 1, 1.5, 2, 2.5, 3, 3.5, 4, 4.5, 5};
  char name[] = "future-past-00.d";
  char* S = name;
  ulim = 8.5;
  vlim = 8.5;

⟨Build future-past transition⟩+≡
  metric = hyperbolic;
  subgroup = subgroup_K;
  direct = true;
  inversion = true; // to cut around unit circle.
  ⟨Initialise Clifford numbers⟩
  ex Fut = clifford_moebius_map(dirac_ONE(), -a * e1,
                               a * e1, dirac_ONE(), lst(x, y), M);

```

Then we proceed with iteration over the frames.

```

⟨Build future-past transition⟩+≡
  for (int j = 0; j < frames; j++) { // the power of transformation
    sprintf(S, "future-past-%.2d.d", j);
    fileout[0] = fopen(S, "w");
    for (int k = 0; k < curves; k++) { // the number of curve
      color_grade = k÷frames;
      init_coord(0);
      ⟨Iteration over a curve⟩
      close_curve(0);
    }
    fclose(fileout[0]);
  }
  cout << endl;

```

A curve is created by rotation in hyperbolic metric and then a Möbius transformations corresponding to the frame number is applied.

```

⟨Iteration over a curve⟩≡
  for (int l = -nodes÷2; l ≤ nodes÷2; l++) // the node number
    try { // There is a chance of singularity!
      float angl = ((j > 0) ? exp(double(j÷exp_scale-3)) : 0);
      res = Fut.subs(1st( a ≡ angl, x ≡ rad[k]*cosh(l÷node_scale),
                        y ≡ rad[k]*sinh(l÷node_scale)));

      get_components;
      if_in_limits(0);
    } catch (exception &p) {
      catch_handle(0);
    }
}

```

5.4. SINGLE NODE CALCULATION

This is a common portion of code for building orbits and transverses. A single entry into `MetaPost` file is calculated and written. Calculations depend for three possible values of the *subgroup*. For each of them we create a special list *a_node* of substitutions for the already symbolically calculated *Moebius*[][].

For subgroup *A* the points are distributed evenly on the unit circle.

```

⟨Generating one entry⟩≡
  switch (subgroup) {
  case subgroup_A:
    vval = 1.0*vi÷(vilimits[subgroup][metric]-1);
    if (metric ≡ hyperbolic) // we need a double set of value for negatives as well
      vval *= 2;
    a_node = 1st(t ≡ f, x ≡ cos(Pi*vval), y ≡ sin(Pi*vval));
    break;

```

For the subgroups *K* and *N* points are distributed evenly on the vertical axis.

```

⟨Generating one entry⟩+≡
  case subgroup_K:
    vval = vpoints[metric][vi];
    a_node = 1st(t ≡ (f * Pi), x ≡ 0, y ≡ vval);
    break;

```

For the subgroup N we need additional points.

```

⟨Generating one entry⟩+≡
  case subgroup_N:
    if (metric ≡ hyperbolic) { // we need a double set of value for negatives as well
      vval = ( ((vi - vilimits[subgroup][metric]÷2) < 0) ? -1 : 1)
        *vpoints[metric][abs(vi - vilimits[subgroup][metric]÷2)];
    } else
      vval = vpoints[metric][vi];
    a_node = lst(t ≡ f, x ≡ 0, y ≡ vval);
    break;
  }

```

And now using values stored above in a_node we do the actual calculation through substitution and write the node into the **MetaPost** file.

```

⟨Generating one entry⟩+≡
  try{
    res = Moebius[subgroup][0].subs(a_node);
    transverse_dir(0);
    get_components;
    if_in_limits(0);
  } catch (exception &p) {
    catch_handle(0);
  }

```

We define the tangents to the transverse lines here.

```

⟨Define transverse directions⟩≡
  switch (subgroup) {
  case subgroup_A:
    a_trans = lst(tr_u ≡ -y, tr_v ≡ x);
    break;
  case subgroup_N:
  case subgroup_K:
    a_trans = lst(tr_u ≡ 0, tr_v ≡ 1);
    break;
  }

```

And here again comes evaluation through substitution.

```

⟨Define transverse directions⟩+≡
  for (int cal=0; cal < 3; cal++)
    trans_dir_sub[cal] = matrix(2, 1,
      lst(trans_dir[subgroup][cal].op(0).subs(a_trans).normal(),
        trans_dir[subgroup][cal].op(1).subs(a_trans).normal()));

```

5.5. CAYLEY TRANSFORMS OF IMAGES

We will need a calculation parameters of parabola into the Cayley transform images. This checks the statement [12, Lemma 2.17]. The calculation is done by linear equation solver *lsolve()* from GiNaC.

```

(Definitions)+≡
  #define calc_par_focal(X) if (direct && (metric == parabolic) \
    ^ (subgroup ≠ subgroup_K)) { \
    up[2][X] = u_res; \
    vp[2][X] = v_res; \
    if (j ≡ 1) { \
      lst eqns, vars ; \
      vars = a, b, c; \
      eqns = a*pow(up[0][X], 2) + b*up[0][X] + c ≡ vp[0][X], \
            a*pow(up[1][X], 2) + b*up[1][X] + c ≡ vp[1][X], \
            a*pow(up[2][X], 2) + b*up[2][X] + c ≡ vp[2][X]; \
      soln[X] = ex.to<lst>(lsolve(eqns, vars)); \
    } \
    /* After calculation is made we store previous values for the next round. */ \
    up[0][X] = up[1][X]; \
    vp[0][X] = vp[1][X]; \
    up[1][X] = up[2][X]; \
    vp[1][X] = vp[2][X]; \
  }

```

We produce two versions of the Cayley transforms for each node of the orbit or transverses lines. This done by simple substitution of *a_node* into *Moebius*[[3,4] symbolically calculated in subsection 4.2.

```

(Producing Cayley transform of the orbit)≡
  cayley = true;
  try { // There is a chance of singularity!
    res = Moebius[subgroup][3].subs(a_node);
    transverse_dir(1);
    get_components;
    if_in_limits(1);
    calc_par_focal(0);
  } catch (exception &p) {
    catch_handle(1);
  }

```

For second type of the Cayley transforms we perform an extra run similar to the above.

```

⟨Producing Cayley transform of the orbit⟩+≡
  try {
    res = Moebius[subgroup][4].subs(a_node);
    transverse_dir(2);
    get_components;
    if_in_limits(2);
    calc_par_focal(1);
  } catch (exception &p) {
    catch_handle(2);
  }
  cayley = false;

```

5.6. NUMERIC CHECK OF FORMULAE

Here is a numeric check of few formulas in the paper about radius and focal length of sections. We calculate focal properties for three types of orbits (circles, parabolas and hyperbolas) of the *subgroup_K*.

```

⟨Check formulas in the paper⟩+≡
  if ((j ≠ -fsteps[subgroup][metric]) ∧ (j ≠ fsteps[subgroup][metric])) // Exclude end-points
    ∧ (subgroup ≡ subgroup_K) ∧ (vval ≠ 0) { // only for that values
    try {
      switch (metric) { // depends from the type of metric

```

The values are calculated for each node on the orbit as follows, see [12, Lemma 2.2]. For *elliptic* orbits of *subgroup_K*: a circle with the centre at $(0, (v + v^{-1})/2)$ and the radius $(v - v^{-1})/2$.

```

⟨Check formulas in the paper⟩+≡
  case elliptic:
    focal.f[1] = ex.to<numeric>(pow(u_res*u_res
      + pow(v_res-(vval+1÷vval)÷2, 2), 0.5)).to_double();
    break;

```

For *parabolic* orbits of *subgroup_K*: a parabola with the focus at $(0, (v + v^{-1})/2)$ and focal length $v^{-1}/2$.

```

⟨Check formulas in the paper⟩+≡
  case parabolic:
    focal.f[1] = ex.to<numeric>(pow(u_res*u_res
      + pow(v_res-(vval+1÷vval÷4), 2), 0.5)-v_res).to_double();
    break;

```

For *hyperbolic* orbits of *sungroup_K*: a hyperbola with the upper focus located at $(0, f)$ with:

$$f = \begin{cases} p - \sqrt{\frac{p^2}{2} - 1}, & \text{for } 0 < v < 1; \text{ and} \\ p + \sqrt{\frac{p^2}{2} - 1}, & \text{for } v \geq 1. \end{cases}$$

and has the focal distance between focuses $2p$.

```

<Check formulas in the paper>+≡
  case hyperbolic:
    p = (vval*vval+1)÷vval÷pow(2,0.5);
    focal = ((vval<1) ? p -pow(p*p÷2-1,0.5) : p+pow(p*p÷2-1,0.5));
    focal.f[1] = ex.to<numeric>(pow(u_res*u_res + pow(v_res-focal, 2) , 0.5)
      -pow(u_res*u_res + pow(v_res-focal+2*p, 2) , 0.5)).to_double();
    break;
  }

```

If the obtained value is reasonably close to the previous one then = sign is printed to *stdout*, otherwise the new value is printed. This produces lines similar to the following:

```
Distance to center is: 3.938=====
```

```

<Check formulas in the paper>+≡
  if ((abs(focal.f[1] - focal.f[0]) < 0.001)
    ∨ (abs(focal.f[1] - focal.f[0]) < 0.001*(abs(focal.f[1])+abs(focal.f[0]))))
    cout << "=";
  else
    printf(" %5.3f", focal.f[1]);
    focal.f[0] = focal.f[1];
  } catch (exception &p) {
    cerr << "*** Got problem in formulas: " << p.what() << endl;
  }
}

```

Note that all check are passed smoothly (see Appendix A), however in the *hyperbolic* case there is “V” shape of switch from positive values to negative and back (with the same absolute value) like this:

Difference to foci is: 2.000===== -2.000===== 2.000=====

This is related to the non-invariance of the upper half plane in the *hyperbolic* [12, § 2.5].

The last check we make is about some properties of Cayley transform in *parabolic* case. All parameters of parabolic orbits were calculated in Subsection 5.5, now we check properties listed in [12, Lemma 2.17].

```
(Definitions)+≡
#define output_focal(X) \
  ex.to<numeric>(focal_u.subs(soln[X]).evalf()).to_double(), \
  ex.to<numeric>(focal_v.subs(soln[X]).evalf()).to_double(), \
  ex.to<numeric>(focal_l.subs(soln[X]).evalf()).to_double()
```

Here is expressions for focal length *focal_l* and vertex (*focal_u*, *focal_v*) of a parabola given by its equation $v = au^2 + bu + c$.

```
(Parabola parameters)≡
focal_l = 1÷(4*a); // focal length
focal_u = b÷(2*a); // u comp of focus
focal_v = c-pow(b÷(2*a), 2); // v comp of focus
```

Properties of parabolas are printed to *stdout* in the form:

```
Parab (A/7/0.368); vert=(1.140,-2.299); l=0.2500;
          second vert=(-1.140,-2.299); l=-0.2500
```

The two vertexes correspond to two Cayley transformations P_e defined by C and P_h defined by CI , see [12, § 2.6].

```
(Check parabolas)≡
if ((metric ≡ parabolic) ∧ (subgroup ≠ subgroup_K))
  try {
    printf("\nParab (%.1s/%2d/% 5.3f); vert=(% 5.3f, % 5.3f); l=% 5.4f",
      &sgroup[subgroup], vi, vval, output_focal(0));
    printf("; second vert=(% 5.3f, % 5.3f); l=% 5.4f", output_focal(1));
```

In the case of *subgroup_A* an additional line

```
Check vertices: -1 and -1
```

is printed. It confirms that vertexes of the orbits under the Cayley transform do belong to the parabolas $v = \pm v^2 - 1$, as stated in [12, 2.17].

```
(Check parabolas)+≡
  if (subgroup ≡ subgroup_A)
    cout << "\nCheck vertices: "
      << ex.to<numeric>(focal.v.subs(soln[0])
        + pow(focal.u.subs(soln[0]), 2).evalf()).to_double()
      << " and "
      << ex.to<numeric>(focal.v.subs(soln[1])
        + pow(focal.u.subs(soln[1]), 2).evalf()).to_double();
  } catch (exception &p) {
    printf("\nParab (%.1s/%2d/%5.3f) is a straight line",
      &sgroup[subgroup], vi, vval);
  }
```

6. How to Get the Code

1. Get the L^AT_EX source of this paper [13] from the arXiv.org.
2. Run the source through L^AT_EX. Three new files (**noweb**, **C++** and **MetaPost** sources) will be created in the current directory.
3. Use it on your own risk under the GNU General Public License [10].

References

1. Christian Bauer, Alexander Frink, Richard Kreckel, and Jens Vollinga. GiNaC is Not a CAS. <http://www.ginac.de/>.
2. F. Brackx, Richard Delanghe, and F. Sommen. *Clifford Analysis*, volume 76 of *Research Notes in Mathematics*. Pitman (Advanced Publishing Program), Boston, MA, 1982.
3. Jonathan Brandmeyer. PyGiNaC—a Python interface to the C++ symbolic math library GiNaC. <http://sourceforge.net/projects/pyginac/>.
4. Jan Cnops. *An introduction to Dirac operators on manifolds*, volume 24 of *Progress in Mathematical Physics*. Birkhäuser Boston Inc., Boston, MA, 2002.
5. R. Delanghe, F. Sommen, and V. Souček. *Clifford Algebra and Spinor-Valued Functions*, volume 53 of *Mathematics and its Applications*. Kluwer Academic Publishers Group, Dordrecht, 1992. A function theory for the Dirac operator, Related REDUCE software by F. Brackx and D. Constales, With 1 IBM-PC floppy disk (3.5 inch).
6. B. A. Dubrovin, A. T. Fomenko, and S. P. Novikov. *Modern geometry—methods and applications. Part I*, volume 93 of *Graduate Texts in Mathematics*. Springer-Verlag, New York, second edition, 1992. The geometry of surfaces, transformation groups, and fields, Translated from the Russian by Robert G. Burns.
7. John W. Eaton et al. GNU Octave—high-level language, primarily intended for numerical computations. <http://www.octave.org/>.
8. Bertfried Fauser. Clifford algebraic remark on the Mandelbrot set of two-component number systems. *Adv. Appl. Clifford Algebras*, 6(1):1–26, 1996.
9. Bertfried Fauser and Rafal Ablamowicz. On the decomposition of Clifford algebras of arbitrary bilinear form. In *Ablamowicz, Rafal (ed.) et al., Clifford algebras and their applications in mathematical physics. Proceedings of the 5th conference, Ixtapa-Zihuatanejo, Mexico, June 27-July 4, 1999. Volume 1: Algebra and physics. Boston, MA: Birkhuser. Prog. Phys. 18, 341-366 . 2000. Zbl # 0955.15018*.
10. Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA. *GNU General Public License*, second edition, 1991. <http://www.gnu.org/licenses/gpl.html>.
11. John D. Hobby. MetaPost: A MetaFont like system with postscript output. <http://www.tug.org/metapost.html>.
12. Vladimir V. Kisil. Elliptic, parabolic and hyperbolic analytic function theory–1: Geometry of invariants. 2005. (To appear).
13. Vladimir V. Kisil. An example of clifford algebras calculations with GiNaC. *Adv. in Appl. Clifford Algebras*, 15(1), 2005. E-print: [arXiv:cs.MS/0410044](http://arxiv.org/abs/cs/0410044).
14. Vladimir V. Kisil and Debapriya Biswas. Elliptic, parabolic and hyperbolic analytic function theory–0: Geometry of domains. In *Complex Analysis and Free Boundary Flows*, volume 1 of *Trans. Inst. Math. of the NAS of Ukraine*, pages 100–118, 2004. E-print: [arXiv:math.CV/0410399](http://arxiv.org/abs/math/0410399).
15. Serge Lang. $SL_2(\mathbf{R})$, volume 105 of *Graduate Texts in Mathematics*. Springer-Verlag, New York, 1985. Reprint of the 1975 edition.
16. Norman Ramsey. Noweb — a simple, extensible tool for literate programming. <http://www.eecs.harvard.edu/~nr/noweb/>.
17. David B. Thompson. The literate programming faq. <http://shelob.ce.ttu.edu/daves/lpfaq/faq.html>.
18. S. Weinzierl and R. Marani. gTybalt - a free computer algebra system. <http://www.fis.unipr.it/~stefanw/gtybalt.html>.

Appendix

A. Graphical and Textual Output of the Program

A sample of graphics produced by the program and post-processed with Meta-Postis shown on Figure 1. Some more examples can be found in [12].

Here is the complete textual output of the program.

```

Metric is: e.
Vect field Direct In Cayley In Cayley1
dA is: 2*x,2*y; -2*y*x,1-y^2+x^2; -1-y^2+x^2,2*y*x
dN is: 1,0; 1/2-y+1/2*y^2-1/2*x^2,-y*x+x; -y*x-y,1/2+x-1/2*y^2+1/2*x^2
dK is: 1-y^2+x^2,2*y*x; -2*y,2*x; -2*y,2*x
Curvature of K-orbits on the v-axis:
(1+y^4-2*y^2)^(-1.5)*(-2*y^5-2*y+4*y^3)

Distance to center is: 3.938=====
Distance to center is: 1.875=====
Distance to center is: 0.750=====
Distance to center is: 0.000=====
Distance to center is: 0.750=====
Distance to center is: 1.333=====
Distance to center is: 2.400=====
Distance to center is: 3.938=====
Distance to center is: 7.969=====

Metric is: p.
Vect field Direct In Cayley In Cayley1
dA is: 2*x,2*y; 2*x,1+2*y+x^2; 2*x,1+2*y-x^2
dN is: 1,0; 1,x; 1,-x
dK is: 1+x^2,2*y*x; 1+x^2,2*y*x+2*x; 1+x^2,2*y*x
Curvature of K-orbits on the v-axis: -2*y

Parab (A/ 0/ 0.000); vert=( 0.000, -0.500); l= 0.5000;
second vert=( 0.000, -0.500); l=-0.5000
Check vertices: -0.5 and -0.5
Parab (A/ 1/ 0.053); vert=( 0.167, -0.528); l= 0.5000;
second vert=(-0.167, -0.528); l=-0.5000
Check vertices: -0.5 and -0.5
Parab (A/ 2/ 0.105); vert=( 0.343, -0.618); l= 0.5000;
second vert=(-0.343, -0.618); l=-0.5000
Check vertices: -0.5 and -0.5
Parab (A/ 3/ 0.158); vert=( 0.541, -0.793); l= 0.5000;
second vert=(-0.541, -0.793); l=-0.5000
Check vertices: -0.5 and -0.5
Parab (A/ 4/ 0.211); vert=( 0.778, -1.106); l= 0.5000;
second vert=(-0.778, -1.106); l=-0.5000
Check vertices: -0.5 and -0.5

```



```

Parab (N/ 2/ 0.250); vert=( 0.000, -0.250); l= 0.5000;
                    second vert=( 0.000, -0.250); l=-0.5000
Parab (N/ 3/ 0.500); vert=( 0.000, 0.000); l= 0.5000;
                    second vert=( 0.000, 0.000); l=-0.5000
Parab (N/ 4/ 1.000); vert=( 0.000, 0.500); l= 0.5000;
                    second vert=( 0.000, 0.500); l=-0.5000
Parab (N/ 5/ 2.000); vert=( 0.000, 1.500); l= 0.5000;
                    second vert=( 0.000, 1.500); l=-0.5000
Parab (N/ 6/ 3.000); vert=( 0.000, 2.500); l= 0.5000;
                    second vert=( 0.000, 2.500); l=-0.5000
Parab (N/ 7/ 6.000); vert=( 0.000, 5.500); l= 0.5000;
                    second vert=( 0.000, 5.500); l=-0.5000
Parab (N/ 8/ 10.000); vert=( 0.000, 9.500); l= 0.5000;
                    second vert=( 0.000, 9.500); l=-0.5000
Parab (N/ 9/ 20.000); vert=( 0.000, 19.500); l= 0.5000;
                    second vert=( 0.000, 19.500); l=-0.5000

Directrice is: 1.875=====
Directrice is: 0.750=====
Directrice is: 0.000=====
Directrice is: -0.750=====
Directrice is: -1.875=====
Directrice is: -2.917=====
Directrice is: -5.958=====
Directrice is: -9.975=====
Directrice is: -19.987=====

Metric is: h.
Vect field  Direct    In Cayley    In Cayley1
dA is: 2*x,2*y; -2*y*x,1-y^2-x^2; 2*y,2*x
dN is: 1,0; 1/2-y+1/2*y^2+1/2*x^2,y*x-x;
        1/2-1/16*(16*y-16*x)*x+1/2*y^2-1/2*x^2,
        1/2-1/16*(16*y-16*x)*y+1/2*y^2-1/2*x^2
dK is: 1+y^2+x^2,2*y*x; 1+y^2+x^2,2*y*x; 1+y^2+x^2,2*y*x
Curvature of K-orbits on the v-axis:
        (-2*y^5-2*y-4*y^3)*(1+y^4+2*y^2)^(-1.5)

Difference to foci is: 8.125 -8.125===== 8.125
Difference to foci is: 4.250= -4.250===== 4.250=
Difference to foci is: 2.500=== -2.500===== 2.500===
Difference to foci is: 2.000===== -2.000===== 2.000=====
Difference to foci is: 2.500===== -2.500===== 2.500=====
Difference to foci is: 3.333===== -3.333===== 3.333=====
Difference to foci is: 5.200===== -5.200== 5.200=====
Difference to foci is: 10.100===== -10.100 10.100=====
Difference to foci is: 100.010===== -100.010 100.010=====

```

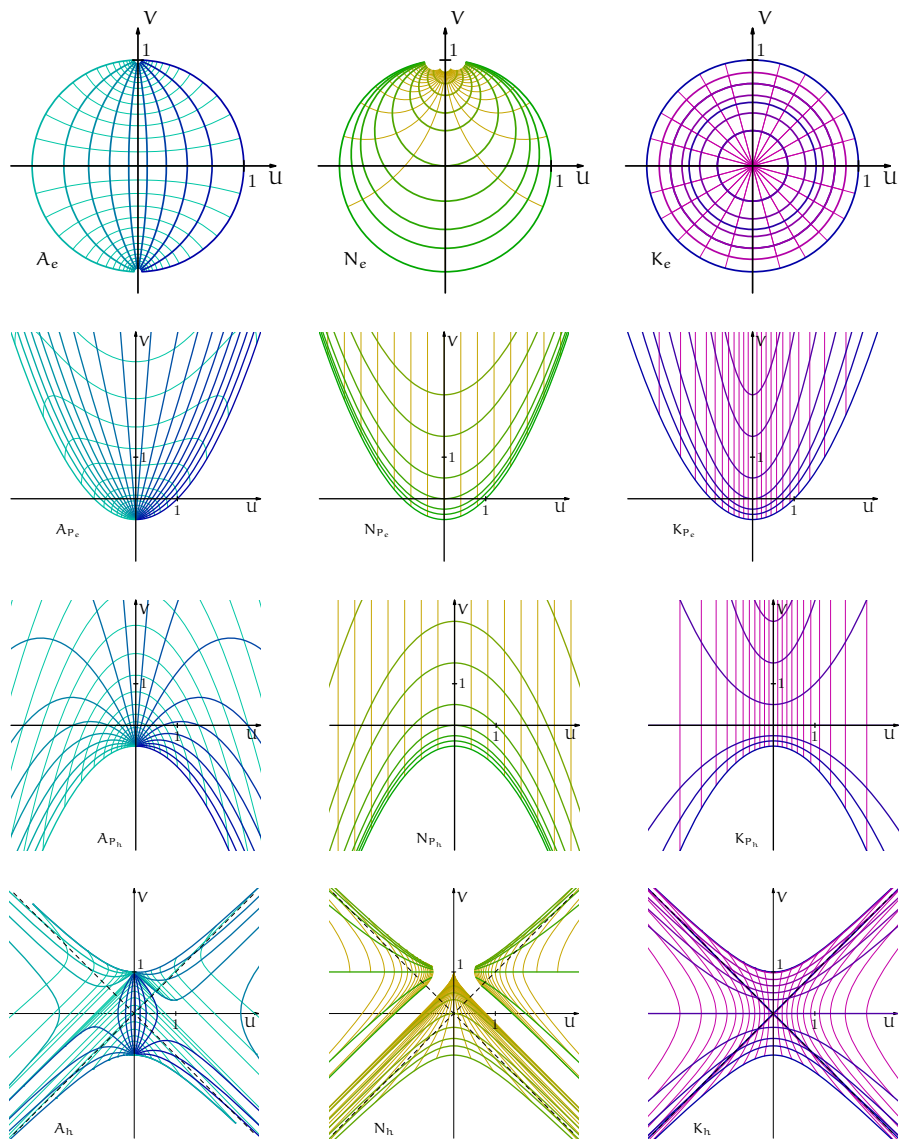


Figure 1. The elliptic, parabolic and hyperbolic unit disks.