
Article:
Kisil, VV (2005) An Example of Clifford Algebras Calculations with GiNaC. Advances in Applied Clifford Algebras, 15 (2). pp. 239-269. ISSN: 0188-7009

https://doi.org/10.1007/s00006-005-0012-1

---

arXiv: cs.MS/0410044, 2004
LEEDS–MATH–PURE–2004-18

Volume 00, Number 0, Pages 000–000

AN EXAMPLE OF CLIFFORD ALGEBRAS CALCULATIONS WITH GiNaC

VLADIMIR V. KISIL

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.

Contents

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 $e_ie_j + e_je_i = B(i,j) + B(j,i)$ 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**);

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. $e_ie_j + e_je_i = 0$)

On leave from Odessa University.

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 $v = (v_0, v_1, \ldots, v_n)$ into the Clifford number $v_0e_0 + v_1e_1 + \cdots + v_ne_n$ 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 $v = (v_0, v_1, \ldots, v_n)$ that $e = v_0c_0 + v_1c_1 + \cdots + v_nc_n$ 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 $(ec_k + c_ke)/pow(c_k, 2)$. 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.

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

3 ⟨* 3⟩≡ 4a ▷

```
⟨Includes 4d⟩
⟨Definitions 5a⟩
⟨Global items 7b⟩
int main (int argc, char**argv)
{
    ⟨C++ variables declaration 8a⟩
    ⟨CiNaC variables declaration 8b⟩
    ⟨Pictures tuning 9a⟩
    ⟨Parabola parameters 19c⟩
```

Defines:
`main`, never used.

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 $SL_2(\mathbb{R})$.

4a ⟨* 3⟩+≡ ◁3 4b▷

```
for (metric = elliptic; metric ≤ hyperbolic; metric++) {
    cout ≪ endl ≪endl ≪ "Metric is: " ≪ metric_name[metric] ≪ "." ≪ endl;
    ⟨Initialise Clifford numbers 9c⟩
    ⟨Calculation of Moebius transformations 10c⟩
    ⟨Calculation of vector fields 11b⟩
```

Uses elliptic 5a, hyperbolic 5a, and metric 7b.

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* from the surrounding cycle in the previous chunk we

(1) build orbits of the subgroups and their transversal curves;
(2) two types of the Cayley transform images of all above curves;
(3) check some formulae in the paper;

We draw all pictures by substitution of numeric values into the symbolic results obtained in the above chunks.

4b ⟨* 3⟩+≡ ◁4a 4c▷

```
    for (subgroup = subgroup_A; subgroup ≤ subgroup_K; subgroup++) {
    /* iteration over subgroups A, N and K */
        ⟨Drawing arrows 12b⟩
        ⟨Building orbits 12c⟩
        ⟨Building transverses 13d⟩
    }
}
```

Uses subgroup 7b, subgroup_A 5a, and subgroup_K 5a.

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].

4c ⟨* 3⟩+≡ ◁4b

```
    ⟨Build future-past transition 14c⟩
}
```

3. Auxiliary matter

Some standard inclusions, but do not forget GiNaC library!

4d ⟨Includes 4d⟩≡ (3)

```
#include <ginac/ginac.h> // At least ver. 1.4.0!
#include <cmath>

using namespace std;
using namespace GiNaC;
```

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

5a ⟨Definitions 5a⟩≡ (3) 5b▷

```
// 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
```

Defines:
elliptic, used in chunks 4a, 10, and 18b.
grey, used in chunk 12b.
hyperbolic, used in chunks 4a, 6a, 10, 14–16, and 18d.
parabolic, used in chunks 10b and 17–19.
subgroup_A, used in chunks 4b, 11, 15c, 16c, and 20.
subgroup_K, used in chunks 4b and 11–19.
subgroup_N, used in chunk 16.

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.

5b ⟨Definitions 5a⟩+≡ (3) ◁5a 5c▷

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

Defines:
init_coord, used in chunks 9b and 15a.
Uses fileout 8a and upos 8a.

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

5c ⟨Definitions 5a⟩+≡ (3) ◁5b 5d▷

```
#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 ")
```

Defines:
close_curve, used in chunks 5d, 14b, and 15a.
put_draw, used in chunks 5d and 9b.
Uses color_name 8a, fileout 8a, subgroup 7b, u 8a, and upos 8a.

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

5d ⟨Definitions 5a⟩+≡ (3) ◁5c 6a▷

```
#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);
#define renew_curve(Y)       close_curve(Y);     \
 put_draw(Y)
```

Defines:
put_point, used in chunk 6a.
renew_curve, used in chunk 6.
Uses close_curve 5c, fileout 8a, put_draw 5c, u 8a, and upos 8a.

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].

6a ⟨Definitions 5a⟩+≡ (3) ◁5d 6b▷

```
#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(lst(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); \
        }
```

Defines:
`if_in_limits`, used in chunks 15–17.
Uses `hyperbolic` 5a, `metric` 7b, `numeric` 8d, `put_point` 5d, `renew_curve` 5d, `subgroup` 7b, `ulim` 9a, and `upos` 8a.

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.

6b ⟨Definitions 5a⟩+≡ (3) ◁6a 6c▷

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

Defines:
`catch_handle`, used in chunks 15–17.
Uses `renew_curve` 5d.

Extracting of numerical values out of Moebius transformations

6c ⟨Definitions 5a⟩+≡ (3) ◁6b 7a▷

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

Defines:
`get_components`, used in chunks 15–17.
Uses `numeric` 8d.

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

7a ⟨Definitions 5a⟩+≡ (3) ◁6c 17a▷

```
#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; \
      } \
}
```

Defines:
transverse_dir, used in chunks 16 and 17.
Uses numeric 8d.

Some global variables which is convenient to use for the *openfile*() function below.

7b ⟨Global items 7b⟩≡ (3) 7c▷

```
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
```

Defines:
metric, used in chunks 4a, 6–10, and 13–19.
sgroup, used in chunks 7c, 11b, 19d, and 20.
subgroup, used in chunks 4–9 and 11–20.
Uses numeric 8d.

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

7c ⟨Global items 7b⟩+≡ (3) ◁7b

```
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");
}
```

Defines:
openfile, used in chunks 12 and 13d.
Uses metric 7b, sgroup 7b, and subgroup 7b.

3.2. **Variables.** First we define variables from the standard C++ classes.

8a ⟨C++ variables declaration 8a⟩≡ (3)

```
FILE *fileout[3]; /* files to pass results to \MetaPost\ */
static char *color_name[]={ "hyp", "par", "ell", "white"}, // A, N, K subgroup colours
*formula[]={"\nDistance to center is:", "\nDirectrice is:",
        "\nDifference to foci is:"};
bool direct = true, // Is it orbit or transverse?
    cayley = false, // Is it the Cayley transform image?
    inversion = false; //Is it future-past inversion?
double u, v, upos[3], vpos[3], udir[3], vdir[3], vval = 0, // coordinates of point, vector, etc
    color_grade, focal_f[2] = {0, 0},
    trans_uf=1, trans_vf=0;
```

Defines:
color_name, used in chunks 5c and 12b.
fileout, used in chunks 5, 12, 13, and 15a.
u, used in chunks 5, 12b, and 19c.
upos, used in chunks 5 and 6a.
v, used in chunks 8, 12, and 19c.
Uses subgroup 7b.

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

8b ⟨CiNaC variables declaration 8b⟩≡ (3) 8c▷

```
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;
```

Uses v 8a.

8c ⟨CiNaC variables declaration 8b⟩+≡ (3) ◁8b 8d▷

```
matrix M(2, 2),          // The metric of the vector space
    C(2, 2), C1(2, 2), CI(2, 2), C1I(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 substituted by a subgroup
```

Uses metric 7b and subgroup 7b.

8d ⟨CiNaC variables declaration 8b⟩+≡ (3) ◁8c 8e▷

```
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);
```

Defines:
numeric, used in chunks 6, 7, 9c, 12b, and 18–20.

8e ⟨CiNaC variables declaration 8b⟩+≡ (3) ◁8d

```
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
```

Uses subgroup 7b and v 8a.

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*.

9a ⟨Pictures tuning 9a⟩≡ (3)

```
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};
```

Defines:
`ulim`, used in chunks 6a and 14c.
`vilimits`, used in chunks 13–16.
Uses `metric` 7b and `subgroup` 7b.

Initialise the set of coordinates for a cycle

9b ⟨Initialisation of coordinates 9b⟩≡ (13a 14a)

```
init_coord(0);
init_coord(1);
put_draw(2);
```

Uses `init_coord` 5b and `put_draw` 5c.

4. Symbolic Clifford Algebra Calculations

This section finally starts to deal with Clifford algebras. We try to make all possible calculations symbolically delaying the numeric substitution to the latest stage. This produces a faster code as well.

4.1. **Initialisation of Clifford Numbers.** We initialise Clifford numbers first. Alternative ways to define *e*, *e0* and *e1* are indicated in comments.

9c ⟨Initialise Clifford numbers 9c⟩≡ (4a 14c) 9d▷

```
signum = numeric(metric-1); // the value of e_2^2
M = -1, 0,
    0, signum;
//e = clifford_unit(mu, indexed(M, symmetric2(),xi, psi));
e = clifford_unit(mu, M);
e0 = e.subs(mu ≡ 0);
e1 = e.subs(mu ≡ 1);
// e0 = lst_to_clifford(lst(1.0, 0), mu, M);
// e1 = lst_to_clifford(lst(0, 1.0), mu, M);
```

Uses `metric` 7b and `numeric` 8d.

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

9d ⟨Initialise Clifford numbers 9c⟩+≡ (4a 14c) ◁9c 10a▷

```
T = dirac_ONE(), e0,  // Transformation to the alternative Cayley map
    e0, dirac_ONE();  // is given by ( 1   e1 )
                                    ( e1  1  )
TI = dirac_ONE(), -e0,  // The inverse of T
    -e0, dirac_ONE();  // is given by ( 1   -e1 )
                                     ( -e1  1  )
```

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

10a ⟨Initialise Clifford numbers 9c⟩+≡ (4a 14c) ◁9d 10b▷

```
switch (metric) {
case elliptic:
case hyperbolic:
    C = dirac_ONE(), -e1,   // First Cayley transform ( 1  −e2 ; σ e2  1 )
        signum*e1, dirac_ONE();
    CI = dirac_ONE(), e1,   // The inverse of C ( 1  e2 ; −σ e2  1 )
        -signum*e1, dirac_ONE();
    C1 = C.mul(T); // Second Cayley transform
    C1I = TI.mul(CI); // The inverse of C1
    break;
```

Uses elliptic 5a, hyperbolic 5a, and metric 7b.

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

10b ⟨Initialise Clifford numbers 9c⟩+≡ (4a 14c) ◁10a

```
case parabolic:
    C = dirac_ONE(), -e1*half,  // First (elliptic) Cayley transform for the parabolic case
        -e1*half, dirac_ONE();
    CI = dirac_ONE(), e1*half,  // The inverse of C
        e1*half, dirac_ONE();
    C1 = dirac_ONE(), -e1*half, // Second (hyperbolic) Cayley transform for the parabolic case
        e1*half, dirac_ONE();
    C1I = dirac_ONE(), e1*half, // The inverse of C1
        -e1*half, dirac_ONE();
    break;
}
```

Uses elliptic 5a, hyperbolic 5a, and parabolic 5a.

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].

10c ⟨Calculation of Moebius transformations 10c⟩≡ (4a) 11a▷

```
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, // Matrix ( 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 $C1$ for operators and finally two Cayley images for points.

11a ⟨Calculation of Moebius transformations 10c⟩+≡ (4a) ◁10c

```
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(
                C1.mul(ex_to<matrix>(Exp[subgroup]).mul(C1I))), 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(
                C1.mul(ex_to<matrix>(Exp[subgroup]))), lst(x, y), M);
    } catch (exception &p) {
        cerr << "*** Got problem in vector fields: " << p.what() << endl;
    }
}
```

Uses catch 12a 16b 17b 17c, subgroup 7b, subgroup_A 5a, and subgroup_K 5a.

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]:

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

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

We calculate *Jacob*ian of the Möbius transformations in order to suply to MetaPost the tangents of the transverse lines.

11b ⟨Calculation of vector fields 11b⟩≡ (4a) 12a▷

```
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)));
            // Transformation of a direction by a Jacobian
            for (int i=0; i < 2; i++)
                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;
}
```

Uses sgroup 7b, subgroup 7b, subgroup_A 5a, and subgroup_K 5a.

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

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

12a ⟨Calculation of vector fields 11b⟩+≡ (4a) ◁11b

```
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 on the v-axis: "
    << normal(Curv[0].subs(x == 0)) << endl;

} catch (exception &p) {
    cerr << "*** Got problem in vector fields: " << p.what() << endl;
}
```

Defines:
catch, used in chunks 11a, 15b, 19a, and 20.
Uses subgroup_K 5a and v 8a.

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 substituion approach is the drawing of vector fields. Three vector fields are drawn by arrows into a MetaPost file.

12b ⟨Drawing arrows 12b⟩≡ (4b)

```
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)); // Get numeric
        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]);
```

Uses color_name 8a, fileout 8a, grey 5a, numeric 8d, openfile 7c, subgroup 7b, u 8a, and v 8a.

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.

12c ⟨Building orbits 12c⟩≡ (4b) 13a▷

```
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
```

Uses fileout 8a and openfile 7c.

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

13a ⟨Building orbits 12c⟩+≡ (4b) ◁12c

```
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 9b⟩
  ⟨Nodes iterations 13b⟩
  ⟨Close all curves 14b⟩
  ⟨Check parabolas 19d⟩
}
⟨Closing all files 13c⟩
```

Uses `metric` 7b, `subgroup` 7b, `subgroup_K` 5a, and `vilimits` 9a.

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.

13b ⟨Nodes iterations 13b⟩≡ (13a)

```
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 15c⟩
    ⟨Check formulas in the paper 18a⟩
    ⟨Producing Cayley transform of the orbit 17b⟩
  }
```

Uses `metric` 7b and `subgroup` 7b.

Closing all files when finishing drawing orbits or transverses

13c ⟨Closing all files 13c⟩≡ (13a 14a)

```
fclose(fileout[0]);
fclose(fileout[1]);
fclose(fileout[2]);
```

Uses `fileout` 8a.

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.

13d ⟨Building transverses 13d⟩≡ (4b) 14a▷

```
direct = false;
fileout[0] = openfile("orbit-t");
fileout[1] = openfile("cayley-t"); // Cayley transform of transverses
fileout[2] = openfile("cayl-a-t"); // Alternative Cayley transform of transverses
color_grade = 1.2;
⟨Define transverse directions 16c⟩;
```

Uses `fileout` 8a and `openfile` 7c.

Thus chunk performs iterations over the transverse lines.

14a ⟨Building transverses 13d⟩+≡ (4b) ◁13d

```
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 9b⟩
    for (int vi = 0; vi < vilimits[subgroup][metric]; vi++) { // iterator over orbits
        vval = vpoints[metric][vi];
        ⟨Generating one entry 15c⟩
        ⟨Producing Cayley transform of the orbit 17b⟩
    }
    ⟨Close all curves 14b⟩
}
⟨Closing all files 13c⟩
```

Uses `metric` 7b, `subgroup` 7b, and `vilimits` 9a.

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

14b ⟨Close all curves 14b⟩≡ (13a 14a)

```
close_curve(0);
close_curve(1);
close_curve(2);
```

Uses `close_curve` 5c.

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.

14c ⟨Build future-past transition 14c⟩≡ (4c) 15a▷

```
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;

metric = hyperbolic;
subgroup = subgroup_K;
direct = true;
inversion = true; // to cut around unit circle.
⟨Initialise Clifford numbers 9c⟩
ex Fut = clifford_moebius_map(dirac_ONE(), -a * e1,
                              a * e1, dirac_ONE(), lst(x, y), M);
```

Defines:
`curves`, used in chunk 15a.
`exp_scale`, used in chunk 15b.
`name`, never used.
`S`, used in chunk 15a.
Uses `hyperbolic` 5a, `metric` 7b, `subgroup` 7b, `subgroup_K` 5a, and `ulim` 9a.

Then we proceed with iteration over the frames.

15a ⟨Build future-past transition 14c⟩+≡ (4c) ◁14c

```
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 15b⟩
        close_curve(0);
    }
    fclose(fileout[0]);
}
cout ≪ endl;
```

Uses close_curve 5c, curves 14c, fileout 8a, init_coord 5b, and S 14c.

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

15b ⟨Iteration over a curve 15b⟩≡ (15a)

```
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(lst( 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);
    }
```

Uses catch 12a 16b 17b 17c, catch_handle 6b, exp_scale 14c, get_components 6c, and if_in_limits 6a.

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.

15c ⟨Generating one entry 15c⟩≡ (13b 14a) 15d▷

```
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 = lst(t ≡ f, x ≡ cos(Pi∗vval), y ≡ sin(Pi∗vval));
    break;
```

Uses hyperbolic 5a, metric 7b, subgroup 7b, subgroup_A 5a, and vilimits 9a.

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

15d ⟨Generating one entry 15c⟩+≡ (13b 14a) ◁15c 16a▷

```
case subgroup_K:
    vval = vpoints[metric][vi];
    a_node = lst(t ≡ (f ∗ Pi), x ≡ 0, y ≡ vval);
    break;
```

Uses metric 7b and subgroup_K 5a.

For the subgroup N we need additional points.

16a ⟨Generating one entry 15c⟩+≡ (13b 14a) ◁15d 16b▷

```
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;
}
```

Uses hyperbolic 5a, metric 7b, subgroup 7b, subgroup_N 5a, and vilimits 9a.

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

16b ⟨Generating one entry 15c⟩+≡ (13b 14a) ◁16a

```
try{
    res = Moebius[subgroup][0].subs(a_node);
    transverse_dir(0);
    get_components;
    if_in_limits(0);
} catch (exception &p) {
    catch_handle(0);
}
```

Defines:
catch, used in chunks 11a, 15b, 19a, and 20.
Uses catch_handle 6b, get_components 6c, if_in_limits 6a, subgroup 7b, and transverse_dir 7a.

We define the tangents to the transverse lines here.

16c ⟨Define transverse directions 16c⟩≡ (13d) 16d▷

```
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;
}
```

Uses subgroup 7b, subgroup_A 5a, subgroup_K 5a, and subgroup_N 5a.

And here again comes evaluation through substitution.

16d ⟨Define transverse directions 16c⟩+≡ (13d) ◁16c

```
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()));
```

Uses subgroup 7b.

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.

17a ⟨Definitions 5a⟩+≡ (3) ◁7a 19b▷

```
#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];                                \
    }
```

Defines:
`calc_par_focal`, used in chunk 17.
Uses `metric` 7b, `parabolic` 5a, `subgroup` 7b, and `subgroup_K` 5a.

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.

17b ⟨Producing Cayley transform of the orbit 17b⟩≡ (13b 14a) 17c▷

```
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);
}
```

Defines:
`catch`, used in chunks 11a, 15b, 19a, and 20.
Uses `calc_par_focal` 17a, `catch_handle` 6b, `get_components` 6c, `if_in_limits` 6a, `subgroup` 7b, and `transverse_dir` 7a.

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

17c ⟨Producing Cayley transform of the orbit 17b⟩+≡ (13b 14a) ◁17b

```
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;
```

Defines:
`catch`, used in chunks 11a, 15b, 19a, and 20.
Uses `calc_par_focal` 17a, `catch_handle` 6b, `get_components` 6c, `if_in_limits` 6a, `subgroup` 7b, and `transverse_dir` 7a.

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*.

18a ⟨Check formulas in the paper 18a⟩≡ (13b) 18b▷

```
if ((j ≠ -fsteps[subgroup][metric]) ∧ (j ≠ fsteps[subgroup][metric]) // End points are weird!
    ∧ (subgroup ≡ subgroup_K) ∧ (vval ≠ 0)) {// only for that values
    try {
        switch (metric) { // depends from the type of metric
```

Uses `metric` 7b, `subgroup` 7b, and `subgroup_K` 5a.

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

18b ⟨Check formulas in the paper 18a⟩+≡ (13b) ◁18a 18c▷

```
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;
```

Uses `elliptic` 5a and `numeric` 8d.

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

18c ⟨Check formulas in the paper 18a⟩+≡ (13b) ◁18b 18d▷

```
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;
```

Uses `numeric` 8d and `parabolic` 5a.

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 $2p$.

18d ⟨Check formulas in the paper 18a⟩+≡ (13b) ◁18c 19a▷

```
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;
}
```

Uses `hyperbolic` 5a and `numeric` 8d.

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

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

Remark 5.1. 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 demonstrates the non-invariance of the upper half plane in the *hyperbolic* case as explained in [12, § 2.5].

19a ⟨Check formulas in the paper 18a⟩+≡ (13b) ◁18d

```
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;
}
}
```

Uses `catch` 12a 16b 17b 17c.

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].

19b ⟨Definitions 5a⟩+≡ (3) ◁17a

```
#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()
```

Defines:
`output_focal`, used in chunk 19d.
Uses `numeric` 8d.

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

19c ⟨Parabola parameters 19c⟩≡ (3)

```
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
```

Uses `u` 8a and `v` 8a.

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 $C1$, see [12, § 2.6].

19d ⟨Check parabolas 19d⟩≡ (13a) 20▷

```
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));
```

Uses `metric` 7b, `output_focal` 19b, `parabolic` 5a, `sgroup` 7b, `subgroup` 7b, and `subgroup_K` 5a.

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].

20 ⟨Check parabolas 19d⟩+≡ (13a) ◁19d

```
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);
}
```

Uses `catch` 12a 16b 17b 17c, `numeric` 8d, `sgroup` 7b, `subgroup` 7b, and `subgroup_A` 5a.

6. How to Get the Code

(1) Get the LaTeX source of this paper [13] from the arXiv.org.
(2) Run the source through LaTeX. 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].

Appendix A. Textual Output of the Program

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:
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,2+2*y+2*x^2;  2*x,2+2*y-2*x^2
  dN is: 1,0; 1,2*x;  1,-2*x
  dK is: 1+x^2,2*y*x; 1+x^2,2*y*x+4*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, -1.000); l= 0.2500; second vert=( 0.000, -1.000); l=-0.2500
Check vertices: -1 and -1
Parab (A/ 1/ 0.053); vert=( 0.083, -1.007); l= 0.2500; second vert=(-0.083, -1.007); l=-0.2500
Check vertices: -1 and -1
Parab (A/ 2/ 0.105); vert=( 0.172, -1.029); l= 0.2500; second vert=(-0.172, -1.029); l=-0.2500
Check vertices: -1 and -1
Parab (A/ 3/ 0.158); vert=( 0.271, -1.073); l= 0.2500; second vert=(-0.271, -1.073); l=-0.2500
Check vertices: -1 and -1
Parab (A/ 4/ 0.211); vert=( 0.389, -1.151); l= 0.2500; second vert=(-0.389, -1.151); l=-0.2500
Check vertices: -1 and -1
Parab (A/ 5/ 0.263); vert=( 0.543, -1.295); l= 0.2500; second vert=(-0.543, -1.295); l=-0.2500
Check vertices: -1 and -1
Parab (A/ 6/ 0.316); vert=( 0.765, -1.586); l= 0.2500; second vert=(-0.765, -1.586); l=-0.2500
Check vertices: -1 and -1
Parab (A/ 7/ 0.368); vert=( 1.140, -2.299); l= 0.2500; second vert=(-1.140, -2.299); l=-0.2500
Check vertices: -1 and -1
Parab (A/ 8/ 0.421); vert=( 1.974, -4.898); l= 0.2500; second vert=(-1.974, -4.898); l=-0.2500
Check vertices: -1 and -1
Parab (A/ 9/ 0.474); vert=( 6.034, -37.410); l= 0.2500; second vert=(-6.034, -37.410); l=-0.2500
Check vertices: -1 and -1
Parab (A/10/ 0.526); vert=(-6.034, -37.410); l= 0.2500; second vert=( 6.034, -37.410); l=-0.2500
Check vertices: -1 and -1
Parab (A/11/ 0.579); vert=(-1.974, -4.898); l= 0.2500; second vert=( 1.974, -4.898); l=-0.2500
```

```
Check vertices: -1 and -1
Parab (A/12/ 0.632); vert=(-1.140, -2.299); l= 0.2500; second vert=( 1.140, -2.299); l=-0.2500
Check vertices: -1 and -1
Parab (A/13/ 0.684); vert=(-0.765, -1.586); l= 0.2500; second vert=( 0.765, -1.586); l=-0.2500
Check vertices: -1 and -1
Parab (A/14/ 0.737); vert=(-0.543, -1.295); l= 0.2500; second vert=( 0.543, -1.295); l=-0.2500
Check vertices: -1 and -1
Parab (A/15/ 0.789); vert=(-0.389, -1.151); l= 0.2500; second vert=( 0.389, -1.151); l=-0.2500
Check vertices: -1 and -1
Parab (A/16/ 0.842); vert=(-0.271, -1.073); l= 0.2500; second vert=( 0.271, -1.073); l=-0.2500
Check vertices: -1 and -1
Parab (A/17/ 0.895); vert=(-0.172, -1.029); l= 0.2500; second vert=( 0.172, -1.029); l=-0.2500
Check vertices: -1 and -1
Parab (A/18/ 0.947); vert=(-0.083, -1.007); l= 0.2500; second vert=( 0.083, -1.007); l=-0.2500
Check vertices: -1 and -1
Parab (A/19/ 1.000); vert=(-0.000, -1.000); l= 0.2500; second vert=( 0.000, -1.000); l=-0.2500
Check vertices: -1 and -1
Parab (N/ 0/ 0.000); vert=( 0.000, -1.000); l= 0.2500; second vert=( 0.000, -1.000); l=-0.2500
Parab (N/ 1/ 0.125); vert=( 0.000, -0.875); l= 0.2500; second vert=( 0.000, -0.875); l=-0.2500
Parab (N/ 2/ 0.250); vert=( 0.000, -0.750); l= 0.2500; second vert=( 0.000, -0.750); l=-0.2500
Parab (N/ 3/ 0.500); vert=( 0.000, -0.500); l= 0.2500; second vert=( 0.000, -0.500); l=-0.2500
Parab (N/ 4/ 1.000); vert=( 0.000,  0.000); l= 0.2500; second vert=( 0.000,  0.000); l=-0.2500
Parab (N/ 5/ 2.000); vert=( 0.000,  1.000); l= 0.2500; second vert=( 0.000,  1.000); l=-0.2500
Parab (N/ 6/ 3.000); vert=( 0.000,  2.000); l= 0.2500; second vert=( 0.000,  2.000); l=-0.2500
Parab (N/ 7/ 6.000); vert=( 0.000,  5.000); l= 0.2500; second vert=( 0.000,  5.000); l=-0.2500
Parab (N/ 8/ 10.000); vert=( 0.000,  9.000); l= 0.2500; second vert=( 0.000,  9.000); l=-0.2500
Parab (N/ 9/ 20.000); vert=( 0.000,  19.000); l= 0.2500; second vert=( 0.000,  19.000); l=-0.2500
Directrice is:
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
  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:
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=============
```

Appendix B. A Sample of Graphics Generated by the Program

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

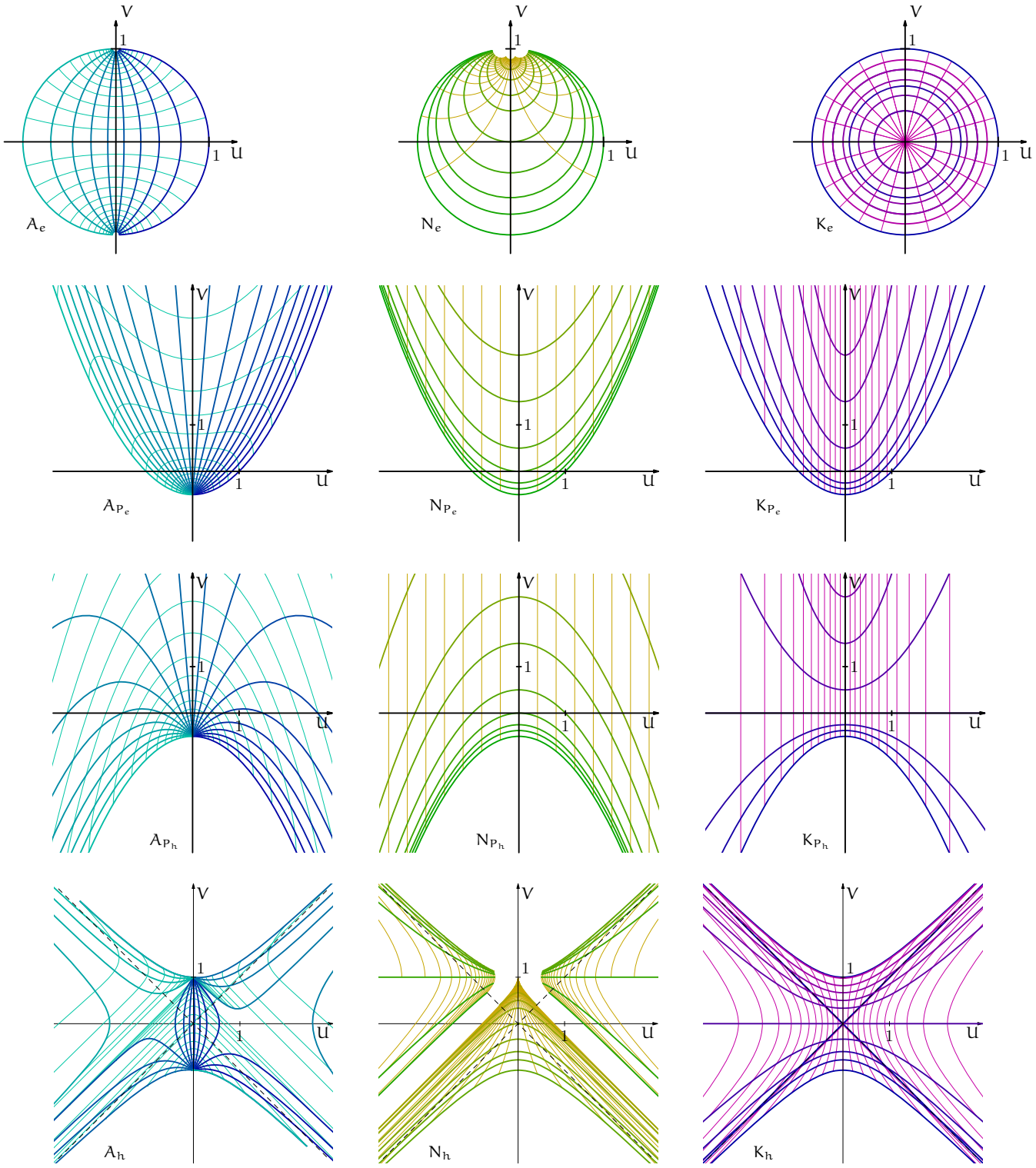

FIGURE 1. The elliptic, parabolic and hyperbolic unit disks.

APPENDIX C. INDEX OF IDENTIFIERS

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 in the sence of F. Klein. 2005. E-print: arXiv:math.CV/yymmnnnn. (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.

[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.

[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.

School of Mathematics, University of Leeds, Leeds LS2 9JT, UK
E-mail address: kisilv@maths.leeds.ac.uk
URL: http://maths.leeds.ac.uk/~kisilv/