COMPUTING WITH QUADRATIC FORMS–THE PROGRAMS FROM THE ZAK PROJECT

RAINER SCHULZE-PILLOT

1. Introduction

In the 1990s Rudolf Scharlau and I had a joint project concerned with computations for and with integral quadratic (and later also hermitian) forms over **Z** and also over the rings of integers of (mainly quadratic) number fields. The project originated in Rudolf Scharlau's group of Diplom and doctoral students, where it also got its name ZAK, I don't really now why (from the German word Zahlkörper (number field) perhaps?). It turned later into a project funded by Deutsche Forschungsgemeinschaft; in this time Alexander Schiemann worked for the project, coordinated the programming work and wrote several C++ programs. The programs developed in the project were the basis of the articles [19, 20, 18, 22, 5] and were used for Schiemann's computation of tables of integral hermitian forms [23]. After Schiemann left academia we continued to use the programs for a while on our HP-UX workstations. When these went out of service it turned out to be difficult to adapt the programs to other environments; I will describe some of the problems later. All experiments I did now are under Linux (Suse 11.0) using gcc 4.3. The present new interest in such computations, in particular in the SAGE project, raises the question whether it is worthwhile (and possible) to revive these programs.

2. The programs

The programs in the ZAK project are centered around the task of classifying positive definite quadratic and hermitian forms. Classifying means here: Enumerate a full set of representatives of isometry classes of forms of fixed dimension, either in a fixed range of discriminants or even in a fixed genus.

2.1. **Class-list.** The case of ternary quadratic forms is somewhat atypical: Here one can (following Seeber) sharpen the usual reduction conditions on a matrix in such a way, that each isometry class contains a unique matrix satisfying the full set of conditions. Consequently, one can just enumerate the matrices of fixed determinant satisfying all the conditions to obtain the desired list. This is done in the programs in our folder Class-list: We have **ternrep**, which takes as input a positive definite ternary form and outputs the unique reduced form in its class, and **ternclass**, which lists all ternaries in a given range of discriminants (with options to list only even, only primitives etc.) Both programs are in C, were written by Alexander Schiemann and are functional (recompiled under Linux using gcc 4.3 and run). Schiemann developed them during the work on his thesis which was later published as [21] and modified them while working with ZAK.
A similar tool is **rep-list**; this generates representatives of genera of forms of specified signature in a given discriminant range. I could compile this program but had runtime errors, probably due to the bigint-routines (see the section about problems).

2.2. **Isometry problem.** In higher dimensions there is no unique reduction. Moreover, from dimension 5 on the reduction conditions become impractical, from dimension 8 on one doesn't even have an explicit finite set of reduction conditions. One can of course still use the LLL algorithm to keep the coefficients of a representative manageable but this does not even come close to finding unique representatives. It is therefore necessary to test quadratic forms for equivalence efficiently. The programs **isom** and **auto(m)** of Bernd Souvignier (who is now, i. e. in 2009, in Nijmegen), described in [13], have been used in our project, with marginal modifications made by Schiemann. The program isom tests for equivalence, the program auto(m) determines the group of units (automorphisms) of a given set of forms. The testing of sets of forms can be used to perform equivalence tests over rings of (quadratic) integers. As far as I know these programs are also still part of MAGMA.

2.3. **Listing forms–Isolist.** During the project a variety of routines was developed for computing various invariants of a set of quadratic forms (also called lattices in the sequel), for example dual lattice, root system, order of the automorphism group, list of short vectors, theta series of degree one and two or Jacobi theta series, dimension of the space of modular forms generated by these theta series, closure of this space under Hecke operators, mass of the set of forms at hand. Schiemann wrote a C++-program **isolist** which takes as input a list of quadratic forms, extracts a set of representatives of the different classes and, depending on options handed to the program, computes various invariants of the forms and organizes the representatives into a list.
More precisely and quoting from Schiemann's documentation, isolist offers the following possibilities:

> "isolist" is a program to sort Z-lattices by isometry and/or compute data (theta-series, order of automorphism group) belonging to these lattices. It can also compare the span of (different kinds of) theta-series of a set of lattices with the span of theta-series of a subset.
>
> isolist performs in different stages (some of which are optional):
>
> **Stage 1:** Parsing of standard input for **Z**-lattices until end-of-file or &End is read. Result is a list of lattices with some related data that was also read from standard input.
>
> **Stage 2:** Perform reduction on the lattices, check the list for isometric lattices and delete all but one representative of isometric ones. Depending on the options some invariants of each lattice may be computed to simplify tests for isometry. Result is a list of representatives of the classes of lattices that where read.
>
> **Stage 3:** Compute the theta-series of the representatives (if not known already) and sort the list depending on the theta-series. Compute additional information about the spaces of different kinds of theta-series belonging to the lattices and print this information on standard output.
>
> **Stage 4:** Compute additional invariants of each representative and print the representative together with the invariants. The order in which the lattices are processed and printed depends on their theta-series or on their number.
>
> The output is in a format that can be processed by isolist again. So you may accumulate information in different runs of isolist (without loss of performance, since the tags &begin_block/&end_block are used to mark the list sorted by isometry).

To give an impression of the way in which Schiemann organized his programs, here is a file list of the folder Isolist:

```
assign.h
asub.c
asub.h
asub_isolist_templ_I.cc
asub_templ.cc
asub_templ.h
bigbin_modular_form.cc
bigbin_modular_form.h
bigfloat_adapt.h
bigfract_call.cc
bigfract.cc
bigfract.h
bigint_adapt.h
bin_IO.h
bin_modular_form.cc
bin_modular_form.h
bin_theta_series.cc
bin_theta_series.h
blocked_vectorC_stack_templ.cc
blocked_vectorC_stack_templ.h
blocked_vectorV_stack_templ.cc
blocked_vectorV_stack_templ.h
block_link_templ.cc
block_link_templ.h
classlist_util.cc
classlist_util.h
COPYING-2.0
COPYRIGHT_isolist
cp_rm_diff
datatypes_templ_I.cc
datavec.cc
datavec_templ.cc
datavec_templ.h
fract.cc
fract.doc
fract.h
Global_DimensionC.cc
Global_DimensionC.doc
Global_DimensionC.h
Glob_int2.cc
handle_uninit_templ.cc
handle_uninit_templ.h
include_param_isolist.h
integer_util_templ.cc
integer_util_templ.h
int_PF.cc
int_PF.h
iso_aut_source.c
iso_aut_source.h
isolist.cc
isolist.doc
isolist.h
isom-list.cc
isom-list.doc
isubspace_templ.cc
isubspace_templ.h
isubspace_templ_I.cc
jac_theta_series.cc
jac_theta_series.h
lidia_param.h
list_errors.cc
list_errors.h
list_with_positions_templ.cc
list_with_positions_templ.h
lower_triangleC_templ.cc
lower_triangleC_templ.h
Makefile
maklib.h
malloc_redefine
malloc_unredefine
math_types.h
matrixC.cc
matrixC_templ.cc
matrixC_templ.h
matrixC_templ_I.cc
matrix_row_ref_templ.h
matrix*_templ.doc
matrix_templ_I.cc
matrix_util_templ.cc
matrix_util_templ.h
matrixV.cc
matrixV_templ.cc
matrixV_templ.h
new_nothrow.cc
new_nothrow.h
new_version_file
option_util.cc
option_util.h
option_util_isolist.cc
option_util_isolist.h
orbits_iter_p.cc
orbits_iter_p_isolist_templ_I.cc
orbits_iter_p_templ.cc
orbits_iter_p_templ.doc
orbits_iter_p_templ.h
ppointer_templ.cc
ppointer_templ.h
README.1st
README.AUTO
README.ISOM
redu_util_isolist_templ_I.cc
redu_util_templ.cc
redu_util_templ.h
s_malloc.c
s_malloc.h
s_malloc.usage
strongly_mod.c
strongly_mod.h
sym_matrixC_templ.cc
sym_matrixC_templ.h
sym_matrix_IO.cc
sym_matrix_IO.h
Tests
theta_series.cc
theta_series.h
types.h
unmaklib.h
vectorC.cc
vectorCp_templ.h
vectorC_templ.cc
vectorC_templ.h
vector_templ_I.cc
vectorV.cc
vectorV_templ.cc
vectorV_templ.h
VERSION
version_isolist.h
Zclassdata2.cc
Zclassdata2.h
Z_isoautC_templ.cc
Z_isoautC_templ.h
Z_isoautC_templ_I.cc
Z_isoaut_options.h
Z_latticeC_class_templ.cc
Z_latticeC_class_templ.h
Z_latticeC_class_templ_I.cc
Z_latticeC_shortvecs_templ.cc
Z_latticeC_shortvecs_templ.h
Z_shortvecs_orbit_util_templ.cc
Z_shortvecs_orbit_util_templ.h
Z_shortvecs_orbit_util_templ_I.cc
```

Attempts to recompile Isolist failed, see section Problems. A different program for organizing lists of lattices is **invar**, it is essentially a part of Hemkemeier's routine **tn** and will be discussed together with it in the next subsection.

2.4. **Neighbouring lattices.** The method of neighbouring lattices has been introduced by Kneser in [10]. Its use for computer calculations was described in [24, 18, 22]. Over **Z** it can be used to determine all classes in a given genus, over the integers of number fields one has to be careful about obtaining all spinor genera in the genus. Our project produced the routines **tn** (written by Boris Hemkemeier) calculating neighbours at the prime 2 of a **Z**-lattice and thereby obtaining all classes in the genus, **pn, pn-sh** (written mainly by Markus Stausberg) for doing the same using neighbours at some prime p, **qn** (written mainly by Fabian Wichelhaus) for

doing the same over a real quadratic field, and **hn** (written by Alexander Schiemann) for doing the same for hermitian forms over the integers of an imaginary quadratic field.
Of these, **hn** could not be recompiled (see section Problems). Some years ago Abshoff created an executable which was used on Linux PCs in Dortmund and Saarbrücken and still runs on older Linux systems, it crashes with linking errors on my present PC.
tn has been maintained by its author Boris Hemkemeier and is available in the web [3]. The programs **pn, pn-sh,qn** could be recompiled, running has not yet been tested, except for one or two runs with **pn-sh**. All three programs have a number of utilities for generating additional information, e. g. the Anzahlmatrix showing the number of neighbours of class A in class B, the shortest vectors etc. The program **tn** has more utilities than **pn,qn** for generating such additional information, e.g. root systems of lattices. These facilities of **tn** have been collected by Schiemann in the program **invar** which takes as input a list of lattices generated by one of the other programs. **invar** could (with marginal changes) be recompiled but has not yet been tested for running.
I include the doc-file of **hn** to give an impression of what it does when properly activated:

```
/*
   Copyright (C) 1997  Alexander Schiemann
   (try 'hn --copyright' for more information)
 */

this file describes VERSION 2.8.6 of hn (15.3.99)

SHORT INFO:
==========

This program (hn) deals with positive definite integral hermitian
forms  over an imaginary quadratic field.
It collects all functionality that I have implemented for
hermitian forms. This is:
- filter for different IO-formats (see: --invar, -I, -O )
- reduction of hermitian forms with respect to Gl(n,O_K) (see: --invar,
  --reduce, --herm_lll )
- calculating invariants for hermitian forms, e.g. automorphism group,
  theta-series: (see: --invar, --autom, --groups, --shell -O )
- sorting of lists of hermitian forms for isometry
  (see: --invar, --isom_test )
- generating the neighbourhood (representatives of all classes)
  of a given form
  (``unmodified normal mode'' of the program).
- heuristic search for forms with large minimum in a neighbourhood
  (modifications of the ``normal mode'').
(see also below under ``OPTIONS AFFECTING THE TYPE OF ACTION ...'')

In its normal mode this program  computes one or more special genera
of integral hermitian forms over an imaginary quadratic field K
using the neighbour method of Kneser. This method for the
hermitian case is described in detail in the
paper "Classification of hermitian forms with the neighbour method"
```

```
that can be obtained at http://www.math.uni-sb.de/~aschiem.
In its normal mode (with isometry tests and construction of orbits
of neighbour vectors under the action of the automorphism group) the
program  works in dimensions (over K) 2,..,10 (,11,12) with neighbours
at small primes P. The runtime is O(N*M*I), where N is the number of
neighbours at P of a lattice, M the number of classes in the union of
special genera that will be computed and I the time consumed by one
isometry test or construction of an automorphism group. Isometry tests
and construction of automorphism groups are subroutines based on the
programs 'isom' and 'bravais' (alias 'autom') by BERND SOUVIGNIER
1995, whom I want to thank for the permission to use the code of his
programs.

SOME REMARKS:
============
There are many options and not all combinations are meaningful. Some
options cancel other options without warning. At the beginning you
should rely on the defaults and only specify options for the 'overall
choices'. Use --defaults to find out which options really are in effect.

If you use 'hn --invar' you should read the help given with 'hn
--invar --help' to understand how to pass a list of lattices (possibly
with known invariants) to 'hn'. Usually you will write the
specification of the imaginary quadratic field, the dimension of the
hermitian space and one or more lattices into a file and  say 'hn
--invar < file'.
Critical applications may need fine-tuning.
-- pass known invariants of lattices to 'hn --invar' as described in
   'hn --invar --help'. Especially known automorphisms may speed up the
   calculation of the automorphism group and isometry tests (use the
   'Aut_subgroup' invariant)
-- try to collect information in successive runs, starting with
   reduction only.    optimize the parameters of each subalgorithm.


DETAILED DESCRIPTION OF OPTIONS:
===============================
Feel free to try combinations of options. Inconsistent combinations
should result in warnings or errors (but may sometimes also lead to
strange results).
We made quite an effort to ensure that the output is not misleading
whatever strange input was made.
Use --defaults to learn which of the options actually remain in effect.

INVOCATION ( [..] stands for optional):
--------------------------------------

hn [options]

INPUT depends on options. In the normal mode, there will be prompts for input
on standard output. The input is
D  dim  lattice  prime
```

```
  where:
D       : specifies an imaginary quadratic field K=Q(sqrt(D))
dim     : dimension (over K) of the hermitian space
lattice : [ list of ideals ] [ {factor}* ]   gram_matrix
          for an exact description of the input format start the
          program with the --verbose (-v ) option
prime   : prime_number or ~prime_number (for the conjugate prime ideal)

OPTIONS : [[-]single_character_options] [[-]key_character[inner_options] ]
          [--word_options]
single-character options: [hH?svltn A'n' T'n' S'n' G'n' D'n']
key-character options   : [O[csbwpPLTOU]  I[bwPLB] ]
word options            : [--invar --recover --recover'filename'
                           --neighbours'n' --neighbour_orbits'n'
                           --only_even --only_odd
                           --max_stable'n'--isom_test --no_isom_test --help
                           --Help --silent --verbose --neighbour_verbose
                           --neighbour_silent --graph_max'i'
                           --trace_isom_search --no_trace_list
                           --blank_format --w_format --pari_format
                           --shells['i'] --Depth'i' --mass --no_mass ...]
Options can be given as single character options (with an optional leading '-'
and optional space between) or as a keycharacter followed by a keyword or an
inner list of character options (ending with a blank) or as a word-option
(that stands for a combination of other options in some cases).
Single character options and word-options may have arguments.
Most options have inverse options. In case of competing options, the last
given overrides the others (important for --recover (see below)).


OPTIONS AFFECTING THE TYPE OF ACTION THAT 'hn' WILL TAKE:
--------------------------------------------------------
(if none of these options is given, the expansion of
 the (whole) neighbour-graph takes place, this is the
 normal action of 'hn'. Modifications of the normal mode are described
 below.)

--version           : show the version number and date

--copyright         : inform about copyright

[-]h                : give this help
[-]H
[-]?
--help
--Help

--help --invar      : give help about the input parsing with --invar

--help --isom       : prints original description of the isometry test
          for sets of bilinear forms over Z (by Bernd Souvignier).
          This test is used within the isometry test of the lattices
          over the ring of integers of a quadratic field here.
          You may set the option for this algorithm using
          `set_isoaut_options(...)' (see below).
--help --autom      : prints original description of the computation
          of the automorphism group of a set of bilinear forms over Z
          (by Bernd Souvignier). This test is used within such
          computations for lattices over the ring of integers of a
          quadratic field here.
          You may set the option for this algorithm using
          `set_isoaut_options(...)' (see below).

--defaults          : show the default settings, i.e. show the settings
          that would take place with the given options except 'defaults'
          (the defaults may depend on given options!)

--recover'filename' : (no space between recover and 'filename')
          (not interactive) if possible a run of hn generates a logfile
          to save partial results in case of a crash. A new logfile gets
          the first non-existing name from hn.log,hn.log1,hn.log2..
          If a run ends successfully the log file is destroyed.
          With the recover option a crashed run that produced the logfile
          'filename' is resumed and new results are appended to that file.
          A recover-job will use the same options as the original
          job with the new options appended to the old ones and
          possibly overriding them. All meaningful changes of parameters
          are allowed with --recover and all other combinations of
          options (e.g. --only_odd recovered with --only_even) will
          (hopefully) lead to an error message and abort the program.
          The representatives of the classes and their order in the
          output of a recovered job may be different to that of a
          completed job with the same input. This is because a
          recovered job may recalculate the last (incomplete)
          expansion with another representative, if a better
          representative was found during the last expansion and
          no '&Neighbour #n'-directive with n>1 has been saved in
          the logfile. Also the counter of the max_stable option is
          reset when a job is restarted.
          The logfile is removed after a complete run unless
          the expansion was terminated because of --stop_mass.

--recover           : (not interactive) same as above but the filename
          will be the first of hn.log,hn.log1,...,hn.log19 with an
          existing file.

--invar             : (not interactive) do not construct lattices but read
          lattices (and related data) from standard input, perform reduction/
          tests for isometry and/or compute invariants of these lattices
          (depending on other options).
          The input is parsed  according to the rules described with
          'hn --help --invar', everything but lines following special 'tags'
          is ignored . Output and log-files are in a suitable format to
          be used as input to 'hn --invar'.
          If a logfile is passed to hn --invar, the invariants
          of the lattices found up to the crash are computed.

--neighbours'n'     : (interactive) Compute at most 'n' neighbours of
          the start lattice without orbit construction and without tests
          for isometric lattices.

--neighbour_orbits'n' : (interactive) As above but compute the automorphism
          group of the start lattice and compute only neighbours belonging to
          different orbits under the action of the automorphism group.

MODIFICATIONS OF THE NORMAL MODE (all interactive):
---------------------------------------------------
--min_search['L' ['M' 'S']] : heuristic option to search for lattices with large
          minimum in large genera (which can not be classified completely).
          Implies --maxmin -G0 and --no_intermediate_mass. 'L' and 'M','S'
          are positive integers.
          A run with --min_search writes not only a logfile (see --recover) but
          also a file named "largest_minimum.D-'disc'.dim'DIM'.det'DET'.tmp"
          (with the values 'disc' the discriminant of K, 'DIM' the K-dimension
          and 'DET' the discriminant of the hermitian lattices. Whenever a
          lattice with a larger minimum is found, the lattice is appended to
          this file (thus a file of such a name may be altered by hn). This
          file is the usual output. If a run ends, it gives the classes that
          were expanded on standard output. In general, this will not be the
          whole neighbourhood.
          The expansion strategy is plainly as discribed below with --maxmin,
          but the list for the unexpanded classes has limited length 'L'.
          If it grows longer, the last class in this list (i.e. that with
          the most short vectors up to the current bound 'maxmin') will be
          dropped.
          Let at a moment 'l' be the length of the list for the unexpanded
          classes. Given a new neighbour with minimum 'm', this neighbour will
          be dropped (without any notice) if (M,S) is given and m<M and there
          are >=S unexpanded lattices better than the new one.
  Otherwise
          if l<L or (l==L and the new lattice would NOT be last of (unexpanded
                     without the one being expanded) in the order described
                     with --maxmin)
             the new lattice is checked against all classes for isometries and
             is added to the list of all and of the unexpanded classes if it
             belongs to a new class.
             if (l_new=l+1) == L, then the last item (without the one being
             expanded) is dropped from unexpanded.
          otherwise (i.e. l==L and the new lattice has the 'most short vectors')
             the lattice is dropped (without any notice).
          NOTE: the list of expanded classes is not limited and
          isometry tests take place. The logfile may contain lattices that are
          dropped later on. It may be recovered (see above). You may change the
          --min_search parameters. This will NOT prevent lattices marked as
          &dropped in the logfile from being dropped, even if the new queue
          of unexpanded lattices is longer.
```

```
    If the isometry tests fail or need to much time, you may combine this
    option with the following ones, which imply --min_search:
--no_isom['k'] :  never test for isometry.
    Every new lattice is dropped under the conditions
    described above. If not (i.e. if the algorithm stated above would
    start isometry tests), the new lattice is dropped
    IF the number of expanded classes with the same theta series
    (up to a bound 'B')+ the number of such lattices that had ever been
    added to the queue 'unexpanded' is >= k   (this is to avoid infinite
    loops). Otherwise it is appended to unexpanded.
    The bound 'B' used above is the current value of theta_bound (see
    also --maxmin). Whenever 'B' changes, all theta-series-counters are
    re-evaluated. You should use -T to give a reasonable start value
    for B.
    It does not seem very useful to restart such a run, although
    this should be possible (in that case all the theta-series-counters
    are set to the number of matching thetaseries in unexpanded).
--jump['k']    : 'k' is an optional integer. Without this option, to
    every class being expanded all neighbours will be computed. If the
    number of neighbours is very large, that may be unefficient. If a
    better lattice shows up, the current expansion is aborted after 'k'
    more neighbours have been constructed unless another even better
    lattice is found. Whenever this occurs, the counter for 'k' is reset.
    If --no_isom is given, the theta-series-counter of the theta series
    belonging to the currently expanded lattice is increased when the
    expansion is aborted because of --jump exactly as in the case of a
    completed expansion. On the contrary, if --no_isom is not active,
    the currently expanded lattice will not be marked 'expanded'; it
    may be expanded again later, if all better lattices have been
    expanded.
    To switch off --jump, give a negative value for 'k'.

    NOTE: successive options --min_search --no_isom --jump will override
    the previous ones, but only with those values that are explicitly
    given. The first option --min_search will provide default values if
    all or some of the parameters are omitted. Repeated calls will only
    override parameters with new values, if those are explicitly given.
    e.g.  --min_search201 5 17  --min_search300  is --min_search300 5 17

--exclude_by_shortvecs'b' 'm' 'f': implies --min_search: 'b' and 'm'
    are non-negative integers and 'f' a floating value between 0 and 1.
    Let L denote the currently expanded lattice (the father) and S be
    the set of vectors of L with length <=b.
    If #S<='m' then a neighbour L(x) of L is ignored if S intersected
    with L_x:=(L intersected with L(x)) has more than #S*'f' elements:
    In short: We may exclude neighbours only by computing how many of
    the vectors in S are in L_x (which is only the computation of one
    scalar-product for each of the vectors).
    If 'b' is 0 then the option is reset.

--pre_exclude_bound'b' 'm': implies --min_search: 'b' and 'm'
```

```
          are non-negative integers. After a neighbour passed the 'exclude_
          by_shortvecs' test (described above) the theta-series is computed
          up to lenght <='b'. If there are more than ‘m’ vectors up to length
          'b' then the lattice is dropped.
          If 'b' is 0 then the option is reset.

OTHER GENERAL OPTIONS:
---------------------
[-]s
--silent            : do not print input requests and explanations
[-]v
--verbose           : print input reqests and some explanations on cerr

--reduce            : reduce the lattices that are read from a logfile or
          from standard input (the start lattice/all lattices with --invar)
--no_reduce         : do not reduce the lattices given.

--no_autom          : never compute automorphism groups. This option implies
          -A INT_MAX --no_mass  and disables the orbit construction on the set
          of neighbour-vectors.
--autom             : allow construction of automorphism groups

--isom_test         : only with --invar: test the given classes for isometric
          ones and give a list of the 'best' representatives
--no_isom_test      : only with --invar: do not test the given classes (from
          standard input) for isometric ones

--check_disc        : only with --recover: check the discriminant of lattices
          read from the logfile
--not_check_disc    : only with --recover: do not check the discriminants

--groups            : whenever an automorphism group is computed it is stored.
          This data is useful to avoid recomputation of the automorphism group
          as otherwise necessary for orbit computations. It may also be used
          to speed up tests for isometry (see --iso_use_known_auts). This option
          will increase the storage consumption if there are many classes.
          NOTE: The order of the group of automorphisms will be stored whenever
          it becomes known, regardless of --groups.
          NOTE: the information about the group is lost, if the representative
          is changed because of reduction or by &replacing...
          This option also enables output of group generators to a logfile,
          for a description of this output see -OG below.
          Reading of automorphisms from a logfile or from standard input is
          controlled by -IG (see below). Printing to standard output is
          controlled by -OG (see below).
--no_groups         : no group generators are stored nor printed in the logfile

OPTIONS AFFECTING THE INTERPRETATION OF INPUT:
---------------------------------------------
[-]I[bwPLB]         : affects the input of integers from O_K and the format
          of the hermitian matrices read from stdin (not from a logfile
          when read with --recover and not with --invar):
```

```
          input options specified by (one or more) characters appended to 'I':
          b : input numbers in O_K in blank_format (e.g. 1 0, 0 1, 3 -2)
          w : input of numbers in O_K and K in w_format (compatible to PARI)
              (e.g. 1, w, 3-2w, 2-5w, 2-3*w or  1/3-2w, 1/2+3/5*w )
          P : Gram-matrix input is in PARI mode (e.g. [*,* ; *,*] )
          L : Gram-matrix input is the lower triangle of the hermitian matrix)
          B :    "    "      "  is given as a full matrix or a lower
              triangle. The end of input for a row is given by newline
          G : store elements of the automorphism group that are read from a
              logfile or with --invar from standard input. For the format see
              'hn --invar --help' (either the generators of the full group of
              automorphisms or just some automorphisms may be specified).
              WARNING: of course, the generators are no invariants of the
              class. Nevertheless, they may be given after &invariants in the
              logfile. Then they belong to the most recently given Gram matrix
              of this class. The automorphisms are given as matrices with
              respect to the same base as the Gram matrix. The automorphism
              matrices Gi are subject to Gi*H*(~Gi).transpose() == H   (where
              H is the Gram matrix).
              NOTE: the information about the group is lost, if the
              representative is changed because of reduction or by &replacing...
          N : ignore automorphisms given in a logfile or in standard input

--blank_format      : I/O of numbers in O_K as two components separated by
          Gram matrices as lower triangle

--w_format          : I/O of numbers in O_K and K using 'w' (e.g. 1, w, 3-2w,
          Gram matrices as lower triangle

--pari_format       : I/O of numbers in O_K and Gram matrices in PARI-format,
          (e.g. [2,3-2*w ; 3+2*w, 4] (given conjugate(w)=-w))

--multiply_by'I'    : multiply any given lattice (i.e. multiply every
          ideal of a given lattice) with the ideal 'I', where 'I' is an
          ideal given in standard format (Note: 'I' will not be
          processed until the quadratic field K is initialized.
          at startup everything enclosed by <...> (including
          < and >) is assigned to a string that later defines the
          ideal). Note: in most shells it will be necessary to
          enclose the arguments by ""..."", because '<' and '>'
          stand for the redirection of input. To learn about
          input of an ideal, start the program with --verbose.
          Multiplying will affect every form read from standard
          input (e.g. with --invar) but has no effect when
          logfiles are read

--scale_by'f'       : scale any given hermitian form (i.e. multiply the
          Gram matrix) by the factor 'f', where 'f' is a fract written
          as numerator/denominator without blanks. This will change
          the hermitian space where the form lives in iff f^dim is
          not in Norm(K). Scaling will affect every form read from
          standard input (e.g. with --invar) but has no effect when
          logfiles are read. The result of multiplying and scaling
          (if both --multiply_by and --scale_by are given) must be
          integral for every lattice that is entered.

OPTIONS CONTROLLING THE NEIGHBOUR CONSTRUCTION:
----------------------------------------------
--neighbour_verbose : print each neighbour and the vector used to construct
          it to the logfile or to standard output in case of
          --neighbours or --neighbour_orbits
--neighbour_silent  : do not print the neighbours

--only_even         : only if neighbours are computed at a prime above 2
          and the start lattice is even: Only even neighbours are
          generated. This will result in computing all even
          lattices in the neighbourhood, because the even lattices
          are a connected subset in the neighbour graph. This option may
          not be combined with --recover (only_odd or only_even of the
          original run will be remembered automatically)

--only_odd          : only if neighbours are computed at a prime above 2
          and the start lattice is odd: Only odd neighbours are
          generated. WARNING: The odd lattices are not necessarily
          a connected subset in the neighbour graph. Possibly not
          all odd lattices in the neighbourhood are generated
          This option may not be combined with --recover (only_odd or
          only_even of the original run will be remembered automatically)

--max_stable'n'     : after 'n' successive expansions of classes in one
          depth that did not produce a new class, the next class
          to expand is choosen from another depth (if possible)
          The 1st depth bigger than the current with an unexpanded
          class is choosen or (if there is no such class) the
          smallest depth with an unexpanded class. Here the 'depth'
          of a lattice means the minimal number of neighbour steps
          from the start lattice to this lattice along a path that
          'the program knows about' (not exactly the depth in the
          neighbour-graph)

--stop_mass'bf'     : expansion of the neighbour graph is stopped if the
          mass becomes >='bf', where 'bf' is a bigfract (written in the
          form numerator/denominator without blanks). This option
          implies --graph_max0 and --intermediate_mass. After the run
          the logfile is not deleted. If the mass 'bf' was reached and
          the expansion left incomplete, this is notified in the output.
          if 'bf'==0, a former stop_mass is erased (can be used to
          reset this option in a recovered run).
          if --invar, --neighbours or --neighbour_orbits is given,
          --stop_mass has no effect.

[-]A'n'
--autom_bound'n'    : the order of the automorphism group is used as an
          invariant to avoid isometry tests if more than 'n'
          classes are already found.

[-]T'n'
--theta_bound'n'    : the theta series of each lattice is computed up to
          norm 'n' to avoid isometry tests. This is just a start value.
          When 1000 classes have been found, 'n' will automatically be set
          (increased) such that it is maximal with <=5% of the thetaseries
          being shorter than 'n' (at that time). In a resumed run this
          setting will take place at the beginning if there are more than
          1000 classes in the logfile. If --maxmin is given, theta_bound
          may be increased whenever a lattice with larger minimum is found.

--compute_missing_groups: only with --groups and --iso_use_known_auts,
          redundant if -A0.
          the automorphism group of every new class is computed and stored
          so that it can be used in future isometry tests. NOTE: Differently
          to -A0 the automorphism groups of new neighbours are not
          (necessarily) computed, only after isometry testing has been
          performed and the neighbour found to be in a new class.
          This option guarantees that in each isometry test at
          least the automorphism group of one lattice is known.
--not_compute_missing_groups: switch off --compute_missing_groups

--intermediate_mass : whenever a class is found, the sum of the inverse
          orders of the automorphism groups is written into the logfile.
          (implies --mass).
--no_intermediate_mass: the mass is computed only at the end of the run (or
          never), to avoid intermediate automorphism calculations

--mass              : the mass of the family of lattices (i.e. is the sum of
          the inverse orders of the automorphism groups) is given in output
          at the end of the run.
--no_mass           : the mass is not computed at all
          (implies --no_intermediate_mass).

[-]G'i'
--graph_max'i'      : 'i' is an integer. The adjacence-matrix will be
          computed up to a class number <=i (to avoid quadratic storage
          consumption)

--maxmin['n'['b']]  : 'n','b' optional integers ('n' will be set to the
          maximal minimum+1 of all forms known at the beginning or resuming
          of the neighbour-construction if it is omited).
          Changes the expansion strategy:
          Briefly: Searches for forms with large minima, starts with a
          strategy to find minimum 'n'. Whenever such a form is found, 'n'
          is increased by 1 until the bound 'b' is reached. Then more forms
          with minimum 'b' are searched.
          Case n>0: The next class to be expanded is the best (i.e. smallest)
          with respect to the following total order:
           0. a form with 2*minimum >=n is better (i.e. < ) than
              one with 2*minimum <n.
           1. smallest #{x| h(x,x)<n}   among these with
           2. largest minimum   and among these
           3. minimal number (as detected in the expansion)
           Whenever the new largest minimum l becomes>n,
           n is set to l or to min(b,l) if b was given
           and the unexpanded classes are reordered.
           Whenever 'n' is increased, theta_bound is set to
           max(n-1,theta_bound)
          Case n<=0: Reset this option and resume the usual expansion strategy
           depending on the number and switching depth according to
           --max_stable. Used to reset the option in recovered runs.

OPTIONS CONTROLLING (PARAMETERS OF) SUBALGORITHMS:
-------------------------------------------------

--set_isoaut_options('string'): (FOR EXPERTS, rely on defaults or try
          more than once!) 'string' must not contain ')'.
          and the closing brace must be seperated from the next option by
          whitespace. This option can be used to set all parameters of the
          subalgorithms developed from ISOM and AUTOM of Bernd Souvignier
          which compute the automorphism group/perform isometry test for
          sets of bilinear forms over Z.
          Some options also may be set directly (see below).
          'string' may contain options as described in 'README.ISOM' and
          'README.AUTOM' which you may print using --help --isom or
          --help --autom (see above). Different options in this string
          shall be separated by whitespace. If a suboption of this whole
          option is ommited then reasonable default parameters are used.
          This option is accumulative even interweaved with -D or -B.
          Note: The options will affect all computations of automorphism
          groups as well as isometry tests. Some options are not meaningful
          and will be ignored or overridden depending on the situation.
          more precisely: the -S'n' option with 'n'!=0 leads to the
            computation of a stabilizer subgroup of all automorphisms in
            `autom2'. But such a setting is overridden in all methods of
            herm_lattice_class and hneighbour_orbits. Thus nothing but
            the computation of the automorphism group for output with -OG
            is affected by -S.

[-]D'i'
--isoaut_depth'i'   : 'i' is an integer. the C-subroutines isom2 and bravais2
          (isometries, automorphism groups) work with 'depth i'
          use i=0..6. higher depth increases performance and
          storage consumption of these subroutines
[-]B'i'
--isoaut_bachdepth'i': 'i' an integer. the C-subroutines isom2 and bravais2
          (isometries, automorphism groups) work with 'bachdepth i'
          use i=0..dim. higher depth increases performance if the automorphism
          groups of the lattices are small

--iso_use_known_auts: changes strategy for tests for isometry: if
         (some) automorphisms of the lattice which is passed to isom2
```

```
          as 2nd one are known, these are used to construct orbits of
          the short vectors. This may increase computation time in easy
          cases but should improve performance for most difficult
          cases. implies --groups -IG
           (There is a related parameter ‘‘noaut_norm_threshold’’ that affects
           cases where iso_use_known_auts is set and only one lattice has known
           automorphisms and this lattice has also the smaller maximal diagonal
           entry in its Gram matrix with respect to a Z-base.
           Then minimizing the
           bound for the needed short vectors contradicts use of the known
           automorphisms. This is resolved as follows: If the necessary
           bound for the short vectors increases by a factor <=
           noaut_norm_threshold then this is acceptable and the
           automorphisms are used (that is: the lattice with
           automorphisms is passed as 2nd lattice parameter to
           the Z-isometry test).
           by now, noaut_norm_threshold is set to 1.3   )
--iso_ignore_known_auts: switch off --iso_use_known_auts (does NOT
           reset --groups) see also --compute_missing_groups above
      /* related hidden parameter (for developers only):
      noaut_norm_threshold */

--use_K_linearity   : the calculation of the automorphism group of hermitian
           lattices and isometry test make use of the hermitian
           structure but thus cannot use reduction of the Z-basis
           by the usual LLL. Only O_K-linear transformations are
           done to find a better basis (the hermitian LLL is
           performed). Switch it off, if the basis found by this
           means is too bad and causes isom/autom troubles
--Z_or_K_linearity  : try Z-reduction of the trace form and use either the
           hermitian structure or the Z-reduction in isom2/autom2 depending on
           what strategy leads to smaller lists of ’short vectors’ involved in
           isom2/autom2).

--herm_lll’i’ ’f’   : perform the reduction of the O_K lattices by a
           hermitian version of LLL with blocksize 2<=’i’<=dim and an
           LLL-constant 0<’f’<=1. (disabled to avoid overflow,
           if hn is compiled without dynamic integers (LEDA or LiDIA))

[-]S’n’
--storage_bound’n’  : the maximal size of the workspace for the construction
           of the orbits will be ’n’ kilobytes. This should fit into the
           memory of your machine. If the workspace cannot hold all neighbour
           vectors, the construction of the orbits needs more time.

--orbit_searchdepth’n’ : if there are so many neighbours, that the neighbour
           vectors do not fit into the workspace (see -S), then vectors that
           have a representative in their orbit that can be found with <=’n’
           iterations of ‘representative-search’ (see description of
           orbits_iter_p_templ.h) are no longer held in the workspace.
           Increasing ’n’ increases computation time
           but allows more vectors to be sorted into orbits (given a fixed -S).
          If no more space in the workspace is left and the maximal search-
          depth 'n' is reached, the orbit construction switches to `pseudo
          orbits', i.e. simply gives a set of vectors that covers all orbits
          (as small as is possible with the given limits) but without exactly
          computing the orbits. If this happens, the order of the orbits
          cannot be tracked and the adjacence-matrix will not be computed.


OPTIONS CONTROLLING THE OUTPUT:
------------------------------
[-]l
--trace_list        : list the trace forms with their invariants after the
          hermitian forms. Do not check them for isometries. The format of
          the list is similar to that of 'invar' and 'isolist'

[-]t
--trace_isom_search : check the trace forms for isometries and list them
          the format of this list is the same as above
[-]n
--no_trace_list     : no list of trace forms

--shells[i]         : i is an optional integer. print the numbers of vectors
          of (square)norm 0<n <=b  with
            b=  -i              (if i<0)
            b=  maxnorm         (if i is omitted)
            b=  min(i,maxnorm)  (if i>=0)
          where maxnorm is the maximal norm of a base vector *
          generator of its coefficient ideal intersected with Z.

--shells_all        : output at least all theta coefficients that have been
          read or computed (see --shells above)
--not_shells_all    : switch off --shells_all

[-]O[csbwpPLUOTRFGN]: output options specified by single characters
          appended to 'O':
          c : copy or skip comments in input according to the identifying
              character at the beginning (% : copy; & : skip)
              comments start anywhere in input and end with newline.
          s : skip all comments
          b : output of numbers in O_K in blank_format (e.g. 1 0, 0 1, 3 -2)
          w : output of numbers in O_K and K in w_format
              (e.g. 1, w, 3-2w, 2-5w  or  1/3-2w, 1/2+3/5w )
          p : output of numbers in O_K in PARI_format (e.g. 1+3*w)
          P : matrix output is in PARI mode (e.g. [*,* ; *,*] )
          L : matrix output as the lower triangle of the hermitian
              Gram-matrices
          U : the lattices have the same order in output as in input or as
              they are generated
          O : the lattices are ordered depending on their invariants
          T : print the trace lattice (scaled by 1/2 iff discr=0 mod 4)
              together with each lattice. If L=SUM a_i*e_i (i=1..n) is the
              lattice with respect to a pseudo-base e_i and coefficient
              ideals a_i in O_K, the trace lattice is given with respect to
              the Z-base
              a_1.Zpart()*e_1..a_n.Zpart()*e_n,a_1.Zgen2()*e_1...
                                                       ...a_n.Zgen2()*e_n
          R : print the trace lattice after lll and "triple"-reduction
          F : give representatives of the lattices in the 'almost free'
              form, i.e. with respect to a pseudo-base with at most one
              coefficient ideal that is not the ring of integers.
              NOTE: the representatives found without this option may be
              much better (have smaller coefficients of the trace lattice
              with the respect to the Z-base given above)
          G : print a set of generators of the automorphism groups of all
              lattices. The automorphisms are given as matrices with respect
              to the same base as the Gram matrix. The automorphism
              matrices Gi are subject to  Gi*H*(~Gi).transpose() == H
              (where H is the Gram matrix). (see also 'hn --invar --help')
              If the option --set_isoaut_options(-S'n') with n>0 is given
              and only a subgroup is computed then this subgroup is
              printed after "Aut_subgroup"
          K : print elements of the automorphism group if these have been
              computed but do not compute missing information. Either a
              generating set of the group is printed after 'Aut' or the
              known automorphisms are printed after Aut_subgroup
          N : do not print automorphisms even if they are known.

CAUGHT SIGNALS:
==============
During the construction of orbits of neighbour-vectors the signal SIGINT
is caught. If the process gets this signal, the construction of orbits
is stopped. The pre-orbits found by then are processed further, the rest will
be processed later. This will save computation time but lead to a partition
of the set of neighbour vectors into pre-orbits that may be finer than the
actual orbit partition. The output on standard error indicates, when a (large)
orbit construction is going on by lines of the form
orbits_iter_p<.>: processed 100000 elements, 4450840 pre-orbits
or later
orbits_iter_p<.>: processed 8000000 elements, 367800 pre-orbits.

During the neighbour construction (when the variable 'porbits' holds the pre-
computed orbits or pre-orbits) the signal SIGUSR1 is caught. If the process
gets this signal the currently precomputed orbits are dumped in a file
orbit_dump.XXXX  (where XXXX contains the discriminant of K, the dimension and
the prime) and this is notified in the logfile. In a recovered run, this
information may be used (if some calling parameters match) to avoid repeating
(part of) the orbit construction. This file will automatically be deleted, if
the current expansion is completed.


KNOWN PROBLEMS:
==============
sometimes subalgorithms do not end within an acceptable period of time.
Because of the logfile, it may be possible to turn off this subalgorithm
```

```
or change its parameters and restart without much loss. To detect which
subalgorithm caused the problem, you might
-- look at the intermediate output on standard error and guess
-- look at the output                                 "    "
-- look in the logfile (see --recover)                "    "
-- attach the debugger to the
   running process and look at the backtrace. (If this does not work,
   restart with the program hn.debug (if it exists) and retry).
-- just try all possibilities listed below

The following functions are known to cause problems without notifying it:
autom, isom2 : although in most cases the runtime depends only on the
     dimension (over K) of the lattices (<=10, 11 is possible) , there are
     very rare examples where they fail in dimension 6 (or bigger).
     Try to change depth parameter (-D'n') or Bacher prameter (-B'n').
     For autom: switch off the computation in the special situation with
     --no_mass, -A100000 or in the whole run with --no_autom. It may also be
     useful to improve the reduction using e.g. --herm_lll3 0.99 (see below).

bigherm_lattice::block_lll, block_lll2 : In rare cases there is a numerical
     explosion causing the routine to run for hours or forever. If the
     action is --invar or reading a logfile try to avoid reduction by
     --no_reduce. Or change the parameters of this process, e.g.
     --herm_lll2 0.5

herm_lattice_class::vector_bound, theta_series::theta_series :
     computation of the thetaseries up to a bound. Runtime depends on the
     dimension and the bound. With --shells (or perhaps as a default) this
     bound depends on the current gram matrix, what will cause problems in
     dimensions >=8 in some cases. Use --shells'n' with negative 'n', e.g.
     --shells-1
```

2.5. **Miscellaneous.** In this subsection I list (alphabetically) miscellaneous routines which have been programmed in the project.

decomp: The program **decomp**, written by Frank Vallentin, computes the decomposition of a given lattice into its irreducible components. I could recompile it, but the attempt to run it crashed with

```
*** glibc detected *** decomp: double free or
corruption (fasttop): 0x08055458 ***
```

I refrained from attempts to debug it.

gsymbol: The program **gsymbol**, written by Schiemann computes the genus symbol (as defined in [2]) of a given (definite or indefinite) integral quadratic form over **Z**. It could be recompiled and ran in simple tests.

herm_mass: The program **herm_mass**, written by Schiemann, computes the mass of the hermitian genus of the sum of m squares over a given imaginary quadratic number field. It could not be recompiled.

perfrk: the program **perfrk**, written by Axel Pawellek, computes the perfection rank (see [11]) of a lattice. I could not compile it at first try, but the only problem seems to be a proper link to **Lapack**, being in a hurry I reserved that for later.

rootsystem: The program **rootsystem**, written by Axel Pawellek, computes the root system of a given lattice. I could not recompile it, due to LiDIA problems.

shvec: The program **shvec**, written by Frank Vallentin, computes lists of short vectors and the theta series of degree 2. According to Vallentin it has been "almost not" tested. I could recompile it but didn't test it.

sort-list: This program isn't really well documented. It is written by Schiemann and may have been abandoned for some reason (superseded by gsymbol?). According to comment lines found in the main program it takes a symmetric matrix S (Gram matrix) from standard input and gives the determinant $\det(S)$, for all primes p dividing $2 * \det(S)$ the p-adic symbol of S and the signature of S. The symbols are those in Conway-Sloane. The 2-adic symbol is the unique symbol given there. I didn't seriously try to compile it.

strongly_modular: This program isn't really well documented. It is apparently written by Vallentin and seems just to test whether the input lattice is strongly modular in the sense of Quebbemann [17]. I could compile but didn't yet test it.

3. Problems

One rather easy type of problem is the change of environment. Schiemann wrote his programs for our HP-UX workstations with some alternatives of using egcs on LINUX machines. Consequently, in the makefiles one has to change all path declarations and some of the flags set. In some cases I experimented with that and was successful. Generally this is something at which I am not particularly good but it should not be a serious problem for an expert.
Another not quite so easy type of problems may be that not all storage allocation procedures which Schiemann used are still permitted and working. At the time when the programs were written storage problems were a limiting factor for number theoretic programming and Schiemann spent a lot of effort in optimizing storage allocation (in 1995 my newly bought HP-UX station had 144 (=16+128) MB of RAM and was the biggest machine in the pure math part of the department). My hope is that an experienced programmer can locate the problematic parts of storage allocation and fix the problems.
Apart from that there are two main types of problems:

3.1. Changes in the C standard.

Some changes in the standards for C and C++ require adaptations. Relatively harmless are changes in the names of files that need to be included, e. g. replacing iostream.h by iostream, including stdlib.h and string.h in several places, adding `using namespace std;` in several files. The files making up gsymbol conflict with some other changed conventions which were easy to fix, I list them in my comments file in that directory.
More serious are changes in the allowed programming style in C++. In particular, the way in which Schiemann made friend declarations in his templates is now illegal. It seems that this can be fixed by using forward declarations. At least in gsymbol that appeared to work:
I changed the original header file `sym_matrix_templ.h` in the following way:

```
/* The following forward declarations inserted by RSP, Jan 2009,
in order to avoid errors of illegal use of friends*/
template<class T>
class sym_matrix;
template<class T>
```

```
void ijvertausch(sym_matrix<T>& A,int ii,int jj);
template<class T>
void ijdiag0add(sym_matrix<T>& A,register int ii,register int jj);
template<class T>
int compare(const sym_matrix<T>& A,const sym_matrix<T>& B);
template<class T>
int transform(sym_matrix<T>& A);
template<class T>
void make_primitive(sym_matrix<T>& A,T &factor);
template<class T>
int jacobistep(sym_matrix<T>& A,T& z,T& n,T*& d);
template<class T>
void jacobidet(sym_matrix<T>& f,T& det,int* dimplus,
               int* dimnull,int preserve_matrix);
/*end inserted forward declarations*/
template<class T>
class sym_matrix{
private:
 // the order of elements is essential for constructor-initialisation
 int Dim;
 int maxdim;
 T * mat;

void mcopy(const T* Bmat) // Bmat!=this->mat
 {int i; T *ap; const T *bp;
  for(i=(Dim*(Dim+1))>>1,ap=this->mat,bp=Bmat  ;i;
i--,*(ap++)=*(bp++));}

public:
 // Konstruktoren
 sym_matrix(int vdim=0):Dim(vdim<0? 0:vdim),maxdim(Dim),
     mat((Dim>0)? new T[(maxdim*(maxdim+1))>>1]:(T*)NULL)
  {if(Dim>0 && mat== NULL)
    {cout <<"new failed in sym_matrix<T> constructor\n";exit(-1);};}

 sym_matrix(const sym_matrix<T>& B);/*:
   Dim(B.Dim),maxdim(B.Dim),mat((Dim>0)? new
T[(maxdim*(maxdim+1))>>1]:NULL)
  {if(&B==this){cout<<"initialisation from oneself not possible\n";
exit(-1);}
   mcopy(B.mat);}
*/
#ifdef math_types_h
 sym_matrix(const smatrix& B);               // copy from smatrix
 sym_matrix<T>& operator=(const smatrix& B); //   "          "
#endif
 ~sym_matrix(){delete[]mat;Dim=maxdim=0;mat=NULL;}
   // destructor, leaves behind "empty" matrix

 sym_matrix<T>& operator=(const sym_matrix<T>& B);
 sym_matrix<T>& suck(sym_matrix<T>& B)
// flat copy, leaves f empty
```

```
   {if(&B!=this)
     {if(mat!=NULL) delete[]mat;
      mat=B.mat;Dim=B.Dim;maxdim=B.maxdim;B.mat=NULL;B.Dim=B.maxdim=0;}
    return *this;}
 T& coeff(int i,int j)
  {int h;if(i>j){h=i;i=j;j=h;}
   return this->mat[(j*(j+1)>>1)+i];}
 T coeff(int i,int j)const
  {int h;if(i>j){h=i;i=j;j=h;}
   return this->mat[(j*(j+1)>>1)+i];}
 int principal_minor(int i){if(i<=maxdim){Dim=i;return 1;}
else return 0;}
 int dimension()const{return Dim;}
 int dim()const{return Dim;}
 int max_dimension()const{return maxdim;}
 int max_dim()const{return maxdim;}

 // operators
 sym_matrix<T>& operator*=(const T& n);

 friend int compare<>(const sym_matrix<T>& A,const sym_matrix<T>& B);
 friend void ijvertausch<>(sym_matrix<T>& A,int ii,int jj);
 friend void ijdiag0add<>(sym_matrix<T>& A,register int ii,
register int jj);
 friend int transform<>(sym_matrix<T>& A);
 friend void make_primitive<>(sym_matrix<T>& A,T &factor);
 friend int jacobistep<>(sym_matrix<T>& A,T& z,T& n,T*& d);
 friend void jacobidet<>(sym_matrix<T>& f,T& det,int* dimplus,
              int* dimnull,int preserve_matrix);
/*inserted <> into jacobidet, RSP Jan 2009*/
 };       // end sym_matrix<T>
```

I do not know whether there are more problems of the same type along the way.

3.2. **Routines for big integers.** This is presently a rather serious problem. Schiemann found it convenient to use some routines from LEDA for handling big integers, big floats, and for sorting. As an alternative he provided switches in the Makefile and in selected header files (`include_param.h`) to use LiDIA instead. At the time Schiemann wrote his programs, LEDA was freely available in the scientific domain (though otherwise already a commercial product), meanwhile it is totally commercial and of no use for developing free software. LiDIA can still be downloaded but is no longer maintained and could not be compiled with gcc4.3. Consequently someone has to rearrange the routines that involve bigints and bigfloats; this goes definitely beyond my powers and time constraints.
For an experienced C++ programmer however, it should be manageable: There are, as far as I can see, (see also below) only three LEDA types used by Schiemann, namely *integer, bigfloat, sortseq*, these are well documented in the LEDA manual and one can even obtain the source files. Moreover, Schiemann's files are rather well documented by comment lines and there is only a small number of files dealing with the bigint/bigfloat routines explicitly.
Without that issue being properly addressed or better solved the problem is particularly bad for **hn** and for **isolist**. At present, both programs produce endless error messages and it is hard to distinguish which ones are indirectly caused by

missing LEDA/LiDIA support and which ones are due to changed programming style alone.
Let us look at the LEDA/LiDIA problem more closely: In the Copyright file of the Hn-directory Schiemann writes:

```
 Depending on the switches in the file 'include_param.h',
this program uses the integer interface to GNU's gmp
from LiDIA and the LiDIA type bigfloat.
The common setting is to prefer the data types
'integer' and 'bigfloat' from LEDA instead of
LiDIAs 'bigint','bigfloat'. (LiDIAs bigints need
a lot of time for copying, because they do not follow
a 'handle' concept as LEDAs integers.
With small numbers, LEDAs integers need about 1/10th
of the time compared with LiDIA. But that may depend
on the installation of LiDIA/LEDA on your system)
```

In addition, he uses the LEDA data type `sortseq` for sorting his lists. Information on these types can be obtained from the LEDA website, my version of the ZAK archive contains pdf files obtained from there.

4. Further goals

Of course the new SAGE library for quadratic forms should at least recover the functionality that our old programs had and should be capable of doing the things that MAGMA can do presently.
By work of Hironaka/Sato [4] and Yang [26, 27] local representation densities can at least in principle be explicitly calculated. This should be available in SAGE. Since the formulas of Hironaka/Sato are rather complicated it may be better to use Katsurada's recursion formulae [6, 7] instead. Since the densities are given by known formulas for primes which don't divide the product of the discriminants of the representing and the represented form such a program should also compute the infinite product over all primes of the densities for a pair of global forms and identify the zeta- and $L-$ values occurring in them.
For the cryptography community it would of course be of interest to have the best short vector methods and the best variants of the LLL algorithm available.
It would also be nice to have implementations of the coding theoretic constructions of lattices described in [2] and other places.

Computing with indefinite quadratic forms is not really far developed. It should be easy to write a program that computes the number of spinor genera in a given genus (and hence in indefinite cases the class number); this will also be useful for definite forms. Such a program should then also give for each pair of spinor genera the congruence condition for primes for which the neighbourhood graph contains both spinor genera.
It will be a bit more intricate to actually decide for two given forms whether they are in the same spinor genus (thereby obtaining an equivalence test for indefinite forms in at least 3 variables), but one should try how far one can get. It appears that this is the easiest way to obtain an equivalence test for indefinite forms. The problem becomes easy once one has a rational transformation which transforms one form into the other, so one is lead to also consider the quite hard problem of constructing rational isometries between two forms which are known to be rationally equivalent, for example by the result of a computation of their genus symbols.

For indefinite anisotropic forms over $\mathbf{Q}$ (which exist only in dimension ≤ 4) representatives of orbits of representations can in principle be effectively determined; it would be nice to really do this by computer calculation.

My interest in programming quadratic forms comes presently mainly from applications to modular forms. Especially for Siegel and (quaternion) hermitian modular forms there aren't many methods available for the explicit construction of modular (especially cuspidal) forms, in particular since it is not known whether Hecke eigenvalues determine the form uniquely, and there is still a lot of experiments to be done.
With Schiemann's programs we could compute Siegel theta series of degree 2, but the computation of spaces generated by a given set of theta series did in fact only give a lower bound for the dimension of the space computed since we had no explicit bound available for the number of Fourier coefficients that are needed to characterize a Siegel modular form. With respect to that question there has meanwhile been a lot of progress mainly due to Poor and Yuen [14, 15, 16] (see also the dissertation of Klein [9] for the hermitian case and for a discussion of level $\neq 1$). The last problem also arises for Hilbert modular forms, where bounds (which are not particularly practical) have been derived in [25, 1]. As far as possible such bounds should be implemented into programs that determine bases of spaces of modular forms generated by theta series of the respective type. For the computation of theta series it should also be possible to consider inhomogeneous theta series (theta series attached to cosets of lattices) and perhaps "mixed" Siegel theta series, i. e., theta series of degree n, where the n vectors whose Gram matrix is computed belong to different lattices (such theta series occur for example if one considers the Fourier expansion with respect to certain boundary components). In addition, theta series with spherical harmonics and with characters should be computed in all cases (elliptic, Hilbert, Siegel, hermitian).

Quaternion algebras play an important role both in modular forms theory and in quadratic forms theory. MAGMA has some facilities for handling these; they should be matched by SAGE. Desirable is e. g. an implementation of the algorithm for computing ideal classes which was recently presented in [8], see also [12], and an implementation of the class number formulas of Vigneras (which have been programmed in MAGMA in the Diplom thesis of my PhD student U. Gebhardt). I understand that an implementation of the algorithms for Hilbert modular forms attached to ideals in quaternion algebras that have been presented by Dembelé is under way. Work on similar constructions for orders of arbitrary class number and using harmonic polynomials is the subject of the PhD thesis of U. Gebhardt which is about to be finished; due to the current lack of quaternion algebra routines in SAGE she is using MAGMA again! A good library for Hilbert modular forms could also be used to test conjectures about cohomology groups of Hilbert modular surfaces and to test modularity conjectures.
Once the basic quaternion algebra routines have been established in SAGE it will be desirable to also compute genera of quaternion hermitian lattices and their quaternionic theta series; there is a lot of fascinating conjectures around generalizing e. g. the Jacquet-Langlands correspondence.

References

[1] S. Baba, K. Chakraborty, Y. Petridis: On the number of Fourier coefficients that determine a Hilbert modular form, Proc. Amer. Math. Soc. 130 (2002), no. 9, 2497–2502

[2] J. H. Conway, N. J. A. Sloane: Sphere packings, lattices and groups, third edition, Springer-Verlag, New York, 1999. lxxiv+703 pp

[3] Boris Hemkemeier: Twoneighbours, available at `http://twoneighbours.origo.ethz.ch/wiki/twoneighbours`

[4] Y. Hironaka, F. Sato: Local densities of representations of quadratic forms over p-adic integers (the non-dyadic case). J. Number Theory 83 (2000), no. 1, 106–136.

[5] W. Jagy, I. Kaplansky, A. Schiemann: There are 913 regular ternary forms. Mathematika 44 (1997), no. 2, 332–341.

[6] H. Katsurada: A certain formal power series of several variables attached to local densities of quadratic forms I, J. Number Theory 51 (1995), no. 2, 169–209

[7] H. Katsurada, M. Hisasue: A recursion formula for local densities, J. Number Theory 64 (1997), no. 2, 183–210

[8] M. Kirschmer, J. Voight: Algorithmic enumeration of ideal classes for quaternion orders, Preprint 2008, arXiv:0808.3833v1 [math.NT]

[9] M. Klein: Verschwindungssätze für Hermitesche Modulformen sowie Siegelsche Modulformen zu den Kongruenzuntergruppen $\Gamma_0^{(n)}(N)$ und $\Gamma^{(n)}(N)$, Dissertation Saarbrücken 2004

[10] M. Kneser: Klassenzahlen definiter quadratischer Formen, Arch. Math. 8 (1957), 241–250.

[11] J. Martinet: Les réseaux parfaits des espaces euclidiens.[Perfect lattices of Euclidean spaces] Masson, Paris, 1996. iv+439 pp.

[12] A. Pizer: An algorithm for computing modular forms on $\Gamma_0(N)$. J. Algebra 64 (1980), no. 2, 340–390.

[13] W. Plesken, B. Souvignier: Computing isometries of lattices, J. Symbolic Comput. 24 (1997), no. 3-4, 327–334.

[14] C. Poor, D. Yuen: Linear dependence among Siegel modular forms. Math. Ann. 318 (2000), no. 2, 205–234.

[15] C. Poor, D. Yuen: Computations of spaces of Siegel modular cusp forms. J. Math. Soc. Japan 59 (2007), no. 1, 185–222.

[16] C. Poor, D. Yuen: Dimensions of cusp forms for $\Gamma_0(p)$ in degree two and small weights. Abh. Math. Sem. Univ. Hamburg 77 (2007), 59–80

[17] H.-G. Quebbemann: Atkin-Lehner eigenforms and strongly modular lattices, Enseign. Math. (2) 43 (1997), no. 1-2, 55–65.

[18] R. Scharlau, B. Hemkemeier: Classification of integral lattices with large class number, Math. Comp. 67 (1998), no. 222, 737–749.

[19] R. Scharlau, R. Schulze-Pillot: Extremal Lattices, p. 139-170 in: B.H. Matzat, G.-M. Greuel, G. Hiss (Eds.): Algorithmic Algebra and Number Theory. Springer 1998

[20] R. Scharlau, A. Schiemann, R. Schulze-Pillot: Theta Series of Modular, Extremal, and Hermitian Lattices, in: M.-H. Kim, J. S. Hsia, Y. Kitaoka, R. Schulze-Pillot (Eds): Integral quadratic forms and lattices (Seoul, 1998), 221–233, Contemp. Math., 249, Amer. Math. Soc., Providence, RI, 1999.

[21] A. Schiemann: Ternary positive definite quadratic forms are determined by their theta series. Math. Ann. 308 (1997), no. 3, 507–517.

[22] A. Schiemann: Classification of Hermitian forms with the neighbour method, J. Symbolic Comput. 26 (1998), no. 4, 487–508.

[23] A. Schiemann: Tables of integral hermitian forms, available at `http://www.math.uni-sb.de/ag/schulze/Hermitian-lattices/`

[24] R. Schulze-Pillot: An algorithm for computing genera of ternary and quaternary quadratic forms, p. 134-143 in Proceedings of the International Symposium on Symbolic and Algebraic Computation (ISSAC) Bonn 1991

[25] F. Wichelhaus: Wann verschwinden Hilbertsche Modulformen?, Dissertation Saarbrücken 1999

[26] T. Yang: An explicit formula for local densities of quadratic forms. J. Number Theory 72 (1998), no. 2, 309–356

[27] T. Yang: Local densities of 2-adic quadratic forms. J. Number Theory 108 (2004), no. 2, 287–345