S. Hamdy (Ed.)

LiDIA

A library for computational number theory

Reference Manual

Edition 2.1, May 2001

S. Hamdy (Ed.)

LiDIA— A library for computational number theory
Reference manual, Edition 2.1

Technische Universität Darmstadt
Fachbereich Informatik
Institut für Theoretische Informatik
Alexanderstraße 10
D–64283 Darmstadt
Germany

Contents

Copyright

Chapter 1

Overview

This chapter describes the overall organization of the LiDIA system.

1.1 Introduction

In early 1994 our research group in computational number theory at the Department of Computer Science at the Universität des Saarlandes, Germany, was working on implementations of algorithms for factoring integers, determining discrete logarithms in finite fields, counting points on elliptic curves over finite fields, etc. In those implementations three different multiprecision integer packages were used. The code written for those implementations was hard to read and hardly documented as well. Therefore, many basic routines were written over and over again, often not very efficiently.

Because of this we decided to organize all the software in a library which we called *LiDIA* . The organization of this task, the development of the general concept of the library and the implementation of the basic components like the *kernel*, the *interface* and the majority of the components for doing arithmetic were designed and implemented by Thomas Papanikolaou.

Since we find that *object oriented programming* is appropriate for implementing mathematical algorithms and since C++ belongs to the most accepted programming languages in scientific computing, we decided to use C++ as the *implementation language* for LiDIA. To guarantee easy *portability* of LiDIA we decided to have a very small machine dependent kernel in LiDIA. That kernel currently contains the multiprecision integer arithmetic and a memory manager. All LiDIA objects are implemented in C++ and compiled with many different compilers on various architectures.

Although not in the public domain, LiDIA can be used freely for non commercial purposes (see copyright notice, page 1).

1.2 Design notes

It is a serious problem to decide which multiprecision integer package and which memory manager should be used in LiDIA. There are competing multiprecision integer packages, for example

- the Berkeley MP package;
- the GNU MP package by Torbjörn Granlund (`http://www.swox.com/gmp/`);
- the cln package by Bruno Haible (`http://clisp.cons.org/~haible/packages-cln.html`);
- the piologie package from HiPiLib (`http://www.hipilib.de/piologie.htm`);
- the libI package by Ralf Dentzer (available from `http://www.informatik.tu-darmstadt.de/TI/LiDIA/`);

- the freeLIP package, formerly known as LIP, by Arjen Lenstra, now maintained by Paul Leyland (`ftp://ftp.ox.ac.uk/pub/math/freelip/`);

- the BIGNUM routines of the openssl library (`http://www.openssl.org/`).

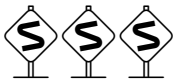
Some of those packages are more efficient on one architecture and some on others[1]. Also, new architectures lead very fast to new multiprecision packages. We decided to make LiDIA independent of a particular kernel and to make it easy to replace the LiDIA kernel. To achieve this, we separate the kernel from the application programs by an *interface* in which the declarations, operators, functions, and procedures dealing with multiprecision integers and the memory management are standardized. All LiDIA classes use the kernel through that interface. They never use the kernel functions directly. Whenever new kernel packages are used only the interface has to be changed appropriately but none of the LiDIA classes has to be altered. In the LiDIA 2.1 package we use the libI package and its memory manager as the default kernel.

We try to develop extremely efficient code for the algorithms we offer in the LiDIA system. Therefore, LiDIA is designed in such a way, that a user can decide in which level of abstraction one implements the programs. Moreover we try to keep the dependency of parts of the library on each other to a minimum and to define interfaces precisely in order to be able to replace modules by more efficient modules in an efficient way.

1.3 Structure of LiDIA

The LiDIA library has 5 levels.

The lowest level contains the *kernel*, which contains a *multi-precision* integer arithmetic and a *memory manager*. Both components are currently written in C to attain maximum speed[2].



The multi-precision integer arithmetic has been an integral part of LiDIA, i.e. the sources have been part of LiDIA's source tree and the multi-precision arithmetic library has been rolled into the LiDIA library. **Beginning with release version 2.1 the multi-precision arithmetic library is no longer a part of LiDIA. In order to use LiDIA, the multi-precision arithmetic library has to be installed, and the user must explicitely link that library to his applications!**

In LiDIA's kernel it is possible to use various multiprecision integer packages since the functions of the kernel are never called directly by any application program of LiDIA but only through the interface on the second level. Therefore, any package supporting the whole functionality of the interface can be used in the kernel. Currently, we support the multiprecision integer packages libI, freelip, GNU MP, cln and piologie.

The second LiDIA level is the *interface* through which C++ applications on higher levels have access to the kernel. By that interface declarations, operators, functions and procedures dealing with multiprecision integers and memory management are standardized. LiDIA applications always use those standards. They never call the underlying kernel directly. The interface for the multiprecision integer arithmetic is realized by the class `bigint` and the interface for the general memory manager is implemented in the class *gmm*.

The third level of LiDIA contains all non parameterized LiDIA applications, the so-called *simple classes*. For example, on the third level there is a class `bigrational` which implements the arithmetic of rational numbers.

On the fourth LiDIA level there are the *parameterized classes* (also called *container classes*). These classes are designed to solve computational problems which are very similar in many concrete applications. In the current version this level includes generic implementations of hash tables, vectors, matrices, power series, and univariate polynomials.

The highest level of LiDIA includes the user interface which consists of an online documentation tool and the *interpreter* lc which makes the LiDIA functions available for interactive use.

[1] As of March 2001, GNU MP seems to be superior, while libI and freelip seem to be deprecated.

[2] However, it is possible to use any other implementation language in the kernel, as long as this language can be called from C++.

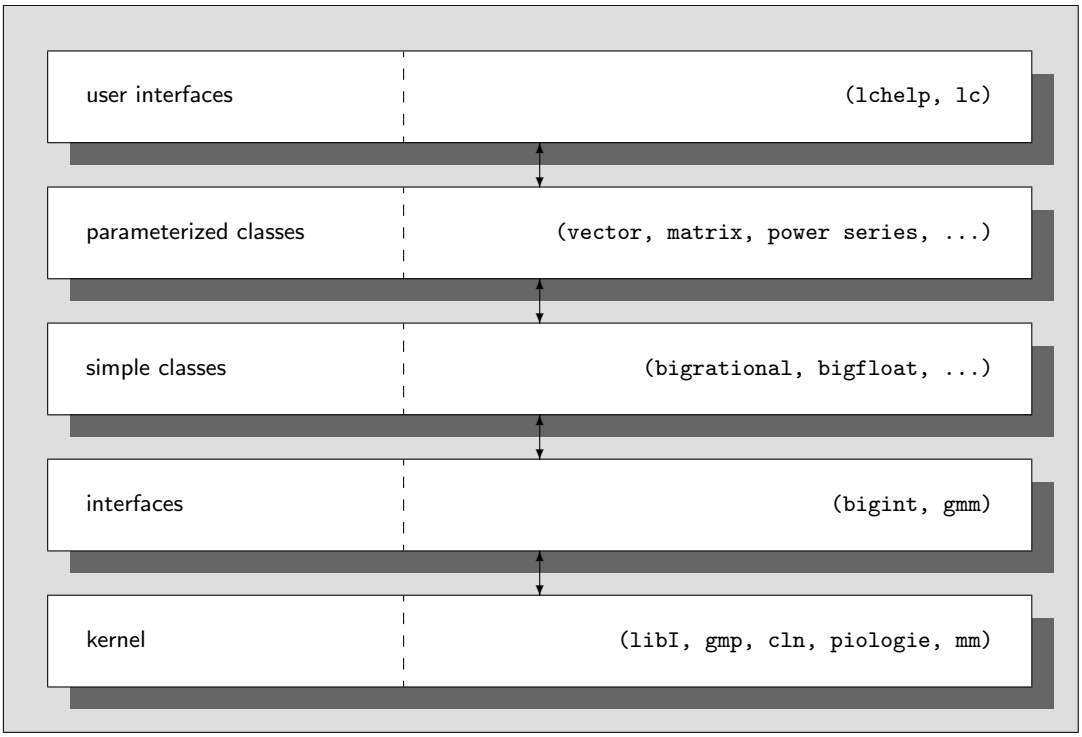


Fig. 1: The levels of LiDIA

1.4 Packages

Since 1994, LiDIA has become quite a big collection of routines and classes, and many people have contributed to its functionality. As the development of LiDIA has shifted towards more and more specialized areas, we have decided to slightly reorganise its design in order to

- remain maintainable and to
- allow an easy integration of new routines.

Therefore, we have divided LiDIA into several packages; the standard packages are the following:

- The LiDIA base package includes all the basic arithmetic and most of the template classes. This package is always required.
- The LiDIA FF package contains the classes which deal with finite fields.
- The LiDIA LA package contains the classes which deal with linear algebra over the ring of rational integers and over $\mathbb{Z}/m\mathbb{Z}$. It requires the LiDIA FF package.
- The LiDIA LT package contains the classes which deal with lattices over the integers and over the reals. It requires the LiDIA FF package, and the LiDIA LA package.
- The LiDIA NF package contains the classes which deal with quadratic number fields, and number fields of higher degree. It requires the LiDIA FF package, the LiDIA LA package, and the LiDIA LT package.
- The LiDIA EC package contains the classes which deal with elliptic curves over the integers and over finite fields. It requires the LiDIA FF package, and the LiDIA LA package.
- The LiDIA ECO package contains classes and code to count the rational points of elliptic curves over GF(p) and GF(2^n).

1.5 Future developments

In the future we will expand the set of classes and algorithms by offering additional packages. Methods which are available in LiDIA 2.1 will also be available in future releases, so as to ensure compatibility of user implemented algorithms with new versions of LiDIA. If you develop algorithms of general interest using LiDIA, we would appreciate it if you let us include them in the LiDIA distribution (possibly as a new package) or send information about your implementation to other LiDIA users.

The list of currently available packages can be found at the LiDIA WWW Home Page of LiDIA:

`http://www.informatik.tu-darmstadt.de/TI/LiDIA`

1.6 Addresses of the LiDIA-group

After having picked up the source of LiDIA 2.1, for example by anonymous ftp via

`ftp://ftp.informatik.tu-darmstadt.de/pub/TI/systems/LiDIA`,

you should subscribe to the LiDIA mailing lists `LiDIA@cdc.informatik.tu-darmstadt.de` and `LiDIA-announce@cdc.informatik.tu-darmstadt.de`. You can do so by sending email to `LiDIA-request@cdc.informatik.tu-darmstadt.de` and `LiDIA-announce-request@cdc.informatik.tu-darmstadt.de`, respectivly. Put the word "subscribe" in the body.

Alternatively, you can use the Web-interface at `http://www.cdc.informatik.tu-darmstadt.de/mailman/listinfo/` to subscribe to the LiDIA mailing lists.

Former LiDIA manuals requested LiDIA users to send an email to `lidia@cdc.informatik.tu-darmstadt.de`. This is no longer desired. Instead, subscribe to `LiDIA` and `LiDIA-announce` to get the latest news on LiDIA.

If you think you have found a bug in LiDIA (or incomplete/incorrect documentation), please report to

`LiDIA-bugs@cdc.informatik.tu-darmstadt.de`

Note that we will probably not be able to reconstruct a bug without a detailed description. We need to know at least the LiDIA version, the operating system, and the compiler that you were using.

1.7 Acknowledgements

The following people have contributed to LiDIA (in alphabetical order):

Detlef Anton, Werner Backes, Jutta Bartholomes, Franz-Dieter Berger, Ingrid Biehl, Emre Binisik, Oliver Braun, John Cremona, Sascha Demetrio, Thomas F. Denny, Roland Dreier, Friedrich Eisenbrand, Dan Everett, Tobias Hahn, Birgit Henhapl, Michael J. Jacobson, Jr., Patrick Kirsch, Thorsten Lauer, Frank J. Lehmann, Markus Maurer, Andreas Müller, Volker Müller, Stefan Neis, Thomas Papanikolaou, Thomas Pfahler, Thorsten Rottschäfer, Dirk Schramm, Victor Shoup, Nigel Smart, Thomas Sosnowski, Anja Steih, Patrick Theobald, Oliver van Sprang, Damian Weber, and Susanne Wetzel.

Special thanks to Thomas Papanikolaou for originating LiDIA and Prof. Johannes Buchmann for supporting the project.

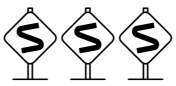
Chapter 2

How to use the LiDIA library

This chapter describes how to build the LiDIA library and executables that use LiDIA.

2.1 Building and installing LiDIA

Obtain the latest sources from `ftp://ftp.informatik.tu-darmstadt.de/TI/systems/LiDIA` or any mirror thereof.



Releases of LiDIA prior to version 2.1 had the multi-precision arithmetic library rolled into the LiDIA library. The neccessary sources (e.g. for libI, cln, or GNU MP) has been part of each LiDIA distribution. This has changed. **Beginning with release version 2.1, the multi-precision arithmetic library is no longer part of LiDIA. In order to use LiDIA, a suitable multi-precision arithmetic library has to be installed on your system, and the user must explicitely link that library to his applications!**

To build LiDIA, your C++ compiler must be ISO C++ compliant to some extent. That is, your C++ compiler must provide the type `bool`, support inlining (implicit and explicit via the keyword `inline`), support `mutable` class members, support ISO C++ style casts (`const_cast<...>` and `static_cast<...>`), support `explicit` contructors, and support explicit template instantiation by ISO C++. Old C++ constructions that conflict with ISO C++ are not supported. Moreover, it is assumed that some basic classes of the standard C++ library such as `iostream`, `fstream`, and `string` are present and work as described in the ISO C++ standard. However, LiDIA makes currently no use of run time type information. (However, this may change in the future). If your C++ compiler doesn't satisfy these requirements, you should upgrade. Sorry for the inconvenience, but this is what a standard is for.

Very few parts of LiDIA require that your system supports the POSIX standard. If your system doesn't support POSIX, these parts will not be built (that is, they will do nothing, e.g. the timer class). All other parts of LiDIA will not be affected. Before configuring/building LiDIA, please check whether you have to set special defines (such as `_POSIX_SOURCE`) for the preprocessor or whether you have to link special libraries (such as `libposix`) to an executable in order to get the POSIX facilities.

2.1.1 Building with `configure`

LiDIA has an Autoconf/`configure`-based configuration system. On Unix-like systems just type

```
./configure
make
```

Additionally, say

```
make examples
```

if you want to build all examples.

Install the library and the headers with

```
make install
```

The default installation directory is `/usr/local`, but this may be altered by giving the `--prefix` option to `configure`.

Previous releases of LiDIA installed the library in the directory *libdir*/LiDIA/*system*/*compiler*/, where *system* denotes the operating system and *compiler* denotes the compiler used. This release and all future releases install the library plainly in *libdir*/. To obtain the old behavior, use the `--libdir` option of `configure`.

If your compiler needs special flags or defines in order to enable the POSIX facilities, pass them to `configure` by setting the `CPPFLAGS` variable; likewise, if special libraries have to be linked to executables in order to get the POSIX facilities, pass them to `configure` by setting the `LIBS` variable. Example:

```
CPPFLAGS=-D_POSIX_SOURCE LIBS=-lposix ./configure
```

Configuration options

All usual `configure` options are available (type `configure --help` to see them all), plus the following ones:

`--enable-inline` If set to 'yes', the multi-precision arithmetic routines from the underlying kernel are inlined, otherwise seperate function calls are generated. The default is 'yes'.

`--enable-exceptions` If set to 'yes', then LiDIA is built with support for exceptions and errors are reported by exceptions. If set to 'no', then LiDIA is built without support for exceptions. (g++ compiler switch '-fno-exceptions'.) Consult the description of class LiDIA::BasicError for details. The default is 'yes'.

`--enable-namespaces` If set to 'yes', then all of LiDIA's symbols will be defined in the name space `LiDIA` (your C++ compiler must support name spaces). The default is 'yes'.

`--enable-assert` If set to 'yes', the assert macros will be activated by not defining `NDEBUG`. The default is 'no'.

`--enable-ff` If set to 'yes', the finite-fields package will be built. The default is 'yes'.

`--enable-la` If set to 'yes', the linear-algebra package will be built. Since this package depends on the finite-fields package, the linear-algebra package will be built only if the finite-fields package is built. The default is 'yes'.

`--enable-lt` If set to 'yes', the lattice package will be built. Since this package depends on the linear-algebra package, the lattice package will be built only if the linear-algebra package is built. The default is 'yes'.

`--enable-nf` If set to 'yes', the number fields package will be built. Since this package depends on the lattice package, the number fields package will be built only if the lattice package is built. The default is 'yes'.

`--enable-ec` If set to 'yes', the elliptic curves package will be built. Since this package depends on the lattice package, the elliptic curves package will be built only if the lattice package is built. The default is 'yes'.

`--enable-eco` If set to 'yes', the elliptic curve order package will be built. Since this package depends on the elliptic curves package, the elliptic curve order package will be built only if the elliptic curves package is built. The default is 'yes'.

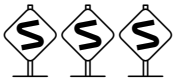
`--enable-gec` If set to 'yes', the elliptic curve generation package will be built. Since this package depends on the elliptic curve order package, the elliptic curve generation package will be built only if the elliptic curve order package is built. The default is 'yes'.

`--with-arithmetic` Determines the multi-precision kernel for LiDIA. Valid values are 'gmp', 'cln', 'piologie', and 'libI'. **Your system must provide the library of your choice before configuring LiDIA!**

Note for Piologie users on Unix-like systems: For some reason the Piologie library is called `piologie.a` instead of `libpiologie.a`. Rename `piologie.a` to `libpiologie.a`, or create a link **before** calling `configure`, otherwise the `configure` script will not find the Piologie library.

`--with-extra-includes` If the headers of the multi-precision library reside in a directory that is not searched by your C++ compiler, then add the path with this option.

`--with-extra-libs` If the multi-precision library resides in a directory that is not searched by your linker, then add the path with this option.



If you build LiDIA with a C++ library for multi-precision arithmetic (e.g. CLN or Piologie), you **must** use the same C++ compiler to build LiDIA that you used to build the multi-precision arithmetic library, otherwise you will not be able to build executables. At least, if you use different C++ compilers, they must have compatible mangling conventions. For example, GNU g++ 2.x and GNU g++ 3.x are incompatible. Actually, `configure` will detect this indirectly, because the test for the multi-precision library will fail. However, there will be no apparent reason for the failure to unaware users. Thus, before reporting a bug, check this first.

LiDIA's build procedure uses GNU Libtool, which adds the following options to `configure`:

`--enable-shared, --enable-static` These options determine whether shared and/or static library versions shall be built. Both options default to 'yes', thus requiring two object files per source file to be made.

`--with-pic, --without-pic` Normally, shared libraries use position-independent code, and static libraries use direct jump instructions which need to be edited by the linker. The `--with-pic` option enforces position-independent code even for static libraries, whereas `--without-pic` does not demand position-independent code even when building shared libraries. Using one of these options can halven compilation time and reduce disk space usage because they save the source files from being compiled twice. Note that only position-independent code can be shared in main memory among applications; using position-dependent code for dynamically linked libraries just defers the linking step to program load time and then requires private memory for the relocated library code.

`--enable-fast-install` LiDIA comes with some example applications, which you may want to run in the build tree without having installed the shared library. To make this work, Libtool creates wrapper scripts for the actual binaries. With `--enable-fast-install=yes`, the (hidden) binaries are prepared for installation and must be run from the (visible) wrapper scripts until the shared library is installed. With `--enable-fast-install=no`, the binaries are linked with the uninstalled library, thus necessitating a relinking step when installing them. As long as you do not run the hidden binaries directly and use the provided `make` targets for installing, you need not care about these details. However, note that installing the example applications without having installed the shared library in the configured *`libdir`* will work only with the setting `--enable-fast-install=yes`, which is the default. This can be important when preparing binary installation images.

Selecting compiler flags

LiDIA's `configure` script does not make any attempts to find out the best compiler flags for building the library. This may change in the future. Instead, please look into the manual of your C++ compiler and set the `CXXFLAGS` environment variable accordingly (see below).

Many compilers offer a `-fast` flag (SUNpro CC, HP aCC) or `-Ofast` flag (MIPSpro CC) that lets the compiler select the best code optimization for the current architecture. If you use GNU g++, you should provide the appropriate `-m` option, see Sect. *GCC Command Options, Hardware Models and Configurations* in the g++ manual (see `http://gcc.gnu.org/onlinedocs/`).

Provide any desired compiler flag by setting the environment variable `CXXFLAGS` before calling the `configure` script, i.e. on the command line type

```
CXXFLAGS="<your desired flags>" ./configure
```

The `configure` script presets the environment variables `CFLAGS` and `CXXFLAGS` with `-O2`, if these are empty or unset. Moreover, if you are building LiDIA with GNU g++, the `configure` script will append `-fno-exceptions` and `-fno-implicit-templates` to `CXXFLAGS` if these are not provided.

If you are not building with GNU g++, you must provide the appropriate compiler flags to avoid implicit template instantiation (see Sect. 2.3 below).

Moreover, LiDIA doesn't make use of C++ exceptions or run time type information (this may change in future releases). Some compilers allow to disable code generation for these C++ features.

2.1.2 Building without `configure`

On non-Unix like systems such as MS Windows or MacOS, `configure` will probably not work.

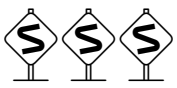
To be finished.

2.2 Building an executable

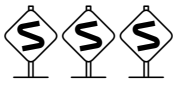
Once LiDIA has been installed properly, its library is used as any other C++ library. We illustrate the use of the LiDIA library by an example. The following little C++ program reads a number from the standard input and prints its factorization on the standard output.

```
#include <LiDIA/rational_factorization.h>

using namespace LiDIA;

int main()
{
    rational_factorization f;
    bigint n;

    cout << "\n Please enter a number: ";
    cin >> n;

    f.assign(n);
    f.factor();

    if (f.is_prime_factorization())
        cout<<"\n Prime Factorization: " << f << endl;
    else
        cout<<"\n Factorization: "<< f << endl;

    return 0;
}
```

It uses the LiDIA factorization class (compare the description of `rational_factorization`). This is done by including `<LiDIA/rational_factorization.h>`. In general, the LiDIA include file for the data type `xxx` is `<LiDIA/xxx.h>`.



All data types and algorithms of LiDIA reside in `libLiDIA.a` **except those for the multi-precision arithmetic**. Thus, unlike with previous releases of LiDIA, you must explicitely link the multi-precision arithmetic library to your executables (see below for examples).



To compile and build your executable, you **must** use the same C++ compiler that you used to build LiDIA, otherwise building your executable will fail. At least, if you use different C++ compilers, they must have compatible mangling conventions. For example, GNU g++ 2.x and GNU g++ 3.x are incompatible. If the linker emits error messages about missing symbols that should

be actually present, this is a good indication that you've used different incompatible C++ compilers. The reason for the failure will not be apparent to unaware users. Thus, before reporting a bug, check this first.

The program `fact.cc` can be compiled and linked using the following command (we assume that you have installed LiDIA and GNU MP in `/usr/local/` and that you have compiled LiDIA on a sparc8 machine using GNU g++):

```
g++ -O fact.cc -I/usr/local/include -L/usr/local/lib -o fact -lLiDIA -lgmp -lm
```

On an OS/2 system, the program can be compiled and linked using the following command (we assume that you have installed LiDIA in `d:\LiDIA` and that you have compiled LiDIA using `EMX`):

```
gcc -O2 fact.cc -Id:/LiDIA/include -Ld:/LiDIA/lib -lLiDIA -lgmp -lbsd -lstdcpp
```

Here is a sample running session:

```
host$ fact <RET>

Please enter an integer 18446744073709551617 <RET>

Prime Factorization:  [(274177,1)(67280421310721,1)]
```

Some programs need information which is stored in the files

`LIDIA_PRIMES` and `GF2n.database`.

The built-in directory, where programs are looking for those files is

`LiDIA_install_dir/share/LiDIA`,

which is the correct path, if you have installed the library via "make install". If you have not installed the library, you must set the environment variables.

2.3 Template instantiation

We describe the method that is used for instantiating templates in LiDIA. All template classes that are used by LiDIA itself are explicitely instantiated. Moreover, some template classes that we expect to be used frequently are also instatiated when LiDIA is built. The advantage of this instantiation method is that it is fool proof and is supported by every C++ compiler.

If you're *not* using GNU g++, then check the compiler manual for the option that inhibits automatic/implicit template instantiation and pass that option to `configure` by adding it to the environment variable `CXXFLAGS`.

For example, if you are using the SUN WorkShop C++ compiler you should type something like

```
CXXFLAGS="-instances=explicit <other flags>" ./configure
```

while a HP aCC user should type something like

```
CXXFLAGS="+inst_none <other flags>" ./configure
```

and a MIPS CC user shoult type something like

```
CXXFLAGS="-ptnone <other flags>" ./configure
```

Note: no guarantee is given that the above flags are the correct ones (this is the reason why we refrained from letting `configure` set these options). In any case, confer your C++ user's manual.

If you want to use the same instantiation mechanism that LiDIA uses, then do the following:

1. Include all files that are necessary to define the type(s) you want to instantiate with.
2. Define `TYPE` (or, in some cases, `TYPE1` and `TYPE2`) appropriately.
3. Some instantiations must be controlled by defining some symbol(s). Look at the particular instantiation source file in `include/LiDIA/instantiate/`.
4. Include the appropriate source file from `include/LiDIA/instantiate/`.

We demonstrate this with an example. Sparse matrices over the ring of integers are instantiated in `src/linear_algebra/instantiate/rm_bigint_sparse.cc`. The code is as follows:

```
#include        <LiDIA/bigint.h>


#define TYPE bigint

#define RING_MATRIX
#define SPARSE


#include        <LiDIA/instantiate/matrix.cc>
```

Note that if you instantiate a template class that is derived from other template classes, you must instantiate these as well. For example, an instantiation of ring matrices requires the instantiation of base matrices with the same type; likewise, an instantiation of field matrices requires the instantiation of ring matrices with the same type.

2.4 LiDIA and other libraries

You may use any other library together with LiDIA without taking any further actions.

2.5 Trouble shooting

Building the library may fail if your C++ compiler is not ISO C++ compliant, doesn't have some required ISO C++ headers (such as `<cstddef>`, `<cctype>`, `<cstdio>`, or `<cstdlib>`, for instance), or doesn't have some required POSIX headers (such as `<unistd.h>`, for instance). Since all major vendors support ISO C++ to some extent, as well as POSIX, this shouldn't happen. Otherwise, you must upgrade your C++ compiler in order to build LiDIA. (Note to HP/UX users: There are two HP C++ compilers, the old, non-ISO C++ compliant CC and the new, ISO C++ compliant aCC. CC will definitively fail to compile LiDIA. However, `configure` will *not* detect aCC, so you must tell `configure` to use aCC by setting the environment variable `CXX` to aCC before you run `configure`.)

If you have problems to compile LiDIA, and you didn't configure LiDIA with the Autoconf `configure` script, then check whether your build environment is sane, i.e. try to compile a small programm that doesn't use LiDIA.

If you have problems to build LiDIA due to memory exhaustion, it will probably help to limit inlining (some packages contain quite large inlined methods and functions). If you are using GNU g++, the appropriate option to limit inlining is `-finline-limit-`N (GCC-2.x) resp. `-finline-limit=`N (GCC-3.x), where N should be chosen to be between 300 and 1000 (GCC's default is 10000). This will also result in faster compilation times and smaller object files for some source files.

Likewise, the GNU g++ compiler flag `-Wreturn-type` might drastically increase memory consumption (see the GCC FAQ, e.g. `http://gcc.gnu.org/faq/`). Note that `-Wall` implies `-Wreturn-type`.

If your linker fails on building an executable, then check that

1. you didn't forget to instantiate all necessary templates that hadn't been already instantiated;
2. you didn't forget to link the multi-precision arithmetik library;
3. you didn't mix up C++ object files and C++ libraries that have been built by different C++ compilers (they are usually not compatible, even different versions of the same C++ compiler may be incompatible).

The LiDIA base package

Chapter 3

Arithmetics over $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$, $\mathbb{C}$

bigint

Name

`bigint` ..... multiprecision integer arithmetic

Abstract

`bigint` is a class that provides a multiprecision integer arithmetic. A variable of type `bigint` can hold an integer with arbitrarily many digits. Since all operators which are available for machine integers (e.g. `int`) are also defined for the type `bigint` you can regard a variable of type `bigint` as an `int` variable without length restriction.

Description

`bigint` is an interface that calls functions provided by the kernel. Thus the representation of variables depends on the data types used in that level. There is a fixed base $B > 0$, which we call `bigint` base and which determines the representation of each integer n by the array $[m_0, m_1, \ldots, m_{l-1}]$, with $l = \lfloor \log_B(n) \rfloor + 1$, $m_i \geq 0$ for $0 \leq i < l$ and

$$n = \mathrm{sgn}(n) \cdot \sum_{i=0}^{l-1} m_i \cdot B^i$$

The coefficients m_i are called *base digits*, l is called the *length* of the representation and the *mantissa* is the array $[m_0, m_1, \ldots, m_{l-1}]$. We assume that the kernel supports such a vector based representation for long integers, i.e., it defines such a base B, the functions `sign` for the calculation of the sign of a long integer, `length` for the length of a long integer, and pointers to the mantissas of represented integers.

Constructors/Destructor

`ct bigint ()`
Initializes with 0.

`ct bigint (const bigint & n)`

`ct bigint (long n)`

`ct bigint (unsigned long n)`

`ct bigint (int n)`
Initializes with n.

`ct bigint (double` n`)`
Initializes with $\lfloor n \rfloor$.

`dt ~bigint ()`

Access Methods

`int` a`.bit (unsigned int` i`) const`
returns the i-th bit of a. Bits are numbered from 0 to a`.bit_length()` $- 1$, where the least significant bit of a has index 0. If $i \geq a$`.bit_length()`, the function always returns 0.

`unsigned long` a`.least_significant_digit () const`
returns the least significant digit of a, i.e., if the mantissa of a is $[m_0, m_1, \ldots, m_{l-1}]$ the function returns m_0 (see basic description, page 19).

`unsigned long` a`.most_significant_digit () const`
returns the most significant digit of a i.e. if the mantissa of a is $[m_0, m_1, \ldots, m_{l-1}]$ the function returns m_{l-1} (see basic description, page 19).

Assignments

Let a be of type `bigint`. The operator `=` is overloaded. The user may also use the following object methods for assignment:

`void` a`.assign_zero ()`
$a \leftarrow 0$.

`void` a`.assign_one ()`
$a \leftarrow 1$.

`void` a`.assign (int` n`)`

`void` a`.assign (long` n`)`

`void` a`.assign (unsigned long` n`)`

`void` a`.assign (double` n`)`

`void` a`.assign (const bigint &` n`)`
$a \leftarrow n$.

Object Modifiers

`void` a`.negate ()`
$a \leftarrow -a$.

`void` a`.inc ()`

`void inc (bigint &` a`)`
$a \leftarrow a + 1$.

`void` a`.dec ()`

`void dec (bigint &` c`)`
$a \leftarrow a - 1$.

`void` a`.multiply_by_2 ()`
$a \leftarrow a \cdot 2$.

`void` a`.divide_by_2 ()`
$a \leftarrow \mathrm{sgn}(a) \cdot \left\lfloor \frac{|a|}{2} \right\rfloor$.

`void` a`.swap (bigint &` b`)`

`void swap (bigint &` a`, bigint &` b`)`
exchanges the values of a and b.

`void` a`.absolute_value ()`
$a \leftarrow |a|$.

`void` a`.abs ()`
$a \leftarrow |a|$.

`void` a`.randomize (const bigint &` b`)`
assigns to a a random value r by means of the random number generator defined in the kernel; $r \in [0, \ldots, b-1]$ if $b > 0$ and $r \in [b+1, \ldots, 0]$ if $b < 0$; if b is equal to zero, the `lidia_error_handler` will be invoked.

Arithmetical Operations

The following operators are overloaded and can be used in exactly the same way as for machine types in C++ (e.g. `int`):

```
                    (unary) -, ++, --
          (binary) +, -, *, /, %, <<, >>, ~, &, |, ^
(binary with assignment) +=, -=, *=, /=, %=, <<=, >>=,
                        &=, |=, ^=
```

To avoid copying, these operations can also be performed by the following functions:

`void negate (bigint &` b`, const bigint &` a`)`
$b \leftarrow -a$.

`void add (bigint &` c`, const bigint &` a`, const bigint &` b`)`

`void add (bigint &` c`, const bigint &` a`, long` b`)`

`void add (bigint &` c`, const bigint &` a`, unsigned long` b`)`

`void add (bigint &` c`, const bigint &` a`, int` b`)`
$c \leftarrow a + b$.

`void subtract (bigint & c, const bigint & a, const bigint & b)`

`void subtract (bigint & c, const bigint & a, long b)`

`void subtract (bigint & c, const bigint & a, unsigned long b)`

`void subtract (bigint & c, const bigint & a, int b)`
$c \leftarrow a - b$.

`void multiply (bigint & c, const bigint & a, const bigint & b)`

`void multiply (bigint & c, const bigint & a, long b)`

`void multiply (bigint & c, const bigint & a, unsigned long b)`

`void multiply (bigint & c, const bigint & a, int b)`
$c \leftarrow a \cdot b$.

`void square (bigint & a, const bigint & b)`
$a \leftarrow b^2$.

`void divide (bigint & c, const bigint & a, const bigint & b)`

`void divide (bigint & c, const bigint & a, long b)`

`void divide (bigint & c, const bigint & a, unsigned long b)`

`void divide (bigint & c, const bigint & a, int b)`
$c \leftarrow a/b$, if $b \neq 0$. Otherwise the `lidia_error_handler` will be invoked.

`void remainder (bigint & c, const bigint & a, const bigint & b)`

`void remainder (bigint & c, const bigint & a, long b)`

`void remainder (bigint & c, const bigint & a, unsigned long b)`

`void remainder (bigint & c, const bigint & a, int b)`

`void remainder (unsigned long & c, const bigint & a, unsigned long b)`

`void remainder (long & c, const bigint & a, long i)`
computes c, with $c \equiv a \pmod{i}$, $|c| < |i|$ and $\operatorname{sgn}(c) = \operatorname{sgn}(a)$.

`long remainder (const bigint & a, long i)`
returns a value c, with $c \equiv a \pmod{i}$, $|c| < |i|$ and $\operatorname{sgn}(c) = \operatorname{sgn}(a)$.

`void div_rem (bigint & q, bigint & r, const bigint & a, const bigint & b)`

`void div_rem (bigint & q, bigint & r, const bigint & a, long b)`

`void div_rem (bigint & q, bigint & r, const bigint & a, unsigned long b)`

`void div_rem (bigint & q, bigint & r, const bigint & a, int b)`

`void div_rem (bigint & q, long & r, const bigint & a, long i)`
computes q and r such that $a = i \cdot q + r$, $|r| < |i|$ and $\operatorname{sgn}(r) = \operatorname{sgn}(a)$.

`void sqrt (bigint & a, const bigint & b)`

a is set to the positive value of $\lfloor\sqrt{b}\rfloor$ if $b \geq 0$. Otherwise the `lidia_error_handler` will be invoked.

`void power (bigint & c, const bigint & a, const bigint & b)`

`void power (bigint & c, const bigint & a, long b)`

$c \leftarrow a^b$. For negative exponents b, the result of this computation will be 0 if $a > 1$, ± 1 respectively if $a = \pm 1$ and 0 if $a = 0$.

Shift Operations

`void shift_left (bigint & c, const bigint & a, long i)`

$c \leftarrow a$<<i, if $i \geq 0$. Otherwise the `lidia_error_handler` will be invoked.

`void shift_right (bigint & c, const bigint & a, long i)`

$c \leftarrow a$>>i, if $i \geq 0$. Otherwise the `lidia_error_handler` will be invoked.

Bit Operations

`void bitwise_and (bigint & c, const bigint & a, const bigint & b)`

$c \leftarrow a \wedge b$.

`void bitwise_or (bigint & c, const bigint & a, const bigint & b)`

$c \leftarrow a \vee b$.

`void bitwise_xor (bigint & c, const bigint & a, const bigint & b)`

$c \leftarrow a \oplus b$.

`void bitwise_not (bigint & c, const bigint & a)`

$c \leftarrow \neg a$.

Comparisons

The binary operators `==`, `!=`, `>=`, `<=`, `>`, `<` and the unary operator `!` (comparison with zero) are overloaded and can be used in exactly the same way as for machine types in C++ (e.g. `int`). Let a be an instance of type `bigint`.

`int a.abs_compare (const bigint & b) const`

`int a.abs_compare (unsigned long b) const`

returns $\mathrm{sgn}(|a| - |b|)$.

`int a.compare (const bigint & b) const`

`int a.compare (unsigned long b) const`

`int a.compare (long b) const`

returns $\mathrm{sgn}(a - b)$.

`bool` a`.is_zero () const`
returns `true` if $a = 0$, `false` otherwise.

`bool` a`.is_one () const`
returns `true` if $a = 1$, `false` otherwise.

`bool` a`.is_gt_zero () const`
returns `true` if $a > 0$, `false` otherwise.

`bool` a`.is_ge_zero () const`
returns `true` if $a \geq 0$, `false` otherwise.

`bool` a`.is_lt_zero () const`
returns `true` if $a < 0$, `false` otherwise.

`bool` a`.is_le_zero () const`
returns `true` if $a \leq 0$, `false` otherwise.

`bool` a`.is_positive () const`
returns `true` if $a > 0$, `false` otherwise.

`bool` a`.is_negative () const`
returns `true` if $a < 0$, `false` otherwise.

`int` a`.sign () const`
returns -1, 0, 1 if $a <, =, > 0$, respectively.

`bool` a`.is_even () const`
returns `true` if $a \bmod 2 = 0$, `false` otherwise.

`bool` a`.is_odd () const`
returns `true` if $a \bmod 2 = 1$, `false` otherwise.

Type Checking and Conversion

Before assigning a `bigint` variable to a machine type variable (e.g. `int`) it is often useful to perform a test which checks whether the assignment can be done without overflow. Let a be an object of type `bigint`. The following methods return `true` if an assignment is possible, `false` otherwise.

`bool` a`.is_char () const`

`bool` a`.is_uchar () const`

`bool` a`.is_short () const`

`bool` a`.is_ushort () const`

`bool` a`.is_int () const`

`bool` a`.is_uint () const`

`bool` a`.is_long () const`

`bool` a`.is_ulong () const`

There methods also exists as procedural versions, however, the object methods are preferred over the procedures.

`bool is_char (const bigint &` a`)`

`bool is_uchar (const bigint &` a`)`

`bool is_short (const bigint &` a`)`

`bool is_ushort (const bigint &` a`)`

`bool is_int (const bigint &` a`)`

`bool is_uint (const bigint &` a`)`

`bool is_long (const bigint &` a`)`

`bool is_ulong (const bigint &` a`)`

`bool` a`.intify (int &` i`) const`

checks whether a can be converted into an `int`. If this can successfully be done, the function will assign $i \leftarrow a$ and return `false`; otherwise, it returns `true` and i will be left unchanged.

`bool` a`.longify (long &` l`) const`

checks whether a can be converted into a `long`. If this can successfully be done, the function will assign $l \leftarrow a$ and return `false`; otherwise, it returns `true` and l will be left unchanged.

`double dbl (const bigint &` a`)`

returns the `double` $x = a$, where x will be set to the special value `infinity`, if a is too large to be stored in a `double` variable; this is used in C++ to denote a value too large to be stored in a `double` variable.

Basic Methods and Functions

Let a be of type `bigint`.

`lidia_size_t` a`.length () const`

returns the number of base digits of the mantissa of a.

`lidia_size_t` a`.bit_length () const`

returns the number of bits of the mantissa of a.

`unsigned int decimal_length (const bigint &` a`)`

return the decimal length of $|a|$.

`void seed (const bigint &` a`)`

initializes the random number generator defined in the kernel with a.

`bigint randomize (const bigint &` a`)`

returns a random value r by means of the random number generator defined in the kernel; $r \in [0, \ldots, a-1]$ if $a > 0$ and $r \in [a+1, \ldots, 0]$ if $a < 0$; if a is equal to zero, the `lidia_error_handler` will be invoked.

High-Level Methods and Functions

`bigint dl (const bigint & a, const bigint & b, const bigint & p)`

computes a solution to $a^x \equiv b \pmod{p}$. The solution is given in the range $0 \leq x < p-1$ and is unique modulo the order of a mod p. If p is not a prime number, `dl()` returns -1. If b is not in the subgroup generated by a, `dl()` returns -2. To use this function include `LiDIA/dlp.h`.

`bool fermat (const bigint & a)`

performs a probabilistic primality test on a using Fermat's theorem (see [17]). The return value will be `false`, if the function recognizes a as composite; otherwise, it will be `true`. For any input $a \leq 1$ this function returns `false`.

`bool a.is_prime (int n = 10) const`

`bool is_prime (const bigint & a, int n = 10)`

performs a probabilistic primality test on a using the Miller-Rabin method (see [17, page 415]). If $a > 1$, then the return value `false` implies that a is composite. If $a > 1$ and `is_prime(a, n)` returns `true`, then a is probably a prime number; the probability, that this answer is wrong is less or equal to $(1/4)^n$. For any input $a \leq 1$ the return value of `is_prime(a, n)` is `false`.

`int jacobi (const bigint & a, const bigint & b)`

returns the value of the Jacobi symbol $\left(\frac{a}{b}\right)$, hence the Legendre symbol when b is an odd prime (see [17, page 28]). The Legendre symbol is defined to be 1 if a is a quadratic residue mod b, 1 if a is a quadratic non-residue mod b and 0 if a is a multiple of b.

`bool cornacchia (bigint & x, bigint & y, const bigint & D, const bigint & p)`

determines (if possible) solutions for $x^2 + |D| \cdot y^2 = 4 \cdot p$, where $D = 0, 1$ modulo 4, $D < 0$, $|D| < 4 \cdot p$, p odd. Sets x, y and returns `true`, if a solution was found. Otherwise it returns `false`.

`void nearest (bigint & z, const bigint & u, const bigint & v)`

computes an integer z with minimal distance $|\frac{u}{v} - z|$ if $v \neq 0$. If there are two possible values, the one with the absolutely greater value will be chosen. If $v = 0$, the `lidia_error_handler` will be invoked.

`bigint a.next_prime () const`

`bigint next_prime (const bigint & a)`

returns the least integer number greater than a that satisfies the primality test in function `is_prime(a)`. For any input $a \leq 1$ this function returns 2.

`bigint a.previous_prime () const`

`bigint previous_prime (const bigint & a)`

returns the greatest integer number smaller than a that satisfies the primality test in function `is_prime(a)`. For any input $a \leq 2$ this function returns 0.

`bigint chinese_remainder (const bigint & a, const bigint & m, const bigint & b, const bigint & n)`

returns the smallest non negative integer which is congruent to a mod m and b mod n. If a simultaneous solution does not exist (note that m and n are not necessarily coprime), the `lidia_error_handler` is invoked.

`void power_mod (bigint & res, const bigint & a, const bigint & n, const bigint & m, int err = 0)`

calculates in *res* the least non-negative residue of a^n modulo m ($res \equiv a^n \bmod m$, $0 \leq res < |m|$) using the right-to-left binary exponentiation method (see [17, page 8]).

For negative exponents n, this function tries to compute the multiplicative inverse of a^{-n}. If it does not exist and $err = 0$, then the `lidia_error_handler` will be invoked. If the multiplicative inverse does not exist and $err \neq 0$, the gcd of a^{-n} and m will be assigned to *res* and the function terminates.

`long a.is_power (bigint & b) const`

`long is_power (bigint & b, const bigint & a)`

tests, whether a is a perfect power. If a is a k-th power, then b is set to k-th root (i.e. $b^k = a$) and k is returned; otherwise the function returns 0.

`void ressol (bigint & r, const bigint & a, const bigint & p)`

returns a square root r mod p with $0 \leq r < p$ by using a method of Shanks. It is assumed that p is prime. A further condition is that $p = 2^s \cdot (2k+1) + 1$ where s fits in a `long`.

`bool a.is_square (bigint & r) const`

`bool is_square (bigint & r, const bigint & a)`

returns `true` if a is a square in $\mathbb{Z}$, and `false` otherwise. If a is a square, r is the root of a.

`bool a.is_square () const`

`bool is_square (const bigint & a)`

returns `true` if a is a square in $\mathbb{Z}$, and `false` otherwise.

Greatest Common Divisor

`bigint gcd (const bigint & a, const bigint & b)`

returns the greatest common divisor (gcd) of a and b as a positive `bigint`.

`bigint bgcd (const bigint & a, const bigint & b)`

calculates the gcd of a and b as a positive `bigint` using a binary method.

`bigint dgcd (const bigint & a, const bigint & b)`

calculates the gcd of a and b as a positive `bigint` using Euclidian division and returns its value.

`bigint xgcd (bigint & u, bigint & v, const bigint & a, const bigint & b)`

returns the gcd of a and b as a positive `bigint` and computes coefficients u and v with $|u| \leq \frac{|b|}{2\gcd(a,b)}$ and $|v| \leq \frac{|a|}{2\gcd(a,b)}$ such that $\gcd(a,b) = u \cdot a + v \cdot b$, i.e. using an extended gcd algorithm.

`bigint xgcd_left (bigint & u, const bigint & a, const bigint & b)`

returns the gcd of a and b as a positive `bigint` and computes a coefficient u such that for some integer t $\gcd(a,b) = u \cdot a + t \cdot b$.

`bigint xgcd_right (bigint & v, const bigint & a, const bigint & b)`

returns the gcd of a and b as a positive `bigint` and computes a coefficient v such that for some integer t $\gcd(a,b) = t \cdot a + v \cdot b$.

`bigint lcm (const bigint & a, const bigint & b)`
returns the least common multiple (lcm) of a and b as a positive `bigint`.

Input/Output

Let a be of type `bigint`.

The `istream` operator `>>` and the `ostream` operator `<<` are overloaded. Furthermore, you can use the following member functions below for writing to and reading from a file in binary or ASCII format, respectively.

`void a.read_from_file (FILE * fp)`
reads a from the binary file `fp` using `fread`.

`void a.write_to_file (FILE * fp)`
writes a to binary file `fp` using `fwrite`.

`void a.scan_from_file (FILE * fp)`
scans a from the ASCII file `fp` using `fscanf`. We assume that the file contains the number in decimal representation.

`void a.print_to_file (FILE * fp)`
prints a in decimal representation to the ASCII file `fp` using `fprintf`.

The input and output format of a `bigint` consist only of the number itself without blanks. Note that you have to manage by yourself that successive `bigint`s have to be separated by blanks.

When entering a `bigint`, the input may be splitted over several lines by ending each line, except the last one (optionally), by a backslash immediately followed by a new line, e.g., you may enter

```
1234567890\
12345678\
232323
```

If a `bigint` is printed, the output may also be splitted over several lines, where each line is ended by a backslash immediately followed by a new line. The number of characters per line can be controlled by the following static member functions.

`void bigint::set_chars_per_line (int cpl)`
When printing a `bigint`, *cpl* characters will be written in one line followed by a backslash and a new line character. If *cpl* is less than 2, line splitting is turned off for printing.

`int bigint::get_chars_per_line ()`
Returns the current value for the number of characters that will be written in one line, when printing a `bigint`.

`LIDIA_CHARS_PER_LINE` is the default value for the number of characters per line which is defined in `LiDIA/LiDIA.h`. It may be changed at compile time.

The following functions allow conversion between `bigint` values and strings:

`int string_to_bigint (const char *s, bigint & a)`
converts the string `s` into the `bigint` a and returns the number of used characters.

`int bigint_to_string (const bigint & a, char *s)`

converts the bigint a into the string `s` and returns the number of used characters. The number of characters allocated for string s must be at least a`.bit_length()`$/3 + 10$.

See also

UNIX manual page mp(3x)

Notes

The member functions `scan_from_file()`, `print_to_file()`, `read_from_file()` and `write_to_file()` are included because there are still C++ compilers which do not handle fstreams correctly. In a future release those functions will disappear.

Examples

```
#include <LiDIA/bigint.h>

int main()
{
    bigint a, b, c;

    cout << "Please enter a : "; cin >> a ;
    cout << "Please enter b : "; cin >> b ;
    cout << endl;
    c = a + b;
    cout << "a + b = " << c << endl;
}
```

For further examples please refer to `LiDIA/src/interfaces/INTERFACE/bigint_appl.cc`.

Author

Thomas Papanikolaou

udigit_mod

Name

`udigit_mod` ..... single precision modular integer arithmetic

Abstract

`udigit_mod` is a class that represents elements of the ring $\mathbb{Z}/m\mathbb{Z}$ by their smallest non-negative residue modulo m. Both, the residue and the modulus are of type `udigit`.

Description

A `udigit_mod` is a pair of two `udigit`s (man, m), where man is called the *mantissa* and m the *modulus*. The modulus is global to all `udigit_mod`s and must be set before a variable of type `udigit_mod` can be declared. This is done by the following statement:

```
udigit_mod::set_modulus(udigit m)
```

By default the modulus of the class is set to 0. Each equivalence class modulo m is represented by its least non-negative representative, i.e. the mantissa of a `udigit_mod` is chosen in the interval $[0, \ldots, |m| - 1]$.

If you have forgotten to set the modulus appropriately when assigning an `udigit` r to an object of type `udigit_mod`, e.g., using the operator `=`, you probably get an error message like "division by zero", because r will be reduced without checking whether the modulus is zero.

Initialization

`void udigit_mod::set_modulus (udigit m)`

sets the global modulus to $|m|$; if $m = 0$, the `lidia_error_handler` will be invoked. If you call the function `set_modulus(m)` after declaration of variables of type `udigit_mod`, all these variables might be incorrect, i.e. they might have mantissas which are not in the proper range.

Constructors/Destructor

`ct udigit_mod ()`

Initializes with 0.

`ct udigit_mod (const udigit_mod & n)`

Initializes with n.

`ct udigit_mod (udigit` n`)`
Initializes the mantissa with n mod m, where m is the global modulus.

`dt ~udigit_mod ()`

Access Methods

Let a be of type `udigit_mod`.

`udigit udigit_mod::get_modulus ()`
Returns the global modulus of the class `udigit_mod`.

`udigit` a`.get_mantissa ()`
returns the mantissa of the `udigit_mod` a. The mantissa is chosen in the interval $[0, \ldots, m-1]$, where m is the global modulus.

Object Modifiers

`void` a`.inc ()`
$a \leftarrow a + 1$.

`void` a`.dec ()`
$a \leftarrow a - 1$.

`void` a`.swap (udigit_mod &` b`)`

`void swap (udigit_mod &` a`, udigit_mod &` b`)`
exchanges the mantissa of a and b.

Assignments

Let a be of type `bigint`. The operator `=` is overloaded. The user may also use the following object methods for assignment:

`void` a`.assign_zero ()`
mantissa(a) $\leftarrow 0$.

`void` a`.assign_one ()`
mantissa(a) $\leftarrow 1$.

`void` a`.assign (const udigit_mod &` n`)`
$a \leftarrow b$.

`void` a`.assign (udigit` n`)`
$a \leftarrow n$ mod m where m is the global modulus.

Comparisons

The binary operators `==`, `!=`, and the unary operator `!` (comparison with zero) are overloaded and can be used in exactly the same way as for machine types in C++ (e.g. `int`). Let a be of type `udigit_mod`.

`bool` a`.is_zero () const`
Returns `true` if the mantissa of a is 0 and `false` otherwise.

`bool` a`.is_one () const`
Returns `true` if the mantissa of a is 1 and `false` otherwise.

`bool` a`.is_equal (const udigit_mod &` b`) const`
Returns `true` if $a = b$ and `false` otherwise.

Arithmetical Operations

The following operators are overloaded and can be used in exactly the same way as for machine types in C++ (e.g. `int`):

```
(unary) -, ++, --
(binary) +, -, *, /
(binary with assignment) +=, -=, *=, /=
```

To avoid copying, these operations can also be performed by the following functions:

`void negate (udigit_mod &` a`, const udigit_mod &` b`)`
$a \leftarrow -b$.

`void add (udigit_mod &` c`, const udigit_mod` a`, const udigit_mod` b`)`

`void add (udigit_mod &` c`, const udigit_mod` a`, udigit` b`)`
$c \leftarrow a + b$.

`void subtract (udigit_mod &` c`, const udigit_mod` a`, const udigit_mod` b`)`

`void subtract (udigit_mod &` c`, const udigit_mod` a`, udigit` b`)`

`void subtract (udigit_mod &` c`, udigit` a`, const udigit_mod` b`)`
$c \leftarrow a - b$.

`void multiply (udigit_mod &` c`, const udigit_mod` a`, const udigit_mod` b`)`

`void multiply (udigit_mod &` c`, const udigit_mod` a`, udigit` b`)`
$c \leftarrow a \cdot b$.

`void square (udigit_mod &` c`, const udigit_mod` a`)`
$c \leftarrow a^2$.

`void divide (udigit_mod &` c`, const udigit_mod` a`, const udigit_mod` b`)`

`void divide (udigit_mod &` c`, const udigit_mod` a`, udigit` b`)`

`void divide (udigit_mod & c, udigit a, const udigit_mod b)`
$c \leftarrow a/b$.

`void invert (udigit_mod & a, const udigit_mod & b)`
$a \leftarrow b^{-1}$.

Input/Output

The input and output format of a `udigit_mod` consists only of its mantissa, since the modulus is global. The `istream` operator `>>` and the `ostream` operator `<<` are overloaded. Furthermore, you can use the following member functions below for writing to and reading from a file.

`void a.read (istream & in)`
Reads a mantissa from `in` and stores it into a. The mantissa is reduced modulo m, where m is the global modulus.

`void a.write (ostream & out) const`
Writes the mantissa of a to `out`.

See also

`udigit`

Notes

The example code can be found in `LiDIA/src/simple_classes/udigit_mod/udigit_mod_appl.cc`.

Examples

```
#include <LiDIA/udigit_mod.h>

int main()
{
    udigit_mod::set_modulus(17);

    udigit_mod c, d, e;

    c = 10;
    d = 9;              //or udigit_mod c(10), d(9), e;

    add(e, d, c);
    cout << e <<''\n''; // output should be 2 ((10+9)%17=2)

    divide(e, c, d);
    cout << e <<''\n''; // output should be 3 (inverse of 9 is 2 and (2*10)%17=3)

    subtract(e, c, d);
    cout << e <<''\n''; // output should be 1 ((10-9)%17=1)
```

```
        multiply(e, c, d);
        cout << e <<''\n''; // output should be 5 ((10*9)%17=5)

        return 0;
    }
```

Author

Thorsten Rottschaefer

bigmod

Name

`bigmod` ..... multiprecision modular integer arithmetic

Abstract

`bigmod` is a class for doing multiprecision modular arithmetic over $\mathbb{Z}/m\mathbb{Z}$. It supports for example arithmetic operations, comparisons and exponentiation.

Description

A `bigmod` is a pair of two `bigint`s (man, m), where *man* is called the *mantissa* and m the *modulus*. The modulus is global to all `bigmod`s and must be set before a variable of type `bigmod` can be declared. This is done by the following statement:

```
bigmod::set_modulus(const bigint & m)
```

Each equivalence class modulo m is represented by its least non-negative representative, i.e. the mantissa of a `bigmod` is chosen in the interval $[0, \ldots, |m| - 1]$. For further details, please refer to the description of the functions `set_modulus` and `normalize`.

Constructors/Destructor

`ct bigmod ()`
Initializes with 0.

`ct bigmod (const bigmod & n)`
Initializes with n.

`ct bigmod (const bigint & n)`

`ct bigmod (long n)`

`ct bigmod (unsigned long n)`

`ct bigmod (int n)`
Initializes the mantissa with n modulo `bigmod::modulus()`.

`ct bigmod (double n)`
Initializes the mantissa with $\lfloor n \rfloor$ modulo `bigmod::modulus()`.

`dt ~bigmod ()`

Initialization

`static void set_modulus (const bigint & m)`

sets the global modulus to $|m|$; if $m = 0$, the `lidia_error_handler` will be invoked. If you call the function `set_modulus(m)` after declaration of variables of type `bigmod`, all these variables might be incorrect, i.e. they might have mantissas which are not in the proper range. If you want to use `bigmod` variables after a global modulus change, you have to normalize them. This normalization can be done with the help of the following two functions:

`void a.normalize ()`

normalizes the `bigmod` a such that the mantissa of a is in the range $[0, \ldots, m-1]$, where m is the global modulus.

`void normalize (bigmod & a, const bigmod & b)`

sets the mantissa of a to the mantissa of b and normalizes a such that the mantissa of a is in the range $[0, \ldots, m-1]$, where m is the global modulus.

Assignments

Let a be of type `bigmod` The operator `=` is overloaded. The user may also use the following object methods for assignment:

`void a.assign_zero ()`

mantissa$(a) \leftarrow 0$.

`void a.assign_one ()`

mantissa$(a) \leftarrow 1$.

`void a.assign (int n)`

`void a.assign (long n)`

`void a.assign (unsigned long n)`

mantissa$(a) \leftarrow n$.

`void a.assign (double n)`

mantissa$(a) \leftarrow \lfloor n \rfloor$.

`void a.assign (const bigint & n)`

mantissa$(a) \leftarrow n$.

`void a.assign (const bigmod & n)`

mantissa$(a) \leftarrow$ mantissa(n).

Access Methods

`const bigint & a.mantissa ()`

`bigint & mantissa (const bigmod & a) const`

returns the mantissa of the `bigmod` a. The mantissa is chosen in the interval $[0, \ldots, m-1]$, where m is the global modulus.

`static const bigint & modulus () const`

returns the value of the global modulus.

Object Modifiers

`void a.negate ()`

$a \leftarrow -a$.

`void a.inc ()`

`void inc (bigmod & a)`

$a \leftarrow a + 1$.

`void a.dec ()`

`void dec (bigmod & a)`

$a \leftarrow a - 1$.

`void a.multiply_by_2 ()`

$a \leftarrow 2 \cdot a$ (done by shifting).

`void a.divide_by_2 ()`

computes a number $b \in [0, \ldots, m-1]$ such that $2b \equiv a \pmod{m}$, where m is the global modulus and sets $a \leftarrow b$ (NOTE: this is not equivalent to multiplying a with the inverse of 2). The function invokes the `lidia_error_handler` if the operation is impossible, i.e. if the mantissa of a is odd and m is even.

`bigint a.invert (int verbose = 0)`

sets $a \leftarrow a^{-1}$ if the inverse of a exists. If the inverse does not exist and *verbose* $= 0$, the `lidia_error_handler` will be invoked. If the inverse does not exist and *verbose* $\neq 0$, then the function prints a warning and returns the gcd of a and the global modulus. In this case a remains unchanged and the program does not exit.

`void a.swap (bigmod & b)`

`void swap (bigmod & a, bigmod & b)`

exchanges the mantissa of a and b.

`void a.randomize ()`

computes a random mantissa of a in the range $[0, \ldots, m-1]$, where m is the global modulus by means of the random number generator defined in the kernel.

Arithmetical Operations

The following operators are overloaded and can be used in exactly the same way as in C++:

```
(unary) -, ++, --
(binary) +, -, *, /
(binary with assignment) +=, -=, *=, /=
```

The operators `/` and `/=` denote "multiplication with the inverse". If the inverse does not exist, the `lidia_error_handler` will be invoked. To avoid copying all operators also exist as functions. Let a be of type `bigmod`.

`void negate (bigmod & a, const bigmod & b)`
$a \leftarrow -b$.

`void add (bigmod & c, const bigmod & a, const bigmod & b)`

`void add (bigmod & c, const bigmod & a, const bigint & b)`

`void add (bigmod & c, const bigmod & a, long b)`

`void add (bigmod & c, const bigmod & a, unsigned long b)`

`void add (bigmod & c, const bigmod & a, int b)`
$c \leftarrow a + b$.

`void subtract (bigmod & c, const bigmod & a, const bigmod & b)`

`void subtract (bigmod & c, const bigmod & a, const bigint & b)`

`void subtract (bigmod & c, const bigmod & a, long b)`

`void subtract (bigmod & c, const bigmod & a, unsigned long b)`

`void subtract (bigmod & c, const bigmod & a, int b)`
$c \leftarrow a - b$.

`void multiply (bigmod & c, const bigmod & a, const bigmod & b)`

`void multiply (bigmod & c, const bigmod & a, const bigint & b)`

`void multiply (bigmod & c, const bigmod & a, long b)`

`void multiply (bigmod & c, const bigmod & a, unsigned long b)`

`void multiply (bigmod & c, const bigmod & a, int b)`
$c \leftarrow a \cdot b$.

`void square (bigmod & c, const bigmod & a)`
$c \leftarrow a^2$.

`void divide (bigmod & c, const bigmod & a, const bigmod & b)`

`void divide (bigmod & c, const bigmod & a, const bigint & b)`

`void divide (bigmod & c, const bigmod & a, long b)`

`void divide (bigmod & c, const bigmod & a, unsigned long b)`

`void divide (bigmod & c, const bigmod & a, int b)`
$c \leftarrow a \cdot b^{-1}$; if the inverse of i does not exist, the `lidia_error_handler` will be invoked.

`void invert (bigmod & a, const bigmod & b)`
sets $a \leftarrow b^{-1}$ if the inverse of b exists. Otherwise the `lidia_error_handler` will be invoked.

`bigmod inverse (const bigmod & a)`
returns a^{-1} if the inverse of a exists. Otherwise the `lidia_error_handler` will be invoked.

`void power (bigmod & c, const bigmod & a, const bigint & b)`

`void power (bigmod & c, const bigmod & a, long b)`
$c \leftarrow a^b$ (done with right-to-left binary exponentiation).

Comparisons

`bigmod` supports the binary operators `==`, `!=` and additionally the unary operator `!` (comparison with zero). Let a be an instance of type `bigmod`.

`bool a.is_equal (const bigmod & b) const`
if $b = a$ return `true`, else `false`.

`bool a.is_equal (const bigint & b) const`

`bool a.is_equal (long b) const`
if $b \geq 0$, `true` is returned if b = mantissa(a), `false` otherwise. If $b < 0$, then `true` is returned if $b +$ `bigmod::modulus()` = mantissa(a), `false` otherwise.

`bool a.is_equal (unsigned long b) const`
if $b \geq 0$, `true` is returned if b = mantissa(a), `false` otherwise.

`bool a.is_zero () const`
returns `true` if $a = 0$, `false` otherwise.

`bool a.is_one () const`
returns `true` if $a = 1$, `false` otherwise.

Type Checking and Conversion

Before assigning a `bigmod` (i.e. the mantissa of a `bigmod`) to a machine type (e.g. `int`) it is often useful to perform a test which checks if the assignment can be done without overflow. Let a be an object of type `bigmod`. The following methods return `true` if the assignment would be successful, `false` otherwise.

`bool a.is_char () const`

`bool a.is_uchar () const`

`bool a.is_short () const`

`bool a.is_ushort () const`

`bool a.is_int () const`

`bool a.is_uint () const`

`bool` a`.is_long () const`

`bool` a`.is_ulong () const`

There methods also exists as procedural versions, however, the object methods are preferred over the procedures.

`bool is_char (const bigmod &` a`)`

`bool is_uchar (const bigmod &` a`)`

`bool is_short (const bigmod &` a`)`

`bool is_ushort (const bigmod &` a`)`

`bool is_int (const bigmod &` a`)`

`bool is_uint (const bigmod &` a`)`

`bool is_long (const bigmod &` a`)`

`bool is_ulong (const bigmod &` a`)`

`double dbl (const bigmod &` a`)`
returns the mantissa of a as a double approximation.

`bool` a`.intify (int &` i`) const`
performs the assignment $i \leftarrow \text{mantissa}(a)$ provided the assignment can be done without overflow. In that case the function returns `false`, otherwise it returns `true` and lets the value of i unchanged.

`bool` a`.longify (long &` l`) const`
performs the assignment $l \leftarrow \text{mantissa}(a)$ provided the assignment can be done without overflow. In that case the function returns `false`, otherwise it returns `true` and lets the value of i unchanged.

Basic Methods and Functions

Let a be of type `bigmod`.

`lidia_size_t` a`.bit_length () const`
returns the bit length of a's mantissa (see `bigint`, page 19).

`lidia_size_t` a`.length () const`
returns the word length of a's mantissa (see `bigint`, page 19).

High-Level Methods and Functions

`bigmod randomize (const bigmod &` a`)`
returns a random `bigmod` b with random mantissa in the range $[0, \ldots, \text{mantissa}(a) - 1]$.

Input/Output

Let a be of type `bigmod`.

The `istream` operator `>>` and the `ostream` operator `<<` are overloaded. Furthermore, you can use the following member functions for writing to and reading from a file in binary or ASCII format, respectively. The input and output format of a `bigmod` consists only of its mantissa, since the modulus is global. Note that you have to manage by yourself that successive `bigmod`s have to be separated by blanks.

`int` x`.read_from_file (FILE *fp)`
reads x from the binary file `fp` using `fread`.

`int` x`.write_to_file (FILE *fp)`
writes x to the binary file `fp` using `fwrite`.

`int` x`.scan_from_file (FILE *fp)`
scans x from the ASCII file `fp` using `fscanf`.

`int` x`.print_to_file (FILE *fp)`
prints x to the ASCII file `fp` using `fprintf`.

`int string_to_bigmod (char *s, bigmod &` a`)`
converts the string `s` to a `bigmod` a and returns the number of converted characters.

`int bigmod_to_string (const bigmod &` a`, char *s)`
converts the `bigmod` a to a string `s` and returns the number of used characters.

See also

`bigint`, `multi_bigmod`

Notes

To use multiple moduli at the same time please use the class `multi_bigmod`.

Examples

```
#include <LiDIA/bigmod.h>

int main()
{
    bigmod a, b, c;

    bigmod::set_modulus(10);        // Global for a, b and c
    cout << "Please enter a : "; cin >> a ;
    cout << "Please enter b : "; cin >> b ;
    cout << endl;
    c = a + b;
    cout << "a + b = " << c << endl;
}
```

For further reference please refer to `LiDIA/src/simple_classes/bigmod_appl.cc`

Author

Markus Maurer, Thomas Papanikolaou

multi_bigmod

Name

`multi_bigmod` ..... multiprecision modular integer arithmetic

Abstract

`multi_bigmod` is a class for doing multiprecision modular arithmetic over $\mathbb{Z}/m\mathbb{Z}$. It supports for example arithmetic operations, comparisons and exponentiation.

Description

A `multi_bigmod` is a pair of two `bigint`s (man, m), where man is called *mantissa* and m *modulus*. Each `multi_bigmod` has got its own modulus which can be set by a constructor or by

a`.set_modulus(const bigint &` m`)`

where a is of type `multi_bigmod`.

Each equivalence class modulo m is represented by its least non-negative representative, i.e. the mantissa of a `multi_bigmod` is chosen in the interval $[0, \ldots, |m| - 1]$.

Constructors/Destructor

If $m = 0$, the `lidia_error_handler` will be invoked.

`ct multi_bigmod ()`
initializes the mantissa and the modulus with 0.

`ct multi_bigmod (const multi_bigmod &` n`)`
initializes with n.

`ct multi_bigmod (const bigint &` n`, const bigint &` m`)`

`ct multi_bigmod (long` n`, const bigint &` m`)`

`ct multi_bigmod (unsigned long` n`, const bigint &` m`)`

`ct multi_bigmod (int` n`, const bigint &` m`)`
initializes the mantissa with n and the modulus with $|m|$.

`ct multi_bigmod (double` n`, const bigint &` m`)`
initializes the mantissa with $\lfloor d \rfloor$ and the modulus with $|m|$.

`dt ~multi_bigmod ()`

Initialization

`void` a`.set_modulus (const bigint &` m`)`
sets the modulus of the `multi_bigmod` a to $|m|$; if $m = 0$, the `lidia_error_handler` will be invoked. The mantissa of a will not automatically be reduced modulo m. You have to call one of the two following normalization functions to reduce the mantissa.

`void` a`.normalize ()`
normalizes the `multi_bigmod` a such that the mantissa of a is in the range $[0, \ldots, m-1]$, where m is the modulus of a.

`void normalize (multi_bigmod &` a`, const multi_bigmod &` b`)`
sets the modulus of a to the modulus of b, the mantissa of a to the mantissa of b and normalizes a such that the mantissa of a is in the range $[0, \ldots, m-1]$, where m is the modulus of b.

Assignments

Let a be of type `multi_bigmod` The operator `=` is overloaded. The user may also use the following object methods for assignment:

`void` a`.assign_zero ()`
mantissa(a) $\leftarrow 0$.

`void` a`.assign_one ()`
mantissa(a) $\leftarrow 1$.

`void` a`.assign_zero (const bigint &` m`)`
mantissa(a) $\leftarrow 0$, modulus(a) $\leftarrow |m|$. If $m = 0$, the `lidia_error_handler` will be invoked.

`void` a`.assign_one (const bigint &` m`)`
mantissa(a) $\leftarrow 1$, modulus(a) $\leftarrow |m|$. If $m = 0$, the `lidia_error_handler` will be invoked.

`void` a`.assign (int` i`, const bigint &` m`)`
mantissa(a) $\leftarrow i$, modulus(a) $\leftarrow |m|$. If $m = 0$, the `lidia_error_handler` will be invoked.

`void` a`.assign (long` i`, const bigint &` m`)`
mantissa(a) $\leftarrow i$, modulus(a) $\leftarrow |m|$. If $m = 0$, the `lidia_error_handler` will be invoked.

`void` a`.assign (unsigned long` i`, const bigint &` m`)`
mantissa(a) $\leftarrow i$, modulus(a) $\leftarrow |m|$. If $m = 0$, the `lidia_error_handler` will be invoked.

`void` a`.assign (const bigint &` b`, const bigint &` m`)`
mantissa(a) $\leftarrow b$, modulus(a) $\leftarrow |m|$. If $m = 0$, the `lidia_error_handler` will be invoked.

`void` a`.assign (const multi_bigmod &` b`)`

$a \leftarrow b$.

`void` a`.set_mantissa (int` i`)`

mantissa(a) $\leftarrow i$ and the mantissa of a will be reduced modulo the modulus of a.

`void` a`.set_mantissa (long` i`)`

mantissa(a) $\leftarrow i$ and the mantissa of a will be reduced modulo the modulus of a.

`void` a`.set_mantissa (unsigned long` i`)`

mantissa(a) $\leftarrow i$ and the mantissa of a will be reduced modulo the modulus of a.

`void` a`.set_mantissa (const bigint &` b`)`

mantissa(a) $\leftarrow b$ and the mantissa of a will be reduced modulo the modulus of a.

`double dbl (const multi_bigmod &` a`)`

returns the mantissa of a as a double approximation.

`bool` a`.intify (int &` i`) const`

performs the assignment $i \leftarrow$ mantissa(a) provided the assignment can be done without overflow. In that case the function returns `false`, otherwise it returns `true` and the value of i will be unchanged.

`bool` a`.longify (long &` i`) const`

performs the assignment $i \leftarrow$ mantissa(a) provided the assignment can be done without overflow. In that case the function returns `false`, otherwise it returns `true` and the value of i will be unchanged.

Access Methods

`bigint mantissa (const multi_bigmod &` a`)`

returns the mantissa of the `multi_bigmod` a. The mantissa is chosen in the interval $[0, \ldots, m-1]$, where m is the modulus of a.

`const bigint &` a`.mantissa () const`

returns the mantissa of the `multi_bigmod` a. The mantissa is chosen in the interval $[0, \ldots, m-1]$, where m is the modulus of a.

`const bigint &` a`.modulus () const`

returns the modulus of the `multi_bigmod` a.

Object Modifiers

`void` a`.negate ()`

$a \leftarrow -a$.

`void` a`.inc ()`

`void inc (multi_bigmod &` a`)`

$a \leftarrow a + 1$.

`void` *a*`.dec ()`

`void dec (multi_bigmod &` *a*`)`

$a \leftarrow a - 1$.

`bigint` *a*`.invert (int` *verbose* `= 0)`

sets $a \leftarrow a^{-1}$, if the inverse of a exists. If the inverse does not exist and *verbose* $= 0$, the `lidia_error_handler` will be invoked. If the inverse does not exist and *verbose* $\neq 0$, then the function prints a warning and returns the gcd of a and its modulus. In this case a remains unchanged and the program does not exit.

`void` *a*`.multiply_by_2 ()`

$a \leftarrow 2 \cdot a$ (using shift-operations).

`void` *a*`.divide_by_2 ()`

computes a number $b \in [0, \ldots, m-1]$ such that $2b \equiv \text{mantissa}(a) \bmod m$, where m is the modulus of a, and sets $\text{mantissa}(a) \leftarrow b$ (note: this is not equivalent to the multiplication of a with the inverse of 2). The `lidia_error_handler` will be invoked if the operation is impossible, i.e. if the mantissa of a is odd and m is even.

`void` *a*`.swap (multi_bigmod &` *b*`)`

`void swap (multi_bigmod &` *a*`, multi_bigmod &` *b*`)`

exchanges the mantissa and the modulus of a and b.

Arithmetical Operations

The binary operations require that the operands have the same modulus; in case of different moduli, the `lidia_error_handler` will be invoked. Binary operations that obtain an operand $a \in \mathbb{Z}/m\mathbb{Z}$, where m is the modulus of a, and an operand $b \in \mathbb{Z}$, i.e. b of type `int`, `long`, `bigint`, map b to $\mathbb{Z}/m\mathbb{Z}$.

The following operators are overloaded and can be used in exactly the same way as in C++:

```
(unary) -, ++, --
(binary) +, -, *, /
(binary with assignment) +=, -=, *=, /=
```

The operators `/` and `/=` denote "multiplication with the inverse". If the inverse does not exist, the `lidia_error_handler` will be invoked. To avoid copying all operators also exist as functions.

`void negate (multi_bigmod &` *a*`, const multi_bigmod &` *b*`)`

$a \leftarrow -b$.

`void add (multi_bigmod &` *c*`, const multi_bigmod &` *a*`, const multi_bigmod &` *b*`)`

`void add (multi_bigmod &` *c*`, const multi_bigmod &` *a*`, const bigmod &` *b*`)`

`void add (multi_bigmod &` *c*`, const multi_bigmod &` *a*`, const bigint &` *b*`)`

`void add (multi_bigmod &` *c*`, const multi_bigmod &` *a*`, long` *b*`)`

`void add (multi_bigmod &` *c*`, const multi_bigmod &` *a*`, unsigned long` *b*`)`

`void add (multi_bigmod & c, const multi_bigmod & a, int b)`
$c \leftarrow a + b$.

`void subtract (multi_bigmod & c, const multi_bigmod & a, const multi_bigmod & b)`

`void subtract (multi_bigmod & c, const multi_bigmod & a, const bigmod & b)`

`void subtract (multi_bigmod & c, const multi_bigmod & a, const bigint & b)`

`void subtract (multi_bigmod & c, const multi_bigmod & a, long b)`

`void subtract (multi_bigmod & c, const multi_bigmod & a, unsigned long b)`

`void subtract (multi_bigmod & c, const multi_bigmod & a, int b)`
$c \leftarrow a - b$.

`void multiply (multi_bigmod & c, const multi_bigmod & a, const multi_bigmod & b)`

`void multiply (multi_bigmod & c, const multi_bigmod & a, const bigmod & b)`

`void multiply (multi_bigmod & c, const multi_bigmod & a, const bigint & b)`

`void multiply (multi_bigmod & c, const multi_bigmod & a, long b)`

`void multiply (multi_bigmod & c, const multi_bigmod & a, unsigned long b)`

`void multiply (multi_bigmod & c, const multi_bigmod & a, int b)`
$c \leftarrow a \cdot b$.

`void square (multi_bigmod & c, const multi_bigmod & a)`
$c \leftarrow a^2$.

`void divide (multi_bigmod & c, const multi_bigmod & a, const multi_bigmod & b)`

`void divide (multi_bigmod & c, const multi_bigmod & a, const bigmod & b)`

`void divide (multi_bigmod & c, const multi_bigmod & a, const bigint & b)`

`void divide (multi_bigmod & c, const multi_bigmod & a, long b)`

`void divide (multi_bigmod & c, const multi_bigmod & a, unsigned long b)`

`void divide (multi_bigmod & c, const multi_bigmod & a, int b)`
$c \leftarrow a \cdot i^{-1}$, if the inverse of i does not exist, the `lidia_error_handler` will be invoked.

`void invert (multi_bigmod & a, const multi_bigmod & b)`
sets $a \leftarrow b^{-1}$, if the inverse of b exists. Otherwise the `lidia_error_handler` will be invoked.

`multi_bigmod inverse (const multi_bigmod & a)`
returns a^{-1}, if the inverse of a exists. Otherwise the `lidia_error_handler` will be invoked.

`void power (multi_bigmod & c, const multi_bigmod & a, const bigint & b)`

`void power (multi_bigmod & c, const multi_bigmod & a, long b)`
$c \leftarrow a^b$ (done with right-to-left binary exponentiation).

Comparisons

`multi_bigmod` supports the binary operators `==`, `!=` and additionally the unary operator `!` (comparison with zero). Let a be an instance of type `multi_bigmod`.

`bool` a`.is_equal (const multi_bigmod &` b`) const`

`bool` a`.is_equal (const bigmod &` b`) const`
if $b = a$ return `true`, else `false`.

`bool` a`.is_equal (const bigint &` b`) const`

`bool` a`.is_equal (long` b`) const`
if $b \geq 0$, `true` is returned if b = mantissa(a), `false` otherwise. If $b < 0$, then `true` is returned if $b +$ `bigmod::modulus()` = mantissa(a), `false` otherwise.

`bool` a`.is_equal (unsigned long` b`) const`
if $b \geq 0$, `true` is returned if b = mantissa(a), `false` otherwise.

`bool` a`.is_zero () const`
returns `true` if mantissa(a) = 0, `false` otherwise.

`bool` a`.is_one () const`
returns `true` if mantissa(a) = 1, `false` otherwise.

Type Checking and Conversion

Before assigning a `multi_bigmod` (i.e. the mantissa of a `multi_bigmod`) to a machine type (e.g. `int`) it is often useful to perform a test which checks whether the assignment could be done without overflow. Let a be an object of type `multi_bigmod`. The following methods return `true` if the assignment would be successful, `false` otherwise.

`bool` a`.is_char () const`

`bool` a`.is_uchar () const`

`bool` a`.is_short () const`

`bool` a`.is_ushort () const`

`bool` a`.is_int () const`

`bool` a`.is_uint () const`

`bool` a`.is_long () const`

`bool` a`.is_ulong () const`

There methods also exists as procedural versions, however, the object methods are preferred over the procedures.

`bool is_char (const multi_bigmod &` a`)`

`bool is_uchar (const multi_bigmod &` a`)`

`bool is_short (const multi_bigmod & a)`

`bool is_ushort (const multi_bigmod & a)`

`bool is_int (const multi_bigmod & a)`

`bool is_uint (const multi_bigmod & a)`

`bool is_long (const multi_bigmod & a)`

`bool is_ulong (const multi_bigmod & a)`

Basic Methods and Functions

Let a be of type `multi_bigmod`.

`lidia_size_t a.bit_length () const`
returns the bit-length of a's mantissa (see class `bigint`, page 19).

`lidia_size_t a.length () const`
returns the word-length of a's mantissa (see class `bigint`, page 19).

High-Level Methods and Functions

`multi_bigmod randomize (const multi_bigmod & a)`
returns a random `multi_bigmod` b with random mantissa in the range $[0, \ldots, \text{mantissa}(b) - 1]$, $\text{modulus}(b) = \text{modulus}(a)$.

`void a.randomize (const bigint & m)`
computes a random mantissa of a in the range $[0, \ldots, |m| - 1]$, $\text{modulus}(a) = |m|$.

Input/Output

Let a be of type `multi_bigmod`.

The `istream` operator `>>` and the `ostream` operator `<<` are overloaded. Furthermore, you can use the following member functions for writing to and reading from a file in binary or ASCII format, respectively. The input and output format of a `multi_bigmod` are

`(mantissa, modulus)`.

`int x.read_from_file (FILE *fp)`
reads x from the binary file `fp` using `fread`.

`int x.write_to_file (FILE *fp)`
writes x to the binary file `fp` using `fwrite`.

`int x.scan_from_file (FILE *fp)`
scans x from the ASCII file `fp` using `fscanf`.

`int` x`.print_to_file (FILE *fp)`
prints x to the ASCII file `fp` using `fprintf`.

`int string_to_multi_bigmod (char *s, multi_bigmod &` a`)`
converts the string s to a `multi_bigmod` a and returns the number of converted characters.

`int multi_bigmod_to_string (const multi_bigmod &` a`, char *s)`
converts the `multi_bigmod` a to a string s and returns the number of used characters.

See also

`bigint`, `bigmod`

Examples

```
#include <LiDIA/multi_bigmod.h>

int main()
{
    multi_bigmod a, b, c;

    a.set_modulus(10);
    b.set_modulus(10);
    cout << "Please enter a : "; cin >> a ;
    cout << "Please enter b : "; cin >> b ;
    cout << endl;
    c = a + b;
    cout << "a + b = " << c << endl;
}
```

For further reference please refer to `LiDIA/src/simple_classes/multi_bigmod_appl.cc`

Author

Markus Maurer, Thomas Papanikolaou

bigrational

Name

`bigrational` ..... multiprecision rational arithmetic

Abstract

`bigrational` is a class for doing multiprecision rational arithmetic. It supports for example arithmetic operations, comparisons, and exponentiation.

Description

A `bigrational` consists of a pair of two `bigint`s (num, den), the numerator num and the denominator den. The `bigrational` (num, den) represents the rational number $\frac{num}{den}$. The numerator num and the denominator den of a `bigrational` are always coprime; the denominator den is positive. Arithmetic is done using the algorithms described in [38].

Constructors/Destructor

`ct bigrational ()`
$num \leftarrow 0$, $den \leftarrow 1$.

`ct bigrational (const bigrational & b)`
$num \leftarrow b$`.numerator()`, $den \leftarrow b$`.denominator()`.

`ct bigrational (const bigint & n, const bigint & d)`
$num \leftarrow n$, $den \leftarrow d$.

`ct bigrational (const bigint & n)`

`ct bigrational (long n)`

`ct bigrational (unsigned long n)`

`ct bigrational (int n)`
$num \leftarrow n$, $den \leftarrow 1$.

`ct bigrational (double d)`
$num \leftarrow$ `bigint(`d`)`, $den \leftarrow 1$.

`dt ~bigrational ()`

Assignments

Let a be of type `bigrational` The operator `=` is overloaded. The user may also use the following object methods for assignment:

`void` a`.assign_zero ()`
$a \leftarrow 0$.

`void` a`.assign_one ()`
$a \leftarrow 1$.

`void` a`.assign (const bigrational &` b`)`
$a \leftarrow b$.

`void` a`.assign (const bigint &` n`, const bigint &` d`)`
$a \leftarrow \frac{n}{d}$.

`void` a`.assign (const bigint &` n`)`

`void` a`.assign (long` n`)`

`void` a`.assign (unsigned long` n`)`

`void` a`.assign (int` n`)`
$a \leftarrow n$.

`void` a`.assign (double` n`)`
$a \leftarrow \lfloor n \rfloor$.

Access Methods

`const bigint &` a`.numerator () const`

`const bigint & numerator (const bigrational &` a`)`
returns the numerator of a.

`const bigint &` a`.denominator () const`

`const bigint & denominator (const bigrational &` a`)`
returns the denominator of a.

Object Modifiers

`void` a`.negate ()`
$a \leftarrow -a$.

`void` a`.inc ()`

`void inc (bigrational &` a`)`
$a \leftarrow a + 1$.

`void` a`.dec ()`

`void dec (bigrational &` a`)`
$a \leftarrow a - 1$.

`void` a`.multiply_by_denominator ()`
Multiplies a by its denominator.

`void` a`.multiply_by_2 ()`
$a \leftarrow 2 \cdot a$ (done by shifting).

`void` a`.divide_by_2 ()`
$a \leftarrow a/2$ (done by shifting).

`void` a`.invert ()`
$a \leftarrow 1/a$, if $a \neq 0$. Otherwise the `lidia_error_handler` will be invoked.

Arithmetical Operations

The following operators are overloaded and can be used in exactly the same way as in C++:

```
                    (unary) -, ++, --
                (binary) +, -, *, /, >>, <<
(binary with assignment) +=, -=, *=, /=, >>=, <<=
```

To avoid copying all operators also exist as functions.

`void negate (bigrational &` a`, const bigrational &` b`)`
$a \leftarrow -b$.

`void add (bigrational &` c`, const bigrational &` a`, const bigrational &` b`)`

`void add (bigrational &` c`, const bigrational &` a`, const bigint &` b`)`

`void add (bigrational &` c`, const bigrational &` a`, long` b`)`

`void add (bigrational &` c`, const bigrational &` a`, unsigned long` b`)`

`void add (bigrational &` c`, const bigrational &` a`, int` b`)`
$c \leftarrow a + b$.

`void subtract (bigrational &` c`, const bigrational &` a`, const bigrational &` b`)`

`void subtract (bigrational &` c`, const bigrational &` a`, const bigint &` b`)`

`void subtract (bigrational &` c`, const bigrational &` a`, long` b`)`

`void subtract (bigrational &` c`, const bigrational &` a`, unsigned long` b`)`

`void subtract (bigrational &` c`, const bigrational &` a`, int` b`)`
$c \leftarrow a - b$.

`void multiply (bigrational & c, const bigrational & a, const bigrational & b)`

`void multiply (bigrational & c, const bigrational & a, const bigint & b)`

`void multiply (bigrational & c, const bigrational & a, long b)`

`void multiply (bigrational & c, const bigrational & a, unsigned long b)`

`void multiply (bigrational & c, const bigrational & a, int b)`
$c \leftarrow a \cdot i$.

`void square (bigrational & c, const bigrational & a)`
$c \leftarrow a^2$.

`void divide (bigrational & c, const bigrational & a, const bigrational & b)`

`void divide (bigrational & c, const bigrational & a, const bigint & b)`

`void divide (bigrational & c, const bigrational & a, long b)`

`void divide (bigrational & c, const bigrational & a, unsigned long b)`

`void divide (bigrational & c, const bigrational & a, int b)`
$c \leftarrow a/i$, if $i \neq 0$. Otherwise the `lidia_error_handler` will be invoked.

`void invert (bigrational & a, const bigrational & b)`
$a \leftarrow 1/b$, if $b \neq 0$. Otherwise the `lidia_error_handler` will be invoked.

`bigrational inverse (const bigrational & a)`
returns $1/a$, if $a \neq 0$. Otherwise the `lidia_error_handler` will be invoked.

`void power (bigrational & c, const bigrational & a, const bigint & i)`

`void power (bigrational & c, const bigrational & a, long i)`
$c \leftarrow a^i$.

Shift Operations

`void shift_left (bigrational & c, const bigrational & a, long i)`
$c \leftarrow a \cdot 2^i$.

`void shift_right (bigrational & c, const bigrational & a, long i)`
$c \leftarrow a/2^i$.

Comparisons

The binary operators `==`, `!=`, `>=`, `<=`, `>`, `<` and the unary operator `!` (comparison with zero) are overloaded and can be used in exactly the same way as in C++. Let a be of type `bigrational`.

`bool a.is_positive () const`
returns `true` if $a > 0$, `false` otherwise.

`bool` a`.is_negative () const`
returns `true` if $a < 0$, `false` otherwise.

`bool` a`.is_zero () const`
returns `true` if $a = 0$, `false` otherwise.

`bool` a`.is_ge_zero () const`
returns `true` if $a \geq 0$, `false` otherwise.

`bool` a`.is_gt_zero () const`
returns `true` if $a > 0$, `false` otherwise.

`bool` a`.is_le_zero () const`
returns `true` if $a \leq 0$, `false` otherwise.

`bool` a`.is_lt_zero () const`
returns `true` if $a < 0$, `false` otherwise.

`bool` a`.is_one () const`
returns `true` if $a = 1$, `false` otherwise.

`int` a`.compare (const bigrational &` b`) const`

`int` a`.compare (const bigint &` b`) const`

`int` a`.compare (long` b`) const`

`int` a`.compare (unsigned long` b`) const`

`int` a`.compare (int` b`) const`
returns $\operatorname{sgn}(a - b)$.

`int` a`.abs_compare (const bigrational &` b`) const`

`int` a`.abs_compare (const bigint &` b`) const`

`int` a`.abs_compare (unsigned long` b`) const`
returns $\operatorname{sgn}(|a| - |b|)$.

`int` a`.sign () const`
returns -1, 0, 1 if $a <, =, > 0$ respectively.

Type Checking and Conversion

Before assigning a `bigrational` to a `bigint` it is often useful to check whether precision is lost, i.e. if the denominator is not equal to one.

`bool is_bigint (const bigrational &` a`) const`
returns `true` if the denominator of a is one, `false`, otherwise.

`double dbl (const bigrational & a) const`

returns the value of a as a double approximation.

`bool a.intify (int & i) const`

performs the assignment $i \leftarrow \lfloor a$`.numerator()/`a`.denominator()`$\rfloor$ provided that this assignment can be done without overflow. In that case the function returns `false`, otherwise it returns `true` and lets the value of i unchanged.

`bool a.longify (long & i) const`

performs the assignment $i \leftarrow \lfloor a$`.numerator()/`a`.denominator()`$\rfloor$ provided that this assignment can be done without overflow. In that case the function returns `false`, otherwise it returns `true` and lets the value of i unchanged.

Basic Methods and Functions

`bigrational abs (const bigrational & a)`

returns $|a|$.

`bigint ceiling (const bigrational & a)`

returns the least integer b with $b \geq a$.

`bigint floor (const bigrational & a)`

returns the greatest integer b with $b \leq a$.

`bigint round (const bigrational & a)`

returns the nearest integer to a.

`bigint truncate (const bigrational & a)`

returns the integer part b of a, i.e. $b \leftarrow \frac{\lfloor a \rfloor + \lceil a \rceil - 1}{2}$.

`bool square_root (bigrational &` *root*`, const bigrational & a)`

returns `true`, if a is square. In this case *root* is a square root of a. Otherwise it returns `false`, but the value of *root* may be changed.

`bool cube_root (bigrational &` *root*`, const bigrational & a)`

returns `true`, if a is a cube. In this case *root* is a cubic root of a. Otherwise it returns `false`, but the value of *root* may be changed.

Input/Output

Let a be of type `bigrational`. `istream` operator `>>` and `ostream` operator `<<` are overloaded. Furthermore, you can use the following member functions for writing to and reading from a file in binary or ASCII format, respectively. Input and output of a `bigrational` have the following format:

num/den

(without blanks). Note that you have to manage by yourself that successive `bigrational`s have to be separated by blanks.

`int string_to_bigrational (char *s, const char *t, bigrational & a)`
converts the strings s and t to `bigints` s', t' and sets $a \leftarrow s'/t'$, returns the total number of converted characters.

`int bigrational_to_string (const bigrational & a, char *s, char *t)`
converts the denominator of a to string s, the numerator of a to string t and returns the number of used characters.

`int a.read_from_file (FILE *fp)`
reads a from `fp` using `fread`.

`int a.write_to_file (FILE *fp)`
writes a to `fp` using `fwrite`.

`int a.scan_from_file (FILE *fp)`
scans a from `fp` using `fscanf`.

`int a.print_to_file (FILE *fp)`
prints a to `fp` using `fprintf`.

See also

`bigint`

Examples

```
#include <LiDIA/bigrational.h>

int main()
{
    bigrational a, b, c;

    cout << "Please enter a : "; cin >> a ;
    cout << "Please enter b : "; cin >> b ;
    cout << endl;
    c = a + b;
    cout << "a + b = " << c << endl;
}
```

For further reference please refer to `LiDIA/src/simple_classes/bigrational_appl.cc`

Author

Volker Müller, Thomas Papanikolaou

xdouble

Name

`xdouble` .....double+double precision arithmetic

Abstract

`xdouble` is a class for (`double`+`double`) arithmetic over $\mathbb{R}$. It supports mainly the same functions as the built-in type double according to IEEE (see [33]). Several conversion routines have been added.

Description

An `xdouble` x is represented by two `doubles` lo, hi, where $|lo| < |hi|$. The addition of lo and hi will give you the value of x

$$x = lo + hi$$

A complete description of the representation, the algorithms for the basic operations (`+, -, *, /`) and the more sophisticated algorithms of `xdouble` can be found in the papers of T.J. Decker, S. Linnainmaa and D. Priest (see [21], [42], [53]).

Constructors/Destructor

`ct xdouble ()`
initializes with 0.

`ct xdouble (double` hi`, double` lo `= 0.0)`
initializes with $hi + lo$.

`ct xdouble (const xdouble &)`
initializes with a `xdouble`.

`ct xdouble (char *)`
string conversion.

`dt ~xdouble ()`
deletes the `xdouble` object.

Assignments

Let a be of type `xdouble`. The operator `=` is overloaded for `xdouble` and any other built-in type. The follwing routines convert an `xdouble` into a built-in type and return the converted element. No overflow check is done.

`double double_xdbl (const xdouble &)`
convert an `xdouble` to a `double`.

`unsigned long ulong_xdbl (const xdouble &)`
convert an `xdouble` to an `unsigned long`.

`long long_xdbl (const xdouble &)`
convert an `xdouble` to a `long`.

`unsigned int uint_xdbl (const xdouble &)`
convert an `xdouble` to an `unsigned int`.

`int int_xdbl (const xdouble &)`
convert an `xdouble` to an `int`.

`unsigned short ushort_xdbl (const xdouble &)`
convert an `xdouble` to an `unsigned short`.

`short short_xdbl (const xdouble &)`
convert an `xdouble` to a `short`.

`unsigned char uchar_xdbl (const xdouble &)`
convert an `xdouble` to an `unsigned char`.

`char char_xdbl (const xdouble &)`
convert an `xdouble` to a `char`.

Access Methods

Let a be of type `xdouble`.

`double` a`.h ()`
returns the *hi* part of a.

`double` a`.l ()`
returns the *lo* part of a.

`xdouble ldexp (const xdouble &` a`, const int` *exp*`)`
returns the result of multiplying the a by 2 raised to the power *exp*.

`xdouble modf (const xdouble &` a`, xdouble *` *id*`)`
breaks the argument a into an integral part and a fractional part, each of which has the same sign as a. The integral part is stored in *id*. Returns the fractional part of a.

Arithmetical Operations

The following operators are overloaded and can be used in exactly the same way as in C++:

```
                    (unary) -
                (binary) +, -, *, /
        (binary with assignment) +=, -=, *=, /=
```

`xdouble sqr (const xdouble & a)`
returns a^2.

`xdouble cub (const xdouble & a)`
returns a^3.

`xdouble sqrt (const xdouble & a)`
returns $\sqrt{a}$, where $a \geq 0$. If $a < 0$, the `lidia_error_handler` will be invoked.

`xdouble pow (const xdouble & a, const xdouble & b)`
returns a^b, where $b > 0$. If $b \leq 0$, the `lidia_error_handler` will be invoked.

`xdouble powint (const xdouble & a, long b)`
returns a^b, where $b > 0$. If $b \leq 0$, the `lidia_error_handler` will be invoked.

`xdouble fmod (const xdouble & a, const int n)`
returns the remainder of dividing a by n.

Comparisons

Let a be of type `xdouble`. The binary operators `==`, `!=`, `>=`, `<=`, `>`, `<` and the unary operator `!` (comparison with zero) are overloaded and can be used in exactly the same way as in C++.

Basic Methods and Functions

Let a be of type `xdouble`.

`int sign (const xdouble & a)`
returns 1 if $a > 0$, 0 if $a = 0$ and 1 otherwise.

`xdouble fabs (const xdouble & a)`
returns $|a|$.

`xdouble ceil (const xdouble & a)`
returns $\lceil a \rceil$, i.e. c is the smallest integer which is greater than or equal to a.

`xdouble floor (const xdouble & a)`
returns $\lfloor a \rfloor$, i.e. c is the largest integer which is less than or equal to a.

`xdouble rint (const xdouble & a)`
returns $\lfloor a + 1/2 \rfloor$, i.e. c is the nearest integer to a.

`xdouble trunc (const xdouble & a)`
returns $\mathrm{sgn}(a) \cdot \lfloor |a| \rfloor$.

`xdouble exp (const xdouble & a)`
returns e^a.

`xdouble log (const xdouble & a)`
returns $\log(a)$ (natural logarithm to base $e \approx 2.71828\ldots$), where $a > 0$. If $a \leq 0$ the `lidia_error_handler` will be invoked.

`xdouble log10 (const xdouble & a)`
returns $\log(a)$ (logarithm to base 10), where $a > 0$. If $a \leq 0$ the `lidia_error_handler` will be invoked.

(Inverse) Trigonometric Functions

Note that the arguments of the trigonometrical functions are supposed to be given as radiants.

`xdouble sin (const xdouble & a)`
returns $\sin(a)$.

`xdouble cos (const xdouble & a)`
returns $\cos(a)$.

`void sin (const xdouble & a, xdouble & sinx, xdouble & cosx)`
$sinx = \sin(a)$ and $cosx = \cos(a)$. Faster than two explicit calls.

`xdouble tan (const xdouble & a)`
returns $\tan(a)$ if $a \neq (2k+1)\,\pi/2$, $k \in \mathbb{Z}$. Otherwise the `lidia_error_handler` will be invoked.

`xdouble asin (const xdouble & a)`
returns $\arcsin(a)$ if $a \in [-1, 1]$. Otherwise the `lidia_error_handler` will be invoked.

`xdouble acos (const xdouble & a)`
returns $\arccos(a)$ if $a \in [-1, 1]$. Otherwise the `lidia_error_handler` will be invoked.

`xdouble atan (const xdouble & a)`
returns $\arctan(a)$.

`xdouble atan2 (const xdouble & a, const xdouble & b)`
returns $\arctan(a/b)$ in the range of $[-\pi, \pi[$, i.e. `atan2(a, b)` calculates the argument (respectively phase) of a, b.

(Inverse) Hyperbolic Trigonometric Functions

`xdouble sinh (const xdouble & a)`
returns $\sinh(a)$.

`xdouble cosh (const xdouble & a)`
returns $\cosh(a)$.

Constants

Useful Constants.

`xdouble Log2 ()`
$\ln 2 \approx 0.69314718055994530941723212145817656807 55$

`xdouble Log10 ()`
$\ln 10 \approx 2.30258509299404568401799145468436420760 1$

`xdouble Pi ()`
$\pi \approx 3.14159265358979323846264338327950288419 72$

`xdouble TwoPi ()`
$2\pi \approx 6.28318530717958647692528676655900576839 43$

`xdouble Pion2 ()`
$\frac{\pi}{2} \approx 1.57079632679489661923132169163975144209 85$

`xdouble Pion4 ()`
$\frac{\pi}{4} \approx 0.78539816339744830961566084581987572104 93$

Input/Output

Let a be of type `xdouble`.

`istream` operator `>>` and `ostream` operator `<<` are overloaded.

`void string_to_xdouble (char * s, xdouble & a)`
converts the string `s` to an `xdouble` a.

`void xdouble_to_string (const xdouble & a, char * s)`
converts the `xdouble` a to a string `s`. The number of characters allocated for s must be at least 42 characters large to carry the digits of a.

See also

`bigfloat`

Notes

Optimization may cause problems with an FPU having more than 64 bits.

Examples

The following program calculates an approximate to the number $\sqrt{\pi}$.

```
#include <LiDIA/xdouble.h>

int main()
{
    xdouble x = sqrt(xdouble::Pi);
    cout << "\n sqrt(pi) = " << x << flush;
    return 0;
}
```

Author

Original code by Keith Briggs. Ported to LiDIA by Werner Backes and Thomas Papanikolaou

bigfloat

Name

`bigfloat` ..... multiprecision floating point arithmetic

Abstract

`bigfloat` is a class for doing multiprecision arithmetic over $\mathbb{R}$. It supports for example arithmetic operations, shift operations, comparisons, the square root function, the exponential function, the logarithmical function, (inverse) trigonometric functions etc.

Description

Before declaring a `bigfloat` it is possible to set the decimal precision p by

```
bigfloat::set_precision(p)
```

Then the *base digit* precision (see class `bigint`, page 19) is

$$t = \left\lceil \frac{p \cdot \log(10)}{\log(base)} \right\rceil + 3 \ ,$$

where *base* is the value of the *bigint base*. If the precision has not been set, a default precision of $t = 5$ `bigint` base digits is used. If the decimal precision is set to p then a `bigfloat` is a pair (m, e), where m is a `bigint` with $|m| < base^p$ and e is a machine integer. It represents the number

$$0.m \cdot 2^e \ .$$

The number m is called the *mantissa* and the number e is called the *exponent*.

A complete description of the floating point representation and the algorithms for the basic operations (`+, -, *, /`) of `bigfloat` can be found in the book of D. Knuth (see [38]). The more sophisticated algorithms which are used in `bigfloat` are implemented according to R.P. Brent (see [8], [10], [9]) and J.M. and P.B. Borwein (see [7]). There one can also find the time- and error-bound analysis of these algorithms.

Constructors/Destructor

`ct bigfloat ()`
initializes with 0.

`ct bigfloat (const bigfloat & n)`

`ct bigfloat (const bigint & n)`

`ct bigfloat (long n)`

`ct bigfloat (unsigned long n)`

`ct bigfloat (int n)`

`ct bigfloat (double n)`
initializes with n.

`ct bigfloat (const bigint & n, const bigint & d)`
This constructor initializes a `bigfloat` with the result of the floating point division of two `bigint`s. Rounding is performed according to the current rounding mode. For this see the function `set_mode(rounding mode)`.

`ct bigfloat (const bigrational & n)`
This constructor initializes a `bigfloat` with the result of the floating point division of the numerator and the denominator of b. Rounding is performed according to the current rounding mode. For this see the functions `set_mode` and `get_mode` below.

`dt ~bigfloat ()`

Initialization

`static void set_precision (long p)`
sets the global decimal precision to p decimal places. Then the base digit precision is $t = \lceil (p \cdot \log(10)/\log(base)) \rceil + 3$, where *base* is the `bigint` base. Thus we have $|m| \leq base^p$ for all `bigfloat`s (m, e), i.e. m is a `bigint` of length at most t. Whenever necessary, internal computations are done with a higher precision.

`static long get_precision ()`
returns the current decimal precision.

`static void set_mode (long m)`
sets the rounding mode for the normalization routine according to the IEEE standard (see [33]). The following modes are available:

- `MP_TRUNC`: round to zero.
- `MP_RND`: round to nearest. If there are two possibilities to do this, round to even.
- `MP_RND_UP`: round to $+\infty$.
- `MP_RND_DOWN`: round to $-\infty$.

`static int get_mode ()`
return the current rounding mode.

Access Methods

Let a be of type `bigfloat`.

`long` a`.exponent ()`
returns the exponent of a.

`bigint` a`.mantissa ()`
returns the mantissa of a.

Assignments

Let a be of type `bigfloat`. The operator `=` is overloaded. The user may also use the following object methods for assignment:

`void` a`.assign_zero ()`
$a \leftarrow 0$.

`void` a`.assign_one ()`
$a \leftarrow 1$.

`void` a`.assign (const bigfloat &` n`)`

`void` a`.assign (const bigint &` n`)`

`void` a`.assign (long` n`)`

`void` a`.assign (unsigned long` n`)`

`void` a`.assign (int` n`)`

`void` a`.assign (double` n`)`
$a \leftarrow n$.

`void` a`.assign (const bigint &` n`, const bigint &` d`)`
$a \leftarrow n/d$ (floating point division) if $d \neq 0$. Otherwise the `lidia_error_handler` will be invoked.

Object Modifiers

`void` a`.negate ()`
$a \leftarrow -a$.

`void` a`.absolute_value ()`
$a \leftarrow |a|$.

`void` a`.inc ()`

`void inc (bigfloat &` a`)`
$a \leftarrow a + 1$.

`void` a`.dec ()`

`void dec (bigfloat &` a`)`
$a \leftarrow a - 1$.

`void` a`.multiply_by_2 ()`
$a \leftarrow a \cdot 2$.

`void` a`.divide_by_2 ()`
$a \leftarrow a/2$.

`void` a`.invert ()`
$a \leftarrow 1/a$, if $a \neq 0$. Otherwise the `lidia_error_handler` will be invoked.

`void` a`.swap (bigfloat &` b`)`

`void swap (bigfloat &` a`, bigfloat &` b`)`
swaps the values of a and b

`void` a`.randomize (const bigint &` n`, long` f`)`
sets a to $m \cdot 2^e$, where $|m| < n$, $|e| \leq f$, and e positive iff f is even. m is chosen randomly using the `randomize(`n`)` function of `bigint` and e using the class `random_generator`. n must not be equal to zero.

Arithmetical Operations

The following operators are overloaded and can be used in exactly the same way as in C++:

```
                    (unary) -, ++, --
                (binary) +, -, *, /, <<, >>
(binary with assignment) +=, -=, *=, /=, <<=, >>=
```

To avoid copying all operators exist also as functions:

`void add (bigfloat &` c`, const bigfloat &` a`, const bigfloat &` b`)`
$c \leftarrow a + b$.

`void add (bigfloat &` c`, const bigfloat &` a`, long` i`)`
$c \leftarrow a + i$.

`void add (bigfloat &` c`, long` i`, const bigfloat &` b`)`
$c \leftarrow i + b$.

`void subtract (bigfloat &` c`, const bigfloat &` a`, const bigfloat &` b`)`
$c \leftarrow a - b$.

`void subtract (bigfloat &` c`, const bigfloat &` a`, long` i`)`
$c \leftarrow a - i$.

`void subtract (bigfloat &` c`, long` i`, const bigfloat &` b`)`
$c \leftarrow i - b$.

`void multiply (bigfloat &` c`, const bigfloat &` a`, const bigfloat &` b`)`
$c \leftarrow a \cdot b$.

`void multiply (bigfloat & c, const bigfloat & a, long i)`
$c \leftarrow a \cdot i$.

`void multiply (bigfloat & c, long i, const bigfloat & b)`
$c \leftarrow i \cdot b$.

`void divide (bigfloat & c, const bigfloat & a, const bigfloat & b)`
$c \leftarrow a/b$, if $b \neq 0$. Otherwise the `lidia_error_handler` will be invoked.

`void divide (bigfloat & c, const bigfloat & a, long i)`
$c \leftarrow a/i$, if $i \neq 0$. Otherwise the `lidia_error_handler` will be invoked.

`void divide (bigfloat & c, long i, const bigfloat & b)`
$c \leftarrow i/b$, if $b \neq 0$. Otherwise the `lidia_error_handler` will be invoked.

`void divide (bigfloat & c, long i, long j)`
$c \leftarrow i/j$ (floating point division) if $j \neq 0$. Otherwise the `lidia_error_handler` will be invoked.

`void negate (bigfloat & a, const bigfloat & b)`
$a \leftarrow -b$.

`void invert (bigfloat & a, const bigfloat & b)`
$a \leftarrow 1/b$, if $b \neq 0$. Otherwise the `lidia_error_handler` will be invoked.

`bigfloat inverse (const bigfloat & a)`
returns $1/a$, if $a \neq 0$. Otherwise the `lidia_error_handler` will be invoked.

`void square (bigfloat & c, const bigfloat & a)`
$c \leftarrow a^2$.

`void sqrt (bigfloat & c, const bigfloat & a)`
$c \leftarrow \sqrt{a}$, where $a \geq 0$. If $a < 0$, the `lidia_error_handler` will be invoked.

`bigfloat sqrt (const bigfloat & a)`
returns $\sqrt{a}$, where $a \geq 0$. If $a < 0$, the `lidia_error_handler` will be invoked.

`void power (bigfloat & c, const bigfloat & a, long i)`
$c \leftarrow a^i$.

`void power (bigfloat & c, const bigfloat & a, const bigfloat & b)`
$c \leftarrow a^b$. If $a < 0$ and b is not an integer, the `lidia_error_handler` will be invoked.

`bigfloat power (const bigfloat & a, const bigfloat & b)`
returns a^b. If $a < 0$ and b is not an integer, the `lidia_error_handler` will be invoked.

Shift Operations

Let a be of type `bigfloat`. In `bigfloat` shifting, i.e. multiplication with powers of 2, is done using exponent manipulation.

`void shift_left (bigfloat & c, const bigfloat & b, long i)`
$c \leftarrow b \cdot 2^i$.

`void shift_right (bigfloat & c, const bigfloat & b, long i)`
$c \leftarrow b/2^i$.

Comparisons

Let a be of type `bigfloat`. The binary operators `==`, `!=`, `>=`, `<=`, `>`, `<` and the unary operator `!` (comparison with zero) are overloaded and can be used in exactly the same way as in C++.

`bool a.is_zero () const`
returns `true` if $a = 0$, `false` otherwise.

`bool a.is_approx_zero () const`
returns `true` if $|a| < base^{-t}$ (see initialization, page 68), `false` otherwise.

`bool a.is_one () const`
returns `true` if $a = 1$, `false` otherwise.

`bool a.is_gt_zero () const`
returns `true` if $a > 0$, `false` otherwise.

`bool a.is_ge_zero () const`
returns `true` if $a \geq 0$, `false` otherwise.

`bool a.is_lt_zero () const`
returns `true` if $a < 0$, `false` otherwise.

`bool a.is_le_zero () const`
returns `true` if $a \leq 0$, `false` otherwise.

`int a.abs_compare (const bigfloat & b) const`
returns $\mathrm{sgn}(|a| - |b|)$.

`int a.compare (const bigfloat & b) const`
returns $\mathrm{sgn}(a - b)$.

`int a.sign () const`
returns 1 if $a > 0$, 0 if $a = 0$ and 1 otherwise.

`int sign (const bigfloat & a)`
returns 1 if $a > 0$, 0 if $a = 0$ and 1 otherwise.

Type Checking and Conversion

Before assigning a `bigfloat` to a machine type (e.g. `int`) it is often useful to perform a test which checks whether the assignment can be done without overflow. Let a be of type `bigfloat`. The following methods return `true` if the assignment would be successful, `false` otherwise. No rounding is done.

`bool` a`.is_char (const bigfloat &)`

`bool` a`.is_uchar (const bigfloat &)`

`bool` a`.is_short (const bigfloat &)`

`bool` a`.is_ushort (const bigfloat &)`

`bool` a`.is_int (const bigfloat &)`

`bool` a`.is_uint (const bigfloat &)`

`bool` a`.is_long (const bigfloat &)`

`bool` a`.is_ulong (const bigfloat &)`

`bool` a`.is_double (const bigfloat &)`

There methods also exists as procedural versions, however, the object methods are preferred over the procedures.

`bool is_char (const bigfloat &` a`)`

`bool is_uchar (const bigfloat &` a`)`

`bool is_short (const bigfloat &` a`)`

`bool is_ushort (const bigfloat &` a`)`

`bool is_int (const bigfloat &` a`)`

`bool is_uint (const bigfloat &` a`)`

`bool is_long (const bigfloat &` a`)`

`bool is_ulong (const bigfloat &` a`)`

`bool is_double (const bigfloat &` a`)`

`void` a`.bigintify (bigint &` b`) const`

$b \leftarrow \lfloor a + 1/2 \rfloor$, i.e. b is the nearest integer to a.

`bool` a`.doublify (double &` d`) const`

tries to perform the assignment $d \leftarrow a$. If successful, i.e. no overflow occurs, the function returns `false` and assigns a to d. Otherwise it returns `true` and leaves the value of d unchanged.

`bool` a`.intify (int &` i`) const`

tries to perform the assignment $i \leftarrow a$. If successful, i.e. no overflow occurs, the function returns `false` and assigns a to i. Otherwise it returns `true` and leaves the value of i unchanged.

`bool` a`.longify (long &` l`) const`

tries to perform the assignment $l \leftarrow a$. If successful, i.e. no overflow occurs, the function returns `false` and assigns a to l. Otherwise it returns `true` and leaves the value of l unchanged.

Basic Methods and Functions

Let a be of type `bigfloat`.

`lidia_size_t` a`.bit_length () const`
returns the length of the mantissa of a in binary digits.

`lidia_size_t` a`.length () const`
returns the length of the mantissa of a in `bigint` base digits.

`bigfloat abs (const bigfloat &` a`)`
returns $|a|$.

`void besselj (bigfloat &` c`, int` n`, const bigfloat &` a`)`
$c \leftarrow J_n(a)$ (Bessel Function of first kind and integer order).

`bigfloat besselj (int` n`, const bigfloat &` a`)`
returns $J_n(a)$ (Bessel Function of first kind and integer order).

`void ceil (bigint &` c`, const bigfloat &` a`)`

`void ceil (bigfloat &` c`, const bigfloat &` a`)`
$c \leftarrow \lceil a \rceil$, i.e. c is the smallest integer which is greater than or equal to a.

`bigfloat ceil (const bigfloat &` a`)`
returns $\lceil a \rceil$, i.e. c is the smallest integer which is greater than or equal to a.

`void floor (bigint &` c`, const bigfloat &` a`)`

`void floor (bigfloat &` c`, const bigfloat &` a`)`
$c \leftarrow \lfloor a \rfloor$, i.e. c is the largest integer which is less than or equal to a.

`bigfloat floor (const bigfloat &` a`)`
returns $\lfloor a \rfloor$, i.e. c is the largest integer which is less than or equal to a.

`void round (bigint &` c`, const bigfloat &` a`)`

`void round (bigfloat &` c`, const bigfloat &` a`)`
$c \leftarrow \lfloor a + 1/2 \rfloor$, i.e. c is the nearest integer to a.

`bigfloat round (const bigfloat &` a`)`
returns $\lfloor a + 1/2 \rfloor$, i.e. c is the nearest integer to a.

`void truncate (bigint &` c`, const bigfloat &` a`)`

`void truncate (bigfloat &` c`, const bigfloat &` a`)`
$c \leftarrow \mathrm{sgn}(a) \cdot \lfloor |a| \rfloor$.

`bigfloat truncate (const bigfloat &` a`)`
returns $\mathrm{sgn}(a) \cdot \lfloor |a| \rfloor$.

`void exp (bigfloat & c, const bigfloat & a)`
$c \leftarrow e^a$.

`bigfloat exp (const bigfloat & a)`
returns $\exp(a) = e^a$ ($e \approx 2.1828\ldots$).

`void log (bigfloat & c, const bigfloat & a)`
$c \leftarrow \log(a)$ (natural logarithm to base $e \approx 2.71828\ldots$), where $a > 0$. If $a \leq 0$, the `lidia_error_handler` will be invoked.

`bigfloat log (const bigfloat & a)`
returns $\log(a)$ (natural logarithm to base $e = 2.71828\ldots$), where $a > 0$. If $a \leq 0$ the `lidia_error_handler` will be invoked.

`bigfloat log2p1 (const bigfloat & a)`
returns $\log_2(1+a)$ (logarithm to base 2). If $a < \sqrt{2}/2$, the Taylor series is used. (written by John Bunda `bunda@centtech.com`)

`long exponent (const bigfloat & a)`
returns the exponent of a.

`bigint mantissa (const bigfloat & a)`
returns the mantissa of a.

(Inverse) Trigonometric Functions

Note that the arguments of the trigonometrical functions are supposed to be given as radiants.

`void sin (bigfloat & c, const bigfloat & a)`
$c \leftarrow \sin(a)$.

`bigfloat sin (const bigfloat & a)`
returns $\sin(a)$.

`void cos (bigfloat & c, const bigfloat & a)`
$c \leftarrow \cos(a)$.

`bigfloat cos (const bigfloat & a)`
returns $\cos(a)$.

`void tan (bigfloat & c, const bigfloat & a)`
$c \leftarrow \tan(a)$ if $a \neq (2k+1)\,\pi/2$, $k \in \mathbb{Z}$. Otherwise the `lidia_error_handler` will be invoked.

`bigfloat tan (const bigfloat & a)`
returns $\tan(a)$ if $a \neq (2k+1)\,\pi/2$, $k \in \mathbb{Z}$. Otherwise the `lidia_error_handler` will be invoked.

`void cot (bigfloat & c, const bigfloat & a)`
$c \leftarrow \cot(a)$ if $a \neq k\,\pi$, $k \in \mathbb{Z}$. Otherwise the `lidia_error_handler` will be invoked.

`bigfloat cot (const bigfloat & a)`
returns $\cot(a)$ if $a \neq k\,\pi$, $k \in \mathbb{Z}$. Otherwise the `lidia_error_handler` will be invoked.

`void asin (bigfloat & c, const bigfloat & a)`
$c \leftarrow \arcsin(a)$ if $a \in [-1, 1]$. Otherwise the `lidia_error_handler` will be invoked.

`bigfloat asin (const bigfloat & a)`
returns $\arcsin(a)$ if $a \in [-1, 1]$. Otherwise the `lidia_error_handler` will be invoked.

`void acos (bigfloat & c, const bigfloat & a)`
$c \leftarrow \arccos(a)$ if $a \in [-1, 1]$. Otherwise the `lidia_error_handler` will be invoked.

`bigfloat acos (const bigfloat & a)`
returns $\arccos(a)$ if $a \in [-1, 1]$. Otherwise the `lidia_error_handler` will be invoked.

`void atan (bigfloat & c, const bigfloat & a)`
$c \leftarrow \arctan(a)$.

`bigfloat atan (const bigfloat & a)`
returns $\arctan(a)$.

`void atan2 (bigfloat & c, const bigfloat & a, const bigfloat & b)`
$c \leftarrow \arctan(a/b)$ in the range of $[-\pi, \pi[$, i.e. `atan2(a, b)` calculates the argument (respectively phase) of a, b.

`bigfloat atan2 (const bigfloat & a, const bigfloat & b)`
returns $\arctan(a/b)$ in the range of $[-\pi, \pi[$, i.e. `atan2(a, b)` calculates the argument (respectively phase) of a, b.

`void acot (bigfloat & c, const bigfloat & a)`
$c \leftarrow \operatorname{arccot}(a)$.

`bigfloat acot (const bigfloat & a)`
returns $\operatorname{arccot}(a)$.

(Inverse) Hyperbolic Trigonometric Functions

`void sinh (bigfloat & c, const bigfloat & a)`
$c \leftarrow \sinh(a)$.

`bigfloat sinh (const bigfloat & a)`
returns $\sinh(a)$.

`void cosh (bigfloat & c, const bigfloat & a)`
$c \leftarrow \cosh(a)$.

`bigfloat cosh (const bigfloat & a)`
returns $\cosh(a)$.

`void tanh (bigfloat & c, const bigfloat & a)`
$c \leftarrow \tanh(a)$.

`bigfloat tanh (const bigfloat & a)`
returns $\tanh(a)$.

`void coth (bigfloat & c, const bigfloat & a)`
$c \leftarrow \coth(a)$.

`bigfloat coth (const bigfloat & a)`
returns $\coth(a)$.

`void asinh (bigfloat & c, const bigfloat & a)`
$c \leftarrow \operatorname{arsinh}(a)$.

`bigfloat asinh (const bigfloat & a)`
returns $\operatorname{arsinh}(a)$.

`void acosh (bigfloat & c, const bigfloat & a)`
$c \leftarrow \operatorname{arcosh}(a)$, where $a \geq 1$.

`bigfloat acosh (const bigfloat & a)`
returns $\operatorname{arcosh}(a)$, where $a \geq 1$.

`void atanh (bigfloat & c, const bigfloat & a)`
$c \leftarrow \operatorname{artanh}(a)$, where $a \in]-1, 1[$.

`bigfloat atanh (const bigfloat & a)`
returns $\operatorname{artanh}(a)$, where $a \in]-1, 1[$.

`void acoth (bigfloat & c, const bigfloat & a)`
$c \leftarrow \operatorname{arcoth}(a)$, where $a \in \mathbb{R} \setminus [-1, 1]$.

`bigfloat acoth (const bigfloat & a)`
returns $\operatorname{arcoth}(a)$, where $a \in \mathbb{R} \setminus [-1, 1]$.

Constants

Constants are calculated according to the precision set in the initialization phase (i.e. `bigfloat::precision`).

`void constant_E (bigfloat & a)`
$a \leftarrow e \approx 2.718281828459\ldots$

`void constant_Pi (bigfloat & a)`
$a \leftarrow \pi \approx 3.141592653589\ldots$

`void constant_Euler (bigfloat & a)`
$a \leftarrow \gamma = 0.5772156649\ldots$

`void constant_Catalan (bigfloat & a)`
$a \leftarrow G \approx 0.9159655941\ldots$

`bigfloat E ()`

`bigfloat Pi ()`

`bigfloat Euler ()`

`bigfloat Catalan ()`

Input/Output

Let a be of type `bigfloat`.

`istream` operator `>>` and `ostream` operator `<<` are overloaded. Furthermore, you can use the following member functions for writing to and reading from a file in binary or ASCII format, respectively. Input and output of a `bigfloat` are in the following format:

$$\pm zE \pm l \quad \text{or} \quad \pm ze \pm l \ ,$$

where z has the form x, $x.y$ or y where x and y are of type int and l is of type long. Note that you have to manage by yourself that successive `bigfloat`s have to be separated by blanks.

`int a.scan_from_file (FILE * fp)`
scans a form the ASCII file `fp` using `fscanf`.

`int a.print_to_file (FILE * fp) const`
prints a in ASCII format to the file `fp` using `fprintf`.

`int string_to_bigfloat (char * s, bigfloat & a)`
converts the string `s` to a `bigfloat` a and returns the number of used characters.

`int bigfloat_to_string (const bigfloat & a, char * s)`
converts the `bigfloat` a to a string `s` and returns the number of used characters. The number of characters allocated for `s` must be at least `a.bit_length()/3 + 10`.

See also

`bigint`, `bigrational`, `bigcomplex`.

Notes

The implementation of the transcendental functions is optimized for the `sun4` architecture.

The member functions `scan_from_file(FILE *)` and `print_to_file(FILE *)` are included because there are still compilers which do not handle `fstream`s correctly. In a future release those functions will disappear.

Storing a double variable d into a `bigfloat` x (i.e. using the assignment $x \leftarrow d$) is done with the precision of the double mantissa. The precision of IEEE doubles is 53 bits).

Examples

The following program calculates the number $\sqrt{\pi}$ to 100 decimal places.

```
#include <LiDIA/bigfloat.h>

int main()
{
    bigfloat::set_precision(100);

    bigfloat x = sqrt(Pi());

    cout << "\n sqrt(Pi) = " << x << flush;
    return 0;
}
```

An extensive example of `bigfloat` can be found in LiDIA's installation directory under `LiDIA/src/simple_classes/bigfloat_appl.cc`

Author

Thomas Papanikolaou

xbigfloat

Name

`xbigfloat` ..... multiprecision floating point arithmetic

Abstract

`xbigfloat` is a class for doing multiprecision arithmetic with approximations to real numbers. It implements the model for computing with approximations described in [15]. It supports basic arithmetic operations, shift operations, comparisons, the square root function, the exponential function, and the logarithmical function. In contrast to the class `bigfloat`, page 67, it allows the specification of error bounds for most of the functions.

Description

An `xbigfloat` is a pair (m, e), where m, e are integers. It represents the number

$$m \cdot 2^{e-b(m)} \ ,$$

where $b(m)$ is the number of bits of m. The number m is called the *mantissa* and e is called the *exponent*. A complete description of that floating point model together with an error analysis can be found in [15].

For the description of the functions, we introduce the notation of absolute and relative approximations. Let $r \in \mathbb{R}$ and $k \in \mathbb{Z}$.

A relative k-approximation to r is an `xbigfloat` $f = (m, e)$ with $b(m)q \leq k + 3$ and such that there exists an $\epsilon \in \mathbb{R}$ with $f = r(1 + \epsilon)$ and $|\epsilon| < 2^{-k}$.

An absolute k-approximation to r is an `xbigfloat` $f = (m, e)$ such that $|f - r| < 2^{-k}$ and $e \geq b(m) - k - 1$.

In the following description of functions, assignment, comparision, and arithmetic are always meant as operations on rational numbers without any errors, e.g., due to rounding.

Constructors/Destructor

`ct xbigfloat ()`
initializes with 0.

`ct xbigfloat (const xbigfloat & n)`

`ct xbigfloat (const bigfloat & n)`

`ct xbigfloat (const bigint & n)`

`ct xbigfloat (long n)`

`ct xbigfloat (unsigned long` n`)`

`ct xbigfloat (int` n`)`

`ct xbigfloat (double` n`)`
initializes with n.

`dt ~xbigfloat ()`

Assignments

Let a be of type `xbigfloat`. The operator `=` is overloaded. The user may also use the following object methods for assignment:

`void` a`.assign_zero ()`
$a \leftarrow 0$.

`void` a`.assign_one ()`
$a \leftarrow 1$.

`void` a`.assign (const xbigfloat &` n`)`

`void` a`.assign (const bigfloat &` n`)`

`void` a`.assign (const bigint &` n`)`

`void` a`.assign (long` n`)`

`void` a`.assign (unsigned long` n`)`

`void` a`.assign (int` n`)`

`void` a`.assign (double` n`)`

`void` a`.assign (xdouble` n`)`
$a \leftarrow n$

Access Methods

Let a be of type `xbigfloat`.

`long` a`.get_exponent () const`
returns the exponent e of a.

`long` a`.exponent () const`
returns the exponent e of a.

`bigint` a`.get_mantissa () const`
returns the mantissa m of a.

`bigint` a`.mantissa () const`
returns the mantissa m of a.

Object Modifiers

`void` a`.negate ()`
$a \leftarrow -a$.

`void` a`.absolute_value ()`
$a \leftarrow |a|$.

`void` a`.inc ()`

`void inc (xbigfloat &` a`)`
$a \leftarrow a + 1$.

`void` a`.dec ()`

`void dec (xbigfloat &` a`)`
$a \leftarrow a - 1$.

`void` a`.multiply_by_2 ()`
$a \leftarrow a \cdot 2$.

`void` a`.divide_by_2 ()`
$a \leftarrow a/2$.

`void` a`.swap (xbigfloat &` b`)`

`void swap (xbigfloat &` a`, xbigfloat &` b`)`
Swaps the values of a and b.

`void` a`.randomize (const bigint &` n`, long` f`)`
Calls the `bigfloat` functions `randomize(`n`,` f`)` to create a random `bigfloat` and assigns it to a.

Arithmetical Operations

The following operators are overloaded. The result of these operations is an `xbigfloat` that represents the rational number which is the result of the operation carried out over the rationals without any rounding error.

`(unary) -`
`(binary) +, -, *, <<, >>`
`(binary with assignment) +=, -=, *=, <<=, >>=`

To avoid copying all operators exist also as functions:

`void add (xbigfloat &` c`, const xbigfloat &` a`, const xbigfloat &` b`)`
$c \leftarrow a + b$.

`void subtract (xbigfloat &` c`, const xbigfloat &` a`, const xbigfloat &` b`)`
$c \leftarrow a - b$.

`void multiply (xbigfloat &` c`, const xbigfloat &` a`, const xbigfloat &` b`)`
$c \leftarrow a \cdot b$.

`void divide (xbigfloat & c, const xbigfloat & a, const xbigfloat & b, long k)`
c is a relative k-approximation to a/b, if $b \neq 0$. Otherwise the `lidia_error_handler` will be invoked.

`void divide (bigint & q, const xbigfloat & a, const xbigfloat & b)`
$q \leftarrow \lfloor a/b \rfloor$, if $b \neq 0$. Otherwise the `lidia_error_handler` will be invoked.

`void square (xbigfloat & c, const xbigfloat & a)`
$c \leftarrow a^2$.

`void sqrt (xbigfloat & c, const xbigfloat & a, long k)`
c is a relative k-approximation to $\sqrt{a}$, where $a \geq 0$. If $a < 0$, the `lidia_error_handler` will be invoked.

`void exp (xbigfloat & c, const xbigfloat & a, long k)`
c is a relative k-approximation to $\exp(a)$.

`void log (xbigfloat & c, const xbigfloat & a, long k)`
c is an absolute k-approximation to $\log(a)$ (natural logarithm to base $e \approx 2.71828\ldots$), where $a > 0$. If $a \leq 0$, the `lidia_error_handler` will be invoked.

Shift Operations

Let a be of type `xbigfloat`. In `xbigfloat` shifting, i.e. multiplication with powers of 2, is done using exponent manipulation.

`void shift_left (xbigfloat & c, const xbigfloat & b, long i)`
$c \leftarrow b \cdot 2^i$.

`void shift_right (xbigfloat & c, const xbigfloat & b, long i)`
$c \leftarrow b/2^i$.

Comparisons

Let a be of type `xbigfloat`. The binary operators `==`, `!=`, `>=`, `<=`, `>`, and `<` are overloaded.

`bool a.is_zero () const`
returns `true` if $a = 0$, `false` otherwise.

`bool a.is_one () const`
returns `true` if $a = 1$, `false` otherwise.

`bool a.is_negative () const`
returns `true` if $a < 0$, `false` otherwise.

`bool a.is_positive () const`
returns `true` if $a > 0$, `false` otherwise.

`bool a.is_equal (const xbigfloat & b) const`
returns `true` if $a = b$, `false` otherwise.

`int` a`.sign () const`
returns 1, if $a > 0$, 0 if $a = 0$, and -1 otherwise.

Basic Methods and Functions

Let a be of type `xbigfloat`.

`long` a`.b_value () const`
returns the b value of a, i.e., the exponent of a, see [15].

`long b_value (const xbigfloat &` a`) const`
returns the b value of a, i.e., the exponent of a, see [15].

`int` a`.get_sign () const`
returns 1, if $a > 0$, 0 if $a = 0$, and -1 otherwise.

`void ceil (bigint &` c`, const xbigfloat &` a`)`
$c \leftarrow \lceil a \rceil$, i.e. c is the smallest integer which is greater than or equal to a.

`void floor (bigint &` c`, const xbigfloat &` a`)`
$c \leftarrow \lfloor a \rfloor$, i.e. c is the largest integer which is less than or equal to a.

`void truncate (xbigfloat &` c`, const xbigfloat &` a`, lidia_size_t` k`)`
truncates the mantissa of a to k bits and assigns the result to c.

`static bool check_relative_error (const xbigfloat &` x`, const xbigfloat &` y`, long` k`, long` c`)`
This function returns `true`, if and only if $|x - y| < v$, where $v = 2^{-k+1} \cdot (2^{b(x)} + 2^{b(y)-c})$.

This can be used to check the correctness of relative errors. I.e., if $k \geq 1$, $c \geq 0$, $|x/z - 1| < 2^{-k}$, and $|y/z - 1| < 2^{-k-c}$, then $|x - y| < v$ for $z \neq 0$.

`static bool check_absolute_error (const xbigfloat &` x`, const xbigfloat &` y`, long` k`, long` c`)`
This function returns `true`, if and only if $|x - y| < v$, where $v = 2^{-k} + 2^{-k-c}$.

This can be used to check the correctness of absolute errors. I.e., if $c \geq 0$, $|x - z| < 2^{-k}$, and $|y - z| < 2^{-k-c}$, then $|x - y| < v$.

Input/Output

Let a be of type `xbigfloat`.

`istream` operator `>>` and `ostream` operator `<<` are overloaded. Input and output of a `xbigfloat` are in the following format:

$$(m, e)$$

For example, to input the number $1/2$, you have to enter $(1, 0)$, because $1/2 = 1 \cdot 2^{0-b(1)}$.

For debugging purposes, the following function is implemented too.

`void` *a*`.print_as_bigfloat (ostream & out = cout) const`

transforms the number into a `bigfloat` object and prints it to the stream `out`. Note that the output of that function depends on the `bigfloat` precision and is only an approximation to the number represented by the `xbigfloat` object.

See also

`bigint`, `bigfloat`.

Examples

The following program computes a relative 5-approximation x to the square root of 2, i.e., $x \leftarrow sqrt(2) \cdot (1+\epsilon)$, $|\epsilon| < 2^{-5}$.

For convenience, the program also prints the value as a `bigfloat`, but note that the result of that output function depends on the precision of `bigfloat` and is not necessarily a relative 5-approximation anymore.

```
#include <LiDIA/xbigfloat.h>

int main()
{
    xbigfloat x;

    sqrt(x,2,5);
    cout << "Relative 5-approximation x to sqrt(2) = " << x << endl;
    cout << "x as bigfloat "; x.print_as_bigfloat(); cout << endl;
    return 0;
}
```

The output of the program is

```
Relative 5-approximation to sqrt(2) = (181,1)
x as bigfloat 1.4140625
```

An extensive example of `xbigfloat` can be found in LiDIA's installation directory under `LiDIA/src/base/simple_classes/xbigfloat_appl.cc`

Author

Markus Maurer

bigcomplex

Name

`bigcomplex` .....multiprecision complex floating-point arithmetic

Abstract

`bigcomplex` is a class for doing multiprecision complex floating point arithmetic. It supports for example arithmetic operations, comparisons and basic triginometric functions.

Description

A `bigcomplex` is a pair of two `bigfloat`s (re, im), where re is called the *real part* and im the *imaginary part*. This pair represents the complex number $re+im \cdot i$, where i is the imaginary unit. Before declaring a `bigcomplex` it is possible to set the decimal precision of the `bigfloat`s (re, im) to p by `bigcomplex::precision(`p`)` or by `bigfloat::precision(`p`)`. Both calls have the same effect and set the `bigfloat` precision to p (compare class `bigfloat`, page 67).

If the precision has not been set, a default precision of $t = 5$ `bigint` base digits is used.

Constructors/Destructor

`ct bigcomplex ()`
Initializes with 0.

`ct bigcomplex (const bigfloat &` x`)`
initializes the real part with x, the imaginary part with 0.

`ct bigcomplex (const bigcomplex &` z`)`
initializes with z.

`ct bigcomplex (const bigfloat &` re`, const bigfloat &` im`)`
initializes the real with re and the imaginary part with im.

`dt ~bigcomplex ()`

Initialization

`static void precision (long` p`)`

sets the global decimal precision of the `bigfloats` re, im to p decimal places (compare `bigfloat`). Note that the `bigfloat` precision is set globally to p by this call. Whenever necessary internal computations are done with a higher precision.

`static void mode (int` m`)`

sets the rounding mode for the normalization routine according to the IEEE standard (see [33]). The following modes are available:

- `MP_TRUNC`: round to zero.
- `MP_RND`: round to nearest. If there are two possibilities to do this, round to even.
- `MP_RND_UP` round to $+\infty$.
- `MP_RND_DOWN` round to $-\infty$.

Type Checking and Conversion

Before assigning a `bigcomplex` to a `bigfloat` it is often useful to check whether the assignment can be done without error, i.e. whether the imaginary part im is zero.

`bool is_bigfloat (const bigcomplex &)`

returns `true` if the imaginary part is zero, `false` otherwise.

Assignments

Let x be of type `bigcomplex`. The operator `=` is overloaded. The user may also use the following object methods for assignment:

`void` x`.assign_zero ()`

$x \leftarrow 0$.

`void` x`.assign_one ()`

$x \leftarrow 1$.

`void` x`.assign (const bigfloat &` y`)`

$x \leftarrow y$.

`void` x`.assign (const bigfloat &` y`, const bigfloat &` z`)`

$x \leftarrow y + z \cdot i$.

`void` x`.assign (const bigcomplex &` y`)`

$x \leftarrow y$.

Access Methods

Let x be of type `bigcomplex`.

`const bigfloat & `x`.real () const`

`const bigfloat & real (const bigcomplex & `x`)`
returns the real part of x.

`const bigfloat & `x`.imag () const`

`const bigfloat & imag (const bigcomplex & `x`)`
returns the imaginary part of x.

Object Modifiers

`void `x`.negate ()`
$x \leftarrow -x$.

`void `x`.invert ()`
$x \leftarrow 1/x$ if $x \neq 0$. Otherwise the `lidia_error_handler` will be invoked.

Arithmetical Operations

The following operators are overloaded and can be used in exactly the same way as in C++:

```
(unary) -
(binary) +, -, *, /
(binary with assignment) +=, -=, *=, /=
```

`void negate (bigcomplex & `x`, const bigcomplex & `y`)`
$x \leftarrow -y$.

`void add (bigcomplex & `x`, const bigcomplex & `y`, const bigcomplex & `z`)`

`void add (bigcomplex & `x`, const bigcomplex & `y`, const bigfloat & `z`)`

`void add (bigcomplex & `x`, const bigfloat & `y`, const bigcomplex & `z`)`
$x \leftarrow y + z$.

`void subtract (bigcomplex & `x`, const bigcomplex & `y`, const bigcomplex & `z`)`

`void subtract (bigcomplex & `x`, const bigcomplex & `y`, const bigfloat & `z`)`

`void subtract (bigcomplex & `x`, const bigfloat & `y`, const bigcomplex & `z`)`
$x \leftarrow y - z$.

`void multiply (bigcomplex & `x`, const bigcomplex & `y`, const bigcomplex & `z`)`

`void multiply (bigcomplex & `x`, const bigcomplex & `y`, const bigfloat & `z`)`

`void multiply (bigcomplex & `x`, const bigfloat & `y`, const bigcomplex & `z`)`
$x \leftarrow y \cdot z$.

`void square (bigcomplex & x, const bigcomplex & y)`
$x \leftarrow y^2$.

`bigcomplex square (const bigcomplex & x)`
returns x^2.

`void divide (bigcomplex & x, const bigcomplex & y, const bigcomplex & z)`

`void divide (bigcomplex & x, const bigcomplex & y, const bigfloat & z)`

`void divide (bigcomplex & x, const bigfloat & y, const bigcomplex & z)`
$x \leftarrow y/z$ if $z \neq 0$. Otherwise the `lidia_error_handler` will be invoked.

`void invert (bigcomplex & x, const bigcomplex & y)`
$x \leftarrow 1/y$ if $y \neq 0$. Otherwise the `lidia_error_handler` will be invoked.

`bigcomplex inverse (const bigcomplex & x)`
returns $1/x$ if $x \neq 0$. Otherwise the `lidia_error_handler` will be invoked.

`void sqrt (bigcomplex & x, const bigcomplex & y)`
$x \leftarrow \sqrt{y}$.

`bigcomplex sqrt (const bigcomplex & x)`
returns $\sqrt{x}$.

`void power (bigcomplex & x, const bigcomplex & y, const bigcomplex & z)`

`void power (bigcomplex & x, const bigcomplex & y, const bigfloat & z)`
sets $x \leftarrow y^z$.

`void power (bigcomplex & x, const bigcomplex & y, long i)`
sets $x \leftarrow y^i$.

`bigcomplex power (const bigcomplex & y, const bigcomplex & z)`

`bigcomplex power (const bigcomplex & y, const bigfloat & z)`
returns y^z.

`bigcomplex power (const bigcomplex & y, long i)`
returns y^i.

Comparisons

`bigcomplex` supports the binary operators `==` and `!=`. Furthermore there exists the following function:

`bool x.is_zero () const`
returns `true` if x is zero, `false` otherwise.

`bool` x`.is_one () const`
returns `true` if x is 1, `false` otherwise.

`bool` x`.is_approx_zero () const`
returns `true` if the real and the imaginary part of x are approximately zero (compare class `bigfloat`, `false` otherwise.

`bool` x`.is_real () const`
returns `true` if the imaginary part of x is zero, `false` otherwise.

`bool` x`.is_approx_real () const`
returns `true` if the imaginary part of x is approximately zero (compare class `bigfloat`, `false` otherwise.

`bool` x`.is_imaginary () const`
returns `true` if the real part of x is zero, `false` otherwise.

`bool` x`.is_approx_imaginary () const`
returns `true` if the real part of x is approximately zero (compare class `bigfloat`, `false` otherwise.

`bool` x`.is_equal (const bigcomplex &` y`) const`
returns `true` if $x = y$, `false` otherwise.

Basic Methods and Functions

`void conj (bigcomplex &` x`, const bigcomplex &` y`)`
returns x as the complex conjugate of y.

`bigcomplex conj (const bigcomplex &` x`)`
returns the complex conjugate of x.

`void exp (bigcomplex &` x`, const bigcomplex &` y`)`
$x \leftarrow \exp(x) = e^y$ ($e \approx 2.71828\ldots$).

`bigcomplex exp (const bigcomplex &` x`)`
returns e^x.

`void log (bigcomplex &` x`, const bigcomplex &` y`)`
$x \leftarrow \log(y)$ (natural logarithm to base $e \approx 2.71828\ldots$).

`bigcomplex log (const bigcomplex &` x`)`
returns $\log(x)$.

`void polar (bigcomplex &` x`, const bigfloat &` r`, const bigfloat &` t`)`
$x \leftarrow r \cdot \cos(t) + r \cdot \sin(t) \cdot i$.

`bigcomplex polar (const bigfloat &` r`, const bigfloat &` t`)`
returns the complex number $r \cdot \cos(t) + r \cdot \sin(t) \cdot i$.

`bigfloat abs (const bigcomplex & x)`
returns $\sqrt{\Re^2 x + \Im^2 x}$.

`bigfloat hypot (const bigcomplex & x)`
returns $|x|$.

`bigfloat arg (const bigcomplex & x)`
returns $\arctan(\Im x / \Re x)$.

`bigfloat norm (const bigcomplex & x)`
returns the norm of x.

`void swap (bigcomplex & x, bigcomplex & y)`
exchanges the values of x and y.

(Inverse) Trigonometric Functions

`void sin (bigcomplex & x, const bigcomplex & y)`
$x \leftarrow \sin(y)$.

`bigcomplex sin (const bigcomplex & x)`
returns $\sin(x)$.

`void cos (bigcomplex & x, const bigcomplex & y)`
$x \leftarrow \cos(y)$.

`bigcomplex cos (const bigcomplex & x)`
returns $\cos(x)$.

(Inverse) Hyperbolic Trigonometric Functions

`void sinh (bigcomplex & x, const bigcomplex & y)`
$x \leftarrow \sinh(y)$.

`bigcomplex sinh (const bigcomplex & x)`
returns $\sinh(x)$.

`void cosh (bigcomplex & x, const bigcomplex & x)`
$x \leftarrow \cosh(y)$.

`bigcomplex cosh (const bigcomplex & x)`
returns $\cosh(x)$.

Input/Output

`istream` operator `>>` and `ostream` operator `<<` are overloaded and can be used in the same way as in C++. If the imaginary part of the `bigcomplex` is zero, then input and output of a `bigcomplex` may be in one of the following formats:

$$re \text{ or } (re, 0) \ ,$$

where *re* is a `bigfloat` in the appropriate format described in the `bigfloat` class description. If the imaginary part of the `bigfloat` is not zero, then input and output have the format

$$(re, im)$$

where *re* and *im* are `bigfloat`s in the appropriate format described in the `bigfloat` class description.

Note that you have to manage by yourself that successive `bigcomplex` numbers have to be separated by blanks.

See also

`bigfloat`

Notes

The structure of `bigcomplex` is strongly based on AT&T's and GNU's complex class. `bigcomplex` is not yet complete. Further methods will be incorporated in the next release.

Examples

```
#include <LiDIA/bigcomplex.h>

int main()
{
    bigcomplex::precision(100);

    bigcomplex a, b, c;

    cout << "Please enter a : "; cin >> a ;
    cout << "Please enter b : "; cin >> b ;
    cout << endl;
    c = a + b;
    cout << "a + b = " << c << endl;
}
```

For further reference please refer to `LiDIA/src/simple_classes/bigcomplex_appl.cc`

Author

Ingrid Biehl, Thomas Papanikolaou

Chapter 4

Vectors and Hash Tables

base_vector

Name

`base_vector < T >` ..... parameterized base vector class

Abstract

The class `base_vector< T >` realizes the concept of a vector as a more advanced array

$$v = (v_i) \in T^m \ , \quad 0 \leq i < m \ .$$

It allows to create vectors and offers elementary access functions. `T` is allowed to be either a built-in type or a class.

Description

A variable v of type `base_vector< T >` consists of a 1-dimensional array `T * data`, an unsigned integer `mode`, a float `expansion_ratio` as well as two integers `size` and `capacity` which describe the length of the vector. We call the number of allocated elements the capacity of v. The size of the vector is the number of items actually used in v. Thus, this value will never decrease; neither will it exceed the capacity.

Both values can individually be changed for a vector v by using the functions v`.set_capacity()` and v`.set_size()`, respectively.

As an example, the declaration

```
base_vector < int > v(10,0)
```

will create a `base_vector< int >` with 10 elements allocated, so v`.capacity` $= 10$ and v`.size` $= 0$. If we use the first s elements of v only, s will be the size of v. In our example, after the declaration, v has v`.size` $= 0$ and if we assign

$$v[0] = 1 \ , \quad v[1] = 2$$

we obtain v`.size` $= 2$. As shown in this example the first element of v has index 0 and the last one has index v`.size` $- 1$.

Furthermore, a `base_vector` v can be of either fixed or expanding size. For so called fixed vectors one can only change the number of allocated items in the vector by using the member function v`.set_capacity()`. I.e. neither is it possible to access the vector at an index which is greater than the vector's capacity, nor may function v`.set_size()` be called with a size greater than this value.

Vectors of expanding size automatically allocate memory if access is required for an index greater than its current capacity or the number of items by using the member function v`.set_size()` exceeds that value. The new amount of memory to be allocated in these cases will then be computed using the expansion ratio of the vector. This is a factor by which the required number of elements will be multiplied before actually reallocating memory. The value of this factor for a vector v is 2.0 by default and may be altered at any time using the member function v`.set_exp_ratio()`. The expansion ratio has to be at least 1.0.

The property, whether a vector is of fixed or expanding size, is stored in a member called `mode`. The value of this member can be either of the constants `FIXED` or `EXPAND`. By default, each vector is of fixed size. Its mode can explicitly be set by either using the function `v.set_mode()` or by adding `FIXED` or `EXPAND` as an additional argument to the corresponding constructor.

In the following descriptions we use v`.size` to label the size and v`.capacity` to label the capacity of vector v.

Constructors/Destructor

`ct base_vector< T > ()`

constructs a vector with capacity 0.

`ct base_vector< T > (lidia_size_t` c`, char` *mode*`)`

constructs a vector with capacity c initialized with values generated by the default constructor for type `T`. *mode* can be either `FIXED` or `EXPAND`.

`ct base_vector< T > (lidia_size_t` c`, lidia_size_t` s`)`

constructs a vector with capacity c and size s initialized with values generated by the default constructor for type `T`.

`ct base_vector< T > (lidia_size_t` c`, lidia_size_t` s`, char` *mode*`)`

constructs a vector with capacity c and size s initialized with values generated by the default constructor for type `T`. *mode* can be either `FIXED` or `EXPAND`.

`ct base_vector< T > (const base_vector< T > &` w`)`

constructs a vector with capacity w`.size` initialized with the elements of w.

`ct base_vector< T > (const base_vector< T > &` w`, char` *mode*`)`

constructs a vector with capacity w`.size` initialized with the elements of w. *mode* can be either `FIXED` or `EXPAND`.

`ct base_vector< T > (const T *` a`, lidia_size_t` l`)`

constructs a vector with capacity l and size l initialized with the first l elements of the array a. If the array a has less than l elements the `lidia_error_handler` will be invoked.

`ct base_vector< T > (const T *` a`, lidia_size_t` l`, char` *mode*`)`

constructs a vector with capacity l and size l initialized with the first l elements of the array a. If the array a has less than l elements the `lidia_error_handler` will be invoked. *mode* can be either `FIXED` or `EXPAND`.

`dt ~base_vector< T > ()`

Access Methods

Let v be of type `base_vector< T >`. Note, that we start the numbering of elements of a `base_vector< T >` with zero.

Access to the size

`lidia_size_t v.size () const`

returns the size of v.

`lidia_size_t v.get_size () const`

returns the size of v.

`void v.set_size (lidia_size_t` s`)`

sets the size of v to s if the capacity of v is large enough. If s exceeds v`.capacity` and v is of expanding size, new memory will be allocated by using the function v`.set_capacity()`. The number of elements allocated in that case is $s \cdot v$`.exp_ratio`. If these actions can not successfully be performed, the `lidia_error_handler` will be invoked.

Access to the capacity

`lidia_size_t` v`.capacity () const`

returns the capacity of v.

`lidia_size_t` v`.get_capacity () const`

returns the capacity of v.

`void` v`.set_capacity (lidia_size_t` c`)`

sets the capacity of v to c by allocating exactly c elements for v. Here, c must be a positive value. If c is 0, the elements of v will be deleted. If an error occurs in this function, the `lidia_error_handler` will be invoked.

`void` v`.reset ()`

deallocates the vector elements and sets v`.capacity` and v`.size` to zero.

`void` v`.kill ()`

deallocates the vector elements and sets v`.capacity` and v`.size` to zero.

Access to the `exp_ratio`

`float` v`.exp_ratio () const`

returns the expansion ratio of v.

`float` v`.get_exp_ratio () const`

returns the expansion ratio of v.

`void` v`.set_exp_ratio (float` e`)`

sets the expansion ratio of v to e. The value of e must be at least 1.0. If e is invalid, the expansion ratio of v will be set to 1.0.

Access to the mode

`char` v`.get_mode () const`

returns the mode of v.

`void` v`.set_mode (char` *mode*`)`

sets mode of v to *mode* which must be either of the two constants `FIXED` or `EXPAND`. If *mode* differs from these values, the mode of v will be set to `FIXED`.

Access to an element

`T & operator[] (lidia_size_t` i`)`

If the mode of v is `FIXED`, the operator returns a reference to the $(i+1)$-st element in v, provided that i is less than the capacity of the vector. If i is too large, the `lidia_error_handler` will be invoked. The size of the vector will be set appropriately after a successful access (i.e. it will be set to $i+1$ if this value is greater than the previous size).

If v is of expanding size, the capacity of the vector will be extended if i is not less than its current value. This is done by a call of the function v`.set_capacity()`. The number of elements that will be allocated is $(i+1) \cdot v$`.exp_ratio`. Provided that the memory reallocation succeeds, the operator will return a reference to the $(i+1)$-st element in v. The size of the vector will be set appropriately.

`const T & operator[] (lidia_size_t` i`) const`

returns a constant reference to the $(i+1)$-st element in v where i has to be non-negative and less than v`.size`. Otherwise the `lidia_error_handler` will be invoked.

`const T & member (lidia_size_t` i`) const`

returns a constant reference to the $(i+1)$-st element in v where i has to be non-negative and less than v`.size`. Otherwise the `lidia_error_handler` will be invoked.

Access to the array of datas

`void` v`.set_data (const T *` d`, lidia_size_t` l`))`

copies l elements $d[0], \ldots, d[l-1]$ to vector v and adjusts its capacity. If array d has less than l elements the behaviour of this function is undefined.

`T *` v`.get_data () const`

returns a pointer to a copy of the vector elements.

Assignments

Let v be of type `base_vector< T >`. The operator `=` is overloaded. For efficiency reasons the following functions are also implemented:

`void assign (base_vector< T > &` v`, const base_vector< T > &` w`)`

allocates w`.size` elements for v using v`.set_capacity()` and copies the entries of w into v.

`void v.assign (lidia_size_t` at`, const base_vector< T > &` w`, lidia_size_t` *from*`,`
`lidia_size_t` *to*`)`

copies the elements $w[from], \ldots, w[to]$ to vector v, starting at position $v[at]$. The corresponding elements of v will be overwritten. The size of v will be adjusted appropriately.

`void` v`.fill (const T &` x`)`

$v[i] \leftarrow x$, $0 \le i <$ v`.size()`.

Reverse Functions

`void` v`.reverse ()`

rearranges the elements of vector v in reverse order.

`void` *v*`.reverse (const base_vector< T > &` *w*`)`

copies the elements of *w* in reverse order into vector *v*. If *w* does not alias *v*, *w*`.size` elements will be allocated for *v* by using *v*`.set_capacity()`.

Swap Functions

`void swap (base_vector< T > &` *v*`, base_vector< T > &` *w*`)`

exchanges the complete information of *v* and *w*.

Concatenation

`void` *v*`.concat (const base_vector< T > u, const base_vector< T > w)`

stores in *v* the concatenation of *u* and *w*. The capacity of *v* will be set to the sum of the sizes of the two vectors *u* and *w*.

Shift Functions

`void` *v*`.shift_left (lidia_size_t` *pos*`, lidia_size_t` *num*`)`

moves the elements of *v* *num* positions to the left starting at *v*[*pos*] thus overwriting some preceeding entries. The size of *v* will be reduced by *num* using the function *v*`.set_size()`. The value of *num* may not exceed *pos*.

`void` *v*`.shift_right (lidia_size_t` *pos*`, lidia_size_t` *num*`)`

copies the elements of *v* *num* positions to the right starting at *v*[*pos*] possibly overwriting some succeeding elements in *v*. The size of *v* will be increased by *num* using the function *v*`.set_size()`.

Insert Functions

`void` *v*`.insert_at (const T &` *x*`, int` *pos*`)`

extends the size of vector *v* by 1 using *v*`.set_size()`, moves the elements of *v* one position to the right starting at *v*[*pos*] and sets *v*[*pos*] $\leftarrow x$. If $pos < 0$ the `lidia_error_handler` will be invoked.

Remove Functions

`void remove_from (lidia_size_t` *pos*`, lidia_size_t` *n* `= 1)`

removes *n* elements from the vector *v* starting at *v*[*pos*]. Any element in *v* succeeding *v*[*pos*] will be moved *n* positions to the left. After a successfull deletion, the size of vector *v* will be diminshed by *n* using *v*`.set_size()`.

Input/Output

The `istream` operator `>>` and the `ostream` operator `<<` are overloaded. The `istream` operator `>>` reads vectors of the form "$[v_0 \ldots v_{n-1}]$" from an input stream.

The `ostream` operator `<<` outputs the vector in the above described format.

See also

`file_vector`, `math_vector`, `sort_vector`

Notes

1. As described in the template introduction (see page 11) for using an instance of type `base_vector< T >` the type `T` has to have at least
   - a swap function `void swap(T &, T&)`,
   - the input operator `>>`,
   - the output operator `<<` and
   - the assignment operator `=`.
2. As usually in C++ the numbering of the elements in a `base_vector` starts with zero.

Examples

```
#include <LiDIA/base_vector.h>

int main()
{
    base_vector < int > u, v, w ;

    cout << " u = " ;
    cin >> u ;

    cout << " v = " ;
    cin >> v ;

    w.concat(u,v);

    cout << w ;
}
```

For further examples please refer to `LiDIA/src/templates/vector/vector_appl.cc`

Author

Frank Lehmann, Markus Maurer, Stefan Neis, Thomas Papanikolaou, Patrick Theobald

math_vector

Name

`math_vector< T >` ..... vectors with basic mathematical operations

Abstract

`math_vector< T >` is a class for doing simple mathematical operations over vectors such as vector addition or the computation of the inner product.

According to the template introduction (see page 11) the vector class `math_vector< T >` is derived from `base_vector< T >`. So you can apply all the functions and operators of class `base_vector< T >` to instances of the type `math_vector< T >`. `T` is allowed to be either a built-in type or a class.

Description

A variable of type `math_vector< T >` contains the same components as `base_vector< T >`.

Constructors/Destructor

Each of the following constructors can get an additional parameter (`FIXED` or `EXPAND`) to indicate the mode of the `math_vector`.

`ct math_vector< T > ()`
constructs a vector with capacity 0.

`ct math_vector< T > (lidia_size_t` c`)`
constructs a vector with capacity c initialized with values generated by the default constructor for type `T`.

`ct math_vector< T > (lidia_size_t` c`, lidia_size_t` s`)`
constructs a vector with capacity c and size s initialized with values generated by the default constructor for type `T`.

`ct math_vector< T > (const math_vector< T > &` w`)`
constructs a vector with capacity $w.size$ initialized with the elements of w.

`ct math_vector< T > (const T *` a`, lidia_size_t` l`)`
constructs a vector with capacity l and size l initialized with the first l elements of the array a.

`dt ~math_vector< T > ()`

Arithmetical Operations

The class `math_vector< T >` supports the following operators:

unary		*op*	`math_vector< T >`	*op* $\in \{-\}$
binary	`math_vector< T >`	*op*	`math_vector< T >`	*op* $\in \{+, -, *\}$
binary with assignment	`math_vector< T >`	*op*	`math_vector< T >`	*op* $\in \{+=, -=\}$
binary	`math_vector< T >`	*op*	`T`	*op* $\in \{+, -, *, /\}$
binary	`T`	*op*	`math_vector< T >`	*op* $\in \{+, -, *\}$
binary with assignment	`math_vector< T >`	*op*	`T`	*op* $\in \{+=, -=, *=, /=\}$

Componentwise combination of two vectors requires that they are both of the same size. If this requirement is not met during a call of these operators, the `lidia_error_handler` will be invoked. If the result vector has not the right size it will be adapted automatically.

Addition

`void add (math_vector< T > & `r`, const math_vector< T > & `v`, const math_vector< T > & `w`)`
adds two `math_vectors` v and w componentwise and stores the result in r.

`void add (math_vector< T > & `r`, const math_vector< T > & `v`, const T & `α`)`
adds α from the right to each component of `math_vector` v and stores the result in r.

`void add (math_vector< T > & `r`, const T & `α`, const math_vector< T > & `v`)`
adds α from the left to each component of `math_vector` v and stores the result in r.

Subtraction

`void subtract (math_vector< T > & `r`, const math_vector< T > & `v`,`
`const math_vector< T > & `w`)`
subtracts `math_vector` w from v componentwise and stores the result in r.

`void subtract (math_vector< T > & `r`, const math_vector< T > & `v`, const T & `α`)`
subtracts α from each component of `math_vector` v and stores the result in r.

`void subtract (math_vector< T > & `r`, const T & `α`, const math_vector< T > & `v`)`
subtracts successively each component of `math_vector` v from α and stores the result in r.

Multiplication

`void multiply (T & `r`, const math_vector< T > & `v`, const math_vector< T > & `w`)`
stores the inner product of v and w in r.

`void multiply (math_vector< T > & `r`, const math_vector< T > & `v`, const T & `α`)`
multiplies each component of `math_vector` v from the right by α and stores the result in r.

`void multiply (math_vector< T > & r, const T & α, const math_vector< T > & v)`
multiplies each component of `math_vector` v from the left by α and stores the result in r.

`void compwise_multiply (math_vector< T > & r, const math_vector< T > & v, const math_vector< T > & w)`
multiplies two `math_vectors` v and w componentwise and stores the result in r.

Division

`void divide (math_vector< T > & r, const math_vector< T > & v, const T & α)`
divides each component of `math_vector` v by α and stores the result in r.

`void compwise_divide (math_vector< T > & r, const math_vector< T > & v, const math_vector< T > & w)`
divides `math_vector` v by w componentwise and stores the result in r.

Negation

`void negate (math_vector< T > & r, const math_vector< T > v)`
negates the `math_vector` v componentwise and stores the result in r.

Sum of squares

`T v.sum_of_squares ()`
returns the sum of the squares of elements of vector v.

`T sum_of_squares (const math_vector< T > & v)`
returns the sum of the squares of elements of vector v.

Comparisons

The binary operators `==` and `!=` are overloaded and can be used for componentwise comparison. Let v be an instance of type `math_vector< T >`.

`bool v.equal (const math_vector< T > & w) const`
returns `true` if the vectors v and w are equal in each component, `false` otherwise.

`bool equal (const math_vector< T > & v, const math_vector< T > & w)`
returns `true` if the vectors v and w are equal in each component, `false` otherwise.

`bool v.unequal (const math_vector< T > & w) const`
returns `true` if the two vectors v and w differ in at least one component, `false` otherwise.

`bool unequal (const math_vector< T > & v, const math_vector< T > & w)`
returns `true` if the two vectors v and w differ in at least one component, `false` otherwise.

See also

`base_vector`, `file_vector`, `sort_vector`

Notes

1. As described in the template introduction (see page 11) for using an instance of type `math_vector< T >` the type `T` has to have at least
   - a swap function `void swap(T &, T&)`,
   - the input operator `>>`,
   - the output operator `<<`,
   - the assignment operator `=`,
   - the multiplication operator `*`,
   - the addition operator `+`,
   - the division operator `/`,
   - the subtraction operator `-`,
   - the unary minus `-` and
   - the equal operator `==`.
2. Beware of the use of expanding vectors. Unpredictable errors might occur in situations like the following. Consider the statement

   ```
   add ( v[2] , v[1] , a );
   ```

   for a vector v with capacity 2. If the parameters of that call are evaluated right to left, the address of v[1] will be stored before the access to v[2] will cause a memory reallocation for v. The latter might change the address of v[1] and thus invalidate the previously stored value.

Examples

```
#include <LiDIA/math_vector.h>

int main()
{
    math_vector < int > u, v, w ;

    cout << " u = " ;
    cin >> u ;

    cout << " v = " ;
    cin >> v ;

    w = u + v;
    cout << w << u - w << (v + w)*10 << flush;
}
```

For further examples please refer to `LiDIA/src/templates/vector/vector_appl.cc`

Author

Frank Lehmann, Markus Maurer, Stefan Neis, Thomas Papanikolaou, Patrick Theobald

sort_vector

Name

`sort_vector< T >` ..... vectors with sort functions

Abstract

Class `sort_vector< T >` contains `base_vector< T >` as a base class. It supports any member and friend function of that class. Additionally it provides functions for sorting and searching, inserting and deleting. `T` can be either a built-in type or a class.

Description

A variable of type `sort_vector< T >` contains the same components as a variable of type `base_vector< T >`. In addition there is a character `sort_dir` and a function pointer `*el_cmp`.

A vector of type `sort_vector< T >` can be sorted according to a certain sort direction. This class provides two standard directions `SORT_VECTOR_UP` and `SORT_VECTOR_DOWN` to sort the elements of a vector in ascending or descending order, respectively. Furthermore, specific compare functions may be used to define a relation of elements of type `T`. These functions define a new sort relation for a vector.

Any compare function used must have a prototye like

`int cmp ( const T &` a`, const T &` b `)` .

The data type for pointers to functions of that type will be called `CMP_FUNC` in the sequel. The return value of a comparison shall be as follows:

< 0	if `(a, b)` is in correct order
$= 0$	if `a, b` are identical
> 0	if `(a, b)` is in inverted order

Any vector v of type `sort_vector< T >` contains a default sort direction according to which it will be sorted if the function v`.sort()` is applied. This default sort direction will be set to `SORT_VECTOR_UP` by any constructor in this class; i.e. vector v will be sorted in ascending order by default. This default value may later be changed for a vector v by using the function v`.set_sort_direction()`.

Constructors/Destructor

Each of the following constructors can get an additional parameter (`FIXED` or `EXPAND`) to indicate the mode of the `sort_vector`.

`ct sort_vector< T > ()`

constructs a vector with capacity 0.

`ct sort_vector< T > (lidia_size_t` *c*`)`

constructs a vector with capacity *c* initialized with values generated by the default constructor for type `T`.

`ct sort_vector< T > (lidia_size_t` *c*`, lidia_size_t` *s*`)`

constructs a vector with capacity *c* and size *s* initialized with values generated by the default constructor for type `T`.

`ct sort_vector< T > (const sort_vector< T > &` *w*`)`

constructs a vector with capacity *w.size* initialized with the elements of *w*.

`ct sort_vector< T > (const T *` *a*`, lidia_size_t` *l*`)`

constructs a vector with capacity *l* and size *l* initialized with the first *l* elements of the array *a*.

`dt ~sort_vector< T > ()`

Sort direction

`char` *v*`.sort_direction () const`

returns the default sort direction of vector *v*. The return value will be one of the predefined constants `SORT_VECTOR_UP`, `SORT_VECTOR_UP` or `SORT_VECTOR_CMP`. The latter indicates that the sort direction of *v* has been set to a certain compare function.

`char` *v*`.get_sort_direction () const`

returns the default sort direction of vector *v*. The return value will be one of the predefined constants `SORT_VECTOR_UP`, `SORT_VECTOR_DOWN` or `SORT_VECTOR_CMP`. The latter indicates that the sort direction of *v* has been set to a certain compare function.

`void` *v*`.set_sort_direction (char` *dir*`)`

sets the default sort-direction for *v* to *dir*. The value of *dir* may be either one of the two predefined constants `SORT_VECTOR_UP` or `SORT_VECTOR_DOWN`. If *dir* is invalid, the standard sort direction of *v* will be left unchanged and the `lidia_error_handler` will be invoked.

`void` *v*`.set_sort_direction (int (*cmp) (const T &` *a*`, const T &` *b*`))`

sets the default sort direction for *v* in a way that function `cmp()` will be used to compare the entries of *v*.

Sort Functions

Class `sort_vector< T >` provides a sort function, which allows to sort a vector in various ways. There are three different prototypes for this sort routine that allow a variety of calls of the functions. Some of the parameters of function `sort()` can obtain default values if they are not specified in a call. These default values will be indicated by "*DEF*".

It is possible to restrict the sort routine to a part of a vector by explicitly passing on the starting and ending indices to this function. Whenever these values are invalid, the `lidia_error_handler` will be invoked.

`void` v`.sort (char sort_dir =` *DEF*`, int` l `=` *DEF*`, int` r `=` *DEF*`)`

sorts the elements $v[l], \ldots, v[r]$ according to the direction v`.sort_dir`. Any other element in v will remain in its position. The value of v`.sort_dir` has to be either one of the two constants `SORT_VECTOR_UP` or `SORT_VECTOR_DOWN`. If it differs from these values, the `lidia_error_handler` will be invoked. If no values for l and r are given, the entire vector will be sorted. If v`.sort_dir` is not specified either, the default sort direction for v will be used.

`void` v`.sort (CMP_FUNC cmp, int` l `=` *DEF*`, int` r `=` *DEF*`)`

sorts the elements $v[l], \ldots, v[r]$ according to the compare function `(*cmp)()`. Any other element in v will remain in its position. The function `(*cmp)()` must be of correct type `CMP_FUNC`. If no values for l and r are given, the entire vector will be sorted.

`void` v`.sort (int` l`, int` r`)`

sorts the elements $v[l], \ldots, v[r]$ according to the default sort direction of v.

Search Functions

As a second feature the class `sort_vector< T >` provides functions for linear and binary search in a vector.

`bool` v`.linear_search (const T &` x`, int &` *pos*`) const`

searches x in vector v. If x can be found, this function will return `true` and *pos* will hold the index of its first occurence in v. Otherwise, the return value will be `false`.

The function `bin_search()` searches for an element x in a vector using a binary search strategy. (Beware of the fact, that this routine can work successfully only if the elements of the vector are sorted in an appropriate way.) Its return value will be `true` if it succeeds, and `false` otherwise. If x can be found in a vector, *pos* will hold the index of its *first* occurence or else, *pos* will be the index, where x can be inserted according to the corresponding sort direction.

Some of the parameters of function `bin_search()` can obtain default values if they are not specified in a call. These default values will be indicated by "*DEF*".

It is possible to restrict the search to a part of a vector by explicitly passing on the starting and ending indices to this function. Whenever these values are invalid, the `lidia_error_handler` will be invoked.

`bool` v`.bin_search (const T &` x`, int &` *pos*`, char sort_dir =` *DEF*`, int` l `=` *DEF*`,`
`int` r `=` *DEF*`) const`

searches for x among the elements $v[l], \ldots, v[r]$ using the sort direction v`.sort_dir`. Its value must be either of the two constants `SORT_VECTOR_UP` or `SORT_VECTOR_DOWN`. If v`.sort_dir` differs from these values, the `lidia_error_handler` will be invoked. If no values for l and r are given, the search will be carried out on the entire vector. If v`.sort_dir` is not specified either, the default sort-direction for v will be used.

`bool` v`.bin_search (const T &` x`, int &` *pos*`, CMP_FUNC cmp, int` l `=` *DEF*`, int` r `=` *DEF*`) const`

searches for x among the elements $v[l], \ldots, v[r]$ using function `(*cmp)()` to compare these entries. The function `(*cmp)()` must be of correct type `CMP_FUNC`. If no values for l and r are given, the search will be carried out on the entire vector. If v`.sort_dir` is not specified either, the default sort direction for v will be used.

`bool` v`.bin_search (const T &` x`, int &` *pos*`, int` l`, int` r`) const`

searches for x among the elements $v[l], \ldots, v[r]$ using the default sort direction of v.

Insert Functions

The class `sort_vector< T >` also supports several functions to insert a new element into a vector. These use the member function v`.set_size()` to extend the size of a vector v by 1. If this cannot successfully be done, the vector will be left unchanged. In that case, the `lidia_error_handler` will be invoked.

An element x can either be inserted at a specified position (using the function `insert_at()` of class `base_vector< T >`) or it can also be automatically inserted at its *correct* position according to the vector's sort direction.

The function `insert()` provides several prototypes that allow to optionally specify a sort direction or a part of the vector in a call to it. If these parameters are missing, *insert*() will refer to the vector default sort direction or consider the entire vector, respectively.

Note, that a vector must have been sorted before using the function `insert()`. The sort direction used in the sort procedure must correspond to the one specified here.

`void` v`.insert (const T &` x`, char sort_dir =` *DEF*`, int` l `=` *DEF*`, int` r `=` *DEF*`)`

uses the function `bin_search()` to determine the correct position of x in v according to the specified sort direction. If l and r are specified, the search for x will be restricted to the part $v[l], \ldots, v[r]$. After x is inserted at the corresponding position, every succeeding element in v will be moved one place to the right. The value of v`.sort_dir` must be either one of the two constants `SORT_VECTOR_UP` or `SORT_VECTOR_DOWN`. If v`.sort_dir` differs from these values, the `lidia_error_handler` will be invoked.

If no values are given for l, r or `sort_dir`, the default values will be passed on to the function `bin_search()`, i.e. the search for x will be carried out on the entire vector and according to its default sort direction.

`void` v`.insert (const T &` x`, CMP_FUNC cmp, int` l `=` *DEF*`, int` r `=` *DEF*`)`

uses the function `bin_search()` to determine the correct position of x in v according to the compare function `(*cmp)()`. If l and r are specified, the search for x will be restricted to the part $v[l], \ldots, v[r]$. After x is inserted at the corresponding position, every succeeding element in v will be moved one place to the right.

If no values are given for l and r, the default values will be passed on to the function `bin_search()`, i.e. the search for x will be carried out on the entire vector.

`void` v`.insert (const T &` x`, int` l`, int` r`)`

uses the function `bin_search()` to determine the correct position of x in v according to the default sort direction of v. The search for x will be restricted to the part $v[l], \ldots, v[r]$. After x is inserted at the corresponding position, every succeeding element in v will be moved one place to the right.

Remove Functions

Furthermore, the class `sort_vector< T >` supports functions to remove elements from a vector. These functions use v`.set_size()` to deminish the size of the vector after the removal.

The function `remove()` initiates `bin_search()` first to find a given element x in a vector. If this search was successfully done, *one* occurrence of x will be removed from v. Any element in v succeeding x will be moved one position to the left and the size of the vector will be diminished by 1. If x cannot be found in the vector, v will be left unchanged and the return value will be `false`.

The function `remove()` also provides several prototypes that allow to optionally specify a sort direction or a part of the vector. If these parameters are missing, `remove()` will refer to the vector default sort direction and consider the entire vector, respectively.

`bool` v`.remove (const T &` x`, char sort_dir =` *DEF*`, int` l `=` *DEF*`, int` r `=` *DEF*`)`

uses the function `bin_search()` to find x in v using the specified sort direction. If l and r are specified, the search for x will be restricted to the part $v[l], \ldots, v[r]$. If x can be found in the vector, its first occurence will be deleted. The value of v`.sort_dir` must be one either of the two constants `SORT_VECTOR_UP` or `SORT_VECTOR_DOWN`. If v`.sort_dir` differs from these values, the `lidia_error_handler` will be invoked.

If no values are given for l, r or v`.sort_dir`, the default values will be passed on to the function `bin_search()`, i.e. the search for x will be carried out on the entire vector and according to its default sort direction.

`bool` v`.remove (const T &` x`, CMP_FUNC cmp, int` l `=` *DEF*`, int` r `=` *DEF*`)`

uses the function v`.bin_search()` to find x in v using the compare function `(*cmp)()`. If l and r are specified, the search for x will be restricted to the part $v[l], \ldots, v[r]$. If x can be found in the vector, its first occurence will be deleted.

If no values are given for l and r, the default values will be passed on to the function `bin_search()`, i.e. the search for x will be carried out on the entire vector.

`bool` v`.remove (const T &` x`, int` l`, int` r`)`

uses the function `bin_search()` to find x in v using the default sort-direction of v. The search for x will be restricted to the part $v[l], \ldots, v[r]$. If x can be found in the vector, its first occurence will be deleted.

See also

`base_vector`, `file_vector`, `math_vector`

Notes

As described in the template introduction (see page 11) for using an instance of type `sort_vector< T >` the type T has to have at least

- a swap function `void swap(T &, T&)`,
- the input operator `>>`,
- the output operator `<<`,
- the assignment operator `=` and
- the compare operators `<` and `>`.

Examples

```
#include <LiDIA/sort_vector.h>

int main()
{
    sort_vector < double > w ;

    cout << " w = " ;
    cin >> w ;

    w.sort();
```

```
        cout << w << flush;
    }
```

For further examples please refer to `LiDIA/src/templates/vector/vector_appl.cc`

Author

Frank Lehmann, Markus Maurer, Stefan Neis, Thomas Papanikolaou, Patrick Theobald

file_vector

Name

`file_vector< T >` ..... vectors with special file operations

Abstract

`file_vector< T >` is a class for doing simple file operations. `T` is allowed to be either a built-in type or a class.

According to the template introduction the class `file_vector< T >` is derived from the class `base_vector< T >`. So you can apply all the functions and operators of class `base_vector< T >` to instances of the type `file_vector< T >`.

Description

A variable of type `file_vector< T >` contains the same components as a variable of type `base_vector< T >`.

Constructors/Destructor

Each of the following constructors can get an additional parameter (`FIXED` or `EXPAND`) to indicate the mode of the `file_vector`.

`ct file_vector< T > ()`
: constructs a vector with capacity 0.

`ct file_vector< T > (lidia_size_t` c`)`
: constructs a vector with capacity c initialized with values generated by the default constructor for type `T`.

`ct file_vector< T > (lidia_size_t` c`, lidia_size_t` s`)`
: constructs a vector with capacity c and size s initialized with values generated by the default constructor for type `T`.

`ct file_vector< T > (const file_vector< T > &` w`)`
: constructs a vector with capacity `w.size` initialized with the elements of w.

`ct file_vector< T > (const T *` a`, lidia_size_t` l`)`
: constructs a vector with capacity l and size l initialized with the first l elements of the array a.

`dt ~file_vector< T > ()`

Input/Output

In the sequel `fp` must always be a file pointer to a file that has already been opened in the appropriate mode (e.g. a non-zero pointer that has been returned by `fopen()`).

`void` v`.print_to_file (FILE *fp) const`

writes a vector in text format to the file, which `fp` is pointing to. The output is of the form "$[v_0 \dots v_n - 1]$", where n is the size of v. This function requires the member function `print_to_file()` for type `T`.

`void` v`.scan_from_file (FILE *fp)`

reads a vector in text format from the file, which `fp` is pointing to. The vector must be stored as "$[v_0 \dots v_n - 1]$". The size and capacity of v will then be set to n. If v can not successfully be read, the `lidia_error_handler` will be invoked. This function requires the member function `scan_from_file()` for type `T`.

`void` v`.write_to_file (FILE *fp) const`

writes a vector in binary format to the file, which `fp` is pointing to. First, the length of v is written to the file. Afterwards, the elements of v are successively stored to `fp` by using the member function `write_to_file()` for elements of type `T`.

`void` v`.read_from_file (FILE *fp)`

reads a vector in binary format from the file, which `fp` is pointing to. The function starts by reading the length of a vector at the current file pointer position. Afterwards, the corresponding number of elements is allocated for v by using the function `v.set_capacity()`. Then, the elements are successively read into v. If v can not successfully be read, the `lidia_error_handler` will be invoked. This function requires the member function `read_from_file()` for type `T`.

The functions read and write from the current file position the pointer is pointing to. The file pointer will *not* be reset after the reading or writing process.

`void` v`.append_a (FILE *fp) const`

appends the elements of vector v to another vector that is stored at the end of a text file. If `fp` indicates the beginning of a vector, this one will be extended by the elements of v. If `fp` is pointing to the end of a file, this function is equivalent to `print_to_file()`. The file-pointer must allow both reading and writing operations. It will, at the end, reside in the same position as prior to the function call. If v can not successfully be written to the file, the `lidia_error_handler` will be invoked.

`void` v`.append_a (FILE *fp, lidia_size_t` n`) const`

appends the $n+1$-st element of v to another vector that is stored at the end of a text file. If `fp` indicates the beginning of a vector, this one will be extended by $v[n]$. If `fp` specifies the end of a file, this function will add a vector of size 1 to it. The file pointer must allow both reading and writing operations. It will, at the end, reside in the same position as prior to the function call. If v can not successfully be written to the file, the `lidia_error_handler` will be invoked.

`void` v`.append_b (FILE *fp) const`

appends the elements of vector v to another vector that is stored at the end of a binary file. If `fp` indicates the beginning of a vector, this one will be extended by the elements of v. If `fp` specifies the end of a file, this function is equivalent to `write_to_file()`. The file pointer must allow both reading and writing operations. It will, at the end, reside in the same position as prior to the function call. If v can not successfully be written to the file, the `lidia_error_handler` will be invoked.

`int` v`.append_b (FILE *fp, lidia_size_t` n`) const`

appends the $n+1$-st element of v to another vector that is stored at the end of a binary file. If `fp` indicates the beginning of a vector, this one will be extended by $v[n]$. If `fp` specifies the end of a file, this function will add a vector of size 1 to it. The file pointer must allow both reading and writing operations. It will, at the end, reside in the same position as prior to the function call. If v can not successfully be written to the file, the `lidia_error_handler` will be invoked.

Remark

The read functions for type `T` should work in a way such that they read the whole stuff which has previously been written by the corresponding write function, e.g. a "`\n`" appended to a type `T` element by the print function must be read by the scan function.

See also

`base_vector`, `math_vector`, `sort_vector`

Notes

As described in the template introduction (see page 11) for using an instance of type `file_vector< T >` the type `T` has to have at least

- a swap function `void swap(T &, T &)`,
- the input operator `>>`,
- the output operator `<<`,
- the assignment operator `=`,
- the function `print_to_file(fp)` (i.e. writing in text-format),
- the function `write_to_file(fp)` (i.e. binary-writing),
- the function `scan_from_file(fp)` (i.e. reading in text-format) and
- the function `read_from_file(fp)` (i.e. binary-writing), where `fp` is of type `FILE *`.

Examples

```
#include <LiDIA/file_vector.h>

int main()
{
    file_vector < double > v;
    cin >> v;

    FILE *dz = fopen("LiDIA-OUTPUT", "w");
    if (dz == NULL)
    {
        cout << "ERROR !! " << flush;
        return 1;
    }
    v.write_to_file(dz);
```

```
        fclose(dz);
        return 0;
    }
```

For further examples please refer to `LiDIA/src/templates/vector/vector_appl.cc`

Author

Frank Lehmann, Markus Maurer, Stefan Neis, Thomas Papanikolaou, Patrick Theobald

lidia_vector

Name

`lidia_vector< T >` ..... parameterized vector-class

Description

The class `lidia_vector< T >` is a combination of the classes `math_vector`, `sort_vector`, and `file_vector`. That means that it puts at disposal all functions of its base classes. Obviously, this derived class requires the same functions for type `T` as the corresponding base classes do.

See also

`base_vector`, `file_vector`, `math_vector`, `sort_vector`

Examples

```
#include <LiDIA/lidia_vector.h>

int main()
{
    lidia_vector < int > u;

    cout << " u = " ;
    cin >> u ;

    u.sort();
    cout << "u = " << u << flush;

    u -= (u*(int)100);
    cout << "u = " << u << flush;

    return 0;
}
```

For further examples please refer to `LiDIA/src/templates/vector/vector_appl.cc`

Author

Frank Lehmann, Markus Maurer, Stefan Neis, Thomas Papanikolaou, Patrick Theobald

hash_table

Name

`hash_table< T >` ..... parameterized hash table class

Abstract

The class `hash_table< T >` represents a hash table consisting of elements which are of some data type `T`. This base type is allowed to be either a built-in type or a class, and its only requirements are that the assignment operator `=`, equality operator `==`, and input/output operators `>>` and `<<` are defined. A function returning a key value for an instance of type `T` must be defined by the user.

Description

An instance of `hash_table< T >` consists of an array of n linked lists whose elements each contain a pointer to an instance of type `T`. The collision resolution scheme is bucketing, i.e., elements which hash to the same location are simply appended to the appropriate linked list (bucket).

Before inserting elements into the hash table, it must be initialized with the member function `initialize`. The number of buckets in the table, n, will be set to the smallest prime number greater than or equal to the parameter passed to `initialize`.

The elements are ordered according to some key function which is defined by the user. This key function must have a prototype like

`bigint get_key (const T & G)`.

The value returned by this function is reduced modulo n to give a valid hash value between 0 and $n-1$, where n is the number of buckets. The key function must be set with the member function `set_key_function`, and should not be changed after elements have been inserted into the table. Note that equality of elements is determined using the `==` operator corresponding to type `T`, and that it is not necessary for equality of key values to imply equality of elements.

Constructors/Destructor

`ct hash_table< T > ()`

`dt ~hash_table< T > ()`

Initialization

Let HT be of type `hash_table< T >`. A `hash_table< T >` must be initialized before use.

`void` HT`.initialize (const lidia_size_t` *table_size*`)`
initializes a `hash_table< T >` with n buckets, where n is the smallest prime number $\geq$ *table_size*. Any subsequent calls to this function will cause the elements in HT to be deleted, and the number of buckets to be reset.

`void` HT`.set_key_function (bigint (*get_key) (const T &` G`))`
sets the key function. The parameter must be a pointer to a function which accepts one parameter of type `T` and returns a `bigint`.

Assignments

Let HT be of type `hash_table< T >`. The operator `=` is overloaded. For efficiency, the following function is also implemented:

`void` HT`.assign (const hash_table< T > &` HT'`)`
HT is assigned a copy of HT'.

Access Methods

Let HT be of type `hash_table< T >`.

`lidia_size_t` HT`.no_of_elements () const`
returns the number of elements currently contained in HT.

`lidia_size_t` HT`.no_of_buckets () const`
returns the number of buckets of HT.

`void` HT`.remove (const T &` G`)`
if G is in the table, it is removed.

`void` HT`.empty ()`
deallocates all elements in HT. The number of buckets remains the same; to change this, reinitialize using `initialize`.

`const T &` HT`.last_entry () const`
returns a constant reference to the last item added to the table. If the table is empty, the `lidia_error_handler` will be evoked.

`void` HT`.hash (const T &` G`)`
adds the item G to the hash table.

`T *` HT`.search (const T &` G`) const`
returns a pointer to the first occurrence of an element in HT that is equal to G according to the `==` operator corresponding to type `T`. If no such element exists, the `NULL` pointer is returned.

Input/Output

Let HT be of type `hash_table< T >`. The `istream` operator `>>` and the `ostream` operator `<<` are overloaded. The input of a `hash_table< T >` consists of n, a `lidia_size_t` representing the number of buckets, m, a second `lidia_size_t` (on a separate line) representing the number of elements in the table, and exactly m values of type `T`, one per line. The output consists of each non-empty bucket being displayed element by element, one bucket per line.

See also

`indexed_hash_table`

Notes

- The base type T must have at least the following capabilities:
  - the assignment-operator `=`,
  - the equality-operator `==`,
  - the input-operator `>>`, and
  - the output-operator `<<`.
- The enumeration of the buckets in a `hash_table` starts with zero (like C++)

Examples

```
#include <LiDIA/hash_table.h>

bigint
ikey(const int & G) { return bigint(G); }

int main()
{
    hash_table < int > HT;
    int i, x, *y;

    HT.initialize(100);
    HT.set_key_function(&ikey);

    cout << "#buckets = " << HT.no_of_buckets() << endl;
    cout << "current size = " << HT.no_of_elements() << "\n\n";

    for (i = 0; i < 4; ++i) {
        cout << "Enter an integer: ";
        cin >> x;
        HT.hash(x);
    }

    cout << "\ncurrent size = " << HT.no_of_elements() << endl;
    x = HT.last_entry();
    cout << "last entry = " << x << endl;
    y = HT.search(x);
    if (y)
        cout << "search succeeded!  Last entry is in the table.\n";
    else
```

```
            cout << "search failed!  Please report this bug!\n";
        cout << "Contents of HT:\n";
        cout << HT << endl;

        return 0;
    }
```

Example:

```
#buckets = 101
current size = 0

Enter an integer: 5
Enter an integer: 12345
Enter an integer: -999
Enter an integer: 207

current size = 4
last entry = 207
search succeeded!  Last entry is in the table.
Contents of HT:
[5] : 5 : 207
[11] : -999
[23] : 12345
```

For further examples please refer to `LiDIA/src/templates/hash_table/hash_table_appl.cc`.

Author

Michael J. Jacobson, Jr.

indexed_hash_table

Name

`indexed_hash_table< T >` ..... hash table with sequential access capability

Abstract

The class `indexed_hash_table< T >` represents a hash table consisting of elements which are of some data type `T` in exactly the same way as the class `hash_table< T >`. However, this class also allows access to the elements in the table by index, i.e., a regular sequential table.

Description

An instance of a `indexed_hash_table< T >` consists of a `hash_table< T >` together with a list of pointers to elements in the `hash_table< T >`. The initialization of the table and definition of the key function work the same as in the class `hash_table< T >`.

Constructors/Destructor

`ct indexed_hash_table< T > ()`

`dt ~indexed_hash_table< T > ()`

Initialization

Let IHT be of type `indexed_hash_table< T >`. The initialization function and key setting function are exactly the same as those of the class `hash_table< T >`.

`void` IHT`.initialize (const lidia_size_t table_size)`

`void` IHT`.set_key_function (bigint (*get_key) (const T &` G`))`

Assignments

Let IHT be of type `indexed_hash_table< T >`. The operator `=` is overloaded. For efficiency, the following function is also implemented:

`void` IHT`.assign (const hash_table< T > &` IHT'`)`

IHT is assigned a copy of IHT'.

Access Methods

Let *IHT* be of type `indexed_hash_table< T >`. The following functions are exactly the same as those of the class `hash_table< T >`.

`lidia_size_t` *IHT*`.no_of_buckets () const`

`lidia_size_t` *IHT*`.no_of_elements () const`

`void` *IHT*`.remove (const T &` G`)`

`void` *IHT*`.empty ()`

`const T &` *IHT*`.last_entry () const`

`void` *IHT*`.hash (const T &` G`)`

`T *` *IHT*`.search (const T &` G`) const`

The following functions are specific to the class `indexed_hash_table< T >`, and are related to the sequential access capability.

`const T & operator[] (lidia_size_t` i`) const`

returns a constant reference to element i in the hash table (zero denotes the first element). If i is greater than or equal to the number of elements in the table or less than 0, the `lidia_error_handler` will be evoked.

`const T &` *IHT*`.member (lidia_size_t` i`) const`

returns a constant reference to element i in the hash table (zero denotes the first element). If i is greater than or equal to the number of elements in the table or less than 0, the `lidia_error_handler` will be evoked.

`void` *IHT*`.remove_from (const lidia_size_t` i`)`

deletes element i from the hash table. If i is greater than or equal to the number of elements in the table or less than 0, the `lidia_error_handler` will be evoked.

Input/Output

Let *IHT* be of type `indexed_hash_table< T >`. The `istream` operator `>>` and the `ostream` operator `<<` are overloaded. Input works in the same way as `hash_table< T >`. Output is done either sequentially, each element in the table printed sequentially one element per line, or in the same way as the class `hash_table< T >`. The following function may be used to set the output style for a particular instance of `indexed_hash_table< T >`.

`void` *IHT*`.output_style (const int` *style*`)`

sets the output style for *IHT*. If *style* = 0, sequential output will be used. If *style* = 1, output will be done in the same style as `hash_table< T >`. The default is sequential output.

See also

`hash_table`

Notes

- The base type `T` must have at least the following capabilities:
  - the assignment-operator `=`,
  - the equality-operator `==`,
  - the input-operator `>>`, and
  - the output-operator `<<`.
- The enumeration of the buckets and the elements both start with zero (like C++)

Examples

```
#include <LiDIA/indexed_hash_table.h>

bigint
ikey(const int & G) { return bigint(G); }

int main()
{
    indexed_hash_table < int > IHT;
    int i, x, *y;

    IHT.initialize(100);
    IHT.set_key_function(&ikey);

    cout << "#buckets = " << IHT.no_of_buckets() << endl;
    cout << "current size = " << IHT.no_of_elements() << "\n\n";

    for (i = 0; i < 4; ++i) {
        cout << "Enter an integer: ";
        cin >> x;
        IHT.hash(x);
    }

    cout << "\ncurrent size = " << IHT.no_of_elements() << endl;
    x = IHT.last_entry();
    cout << "last entry = " << x << endl;
    y = IHT.search(x);
    if (y)
        cout << "search succeeded!  Last entry is in the table.\n";
    else
        cout << "search failed!  Please report this bug!\n";

    cout << "\nsequential access:\n";
    for (i=0; i<3; ++i)
    cout << "element " << i << " = " << IHT[i] << endl;

    cout << "\nContents of IHT:\n";
    cout << IHT << endl;

    IHT.output_style(1);
    cout << "\nAs regular hash table:\n";
    cout << IHT << endl;

    return 0;
}
```

Example:

```
#buckets = 101
current size = 0

Enter an integer: 5
Enter an integer: 12345
Enter an integer: -999
Enter an integer: 207

current size = 4
last entry = 207
search succeeded!  Last entry is in the table.

sequential access:
element 0 = 5
element 1 = 12345
element 2 = -999

Contents of IHT:
0: 5
1: 12345
2: -999
3: 207


As regular hash table:
[5] : 5 : 207
[11] : -999
[23] : 12345
```

For further examples please refer to `LiDIA/src/templates/hash_table/indexed_hash_table_appl.cc`.

Author

Michael J. Jacobson, Jr.

Chapter 5

Matrices

Please note that the LiDIA base package only contains the general template matrix classes. Classes which deal with the more specialized linear algebra over the ring of rational integers and over $\mathbb{Z}/m\mathbb{Z}$ can be found in the LiDIA LA package (see page 331).

base_matrix

Name

`base_matrix< T >` ..... parameterized base matrix class

Abstract

The class `base_matrix< T >` allows to order elements of type `T` which is allowed to be either a built-in type or a class in form of a matrix.

$$A = (a_{i,j}) \in T^{m \times n} \ ,$$

where $0 \leq i < m$ and $0 \leq j < n$. It allows to create matrices and offers elementary access functions.

Description

A variable of type `base_matrix< T >` consists of a 2-dimensional array `T** value` and two elements (`rows, columns`) of type `lidia_size_t` which store the number of rows and columns of the matrix, respectively. Additionally a variable of type `base_matrix< T >` contains two unsigned integers (`print_mode`, (`print_mode`). The member `print_mode` determines in which format the matrix will be printed. The member `lattice_mode` stores informations used by the lattice classes.

In the following descriptions we use A`.rows` to label the number of rows and A`.columns` to label the number of columns of the matrix A.

Constructors/Destructor

If $a \leq 0$ or $b \leq 0$ or $v =$ `NULL` in one of the following constructors, the `lidia_error_handler` will be invoked.

`ct base_matrix< T > ()`
constructs a 1×1 matrix initialized with values generated by the default constructor of type `T`.

`ct base_matrix< T > (lidia_size_t` a`, lidia_size_t` b`)`
constructs an $a \times b$ matrix initialized with values generated by the default constructor of type `T`.

`ct base_matrix< T > (lidia_size_t` a`, lidia_size_t` b`, const T **` v`)`
constructs an $a \times b$ matrix initialized with the values of the 2-dimensional array v. The behaviour of this constructor is undefined if the array v has less than a rows or less than b columns.

`ct base_matrix< T > (const base_matrix< T > &` A`)`
constructs a copy of matrix A.

`dt ~base_matrix< T > ()`

Assignments

Let A be of type `base_matrix< T >`. The operator `=` is overloaded. For efficiency reasons, the following functions are also implemented:

`void` A`.assign (const base_matrix< T > &` B`)`

`void assign (base_matrix< T > &` A`, const base_matrix< T > &` B`)`

sets the dimensions of matrix A to the dimensions of matrix B and copies each entry of B to the corresponding position in A.

Access Methods

Let A be of type `base_matrix< T >`. Note that the numbering of columns and rows starts with zero.

Access to the number of rows

`lidia_size_t` A`.get_no_of_rows () const`

returns the number of rows of matrix A.

`void` A`.set_no_of_rows (lidia_size_t` k`)`

changes the number of rows of matrix A into k. If $k < A$`.rows`, the rows $k, \ldots, (A$`.rows` $- 1)$ are discarded. If $k > A$`.rows`, this function adds $k - A$`.rows` new rows initialized with values generated by the default constructor of type `T`. The `lidia_error_handler` will be invoked if $k < 1$.

Access to the number of columns

`lidia_size_t` A`.get_no_of_columns () const`

returns the number of columns of matrix A.

`void` A`.set_no_of_columns (lidia_size_t` k`)`

changes the number of columns of matrix A into k. If $k < A$`.columns`, then the columns $i, \ldots, (A$`.columns` $- 1)$ are discarded. If $k > A$`.columns`, then this function adds $k - A$`.columns` new columns initialized with values generated by the default-constructor for type `T`. The `lidia_error_handler` will be invoked if $k < 1$.

Access to the number of rows and columns

`void` A`.resize (lidia_size_t` i`, lidia_size_t` j`)`

changes the number of rows into i and the number of columns of matrix A into j by calling the functions A`.set_no_of_rows(`i`)` and A`.set_no_of_columns(`j`)`. The `lidia_error_handler` is invoked if $i < 1$ or $j < 1$.

`void` A`.reset ()`

changes the number of columns and number of rows of the matrix A to 1.

Note: Memory is deallocated.

`void` A`.kill ()`

changes the number of columns and number of rows of the matrix A to 1.

Note: Memory is deallocated.

Access to a row

`base_vector< T > A.operator[] (lidia_size_t i) const`

returns a `base_vector` v created by copying the values of row i. If $i \geq A$.`rows` or $i < 0$ the `lidia_error_handler` will be invoked. Note, that the size of the vector that is returned has v.`size` $= A$.`columns` and v.`capacity` $= A$.`columns`.

`T * A.row (lidia_size_t i) const`

allocates memory and returns a copy of row i. If $i \geq A$.`rows` or $i < 0$, the `lidia_error_handler` will be invoked.

`T * A.get_row (lidia_size_t i) const`

allocates memory and returns a copy of row i. If $i \geq A$.`rows` or $i < 0$, the `lidia_error_handler` will be invoked.

`void A.row (T * v, lidia_size_t i) const`

copies the values of row i to array v. If $i \geq A$.`rows` or $i < 0$ or $v =$ `NULL`, the `lidia_error_handler` will be invoked. The behaviour of this function is undefined if the array v has less than A.`columns` elements.

`void A.get_row (T * v, lidia_size_t i) const`

copies the values of row i to array v. If $i \geq A$.`rows` or $i < 0$ or $v =$ `NULL`, the `lidia_error_handler` will be invoked. The behaviour of this function is undefined if the array v has less than A.`columns` elements.

`base_vector< T > A.row_vector (lidia_size_t i) const`

returns a `base_vector` v created by copying the values of row i. If $i \geq A$.`rows` or $i < 0$ the `lidia_error_handler` will be invoked. Note, that the size of the vector that is returned has v.`size` $= A$.`columns` and v.`capacity` $= A$.`columns`.

`base_vector< T > A.get_row_vector (lidia_size_t i) const`

returns a `base_vector` v created by copying the values of row i. If $i \geq A$.`rows` or $i < 0$ the `lidia_error_handler` will be invoked. Note, that the size of the vector that is returned has v.`size` $= A$.`columns` and v.`capacity` $= A$.`columns`.

`void A.row_vector (base_vector< T > & v, lidia_size_t i) const`

copies the values of row i to `base_vector` v. If $i \geq A$.`rows` or $i < 0$ the `lidia_error_handler` will be invoked. Note, that the size of vector v is adapted automatically.

`void A.get_row_vector (base_vector< T > & v, lidia_size_t i) const`

copies the values of row i to `base_vector` v. If $i \geq A$.`rows` or $i < 0$ the `lidia_error_handler` will be invoked. Note, that the size of vector v is adapted automatically.

`void A.sto_row (const T * v, lidia_size_t j, lidia_size_t i)`

stores j entries of array v in row i of matrix A starting at column zero. If $j > A$.`columns` or $j \leq 0$ or $i \geq A$.`rows` or $i < 0$, the `lidia_error_handler` will be invoked.

`void A.sto_row_vector (const base_vector< T > & v, lidia_size_t j, lidia_size_t i)`

stores j entries of `base_vector` v in row i of matrix A starting at column zero. If $j > A$.`columns` or $j \leq 0$ or $i \geq A$.`rows` or $i < 0$, the `lidia_error_handler` will be invoked.

Access to a column

`base_vector< T > A.operator() (lidia_size_t i) const`

returns a `base_vector` v created by copying the values of column i. If $i \geq A$.`columns` or $i < 0$ the `lidia_error_handler` will be invoked. Note, that the size of the vector that is returned has v.`size` = A.`rows` and v.`capacity` = A.`rows`.

`T * A.column (lidia_size_t i) const`

allocates memory and returns a copy of column i. If $i \geq A$.`columns` or $i < 0$, the `lidia_error_handler` will be invoked.

`T * A.get_column (lidia_size_t i) const`

allocates memory and returns a copy of column i. If $i \geq A$.`columns` or $i < 0$, the `lidia_error_handler` will be invoked.

`void A.column (T * v, lidia_size_t i) const`

copies the values of column i to array v. If $i \geq A$.`columns` or $i < 0$ or v = `NULL`, the `lidia_error_handler` will be invoked. The behaviour of this function is undefined if the array v has less than A.`rows` elements.

`void A.get_column (T * v, lidia_size_t i) const`

copies the values of column i to array v. If $i \geq A$.`columns` or $i < 0$ or v = `NULL`, the `lidia_error_handler` will be invoked. The behaviour of this function is undefined if the array v has less than A.`rows` elements.

`base_vector< T > A.column_vector (lidia_size_t i) const`

returns a `base_vector` v created by copying the values of column i. If $i \geq A$.`columns` or $i < 0$ the `lidia_error_handler` will be invoked. Note, that the size of the vector that is returned has v.`size` = A.`rows` and v.`capacity` = A.`rows`.

`base_vector< T > A.get_column_vector (lidia_size_t i) const`

returns a `base_vector` v created by copying the values of column i. If $i \geq A$.`columns` or $i < 0$ the `lidia_error_handler` will be invoked. Note, that the size of the vector that is returned has v.`size` = A.`rows` and v.`capacity` = A.`rows`.

`void A.column_vector (base_vector< T > & v, lidia_size_t i) const`

copies the values of column i to `base_vector` v. If $i \geq A$.`columns` or $i < 0$, the `lidia_error_handler` will be invoked. Note, that the size of vector v is adapted automatically.

`void A.get_column_vector (base_vector< T > & v, lidia_size_t i) const`

copies the values of column i to `base_vector` v. If $i \geq A$.`columns` or $i < 0$, the `lidia_error_handler` will be invoked. Note, that the size of vector v is adapted automatically.

`void A.sto_column (const T * v, lidia_size_t j, lidia_size_t i)`

stores j entries of array v in column i of matrix A starting at row zero. If $j > A$.`rows` or $j \leq 0$ or $i \geq A$.`columns` or $i < 0$, the `lidia_error_handler` will be invoked.

`void A.sto_column_vector (const base_vector< T > & v, lidia_size_t j, lidia_size_t i)`

stores j entries of `base_vector` v in column i of matrix A starting at row zero. If $j > A$.`rows` or $j \leq 0$ or $i \geq A$.`columns` or $i < 0$, the `lidia_error_handler` will be invoked.

Access to an element

`T & A.operator() (lidia_size_t i, lidia_size_t j)`
returns a reference of $a_{i,j}$. If $i \geq A$.`rows` or $i < 0$ or $j \geq A$.`columns` or $j < 0$, the `lidia_error_handler` will be invoked.

`const T & A.member (lidia_size_t i, lidia_size_t j) const`
returns a constant reference of $a_{i,j}$. If $i \geq A$.`rows` or $i < 0$ or $j \geq A$.`columns` or $j < 0$, the `lidia_error_handler` will be invoked.

`void A.sto (lidia_size_t i, lidia_size_t j, const T & x)`
stores x as $a_{i,j}$. If $i \geq A$.`rows` or $i < 0$ or $j \geq A$.`columns` or $j < 0$, the `lidia_error_handler` will be invoked.

Access to the print mode

`unsigned long A.get_print_mode () const`
returns the print mode for printing matrix A (for further information see the description of Input / Output, page 140).

`void A.set_print_mode (unsigned long i)`
sets the print mode for matrix A to i (for further information see the description of Input / Output, page 140).

Access to the array of values

`T ** A.get_data () const`
returns a pointer to a copy of the array of values of matrix A.

`void A.set_data (const T ** v, lidia_size_t r, lidia_size_t c)`
sets the number of rows of matrix A to r and the number of columns of matrix A to c and copies the values of the $(r \times c)$-array v to the array of values of matrix A. The `lidia_error_handler` will be invoked if $r < 1$ or $c < 1$.

Split Functions

Let A be of type `base_matrix< T >`. The following split functions allow area overlap of the submatrices or subvectors in the following sense:

According to the dimensions of the parameters and the special scheme described in the respective split function, the values of the member matrix are copied to the parameters.

Example:
Look at the matrix

$$A = \begin{pmatrix} 12 & 23 & 62 & 89 \\ 90 & 19 & 80 & 73 \\ 45 & 90 & 32 & 91 \end{pmatrix} .$$

Furthermore, let B be a (2×3)-`base_matrix`, C a (2×2)-`base_matrix`, D a (3×2)-`base_matrix` and E a (3×3)-`base_matrix`.

After executing A`.split_t(`B`,` C`,` D`,` E`)` for the matrices B, C, D and E hold the following:

$$A = \begin{pmatrix} 12 & 23 & 62 & 89 \\ 90 & 19 & 80 & 73 \\ 45 & 90 & 32 & 91 \end{pmatrix}$$

and

$$B = \begin{pmatrix} 12 & 23 & 62 \\ 90 & 19 & 80 \end{pmatrix} \quad , \quad C = \begin{pmatrix} 62 & 89 \\ 80 & 73 \end{pmatrix} \quad , \quad D = \begin{pmatrix} 12 & 23 \\ 90 & 19 \\ 45 & 90 \end{pmatrix} \quad , \quad E = \begin{pmatrix} 23 & 62 & 89 \\ 19 & 80 & 73 \\ 90 & 32 & 91 \end{pmatrix} \quad .$$

If the given conditions in the descriptions of the following functions are not satisfied, the `lidia_error_handler` will be invoked.

Total split

`void` A`.split_t (base_matrix< T > &` B`, base_matrix< T > &` C`, base_matrix< T > &` D`,`
`base_matrix< T > &` E`) const`

takes matrix A apart into the submatrices B, C, D and E, with

$$A = \begin{pmatrix} B & C \\ D & E \end{pmatrix} \quad ,$$

where the following conditions have to be satisfied:

- B`.rows`, C`.rows`, D`.rows`, E`.rows` $\leq$ A`.rows`;
- B`.columns`, C`.columns`, D`.columns`, E`.columns` $\leq$ A`.columns`.

Horizontal split

`void` A`.split_h (base_matrix< T > &` B`, base_matrix< T > &` C`) const`

takes matrix A apart into the submatrices B and C, with $A = \begin{pmatrix} B & C \end{pmatrix}$, where the following conditions have to be satisfied:

- B`.rows`, C`.rows` $\leq$ A`.rows`;
- B`.columns`, C`.columns` $\leq$ A`.columns`.

`void` A`.split_h (T *` v`, base_matrix< T > &` C`) const`

takes matrix A apart into the array v and the submatrix C, with $A = \begin{pmatrix} v & C \end{pmatrix}$, where the following conditions have to be satisfied:

- $v \neq$ `NULL`;
- C`.rows` $\leq$ A`.rows`;
- C`.columns` $\leq$ A`.columns`.

The behaviour of this function is undefined if v has less than A`.rows` elements.

`void` A`.split_h (base_vector< T > &` v`, base_matrix< T > &` C`) const`

takes matrix A apart into the vector v and the submatrix C, with $A = \begin{pmatrix} v & C \end{pmatrix}$, where the following conditions have to be satisfied:

- v`.size` $\leq A$`.rows`;
- C`.rows` $\leq A$`.rows`;
- C`.columns` $\leq A$`.columns`.

`void` A`.split_h (base_matrix< T > &` B`, T *` v`) const`

takes matrix A apart into the submatrix B and the array v, with $A = \begin{pmatrix} B & v \end{pmatrix}$, where the following conditions have to be satisfied:

- $v \neq$ `NULL`;
- B`.rows` $\leq A$`.rows`;
- B`.columns` $\leq A$`.columns`.

The behaviour of this function is undefined if v has less than A`.rows` elements.

`void` A`.split_h (base_matrix< T > &` B`, base_vector< T > &` v`) const`

takes matrix A apart into the submatrix B and the vector v, with $A = \begin{pmatrix} B & v \end{pmatrix}$, where the following conditions have to be satisfied:

- v`.size` $\leq A$`.rows`;
- B`.rows` $\leq A$`.rows`;
- B`.columns` $\leq A$`.columns`.

Vertical split

`void` A`.split_v (base_matrix< T > &` B`, base_matrix< T > &` C`) const`

takes matrix A apart into the submatrices B and C, with $A = \begin{pmatrix} B \\ C \end{pmatrix}$, where the following conditions have to be satisfied:

- B`.rows`, C`.rows` $\leq A$`.rows`;
- B`.columns`, C`.columns` $\leq A$`.columns`.

`void` A`.split_v (T *` v`, base_matrix< T > &` C`) const`

takes matrix A apart into the array v and the submatrix C, with $A = \begin{pmatrix} v \\ C \end{pmatrix}$, where the following conditions have to be satisfied:

- $v \neq$ `NULL`;
- C`.rows` $\leq A$`.rows`;
- C`.columns` $\leq A$`.columns`.

The behaviour of this function is undefined if v has less than A`.columns` elements.

`void` A`.split_v (base_vector< T > &` v`, base_matrix< T > &` C`) const`

takes matrix A apart into the vector B and the submatrix C, with $A = \begin{pmatrix} v \\ C \end{pmatrix}$, where the following conditions have to be satisfied:

- v`.size` $\leq A$`.columns`;
- C`.rows` $\leq A$`.rows`;
- C`.columns` $\leq A$`.columns`.

`void` A`.split_v (base_matrix< T > &` B`, T *` v`) const`

takes matrix A apart into the submatrix B and the array v, with $A = \begin{pmatrix} B \\ v \end{pmatrix}$, where the following conditions have to be satisfied:

- $v \neq$ `NULL`;
- B`.rows` $\leq A$`.rows`;
- B`.columns` $\leq A$`.columns`.

The behaviour of this function is undefined if v has less than A`.columns` elements.

`void` A`.split_v (base_matrix< T > &` B`, base_vector< T > &` v`) const`

takes matrix A apart into the submatrix B and the vector v, with $A = \begin{pmatrix} B \\ v \end{pmatrix}$, where the following conditions have to be satisfied:

- v`.size` $\leq A$`.columns`;
- B`.rows` $\leq A$`.rows`;
- B`.columns` $\leq A$`.columns`.

Compose Functions

Let A be of type `base_matrix< T >`. The following compose functions allow area overlap of the submatrices or subvectors in the following sense:

According to the dimensions of the parameters, the order of the parameters in the prototype and the special scheme described in the respective compose function, the values of the parameters are copied to the member matrix.

Example:

Look at the matrices

$$B = \begin{pmatrix} 12 & 23 & 30 \\ 90 & 19 & 40 \end{pmatrix} \quad , \quad C = \begin{pmatrix} 24 & 89 \\ 67 & 73 \end{pmatrix} \quad , \quad D = \begin{pmatrix} 21 \\ 92 \\ 45 \end{pmatrix} \quad , \quad E = \begin{pmatrix} 12 & 88 & 73 \\ 99 & 33 & 91 \end{pmatrix}$$

Further let A be a (3×4)-`base_matrix`. After executing A`.compose_t(`B`, `C`, `D`, `E`)` for A the following holds:

$$A = \begin{pmatrix} 21 & 23 & 24 & 89 \\ 92 & 12 & 88 & 73 \\ 45 & 99 & 33 & 91 \end{pmatrix}$$

The matrices B, C, D and E remain unchanged.

If the given conditions in the descriptions of the following functions are not satisfied, the `lidia_error_handler` will be invoked.

Total compose

`void` A`.compose_t (const base_matrix< T > &` B`, const base_matrix< T > &` C`,`
`const base_matrix< T > &` D`, const base_matrix< T > &` E`)`

composes the matrices B, C, D and E to the matrix A, with $A = \begin{pmatrix} B & C \\ D & E \end{pmatrix}$, where the following conditions have to be satisfied:

- B`.rows`, C`.rows`, D`.rows`, E`.rows` $\leq A$`.rows`;
- B`.columns`, C`.columns`, D`.columns`, E`.columns` $\leq A$`.columns`.

Horizontal compose

`void` A`.compose_h (const base_matrix< T > &` B`, const base_matrix< T > &` C`)`

composes the matrices B and C to the matrix A, with $A = \begin{pmatrix} B & C \end{pmatrix}$, where the following conditions have to be satisfied:

- B`.rows`, C`.rows` $\leq A$`.rows`;
- B`.columns`, C`.columns` $\leq A$`.columns`.

`void` A`.compose_h (const T *` v`, const base_matrix< T > &` C`)`

composes the array v and the matrix C to the matrix A, with $A = \begin{pmatrix} v & C \end{pmatrix}$, where the following conditions have to be satisfied:

- $v \neq$ `NULL`;
- C`.rows` $\leq A$`.rows`;
- C`.columns` $\leq A$`.columns`.

The behaviour of this function is undefined if v has less than A`.rows` elements.

`void` A`.compose_h (const base_vector< T > &` v`, const base_matrix< T > &` C`)`

composes the vector v and the matrix C to the matrix A, with $A = \begin{pmatrix} v & C \end{pmatrix}$, where the following conditions have to be satisfied:

- v`.size` $\leq A$`.rows`;
- C`.rows` $\leq A$`.rows`;
- C`.columns` $\leq A$`.columns`.

`void` A`.compose_h (const base_matrix< T > &` B`, const T *` v`)`

composes the matrix B and the array v to the matrix A, with $A = \begin{pmatrix} B & v \end{pmatrix}$, where the following conditions have to be satisfied:

- $v \neq$ `NULL`;
- B`.rows` $\leq A$`.rows`;
- B`.columns` $\leq A$`.columns`.

The behaviour of this function is undefined if v has less than A`.rows` elements.

`void` A`.compose_h (const base_matrix< T > &` B`, const base_vector< T > &` v`)`

composes the matrix B and the vector v to the matrix A, with $A = \begin{pmatrix} B & v \end{pmatrix}$, where the following conditions have to be satisfied:

- v`.size` $\leq A$`.rows`;
- B`.rows` $\leq A$`.rows`;
- B`.columns` $\leq A$`.columns`.

Vertical compose

`void` A`.compose_v (const base_matrix< T > &` B`, const base_matrix< T > &` C`)`

composes the matrices B and C to the matrix A, with $A = \begin{pmatrix} B \\ C \end{pmatrix}$, where the following conditions have to be satisfied:

- B`.rows`, C`.rows` $\leq A$`.rows`;
- B`.columns`, C`.columns` $\leq A$`.columns`.

`void` A`.compose_v (const T *` v`, const base_matrix< T > &` C`)`

composes the array v and the matrix C to the matrix A, with $A = \begin{pmatrix} v \\ C \end{pmatrix}$, where the following conditions have to be satisfied:

- $v \neq$ `NULL`;
- C`.rows` $\leq A$`.rows`;
- C`.columns` $\leq A$`.columns`.

The behaviour of this function is undefined if v has less than A`.columns` elements.

`void` A`.compose_v (const base_vector< T > &` v`, const base_matrix< T > &` C`)`

composes the vector v and the matrix C to the matrix A, with $A = \begin{pmatrix} v \\ C \end{pmatrix}$, where the following conditions have to be satisfied:

- v`.size` $\leq A$`.columns`;
- C`.rows` $\leq A$`.rows`;
- C`.columns` $\leq A$`.columns`.

`void` A`.compose_v (const base_matrix< T > &` B`, const T *` v`)`

composes the matrix B and the array v to the matrix A, with $A = \begin{pmatrix} B \\ v \end{pmatrix}$, where the following conditions have to be satisfied:

- $v \neq$ `NULL`;
- B`.rows` $\leq A$`.rows`;
- B`.columns` $\leq A$`.columns`.

The behaviour of this function is undefined if v has less than A`.columns` elements.

`void` A`.compose_v (const base_matrix< T > &` B`, const base_vector< T > &` v`)`

composes the matrix B and the vector v to the matrix A, with $A = \begin{pmatrix} B \\ v \end{pmatrix}$, where the following conditions have to be satisfied:

- v`.size` $\leq A$`.columns`;
- B`.rows` $\leq A$`.rows`;
- B`.columns` $\leq A$`.columns`.

Swap Functions

`void swap (base_matrix< T > &` A`, base_matrix< T > &` B`)`

swaps A and B.

`void` A`.swap_columns (lidia_size_t` i`, lidia_size_t` j`)`
swaps the entries of the columns i and j in A. If $0 \leq i, j < A$`.columns` doesn't hold, the `lidia_error_handler` will be invoked.

`void` A`.swap_rows (lidia_size_t` i`, lidia_size_t` j`)`
swaps the entries of the rows i and j in A. If $0 \leq i, j < A$`.rows` doesn't hold, the `lidia_error_handler` will be invoked.

Diagonal Functions

`void` A`.diag (const T &` a`, const T &` b`)`
stores a at all positions (i, i) of matrix A with $0 \leq i < \min(A$`.rows`$, A$`.columns`$)$ and b at all other positions.

`void diag (base_matrix< T > &` A`, const T &` a`, const T &` b`)`
stores a at all positions (i, i) of matrix A with $0 \leq i < \min(A$`.rows`$, A$`.columns`$)$ and b at all other positions.

Transpose Functions

`base_matrix< T >` A`.trans () const`
returns A^T.

`base_matrix< T > trans (const base_matrix< T > &` A`)`
returns A^T.

`void` A`.trans (const base_matrix< T > &` B`)`
stores B^T to A.

`void trans (base_matrix< T > &` A`, const base_matrix< T > &` B`)`
stores B^T to A.

Input/Output

The `istream` operator `>>` and the `ostream` operator `<<` are overloaded. The `istream` operator `>>` reads matrices in any of the supported formats from istream.

The `ostream` operator `<<` outputs the matrix in the format given by the print mode (see Access Functions, page 133, for further information on how to set the print mode). By default, the "`beautify mode`" is used. The following table shows, which values of print mode correspond to the different output format.

`print_mode`	format
0	"beautify"
1	LiDIA
2	PARI
3	MATHEMATICA
4	MAPLE
5	KANT

- By beautify format we mean the following ASCII format, which should be used only to output matrices on the screen, since it cannot be read by our library:

$$\begin{array}{cccccc} (& a_{0,0} & a_{0,1} & \dots & a_{0,c-1} &) \\ (& a_{1,0} & \dots & \dots & a_{1,c-1} &) \\ (& \vdots & \vdots & \vdots & \vdots &) \\ (& a_{r-1,0} & \dots & \dots & a_{r-1,c-1} &) \end{array}$$

- By LiDIA format we mean the following ASCII format:

$$r \sqcup c \sqcup a_{0,0} \sqcup a_{0,1} \sqcup \cdots \sqcup a_{0,c-1} \sqcup a_{1,0} \sqcup \cdots \sqcup a_{1,c-1} \sqcup \cdots \sqcup a_{r-1,0} \sqcup \cdots \sqcup a_{r-1,c-1} \ ,$$

  where $r = A$.`rows` and $c = A$.`columns`.

- By PARI format we mean the following ASCII format:

$$[a_{0,0}, a_{0,1}, \dots, a_{0,c-1}; a_{1,0}, \dots, a_{1,c-1}; \ \dots; a_{r-1,0}, \dots, a_{r-1,c-1}] \ ,$$

  where $r = A$.`rows` and $c = A$.`columns`.

- By MATHEMATICA format we mean the following ASCII format:

$$\{\{a_{0,0}, a_{0,1}, \dots, a_{0,c-1}\}, \{a_{1,0}, \dots, a_{1,c-1}\}, \dots, \{a_{r-1,0}, \dots, a_{r-1,c-1}\}\} \ ,$$

  where $r = A$.`rows` and $c = A$.`columns`.

- By MAPLE format we mean the following ASCII format:

$$\begin{array}{rccccc} array(1..r, 1..c, [& (1,1) = a_{0,0}, & (1,2) = a_{0,1}, & \dots, & (1,c) = a_{0,c-1}, \\ & (2,1) = a_{1,0}, & \dots & \dots, & (2,c) = a_{1,c-1}, \\ & \vdots & \vdots & \vdots & \vdots \\ & (r,1) = a_{r-1,0}, & \dots & \dots, & (r,c) = a_{r-1,c-1}]); \end{array}$$

  where $r = A$.`rows` and $c = A$.`columns`.

- By KANT format we mean the following ASCII format:

$$LIDIA := Mat(Z, [[a_{0,0}, a_{0,1}, \dots, a_{0,c-1}], [a_{1,0}, \dots, a_{1,c-1}], \dots, [a_{r-1,0}, \dots, a_{r-1,c-1}]])$$

  where $r = A$.`rows` and $c = A$.`columns`.

Interfaces

The following member functions support writing to and reading from ASCII files in the standard format of other computer algebra programs:

`void` A`.write_to_stream (ostream & out) const`
writes matrix A to ostream *out* in LiDIA format.

`void` A`.read_from_stream (istream & in)`
reads LiDIA matrix A from istream *in*.

`void` A`.write_to_gp (ostream & out) const`
writes matrix A to ostream *out* in GP format.

`void` A`.read_from_gp (istream & in)`
reads GP matrix A from istream.

`void` *A*`.write_to_mathematica (ostream & out) const`
writes matrix *A* to ostream *out* in MATHEMATICA format.

`void` *A*`.read_from_mathematica (istream & in)`
reads MATHEMATICA matrix *A* from istream.

`void` *A*`.write_to_maple (ostream & out) const`
writes matrix *A* to ostream *out* in MAPLE format.

`void` *A*`.read_from_maple (istream & in)`
reads MAPLE matrix *A* from istream.

`void` *A*`.write_to_kash (ostream & out) const`
writes matrix *A* to ostream *out* in KASH format.

`void` *A*`.read_from_kash (istream & in)`
reads KASH matrix *A* from istream.

See also

`math_matrix`, `bigint_matrix`, `base_vector`, `math_vector`

Bugs

By now, the function reading and writing matrices in GP-format don't understand the use of '\' as a multiple line break symbol.

Notes

1. As described in the template introduction (see page 11) for using an instance of type `base_matrix< T >` the type `T` has to have at least least have
   - a swap-function `void swap(T &, T&)`,
   - the input-operator `>>`,
   - the output-operator `<<` and
   - the assignment-operator `=`.
2. As usually in C++ the numbering of columns and rows starts with zero.

Examples

```
#include <LiDIA/base_matrix.h>
#include <LiDIA/bigint.h>

int main()
{
    base_matrix < bigint > A(4,5);
    register int i, j;
```

```
        for (i = 0; i < A.get_no_of_rows(); i++)
            for (j = 0; j < A.get_no_of_columns(); j++)
                A.sto(i,j,(bigint)(i+j));
        cout << A << flush;

        return 0;
    }
```

For further examples please refer to `LiDIA/src/templates/matrix/base_matrix_appl.cc`.

Author

Stefan Neis, Patrick Theobald

math_matrix

Name

`math_matrix< T >` ..... parametrized matrix class with basic mathematical operations

Abstract

`math_matrix< T >` is a class for doing simple mathematical operations over matrices of type `T` which is allowed to be either a built-in type or a class such as matrix multiplication or addition.

According to the template introduction (see page 11) the class `math_matrix< T >` is derived from `base_matrix< T >`. So you can apply all the functions and operators of class `base_matrix< T >` to instances of the type `math_matrix< T >`.

Description

A variable of type `math_matrix< T >` contains the same components as `base_matrix< T >`.

In the following descriptions we use A`.rows` to label the number of rows and A`.columns` to label the number of columns of matrix A.

Constructors/Destructor

If $a \leq 0$ or $b \leq 0$ or $v =$ `NULL` in one of the following constructors the `lidia_error_handler` will be invoked.

`ct math_matrix< T > ()`
constructs a 1×1 matrix initialized with values generated by the default constructor for type `T`.

`ct math_matrix< T > (lidia_size_t` a`, lidia_size_t` b`)`
constructs an $a \times b$ matrix initialized with values generated by the default constructor for type `T`.

`ct math_matrix< T > (lidia_size_t` a`, lidia_size_t` b`, const T **` v`)`
constructs an $a \times b$ matrix initialized with the values of the 2-dimensional array v. The behaviour of this constructor is undefined if the array v has less than a rows or less than b columns.

`ct math_matrix< T > (const math_matrix< T > &` A`)`
constructs a copy of matrix A.

`dt ~math_matrix< T > ()`

Arithmetical Operations

The class `math_matrix< T >` supports the following operators:

unary		*op*	`math_matrix< T >`	*op* $\in \{$`-`$\}$
binary	`math_matrix< T >`	*op*	`math_matrix< T >`	*op* $\in \{$`+, -, *`$\}$
binary with assignment	`math_matrix< T >`	*op*	`math_matrix< T >`	*op* $\in \{$`+=, -=, *=`$\}$
binary	`math_matrix< T >`	*op*	`T`	*op* $\in \{$`+, -, *, /`$\}$
binary	`T`	*op*	`math_matrix< T >`	*op* $\in \{$`+, -, *`$\}$
binary with assignment	`math_matrix< T >`	*op*	`T`	*op* $\in \{$`+=, -=, *=, /=`$\}$
binary	`math_matrix< T >`	*op*	`(T *)`	*op* $\in \{$`*`$\}$
binary	`(T *)`	*op*	`math_matrix< T >`	*op* $\in \{$`*`$\}$
binary	`math_matrix< T >`	*op*	`math_vector< T >`	*op* $\in \{$`*`$\}$
binary	`math_vector< T >`	*op*	`math_matrix< T >`	*op* $\in \{$`*`$\}$

Here the operators operating on two matrices and the unary minus implement the usual operations known from linear algebra. If the dimensions of the operands do not satisfy the usual restrictions, the `lidia_error_handler` will be invoked. Note that the dimensions of the resulting matrix are adapted to the right dimensions known from linear algebra if necessary.

The operators

```
math_matrix< T > * (T *)
(T *) * math_matrix< T >
math_matrix< T > * math_vector< T >
math_vector< T > * math_matrix< T >
```

realize the matrix vector multiplication and the vector matrix multiplication. If the number of elements does not satisfy the usual restrictions known from linear algebra the `lidia_error_handler` will be invoked in case of vectors or the behaviour is undefined in case of arrays.

The operators *op* which have a single element of type `T` in their list of arguments perform the operation *op* componentwise, e.g. `T * math_matrix< T >` multiplies each entry of the matrix by the given scalar.

To avoid copying, all operators also exist as functions and of course the restrictions for the dimensions are still valid:

Addition

`void add (math_matrix< T > & `C`, const math_matrix< T > & `A`, const math_matrix< T > & `B`)`
$C \leftarrow A + B$.

`void add (math_matrix< T > & `C`, const math_matrix< T > & `B`, const T & `e`)`
Let $r = B$`.rows` and $c = B$`.columns`.

$$\begin{pmatrix} c_{0,0} & \dots & c_{0,c-1} \\ \vdots & \ddots & \vdots \\ c_{r-1,0} & \dots & c_{r-1,c-1} \end{pmatrix} = \begin{pmatrix} b_{0,0}+e & \dots & b_{0,c-1}+e \\ \vdots & \ddots & \vdots \\ b_{r-1,0}+e & \dots & b_{r-1,c-1}+e \end{pmatrix} .$$

`void add (math_matrix< T > & `C`, const T & `e`, const math_matrix< T > & `B`)`

Let $r = B$.`rows` and $c = B$.`columns`.

$$\begin{pmatrix} c_{0,0} & \dots & c_{0,c-1} \\ \vdots & \ddots & \vdots \\ c_{r-1,0} & \dots & c_{r-1,c-1} \end{pmatrix} = \begin{pmatrix} e + b_{0,0} & \dots & e + b_{0,c-1} \\ \vdots & \ddots & \vdots \\ e + b_{r-1,0} & \dots & e + b_{r-1,c-1} \end{pmatrix} .$$

Subtraction

`void subtract (math_matrix< T > & `C`, const math_matrix< T > & `A`,`
`  const math_matrix< T > & `B`)`

$C \leftarrow A - B$.

`void subtract (math_matrix< T > & `C`, const math_matrix< T > & `B`, const T & `e`)`

Let $r = B$.`rows` and $c = B$.`columns`.

$$\begin{pmatrix} c_{0,0} & \dots & c_{0,c-1} \\ \vdots & \ddots & \vdots \\ c_{r-1,0} & \dots & c_{r-1,c-1} \end{pmatrix} = \begin{pmatrix} b_{0,0} - e & \dots & b_{0,c-1} - e \\ \vdots & \ddots & \vdots \\ b_{r-1,0} - e & \dots & b_{r-1,c-1} - e \end{pmatrix} .$$

`void subtract (math_matrix< T > & `C`, const T & `e`, const math_matrix< T > & `B`)`

Let $r = B$.`rows` and $c = B$.`columns`.

$$\begin{pmatrix} c_{0,0} & \dots & c_{0,c-1} \\ \vdots & \ddots & \vdots \\ c_{r-1,0} & \dots & c_{r-1,c-1} \end{pmatrix} = \begin{pmatrix} e - b_{0,0} & \dots & e - b_{0,c-1} \\ \vdots & \ddots & \vdots \\ e - b_{r-1,0} & \dots & e - b_{r-1,c-1} \end{pmatrix} .$$

Multiplication

`void multiply (math_matrix< T > & `C`, const math_matrix< T > & `A`,`
`  const math_matrix< T > & `B`)`

$C \leftarrow A \cdot B$.

`void multiply (math_matrix< T >& `C`, const math_matrix< T > & `B`, const T & `e`)`

Let $r = B$.`rows` and $c = B$.`columns`.

$$\begin{pmatrix} c_{0,0} & \dots & c_{0,c-1} \\ \vdots & \ddots & \vdots \\ c_{r-1,0} & \dots & c_{r-1,c-1} \end{pmatrix} = \begin{pmatrix} b_{0,0} \cdot e & \dots & b_{0,c-1} \cdot e \\ \vdots & \ddots & \vdots \\ b_{r-1,0} \cdot e & \dots & b_{r-1,c-1} \cdot e \end{pmatrix} .$$

`void multiply (math_matrix< T >& C, const T & e, const math_matrix< T > & B)`

Let $r = B$.`rows` and $c = B$.`columns`.

$$\begin{pmatrix} c_{0,0} & \dots & c_{0,c-1} \\ \vdots & \ddots & \vdots \\ c_{r-1,0} & \dots & c_{r-1,c-1} \end{pmatrix} = \begin{pmatrix} e \cdot b_{0,0} & \dots & e \cdot b_{0,c-1} \\ \vdots & \ddots & \vdots \\ e \cdot b_{r-1,0} & \dots & e \cdot b_{r-1,c-1} \end{pmatrix} .$$

`void compwise_multiply (math_matrix< T >& C, const math_matrix< T > & A,`
`const math_matrix< T > & B)`

Let $r = B$.`rows` $= A$.`rows` and $c = B$.`columns` $= A$.`columns`.

$$\begin{pmatrix} c_{0,0} & \dots & c_{0,c-1} \\ \vdots & \ddots & \vdots \\ c_{r-1,0} & \dots & c_{r-1,c-1} \end{pmatrix} = \begin{pmatrix} a_{0,0} \cdot b_{0,0} & \dots & a_{0,c-1} \cdot b_{0,c-1} \\ \vdots & \ddots & \vdots \\ a_{r-1,0} \cdot b_{r-1,0} & \dots & a_{r-1,c-1} \cdot b_{r-1,c-1} \end{pmatrix} .$$

`void multiply (T *& v, const math_matrix< T > & A, const T * w)`

assigns the result of the muliplication $A \cdot w$ to v (matrix vector multiplication). If no memory has been allocated for the array w, the `lidia_error_handler` will be invoked. If the array w has less than A.`columns` elements, the behaviour of this function is undefined. Note that if $v =$ `NULL`, this function allocates memory in the right size and if the array v has less than A.`rows` elements, the behaviour of this function is undefined.

`void multiply (math_vector< T > & v, const math_matrix< T > & A,`
`const math_vector< T > & w)`

assigns the result of the muliplication $A \cdot w$ to vector v (matrix vector multiplication). If $w.size \neq A$.`columns`, the `lidia_error_handler` will be invoked. Note, that if vector v has less than A.`rows` elements, the size is adapted automatically.

`void multiply (T *& v, const T * w, const math_matrix< T > & A)`

assigns the result of the muliplication $w \cdot A$ to v (vector matrix multiplication). If no memory has been allocated for the array w, the `lidia_error_handler` will be invoked. If the array w has less than A.`rows` elements, the behaviour of this function is undefined. Note that if $v =$ `NULL`, this function allocates memory in the right size and if array v has less than A.`columns` elements, the behaviour of this function is undefined.

`void multiply (math_vector< T > & v, const math_vector< T > & w,`
`const math_matrix< T > & A)`

assigns the result of the muliplication $w \cdot A$ to vector v (vector matrix multiplication). If $w.size \neq A$.`rows`, the `lidia_error_handler` will be invoked. Note that if vector v has less than A.`columns` elements, the size is adapted automatically.

Division

`void divide (math_matrix< T >& C, const math_matrix< T > & B, const T & e)`

Let $r = B$.`rows` and $c = B$.`columns`.

$$\begin{pmatrix} c_{0,0} & \dots & c_{0,c-1} \\ \vdots & \ddots & \vdots \\ c_{r-1,0} & \dots & c_{r-1,c-1} \end{pmatrix} = \begin{pmatrix} b_{0,0}/e & \dots & b_{0,c-1}/e \\ \vdots & \ddots & \vdots \\ b_{r-1,0}/e & \dots & b_{r-1,c-1}/e \end{pmatrix} .$$

`void compwise_divide (math_matrix< T >&` C`, const math_matrix< T > &` A`, const math_matrix< T > &` B`)`

Let $r = B$`.rows` $= A$`.rows` and $c = B$`.columns` $= A$`.columns`.

$$\begin{pmatrix} c_{0,0} & \cdots & c_{0,c-1} \\ \vdots & \ddots & \vdots \\ c_{r-1,0} & \cdots & c_{r-1,c-1} \end{pmatrix} = \begin{pmatrix} a_{0,0}/b_{0,0} & \cdots & a_{0,c-1}/b_{0,c-1} \\ \vdots & \ddots & \vdots \\ a_{r-1,0}/b_{r-1,0} & \cdots & a_{r-1,c-1}/b_{r-1,c-1} \end{pmatrix} .$$

Negation

`void negate (math_matrix< T >&` B`, const math_matrix< T > &` A`)`

$B \leftarrow -A$.

Comparisons

The binary operators `==` and `!=` are overloaded and can be used for comparison by components. Let A be an instance of type `math_matrix< T >`.

`bool` A`.equal (const math_matrix< T > &` B`) const`

returns `true` if A and B are identical, `false` otherwise.

`bool equal (const math_matrix< T > &` A`, const math_matrix< T > &` B`)`

returns `true` if A and B are identical, `false` otherwise.

`bool` A`.unequal (const math_matrix< T > &` B`) const`

returns `false` if A and B are identical, `true` otherwise.

`bool unequal (const math_matrix< T > &` A`, const math_matrix< T > &` B`)`

returns `false` if A and B are identical, `true` otherwise.

Trace

`void` A`.trace (T &` tr`) const`

assigns the trace of matrix A to tr.

`T` A`.trace () const`

returns the trace of matrix A.

`T trace (const math_matrix< T > &` A`)`

returns the trace of matrix A.

See also

`base_matrix`, `bigint_matrix`, `base_vector`, `math_vector`

Notes

As described in the template introduction (see page 11) for using an instance of type `math_matrix< T >` the type `T` has to have at least

- a swap-function `void swap(T &, T&)`,
- the input-operator `>>`,
- the output-operator `<<`,
- the assignment-operator `=`,
- the multiply-operator `*`,
- the addition-operator `+`,
- the division-operator `/`,
- the subtract-operator `-`,
- the unary minus `-` and
- the equal-operator `==`.

Examples

```
#include <LiDIA/math_matrix.h>

int main()
{
    math_matrix < double > A, B;

    cin >> A >> B;
    cout << "trace(A) = " << trace(A) << endl;
    cout << "trace(A+B) = " << trace(A+B) << endl;
    cout << "trace(A*B) = " << trace(A*B) << endl;

    return 0;
}
```

For further examples please refer to `LiDIA/src/simple_classes/math_matrix_appl.cc`.

Author

Stefan Neis, Patrick Theobald

matrix_GL2Z

Name

`matrix_GL2Z` ..... multiprecision $\mathrm{GL}(2,\mathbb{Z})$ arithmetic

Abstract

`matrix_GL2Z` is a class for doing multiprecision arithmetic with matrices of $\mathrm{GL}(2,\mathbb{Z})$. It was designed especially for the algorithms which deal with unimodular transformations of variables of binary quadratic forms.

Description

A `matrix_GL2Z` consists of a quadruple of `bigint`s

$$\begin{pmatrix} s & u \\ t & v \end{pmatrix}$$

with $sv - tu = \pm 1$.

Constructors/Destructor

`ct matrix_GL2Z ()`
initializes with the identity matrix.

`ct matrix_GL2Z (const bigint & a, const bigint & c, const bigint & b, const bigint & d)`
initializes with the matrix $\begin{pmatrix} a & b \\ c & d \end{pmatrix}$. If the determinant is $\neq \pm 1$, the `lidia_error_handler` will be invoked.

`ct matrix_GL2Z (const bigint_matrix & U)`
constructs a copy of the matrix U. If U is not 2×2 or does not have determinant ± 1, the `lidia_error_handler` will be invoked.

`ct matrix_GL2Z (const matrix_GL2Z & U)`
constructs a copy of the matrix U.

`dt ~matrix_GL2Z ()`

Assignments

The operator `=` is overloaded, and assignments from both `matrix_GL2Z` and `bigint_matrix` are possible. The user may also use the following object methods for assignment:

`void` U`.assign_zero ()`
$U \leftarrow \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix}$

`void` U`.assign_one ()`
$U \leftarrow \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$

`void` U`.assign (const matrix_GL2Z &` V`)`
$U \leftarrow V$

`void` U`.assign (const bigint_matrix &` V`)`
$U \leftarrow V$ if V is a 2×2 matrix with determinant ± 1, otherwise the `lidia_error_handler` is invoked.

Access Methods

Let U be of type `matrix_GL2Z`.

`int` U`.det () const`
returns the value of the determinant of U.

`bigint` U`.get_s () const`
returns the value of s.

`bigint` U`.get_t () const`
returns the value of t.

`bigint` U`.get_u () const`
returns the value of u.

`bigint` U`.get_v () const`
returns the value of v.

`bigint` U`.operator() (int` i`, int` j`) const`
returns $U(i,j)$ if $i,j \in \{0,1\}$ (element of the i-th row and j-th column). If i or j are not in $\{0,1\}$, the `lidia_error_handler` will be invoked.

Arithmetical Operations

The operators `*`, `/`, `*=`, and `/=` are overloaded. To avoid copying, these operations can also be performed by the following functions:

`void multiply (matrix_GL2Z &` S`, const matrix_GL2Z &` U`, const matrix_GL2Z &` V`)`
$S = U \cdot V$.

`void divide (matrix_GL2Z & `S`, const matrix_GL2Z & `U`, const matrix_GL2Z & `V`)`
$S = U \cdot V^{-1}$.

`void `U`.invert ()`
$U = U^{-1}$.

`matrix_GL2Z inverse (const matrix_GL2Z `U`)`
returns the U^{-1}.

Comparisons

The operators `==` and `!=` are overloaded. Let U be of type `matrix_GL2Z`.

`bool `U`.is_zero () const`
Returns `true` if $U = \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix}$ and `false` otherwise.

`bool `U`.is_one () const`
Returns `true` if $U = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$ and `false` otherwise.

`bool `U`.is_equal (const matrix_GL2Z & `V`) const`
Returns `true` if $U = V$ and `false` otherwise.

Input/Output

The operators `<<` and `>>` are overloaded. Input and output of a `matrix_GL2Z` have the following ASCII-format, where s, t, u and v are `bigint`s:

```
( s u )
( t v )
```

See also

`quadratic_form`

Author

Thomas Papanikolaou

Chapter 6

Polynomials

polynomial

Name

`polynomial< T >` ..... parameterized polynomials

Abstract

`polynomial< T >` is a class for doing elementary polynomial computations. A variable of type `polynomial< T >` can hold polynomials of arbitrary degree.

Description

A variable f of type `polynomial< T >` is internally represented as a coefficient array with entries of type `T`. The zero polynomial is a zero length array; otherwise, $f[0]$ is the constant-term, and $f[\deg(f)]$ is the leading coefficient, which must always be non-zero.

Constructors/Destructor

`ct polynomial< T > ()`
initializes a zero polynomial.

`ct polynomial< T > (T` x`)`
initializes with the constant polynomial x.

`ct polynomial< T > (const T *` v`, lidia_size_t` d`)`
initializes with the polynomial $\sum_{i=0}^{d} v_i x^i$. The behaviour of this constructor is undefined if the array v has less than $d+1$ elements.

`ct polynomial< T > (const vector< T >` v`)`
initializes with the polynomial $\sum_{i=0}^{n-1} v_i x^i$ where $n = v$`.get_size()`.

`ct polynomial< T > (const polynomial< T > &` f`)`
initializes with a copy of the polynomial f.

`dt ~polynomial< T > ()`

Assignments

Let f be of type `polynomial< T >`.

The operator `=` is overloaded. For efficiency reasons, the following functions are also implemented:

`void` f`.assign_zero ()`
sets f to the zero polynomial.

`void` f`.assign_one ()`
sets f to the polynomial $1 \cdot x^0$.

`void` f`.assign_x ()`
sets f to the polynomial $1 \cdot x^1 + 0 \cdot x^0$.

`void` f`.assign (const T &` a`)`
$f \leftarrow a \cdot x^0$.

`void` f`.assign (const polynomial< T > &` g`)`
$f \leftarrow g$.

Basic Methods and Functions

Let f be of type `polynomial< T >`.

`void` f`.remove_leading_zeros ()`
removes leading zeros. Afterwards, if f is not the zero polynomial, f`.lead_coeff()` $\neq 0$. Note that leading zeros will break the code, therefore, this function is usually called automatically. However, if you manipulate the coefficients by hand or if you used `set_degree()`, you may need to explicitly call this function.

`void swap (polynomial< T > &` a`, polynomial< T > &` b`)`
exchanges the values of a and b.

Access Methods

Let f be of type `polynomial< T >`.

`lidia_size_t` f`.degree () const`
returns the degree of f. For the zero polynomial we return -1.

`void` f`.set_degree (lidia_size_t` n`)`
sets the degree of the polynomial to n. If the polynomial is not modified (by one of the following functions) in such a way that it really has this pretended degree before doing computations with the polynomial, this leads to undefined behaviour. The value of f is always unchanged.

`bigint &` f`.operator[] (lidia_size_t` i`)`
returns a reference to the coefficient of x^i of f. i must be non-negative and less than or equal to the degree of f. This operator may be used to assign a value to a coefficient.

`int` f`.set_data (const T *` d`, lidia_size_t` l`)`

sets f to the polynomial $\sum_{i=0}^{l-1} v_i x^i$. The behaviour of this function is undefined if the array v has less than l elements.

`T *` f`.get_data () const`

returns a pointer to a copy of the array of coefficients of f. If f is the zero polynomial it returns a pointer to an array with one element which is zero.

`bigint` f`.lead_coeff () const`

returns $f[\deg(f)]$, i.e. the leading coefficient of f; returns zero if f is the zero polynomial.

`bigint` f`.const_term () const`

returns $f[0]$, i.e. the constant term of f; returns zero if f is the zero polynomial.

Arithmetical Operations

The following operators are overloaded:

unary		*op*	`polynomial< T >`	*op* $\in \{$`-`$\}$
binary	`polynomial< T >`	*op*	`polynomial< T >`	*op* $\in \{$`+, -, *`$\}$
binary with assignment	`polynomial< T >`	*op*	`polynomial< T >`	*op* $\in \{$`+=, -=, *=`$\}$
binary	`polynomial< T >`	*op*	`T`	*op* $\in \{$`+, -, *`$\}$
binary	`T`	*op*	`polynomial< T >`	*op* $\in \{$`+, -, *`$\}$
binary with assignment	`polynomial< T >`	*op*	`T`	*op* $\in \{$`+=, -=, *=`$\}$

To avoid copying, these operations can also be performed by the following functions (Let f be of type `polynomial< T >`).

`void negate (polynomial< T > &` g`, const polynomial< T > &` h`)`

$g \leftarrow -h$.

`void add (polynomial< T > &` f`, const polynomial< T > &` g`, const polynomial< T > &` h`)`

$f \leftarrow g + h$.

`void add (polynomial< T > &` f`, const polynomial< T > &` g`, const T &` a`)`

$f \leftarrow g + a$.

`void add (polynomial< T > &` f`, const T &` a`, const polynomial< T > &` g`)`

$f \leftarrow g + a$.

`void subtract (polynomial< T > &` f`, const polynomial< T > &` g`,`
`const polynomial< T > &` h`)`

$f \leftarrow g - h$.

`void subtract (polynomial< T > &` f`, const polynomial< T > &` g`, const T &` a`)`

$f \leftarrow g - a$.

`void subtract (polynomial< T > & `f`, const T & `a`, const polynomial< T > & `g`)`
$f \leftarrow a - g$.

`void multiply (polynomial< T > & `f`, const polynomial< T > & `g`,`
`const polynomial< T > & `h`)`
$f \leftarrow g \cdot h$.

`void multiply (polynomial< T > & `f`, const polynomial< T > & `g`, const T & `a`)`
$f \leftarrow g \cdot a$.

`void multiply (polynomial< T > & `f`, const T & `a`, const polynomial< T > & `g`)`
$f \leftarrow g \cdot a$.

`void power (polynomial< T > & `f`, const polynomial< T > & `g`, const bigint & `e`)`
$f \leftarrow g^e$.

Comparisons

The binary operators `==`, `!=` are overloaded.

Let f be an instance of type `polynomial< T >`.

`bool `f`.is_zero () const`
returns `true` if f is the zero polynomial; `false` otherwise.

`bool `f`.is_one () const`
returns `true` if f is a constant polynomial and the constant coefficient equals 1; `false` otherwise.

`bool `f`.is_x () const`
returns `true` if f is a polynomial of degree 1, the leading coefficient equals 1 and the constant coefficient equals 0; `false` otherwise.

High-Level Methods and Functions

Let f be an instance of type `polynomial< T >`.

`void derivative (polynomial< T > & `f`, const polynomial< T > & `g`)`
$f \leftarrow g'$, i.e. the (formal) derivative of g.

`base_polynomial< T > derivative (const base_polynomial< T > & `g`)`
returns g', i.e. the (formal) derivative of g.

`bigint `f`.operator() (const bigint & `a`) const`
returns $\sum_{i=0}^{\deg(f)} f_i \cdot a^i$, where $f = \sum_{i=0}^{\deg(f)} f_i \cdot x^i$.

Input/Output

The `istream` operator `>>` and the `ostream` operator `<<` are overloaded.
Let $f = \sum_{i=0}^{n} f_i \cdot x^i$ where n denotes the degree of the polynomial f.

We support two different I/O-formats:

- The more simple format is

  ```
  [ f_0 f_1 ... f_n ]
  ```

  with elements f_i of type `T`. Leading zeros will be removed at input.

- The more comfortable format (especially for sparse polynomials) is

  ```
  f_n * x^n + ... + f_2 * x^2 + f_1 * x + f_0
  ```

  At output, zero coefficients are omitted, as well as you may omit them at input. In fact the input is even much less restrictive: you may ommit the "*" and you also may permute the monomials, writing e.g. $a_0 + a_n x^n + a_1 * x + \ldots$.

Both formats may be used as input — they are distiguished automatically by the first character of the input, being '[' or not '['. The `ostream` operator `<<` always uses the second format.

See also

`polynomial< bigint >`, `polynomial< bigrational >`, `polynomial< bigfloat >`, `polynomial< bigcomplex >`, `Fp_polynomial`

Notes

Special thanks to Victor Shoup and Thomas Papanikolaou, who gave permission to rewrite and include their code.

Bugs

Since this is a template class, I have to rely on other classes for reading input and producing output. Therefore I don't see, how to prevent "ugly" output like

```
3 * x^4 + -5 * x^2 + -7 .
```

Examples

```
#include <LiDIA/polynomial.h>

int main()
{
    polynomial< bigfloat > f, g, h;

    cout << "Please enter f : "; cin >> f;
    cout << "Please enter g : "; cin >> g;

    h = f * g;
```

```
        cout << "f * g   =  " << h << endl;;

        return 0;
    }
```

Author

Stefan Neis, Thomas Papanikolaou, Victor Shoup

polynomial over bigint, bigrational, bigfloat, bigcomplex

Name

`polynomial< bigint >`,
`polynomial< bigrational >`,
`polynomial< bigfloat >`,
`polynomial< bigcomplex >` ..... specializations of the polynomials

Abstract

`polynomial< bigint >`, `polynomial< bigrational >`, `polynomial< bigfloat >`, and `polynomial< bigcomplex >` are classes for computations with polynomials over `bigint`s, `bigfloat`s, `bigrational`s and `bigcomplex`es. These classes are spezializations of the general `polynomial< T >`, and you can apply all the functions and operators of the general class `polynomial< T >`. Moreover these classes support some additional functionality; especially functions involving division (or pseudo-division for `bigint`s) are offered.

In the following we will describe the additional functions of the classes `polynomial< bigint >`, `polynomial< bigrational >`, `polynomial< bigfloat >`, and `polynomial< bigcomplex >` We will use the parameter `type` to describe the fact that the corresponding function exists in all four classes and has the same functionality in each case. Differences in the classes will be explained separately.

Description

The specializations use the same representation as the general class `polynomial< T >`.

Constructors/Destructor

The specializations support the same constructors as the general type.

Basic Methods and Functions

There are automatic upcasts, which convert a `polynomial< bigint >` to a `polynomial< bigrational >`, a `polynomial< bigfloat >` or a `polynomial< bigcomplex >`. In the same way conversions from a `polynomial< bigrational >` to a `polynomial< bigfloat >` or a `polynomial< bigcomplex >` and from a `polynomial< bigfloat >` to a `polynomial< bigcomplex >` are supported. Thus it is possible to compute e.g. the complex roots of a `polynomial< bigint >` without having to convert it to a `polynomial< bigcomplex >` explicitly.

Arithmetical Operations

In addition to the operators of `polynomial< T >` the following operators are overloaded:

binary	`polynomial< type >`	*op*	`polynomial< type >`	*op* $\in \{$`/`,`%`$\}$
binary with assignment	`polynomial< type >`	*op*	`polynomial< type >`	*op* $\in \{$`/=`,`%=`$\}$
binary	`polynomial< type >`	*op*	`type`	*op* $\in \{$`/`$\}$
binary with assignment	`polynomial< type >`	*op*	`type`	*op* $\in \{$`/=`$\}$

Note that over the `bigint`s these operators involve pseudo-division. So do the following functions which can be used instead of the operators to avoid copying. (Let f be of type `polynomial< type >`.):

`void div_rem (polynomial< type > & q, const polynomial< type > & r, const polynomial< type > & f, const polynomial< type > & g)`

$f = q \cdot g + r$, where $\deg(r) < \deg(g)$.

`void divide (polynomial< type > & q, const polynomial< type > & f, const polynomial< type > & g)`

Computes a polynomial q, such that $\deg(f - q \cdot g) < \deg(g)$.

`void divide (polynomial< type > & q, const polynomial< type > & f, const type & a)`

$q = f/a$; for `polynomial< bigint >` the `lidia_error_handler` is invoked, if a does not divide every coefficient of f.

`void remainder (polynomial< type > & r, const polynomial< type > & f, const polynomial< type > & g)`

Computes r with $r \equiv f \pmod{g}$ and $\deg(r) < \deg(g)$.

`void power_mod (polynomial< type > & x, const polynomial< type > & a, const bigint & e, const polynomial< type > & f)`

$x \equiv a^e \pmod{f}$. For `polynomial< bigint >` the result may be a multiple of x, since pseudo-division is used to reduce a given polynomial mod f.

Greatest Common Divisor

`polynomial< type > gcd (const polynomial< type > & f, const polynomial< type > & g)`

returns $\gcd(f, g)$. Note that these implementations are simply using the euclidean algorithm and thus are far from being very efficient (especially for `polynomial< bigint >` and `polynomial< bigrational >`). For `polynomial< bigint >` the euclidean algorithm uses pseudo-division thus possibly producing a multiple of the "real" gcd, which one would obtain over the bigrationals.

`polynomial< type > xgcd (polynomial< type > & s, polynomial< type > & t, const polynomial< type > & f, const polynomial< type > & g)`

returns $\gcd(f, g) = s \cdot f + t \cdot g$. This function has the same efficiency problems as "gcd".

High-Level Methods and Functions

`bigint cont (const polynomial< bigint > & f)`
returns the content of the polynomial f, i.e. the gcd of all coefficients.

`polynomial< bigint > pp (const polynomial< bigint > & f)`
returns the primitive part of the polynomial f, i.e. f divided by its content.

`lidia_size_t no_of_real_roots (const polynomial< bigint > & f)`
return the number of real roots of the polynomial f.

`bigint resultant (const polynomial< bigint > & f, const polynomial< bigint > & g)`
returns the resultant of two polynomials f and g.

`bigint discriminant (const polynomial< bigint > & f)`
returns the discriminant of the polynomials f.

Integration

`void integral (polynomial< type > & f, const polynomial< type > & g)`
computes f, such that $f' = g$.

`polynomial< type > integral (const polynomial< type > & g)`
returns a polynomial f, such that $f' = g$.

`bigcomplex integral (const bigfloat & a, const bigfloat & b,`
`const polynomial< bigcomplex > & f)`
computes $\int_a^b f(x)dx$. Note that this function may also be applied to a `polynomial< type >` since there are automatic casts to `polynomial< bigcomplex >`.

Computation of Complex Roots

Note that the following functions may also be applied to a `polynomial< type >` since there are automatic casts to `polynomial< bigcomplex >`.

`bigcomplex root (const polynomial< bigcomplex > & f, const bigcomplex & s)`
returns an arbitrary root of f, using s as a first approximation to the root.

`void cohen (const polynomial< bigcomplex > & f, bigcomplex * w, int r, int & c)`
computes the zeros of a squarefree polynomial f and stores them in the array w starting at position c. If less than $c + \deg(f)$ elements are allocated for w before calling this routine, this results in undefined behaviour. The flag r may be set, if all coefficients of the polynomial are real numbers (This will speed up the computation). The function will return with $w[c]$ being the first unused element of the array w.

`void roots (const polynomial< bigcomplex > & f, bigcomplex * w)`
computes the zeros of an arbitrary polynomial f and stores them in the array w. If less than $\deg(f)$ elements are allocated for w before calling this routine, this results in undefined behaviour.

Input/Output

The `istream` operator `>>` and the `ostream` operator `<<` are overloaded. The input formats and the output format are the same as for the general class.

See also

`bigint`, `polynomial< T >`

Examples

```
#include <LiDIA/polynomial.h>

int main()
{
    polynomial< bigint > f, g, h;

    cout << "Please enter f : "; cin >> f;
    cout << "Please enter g : "; cin >> g;

    h = f * g;

    cout << "f * g = " << h << endl;

    bigcomplex * zeros = new bigcomplex[h.degree()+1];

    roots(h, zeros);
    cout << "Zeros of " << h << "are :" << endl;
    for (long int i = 0; i < h.degree(); i++)
    cout << "\n\t" << zeros[i];

    return 0;
}
```

Author

Roland Dreier, Stefan Neis, Thomas Papanikolaou, Nigel Smart, Damian Weber

Chapter 7

Factorization

Please not that the description of the classes which deal with the factorization of polynomials over finite fields can be found in the section "LiDIA FF package" (see page 249); factorization routines for algebraic ideals are part of the LiDIA NF package (see page 397).

rational_factorization

Name

`rational_factorization` .....class for factoring rational numbers and representing factorizations

Abstract

`rational_factorization` is a class for factoring non-zero rational numbers and holding factorizations. At the moment *trial division (TD)*, the *elliptic curve method (ECM)* (see [40]) and the *quadratic sieve (QS)* (see [52]) are supported. Depending on the size of the number to be factored, different combinations of *TD*, *ECM* and *QS* are chosen. In addition it is possible to use own strategies by changing various parameters of *TD* and *ECM*.

Description

The factorization of a rational number is internally represented by an `int` variable *sign* and a vector of pairs $(base, exp)$, where *base* is of type `bigint` and *exp* of type `int`. Let in the following $(base_i, exp_i)$ denote the i-th vector component and l the length of the vector. The length of a `rational_factorization` is defined as length of "its vector". Then this `rational_factorization` represents the rational number

$$f = sign \cdot \prod_{i=0}^{l-1} base_i^{exp_i} \quad .$$

The value of *sign* is either 1 or -1. The values of the bases $base_i$ are always positive. The vector is sorted according to the size of the value of the *base* component. Note that the rational number 0 can not be represented by a `rational_factorization`. Different bases $base_i$, $base_j$ $(i \neq j)$ of the vector are not equal, but not necessarily coprime. In particular, a `rational_factorization` does in general not represent the prime factorization of a rational number.

Constructors/Destructor

All constructors invoke the `lidia_error_handler` if the value of the input variable is 0.

`ct rational_factorization ()`
initializes the variable with 1.

`ct rational_factorization (int` x`, int` e `= 1)`
initializes the variable with x^e.

`ct rational_factorization (unsigned int` x`, int` e `= 1)`
initializes the variable with x^e.

`ct rational_factorization (long` x`, int` e `= 1)`
initializes the variable with x^e.

`ct rational_factorization (unsigned long` x`, int` e `= 1)`
initializes the variable with x^e.

`ct rational_factorization (const bigint &` x`, int` e `= 1)`
initializes the variable with x^e.

`ct rational_factorization (const bigrational &` x`, int` e `= 1)`
initializes the variable with x^e.

`ct rational_factorization (const rational_factorization &` x`, int` e `= 1)`
initializes the variable with x^e.

`dt ~rational_factorization ()`

Assignments

Let a be of type `rational_factorization`. The operator `=` is overloaded and the following assignment functions exist. If you try to assign 0 to a variable of type `rational_factorization`, the `lidia_error_handler` will be invoked.

`void a.assign (long` i`, int` e `= 1)`
$a \leftarrow i^e$.

`void a.assign (const bigint &` I`, int` e `= 1)`
$a \leftarrow I^e$.

`void a.assign (const bigrational &` J`, int` e `= 1)`
$a \leftarrow J^e$.

`void a.assign (const rational_factorization &` F`, int` e `= 1)`
$a \leftarrow F^e$.

Access Methods

`lidia_size_t` a`.no_of_comp () const`
returns the number of components of the `rational_factorization` a, which is the number of different (*base*, *exp*)-pairs.

`int` a`.sign () const`
returns the sign of the rational number represented by a.

`bigint` a`.base (lidia_size_t` i`) const`
returns $base_i$. If $i < 0$ or if i is greater or equal than the number of components of a, the `lidia_error_handler` will be invoked.

`int` a`.exponent (lidia_size_t` i`) const`

returns exp_i. If $i < 0$ or if i is greater or equal than the number of components of a, the `lidia_error_handler` will be invoked.

`void` a`.set_exponent (lidia_size_t` i`, int` e`)`

set $exp_i \leftarrow e$. If $i < 0$ or if i is greater or equal than the number of components of a, the `lidia_error_handler` will be invoked.

Arithmetical Operations

The following operators are overloaded and can be used in exactly the same way as for machine types in C++ (e.g. `int`) :

`(binary) *, /`

Let a be of type `rational_factorization`. To avoid copying, these operations can also be performed by the following functions:

`void multiply (rational_factorization &` c`, const rational_factorization &` a`, const rational_factorization &` b`)`

$c \leftarrow a \cdot b$.

`void` a`.square ()`

`void square (rational_factorization &` c`, const rational_factorization &` a`)`

$c \leftarrow a^2$.

`void divide (rational_factorization &` c`, const rational_factorization &` a`, const rational_factorization &` b`)`

$c \leftarrow a/b$.

`void` a`.invert ()`

`void invert (rational_factorization &` c`, const rational_factorization &` a`)`

$c \leftarrow a^{-1}$.

Comparisons

The binary operators `==` and `!=` are overloaded and can be used in exactly the same way as for machine types in C++ (e.g. `int`).

High-Level Methods and Functions

General High-Level Functions

`void` a`.verbose (int` *state*`)`

sets the amount of information which is printed during computations. If *state* = 1 then the factorization functions print information about the strategy, for *state* = 0 there is no such output. The predefined value is *state* = 0. The verbose mode is stored in a static class variable such that the output behavior is the same for all variables of type `rational_factorization`.

`bool` *a*`.is_prime_factor (lidia_size_t` *i*`)`

returns `true` if $base_i$ is probably a prime number, otherwise `false`. For checking primality the `bigint` function `is_prime()` is used, which uses a probabilistic primality test.

`bool` *a*`.is_prime_factorization ()`

returns `true` if *a* is probably a prime factorization of a rational number, `false` otherwise. For checking primality the `bigint` function `is_prime()`, which uses a probabilistic primality test on each base element of *a*, is used.

`void` *a*`.refine ()`

refines the factorization *a* such that the bases of *a* are pairwise coprime.

`bool` *a*`.refine (const bigint &` *x*`)`

refines the factorization *a* by computing greatest common divisors of *x* with every base occurring in *a*. If *x* has a proper common divisor with at least one base, the return-value of this function will be `true`; otherwise it will be `false`.

`bool` *a*`.refine_comp (lidia_size_t` *i*`, const bigint &` *x*`)`

refines the factorization *a* by computing the greatest common divisor of *x* with $base_i$ in *a*. If *x* has a proper common divisor with $base_i$, the return-value of this function will be `true`; otherwise it will be `false`.

`void` *a*`.factor_comp (lidia_size_t` *i*`, int` *upper_bound* `= 34)`

tries to find factors of $base_i$. A combination of *TD*, *ECM* and *QS* is used. The *i*-th component of *a* is erased, found factors (with appropriate exponents) are added to the vector of *a* and the vector is resorted. If the default parameter *upper_bound* is set, *ECM* tries to find factors up to *upper_bound* digits.

`void` *a*`.factor (int` *upper_bound* `= 34)`

tries to factor all base elements of *a* with a combination of *TD*, *ECM* and *QS* and changes *a* appropriately. If you have more information about all bases (for example no prime factors smaller than 10^6 in any base of *a*) you may get the results more efficiently by using the function *a*`.ecm()` or *a*`.mpqs_comp()`. (see below)

`rational_factorization factor (const bigint &` *N*`)`

tries to factor the number *N* with a combination of *TD*, *ECM* and *QS* and returns the `rational_factorization` representing the number *N*.

`sort_vector < bigint > divisors (rational_factorization &` *f*`)`

return a sorted vector of all positive divisors of the number represented by *f* (in ascending order). If *f* is no prime factorization, the function tries to refine *f* to a prime factorization. In this case, the computed prime factorization is assigned to *f*.

`sort_vector < bigint > divisors (const bigint &` *N*`)`

return a sorted vector of all positive divisors of *N* (in ascending order).

`bigrational evaluate (const rational_factorization &` *f*`)`

return the rational number represented by *f*.

`bigint evaluate_to_bigint (const rational_factorization &` *f*`)`

return the integer represented by *f*. If *f* represents no integer, the `lidia_error_handler` is invoked.

Special High-Level Functions

`void` *a*`.trialdiv_comp (lidia_size_t` *i*`, unsigned int` *upper_bound* `= 1000000, unsigned int` *lower_bound* `= 1)`

tries to factor the element $base_i$ of a by using *TD* with all primes p with $lower\_bound < p < upper\_bound$. The i-th component of a is erased, found factors (with appropriate exponents) are added to the vector of a and the vector is resorted. $base_i$ is not tested for primality.

`void` *a*`.trialdiv (unsigned int` *upper_bound* `= 1000000, unsigned int` *lower_bound* `= 1)`

tries to factor all base elements of a with *TD* using all primes p with $lower\_bound < p < upper\_bound$. The vector of a is changed according to the rules used in *a*`.trialdiv_comp(`*i*`)`. The base elements are not tested for primality.

`rational_factorization trialdiv (const bigint &` *N*`, unsigned int` *upper_bound* `= 1000000, unsigned int` *lower_bound* `= 1)`

tries to factor the number N with *TD* using all primes p with $lower\_bound < p < upper\_bound$ and returns the `rational_factorization` representing the number N. N is not tested for primality.

The following functions require some understanding of the underlying strategy of ECM. A detailed description of the implementation of *TD* and *ECM* can be found in [4] and [48]. *ECM* uses the following strategy, which is parameterized by the `int` variables *lower_bound*, *upper_bound* and *step*. These variables can be chosen by the user. It starts to look for factors with *lower_bound* decimal digits. After having tried a "few" curves and not having found a factor the parameters of *ECM* are changed and we try to find (*lower_bound* + *step*)-digit factors, (*lower_bound* + 2 *step*)-digit factors and so on, until the decimal size of the factors we are looking for exceeds *upper_bound*. The number of curves which is used for finding a d-digit factor is chosen such that we find a d-digit factor with probability of at least 50% if it exists. Note that found factors can be composite. The built-in default values are $lower\_bound = 6$, $step = 3$ and *upper_bound* is set to half the decimal length of the number to be factored. The bounds $lower\_bound \geq 6$ and $upper\_bound \leq 34$ are limited because we have only precomputed parameters for *ECM* for factors of that size.

`void` *a*`.ecm_comp (lidia_size_t` *i*`, int` *upper_bound* `= 34, int` *lower_bound* `= 6, int` *step* `= 3)`

tries to find factors of $base_i$ with decimal length k, $lower\_bound \leq k$ and $k \leq upper\_bound$, by means of *ECM* according to the strategy explained above. Note that found factors of $base_i$ can be composite. If the value of *upper_bound* is 34, then *upper_bound* is set to $\min(34, \frac{1}{2}\lfloor \log_1 0(base_i) \rfloor + 1)$. The vector of a is changed appropriately. Unreasonable input ($step \leq 0$, *lower_bound* or *upper_bound* out of range etc.) leads to a call of the `lidia_warning_handler`, the parameter is set to a predefined value.

`void` *a*`.ecm (int` *upper_bound* `= 34, int` *lower_bound* `= 6, int` *step* `= 3)`

tries to factor all base elements of a with *ECM* using the strategy described in *a*`.ecm_comp(`*i*`)` and changes a appropriately. Note that the result is not necessarily a prime factorization.

`rational_factorization ecm (const bigint &` *N*`, int` *upper_bound* `= 34, int` *lower_bound* `= 6, int` *step* `= 3)`

tries to factor the number N with *ECM* using the strategy described in *a*`.ecm_comp(`*i*`)` and returns the `rational_factorization` representing the number N. Note that the result is not necessarily a prime factorization.

The version of *QS* used in LiDIA is called self initializing multiple polynomial large prime quadratic sieve. A description of the algorithm can be found in [1], [22] or [60]. The linear equation step uses an implementation of the Block Lanczos algorithm for $\mathbb{Z}/2\mathbb{Z}$.

Note that the running time of *QS* depends on the size of the number to be factored, not on the size of the factors.

`void` a`.mpqs_comp (lidia_size_t` i`)`

tries to factor $base_i$ using the large prime variation of the quadratic sieve. For bases with more than 75 digits the `lidia_warning_handler` will be invoked, the `rational_factorization` a is not changed. Notice that as QS creates temporary files, you need write permission either in the `/tmp`-directory or in the directory from which QS is called.

`void` a`.mpqs (const bigint &` N`)`

tries to factor N using a`.mpqs_comp` (see above).

Input/Output

The `istream` operator `>>` and the `ostream` operator `<<` are overloaded. The `istream` operator `>>` expects the input of a `rational_factori-zation` in the following format:

$$[(base_0, exp_0)\,(base_1, exp_1)\,\ldots\,(base_{l-1}, exp_{l-1})]$$

In the input $base_i$ is allowed to be an arbitrary non-zero rational integer (i.e., of type `bigint`). The input function computes the internal format which was explained in the beginning.

See also

`bigint`, `bigrational`

Warnings

ECM fails to factor rational numbers that only consist of prime factors smaller than 1000. Therefore it is strongly recommended to use the function `factor()` or to ensure with the function `trialdiv()`, that there are no such small prime factors left before calling functions which use *ECM*.

Examples

```
#include <LiDIA/rational_factorization.h>

int main()
{
    rational_factorization f;
    bigint n;

    cout << "enter an integer n: ";
    cin >> n;             //  input n
    f.assign(n);          //  assign n as trivial factorization to f
    f.verbose(1);         //  we want a lot of informations
    f.factor();           //  calculate prime factorization
                          //  using the factor function
    cout << f;            //  output f

    cout << "enter an integer n: ";
    cin >> n;             //  input n
    f.assign(n);          //  assign n as trivial factorization to f
    f.verbose(1);         //  we want a lot of informations
    f.trialdiv();         //  calculate prime factorization using trialdiv
```

```
        cout << f;            //  output f

        cout << "enter an integer n: ";
        cin >> n;             //  input n
        f.assign(n);          //  assign n as trivial factorization to f
        f.verbose(1);         //  we want a lot of informations
        f.ecm_comp(0);        //  calculate prime factorization using ecm
        cout << f;            //  output f

        cout << "enter an integer n: ";
        cin >> n;             //  input n
        f.assign(n);          //  assign n as trivial factorization to f
        f.verbose(1);         //  we want a lot of informations
        f.mpqs_comp(0);       //  calculate prime factorization using
                              //  quadratic sieve
        cout << f;            //  output f

        return 0;
    }
```

For further references please refer to `LiDIA/src/simple_classes/factorization/rational_factorization_appl.cc`

Author

Franz-Dieter Berger, Thomas F. Denny, Andreas Müller, Volker Müller, Thomas Sosnowski

single_factor

Name

`single_factor< T >` ..... parameterized class for factoring elements of type `T`

Abstract

Including `LiDIA/single_factor.h` in an application allows the use of the C++ type

```
single_factor< T >
```

for some data type `T` which is allowed to be either a built-in type or a class.

`single_factor< T >` is a class for factoring elements of type `T` and for storing factors of a `factorization< T >`. It offers elementary arithmetical operations and information concerning the primality of the represented element. This class is used in the class `factorization< T >`.

In LiDIA, factorization algorithms are implemented for the following types:

- `single_factor< Fp_polynomial >` (factorization of polynomials over finite prime fields)
- `single_factor< polynomial< gf_p_element > >` (factorization of polynomials over finite fields).

In the next release we plan to offer factorization algorithms for the following additional types:

- `single_factor< bigint >` (factorization of integers, replacing `rational_factorization`)
- `single_factor< ideal >` (factorization of ideals).

Description

A `single_factor< T >` is internally represented by a variable of type `T` named *base* and a variable named *state* containing information about the primality or compositeness of *base*.

state can hold '`prime`', '`not_prime`' or '`unknown`'; the latter indicating that nothing is known about primality or compositeness of *base* so far.

A test for primality or compositeness is implemented in the function `is_prime_factor()`. Note that these tests may be based on tests with limited capabilities. For example in the case of `bigint`s the Miller-Rabin-test is used which can prove compositeness, however, if an integer is claimed to be prime, this might actually be wrong — although this is highly improbable.

Constructors/Destructor

`ct single_factor< T > ()`
initializes the `single_factor< T >` with the identity element ("1") of the multiplication for elements of type `T`.

`ct single_factor< T > (const T & a)`
initializes the `single_factor< T >` with an element a of type `T`. The *state* is set to "unknown".

`ct single_factor< T > (const single_factor< T > & a)`
initializes the `single_factor< T >` with the `single_factor< T > a`.

`dt ~single_factor< T > ()`

Assignments

The operator `=` is overloaded and moreover the following assignment functions exist. (Let a be an instance of type `single_factor< T >`.)

`void a.assign (const single_factor< T > & b)`
$a \leftarrow b$.

`void a.assign (const T & b)`
base of a is set to b; *state* of a is set to `unknown`.

Comparisons

The binary operators `==`, `!=`, `<`, `<=`, `>=`, `>` are overloaded. Note that although these operators may do a meaningful comparision of the elements represented by a `single_factor< T >` (e.g. for `bigint`s), the operators may as well implement a completely artifical ordering relation (e.g. a lexical ordering), since this relation is only needed for sorting vectors of type `single_factor< T >`.

Arithmetical Operations

The operators `(binary) *, /` are overloaded, however using operator `/` is only a well-defined operation, if the divisor really divides the first operand. Multiplication and division can also be performed by the functions `multiply` and `divide`, which avoid copying and therfore are faster than the operators.

`void multiply (single_factor< T > & c, const single_factor< T > & a,`
`const single_factor< T > & b)`
$c \leftarrow a \cdot b$.

`void divide (single_factor< T > & c, const single_factor< T > & a,`
`const single_factor< T > & b)`
$c \leftarrow a/b$. Note that the behaviour of this function is undefined if b is not a divisor of a.

`void gcd (single_factor< T > & c, const single_factor< T > & a,`
`const single_factor< T > & b)`
$c \leftarrow \gcd(a, b)$.

`lidia_size_t ord_divide (const single_factor< T > & a, single_factor< T > & b)`
returns the largest non-negative integer e such that a^e divides b. b is divided by a^e.

Access Methods

Let a be an instance of type `single_factor< T >`.

`T & a.base ()`
returns a reference to the element of type `T` represented by a.

`const T & a.base () const`
returns a reference to the element of type `T` represented by a.

High-Level Methods and Functions

Let a be an instance of type `single_factor< T >`.

Queries

`bool a.is_prime_factor (int test = 0)`
returns `true`, if f is a prime factorization according to a test in class `single_factor< T >`. This test either can be a *compositeness test*, which decides whether f is composite or probably prime or a *primality test* which proofs that f is prime. If $test \neq 0$, an explicit test is done if we do not yet know, whether or not a is prime, otherwise only *state* is checked.

Modifying Operations

`void swap (single_factor< T > a, single_factor< T > b)`
swaps the values of a and b.

`T a.extract_unit ()`
sets a to a fixed representant of the equivalence class aU where U is the group of units of type `T`. The unit ϵ by which we have to multiply the new value of a to obtain the old one is returned.

`factorization< T > a.factor ()`
returns a factorization of the element of type `T` represented by the `single_factor< T >` a, if a suitable routine for `single_factor< T >` is defined. Otherwise, you only get a message saying that no factorization routine has been implemented for that type.

Input/Output

The `istream` operator `>>` and the `ostream` operator`<<` are overloaded. By now, the I/O-format is identical to that of elements of type `T`.

Notes

The type `T` has to offer the following functions:

- the assignment operator `=`
- the operator `=` (`int`), if you don't specialize the constructors

- a swap function `void swap(T&, T&)`
- the friend-functions `multiply`, `divide`, `gcd`
- the input-operator `>>` and
- the output-operator `<<`
- the comparison-operators `<`, `<=`, `==`, `!=`

See also

`factorization< T >`, `bigint`, `Fp_polynomial`, `polynomial< gf_p_element >`, `module`, `alg_ideal`.

Author

Oliver Braun, Thomas F. Denny, Stefan Neis, Thomas Pfahler

factorization

Name

`factorization< T >` ..... parameterized class for computing with factorizations

Abstract

Including `LiDIA/factorization.h` in an application allows the use of the C++ type `factorization< T >` for some data type `T` which is allowed to be either a built-in type or a class.

`factorization< T >` is a class for holding factorizations. There are elementary access functions, functions for computing with factorizations and modifying operations.

The class `factorization< T >` is the return type of the factorization algorithms in the class `single_factor< T >`.

Description

The factorization of an element of type `T` is internally represented by

- a unit ϵ, which is automatically extracted from all bases (see function `extract_unit()` of class `single_factor< T >`) and
- two *vector*s p and c of pairs $p_i = (ibase_i(p), exp_i(p))$ and $c_i = (ibase_i(c), exp_i(c))$ with $l(p)$ and $l(c)$ components, respectively. These vectors represent the factors assumed to be prime (or irreducible) and those where we know nothing about primality (or irreducibility), respectively. In an abuse of language, we call the bases of the components of c, i.e. the components which might be composite factors, the "composite components", although this is not generally correct.

  $(ibase_i(x), exp_i(x))$ denotes the i-th component of the vector x, where

  - $ibase_i(x)$ is of type `single_factor< T >`, and
  - $exp_i(x)$ is of type `lidia_size_t`.

So a `factorization< T >` f represents the element

$$f = \epsilon \cdot \prod_{i=0}^{l(p)-1} ibase_i\texttt{.base}(p)^{exp_i(p)} \prod_{i=0}^{l(c)-1} ibase_i\texttt{.base}(c)^{exp_i(c)} \ .$$

Different bases $ibase_i(x), ibase_j(y)$ $(i \neq j)$ of the vectors are not necessarily coprime. Especially, a `factorization< T >` does not generally represent a prime factorization.

Constructors/Destructor

`ct factorization< T > ()`
initializes a factorization of length zero. ϵ is set to "1" (the identity element of the multiplication for elements of type `T`).

`ct factorization< T > (const factorization< T > & f)`
initializes the factorization with a copy of f.

`ct factorization< T > (const single_factor< T > & a)`
initializes the factorization with a.

`dt ~factorization< T > ()`

Assignments

The operator `=` is overloaded and additionally the following assignment functions exist:

Let f be an instance of type `factorization< T >`.

`void` f`.assign (const factorization< T > & g)`
$f \leftarrow g$.

`void` f`.assign (const single_factor< T > & a)`
$f \leftarrow a$.

Comparisons

The binary operators `==` and `!=` are overloaded.

Two `factorization< T >`s f and g are considered to be equal if the value represented by f is equal to the value represented by g. Note that this can be checked efficiently by using factor refinement.

Arithmetical Operations

The following operators are overloaded:

`(binary) *, /`

These operations can also be performed by the functions `multiply()` and `divide()` (which avoid copying and are therefore faster than the operators).

`void multiply (factorization< T > & h, const factorization< T > & f,`
`const factorization< T > & g)`
$h = f \cdot g$, i.e. the components of the factorizations of f and g are combined to the factorization h, such that h represents the element $f \cdot g$.

`void multiply (factorization< T > & h, const factorization< T > & f,`
`const single_factor< T > & a)`
$h = f \cdot a$, i.e. the components of the factorization of f and the `single_factor< T >` a are combined to the factorization h, such that h represents the element $f \cdot a$.

`void divide (factorization< T > & h, const factorization< T > & f, const factorization< T > & g)`
$h = f/g$, i.e. the components of the factorizations of f and g are combined to the factorization h, such that h represents the element f/g.

In the following, let f be an instance of type `factorization< T >`.

`void f.invert ()`
f is inverted (by negating the exponents).

`void invert (factorization< T > & h, const factorization< T > & f)`
sets $h = f^{-1}$.

`factorization< T > inverse (factorization< T > & f)`
returns f^{-1}.

`void f.square ()`
sets $f = f^2$, i.e. all exponents in the representation of f are multiplied by 2 and the unit is squared.

`void square (factorization< T >& h, const factorization< T >& f)`
sets $h = f^2$.

`void f.power (lidia_size_t i)`
sets $f = f^i$, i.e. all exponents in the represenation of f are multiplied by i and the unit is replaced by its i-th power.

`void power (factorization< T >& h, const factorization< T >& f, lidia_size_t i)`
sets $h = f^i$.

Access Methods

Let f be an instance of type `factorization< T >`.

`single_factor< T > f.prime_base (lidia_size_t i)`
returns $ibase_i(p)$. If $i < 0$ or if i is greater than or equal to the number of components of p, the `lidia_error_handler` will be invoked.

`single_factor< T > f.composite_base (lidia_size_t i)`
returns $ibase_i(c)$. If $i < 0$ or if i is greater than or equal to the number of components of c, the `lidia_error_handler` will be invoked.

`lidia_size_t f.prime_exponent (lidia_size_t i)`
returns $exp_i(p)$. If $i < 0$ or if i is greater than or equal to the number of components of p, the `lidia_error_handler` will be invoked.

`lidia_size_t f.composite_exponent (lidia_size_t i)`
returns $exp_i(c)$. If $i < 0$ or if i is greater than or equal to the number of components of c, the `lidia_error_handler` will be invoked.

`lidia_size_t` f`.no_of_prime_components ()`

returns the number of components of p, which is the number of $(ibase, exp)$-pairs of p.

`lidia_size_t` f`.no_of_composite_components ()`

returns the number of components of c, which is the number of $(ibase, exp)$-pairs of c.

`lidia_size_t` f`.no_of_components ()`

returns the number of components of f, which is the sum of the number of prime and composite components.

`T` f`.unit ()`

returns the unit ϵ.

`T` f`.value ()`

returns the element of type `T` that is represented by the factorization of f. Note that this computation will take a long time and will consume lots of memory, if the exponents involved in the computation are large.

High-Level Methods and Functions

Let f be an instance of type `factorization< T >`.

Modifying Operations

`void` f`.replace (lidia_size_t` *pos*`, factorization< T >` g`)`

This function is used to replace a composite component by its factorization. It replaces the composite component at position *pos* by the composite components of the factorization g. The prime components of g are appended to the prime components of f.

`void` f`.sort ()`

sorts the prime components as well as the composite componentes of the factorization in ascending order according to the size of the `single_factor< T >` and the size of the exponents, so that $f.ibase(i) \leq f.ibase(j)$ for all $i < j$ and, if $f.ibase(i) = f.ibase(j)$, $f.exp(i) \leq f.exp(j)$. The operators "`<=`" and "`==`" are to be defined in the class `single_factor< T >`.

`void` f`.normalize ()`

sorts the factorization f and collects equal bases, such that $f.ibase_i(p) \neq f.ibase_j(p)$, $f.ibase_i(c) \neq f.ibase_j(c)$ for all $i \neq j$, and $f.ibase_i(p) \neq f.ibase_j(c)$ for all i, j. "`!=`" has to be defined in the class `single_factor< T >`.

`void` f`.refine ()`

refines the factorization f such that the bases of f are pairwise coprime, i.e. $\gcd\big(f.ibase_i(x), f.ibase_j(y)\big) =' 1'$ for $x, y \in \{p, c\}$ and all i, j. The function `gcd` has to be defined in the class `single_factor< T >`.

`void` f`.factor ()`

factors all composite components and replaces each such component $f.ibase_i(c)$ by its factorization `factor(`$f.ibase_i(c)$`)`, $i = 0, \ldots,$ f`.no_of_composite_components() - 1` using the function `replace` describe above. The function `factor` must be defined in the class `single_factor< T >`.

Queries

`bool` *f*`.is_prime_factorization (int` *test* `= 0)`

returns `true`, if *f* is a prime factorization according to the test in class `single_factor< T >`. This test can either be a *compositeness test*, which decides whether *f* is composite or probably prime or a *primality test* which proofs that *f* is prime.

If $test \neq 0$, an explicit test is done for every composite component with unknown prime state, otherwise we only check, whether there are composite components.

`bool` *f*`.is_sorted ()`

returns `true`, if *f* has already been sorted using the function *f*`.sort()` (which is decided by checking a flag).

`bool` *f*`.is_normalized ()`

returns `true`, if *f* has been normalized using the function *f*`.normalize()` (which is decided by checking a flag).

`bool` *f*`.is_refined ()`

returns `true`, if *f* has been refined using the function *f*`.refine()` (which is decided by checking a flag).

Input/Output

The `istream` operator `>>` and the `ostream` operator`<<` are overloaded. The `ostream` operator `<<` prints a `factorization< T >` in the following format:

$$[\epsilon, [ibase_0(p).base, exp_0(p)], [ibase_1(p).base, exp_1(p)], \ldots, [ibase_{l-1}(p).base, exp_{l(p)-1}(p)],$$
$$[ibase_0(c).base, exp_0(c)], [ibase_1(c).base, exp_1(c)], \ldots, [ibase_{l-1}(c).base, exp_{l(c)-1}(c)]\,]$$

where ϵ is a unit (as explained at the beginning), If this unit is the neutral element of multiplication, it is omitted unless $l(p) = l(c) = 0$.

The `istream` operator `>>` assumes all components to be composite, so the input format is the following (again, the unit is optional unless $l(c) = 0$):

$$[\epsilon, [ibase_0(c).base, exp_0(c)], \ldots, [ibase_{l-1}(c).base, exp_{l(c)-1}(c)]]$$

So in practice input/output looks like

```
[ [ x^2 - 1 mod 5, 1], [x^2 + 1 mod 5, 2] ].
```

Notes

- The type `T` must at least have the following functions:
  - the assignment operator `=`
  - the operator `=` (`int`), if you don't specialize the constructors
  - a swap function `void swap(T&, T&)`
  - the friend-functions `multiply`, `divide`, `gcd`
  - the input-operator `>>` and
  - the output-operator `<<`
  - the comparison-operators `<`, `<=`, `==`, `!=`
- The numeration of the components in a `factorization< T >` starts with zero. (As usually in C++)

See also

`single_factor`, `sort_vector`

Examples

```
#include <LiDIA/Fp_polynomial.h>
#include <LiDIA/factorization.h>

int main()
{
    factorization< Fp_polynomial > f;

    cout << "Please enter a factorization of type Fp_polynomial:";
    cout << endl;

    cin >> f;

    cout << "You entered a factorization with ";
    cout << f.no_of_components();
    cout << " components." << endl << endl;

    cout << "The value represented by the factorization is " << endl;
    cout << f.value() << "." << endl << endl;

    f.factor_all_components();

    cout << "The prime factorization is :" << endl;
    cout << f <<endl;

    return 0;
}
```

Author

Oliver Braun, Thomas F. Denny, Stefan Neis, Thomas Pfahler

single_factor< bigint >

Name

`single_factor< bigint >` ..... a single factor of a bigint

Abstract

`single_factor< bigint >` is used for storing a single factor of a factorization of rational integers (see the general template class `factorization< T >`). It is a specialization of `single_factor< T >` with some additional functionality, namely different factorization algorithms.

Description

`single_factor< bigint >` is used for storing a single factor of a factorization of rational integers (see the general template class `factorization< T >`). It is a specialization of `single_factor< T >` with some additional functionality, namely different factorization algorithms. All functions for `single_factor< T >` can be applied to objects of class `single_factor< bigint >`, too. These basic functions are not described here any further; you will find the description of the latter in `single_factor< T >`.

At the moment we have implemented the following factorization algorithms: trial division (TD), Pollard Rho algorithm, Pollard $p-1$ algorithm, Williams $p+1$ algorithm, the elliptic curve method (*ECM*) and the self initializing variant of the Quadratic sieve (*MPQS*) with Block Lanczos algorithm as system solver. The Pollard $p-1$ algorithm, Williams $p+1$ algorithm, and *ECM* use the so called Improved Standard Continuation. A description of the theory of these algorithms can for example be found in [54], [1], and in several diploma theses (see [22], [60], [4], [48]).

We do not describe the general template functions offered by `single_factor< T >`. A description for these functions can be found in the manual for this template class. Here we concentrate on the different factorization algorithms implemented for factorization of rational integers. These factorization algorithm usually stop after a non trivial factor of the input number has been found. Therefore note that the result of a factorization algorithm is not necessary a prime factorization. Moreover it is possible that the returned factorization just has one member, namely the input number itself. This can happen when the used factorization algorithm has not found a proper factor.

Factorization Algorithms

Let a be an instance of type `single_factor< bigint >`.

It should be mentioned that all factorization implementations which have some integer x as input return in any case a factorization which has the same value as x. If a factor of x has been found, then the result is a non trivial factorization; otherwise the result is a trivial factorization holding only x itself.

`factorization< bigint > `*a*`.TrialDiv (unsigned int` *upper_bound* `= 1000000,`
`unsigned int` *lower_bound* `= 1)`

tries to factor the integer stored in *a* by using *TD* with all primes *p* satisfying *lower_bound* $\leq$ *p* $\leq$ *upper_bound*. For unreasonable parameters as negative values for *lower_bound*, the `lidia_error_handler` is invoked. Found factors (with appropriate exponents) are returned as factorization. If the found factorization is a prime factorization, then the complete prime factorization is returned and *a* is set to one. Otherwise found factors are returned as factorization and *a* is set to the remaining composite part.

`factorization< bigint > TrialDiv (const bigint &` *x*`, unsigned int` *upper_bound* `= 1000000,`
`unsigned int` *lower_bound* `= 1)`

tries to factor the integer *x* by using *TD* with all primes *p* satisfying *lower_bound* $\leq p \leq$ *upper_bound*. The function returns the factorization found with this method. Unreasonable input leads to a call to the `lidia_error_handler`.

`factorization< bigint > `*a*`.PollardPminus1 (int` *size* `= 9)`

tries to factor the integer stored in *a* by using Pollard's $(p-1)$ method. Parameters are chosen as in *ECM* for finding factors of size *size* with some probability. If a factor is found, the factorization of *a* is returned and *a* is set to one; otherwise *a* remains unchanged and an empty factorization is returned. Unreasonable values for *size* lead to a call of the `lidia_error_handler`.

`factorization< bigint > PollardPminus1 (const bigint &` *x*`, int` *size* `= 9)`

tries to factor the integer *x* by using Pollard's $(p-1)$ method. Parameters are chosen as in *ECM* to find factors of size *size* with some probability. A factorization of *x* is returned.

`factorization< bigint > `*a*`.PollardRho (int` *size* `= 7)`

tries to factor the integer stored in *a* by using Pollard's rho method. Parameters are chosen for finding factors of size *size* with some probability. If a factor is found, the factorization is returned and *a* is set to one; otherwise *a* remains unchanged and an empty factorization is returned. Unreasonable values for *size* lead to a call of the `lidia_error_handler`.

`factorization< bigint > PollardRho (const bigint &` *x*`, int` *size* `= 9)`

tries to factor the integer *x* by using Pollard's rho method. Parameters are chosen for finding factors of size *size*. A factorization of *x* is returned. Unreasonable values for *size* lead to a call of the `lidia_error_handler`.

`factorization< bigint > `*a*`.WilliamsPplus1 (int` *size* `= 9)`

tries to factor the integer stored in *a* by using Williams $p+1$ method. Parameters are chosen as in *ECM* to find factors of size *size* with some probability. If a factor is found, the factorization is returned and *a* is set to one; otherwise *a* is unchanged and an empty factorization is returned. Unreasonable values for *size* lead to a call of the `lidia_error_handler`.

`factorization< bigint > WillaimsPplus1 (const bigint &` *x*`, int` *size* `= 9)`

tries to factor the integer *x* by using Williams $p+1$ method. Parameters are chosen as in *ECM* to find factors of size *size* with some probability. A factorization of *x* is returned. Unreasonable values for *size* lead to a call to the `lidia_error_handler`.

`factorization< bigint > `*a*`.Fermat ()`

tries to factor the integer stored in *a* by using Fermat's method with a fixed number of iterations. If a factor is found, this factorization is returned and *a* is set to one; otherwise *a* is unchanged and an empty factorization is returned.

`factorization< bigint > Fermat (const bigint & x)`

tries to factor the integer stored in x by using Fermat's method with a fixed number of iterations. A factorization of x is returned.

The following functions require some understanding of the underlying strategy of *ECM*. A detailed description of the implementation of *TD* and *ECM* can be found in [4] and [48]. *ECM* uses the following strategy, which is parameterized by the int variables *lower_bound*, *upper_bound* and *step*. These variables can be chosen by the user. It starts to look for factors with *lower_bound* decimal digits. After having tried a "few" curves and not having found a factor the parameters of *ECM* are changed and we try to find (*lower_bound* + *step*)-digit factors, (*lower_bound* + 2 *step*)-digit factors and so on, until the decimal size of the factors we are looking for exceeds *upper_bound*. The number of curves which is used for finding a d-digit factor is chosen such that we find a d-digit factor with probability of at least 50% if it exists. Note that found factors can be composite. The built-in default values are *lower_bound* $= 6$, *step* $= 3$ and *upper_bound* is set to half the decimal length of the number to be factored. The bounds *lower_bound* ≥ 6 and *upper_bound* ≤ 34 are limited because we have only precomputed parameters for *ECM* for factors of that size.

`factorization< bigint > a.ECM (int upper_bound = 34, int lower_bound = 6, int step = 3, bool jump_to_QS = false)`

tries to factor the integer stored in a by using the ECM method with the strategy described above. If `jump_to_QS` is set to `true`, then the function uses some heuristic to check whether *MPQS* would be faster, if so, it starts the *MPQS* algorithm to factor the integer. The function returns a factorization of the integer stored in a. If a proper factor was found, then a is set to one.

`factorization< bigint > ECM (const bigint & x, int upper_bound = 34, int lower_bound = 6, int step = 3)`

tries to factor the integer stored in x by using the *ECM* method with the strategy described above. A factorization of x is returned.

The version of *MPQS* used in LiDIA is called self initializing multiple polynomial large prime quadratic sieve. A description of the algorithm can be found in [1], [22] or [60]. The linear equation step uses an implementation of the Block Lanczos algorithm for $\mathbb{Z}/2\mathbb{Z}$. Note that the running time of *MPQS* depends on the size of the number to be factored, not on the size of the factors.

`factorization< bigint > a.MPQS ()`

returns a factorization of the integer stored in a by using the *MPQS* method.

`factorization< bigint > MPQS (const bigint & x)`

returns a factorization of the integer x by using the *MPQS* method.

`factorization< bigint > a.factor (int size = 34) const`

returns a factorization of a using a strategy which tries to apply an "optimal" combination of the above algorithms. Parameters are chosen such that the algorithms whose running time depends on the size of the factor will find factors of size *size* with 50 % probability. A factorization of a is returned, a is set to one.

`factorization< bigint > sf_factor (const bigint & x, int size = 34)`

returns a factorization of x using a strategy which tries to apply an "optimal" combination of the above algorithms. Parameters are chosen such that the algorithms whose running time depends on the size of the factor will find factors of size *size* with 50 % probability.

`factorization< bigint > a.completely_factor () const`

returns a prime factorization of a using a strategy which tries to apply always the fastest of the above algorithms. a is set to one.

`factorization< bigint > completely_factor (const bigint & x)`

returns a prime factorization of x using a strategy which tries to apply an "optimal" combination of the above algorithms.

See also

`factorization< T >`, `bigint`, `single_factor< T >`, `rational_factorization`.

Warnings

ECM fails to factor integers that only consist of prime factors smaller than 1000. Therefore it is strongly recommended to use the function `sf_factor` or to ensure with the function `TrialDiv`, that there are no such small prime factors left before calling functions which use *ECM*.

Examples

```
#include <LiDIA/bigint.h>
#include <LiDIA/factorization.h>

int main()
{
    bigint f;
    factorization< bigint > u;

    cout << "Please enter f : "; cin >> f ;

    u = sf_factor(f);

    cout << "\nFactorization of f : " << u << endl;

    return 0;
}
```

For further examples please refer to `LiDIA/src/templates/factorization/bigint/fact_bi_appl.cc`.

Author

Volker Müller, based on previous implementations of Oliver Braun and Emre Binisik.

Chapter 8

Miscellaneous

modular_functions

Name

modular_functions ..... some number-theoretical functions

Abstract

Including `LiDIA/modular_functions.h` allows the computation of some number-theoretical functions like the modular function j or Dedekind's η-function.

Description

With this package the user can compute the value $g(\tau)$, where τ is in the upper complex half plane $\mathfrak{h}$ and g denotes one of the following functions:

- $g = j$ with the modular function j.
- $g = \eta$, where η denotes Dedekind's η-function.
- $g \in \{\mathfrak{f}, \mathfrak{f}_1, \mathfrak{f}_2\}$ with the Weber functions $\mathfrak{f}_i$.
- $g = \gamma_2$ with the cube root γ_2 of j which is real valued on the imaginary axis.

The values of these functions can be computed very efficiently, if τ is in the domain $D := \{z \in \mathfrak{h} : |\mathrm{Re}(\tau)| \leq 1/2, |\tau| \geq 1\}$. It is well known that each $\tau \in \mathfrak{h}$ is equivalent to some $\tau' \in D$ under the action of $\mathrm{SL}(2, \mathbb{Z})$, and transformation formulae for all functions g are known. If the user is sure that $\tau \in D$, he can set the boolean `is_fundamental` to `true` to avoid the application of the transformation.

`bigcomplex modular_j (const bigcomplex & τ, bool is_fundamental = false)`

returns the value $j(\tau)$ within the precision, which is set by `bigfloat::set_precision()`. If `is_fundamental` is set to `true`, then τ is assumed to be in the fundamental domain D. Otherwise, τ is first shifted to D.

`bigcomplex gamma_2 (const bigcomplex & τ, bool is_fundamental = false)`

returns the value $\gamma_2(\tau)$ within the precision, which is set by `bigfloat::set_precision()`. If `is_fundamental` is set to `true`, then τ is assumed to be in the fundamental domain D. Otherwise, τ is first shifted to D.

`bigcomplex dedekind_eta (const bigcomplex & τ, bool is_fundamental = false)`

returns the value $\eta(\tau)$ within the precision, which is set by `bigfloat::set_precision()`. If `is_fundamental` is set to `true`, then τ is assumed to be in the fundamental domain D. Otherwise, τ is first shifted to D.

`bigcomplex weber_f (const bigcomplex & τ, bool is_fundamental = false)`

returns the value $\mathfrak{f}(\tau)$ within the precision, which is set by `bigfloat::set_precision()`. If `is_fundamental` is set to `true`, then τ is assumed to be in the fundamental domain D. Otherwise, the relation $\mathfrak{f}(\tau) = \zeta_{48}^{-1} \cdot \frac{\eta((\tau+1)/2)}{\eta(\tau)}$ is used.

`bigcomplex weber_f1 (const bigcomplex & τ, bool is_fundamental = false)`

returns the value $\mathfrak{f}_1(\tau)$ within the precision, which is set by `bigfloat::set_precision()`. If `is_fundamental` is set to `true`, then τ is assumed to be in the fundamental domain D. Otherwise, the relation $\mathfrak{f}_1(\tau) = \frac{\eta(\tau/2)}{\eta(\tau)}$ is used.

`bigcomplex weber_f2 (const bigcomplex & τ, bool is_fundamental = false)`

returns the value $\mathfrak{f}_2(\tau)$ within the precision, which is set by `bigfloat::set_precision()`. If `is_fundamental` is set to `true`, then τ is assumed to be in the fundamental domain D. Otherwise, the relation $\mathfrak{f}_2(\tau) = \sqrt{2} \cdot \frac{\eta(2\tau)}{\eta(\tau)}$ is used.

See also

`bigcomplex`

Examples

```
#include <LiDIA/modular_functions.h>

using namespace LiDIA;

int
main()
{
    bigcomplex tau;

    std::cout << "Please enter a bigcomplex tau >> ";
    std::cin  >> tau;

    bigfloat::set_precision( 1000 );

    std::cout << "j( tau ) = " << modular_j( tau ) << std::endl;

    return( 0 );
}
```

Author

Harald Baier

number-theoretic functions

Name

number-theoretic functions ..... a collection of basic number-theoretic functions

Abstract

Including `LiDIA/nmbrthry_functions.h` allows the use of some basic number-theoretic functions, e.g., computing all divisors of an integer number.

Description

`sort_vector < bigint > divisors (rational_factorization & f)`

returns a sorted vector of all positive divisors of the number represented by f (in ascending order). If f is no prime factorization, the function tries to refine f to a prime factorization. In this case, the computed prime factorization is assigned to f.

`sort_vector < bigint > divisors (const bigint & N)`

returns a sorted vector of all positive divisors of N (in ascending order).

`sort_vector < bigint > all_divisors (rational_factorization & f)`

returns a sorted vector of all positive and negative divisors of the number represented by f (in ascending order). If f is no prime factorization, the function tries to refine f to a prime factorization. In this case, the computed prime factorization is assigned to f.

`sort_vector < bigint > all_divisors (const bigint & N)`

returns a sorted vector of all positive and negative divisors of N (in ascending order).

`sort_vector < bigint > square_free_divisors (rational_factorization & f)`

returns a sorted vector of all positive square-free divisors of the number represented by f (in ascending order). If f is no prime factorization, the function tries to refine f to a prime factorization. In this case, the computed prime factorization is assigned to f.

`sort_vector < bigint > square_free_divisors (const bigint & N)`

returns a sorted vector of all positive square-free divisors of N (in ascending order).

`sort_vector < bigint > all_square_free_divisors (rational_factorization & f)`

returns a sorted vector of all positive and negative square-free divisors of the number represented by f (in ascending order). If f is no prime factorization, the function tries to refine f to a prime factorization. In this case, the computed prime factorization is assigned to f.

`sort_vector < bigint > all_square_free_divisors (const bigint & N)`

returns a sorted vector of all positive and negative square-free divisors of N (in ascending order).

`sort_vector < bigint > square_divides_n_divisors (rational_factorization & f)`

returns a sorted vector of all positive square-free divisors of the number represented by f (in ascending order). If f is no prime factorization, the function tries to refine f to a prime factorization. In this case, the computed prime factorization is assigned to f.

`sort_vector < bigint > square_divides_n_divisors (const bigint & N)`

returns a sorted vector of all positive square-free divisors of N (in ascending order).

`sort_vector < bigint > all_square_divides_n_divisors (rational_factorization & f)`

returns a sorted vector of all positive and negative square-free divisors of the number represented by f (in ascending order). If f is no prime factorization, the function tries to refine f to a prime factorization. In this case, the computed prime factorization is assigned to f.

`sort_vector < bigint > all_square_divides_n_divisors (const bigint & N)`

returns a sorted vector of all positive and negative square-free divisors of N (in ascending order).

See also

`bigint`, `rational_factorization`

Examples

```
#include <LiDIA/nmbrthry_functions.h>

int main()
{
    bigint n;

    cout << "Please enter a bigint n = ";
    cin  >> n;

    cout << "all divisors of " << n << ": " << divisors(n) << endl;

    return 0;
}
```

Author

Markus Maurer, Volker Müller

crt_table/crt

Name

`crt_table`, `crt` ..... Chinese Remainder Theorem

Abstract

`crt_table` and `crt` are classes that allow explicit use of the Chinese Remainder Theorem.

A common method in computer algebra is to replace a multiprecision computation by several single-precision computations and one application of the Chinese Remainder Theorem. This is applicable for both, rational and modular integers. Thus classes `crt_table` and `crt` provide interfaces to type `bigint` and `bigmod` and to different objects over these types; i.e. single objects, pointer-arrays, `base_vector`s and `base_matrix`es.

The class `crt_table` provides internally stored tables which can be accessed by the class `crt`. The class `crt` can be seen as a black box that can reduce numbers (represented as `bigint` or `bigmod`) modulo single-precision prime numbers, as well as combine the modular results to the final result of type `bigint` or `bigmod`.

Description

The Chinese Remainder Theorem says, that given n moduli m_i $(1 \leq i \leq n)$ of pairwise coprime integers and a set of congruential equations $x \equiv c_i \pmod{m_i}$ $(1 \leq i \leq n)$ for an arbitrary integer x, there is a unique solution c for x modulo the product $M = \prod_{i=1}^{n} m_i$; say $x \equiv c \pmod{M}$.

Let us assume in this situation that we know an upper bound for the absolute value of x; $|x| \leq B$. Furthermore the number n of moduli shall be chosen in a way that $B < \frac{M}{2}$. If we then represent c by its absolutely least residue modulo M, we know that c and x coincide. Thus x is uniquely determined by the above congruential equations.

If we consider x to be the result of some algorithm, this implies an alternative strategy for computations with multiprecision integers. If we want to compute some results over the integers, and if we know an upper bound on their absolute values, we can choose an appropriate number of single-precision (pairwise coprime) moduli and perform the algorithm modulo each of them while using a suitable single-precision modular arithmetic. As a result of that we will obtain a set of congruential equations that we can use as an input to the Chinese Remainder Theorem. Its application will provide the correct multiprecision result of the computation.

Not only is this method suitable for rational integers, we can also easily extend it componentwise to vectors or matrices with integer entries. We then have to find an upper bound on *all* the coefficients of the respective object.

Moreover, this strategy is also applicable for multiprecision modular arithmetic. Here, we consider the residues as integers, compute the result of the problem over $\mathbb{Z}$ and accomplish by using some reduction-steps to compute residues again.

The classes `crt_table` and `crt` allow finding the solution of one application of the Chinese Remainder Theorem by successively presenting the residues c_i (in the notation above) to an object of type `crt`. The moduli m_i are prime numbers provided by the two classes. They can be generated automatically by initializing a `crt_table`

object with the upper bound B on the absolute values of the results. The system then chooses primes, until the product M of all of them exceeds $2 \cdot B$. Another possibiltiy is to explicitely store a vector of prime numbers into a `crt_table` object. If the product of these primes is M, this object can be used to compute integers of absolute value $\frac{M}{2}$ at most.

To allow prime numbers of arbitrary shape to be generated, the internal prime number generator may be replaced by a corresponding user-defined function.

A standard application for these classes looks as follows:

1. Determine an upper bound B for the absolute values of the numbers that result from your algorithm. For computations of type `bigmod` consider the residues as rational integers.
2. Create an object T of type `crt_table` and initialize it with bound B and when computing with `bigmod` also with the corresponding modulus (by using a constructor or `init()` member function of class `crt_table`). Then T contains a list of prime numbers you have to use for modular computations.
3. Create an object C of type `crt` and initialize it with the previously generated object T, thus providing C with all the internal data of T (by using a constructor or `init()` member function of class `crt`).
4. Perform the computation modulo any prime contained in the internal list of T (see member function `get_prime()`) and store the respective results in C (by using a suitable `combine()` member function).
5. Read the final results out of C (by using a suitable `get_result()` member function).

Thus, this method does *not* require to store each residue modulo every prime before an application of the Chinese Remainder Theorem.

Class crt_table

An instance of type `crt_table` can be initialized by either a constructor or by a suitable `init()` member function. These functions require an upper bound B for the absolute values of the results of your computation. When computing with type `bigmod` the residues are considered as rational integers, and the global modulus must then be given to each initialization as parameter P. Using a modulus when initializing an instance sets its mode to `MODULAR`; otherwise, the mode will be set to `INTEGER`. The mode of an instance defines whether a `crt` object by which it is used, provides interfaces to `bigint` (`INTEGER`) or to `bigmod` (`MODULAR`) only.

Any object of class `crt_table` contains a reference counter to control whether or not it is used by a `crt` object. One may initialize (or clear) an instance only if this counter is zero.

Constructors/Destructor

`ct crt_table ()`

creates an empty instance, no tables are initialized.

`ct crt_table (const bigint &` B`)`

generates primes, beginning with initial seed $2^{31} - 1$ to the internal prime number generator. When using the default prime generator, the prime table will consist of subsequent prime numbers less than $2^{31} - 1$ and in descending order. This function initializes the internal tables for computations where the absolute values of the later results must not exceed B. (`INTEGER` mode)

`ct crt_table (const bigint &` B`, const sdigit &` *prime_bound*`)`

generates primes, beginning with initial seed *prime_bound* to the internal prime number generator; *prime_bound* must be positve. When using the default prime generator, the prime table will consist of subsequent prime numbers less than *prime_bound* and in descending order. This function initializes the internal tables for computations where the absolute values of the later results must not exceed B. (`INTEGER` mode)

`ct crt_table (const base_vector< sdigit > & `*Primes*`)`

copies all numbers in *Primes* into the internal prime table and initializes the internal tables. If any of these numbers is negative, its absolute value will be considered; if *Primes* contains at least two entries with a non-trivial greatest common divisor, a call to the `lidia_error_handler` will be invoked. The absolute values of the later results must be bounded by $\frac{M}{2}$, where M is the product of all numbers in vector *Primes* (`INTEGER` mode). (Actually, those do not have to be primes but only pairwise coprime.)

`ct crt_table (const bigint & `B`, const bigint & `P`)`

generates primes, beginning with initial seed $2^{31} - 1$ to the internal prime number generator. When using the default prime generator, the prime table will consist of subsequent prime numbers less than $2^{31} - 1$ and in descending order. This function initializes the internal tables for computations where the absolute values of the later results must not exceed B. If P is non-zero, precomputations are performed modulo P (`MODULAR` mode); otherwise P will be ignored (`INTEGER` mode).

`ct crt_table (const bigint & `B`, const sdigit & `*prime_bound*`, const bigint & `P`)`

generates primes, beginning with initial seed *prime_bound* to the internal prime number generator; *prime_bound* must be positve. When using the default prime generator, the prime table will consist of subsequent prime numbers less than *prime_bound* and in descending order. This function initializes the internal tables for computations where the absolute values of the later results must not exceed B. If P is non-zero, precomputations are performed modulo P (`MODULAR` mode); otherwise P will be ignored (`INTEGER` mode).

`ct crt_table (const base_vector< sdigit > & `*Primes*`, const bigint & `P`)`

copies all numbers in *Primes* into the internal prime table and initializes the internal tables. If any of these numbers is negative, its absolute value will be considered; if *Primes* contains at least two entries with a non-trivial greatest common divisor, a call to the `lidia_error_handler` will be invoked. The absolute values of the later results must be bounded by $\frac{M}{2}$, where M is the product of all numbers in vector *Primes* (`INTEGER` mode). (Actually, those do not have to be primes but only pairwise coprime.) If P is non-zero, precomputations are performed modulo P (`MODULAR` mode); otherwise, P will be ignored (`INTEGER` mode).

`dt ~crt_table ()`

Initialization

The initialization functions allow to create the internal data of a `crt_table` object after its definition or to reset an object for new computations. Modifying an object for which there is still a reference in some instance of class `crt`, invokes the `lidia_error_handler`.

Let a be of type `crt_table`.

`void `a`.init (const bigint & `B`)`

initializes a like the constructor `crt_table(`B`)`.

`void `a`.init (const bigint & `B`, const sdigit & `*prime_bound*`)`

initializes a like the constructor `crt_table(`B`, `*prime_bound*`)`.

`void `a`.init (const base_vector< sdigit > & `*Primes*`)`

initializes a like the constructor `crt_table(`*Primes*`)`.

`void `a`.init (const bigint & `B`, const bigint & `P`)`

initializes a like the constructor `crt_table(`B`, `P`)`.

`void` *a*`.init (const bigint &` *B*`, const sdigit &` *prime_bound*`, const bigint &` *P*`)`
initializes *a* like the constructor `crt_table(`*B*`,` *prime_bound*`,` *P*`)`.

`void` *a*`.init (const base_vector< sdigit > &` *Primes*`, const bigint &` *P*`)`
initializes *a* like the constructor `crt_table(`*Primes*`,` *P*`)`.

`void` *a*`.clear ()`
deletes all internally stored tables.

Access Methods

Let *a* be of type `crt_table`.

`lidia_size_t` *a*`.number_of_primes () const`
returns number of internally stored primes for which precomputations have been done.

`sdigit` *a*`.get_prime (lidia_size_t` *ix*`) const`
returns prime with index *ix* from the table. The `lidia_error_handler` will be invoked if index is negative or not less than *a*`.number_of_primes()`.

`lidia_size_t` *a*`.how_much_to_use (bigint` *B*`) const`
If each result of the computation has an absolute value of at most B but a has been initialized with some greater bound, it might be sufficient to regard computations only modulo the first l primes from the internal table instead of using all of them. In that case, the function returns l; it returns -1 if the table is not large enough.

This option is only possible in `INTEGER` mode. When initialized with a modulus, one always has to use every prime that has been generated before. Thus in `MODULAR` mode, always the number of internal primes will be returned.

`void` *a*`.set_prime_generator (sdigit (*np) (const sdigit &` *q*`) = nil)`
changes the internal prime number generator to the function *np* is pointing to. This must be a function which has a `const` reference to an `sdigit` as the only argument and returns an `sdigit`. Internally, only positive prime numbers are used. Thus, any negative return-value of `np` will be multiplied by -1. Furthermore, one has to consider the fact, that $2^{31} - 1$ will be given to this function by a constructor or an `init()` member function. Thus *np* should be able to accept this value as an input. If no function pointer is given, the prime generator will be reset to the default routine, which returns the next prime less than *q*.

Class crt

Member functions of this class are not applicable to an instance, unless it is assigned a reference to an object T of class `crt_table`. (The latter must already be initialized.) This can be done by means of either a constructor or an `init()` member function.

Depending on the mode (`INTEGER` or `MODULAR`) of T, all input data and final results will be assumed to be of type `bigint` or `bigmod`. Only the corresponding member functions are enabled.

Constructors/Destructor

`ct crt ()`
creates an empty instance of the class crt.

`ct crt (crt_table & `M`)`

creates an instance and stores a reference to M. The reference counter of M will therefore be modified.

`dt ~crt ()`

deletes all internal data and decreases the reference counter of the corresponding `crt_table` object by one.

Initialization

Let a be of type `crt`.

`void `a`.init (crt_table & `M`)`

stores a reference to table M, that will then be used during the reduction and combining steps. If M has not been initialized before, this will invoke the `lidia_error_handler`.

`void `a`.reset ()`

deletes any information that is stored in a, but leaves the reference to an `crt_table` object unchanged.

`void `a`.clear ()`

deletes any information that is stored in a, including the reference to the `crt_table` object.

Basic Methods and Functions

`sdigit `a`.get_prime (lidia_size_t `ix`) const`

returns the prime with index ix from underlying `crt_table` object. The `lidia_error_handler` will be invoked if ix is negative or not less than the number of internal primes.

`lidia_size_t `a`.number_of_primes () const`

returns number of internally stored primes for which precomputations have been made in the uderlying `crt_table` object.

`lidia_size_t `a`.how_much_to_use (const bigint & `B`) const`

If each result of the computation has an absolute value of at most B but the `crt_table` object referred to has been initialized with some greater bound, it might be sufficient to regard computations only modulo the first l primes from the internal table, instead of using all of them. In that case, the function returns l. It returns -1 if the table is not large enough.

This option is only possible in `INTEGER` mode. In `MODULAR` mode one always has to use every prime that has been generated before. Thus in the latter case, always the number of internal primes will be returned.

Arithmetical Operations

Compute Residues

Let a be of type `crt`.

The `reduce()` member functions provide easy reduction modulo the internal primes. They do not modify any information stored in a. To compute the residues, the remainder functions of class `bigint` are used.

`void` *a*`.reduce (sdigit &` *small*`, const bigint &` *big*`, lidia_size_t` *ix*`) const`

small $\leftarrow$ *big* mod *p*, for $p =$ *a*`.get_prime(`*ix*`)`.

`void` *a*`.reduce (sdigit*` *small_vec*`, const bigint*` *big_vec*`, long` *bsize*`, lidia_size_t` *ix*`) const`

small_vec[*i*] $\leftarrow$ *big_vec*[*i*] mod *p* for $0 \leq i <$ *bsize*, and $p =$ *a*`.get_prime(`*ix*`)`. This function does not allocate any space for *small_vec*. Thus it must be a pointer to an array of appropriate size.

`void` *a*`.reduce (base_vector< sdigit > &` *small_vec*`, const base_vector< bigint > &` *big_vec*`, lidia_size_t` *ix*`) const`

small_vec[*i*] $\leftarrow$ *big_vec*[*i*] mod *p* for $0 \leq i <$ *big_vec*`.size()`, and $p =$ *a*`.get_prime(`*ix*`)`. The size and capacity of *small_vec* are set to the size of *big_vec*.

`void` *a*`.reduce (base_matrix< sdigit > &` *small_mat*`, const base_matrix< bigint > &` *big_mat*`, lidia_size_t` *ix*`) const`

small_mat[*i*][*j*] $\leftarrow$ *big_mat*[*i*][*j*] mod *p* for $0 \leq i < r, 0 \leq j < c$, and $p =$ *a*`.get_prime(`*ix*`)`. Here r and c represent the numbers of rows and columns of matrix *big_mat*. The size of *small_mat* is adjusted to that of *big_mat*.

`void` *a*`.reduce (sdigit &` *small*`, const bigmod &` *big*`, lidia_size_t` *ix*`) const`

small $\leftarrow$ *big*`.mantissa()` mod *p* for $p =$ *a*`.get_prime(`*ix*`)`.

`void` *a*`.reduce (sdigit*` *small_vec*`, const bigmod*` *big_vec*`, long` *bsize*`, lidia_size_t` *ix*`) const`

small_vec[*i*] $\leftarrow$ *big_vec*[*i*]`.mantissa()` mod *p* for $0 \leq i <$ *bsize*, and $p =$ *a*`.get_prime(`*ix*`)`. This function does not allocate any space for *small_vec*. Thus it must be a pointer to an array of appropriate size.

`void` *a*`.reduce (base_vector< sdigit > &` *small_vec*`, const base_vector< bigmod > &` *big_vec*`, lidia_size_t` *ix*`) const`

small_vec[*i*] $\leftarrow$ *big_vec*[*i*]`.mantissa()` mod *p* for $0 \leq i <$ *big_vec*`.size()`, and $p =$ *a*`.get_prime(`*ix*`)`. The size and capacity of *small_vec* are set to the size of *big_vec*.

`void` *a*`.reduce (base_matrix< sdigit > &` *small_mat*`, const base_matrix< bigmod > &` *big_mat*`, lidia_size_t` *ix*`) const`

small_mat[*i*][*j*] $\leftarrow$ *big_mat*[*i*][*j*]`.mantissa()` mod *p* for $0 \leq i < r, 0 \leq j < c$, and $p =$ *a*`.get_prime(`*ix*`)`. Here, r and c represent the number of rows and columns of matrix *big_mat*. The size of *small_mat* is adjusted to that of *big_mat*.

Add Congruencial Equations

The results of computations modulo a particular prime are combined to the final result via the `combine()` member functions. You may not combine a result modulo a specific prime twice. The first call to any of these functions defines the data structure of the input and final result: single, pointer array, `base_vector`, or `base_matrix`. If, for example, the first function call is *a*`.combine(`*u*`,` ix_1`)` with an `sdigit` *u*, then every attempt to use *a*`.combine(`*uvec*`,` ix_2`)` with *uvec* as a `base_vector< sdigit >` will invoke the `lidia_error_handler`. Furthermore, you may not use objects of different size in subsequent calls to the function. This invokes the `lidia_error_handler`.

To prematurely terminate a computation and allow the use of a `crt` object with some other input, one may use either of the two functions `reset()` and `clear()`.

`void` *a*`.combine (const sdigit &` *small*`, lidia_size_t` *ix*`)`

stores the result modulo *a*`.get_prime(`*ix*`)`.

`void` *a*`.combine (const sdigit*` *small_vec*`, long` *l*`, lidia_size_t` *ix*`)`

stores the result modulo *a*`.get_prime(`*ix*`)`.

`void` *a*`.combine (const base_vector< sdigit > &` *small_vec*`, lidia_size_t` *ix*`)`

stores the result modulo *a*`.get_prime(`*ix*`)`.

`void` *a*`.combine (const base_matrix< sdigit > &` *small_mat*`, lidia_size_t` *ix*`)`

stores the result modulo *a*`.get_prime(`*ix*`)`.

Reading Results

Having defined the congruencial equations, one can now compute the result of the Chinese Remainder Theorem. In MODULAR mode, congruences must be defined for each internal prime. The INTEGER mode allows to choose arbitrarily many congruences. Rational integers (`bigint`) will then be computed as the absolutely least residues modulo the product of all primes, that have been used before.

A call of the `get_result()` member function is only possible, if its argument corresponds to preceeding uses of `combine()`. I.e. if congruences have been defined for matrices, then *a*`.get_result(`*big_mat*`)` is the only applicable function to obtain the solution.

After a call to the function `get_result()`, the information about the object that was used will no longer be stored. Thus *a* can later arbitrarily be used again.

`void` *a*`.get_result (bigint &` *big*`)`

computes in *big* the result of the Chinese Remainder Theorem according to the previously combined congruences.

`void` *a*`.get_result (bigint*&` *big_vec*`, lidia_size_t &` *l*`)`

computes in *big_vec*$[i]$ $(0 \leq i < l)$ the result of the Chinese Remainder Theorem according to the previously combined congruences. First, l must be the size of the array *big_vec* is pointing to. If it differs from the size of the internal object, *big_vec* will be reallocated appropriately and its new size will be assigned to l.

`void` *a*`.get_result (base_vector< bigint > &` *big_vec*`)`

computes in *big_vec*$[i]$ $(0 \leq i < l)$ the result of the Chinese Remainder Theorem according to the previously combined congruences. The capacity and size of vector *big_vec* will previously be set to l, which is the size of the internal object.

`void` *a*`.get_result (base_matrix< bigint > &` *big_mat*`)`

computes in *big_mat*$[i][j]$ $(0 \leq i < r, 0 \leq j < c)$ the result of the Chinese Remainder Theorem according to the previously combined congruences. The number of rows and colums of *big_mat* will previously be set to r and c, respectively. Here r and c represent the size of the internal matrix.

`void` *a*`.get_result (bigmod &` *big*`)`

computes in *big* the result of the Chinese Remainder Theorem according to the previously combined congruences.

`void` *a*`.get_result (bigmod*&` *big_vec*`, lidia_size_t &` *l*`)`

computes in *big_vec*$[i]$ $(0 \leq i < l)$ the result of the Chinese Remainder Theorem according to the previously combined congruences. First, l must be the size of the array *big_vec* is pointing to. If it differs from the size of the internal object, *big_vec* will be reallocated appropriately and the new size will be assigned to l.

`void` *a*`.get_result (base_vector< bigmod > &` *big–vec*`)`

computes in *big–vec*$[i]$ $(0 \leq i < l)$ the result of the Chinese Remainder Theorem according to the previously combined congruences. The capacity and size of vector *big–vec* will previously be set to l, which is the size of the internal object.

`void` *a*`.get_result (base_matrix< bigmod > &` *big–mat*`)`

computes in *big–mat*$[i][j]$ $(0 \leq i < r, 0 \leq j < c)$ the result of the Chinese Remainder Theorem according to the previously combined congruences. The number of rows and colums of *big–mat* will previously be set to r and c, respectively. Here r and c represent the size of the internal matrix.

Examples

```
#include <LiDIA/crt.h>

int main ()
{
    int i, n;
    bigint big, B;
    sdigit small;

    crt_table T;
    crt       C;

    cout << " Bound for absolute value : ";
    cin  >> B;

    T.init ( B );
    C.init ( T );
    n = C.number_of_primes();

    cout << " " << n << " congruences needed\n\n";

    for (i = 0; i < n; i++) {
        cout << " Congruence modulo " ;
        cout << C.get_prime(i) << " : ";
        cin >> small;
        C.combine (small, i);
    }
    C.get_result ( big );
    cout << "\n Solution : " << big << endl;

    return 0;
}
```

For further reference please refer to `LiDIA/src/simple_classes/chinese_rem/crt_appl.cc`

See also

`base_vector`, `base_matrix`, `bigint`, `bigmod`

Author

Frank J. Lehmann, Thomas Pfahler

prime_list

Name

`prime_list` ..... prime number calculation

Abstract

`prime_list` is a class that provides efficient calculation and storage of prime number intervals in memory. Loading and saving lists from/to files is also supported.

Description

`prime_list` implements five algorithms to calculate prime number intervals with different speed and memory usage.

algorithm	memory requirement	example
sieve of Erathostenes (mode 'E')	$(ub - lb) + \min(lb, \sqrt{ub})$	1000000
$6k \pm 1$ sieve (mode 'K')	$((ub - lb) + \min(lb, \sqrt{ub}))/3$	333333
sieve of Erathostenes bit-level (mode 'B')	$((ub - lb) + \min(lb, \sqrt{ub}))/8$	125000
$6k \pm 1$ bit sieve (mode '6', default)	$((ub - lb) + \min(lb, \sqrt{ub}))/24$	41666
interval sieve (mode 'I')	$\min(1000000, \max((ub - lb), \sqrt{ub})) + \sqrt{ub}/\log(\sqrt{ub}) \cdot 4$	1000574

"memory requirement" estimates the approximate amount of memory in bytes used *during calculation* of an interval from lb (lower bound) to ub (upper bound) on a 32-bit machine.

The column example shows the memory requirement for an interval of primes from 2 to 10^6.

Under normal conditions the $6k \pm 1$ bit sieve is the fastest algorithm and therefore used by default. To get an exact comparison of the algorithms on your machine, you can run the benchmark application `prime_list_bench_appl`.

After calculation only the differences between neighbored primes are stored, so (in the default configuration) each prime in the list needs just one byte of memory. Additionally the differences are organized in blocks to speed up random access to the list.

Besides memory the used data types limit the primes that can be calculated. By default the upper bound is 2^{32} on a 32-bit machine and 2^{38} on a 64-bit machine. But `prime_list` can easily be reconfigured to support greater primes by changing the following types:

name	default type
PRIME_LIST_NUMBER	unsigned long
PRIME_LIST_COUNTER	long
PRIME_LIST_FLOAT_NUMBER	double
PRIME_LIST_DIFF	unsigned char

For more information, see `LiDIA/include/LiDIA/prime_list.h`.

Constructors/Destructor

`ct prime_list ()`
Creates an empty list.

`ct prime_list (PRIME_LIST_NUMBER` *upper_bound*`, char` *mode* `= '6')`
Creates a list with primes from 2 to *upper_bound*. Uses the algorithm specified by *mode*.

`ct prime_list (PRIME_LIST_NUMBER` *lower_bound*`, PRIME_LIST_NUMBER` *upper_bound*`,`
`char` *mode* `= '6')`
Creates a list with primes from *lower_bound* to *upper_bound*. Uses the algorithm specified by *mode*.

`ct prime_list (const char *filename, lidia_size_t` *max_number_of_primes* `= 0)`
Loads primes from a file. The numbers in the file must be primes, sorted in ascending or descending order and there may be no primes missing inside the interval. The number of primes being read can be limited by *max_number_of_primes*.

`ct prime_list (const prime_list &` *A*`)`
Copies primes from *A*.

`dt ~prime_list ()`
Deletes the `prime_list` object.

Basic Methods and Functions

Let *pl* be of type `prime_list`.

`PRIME_LIST_NUMBER` *pl*`.get_lower_bound () const`
Returns the lower bound of *pl*.

`PRIME_LIST_NUMBER` *pl*`.get_upper_bound () const`
Returns the upper bound of *pl*.

`void` *pl*`.set_lower_bound (PRIME_LIST_NUMBER` *lower_bound*`, char` *mode* `= '6')`
Sets a new lower bound for *pl* and deletes primes or calculates new ones (with the algorithm specified by *mode*) as necessary.

`void` *pl*`.set_upper_bound (PRIME_LIST_NUMBER` *upper_bound*`, char` *mode* `= '6')`
Sets a new upper bound for *pl* and deletes primes or calculates new ones (with the algorithm specified by *mode*) as necessary.

`void` *pl*`.resize (PRIME_LIST_NUMBER` *lower_bound*`, PRIME_LIST_NUMBER` *upper_bound*`,`
`char` *mode* `= '6')`

Sets a new lower and upper bound for *pl* and deletes primes and/or calculates new ones (with the algorithm specified by *mode*) as necessary.

`bool` *pl*`.is_empty () const`

Returns `true` if *pl* is empty, `false` otherwise.

`lidia_size_t` *pl*`.get_number_of_primes () const`

Returns the number of primes in *pl*.

`bool` *pl*`.is_element (PRIME_LIST_NUMBER` *prime*`) const`

Returns `true` if *pl* contains *prime*, `false` otherwise.

`lidia_size_t` *pl*`.get_index(PRIME_LIST_NUMBER` *prime* `() const`

Returns the index of *prime* in *pl* or -1 if *prime* is not in *pl*.

Access Methods

Let *pl* be of type `prime_list`.

`PRIME_LIST_NUMBER` *pl*`.get_current_prime () const`

Returns the current prime of *pl*. This is the prime returned by the latest called $get_???_prime$ function.

`lidia_size_t` *pl*`.get_current_index () const`

Returns the index of the current prime of *pl*.

`PRIME_LIST_NUMBER` *pl*`.get_first_prime () const`

Returns the first prime in *pl* or 0 if *pl* is empty. Updates the current prime.

`PRIME_LIST_NUMBER` *pl*`.get_first_prime (PRIME_LIST_NUMBER` *prime*`) const`

Returns the first number in *pl* which is greater than or equal to *prime* or 0 if *prime* is greater than the last prime in *pl*. Updates the current prime.

`PRIME_LIST_NUMBER` *pl*`.get_next_prime () const`

Returns the number that follows the current prime in *pl* or 0 if the current prime is already the last prime in *pl*. Updates the current prime.

`PRIME_LIST_NUMBER` *pl*`.get_last_prime () const`

Returns the last prime in *pl* or 0 if *pl* is empty. Updates the current prime.

`PRIME_LIST_NUMBER` *pl*`.get_last_prime (PRIME_LIST_NUMBER` *prime*`) const`

Returns the last number in *pl* which is less than or equal to *prime* or 0 if *prime* is less than the first prime in *pl*. Updates the current prime.

`PRIME_LIST_NUMBER` *pl*`.get_prev_prime () const`

Returns the number that precedes the current prime in *pl* or 0 if the current prime is already the first prime in *pl*. Updates the current prime.

`PRIME_LIST_NUMBER` *pl*`.get_prime (lidia_size_t` *index*`) const`

Returns the number at position *index* (starting with 0) in *pl* or 0 if *index* is out of bounds. Updates the current prime.

The overloaded operator `[]` can be used instead of `get_prime`.

Input/Output

Let *pl* be of type `prime_list`.

`void` *pl*`.load_from_file (const char *filename, lidia_size_t` *max_number_of_primes* `= 0)`

Deletes the current content of *pl* and loads primes from a file. The numbers in the file must be primes, sorted in ascending or descending order and there may be no primes missing inside the interval. The number of primes being read can be limited by *max_number_of_primes*.

`void` *pl*`.save_to_file (const char *filename, bool descending = false) const`

Writes all primes in *pl* to a file in ascending or descending order (according to the value of `descending`) with one number per line. If the file already exists, it is overwritten.

Assignments

Let *pl* be of type `prime_list`.

`void` *pl*`.assign (const prime_list &` *A*`)`

Deletes the current content of *pl* and copies primes from *A*.

The overloaded operator `=` can be used instead of `assign`.

Examples

```
#include <LiDIA/prime_list.h>

int main()
{
    unsigned long lb, ub, p;

    cout << "Please enter lower bound: "; cin >> lb ;
    cout << "Please enter upper bound: "; cin >> ub ;
    cout << endl;
    prime_list pl(lb, ub);
    p = pl.get_first_prime();
    while (p)
    {
        cout << p << endl;
        p = pl.get_next_prime();
    }

    return 0;
}
```

Author

Dirk Schramm

power_functions

Name

`power functions` .....a collection of template functions for exponentiation

Abstract

Including `LiDIA/power_functions.h` allows the use of some power functions, e.g., left-to-right and right-to-left exponentiation.

Description

Exponentiation is a very basic routine that is used for a variety of mathematical objects. Therefore, we have written template versions of some powering methods to compute a^b, where b can be of type `unsigned long` or `bigint` and the element a must be of type `T` that provides the functions

1. a`.is_one()`
2. a`.assign_one()`
3. `operator = (const T &)`
4. `multiply (T &, const T &, const T &)`
5. `square (T &, const T &, const T &)`.

More precisely, these methods can be applied to elements of a monoid, where the term *one* stands for the neutral element of the monoid.

We assume, that every class, which uses these routines, implements their own power function, that handles the case of negative exponents appropriately and then calls the template function. To avoid name conflicts, all functions have a prefix `lidia_`.

In general, you should use the functions `lidia_power(...)`, because they are interface functions, that call the fastest method. If a new, faster routine is implemented, the interface function will be changed appropriately and you don't have to change your own code.

`LiDIA/include/LiDIA/power_functions.cc` contains the implementation of the functions.

`void lidia_power (T &` c`, const T &` a`, const E &` b`)`

interface function that computes $c \leftarrow a^b$ by calling `lidia_power_left_to_right`. E can be either of type `bigint` or `unsigned long`.

`void lidia_power (T & c, const T & a, const E & b, void (*fast_mul) (T &, const T &, const T &))`

interface function that computes $c \leftarrow a^b$ by calling `lidia_power_left_to_right`. E can be either of type `bigint` or `unsigned long`.

`void lidia_power_left_to_right (T & c, const T & a, bigint b)`

uses repeated squaring, reading the bits from the highest to the lowest bit, to compute $c \leftarrow a^b$. If the exponent b is negative, the `lidia_error_handler` is called.

`void lidia_power_left_to_right (T & c, const T & a, unsigned long b)`

uses repeated squaring, reading the bits from the highest to lowest bit, to compute $c \leftarrow a^b$.

`void lidia_power_left_to_right (T & c, const T & a, bigint b, void (*fast_mul) (T &, const T &, const T &))`

uses repeated squaring, reading the bits from the highest to the lowest bit, to compute $c \leftarrow a^b$. It uses the function `fast_mul` to multiply the numbers, if the current examined bit is one. If the exponent b is negative, the `lidia_error_handler` is called.

`void lidia_power_left_to_right (T & c, const T & a, unsigned long b, void (*fast_mul) (T &, const T &, const T &))`

uses repeated squaring, reading the bits from the highest to lowest bit, to compute $c \leftarrow a^b$. It uses the function `fast_mul` to multiply the numbers, if the current examined bit is one.

`void lidia_power_right_to_left (T & c, const T & a, bigint b)`

uses repeated squaring, reading the bits from the lowest to highest bit, to compute $c \leftarrow a^b$. If the exponent b is negative, the `lidia_error_handler` is called.

`void lidia_power_right_to_left (T & c, const T & a, unsigned long b)`

uses repeated squaring, reading the bits from the lowest to highest bit, to compute $c \leftarrow a^b$.

`void lidia_power_sliding_window (T & c, const T & a, const bigint & b, unsigned long k)`

compute $c \leftarrow a^b$ by repeated squaring and multipliciation with precomputed powers of a, using a window of size k. Note that this algorithm precomputes $2^k - 1$ powers of a. If the exponent b is negative, the `lidia_error_handler` is called. If the window size k is one, this algorithm is equivalent to `lidia_power_left_to_right`.

See also

`bigint`

Examples

```
#include <LiDIA/bigint.h>
#include <LiDIA/random_generator.h>
#include <LiDIA/power_functions.h>
#include <LiDIA/timer.h>
#include <assert.h>

int main()
```

```
{
    bigint powl, powr;
    bigint a;
    long exp;
    bigint bigint_exp;

    random_generator rg;

    cout << "Testing repeated squaring left_to_right ";
    cout << "and right_to_left using bigints.\n";
    cout << "Base:     10 decimal digits.\n";
    cout << "Exponent: less than 100000.\n";
    cout << endl;
    cout << "This may take a while ...\n";
    cout << endl;
    cout.flush();

    // num, den randomly chosen, less than 10^10;
    bigint tmp = 10;
    lidia_power(tmp,tmp,tmp);

    a = randomize(tmp);

    // exp randomly chosen
    rg >> exp;

    exp %= 100000;

    bigint_exp = exp;


    timer T1, T2;

    T1.start_timer();
    lidia_power_right_to_left(powr, a, bigint_exp);
    T1.stop_timer();

    T2.start_timer();
    lidia_power_left_to_right(powl, a, bigint_exp);
    T2.stop_timer();

    cout << "power_right_to_left(bigint): " << T1 << endl;
    cout << "power_left_to_right(bigint): " << T2 << endl;

    // verify result
    assert(powl == powr);

    T1.start_timer();
    lidia_power_right_to_left(powr, a, exp);
    T1.stop_timer();

    // verify result
    assert(powl == powr);

    T2.start_timer();
    lidia_power_left_to_right(powl, a, exp);
    T2.stop_timer();
```

```
        cout << "power_right_to_left(long): " << T1 << endl;
        cout << "power_left_to_right(long): " << T2 << endl;

        // verify result
        assert(powl == powr);

        return 0;
    }
```

See LiDIA/src/templates/template_lib/power_functions_appl.cc.

Author

Markus Maurer, Stefan Neis

Chapter 9

System and Portability

timer

Name

`timer` ..... timer class

Abstract

`timer` is a class of timer utilities. It can be used for example to calculate the elapsed time while executing a function, the time spent in the system and the time spent in the execution of the function.

Description

`timer` is a class which simulates the functions of a chronometer. An instance T of the data type `timer` is an object consisting of the integer variables `user_time`, `sys_time` and local variables which are used to store elapsed times.

A possible scenario of the use of `timer` is to start it at any place t_1 of the program then stop it at t_2 and print the elapsed real, system and user time between t_2 and t_1. Continuing the timing and stopping the timer once again at t_3 stores in `T` the sum of the elapsed times between t_1 and t_2 and t_2 and t_3.

Constructors/Destructor

`ct timer ()`

`dt ~timer ()`

Initialization

`int T.set_print_mode (int` m`)`

this function sets the output mode m of the `T.print()` function for `T` and returns the old print mode. There are two predefined output modes: `TIME_MODE` and `HMS_MODE`. The output of the `T.print()` function when `TIME_MODE` is set, has the following format:

`rrr real uuu user sss sys`,

where `rrr, uuu, sss` denote the real, user and system time respectively (in milliseconds). In `HMS_MODE` the `T.print()` function outputs the elapsed real time in the format

`hhh hour, mmm min, nnn sec, ttt msec`,

where hhh, mmm, nnn and ttt denote the consumed hours, minutes, seconds and milliseconds respectively.

`void T.start_timer ()`
this function initializes the timer with the current time and sets the elapsed time for this timer to zero.

Access Methods

`int T.get_print_mode () const`
returns the output mode of the timer `T`. This mode is used by the `T.print()` function.

`long T.user_time () const`
returns the user time elapsed during the last call of `T.start_timer()`. The time is given in milliseconds.

`long T.sys_time () const`
returns the system time elapsed during the last call of `T.start_timer()`. The time is given in milliseconds.

`long T.real_time () const`
returns `T.user_time()` + `T.sys_time()`.

High-Level Methods and Functions

`void T.stop_timer ()`
this function adds in `T` the elapsed time (user + system) since the last call of `T.start_timer()` or `T.cont_timer()`.

`void T.cont_timer ()`
this function sets the starting point for timing to the current time without changing the elapsed time in `T`.

For doing statistical computations, the operators `+`, `-`, and `/` with a variable of type `double` are available. Note that negative results are set to zero.

Input/Output

The `ostream` operator `<<` is overloaded. Furthermore there exists the following member function for the output:

`void T.print (ostream & out = cout) const`
prints the times elapsed between the last subsequent calls of `T.start_timer()` and `T.stop_timer()` according to the print mode of `T`. The default output stream is the standard output.

See also

getrusage(2)

Warnings

Subsequent calls of `stop_timer()` produce wrong results.

Examples

```
#include <LiDIA/timer.h>

int main()
{
    long i;
    timer x;

    x.set_print_mode(HMS_MODE);

    x.start_timer();
    for (i=0; i < 0x7fffff; i++);
    x.stop_timer();
    cout << endl;
    x.print();
    cout << endl;
    x.cont_timer();
    for (i=0; i < 0x7fffff; i++);
    x.stop_timer();
    cout << endl;
    x.print();
    cout << endl;

    return 0;
}
```

For further references please refer to `LiDIA/src/system/timer_appl.cc`

Author

Thomas Papanikolaou

system_handlers

Name

`system handlers` ..... warning, memory and debug handlers as well as legacy error handlers

Abstract

LiDIA's system handlers are functions which allow easy warning, memory and debug handling in LiDIA programs. LiDIA's error handling routines used to follow the same pattern. If you configured LiDIA with `--enable-exceptions=no`, then they still do. Otherwise, LiDIA's subroutines will throw exceptions derived from `LiDIA::BasicError`.

Description

In the following let `xxx` stand for `lidia_warning, lidia_error, lidia_memory` or `lidia_debug`. A `xxx_handler` in LiDIA consists of three parts:

`void default_xxx_handler ()`
: this is the function which realizes the handling proposed by the LiDIA system. For example, the `default_error_handler` prints an error message and then exits the program.

`xxx_handler_ptr xxx_handler ()`
: this is a pointer which by default points to the `default_xxx_handler`.

`xxx_handler_ptr set_xxx_handler (xxx_handler_ptr)`
: this is a function which allows to replace the default handler by a user-defined handler. The new handler must have the type `xxx_handler_ptr`.

According to the description above, the following handlers are available:

```
typedef void (*error_handler_ptr)(char *, char *);
error_handler_ptr lidia_error_handler;
```

The *lidia_error_handler* must be called with two character pointers (strings). The first string should include the name of the class in which the error occured and the second the exact position and the description of the error. The `lidia_error_handler` produces an error message of the format

```
lidia_error_handler::classname::exact position and description
```

and exits using `abort()`.

An extended version of `lidia_error_handler` is `lidia_error_handler_c`. This function must be called with two character pointers and a non-empty, valid piece of C++ code (commas are not allowed in this code). On call, it executes the code and the `lidia_error_handler`. `lidia_error_handler_c` is implemented by the macro

```
#define lidia_error_handler_c(f, m, code)\
{ code; lidia_error_handler(f, m); }
```

The `lidia_warning_handler` and the `lidia_debug_handler` work in the same way but in contrast to the `error_handler` they do not abort execution. To include warning and debug handling in LiDIA, the following compilation flags have to be defined before building the library: `-DLIDIA_WARNINGS -DLIDIA_DEBUG=level`

`level` can be any integer greater than zero. The greater the `level` the less debug information will be produced on execution. The default value of `LIDIA_DEBUG` is 1 (print all debug information).

```
#define debug_handler(file, msg)\
lidia_debug_handler(file, msg)

#define debug_handler_l(file, msg, level)\
if (LIDIA_DEBUG <= level)\
lidia_debug_handler(file, msg)

#define debug_handler_c(name, msg, level, code)\
{ if (LIDIA_DEBUG <= level)\
lidia_debug_handler(name, msg);\
code; }
```

Finally, the `memory_handler` is intented to be used as a check after allocating vectors of objects. Therefore, it has an additional pointer argument (the vector the user tried to allocate). If the allocation was not successful (memory exhausted), a message is printed and the program aborts execution. Otherwise no message is printed. Again, to allow memory handling in LiDIA, it is necessary to define the compilation flag `-DLIDIA_MEMORY`

See also

UNIX manual page abort(3)

Warnings

The compilation flags `LIDIA_WARNINGS`, `LIDIA_DEBUG` and `LIDIA_MEMORY` must be defined before compiling LiDIA.

Examples

```
#include <LiDIA/error.h>
#include <LiDIA/memory.h>

int main()
{
    char *s;
```

```
        if (1 != (2 - 1))
            lidia_error_handler("main()", "if::math. error");

        s = new char[100000];

        memory_handler(s, "main()", "allocating 100000 chars");

        return 0;
    }
```

Author

Thomas Papanikolaou

basic_error

Name

`basic_error` ..... root of LiDIA's exception hierarchy

Abstract

As long as you did not disable exception handling during configuration, LiDIA reports errors by throwing exceptions. All exceptions thrown are derived from class `basic_error`. The exception hierarchy is still under development, so the actual type of the thrown exceptions is subject to change. For the time beeing, the only guarantee is that `catch(const basic_error& ex) { /*...*/ }` will catch all exceptions thrown by LiDIA.

Description

Class `basic_error` is publicly derived from `std::exception`. It is designed as an abstract base class of LiDIA's exception hierarchy, so you cannot create objects of type `basic_error`.

In order to allow a smooth transition from LiDIA's legacy error handlers, a concrete subclass `generic_error` is provided. But you must not rely on the thrown exceptions to be of type `generic_error` since we may implement specific exception classes without further notice.

Let `e` be an object of type `basic_error`.

Constructors/Destructor

`ct basic_error (const std::string& msg)`

Constructs an exception object. The argument is passed on to the constructor of the base class `std::runtime_error`. This constructor is protected.

`dt virtual ~basic_error ()`

Destroys an exception. This destructor offers the "nothrow-guarantee". Every subclass must explicitly declare its destructor to be nonthrowing with `throw()`.

Access Methods

`virtual const std::string& e.offendingClass () const`

Returns a textual description where the error occured. This method is pure virtual and must be overridden in all concrete subclasses.

`virtual const char* e.what () const`

Returns a textual description of the exception. This method overrides `std::exception::what()`

High-Level Methods and Functions

`void e.traditional_error_handler () const`

Calls LiDIA's traditional error handling module via `lidia_error_handler(classname, msg)`. This method does therefore not return. `classname` and `msg` are determined by the protected virtual member function `traditional_error_handler_impl(string&, string&)`.

See also

`lidia_error_handler`

Warnings

You must not configure LiDIA with `--enable-exceptions=no` if you want LiDIA to throw exceptions.

Examples

```
#include <iostream>
#include <LiDIA/error.h>
#include <LiDIA/bigint.h>
#include <LiDIA/base_vector.h>

int main()
{
    LiDIA::base_vector<LiDIA::bigint> v(10, FIXED);

    try {
      // manipulate v ...
      std::cout << v[10] << std::endl;
    }
    catch(const LiDIA::basic_error& ex) {
      if(/* can we deal with the error? */) {
        std::clog << "There was a problem in " << ex.offendingClass() << ":\n"
                  << ex.what() << "\n\n"
                  << "We can nevertheless continue.\n".
        // ...
      }
      else { // Give up
        ex.traditional_error_handler();
      }
    }
    return 0;
}
```

Author

Christoph Ludwig

generic_error

Name

`generic_error` ..... concrete exception class for general errors

Abstract

LiDIA throws a `generic_error` if, for some particular error situation, no specialized exception class is provided.

Description

Class `generic_error` is publicly derived from `basic_error`. Its interface gives access to two strings that correspond to to the arguments of `lidia_error_handler`. `traditional_error_handler` yields the same behaviour as a call to `lidia_error_handler` in LiDIA 2.0 and earlier.

Let `e` be an object of type `generic_error`.

Constructors/Destructor

`explicit ct generic_error (const std::string& what_msg)`

This constructor is equivalent to `generic_error("<generic_error>", what_msg)`.

`ct generic_error (const std::string& classname, const std::string& what_msg)`

Constructs a new generic error object. The semantic of `classname` and `what_msg` is the same as in `lidia_error_handler()`. When the constructor returns the invariants `classname == this->offendingClass()` and `what_msg == this->what` will hold.

`dt virtual ~generic_error ()`

Frees all resources hold by this object.

Access Methods

`virtual const std::string& e.offendingClass () const`

Returns a textual description where the error occured. Corresponds to the first parameter of `lidia_error_handler`.

See also

`lidia_error_handler, basic_error`

Warnings

LiDIA's exception hierarchy is still experimental. We may change it without prior notice.

Author

Christoph Ludwig

cast_error

Name

`cast_error` ..... exception indicating that a cast failed

Abstract

If LiDIA has to perform a cast but cannot guarantee that the cast will succeed (e. g. in `precondition_error::paramValue<T>()`) then it will throw a `cast_error` in case of failure.

Description

Class `cast_error` is publicly derived from `basic_error` as well as `std::bad_cast`.

Let `e` be an object of type `cast_error`.

Constructors/Destructor

`ct cast_error (const std::string& classname, const std::string& what_msg)`

Constructs a new cast error object. The semantic of `classname` and `what_msg` is the same as in `lidia_error_handler()`. When the constructor returns the invariants `classname == this->offendingClass()` and `what_msg == this->what()` will hold.

`dt virtual ~cast_error ()`

Frees all resources hold by this object.

Access Methods

`virtual const std::string& e.offendingClass () const`

Returns a textual description where the error occured. Corresponds to the first parameter of `lidia_error_handler`.

`virtual const char* e.what () const`

Returns a textual description of the error. Corresponds to the second parameter of `lidia_error_handler`.

This method overrides `basic_error::what()` as well as `std::bad_cast::what()` and thereby resolves any ambiguity.

See also

`lidia_error_handler, basic_error, std::bad_cast`

Warnings

LiDIA's exception hierarchy is still experimental. We may change it without prior notice.

Author

Christoph Ludwig

precondition_error

Name

`precondition_error` ..... exception indicating that a precondition was not met

Abstract

Some LiDIA routines enforce their preconditions. If they detect a failed precondition check they throw a `precondition_error`, typically via a call of `precondition_error_handler()`.

Description

Class `precondition_error` is publicly derived from `basic_error` as well as `std::logic_error`. It stores a textual representation of the prototype of the function that detected the failed precondition as well as the class or source file the function belongs to and an error message.

Furthermore, one can add arbitrary data objects together with strings describing their name and their requirements at runtime. There are no relevant restrictions with respect to number or type of the data objects. The added objects can be accessed through a rudimentary vector-like interface.

Let `e` be an object of type `precondition_error`.

Typedefs

`precondition_error::size_type`

An integral type that can hold the number of added data objects.

Constructors/Destructor

```
ct precondition_error (const std::string& proto, const std::string& where,
                       const std::string& what_msg)
```

Constructs a new precondition error object. The semantic of `where` and `what_msg` is the same as in `lidia_error_handler()`. When the constructor returns the postconditions `proto == e.prototype()`, `where == e.offendingClass()`, `what_msg == e.what()`, and `0 == e.noParams()` hold.

```
ct precondition_error (const precondition_error& pce)
```

Constructs a copy of `pce`. The copy strategy is *deep-copy*, i. e. all added data objects will be copied, too.

```
dt virtual ~cast_error ()
```

Frees all resources hold by this object. All added data objects will be destroyed.

Assignments

`precondition_error& e.operator= (const precondition_error& pce)`

Makes `e` a copy of `pce`. The copy strategy is *deep-copy*, i. e. all added data objects will be copied, too. All data objects previously held by `e` will be freed.

This operator offers the strong exception safety guarantee, i. e. the state of `e` will only be changed if the copy succeeds.

Access Methods

`virtual const std::string& e.prototype () const`

Returns the prototype of the function that detected the failed precondition.

`virtual const std::string& e.offendingClass () const`

Returns a textual description where the error occured. Corresponds to the first parameter of `lidia_error_handler`.

`virtual const char* e.what () const`

Returns a textual description of the error. Corresponds to the second parameter of `lidia_error_handler`.

This method overrides `basic_error::what()` as well as `std::logic_error::what()` and thereby resolves any ambiguity.

`size_type e.noParams () const`

The number of data objects added to `e`.

`std::string& e.paramNameStr (size_type` n`) const`

The name of the parameter added at position n.

`paramNameStr` throws an `index_out_of_bounds_error` if $0 \leq n <$ `e.noParams()` does not hold.

`std::string& e.paramValueStr (size_type` n`) const`

A textual representation of the value of the parameter added at position n. If the parameter is of type `T`, then the representation is generated by a call to `std::ostream& operator<<(std::ostream&, const T& t)`.

`paramValueStr` throws an `index_out_of_bounds_error` if $0 \leq n <$ `e.noParams()` does not hold.

`std::string& e.paramConditionStr (size_type` n`) const`

A description of the requirement on the parameter added at position n.

`paramConditionStr` throws an `index_out_of_bounds_error` if $0 \leq n <$ `e.noParams()` does not hold.

`const T& e.paramValueStr< T > (size_type` n`) const`

The value of the parameter added at position n..

`paramValueStr` throws an `index_out_of_bounds_error` if $0 \leq n <$ `e.noParams()` does not hold and a `cast_error` if the parameter cannot be casted to the type `const T`.

`size_type e.indexOf (const std::string& paramName) const`

Returns the smalles nonnegative index n such that `paramName == e.paramNameStr(`n`)` holds. If no such index exists, `indexOf` returns `e.noParams()`.

Basic Methods and Functions

```
virtual void e.traditional_error_handler_implementation (std::string& classname,
                                                         std::string& msg) const
```

Prints the prototype and all added parameters with their values, names, and requirements to `std::cerr`. `classname` and `msg` are set to `e.offendigClass()` and `e.what()`, respectively.

The effect is that a call to `e.traditional_error_handler()` will yield the same side effects as a call to the (undocumented) function templates `lidia_error_handler< T1, ..., Tn >` in LiDIA 2.0.

```
void precondition_error_handler< T1, ..., Tn > (const T1& para_1, const char* name_1,
                                                const char* cond_1, ...,
                                                const Tn& para_n, const char* name_n,
                                                const char* cond_n, const char* proto,
                                                const char* where, const char* what_msg)
```

A function template that creates a `precondition_error`, adds the parameters and their description to it and passes it on to `error_handler`.

`precondition_error_handler` is defined for up to twelve template arguments. If you need a spezialization that LiDIA does not provide yet you can include the function template definitions from `LiDIA/precondition_error.cc`.

See also

`lidia_error_handler, basic_error, std::bad_cast`

Warnings

LiDIA's exception hierarchy is still experimental. We may change it without prior notice.

Author

Christoph Ludwig

index_out_of_bounds_error

Name

`index_out_of_bounds_error` ..... a sequence was subscripted with an invalid index.

Abstract

An `index_out_of_bounds_error` is thrown if an element access in a sequence or container fails due to an invalid index.

Description

Class `index_out_of_bounds_error` is publicly derived from `basic_error` as well as `std::out_of_range`.

Let `e` be an object of type `index_out_of_bounds_error`.

Constructors/Destructor

`ct index_out_of_bounds_error (const std::string& classname, unsigned long` n`)`

Constructs a new `index_out_of_bounds_error` object. `classname` describes the class and / or function where the error was detected. When the constructor returns the invariant `classname == e.offendingClass()` and n `== e.offendingIndex()` will hold.

`dt virtual ~index_out_of_bounds_error ()`

Frees all resources hold by this object.

Access Methods

`unsigned long e.offendingIndex () const`

Returns the index that prompted the exception to be thrown.

`virtual const std::string& e.offendingClass () const`

Returns a textual description where the error occured. Corresponds to the first parameter of `lidia_error_handler`.

`virtual const char* e.what () const`

Returns `"index out of bounds"`. Corresponds to the second parameter of `lidia_error_handler`.

This method overrides `basic_error::what()` as well as `std::out_of_range::what()` and thereby resolves any ambiguity.

See also

`lidia_error_handler, basic_error, std::out_of_range`

Warnings

LiDIA's exception hierarchy is still experimental. We may change it without prior notice.

Author

Christoph Ludwig

gmm

Name

gmm ..... general memory manager

Abstract

gmm is the memory management interface of LiDIA. It provides standardized calls to the underlying memory manager (which can be either the system's manager or a user implemented one). gmm assumes an uncooperative environment. This means that no special class support is needed in order to add memory management to the class.

Description

gmm is implemented as a class providing a small set of functions for allocation, memory resizing and deallocation. Using this functions overloads the `new` and `delete` operators. A user class is registered to gmm by simple inheritance, i.e.

```
#include <LiDIA/gmm.h>

class user_class: public gmm { ...  };
```

A variable of the class `user_class` can then be allocated by simply typing

```
user_class *a = new user_class;
```

or equivalently

```
user_class *a = new (NoGC) user_class;
```

The additional argument after the `new` keyword denotes the method to be used for the allocation of a. In order to support garbage collecting and non-garbage collecting memory managers gmm provides three modes

1. `NoGC`
   allocates an uncollectable item.
2. `AtGC`
   allocates a collectable atomic item, i.e. an object containing no pointers in it.
3. `GC`
   allocates a collectable item.

Now, if gmm is implemented by malloc(), then these three modes are equal (allocate always an uncollectable object). However, if a garbage-collector is used like [6], then these three modes are different.

A note on the philosophy of this approach. Garbage collection is expensive in execution resources. Although in some examples it does as well as by-hand collection (i.e. the user is responsible to free allocated memory, for example be using delete), in general it is 10% to 20% slower (most of the time even worse than that). Therefore, LiDIA functions and classes are written using by-hand collection (so that there is no memory leak if a non-collecting manager is used). It is clear that avoiding the use of automatic collection implies more work. However, we do not want to lose this factor of more than 10%.

Basic Methods and Functions

`static void * allocate (size_t` *NMEMB*`, size_t` *SIZE*`)`

allocates memory for an array of *NMEMB* elements of *SIZE* bytes each and returns a pointer to the allocated memory. The memory is set to zero. For `allocate(`*NMEMB*`, `*SIZE*`)`, the value returned is a pointer to the allocated memory, which is suitably aligned for any kind of variable or `NULL` if the request fails.

`static void * allocate (size_t` *SIZE*`)`

allocates *SIZE* bytes and returns a pointer to the allocated memory. The memory ist not cleared. For `allocate(`*SIZE*`)`, the value returned is a pointer to the allocated memory, which is suitably aligned for any kind of variable or `NULL` if the request fails.

`static void * allocate_uncollectable (size_t` *SIZE*`)`

allocates *SIZE* bytes and returns a pointer to the allocated memory. The memory ist not cleared and will not be collected by the implemented memory manager. For `allocate_uncollectable(`*SIZE*`)`, the value returned is a pointer to the allocated memory, which is suitably aligned for any kind of variable or `NULL` if the request fails.

`static void * allocate_atomic (size_t` *SIZE*`)`

allocates *SIZE* bytes and returns a pointer to the allocated memory. `allocate_atomic(`*SIZE*`)` assumes that there are no relevant pointers in the object. The memory ist not cleared. For `allocate_atomic(`*SIZE*`)`, the value returned is a pointer to the allocated memory, which is suitably aligned for any kind of variable or `NULL` if the request fails.

`static void * resize (void * PTR, size_t` *NSIZE*`, size_t` *OSIZE*`)`

changes the size of the memory block pointed to by `PTR` to *NSIZE* bytes. The contents will be unchanged to the minimum of the $OSIZE$ and $NSIZE$ sizes. Newly allocated memory will be uninitialized. If `PTR` is `NULL`, the call is equivalent to `allocate(`*SIZE*`)`. If *NSIZE* is equal to zero, the call is equivalent to `release(PTR)`. Unless PTR is `NULL`, it must have been returned by an earlier call of `allocate(`*SIZE*`)` or `resize(PTR, `*NSIZE*`, `*OSIZE*`)`. `resize(PTR, `*NSIZE*`, `*OSIZE*`)` returns a pointer to the newly allocated memory, which is suitably aligned for any kind of variable and may be different from `PTR` or `NULL` if the request fails or if *NSIZE* was equal to 0.

`static void release (void *PTR)`

frees the memory space pointed to by `PTR`, which must have been returned by a previous call of `allocate(`*SIZE*`)`. If `PTR` is `NULL`, no operation is performed.

`static void collect ()`

initiates a garbage collection if the implemented memory manager supports garbage collection.

`void * operator new (size_t` *SIZE*`)`

operator new overloading. Calls `allocate(`*SIZE*`)`.

`void * operator new (size_t` *SIZE*`, memory_manager_mode` *MODE*`)`
operator new overloading with a predefined *MODE*. Mode can be one of `NoGC` (the default), `AtGC` or `GC`.

`void * operator delete (void *PTR)`
operator delete overloading. Class `release(PTR)`.

If the compiler allows the overloading of the `new` and `delete` operators for arrays, the following functions are also available:

`void * operator new[] (size_t` *SIZE*`)`

`void * operator new[] (size_t` *SIZE*`, memory_manager_mode` *MODE*`)`

See also

UNIX manual page malloc(3).

Author

Thomas Papanikolaou

random_generator

Name

`random_generator` ..... a class for producing random numbers

Abstract

`random_generator` is a class that can be used to create random numbers for built-in types `int` and `long`. Using this class exclusively and no operating system functions like `random()`, `srandom()` etc. allows to collect the operating system dependend parts of the process of generating random numbers for built-in types in the implementation of this class. Especially, the initialization of the random number generator is done by the class `random_generator`.

Description

`random_generator` is an interface that calls functions provided by the operating system. The implementation of the internal functions depends on the platform, LiDIA is running on.

Constructors/Destructor

`ct random_generator ()`
Initializes the random number generator.

`dt ~random_generator ()`
Deletes the random generator object.

Assignments

The operator `>>` is overloaded and is used to produce random numbers.

`random_generator & operator>> (random_generator & rg, int & i)`
produces a randomly chosen number and assigns it to i.

`random_generator & operator>> (random_generator & rg, long & l)`
produces a randomly chosen number and assigns it to l.

Examples

```
#include <LiDIA/random_generator.h>

int main()
{
    random_generator rg;
    int i, j;

    rg >> i >> j;

    cout << "i == " << i << ", j == " << j << endl;
    cout.flush();

    return 0;
}
```

For further examples please refer to `LiDIA/src/portabiliy/random_generator_appl.cc`.

Author

Markus Maurer

lidia_signal

Name

`lidia_signal` ..... stacks for signals

Abstract

`lidia_signal` is a class that contains a collection of stacks, one for each signal. It is used by those functions of LiDIA, that define their own handlers for signals to guarantee that user defined handlers will be reinstalled, when the program leaves the scope of the LiDIA functions.

Description

The class `lidia_signal` contains a stack for each signal, that is defined by the system. It assumes the following scenario: A user (i.e., a function outside the LiDIA library) installs a handler for a signal and then calls some LiDIA function. That function and further LiDIA routines also install handlers for this signal, but exclusively by using the `lidia_signal` class. When the first LiDIA function installs a handler for the signal, the user defined signal is stored on bottom of the stack and the LiDIA handler above. Every further LiDIA handler is stored on top of the stack. Before the program returns to the user (leaves the scope of the LiDIA functions), all LiDIA functions must have uninstalled their handlers. If the stack only contains one (the user defined) handler, the `lidia_signal` class reinstalls that handler and cleans the stack.

To obtain automatic deinstallation of a handler, the class `lidia_signal` only provides a constructor and a destructor.

Please use the macro `LIDIA_SIGNAL_FUNCTION` to define a handler for a signal. Then the return and parameter type of the handler is set appropriately; see definition of function `myhandler` in the example at the end of the description. The macro is defined in the file `LiDIA/lidia_signal.h`.

The implementation of the class `lidia_signal` can be found in the directory `LiDIA/src/portability`.

`LiDIA/lidia_signal.h` defines all signals for which a handler can be installed. It follows the list of those signals:

1. `LIDIA_SIGHUP` hangup
2. `LIDIA_SIGINT` interrupt (rubout)
3. `LIDIA_SIGQUIT` quit (ASCII FS)
4. `LIDIA_SIGILL` illegal instruction (not reset when
5. `LIDIA_SIGTRAP` trace trap (not reset when
6. `LIDIA_SIGIOT` IOT instruction
7. `LIDIA_SIGABRT` used by abort, replace SIGIOT

8. LIDIA_SIGEMT EMT instruction
9. LIDIA_SIGFPE floating point exception
10. LIDIA_SIGKILL kill (cannot be caught or ignored)
11. LIDIA_SIGBUS bus error
12. LIDIA_SIGSEGV segmentation violation
13. LIDIA_SIGSYS bad argument to system call
14. LIDIA_SIGPIPE write on a pipe with
15. LIDIA_SIGALRM alarm clock
16. LIDIA_SIGTERM software termination signal from kill
17. LIDIA_SIGUSR1 user defined signal 1
18. LIDIA_SIGUSR2 user defined signal 2
19. LIDIA_SIGCLD child status change
20. LIDIA_SIGCHLD child status change alias (POSIX)
21. LIDIA_SIGPWR power-fail restart
22. LIDIA_SIGWINCH window size change
23. LIDIA_SIGURG urgent socket condition
24. LIDIA_SIGPOLL pollable event occured
25. LIDIA_SIGIO socket I/O possible (SIGPOLL alias)
26. LIDIA_SIGSTOP stop (cannot be caught or
27. LIDIA_SIGTSTP user stop requested from tty
28. LIDIA_SIGCONT stopped process has been continued
29. LIDIA_SIGTTIN background tty read attempted
30. LIDIA_SIGTTOU background tty write attempted
31. LIDIA_SIGVTALRM virtual timer expired
32. LIDIA_SIGPROF profiling timer expired
33. LIDIA_SIGXCPU exceeded cpu limit
34. LIDIA_SIGXFSZ exceeded file size limit
35. LIDIA_SIGWAITING process's lwps are blocked

Constructors/Destructor

`ct lidia_signal (int` *sig*`, lidia_signal_handler_t h)`

stores signal *sig* and installs handler `h` for signal *sig*. If *sig* can not be found in the signal list, the `lidia_error_handler` is called.

`dt ~lidia_signal ()`

uninstalls the handler for the stored signal.

Basic Methods and Functions

`static void print_stack (int` *sig*`)`

Prints the content of the stack of signal *sig* to `stdout`. If *sig* can not be found in the signal list, the `lidia_error_handler` is called.

Examples

```
#include <LiDIA/lidia_signal.h>

LIDIA_SIGNAL_FUNCTION(myhandler)
{
    cout << " Hello, this is the myhandler function.\n";
    cout << " I have caught your LIDIA_SIGINT.\n";
    cout << endl;
    cout.flush();
}


void foo()
{
    // install myhandler for LIDIA_SIGINT
    lidia_signal s(LIDIA_SIGINT,myhandler);

    cout << endl;
    cout << " If you type Ctrl-C within the next 15 seconds,\n";
    cout << " the function myhandler is called.\n";
    cout << " sleep(15) ...\n\n";
    cout.flush();

    sleep(15);

    // The automatic call of the destructor reinstalls
    // the previous handler for LIDIA_SIGINT.
}


int main()
{
    cout << endl;
    cout << "Going into function foo ..." << endl;

    foo();

    cout << "Back from foo.\n";
    cout << "The default handler for LIDIA_SIGINT is installed again.\n";
    cout << "sleep(15) ...\n\n";
    cout.flush();

    sleep(15);

    return 0;
}
```

Please refer to `LiDIA/src/portability/lidia_signal_appl.cc`.

Author

Sascha Demetrio, Markus Maurer

The LiDIA FF package

Chapter 10

Arithmetics over $\mathrm{GF}(2^n)$, $\mathrm{GF}(p^n)$

gf2n

Name

`gf2n` ..... $\mathrm{GF}(2^n)$ arithmetic

Abstract

`gf2n` is a class for doing arithmetic in $\mathrm{GF}(2^n)$, the finite field with 2^n elements. It supports for example arithmetic operations, comparisons, I/O of $\mathrm{GF}(2^n)$-elements and basic operations like computing a generator of the multiplicative group. We use a polynomial representation for $\mathrm{GF}(2^n)$ with a precomputed sparse modulus.

Description

We use the polynomial representation for elements in the finite field $\mathrm{GF}(2^n)$. A `gf2n` element is represented by an array of `unsigned long`s and a global modulus. This modulus must be set by the procedure `gf2n_init`, before any variable of type `gf2n` can be declared. The modulus can be chosen from a given database or you can select it yourself. For a detailed description of the different inputs of this initialization procedure, see section initialization.

Each equivalence class modulo the global modulus is represented by a polynomial over $\mathrm{GF}(2)$, whose degree is smaller than the degree of the modulus. This polynomial is internally stored in the bit field given by the array of `unsigned long`s.

The format of the input/output of `gf2n` elements is handled with a special class `gf2nIO`. Available choices are described in section Input/Output.

Constructors/Destructor

`ct gf2n ()`
initialize variable with 0.

`ct gf2n (const gf2n & `x`)`
initialize variable with x.

`ct gf2n (unsigned long `ui`)`
initialize variable with the binary representation of ui, i.e. if $ui = \sum_{i=0}^{k} b_i\, 2^i$ with $b_i \in \{0,1\}$, then initialize the variable with the equivalence class represented by the polynomial $\sum_{i=0}^{k} b_i x^i$. If ui is greater or equal than the number of elements in $\mathrm{GF}(2^n)$, the `lidia_error_handler` is invoked.

`ct gf2n (const bigint & b)`

initializes the declared variable with the binary representation of b, i.e. if $b = \sum_{i=0}^{k} b_i 2^i$ with $b_i \in \{0, 1\}$, then initialize the variable with the equivalence class represented by the polynomial $\sum_{i=0}^{k} b_i x^i$. If b is greater or equal than the number of elements in GF(2^n), the `lidia_error_handler` is invoked.

`dt ~gf2n ()`

Initialization

One of the two following initialization routines must be called before any object of type `gf2n` can be declared. Note that because of this fact it is not possible to use static or global variables of type `gf2n`. It is however possible to call a initializing routine for `gf2n` several times in a program. After each such call, existing variables of type `gf2n` have to be reinitialized before further usage.

`void gf2n_init (unsigned int d, gf2nIO::base base = gf2nIO::Dec)`

the function selects a sparse irreducible polynomial of degree d over GF(2) in a precomputed database and initializes the global modulus with this polynomial. Per default the database is read from the file `LIDIA_INSTALL_DIR/lib/LiDIA/GF2n.database`, where `LIDIA_INSTALL_DIR` denotes the directory where LiDIA was installed by the installation procedure. If you want to use another location for the database, you can set the environment variable `LIDIA_GF2N` to point to this location. Furthermore the initialization function sets the output base of `gf2nIO` to `base`, for a detailed description of `gf2nIO` see the section on Input/Output.

`void gf2n_init (char * m, gf2nIO::base base = gf2nIO::Dec)`

the global modulus of `gf2n` is initialized with the polynomial, which is described in the string `m`. This string must contain the exponents of 1-coefficients of the polynomial, separated by a blank (example: the string "123 45 0" represents the polynomial $x^{123} + x^{45} + x^0$). Note that the irreducibility of the modulus is not checked. The output base of `gf2nIO` is set to `base`, for a detailed description of `gf2nIO` see the section on Input/Output.

`void` x`.re_initialize ()`

if the variable x was declared before the latest call to one of the previous initialization routines, then x has to be reinitialized before further usage. This function discards the old value of x, changes the internal size appropriately and initializes x with 0.

Assignments

The assignment operator `=` is overloaded. Furthermore there exist the following assignment functions:

`void` x`.assign_zero ()`

$x \leftarrow 0$.

`void` x`.assign_one ()`

$x \leftarrow 1$.

`void` x`.assign (const gf2n &` a`)`

$x \leftarrow a$.

Arithmetical Operations

The following operators are overloaded and can be used in exactly the same way as in C++ (note that addition and subtraction are the same in fields of characteristic 2, but nevertheless we support both operators).

```
(binary) +, -, *, /
(binary with assignment) +=, -=, *=, /=
```

To avoid copying, all operators also exist as functions:

`void add (gf2n & `x`, const gf2n & `y`, const gf2n & `z`)`
$x \leftarrow y + z$.

`void subtract (gf2n & `x`, const gf2n & `y`, const gf2n & `z`)`
$x \leftarrow y - z$.

`void multiply (gf2n & `x`, const gf2n & `y`, const gf2n & `z`)`
$x \leftarrow y \cdot z$.

`void divide (gf2n & `x`, const gf2n & `y`, const gf2n & `z`)`
$x \leftarrow y/z$. If $z = 0$, the `lidia_error_handler` is invoked.

`void square (gf2n & `x`, const gf2n & `y`)`
$x \leftarrow y^2$.

`void sqrt (gf2n & `x`, const gf2n & `y`)`
compute x with $x^2 = y$.

`gf2n sqrt (const gf2n & `y`)`
return x such that $x^2 = y$.

`void `x`.invert ()`
set $x \leftarrow x^{-1}$. If $x = 0$, the `lidia_error_handler` is invoked.

`void invert (gf2n & `x`, const gf2n & `y`)`
set $x \leftarrow y^{-1}$. If $y = 0$, the `lidia_error_handler` is invoked.

`gf2n inverse (const gf2n & `y`)`
return y^{-1}. If $y = 0$,the `lidia_error_handler` is invoked.

`void power (gf2n & `x`, const gf2n & `y`, const & bigint `i`)`
$x \leftarrow y^i$. If $y = 0$ and $i < 0$, the `lidia_error_handler` is invoked.

`void power (gf2n & `x`, const gf2n & `y`, long `l`)`
$x \leftarrow y^l$. If $y = 0$ and $l < 0$, the `lidia_error_handler` is invoked.

Comparisons

`gf2n` supports the binary operators `==` and `!=`. Let a be an instance of type `gf2n`.

`bool` a`.is_zero () const`
returns `true` if $a = 0$, `false` otherwise.

`bool` a`.is_one () const`
returns `true` if $a = 1$, `false` otherwise.

Basic Methods and Functions

Let x be of type `gf2n`.

`static unsigned int get_absolute_degree ()`
return the extension degree n of the current field GF(2^n) over the prime field GF(2).

`unsigned int` x`.relative_degree ()`
return the minimal extension degree k of the field GF(2^k) over the prime field GF(2) where x is an element of GF(2^k).

`void swap (gf2n &` a`, gf2n &` b`)`
exchange the values of a and b.

`udigit hash (const gf2n &` a`)`
return a hash value for a.

`void` x`.randomize (unsigned int` d `=` n`)`
set x to a random element in GF(2^d). If GF(2^d) is no subfield of GF(2^n) (i.e. d is no divisor of n), then the `lidia_error_handler` is invoked.

`gf2n randomize (const gf2n &` a`, unsigned int` d `=` n`)`
return a random element in GF(2^d). If GF(2^d) is no subfield of GF(2^n) (i.e. d is no divisor of n), then the `lidia_error_handler` is invoked.

`int` x`.trace ()`
return the trace of x as `int`.

High-Level Methods and Functions

`bigint compute_order (const gf2n &` a`)`
compute the order of a. Note that computing the order of an element might imply factoring the order of the multiplicative group of GF(2^n) which can take a while for extension degrees bigger than 150.

`gf2n get_generator (unsigned int` d `=` n`)`
return a generator of the multiplicative group of GF(2^d). If d does not divide n, then the `lidia_error_handler` is invoked. Note that computing a generator might imply factoring the order of the multiplicative group of GF(2^n) which can take a while for extension degrees bigger than 150.

`bool solve_quadratic (gf2n & r, const gf2n & a1, const gf2n & a0)`

The function tries to determine a root of the quadratic polynomial $Y^2 + a_1Y + a_0$. If this polynomial splits, the function sets r to one of the roots and returns `true`, otherwise the function returns `false` and does not change r.

In addition to this general function, there exists a moer optimized function which uses precomputed tables. This function is much faster, but the table precomputation takes some time, such that it's usage is only recommended for many simultaneous root computations. Note that the precomputation routine must be called by the user. Releasing the tables after usage is also the responsibility of the user. Moreover after each call reinitialization of a `gf2n`, the tables are also automatically deleted and must be recomputed.

`static void initialize_table_for_solve_quadratic ()`

initialize internal tables for fast solving of qudratic equations. This function must be called before the improved algorithm can be used. Tables must be released manually after usage by calling the following function.

`static void delete_table_for_solve_quadratic ()`

delete internal tables used for fast solving of qudratic equations.

`bool solve_quadratic_with_table (gf2n & c, const gf2n & a1, const gf2n & a0, bool certainly_factors = false)`

The function tries to determine a root of the quadratic polynomial $Y^2 + a_1Y + a_0$ with the help of precomputed tables. Note that these tables must be initialized beforehand with the initialization function described above. If the polynomial splits, the function sets r to one of the roots and returns `true`, otherwise the function returns `false` and does not change r. If `certainly_factors` is `true`, then the reducibility of the polynomial is not checked but assumed. The behavior of the function is undefined if not used properly.

Input/Output

The `istream` operator `>>` and the `ostream` operator `<<` are implemented. The format of the I/O of `gf2n` elements can be chosen with the help of the class `gf2nIO`. Note that the I/O-format stored in `gf2nIO` is valid for all variables of type `gf2n`.

Input of `gf2n` elements is per default expected as decimal number. Moreover one of the two prefixes `dec:`, `hex:` can be used to denote decimal, hexadecimal interpretation of the input, respectively. Example: you can input the "polynomial" $x^4 + x + 1$ as either `19`, `dec:19` or `hex:a4`.

At the moment, the class `gf2nIO` allows you to choose a base for Output and an output prefix. Available bases are given as `enum base { Dec, Hex }`. Output of an `gf2n` variable is then done as decimal, hexadecimal number with the chosen prefix, respectively. This Output format can be chosen during initialization with the functions `gf2n_init` or it can be changed at any time with the help of the following functions.

`gf2nIO::void setbase (gf2nIO::base b)`

set `gf2nIO::base` to base b. If b is no valid base, the `lidia_error_handler` is invoked.

`gf2nIO::base showbase ()`

return `gf2nIO::base`.

`gf2nIO::void setprefix (gf2nIO::base b)`

set the output-prefix to the standard prefix for base b.

`gf2nIO::void setprefix (char * p)`
set the output prefix to `p`.

`gf2nIO::void noprefix ()`
set the output prefix to the empty string.

`char * showprefix ()`
returns the actual output prefix.

Examples

The following program computes a generator of a given field GF(2^n).

```
#include <LiDIA/gf2n.h>

int main()
{
    unsigned int degree;

    cout << "Degree: ";  cin >> degree;
    gf2n_init(degree);
    gf2n a;
    a = get_generator();
    cout << "Generator of GF(2^"<< degree <<") = " << a;

    return 0;
}
```

Notes

The classes `gf2n` and `gf2nIO` are not yet complete. In future we will add more different input/output formats and make the I/O of `gf2n` elements easier.

Author

Franz-Dieter Berger, Patric Kirsch, Volker Müller

galois_field

Name

`galois_field` ..... galois field

Abstract

A variable of type `galois_field` represents a finite field GF(q).

Description

A variable of type `galois_field` contains a reference to an object of type `galois_field_rep`. In objects of the latter type we store the characteristic, which is of type `bigint`, the absolute degree, which is of type `lidia_size_t`, and the defining polynomial for the field. We furthermore store the factorization of the order of the multiplicative group GF(q)*. However, this factorization is computed only when explicitely needed.

Any instance of type `galois_field` can be deleted when its functionality is no longer required. Instances of type `gf_element` do not depend on the existence of objects of type `galois_field` which they might have been initialized with.

Constructors/Destructor

`ct galois_field ()`

initializes with a reference to a mainly useless dummy field.

`ct galois_field (const bigint & `*characteristic*`, lidia_size_t `*degree*` = 1)`

constructs a field by giving the *characteristic* and *degree* of it. The default degree is 1. If *characteristic* is not a prime number, the behaviour of the program is undefined.

`ct galois_field (const bigint & `*characteristic*`, lidia_size_t `*degree*`,`
`const rational_factorization & `*fact*`)`

constructs a field by giving the *characteristic*, the *degree* and the factorization *fact* of the number of elements in the multiplicative group of the constructed field. If *fact* is not a factorization of $characteristic^{degree} - 1$ (it does not have to be a prime factorization), the behaviour of the program is undefined.

`ct galois_field (const Fp_polynomial & `g`)`

constructs the finite field $\mathbb{F}_p[X]/(g(X)\mathbb{F}_p[X]) \cong \mathbb{F}_p^n$ by giving a monic irreducible polynomial g of degree $n = g$`.degree()` over $\mathbb{Z}/p\mathbb{Z}$, where $p = g$`.modulus()`. The behaviour of the program is undefined if g is not irreducible.

`ct galois_field (const Fp_polynomial & g, const rational_factorization & fact)`
constructs the finite field $\mathbb{F}_p[X]/(g(X)\mathbb{F}_p[X]) = \mathbb{F}_p^n$ by giving a monic irreducible polynomial g of degree $n = g$`.degree()` over $\mathbb{Z}/p\mathbb{Z}$, where $p = g$`.modulus()`. The behaviour of the program is undefined if f is not irreducible. If *fact* is not a factorization of $p^n - 1$ (it does not have to be a prime factorization), the behaviour of the program is undefined.

`ct galois_field (const galois_field & f2)`
constructs a copy of f_2.

`dt ~galois_field ()`

Access Methods

Let f be an instance of type `galois_field`.

`const bigint & f.characteristic () const`
returns the characteristic of the field f.

`lidia_size_t f.degree () const`
returns the degree of the field f.

`const bigint & f.number_of_elements () const`
returns the number of elements of the field f.

`const rational_factorization & f.factorization_of_mult_order () const`
returns the prime factorization of the number of elements in the multiplicative group of f.

`Fp_polynomial f.irred_polynomial () const`
returns the defining polynomial of f.

Assignments

Let f be an instance of type `galois_field`. The operator `=` is overloaded. For efficiency reasons, the following function is implemented:

`void f.assign (const galois_field & f2)`
$f = f_2$.

Comparisons

The binary operators `==`, `!=` are overloaded and have the usual meaning. Furthermore, the operators `<`, `>` are overloaded: Let f and `f2` be instances of type `galois_field`. Then f `<` f_2 returns `true` if f is a proper subfield of f_2, and `false` otherwise. Operator `>` is defined analogously.

Input/Output

The `istream` operator `>>` and the `ostream` operator `<<` are overloaded.

`istream & operator >> (istream & in, galois_field & f)`

reads a field from `istream in`. The input must be either of the form "[p, n]", where p is the characteristic and n the degree of the field f, or it must be of the form "$f(X)$ mod p", where $f(X)$ is a polynomial of degree n which is irreducible modulo p.

`ostream & operator << (ostream & out, const galois_field & f)`

writes the field f to `ostream out` in form of "$f(X)$ mod p", where p is the characteristic and f defining polynomial of the field f.

See also

`gf_element`

Examples

see example for class `gf_element`

Author

Detlef Anton, Franz-Dieter Berger, Stefan Neis, Thomas Pfahler

gf_element

Name

`gf_element` ..... an element over a finite field

Abstract

`gf_element` is a class that provides arithmetics over a finite field. If the finite field is defined by a polynomial $f(X)$ mod p, then a variable of type `gf_element` holds a representation of a field element by its polynomial representation.

Description

A variable of type `gf_element` consists of an instance of type `galois_field` (indicating the field over which the element is defined) and the representation of the element. The internal representation varies for different classes of finite fields. Currently, we distinguish finite prime fields, extension fields of odd characteristic, and extension fields of characteristic 2. However, the interface of the class `gf_element` is completely independent of the internal representation.

Let e be an instance of type `gf_element`. If ff = e`.get_field()` and g denotes the polynomial ff`.irred_polynomial()`, then e represents the element (e`.polynomial_rep()` mod $g(X)$) $\in \mathbb{F}_p[X]/(g(X)\mathbb{F}_p[X])$, where $p = ff$`.characteristic()`.

Constructors/Destructor

`ct gf_element ()`
initializes the `gf_element` over a mainly useless dummy field.

`ct gf_element (const galois_field & `K`)`
constructs an element over the finite field K. The element is initialized with zero.

`ct gf_element (const gf_element & `e`)`
constructs a copy of e.

`dt ~gf_element ()`

Access Methods

Let `e` be an instance of type `gf_element`.

`galois_field` e`.get_field () const`
returns the field over which the element `e` is defined.

`const Fp_polynomial &` e`.polynomial_rep () const`
returns the polynomial representation of the element `e`.

Assignments

Let `e` be an instance of type `gf_element`.

`void` e`.set_polynomial_rep (const Fp_polynomial &` g`)`
sets e to the element g mod e`.get_field().irred_polynomial()`.

`void` e`.assign_zero ()`
$e \leftarrow 0$.

`void` e`.assign_one ()`
$e \leftarrow 1$.

`void` e`.assign_zero (const galois_field &` f`)`
e is assigned the zero element of the field ff.

`void` e`.assign_one (const galois_field &` f`)`
e is assigned the element 1 of the field ff.

`void` e`.assign (const bigint &` a`)`
e is assigned the element $a \cdot 1$ of the field ff.

The operator `=` is overloaded. For efficiency reasons, the following function is also implemented:

`void` e`.assign (const gf_element &` a`)`
$e \leftarrow a$.

Arithmetical Operations

Let `e` be an instance of type `gf_element`.

The following operators are overloaded and can be used in exactly the same way as for machine types in C++ (e.g. `int`) :

`(binary) +, -, *, /, +=, -=, *=, /=`

To avoid copying, these operations can also be performed by the following functions:

`void add (gf_element &` c`, const gf_element &` a`, const gf_element &` b`)`
$c \leftarrow a + b$.

`void add (gf_element &` c`, const bigint &` a`, const gf_element &` b`)`
$c \leftarrow a \cdot 1 + b$ over the field b`.get_field()`.

`void add (gf_element & `c`, const gf_element & `a`, const bigint & `b`)`
$c \leftarrow a + b \cdot 1$ over the field a`.get_field()`.

`void subtract (gf_element & `c`, const gf_element & `a`, const gf_element & `b`)`
$c \leftarrow a - b$.

`void subtract (gf_element & `c`, const bigint & `a`, const gf_element & `b`)`
$c \leftarrow a \cdot 1 - b$ over the field b`.get_field()`.

`void subtract (gf_element & `c`, const gf_element & `a`, const bigint & `b`)`
$c \leftarrow a - b \cdot 1$ over the field a`.get_field()`.

`void multiply (gf_element & `c`, const gf_element & `a`, const gf_element & `b`)`
$c \leftarrow a \cdot b$.

`void multiply (gf_element & `c`, const bigint & `a`, const gf_element & `b`)`
$c \leftarrow a \cdot 1 \cdot b$ over the field b`.get_field()`.

`void multiply (gf_element & `c`, const gf_element & `a`, const bigint & `b`)`
$c \leftarrow a \cdot b \cdot 1$ over the field a`.get_field()`.

`void divide (gf_element & `c`, const gf_element & `a`, const gf_element & `b`)`
$c \leftarrow a/b$, if $b \neq 0$. Otherwise the `lidia_error_handler` will be invoked.

`void divide (gf_element & `c`, const bigint & `a`, const gf_element & `b`)`
$c \leftarrow (a \cdot 1)/b$ over the field b`.get_field()`, if $b \neq 0$. Otherwise the `lidia_error_handler` will be invoked.

`void divide (gf_element & `c`, const gf_element & `a`, const bigint & `b`)`
$c \leftarrow a/(b \cdot 1)$ over the field a`.get_field()`, if $b \neq 0$. Otherwise the `lidia_error_handler` will be invoked.

`void `e`.negate ()`
$e \leftarrow -e$.

`void negate (gf_element & `a`, const gf_element & `b`)`
$a \leftarrow -b$.

`void `e`.multiply_by_2 ()`
$e \leftarrow 2 \cdot e$.

`void `e`.invert ()`
$e \leftarrow e^{-1}$, if $e \neq 0$. Otherwise the `lidia_error_handler` will be invoked.

`void invert (gf_element & `a`, const gf_element & `b`)`
$a \leftarrow b^{-1}$, if $b \neq 0$. Otherwise the `lidia_error_handler` will be invoked.

`gf_element inverse (const gf_element & `a`)`
returns a^{-1}, if $a \neq 0$. Otherwise the `lidia_error_handler` will be invoked.

`gf_element sqrt (const gf_element & a)`
returns x with $x^2 = a$.

`void square (gf_element & a, const gf_element & b)`
$a \leftarrow b^2$.

`void power (gf_element & c, const gf_element & a, const bigint & e)`
$c \leftarrow a^e$.

`void pth_power (gf_element & c, const gf_element & a, lidia_size_t e)`
$c \leftarrow a^{p^e}$, where p is the characteristic of the corresponding field.

Comparisons

Let `e` be an instance of type `gf_element`.

The binary operators `==`, `!=` are overloaded.

`bool e.is_zero () const`
returns `true` if $e = 0$, `false` otherwise.

`bool e.is_one () const`
returns `true` if $e = 1$, `false` otherwise.

Basic Methods and Functions

Let `e` be an instance of type `gf_element`.

`void e.randomize ()`
sets e to a random element of the field `e.get_field()`.

`void e.randomize (lidia_size_t d)`
sets e to a random element of the subfield K of `e.get_field()`, where K has (absolute) degree d. If d is not a divisor of `e.get_field().degree()`, then the `lidia_error_handler` is invoked.

`void swap (gf_element & a, gf_element & b)`
swaps the values of a and b.

`udigit hash (const gf_element & a)`
returns a hash value for a.

High-Level Methods and Functions

Let e be an instance of type `gf_element`.

`bigint e.order () const`
returns the order of e in the multiplicative group of `e.get_field()`.

`multi_bigmod` e`.trace () const`
returns the trace of e.

`multi_bigmod` e`.norm () const`
returns the norm of e.

`bool` e`.is_square () const`
returns `true` if e is a square, `false` otherwise.

`bool` e`.is_primitive_element () const`
checks whether the order of e in the multiplicative group is $p^n - 1$, where p is the characteristic and n is the degree of the corresponding field.

`bool` e`.is_free_element () const`
checks whether e is a free element, i.e. the generator of a normal base.

`gf_element` e`.assign_primitive_element (const galois_field &` *ff*`)`
sets e to a generator of the multiplicative group of the field *ff*.

`unsigned int` e`.get_absolute_degree () const`
returns the extension degree n of the current field over which e is defined.

`unsigned int` e`.get_relative_degree () const`
returns the minimal extension degree k of the field $\mathrm{GF}(p^k)$ over the prime field $\mathrm{GF}(p)$ such that e is an element of $\mathrm{GF}(p^k)$ (p denotes the characteristic of the corresponding field).

`bigint` e`.lift_to_Z () const`
If e is an element of a prime field $\mathrm{GF}(p)$ of characteristic p, then return the least non negative residue of the residue class of the class e mod p. Otherwise the `lidia_error_handler` is invoked.

`bool` e`.solve_quadratic (const gf_element&` a_1`, const gf_element&` a_0`)`
If the polynomial $X^2 + a_1X + a_0$ has a root over the field over which a_1 and a_0 are defined, then e is set to a root and `true` is returned. If the polynomial does not have a root, `false` is returned. If a_1 and a_0 are defined over different fields, the `lidia_error_handler` is invoked.

Input/Output

We distinguish between *verbose* and *short* I/O format. Let e be an instance of type `gf_element`. The verbose format is "(g(X), f(X))" where $g(X) =$ e`.polynomial_rep()` and $f(X) =$ `get_field().irred_polynomial()`. In case of a prime field $\mathrm{GF}(p) \cong \mathbb{Z}/p\mathbb{Z}$, the element m mod p can also be written as `(m, p)`.

For the short format, we assume that the finite field over which the element is defined is already known, i.e. if e`.get_field()` does not return the dummy field. We distinguish the following cases, depending on the finite field, which we will denote by K:

- $K = \mathrm{GF}(p)$ is a prime field: "`(m)`" means the element m mod $p \in K$;
- $K = \mathrm{GF}(p^n), p \neq 2, n > 1$: "`(g(X))`" means the element $g(X)$ mod K`.irred_polynomial()`;
- $K = \mathrm{GF}(2^n)$, $n > 1$: "`(Dec:`*d*`)`", and "`(Hex:`*d*`)`" mean the element $\sum_i a_i X^i$ mod 2 for $d = \sum_i a_i 2^i$, where d is an integer in decimal, or hexadecimal representation, respectively.

- $K = GF(p^n)$: "d" means the element $(\sum_i a_i X^i)$ mod K`.irred_polynomial()` for $d = \sum_i a_i p^i$, where d is an integer in decimal.

The `istream` operator `>>` and the `ostream` operator `<<` are overloaded.

`static void set_output_format (unsigned int` m`)`

sets the output format for all elements of the class `gf_element` to *verbose* if $m \neq 0$. Otherwise the output format will be *short*.

`unsigned int get_output_format ()`

returns 0, if the output format is set to "short", and 1 otherwise.

`istream & operator >> (istream & in, gf_element &` e`)`

reads an element of a finite field from `istream in`. The input must be of the format described above. If the input is in the short format, the field over which the element is defined must already be set beforehand, otherwise the `lidia_error_handler` is invoked.

`ostream & operator << (ostream & out, const gf_element &` e`)`

writes the element e to `ostream out` in verbose or short format, according to `gf_element::get_output_format()`.

See also

`galois_field`

Examples

```
#include <LiDIA/gf_element.h>

int main()
{
    bigint p;
    lidia_size_t d;

    cout << "Please enter the characteristic ";
    cout << "of the finite field : "; cin >> p;
    cout << "Please enter the degree of the ";
    cout << "finite field : "; cin >> d;

    galois_field field(p, d);
    cout << "This field has ";
    cout << field.number_of_elements();
    cout <<" elements.\n";

    cout << "The defining polynomial of the field is\n";
    field.irred_polynomial().pretty_print();
    cout << endl;

    gf_element elem;
    elem.assign_primitive_element(field);
    cout << elem << " is a primitive element in this field.\n";

    return 0;
}
```

Author

Detlef Anton, Franz-Dieter Berger, Stefan Neis, Thomas Pfahler

Chapter 11

Polynomials, Rational Functions, and Power Series

polynomial over gf_element

Name

`polynomial< gf_element >` ..... specialization of the polynomials

Abstract

`polynomial< gf_element >` is a class for computations with polynomials over finite fields. This class is a specialization of the general `polynomial< T >`, and you can apply all the functions and operators of the general class `polynomial< T >`. Moreover these classes support some additional functionality.

In the following we will describe the additional functions of the class `polynomial< gf_element >`.

Description

The specializations use the representation of the general class `polynomial< T >` in addition with a reference to an element of the class `galois_field` characterizing the field over which the polynomial is defined. So, every element of the class `polynomial< gf_element >` has "its own field", allowing to use polynomials over different finite fields at the same time. This field must be set explicitely if the polynomial is not the result of an arithmetical operation.

Constructors/Destructor

The specialization supports the same constructors as the general type, and the following one:

`ct polynomial< gf_element > (const galois_field &` K`)`
: initializes a zero polynomial over the field which is defined by K.

Basic Methods and Functions

All arithmetical operations are done in the field over which the polynomials are defined. In principal, the `lidia_error_handler` is invoked if the `const` arguments are not defined over the same finite fields. The "resulting" polynomial receives the field of the "input" polynomials.

Let f be of type `polynomial< gf_element >`:

`const galois_field &` f`.get_field () const`
: returns the finite field over which f is defined.

`void` f`.set_field (const galois_field &` K`)`
: sets the field over which f is defined. f is set to zero.

`void` f`.assign_zero (const galois_field &` K`)`
sets the field over which f is defined. f is set to zero.

`void` f`.assign_one (const galois_field &` K`)`
sets the field over which f is defined. f is set to the polynomial $1 \cdot x^0$.

`void` f`.assign_x (const galois_field &` K`)`
sets the field over which f is defined. f is set to the polynomial $1 \cdot x^1$.

Arithmetical Operations

In addition to the operators of `polynomial< T >` the following operators are overloaded:

binary	`polynomial< gf_element >`	*op*	`polynomial< gf_element >`	*op* $\in \{$`/`,`%`$\}$
binary with assignment	`polynomial< gf_element >`	*op*	`polynomial< gf_element >`	*op* $\in \{$`/=`,`%=`$\}$
binary	`polynomial< gf_element >`	*op*	`gf_element`	*op* $\in \{$`/`$\}$
binary with assignment	`polynomial< gf_element >`	*op*	`gf_element`	*op* $\in \{$`/=`$\}$

To avoid copying, these operations can also be performed by the following functions. Let f be of type `polynomial< gf_element >`.

`void div_rem (polynomial< gf_element > &` q`,const polynomial< gf_element > &` r`,`
`const polynomial< gf_element > &` f`, const polynomial< gf_element > &` g`)`
$f \leftarrow q \cdot g + r$, where $\deg(r) < \deg(g)$.

`void divide (polynomial< gf_element > &` q`, const polynomial< gf_element > &` f`,`
`const polynomial< gf_element > &` g`)`
Computes a polynomial q, such that $\deg(f - q \cdot g) < \deg(g)$.

`void divide (polynomial< gf_element > &` q`, const polynomial< gf_element > &` f`,`
`const gf_element &` a`)`
$q \leftarrow f/a$.

`void remainder (polynomial< gf_element > &` r`, const polynomial< gf_element > &` f`,`
`const polynomial< gf_element > &` g`)`
Computes r with $r \equiv f \pmod{g}$ and $\deg(r) < \deg(g)$.

`void invert (polynomial< gf_element > &` f`, const polynomial< gf_element > &` g`,`
`lidia_size_t` m`)`
$f \equiv g^{-1} \pmod{x^m}$. The constant term of g must be non-zero, otherwise the `lidia_error_handler` is invoked.

Greatest Common Divisor

`polynomial< gf_element > gcd (const polynomial< gf_element > &` f`,`
`const polynomial< gf_element > &` g`)`
returns $\gcd(f, g)$.

`polynomial< gf_element > xgcd (polynomial< gf_element > & `s`, polynomial< gf_element > & `t`, const polynomial< gf_element > & `f`, const polynomial< gf_element > & `g`)`

computes s and t such that $\gcd(f, g) = s \cdot f + t \cdot g$ and returns $\gcd(s, t)$

Modular Arithmetic without pre-conditioning

`void multiply_mod (polynomial< gf_element > & `g`, const polynomial< gf_element > & `a`, const polynomial< gf_element > & `b`, const polynomial< gf_element > & `f`)`

$g \leftarrow a \cdot b \bmod f$.

`void square_mod (polynomial< gf_element > & `g`, const polynomial< gf_element > & `a`, const polynomial< gf_element > & `f`)`

$g \leftarrow a^2 \bmod f$.

`void multiply_by_x_mod (polynomial< gf_element > & `g`, const polynomial< gf_element > & `a`, const polynomial< gf_element > & `f`)`

$g \leftarrow a \cdot x \bmod f$.

`void invert_mod (polynomial< gf_element > & `g`, const polynomial< gf_element > & `a`, const polynomial< gf_element > & `f`)`

$g \leftarrow a^{-1} \bmod f$. The `lidia_error_handler` is invoked if a is not invertible.

`bool invert_mod_status (polynomial< gf_element > & `g`, const polynomial< gf_element > & `a`, const polynomial< gf_element > & `f`)`

returns `true` and sets $g \leftarrow a^{-1} \bmod f$, if $\gcd(a, f) = 1$; otherwise returns `false` and sets $g \leftarrow \gcd(a, f)$.

`void power_mod (polynomial< gf_element > & `g`, const polynomial< gf_element > & `a`, const bigint & `e`, const polynomial< gf_element > & `f`)`

$g \leftarrow a^e \bmod f$. The polynomial f and the exponent e may not alias an output.

`void power_x_mod (polynomial< gf_element > & `g`, const bigint & `e`, const polynomial< gf_element > & `f`)`

$g \leftarrow x^e \bmod f$. The polynomial f and the exponent e may not alias an output.

`void power_x_plus_a_mod (polynomial< gf_element > & `g`, const bigint & `a`, const bigint & `e`, const polynomial< gf_element > & `f`)`

$g \leftarrow (x + a)^e \bmod f$. The polynomial f and the exponent e may not alias an output.

High-Level Methods and Functions

Let f be an instance of type `polynomial< gf_element >`.

`void cyclic_reduce (polynomial< gf_element > & `f`, const polynomial< gf_element > & `g`, lidia_size_t m)`

$f \leftarrow g \bmod x^m - 1$.

`base_vector< gf_element > find_roots (const polynomial< gf_element > & `f`)`

returns the list of roots of f. Assumes that f is monic and has $\deg(f)$ distinct roots; otherwise the behaviour of this function is undefined.

`polynomial< gf_element > randomize (const galois_field & K, lidia_size_t n)`
returns a random polynomial of degree n over the field defined by K.

Integration

`void integral (polynomial< gf_element > & f, const polynomial< gf_element > & g)`
computes f, such that $f' = g$.

`polynomial< gf_element > integral (const polynomial< gf_element > & g)`
returns a polynomial f, such that $f' = g$.

Input/Output

The `istream` operator `>>` and the `ostream` operator `<<` are overloaded. The input formats and the output format are the same as for the general class.

See also

`galois_field`, `gf_element`, `polynomial`

Author

Thomas Pfahler, Stefan Neis

Fp_polynomial

Name

`Fp_polynomial` ..... polynomials over finite prime fields

Abstract

`Fp_polynomial` is a class for doing very fast polynomial computation modulo a prime $p > 2$. A variable of type `Fp_polynomial` can hold polynomials of arbitrary length. Each polynomial has its own modulus p, which is of type `bigint`.

Description

A variable f of type `Fp_polynomial` is internally represented as a coefficient array with entries of type `bigint`. The zero polynomial is a zero length array; otherwise, $f[0]$ is the constant-term, and $f[\deg(f)]$ is the leading coefficient, which is always non-zero (except for $f = 0$). Furthermore, every `Fp_polynomial` carries a pointer to its modulus, thus allowing to use polynomials over different prime fields at the same time. This modulus must be set explicitely if f is not the result of an arithmetical operation.

For multiplication and division, we have implemented a very efficient FFT-arithmetic.

Constructors/Destructor

`ct Fp_polynomial ()`
initializes a zero polynomial.

`ct Fp_polynomial (const Fp_polynomial & f)`
initializes with a copy of the polynomial f.

`ct Fp_polynomial (const polynomial< bigint > & f, const bigint & p)`
initializes with $f \bmod p$.

`dt ~Fp_polynomial ()`

Assignments

Let f be of type `Fp_polynomial`.

The operator `=` is overloaded. For efficiency reasons, the following functions are also implemented:

`void` f`.assign_zero ()`
sets f to the zero polynomial. If f`.modulus()` $= 0$, the `lidia_error_handler` is invoked.

`void` f`.assign_zero (const bigint &` p`)`
sets f to the zero polynomial modulo p. If $p < 2$, the `lidia_error_handler` is invoked.

`void` f`.assign_one ()`
sets f to the polynomial $1 \cdot x^0$. If f`.modulus()` $= 0$, the `lidia_error_handler` is invoked.

`void` f`.assign_one (const bigint &` p`)`
sets f to the polynomial $1 \cdot x^0$ modulo p. If $p < 2$, the `lidia_error_handler` is invoked.

`void` f`.assign_x ()`
sets f to the polynomial $1 \cdot x^1 + 0 \cdot x^0$. If f`.modulus()` $= 0$, the `lidia_error_handler` is invoked.

`void` f`.assign_x (const bigint &` p`)`
sets f to the polynomial $1 \cdot x^1 + 0 \cdot x^0$ modulo p. If $p < 2$, the `lidia_error_handler` is invoked.

`void` f`.assign (const Fp_polynomial &` g`)`
$f \leftarrow g$.

`void` f`.assign (const bigint &` a`)`
$f \leftarrow a \cdot x^0$. If f`.modulus()` $= 0$, the `lidia_error_handler` is invoked.

`void` f`.assign (const base_vector< bigint > &` v`, const bigint &` p`)`
$f \leftarrow \sum_{i=0}^{\texttt{v.size()}-1} v[i] \cdot x^i \bmod p$. p must be prime; otherwise, the behaviour of this class is undefined. If $p < 2$, the `lidia_error_handler` is invoked. Leading zeros will be automatically removed.

Basic Methods and Functions

Let f be of type `Fp_polynomial`.

`void` f`.set_modulus (const bigint &` p`)`
sets the modulus for the polynomial f to p. p must be ≥ 2 and prime. The primality of p is not tested, but be aware that the behavior of this class is not defined if p is not a prime. f is set to zero.

`void` f`.set_modulus (const Fp_polynomial &` g`)`
sets the modulus for f to g`.modulus()`. f is assigned the zero polynomial.

`const bigint &` f`.modulus () const`
returns the modulus of f. If f has not been assigned a modulus yet (explicitely by f`.set_modulus()` or implicitely by an operation), the value zero is returned.

`void` f`.set_max_degree (lidia_size_t` n`)`
pre-allocates space for coefficients up to degree n. This has no effect if $n < \deg(f)$ or if sufficient space was already allocated. The value of f is always unchanged.

`void` f`.remove_leading_zeros ()`

removes leading zeros. Afterwards, if f is not the zero polynomial, f`.lead_coeff()` $\neq 0$. If f`.modulus()` $= 0$, the `lidia_error_handler` is invoked.

`void` f`.make_monic ()`

divides f by its leading coefficient; afterwards, f`.lead_coeff` $= 1$. If $f = 0$ or f`.modulus()` $= 0$, the `lidia_error_handler` is invoked.

`void` f`.kill ()`

deallocates the polynomial's coefficients and sets f to zero. f`.modulus()` is set to zero.

Access Methods

Let f be of type `Fp_polynomial`. All returned coefficients lie in the interval $[0, \ldots, f$`.modulus()` $- 1]$.

`lidia_size_t` f`.degree () const`

returns the degree of f. The zero polynomial has degree -1.

`bigint &` f`.operator[] (lidia_size_t` i`)`

returns a reference to the coefficient of x^i of f. i must be non-negative.

IMPORTANT NOTE: Assignments to coefficients via this operator are only possible by the following functions:

- `operator=(const bigint &)`
- `bigint::assign_zero()`
- `bigint::assign_one()`
- `bigint::assign(const bigint &)`

Any other assignment (e.g. `square(f[0], a)`) will only cause compiler warnings, but your program will not run correctly. The degree of f is adapted automatically after any assignment. Read access is always possible.

`const bigint &` f`.operator[] (lidia_size_t` i`) const`

returns a constant reference to the coefficient of x^i of f. i must be non-negative. If i exceeds the degree of f, a reference to a bigint of value zero is returned.

`bigint &` f`.lead_coeff () const`

returns $f[\deg(f)]$, i.e. the leading coefficient of f; returns zero if f is the zero polynomial.

`bigint &` f`.const_term () const`

returns $f[0]$, i.e. the constant term of f; returns zero if f is zero polynomial.

To avoid copying, coefficient access can also be performed by the following functions:

`void` f`.get_coefficient (bigint &` a`, int` i`) const`

$a \leftarrow c_i$, where $f = \sum_{k=0}^{\deg(f)} c_k \cdot x^k$. $a \leftarrow 0$, if $i > \deg(f)$. The `lidia_error_handler` is invoked if $i < 0$.

`void` f`.set_coefficient (const bigint &` a`, lidia_size_t` i`)`

sets coefficient of x^i to $a \bmod f$`.modulus()`; if $i > \deg(f)$, all coefficients c_j of x^j with $j = \deg(f) + 1, \ldots, i-1$ are set to zero; the degree of f is adapted automatically. The `lidia_error_handler` is invoked if $i < 0$.

`void` f`.set_coefficient (lidia_size_t` i`)`

This is equivalent to f`.set_coefficient(1,i)`. Sets coefficient of x^i to 1; if $i > \deg(f)$, all coefficients c_j of x^j with $j = \deg(f) + 1, \ldots, i-1$ are set to zero; the degree of f is adapted automatically. The `lidia_error_handler` is invoked if $i < 0$.

Arithmetical Operations

All arithmetical operations are done modulo the modulus assigned to each polynomial. Input variables of type `bigint` are automatically reduced. In principal, the `lidia_error_handler` is invoked if the moduli of the `const` arguments are not the same. The "resulting" polynomial receives the modulus of the "input" polynomials.

The following operators are overloaded and can be used in exactly the same way as for machine types in C++ (e.g. `int`) :

`(unary) -`
`(binary) +, -, *, /, %`
`(binary with assignment) +=, -=, *=, /=, %=`

Let f be of type `Fp_polynomial`. To avoid copying, these operations can also be performed by the following functions:

`void` f`.negate ()`
$f \leftarrow -f$.

`void negate (Fp_polynomial &` g`, const Fp_polynomial &` f`)`
$g \leftarrow -f$.

`void add (Fp_polynomial &` f`, const Fp_polynomial &` g`, const Fp_polynomial &` h`)`
$f \leftarrow g + h$.

`void add (Fp_polynomial &` f`, const Fp_polynomial &` g`, const bigint &` a`)`
$f \leftarrow g + a \bmod p$, where $p =$ g`.modulus()`.

`void add (Fp_polynomial &` f`, const bigint &` a`, const Fp_polynomial &` g`)`
$f \leftarrow g + a \bmod p$, where $p =$ g`.modulus()`.

`void subtract (Fp_polynomial &` f`, const Fp_polynomial &` g`, const Fp_polynomial &` h`)`
$f \leftarrow g - h$.

`void subtract (Fp_polynomial &` f`, const Fp_polynomial &` g`, const bigint &` a`)`
$f \leftarrow g - a \bmod p$, where $p =$ g`.modulus()`.

`void subtract (Fp_polynomial &` f`, const bigint &` a`, const Fp_polynomial &` g`)`
$f \leftarrow a - g \bmod p$, where $p =$ g`.modulus()`.

`void multiply (Fp_polynomial & f, const Fp_polynomial & g, const Fp_polynomial & h)`
$f \leftarrow g \cdot h$.

`void multiply (Fp_polynomial & f, const Fp_polynomial & g, const bigint & a)`
$f \leftarrow g \cdot a \bmod p$, where $p =$ `g.modulus()`.

`void multiply (Fp_polynomial & f, const bigint & a, const Fp_polynomial & g)`
$f \leftarrow g \cdot a \bmod p$, where $p =$ `g.modulus()`.

`void square (Fp_polynomial & f, const Fp_polynomial & g)`
$f \leftarrow g^2$.

`void divide (Fp_polynomial & q, const Fp_polynomial & f, const Fp_polynomial & g)`
$q \leftarrow f/g$.

`void divide (Fp_polynomial & q, const Fp_polynomial & f, const bigint & a)`
$q \leftarrow f/a \bmod p$, where $p =$ `f.modulus()`.

`void remainder (Fp_polynomial & r, const Fp_polynomial & f, const Fp_polynomial & g)`
$r \leftarrow f \bmod g$.

`void div_rem (Fp_polynomial & q, const Fp_polynomial & r, const Fp_polynomial & f,`
`const Fp_polynomial & g)`
$f \leftarrow q \cdot g + r$, where $0 \leq \deg(r) < \deg(g)$.

`void invert (Fp_polynomial & f, const Fp_polynomial & g, lidia_size_t m)`
$f \leftarrow g^{-1} \bmod x^m$. The constant term of g must be non-zero, otherwise the `lidia_error_handler` is invoked.

`void power (Fp_polynomial & x, const Fp_polynomial & g, lidia_size_t e)`
$f \leftarrow g^e$.

Greatest Common Divisor

`void gcd (Fp_polynomial & d, const Fp_polynomial & f, const Fp_polynomial & g)`
$d \leftarrow \gcd(f, g)$.

`Fp_polynomial gcd (const Fp_polynomial & f, const Fp_polynomial & g)`
returns $\gcd(f, g)$.

`void xgcd (Fp_polynomial & d, Fp_polynomial & s, Fp_polynomial & t,`
`const Fp_polynomial & f, const Fp_polynomial & g)`
$d \leftarrow \gcd(f, g) = s \cdot f + t \cdot g$.

Modular Arithmetic without pre-conditioning

`void multiply_mod (Fp_polynomial & g, const Fp_polynomial & a, const Fp_polynomial & b, const Fp_polynomial & f)`
$g \leftarrow a \cdot b \bmod f$.

`void square_mod (Fp_polynomial & g, const Fp_polynomial & a, const Fp_polynomial & f)`
$g \leftarrow a^2 \bmod f$.

`void multiply_by_x_mod (Fp_polynomial & g, const Fp_polynomial & a, const Fp_polynomial & f)`
$g \leftarrow a \cdot x \bmod f$.

`void invert_mod (Fp_polynomial & g, const Fp_polynomial & a, const Fp_polynomial & f)`
$g \leftarrow a^{-1} \bmod f$. The `lidia_error_handler` is invoked if a is not invertible.

`bool invert_mod_status (Fp_polynomial & g, const Fp_polynomial & a, const Fp_polynomial & f)`
returns `true` and sets $g \leftarrow a^{-1} \bmod f$, if $\gcd(a, f) = 1$; otherwise returns `false` and sets $g \leftarrow \gcd(a, f)$.

`void power_mod (Fp_polynomial & g, const Fp_polynomial & a, const bigint & e, const Fp_polynomial & f)`
$g \leftarrow a^e \bmod f$.

`void power_x_mod (Fp_polynomial & g, const bigint & e, const Fp_polynomial & f)`
$g \leftarrow x^e \bmod f$.

`void power_x_plus_a_mod (Fp_polynomial & g, const bigint & a, const bigint & e, const Fp_polynomial & f)`
$g \leftarrow (x + a)^e \bmod f$.

Comparisons

The binary operators `==`, `!=` are overloaded and can be used in exactly the same way as for machine types in C++ (e.g. `int`).

Let f be an instance of type `Fp_polynomial`.

`bool` f`.is_zero () const`
returns `true` if f is the zero polynomial; `false` otherwise.

`bool` f`.is_one () const`
returns `true` if f is a constant polynomial and the constant coefficient equals 1; `false` otherwise.

`bool` f`.is_x () const`
returns `true` if f is a polynomial of degree 1, the leading coefficient equals 1 and the constant coefficient i equals 0; `false` otherwise.

`bool` f`.is_monic () const`
returns `true` if f is a monic polynomial, i.e. if f is not the zero polynomial and f`.lead_coeff()` $= 1$.

High-Level Methods and Functions

Let f be an instance of type `Fp_polynomial`.

`void shift_left (Fp_polynomial & f, const Fp_polynomial & g, lidia_size_t n)`
$f \leftarrow g \cdot x^n$.

`void shift_right (Fp_polynomial & f, const Fp_polynomial & g, lidia_size_t n)`
$f \leftarrow g/x^n$.

`void trunc (Fp_polynomial & f, const Fp_polynomial & g, lidia_size_t n)`
$f \leftarrow g \pmod{x^n}$.

`void derivative (Fp_polynomial & f, const Fp_polynomial & g)`
$f \leftarrow g'$, i.e. the derivative of g.

`void f.randomize (lidia_size_t n)`
f is assigned a random polynomial of degree n modulo `f.modulus()`. If `f.modulus()` $= 0$, the `lidia_error_handler` is invoked.

`void randomize (Fp_polynomial & f, const bigint & p, lidia_size_t n)`
f is assigned a random polynomial modulo p of degree n.

`bigint f.operator() (const bigint & a) const`
returns $\sum_{i=0}^{\deg(f)} c_i \cdot a^i \bmod p$, where $f = \sum_{i=0}^{\deg(f)} c_i \cdot x^i$ and $p =$ `f.modulus()`. If `f.modulus()` $= 0$, the `lidia_error_handler` is invoked.

`void build_from_roots (const base_vector< bigint > & v)`
computes the polynomial $\prod_{i=0}^{v.\mathtt{size}()-1}(x - a[i]) \bmod$ `f.modulus()`. The `lidia_error_handler` is invoked if `f.modulus()` $= 0$.

`void cyclic_reduce (Fp_polynomial & f, const Fp_polynomial & g, lidia_size_t m)`
$f \leftarrow g \bmod x^m - 1$.

`void add_multiple (Fp_polynomial & f, const Fp_polynomial & g, const bigint & s,`
`lidia_size_t n, const Fp_polynomial & h)`
$f \leftarrow g + s \cdot x^n \cdot h \bmod p$, where $p =$ `g.modulus()`.

`bool prob_irred_test (const Fp_polynomial & f, lidia_size_t` *iter* `= 1)`
performs a fast, probabilistic irreducibility test. The test can err only if f is reducible modulo `f.modulus()`, and the error probability is bounded by `f.modulus()`$^{-iter}$.

`bool det_irred_test (const Fp_polynomial & f)`
performs a recursive deterministic irreducibility test.

`void build_irred (Fp_polynomial & f, cnost bigint & p, lidia_size_t n)`
$f \leftarrow$ a monic irreducible polynomial modulo p of degree n.

`void build_random_irred (Fp_polynomial & f, const Fp_polynomial & g)`
constructs a random monic irreducible polynomial f of degree $\deg(g)$ over $\mathbb{Z}/p\mathbb{Z}$, where $p =$ `g.modulus()`. Assumes g to be a monic irreducible polynomial (otherwise, the behaviour of this function is undefined).

`base_vector< bigint > find_roots (const Fp_polynomial & f, int flag = 0)`

returns the list of roots of f (without multiplicities). If *flag* $\neq 0$, f must be monic and the product of $\deg(f)$ distinct roots; otherwise no assumptions on f are made.

`bigint find_root (const Fp_polynomial & f)`

returns a single root of f. Assumes that f is monic and splits into distinct linear factors.

`void swap (Fp_polynomial & f, Fp_polynomial & g)`

swaps the values of f and g.

Input/Output

The `istream` operator `>>` and the `ostream` operator `<<` are overloaded. Let $f = \sum_{i=0}^{n} c_i \cdot x^i \bmod p$ where n denotes the degree and p the modulus of the polynomial f.

We support two different I/O-formats:

- The more simple format is

  ```
  [ c_0 c_1 ... c_n ] mod p
  ```

  with integers c_i, p. All numbers will be reduced modulo p at input; leading zeros will be removed.

- The more comfortable format (especially for sparse polynomials) is

  ```
  c_n * x^n + ... + c_2 * x^2 + c_1 * x + c_0 mod p
  ```

  At output, zero coefficients are omitted, as well as you may omit them at input. Even

  ```
  -x -x^2 +3*x^2 -17 +2 mod 5
  ```

  will be accepted.

Both formats may be used as input — they are distiguished automatically by the first character of the input, being '[' or not '['. The `ostream` operator `<<` always uses the first format. The second output format can be obtained using the member function

`void f.pretty_print (ostream & os) const`

See also

`bigint`bigint, `polynomial< T >`polynomial

Notes

If you want to do many computations modulo some fixed `Fp_polynomial` f, use class `Fp_poly_modulus`.

Warnings

Assignments to coefficients of polynomials via `operator[](lidia_size_t)` (without `const`) are only possible by the following functions:

- `operator=(const bigint &)`
- `bigint::assign_zero()`
- `bigint::assign_one()`
- `bigint::assign(const bigint &)`

Any other assignment (e.g. `square(f[0], a)`) will only cause compiler warnings, but your program will not run correctly.

Read access is always possible.

Examples

```
#include <LiDIA/Fp_polynomial.h>

int main()
{
    Fp_polynomial f, g, h;

    cout << "Please enter f : "; cin >> f;
    cout << "Please enter g : "; cin >> g;

    h = f * g;

    cout << "f * g  =  ";
    h.pretty_print();
    cout << endl;

    return 0;
}
```

Input example:

```
Please enter f : 14*x^4 - 43*x^3 + 1  mod 97
Please enter g : -x^3 + x - 2  mod 97

f * g  =  83*x^7 + 43*x^6 + 14*x^5 + 26*x^4 + 85*x^3 + x + 95 mod 97
```

Author

Victor Shoup (original author), Thomas Pfahler

Fp_poly_modulus

Name

`Fp_poly_modulus` ..... class for efficient computations modulo polynomials over finite prime fields
`Fp_poly_multiplier` ..... class for efficient multiplications modulo polynomials over finite prime fields

Abstract

If you need to do a lot of arithmetic modulo a fixed `Fp_polynomial` f, build a `Fp_poly_modulus` F for f. This pre-computes information about f that speeds up the computation a great deal, especially for large polynomials.

If you need to compute the product $a \cdot b$ mod f for a fixed `Fp_polynomial` b, but for many `Fp_polynomials` a (for example, when computing powers of b modulo f), it is much more efficient to first build a `Fp_poly_multiplier` B for b, and then use the multiplication routine below.

Description

The pre-computations for variables of type `Fp_poly_modulus` as well as for variables of type `Fp_poly_multiplier` consist of evaluating FFT-representations. For further description, see [59]. However, if the degree of the polynomials involved is small, pre-computations are not necessary. In this case, arithmetic with `Fp_poly_modulus` or `Fp_poly_multiplier` is not faster than modular arithmetic without pre-conditioning.

Constructors/Destructor

`ct Fp_poly_modulus ()`
: no initialization is done; you must call the member-function `build` (see below) before using this `Fp_poly_modulus` in one of the arithmetical functions.

`ct Fp_poly_modulus (const Fp_polynomial & f)`
: initializes for computations modulo f.

`dt ~Fp_poly_modulus ()`

`ct Fp_poly_multiplier ()`
: no initialization is done. The member-function `build` (see below) must be called before this instance can be used for multiplications.

`ct Fp_poly_multiplier (const Fp_polynomial & b, const Fp_poly_modulus & F)`
: initializes for multiplications with b modulo F`.modulus()`.

`dt ~Fp_poly_multiplier ()`

Basic Methods and Functions

Let F be of type `Fp_poly_modulus`. Let B be of type `Fp_poly_multiplier`.

`void` F`.build (const Fp_polynomial &` f`)`
initializes for computations modulo f.

`void` B`.build (const Fp_polynomial &` b`, const Fp_poly_modulus &` F`)`
initializes for multiplications with b modulo F`.modulus()`. If $\deg(b) \geq \deg(F$`.modulus()`$)$, the `lidia_error_handler` is invoked.

Access Methods

Let F be of type `Fp_poly_modulus`. Let B be of type `Fp_poly_multiplier`.

`const Fp_polynomial &` F`.modulus () const`
returns a constant reference to the polynomial for which F was build.

`const Fp_polynomial &` B`.multiplier () const`
returns a constant reference to the polynomial for which B was build.

Arithmetical Operations

As described in the section `Fp_polynomial`, each polynomial has its own modulus p of type `bigint`. The same is true for `Fp_poly_modulus` and `Fp_poly_multiplier` for they are only additional representations of polynomials over finite prime fields. Therefore, if the moduli of the `const` arguments are not the same, the `lidia_error_handler` is invoked. The "resulting" polynomial receives the modulus of the "input" polynomials.

`void remainder (Fp_polynomial &` g`, const Fp_polynomial &` a`, const Fp_poly_modulus &` F`)`
$g \leftarrow a \bmod F$`.modulus()`. No restrictions on $\deg(a)$.

`void multiply (Fp_polynomial &` g`, const Fp_polynomial &` a`, const Fp_polynomial &` b`,`
`const Fp_poly_modulus &` F`)`
$g \leftarrow a \cdot b \bmod F$`.modulus()`. If $\deg(a)$ or $\deg(b) \geq \deg(F$`.modulus()`$)$, the `lidia_error_handler` is invoked.

`void multiply (Fp_polynomial &` g`, const Fp_polynomial &` a`, const Fp_poly_multiplier &` B`,`
`const Fp_poly_modulus &` F`)`
$g \leftarrow a \cdot B$`.multiplier()` $\bmod F$`.modulus()`. If B is not initialized with F or $\deg(a) \geq \deg(F$`.modulus()`$)$, the `lidia_error_handler` is invoked.

`void square (Fp_polynomial &` g`, const Fp_polynomial &` a`, const Fp_poly_modulus &` F`)`
$g \leftarrow a^2 \bmod F$`.modulus()`. If $\deg(a) \geq \deg(F$`.modulus()`$)$, the `lidia_error_handler` is invoked.

`void power (Fp_polynomial &` g`, const Fp_polynomial &` a`, const bigint &` e`,`
`const Fp_poly_modulus &` F`)`
$g \leftarrow a^e \bmod F$`.modulus()`. If $e < 0$, the `lidia_error_handler` is invoked.

`void power_x (Fp_polynomial &` g`, const bigint &` e`, const Fp_poly_modulus &` F`)`
$g \leftarrow x^e \bmod F$`.modulus()`. If $e < 0$, the `lidia_error_handler` is invoked.

```
void power_x_plus_a (Fp_polynomial & g, const bigint & a, const bigint & e,
                     const Fp_poly_modulus & F)
```

$g \leftarrow (x+a)^e \bmod F$.`modulus()`. If $e < 0$, the `lidia_error_handler` is invoked.

High-Level Methods and Functions

```
void prob_min_poly (Fp_polynomial & h, const Fp_polynomial & g, lidia_size_t m,
                    const Fp_poly_modulus & F)
```

computes the monic minimal polynomial h of $g \bmod F$.`modulus()`. m is an upper bound on the degree of the minimal polynomial. The algorithm is probabilistic, always returns a divisor of the minimal polynomial, and returns a proper divisor with probability at most m/g.`modulus()`.

```
void min_poly (Fp_polynomial & h, const Fp_polynomial & g, lidia_size_t m,
               const Fp_poly_modulus & F)
```

same as `prob_min_poly`, but guarantees that result is correct.

```
void irred_poly (Fp_polynomial & h, const Fp_polynomial & g, lidia_size_t m,
                 const Fp_poly_modulus & F)
```

same as `prob_min_poly`, but assumes that F.`modulus()` is irreducible (or at least that the minimal polynomial of g is itself irreducible). The algorithm is deterministic (and hence is always correct).

```
void compose (Fp_polynomial & c, const Fp_polynomial & g, const Fp_polynomial & h,
              const Fp_poly_modulus & F)
```

$c \leftarrow g(h) \bmod F$.`modulus()`.

```
void trace_map (Fp_polynomial & w, const Fp_polynomial & a, lidia_size_t d,
                const Fp_poly_modulus & F, const Fp_polynomial & b)
```

$w \leftarrow \sum_{i=0}^{d-1} a^{q^i} \bmod F$.`modulus()`. It is assumed that $d \geq 0$, q is a power of F.`modulus().modulus()`, and $b \equiv x^q \pmod{F.\texttt{modulus()}}$, otherwise the behaviour of this function is undefined.

```
void power_compose (Fp_polynomial & w, const Fp_polynomial & b, lidia_size_t d,
                    const Fp_poly_modulus & F)
```

$w \leftarrow x^{q^d} \bmod F$.`modulus()`. It is assumed that $d \geq 0$, q is a power of F.`modulus().modulus()`, and $b \equiv x^q \pmod{F.\texttt{modulus()}}$, otherwise the behaviour of this function is undefined.

See also

`Fp_polynomial`

Examples

```
#include <LiDIA/Fp_polynomial.h>
#include <LiDIA/Fp_poly_modulus.h>
#include <LiDIA/Fp_poly_multiplier.h>

int main()
{
    Fp_polynomial a, b, f, x, y, z;
    Fp_poly_modulus F;
```

```
    Fp_poly_multiplier B;

    cout << "Please enter a : "; cin >> a;
    cout << "Please enter b : "; cin >> b;
    cout << "Please enter f : "; cin >> f;

    multiply_mod(x, a, b, f);

    F.build(f);
    multiply(y, a, b, F);

    B.build(b, F);
    multiply(z, a, B, F);

    //now, x == y == z

    cout << "a * b  mod  f  =  ";
    x.pretty_print(cout);
    cout << endl;

    return 0;
}
```

Example:

```
Please enter a : x^4 + 2 mod 97
Please enter b : x^3 + 1 mod 97
Please enter f : x^5 - 1 mod 97

a * b  mod  f  = x^4 + 2*x^3 + x^2 + 2 mod 97
```

Author

Victor Shoup (original author), Thomas Pfahler

gf_poly_modulus

Name

`gf_poly_modulus` ..... class for efficient computations modulo polynomials over finite fields

Abstract

If you need to do a lot of arithmetic modulo a fixed `polynomial< gf_element >` f, build a `gf_poly_modulus` F for f. This pre-computes information about f that speeds up the computation a great deal, especially for large polynomials.

Description

The pre-computations for variables of type `gf_poly_modulus` as well consist of evaluating special representations. For further description, see [50].

Constructors/Destructor

`ct gf_poly_modulus ()`
: no initialization is done; you must call the member-function `build()` (see below) before using this `gf_poly_modulus` in one of the arithmetical functions.

`ct gf_poly_modulus (const polynomial< gf_element > & f)`
: initializes for computations modulo f.

`dt ~gf_poly_modulus ()`

Basic Methods and Functions

Let F be of type `gf_poly_modulus`.

`void F.build (const polynomial< gf_element > & f)`
: initializes for computations modulo f.

Access Methods

Let F be of type `gf_poly_modulus`.

`const polynomial< gf_element > & F.modulus () const`
: returns a constant reference to the polynomial for which F was build.

Arithmetical Operations

As described in the section `polynomial< gf_element >`, each polynomial carries a reference to an element of type `galois_field` representing the finite field over which the polynomial is defined. Therefore, if the finite fields of the `const` arguments are not the same, the `lidia_error_handler` is invoked. The "resulting" polynomial receives the finite field of the "input" polynomials.

```
void remainder (polynomial< gf_element > & g, const polynomial< gf_element > & a,
                const gf_poly_modulus & F)
```
$g \leftarrow a \bmod F$`.modulus()`. No restrictions on $\deg(a)$.

```
void multiply (polynomial< gf_element > & g, const polynomial< gf_element > & a,
               const polynomial< gf_element > & b, const gf_poly_modulus & F)
```
$g \leftarrow a \cdot b \bmod F$`.modulus()`. If $\deg(a)$ or $\deg(b) \geq \deg(F$`.modulus()`$)$, the `lidia_error_handler` is invoked.

```
void square (polynomial< gf_element > & g, const polynomial< gf_element > & a,
             const gf_poly_modulus & F)
```
$g \leftarrow a^2 \bmod F$`.modulus()`. If $\deg(a) \geq \deg(F$`.modulus()`$)$, the `lidia_error_handler` is invoked.

```
void power (polynomial< gf_element > & g, const polynomial< gf_element > & a,
            const bigint & e, const gf_poly_modulus & F)
```
$g \leftarrow a^e \bmod F$`.modulus()`. If $e < 0$, the `lidia_error_handler` is invoked.

```
void power_x (polynomial< gf_element > & g, const bigint & e, const gf_poly_modulus & F)
```
$g \leftarrow x^e \bmod F$`.modulus()`. If $e < 0$, the `lidia_error_handler` is invoked.

```
void power_x_plus_a (polynomial< gf_element > & g, const bigint & a, const bigint & e,
                     const gf_poly_modulus & F)
```
$g \leftarrow (x + a)^e \bmod F$`.modulus()`. If $e < 0$, the `lidia_error_handler` is invoked.

High-Level Methods and Functions

```
void trace_map (polynomial< gf_element > & w, const polynomial< gf_element > & a,
                lidia_size_t d, const gf_poly_modulus & F,
                const polynomial< gf_element > & b)
```
$w \leftarrow \sum_{i=0}^{d-1} a^{q^i} \bmod F$`.modulus()`. It is assumed that $d \geq 0$, q is a power of the number of elements over which these polynomials are defined, and $b \equiv x^q \pmod{F\text{.modulus()}}$, otherwise the behaviour of this function is undefined.

See also

`polynomial< gf_element >`, `poly_modulus`

Examples

```
#include <LiDIA/polynomial.h>
#include <LiDIA/gf_element.h>

...
```

```
        polynomial< gf_element >  a, b, f, x, y;
        gf_poly_modulus F;

    ...

        multiply_mod(x, a, b, f);

        F.build(f);
        multiply(y, a, b, F);

        //now, x == y

    ...
```

Author

Victor Shoup (ideas), Thomas Pfahler

Fp_rational_function

Name

`Fp_rational_function` ..... rational functions over finite prime fields

Abstract

`Fp_rational_function` is a class for doing very fast computations with rational functions modulo an odd prime. This class is an extension of the polynomial class `Fp_polynomial`.

Description

`Fp_rational_function` is a class for doing very fast computations with rational functions modulo an odd prime p. A variable f of type `Fp_rational_function` is internally represented with two pointers to instances of type `Fp_polynomial`, the numerator and the denominator of f. Each `Fp_rational_function` has it's own modulus which must be set explicitely if f is not the result of an arithmetical operation (note the equivalence to `Fp_polynomial`). We use a "lazy evaluation" strategy in all operations, i.e. numerator and denominator of a `Fp_rational_function` are not necessarily coprime. There exists however a function to achieve coprimeness. Similar as in `Fp_polynomial` several high level functions like evaluation at a `bigint`, derivation and arithmetical operations exist. In addition to these ordinary arithmetical operations we have implemented modular versions where the numerator and the denominator are reduced modulo some given `Fp_polynomial`. These modular versions also exist for a polynomial modulus given by an instance of `poly_modulus`.

Constructors/Destructor

`ct Fp_rational_function ()`
: initializes a zero rational function with uninitialized modulus.

`ct Fp_rational_function (const bigint & p)`
: initializes a zero rational function with modulus p. If p is no odd prime number, the `lidia_error_handler` is invoked.

`ct Fp_rational_function (const Fp_polynomial & f)`
: initializes with a copy of the polynomial f. The modulus is set to the modulus of f.

`ct Fp_rational_function (const Fp_polynomial & n, const Fp_polynomial & d)`
: initializes with the rational function n/d. If the moduli of n and d are different, the `lidia_error_handler` is invoked.

`ct Fp_rational_function (const Fp_rational_function & f)`
initializes with a copy of the rational function f. The modulus is set to the modulus of f.

`dt ~Fp_rational_function ()`

Assignments

Let f be of type `Fp_rational_function`. The operator `=` is overloaded. All assignment functions set the modulus of the result to the modulus of the input variable (if possible). For efficiency reasons, the following functions are also implemented:

`void f.assign_zero ()`
sets f to the zero rational function. If the modulus of f is zero, the `lidia_error_handler` is invoked.

`void f.assign_one ()`
sets f to the rational function $1 \cdot x^0$. If the modulus of f is zero, the `lidia_error_handler` is invoked.

`void f.assign_x ()`
sets f to the rational function $1 \cdot x^1 + 0 \cdot x^0$. If the modulus of f is zero, the `lidia_error_handler` is invoked.

`void f.assign (const Fp_rational_function & g)`
$f \leftarrow g$.

`void f.assign (const Fp_polynomial & g)`
$f \leftarrow g$.

`void f.assign (const Fp_polynomial & n, const Fp_polynomial & d)`
sets $f \leftarrow n/d$. If the moduli of n and d are different, the `lidia_error_handler` is invoked.

`void f.assign_numerator (const Fp_polynomial & g)`
set the numerator of f to g. If the modulus of g is different from the modulus of f, the `lidia_error_handler` is invoked.

`void f.assign_denominator (const Fp_polynomial & g)`
set the denominator of f to g. If the modulus of g is different from the modulus of f, the `lidia_error_handler` is invoked.

Basic Methods and Functions

Let f be of type `Fp_rational_function`.

`void f.set_modulus (const bigint & p)`
sets the modulus for the rational function f to p. p must be an odd prime. The primality of p is not tested, but be aware that the behaviour of this class is not defined if p is not prime (feature inherited from `Fp_polynomial`).

`const bigint & f.modulus () const`
returns the modulus of f. If f has not been assigned a modulus yet (explicitely by `f.set_modulus()` or implicitly by an operation), the value zero is returned.

`void` f`.kill ()`
deallocates the rational function's coefficients and sets f to zero. The modulus of f is also set to zero.

Access Methods

Let f be of type `Fp_rational_function`. All returned coefficients of the following functions lie in the interval $[0, \ldots, p-1]$, where p is the modulus of f.

`Fp_polynomial &` f`.numerator ()`
returns a reference to the numerator of f.

`const Fp_polynomial &` f`.numerator () const`
returns a `const` reference to the numerator of f.

`Fp_polynomial &` f`.denominator ()`
returns a reference to the denominator of f.

`const Fp_polynomial &` f`.denominator () const`
returns a `const` reference to the denominator of f.

`lidia_size_t` f`.degree_numerator () const`
returns the degree of the numerator of f. For the zero rational function, the return value is -1.

`lidia_size_t` f`.degree_denominator () const`
returns the degree of the denominator of f. For the zero rational function the return value is -1.

`const bigint &` f`.lead_coefficient_numerator () const`
returns the leading coefficient of the numerator of f (zero if f is the zero rational function).

`const bigint &` f`.lead_coefficient_denominator () const`
returns the leading coefficient of the denominator of f. If f is the zero rational function, the `lidia_error_handler` is invoked.

`const bigint &` f`.const_term_numerator () const`
returns the constant term of the numerator of f (zero if f is the zero rational function).

`const bigint &` f`.const_term_denominator () const`
returns the constant term of the denominator of f. If f is the zero rational function, the `lidia_error_handler` is invoked.

`void` f`.get_coefficient_numerator (bigint &` a`, lidia_size_t` i`) const`
sets $a \leftarrow c_i$, where c_i is the coefficient of x^i of the numerator of f. If i is bigger than the degree of the numerator of f, then a is set to zero. For negative values of i the `lidia_error_handler` is invoked.

`void` f`.get_coefficient_denominator (bigint &` a`, lidia_size_t` i`) const`
sets $a \leftarrow c_i$, where c_i is the coefficient of x^i of the denominator of f. If i is bigger than the degree of the denominator of f, then a is set to zero. For negative values of i the `lidia_error_handler` is invoked.

`void` f`.set_coefficient_numerator (const bigint &` a`, lidia_size_t` i`)`

sets coefficient of x^i of the numerator of f to $a \pmod p$, where p is the modulus of f. Note that the function changes the degree of the numerator of f if i is bigger than the degree of the numerator of f. The `lidia_error_handler` is invoked for negative values of i.

`void` f`.set_coefficient_numerator (lidia_size_t` i`)`

sets coefficient of x^i of the numerator of f to one. Note that the function changes the degree of the numerator of f if i is bigger than the degree of the numerator of f. The `lidia_error_handler` is invoked for negative values of i.

`void` f`.set_coefficient_denominator (const bigint &` a`, lidia_size_t` i`)`

sets coefficient of x^i of the denominator of f to $a \pmod p$, where p is the modulus of f. Note that the function changes the degree of the denominator of f if i is bigger than the degree of the denominator of f. The `lidia_error_handler` is invoked for negative values of i.

`void` f`.set_coefficient_denominator (lidia_size_t` i`)`

sets coefficient of x^i of the denominator of f to one. Note that the function changes the degree of the denominator of f if i is bigger than the degree of the denominator of f. The `lidia_error_handler` is invoked for negative values of i.

Arithmetical Operations

All arithmetical operations are done modulo the modulus assigned to the input instances of `Fp_rational_function`. Input variables of type `bigint` are automatically reduced. The `lidia_error_handler` is invoked if the moduli of the input instances are different. The modulus of the result is determined by the modulus of the input rational functions.

The following operators are overloaded and can be used in exactly the same way as for machine types in C++ (e.g. `int`) :

```
                    (unary) -
                (binary) +, -, *, /
(binary with assignment) +=, -=, *=, /=
```

Let f be of type `Fp_rational_function`. To avoid copying, these operations can also be performed by the following functions:

`void` f`.negate ()`

$f \leftarrow -f$.

`void negate (Fp_rational_function &` f`, const Fp_rational_function &` g`)`

$f \leftarrow -g$.

`void add (Fp_rational_function &` f`, const Fp_rational_function &` g`,`
`const Fp_rational_function &` h`)`

$f \leftarrow g + h$.

`void subtract (Fp_rational_function &` f`, const Fp_rational_function &` g`,`
`const Fp_rational_function &` h`)`

$f \leftarrow g - h$.

`void multiply (Fp_rational_function & `f`, const Fp_rational_function & `g`,`
`  const Fp_rational_function & `h`)`
$f \leftarrow g \cdot h$.

`void multiply (Fp_rational_function & `f`, const Fp_rational_function & `g`,`
`  const Fp_polynomial & `a`)`
$f \leftarrow g \cdot a$.

`void multiply (Fp_rational_function & `f`, const Fp_polynomial & `a`,`
`  const Fp_rational_function & `g`)`
$f \leftarrow g \cdot a$.

`void multiply (Fp_rational_function & `f`, const Fp_rational_function & `g`,`
`  const bigint & `a`)`
$f \leftarrow g \cdot a$.

`void multiply (Fp_rational_function & `f`, const bigint & `a`,`
`  const Fp_rational_function & `g`)`
$f \leftarrow g \cdot a$.

`void `f`.multiply_by_2 ()`
$f \leftarrow 2 \cdot f$.

`void square (Fp_rational_function & `f`, const Fp_rational_function & `g`)`
$f \leftarrow g^2$.

`void divide (Fp_rational_function & `f`, const Fp_rational_function & `g`,`
`  const Fp_rational_function & `h`)`
$f \leftarrow g/h$.

`void divide (Fp_rational_function & `f`, const Fp_rational_function & `g`,`
`  const Fp_polynomial & `h`)`
$f \leftarrow g/h$.

`void divide (Fp_rational_function & `f`, const Fp_polynomial & `g`,`
`  const Fp_rational_function & `h`)`
$f \leftarrow g/h$.

`void `f`.invert ()`
$f \leftarrow 1/f$.

`void invert (Fp_rational_function & `f`, const Fp_rational_function & `g`)`
$f \leftarrow 1/g$.

`void div_rem (Fp_polynomial & `q`, Fp_rational_function & `f`)`
determine for given input value $f = n/d$ a polynomial q such that $n = q \cdot d + r$ with $\deg(r) < \deg(d)$. f is set to r/d.

`void shift (Fp_rational_function & `g`, const Fp_rational_function & `f`, lidia_size_t `n`)`
sets $g \leftarrow f \cdot x^n$.

Comparisons

The binary operators `==`, `!=` are overloaded and can be used in exactly the same way as for machine types in C++ (e.g. `int`). Let f be an instance of type `Fp_rational_function`.

`bool` f`.is_zero () const`
returns `true` if f is the zero rational function; `false` otherwise.

`bool` f`.is_one () const`
returns `true` if f is one; `false` otherwise.

High-Level Methods and Functions

Let f be an instance of type `Fp_rational_function`.

`void` f`.randomize (lidia_size_t` n`, lidia_size_t` d `= 0)`
assigns to f a randomly chosen rational function where the degree of the numerator and the denominator is n, d, respectively. If the modulus of f is zero, the `lidia_error_handler` is invoked.

`bigint` f`.operator () (const bigint &` a`) const`
evaluate the rational function f at the `bigint` a modulo the modulus of f and return the result. If the denominator of f is evaluated to zero, the `lidia_error_handler` is invoked.

`void derivative (Fp_rational_function &` f`, const Fp_rational_function &` g`)`
sets f to the derivative of g.

`Fp_rational_function derivative (const Fp_rational_function &` g`)`
returns the derivative of f.

`void swap (Fp_rational_function &` f`, Fp_rational_function &` g`)`
swaps the values of f and g.

Modular Arithmetic without pre-conditioning

All arithmetical operations shown above can also be performed modulo some given input variable f of type `Fp_polynomial`. These functions do not invert the denominator of the rational function modulo f, but perform modular polynomial operations for numerator and denominator ("lazy evaluation"). Therefore the degree of the numerator and the denominator of the resulting rational function are bounded by the degree of f. The `lidia_error_handler` is invoked if the moduli of the input instances and the polynomial modulus f are different. The modulus of the result is given as the modulus of the input rational functions.

`void` g`.reduce (const Fp_polynomial &` f`)`
reduce the numerator and the denominator of g modulo f.

`void add_mod (Fp_rational_function &` g`, const Fp_rational_function &` a`,`
`const Fp_rational_function &` b`, const Fp_polynomial &` f`)`
determines $g \leftarrow a + b \bmod f$.

`void subtract_mod (Fp_rational_function &` g`, const Fp_rational_function &` a`,`
`const Fp_rational_function &` b`, const Fp_polynomial &` f`)`
determines $g \leftarrow a - b \bmod f$.

`void multiply_mod (Fp_rational_function & g, const Fp_rational_function & a, const Fp_rational_function & b, const Fp_polynomial & f)`

determines $g \leftarrow a \cdot b \bmod f$.

`void square_mod (Fp_rational_function & g, const Fp_rational_function & a, const Fp_polynomial & f)`

determines $g \leftarrow a^2 \bmod f$.

`void divide_mod (Fp_rational_function & g, const Fp_rational_function & a, const Fp_rational_function & b, const Fp_polynomial & f)`

determines $g \leftarrow a/b \bmod f$.

`void convert_mod (Fp_polynomial & g, const Fp_rational_function & a, const Fp_polynomial & f)`

sets g to a polynomial which is equivalent to $a \pmod f$. If the denominator of a is not invertible modulo f, the `lidia_error_handler` is invoked.

`bool convert_mod_status (Fp_polynomial & g, const Fp_rational_function & a, const Fp_polynomial & f)`

If the denominator of a can be inverted modulo f, then the function sets g to a polynomial which is equivalent to $a \pmod f$ and returns `true`. Otherwise a is set to the gcd of the denominator of a and f and `false` is returned.

`bool equal_mod (const Fp_rational_function & g, const Fp_rational_function & h, const Fp_polynomial & f)`

returns `true` if g and h are equivalent modulo f; `false` otherwise.

Modular Arithmetic with pre-conditioning

If you want to do a lot of modular computations modulo the same modulus, then it is worthwhile to precompute some information about the modulus and store these information in a variable of type `poly_modulus` (see `poly_modulus`). Therefore all modular arithmetical operations shown above can also be performed modulo a given input variable f of type `poly_modulus`. The behaviour of these functions is exactly the same as for the modular functions without pre-conditioning.

Note that the class `poly_modulus` assumes that the degree of input polynomials is bounded by the degree of the polynomial modulus. This condition can be assured with the following member function which should therefore be called before the functions for modular arithmetic are used.

`void g.reduce (const poly_modulus & f)`

reduces the numerator and the denominator of g modulo the polynomial stored in f.

`void add (Fp_rational_function & g, const Fp_rational_function & a, const Fp_rational_function & b, const poly_modulus & f)`

determines $g \leftarrow a + b \bmod f$.

`void subtract (Fp_rational_function & g, const Fp_rational_function & a, const Fp_rational_function & b, const poly_modulus & f)`

determines $g \leftarrow a - b \bmod f$.

`void multiply (Fp_rational_function & g, const Fp_rational_function & a, const Fp_rational_function & b, const poly_modulus & f)`

determines $g \leftarrow a \cdot b \bmod f$.

```
void square (Fp_rational_function & g, const Fp_rational_function & a,
             const poly_modulus & f)
```
determines $g \leftarrow a^2 \bmod f$.

```
void divide (Fp_rational_function & g, const Fp_rational_function & a,
             const Fp_rational_function & b, const poly_modulus & f)
```
determines $g \leftarrow a/b \bmod f$.

```
bool equal (const Fp_rational_function & g, const Fp_rational_function & h,
            const poly_modulus & f)
```
returns `true` if g and h are equivalent modulo f; `false` otherwise.

Input/Output

The `istream` operator `>>` and the `ostream` operator `<<` are overloaded. We support two different I/O-formats:

- The more simple format is

  ```
  [ c_n ... c_0 ] / [ d_m ... d_0 ] mod p
  ```

  with integers c_i, d_j, p. Here c_i (d_i) is the coefficient of the numerator (denominator) of x^i, respectively. All numbers will be reduced modulo p at input; leading zeros will be removed.

- The more comfortable format (especially for sparse rational functions) is

  ```
  c_n * x^n + ... + c_2 * x^2 + c_1 * x + c_0 / d_m *
  x^m + ... + d_2 * x^2 + d_1 * x + d_0 mod p
  ```

  At output zero coefficients are omitted, as well as you may omit them at input.

In the case you want to input a polynomial, it is in both formats also possible to use the input format of class `Fp_polynomial` (i.e. ignore '/' and the denominator part). In this case the input format looks like

```
[ c_n ... c_0 ] mod p   ,    c_n*x^n + ... + c_0 mod p ,
```

respectively.

Both formats may be used as input — they are distinguished automatically by the first character of the input, being '[' or not '['. In addition to the `istream` operator `>>` the following function exists:

```
void f.read (istream & is = cin)
```

The `ostream` operator `<<` always uses the second output format. The first output format can be obtained using the member function

```
void f.print (ostream & os = cout) const
```

See also

`bigint`, `Fp_polynomial`, `poly_modulus`.

Examples

```
#include <LiDIA/Fp_rational_function.h>

int main()
{
    Fp_rational_function f, g, h;

    cout << "Please enter f : "; cin >> f;
    cout << "Please enter g : "; cin >> g;

    h = f * g;
    cout << "\n f * g   =  " << h;

    return 0;
}
```

Author

Volker Müller

power_series

Name

`dense_power_series< T >`, `sparse_power_series< T >` . . an arithmetic for approximations of power series

Abstract

`dense_power_series< T >` and `sparse_power_series< T >` are C++ template classes which represent series of the form

$$\sum_{i=n}^{\infty} a_i \cdot X^i \quad , \quad a_n \neq 0,\, a_i = 0,\, i < n,\, n \in \mathbb{Z}$$

by approximations

$$\sum_{i=n}^{M} a_i \cdot X^i \quad , \quad a_n \neq 0,\, M \in \mathbb{Z}_{\geq n}$$

where the coefficients a_i are elements of type `T` and the indices i of type `lidia_size_t`. The difference between `dense_power_series< T >` and `sparse_power_series< T >` is that in the dense representation all coefficients a_i, $n \leq i \leq M$, are stored, whereas in the sparse representation only the non-zero ones together with their corresponding exponents are kept. The two classes provide basic operations on power series like comparisons, addition, multiplication etc.

Description

An approximation of a power series

$$\sum_{i=n}^{M} a_i \cdot X^i \quad , \quad a_n \neq 0,\, M \in \mathbb{Z}_{\geq n}$$

can be represented in two different ways. A `dense_power_series< T >` stores the exponent n and the vector of the coefficients, e.g.

$$(n, a_n, \ldots, a_M) \ .$$

A `sparse_power_series< T >` only holds the non-zero coefficients together with their corresponding exponents, e.g.

$$\Big((a_{i_1}, i_1), \ldots, (a_{i_k}, i_k) \Big) \quad , \quad a_{i_j} \neq 0,\, n \leq i_j \leq M.$$

The exponent n is called the first exponent of the approximation, because it is the smallest exponent which belongs to a non-zero coefficient. Similarly, we call M the last exponent of the approximation, because it is the largest exponent of the series which the corresponding coefficient is known of. If all coefficients of an approximation are zero, so that there is no element $a_n \neq 0$, we can imagine this zero-approximation to be represented as

$$a_M \cdot X^M \quad , \quad a_M = 0$$

and we set the first and last exponent to M.

The coefficients a_i are type `T` elements, where `T` must be a C++ class which provides the following operators, member and friend functions

`istream operator>> (istream, T)`

`ostream operator<< (ostream, T)`

`bool operator== (T, T)`

`bool operator!= (T, T)`

`bool` a`.is_zero () const`
returns `true` if $a = 0$, `false` otherwise.

`bool` a`.is_one () const`
returns `true` if $a = 1$, `false` otherwise.

`void` a`.assign_zero ()`
$a \leftarrow 0$.

`void` a`.assign_one ()`
$a \leftarrow 1$.

`void add (T &` c`, T &` a`, T &` b`)`
$c \leftarrow a + b$.

`void subtract (T &` c`, T &` a`, T &` b`)`
$c \leftarrow a - b$.

`void multiply (T &` c`, T &` a`, T &` b`)`
$c \leftarrow a \cdot b$.

`void invert (T &` c`, T &` a`)`
$c \leftarrow 1/a$.

In the following description of operators, member and friend functions we will use the type psr as an abbreviation which can be substituted by `sparse_power_series< T >` or `dense_power_series< T >`, respectively, because nearly all functions are defined for both, sparse and dense representation.

Furthermore, for an approximation a of a series, we denote the first and last exponent of a (as defined above) by $f(a)$ and $l(a)$, respectively. By a_i we denote that coefficient of a which belongs to the exponent $i \in \mathbb{Z}$.

Constructors/Destructor

`ct psr ()`
sets the series to an uninitialized state; the first and last exponent are undefined. Any function call, beside setting coefficients, the series is involved in causes a call of the `lidia_error_handler`.

`ct psr (const T &` z`, lidia_size_t` l`)`
initializes the series with $z \cdot X^0 + \sum_{i=1}^{l} 0 \cdot X^i$
$f(this) = 0$, if $z \neq 0$, l otherwise $l(this) = l$.

`ct psr (const base_vector< T > & v, lidia_size_t f)`

initializes the series with $\sum_{i=f}^{f+v.\texttt{size()}-1} v[i-f] \cdot X^i$
$f \leq f(this) \leq l(this)$
$l(this) = f + v.\texttt{size()} - 1$.

`ct psr (const psr< T > & c)`

initializes the series with c.

Assignments

The operator= is overloaded. More precisely, the following assignments are possible:

`const dense_power_series & operator= (const dense_power_series< T > &)`

`const dense_power_series & operator= (const sparse_power_series< T > &)`

`const sparse_power_series & operator= (const sparse_power_series< T > &)`

`const sparse_power_series & operator= (const dense_power_series< T > &)`

`void a.assign_zero (lidia_size_t l)`

$a \leftarrow 0 \cdot X^l$
$f(c) \leftarrow l$
$l(c) \leftarrow l$.

`void a.assign_one (lidia_size_t l)`

$a \leftarrow 1 \cdot X^0 + \sum_{i=1}^{l} 0 \cdot X^i$
$f(a) \leftarrow 0$
$l(a) \leftarrow l$
If l is less than zero the `lidia_error_handler` will be invoked.

Arithmetical Operations

The operators `+`, `+=`, `-`, `-=`, `*`, `*=`, `/`, and `/=` are overloaded.

To avoid copying, these operations can also be performed by the following functions, where the output may always alias the input:

`void add (psr & c, const psr & a, const psr & b)`

$c \leftarrow a + b$
$l(c) \leftarrow \min(l(a), l(b))$
$\min(f(a), f(b)) \leq f(c) \leq l(c)$

`void add (psr & c, const psr & a, const T & b)`

$c \leftarrow a + \left(b \cdot X^0 + \sum_{i=1}^{l(a)} 0 \cdot X^i\right)$
$l(c) \leftarrow l(a)$
$f(a) \leq f(c) \leq l(c)$

`void add (psr & c, const T & a, const psr & b)`

$c \leftarrow \left(a \cdot X^0 + \sum_{i=1}^{l(b)} 0 \cdot X^i\right) + b$
$l(c) \leftarrow l(b)$
$f(b) \leq f(c) \leq l(c)$

`void subtract (psr & c, const psr & a, const psr & b)`

$c \leftarrow a - b$
$l(c) \leftarrow \min(l(a), l(b))$
$\min(f(a), f(b)) \leq f(c) \leq l(c)$

`void subtract (psr & c, const psr & a, const T & b)`

$c \leftarrow a - \left(b \cdot X^0 + \sum_{i=1}^{l(a)} 0 \cdot X^i\right)$
$l(c) \leftarrow l(a)$
$f(a) \leq f(c) \leq l(c)$

`void subtract (psr & c, const T & a, const psr & b)`

$c \leftarrow \left(a \cdot X^0 + \sum_{i=1}^{l(b)} 0 \cdot X^i\right) - b$
$l(c) \leftarrow l(b)$
$f(b) \leq f(c) \leq l(c)$

`void multiply (psr & c, const psr & a, const psr & b)`

$c \leftarrow a \cdot b$
$f(c) \leftarrow f(a) + f(b)$
$l(c) \leftarrow f(c) + \min(l(a) - f(a), l(b) - f(b))$
If a and b refer to the same object the `square(`c`,` a`)`-function is called in the case of `dense_power_series< T >`.

`void multiply (psr & c, const psr & a, const T & b)`

$c \leftarrow \sum_{i=f(a)}^{l(a)} (a_i \cdot b) \cdot X^i$
$f(c) \leftarrow f(a)$ if $b \neq 0$, otherwise $f(c) \leftarrow l(c)$
$l(c) \leftarrow l(a)$

`void multiply (psr & c, const T & a, const psr & b)`

$c \leftarrow \sum_{i=f(b)}^{l(b)} (a \cdot b_i) \cdot X^i$
$f(c) \leftarrow f(b)$ if $a \neq 0$, otherwise $f(c) \leftarrow l(c)$
$l(c) \leftarrow l(b)$

`void invert (psr & c, const psr & a)`

$c \leftarrow 1/a$
$f(c) \leftarrow -f(a)$
$l(c) \leftarrow -2 \cdot f(a) + l(a)$
If a is not invertible, the `lidia_error_handler` will be invoked.

`void divide (psr & c, const psr & a, const psr & b)`

$c \leftarrow a/b$
$f(c) \leftarrow f(a) - f(b)$
$l(c) \leftarrow f(c) + \min(l(a) - f(a), l(b) - f(b))$
If b is not invertible, the `lidia_error_handler` will be invoked.

`void divide (psr & c, const psr & a, const T & b)`

$c \leftarrow \sum_{i=f(a)}^{l(a)} (a_i \cdot (1/b)) \cdot X^i$
$f(c) \leftarrow f(a)$
$l(c) \leftarrow l(a)$

`void divide (psr & c, const T & a, const psr & b)`

$c \leftarrow \sum_{i=f(d)}^{l(d)} (a \cdot d_i) \cdot X^i$ where $d = 1/b$
$f(c) \leftarrow -f(b)$ if $a \neq 0$, otherwise $f(c) \leftarrow l(c)$
$l(c) \leftarrow -2 \cdot f(b) + l(b)$
If b is not invertible, the `lidia_error_handler` will be invoked.

`void square (psr & c, const psr & a)`

$c \leftarrow a^2$
$f(c) \leftarrow 2 \cdot f(a)$
$l(c) \leftarrow f(a) + l(a)$
At the moment, we use algorithms with quadratic running time to compute the product of two series. Therefore squaring is twice as fast as multiplying in the case of `dense_power_series< T >`.

`void power (psr & c, const psr & a, long n)`

$c \leftarrow a^n$
If $n = 0$ and a is a zero-approximation then $c \leftarrow 1 \cdot X^0$, $l(c) \leftarrow 0$;
if $n = 0$ and a is not a zero-approximation then $c \leftarrow 1 + \sum_{i=1}^{l(a)-f(a)} 0 \cdot X^i$;
if $n \neq 0$ and a is a zero-approximation then $c \leftarrow 0 \cdot X^n \cdot l(a)$, $f(c) \leftarrow l(c) \leftarrow n \cdot l(a)$;
otherwise $f(c) \leftarrow n \cdot f(a)$ and $l(c) \leftarrow (n-1) \cdot f(a) + l(a)$.

Let a be of type `psr`.

`void a.multiply_by_xn (lidia_size_t n)`

$a \leftarrow \sum_{i=f(a)}^{l(a)} a_i \cdot X^i + n$
$f(a) \leftarrow f(a) + n$
$l(a) \leftarrow l(a) + n$

`void a.compose (lidia_size_t n)`

$a \leftarrow \sum_{i=f(a)}^{l(a)} a_i \cdot (X^n)^i$
$f(a) \leftarrow f(a) \cdot n$
$l(a) \leftarrow l(a) \cdot n$

Comparisons

The binary operators `==` and `!=` are overloaded. Let a be of type `psr`.

`bool a.is_zero () const`
returns `true` if $a = 0 \cdot X^l, l \in \mathbb{Z}$, `false` otherwise.

`bool a.is_one () const`
returns `true` if $a = 1 + \sum_{i=1}^{l} 0 \cdot X^i, l \in \mathbb{Z}_{>0}$, `false` otherwise.

Basic Methods and Functions

Let a be of type `psr`.

`lidia_size_t a.get_first () const`
returns the first exponent of a.

`lidia_size_t` a`.get_last () const`

returns the last exponent of a.

`void` a`.set_coeff (const T &` z`, lidia_size_t` e`)`

sets the coefficient with exponent e to z. If e is less than $f(a)$ all coefficients with exponent i, $e < i < f(a)$, are set to zero just as all coeffients with exponent i, $l(a) < i < e$, in the case that e is greater than $l(a)$.

`void` a`.set (const psr &` b`)`

$a \leftarrow b$.

`void` a`.set (const T &` z`, lidia_size_t` l`)`

$a \leftarrow z \cdot X^0 + \sum_{i=1}^{l} 0 \cdot X^i$. If l is less than zero, a is set to $0 \cdot X^l$.

`void` a`.get_coeff (T &` z`, lidia_size_t` e`)`

assigns a_e to z. If e is greater than $l(a)$ the `lidia_error_handler` will be invoked.

`T` a`.operator[] (lidia_size_t` e`)`

returns a copy of a_e. If e is greater than $l(a)$ the `lidia_error_handler` will be invoked.

`void` a`.clear ()`

deletes all coefficients in a and resets a to an uninitialized state in the same way the default constructor does.

`void` a`.reduce_last (lidia_size_t` l`)`

If a is of the form $a = \sum_{i=f(a)}^{l(a)} a_i \cdot X^i$ before the call of this function, it will be set to $a \leftarrow \sum_{i=f(a)}^{l} a_i \cdot X^i$, where the coefficients with exponents between $l+1$ and $l(a)$ are cut off and $l(a)$ will be decreased to l. If l is greater than $l(a)$ the `lidia_error_handler` will be invoked. The memory reserved for the invalid coefficients, i.e. those coefficients that have been cut off before, is not deallocated, but these coefficients are not accessible anymore.

`void` a`.normalize ()`

deallocates memory which is occupied by invalid coefficients. This function is useful after several calls of a`.reduce_last()`

`void swap (psr &` a`, psr &` b`)`

exchanges the approximations of a and b.

Let a be of type `sparse_power_series< T >`.

`void` a`.set (const base_vector< T > &` v`, const base_vector< lidia_size_t > &` e`,`
`lidia_size_t` l`)`

builds the `power_series` a using the coefficient vector v and the exponent vector e. The coefficient $v[i]$ obtains the exponent $e[i]$ for $0 \leq i <$ v`.size()`. Furthermore, the last exponent of a is set to l and l must be greater than or equal to the maximum of the exponents $e[i]$.

`void` a`.set (const T*` v`, const lidia_size_t*` e`, lidia_size_t` sz`, lidia_size_t` l`)`

builds the `power_series` a using the coefficient vector v and the exponent vector e. The coefficient $v[i]$ obtains the exponent $e[i]$ for $0 \leq i < sz$. Furthermore, the last exponent of a is set to l and l must be greater than or equal to the maximum of the exponents $e[i]$.

`void` a`.get (base_vector< T > &` v`, base_vector< lidia_size_t > &` e`)`

assigns the non-zero coefficients of a to v and stores the corresponding exponents in e, i.e. $e[i]$ holds the exponent of $v[i]$ and the exponents are in increasing order. The capacity and the sizes of v and e are set to the number of non-zero elements of a. In the special case that a is a zero-approximation, the sizes and capacities of v and e are set to one, $v[0]$ is set to zero and $e[0]$ to $l(a)$.

`void` a`.get (T* &` v`, lidia_size_t* &` e`, lidia_size_t &` sz`)`

assigns the non-zero coefficients of a to v and stores the corresponding exponents in e, i.e. $e[i]$ holds the exponent of $v[i]$ and the exponents are in increasing order. The variable sz holds the size of v and e which is the number of elements of v. Especially, if a is a zero-approximation, the sizes of v and e are set to one, $v[0]$ is set to zero and $e[0]$ to $l(a)$.

Let a be of type `dense_power_series< T >`.

`void` a`.set (const base_vector< T > &` v`, lidia_size_t` f`)`
$a \leftarrow \sum_{i=f}^{f+\texttt{v.size()}-1} v[i-f] \cdot X^i$.

`void` a`.set (const T*` v`, lidia_size_t` sz`, lidia_size_t` f`)`
$a \leftarrow \sum_{i=f}^{f+sz-1} v[i-f] \cdot X^i$.

`void` a`.get (base_vector< T > &` v`)`
$v[i-f(a)] \leftarrow a_i$, $f(a) \le i \le l(a)$
v`.capacity()` $\leftarrow$ v`.size()` $= l(a) - f(a) + 1$
Especially, if a is a zero-approximation, the size and capacity of v are set to one and $v[0]$ is set to zero.

`void` a`.get (T* &` v`, lidia_size_t &` sz`)`
sets $sz \leftarrow l(a) - f(a) + 1$, allocates sz coefficients for v, and sets
$v[i-f(a)] \leftarrow a_i$, $f(a) \le i \le l(a)$. Especially, if a are a zero-approximation, v only holds one zero-element.

For efficiency reasons, the class `dense_power_series< T >` puts at disposal the following operator :

`T &` a`.operator() (lidia_size_t` e`)`

returns a reference to the coefficient with exponent e.

This operator allows the setting of coefficients without initializing a temporary variable, x, as it is necessary if the member function a`.set_coeff(T &` x `,lidia_size_t)` is used. Furthermore, it is possible to read coefficients without copying, as it is necessary when using the operator `T` a`.operator[](lidia_size_t)`.

But the `operator()` always assumes that the user intends to set the returned coefficient. Therefore, whenever e is less than $f(a)$, memory for the coefficients with exponent i, $e \le i < f(a)$, is allocated just as for the coefficients with exponent i, $l(a) < i \le e$, in the case that e is greater than $l(a)$. These coefficients are initialized with zero.

If the operator is used for reading cofficients and, by mistake, the exponent e is greater than $l(a)$, this operation may cause an incorrect approximation, because it sets the last exponent to e and the coefficients in between to zero.

Furthermore, zero coefficients whose exponents are less than the first exponent of an approximation should not be set using this operator, because then these coefficients are explicitly stored. The class `dense_power_series< T >` is able to handle leading zeros, but the running time of arithmetical operations increases.

Input/Output

The `istream` operator `>>` and the `ostream` operator `<<` are overloaded.

The input and output format of type `dense_power_series< T >` is of the form

$$[f\ [a_f \ldots a_l]]$$

and symbolizes the approximation

$$\sum_{i=f}^{l} a_i \cdot X^i \ .$$

The operator`>>` can handle leading zeros, i.e. $a_f = 0$.

The input and output format of type `sparse_power_series< T >` is of the form

$$[[(b_{i_1}, i_1), \ldots, (b_{i_k}, i_k)]\ l]$$

and symbolizes the approximation

$$\sum_{i=\min(i_1,\ldots,i_k)}^{l} a_i \cdot X^i \ ,$$

with $a_i = b_i$, if $i \in \{i_1, \ldots, i_k\}$, 0 otherwise. The operator`>>` can handle zero coefficients, i.e. $b_i = 0$.

Notes

The two data structures `dense_power_series< T >` and `sparse_power_series< T >` use the LiDIA class `base_vector< T >` to store the coefficients of a series. Therefore, the number of necessary memory allocations when initializing a series by setting coefficients, i.e. using the member function `set_coeff()` or the `operator()`, depends on the order in which the coefficients are set. To minimize the running time of this process, setting of coefficients should be done in the following way :

`dense_power_series< T >`

1. Set the coefficient that belongs to the first exponent.
2. Set the coefficient that belongs to the last exponent.
3. Set the remaining non-zero coefficients.

`sparse_power_series< T >`

1. Set the coefficient that belongs to the first exponent.
2. Set the non-zero coefficient with the largest exponent.
3. Set the remaining non-zero coefficients and if the coefficient which corresponds to the last exponent is zero, set this coefficient too.

If you must instantiate the power series classes with one of your own types, you have to change two instantiation files:

1. Add an entry for your type in `src/templates/powser/makefile.inst` (s. `Lp_bigint.o`).
2. Add an entry for `spc< your type >` in `src/templates/vector/makefile.inst` (s. `Lv_spc_bigint.o`).

For the general concept of template instantiation in LiDIA we refer to section 2.3 of the manual.

Examples

```
#include <LiDIA/dense_power_series.h>

main()
{
    dense_power_series< bigint > a, b, c;

    cout << "Please enter a : "; cin >> a ;
    cout << "Please enter b : "; cin >> b ;
    cout << endl;

    c = a + b;

    cout << "a + b = " << c << endl;
}
```

For further examples please refer to `LiDIA/src/templates/dense_power_series_appl.cc` and `LiDIA/src/templates/sparse_power_series_appl.cc`.

Author

Frank Lehmann, Markus Maurer

Chapter 12

Factorization

The description of the general factorization template class can be found in the section “LiDIA base package” (see page 181).

single_factor< Fp_polynomial >

Name

`single_factor< Fp_polynomial >` .....a single factor of a polynomial over finite prime fields

Abstract

`single_factor< Fp_polynomial >` is used for storing factorizations of polynomials over finite prime fields (see `factorization`). It is a specialization of `single_factor< T >` with some additional functionality. All functions for `single_factor< T >` can be applied to objects of class `single_factor< Fp_polynomial >`, too. These basic functions are not described here any further; you will find the description of the latter in `single_factor< T >`.

Description

Modifying Operations

Let a be an instance of type `single_factor< Fp_polynomial >`.

`bigint` a`.extract_unit ()`
: returns the leading coefficient of a`.base()` and converts it to a monic polynomial (by dividing it by its leading coefficient).

Factorization Algorithms

The implementation of the factorization algorithms is described in [50].

Let a be an instance of type `single_factor< Fp_polynomial >`.

`factorization< Fp_polynomial > square_free_decomp (const Fp_polynomial &` f`)`
: returns the factorization of f into square-free factors. f must be monic, otherwise the `lidia_error_handler` is invoked.

`factorization< Fp_polynomial >` a`.square_free_decomp () const`
: returns `square_free_decomp(`a`.base())`.

`factorization< Fp_polynomial > berlekamp (const Fp_polynomial &` f`)`
: returns the complete factorization of f using Berlekamp's factoring algorithm. If f is the zero polynomial, the `lidia_error_handler` is invoked.

`factorization< Fp_polynomial > `a`.berlekamp () const`

returns `berlekamp(`a`.base())`.

`factorization< Fp_polynomial > sf_berlekamp (const Fp_polynomial &` f`)`

returns the complete factorization of f using Berlekamp's factoring algorithm. f must be monic and square-free with a non-zero constant term; otherwise the behaviour of this function is undefined.

`factorization< Fp_polynomial > `a`.sf_berlekamp () const`

returns `sf_berlekamp(`a`.base())`.

`factorization< Fp_polynomial > can_zass (const Fp_polynomial &` f`)`

returns the complete factorization of f using `sf_can_zass`. If f is the zero polynomial, the `lidia_error_handler` is invoked. Note that as `ddf` creates temporary files, you need write permission either in the `/tmp`-directory or in the directory from which it is called.

This function uses less memory than `berlekamp` when factoring polynomials of large degrees.

`factorization< Fp_polynomial > `a`.can_zass () const`

returns `can_zass(`a`.base())`.

`factorization< Fp_polynomial > sf_can_zass (const Fp_polynomial &` f`)`

returns the complete factorization of f using `ddf` and the von zur Gathen/Shoup algorithm for factoring polynomials whose irreducible factors all have the same degree.

f must be monic and square-free, otherwise the behaviour of this function is undefined. Note that as `ddf` creates temporary files, you need write permission either in the `/tmp` directory or in the directory from which it is called.

This function uses less memory than `sf_berlekamp` when factoring polynomials of large degrees.

`factorization< Fp_polynomial > `a`.sf_can_zass () const`

returns `sf_can_zass(`a`.base())`.

`factorization< Fp_polynomial > ddf (const Fp_polynomial &` f`)`

performs a "distinct degree factorization".

Returns the factorization of f into a product of factors, where each factor is the product of distinct irreducibles all of the same degree, using Shoup's algorithm [59]. *The exponents* of this factorization do not give the multipicities of these factors (which would be 1) *but the degrees* of the irreducibles.

f must be monic and square-free, otherwise the behaviour of this function is undefined. Note that as `ddf` creates temporary files, you need write permission either in the `/tmp` directory or in the directory from which it is called.

`factorization< Fp_polynomial > `a`.ddf () const`

returns `ddf(`a`.base())`.

`factorization< Fp_polynomial > edf (const Fp_polynomial &` f`, lidia_size_t` d`)`

performs an "equal degree factorization". Assumes that f is a monic polynomial whose irreducible factors all have degree d (otherwise, the behaviour of this function is undefined); returns the complete factorization of f using an algorithm of von zur Gathen/Shoup [62].

`factorization< Fp_polynomial > `a`.edf (lidia_size_t` d`) const`

returns `edf(`a`.base(),` d`)`.

`factorization< Fp_polynomial > factor (const Fp_polynomial & f)`

returns the complete factorization of f using a strategy which tries to apply always the fastest of the above algorithms. It also includes special variants for factoring binomials. The strategy is explained in detail in [50]. If f is the zero polynomial, the `lidia_error_handler` is invoked.

`factorization< Fp_polynomial > a.factor () const`

returns `factor(a.base())`.

See also

`factorization< T >`, `Fp_polynomial`

Warnings

Using `can_zass`, `sf_can_zass` or `ddf` requires write permission in the `/tmp` directory or in the directory from which they are called. This is necessary because `ddf` creates temporary files.

Examples

```
#include <LiDIA/Fp_polynomial.h>
#include <LiDIA/factorization.h>

int main()
{
    Fp_polynomial f;
    factorization< Fp_polynomial > u;

    cout << "Please enter f : "; cin >> f ;
    u = factor(f);
    cout << "\nFactorization of f :\n" << u << endl;

    return 0;
}
```

For further examples please refer to `LiDIA/src/templates/factorization/Fp_polynomial/Fp_pol_factor_appl.cc`.

Author

Victor Shoup, Thomas Pfahler

single_factor< gf_polynomial >

Name

`single_factor< polynomial< gf_p_element > >` .....a single factor of a polynomial over finite fields

Abstract

`single_factor< polynomial< gf_p_element> >` is used for storing factorizations of polynomials over finite fields (see `factorization`). It is a specialization of `single_factor< T >` with some additional functionality. All functions for `single_factor< T >` can be applied to objects of class `single_factor< polynomial< gf_p_element > >`, too. These basic functions are not described here any further; you will find the description of the latter in `single_factor< T >`.

Description

Modifying Operations

Let a be an instance of type `single_factor< polynomial< gf_p_element > >`.

`bigint` a`.extract_unit ()`
: returns the leading coefficient of a`.base()` and converts it to a monic polynomial (by dividing it by its leading coefficient).

Factorization Algorithms

The implementation of the factorization algorithms is described in [50].

Let a be an instance of type `single_factor< polynomial< gf_p_element > >`.

`factorization< polynomial< gf_p_element> > square_free_decomp (const polynomial< gf_p_element > &` f`)`
: returns the factorization of f into square-free factors. f must be monic, otherwise the `lidia_error_handler` is invoked.

`factorization< polynomial< gf_p_element> >` a`.square_free_decomp () const`
: returns `square_free_decomp(`a`.base())`.

`factorization< polynomial< gf_p_element > > berlekamp (const polynomial< gf_p_element > &` f`)`
: returns the complete factorization of f using Berlekamp's factoring algorithm. If f is the zero polynomial, the `lidia_error_handler` is invoked.

`factorization< polynomial< gf_p_element > > a.berlekamp () const`

returns `berlekamp(a.base())`.

`factorization< polynomial< gf_p_element > > sf_berlekamp (const polynomial< gf_p_element > & f)`

returns the complete factorization of f using Berlekamp's factoring algorithm. f must be monic and square-free with a non-zero constant term; otherwise the behaviour of this function is undefined.

`factorization< polynomial< gf_p_element > > a.sf_berlekamp () const`

returns `sf_berlekamp(a.base())`.

`factorization< polynomial< gf_p_element > > can_zass (const polynomial< gf_p_element > & f)`

returns the complete factorization of f using `sf_can_zass`. If f is the zero polynomial, the `lidia_error_handler` is invoked.

`factorization< polynomial< gf_p_element > > a.can_zass () const`

returns `can_zass(a.base())`.

`factorization< polynomial< gf_p_element > > sf_can_zass (const polynomial< gf_p_element > & f)`

returns the complete factorization of f using a DDF according to Cantor/Zassenhaus, and the von zur Gathen/Shoup algorithm for factoring polynomials whose irreducible factors all have the same degree. f must be monic and square-free, otherwise the behaviour of this function is undefined.

`factorization< polynomial< gf_p_element > > a.sf_can_zass () const`

returns `sf_can_zass(a.base())`.

`factorization< polynomial< gf_p_element > > factor (const polynomial< gf_p_element > & f)`

returns the complete factorization of f using a strategy which tries to apply always the fastest of the above algorithms. It also includes special variants for factoring binomials. The strategy is explained in detail in [50]. If f is the zero polynomial, the `lidia_error_handler` is invoked.

`factorization< polynomial< gf_p_element > > a.factor () const`

returns `factor(a.base())`.

`bool  det_irred_test (const polynomial< gf_p_element > & f)`

performs a deterministic irreducibility test (based on `berlekamp`).

See also

`factorization`, `polynomial< gf_p_element >`

Examples

For examples please refer to `LiDIA/src/templates/factorization/gf_polynomial/gf_pol_factor_appl.cc`.

Notes

The main ideas for speeding up polynomial computation and factorization algorithms are attributed to Victor Shoup [59]

Author

Thomas Pfahler

Chapter 13

Miscellaneous

fft_prime

Name

`fft_prime` ..... fast fourier transformation modulo a prime

Abstract

`fft_prime` is a class that provides routines for computing discrete fourier transformations in finite prime fields with single precision characteristic.

Description

An object of type `fft_prime` stores information which are necessary to multiply single precision polynomials over a finite prime field using the Discrete Fourier Transformation (DFT).

To initialize an `fft_prime` object, the user must set the characteristic of the prime field using the member function `set_prime`. Then one of the functions for multiplying polynomials using DFT can be used.

For further information on the DFT algorithm, we refer to [19].

Constructors/Destructor

`ct fft_prime ()`
Initializes the characteristic of the field with zero.

`ct fft_prime (const fft_prime & a)`
Initializes a copy of a.

`dt ~fft_prime ()`
deletes all tables

Assignments

Let a be of type `fft_prime`. The operator `=` is overloaded. For efficiency reasons, the following function is also implemented:

`void a.assign (const fft_prime & b)`
$a \leftarrow b$.

Basic Methods and Functions

Let a be of type `fft_prime`.

`void` a`.set_prime (udigit` q`)`

Sets $p \leftarrow q$, where p is the characteristic of the finite prime field stored in the class. The parameter q must be a prime number larger than 2. The primality of q is not verified, but the behaviour is undefined, if q is not a prime number larger than 2. All further computations are done modulo q.

`udigit` a`.get_prime () const`

Returns the prime number which was set by the last call of `set_prime`.

`lidia_size_t` a`.get_max_degree () const`

Returns m, where 2^m is the largest possible convolution size for the current prime field. If the characteristic is zero, e.g. it has not been initialized, the `lidia_error_handler` is called.

High-Level Methods and Functions

`bool multiply_fft (void*` x`, int` *type*`, const void* const` pa`, lidia_size_t` da`,`
`const void* const` pb`, lidia_size_t` db`, fft_prime &` q`)`

pa and pb must represent polynomials of degree da and db modulo p, respectively, where $p = q$`.get_prime()`. The coefficients must be represented by the least non-negative element in the equivalence class modulo p. The constant term of the polynomials pa and pb must be stored in $pa[0]$ and $pb[0]$, respectively.

The product $pa \cdot pb$ is assigned to x. Adequate space for x must be allocated by the caller. The minimum size for x is the $da + db + 1$.

The parameters x, pa, and pb can be either of type `udigit` or `udigit_mod`. In the first case the parameter *type* has to be 0, in second case 1.

Squarings are detected automatically. Input may alias output.

If da (db) is less than or equal to zero, the `lidia_error_handler` is called.

The function returns false, if the next power of two larger than $da + db$ does not divide $p - 1$. In that case a DFT transformation is not possible. Otherwise, it returns true.

See also

`udigit`

Examples

The code of the example can be found in: `LiDIA/src/simple_classes/fft_prime/fft_prime_appl.cc`.

```
#include <LiDIA/udigit.h>
#include <LiDIA/fft_prime.h>

int main()
{
    lidia_size_t i, da ,db;
    fft_prime fft;
```

```
    fft.set_prime(17);

    fft_prime_t x[9], a[4], b[4];
    // pa =    1  + 2x^1  + 3x^2  + 4x^3
    a[0]=1; a[1]=2; a[2]=3; a[3]=4; da = 3;

    // pb =    1  + 2x^1  + 3x^2  + 4x^3
    b[0]=1; b[1]=2; b[2]=3; b[3]=4; db = 3; // a and b have degree 3 !!

    multiply_fft(x, a, da, b, db, fft);

    // output
    cout << "\n  pa:   ";
    for (i = 0; i <= da; i++){
        cout << a[i] << " ";
    }

    cout << "\n  pb:   ";
    for (i = 0; i <= db; i++) {
        cout << b[i] << " ";
    }

    cout << "\nproduct:";
    for (i = 0; i <= da + db + 1; i++) {
        cout << x[i] << " ";
    }

    return 0;
}
```

Author

Thomas Pfahler, Thorsten Rottschaefer, and Victor Shoup

The LiDIA LA package

The LiDIA LA package contains the classes which deal with linear algebra over the ring of rational integers and over $\mathbb{Z}/m\mathbb{Z}$. It requires the LiDIA base package and the LiDIA FF package. The description of the general matrix template classes can be found in the section "LiDIA base package" (see page 127).

Chapter 14

Matrices

bigint_matrix

Name

`bigint_matrix` ..... Linear algebra over $\mathbb{Z}$

Abstract

`bigint_matrix` is a class for doing linear algebra over the ring of rational integers. This class may either be used for representing a modul, which is generated by the columns of the matrix

$$A = (a_{i,j}) \in \mathbb{Z}^{m \times n} \ , \quad 0 \leq i < m, 0 \leq j < n$$

or for representing a homomorphism by its matrix. In the first case, the class supports for example computing bases, hermite normal form, smith normal form etc., in the second case it may be used for computing the determinant, the characteristic polynomial, the image, the kernel etc. of the homomorphism.

According to the template introduction (see page 11) the class `bigint_matrix` is derived from `math_matrix< bigint >`. So you can apply all the functions and operators of class `base_matrix< bigint >` and `math_matrix< bigint >` to instances of type `bigint_matrix`.

Description

A variable of type `bigint_matrix` contains the same components as a `math_matrix< bigint >`.

As usually in the following descriptions we use A`.rows` to label the number of rows and A`.columns` to label the number of columns of matrix A.

For some computations modular arithmetic is used, for which we use an external table of prime numbers. By default, this table is taken out of the file `LIDIA_INSTALL_DIR/lib/LiDIA/LIDIA_PRIMES`, where `LIDIA_INSTALL_DIR` denotes the directory, where LiDIA was installed by the installation procedure. For efficiency reasons, we assume at first, that 300 prime numbers are sufficient. If this is not the case, the number of used prime numbers is successively doubled.

If you want to use some other file location for the file containing the prime numbers, you have to set the environment variable `LIDIA_PRIMES_NAME` to point to this location. Depending on your architecture quite some gain in running time may be achieved by using lists of larger prime numbers, e.g. if you have a 64 bit CPU using our list of 26 bit primes (optimized for SPARC V7/V8 architectures) is just a convenient though slow way to get our library to work. If you do computations with very large numbers, you might consider to start with a larger buffer for the prime numbers by setting the environment variable `LIDIA_PRIMES_SIZE` to some larger number, e.g. to 6000.

Constructors/Destructor

If $a \leq 0$ or $b \leq 0$ or $v =$ `NULL` in one of the following constructors the `lidia_error_handler` will be invoked.

`ct bigint_matrix ()`
constructs a 1×1 matrix initialized with zero.

`ct bigint_matrix (lidia_size_t` a`, lidia_size_t` b`)`
constructs an $a \times b$ matrix initialized with zero.

`ct bigint_matrix (lidia_size_t` a`, lidia_size_t` b`, const bigint ** ` v`)`
constructs an $a \times b$ matrix initialized with the values of the 2-dimensional array v. The behaviour of this constructor is undefined if the array v has less than a rows or less than b columns.

`ct bigint_matrix (const base_matrix< bigint > & ` A`)`
constructs a copy of matrix A.

`dt ~bigint_matrix ()`

Arithmetical Operations

In addition to the arithmetical functions and operators of class `math_matrix< T >` you can use the following operators and function:

The operators `bigint_matrix % bigint` and `bigint_matrix %= bigint` reduce componentwise each entry of the given matrix modulo the given `bigint`.

To avoid copying this operator also exist as function:

`void remainder (bigint_matrix & ` A`, const bigint_matrix & ` B`, const bigint & ` e`)`
Let $r = B$`.rows` and $c = B$`.columns`.

$$\begin{pmatrix} a_{0,0} & \dots & a_{0,c-1} \\ \vdots & \ddots & \vdots \\ a_{r-1,0} & \dots & a_{r-1,c-1} \end{pmatrix} \leftarrow \begin{pmatrix} b_{0,0} \bmod e & \dots & b_{0,c'-1} \bmod e \\ \vdots & \ddots & \vdots \\ b_{r-1,0} \bmod e & \dots & b_{r'-1,c'-1} \bmod e \end{pmatrix} .$$

As modulo operation on $\mathbb{Z}$ we use the best remainder function for integers.

Comparisons

Let A be an variable of type `bigint_matrix`.

`bool` A`.is_column_zero (lidia_size_t` i`) const`

`bool is_column_zero (const bigint_matrix & ` A`, lidia_size_t` i`)`
returns `true` if all entries of column i of matrix A are zero, `false` otherwise.

`bool` A`.is_row_zero (lidia_size_t` i`) const`

`bool is_row_zero (const bigint_matrix & ` A`, lidia_size_t` i`)`
returns `true` if all entries of row i of matrix A are zero, `false` otherwise.

`bool` A`.is_matrix_zero () const`

`bool is_matrix_zero (const bigint_matrix & ` A`)`
returns `true` if all entries of matrix A are zero, and `false` otherwise.

Norms and Bounds

`bigint` A`.max () const`

`bigint max (const bigint_matrix &` A`)`
returns the maximum of all components of matrix A.

`void` A`.max (bigint &` x`) const`
$x \leftarrow$ `max(`A`)`.

`bigint` A`.max_abs () const`

`bigint max_abs (const bigint_matrix &` A`)`
returns the maximum of the absolute values of all components of matrix A.

`void` A`.max_abs (bigint &` x`) const`
$x \leftarrow$ `max_abs(`A`)`

`bigint` A`.max_pos (lidia_size_t &` x`, lidia_size_t &` y`) const`

`bigint max_pos (const bigint_matrix &` A`, lidia_size_t &` x`, lidia_size_t &` y`)`
returns the element $a_{x,y}$ which is the maximum of all components of matrix A.

`void` A`.max_pos (bigint &` m`, lidia_size_t &` x`, lidia_size_t &` y`) const`
$m \leftarrow$ `max_pos(`A`,` x`,` y`)`.

`bigint` A`.max_abs_pos (lidia_size_t &` x`, lidia_size_t &` y`) const`

`bigint max_abs_pos (const bigint_matrix &` A`, lidia_size_t &` x`, lidia_size_t &` y`)`
returns the element $a_{x,y}$ which is the maximum of the absolute values of all components of matrix A.

`void` A`.max_abs_pos (bigint &` m`, lidia_size_t &` x`, lidia_size_t &` y`) const`
$m \leftarrow$ `max_abs_pos(`A`,` x`,` y`)`

`bigint` A`.min () const`

`bigint min (const bigint_matrix &` A`)`
returns the minimum of all components of A.

`void` A`.min (bigint &` x`) const`
$x \leftarrow$ `min(`A`)`

`bigint` A`.min_abs () const`

`bigint min_abs (const bigint_matrix &` A`)`
returns the minimum of the absolute values of all components of matrix A.

`void` A`.min_abs (bigint &` x`) const`
$x \leftarrow$ `min_abs(`A`)`

`bigint` A`.min_pos (lidia_size_t &` x`, lidia_size_t &` y`) const`

`bigint min_pos (const bigint_matrix & A, lidia_size_t & x, lidia_size_t & y)`
returns the element $a_{x,y}$ which is the minimum of all components of A.

`void A.min_pos (bigint & m, lidia_size_t & x, lidia_size_t & y) const`
$m \leftarrow$ `min_pos(A, x, y)`

`bigint A.min_abs_pos (lidia_size_t & x, lidia_size_t & y) const`

`bigint min_abs_pos (const bigint_matrix & A, lidia_size_t & x, lidia_size_t & y)`
returns the element $a_{x,y}$ which is the minimum of the absolute values of all components of matrix A.

`void A.min_abs_pos (bigint & x, lidia_size_t & x, lidia_size_t & y) const`
$x \leftarrow$ `min_abs_pos(A, x, y)`

`bigint A.hadamard () const`

`bigint hadamard (const bigint_matrix & A)`
returns the Hadamard bound of A.

`void A.hadamard (bigint & x) const`
$x \leftarrow$ `hadamard(A)`

`void A.row_norm (bigint & x, lidia_size_t i, long j = 2)`
sets $x \leftarrow \sum_{l=0}^{A.\texttt{columns}-1} |a_{i,l}^j|$. If $0 \leq i < A.$`rows` doesn't hold, the `lidia_error_handler` will be invoked.

`bigint A.row_norm (lidia_size_t i, long j = 2)`

`bigint row_norm (const bigint_matrix & A, lidia_size_t i, long j = 2)`
returns $\sum_{l=0}^{A.\texttt{columns}-1} |a_{i,l}^j|$. If not $0 \leq i < A.$`rows` the `lidia_error_handler` will be invoked.

`void A.column_norm (bigint & x, lidia_size_t i, long j = 2)`
sets $x \leftarrow \sum_{l=0}^{A.\texttt{rows}-1} |a_{l,i}^j|$. If not $0 \leq i < A.$`columns` the `lidia_error_handler` will be invoked.

`bigint A.column_norm (lidia_size_t i, long j = 2)`

`bigint column_norm (const bigint_matrix & A, lidia_size_t i, long j = 2)`
returns $\sum_{l=0}^{A.\texttt{rows}-1} |a_l, i^j|$. If not $0 \leq i < A.$`columns` the `lidia_error_handler` will be invoked.

Randomize

`void A.randomize (const bigint & b)`
fills matrix A with random values between 0 and b. The dimensions of matrix A remain unchanged.

`void A.randomize_with_det (const bigint & b, const bigint & DET)`
generates a random matrix A with determinant DET. Internally this function creates two triangular matrices with values between 0 and b. One of these matrices has the determinant 1 and the other determinant DET. So the product of these matrices is the wanted matrix A. If matrix A is not quadratic, the `lidia_error_handler` will be invoked.

Linear Algebra

If the dimensions of the arguments in the following functions don't satisfy the usual restrictions known from linear algebra, the `lidia_error_handler` will be invoked. Some of these are marked by (I). These functions are interface functions to select the algorithm, which is the fastest one on big, dense matrices. Depending on your application, you might want to choose some other algorithm from the list of algorithms described in the next section.

`void` A`.adj (const bigint_matrix &` B`)`

`bigint_matrix adj (const bigint_matrix &` B`)`

stores the adjoint matrix of B to A.

`void` A`.charpoly (base_vector< bigint > &` v`) const`

`void charpoly (base_vector< bigint > &` v`, const bigint_matrix &` A`)`

stores the coefficients of the characteristic polynomial of A to v, where $v[i]$ contains the coefficient of x^i (see [47]). Note that the size of vector v will be adapted if necessary.

`bigint *` A`.charpoly () const`

`bigint * charpoly (const bigint_matrix &` A`)`

returns an array v of coefficients of the characteristic polynomial of A, where $v[i]$ contains the coefficient of x^i (see [47]).

`bigint` A`.det () const`

`bigint det (const bigint_matrix &` A`)`

returns the determinant of matrix A.

`void` A`.det (x) const`

$x \leftarrow$ `det(`A`)`.

`void` A`.hnf ()`

transforms A into hermite normal form. (I)

`bigint_matrix hnf (const bigint_matrix &` A`)`

returns the hermite normal form of A. (I)

`void` A`.hnf (bigint_matrix &` T`)`

transforms A in hermite normal form and sets T to the corresponding transformation matrix. That means: $A \cdot T = \mathrm{HNF}(A)$ (I).

`bigint_matrix hnf (const bigint_matrix &` A`, bigint_matrix &` T`)`

returns the hermite normal form of A and sets T to the corresponding transformation matrix. That means: $A \cdot T = \mathrm{HNF}(A)$ (I).

`void` A`.image (const bigint_matrix &` B`)`

stores a base of the image of the homomorphism defined by B to A. That means, the columns of matrix A form a generating system of this image.

`bigint_matrix image (const bigint_matrix &` B`)`

returns a base of the image of the homomorphism defined by B.

`void` A`.invimage (const bigint_matrix &` B`, const bigint *` v`)`

stores the solutions of the linear equation system $B \cdot x = v$ to matrix A. The last column of matrix A is a solution x of the system and the other columns form a generating system of the kernel of the homomorphism corresponding to matrix B. If there is no solution, matrix A has only one column of zeros. If no memory is allocated for array v, the `lidia_error_handler` will be invoked. The behaviour of this function is undefined if v has less than B`.rows` elements.

`bigint_matrix invimage (const bigint_matrix &` B`, const bigint *` v`)`

returns a `bigint_matrix` A with the following properties: The last column of matrix A is a solution x of the system $B \cdot x = v$ and the other columns form a generating system of the kernel of the homomorphism corresponding to matrix B. If there is no solution, matrix A has only one column of zeros. If no memory is allocated for array v, the `lidia_error_handler` will be invoked. The behaviour of this function is undefined if v has less than B`.rows` elements.

`void` A`.invimage (const bigint_matrix &` B`, const math_vector< bigint > &` v`)`

stores the solutions of the linear equation $B \cdot x = v$ to matrix A. The last column of matrix A is a solution x of the system and the other columns form a generating system of the kernel of the homomorphism corresponding to matrix B. If there is no solution, matrix A has only one column of zeros. If v`.size` $\neq$ B`.rows`, the `lidia_error_handler` will be invoked.

`bigint_matrix invimage (const bigint_matrix &` B`, const math_vector< bigint > &` v`)`

returns a `bigint_matrix` A with the following properties: The last column of matrix A is a solution x of the system $B \cdot x = v$ and the other columns form a generating system of the kernel of the homomorphism corresponding to matrix B. If there is no solution, matrix A has only one column of zeros. If v`.size` $\neq$ B`.rows`, the `lidia_error_handler` will be invoked.

`void` A`.kernel (const bigint_matrix &` B`)`

stores a basis of the kernel of the homomorphism defined by matrix B to A (see [47]), i.e. the columns of the matrix A form a generating system of the kernel (I).

`bigint_matrix kernel (const bigint_matrix &` B`)`

stores a basis of the kernel of the homomorphism defined by matrix B to A (see [47]), i.e. the columns of the matrix A form a generating system of the kernel (I).

`bigint` A`.latticedet () const`

`bigint latticedet (const bigint_matrix &` A`)`

returns a positive multiple of the determinant of the lattice generated by the columns of A (see [24]) (I).

`void` A`.latticedet (bigint &` x`) const`

$x \leftarrow$ `latticedet(`A`)` (I).

`lidia_size_t *` A`.lininc () const`

`lidia_size_t * lininc (const bigint_matrix &` A`)`

returns an array of type `lidia_size_t` with the rank of A (at position zero) followed by the indices of the linearly independent columns of A.

`void` A`.lininc (base_vector< lidia_size_t > &` v`) const`

`void lininc (base_vector< lidia_size_t > &` v`, const bigint_matrix &` A`)`

stores the rank of A (at position zero) in `base_vector` v followed by the indices of the linearly independent columns of A. Note that the size of vector v will be adapted if necessary.

`lidia_size_t *` A`.lininr () const`

`lidia_size_t * lininr (const bigint_matrix &` A`)`

returns an array of type `lidia_size_t` with the rank of A (at position zero) followed by the indices of the linearly independent rows of A (see [47]).

`void` A`.lininr (base_matrix< lidia_size_t > &` v`) const`

`void lininr (base_matrix< lidia_size_t> &` v`, const bigint_matrix &` A`)`

stores the rank of A (at position zero) in `base_vector` v followed by the indices of the linearly independent rows of A. Note that the size of vector v will be adapted if necessary.

`lidia_size_t` A`.rank () const`

`lidia_size_t rank (const bigint_matrix &` A`)`

returns the rank of matrix A (see [47]).

`void` A`.reginvimage (const bigint_matrix &` B`, const bigint_matrix &` C`)`

solves the equation systems $B \cdot X = C$ in the following sense:

The function calculates `bigint` multipliers $g_0, \ldots, g_{m-1}$, where $m = C$`.columns` and a matrix X, such that

$$B \cdot X = C \cdot \begin{pmatrix} g_0 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & g_{m-1} \end{pmatrix}$$

and sets the matrix

$$A \leftarrow \begin{pmatrix} X \\ g_0 \cdots g_{m-1} \end{pmatrix}$$

if B is regular. Otherwise the `lidia_error_handler` will be invoked.

`bigint_matrix reginvimage (const bigint_matrix &` B`, const bigint_matrix &` C`)`

solves the equation systems $B \cdot X = C$ in the following sense:

The function calculates `bigint` multipliers $g_0, \ldots, g_{m-1}$, where $m = C$`.columns` and a matrix X, such that

$$B \cdot X = C \cdot \begin{pmatrix} g_0 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & g_{m-1} \end{pmatrix}$$

and returns the matrix

$$\begin{pmatrix} X \\ g_0 \cdots g_{m-1} \end{pmatrix}$$

if B is regular. Otherwise the `lidia_error_handler` will be invoked.

`void` A`.solve (const bigint_matrix &` B`, const bigint *` v`)`

stores the solutions of the linear equation $B \cdot x = v$ to matrix A. The last column of matrix A is a solution x of the system and the other columns form a generating system of the kernel of the homomorphism corresponding to matrix B. If there is no solution, matrix A has only one column of zeros. If no memory is allocated for the array v, the `lidia_error_handler` will be invoked. The behaviour of this function is undefined if v has less than B`.rows` elements.

`bigint_matrix solve (const bigint_matrix & B, const bigint * v)`

returns a `bigint_matrix` A with the following properties: The last column of matrix A is a solution x of the system $B \cdot x = v$ and the other columns form a generating system of the kernel of the homomorphism corresponding to matrix B. If there is no solution, matrix A has only one column of zeros. If no memory is allocated for the array v, the `lidia_error_handler` will be invoked. The behaviour of this function is undefined if v has less than B`.rows` elements.

`void A.solve (const bigint_matrix & B, const math_vector< bigint > & v)`

stores the solutions of the linear equation $B \cdot x = v$ to matrix A. The last column of matrix A is a solution x of the system and the other columns form a generating system of the kernel of the homomorphism corresponding to matrix B. If there is no solution, matrix A has only one column of zeros. If v`.size` $\neq$ B`.rows`, the `lidia_error_handler` will be invoked.

`bigint_matrix solve (const bigint_matrix & B, const math_vector< bigint > & v)`

returns a `bigint_matrix` A with the following properties: The last column of matrix A is a solution x of the system $B \cdot x = v$ and the other columns form a generating system of the kernel of the homomorphism corresponding to matrix B. If there is no solution, matrix A has only one column of zeros. If v`.size` $\neq$ B`.rows`, the `lidia_error_handler` will be invoked.

`void A.snf ()`

transforms A to Smith Normal Form (I).

`bigint_matrix snf (const bigint_matrix & A)`

returns the Smith Normal Form of A(I).

`void A.snf (bigint_matrix & B, bigint_matrix & C)`

transforms A in Smith Normal Form and sets B, C to the corresponding transformation matrices with $\mathrm{SNF}(A) = B \cdot A \cdot C$ (I).

`bigint_matrix snf (const bigint_matrix & A, bigint_matrix & B, bigint_matrix & C)`

returns the Smith Normal Form of A and sets B, C to the corresponding transformation matrices with $\mathrm{SNF}(A) = B \cdot A \cdot C$ (I).

Special functions

Multiple of the lattice determinant

1. Method:

   `bigint A.latticedet1 () const`

   `bigint latticedet1 (const bigint_matrix & A)`

   returns a positive multiple of the determinant of the lattice generated by the columns of A (see [24]).

   `void A.latticedet1 (bigint & x) const`

   $x \leftarrow$ `latticedet1(`A`)`.

2. Method:

   `bigint A.latticedet2 () const`

`bigint latticedet2 (const bigint_matrix & A)`
returns a positive multiple of the determinant of the lattice generated by the columns of A. First this functions computes two regular submatrices with full rank. Then it computes the determinants DET_1 and DET_2 of these matrices. Finally it returns the greates common divisor of DET_1 and DET_2.

`void A.latticedet2 (bigint & x) const`
$x \leftarrow$ `latticedet2(`A`)`

3. Method:

`bigint A.latticedet3 () const`

`bigint latticedet3 (const bigint_matrix & A)`
returns a positive multiple of the determinant of the lattice generated by the columns of A. In the first step the columns of matrix A are sorted by the euclidian norm. Next it computes a regular submatrix with full rank generated by columns with norm as small as possible. Finally it returns the determinant of this builded matrix.

`void A.latticedet3 (bigint & x) const`
$x \leftarrow$ `latticedet3(`A`)`

Kernel

1. Method:

`void A.kernel1 (const bigint_matrix & B)`
stores a basis of the kernel of the homomorphism generated by matrix B to A.

`bigint_matrix kernel1 (const bigint_matrix & B)`
returns a matrix where the columns form a base of the kernel of the homomorphism generated by matrix B.

The underlying algorithm of these two functions is described in [47].

2. Method:

`void A.kernel2 (const bigint_matrix & B)`
stores a basis of the kernel of the homomorphism with matrix B to A.

`bigint_matrix kernel2 (const bigint_matrix & B)`
returns a matrix where the columns form a base of the kernel of the homomorphism generated by matrix B.

The underlying algorithm of these two functions works as follow:

In a first step the algorithm computes the Hermite Normal Form of matrix B and the corresponding transformation matrix TR by using `B.hnf_havas()` (Description of this algorithm follows later.) Then it counts the number of leading zero columns in the normal form, say l is this number. Finally the leading l columns of the matrix TR build the searched basis.

The differences in the results of these functions are that on the one hand `kernel1` usually creates a basis with smaller entries than `kernel2` but on the other hand the kernel2 functions are faster.

Hermite Normal Form

The following functions transform the given matrix in Hermite Normal Form (HNF). A matrix A in HNF is an upper triangular matrix with:

$$a_{i,A.\texttt{columns}-A.\texttt{rows}+i} > a_{i,j} \geq 0 \quad \forall j > (A.\texttt{columns} - A.\texttt{rows} + i)$$

In these functions we assume that for the given matrices the rank is equal to the number of rows. If this condition is not satisfied, the functions using modular methods for the computation (i.e. the functions, whose names start with `hnfmod`) invoke the `lidia_error_handler`, whereas the other functions terminate at the position, where this is detected, returning a partially HNF reduced matrix.

1. Method:

   `void` A`.hnf_simple ()`
   transforms A to Hermite Normal Form using Gaussian Elimination.

   `bigint_matrix hnf_simple (const bigint_matrix &` A`)`
   returns the Hermite Normal Form of A using Gaussian Elimination.

   `void` A`.hnf_simple (bigint_matrix &` T`)`
   transforms A in Hermite Normal Form and sets T to the corresponding transformation matrix (i.e. $\mathrm{HNF}(A) = A \cdot T$) using a variant of Gaussian Elimination.

   `bigint_matrix hnf_simple (const bigint_matrix &` A`, bigint_matrix &` T`)`
   returns the Hermite Normal Form of A and sets T to the corresponding transformation matrix (i.e. $\mathrm{HNF}(A) = A \cdot T$) using a variant of Gaussian Elimination.

2. Method:

   `void` A`.hnf_havas ()`
   transforms A to Hermite Normal Form using results of [30] [31] and [32].

   `bigint_matrix hnf_havas (const bigint_matrix &` A`)`
   returns the Hermite Normal Form of A using results of [30] [31] and [32].

   `void` A`.hnf_havas (bigint_matrix &` T`)`
   transforms A in Hermite Normal Form and sets T to the corresponding transformation matrix (i.e. $\mathrm{HNF}(A) = A \cdot T$) using results of [30] [31] and [32].

   `bigint_matrix hnf_havas (const bigint_matrix &` A`, bigint_matrix &` T`)`
   returns the Hermite Normal Form of A and sets T to the corresponding transformation matrix (i.e. $\mathrm{HNF}(A) = A \cdot T$) using results of [30] [31] and [32].

3. Method:

   `void` A`.hnf_havas_cont ()`
   transforms A to Hermite Normal Form using results of [30] [31] and [32]. In difference to A`.hnf_havas()`, this algorithm produces always an upper triangular matrix even if the rank of matrix A is not equal to the number of rows of matrix A.

   `bigint_matrix hnf_havas_cont (const bigint_matrix &` A`)`
   returns the Hermite Normal Form of A using results of [30] [31] and [32]. In difference to `hnf_havas(A)`, this algorithm produces always an upper triangular matrix even if the rank of matrix A is not equal to the number of rows of matrix A.

`void` A`.hnf_havas_cont (bigint_matrix &` B`)`
transforms A in Hermite Normal Form and sets T to the corresponding transformation matrix (i.e. HNF$(A) = A \cdot T$) using results of [30] [31] and [32]. In difference to A`.hnf_havas(B)` this algorithm produces always an upper triangular matrix even if the rank of matrix A is not equal to the number of rows of matrix A.

`bigint_matrix hnf_havas_cont (const bigint_matrix &` A`, bigint_matrix &` T`)`
returns the Hermite Normal Form of A and sets T to the corresponding transformation matrix (i.e. HNF$(A) = A \cdot T$) using results of [30] [31] and [32]. In difference to `hnf_havas(A,T)` this algorithm produces always an upper triangular matrix even if the rank of matrix A is not equal to the number of rows of matrix A.

4. Method:

`void` A`.hnfmod_dkt ()`
transforms A to Hermite Normal Form using the algorithm of [23].

`bigint_matrix hnfmod_dkt (const bigint_matrix &` A`)`
returns the Hermite Normal Form of A using the algorithm of [23].

`void` A`.hnfmod_dkt (const bigint &` x`)`
transforms A in Hermite Normal Form using the algorithm of [23] assuming that x is a multiple of the lattice determinant.

`bigint_matrix hnfmod_dkt (const bigint_matrix &` A`, const bigint &` x`)`
returns the Hermite Normal Form of A using the algorithm of [23] assuming that x is a multiple of the lattice determinant.

5. Method:

`void` A`.hnfmod_cohen ()`
transforms A to Hermite Normal Form using the algorithm of [17].

`bigint_matrix hnfmod_cohen (const bigint_matrix &` A`)`
returns the Hermite Normal Form of A using the algorithm of [17].

`void` A`.hnfmod_cohen (const bigint &` x`)`
transforms A in Hermite Normal Form using the algorithm of [17] assuming that x is a multiple of the lattice determinant.

`bigint_matrix hnfmod_cohen (const bigint_matrix &` A`, const bigint &` x`)`
returns the Hermite Normal Form of A using the algorithm of [17] assuming that x is a multiple of the lattice determinant.

6. Method:

`void` A`.hnfmod_mueller (bigint_matrix &` T`)`
transforms A in Hermite Normal Form and sets T to the corresponding transformation matrix with HNF$(A) = A \cdot T$ (see [47]).

`bigint_matrix hnfmod_mueller (const bigint_matrix &` A`, bigint_matrix &` T`)`
returns the Hermite Normal Form of A and sets T to the corresponding transformation matrix with HNF$(A) = A \cdot T$ (see [47]).

Smith Normal Form

The following functions tranform the given matrix into Smith Normal Form (SNF). A matrix in SNF is a diagonal matrix $A = (a_{i,j})$ $(0 \le i \le n,\ 0 \le j \le n)$ such that for all $0 \le i < n$:

$$\begin{aligned} a_{i,i} &\ge 0 \\ a_{n,n} &\ge 0 \\ a_{i,i} &\mid a_{i+1,i+1} \end{aligned}$$

1. Method:

   `void` A`.snf_simple ()`
   transforms A to Smith Normal Form using a variant of Gauß-Jordan elimination.

   `bigint_matrix snf_simple (const bigint_matrix &` A`)`
   returns the Smith Normal Form of A using a variant of Gauß-Jordan elimination.

   `void` A`.snf_simple (bigint_matrix &` C`, bigint_matrix &` B`)`
   transforms A in Smith Normal Form and sets B and C to the corresponding transformation matrices (i.e. $\mathrm{SNF}(A) = C \cdot A \cdot B$) using a variant of Gauß-Jordan elimination.

   `bigint_matrix snf_simple (const bigint_matrix &` A`, bigint_matrix &` C`, bigint_matrix &` B`)`
   returns the Smith Normal Form of A and sets B and C to the corresponding transformation matrices (i.e. $\mathrm{SNF}(A) = C \cdot A \cdot B$) using a variant of Gauß-Jordan elimination.

2. Method:

   `void` A`.snf_hartley ()`
   transforms A to Smith Normal Form using a variant of the algorithm of Hartley and Hawkes.

   `bigint_matrix snf_hartley (const bigint_matrix &` A`)`
   returns the Smith Normal Form of A using a variant of the algorithm of Hartley and Hawkes.

   `void` A`.snf_hartley (bigint_matrix &` C`, bigint_matrix &` B`)`
   transforms A in Smith Normal Form and sets B and C to the corresponding transformation matrices (i.e. $\mathrm{SNF}(A) = C \cdot A \cdot B$) using a variant of the algorithm of Hartley and Hawkes.

   `bigint_matrix snf_hartley (const bigint_matrix &` A`, bigint_matrix &` C`, bigint_matrix &` B`)`
   returns the Smith Normal Form of A and sets B and C to the corresponding transformation matrices (i.e. $\mathrm{SNF}(A) = C \cdot A \cdot B$) using a variant of the algorithm of Hartley and Hawkes.

3. Method:

   `void` A`.snf_havas ()`
   transforms A to Smith Normal Form using results of [30] [31] and [32].

   `bigint_matrix snf_havas (const bigint_matrix &` A`)`
   returns the Smith Normal Form of A using results of [30] [31] and [32].

   `void` A`.snf_havas (bigint_matrix &` C`, bigint_matrix &` B`)`
   transforms A in Smith Normal Form and sets B and C to the corresponding transformation matrices (i.e. $\mathrm{SNF}(A) = C \cdot A \cdot B$) using results of [30] [31] [32].

`bigint_matrix snf_havas (const bigint_matrix & A, bigint_matrix & C, bigint_matrix & B)`
returns the Smith Normal Form of A and sets B and C to the corresponding transformation matrices (i.e. $\mathrm{SNF}(A) = C \cdot A \cdot B$) using results of [30] [31] [32].

The following functions are based on results of [30], [31] and [32]. The extensions `add`, `mult`, `new` specify how column norms and row norms are combined for choosing the pivot strategy and the paramter k is used to select the L_k-norm which is used as described in these papers (default: $k = 1$). If $k < 0$ the behaviour of the functions is undefined.

Member functions transform the given (member) matrix in Smith Normal Form. Functions return the Smith Normal Form of the first argument.

All functions with more then two arguments compute also the corresponding transformation matrices B and C with $\mathrm{SNF}(A) = C \cdot A \cdot B$:

`void A.snf_mult (lidia_size_t k =1)`

`bigint_matrix snf_mult (const bigint_matrix & A, lidia_size_t k = 1)`

`void A.snf_mult (bigint_matrix & C, bigint_matrix & B, lidia_size_t k =1)`

`bigint_matrix snf_mult (const bigint_matrix & A, bigint_matrix & C, bigint_matrix & B, lidia_size_t k = 1)`

`void A.snf_add (lidia_size_t k = 1)`

`bigint_matrix snf_add (const bigint_matrix & A, lidia_size_t k = 1)`

`void A.snf_add (bigint_matrix & C, bigint_matrix & B, lidia_size_t k = 1)`

`bigint_matrix snf_add (const bigint_matrix & A, bigint_matrix & C, bigint_matrix & B, lidia_size_t k = 1)`

`void A.snf_new (lidia_size_t k = 1)`

`bigint_matrix snf_new (const bigint_matrix & A, lidia_size_t k = 1)`

`void A.snf_new (bigint_matrix & C, bigint_matrix & B, lidia_size_t k = 1)`

`bigint_matrix snf_new (const bigint_matrix & A, bigint_matrix & C, bigint_matrix & B, lidia_size_t k = 1)`

The following functions use modular algorithms, which are based on [17] and [23].

1. Method:

`void A.snfmod_dkt ()`
transforms A to Smith Normal Form [23].

`bigint_matrix snfmod_dkt (const bigint_matrix & A)`
returns the Smith Normal Form of A [23].

`void A.snfmod_dkt (const bigint & x)`
transforms A to Smith Normal Form assuming that x is a multiple of the lattice determinant [23].

`bigint_matrix snfmod_dkt (const bigint_matrix & A, const bigint & x)`
returns the Smith Normal Form of A assuming that x is a multiple of the lattice determinant [23].

2. Method:

   `void` A`.snfmod_cohen ()`
   transforms A to Smith Normal Form [17].

   `bigint_matrix snfmod_cohen (const bigint_matrix &` A`)`
   returns the Smith Normal Form of A [17].

   `void` A`.snfmod_cohen (const bigint &` x`)`
   transforms A to Smith Normal Form assuming that x is a multiple of the lattice determinant [17].

   `bigint_matrix snfmod_cohen (const bigint_matrix &` A`, const bigint &` x`)`
   returns the Smith Normal Form of A assuming that x is a multiple of the lattice determinant [17].

See also

`base_matrix`, `math_matrix`, `base_vector`, `math_vector`

Notes

By inspection of the file `LiDIA/bigint_matrix.h` you will discover additional routines for computing normal forms, the kernel, ... These versions are not yet completely tested, but should be faster than the well tested functions, that are documented. In the next release these functions should be included. So, if you feel a bit adventurous you might try them and help us to debug them.

Examples

```
#include <LiDIA/bigint_matrix.h>

int main()
{
    lidia_size_t c = 7, d = 5;

    bigint_matrix A(c,c);
    A.randomize((bigint)10);

    bigint *tmp = A.row(5);
    A.sto_row(tmp,7,3);

    bigint_matrix B, C;

    B = A;
    A.hnf_simple();
    cout << "hnf_simple" << A << flush;

    B = A;
    A.hnf_havas();
    cout << "hnf_havas" << A << flush;

    B = A;
    A.hnf_havas_cont();
    cout << "hnf_havas_cont" << A << flush;
```

```
        B = A;
        A.hnf_havas_cont(C);
        cout << A << B*C << flush;

        return 0;
    }
```

For further examples please refer to `LiDIA/src/simple_classes/bigint_matrix_appl.cc`.

Author

Stefan Neis, Patrick Theobald

bigmod_matrix

Name

`bigmod_matrix` ..... Linear algebra over $\mathbb{Z}/m\mathbb{Z}$

Abstract

`bigmod_matrix` is a class for doing linear algebra over the residue class ring $\mathbb{Z}/m\mathbb{Z}$, where m may be any non-negative integer (for $m = 0$ we do the computations over $\mathbb{Z}$). This class may either be used for representing a modul, which is generated by the columns of the matrix

$$A = (a_{i,j}) \in (\mathbb{Z}/m\mathbb{Z})^{k \times l} \quad , \quad 0 \leq i < k, 0 \leq j < l$$

or for representing a homomorphism by its matrix. In the first case, the class supports for example computing "standard generating systems", in the second case it may be used for computing the image or the kernel of the homomorphism.

This version is an experimental version which mainly supplies routines needed by `modules` and `ideals` in algebraic number fields. However, we believe, our new algorithms might be of interest also to other people. Note however, that the interface of this class is not yet very stable and will change in future releases. Especially it is not yet very consistent with the template classes for matrices.

Description

A variable of type `bigmod_matrix` is derived from `bigint_matrix` and contains an additional `bigint` m which describes the modulus of the matrix. Due to this derivation step, you may call all functions written for `bigint_matrices` also on `bigmod_matrices`. Note however, that the functions inherited from `bigint_matrix` will not reduce their results modulo m, but the functions written for `bigmod_matrix` rely on a reduced input! However quite some functions of `bigint_matrix` can be used safely anyway (for example `transpose` or `diag` with reasonable arguments). However, be very careful, which functions you do use. Especially the `sto_column_vector` or `sto_row_vector` functions should be avoided.

As usually in the following descriptions we use A`.rows` to label the number of rows and A`.columns` to label the number of columns of matrix A.

Constructors/Destructor

If $a \leq 0$ or $b \leq 0$ in one of the following constructors the `lidia_error_handler` will be invoked.

`ct bigmod_matrix ()`
constructs a 1×1 matrix over $\mathbb{Z}$ initialized with zero.

`ct bigmod_matrix (lidia_size_t` a`, lidia_size_t` b`)`
constructs an $a \times b$ matrix over $\mathbb{Z}$ initialized with zero.

ct bigmod_matrix (lidia_size_t a, lidia_size_t b, const bigint & m)
constructs an $a \times b$ matrix over $\mathbb{Z}/m\mathbb{Z}$ initialized with zero.

ct bigmod_matrix (const bigmod_matrix & A)
constructs a copy of matrix A.

ct bigmod_matrix (const base_matrix< bigint > & A, const bigint & m = 0)
constructs a copy of matrix A where every entry is reduced modulo m.

dt ~bigmod_matrix ()

Assignments

Let A be of type bigmod_matrix. The operator = is overloaded. For consistency reasons, the following functions are also implemented:

void A.assign (const bigmod_matrix & B)

void assign (bigmod_matrix & A, const bigmod_matrix & B)
sets the dimensions and the modulus of matrix A to the dimensions and the modulus of matrix B and copies each entry of B to the corresponding position in A.

void A.assign (const base_matrix< bigint > & B, const bigint & m = 0)

void assign (bigmod_matrix & A, const base_matrix< bigint > & B, const bigint & m = 0)
sets the dimensions of matrix A to the dimensions of matrix B, sets the modulus of matrix A to m, and reduces each entry of B modulo m and stores it to the corresponding position in A.

Access Methods

Let A be of type bigmod_matrix. Note that the numbering of columns and rows starts with zero. Also note that for reading access the functions of class bigint_matrix and thereby of class base_matrix< bigint > can safely be used. However for storing elements, we would like to reduce the elements by the modulus of the matrix. For this purpose the following functions (and only these!!) are supported.

void A.sto_row (const bigint * v, lidia_size_t j, lidia_size_t i)
stores j entries of array v in row i of matrix A starting at column zero, reducing them by the modulus of A. If $j > A$.columns or $j \leq 0$ or $i \geq A$.rows or $i < 0$, the lidia_error_handler will be invoked.

void A.sto_column (const bigint * v, lidia_size_t j, lidia_size_t i)
stores j entries of array v in column i of matrix A starting at row zero, reducing them by the modulus of A. If $j > A$.rows or $j \leq 0$ or $i \geq A$.columns or $i < 0$, the lidia_error_handler will be invoked.

void A.sto (lidia_size_t i, lidia_size_t j, const bigint & x)
stores x as $a_{i,j}$ reducing it modulo the modulus of A. If $i \geq A$.rows or $i < 0$ or $j \geq A$.columns or $j < 0$, the lidia_error_handler will be invoked.

const bigint & A.get_modulus () const

const bigint & get_modulus (const bigmod_matrix & A)
returns the modulus of the matrix A.

`void` A`.set_modulus (const bigint &` m`)`

`void set_modulus (bigmod_matrix &` A`, const bigint &` m`)`
sets the modulus of the matrix A to m. Note that the entries of the matrix are not changed!

Lifting and Reducing

Let A be of type `bigmod_matrix`.

`void` A`.reduce (const bigint &` m`)`

`void reduce (bigmod_matrix &` A`, const bigint &` m`)`
This interprets A as generating a module over $\mathbb{Z}/m\mathbb{Z}$ (and accordingly may modify A). Note that m must divide the previously set modulus, since otherwise this is not considered to be a valid operation and the `lidia_error_handler` will be invoked.

`void` A`.lift (const bigint &` m`)`

`void lift (bigmod_matrix &` A`, const bigint &` m`)`
This interprets A as generating a module over $\mathbb{Z}/m\mathbb{Z}$ (and accordingly may modify A). Note that m must be a multiple of the previously set modulus, since otherwise this is not considered to be a valid operation and the `lidia_error_handler` will be invoked.

Split Functions

Let A be of type `bigmod_matrix`. The following split functions allow area overlap of the submatrices as described in the documentation of the class `base_matrix< T >`.

If the given conditions in the descriptions of the following functions are not satisfied, the `lidia_error_handler` will be invoked.

`void` A`.split (bigmod_matrix &` B`, bigmod_matrix &` C`, bigmod_matrix &` D`,`
`bigmod_matrix &` E`) const`
takes matrix A apart into the submatrices B, C, D, and E, with

$$A = \begin{pmatrix} B & C \\ D & E \end{pmatrix} ,$$

where the following conditions have to be satisfied:

- B`.rows`, C`.rows`, D`.rows`, E`.rows` $\leq$ A`.rows`
- B`.columns`, C`.columns`, D`.columns`, E`.columns` $\leq$ A`.columns`

The entries that are stored into B, C, D, and E, respectively, are always reduced modulo the corresponding modulus.

`void` A`.split_h (bigmod_matrix &` B`, bigmod_matrix &` C`) const`

takes matrix A apart into the submatrices B and C, with $A = \begin{pmatrix} B & C \end{pmatrix}$ where the following conditions have to be satisfied:

- B`.rows`, C`.rows` $\leq A$`.rows`
- B`.columns`, C`.columns` $\leq A$`.columns`

The entries that are stored into B and C, respectively, are always reduced modulo the corresponding modulus.

`void` A`.split_v (bigmod_matrix &` B`, bigmod_matrix &` C`) const`

takes matrix A apart into the submatrices B and C, with $A = \begin{pmatrix} B \\ C \end{pmatrix}$, where the following conditions have to be satisfied:

- B`.rows`, C`.rows` $\leq A$`.rows`
- B`.columns`, C`.columns` $\leq A$`.columns`

The entries that are stored into B and C, respectively, are always reduced modulo the corresponding modulus.

Compose Functions

Let A be of type `bigmod_matrix`. The following compose functions allow area overlap of the submatrices as described in the documentation of class `base_matrix< T >`. The matrices B, C, D and E remain unchanged.

If the given conditions in the descriptions of the following functions are not satisfied, the `lidia_error_handler` will be invoked.

`void` A`.compose (const bigmod_matrix &` B`, const bigmod_matrix &` C`,`
`const bigmod_matrix &` D`, const bigmod_matrix &` E`)`

composes the matrices B, C, D and E to the matrix A, with $A = \begin{pmatrix} B & C \\ D & E \end{pmatrix}$ where the following conditions have to be satisfied:

- B`.rows`, C`.rows`, D`.rows`, E`.rows` $\leq A$`.rows`
- B`.columns`, C`.columns`, D`.columns`, E`.columns` $\leq A$`.columns`

The entries that are stored into A are always reduced modulo the modulus of A.

`void` A`.compose_h (const bigmod_matrix &` B`, const bigmod_matrix &` C`)`

composes the matrices B and C to the matrix A, with $A = \begin{pmatrix} B & C \end{pmatrix}$ where the following conditions have to be satisfied:

- B`.rows`, C`.rows` $\leq A$`.rows`
- B`.columns`, C`.columns` $\leq A$`.columns`

The entries that are stored into A are always reduced modulo the modulus of A.

`void` A`.compose_v (const bigmod_matrix &` B`, const bigmod_matrix &` C`)`

composes the matrices B and C to the matrix A, with $A = \begin{pmatrix} B \\ C \end{pmatrix}$ where the following conditions have to be satisfied:

- B`.rows`, C`.rows` $\leq A$`.rows`
- B`.columns`, C`.columns` $\leq A$`.columns`

The entries that are stored into A are always reduced modulo the modulus of A.

Swap Functions

Obviously, the function `swap_rows` and `swap_columns` of class `bigint_matrix` (or more precisely of `base_matrix< T >`) can be safely used. Moreover there is the following function.

`void swap (bigmod_matrix &` A`, bigmod_matrix &` B`)`

swaps A and B.

Arithmetical Operations

The class `bigmod_matrix` supports the following operators, which always involve reduction by the modulus of the input matrices.

unary		*op*	`bigmod_matrix`	*op* $\in \{$`-`$\}$
binary	`bigmod_matrix`	*op*	`bigmod_matrix`	*op* $\in \{$`+, -, *`$\}$
binary with assignment	`bigmod_matrix`	*op*	`bigmod_matrix`	*op* $\in \{$`+=, -=, *=`$\}$
binary	`bigmod_matrix`	*op*	`bigint`	*op* $\in \{$`+, -, *`$\}$
binary with assignment	`bigmod_matrix`	*op*	`bigint`	*op* $\in \{$`+=, -=, *=`$\}$
binary	`bigmod_matrix`	*op*	`(bigint *)`	*op* $\in \{$`*`$\}$

Here the operators operating on two matrices and the unary minus implement the usual operations known from linear algebra. Especially the moduli of the input matrices have to be the same, otherwise the `lidia_error_handler` will be invoked; if the dimensions of the operands do not satisfy the usual restrictions, the `lidia_error_handler` will also be invoked. Note that the dimensions of the resulting matrix are adapted to the right dimensions known from linear algebra if necessary.

The operator `bigmod_matrix * (bigint *)` realizes the matrix-vector-multiplication. If the number of elements does not satisfy the usual restrictions known from linear algebra the behaviour is undefined.

The operators *op* which have a single element of type `bigint` in their list of arguments perform the operation *op* componentwise and reduce the result by the modulus of the matrix. E.g. `bigmod_matrix`·`bigint` multiplies each entry of the matrix by the given scalar, and reduces by the modulus of the matrix.

To avoid copying, all operators also exist as functions and of course the restrictions for the dimensions are still valid:

`void add (bigmod_matrix &` C`, const bigmod_matrix &` A`, const bigmod_matrix &` B`)`

$C \leftarrow A + B$.

`void add (bigmod_matrix & `C`, const bigmod_matrix & `B`, const bigint & `e`)`
Let $r = B$`.rows` and $c = B$`.columns`.

$$\begin{pmatrix} c_{0,0} & \dots & c_{0,c-1} \\ \vdots & \ddots & \vdots \\ c_{r-1,0} & \dots & c_{r-1,c-1} \end{pmatrix} \leftarrow \begin{pmatrix} b_{0,0}+e & \dots & b_{0,c-1}+e \\ \vdots & \ddots & \vdots \\ b_{r-1,0}+e & \dots & b_{r-1,c-1}+e \end{pmatrix} .$$

`void subtract (bigmod_matrix & `C`, const bigmod_matrix & `A`, const bigmod_matrix & `B`)`
$C \leftarrow A - B$.

`void subtract (bigmod_matrix & `C`, const bigmod_matrix & `B`, const bigint & `e`)`
Let $r = B$`.rows` and $c = B$`.columns`.

$$\begin{pmatrix} c_{0,0} & \dots & c_{0,c-1} \\ \vdots & \ddots & \vdots \\ c_{r-1,0} & \dots & c_{r-1,c-1} \end{pmatrix} \leftarrow \begin{pmatrix} b_{0,0}-e & \dots & b_{0,c-1}-e \\ \vdots & \ddots & \vdots \\ b_{r-1,0}-e & \dots & b_{r-1,c-1}-e \end{pmatrix} .$$

`void multiply (bigmod_matrix & `C`, const bigmod_matrix & `A`, const bigmod_matrix & `B`)`
$C \leftarrow A \cdot B$.

`void multiply (bigmod_matrix& `C`, const bigmod_matrix & `B`, const bigint & `e`)`
Let $r = B$`.rows` and $c = B$`.columns`.

$$\begin{pmatrix} c_{0,0} & \dots & c_{0,c-1} \\ \vdots & \ddots & \vdots \\ c_{r-1,0} & \dots & c_{r-1,c-1} \end{pmatrix} \leftarrow \begin{pmatrix} b_{0,0} \cdot e & \dots & b_{0,c-1} \cdot e \\ \vdots & \ddots & \vdots \\ b_{r-1,0} \cdot e & \dots & b_{r-1,c-1} \cdot e \end{pmatrix} .$$

Note that in this special case reduction is done by the modulus of C. This is helpful in some contexts while disturbing in others!

`void multiply (bigint * & `v`, const bigmod_matrix & `A`, const bigint * `w`)`
assigns the result of the muliplication $A \cdot w$ to v (matrix-vector-multiplication). If a suitable amount of memory has not been allocated for the arrays w and v, the behaviour of this function is undefined.

`void negate (bigmod_matrix& `B`, const bigmod_matrix & `A`)`
$B \leftarrow -A$.

Comparisons

The binary operators `==` and `!=` are overloaded and can be used for comparison by components. Let A be an instance of type `bigmod_matrix`.

`bool `A`.equal (const bigmod_matrix & `B`) const`
returns `true` if A and B are identical, `false` otherwise.

`bool equal (const bigmod_matrix & `A`, const bigmod_matrix & `B`)`
returns `true` if A and B are identical, `false` otherwise.

`bool` A`.unequal (const bigmod_matrix &` B`) const`
returns `false` if A and B are identical, `true` otherwise.

`bool unequal (const bigmod_matrix &` A`, const bigmod_matrix &` B`)`
returns `false` if A and B are identical, `true` otherwise.

Linear Algebra

If the dimensions of the arguments in the following functions don't satisfy the usual restrictions known from linear algebra, the `lidia_error_handler` will be invoked. Linear algebra over $\mathbb{Z}/m\mathbb{Z}$ can be done following two philosophies: Either you are content to get a partial result and a factor of m (for example for reducing the partial result and using the chinese remainder theorem. This however is not very good for "small" m) or you want a result over $\mathbb{Z}/m\mathbb{Z}$ no matter what. Since this can also be done efficiently, we offer both methods. So far, however, not both versions of all functions are supported.

`void` A`.adj (const bigmod_matrix &` B`, bigint &` f`)`

`bigmod_matrix adj (const bigmod_matrix &` B`, bigint &` f`)`
stores either the adjoint matrix of B to A or a non-trivial factor of m to f.

`void` A`.inv (const bigmod_matrix &` B`, bigint &` f`)`

`bigmod_matrix inv (const bigmod_matrix &` B`, bigint &` f`)`
stores either the inverse matrix of B to A or a non-trivial factor of m to f.

`bigint *` A`.charpoly (bigint &` f`) const`

`bigint * charpoly (const bigmod_matrix &` A`, bigint &` f`)`
returns either an array v of coefficients of the characteristic polynomial of A, where $v[i]$ contains the coefficient of x^i or sets f to some non-trivial factor of m.

`bigint` A`.det (bigint &` f`) const`

`bigint det (const bigmod_matrix &` A`, bigint &` f`)`
either returns the determinant of matrix A or sets f to some non-trivial factor of m.

`void` A`.det (bigint &` x`, bigint &` f`) const`
either $x \leftarrow$ `det`(A) or f is set to some non-trivial factor of m.

`void` A`.image (const bigmod_matrix &` B`, bigint &` f`)`
either stores a generating system of the image of the homomorphism defined by B to A or sets f to some non-trivial factor of m.

`bigmod_matrix image (const bigmod_matrix &` B`, bigint &` f`)`
either returns a generating system of the image of the homomorphism defined by B or sets f to some non-trivial factor of m.

`void` A`.image (const bigmod_matrix &` B`)`
stores a generating system of the image of the homomorphism defined by B to A.

`bigmod_matrix image (const bigmod_matrix & `B`)`
returns a generating system of the image of the homomorphism defined by B.

`void `A`.unique_image (const bigmod_matrix & `B`)`
stores a uniquely determined generating system of the image of the homomorphism defined by B to A .

`bigmod_matrix unique_image (const bigmod_matrix & `B`)`
returns a uniquely determined generating system of the image of the homomorphism defined by B.

`void `A`.kernel (const bigmod_matrix & `B`, bigint & `f`)`
either stores a generating system of the kernel of the homomorphism defined by matrix B to A or sets f to some non-trivial factor of m.

`bigmod_matrix kernel (const bigmod_matrix & `B`, bigint & `f`)`
either returns a generating system of the kernel of the homomorphism defined by matrix B to A or sets f to some non-trivial factor of m.

`void `A`.kernel (const bigint_matrix & `B`)`
stores a generating system of the kernel of the homomorphism defined by matrix B to A.

`bigint_matrix kernel (const bigint_matrix & `B`)`
returns a generating system of the kernel of the homomorphism defined by matrix B to A.

Input/Output

The `istream` operator `>>` and the `ostream` operator `<<` are overloaded. The `istream` operator `>>` reads matrices in the following format from istream:

$$r \sqcup c \sqcup m \sqcup a_{0,0} \sqcup a_{0,1} \sqcup \cdots \sqcup a_{0,c-1} \sqcup a_{1,0} \sqcup \cdots \sqcup a_{1,c-1} \sqcup \cdots \sqcup a_{r-1,0} \sqcup \cdots \sqcup a_{r-1,c-1}$$

where $r = A$`.rows`, $c = A$`.columns`, and $m = A.modulus$.

The `ostream` operator `<<` outputs the matrix in a beautified format which looks as follows:

$$\begin{array}{cccccc} (& a_{0,0} & a_{0,1} & \dots & a_{0,c-1} &) \\ (& a_{1,0} & \dots & \dots & a_{1,c-1} &) \\ (& \vdots & \vdots & \vdots & \vdots &) \\ (& a_{r-1,0} & \dots & \dots & a_{r-1,c-1} &) \end{array} \mod m \ .$$

For supporting more input and output formats we suggest to read and write `bigint_matrices` and the modulus separately, thus enabling all the formats supported by `bigint_matrix`.

Warnings

The behaviour of the function

`void multiply (bigmod_matrix& `C`, const bigmod_matrix & `B`, const bigint & `e`)`

may be surprising sometimes, since the modulus of C is *not* set to the modulus of B and reduction is done by the modulus of C, *not* by the modulus of B.

See also

`base_matrix`, `math_matrix`, `bigint_matrix`

Notes

1. As described in the introduction this version is very preliminary.
2. As usually in C++ the numbering of columns and rows starts with zero.

Author

Stefan Neis, Patrick Theobald

The LiDIA LT package

The LiDIA LT package contains the classes which deal with lattices over the integers and over the reals. It requires the LiDIA base package, the LiDIA FF package, and the LiDIA LA package.

Chapter 15

Lattices

bigint_lattice

Name

`bigint_lattice` ..... lattice algorithms

Abstract

The class `bigint_lattice` is derived from the class `bigint_matrix` and therefore inherits all their functions. In addition the class `bigint_lattice` provides algorithms for shortest vector computations, lattice reduction of lattice bases or linearly independent Gram matrices as well as computations of lattice bases and relations from generating systems or linearly dependent Gram matrices. A lattice L is given by $A = (a_0, \ldots, a_{k-1}) \in M^{n \times k}$ $(n, k \in \mathbb{N})$ where M consists of elements of type `bigint` and A is a linearly (in)dependent Gram matrix, a generating system or a lattice basis of L.

Description

A variable of type `bigint_lattice` cosists of the same components as `bigint_matrix`.

Let A be of class `bigint_lattice`. In the present version of LiDIA the following algorithms are implemented:

1. Generating systems:
   - The MLLL algorithm [51] which computes for a $k-1$-dimensional lattice L represented by a generating system $A = (a_0, \ldots, a_{k-1}) \in M^{k-1 \times k}$ a reduced basis $B = (b_0, \ldots, b_{k-2})$ as well as a relation i.e., integers $x_0, \ldots, x_{k-1} \in \mathbb{Z}$ satisfying $\sum_{i=0}^{k-1} x_i a_i = 0$.
   - The Buchmann-Kessler algorithm [13] which computes a reduced basis for a lattice represented by a generating system A.
   - The Schnorr-Euchner algorithm [55] (modified for generating systems) as well as variations [64] which compute an LLL reduced basis of the lattice L represented by the generating system A.
2. Lattice bases:
   - The original Schnorr-Euchner lattice reduction algorithm [55] as well as variations [64].
   - The Benne de Weger variation [20] of the original LLL algorithm [39] which avoids rational arithmetic.
   - The Fincke-Pohst algorithm for computing shortest vectors of a lattice $L = \mathbb{Z}a_0 \oplus \cdots \oplus \mathbb{Z}a_{k-1}$ [25].
   - The algorithm of Babai [2] for computing a close lattice vector to a given vector.
3. Gram matrices:
   - The original Schnorr-Euchner lattice reduction algorithm [55] as well as variations [64].

In the following we will explain the constructors, destructors and basic functions of the classes `bigint_lattice`. Lattices can be stored column by column or row by row. Independent of the the storage mode either the columns or the rows of the lattice can be reduced. The following description is done with respect to column-oriented lattices i.e the reduction is done columnwise. For row-oriented lattices one has to change the representation. The corresponding computations will then be adjusted automatically.

Constructors/Destructor

For the following constructors the variable A is assumed to represent a lattice. If in the following constructors $n \leq 1$, $k \leq 1$, $n < k$, $A =$ `NULL` or the dimension of A does not correspond to the input parameters which declare the dimension, the `lidia_error_handler` will be invoked.

`ct bigint_lattice ()`
constructs a 1×1 `bigint_lattice`.

`ct bigint_lattice (lidia_size_t` n`, lidia_size_t` k`)`
constructs a `bigint_lattice` of k vectors embedded in an n-dimensional vector space initialized with zero.

`ct bigint_lattice (const bigint_matrix &` A`)`
constructs a copy of A.

`ct bigint_lattice (lidia_size_t` n`, lidia_size_t` k`, bigint** &` A`)`
constructs an $n \times k$-`bigint_lattice` initialized with the values of the 2-dimensional array A where $*A$ is a row vector.

`dt ~bigint_lattice ()`

Initialization

Let A be of type `bigint_lattice`. If none of the following functions is called explicitly by the user, A will be considered as column-oriented generating system representing the lattice L. Changes of the lattice by the user will automatically result in the deletion of the basis and Gram flag.

`void` A`.set_basis_flag ()`
sets the basis flag without checking whether A satisfies the basis property or not. For future computations A is now considered as basis representing the lattice L. Setting the flag without knowing whether A is a basis or not may result in abortions of further computations or wrong results. By default the representation is set as generating system.

`void` A`.delete_basis_flag ()`
sets the basis flag to `false`. For future computations A is now considered as generating system representing the lattice L.

`void` A`.set_gram_flag ()`
sets the Gram flag without checking whether A satisfies the properties of a Gram matrix or not. For future computations A is now considered as Gram matrix representing the lattice L. Setting the flag without knowing whether A is a Gram matrix or not may result in abortions of further computations or wrong results. By default the Gram flag is not set.

`void` A`.delete_gram_flag ()`
sets the Gram flag to `false`.

`void` A`.set_red_orientation_columns ()`
sets the orientation such that the reduction will be done column-oriented. By default the orientation is taken as column-oriented.

`void` A`.set_red_orientation_rows ()`

sets the orientation such that the reduction will be done row-oriented. By default the orientation is taken as column-oriented.

Basic Methods and Functions

Let A be of type `bigint_lattice`.

`bool` A`.check_basis ()`

returns `true` and sets the basis flag if the columns or rows of A are linearly independent according to the orientation of A, `false` otherwise.

`bool` A`.check_gram ()`

returns `true` and sets the Gram flag if A is symmetric and positive semi-definete and therefore satisfies the properties of a Gram matrix, `false` otherwise.

`bool` A`.get_basis_flag ()`

returns `true` if the basis flag is set, `false` otherwise.

`bool` A`.get_gram_flag ()`

returns `true` if the Gram flag is set, `false` otherwise.

`bool` A`.get_red_orientation ()`

returns `true` for column-oriented lattices, `false` otherwise.

`bool` A`.lll_check (double` y`)`

returns `true` if the basis is reduced for the reduction parameter y, `false` otherwise. The `lidia_error_handler` will be invoked if the function is applied to generating systems or the Gram flag is set.

`bool` A`.lll_check (sdigit` p`, sdigit` q`)`

returns `true` if the basis is reduced for the reduction parameter $\frac{p}{q}$, `false` otherwise. The `lidia_error_handler` will be invoked if the function is applied to generating systems or the Gram flag is set.

`double` A`.lll_check_search ()`

if A is LLL reduced, this function will return the largest $y \in]\frac{1}{2}, 1]$ for which the basis A is still LLL reduced, see [39, 55]. If the basis A is not LLL reduced, 0.0 will be returned. The `lidia_error_handler` will be invoked if the function is applied to generating systems or the Gram flag is set.

`void` A`.lll_check_search (sdigit &` p`, sdigit &` q`)`

if A is LLL reduced, this function will return the largest $\frac{p}{q} \in]\frac{1}{2}, 1]$ for which the basis A is still LLL reduced, see [39, 55]. If the basis A is not LLL reduced, 0 will be returned. The `lidia_error_handler` will be invoked if the function is applied to generating systems or the Gram flag is set.

`void` A`.randomize_vectors ()`

permutes the vectors of the given lattice randomly.

`void` *A*`.sort_vectors (bin_cmp_func)`

sorts the vectors of the given lattice such that their lengths (length in the meaning of the `bin_cmp_func`, which is defined as `int (*bin_cmp_func)(const bigint*, const bigint*, long))` are in descending order. The function `bin_cmp_func` has to return -1 (1) if the first vector is shorter (longer) as the second one and 0 if they have the same length.

`void` *A*`.sort_big_vectors ()`

sorts the vectors of the given lattice such that their lengths (by means of the Euclidian norm) are in descending order.

`void` *A*`.sort_small_vectors ()`

sorts the vectors of the given lattice such that their lengths (by means of the Euclidian norm) are in ascending order.

Lattice Reduction

The following functions can be used for performing LLL reductions. These functions are interface functions and point to the fastest algorithms described in the section 'Special Functions' for solving arbitrarily chosen lattice instances. Depending on the application of the lattice problem you might want to use a different algorithm mentioned in that section though. Please note that those algorithms might change in the future and we therefore suggest to use the interface functions. At the moment the interface functions point to the `lll_schnorr_euchner_orig` implementation.

The union `lattice_info` is defined as:

```
typedef union {
          struct {
            lidia_size_t rank;
            double y;
            sdigit y_nom;
            sdigit y_denom;
            sdigit reduction_steps;
            sdigit correction_steps;
            sdigit swaps;
          } lll;
        } lattice_info;
```

So far `lattice_info` provides only information about the performed LLL reduction (e.g. number of swaps, reduction or correction steps) but in the future the union will be extended for providing information about the performance of other lattice algorithms.

In order to use a self-defined scalar product one has to define four functions corresponding to the following typedefs:

```
typedef void (*scal_dbl)(double&, double*, double*, lidia_size_t);
typedef void (*scal_xdbl)(xdouble&, xdouble*, xdouble*, lidia_size_t);
typedef void (*scal_bin)(bigint&, bigint*, bigint*, lidia_size_t);
typedef void (*scal_bfl)(bigfloat&, bigfloat*, bigfloat*, lidia_size_t);
```

The first parameter is the return value, the second and third parameter stand for the input vectors and the last parameter indicates the length of the vectors. With the addresses of the functions one defines the actual scalar product as:

```
typedef struct {
          scal_dbl dbl;
```

```
    scal_xdbl xdbl;
    scal_bin bin;
    scal_bfl bfl;
  } user_SP;
```

Please note that it is allowed to change the precision for `bigfloat`s within `void (*scal_bfl)(bigfloat&, bigfloat*, bigfloat*, lidia_size_t);` but one has to guarantee that after the computation of the scalar product the precision is set back to its original value (before entering the function).

Let A be of type `bigint_lattice`.

`void` A`.lll (double` y`, sdigit` *factor* `= 1)`

`bigint_lattice lll (const bigint_lattice &` A`, double` y`, sdigit` *factor* `= 1)`

`void` A`.lll (double` y`, lattice_info &` li`, user_SP, sdigit` *factor* `= 1)`

`bigint_lattice lll (const bigint_lattice &` A`, double` y`, lattice_info &` li`, user_SP, sdigit` *factor* `= 1)`

`void` A`.lll (double` y`, lattice_info &` li`, sdigit` *factor* `= 1)`

`bigint_lattice lll (const bigint_lattice &` A`, double` y`, lattice_info &` li`, sdigit` *factor* `= 1)`

`void` A`.lll (double` y`, user_SP, sdigit` *factor* `= 1)`

`bigint_lattice lll (const bigint_lattice &` A`, double` y`, user_SP, sdigit` *factor* `= 1)`

if A is not a Gram matrix, these functions compute $A = (\underbrace{a_0, \ldots, a_r - 1}_{=B}, \underbrace{0, \ldots, 0}_{k-r})$, where B is an LLL reduced basis of L with rank r. For linearly (in)dependent Gram matrices the computations are done according to [17]. $y \in]\frac{1}{2}, 1]$ is the reduction parameter. With *factor* the precision for the approximations of computations (depending on the algorithm) will be determined as (*factor*$\cdot$ `double` precision). For *factor* $= 1$ (*factor* $= 2$) the approximations are done by using `double`s (`xdouble`s), otherwise `bigfloat`s with appropriate precision will be used.

`void` A`.lll (math_matrix< bigint > &` T`, double` y`, sdigit` *factor* `= 1)`

`bigint_lattice lll (const bigint_lattice &` A`, math_matrix< bigint > &` T`, double` y`, sdigit` *factor* `= 1)`

`void` A`.lll (math_matrix< bigint > &` T`, double` y`, lattice_info &` li`, user_SP, sdigit` *factor* `= 1)`

`bigint_lattice lll (const bigint_lattice &` A`, math_matrix< bigint > &` T`, double` y`, lattice_info &` li`, user_SP, sdigit` *factor* `= 1)`

`void` A`.lll (math_matrix< bigint > &` T`, double` y`, lattice_info &` li`, sdigit` *factor* `= 1)`

`bigint_lattice lll (const bigint_lattice &` A`, math_matrix< bigint > &` T`, double` y`, lattice_info &` li`, sdigit` *factor* `= 1)`

`void` A`.lll (math_matrix< bigint > &` T`, double` y`, user_SP, sdigit` *factor* `= 1)`

`bigint_lattice lll (const bigint_lattice & A, math_matrix< bigint > & T, double y, user_SP, sdigit factor = 1)`

if A is not a Gram matrix, these functions compute the transformation matrix T and $A = (\underbrace{b_0, \ldots, b_{r-1}}_{=B}, \underbrace{0, \ldots, 0}_{k-r})$ where B is an LLL reduced basis of L with rank r. The last $k - r$ columns of T are relations for the original generating system. For linearly (in)dependent Gram matrices the computations are done according to [17]. $y \in]\frac{1}{2}, 1]$ is the reduction parameter. With *factor* the precision for the approximations of computations (depending on the algorithm) will be determined as (*factor*· `double` precision). For *factor* = 1 (*factor* = 2) the approximations are done by using `double`s (`xdouble`s), otherwise `bigfloat`s with appropriate precision will be used.

Shortest Vector Computation

Let A be of type `bigint_lattice`.

`lidia_size_t A.shortest_vector (math_matrix< bigint > & V, lidia_size_t p = 20)`

`lidia_size_t shortest_vector (const bigint_lattice & A, math_matrix< bigint > & V, lidia_size_t p = 20)`

returns the squared length of the shortest vector(s) of A and computes the shortest vectors V of A using the Fincke-Pohst algorithm [25]. For Gram matrices the shortest vector(s) can not be computed explicitly and therefore V is a 1×1 matrix with entry zero. By default p is set to the standard `bigfloat` precision. Depending on the application it might be necessary to use a higher precision in order to achieve a speed-up of the computation. The `lidia_error_handler` will be invoked if the basis flag is not set.

Close Vector Computation

Let A be of type `bigint_lattice`.

`void A.close_vector (const base_vector< bigint > & v, base_vector< bigint >& w, sdigit factor = 1)`

`base_vector< bigint > close_vector (const bigint_lattice & A, const base_vector< bigint > & v, sdigit factor = 1)`

computes a close lattice vector to a given vector v by using the algorithm of Babai [2]. With *factor* the precision for the approximations of computations of the LLL reduction which is part of the algorithm of Babai, will be determined as (*factor*· `double` precision). For *factor* = 1 (*factor* = 2) the approximations are done by using `doubles` (`xdoubles`), otherwise `bigfloats` with appropriate precision will be used. The `lidia_error_handler` will be invoked if the Gram flag is set or the basis flag is not set.

Special Functions

The union `lattice_info` is defined as:

```
typedef union {
          struct {
            lidia_size_t rank;
            double y;
            sdigit y_nom;
            sdigit y_denom;
            sdigit reduction_steps;
            sdigit correction_steps;
```

```
        sdigit swaps;
      } lll;
    } lattice_info;
```

So far `lattice_info` provides only information about the performed LLL reduction (e.g. number of swaps, reduction or correction steps) but in the future the union will be extended for providing information about the performance of other lattice algorithms.

In order to use a self-defined scalar product one has to define four functions corresponding to the following typedefs:

```
typedef void (*scal_dbl)(double&, double*, double*, lidia_size_t);
typedef void (*scal_xdbl)(xdouble&, xdouble*, xdouble*, lidia_size_t);
typedef void (*scal_bin)(bigint&, bigint*, bigint*, lidia_size_t);
typedef void (*scal_bfl)(bigfloat&, bigfloat*, bigfloat*, lidia_size_t);
```

The first parameter is the return value, the second and third parameter stand for the input vectors and the last parameter indicates the length of the vectors. With the addresses of the functions one defines the actual scalar product as:

```
typedef struct {
          scal_dbl dbl;
          scal_xdbl xdbl;
          scal_bin bin;
          scal_bfl bfl;
        } user_SP;
```

Please note that it is allowed to change the precision for `bigfloat`s within `void (*scal_bfl)(bigfloat&, bigfloat*, bigfloat*, lidia_size_t);` but one has to guarantee that after the computation of the scalar product the precision is set back to its original value (before entering the function).

Let A be of type `bigint_lattice`.

`void` A`.buchmann_kessler (math_matrix< bigint > &` T`, double` y`)`

`bigint_lattice buchmann_kessler (const bigint_lattice &` A`, math_matrix< bigint > &` T`, double` y`)`

`void` A`.buchmann_kessler (math_matrix< bigint > &` T`, double` y`, lattice_info &` li`)`

`bigint_lattice buchmann_kessler (const bigint_lattice &` A`, math_matrix< bigint > &` T`, double` y`, lattice_info &` li`)`

these functions compute the transformation matrix T as well as $A = (\underbrace{b_0, \ldots, b_{r-1}}_{=B}, \underbrace{0, \ldots, 0}_{k-r})$ where B is an LLL reduced basis of L with rank r by using the Buchmann-Kessler algorithm [13]. The algorithm is optimized for computing $\mathbb{Z}$-Bases of orders and might not work properly for arbitrarily chosen lattices. The last $k-r$ columns of T are relations for the original generating system. The `lidia_error_handler` will be invoked if the basis or Gram flag is set.

`void` A`.lll_schnorr_euchner_orig (double` y`, sdigit` *factor* `= 1)`

`bigint_lattice lll_schnorr_euchner_orig (const bigint_lattice &` A`, double` y`, sdigit` *factor* `= 1)`

`void` A`.lll_schnorr_euchner_orig (double` y`, lattice_info &` li`, user_SP, sdigit` *factor* `= 1)`

`bigint_lattice lll_schnorr_euchner_orig (const bigint_lattice &` A`, double` y`, lattice_info &` li`, user_SP, sdigit` *factor* `= 1)`

`void` A`.lll_schnorr_euchner_orig (double` y`, lattice_info &` li`, sdigit` *factor* `= 1)`

`bigint_lattice lll_schnorr_euchner_orig (const bigint_lattice &` A`, double` y`,`
`lattice_info &` li`, sdigit` *factor* `= 1)`

`void` A`.lll_schnorr_euchner_orig (double` y`, user_SP, sdigit` *factor* `= 1)`

`bigint_lattice lll_schnorr_euchner_orig (const bigint_lattice &` A`, double` y`, user_SP,`
`sdigit` *factor* `= 1)`

if A is not a Gram matrix, these functions compute $A = (\underbrace{b_0, \ldots, b_{r-1}}_{=B}, \underbrace{0, \ldots, 0}_{k-r})$ where B is an LLL reduced basis of L with rank r by using the original Schnorr-Euchner algorithm [55]. For linearly (in)dependent Gram matrices the computations are done according to [17]. $y \in]\frac{1}{2}, 1]$ is the reduction parameter. With *factor* the precision for the approximations of computations (depending on the algorithm) will be determined as (*factor*· `double` precision). For *factor* $= 1$ (*factor* $= 2$) the approximations are done by using `doubles` (`xdoubles`), otherwise `bigfloats` with appropriate precision will be used.

`void` A`.lll_schnorr_euchner_orig (math_matrix< bigint > &` T`, double` y`, sdigit` *factor* `= 1)`

`bigint_lattice lll_schnorr_euchner_orig (const bigint_lattice &` A`, math_matrix< bigint >`
`&` T`, double` y`, sdigit` *factor* `= 1)`

`void` A`.lll_schnorr_euchner_orig (math_matrix< bigint > &` T`, double` y`, lattice_info &` li`,`
`user_SP, sdigit` *factor* `= 1)`

`bigint_lattice lll_schnorr_euchner_orig (const bigint_lattice &` A`,`
`math_matrix< bigint > &` T`, double` y`,`
`lattice_info &` li`, user_SP, sdigit` *factor* `= 1)`

`void` A`.lll_schnorr_euchner_orig (math_matrix< bigint > &` T`, double` y`, lattice_info &` li`,`
`sdigit` *factor* `= 1)`

`bigint_lattice lll_schnorr_euchner_orig (const bigint_lattice &` A`,`
`math_matrix< bigint > &` T`, double` y`,`
`lattice_info &` li`, sdigit` *factor* `= 1)`

`void` A`.lll_schnorr_euchner_orig (math_matrix< bigint > &` T`, double` y`, user_SP,`
`sdigit` *factor* `= 1)`

`bigint_lattice lll_schnorr_euchner_orig (const bigint_lattice &` A`, math_matrix< bigint >`
`&` T`, double` y`, user_SP, sdigit` *factor* `= 1)`

if A is not a Gram matrix, these functions computes the transformation matrix T and $A = (\underbrace{b_0, \ldots, b_r - 1}_{=B}, \underbrace{0, \ldots, 0}_{k-r})$ where B is an LLL reduced basis of L with rank r by using the orignal Schnorr-Euchner algorithm [55]. The last $k - r$ columns of T are relations for the original generating system. For linearly (in)dependent Gram matrices the computations are done according to [17]. $y \in]\frac{1}{2}, 1]$ is the reduction parameter. With *factor* the precision for the approximations of computations (depending on the algorithm) will be determined as (*factor*· `double` precision). For *factor* $= 1$ (*factor* $= 2$) the approximations are done by using `doubles` (`xdoubles`), otherwise `bigfloats` with appropriate precision will be used.

`void` A`.lll_schnorr_euchner_fact (double` y`, sdigit` *factor* `= 1)`

`bigint_lattice lll_schnorr_euchner_fact (const bigint_lattice &` A`, double` y`,`
`sdigit` *factor* `= 1)`

`void` A`.lll_schnorr_euchner_fact (double` y`, lattice_info &` li`, user_SP, sdigit` *factor* `= 1)`

`bigint_lattice lll_schnorr_euchner_fact (const bigint_lattice & `A`, double `y`, lattice_info & `li`, user_SP, sdigit `*factor* ` = 1)`

`void `A`.lll_schnorr_euchner_fact (double `y`, lattice_info & `li`, sdigit `*factor* ` = 1)`

`bigint_lattice lll_schnorr_euchner_fact (const bigint_lattice & `A`, double `y`, lattice_info & `li`, sdigit `*factor* ` = 1)`

`void `A`.lll_schnorr_euchner_fact (double `y`, user_SP, sdigit `*factor* ` = 1)`

`bigint_lattice lll_schnorr_euchner_fact (const bigint_lattice & `A`, double `y`, user_SP, sdigit `*factor* ` = 1)`

if A is not a Gram matrix, these functions compute $A = (\underbrace{b_0, \ldots, b_{r-1}}_{=B}, \underbrace{0, \ldots, 0}_{k-r})$ where B is an LLL reduced basis of L with rank r by using a variation of the Schnorr-Euchner algorithm [64]. For linearly (in)dependent Gram matrices the computations are done according to [17]. $y \in]\frac{1}{2}, 1]$ is the reduction parameter. With *factor* the precision for the approximations of computations (depending on the algorithm) will be determined as (*factor*· double precision). For *factor* = 1 (*factor* = 2) the approximations are done by using `double`s (`xdouble`s), otherwise `bigfloat`s with appropriate precision will be used. This function is not completely tested yet but should be faster since one can use `double`s and `xdouble`s for doing the approximations even for higher dimensions and larger lattice entries.

`void `A`.lll_schnorr_euchner_fact (math_matrix< bigint > & `T`, double `y`, sdigit `*factor* ` = 1)`

`bigint_lattice lll_schnorr_euchner_fact (const bigint_lattice & `A`, math_matrix< bigint > & `T`, double `y`, sdigit `*factor* ` = 1)`

`void `A`.lll_schnorr_euchner_fact (math_matrix< bigint > & `T`, double `y`, lattice_info & `li`, user_SP, sdigit `*factor* ` = 1)`

`bigint_lattice lll_schnorr_euchner_fact (const bigint_lattice & `A`, math_matrix< bigint > & `T`, double `y`, lattice_info & `li`, user_SP, sdigit `*factor* ` = 1)`

`void `A`.lll_schnorr_euchner_fact (math_matrix< bigint > & `T`, double `y`, lattice_info & `li`, sdigit `*factor* ` = 1)`

`bigint_lattice lll_schnorr_euchner_fact (const bigint_lattice & `A`, math_matrix< bigint > & `T`, double `y`, lattice_info & `li`, sdigit `*factor* ` = 1)`

`void `A`.lll_schnorr_euchner_fact (math_matrix< bigint > & `T`, double `y`, user_SP, sdigit `*factor* ` = 1)`

`bigint_lattice lll_schnorr_euchner_fact (const bigint_lattice & `A`, math_matrix< bigint > & `T`, double `y`, user_SP, sdigit `*factor* ` = 1)`

if A is not a Gram matrix, these functions compute the transformation matrix T and $A = (\underbrace{b_0, \ldots, b_{r-1}}_{=B}, \underbrace{0, \ldots, 0}_{k-r})$ where B is an LLL reduced basis of L with rank r by using a variation of the Schnorr-Euchner algorithm [64]. The last $k - r$ columns of T are relations for the original generating system. For linearly (in)dependent Gram matrices the computations are done according to [17]. $y \in]\frac{1}{2}, 1]$ is the reduction parameter. With *factor* the precision for the approximations of computations (depending on the algorithm) will be determined as (*factor*· `double` precision). For *factor* = 1 (*factor* = 2) the approximations are done by using `double`s (`xdouble`s), otherwise `bigfloat`s with appropriate precision will be used. This function is not completely tested yet but should be faster since one can use `double`s and `xdouble`s for doing the approximations even for higher dimensions and larger lattice entries.

`void` A`.lll_benne_de_weger (sdigit` p`, sdigit` q`)`

`bigint_lattice lll_benne_de_weger (const bigint_lattice &` A`, sdigit` p`, sdigit` q`)`

`void` A`.lll_benne_de_weger (sdigit` p`, sdigit` q`, lattice_info &` li`)`

`bigint_lattice lll_benne_de_weger (const bigint_lattice &` A`, sdigit` p`, sdigit` q`, lattice_info &` li`)`

reduces the lattice A by using the algorithm of Benne de Weger [20]. The `lidia_error_handler` will be invoked if the generating system or Gram flag is set.

`void` A`.lll_benne_de_weger (math_matrix< bigint > &` T`, sdigit` p`, sdigit` q`)`

`bigint_lattice lll_benne_de_weger (const bigint_lattice &` A`, math_matrix< bigint > &` T`, sdigit` p`, sdigit` q`)`

`void` A`.lll_benne_de_weger (math_matrix< bigint > &` T`, sdigit` p`, sdigit` q`, lattice_info &` li`)`

`bigint_lattice lll_benne_de_weger (const bigint_lattice &` A`, math_matrix< bigint > &` T`, sdigit` p`, sdigit` q`, lattice_info &` li`)`

reduces the lattice A and computes the transformation matrix T by using the algorithm of Benne de Weger [20]. The `lidia_error_handler` will be invoked if the generating system or Gram flag is set.

`void` A`.mlll (double` y`, bigint *&` v`)`

`bigint_lattice mlll (const bigint_lattice &` A`, double` y`, bigint *&` v`)`

`void` A`.mlll (double` y`, base_vector< bigint > &` v`)`

`bigint_lattice mlll (const bigint_lattice &` A`, double` y`, base_vector< bigint > &` v`)`

`void` A`.mlll (double` y`, bigint *&` v`, lattice_info &` li`)`

`bigint_lattice mlll (const bigint_lattice &` A`, double` y`, bigint *&` v`, lattice_info &` li`)`

`void` A`.mlll (double` y`, base_vector< bigint > &` v`, lattice_info &` li`)`

`bigint_lattice mlll (const bigint_lattice &` A`, double` y`, base_vector< bigint > &` v`, lattice_info &` li`)`

these functions compute a $k-2$-dimensional reduced basis of $A = (a_0, \ldots, a_{k-1}) \in M^{k-1} \times k$ as well as a vector v satisfying $\sum_{i=0}^{k-1} v_i a_i = 0$ by using the modified LLL-algorithm [51]. $y \in]\frac{1}{2}, 1]$ is the reduction parameter. The `lidia_error_handler` will be invoked if the basis or Gram flag is set.

See also

`base_matrix`, `math_matrix`, `bigint_matrix`, `bigfloat_lattice`

Examples

```
#include <LiDIA/bigint_lattice.h>

void double_scalar(double &x, double *a, double *b,
                   lidia_size_t len)
{
```

```
    lidia_size_t i;

    x = 0;
    for(i = 0; i < len; i++)
        x += a[i] * b[i];
}


void xdouble_scalar(xdouble &x, xdouble *a, xdouble *b,
                    lidia_size_t len)
{
    lidia_size_t i;

    x = 0;
    for(i = 0; i < len; i++)
        x += a[i] * b[i];
}


void bigfloat_scalar(bigfloat &x, bigfloat *a, bigfloat *b,
                     lidia_size_t len)
{
    lidia_size_t i;

    x = 0;
    for(i = 0; i < len; i++)
        x += a[i] * b[i];
}


void bigint_scalar(bigint &x, bigint *a, bigint *b,
                   lidia_size_t len)
{
    lidia_size_t i;

    x = 0;
    for(i = 0; i < len; i++)
        x += a[i] * b[i];
}


int main()
{
    bigint_lattice L,B;
    math_matrix < bigint > SV;
    lattice_info li;
    bool gram, basis;
    lidia_size_t sv_len;
    user_SP usp = { double_scalar, xdouble_scalar,
                    bigint_scalar, bigfloat_scalar};

    cin >> L;

    cout << "Lattice:" << endl;
    cout << L << endl;

    gram = L.check_gram();
    basis = L.check_basis();
```

```
    B.assign(L);

    L.lll(0.99, li, usp);

    cout << "The reduced lattice is:" << endl;
    cout << L << endl;

    if(basis)
        cout << "The lattice was represented by a basis." << endl;
    else
        cout << "The lattice was represented by a generating system.\n" << endl;

    if(gram)
        cout << "The lattice was represented by a gram matrix." << endl;
    if(basis) {
        sv_len = B.shortest_vector(SV,50);
        cout << "\n The squared length of the shortest vector is: ";
        cout << sv_len << endl;
        if(!gram) {
            cout << "The shortest vector(s) is(are):" << endl;
            cout << SV;
        }
        cout << endl;
    }

    cout << "Some statistics:" << endl;
    cout << "Number of reduction steps:\t" << li.lll.reduction_steps << endl;
    cout << "Number of correction steps:\t" << li.lll.correction_steps << endl;
    cout << "Number of swaps:\t\t" << li.lll.swaps << endl;
    cout << "Reduction parameter was:\t" << li.lll.y << endl;

    return 0;
}
```

Example:

```
***********************************************

Lattice:

( 1212  911   450  288  116 )
( 938   709   278  254  108 )
( 1154  915   281  308  137 )
( 456   376   119  105  47  )
( 1529  1207  432  382  166 )

The reduced lattice is:           Some statistics:

( -6  -4   4  5   6  )            Number of reduction steps:      56
( -2   4  -6  5  -2  )            Number of correction steps:     34
(  1  -3  -2  7  -2  )            Number of swaps:                42
(  3   2  -1  5   10 )            Reduction parameter was:        0.99
(  2   3   6  1  -4  )

The lattice was represented by a basis.

The squared length of the shortest vector is: 54
```

```
The shortest vector(s) is(are):

(  4   6 )
( -4   2 )
(  3  -1 )
( -2  -3 )
( -3  -2 )

**************************************************

Lattice:

( 13029  10136  5458  3711  3358 )
( 10136  7894   4254  2882  2616 )
( 5458   4254   2300  1543  1418 )
( 3711   2882   1543  1068  945  )
( 3358   2616   1418  945   878  )

The reduced lattice is:            Some statistics:

( 8  2   2   3    0  )            Number of reduction steps:      33
( 2  9   3   3    0  )            Number of correction steps:     20
( 2  3   11  5   -4  )            Number of swaps:                20
( 3  3   5   12   1  )            Reduction parameter was:        0.99
( 0  0  -4   1    12 )

The lattice was represented by a basis.
The lattice was represented by a gram matrix.

The squared length of the shortest vector is: 18

**************************************************

Lattice:

( 1196  2316  661   929   882   117 )
( 1208  2460  810   1294  1187  154 )
( 1771  2398  741   1369  1075  126 )
( 1353  2373  845   1441  1389  184 )
( 2122  3507  1124  1937  1637  202 )

The reduced lattice is:            Some statistics:

( 0  0  0  -1   2  0 )             Number of reduction steps:      158
( 1  0  0   0   0  0 )             Number of correction steps:     90
( 0  1  0   0   0  0 )             Number of swaps:                140
( 0  0  1  -1  -1  0 )             Reduction parameter was:        0.99
( 0  0  1   0   2  0 )

The lattice was represented by a generating system.
```

Author

Werner Backes, Thorsten Lauer, Oliver van Sprang, Susanne Wetzel
(code fragments written by Jutta Bartholomes)

bigfloat_lattice

Name

`bigfloat_lattice` ..... lattice algorithms

Abstract

The class `bigfloat_lattice` is derived from the class `math_matrix` and therefore inherits all their functions. In addition the class `bigfloat_lattice` provides algorithms for shortest vector computations, lattice reduction of lattice bases or linearly independent Gram matrices as well as computations of lattice bases and relations from generating systems or linearly dependent Gram matrices. A lattice L is given by $A = (a_0, \ldots, a_{k-1}) \in M^{n \times k}$ $(n, k \in \mathbb{N})$ where M consists of elements of type `bigfloat` and A is a linearly (in)dependent Gram matrix, a generating system or a lattice basis of L.

Description

A variable of type `bigfloat_lattice` cosists of the same components as `math_matrix< bigfloat >`.

Let A be of class `bigfloat_lattice`. In the present version of LiDIA the following algorithms are implemented:

1. Generating systems:
   - The MLLL algorithm [51] which computes for a $k-1$-dimensional lattice L represented by a generating system $A = (a_0, \ldots, a_{k-1}) \in M^{k-1 \times k}$ a reduced basis $B = (b_0, \ldots, b_{k-2})$ as well as a relation i.e., integers $x_0, \ldots, x_{k-1} \in \mathbb{Z}$ satisfying $\sum_{i=0}^{k-1} x_i a_i = 0$.
   - The Buchmann-Kessler algorithm [13] which computes a reduced basis for a lattice represented by a generating system A.
   - The original Schnorr-Euchner algorithm [55] (modified for generating systems) as well as several variations [64] which compute an LLL reduced basis of the lattice L represented by the generating system A.
2. Lattice bases:
   - The original Schnorr-Euchner lattice reduction algorithm [55] as well as variations [64].
   - The algorithm of Babai [2] for computing a close lattice vector to a given vector.
3. Gram matrices:
   - The original Schnorr-Euchner lattice reduction algorithm [55] as well as variations [64].

In the following we will explain the constructors, destructors and basic functions of the classes `bigfloat_lattice`. Lattices can be stored column by column or row by row. Independent of the the storage mode either the columns or the rows of the lattice can be reduced. The following description is done with respect to column-oriented lattices i.e the reduction is done columnwise. For row-oriented lattices one has to change the representation. The corresponding computations will then be adjusted automatically.

Constructors/Destructor

For the following constructors the variable A is assumed to represent a lattice. If in the following constructors $n \leq 1$, $k \leq 1$, $n < k$, $A =$ `NULL` or the dimension of A does not correspond to the input parameters which declare the dimension, the `lidia_error_handler` will be invoked.

`ct bigfloat_lattice ()`

constructs a 1×1 `bigfloat_lattice`.

`ct bigfloat_lattice (lidia_size_t` n`, lidia_size_t` k`)`

constructs a `bigfloat_lattice` of k vectors embedded in an n-dimensional vector space initialized with zero.

`ct bigfloat_lattice (const bigfloat_matrix &` A`)`

constructs a copy of A.

`ct bigfloat_lattice (lidia_size_t` n`, lidia_size_t` k`, bigfloat** &` A`)`

constructs an $n \times k$-`bigfloat_lattice` initialized with the values of the 2-dimensional array A where $*A$ is a row vector.

`dt ~bigfloat_lattice ()`

Initialization

Let A be of type `bigfloat_lattice`. If none of the following functions is called explicitly by the user, A will be considered as column-oriented generating system representing the lattice L. Changes of the lattice by the user will automatically result in the deletion of the basis and gram flag.

`void` A`.set_basis_flag ()`

sets the basis flag without checking whether A satisfies the basis property or not. For future computations A is now considered as basis representing the lattice L. Setting the flag without knowing whether A is a basis or not may result in abortions of further computations or wrong results. By default the representation is set as generating system.

`void` A`.delete_basis_flag ()`

sets the basis flag to `false`. For future computations A is now considered as generating system representing the lattice L.

`void` A`.set_gram_flag ()`

sets the gram flag without checking whether A satisfies the properties of a Gram matrix or not. For future computations A is now considered as Gram matrix representing the lattice L. Setting the flag without knowing whether A is a Gram matrix or not may result in abortions of further computations or wrong results. By default the gram flag is not set.

`void` A`.delete_gram_flag ()`

sets the gram flag to `false`.

`void` A`.set_red_orientation_columns ()`

sets the orientation such that the reduction will be done column-oriented. By default the orientation is taken as column-oriented.

`void` A`.set_red_orientation_rows ()`

sets the orientation such that the reduction will be done row-oriented. By default the orientation is taken as column-oriented.

Basic Methods and Functions

Let A be of type `bigfloat_lattice`.

`bool` A`.check_basis ()`

returns `true` and sets the basis flag if the columns or rows of A are linearly independent according to the orientation of A, `false` otherwise.

`bool` A`.check_gram ()`

returns `true` and sets the gram flag if A is symmetric and positive semi-definete and therefore satisfies the properties of a Gram matrix, `false` otherwise.

`bool` A`.get_red_orientation ()`

returns `true` for column-oriented lattices, `false` otherwise.

`bool` A`.lll_check (double` y`)`

returns `true` if the basis is reduced for the reduction parameter y, `false` otherwise. The `lidia_error_handler` will be invoked if the function is applied to generating systems or the gram flag is set.

`bool` A`.lll_check (sdigit` p`, sdigit` q`)`

returns `true` if the basis is reduced for the reduction parameter $\frac{p}{q}$, `false` otherwise. The `lidia_error_handler` will be invoked if the function is applied to generating systems or the gram flag is set.

`double` A`.lll_check_search ()`

if A is LLL reduced, this function will return the largest $y \in]\frac{1}{2}, 1]$ for which A is still LLL reduced, see [39, 55]. If A is not LLL reduced, 0.0 will be returned. The `lidia_error_handler` will be invoked if the function is applied to generating systems or the gram flag is set.

`void` A`.lll_check_search (sdigit` p`&, sdigit&` q`)`

if A is LLL reduced, this function will return the largest $\frac{p}{q} \in]\frac{1}{2}, 1]$ for which A is still LLL reduced, see [39, 55]. If A is not LLL reduced, 0 will be returned. The `lidia_error_handler` will be invoked if the function is applied to generating systems or the gram flag is set.

`void` A`.randomize_vectors ()`

permutes the vectors of the given lattice randomly.

`void` A`.sort_vectors (bfl_cmp_func)`

sorts the vectors of the given lattice such that their lengths (length in the meaning of the bfl_cmp_func, which is defined as `int (*bfl_cmp_func)(const bigfloat*, const bigfloat*, long))` are in descending order. The function `bfl_cmp_func` has to return -1 (1) if the first vector is shorter (longer) as the second one and 0 if they have the same length.

`void` A`.sort_big_vectors ()`

sorts the vectors of the given lattice such that their lengths (by means of the Euclidian norm) are in descending order.

`void` *A*`.sort_small_vectors ()`

sorts the vectors of the given lattice such that their lengths (by means of the Euclidian norm) are in ascending order.

Lattice Reduction

The following functions can be used for performing LLL reductions. These functions are interface functions and point to the fastest algorithms described in the section 'Special Functions' for solving arbitrarily chosen lattice instances. Depending on the application of the lattice problem you might want to use a different algorithm mentioned in that section though. Please note that those algorithms might change in the future and we therefore suggest to use the interface functions. At the moment the interface functions point to the `lll_schnorr_euchner_fact` implementation.

The union `lattice_info` is defined as:

```
typedef union {
          struct {
            lidia_size_t rank;
            double y;
            sdigit y_nom;
            sdigit y_denom;
            sdigit reduction_steps;
            sdigit correction_steps;
            sdigit swaps;
          } lll;
        } lattice_info;
```

So far `lattice_info` provides only information about the performed LLL reduction (e.g. number of swaps, reduction or correction steps) but in the future the union will be extended for providing information about the performance of other lattice algorithms.

In order to use a self-defined scalar product one has to define four functions corresponding to the following typedefs:

```
typedef void (*scal_dbl)(double&, double*, double*, lidia_size_t);
typedef void (*scal_xdbl)(xdouble&, xdouble*, xdouble*, lidia_size_t);
typedef void (*scal_bin)(bigint&, bigint*, bigint*, lidia_size_t);
typedef void (*scal_bfl)(bigfloat&, bigfloat*, bigfloat*, lidia_size_t);
```

The first parameter is the return value, the second and third parameter stand for the input vectors and the last parameter indicates the length of the vectors. With the addresses of the functions one defines the actual scalar product as:

```
typedef struct {
          scal_dbl dbl;
          scal_xdbl xdbl;
          scal_bin bin;
          scal_bfl bfl;
        } user_SP;
```

Please note that it is allowed to change the precision for `bigfloat`s within `void (*scal_bfl)(bigfloat&, bigfloat*, bigfloat*, lidia_size_t);` but one has to guarantee that after the computation of the scalar product the precision is set back to its original value (before entering the function).

Let *A* be of type `bigfloat_lattice`.

`void` *A*`.lll (double` *y*`, sdigit` *factor* `= 1)`

`bigfloat_lattice lll (const bigfloat_lattice & A, double y, sdigit factor = 1)`

`void A.lll (double y, lattice_info & li, user_SP, sdigit factor = 1)`

`bigfloat_lattice lll (const bigfloat_lattice & A, double y, lattice_info & li, user_SP, sdigit factor = 1)`

`void A.lll (double y, lattice_info & li, sdigit factor = 1)`

`bigfloat_lattice lll (const bigfloat_lattice & A, double y, lattice_info & li, sdigit factor = 1)`

`void A.lll (double y, user_SP, sdigit factor = 1)`

`bigfloat_lattice lll (const bigfloat_lattice & A, double y, user_SP, sdigit factor = 1)`

if A is not a Gram matrix, these functions compute $A = (\underbrace{a_0,\ldots,a_{r-1}}_{=B},\underbrace{0,\ldots,0}_{k-r})$ where B is an LLL reduced basis of L with rank r. For linearly (in)dependent Gram matrices the computations are done according to [17]. $y \in]\frac{1}{2},1]$ is the reduction parameter. With *factor* the precision for the approximations of computations (depending on the algorithm) will be determined as (*factor*· `double` precision). For *factor* $= 1$ (*factor* $= 2$) the approximations are done by using `double`s (`xdouble`s), otherwise `bigfloat`s with appropriate precision will be used.

`void A.lll (math_matrix< bigfloat > & T, double y, sdigit factor = 1)`

`bigfloat_lattice lll (const bigfloat_lattice & A, math_matrix< bigfloat > & T, double y, sdigit factor = 1)`

`void A.lll (math_matrix< bigint > & T, double y, sdigit factor = 1)`

`bigfloat_lattice lll (const bigfloat_lattice & A, math_matrix< bigint > & T, double y, sdigit factor = 1)`

`void A.lll (math_matrix< bigfloat > & T, double y, lattice_info & li, user_SP, sdigit factor = 1)`

`bigfloat_lattice lll (const bigfloat_lattice & A, math_matrix< bigfloat > & T, double y, lattice_info & li, user_SP, sdigit factor = 1)`

`void A.lll (math_matrix< bigfloat > & T, double y, lattice_info & li, sdigit factor = 1)`

`bigfloat_lattice lll (const bigfloat_lattice & A, math_matrix< bigfloat > & T, double y, lattice_info & li, sdigit factor = 1)`

`void A.lll (math_matrix< bigint > & T, double y, lattice_info & li, user_SP, sdigit factor = 1)`

`bigfloat_lattice lll (const bigfloat_lattice & A, math_matrix< bigint > & T, double y, lattice_info & li, user_SP, sdigit factor = 1)`

`void A.lll (math_matrix< bigint > & T, double y, lattice_info & li, sdigit factor = 1)`

`bigfloat_lattice lll (const bigfloat_lattice & A, math_matrix< bigint > & T, double y, lattice_info & li, sdigit factor = 1)`

`void A.lll (math_matrix< bigfloat > & T, double y, user_SP, sdigit factor = 1)`

`bigfloat_lattice lll (const bigfloat_lattice & A, math_matrix< bigfloat > & T, double y, user_SP, sdigit factor = 1)`

`void` A`.lll (math_matrix< bigint > &` T`, double` y`, user_SP, sdigit` *factor* `= 1)`

`bigfloat_lattice lll (const bigfloat_lattice &` A`, math_matrix< bigint > &` T`, double` y`, user_SP, sdigit` *factor* `= 1)`

if A is not a Gram matrix, these functions compute the transformation matrix T and $A = (\underbrace{b_0, \ldots, b_{r-1}}_{=B}, \underbrace{0, \ldots, 0}_{k-r})$ where B is an LLL reduced basis of L with rank r. The last $k - r$ columns of T are relations for the original generating system. For linearly (in)dependent Gram matrices the computations are done according to [17]. $y \in]\frac{1}{2}, 1]$ is the reduction parameter. With *factor* the precision for the approximations of computations (depending on the algorithm) will be determined as (*factor*· `double` precision). For *factor* $= 1$ (*factor* $= 2$) the approximations are done by using `double`s (`xdouble`s), otherwise `bigfloat`s with appropriate precision will be used.

Close Vector Computation

Let A be of type `bigfloat_lattice`.

`void` A`.close_vector (const base_vector< bigfloat > &` v`, base_vector< bigfloat > &` w`, sdigit` *factor* `= 1)`

`base_vector< bigfloat > close_vector (const bigfloat_lattice &` A`, const base_vector< bigfloat >&` v`, sdigit` *factor* `= 1)`

computes a close lattice vector to a given vector v by using the algorithm of Babai [2]. With *factor* the precision for the approximations of computations of the LLL reduction which is part of the algorithm of Babai, will be determined as (*factor*· `double` precision). For *factor* $= 1$ (*factor* $= 2$) the approximations are done by using `double`s (`xdouble`s), otherwise `bigfloat`s with appropriate precision will be used. The `lidia_error_handler` will be invoked if the gram flag is set or the basis flag is not set.

Special Functions

The union `lattice_info` is defined as:

```
typedef union {
          struct {
            lidia_size_t rank;
            double y;
            sdigit y_nom;
            sdigit y_denom;
            sdigit reduction_steps;
            sdigit correction_steps;
            sdigit swaps;
          } lll;
        } lattice_info;
```

So far `lattice_info` provides only information about the performed LLL reduction (e.g. number of swaps, reduction or correction steps) but in the future the union will be extended for providing information about the performance of other lattice algorithms.

In order to use a self-defined scalar product one has to define four functions corresponding to the following typedefs:

```
typedef void (*scal_dbl)(double&, double*, double*, lidia_size_t);
typedef void (*scal_xdbl)(xdouble&, xdouble*, xdouble*, lidia_size_t);
typedef void (*scal_bin)(bigint&, bigint*, bigint*, lidia_size_t);
typedef void (*scal_bfl)(bigfloat&, bigfloat*, bigfloat*, lidia_size_t);
```

The first parameter is the return value, the second and third parameter stand for the input vectors and the last parameter indicates the length of the vectors. With the addresses of the functions one defines the actual scalar product as:

```
typedef struct {
          scal_dbl dbl;
          scal_xdbl xdbl;
          scal_bin bin;
          scal_bfl bfl;
        } user_SP;
```

Please note that it is allowed to change the precision for `bigfloat`s within `void (*scal_bfl)(bigfloat&, bigfloat*, bigfloat*, lidia_size_t);` but one has to guarantee that after the computation of the scalar product the precision is set back to its original value (before entering the function).

Let A be of type `bigfloat_lattice`.

`void` A`.buchmann_kessler (math_matrix< bigfloat > &` T`, double` y`)`

`bigfloat_lattice buchmann_kessler (const bigfloat_lattice &` A`, math_matrix< bigfloat > &` T`, double` y`)`

`void` A`.buchmann_kessler (math_matrix< bigint >&` T`, double` y`)`

`bigint_lattice buchmann_kessler (const bigfloat_lattice &` A`, math_matrix< bigint >&` T`, double` y`)`

`void` A`.buchmann_kessler (math_matrix< bigfloat > &` T`, double` y`, lattice_info &` li`)`

`bigfloat_lattice buchmann_kessler (const bigfloat_lattice &` A`, math_matrix< bigfloat > &` T`, double` y`, lattice_info &` li`)`

`void` A`.buchmann_kessler (math_matrix< bigint >&` T`, double` y`, lattice_info &` li`)`

`bigint_lattice buchmann_kessler (const bigfloat_lattice &` A`, math_matrix< bigint > &` T`, double` y`, lattice_info &` li`)`

these functions compute the transformation matrix T as well as $A = (\underbrace{b_0, \ldots, b_{r-1}}_{=B}, \underbrace{0, \ldots, 0}_{k-r})$ where B is an LLL reduced basis of L with rank r by using the Buchmann-Kessler algorithm [13]. The last $k - r$ columns of T are relations for the original generating system. The algorithm is optimized for computing $\mathbb{Z}$-Bases of orders and might not work properly for arbitrarily chosen lattices. The `lidia_error_handler` will be invoked if the basis or gram flag is set.

`void` A`.lll_schnorr_euchner_orig (double` y`, sdigit` *factor* `= 1)`

`bigfloat_lattice lll_schnorr_euchner_orig (const bigfloat_lattice &` A`, double` y`, sdigit` *factor* `= 1)`

`void` A`.lll_schnorr_euchner_orig (double` y`, lattice_info &` li`, user_SP, sdigit` *factor* `= 1)`

`bigfloat_lattice lll_schnorr_euchner_orig (const bigfloat_lattice &` A`, double` y`, lattice_info &` li`, user_SP, sdigit` *factor* `= 1)`

`void` A`.lll_schnorr_euchner_orig (double` y`, lattice_info &` li`, sdigit` *factor* `= 1)`

`bigfloat_lattice lll_schnorr_euchner_orig (const bigfloat_lattice &` A`, double` y`, lattice_info &` li`, sdigit` *factor* `= 1)`

`void` A`.lll_schnorr_euchner_orig (double` y`, user_SP, sdigit` *factor* `= 1)`

```
bigfloat_lattice lll_schnorr_euchner_orig (const bigfloat_lattice & A, double y,
                                           user_SP, sdigit factor = 1)
```

if A is not a Gram matrix, these functions compute $A = (\underbrace{b_0, \ldots, b_{r-1}}_{=B}, \underbrace{0, \ldots, 0}_{k-r})$ where B is an LLL reduced basis of L with rank r by using the original Schnorr-Euchner algorithm [55]. For linearly (in)dependent Gram matrices the computations are done according to [17]. $y \in]\frac{1}{2}, 1]$ is the reduction parameter. With *factor* the precision for the approximations of computations (depending on the algorithm) will be determined as (*factor*· `double` precision). For *factor* = 1 (*factor* = 2) the approximations are done by using `double`s (`xdouble`s), otherwise `bigfloat`s with appropriate precision will be used.

```
void A.lll_schnorr_euchner_orig (math_matrix< bigfloat > & T, double y,
                                 sdigit factor = 1)

bigfloat_lattice lll_schnorr_euchner_orig (const bigfloat_lattice & A, math_matrix<
                                           bigfloat > & T, double y, sdigit factor = 1)

void A.lll_schnorr_euchner_orig (math_matrix< bigint > & T, double y, sdigit factor = 1)

bigfloat_lattice lll_schnorr_euchner_orig (const bigfloat_lattice & A, math_matrix<
                                           bigint > & T, double y, sdigit factor = 1)

void A.lll_schnorr_euchner_orig (math_matrix< bigfloat > & T, double y,
                                 lattice_info & li, user_SP, sdigit factor = 1)

bigfloat_lattice lll_schnorr_euchner_orig (const bigfloat_lattice & A,
                                           math_matrix< bigfloat > & T, double y,
                                           lattice_info & li, user_SP, sdigit factor = 1)

void A.lll_schnorr_euchner_orig (math_matrix< bigfloat > & T, double y,
                                 lattice_info & li, sdigit factor = 1)

bigfloat_lattice lll_schnorr_euchner_orig (const bigfloat_lattice & A,
                                           math_matrix< bigfloat > & T, double y,
                                           lattice_info & li, sdigit factor = 1)

void A.lll_schnorr_euchner_orig (math_matrix< bigint > & T, double y, lattice_info & li,
                                 user_SP, sdigit factor = 1)

bigfloat_lattice lll_schnorr_euchner_orig (const bigfloat_lattice & A,
                                           math_matrix< bigint > & T, double y,
                                           lattice_info & li, user_SP, sdigit factor = 1)

void A.lll_schnorr_euchner_orig (math_matrix< bigint > & T, double y, lattice_info & li,
                                 sdigit factor = 1)

bigfloat_lattice lll_schnorr_euchner_orig (const bigfloat_lattice & A,
                                           math_matrix< bigint > & T, double y,
                                           lattice_info & li, sdigit factor = 1)

void A.lll_schnorr_euchner_orig (math_matrix< bigfloat > & T, double y, user_SP,
                                 sdigit factor = 1)

bigfloat_lattice lll_schnorr_euchner_orig (const bigfloat_lattice & A,
                                           math_matrix< bigfloat > & T, double y,
                                           user_SP, sdigit factor = 1)

void A.lll_schnorr_euchner_orig (math_matrix< bigint > & T, double y, user_SP,
                                 sdigit factor = 1)
```

```
bigfloat_lattice lll_schnorr_euchner_orig (const bigfloat_lattice & A,
                                           math_matrix< bigint > & T, double y, user_SP,
                                           sdigit factor = 1)
```

if A is not a Gram matrix, these functions compute the transformation matrix T and $A = (\underbrace{b_0, \ldots, b_{r-1}}_{=B}, \underbrace{0, \ldots, 0}_{k-r})$ where B is an LLL reduced basis of L with rank r by using the orignal Schnorr-Euchner algorithm [55]. The last $k - r$ columns of T are relations for the original generating system. For linearly (in)dependent Gram matrices the computations are done according to [17]. $y \in]\frac{1}{2}, 1]$ is the reduction parameter. With *factor* the precision for the approximations of computations (depending on the algorithm) will be determined as (*factor*· `double` precision). For *factor* = 1 (*factor* = 2) the approximations are done by using `doubles` (`xdoubles`), otherwise `bigfloats` with appropriate precision will be used.

```
void A.lll_schnorr_euchner_fact (double y, sdigit factor = 1)

bigfloat_lattice lll_schnorr_euchner_fact (const bigfloat_lattice & A, double y,
                                           sdigit factor = 1)

void A.lll_schnorr_euchner_fact (double y, lattice_info & li, user_SP, sdigit factor = 1)

bigfloat_lattice lll_schnorr_euchner_fact (const bigfloat_lattice & A, double y,
                                           lattice_info & li, user_SP, sdigit factor = 1)

void A.lll_schnorr_euchner_fact (double y, lattice_info & li, sdigit factor = 1)

bigfloat_lattice lll_schnorr_euchner_fact (const bigfloat_lattice & A, double y,
                                           lattice_info & li, sdigit factor = 1)

void A.lll_schnorr_euchner_fact (double y, user_SP, sdigit factor = 1)

bigfloat_lattice lll_schnorr_euchner_fact (const bigfloat_lattice & A, double y,
                                           user_SP, sdigit factor = 1)
```

if A is not a Gram matrix, these functions compute $A = (\underbrace{b_0, \ldots, b_{r-1}}_{=B}, \underbrace{0, \ldots, 0}_{k-r})$ where B is an LLL reduced basis of L with rank r by using a variation of the Schnorr-Euchner algorithm [64]. For linearly (in)dependent Gram matrices the computations are done according to [17]. $y \in]\frac{1}{2}, 1]$ is the reduction parameter. With *factor* the precision for the approximations of computations (depending on the algorithm) will be determined as (*factor*· `double` precision). For *factor* = 1 (*factor* = 2) the approximations are done by using `doubles` (`xdoubles`), otherwise `bigfloats` with appropriate precision will be used. In general these functions should be much faster than the ones using the original Schnorr-Euchner algorithm.

```
void A.lll_schnorr_euchner_fact (math_matrix< bigfloat > & T, double y,
                                 sdigit factor = 1)

bigfloat_lattice lll_schnorr_euchner_fact (const bigfloat_lattice & A, math_matrix<
                                           bigfloat > & T, double y, sdigit factor = 1)

void A.lll_schnorr_euchner_fact (math_matrix< bigint > & T, double y, sdigit factor = 1)

bigfloat_lattice lll_schnorr_euchner_fact (const bigfloat_lattice & A, math_matrix<
                                           bigint > & T, double y, sdigit factor = 1)

void A.lll_schnorr_euchner_fact (math_matrix< bigfloat > & T, double y,
                                 lattice_info & li, user_SP, sdigit factor = 1)

bigfloat_lattice lll_schnorr_euchner_fact (const bigfloat_lattice & A,
                                           math_matrix< bigfloat > & T, double y,
                                           lattice_info & li, user_SP, sdigit factor = 1)
```

```
void A.lll_schnorr_euchner_fact (math_matrix< bigfloat > & T, double y,
                                 lattice_info & li, sdigit factor = 1)

bigfloat_lattice lll_schnorr_euchner_fact (const bigfloat_lattice & A,
                                           math_matrix< bigfloat > & T, double y,
                                           lattice_info & li, sdigit factor = 1)

void A.lll_schnorr_euchner_fact (math_matrix< bigint > & T, double y, lattice_info & li,
                                 user_SP, sdigit factor = 1)

bigfloat_lattice lll_schnorr_euchner_fact (const bigfloat_lattice & A,
                                           math_matrix< bigint > & T, double y,
                                           lattice_info & li, user_SP, sdigit factor = 1)

void A.lll_schnorr_euchner_fact (math_matrix< bigint > & T, double y, lattice_info & li,
                                 sdigit factor = 1)

bigfloat_lattice lll_schnorr_euchner_fact (const bigfloat_lattice & A,
                                           math_matrix< bigint > & T, double y,
                                           lattice_info & li, sdigit factor = 1)

void A.lll_schnorr_euchner_fact (math_matrix< bigfloat > & T, double y, user_SP,
                                 sdigit factor = 1)

bigfloat_lattice lll_schnorr_euchner_fact (const bigfloat_lattice & A,
                                           math_matrix< bigfloat > & T, double y,
                                           user_SP, sdigit factor = 1)

void A.lll_schnorr_euchner_fact (math_matrix< bigint > & T, double y, user_SP,
                                 sdigit factor = 1)

bigfloat_lattice lll_schnorr_euchner_fact (const bigfloat_lattice & A,
                                           math_matrix< bigint > & T, double y, user_SP,
                                           sdigit factor = 1)
```

if A is not a Gram matrix, these functions compute the transformation matrix T and $A = (\underbrace{b_0, \ldots, b_{r-1}}_{=B}, \underbrace{0, \ldots, 0}_{k-r})$ where B is an LLL reduced basis of L with rank r by using a variation of the Schnorr-Euchner algorithm [64]. The last $k - r$ columns of T are relations for the original generating system. For linearly (in)dependent Gram matrices the computations are done according to [17]. $y \in]\frac{1}{2}, 1]$ is the reduction parameter. With *factor* the precision for the approximations of computations (depending on the algorithm) will be determined as (*factor*· `double` precision). For *factor* = 1 (*factor* = 2) the approximations are done by using `double`s (`xdouble`s), otherwise `bigfloat`s with appropriate precision will be used. In general these functions should be much faster than the ones using the original Schnorr-Euchner algorithm.

```
void A.mlll (double y, bigint *& v)

bigfloat_lattice mlll (const bigfloat_lattice & A, double y, bigint *& v)

void A.mlll (double y, bigfloat *& v)

bigfloat_lattice mlll (const bigfloat_lattice & A, double y, bigfloat *& v)

void A.mlll (double y, base_vector< bigint > & v)

bigfloat_lattice mlll (const bigfloat_lattice & A, double y, base_vector< bigint > & v)

void A.mlll (double y, base_vector< bigfloat > & v)
```

```
bigfloat_lattice mlll (const bigfloat_lattice & A, double y,
                       base_vector< bigfloat > & v)

void A.mlll (double y, lattice_info & li, bigint *& v)

bigfloat_lattice mlll (const bigfloat_lattice & A, double y, lattice_info & li,
                       bigint *& v)

void A.mlll (double y, lattice_info & li, bigfloat*& v)

bigfloat_lattice mlll (const bigfloat_lattice & A, double y, lattice_info & li,
                       bigfloat *& v)

void A.mlll (double y, lattice_info & li, base_vector< bigint > & v)

bigfloat_lattice mlll (const bigfloat_lattice & A, double y, lattice_info & li,
                       base_vector< bigint > & v)

void A.mlll (double y, lattice_info & li, base_vector< bigfloat > & v)

bigfloat_lattice mlll (const bigfloat_lattice & A, double y, lattice_info & li,
                       base_vector< bigfloat > & v)
```

these functions compute a $k-2$-dimensional reduced basis of $A = (a_0, \ldots, a_k - 1) \in M^{n \times k}$ as well as a vector v satisfying $\sum_{i=0}^{k-1} v_i a_i = 0$ by using the modified LLL-algorithm [51]. $y \in]\frac{1}{2}, 1]$ is the reduction parameter. The `lidia_error_handler` will be invoked if the basis or gram flag is set.

See also

`base_matrix`, `math_matrix`, `bigint_lattice`

Examples

```
#include <LiDIA/bigfloat_lattice.h>

void double_scalar(double &x, double *a, double *b,
                   lidia_size_t len)
{
    lidia_size_t i;

    x = 0;
    for(i = 0; i < len; i++)
        x += a[i] * b[i];
}


void xdouble_scalar(xdouble &x, xdouble *a, xdouble *b,
                    lidia_size_t len)
{
    lidia_size_t i;

    x = 0;
    for(i = 0; i < len; i++)
        x += a[i] * b[i];
}
```

```
void bigfloat_scalar(bigfloat &x, bigfloat *a, bigfloat *b,
                     lidia_size_t len)
{
    lidia_size_t i;
    long old_prec;

    old_prec = x.get_precision();
    x.precision(old_prec*4);
    x = 0;
    for(i = 0; i < len; i++)
        x += a[i] * b[i];
    x.precision(old_prec);
}


void bigint_scalar(bigint &x, bigint *a, bigint *b,
                   lidia_size_t len)
{
    lidia_size_t i;

    x = 0;
    for(i = 0; i < len; i++)
        x += a[i] * b[i];
}


int main()
{
    bigfloat_lattice L;
    lattice_info li;
    bool gram, basis;
    user_SP usp = { double_scalar, xdouble_scalar,
                    bigint_scalar, bigfloat_scalar};

    cin >> L;

    cout << "Lattice:" << endl;
    cout << L << endl;

    gram = L.check_gram();
    basis = L.check_basis();

    L.lll(0.99, li, usp);

    cout << "The reduced lattice is:" << endl;
    cout << L << endl;

    if(basis)
        cout << "The lattice was represented "
                "by a basis." << endl;
    else
        cout << "The lattice was represented "
                "by a generating system." << endl;

    if(gram)
        cout << "The lattice was represented by a gram matrix." << endl;
    cout << "\nSome statistics:" << endl;
```

```
    cout << "Number of reduction steps:\t";
    cout << li.lll.reduction_steps << endl;
    cout << "Number of correction steps:\t";
    cout << li.lll.correction_steps << endl;
    cout << "Number of swaps:\t\t";
    cout << li.lll.swaps << endl;
    cout << "Reduction parameter was:\t";
    cout << li.lll.y << endl;

    return 0;
}
```

Example:

```
************************************************

Lattice:

( 15.5    17.46   36.24  12.1   2.5   )
( 145.5   57.5    158.4  44.75  10.2  )
( 218.56  27.78   117    13.1   37    )
( 211.5   119.75  28     23.44  11.66 )
( 77.2    369.85  41.6   34.14  68.76 )

The reduced lattice is:                 Some statistics:

( -6.96  -4.7   -9.6    19.06 -103.5 )
(  26.55 -3.9   -34.55  18.2   8.25  )   Number of reduction steps:   22
(  3.46  -55.62  23.9   9.64   27.4  )   Number of correction steps:  12
( -15.96  7.35  -11.78  39.4   24.77 )   Number of swaps:             16
(  7.64  -0.15   34.62  26.5  -21.13 )   Reduction parameter was:     0.99

The lattice was represented by a basis.

************************************************

Lattice:

( 84.43   141.58  237.04  195.19  109.83 )
( 141.58  424.59  414.22  608.8   278.13 )
( 237.04  414.22  710.7   622.39  320.35 )
( 195.19  608.8   622.39  978.26  440.06 )
( 109.83  278.13  320.35  440.06  229.41 )

The reduced lattice is:                  Some statistics:

(  26.17 -12.37  -8.15   -1.11  -7.8   )  Number of reduction steps:  10
( -12.37  29.23   0.5e-1 -6      1.03  )  Number of correction steps:  7
( -8.15   0.5e-1  48.33   6.77   15.26 )  Number of swaps:              9
( -1.11  -6       6.77    73.79 -0.29  )  Reduction parameter was:    0.99
( -7.8    1.03    15.26  -0.29   83.61 )

The lattice was represented by a basis.
The lattice was represented by a gram matrix.
```

Author

Werner Backes, Thorsten Lauer, Oliver van Sprang, Susanne Wetzel (code fragments written by Jutta Bartholomes)

The LiDIA NF package

Chapter 16

Quadratic Number Fields

quadratic_form

Name

`quadratic_form` ..... binary quadratic forms

Abstract

`quadratic_form` is a class which represents binary quadratic forms, i.e., polynomials of the form $f(X,Y) = aX^2 + bXY + cY^2 \in \mathbb{Z}[X,Y]$. It supports many algorithms which evolve in the study of binary quadratic forms, including equivalence testing, composition, etc.

Description

A `quadratic_form` consists of an ordered triple (a,b,c) which represents the form $f(X,Y) = aX^2+bXY+cY^2$. The variables a, b, and c are all of type `bigint`. In addition to a, b, and c, a pointer to a `quadratic_order` is stored with each instance of `quadratic_form`. If $\Delta = b^2 - 4ac$, the discriminant of f, is not a perfect square, the pointer points to a `quadratic_order` of discriminant Δ. If Δ is a perfect square (i.e., f is irregular), it points to a `quadratic_order` of discriminant 0.

Constructors/Destructor

`ct quadratic_form ()`

`ct quadratic_form (const bigint & a, const bigint & b, bigint & c)`
initializes with the quadratic form (a,b,c).

`ct quadratic_form (const long a, const long b, const long c)`
initializes with the quadratic form (a,b,c).

`ct quadratic_form (const qi_class & A)`
initializes with the quadratic form associated with the reduced ideal A.

`ct quadratic_form (const qi_class_real & A)`
initializes with the quadratic form associated with the reduced ideal A.

`ct quadratic_form (const quadratic_ideal & A)`
initializes with the quadratic form associated with the ideal A. If A has a fractional multiplier, it is simply ignored.

`ct quadratic_form (const quadratic_form & f)`
initializes a copy of f.

`dt ~quadratic_form ()`

Assignments

Let f be of type `quadratic_form`. The operator = is overloaded. For efficiency reasons, the following functions are also implemented:

`void f.assign_zero ()`
$f \leftarrow (0,0,0)$.

`void f.assign_one ()`
f is set to the unit form (first coefficient is 1) with the same discriminant as f. If the discriminant of f is 0, then the discriminant of the most recently referenced `quadratic_order` is used.

`void f.assign_one (const bigint & Δ)`
if Δ is a quadratic discriminant, f is set to the unit form (first coefficient is 1) of discriminant Δ, otherwise the `lidia_error_handler` will be evoked.

`void f.assign (const bigint & a, const bigint & b, const bigint & c)`
sets f to the form (a,b,c).

`void f.assign (const long a, const long b, const bigint & c)`
sets f to the form (a,b,c).

`void f.assign (const qi_class & A)`
sets f to the quadratic form associated with the reduced ideal A.

`void f.assign (const qi_class_real & A)`
sets f to the quadratic form associated with the reduced ideal A.

`void f.assign (const quadratic_ideal & A)`
sets f to the quadratic form associated with the ideal A. If A has a fractional multiplier, it is simply ignored.

`void f.assign (const quadratic_form & g)`
$f \leftarrow g$.

Access Methods

Let f be of type `quadratic_form`.

`bigint f.get_a () const`
returns the value of a.

`bigint` f`.get_b () const`
returns the value of b.

`bigint` f`.get_c () const`
returns the value of c.

`bigint` f`.discriminant () const`
returns the discriminant of f.

`quadratic_order & ` f`.which_order () const`
returns a reference to the `quadratic_order` of the same discriminant as f, or the empty `quadratic_order` (discriminant 0) if f is irregular.

`quadratic_order & which_order (const quadratic_form & ` f`)`
returns a reference to the `quadratic_order` of the same discriminant as f, or the empty `quadratic_order` (discriminant 0) if f is irregular.

Arithmetical Operations

Let f be of type `quadratic_form`. The following operators are overloaded and can be used in exactly the same way as in the programming language C++.

```
(unary) -
(binary) *, /
(binary with assignment) *=, /=
```

Note: By $-f$ and f^{-1} we denote the conjugate of f, by $f \cdot g$ we denote the composition of f and g, and by f/g we denote $f \cdot g^{-1}$.

To avoid copying, these operations can also be performed by the following functions:

`void compose (quadratic_form & ` f`, const quadratic_form & ` g_1`, const quadratic_form & ` g_2`)`
$f \leftarrow g_1 \cdot g_2$. If g_1 and g_2 do not have the same discriminant, the `lidia_error_handler` will be evoked.

`void compose_reduce (quadratic_form & ` f`, const quadratic_form & ` g_1`,`
`const quadratic_form & ` g_2`)`
f is set to a reduced form equivalent to $g_1 \cdot g_2$. If g_1 and g_2 do not have the same discriminant, the `lidia_error_handler` will be evoked.

`void nucomp (quadratic_form & ` f`, const quadratic_form & ` g_1`, const quadratic_form & ` g_1`)`
f is set to a reduced form equivalent to $g_1 \cdot g_2$ using the NUCOMP algorithm of Shanks [17]. Both g_1 and g_2 must be positive of negative definite, otherwise the `lidia_error_handler` will be evoked.

`void` f`.conjugate ()`
the coefficient b of f is negated.

`void get_conjugate (quadratic_form & ` f`, const quadratic_form & ` g`)`
f is set to g with the b coefficient negated.

`quadratic_form get_conjugate (quadratic_form & ` f`)`
f is returned with its b coefficient negated.

`void divide (quadratic_form & f, const quadratic_form & g1, const quadratic_form & g2)`

$f \leftarrow g_1 \cdot g_2^{-1}$. If g_1 and g_2 are not of the same quadratic order or if $g_2 = (0,0,0)$, the `lidia_error_handler` will be evoked.

`void divide_reduce (quadratic_form & f, const quadratic_form & g1, const quadratic_form & g2)`

f is set to a reduced form equivalent to $g_1 \cdot g_2^{-1}$. If g_1 and g_2 are not of the same quadratic order or if $g_2 = (0,0,0)$, the `lidia_error_handler` will be evoked.

`void square (quadratic_form & f, const quadratic_form & g)`

$f \leftarrow g^2$.

`void square_reduce (quadratic_form & f, const quadratic_form & g)`

f is set to a reduced form equivalent to g^2.

`void nudupl (quadratic_form & f, const quadratic_form & g)`

f is set to a reduced form equivalent to g^2 using the NUDUPL algorithm of Shanks [17]. The form g must be positive of negative definite, otherwise the `lidia_error_handler` will be evoked.

`void power (quadratic_form & f, const quadratic_form & g, const bigint & i)`

$f \leftarrow g^i$. If $i < 0$, h^i is computed where h is equal to the conjugate of g.

`void power_reduce (quadratic_form & f, const quadratic_form & g, const bigint & i)`

f is set to a reduced form equivalent to g^i. If $i < 0$, h^i is computed where h is equal to the conjugate of g.

`void nupower (quadratic_form & f, const quadratic_form & g, const bigint & i)`

f is set to a reduced form equivalent to g^i using the NUCOMP and NUDUPL algorithms of Shanks [17] to compute the intermediate products. If $i < 0$, h^i is computed where h is equal to the conjugate of g. The form g must be positive of negative definite, otherwise the `lidia_error_handler` will be evoked.

`void power (quadratic_form & f, const quadratic_form & g, const long i)`

`void power_reduce (quadratic_form & f, const quadratic_form & g, const long i)`

`void nupower (quadratic_form & f, const quadratic_form & g, const long i)`

Comparisons

The binary operators `<=`, `==`, `>=`, `!=`, `<`, `>` are overloaded and mean the usual lexicographic ordering of the triples (a,b,c). The unary operator `!` (comparison with $(0,0,0)$) is also overloaded.

Let f be an instance of type `quadratic_form`.

`bool f.is_zero () const`

returns `true` if $f = (0,0,0)$, `false` otherwise.

`bool f.is_one () const`

returns `true` if the a coefficient of f is 1, `false` otherwise.

`bool` f`.is_equal (const quadratic_form &` g`) const`

returns `true` if f and g are equal, `false` otherwise.

`int` f`.compare (const quadratic_form &` g`) const`

returns -1 if $f < g$, 0 if $f = g$, and 1 if $f > g$.

`bool` f`.abs_compare (const quadratic_form &` g`) const`

returns -1 if $f < g$, 0 if $f = g$, and 1 if $f > g$, where the coefficients are taken to be the absolute values of the actual coefficients.

Basic Methods and Functions

`operator qi_class () const`

cast operator which implicitly converts a `quadratic_form` to a `qi_class`. The current order of `qi_class` will be set to the order corresponding to the form.

`operator qi_class_real () const`

cast operator which implicitly converts a `quadratic_form` to a `qi_class`. The current order of `qi_class` and `qi_class_real` will be set to the order corresponding to the form.

`operator quadratic_ideal () const`

cast operator which implicitly converts a `quadratic_form` to a `quadratic_ideal`.

`void swap (quadratic_form &` f`, quadratic_form &` g`)`

exchanges the values of f and g.

High-Level Methods and Functions

Let f be of type `quadratic_form`. The operator `()` is overloaded, and can be used in place of the function `eval`.

`int` f`.definiteness () const`

returns 1 if f is positive definite, -1 if it is negative definite, 0 if it is indefinite, and 2 if it is irregular.

`bool` f`.is_pos_definite () const`

returns `true` if f is positive definite, `false` otherwise.

`bool` f`.is_pos_semidefinite () const`

returns `true` if f is positive semidefinite, `false` otherwise.

`bool` f`.is_indefinite () const`

returns `true` if f is indefinite and regular, `false` otherwise.

`bool` f`.is_neg_definite () const`

returns `true` if f is negative definite, `false` otherwise.

`bool` f`.is_neg_semidefinite () const`

returns `true` if f is negative semidefinite, `false` otherwise.

`bool` f`.is_regular () const`
returns `true` if f is regular, `false` otherwise.

`bigint` f`.content () const`
returns the content of f, the gcd of its coefficients.

`bool` f`.is_primitive () const`
returns `true` if f is primitive (content is one), `false` otherwise.

`bigint` f`.eval (const bigint &` x`, const bigint &` y`) const`
returns $f(x, y)$.

`void` f`.transform (const matrix_GL2Z &` U`)`
$f \leftarrow fU$.

`void` f`.transform (const bigint_matrix &` U`)`
$f \leftarrow fU$ (U is cast to a `matrix_GL2Z`).

`bool generate_prime_form (quadratic_form &` f`, const bigint &` p`, const bigint &` Δ`)`
attempts to set f to the prime form corresponding to p of discriminant Δ (first coefficient is p). If successful (the Kronecker symbol $\left(\frac{\Delta}{p}\right) \neq -1$), `true` is returned, otherwise `false` is returned. If Δ is not a quadratic discriminant, the `lidia_error_handler` will be evoked.

Normalization and Reduction

`bool` f`.is_normal ()`
returns `true` if f is normal, `false` otherwise.

`void` f`.normalize ()`
f is set to its normalization.

`void` f`.normalize (matrix_GL2Z &` U`)`
f is set to its normalization. U is set to the matrix product of U and T, where T is the transformation matrix which was used to effect the normalization.

`bool` f`.is_reduced ()`
returns `true` if f is reduced, `false` otherwise.

`void` f`.reduce ()`
If f is regular, then f is set to the first reduced form which is reached by successively applying the reduction operator ρ, unless f itself is already reduced. If f is irregular, then f is set to the reduced form in its proper equivalence class.

`void` f`.reduce (matrix_GL2Z &` U`)`
If f is regular, then f is set to the first reduced form which is reached by successively applying the reduction operator ρ, unless f itself is already reduced. If f is irregular, then f is set to the reduced form in its proper equivalence class. U is multiplied by the transformation matrix used to effect the reduction.

`void` f`.rho ()`

f is set to the result of applying the reduction operator ρ to f.

`void` f`.rho (matrix_GL2Z &` U`)`

f is set to the result of applying the reduction operator ρ to f. U is multiplied by the transformation matrix used to effect the operation.

`void` f`.inverse_rho ()`

f is set to the result of applying the inverse reduction operator to f.

`void` f`.inverse_rho (matrix_GL2Z &` U`)`

f is set to the result of applying the inverse reduction operator to f. U is multiplied by the transformation matrix used to effect the operation.

Equivalence and principality testing

The parameter U is optional in all equivalence and principality functions.

`bool` f`.is_equivalent (const quadratic_form &` g`, matrix_GL2Z &` U`) const`

returns `true` if f and g are equivalent, `false` otherwise. U is multiplied by the transformation matrix which transforms g to f if f and g are equivalent otherwise U remains unchanged.

`bool` f`.is_prop_equivalent (const quadratic_form &` g`, matrix_GL2Z &` U`) const`

returns `true` if f and g are properly equivalent, `false` otherwise. U is multiplied by the transformation matrix which transforms f to g if f and g are properly equivalent, otherwise U remains unchanged.

`bool` f`.is_principal (matrix_GL2Z &` U`) const`

returns `true` if f is principal, `false` otherwise. U is multiplied by the transformation matrix which transforms the unit form to f if f is principal, otherwise U remains unchanged.

Orders, discrete logarithms, and subgroup structures

`bigint` f`.order_in_CL ()`

returns the order of f in the class group of forms of the same discriminant, i.e., the least positive integer x such that f^x is equivalent to the unit form. If f is not primitive, 0 is returned.

`bool` f`.DL (const quadratic_form &` g`, bigint &` x`) const`

returns `true` if f belongs to the cyclic subgroup generated by g, `false` otherwise. If so, x is set to the discrete logarithm of f to the base g, i.e., the least non-negative integer x such that g^x is equivalent to f. If not, x is set to the order of g. If either f or g is not primitive, `false` is returned and x is set to 0.

`base_vector< bigint > subgroup (base_vector< quadratic_form > &` G`)`

returns the vector of elementary divisors representing the structure of the subgroup generated by the classes corresponding to the primitive forms contained in G.

`bigfloat` f`.regulator ()`

returns an approximation of the regulator of the quadratic order of the same discriminant as f. If f is not indefinite, 1 is returned.

`bigint` f`.class_number ()`
returns the number of equivalence classes of forms with the same discriminant as f.

`base_vector< bigint >` f`.class_group ()`
returns the vector of elementary divisors representing the structure of the class group of forms with the same discriminant as f.

`base_vector< quadratic_form > compute_class_group (const bigint &` Δ`)`
returns the vector of all reduced representatives of the class group of discriminant Δ.

`matrix_GL2Z` f`.fundamental_automorphism ()`
returns the fundamental automorphism of the quadratic order of the same discriminant as f.

Representations

`bool` f`.representations (sort_vector < pair < bigint, bigint > > &` *Reps*`,`
`const bigint &` N`)`
returns `true` if there exists at least one representation of the integer N by the form f, `false` otherwise. If exactly r representations exist, *Reps* contains r ordered pairs corresponding to these representations. If infinitely many representations exist, *Reps* will contain the smallest inequivalent representations.

Input/Output

The operators `<<` and `>>` are overloaded. Input is as follows:

```
(a b c)
```

and corresponds to the form $f = (a, b, c)$.

See also

`matrix_GL2Z`, `quadratic_order`, `qi_class`, `qi_class_real`, `quadratic_ideal`

Notes

The order, discrete logarithm, subgroup, regulator, class number, and class group algorithms are not yet implemented for irregular forms.

Examples

```
#include <LiDIA/quadratic_order.h>

int main()
{
    // This program reads a quadratic form f and transforms
    // it to an equivalent reduced form.  It keeps track of
    // the transformation in a matrix U of GL(2, Z) and
```

```
    // transforms the reduced form with the inverse of
    // this matrix.

    quadratic_form f,g;

    matrix_GL2Z U;

    cout << "Please enter a form! Example (9 5 3):  ";

    cin >> f;
    g = f;

    f.reduce(U);

    cout << "An equivalent reduced form is:  " << f << endl;

    U.invert();
    f.transform(U);

    cout << "Transforming f with the inverse transformation yields:  " << f
         << endl

    if (f == g)
        cout << "The result is correct!" << endl;
    else
        cout << "The result is incorrect" << endl;

    return 0;
}
```

Example:

```
Please enter a form! Example (9, 5, 3):  (243, -371, 145)
An equivalent reduced form is:  (17, -13, 51)
Transforming f with the inverse transformation yields:  (243, -371, 145)
The result is correct!
```

For further examples please refer to `LiDIA/src/packages/quadratic_order/quadratic_form_appl.cc`

Author

Friedrich Eisenbrand, Michael J. Jacobson, Jr., Thomas Papanikolaou

quadratic_order

Name

`quadratic_order` ..... class describing quadratic orders

Abstract

`quadratic_order` is a class for representing quadratic orders and computing many related invariants and functions, such as the ideal class number, regulator, L-function, Littlewood indices, etc ...

Description

A `quadratic_order` $\mathcal{O}$ is represented simply by a `bigint` Δ, which is a quadratic discriminant, i.e., a non-square integer congruent to 0 or 1 modulo 4. Unlike the case of higher degree orders (see the description of `order`), we do not require a multiplication table to perform arithmetic with elements in the order since we can represent any element in $\mathcal{O}$ as $a + b(\Delta + \sqrt{\Delta})/2$ where $a, b \in \mathbb{Z}$.

Several invariants related to a specific quadratic order are stored along with the discriminant, since some of them can be very time-consuming to compute, and knowledge of them often simplifies other computations. We store the following invariants once they have been computed:

- factorization of Δ (`rational_factorization`)
- ideal class number (`bigint`)
- approximation of the regulator (`bigfloat`)
- elementary divisors of the ideal class group (`base_vector< bigint >`)
- generators of each cyclic subgroup in the decomposition of the ideal class group (`base_vector` of `qi_class`)
- approximation of $L(1, \chi)$ (`bigfloat`)
- approximation of $C(\Delta)$ (`bigfloat`)

Constructors/Destructor

`ct quadratic_order ()`
initializes an empty quadratic order ($\Delta = 0$)).

`ct quadratic_order (long` Δ`)`
if Δ is a quadratic discriminant, initializes a `quadratic_order` of discriminant Δ, otherwise an empty quadratic order ($\Delta = 0$) is initialized.

`ct quadratic_order (const bigint & Δ)`

if Δ is a quadratic discriminant, initializes a `quadratic_order` of discriminant Δ, otherwise an empty quadratic order ($\Delta = 0$) is initialized.

`ct quadratic_order (const quadratic_order & 𝒪)`

initializes a copy of $\mathcal{O}$.

`dt ~quadratic_order ()`

Initialization

Let $\mathcal{O}$ be of type `quadratic_order`.

`static void quadratic_order::verbose (int` *state*`)`

sets the amount of information which is printed during computations. If *state* = 1, then the elapsed CPU time of some computations is printed. If *state* = 2, then extra run-time data is printed during the execution of the subexponential class group algorithm and verifications. If *state* = 0, no extra information is printed. The default is *state* = 0.

`static void quadratic_order::verification (int` *level*`)`

sets the level of post-computation verification performed. If *level* = 0 no extra verification is performed, otherwise, the following levels of verification are used:

- *level* = 1 — regulator is verified, orders of generators of the class group are verified
- *level* = 2 — same as 1, plus all prime ideals not in the factor base with norm less than Bach's bound are factored over the factor base (only for the subexponential algorithm)
- *level* = 3 — same as 2, plus all prime ideals with norm less than Bach's bound are represented over a system of generators (only for the subexponential algorithm)

Note that level 2 is sufficient to provide a conditional verification of the output of the subexponential algorithm under the Extended Riemann Hypothesis (ERH). The success of the verification will be output only if the verbose mode is set to a non-zero value, but any failures are always output. The default is *level* = 0.

Assignments

Let $\mathcal{O}$ be of type `quadratic_order`. The operator `=` is overloaded. For efficiency reasons, the following functions are also implemented:

`bool 𝒪.assign (long Δ)`

if D is a quadratic discriminant, $\mathcal{O}$ is set to the quadratic order of discriminant D and `true` is returned, otherwise $\mathcal{O}$ is set to an empty quadratic order ($\Delta = 0$) and `false` is returned.

`bool 𝒪.assign (const bigint & Δ)`

if D is a quadratic discriminant, $\mathcal{O}$ is set to the quadratic order of discriminant D and `true` is returned, otherwise $\mathcal{O}$ is set to an empty quadratic order ($\Delta = 0$) and `false` is returned.

`void 𝒪.assign (const quadratic_order & 𝒪₂)`

$\mathcal{O} \leftarrow \mathcal{O}_2$.

Access Methods

Let $\mathcal{O}$ be of type `quadratic_order`.

`bigint` $\mathcal{O}$`.discriminant () const`
returns the discriminant Δ of $\mathcal{O}$.

Comparisons

The binary operators `==`, `!=`, `<=`, `<` (true subset), `>=`, `>` and the unary operator `!` (comparison with (0)) are overloaded and can be used in exactly the same way as in the programming language C++. Let $\mathcal{O}$ be of type `quadratic_order`.

`bool` $\mathcal{O}$`.is_zero () const`
returns `true` if the discriminant is zero, `false` otherwise.

`bool` $\mathcal{O}$`.is_equal (const quadratic_order &` $\mathcal{O}_2$`) const`
returns `true` if $\mathcal{O} = \mathcal{O}_2$, `false` otherwise.

`bool` $\mathcal{O}$`.is_subset (quadratic_order &` $\mathcal{O}_2$`)`
returns `true` if $\mathcal{O}$ is a subset of $\mathcal{O}_2$, `false` otherwise. The discriminant is factored if necessary.

`bool` $\mathcal{O}$`.is_proper_subset (quadratic_order &` $\mathcal{O}_2$`)`
returns `true` if $\mathcal{O}$ is a proper subset of $\mathcal{O}_2$, `false` otherwise. The discriminant is factored if necessary.

Basic Methods and Functions

`bool is_quadratic_discriminant (const long` Δ`)`
returns `true` if D is a quadratic discriminant, `false` otherwise.

`bool is_quadratic_discriminant (const bigint &` Δ`)`
returns `true` if D is a quadratic discriminant, `false` otherwise.

`bool` $\mathcal{O}$`.is_imaginary () const`
returns `true` if $\mathcal{O}$ is an imaginary quadratic order (negative discriminant), `false` otherwise.

`bool` $\mathcal{O}$`.is_real () const`
returns `true` if $\mathcal{O}$ is a real quadratic order (positive discriminant), `false` otherwise.

`void swap (quadratic_order &` A`, quadratic_order &` B`)`
exchanges A and B.

High-Level Methods and Functions

Let $\mathcal{O}$ be of type `quadratic_order`.

`int` $\mathcal{O}$`.bach_bound ()`
returns a constant such that the prime ideals of norm less than this constant generate the class group (under the ERH). This function assumes that $\mathcal{O}$ is not maximal.

`int` $\mathcal{O}$`.bach_bound (bool` *is_max*`)`

returns a constant such that the prime ideals of norm less than this constant generate the class group (under the ERH). If the parameter is `true`, $\mathcal{O}$ will be assumed to be maximal, otherwise non-maximal.

`bigint` $\mathcal{O}$`.conductor ()`

returns the conductor of $\mathcal{O}$, i.e., the integer f such that $\Delta = f^2\Delta_0$ where Δ_0 is a fundamental discriminant (the discriminant of a maximal order). The factorization of the discriminant is computed if it has not been already.

`bool` $\mathcal{O}$`.is_maximal ()`

returns `true` if $\mathcal{O}$ is a maximal order, `false` otherwise. The factorization of the discriminant is computed if it has not been already.

`quadratic_order` $\mathcal{O}$`.maximize ()`

returns the maximal order in which $\mathcal{O}$ is contained. The factorization of the discriminant is computed if it has not been already.

L-functions, Littlewood indices, C-function

The function $L(s,\chi) = \sum_{n=1}^{\infty} \frac{\chi(n)}{n^s}$, where χ is taken to be the Kronecker symbol, is intimately related to computations in class groups of quadratic orders, especially at $s = 1$, due to the analytic class number formula of Dirichlet.

The Littlewood indices [43, 58] are values which test the accuracy of Littlewood's bounds on $L(1,\chi)$ and a related function $L_\Delta(1) = (2 - \left(\frac{\Delta}{2}\right))/2\, L(1,\chi)$ defined by Shanks [58]. Under the ERH, we expect the lower Littlewood indices to be greater than 1 for all discriminants Δ (with a few exceptions of very small discriminants) and that they should come close to 1 for discriminants with exceptionally small values of $L(1,\chi)$ or $L_D(1)$. Similarly, we expect the upper Littlewood indices to be less than 1 and to approach 1 for discriminants with exceptionally large values of $L(1,\chi)$ or $L_\Delta(1)$.

The function $C(\Delta) = \prod_{p\geq 3} 1 - \left(\frac{\Delta}{p}\right)/(p-1)$ provides an indication of whether a related quadratic polynomial will have an asymptotically large number of prime values, based on a conjecture of Hardy and Littlewood [29, 26]. If Δ is odd, then for $A = (1-\Delta)/4$ the polynomial $x^2 + x + A$ will have a large asymptotic density of prime values when $C(\Delta)$ is large. If Δ is even, then for $A = \Delta/4$ the corresponding polynomial is $x^2 + A$.

`int kronecker (const bigint &` D`, const bigint &` p`)`

computes the value of the Kronecker symbol $\left(\frac{D}{p}\right)$, where p is prime.

`long` $\mathcal{O}$`.generate_optimal_Q ()`

generates the optimal value of Q for the function `estimate_L1` to compute a sufficiently accurate estimate from which a value h^* can be computed satisfying $h^* < h < 2h^*$ for imaginary orders and $h^* < hR < 2h^*$ for real orders.

`long` $\mathcal{O}$`.get_optimal_Q ()`

returns, from a precomputed list, a value of Q that is sufficiently large to satisfy the conditions given above.

`long` $\mathcal{O}$`.generate_optimal_Q_cnum (const bigfloat &` h_2`, const bigfloat &` t`)`

generates the optimal value of Q for the function `estimate_L1` to compute a sufficiently accurate estimate to prove that $h = h_2$ when the fractional part of the approximation of h is less that t. See [46] for more details.

`long` $\mathcal{O}$`.get_optimal_Q_cnum ()`

returns, from a precomputed list, a value of Q that is sufficiently large to satisfy the conditions given above.

`bigfloat` $\mathcal{O}$`.estimate_L (const int` s`, const long` Q`)`

computes a truncated product approximation of $L(s, \chi)$ using primes less than Q.

`bigfloat` $\mathcal{O}$`.estimate_L1 (const long` Q`)`

computes an approximation of $L(1, \chi)$ using Bach's averaging method [3] with primes less than $2Q$.

`bigfloat` $\mathcal{O}$`.estimate_L1_error (const long` Q`)`

computes an estimate of the error (conditional on EHR) in the approximation of $L(1, \chi)$ using Bach's method [3] with primes less than $2Q$.

`bigfloat` $\mathcal{O}$`.Lfunction ()`

computes an approximation of $L(1, \chi)$ via the analytic class number formula. The class number and regulator are computed if they have not been already.

`bigfloat` $\mathcal{O}$`.LDfunction ()`

returns an approximation of $L_\Delta(1)$. The class number and regulator are computed if they have not been already.

`bigfloat` $\mathcal{O}$`.LLI ()`

returns an approximation of the lower Littlewood index related to a lower bound on $L(1, \chi)$ due to Littlewood. The class number and regulator are computed if they have not been already.

`bigfloat` $\mathcal{O}$`.LLI_D ()`

returns an approximation of the lower Littlewood index related to a lower bound on $L_\Delta(1)$ due to Shanks. The class number and regulator are computed if they have not been already.

`bigfloat` $\mathcal{O}$`.ULI ()`

returns an approximation of the upper Littlewood index related to an upper bound on $L(1, \chi)$ due to Littlewood. The class number and regulator are computed if they have not been already.

`bigfloat` $\mathcal{O}$`.ULI_D ()`

returns an approximation of the upper Littlewood index related to an upper bound on $L_\Delta(1)$ due to Shanks. The class number and regulator are computed if they have not been already.

`bigfloat` $\mathcal{O}$`.Cfunction ()`

returns an approximation of the function $C(\Delta)$ accurate to 8 significant digits. The class number, regulator, and the prime factorization of Δ are computed if they have not been already.

Regulator, class number, and class group

The regulator is defined as the natural logarithm of the fundamental unit and the class number is the order of the group of ideal equivalence classes. We denote the regulator by R, the class group by Cl and the class number by h. By the structure of the class group, we mean the unique positive integers $m_1, \ldots, m_k$ with $m_1 > 1$, $m_i \mid m_{i+1}$ for $1 \leq i \leq k$ such that there is an isomorphism $\phi : \mathrm{Cl} \to \mathbb{Z}/m_1\mathbb{Z} \times \cdots \times \mathbb{Z}/m_k\mathbb{Z}$. The integers m_i are called the elementary divisors of the class group.

The following general functions are provided, which intelligently select an appropriate algorithm based on the size of the discriminant.

`bigfloat` $\mathcal{O}$`.regulator ()`

returns an approximation of the regulator of $\mathcal{O}$. If the regulator has not already been computed, either `regulator_BJT`, `regulator_shanks`, or `regulator_subexp` is used, depending on the size of the discriminant. If $\mathcal{O}$ is imaginary, the regulator is set to 1.

`bigint` $\mathcal{O}$`.class_number ()`

returns the ideal class number of $\mathcal{O}$. If the class number has not already been computed, either `class_number_shanks` or `class_group_subexp` is used, depending on the size of the discriminant.

`base_vector< bigint >` $\mathcal{O}$`.class_group ()`

returns the vector of invariants representing the structure of the ideal class group of $\mathcal{O}$. If the class group has not already been computed, either `class_group_BJT`, `class_group_shanks`, `class_group_randexp`, `class_group_mpqs`, or `class_group_siqs` depending on the size of the discriminant.

The following functions all make use of a specific algorithm.

`bigfloat` $\mathcal{O}$`.regulator_BJT ()`

computes an unconditionally correct approximation of the regulator of $\mathcal{O}$ using a variation of the algorithm of [5] which has unconditional complexity $O(\sqrt{R})$. If $\mathcal{O}$ is not real, the regulator is set to 1. If the regulator has already been computed, the function simply returns this value.

`bigfloat` $\mathcal{O}$`.regulator_shanks ()`

computes an unconditionally correct approximation of the regulator of $\mathcal{O}$ using the algorithm of [37] based on the baby-step giant-step method in [46], which has complexity $O(\Delta^{1/5})$ under the ERH. If $\mathcal{O}$ is not real, the regulator is set to 1. If the regulator has already been computed, the function simply returns this value.

`bool` $\mathcal{O}$`.verify_regulator ()`

returns true if the regulator is correct, as determined by a polynomial-time verification.

`bigint` $\mathcal{O}$`.class_number_shanks ()`

computes the ideal class number of $\mathcal{O}$ using the algorithm of [26] based on the ideas of [41] for imaginary quadratic orders, and the algorithm of [37] based on the ideas in [46] for real quadratic orders. Both algorithms have complexity $O(|\Delta|^(1/5))$, and the correctness and the complexity are both conditional on the truth of the ERH. If the class number has already been computed, the function simply returns this value.

`base_vector< bigint >` $\mathcal{O}$`.class_group_BJT ()`

computes the structure of the ideal class group of $\mathcal{O}$ using the method of [12] for imaginary orders and [34] for real orders, which have complexity $O(\sqrt{h})$ and $O(\sqrt{hR})$ respectively. The correctness of both algorithms is conditional on the truth of the ERH but the complexity results are not. If the class group has already been computed, the function simply returns this value.

`base_vector< bigint >` $\mathcal{O}$`.class_group_shanks ()`

computes the structure of the ideal class group of $\mathcal{O}$ using variations of the methods of [56] which have complexity $O(|\Delta|^(1/4))$. The correctness and complexity of both algorithms are conditional on the truth of the ERH. If the class group has already been computed, the function simply returns this value.

`base_vector< bigint > `$\mathcal{O}$`.class_group_randexp ()`

computes the structure of the ideal class group of $\mathcal{O}$ using a practical variation of the sub-exponential method of Hafner and McCurley [28] due to Düllmann [24] for imaginary orders and Cohen, Diaz y Diaz, and Olivier [18] for real orders, as presented in [36]. Note that if $\mathcal{O}$ is real, this function also computes the regulator. If the class group has already been computed, the function simply returns this value. Turning the verification off with `quadratic_order::verification(0)` will result in much faster execution of this algorithm, but the result is no longer certifiably correct under the ERH since a smaller factor-base is used than is theoretically necessary. However, as experimental results show [34], it is extremely unlikely that the computed result will be false.

`base_vector< bigint > `$\mathcal{O}$`.class_group_mpqs ()`

computes the structure of the ideal class group of $\mathcal{O}$ using the algorithm of [35] as presented in [36], in which the relation generation strategy is based on the MPQS factoring algorithm. Note that if $\mathcal{O}$ is real, this function also computes the regulator. If the class group has already been computed, the function simply returns this value. Turning the verification off with `quadratic_order::verification(0)` will result in much faster execution of this algorithm, but the result is no longer certifiably correct under the ERH since a smaller factor-base is used than is theoretically necessary. However, as experimental results show [34], it is extremely unlikely that the computed result will be false.

`base_vector< bigint > `$\mathcal{O}$`.class_group_siqs ()`

computes the structure of the ideal class group of $\mathcal{O}$ using an algorithm from [36], in which the relation generation strategy is based on the self-initializing MPQS factoring algorithm. Note that if $\mathcal{O}$ is real, this function also computes the regulator. If the class group has already been computed, the function simply returns this value. Turning the verification off with `quadratic_order::verification(0)` will result in much faster execution of this algorithm, but the result is no longer certifiably correct under the ERH since a smaller factor-base is used than is theoretically necessary. However, as experimental results show [34], it is extremely unlikely that the computed result will be false.

`base_vector< bigint > `$\mathcal{O}$`.class_group_lp ()`

computes the structure of the ideal class group of $\mathcal{O}$ using an algorithm from [36], in which the relation generation strategy is based on the self-initializing MPQS factoring algorithm with large prime variant. Note that if $\mathcal{O}$ is real, this function also computes the regulator. If the class group has already been computed, the function simply returns this value. Turning the verification off with `quadratic_order::verification(0)` will result in much faster execution of this algorithm, but the result is no longer certifiably correct under the ERH since a smaller factor-base is used than is theoretically necessary. However, as experimental results show [34], it is extremely unlikely that the computed result will be false.

`bool `$\mathcal{O}$`.verify_class_group (int level)`

returns true if the class group is correct as determined by a polynomial-time verification, using the given verification level

`bool `$\mathcal{O}$`.verify_class_group ()`

returns true if the class group is correct as determined by a polynomial-time verification, using the global verification level.

Additional functions

`rational_factorization `$\mathcal{O}$`.factor_h ()`

returns a `rational_factorization` of the class number. The class number is computed if it has not been already.

`bigint` $\mathcal{O}$`.exponent ()`

returns the exponent of the class group, i.e., the largest cyclic factor. The class group is computed if it has not been already.

`int` $\mathcal{O}$`.p_rank (const bigint &` p`)`

returns the p-rank of the ideal class group. The class group is computed if it has not been already.

`base_vector< qi_class >` $\mathcal{O}$`.generators ()`

returns a vector of generators of the cyclic subgroups of the ideal class group of $\mathcal{O}$. The class group is computed if it has not been already.

`rational_factorization` $\mathcal{O}$`.factor_discriminant ()`

returns the prime factorization of the discriminant of $\mathcal{O}$ using the 2-Sylow subgroup of the class group. The class group and a set of generators are computed if they have not been already. At present, this function is only implemented for imaginary orders. If $\mathcal{O}$ is real, the discriminant is simply factored with the `rational_factorization` function `factor`.

`math_vector < bigint >` $\mathcal{O}$`.represent_over_generators (qi_class &` A`)`

returns an exponent vector corresponding to the representation of A over the system of generators of the class group.

Input/Output

Let $\mathcal{O}$ be of type `quadratic_order`. The `istream` operator `>>` and the `ostream` operator `<<` are overloaded. Input consists of reading a `bigint` value for the discriminant. Output of a `quadratic_order` has the following format:

```
Quadratic Order:
   Delta = (discriminant of QO) (decimal digits)
         = (prime factorization of the discriminant of QO)
   R = (regulator)
   h = (class number)
   CL = (vector of class group invariants)
   generators = (vector of generators of the cyclic subgroups)
   L(1,X) = (L function)
   C(Delta) = (C function)
```

The discriminant is always displayed, while the other invariants are displayed only if they have been computed.

See also

`qi_class`, `qi_class_real`, `order`

Examples

```
#include <LiDIA/quadratic_order.h>

int main()
{
    bigint D;
```

```
    quadratic_order QO;

    do {
        cout << "Please enter a quadratic discriminant: "; cin >> D;
    } while (!QO.assign(D));
    cout << endl;

    QO.class_group();
    QO.generators();
    QO.Lfunction();

    cout << QO;

    cout << "   ULI = " << QO.ULI() << endl;
    cout << "   LLI = " << QO.LLI() << endl;

    return 0;
}
```

Example:

```
Please enter a quadratic discriminant: -501510767

Quadratic Order:
   Delta = -501510767 (9)
   h = 18522
   CL = [ 7 7 378 ]
   generators = [ (3286, 1657) (1144, 439) (2934, 61) ]
   L(1,X) = 2.5983498285787
   ULI = 0.243356600232444
   LLI = 16.865670804095
```

For further examples please refer to `LiDIA/src/packages/quadratic_order/quadratic_order_appl.cc`.

Author

Michael J. Jacobson, Jr.

qi_class

Name

`qi_class` ..... ideal equivalence classes of quadratic orders

Abstract

`qi_class` is a class for representing and computing with the ideal equivalence classes of quadratic orders. It supports basic operations like multiplication as well as more complex operations such as computing the order of an equivalence class in the class group.

Description

A `qi_class` is represented by a reduced, invertible quadratic ideal given in standard representation, i.e., an ordered pair of `bigint`s (a, b) representing the $\mathbb{Z}$-module $[a\mathbb{Z} + (b + \sqrt{\Delta})/2\,\mathbb{Z}]$, where Δ is the discriminant of a quadratic order. If the quadratic order is imaginary, then this ideal is the unique representative of its equivalence class. If not, then it is one of a finite set of reduced representatives.

This class was designed to be used for computations in the ideal equivalence class group of quadratic orders; the class `quadratic_ideal` describes general fractional quadratic ideals, and can be used for more general applications.

The main difference between this class and `quadratic_ideal` is that a pointer to a `quadratic_order` is not stored with every instance of the class. Instead, there is a global quadratic order to which every current `qi_class` belongs. This strategy is similar to that used for the global modulus of the `bigmod` class and is logical here, since most computations using this class will be related to only one quadratic order at a time. Of course, if the global order is changed, all the existing instances of `qi_class` are invalidated.

Constructors/Destructor

`ct qi_class ()`

`ct qi_class (const bigint & `a_2`, const bigint & `b_2`)`

if $[a_2\mathbb{Z} + (b_2 + \sqrt{\Delta})/2\,\mathbb{Z}]$ is an ideal of the current quadratic order, A is set to a reduced representative of its equivalence class. If not, or if the resulting reduced representative is not invertible, the ideal will be set to the zero ideal.

`ct qi_class (const long & `a_2`, const long & `b_2`)`

if $[a_2\mathbb{Z} + (b_2 + \sqrt{\Delta})/2\,\mathbb{Z}]$ is an ideal of the current quadratic order, A is set to a reduced representative of its equivalence class, If not, or if the resulting reduced representative is not invertible, the ideal will be set to the zero ideal.

`ct qi_class (const quadratic_form & f)`

if f is regular and primitive, initializes with a reduced representative of the equivalence class to which the positively oriented ideal corresponding to f belongs, otherwise the `qi_class` will be initialized to zero. Note that the current order will be set to f`.which_order()`.

`ct qi_class (const quadratic_ideal & A)`

if A is invertible, initializes with a reduced representative of the equivalence class of A, otherwise the `qi_class` will be initialized to zero. Note that the current order will be set to A`.which_order()`.

`ct qi_class (const qi_class_real & A)`

initializes a copy of A

`ct qi_class (const qi_class & A)`

initializes a copy of A

`dt ~qi_class ()`

Initialization

Before any arithmetic is done with a `qi_class`, the current quadratic order must be initialized. If this order is real, it is also valid for all instances of `qi_class_real`.

`static void qi_class::set_current_order (quadratic_order & O)`

sets the current quadratic order to $\mathcal{O}$.

`static void qi_class::verbose (int` *state*`)`

sets the amount of information which is printed during computations. If *state* = 1, then the elapsed CPU time of some computations is printed. If *state* = 2, then extra information will be output during verifications. If *state* = 0, no extra information is printed. The default is *state* = 0.

`static void qi_class::verification (int` *level*`)`

sets the *level* of post-computation verification performed. If *level* = 0 no extra verification is performed. If *level* = 1, then verification of the order and discrete logarithm computations will be performed. The success of the verification will be output only if the verbose mode is set to a non-zero value, but any failures are always output. The default is *level* = 0.

Assignments

Let A be of type `qi_class`. The operator `=` is overloaded. For efficiency reasons, the following functions are also implemented:

`void A.assign_zero ()`

$A \leftarrow (0)$, the zero ideal.

`void A.assign_one ()`

$A \leftarrow (1)$, the whole order. If no current order has been defined, the `lidia_error_handler` will be evoked.

`void` A`.assign_principal (const bigint &` x`, const bigint &` y`)`

sets A to a reduced representative of the principal ideal generated by $x+y(\Delta+\sqrt{\Delta})/2$, where Δ is the discriminant of the current quadratic order. If no current order has been defined, the `lidia_error_handler` will be evoked.

`bool` A`.assign (const bigint &` a_2`, const bigint &` b_2`)`

if $[a_2\mathbb{Z} + (b_2 + \sqrt{\Delta})/2\,\mathbb{Z}]$ is an ideal of the current quadratic order, A is set to a reduced representative of its equivalence class and `true` is returned. If not, or if the resulting reduced representative is not invertible, the ideal is set to zero ans `false` is returned. If no current order has been defined, the `lidia_error_handler` will be evoked.

`bool` A`.assign (const long` a_2`, const long` b_2`)`

if $[a_2\mathbb{Z} + (b_2 + \sqrt{\Delta})/2\,\mathbb{Z}]$ is an ideal of the current quadratic order, A is set to a reduced representative of its equivalence class and `true` is returned. If not, or if the resulting reduced representative is not invertible, the ideal is set to zero ans `false` is returned. If no current order has been defined, the `lidia_error_handler` will be evoked.

`void` A`.assign (const quadratic_form &` f`)`

if f is regular and primitive, sets A to a reduced representative of the equivalence class to which the positively oriented ideal corresponding to f belongs, otherwise the `lidia_error_handler` will be evoked. Note that the current order will be set to f`.which_order()`.

`void` A`.assign (const quadratic_ideal &` B`)`

if B is invertible, sets A to a reduced representative of the equivalence class of B, otherwise the `lidia_error_handler` will be evoked. Note that the current order will be set to `B.which_order()`.

`void` A`.assign (const qi_class_real &` B`)`

$A \leftarrow B$.

`void` A`.assign (const qi_class &` B`)`

$A \leftarrow B$.

Access Methods

Let A be of type `qi_class`.

`bigint` A`.get_a () const`

returns the coefficient a in the representation $A = [a\mathbb{Z} + (b + \sqrt{\Delta})/2\,\mathbb{Z}]$, where Δ is the discriminant of the current quadratic order.

`bigint` A`.get_b () const`

returns the coefficient b in the representation $A = [a\mathbb{Z} + (b + \sqrt{\Delta})/2\,\mathbb{Z}]$, where Δ is the discriminant of the current quadratic order.

`bigint` A`.get_c () const`

returns the value $c = (b^2 - \Delta)/(4a)$ corresponding to the representation $A = [a\mathbb{Z} + (b + \sqrt{\Delta})/2\,\mathbb{Z}]$, where Δ is the discriminant of the current quadratic order.

`static bigint` A`.discriminant () const`

returns the discriminant of the current quadratic order.

`static quadratic_order & A.get_current_order () const`

returns a reference to the current quadratic order.

Arithmetical Operations

Let A be of type `qi_class`. The following operators are overloaded and can be used in exactly the same way as in the programming language C++.

```
(unary) -
(binary) *, /
(binary with assignment) *=, /=
```

Note: By $-A$ and A^{-1} we denote the inverse of A and by A/B we denote $A \cdot B^{-1}$.

To avoid copying, these operations can also be performed by the following functions:

`void multiply (qi_class & C, const qi_class & A, const qi_class & B)`

C is set to a reduced representative of $A \cdot B$. The appropriate function `multiply_imag` or `multiply_real` is used, depending on whether the current order is imaginary or real.

`void multiply_imag (qi_class & C, const qi_class & A, const qi_class & B)`

C is set to a reduced representative of $A \cdot B$. In the interest of efficiency, no checking is done to ensure that the current order is actually imaginary.

`void nucomp (qi_class & C, const qi_class & A, const qi_class & B)`

C is set to a reduced representative of $A \cdot B$ using the NUCOMP algorithm of Shanks [17]. In the interest of efficiency, no checking is done to ensure that the current order is actually imaginary.

`void multiply_real (qi_class & C, const qi_class & A, const qi_class & B)`

C is set to a reduced representative of $A \cdot B$. In the interest of efficiency, no checking is done to ensure that the current order is actually real.

`void A.invert ()`

$A \leftarrow A^{-1}$.

`void inverse (qi_class & A, const qi_class & B)`

$A \leftarrow B^{-1}$.

`qi_class inverse (qi_class & A)`

A^{-1} is returned.

`void divide (qi_class & C, const qi_class & A, const qi_class & B)`

C is set to a reduced representative of $A \cdot B^{-1}$ unless $B = (0)$, in which case the `lidia_error_handler` will be evoked.

`void square (qi_class & C, const qi_class & A)`

C is set to a reduced representative of A^2. The appropriate function `square_imag` or `square_real` is used, depending on whether the current order is imaginary or real.

`void square_imag (qi_class & C, const qi_class & A)`

C is set to a reduced representative of A^2. In the interest of efficiency, no checking is done to ensure that the current order is actually imaginary.

`void nudupl (qi_class & C, const qi_class & A)`

C is set to a reduced representative of A^2 using the NUDUPL algorithm of Shanks [17]. In the interest of efficiency, no checking is done to ensure that the current order is actually imaginary.

`void square_real (qi_class & C, const qi_class & A)`

C is set to a reduced representative of A^2. In the interest of efficiency, no checking is done to ensure that the current order is actually real.

`void power (qi_class & C, const qi_class & A, const bigint & i)`

C is set to a reduced representative of A^i. If $i < 0$, B^i is computed where $B = A^{-1}$. The appropriate function `power_imag` of `power_real` is used, depending on whether the current order is imaginary or real.

`void power (qi_class & C, const qi_class & A, const long i)`

`void power_imag (qi_class & C, const qi_class & A, const bigint & i)`

C is set to a reduced representative of A^i. If $i < 0$, B^i is computed where $B = A^{-1}$. in the interest of efficiency, no checking is done to ensure that the current order is actually imaginary.

`void power_imag (qi_class & C, const qi_class & A, const long i)`

`void power_real (qi_class & C, const qi_class & A, const bigint & i)`

C is set to a reduced representative of A^i. If $i < 0$, B^i is computed where $B = A^{-1}$. In the interest of efficiency, no checking is done to ensure that the current order is actually real.

`void power_real (qi_class & C, const qi_class & A, const long i)`

`void nupower (qi_class & C, const qi_class & A, const bigint & i)`

C is set to a reduced representative of A^i using the NUCOMP and NUDUPL algorithms of Shanks [17]. If $i < 0$, B^i is computed where $B = A^{-1}$. In the interest of efficiency, no checking is done to ensure that the current order is actually imaginary.

`void nupower (qi_class & C, const qi_class & A, const long i)`

Comparisons

The binary operators `==`, `!=`, and the unary operator `!` (comparison with (0)) are overloaded and can be used in exactly the same way as in the programming language C++. Let A be an instance of type `qi_class`.

`bool A.is_zero () const`

returns `true` if $A = (0)$, `false` otherwise.

`bool A.is_one () const`

returns `true` if $A = (1)$, `false` otherwise.

`bool A.is_equal (const qi_class & B) const`

returns `true` if A and B are equal, `false` otherwise. Note that this function tests whether the representative ideals of the equivalence classes A and B are equal. Use `is_equivalent` for equivalence testing.

Basic Methods and Functions

`operator qi_class_real () const`
cast operator which implicitly converts a `qi_class` to a `qi_class_real`.

`operator quadratic_ideal () const`
cast operator which implicitly converts a `qi_class` to a `quadratic_ideal`.

`operator quadratic_form () const`
cast operator which implicitly converts a `qi_class` to a `quadratic_form`.

`void swap (qi_class & A, qi_class & B)`
exchanges the values of A and B.

High-Level Methods and Functions

Let A be of type `qi_class`.

`bool generate_prime_ideal (qi_class & A, const bigint & p)`
attempts to set A to a reduced representative of the prime ideal lying over the prime p. If successful, i.e., the Kronecker symbol $(\Delta/p) \neq -1$, and the prime ideal is invertible, `true` is returned, `false` otherwise. If no current order has been defined, the `lidia_error_handler` will be evoked.

Reduction operator

`void A.rho ()`
applies the reduction operator ρ once to A, resulting in another reduced representative of the same equivalence class. If the current quadratic order is not real, the `lidia_error_handler` will be evoked.

`void apply_rho (qi_class & C, const qi_class & A)`
sets C to the result of the reduction operator ρ being applied once to A. If the current quadratic order is not real, the `lidia_error_handler` will be evoked.

`qi_class apply_rho (qi_class & A)`
returns a `qi_class` corresponding to the result of the reduction operator ρ being applied once to A. If the current quadratic order is not real, the `lidia_error_handler` will be evoked.

`void A.inverse_rho ()`
applies the inverse reduction operator ρ^{-1} once to A, resulting in another reduced representative of the same equivalence class. If the current quadratic order is not real, the `lidia_error_handler` will be evoked.

`void apply_inverse_rho (qi_class & C, const qi_class & A)`
sets C to the result of the inverse reduction operator ρ^{-1} being applied once to A. If the current quadratic order is not real, the `lidia_error_handler` will be evoked.

`qi_class apply_inverse_rho (qi_class & A)`
returns a `qi_class` corresponding to the result of the inverse reduction operator ρ^{-1} being applied once to A. If the current quadratic order is not real, the `lidia_error_handler` will be evoked.

Equivalence and principality testing

`bool` A`.is_equivalent (const qi_class &` B`) const`

returns `true` if A and B are in the same ideal equivalence class, `false` otherwise. If the current quadratic order is imaginary, this is simply an equality test. If the current order is real, the `qi_class_real` function of the same name is used.

`bool` A`.is_principal () const`

returns `true` if A is principal, `false` otherwise. If the current quadratic order is imaginary, this is simply an equality test with (1). If the current order is real, the `qi_class_real` function of the same name is used.

Orders of elements in the class group

In the following functions, if no current order has been defined, the `lidia_error_handler` will be evoked.

`bigint` A`.order_in_CL () const`

returns the order of A in the ideal class group of the current quadratic order, i.e., the least positive integer x such that A^x is equivalent to (1). It uses one of the following order functions depending on whether the class number is known and the size of the discriminant.

`bigint` A`.order_BJT (long` v`) const`

computes the order of A using the unconditional method of [12] for imaginary orders and an unpublished variation for real orders, assuming that the class number is not known. The parameter v is the size of the initial step-width. This parameter is ignored if the corresponding quadratic order is real.

`bigint` A`.order_h () const`

computes the order of A using the method of [57] where the class number is known. The class number of the current quadratic order is computed if it has not been already.

`bigint` A`.order_mult (bigint &` h`, rational_factorization &` *hfact*`) const`

computes the order of A using the method of [57] where h is a multiple of the order and *hfact* is a `rational_factorization` of h.

`bigint` A`.order_shanks () const`

computes the order of A using the baby-step giant-step method of [57] of complexity $O(|\Delta|^(1/5))$. The result is correct, but the complexity is conditional on the ERH.

`bigint` A`.order_subexp () const`

computes the order of A using a sub-exponential method whose complexity is conditional on the ERH. The algorithms for both real and imaginary orders simply compute the class number with a sub-exponential algorithm and then use `order_h`.

`bool` A`.verify_order (const bigint &` x`) const`

returns true if x is the order of A in the class group.

Discrete logarithms in the class group

In the following functions, if no current order has been defined, the `lidia_error_handler` will be evoked.

`bool` A`.DL (const qi_class &` G`, bigint &` x`) const`

returns `true` if A belongs to the cyclic subgroup generated by G, `false` otherwise. If so, x is set to the discrete logarithm of A to the base G, i.e., the least non-negative integer x such that G^x is equivalent to A. If not, x is set to the order of G. At the moment, `DL_BJT` is the only algorithm that is implemented, and this function simply calls it.

`bool` A`.DL_BJT (const qi_class &` G`, bigint &` x`, long` v`) const`

computes the discrete logarithm of A to the base G using the unconditional method of [12] for imaginary orders and an unpublished variation for real orders, assuming that the class number is not known.

`bool` A`.DL_subexp (const qi_class &` G`, bigint &` x`) const`

computes the discrete logarithm of A to the base G using the sub-exponential method of [36], whose complexity is conditional on the ERH.

`bool` A`.verify_DL (const qi_class &` G`, const bigint &` x`, const bool is_DL) const`

if is_DL is `true` tests whether G^x is equivalent to A. Otherwise, tests whether x is the order of G.

Structure of a subgroup of the class group

In the following functions, if no current order has been defined, the `lidia_error_handler` will be evoked.

`base_vector< bigint > subgroup (base_vector< qi_class > &` G`)`

returns the vector of elementary divisors representing the structure of the subgroup generated by the classes contained in G. At the moment, `subgroup_BJT` is the only algorithm that is implemented, and this function simply calls it.

```
base_vector< bigint > subgroup_BJT (base_vector< qi_class > & G,
                                    base_vector< long > & v)
```

computes the structure of the subgroup generated by G using the unconditional methods of [12] for imaginary orders and an unpublished variation for real orders, assuming that the class number is not known.

Input/Output

Let A be of type `qi_class`. The `istream` operator `>>` and the `ostream` operator `<<` are overloaded. The input of a `qi_class` consists of `(a b)`, where a and b are integers corresponding to the representation $A = [a\mathbb{Z} + (b + \sqrt{\Delta})/2\,\mathbb{Z}]$. The output has the format (a, b). For input, the current quadratic order must be set first, otherwise the `lidia_error_handler` will be evoked.

See also

`qi_class_real`, `quadratic_order`, `quadratic_ideal`, `quadratic_form`

Examples

```
#include <LiDIA/quadratic_order.h>

int main()
{
```

```
    quadratic_order QO;
    qi_class A, B, C;
    bigint D, p, x;

    do {
        cout << "Please enter a quadratic discriminant: "; cin >> QO ;
    } while (!QO.assign(D));
    cout << endl;

    qi_class::set_current_order(QO);

    /* compute 2 prime ideals */
    p = 3;
    while (!generate_prime_ideal(A, p))
        p = next_prime(p);
    p = next_prime(p);
    while (!generate_prime_ideal(B, p))
        p = next_prime(p);

    cout << "A = " << A << endl;
    cout << "B = " << B << endl;

    power(C, B, 3);
    C *= -A;
    square(C, C);
    cout << "C = (A^-1*B^3)^2 = " << C << endl;

    cout << "|<C>| = " << C.order_in_CL() << endl;

    if (A.DL(B, x))
        cout << "log_B A = " << x << endl;
    else
        cout << "no discrete log:  |<B>| = " << x << endl;

    return 0;
}
```

Example:

```
Please enter a quadratic discriminant: -82636319

A = (3, 1)
B = (5, 1)
C = (A^-1*B^3)^2 = (2856, -271)
|<C>| = 20299
log_B A = 17219
```

For further examples please refer to `LiDIA/src/packages/quadratic_order/qi_class_appl.cc`.

Author

Michael J. Jacobson, Jr.

qi_class_real

Name

`qi_class_real` ..... ideal equivalence classes of real quadratic orders

Abstract

`qi_class_real` is a class for representing and computing with the ideal equivalence classes of real quadratic orders, and in particular, for computing in the infrastructure of an equivalence class. It supports basic operations like multiplication as well as more complex operations such as equivalence and principality testing.

Description

A `qi_class_real` is represented by a reduced, invertible quadratic ideal given in standard representation, i.e., an ordered pair of `bigint`s (a, b) representing the $\mathbb{Z}$-module $[a\mathbb{Z} + (b + \sqrt{\Delta})/2\,\mathbb{Z}]$, where $\Delta > 0$ is the discriminant of a real quadratic order. This representative of an equivalence class is not unique, but it is one of a finite set of representatives. A distance δ (logarithm of a minimum) is also stored with the ideal, so that computations in the infrastructure of an equivalence class are possible. It is the user's responsibility to keep track of whether a specific distance is taken from the unit ideal or some other ideal.

This class is derived from `qi_class`, and most of the functions work in exactly the same way. Additional functions as well as the functions that work differently from `qi_class` are detailed below.

Constructors/Destructor

Constructors have the same syntax as `qi_class`. The distance is always initialized to 0. If the current quadratic order is not real, the `qi_class_real` will be initialized to zero.

`ct qi_class_real ()`

`ct qi_class_real (const bigint & `a_2`, const bigint & `b_2`)`

`ct qi_class_real (const long `a_2`, const long `b_2`)`

`ct qi_class_real (const quadratic_form & `f`)`

`ct qi_class_real (const quadratic_ideal & `A`)`

`ct qi_class_real (const qi_class & `A`)`

`ct qi_class_real (const qi_class_real & `A`)`

`dt ~qi_class_real ()`

Initialization

Before any arithmetic is done with a `qi_class_real`, the current quadratic order must be initialized. Note that this quadratic order is the same as that used for `qi_class`.

`static void qi_class_real::set_current_order (quadratic_order & `$\mathcal{O}$`)`

`static void qi_class_real::verbose (int` *state*`)`

sets the amount of information which is printed during computations. If *state* = 1, then the elapsed CPU time of some computations is printed. If *state* = 2, then extra information is output during verifications. If *state* = 0, no extra information is printed. The default is *state* = 0.

`static void qi_class_real::verification (int` *level*`)`

sets the *level* of post-computation verification performed. If *level* = 0 no extra verification is performed. If *level* = 1, then verification of the principality and equivalence tests will be performed. The success of the verification will be output only if the verbose mode is set to a non-zero value, but any failures are always output. The default is *level* = 0.

Assignments

Let A be of type `qi_class_real`. Assignment is the same as in `qi_class`, with the exception that if the current quadratic order is not real, the `lidia_error_handler` will be evoked. Also, some of the functions admit an optional parameter specifying the distance of A. If this parameter is omitted, the distance will be set to 0.

`void` A`.assign_zero ()`

`void` A`.assign_one ()`

`void` A`.assign_principal (const bigint &` x`, const bigint &` y`, const bigfloat &` *dist* `= 0.0)`

`bool` A`.assign (const bigint &` a_2`, const bigint &` b_2`, const bigfloat &` *dist* `= 0.0)`

`bool` A`.assign (const long` a_2`, const long` b_2`, const bigfloat &` *dist* `= 0.0)`

`void` A`.assign (const quadratic_form &` f`, const bigfloat &` *dist* `= 0.0)`

`void` A`.assign (const quadratic_ideal &` B`, const bigfloat &` *dist* `= 0.0)`

`void` A`.assign (const qi_class &` B`, const bigfloat &` *dist* `= 0.0)`

`void` A`.assign (const qi_class_real &` B`)`

Access Methods

Let A be of type `qi_class_real`. The access functions are the same as in `qi_class`, with the addition of the function `get_distance`.

`bigint` A`.get_a () const`

`bigint` A`.get_b () const`

`bigint` A`.get_c () const`

`bigfloat` A`.get_distance () const`

returns the distance of A.

`static bigint A.discriminant () const`

`static quadratic_order & A.get_current_order () const`

Arithmetical Operations

Let A be of type `qi_class_real`. The arithmetic operations are identical to `qi_class`, with the exception that the distance of the result is also computed.

`void multiply (qi_class_real & C, const qi_class_real & A, const qi_class_real & B)`

`void multiply_real (qi_class_real & C, const qi_class_real & A,`
`const qi_class_real & B)`

`void A.invert ()`

`void inverse (qi_class_real & A, const qi_class_real & B)`

`qi_class_real inverse (qi_class_real & A)`

`void divide (qi_class_real & C, const qi_class_real & A, const qi_class_real & B)`

`void square (qi_class_real & C, const qi_class_real & A)`

`void square_real (qi_class_real & C, const qi_class_real & A)`

`void power (qi_class_real & C, const qi_class_real & A, const bigint & i)`

`void power (qi_class_real & C, const qi_class_real & A, const long i)`

`void power_real (qi_class_real & C, const qi_class_real & A, const bigint & i)`

`void power_real (qi_class_real & C, const qi_class_real & A, const long i)`

Comparisons

Comparisons are identical to `qi_class`. Let A be an instance of type `qi_class_real`.

`bool A.is_zero () const`

`bool A.is_one () const`

`bool A.is_equal (const qi_class_real & B) const`

Basic Methods and Functions

`operator qi_class () const`
cast operator which implicitly converts a `qi_class_real` to a `qi_class`.

`operator quadratic_ideal () const`
cast operator which implicitly converts a `qi_class_real` to a `quadratic_ideal`.

`operator quadratic_form () const`
cast operator which implicitly converts a `qi_class_real` to a `quadratic_form`.

`void swap (qi_class_real & A, qi_class_real & B)`
exchanges the values of A and B.

High-Level Methods and Functions

Let A be of type `qi_class_real`. The following two functions are the same as the corresponding `qi_class` functions.

`bool generate_prime_ideal (qi_class_real & `A`, const bigint & `p`)`
the distance of A is set to 0.

Reduction operator

`void `A`.rho ()`
applies the reduction operator ρ once to A, resulting in another reduced representative of the same equivalence class. The distance of A is adjusted appropriately.

`void apply_rho (qi_class_real & `C`, const qi_class_real & `A`)`
sets C to the result of the reduction operator ρ being applied once to A. The distance of C is set appropriately.

`qi_class_real apply_rho (qi_class_real & `A`)`
returns a `qi_class_real` corresponding to the result of the reduction operator ρ being applied once to A. The distance of the return value is set appropriately.

`void `A`.inverse_rho ()`
applies the inverse reduction operator ρ^{-1} once to A, resulting in another reduced representative of the same equivalence class. The distance of A is adjusted appropriately.

`void apply_inverse_rho (qi_class_real & `C`, const qi_class_real & `A`)`
sets C to the result of the inverse reduction operator ρ^{-1} being applied once to A. The distance of C is set appropriately.

`qi_class_real apply_inverse_rho (qi_class_real & `A`)`
returns a `qi_class_real` corresponding to the result of the inverse reduction operator ρ^{-1} being applied once to A. The distance of the return value is set appropriately.

`qi_class_real nearest (qi_class_real & `S`, const bigfloat & `E`)`
returns the `qi_class_real` in the same class as S whose distance from S is as close to E in value as possible.

`bigfloat `A`.convert_distance (const qi_class_real & `B`, const bigfloat & `*dist*`) const`
given that the distance from B to A is close to *dist*, returns a more precise approximation of the distance from B to A.

Equivalence and principality testing

In the following functions, if no current order has been defined, the `lidia_error_handler` will be evoked. The parameter *dist* is optional in all the equivalence and principality testing functions.

`bool `A`.is_equivalent (const qi_class_real & `B`, bigfloat & `*dist*`) const`
returns `true` if A and B are in the same ideal equivalence class, `false` otherwise. If A and B are equivalent, *dist* is set to the distance from B to A, otherwise it is set to the regulator of the current quadratic order. Either `is_equivalent_buch` or `is_equivalent_subexp` is used, depending on the size of the discriminant.

`bool` A`.is_equivalent_buch (const qi_class_real &` B`, bigfloat &` *dist*`) const`

tests whether A and B are equivalent using the unconditional method of [5] assuming that the regulator is not known.

`bool` A`.is_equivalent_subexp (const qi_class_real &` B`, bigfloat &` *dist*`) const`

tests whether A and B are equivalent using a variation of the sub-exponential method of [36] whose complexity is conditional on the ERH.

`bool` A`.verify_equivalent (const qi_class_real &` B`, const bigfloat &` x`,`
`const bool` *is_equiv*`) const`

verifies whether x is the distance from B to A if *is_equiv* is `true`, otherwise it verifies that X is the regulator.

`bool` A`.is_principal (bigfloat &` *dist*`) const`

returns `true` if A is principal, `false` otherwise. If A is principal, *dist* is set to the distance from (1), otherwise it is set to the regulator of the current quadratic order. Either `is_principal_buch` or `is_principal_subexp` is used, depending on the size of the discriminant.

`bool` A`.is_principal_buch (bigfloat &` *dist*`) const`

tests whether A is principal using the unconditional method of [5] assuming that the regulator is not known.

`bool` A`.is_principal_subexp (bigfloat &` *dist*`) const`

tests whether A is principal using a variation of the sub-exponential method of [36] whose complexity is conditional on the ERH.

`bool` A`.verify_principal (const bigfloat &` x`, const bool` *is_prin*`) const`

verifies whether x is the distance from (1) to A if *is_prin* is `true`, otherwise it verifies that X is the regulator.

Orders of elements in the class group

These functions work in exactly the same way as `qi_class`.

`bigint` A`.order_in_CL () const`

`bigint` A`.order_BJT (long` v`) const`

`bigint` A`.order_h () const`

`bigint` A`.order_mult (bigint &` h`, rational_factorization &` $hfact$`) const`

`bigint` A`.order_Shanks/Wrench:1962 () const`

`bigint` A`.order_subexp () const`

`bool` A`.verify_order (const bigint &` x`) const`

Discrete logarithms in the class group

These functions work in exactly the same way as `qi_class`.

```
bool A.DL (const qi_class_real & G, bigint & x) const

bool A.DL_BJT (const qi_class_real & G, bigint & x, long v) const

bool A.DL_subexp (const qi_class_real & G, bigint & x) const

bool A.verify_DL (const qi_class_real & G, const bigint & x, const bool is-DL) const
```

Structure of a subgroup of the class group

These functions work in exactly the same way as `qi_class`.

```
base_vector< bigint > subgroup (base_vector< qi_class_real > & G)

base_vector< bigint > subgroup_BJT (base_vector< qi_class_real > & G,
                                    base_vector< long > & v)
```

Input/Output

Let A be of type `qi_class_real`. The `istream` operator `>>` and the `ostream` operator `<<` are overloaded. The input of a `qi_class_real` consists of `(a b dist)`, where a and b are integers corresponding to the representation $A = [a\mathbb{Z} + (b + \sqrt{\Delta})/2\,\mathbb{Z}]$, and *dist* is the distance corresponding to A. The output has the format $(a, b, dist)$. For input, the current quadratic order must be set first, otherwise the `lidia_error_handler` will be evoked.

See also

`qi_class`, `quadratic_order`, `quadratic_ideal`, `quadratic_form`

Examples

```
#include <LiDIA/quadratic_order.h>

int main()
{
    quadratic_order QO;
    qi_class_real A, B, C, U;
    bigint D, p, x;
    bigfloat dist;

    do {
        cout << "Please enter a quadratic discriminant: ";
        cin >> D;
    } while (!QO.assign(D));
    cout << endl;

    qi_class_real::set_current_order(QO);

    /* compute 2 prime ideals */
    p = 3;
    while (!generate_prime_ideal(A, p))
        p = next_prime(p);
    p = next_prime(p);
```

```
    while (!generate_prime_ideal(B, p))
        p = next_prime(p);

    cout << "A = " << A << endl;
    cout << "B = " << B << endl;

    power(C, B, 3);
    C *= -A;
    square(C, C);
    cout << "C = (A^-1*B^3)^2 = " << C << endl;

    cout << "|<C>| = " << C.order_in_CL() << endl;

    cout << "A principal? - " << A.is_principal(dist) << endl;
    cout << "distance = " << dist << endl;
    U.assign_one();
    C = nearest(U, dist);
    cout << C << "\n\n";

    cout << "A equivalent to B? - " << A.is_equivalent(B, dist) << endl;
    cout << "distance = " << dist << endl;
    C = nearest(B, dist);
    cout << C << endl;

    return 0;
}
```

Example:

```
Please enter a quadratic discriminant: 178936222537081

A = (3, 13376701, 0)
B = (5, 13376701, 0)
C = (A^-1*B^3)^2 = (140625, 13334741, 2.1972245773362194)
|<C>| = 65
A principal? - 0
distance = 905318.66551222335
(1, 13376703, 905318.66551222335)

A equivalent to B? - 0
distance = 905318.66551222335
(5, 13376701, 905318.66551222335)
```

For further examples please refer to `LiDIA/src/packages/quadratic_order/qi_class_real_appl.cc`.

Author

Michael J. Jacobson, Jr.

quadratic_ideal

Name

`quadratic_ideal` ..... fractional ideals of quadratic orders

Abstract

`quadratic_ideal` is a class for representing and computing with fractional ideals of quadratic orders. It supports basic operations like ideal multiplication as well as more complex operations such as computing the order of an equivalence class in the class group.

Description

A `quadratic_ideal` is represented as an ordered triple (q, a, b) (a and b are `bigint`, q is `bigrational`) representing the $\mathbb{Z}$-module $q[a\mathbb{Z} + (b + \sqrt{\Delta})/2\,\mathbb{Z}]$. In addition to a, b, and q, a pointer to the `quadratic_order` to which the ideal belongs is stored with each instance of `quadratic_ideal`.

Let I be a real quadratic ideal and $a \in I$. We call α a minimum of I, if $\alpha > 0$ and there is no non-zero number $\beta \in I$ such that $|\beta| < |\alpha|$ and $|\sigma\beta| < |\sigma\alpha|$, where σ denotes the conjugate map. We call I reduced, if 1 is a minimum in I, which is equivalent to the fact that $\left|\sqrt{\Delta} - 2|a|\right| < b < \sqrt{\Delta}$, i.e., the corresponding form a, b, c is reduced, and $q = 1/a$. Similarly, a imaginary quadratic ideal I is called reduced, if $a \leq c$ and $b \geq 0$ if $a = c$ and $q = 1/a$. Here, $c = (b^2 - \Delta)/(4a)$.

This class is meant to be used for general computations with quadratic ideals. The classes `qi_class` and `qi_class_real` should be used for computations specific to quadratic ideal equivalence classes.

Constructors/Destructor

`ct quadratic_ideal ()`

initializes with the zero ideal.

`ct quadratic_ideal (const bigint & `a_2`, const bigint & `b_2`, bigrational & `q_2`, quadratic_order & `$\mathcal{O}_2$` = last_order)`

if $q_2[a_2\mathbb{Z} + (b_2 + \sqrt{\Delta})/2\,\mathbb{Z}]$ is an ideal of $\mathcal{O}_2$, A is set to it, otherwise A is set to the zero ideal. If $\mathcal{O}_2$ is omitted, the most-recently accessed `quadratic_order` will be used.

`ct quadratic_ideal (const long `a_2`, const long `b_2`, bigrational & `q_2`, quadratic_order & `$\mathcal{O}_2$` = last_order)`

if $q_2[a_2\mathbb{Z} + (b_2 + \sqrt{\Delta})/2\,\mathbb{Z}]$ is an ideal of $\mathcal{O}_2$, A is set to it, otherwise A is set to the zero ideal. If $\mathcal{O}_2$ is omitted, the most-recently accessed `quadratic_order` will be used.

`ct quadratic_ideal (const quadratic_form & f)`
initializes with the positively oriented ideal associated with f. If f is not regular, the `quadratic_ideal` will be initialized to zero.

`ct quadratic_ideal (const qi_class & A)`
initializes with the reduced ideal A.

`ct quadratic_ideal (const qi_class_real & A)`
initializes with the reduced ideal A.

`ct quadratic_ideal (const quadratic_ideal & A)`
initializes a copy of A.

`ct quadratic_ideal (const quadratic_order & O)`
initializes with the ideal O.

`dt ~quadratic_ideal ()`

Assignments

Let A be of type `quadratic_ideal`. The operator `=` is overloaded. For efficiency reasons, the following functions are also implemented. Note that the parameter $\mathcal{O}_2$ is optional. If it is omitted, the new ideal will belong to the most recently accessed quadratic order.

`void A.assign_zero (quadratic_order & O2 = last_order)`
$A = (0)$, the zero ideal.

`void A.assign_one (quadratic_order & O2 = last_order)`
$A = (1)$, the whole order.

`void A.assign_principal (const bigint & x, const bigint & y,`
`quadratic_order & O2 = last_order)`
sets A to the principal ideal generated by $x + y(\Delta + \sqrt{\Delta})/2$, where Δ is the discriminant of $\mathcal{O}_2$.

`bool A.assign (const bigint & a2, const bigint & b2, const bigrational & q2,`
`quadratic_order & O2 = last_order)`
if $q_2[a_2\mathbb{Z} + (b_2 + \sqrt{\Delta})/2\,\mathbb{Z}]$ is an ideal of $\mathcal{O}_2$, A is set to it and `true` is returned. Otherwise, A will be set to zero and `false` is returned.

`bool A.assign (const long a2, const long b2, const bigrational & q2,`
`quadratic_order & O2 = last_order)`
if $q_2[a_2\mathbb{Z} + (b_2 + \sqrt{\Delta})/2\,\mathbb{Z}]$ is an ideal of $\mathcal{O}_2$, A is set to it and `true` is returned. Otherwise, A will be set to zero and `false` is returned.

`void A.assign (const quadratic_form & f)`
sets A to the positively oriented ideal associated with f. If f is not regular, the `lidia_error_handler` will be evoked.

`void A.assign (const qi_class & B)`
sets A to the reduced ideal B.

`void` A`.assign (const qi_class_real &` B`)`
sets A to the reduced ideal B.

`void` A`.assign (const quadratic_ideal &` B`)`
$A \leftarrow B$.

`bool` A`.assign_order (quadratic_order &` $\mathcal{O}_2$`)`
changes the quadratic order to which A belongs. If A is not an ideal in $\mathcal{O}_2$, nothing is changed and false is returned. Otherwise, true is returned.

`bool` A`.set_order (quadratic_order &` $\mathcal{O}_2$`)`
changes the quadratic order to which A belongs. If A is not an ideal in $\mathcal{O}_2$, nothing is changed and false is returned. Otherwise, true is returned.

`bool` A`.assign (const bigrational &` nq`, const quadratic_number_standard &` q_1`,`
`const quadratic_number_standard &` q_2`)`
If $q(\mathbb{Z}q_1 + \mathbb{Z}q_2)$ is a quadratic ideal, i.e., a $\mathbb{Z}$ module of rank 2, the function returns true, and assigns the standard representation of the module to A. Otherwise the function returns false, and initializes the ideal with the order.

Access Methods

Let A be of type `quadratic_ideal`.

`bigint` A`.get_a () const`
returns the coefficient a in the representation (q, a, b) of A.

`bigint` A`.get_b () const`
returns the coefficient b in the representation (q, a, b) of A.

`bigrational` A`.get_q () const`
returns the coefficient q in the representation (q, a, b) of A.

`bigint` A`.get_c () const`
returns the value $c = (b^2 - \Delta)/(4a)$ corresponding to the representation (q, a, b) of A, where Δ is the discriminant of the quadratic order to which A belongs.

`const quadratic_order &` A`.which_order () const`
returns a reference to the `quadratic_order` of which A belongs.

`const quadratic_order &` A`.get_order () const`
returns a reference to the `quadratic_order` of which A belongs.

`const quadratic_order & which_order (const quadratic_ideal &` A`)`
returns a reference to the `quadratic_order` of which A belongs.

`bigint` A`.discriminant () const`
returns the discriminant of the quadratic order to which A belongs.

Arithmetical Operations

Let A be of type `quadratic_ideal`. The following operators are overloaded and can be used in exactly the same way as in the programming language C++.

```
(unary) -
(binary) +, *, /
(binary with assignment) *=, /=
```

Note: By $-A$ and A^{-1} we denote the inverse of A if it exists, by A/B we denote $A \cdot B^{-1}$, and by $A + B$ the set $\{a + b | a \in A, b \in B\}$.

To avoid copying, these operations can also be performed by the following functions:

`void add (quadratic_ideal & C, const quadratic_ideal & A, const quadratic_ideal & B)`
$C \leftarrow A + B$ if A and B are of the same quadratic order; otherwise the `lidia_error_handler` will be evoked.

`void multiply (quadratic_ideal & C, const quadratic_ideal & A,`
`const quadratic_ideal & B)`
$C \leftarrow A \cdot B$. If A and B do not belong to the same quadratic order, the `lidia_error_handler` will be evoked.

`void multiply (quadratic_ideal & J, const quadratic_ideal & I,`
`const quadratic_number_standard & g)`
$J \leftarrow I \cdot (g \cdot O)$, where O is the order of I and g.

`void A.conjugate ()`
A is set to the conjugate of A.

`void get_conjugate (quadratic_ideal & A, const quadratic_ideal & B)`
A is set to the conjugate of B.

`quadratic_ideal get_conjugate (quadratic_ideal & A)`
returns the conjugate of A.

`void A.invert ()`
If A is invertible, $A \leftarrow A^{-1}$, otherwise A is set to its conjugate.

`void inverse (quadratic_ideal & A, const quadratic_ideal & B)`
If B is invertible, $A \leftarrow B^{-1}$, otherwise A is set to the conjugate of B.

`quadratic_ideal inverse (quadratic_ideal & A)`
If A is invertible, A^{-1} is returned, otherwise the conjugate of A is returned.

`void divide (quadratic_ideal & C, const quadratic_ideal & A, const quadratic_ideal & B)`
$C = A \cdot B^{-1}$. If A and B are not of the same quadratic order or $B = (0)$, the `lidia_error_handler` will be evoked. If B is not invertible, C is set to the product of A and the conjugate of B.

`void divide (quadratic_ideal & J, const quadratic_ideal & I,`
`const quadratic_number_standard & g)`
$J \leftarrow I/(g \cdot O)$, where O is the order of I and g.

`void square (quadratic_ideal & C, const quadratic_ideal & A)`
$C \leftarrow A^2$.

`void power (quadratic_ideal & C, const quadratic_ideal & A, const bigint & i)`
$C \leftarrow A^i$. If $i < 0$, B^i is computed where $B = A^{-1}$ or the conjugate of A if A is not invertible.

`void power (quadratic_ideal & C, const quadratic_ideal & A, const long i)`
$C \leftarrow A^i$. If $i < 0$, B^i is computed where $B = A^{-1}$ or the conjugate of A if A is not invertible.

Comparisons

The binary operators `==`, `!=`, `<=`, `<` (true subset), `>=`, `>` and the unary operator `!` (comparison with (0)) are overloaded and can be used in exactly the same way as in the programming language C++. Let A be an instance of type `quadratic_ideal`.

`bool A.is_zero () const`
returns `true` if $A = (0)$, `false` otherwise.

`bool A.is_one () const`
returns `true` if $A = (1)$, `false` otherwise.

`bool A.is_equal (const quadratic_ideal & B) const`
returns `true` if A and B are equal, `false` otherwise.

`bool A.is_subset (const quadratic_ideal & B) const`
returns `true` if A is a subset of B, `false` otherwise.

`bool A.is_proper_subset (const quadratic_ideal & B) const`
returns `true` if A is a proper subset of B, `false` otherwise.

Basic Methods and Functions

`operator quadratic_form () const`
cast operator which implicitly converts a `quadratic_ideal` to a `quadratic_form`.

`operator qi_class () const`
cast operator which implicitly converts a `quadratic_ideal` to a `qi_class`. The current order of `qi_class` will be set to the order of the ideal.

`operator qi_class_real () const`
cast operator which implicitly converts a `quadratic_ideal` to a `qi_class_real`. The current order of `qi_class` and `qi_class_real` will be set to the order of the ideal.

`void swap (quadratic_ideal & A, quadratic_ideal & B)`
exchanges the values of A and B.

High-Level Methods and Functions

Let A be of type `quadratic_ideal`.

`bigrational` A`.smallest_rational () const`
: returns the smallest rational number in A. This is simply the value of qa, where q and a are coefficients in the representation of A.

`bigint` A`.denominator () const`
: returns the smallest positive integer d such that $d \cdot A$ is integral. This is simply the denominator of q, where q is the coefficient of the representation of A.

`bigint` A`.multiply_by_denominator () const`
: multiplies A by its denominator d, where d is the smallest positive integer d such that $d \cdot A$ is integral.

`bool` A`.is_integral () const`
: returns `true` if A is an integral ideal, `false` otherwise.

`bigint` A`.conductor () const`
: returns the conductor of A.

`bool` A`.is_invertible () const`
: returns `true` if A is invertible, `false` otherwise.

`bigrational` A`.norm () const`
: returns the norm of A.

`quadratic_order` A`.ring_of_multipliers ()`
: returns the ring of multipliers of A.

`bool generate_prime_ideal (quadratic_ideal A, const bigint &` p`,`
`quadratic_order & ` $\mathcal{O}_2$ `= last_order)`
: attempts to set A to the prime ideal lying over the prime p. If successful (the Kronecker symbol $\left(\frac{\Delta}{p}\right) \neq -1$), `true` is returned, otherwise `false` is returned. Note that the parameter $\mathcal{O}_2$ is optional. If it is omitted, the new ideal will belong to the most recently accessed quadratic order.

Reduction

For the reduction operator functions, if A is reduced and belongs to an imaginary quadratic order, it will not be modified.

`bool` A`.is_reduced () const`
: returns `true` if A is reduced, `false` otherwise.

`void` A`.reduce ()`
: reduces A.

`void` A`.reduce (quadratic_number_standard & ` g`)`
: computes g such that A/g is reduced and assigns $A \leftarrow A/g$.

`void` A`.rho ()`
applies the reduction operator ρ once to A.

`void` A`.rho (quadratic_number_standard &` g`)`
computes g sucht that $\rho(A) = A/g$ and assigns $A \leftarrow A/g$.

`void apply_rho (quadratic_ideal &` C`, const quadratic_ideal &` A`)`
sets C to the result of the reduction operator ρ being applied once to A.

`qi_class apply_rho (quadratic_ideal &` A`)`
returns a `qi_class` corresponding to the result of the reduction operator ρ being applied once to A.

`void` A`.inverse_rho ()`
applies the inverse reduction operator once to A.

`void` A`.inverse_rho (quadratic_number_standard &` g`)`
computes g sucht that $\rho^{-1}(A) = A/g$ and assigns $A \leftarrow A/g$.

`void apply_inverse_rho (quadratic_ideal &` C`, const quadratic_ideal &` A`)`
sets C to the result of the inverse reduction operator ρ^{-1} being applied once to A.

`qi_class apply_inverse_rho (quadratic_ideal &` A`)`
returns a `qi_class` corresponding to the result of the inverse reduction operator ρ^{-1} being applied once to A.

Prescribed logarithm

Let Ln denote the Lenstra logarithm: $\mathrm{Ln}(x) = 1/2 \ln |x/\sigma(x)|$.

`void` A`.local_close (quadratic_number_standard &` α`, xbigfloat &` a`, xbigfloat` t`, long` k`)`
The function transforms A into a reduced ideal $B = A/\alpha$, where α is a minimum in A with $|t - \mathrm{Ln}(\alpha)| < |t - \mathrm{Ln}(\beta)| + 2^{-k+1}$ for any minimum β in A. It also computes an absolute k-approximation a to $\mathrm{Ln}(\alpha)$. The function uses reduction, ρ, and inverse ρ operations to determine B. If A is not a real quadratic ideal, the `lidia_error_handler` is called.

`void` A`.order_close (quadratic_number_power_product &` α`, xbigfloat &` a`, xbigfloat` t`, long` k`)`
If A is a real quadratic order, the function transforms A into a reduced ideal $B = A/\alpha$, where α is a minimum in A with $|t - \mathrm{Ln}(\alpha)| < |t - \mathrm{Ln}(\beta)| + 2^{-k+1}$ for any minimum β in A. It also computes an absolute k-approximation a to $\mathrm{Ln}(\alpha)$. The function uses repeated squaring to determine B. If A is not a real quadratic order, the `lidia_error_handler` is called.

`void` A`.close (quadratic_number_power_product &` α `xbigfloat &` a`, xbigfloat` t`, long` k`)`
The function transforms A into a reduced ideal $B = A/\alpha$, where α is a minimum in A with $|t - \mathrm{Ln}(\alpha)| < |t - \mathrm{Ln}(\beta)| + 2^{-k+1}$ for any minimum β in A. It also computes an absolute k-approximation a to $\mathrm{Ln}(\alpha)$. The function uses `order_close` and `local_close` to determine B. If A is not a real quadratic ideal, the `lidia_error_handler` is called.

Equivalence and principality testing

`bool` A`.is_equivalent (const quadratic_ideal &` B`) const`

returns `true` if A and B are in the same ideal equivalence class, `false` otherwise.

`bool` A`.is_principal () const`

returns `true` if A is principal, `false` otherwise.

Orders, discrete logarithms, and subgroup structures

`bigint` A`.order_in_CL ()`

returns the order of A in the ideal class group of its quadratic order, i.e. the least positive integer x such that A^x is equivalent to (1). It uses the `qi_class` function of the same name. If A is not invertible, 0 is returned.

`bool` A`.DL (const quadratic_ideal &` G`, bigint &` x`) const`

returns `true` if A belongs to the cyclic subgroup generated by G, `false` otherwise. If so, x is set to the discrete logarithm of A to the base G, i.e., the least non-negative integer x such that G^x is equivalent to A. If not, x is set to the order of G. The function uses the `qi_class` function of the same name. If either A or G is not invertible, `false` is returned and x is set to 0.

`base_vector< bigint > subgroup (base_vector< quadratic_ideal > &` G`)`

returns the vector of elementary divisors representing the structure of the subgroup generated by the classes corresponding to the invertible ideals contained in G. The function uses the `qi_class` function of the same name.

`bigfloat` A`.regulator ()`

returns an approximation of the regulator of the quadratic order to which A belongs. If A belongs to an imaginary quadratic order, 1 is returned.

`bigint` A`.class_number ()`

returns the order of the ideal class group of the quadratic order to which A belongs.

`base_vector< bigint >` A`.class_group ()`

returns the vector of elementary divisors representing the structure of the ideal class group of the quadratic order to which A belongs.

Input/Output

Let A be of type `quadratic_ideal`. The `istream` operator `>>` and the `ostream` operator `<<` are overloaded. The input of a `quadratic_ideal` consists of `(q a b)`, where q is a `bigrational` and a and b are integers corresponding to the representation $A = q[a\mathbb{Z} + (b + \sqrt{\Delta})/2\,\mathbb{Z}]$. The ideal is assumed to belong to the most recently accessed quadratic order. The output has the format $((q), a, b)$.

See also

`quadratic_order`, `qi_class`, `qi_class_real`, `quadratic_form`

Examples

```
#include <LiDIA/quadratic_order.h>
#include <LiDIA/quadratic_ideal.h>

int main()
{
    quadratic_order QO;
    quadratic_ideal A, B, C;
    bigint D, p, x;

    do {
        cout << "Please enter a quadratic discriminant: "; cin >> D;
    } while (!QO.assign(D));
    cout << endl;

    /* compute 2 prime ideals */
    p = 3;
    while (!generate_prime_ideal(A,p,QO))
        p = next_prime(p);
    p = next_prime(p);
    while (!generate_prime_ideal(B,p,QO))
        p = next_prime(p);

    cout << "A = " << A << endl;
    cout << "B = " << B << endl;

    power(C,B,3);
    C *= -A;
    square(C, C);
    cout << "C = (A^-1*B^3)^2 = " << C << endl;

    cout << "|<C>| = " << C.order_in_CL() << endl;

    C.reduce();
    cout << "C reduced = " << C << endl;

    cout << "ring of multipliers of C:\n";
    cout << C.ring_of_multipliers() << endl;

    if (A.DL(B, x))
        cout << "log_B A = " << x << endl << flush;
    else
        cout << "no discrete log:  |<B>| = " << x << endl << flush;

    return 0
}
```

Example:

```
Please enter a quadratic discriminant: -82636319

A = ((1), 3, 1)
B = ((1), 5, 1)
C = (A^-1*B^3)^2 = ((1/9), 140625, 103541)
|<C>| = 20299
C reduced = ((1), 2856, -271)
ring of multipliers of C:
```

```
Quadratic Order:
   Delta = -82636319 (8)

log_B A = 17219
```

For further examples please refer to `LiDIA/src/packages/quadratic_order/quadratic_ideal_appl.cc`.

Author

Michael J. Jacobson, Jr.

quadratic_number_standard

Name

`quadratic_number_standard` .....elements of quadratic number fields in standard representation.

Abstract

`quadratic_number_standard` is a class that represents elements of quadratic number fields in standard representation. It supports the standard operations for quadratic numbers.

Description

A `quadratic_number_standard` is a triple (a, b, d) of integers, $\gcd(a, b, d) = 1$, and $d > 0$, together with a `quadratic_order` $\mathcal{O}$ of discriminant Δ. It represents the number

$$(a + b\sqrt{\Delta})/d \ ,$$

in the field of fractions of $\mathcal{O}$.

Constructors/Destructor

`ct quadratic_number_standard ()`

creates a quadratic number object with coefficients $(0, 0, 1)$. If the order $\mathcal{O}$ = `quadratic_order::qo_l.last()` has been set by the user, i.e., is different from its initialization value, than the object is initialized by the zero element of $\mathcal{O}$. Otherwise, the object is uninitialized. This method of implicitely assigning quadratic orders to quadratic numbers can be used, e.g., when reading a `base_vector` of numbers, given by their coefficients, from a stream.

`ct quadratic_number_standard (const quadratic_number_standard & q)`

initializes with q.

`dt ~quadratic_number_standard ()`

deletes the quadratic number object.

Assignments

Let q be of type `quadratic_number_standard`. The operator `=` is overloaded. The following functions are also implemented:

`void` q`.assign_order (const quadratic_order &` $\mathcal{O}$`)`

Changes the quadratic order of q to $\mathcal{O}$. I.e., $q \leftarrow (a + b\sqrt{\Delta})/d$, where a, b, d are the coefficients of q and Δ is the discriminant of $\mathcal{O}$.

`void` q`.set_order (const quadratic_order &` $\mathcal{O}$`)`

Changes the quadratic order of q to $\mathcal{O}$. I.e., $q \leftarrow (a + b\sqrt{\Delta})/d$, where a, b, d are the coefficients of q and Δ is the discriminant of $\mathcal{O}$.

`void` q`.set (const bigint &` pa`, const bigint &` pb`, const bigint &` pd`)`

Assigns (pa, pb, pd) to the coefficients (a, b, d) of q.

`void` q`.assign (const bigint &` pa`, const bigint &` pb`, const bigint &` pd`)`

Assigns (pa, pb, pd) to the coefficients (a, b, d) of q.

`void` q`.assign_one ()`

Assigns $(1, 0, 1)$ to the coefficients (a, b, d) of q.

`void` q`.assign_one (const quadratic_order &` $\mathcal{O}$`)`

Changes the order of q to $\mathcal{O}$ and assigns 1.

`void` q`.assign_zero ()`

Assigns $(0, 0, 1)$ to the coefficients (a, b, d) of q.

`void` q`.assign_zero (const quadratic_order &` $\mathcal{O}$`)`

Changes the order of q to $\mathcal{O}$ and assigns 0.

`void` q`.assign (const quadratic_number_standard &` x`)`

$q \leftarrow x$.

`void` q`.randomize ()`

Generates coefficients (a, b, d) at random using the `bigint randomize`(Δ) function, where Δ is the discriminant of the order of q. If the order of q is not set, the `lidia_error_handler` will be called.

`void swap (quadratic_number_standard &` q`, quadratic_number_standard &` r`)`

Swaps q and r.

Access Methods

Let q be of type `quadratic_number_standard`.

`bigint` q`.get_discriminant () const`

Returns the discriminant of the order of q. If the order of q is not set, the error handler will be called.

`const quadratic_order &` q`.which_order () const`

Returns a `const` reference to the order of q. If the order of q is not set, the error handler will be called. As usual, the `const` reference should only be used to create a copy or to immediately apply another operation to it, because after the next operation applied to q the obtained reference may be invalid.

`const quadratic_order & q.get_order () const`

Returns a `const` reference to the order of q. If the order of q is not set, the error handler will be called. As usual, the `const` reference should only be used to create a copy or to immediately apply another operation to it, because after the next operation applied to q the obtained reference may be invalid.

`void q.get (bigint & pa, bigint & pb, bigint & pd) const`

Assigns the coefficients (a, b, d) to (pa, pb, pd).

`const bigint & q.get_a () const`

Returns a `const` reference to the coefficient a of q. As usual, the `const` reference should only be used to create a copy or to immediately apply another operation to it, because after the next operation applied to q the obtained reference may be invalid.

`const bigint & q.get_b () const`

Returns a `const` reference to the coefficient b of q. As usual, the `const` reference should only be used to create a copy or to immediately apply another operation to it, because after the next operation applied to q the obtained reference may be invalid.

`const bigint & q.get_d () const`

Returns a `const` reference to the coefficient d of q. As usual, the `const` reference should only be used to create a copy or to immediately apply another operation to it, because after the next operation applied to q the obtained reference may be invalid.

Arithmetical Operations

The following operators are overloaded.

```
                    (unary) -
                (binary) +, -, *, /
(binary with assignment) +=, -=, *=, /=
```

Furthermore, the following functions are implemented.

```
void add (quadratic_number_standard & q, const quadratic_number_standard & r,
          const quadratic_number_standard & s)
```
$q \leftarrow r + s$.

```
void q.negate ()
```
$q \leftarrow -q$.

```
void subtract (quadratic_number_standard & q, const quadratic_number_standard & r,
               const quadratic_number_standard & s)
```
$q \leftarrow r - s$.

```
void multiply (quadratic_number_standard & q, const quadratic_number_standard & r,
               const quadratic_number_standard & s)
```
$q \leftarrow r \cdot s$.

```
void multiply (quadratic_number_standard & q, const quadratic_number_standard & r,
               const bigint & n)
```
$q \leftarrow r \cdot n$.

`void multiply (quadratic_number_standard & q, const bigint & n,`
`const quadratic_number_standard & s)`
$q \leftarrow n \cdot s$.

`void multiply (quadratic_number_standard & q, const quadratic_number_standard & r,`
`const bigrational & n)`
$q \leftarrow r \cdot n$.

`void multiply (quadratic_number_standard & q, const bigrational & n,`
`const quadratic_number_standard & s)`
$q \leftarrow n \cdot s$.

`void q.multiply_by_denominator ()`
$q \leftarrow (a, b, 1)$, where a, b are the coefficients of q.

`void q.invert ()`
$q \leftarrow 1/q$.

`void inverse (quadratic_number_standard & q, const quadratic_number_standard & r)`
$q \leftarrow 1/r$.

`void divide (quadratic_number_standard & q, const quadratic_number_standard & r,`
`const quadratic_number_standard & s)`
$q \leftarrow r/s$.

`void divide (quadratic_number_standard & q, const quadratic_number_standard & r,`
`const bigrational & n)`
$q \leftarrow r/n$.

`void divide (quadratic_number_standard & q, const bigrational & n,`
`const quadratic_number_standard & s)`
$q \leftarrow n/s$.

`void square (quadratic_number_standard & q, const quadratic_number_standard & r)`
$q \leftarrow r^2$.

`void power (quadratic_number_standard & q, const quadratic_number_standard & r,`
`unsigned long e)`
$q \leftarrow r^e$.

`void power (quadratic_number_standard & q, const quadratic_number_standard & r,`
`const bigint & e)`
$q \leftarrow r^e$.

Comparisons

Let q be of type `quadratic_number_standard`. The binary operators `==` and `!=` are overloaded for comparison with `quadratic_number_standard`, `bigint`, and `bigrational`.

Furthermore, the follwing functions are implemented.

`bool` q`.is_zero () const`
returns `true` if $q = 0$, `false` otherwise.

`bool` q`.is_one () const`
returns `true` if $q = 1$, `false` otherwise.

`bool` q`.is_negative () const`
returns `true` if $q < 0$, `false` otherwise.

`bool` q`.is_positive () const`
returns `true` if $q \geq 0$, `false` otherwise.

`bool` q`.is_integer () const`
returns `true` if $q \in \mathbb{Z}$, `false` otherwise.

`bool` q`.is_rational_number () const`
returns `true` if $q \in \mathbb{Q}$, `false` otherwise.

Basic Methods and Functions

Let Ln denote the Lenstra logarithm: $\mathrm{Ln}(x) = 1/2|x/\sigma(x)|$. Let q be of type `quadratic_number_standard`.

`void` q`.absolute_value ()`
$q \leftarrow |q|$.

`void` q`.conjugate ()`
$q \leftarrow \sigma(q)$.

`void conjugate (quadratic_number_standard &` q`, const quadratic_number_standard &` r`)`
$q \leftarrow \sigma(r)$.

`int` q`.get_sign () const`
Returns -1, if $q < 0$, and 1 otherwise.

`bigrational` q`.norm () const`
Returns the norm of q.

`void` q`.norm (bigrational &` n`) const`
Assigns the norm of q to n.

`bigrational norm (const quadratic_number_standard &` q`)`
Returns the norm of q.

`bigrational` q`.trace () const`
Returns the trace of q.

`void` q`.trace (bigrational &` n`) const`
Assigns the trace of q to n.

`bigrational trace (const quadratic_number_standard & q)`
Returns the trace of q.

`xbigfloat q.get_absolute_Ln_approximation (long k) const`
Returns an absolute k-approximation to $\mathrm{Ln}(q)$. See `xbigfloat`.

`void q.get_absolute_Ln_approximation (xbigfloat & l, long k) const`
Assigns an absolute k-approximation to $\mathrm{Ln}(q)$ to l. See `xbigfloat`.

`xbigfloat q.get_relative_approximation (long k) const`
Returns a relative k-approximation to q. See `xbigfloat`.

`void q.get_relative_approximation (xbigfloat & l, long k) const`
Assigns a relative k-approximation to q to l. See `xbigfloat`.

`bool q.check_Ln_correctness (lidia_size_t` *trials* `= 5) const`
Verifies the function `q.get_absolute_Ln_approximation` by approximating the Ln with two accuracies and checking the correctness of the approximations using the function `check_absolute_error` of `xbigfloat`. This test is repeated *trials* times. It returns `true`, if all tests succeed and `false` otherwise.

Input/Output

Let q be of type `quadratic_number_standard`.

`istream` operator `>>` and `ostream` operator `<<` are overloaded. Input and output of a `quadratic_number_standard` are in the following format:

$$(a, b, d)$$

Note, that in the current implementation, it is not possible to input the quadratic order together with the coefficients. The order must be set before reading the coefficients. This may occur either explicitely by calling, e.g., the `assign_order` function, or implicitely (see default constructor).

For example, to input the number $(1 + \sqrt{5})/2$, you have to create a quadratic order $\mathcal{O}$ with discriminant 5. Then you may create the `quadratic_number_standard` object q, call `q.assign_order(`$\mathcal{O}$`)`, and read in $(1, 1, 2)$ from stream. Alternatively, you may set `quadratic_order::qo_l.set_last(&`$\mathcal{O}$`)`, create the `quadratic_number_standard` object q using the default constructor, and read in $(1, 1, 2)$ from stream.

This behaviour is not that satisfactory and may be changed in the future.

`int string_to_quadratic_number_standard (const char* s, quadratic_number_standard & q)`
Assumes that `s` starts with any number of blanks, followed by (a, b, d). Reads a, b, d and assign them as coefficients to q. Returns the number of characters read from `s`. The quadratic order of q must be set, before calling this function.

See also

`xbigfloat`, `quadratic_order`, `quadratic_number_power_product`.

Examples

```
#include <LiDIA/quadratic_order.h>
#include <LiDIA/quadratic_number_standard.h>

int main()
{
    quadratic_order O;
    quadratic_number_standard q;

    O.assign(5);
    q.assign_order(O);

    q.assign(1, 1, 2);
    square(q,q);

    cout << "((1+sqrt(5)/2)^2 = " << q << endl;
    return 0;
}
```

The output of the program is

```
((1+sqrt(5)/2)^2 = (3,1,2)
```

An extensive example of `quadratic_number_standard` can be found in LiDIA's installation directory under `LiDIA/src/packages/quadratic_order/quadratic_number_standard_appl.cc`

Author

Markus Maurer

quadratic_number_power_product_basis

Name

`quadratic_number_power_product_basis` .....basis for power products of quadratic numbers.

Abstract

`quadratic_number_power_product_basis` is a class that represents a basis for a power product of quadratic numbers. Its main use is to reduce memory needed by power products by providing a basis that can be shared by several power products and to increase efficiency in power product computations by speeding up the computation of approximations to the logarithms of the elements of the basis.

Description

A `quadratic_number_power_product_basis` is an array b of elements of type `quadratic_number_standard`. Let n be the number of elements in b. Then

$$b = (b_0, \ldots, b_{n-1}) \ ,$$

where b_i $(0 \leq i < n)$ represents a quadratic number. In principle, it is possible that the numbers in the basis belong to different quadratic number fields.

Constructors/Destructor

`ct quadratic_number_power_product_basis ()`
creates a basis with no elements.

`dt ~quadratic_number_power_product_basis ()`
deletes the object.

Assignments

Let b be of type `quadratic_number_power_product_basis`. The operator `=` is overloaded. The following functions are also implemented:

`void` b`.assign (const quadratic_number_power_product_basis &` c`)`
c is copied to b.

`void` b`.set_basis (const base_vector< quadratic_number_standard > &` c`)`
$b \leftarrow (c[0], \ldots, c[s-1])$, where $s = c$`.get_size()`.

`void` b`.set_basis (const quadratic_number_standard &` c`)`

$b \leftarrow (c)$.

`void` b`.concat (const quadratic_number_power_product_basis &` c`,`
`const quadratic_number_standard &` q`)`

$b \leftarrow (c_0, \ldots, c_{n-1}, q)$, where n is the number of elements in c.

`void` b`.concat (const quadratic_number_power_product_basis &` c`,`
`const quadratic_number_power_product_basis &` d`)`

$b \leftarrow (c_0, \ldots, c_{m-1}, d_0, \ldots, d_{n-1})$, where m and n are the number of elements in c and d, respectively.

Access Methods

Let b be of type `quadratic_number_power_product_basis`.

`const base_vector< quadratic_number_standard > & `b`.get_basis () const`

Returns a `const` reference to a vector $[b_0, \ldots, b_{n-1}]$, where n is the number of elements in b. As usual, the `const` reference should only be used to create a copy of the vector or to immediately apply another operation on the vector, because after the next operation applied to b the obtained reference may be invalid.

`lidia_size_t` b`.get_size () const`

Returns the number of elements of b.

`const quadratic_number_standard & `b`.operator[] (lidia_size_t` i`) const`

Returns a `const` reference to b_i, if $0 \leq i < n$, where n is the number of elements in b. If i is outside that range, the `lidia_error_handler` will be called. As usual, the `const` reference should only be used to create a copy or to immediately apply another operation to it, because after the next operation applied to b the obtained reference may be invalid.

`const quadratic_order & `b`.get_order () const`

Returns a `const` reference to the quadratic order of b_0. If the number of elements of b is zero, the `lidia_error_handler` will be called. This function is for convenience in the case, where all numbers are defined over the same order. As usual, the `const` reference should only be used to create a copy or to immediately apply another operation to it, because after the next operation applied to b the obtained reference may be invalid.

Comparisons

`bool operator== (const quadratic_number_power_product_basis &` c`,`
`const quadratic_number_power_product_basis &` d`)`

Returns `true`, if the address of c is equal to the address of d. Otherwise, it returns `false`.

Basic Methods and Functions

Let b be of type `quadratic_number_power_product_basis`.

`const xbigfloat & b.get_absolute_Ln_approximation (long k, lidia_size_t i) const`

Returns a `const` reference to an absolute k-approximation to $\mathrm{Ln}(b_i)$, where Ln is the Lenstra logarithm (see `xbigfloat`, `quadratic_number_standard`). If $i < 0$ or $n \leq i$, where n is the number of elements in b, the `lidia_error_handler` will be called. Approximations to the logarithm and their accuracies are internally stored. Hence, if k is less than or equal to the currently stored accuracy, the previously computed approximation is only truncated to create the absolute k approximation. Only if k is larger, a new approximation is computed.

As usual, the `const` reference should only be used to create a copy or to immediately apply another operation to it, because after the next operation applied to b the obtained reference may be invalid.

`void b.absolute_Ln_approximations (const matrix< bigint > & M, long k)`

This function computes approximations to the logarithms of the base elements, that are internally stored. More precisely, let n be the number of elements of b. M must be a matrix with n rows; otherwise the `lidia_error_handler` is called. Let m be the number of columns of M. For each $0 \leq i < n$ the function computes and internally stores an absolute $k + 2 + b(n) + t$ approximation l_i to $\mathrm{Ln}(b_i)$, where $t = \max_{0 \leq j < m} b(M_i, j)$, where M_i, j is the element in row i and column j of M. This implies that $\left| \sum_{i=0}^{n-1} e_{i,j} l_i - \mathrm{Ln}(\prod_{i=0}^{n-1} b_i^{M_{i,j}}) \right| < 2^{-k-2}$ for each $0 \leq j < m$. Hence, when building power products where the basis is b and the exponents are given by the columns of M, no recomputations of logarithms is necessary, when asking for absolute k-approximations to the logarithm of the power product.

`bool b.check_Ln_correctness (lidia_size_t trials = 5) const`

Uses the function `check_Ln_correctness(`*trials*`)` of the class `quadratic_number_standard` to check the correctness of the logarithm approximation for the base elements. Returns `true`, if all the tests work correct and `false` otherwise.

`void b.conjugate ()`

Replaces b_i by its conjugate for each $0 \leq i < n$, where n is the number of elements of b.

Input/Output

`istream` operator `>>` and `ostream` operator `<<` are overloaded. Input and output of a `quadratic_number_power_product_basis` are in the format of a `base_vector` of elements of type `quadratic_number_standard`.

See also

`xbigfloat`, `base_vector`, `quadratic_number_standard`, `quadratic_number_power_product`, `quadratic_order`.

Author

Markus Maurer

quadratic_number_power_product

Name

`quadratic_number_power_product` ..... power products of quadratic numbers.

Abstract

`quadratic_number_power_product` is a class that represents power products of quadratic numbers. It uses the class `quadratic_number_power_product_basis` to represent the basis of the power product and a vector of `bigint` to represent the exponents.

Description

A `quadratic_number_power_product` is a pair $p = (b, e)$, where b is of type `quadratic_number_power_product_basis` and e is of type `math_vector` over `bigint`. Let $b = (b_0, \ldots, b_{n-1})$ and $e = (e_0, \ldots, e_{n-1})$. Then the pair p represents the power product

$$p = \prod_{i=0}^{n-1} b_i^{e_i} \quad .$$

The class manages a list of elements of type `quadratic_number_power_product_basis`. When a power product is the result of an operation, then the basis object for the result is the same as that for the operands and only a reference is copied. This applies for all operations on power products, unless stated otherwise.

Constructors/Destructor

`ct quadratic_number_power_product ()`
creates a power product with no elements.

`ct quadratic_number_power_product (const quadratic_number_power_product & q)`
initializes with q.

`dt ~quadratic_number_power_product ()`
deletes the object.

Assignments

Let p be of type `quadratic_number_power_product`. The operator `=` is overloaded for `quadratic_number_power_product` and `quadratic_number_standard`. The following functions are also implemented:

`void` p`.set_basis (const quadratic_number_power_product_basis &` x`)`

Assigns a copy of x to the basis of p and leaves the exponent vector unchanged.

`void` p`.set_basis (const quadratic_number_power_product &` x`)`

Assigns the basis of x to the basis of p.

`void` p`.set_exponents (const base_vector< exp_type > &` e`)`

Assigns e to the exponent vector of p.

`void` p`.reset ()`

Deletes the basis and the exponent vector of p.

`void` p`.assign (const quadratic_number_power_product &` q`)`

$p = q$.

`void` p`.assign (const quadratic_number_standard &` g`)`

$p = g^1$.

`void` p`.assign_one (const quadratic_order &` $\mathcal{O}$`)`

$p = 1^1$ with $1 \in \mathcal{O}$.

`void` p`.assign (const quadratic_ideal &` I`, const xbigfloat &` t`, int` s`)`

Let $\mathcal{O}$ be the real quadratic order of I. It is assumed that I is a principal ideal in $\mathcal{O}$ with $a\mathcal{O} = I$ and $|t - \mathrm{Ln}(a)| < 2^{-4}$, $\mathrm{sgn}(a) = s$. Then a is uniquely determined by I, t, s. The function initializes the power product by a compact representation of a. If the above conditions are not satisfied, the result is undefined. The $\mathrm{sgn}(a)$ is -1, if $a < 0$, and 1 otherwise.

Let $\mathcal{O}$ be a quadratic order of discriminant Δ, K the field of fractions, and $\alpha \in K$. Extending the definition in [16] from quadratic integers to numbers in K, we say that α is in compact representation with respect to Δ, if

$$\alpha = \gamma \prod_{j=1}^{k} \left(\frac{\alpha_j}{d_j} \right)^{2^{k-j}} ,$$

where $\gamma \in K$, $\alpha_j \in \mathcal{O}$, $d_j \in \mathbb{Z}_{>0}$, $1 \le j \le k \le \max\{1, \ln_2 \ln_2(H(\alpha)d(\alpha)) + 2\}$,

$$d(\gamma) \le d(\alpha) \ , \quad H(\gamma) \le |N(\alpha)| d(\alpha) \ ,$$

and

$$0 < d_j \le |\Delta|^{1/2} \ , \quad H(\alpha_j) \le 2|\Delta|^{7/4} \quad \text{for } 1 \le j \le k.$$

Here, $H(\alpha)$ is the height, $N(\alpha)$ the norm, and $d(\alpha)$ the denominator of α. For further details see [45].

`void swap (const quadratic_number_power_product &` p`,`
`const quadratic_number_power_product &` q`)`

Swaps p and q.

Access Methods

Let p be of type `quadratic_number_power_product`.

`const quadratic_number_power_product_basis & p.get_basis ()`

Returns a `const` reference to the basis of p. As usual, the `const` reference should only be used to create a copy or to immediately apply another operation to it, because after the next operation applied to p the obtained reference may be invalid.

`const base_vector< bigint > & p.get_exponents ()`

Returns a `const` reference to the vector of exponents of p. As usual, the `const` reference should only be used to create a copy or to immediately apply another operation to it, because after the next operation applied to p the obtained reference may be invalid.

`const quadratic_order & p.get_order ()`

Returns a `const` reference to the order of the first element in the basis of p. If there is no element in the basis of p, the `lidia_error_handler` is called. This function is for convenience, in the case, that all base elements are defined over the same order. As usual, the `const` reference should only be used to create a copy or to immediately apply another operation to it, because after the next operation applied to p the obtained reference may be invalid.

`bool p.is_initialized ()`

Returns `true`, if a basis and a vector of exponents has been assigned to p, and if both are of same length. Otherwise, it returns `false`.

`int p.get_sign ()`

Returns 1, if $p \geq 0$, and -1 otherwise.

Arithmetical Operations

Let p be of type `quadratic_number_power_product`. In the following, writing $(b, c), (e, f)$ means, that the basis b is concatenated with c and that the exponent vector e is concatenated with f.

`void p.negate ()`

$p \leftarrow -p$.

`void p.multiply (const quadratic_number_power_product & x,`
`const quadratic_number_standard & y)`

Let $x = (b, e)$. Then $p \leftarrow \big((b, y), (e, 1)\big)$.

`void p.multiply (const quadratic_number_standard & y,`
`const quadratic_number_power_product & x)`

Let $x = (b, e)$. Then $p \leftarrow \big((b, y), (e, 1)\big)$.

`void p.multiply (const quadratic_number_power_product & x,`
`const quadratic_number_power_product & y)`

Let $x = (b, e)$ and $y = (c, f)$. Then $p \leftarrow \big((b, c), (e, f)\big)$.

`void p.invert (const quadratic_number_power_product & x)`

Let $p = (b, e)$. Then $p \leftarrow (b, -e)$.

`void p.invert_base_elements ()`

Let $p = (b, e)$. Then $p \leftarrow (b^{-1}, e)$, i.e., each element of the basis is inverted.

`void` p`.divide (const quadratic_number_power_product &` x`,`
`const quadratic_number_standard &` y`)`
Let $x = (b, e)$. Then $p \leftarrow \big((b, y), (e, -1)\big)$.

`void` p`.divide (const quadratic_number_standard &` y`,`
`const quadratic_number_power_product &` x`)`
Let $x = (b, e)$. Then $p \leftarrow \big((b, y), (-e, 1)\big)$.

`void` p`.divide (const quadratic_number_power_product &` x`,`
`const quadratic_number_power_product &` y`)`
Let $x = (b, e)$ and $y = (c, f)$. Then $p \leftarrow \big((b, c), (e, -f)\big)$.

`void` p`.square (const quadratic_number_power_product &` x`)`
Let $x = (b, e)$. Then $p \leftarrow (b, 2 \cdot e)$.

`void` p`.power (const quadratic_number_power_product &` x`, const bigint &` f`)`
Let $x = (b, e)$. Then $p \leftarrow (b, f \cdot e)$.

Comparisons

The binary operators `==` and `!=` are overloaded for comparison with `quadratic_number_power_product` and `quadratic_number_standard`.

These functions are not efficient, because they evaluate the power products for comparison.

Basic Methods and Functions

Let p be of type `quadratic_number_power_product`.

`void` p`.conjugate ()`
Maps p to its conjugate by replacing each element of the basis of p by its conjugate.

`void` p`.norm_modulo (bigint &` *num*`, bigint &` *den*`, const bigint &` m`) const`
Returns *num* and *den*, i.e., the numerator and denominator of the norm of p modulo m, respectively. If m is zero, the `lidia_error_handler` will be called. It is $0 \leq num, den < m$.

`quadratic_number_standard` p`.evaluate () const`
Returns the evaluation of p.

High-Level Methods and Functions

`void` p`.compact_representation (const quadratic_ideal &` I`)`
Let $\mathcal{O}$ be the real quadratic order of I. If $p\mathcal{O} = qI$, then this function transforms p into compact representation. Otherwise, the result is undefined. For a definition of compact representation, see the corresponding `assign` function of the class.

`void` p`.get_short_principal_ideal_generator (xbigfloat &` l`, bigint &` z`, const quadratic_unit &` ϱ`, long` b`, long` k`)`

Let $\mathcal{O}$ be the real quadratic order of p, ϱ its fundamental unit, and let $R = |\operatorname{Ln}(\varrho)|$ denote the regulator of $\mathcal{O}$. If $|b - b(\operatorname{Ln}(\varrho))| \leq 1$ and p is a non-rational number in the field of fractions of $\mathcal{O}$, then this functions computes z, such that either $-R/2 < \operatorname{Ln}(p) - zR \leq R/2$ or $0 \leq \operatorname{Ln} p - zR < R$ and also an absolute k approximation to $\operatorname{Ln}(a)$, where $a = p/\varrho^z$. If the conditions are not satisfied, the result is undefined.

If $k \geq 4$, then l and the ideal $I = p\mathcal{O}$ can be used to compute a compact representation of a using the `assign` function of the class. If the caller is only interested in z, he should choose $k = -b + 6$.

NOTE: The interface of that function will be changed in the future, so that it is not necessary to give the parameters ϱ and b anymore.

Ln approximation

Let Ln denote the Lenstra logarithm: $\operatorname{Ln}(x) = 1/2 \ln |x/\sigma(x)|$.

`void` p`.get_absolute_Ln_approximation (xbigfloat &` l`, long` k`) const`

Computes an absolute k-approximation l to $\operatorname{Ln}(p)$. See `xbigfloat` for definition of absolute approximation.

`void` p`.absolute_Ln_approximation (xbigfloat &` l`, long` k`) const`

Computes an absolute k-approximation l to $\operatorname{Ln}(p)$. See `xbigfloat` for definition of absolute approximation.

`xbigfloat` p`.get_absolute_Ln_approximation (long` k`) const`

Returns an absolute k-approximation to $\operatorname{Ln}(p)$. See `xbigfloat` for definition of absolute approximation.

`xbigfloat` p`.absolute_Ln_approximation (long` k`) const`

Returns an absolute k-approximation to $\operatorname{Ln}(p)$. See `xbigfloat` for definition of absolute approximation.

`void` p`.get_relative_Ln_approximation (xbigfloat &` l`, long` k`, long` m`) const`

Computes a relative k-approximation l to $\operatorname{Ln}(p)$. It requires $|\operatorname{Ln} p| > 2^m$. See `xbigfloat` for definition of relative approximation.

`xbigfloat` p`.get_relative_Ln_approximation (long` k`, long` m`) const`

Returns a relative k-approximation to $\operatorname{Ln}(p)$. It requires $|\operatorname{Ln} p| > 2^m$. See `xbigfloat` for definition of relative approximation.

`xbigfloat` p`.relative_Ln_approximation (long` k`, long` m`) const`

Returns a relative k-approximation to $\operatorname{Ln}(p)$. It requires $|\operatorname{Ln} p| > 2^m$. See `xbigfloat` for definition of relative approximation.

Operations on quasi-units, i.e., numbers of $\mathbb{Q}^* \cdot \mathcal{O}^*$

Let u be a power product of quadratic numbers. We call u a *quasi-unit* of $\mathcal{O}$, if $u \in \mathbb{Q}^* \cdot \mathcal{O}^*$, for a real quadratic order $\mathcal{O}$, i.e., u is the product of a rational number and a unit of $\mathcal{O}$. The set $\mathbb{Q}^* \cdot \mathcal{O}^*$ together with multiplication forms a group with subgroup $(\mathbb{Q}^*, \cdot)$. The factor group $\mathbb{Q}^* \cdot \mathcal{O}^* / \mathbb{Q}^*$ is isomorphic to $\mathcal{O}^*$ by mapping $a \cdot \mathbb{Q}^*$ to a for $a \in \mathcal{O}^*$. If u is a unit, q a non-zero rational number, then $\pm u$ are the only units in the equivalence class $(qu)\mathbb{Q}^*$. Here, $\mathbb{Q}^*$ denotes $\mathbb{Q} \setminus \{0\}$.

`void` u`.compact_representation_of_unit ()`

Transforms u into a compact representation of the positive unit in the equivalence class $u\mathbb{Q}^*$, assuming that u is a quasi unit of its order $\mathcal{O}$. Calls u`.compact_representation(`$\mathcal{O}$`)`. If u is not a quasi unit, the result is undefined. For a definition of compact representation, see the corresponding `assign` function of the class.

`bool` u`.is_rational (long` m`) const`

Let u be a quasi-unit of the real quadratic order $\mathcal{O}$ and let $R > 2^m$, where R is the regulator of $\mathcal{O}$. This functions returns `true`, if u is rational, and `false` otherwise. If u is not a quasi-unit of its order, the result is undefined.

`void` u`.generating_unit (const quadratic_number_power_product &` q_1`,`
`const quadratic_number_power_product &` q_2`, long` m`)`

Let q_1 and q_2 be quasi-units of the real quadratic order $\mathcal{O}$ and let $R > 2^m$, where R is the regulator of $\mathcal{O}$. The function computes a quasi-unit u of $\mathcal{O}$, such that the subgroup generated by the canonical embedding of u in $\mathcal{O}^*$ is the same as the subgroup generated by the embeddings of q_1 and q_2.

`void` u`.generating_unit (bigint &` x`, bigint &` y`, bigint &` M_1`, bigint &` M_2`,`
`const quadratic_number_power_product &` q_1`,`
`const quadratic_number_power_product &` q_2`, long` m`)`

Let q_1 and q_2 be quasi-units of a real quadratic order $\mathcal{O}$ and let $R > 2^m$, where R is the regulator of $\mathcal{O}$. The function computes a quasi-unit u of $\mathcal{O}$, such that the subgroup generated by the canonical embedding of u in $\mathcal{O}^*$ is the same as the subgroup generated by the embeddings of q_1 and q_2.

Identify u, q_1, and q_2 by its embeddings. The function also returns M_1 and M_2 such that $u^{M_1} = q_1$ and $u^{M_2} = q_2$, as well as x and y such that $q_1^x q_2^y = u$, respectively. It is $xM_1 + yM_2 = 1$.

`void` u`.generating_unit (base_vector< quadratic_number_power_product > &` q`, long` m`)`

Let q be a vector of quasi-units of a real quadratic order $\mathcal{O}$ and let $R > 2^m$, where R is the regulator of $\mathcal{O}$. The function computes a quasi-unit u of $\mathcal{O}$, such that the subgroup generated by the canonical embedding of u in $\mathcal{O}^*$ is the same as the subgroup generated by the embeddings of the quasi-units in q.

`void` u`.generating_unit (const matrix< bigint > &` M`,`
`const base_vector< quadratic_number_standard > &` v`, long` l`)`

Let M be a matrix with m rows and n columns and e be a vector of size m of quadratic numbers. It is assumed that $q_j = \prod_{i=0,\ldots,m-1} v_i^{M(i,j)}$ for $j = 0, \ldots, n-1$ are quasi-units of a real quadratic order $\mathcal{O}$ and $R > 2^l$, where R is the regulator of $\mathcal{O}$. The function computes a quasi-unit u of $\mathcal{O}$, such that the subgroup generated by the canonical embedding of u in $\mathcal{O}^*$ is the same as the subgroup generated by the embeddings of the quasi-units $q_0, \ldots, q_n - 1$.

`void` u`.generating_unit (const char * units_input_file)`

Assumes that the `units_input_file` contains a matrix M over `bigint`, a `base_vector` v over `quadratic_number_standard` and a `long` l. The functions reads M, v, and l and calls u`.generating_unit(`M`,` v`,` l`)` to determine the generating quasi-unit u.

Note that `quadratic_order::qo_l.set_last` must be called with the quadratic order of the quadratic numbers in v. See input operator of `quadratic_number_standard` for further details.

`void quadratic_number_power_product::remove_rationals_from_quasi_units (base_vector<`
`quadratic_number_power_product`
`> &` p`, long` l`)`

Let p be a vector of quasi units of an order $\mathcal{O}$ and let the regulator of R be strictly larger than 2^l. The function removes those quasi units from the vector q that a rational numbers and adjusts the size of q.

Tests for units and quasi units

Let u be of type `quadratic_number_power_product`.

If a test for quasi unit on u returns `false`, then u is not a unit. If a test for unit on $u/\sigma(u)$ returns `false`, then u is not a quasi unit. In this way, tests for units can be applied for quasi units and vice versa. This is because a unit is a quasi unit too and if $v = qu$ is a quasi unit, $q \in \mathbb{Q}^*$, $u \in \mathcal{O}^*$, then $v/\sigma(v) = u/\sigma(u) = u^2/N(u) = u^2$ is a unit.

`bool` *u*`.could_be_unit_norm_test (int` *trials* `= 5) const`

If the function returns `false`, u is not a unit. The function computes the numerator n and denominator d of the norm modulo some integer m, respectively. It then checks whether $n \equiv d$ modulo m. If this is not the case, the function returns `false`. If the test is positive for *trials* chosen moduli m, the function returns `true`.

Note that numbers of the form $a/\sigma(a)$ always have norm 1, but are not necessarily units.

`bool` *u*`.could_be_quasi_unit_close_test () const`

If the function returns `false`, u is not a quasi unit of its order $\mathcal{O}$. The function approximates $\mathrm{Ln}(u)$ by t close enough, such that one out of three ideals that are close to t must be $\mathcal{O}$. If this is not the case, the function returns `false`, otherwise it returns `true`. It uses the `order_close` function of `quadratic_ideal` to do this test.

`int` *u*`.could_be_unit_refinement_test () const`

If the function returns 0, u is not a unit in its order $\mathcal{O}$. If it returns 1, then u is a unit in $\mathcal{O}$. If it returns -1, then u is a unit in a overorder of $\mathcal{O}$. The function uses factor refinement, to determine its answer. See [11]. This method is rather slowly compared to the other unit tests.

Input/Output

Let $p = (b, e)$ be of type `quadratic_number_power_product`.

`istream` operator `>>` and `ostream` operator `<<` are overloaded. Input and output of a `quadratic_number_power_product` are in the following format:

$$(b, e)$$

For example, to input the power product $(1 + \sqrt{5})/2$, you have to do the following. Because elements of type `quadratic_number_standard` are entered only by their coefficients, you first have to initialize a quadratic order $\mathcal{O}$ with the discriminant 5 and to set

```
quadratic_order::qo_l.set_last(&O)
```

Then you can call the operator `>>` and enter $([(1, 1, 2)], [1])$.

This behaviour may be improved in the future.

See also

`bigfloat`, `quadratic_order`, `quadratic_unit`.

Examples

The following program generates a randomly chosen power product in the field of fractions of the real quadratic order of discriminant 5. Then it squares the power product.

```
#include <LiDIA/quadratic_order.h>
#include <LiDIA/quadratic_number_standard.h>
#include <LiDIA/quadratic_number_power_product.h>
#include <LiDIA/quadratic_number_power_product_basis.h>
#include <LiDIA/base_vector.h>
#include <LiDIA/bigint.h>
#include <LiDIA/random_generator.h>

int main()
{
    // Generate order.
    //
    quadratic_order O;
    O.assign(5);

    // Generate
    // base_vector< quadratic_number > v and
    // base_vector< bigint > e.
    //
    base_vector< quadratic_number_standard > v;
    base_vector< bigint > e;
    lidia_size_t j, sizePPB;

    sizePPB = 5;
    v.set_capacity(sizePPB);
    e.set_capacity(sizePPB);

    for (j=0; j < sizePPB; j++) {
        v[j].assign_order(O);
        v[j].randomize();
        e[j].randomize(5);
    }

    // Generate power product basis b from v.
    //
    quadratic_number_power_product_basis b;
    b.set_basis(v);

    // Generate power product from b and e.
    //
    quadratic_number_power_product p;
    p.set_basis(b);
    p.set_exponents(e);

    cout << " p = " << p << endl;

    // Square the power product.
    //
    p.square(p);

    // Print result.
    //
    cout << " p^2 = " << p << endl;

    return 0;
}
```

For example, the program could produce the following output.

```
p = ([ (3,2,3) (4,1,3) (3,3,2) (3,3,1) (0,1,1) ],[ 3 0 0 4 2 ])
p^2 = ([ (3,2,3) (4,1,3) (3,3,2) (3,3,1) (0,1,1) ],[ 6 0 0 8 4 ])
```

An extensive example for the use of `quadratic_number_power_product` can be found in LiDIA's installation directory under `LiDIA/src/packages/quadratic_order/quadratic_number_power_product_appl.cc`

Author

Markus Maurer

Chapter 17

Arbitrary Algebraic Number Fields

number_field

Name

`number_field` .....algebraic number fields.

Abstract

`number_field` is a class for representing algebraic number fields $\mathbb{Q}(b)$, where b is a root of a monic irreducible polynomial with coefficients in the rational integers.

Description

A `number_field` consists of a $\mathbb{Z}$-basis of an algebraic number field. This basis is stored using the internal class `nf_base`.

Constructors/Destructor

`ct number_field ()`

initializes with the number field generated by the current `nf_base`. This is the base of the `number_field` or the `order` that was either constructed or read the last before invoking this constructor. If no `number_field` or `order` has been used so far, this defaults to a (mostly useless) dummy field.

`ct number_field (const polynomial< bigint > &)`

initializes with the given polynomial. If the given polynomial is not irreducible or not monic, this results in undefined behaviour.

`ct number_field (const bigint * v, lidia_size_t deg)`

initializes the field by the number field $\mathbb{Q}(b)$, where b is a root of

$$\sum_{i=0}^{deg} v[i]x^i \ .$$

If this polynomial is not irreducible or not monic, this results in undefined behaviour.

`ct number_field (const order & O)`

initializes with the quotient field of the order $\mathcal{O}$.

`ct number_field (const number_field & F)`

initializes with a copy of the field F.

`dt ~number_field ()`

Assignments

Let F be of type `number_field`. The operator `=` is overloaded. For consistency reasons, the following function is also implemented:

`void` F`.assign (const number_field &` K`)`
$F \leftarrow K$.

Access Methods

`nf_base *` F`.which_base () const`
returns the pointer to the `nf_base` used to describe F.

`nf_base * which_base (const number_field &` F`)`
returns the pointer to the `nf_base` used to describe F.

`const polynomial< bigint > &` F`.which_polynomial () const`
returns the generating polynomial of F.

`const polynomial< bigint > & which_polynomial (const number_field &` F`)`
returns the generating polynomial of F.

Basic Methods and Functions

Let F be an instance of type `number_field`.

`lidia_size_t` F`.degree () const`
returns the degree of the field F over $\mathbb{Q}$.

`lidia_size_t degree (const number_field &` F`)`
returns the degree of the field F over $\mathbb{Q}$.

`operator nf_base * () const`
cast operator which implicitly converts a `number_field` to a pointer to the representation of its base.

`lidia_size_t` F`.no_of_real_embeddings () const`
returns the number of embeddings of the field F into $\mathbb{R}$.

`lidia_size_t no_of_real_embeddings (const number_field &` F`)`
returns the number of embeddings of the field F into $\mathbb{R}$.

High-Level Methods and Functions

Let F be an instance of type `number_field`.

`const bigfloat & F.get_conjugate (lidia_size_t i, lidia_size_t j) const`

for $0 \leq i < \deg(F)$ and $0 < j \leq \deg(F)$ this functions returns the j-th conjugate of the i-th element of the basis representing F. If i or j are not within the indicated bounds, the `lidia_error_handler` will be invoked. Note that the order of the conjugates is randomly determined, but once it is determined, it remains fixed. The first F`.no_of_real_embeddings()` conjugates always correspond to the real embeddings of F.

`const math_matrix< bigfloat > & F.get_conjugates () const`

returns a reference to the matrix that is used internally to store the conjugates. The i-th column of the matrix contains the conjugates of the i-th element of the basis representing F.

Input/Output

`istream` operator `>>` and `ostream` operator `<<` are overloaded. We support four formats for the input of a `number_field`:

- You may input the generating polynomial of the `number_field`; however —- due to the limitations of the class `istream` — only the verbose format for polynomials is supported and the polynomial has to start with the variable, e.g.

  $$x^n + \cdots + a_1 x + a_0 \ .$$

  This represents the field $\mathbb{Q}[b]$ where b is a root of the given polynomial, which must be monic and irreducible. The powers $\{1, b, b^2, \ldots, b^{n-1}\}$ of b will be used as the basis of the `number_field`.

- You may give a pair of a (monic and irreducible) generating polynomial f and a transformation matrix T in the form

  $$(f \sqcup T) \ .$$

  Any format supported by `polynomial< bigint >` may be used to give the polynomial. Since any matrix over the rationals would be an acceptable transformation matrix we use a `bigint_matrix` (supporting any format supported by `bigint_matrix`) that is optionally followed by a common denominator in the form "/d" to give the transformation matrix.

  This also represents the field generated by the given polynomial, however in contrast to the previous format $\{1, b, b^2, \ldots, b^{n-1}\} \cdot T$ will be used as the basis of the `number_field`.

- You may only input a transformation matrix. This will be interpreted as a transformation with respect to the current `nf_base`. This is the base of the `number_field` or the `order` that was constructed or read the last before invoking this operator. If there is no current `nf_base`, the `lidia_error_handler` will be invoked. Any format supported by `bigint_matrix` optionally followed by a common denominator in the form "/d" can be used to give this transformation matrix.

- Finally, you may input a multiplication table, describing the basis of the `number_field`: A basis $\{w_1, \ldots, w_n\}$ is uniquely determined by constants $MT(i,j,k)$ with

  $$w_i \cdot w_j = \sum_{k=1}^{n} MT(i,j,k) w_k \ .$$

  Since the multiplication table

  $$\begin{pmatrix} (MT(1,1,1), \ldots, MT(1,1,n)) & \ldots & (MT(1,n,1), \ldots, MT(1,n,n)) \\ \vdots & \vdots & \vdots \\ (MT(n,1,1), \ldots, MT(n,1,n)) & \ldots & (MT(n,n,1), \ldots, MT(n,n,n)) \end{pmatrix} ,$$

is symmetric, we represent such a table by its lower left part, i.e. by the following `bigint_matrix`

$$
\begin{array}{lccr}
(& MT(1,1,1) & \dots & MT(1,1,n) &) \\
(& MT(2,1,1) & \dots & MT(2,1,n) &) \\
(& MT(2,2,1) & \dots & MT(2,2,n) &) \\
(& \vdots & \vdots & \vdots &) \\
(& MT(i,1,1) & \dots & MT(i,1,n) &) \\
(& \vdots & \vdots & \vdots &) \\
(& MT(i,i,1) & \dots & MT(i,i,n) &) \\
(& MT(i+1,1,1) & \dots & MT(i+1,1,n) &) \\
(& \vdots & \vdots & \vdots &) \\
(& MT(i+1,i+1,1) & \dots & MT(i+1,i+1,n) &) \\
(& \vdots & \vdots & \vdots &) \\
(& MT(n,1,1) & \dots & MT(n,1,n) &) \\
(& \vdots & \vdots & \vdots &) \\
(& MT(n,n,1) & \dots & MT(n,n,n) &)
\end{array}
.
$$

As for the previous formats, any format supported by `bigint_matrix` can be used.

The output of a `number_field` always uses either the first or the second format, depending on whether a transformation matrix is necessary to describe the base of the `number_field` or not.

Note that you have to manage by yourself that successive `order`s may have to be separated by blanks, depending on what format you are using for reading and writing matrices.

Notes

Due to the automatic casts to and from `nf_base *` there should be no need to use this (undocumented) internal class, although — for the sake of completeness — it has been mentioned several times.

See also

`order`, `polynomial< bigint >`

Examples

see the documentation of "`order`" for an example.

Author

Stefan Neis

order

Name

`order` ..... orders in algebraic number fields.

Abstract

`order` is a class for representing orders of algebraic number fields and for computing elementary invariants like the discriminant and for maximizing the order at a given prime.

Description

An `order`, exactly like a `number_field`, consists of its $\mathbb{Z}$-basis. This basis is stored using the internal class `nf_base`.

Constructors/Destructor

`ct order (const nf_base * `$base_1$` = nf_base::current_base)`

initializes with the base represented by $base_1$. If no pointer to an `nf_base` is given, the current base is used, which is the base of the `number_field` or the `order` that was constructed or read the last before invoking this constructor. If no `number_field` or `order` has been used so far, this defaults to a (mostly useless) dummy order.

`ct order (const polynomial< bigint > & `p`,`
`const base_matrix< bigint > & `T` = bigint_matrix(), const bigint & `d` = 1)`

if the second and third argument are omitted, this constructor initializes an order with the equation order of the polynomial p. If this polynomial is not irreducible or not monic, this results in undefined behaviour. If T and d are given also, this constructs an order with basis $(1, x, \ldots, x^{\deg(p)}) \cdot T \cdot 1/d$.

`ct order (const bigint_matrix & `A`)`

initializes with the given matrix A as multiplication table. A must be in the format described in the section Input/Output of the documentation for class `number_field` (see page 473) .

`ct order (const bigint_matrix & `A`, const bigint & `d`,`
`nf_base * `$base$` = nf_base::current_base)`

initializes with the order generated by the columns of $\frac{1}{d} \cdot B \cdot A$, where B is the base pointed to by $base$. Note that due to the automatic casts you may also use an `order` or a `number_field` as third argument. If no third argument is given, the current base is used, which is the base of the `number_field` or the `order` that was constructed or read the last before invoking this constructor. If no `number_field` or `order` has been used so far, this defaults to a (mostly useless) dummy field.

`ct order (const number_field & F)`
initializes with the maximal `order` of the `number_field` F.

`ct order (const order & O)`
initializes with a copy of the order $\mathcal{O}$.

`dt ~order ()`

Assignments

Let $\mathcal{O}$ be of type `order`. The operator `=` is overloaded. For consistency, the following function is also implemented:

`void O.assign (const order & Q)`
$\mathcal{O} \leftarrow \mathcal{Q}$.

Access Methods

Let $\mathcal{O}$ be an order in $\mathbb{Z}[X]/(p)$ for a polynomial p and let

$$(w_1, \ldots, w_n) = \frac{1}{d}(1, x, \ldots, x^{n-1}) \cdot T$$

be the basis of $\mathcal{O}$, where d is a `bigint` and T is a `bigint_matrix`. Then MT with

$$w_i \cdot w_j = \sum_{k=1}^{n} MT(i,j,k) w_k$$

is the multiplication table of the order.

`const bigint & O.MT (long i, long j, long k) const`
returns $MT(i,j,k)$.

`const bigint_matrix & O.base_numerator () const`
returns the `bigint_matrix` T described above, if T has been computed. Otherwise it returns a 1×1-matrix containing 0.

`const bigint_matrix & base_numerator (const order & O)`
returns the `bigint_matrix` T described above, if T has been computed. Otherwise it returns a 1×1-matrix containing 0.

`const bigint & O.base_denominator () const`
returns the `bigint` d described above, if T has been computed. Otherwise it returns 1.

`const bigint &  base_denominator (const order & O)`
returns the `bigint` d described above, if T has been computed. Otherwise it returns 1.

`nf_base * O.which_base () const`
returns the pointer to the `nf_base` used to describe $\mathcal{O}$.

`nf_base * which_base (const order & O)`
returns the pointer to the `nf_base` used to describe $\mathcal{O}$.

`const polynomial< bigint > & O.which_polynomial () const`
returns the polynomial p described above.

`const polynomial< bigint > & which_polynomial (const order & O)`
returns the polynomial p described above.

Comparisons

The binary operators `==`, `!=`, `<=`, `<` (true subset), `>=`, and `>` are overloaded and can be used in exactly the same way as in the programming language C++ if the orders to be compared happen to be represented using the same polynomial. If this is not the case the `lidia_error_handler` will be invoked.

Basic Methods and Functions

Let $\mathcal{O}$ be an instance of type `order`.

`long O.degree () const`
returns the rank of $\mathcal{O}$ as module over $\mathbb{Z}$.

`long degree (const order & O)`
returns the rank of $\mathcal{O}$ as module over $\mathbb{Z}$.

`lidia_size_t O.no_of_real_embeddings () const`
returns the number of embeddings of $\mathcal{O}$ into $\mathbb{R}$.

`lidia_size_t no_of_real_embeddings (const order & O)`
returns the number of embeddings of $\mathcal{O}$ into $\mathbb{R}$.

`void swap (order & O, order & Q)`
swaps $\mathcal{O}$ and $\mathcal{Q}$.

`operator alg_ideal () const`
cast operator, which implicitly converts an `order` to an `alg_ideal` over this `order`.

High-Level Methods and Functions

Let $\mathcal{O}$ be an instance of type `order`.

`const bigfloat & O.get_conjugate (lidia_size_t i, lidia_size_t j) const`
or $0 \leq i < \mathcal{O}$`.degree()` and $0 < j \leq \mathcal{O}$`.degree()` this functions returns the j-th conjugate of the i-th element of the basis representing $\mathcal{O}$. If i or j are not within the indicated bounds, the `lidia_error_handler` will be invoked. Note that the order of the conjugates is randomly determined, but once it is determined, it remains fixed. The first $\mathcal{O}$`.no_of_real_embeddings()` conjugates always correspond to the real embeddings of $\mathcal{O}$.

`const math_matrix< bigfloat > & `$\mathcal{O}$`.get_conjugates () const`

returns a reference to the matrix that is used internally to store the conjugates. The i-th column of the matrix contains the conjugates of the i-th element of the basis representing $\mathcal{O}$.

`const math_vector< bigint > & `$\mathcal{O}$`.get_one () const`

returns the representation of the rational number 1 in $\mathcal{O}$.

`bigint disc (const order & `$\mathcal{O}$`)`

returns the discriminant of $\mathcal{O}$.

`module `$\mathcal{O}$`.pseudo_radical (const bigint & `p`) const`

returns the p-radical, if p is a prime number. Otherwise, this routine returns the so-called pseudo-radical (see [14]).

`int `$\mathcal{O}$`.dedekind (const bigint & `p`, polynomial< bigint > & `h`) const`

returns 1, if p is an index divisor of the equation order of the polynomial used to describe the base of $\mathcal{O}$. In this case $h(x)$ and p generate the p-radical, where x is a root of the generating polynomial of the field. If p is not a prime number the behaviour of this function is undefined.

`order `$\mathcal{O}$`.maximize (const bigint & `m`) const`

returns an order which is maximized at all primes which divide m but whose squares don't divide m.

`order `$\mathcal{O}$`.maximize () const`

returns the maximal order containing $\mathcal{O}$. Since this involves factoring the discriminant of the order, this might take a fairly long time.

Input/Output

`istream` operator `>>` and `ostream` operator `<<` are overloaded. Input and output of an `order` support exactly the same formats as input and output of a `number_field`, so see the description of `number_field` for the (lengthy) description of these formats (page 473).

Warnings

Some of the functions and arguments described as `const` in this documentation are only `const`, if your compiler supports the new ANSI C++ keyword "`mutable`". Otherwise you might get some warnings while compiling your own programs. For more detailed information which functions are concerned have a look at `LiDIA/alg_number.h` and look for the `LIDIA_NO_MUTABLE` conditionals.

See also

`bigint`, `base_vector`, `module`, `alg_number`, `polynomial< bigint >`

Examples

```
#include <LiDIA/alg_number.h>

int main()
```

```
{
    order O;                            // a dummy order
    number_field F;                     // a dummy field
    cin >> O;                           // Now the base of O is current_base
    alg_number a = bigint(1);
    cout << O;
    O = O.maximize();                   // Now the current_base is the base
                                        // of the maximal order!

    for (long zaehl = 1; zaehl <= 2; zaehl++) {
        alg_number one = bigint(1);
        cout << "one = " << one << endl;
        cout << "one + 1 = " << (one + 1) << endl;

        alg_number two = bigint(2);
        cout << "two = " << two << endl;

        if (zaehl != 2){
            cin >> F;                   // now the base of F is the current_base
            O.assign(order(F));         // O is the maximal order of F;
           // The numbers used in the next iteration will be numbers in order(F)!!
        }
        else cout << endl;
    }

    return 0;
}
```

Author

Stefan Neis

alg_number

Name

`alg_number` ..... arithmetic for algebraic numbers

Abstract

`alg_number` is a class for doing multiprecision arithmetic for algebraic numbers. It supports for example arithmetic operations, comparisons, and computations of norm and trace.

Description

An `alg_number` consists of a triple $(num, den, \mathcal{O})$, where the numerator $num = (num_0, \ldots, num_{n-1})$ is a `math_vector< bigint >` of length n, the denominator den is a `bigint`, and $\mathcal{O}$ is a pointer to the `nf_base`, that is used to represent the number. The `alg_number` $(num, den, \mathcal{O})$ represents the algebraic number

$$\frac{1}{den} \sum_{i=0}^{n-1} num_i w_i \ ,$$

where w_i are the base elements described by the `nf_base` pointed to by $\mathcal{O}$. The components of the numerator num and the denominator den of an `alg_number` are always coprime, the denominator den is positive.

Constructors/Destructor

In the following constructors there is an optional pointer to an `nf_base`. Instead of this pointer, a `number_field` or an `order` may be given, since there are suitable automatic casts. If however, this argument is completely omitted, the number is generated with respect to `nf_base::current_base`, which is the base of the `number_field` or `order` that was constructed or read last. If no `number_field` or `order` has been constructed or read so far, this defaults to a dummy base. Computations with algebraic numbers over this dummy base will lead to unpredictable results or errors, if you do not assign meaningful values to such algebraic numbers before using them.

`ct alg_number (nf_base * ` $\mathcal{O}$ `= nf_base::current_base)`

initializes with zero with respect to the given base.

`ct alg_number (const bigint &, nf_base * ` $\mathcal{O}_1$ `= nf_base::current_base)`

lifts the given `bigint` algebraic number field described by the given `nf_base`.

`ct alg_number (const base_vector< bigint > & ` v`, const bigint & ` d `= 1,`
`nf_base * ` $\mathcal{O}_1$ `= nf_base::current_base)`

initializes with the algebraic number $\frac{1}{d} \sum_{i=0}^{n-1} v[i] w_i$ if v has exactly n components. Otherwise the `lidia_error_handler` will be invoked.

`ct alg_number (const bigint *& `v`, const bigint & `d` = 1, nf_base * `$\mathcal{O}_1$` = nf_base::current_base)`

initializes with the algebraic number $\frac{1}{d}\sum_{i=0}^{n-1} v[i]w_i$. The behaviour of this constructor is undefined if the array v has less than n components.

`ct alg_number (const alg_number & `a`)`

initializes with a copy of the algebraic number a.

`dt ~alg_number ()`

Assignments

Let a be of type `alg_number`. The operator `=` is overloaded. For efficiency reasons, the following functions are also implemented:

`void `a`.assign_zero ()`

$a \leftarrow 0$. Note that this embedds zero into the number field which a is a member of.

`void `a`.assign_one ()`

$a \leftarrow 1$. Note that this embedds one into the number field which a is a member of.

`void `a`.assign (const bigint & `b`)`

$a \leftarrow b$. Note that this embedds b into the number field which a is a member of.

`void `a`.assign (const alg_number & `b`)`

$a \leftarrow b$.

Access Methods

Let a be of type `alg_number` with $a = (num, den, \mathcal{O})$.

`const math_vector< bigint > & `a`.coeff_vector () const`

returns the coefficient vector num of a.

`const math_vector< bigint > & coeff_vector (const alg_number & `a`)`

returns the coefficient vector num of a.

`alg_number `a`.numerator () const`

returns the `alg_number` $(num, 1, \mathcal{O})$.

`alg_number numerator (const alg_number & `a`)`

returns the `alg_number` $(num, 1, \mathcal{O})$.

`const bigint & `a`.denominator () const`

returns the denominator den of a.

`const bigint & denominator (const alg_number & `a`)`

returns the denominator den of a.

`nf_base * a.which_base () const`
returns the pointer $\mathcal{O}$.

`nf_base * which_base (const alg_number & a)`
returns the pointer $\mathcal{O}$.

Arithmetical Operations

The following operators are overloaded and can be used in exactly the same way as in the programming language C++. If you use the binary operators on two algebraic numbers a and b, this is only successful if `a.which_order()` $=$ `b.which_order()`, otherwise the `lidia_error_handler` will be invoked.

unary		*op*	`alg_number`	*op* $\in \{$`-`$\}$
binary	`alg_number`	*op*	`alg_number`	*op* $\in \{$`+, -, *, /`$\}$
binary with assignment	`alg_number`	*op*	`alg_number`	*op* $\in \{$`+=, -=, *=, /=`$\}$
binary	`alg_number`	*op*	`bigint`	*op* $\in \{$`+, -, *, /`$\}$
binary	`bigint`	*op*	`alg_number`	*op* $\in \{$`+, -, *, /`$\}$
binary with assignment	`alg_number`	*op*	`bigint`	*op* $\in \{$`+=, -=, *=, /=`$\}$

To avoid copying all operators also exist as functions.

`void add (alg_number & c, const alg_number & a, const alg_number & b)`
$c \leftarrow a + b$ if a and b are members of the same order. Otherwise the `lidia_error_handler` will be invoked.

`void add (alg_number & c, const alg_number & a, const bigint & i)`
$c \leftarrow a + i$.

`void add (alg_number & c, const bigint & i, const alg_number & b)`
$c \leftarrow i + b$.

`void subtract (alg_number & c, const alg_number & a, const alg_number & b)`
$c \leftarrow a - b$ if a and b are members of the same order. Otherwise the `lidia_error_handler` will be invoked.

`void subtract (alg_number & c, const alg_number & a, const bigint & i)`
$c \leftarrow a - i$.

`void subtract (alg_number & c, const bigint & i, const alg_number & b)`
$c \leftarrow i - b$.

`void multiply (alg_number & c, const alg_number & a, const alg_number & b)`
$c \leftarrow a \cdot b$ if a and b are members of the same order. Otherwise the `lidia_error_handler` will be invoked.

`void multiply (alg_number & c, const alg_number & a, const bigint & i)`
$c \leftarrow a \cdot i$.

`void multiply (alg_number & c, const bigint & i, const alg_number & b)`
$c \leftarrow i \cdot b$.

`void divide (alg_number & c, const alg_number & a, const alg_number & b)`
$c \leftarrow a/b$ if $b \neq 0$ and if a and b are members of the same order. Otherwise the `lidia_error_handler` will be invoked.

`void divide (alg_number & c, const alg_number & a, const bigint & i)`
$c \leftarrow a/i$ if $i \neq 0$. Otherwise the `lidia_error_handler` will be invoked.

`void divide (alg_number & c, const bigint & i, const alg_number & b)`
$c \leftarrow i/b$ if $b \neq 0$. Otherwise the `lidia_error_handler` will be invoked.

`void` a`.negate ()`
$a \leftarrow -a$.

`void negate (alg_number & a, const alg_number & b)`
$a \leftarrow -b$.

`void` a`.multiply_by_2 ()`
$a \leftarrow 2 \cdot a$ (done by shifting).

`void` a`.divide_by_2 ()`
$a \leftarrow a/2$ (done by shifting).

`void` a`.invert ()`
$a \leftarrow 1/a$ if $a \neq 0$. Otherwise the `lidia_error_handler` will be invoked.

`void invert (alg_number & c, const alg_number & a)`
$c \leftarrow 1/a$ if $a \neq 0$. Otherwise the `lidia_error_handler` will be invoked.

`alg_number inverse (const alg_number & a)`
returns $1/a$ if $a \neq 0$. Otherwise the the `lidia_error_handler` will be invoked.

`void square (alg_number & c, const alg_number & a)`
$c \leftarrow a^2$.

`void power (alg_number & c, const alg_number & a, const bigint & i)`
$c \leftarrow a^i$.

`void power_mod_p (alg_number & c, const alg_number & a, const bigint & i,`
`const bigint & p)`
$c \leftarrow a^i \bmod p$, i.e. every component of the result is reduced modulo p. We assume, that a has denominator 1.

Comparisons

The binary operators `==`, `!=`, and the unary operator `!` (comparison with zero) are overloaded and can be used in exactly the same way as in the programming language C++ if the numbers to be compared are members of the same order. Otherwise the `lidia_error_handler` will be invoked.

In addition we offer the following functions for frequently needed special cases.

Let a be an instance of type `alg_number`.

`bool` a`.is_zero () const`
returns `true` if $a = 0$, `false` otherwise.

`bool` a`.is_one () const`
returns `true` if $a = 1$, `false` otherwise.

Basic Methods and Functions

Let $a = (num, den, \mathcal{O})$ be an instance of type `alg_number`.

`lidia_size_t` a`.degree () const`
returns the degree of the number field described by the base pointed to by $\mathcal{O}$.

`lidia_size_t degree (const alg_number &` a`)`
returns the degree of the number field described by the base pointed to by $\mathcal{O}$.

`void` a`.normalize ()`
normalizes the `alg_number` a such that the gcd of the elements representing the numerator and the denominator is 1 and that the denominator is positive.

`void swap (alg_number &` a`, alg_number &` b`)`
exchanges the values of a and b.

High-Level Methods and Functions

Let a be an instance of type `alg_number`.

`bigfloat` a`.get_conjugate (lidia_size_t j) const`
for $0 < j \leq \deg(\mathcal{O})$ this functions returns the j-th conjugate of a. If j is not within the indicated bounds, the `lidia_error_handler` will be invoked.

`math_vector< bigfloat >` a`.get_conjugates () const`
returns the vector of conjugates of a.

`bigint_matrix rep_matrix (const alg_number &` a`)`
returns the representation matrix of the numerator of a, i.e. the matrix of the endomorphism $K \rightarrow K, x \mapsto a \cdot den(\mathrm{a}) \cdot \mathrm{x}$ with respect to the basis of the number field K pointed to by $\mathcal{O}$.

`bigrational norm (const alg_number &` a`)`
returns the norm of a.

`bigrational trace (const alg_number &` a`)`
returns the trace of a.

`polynomial< bigint > charpoly (const alg_number &` a`)`
returns the characteristic polynomial of a.

Input/Output

`istream` operator `>>` and `ostream` operator `<<` are overloaded. Input and output of an `alg_number` have the following format: $[a_0 \dots a_{n-1}]/den$.

If the denominator is 1 we omit it, i.e. in this case the format is $[a_0 \dots a_{n-1}]$.

Note that you have to manage by yourself that successive `alg_number`s may have to be separated by blanks.

Since we didn't want to read the $\mathcal{O}$-component of each number, we always assume, that we are reading a representation with coefficients relative to the base pointed to by `nf_base::current_base`. By default, `nf_base::current_base` points to the basis of the last number field or order that was read or constructed, but you may set it to previously defined bases,e.g. by calling an constructor of the class `order` with suitable arguments. If `nf_base::current_base` was not set before, the number will use some useless dummy base, which will produce unpredictable results, if you use such a number in a computation before assigning a meaningful value to it. (Note: Exactly the same mechanism using the same variables is used for reading and writing modules and ideals).

See also

`bigint`, `module`, `ideal`, `number_field`, `order`

Examples

For an example please refer to `LiDIA/src/packages/alg_number/alg_number_appl.cc`

Author

Stefan Neis

module/alg_ideal

Name

`module/alg_ideal` ..... free $\mathbb{Z}$-modules and ideals in algebraic number fields

Abstract

`module` is a class for doing module arithmetic in number fields or more precisely in orders. `alg_ideal` is a class for doing ideal arithmetic. They support for example arithmetic operations (including pseudo-division), comparisons, and the computation of the ring of multipliers of a module.

Both classes are essentially the same, however for ideals multiplication and divsion can use faster algorithms than for modules.

Description

A `module` as well as an `alg_ideal` consists of a triple $(num, den, \mathcal{O})$, where the numerator $num = (\underline{num}_0, \ldots, \underline{num}_{m-1})$ is a `bigmod_matrix` of size $n \times m$ (n is the degree of the number field) with coefficients from $\mathbb{Z}/M\mathbb{Z}$ for some M, the denominator den is a `bigint`, and $\mathcal{O}$ is a pointer to the `nf_base`, that is used to represent the module. The `module`(`alg_ideal`) $(num, den, \mathcal{O})$ represents the module (ideal)

$$\sum_{j=0}^{m-1} \frac{1}{den} \left(\sum_{i=0}^{n-1} num_{i,j} w_i \right) \cdot \mathbb{Z} + M\mathcal{O} \ ,$$

where w_i are the base elements of the `nf_base` pointed to by $\mathcal{O}$. The components of the numerator num and the denominator den of a `module` are always coprime, the denominator den is positive.

Note that since each ideal is a module as well, `alg_ideal` inherits from `module`, so that you can use all functions described for modules for `alg_ideals` as well. Note however, that a couple of extra functions are available for `alg_ideals`.

Constructors/Destructor

`ct module (nf_base * `$\mathcal{O}_1$` = nf_base::current_base)`

initializes with the zero ideal.

`ct module (const alg_number & `a`, const alg_number & `b` = 0)`

initializes with the ideal generated by the algebraic numbers a and b. This constructor will be removed in the next release.

`ct module (const base_matrix< bigint > & A, const bigint & d = 1,`
`nf_base * O1 = nf_base::current_base)`

initializes the module with the ideal(!) generated by the columns of A divided by d. If the number of rows of A does not match the degree of the number field with the base $\mathcal{O}_1$ is pointing to, the `lidia_error_handler` will be invoked. The behaviour of this constructor will change in the next release. Please use the class `alg_ideal` for representing ideals.

`ct module (const bigmod_matrix & A, const bigint & d = 1,`
`nf_base * O1 = nf_base::current_base)`

initializes with the $\mathbb{Z}$-module generated by the columns of A and the columns of $m \cdot I$ divided by d. Here m denotes the modulus of the `bigmod_matrix` and I denotes an identity matrix with as many rows as A. If the number of rows of A does not match the degree of the number field with the base $\mathcal{O}_1$ is pointing to, the `lidia_error_handler` will be invoked.

`ct module (const module & M)`

initializes with a copy of the module M.

`dt ~module ()`

`ct alg_ideal (nf_base * O1 = nf_base::current_base)`

initializes with the zero ideal.

`ct alg_ideal (const bigint & a, const alg_number & b = 0)`

initializes with the ideal generated by the integer a and the algebraic number b. Note that this will generate an ideal over the order, in which b is contained. Especially, if b is omitted, it will be an ideal with respect to the base pointed to by `nf_base::current_base`.

`ct alg_ideal (const alg_number & a, const alg_number & b = 0)`

initializes with the ideal generated by the algebraic numbers a and b.

`ct alg_ideal (const base_matrix< bigint > & A, const bigint & d = 1,`
`nf_base * O1 = nf_base::current_base)`

initializes with the ideal generated by the columns of A (interpreted as algebraic numbers) divided by d. If the number of rows of A does not match the degree of the number field with the base $\mathcal{O}_1$ is pointing to, the `lidia_error_handler` will be invoked.

`ct alg_ideal (const bigmod_matrix & A, const bigint & d = 1,`
`nf_base * O1 = nf_base::current_base)`

initializes with the $\mathbb{Z}$-module generated by the columns of A and the columns of $m \cdot I$ divided by d. Here m denotes the modulus of the `bigmod_matrix` and I denotes an identity matrix with as many rows as A. If the number of rows of A does not match the degree of the number field with the base $\mathcal{O}_1$ is pointing to, the `lidia_error_handler` will be invoked. If the resulting module does not have full rank, we use the ideal generated by the numbers of the module, otherwise we assume that the matrix really describes an ideal. If this is not true, operations using this ideal may lead to wrong results!

`ct alg_ideal (const alg_ideal & M)`

initializes with a copy of the ideal M.

`dt ~alg_ideal ()`

Assignments

Let M be of type `module` or `alg_ideal`. The operator `=` is overloaded. For efficiency reasons, the following functions are also implemented:

`void` M`.assign_zero ()`
$M \leftarrow (0)$, i.e. M is set to the zero ideal.

`void` M`.assign_one ()`
$M \leftarrow (1)$, i.e. M is set to the whole order generated by the base elements of the `nf_base`.

`void` M`.assign_whole_order ()`
$M \leftarrow (1)$.

`void` M`.assign (const bigint &` b`)`
$M \leftarrow (b)$, i.e. M is set to the principal ideal generated by b.

`void` M`.assign (const alg_number &` b`)`
$M \leftarrow (b)$.

`void` M`.assign (const module &` N`)`
if M is of type `alg_ideal`, then M is set to the ideal generated by the elements of N, otherwise M is set to N.

`void` M`.assign (const alg_ideal &` N`)`
$M \leftarrow N$.

Access Methods

Let M be of type `module` or `alg_ideal` with $M = (num, den, \mathcal{O})$.

`const bigmod_matrix &` M`.coeff_matrix () const`
returns the numerator *num* of the description of M.

`const bigmod_matrix & coeff_matrix (const module/alg_ideal &` M`)`
returns the numerator *num* of the description of M.

`module/alg_ideal` M`.numerator () const`
Depending on the type of M, this returns either the `module` or the `alg_ideal` $(num, 1, \mathcal{O})$.

`module/alg_ideal numerator (const module/alg_ideal &` M`)`
Depending on the type of M, this returns either the `module` or the `alg_ideal` $(num, 1, \mathcal{O})$.

`const bigint &` M`.denominator () const`
returns the denominator *den* of the description of M.

`const bigint & denominator (const module/alg_ideal &` M`)`
returns the denominator *den* of the description of M.

`nf_base *` M`.which_base () const`
returns the pointer $\mathcal{O}$ of the description of M.

`nf_base * which_base (const module/alg_ideal &` M`)`
returns the pointer $\mathcal{O}$ of the description of M.

Arithmetical Operations

The following operators are overloaded and can be used in exactly the same way as in the programming language C++. If you use these operators on two modules/ideals a and b, this is only successful if `a.which_base()` = `b.which_base()`, otherwise the `lidia_error_handler` will be invoked.

`(binary) +, &, *, /`
`(bin.~with assignment) +=, &=, *=, /=`

Note: By M `&` N we denote the intersection of the modules M and N.

To avoid copying all operators also exist as functions:

`void add (module & c, const module & a, const module & b)`
$c \leftarrow a + b$, if a and b are over the same order. Otherwise the `lidia_error_handler` will be invoked.

`void intersect (module & c, const module & a, const module & b)`
$c \leftarrow a \cap b$, if a and b are over the same order. Otherwise the `lidia_error_handler` will be invoked.

`void multiply (module & c, const module & a, const module & b)`
$c \leftarrow a \cdot b$, if a and b are over the same order. Otherwise the `lidia_error_handler` will be invoked.

`void multiply (module & c, const module & a, const bigint & i)`
$c \leftarrow a \cdot i$.

`void multiply (module & c, const bigint & i, const module & b)`
$c \leftarrow i \cdot b$.

`void multiply (alg_ideal & c, const alg_ideal & a, const alg_ideal & b)`
$c \leftarrow a \cdot b$, if a and b are over the same order. Otherwise the `lidia_error_handler` will be invoked. Note that the result will be the same if the multiply function for modules is used, however this special case should be faster.

`void multiply (alg_ideal & c, const alg_ideal & a, const alg_number & b)`
$c \leftarrow a \cdot b$, if a and b are over the same order. Otherwise the `lidia_error_handler` will be invoked.

`void multiply (alg_ideal & c, const alg_number & b, const alg_ideal & a)`
$c \leftarrow a \cdot b$, if a and b are over the same order. Otherwise the `lidia_error_handler` will be invoked.

`void divide (module & c, const module & a, const module & b)`
$c \leftarrow a/b$, if $b \neq 0$ and if a and b are over the same order $\mathcal{O}$. Otherwise the `lidia_error_handler` will be invoked. If b is not invertible, we compute the maximal module c with $b \cdot c \subset a$. Note that although this may seem a bit strange, it is a natural way to extend the notion of ideal division.

`void divide (module & c, const module & a, const bigint & i)`
$c \leftarrow a/i$, if $i \neq 0$. Otherwise the `lidia_error_handler` will be invoked.

`void divide (alg_ideal & c, const alg_ideal & a, const alg_ideal & b)`
$c \leftarrow a/b$, if $b \neq 0$ and if a and b are over the same order $\mathcal{O}$. Otherwise the `lidia_error_handler` will be invoked. If b is not invertible, we compute the maximal module c with $b \cdot c \subset a$. Note that although this may seem a bit strange, it is a natural way to extend the notion of ideal division.

`void divide (alg_ideal & c, const alg_ideal & a, const alg_number & b)`
$c \leftarrow a/b$, if $b \neq 0$. Otherwise the `lidia_error_handler` will be invoked.

`void a.invert ()`
$a \leftarrow 1/a$, if a is an invertible ideal. Otherwise we set a to the module $a' := \{x \in \mathbb{K} : x \cdot a \subseteq \mathcal{O}\}$.

`void invert (module & a, const module & b)`
$a \leftarrow 1/b$, if b is an invertible ideal. Otherwise a is set to $b' := \{x \in \mathbb{K} : x \cdot b \subseteq \mathcal{O}\}$.

`module inverse (const module & a)`
returns $1/a$, if a is an invertible ideal. Otherwise we return the module $a' := \{x \in \mathbb{K} : x \cdot a \subseteq \mathcal{O}\}$.

`alg_ideal inverse (const alg_ideal & a)`
returns $1/a$, if a is an invertible ideal. Otherwise we return the ideal $a' := \{x \in \mathbb{K} : x \cdot a \subseteq \mathcal{O}\}$.

`void square (module & c, const module & a)`
$c \leftarrow a^2$.

`void power (module & c, const module & a, const bigint & i)`
$c \leftarrow a^i$.

Comparisons

The binary operators `==`, `!=`, `<=`, `<` (true subset), `>=`, `>`, and the unary operator `!` (comparison with zero) are overloaded and can be used in exactly the same way as in the programming language C++ if the modules/ideals to be compared are given relative to the same basis. Otherwise the `lidia_error_handler` will be invoked. The operators may also be used to compare modules with ideals.

Let M be an instance of type `module` or `alg_ideal`.

`bool` M`.is_zero () const`
returns `true` if $M = (0)$, `false` otherwise.

`bool` M`.is_one () const`
returns `true` if $M = (1)$, `false` otherwise.

`bool` M`.is_whole_order () const`
returns `true` if $M = (1)$, `false` otherwise.

Basic Methods and Functions

Let $M = (num, den, \mathcal{O})$ be an instance of type `module` or `alg_ideal`.

`bigint_matrix` M`.z_basis () const`
returns a $\mathbb{Z}$-basis of the module or ideal M.

`bigint_matrix z_basis (const module/alg_ideal &` M`) const`
returns a $\mathbb{Z}$-basis of the module or ideal M.

`long` M`.degree () const`

returns the rank of the order pointed to by $\mathcal{O}$ as module over $\mathbb{Z}$.

`long degree (const module/alg_ideal &` M`)`

returns the rank of the order pointed to by $\mathcal{O}$ as module over $\mathbb{Z}$.

`void` M`.normalize ()`

normalizes the `module` or `alg_ideal` M such that the gcd of the elements representing the numerator and the denominator is 1 and that the denominator is positive.

`void swap (module/alg_ideal &` M`, module/alg_ideal &` N`)`

exchanges the values of M and N. Note that using swap on parameters of different type will lead to unpredictable results, i.e. the compiler does not (yet) complain.

High-Level Methods and Functions

Let M be an instance of type `module` or `alg_ideal`.

`bigrational norm (const module/alg_ideal &` M`)`

returns the norm of M.

`bigrational exp (const module/alg_ideal &` M`)`

returns the exponent of M, i.e. if $M = (num, den, \mathcal{O})$ it computes the smallest integer e, such that $e \cdot (*\mathcal{O}) \subseteq den \cdot M$ and then it returns $\frac{e}{den}$.

`order` M`.ring_of_multipliers (const bigint &` p`) const`

returns the ring of multipliers of the module M modulo p. Note that it is not necessary, that p is prime.

Input/Output

`istream` operator `>>` and `ostream` operator `<<` are overloaded. Input and output of a `module` or an `alg_ideal` have the following format: [`bigint_matrix`]/den, where [`bigint_matrix`] denotes a matrix given in any format understood by the class `bigint_matrix`. If the denominator is 1, we omit it, i.e. in this case the formats are the same as for the class `bigint_matrix`.

Note that you have to manage by yourself that a `module` or an `alg_ideal` may have to be separated by blanks from other input or output.

Since we didn't want to read the $\mathcal{O}$-component of each module/ideal, we always assume, that we are reading a representation with coefficients relative to the base pointed to by `nf_base::current_base`. By default, `nf_base::current_base` points to the basis of the last number field or order that was read or constructed, but you may set it to previously defined bases e.g. by calling a constructor of the class `order` with suitable arguments. If `nf_base::current_base` was not set before, the module will use some useless dummy base, which will produce unpredictable results, if you use such a module in a computation before assigning a meaningful value to it. (Note: Exactly the same mechanism using the same variables is used for reading and writing algebraic numbers).

Warnings

For backward compatibility some of the constructors do not yet behave as one would expect, for example `module(a,b)` with to `alg_numbers` a and b should construct the module $a\mathbb{Z} + b\mathbb{Z}$, but in fact it does construct

the ideal generated by a and b, i.e. the module $a\mathcal{O} + b\mathcal{O}$. However we do recommend to use the new class `alg_ideal` for this purpose, which will support this constructor also in the future (`module` will not !!).

See also

`bigint`, `bigmod_matrix`, `alg_number`, `order`

Examples

For further references please refer `LiDIA/src/packages/alg_number/module_appl.cc`

Author

Stefan Neis

prime_ideal

Name

`prime_ideal` ..... prime ideals in algebraic number fields

Abstract

`prime_ideal` is a class for representing prime ideals in algebraic number fields. Since each prime ideal obviously is an ideal, you can do everything you can do with an `alg_ideal` also with a `prime_ideal`. This is realised by a cast operator, which implicitly converts a `prime_ideal` to an `alg_ideal`. Note especially, that operations on `prime_ideal`s always have an `alg_ideal` as result.

However this class offers some additional functionality which either makes no sense for a general `alg_ideal` or would be of much less interest in the general case.

Description

A `prime_ideal` consists of a prime p (represented by a `bigint`) and an `alg_number` a which especially contains the pointer to an `nf_base` B, such that B is the basis of the `order` O over which the prime ideal is defined and that p and a generate the prime ideal as O-ideal. Additionaly the `lidia_size_t`s e and f contain the ramification index and the degree of inertia of the prime ideal, respectively. An additional `alg_number` contains information needed to compute valuations.

Constructors/Destructor

`ct prime_ideal ()`

constructs a dummy prime ideal initialized with useless values. Don't try to do any computation with such a prime ideal!

`ct prime_ideal (const bigint & p, const alg_number & a, lidia_size_t e = 0, lidia_size_t f = 0)`

initializes with the prime ideal generated by the prime p and the algebraic number a. If e and f are not given, their values are computed. If incorrect data is given to this constructor (e.g. if p and a do not generate a prime ideal or if the values given for e and f are wrong), this will result in unpredictable behaviour.

`ct prime_ideal (const bigint & p, const nf_base * O = nf_base::current_base)`

initializes with a prime p that is inert over the order generated by the `nf_base` pointed to by $\mathcal{O}$. If p is not an inert prime, this will result in unpredictable behaviour.

`dt ~prime_ideal ()`

Access Methods

Let P be of type `prime_ideal` with $P = (p, a, \mathcal{O})$.

`const bigint & P.first_generator () const`
returns the prime p of the description of P.

`const alg_number & P.second_generator () const`
returns the algebraic number a of the description of P.

`lidia_size_t P.ramification_index () const`
returns the ramification index of the prime ideal P.

`lidia_size_t P.degree_of_inertia () const`
returns the degree of inertia of the prime ideal P.

`const bigint & P.base_prime () const`
returns the prime p which is divided by P, i.e. the function returns the first generator.

Basic Methods and Functions

`operator alg_ideal () const`
cast operator, which implicitly converts a `prime_ideal` to an `alg_ideal` over the same `order`.

`void swap (prime_ideal & a, prime_ideal & b)`
exchanges the values of a and b.

High-Level Methods and Functions

`long ord (const prime_ideal & P, const alg_ideal & M)`
returns $\mathrm{ord}_P(M)$, i.e. the maximal power of P which divides M.

`long ord (const prime_ideal & P, const alg_number & a)`
returns $\mathrm{ord}_P(a)$, i.e. the maximal power of P which divides the principal ideal generated by a.

Input/Output

`istream` operator `>>` and `ostream` operator `<<` are overloaded. Input and output of a `prime_ideal` have the following format: $< p, a >$, where p and a are the generators of the prime ideal. In this representation p is a `bigint` and a is an `alg_number`. If a is 0, output will be simplified to the format $< p >$.

Warnings

Due to a last minute redesign, this class lacks several functions that it should have for compatibility with the class `alg_ideal`. This will be fixed in the next version.

See also

`alg_number`, `alg_ideal`, `order`

Author

Stefan Neis

Chapter 18

Factorization

The description of the general factorization template class can be found in the section “LiDIA base package” (see page 181).

single_factor< alg_ideal >

Name

`single_factor< alg_ideal >` ..... a factor of an ideal in an algebraic number field

Abstract

`single_factor< alg_ideal >` is used for storing factorizations of ideals of algebraic number fields (see `factorization`). It is a specialization of `single_factor< T >` with some additional functionality. All functions for `single_factor< T >` can be applied to objects of class `single_factor< alg_ideal >`, too. These basic functions are not described here any further; you will find the description of the latter in `single_factor< T >`.

Description

Access Methods

Let a be an instance of type `single_factor< alg_ideal >`.

`prime_ideal` a`.prime_base () const`

If a`.is_prime_factor()` returns `true`, then a represents a `prime_ideal` which is returned by this function. If a`.is_prime_factor()` returns `false`, calling this function will result in unpredictable behaviour.

High-Level Methods and Functions

Let a be an instance of type `single_factor< alg_ideal >`.

Queries

`bool` a`.is_prime_factor (int` *test* `= 0)`

Note that contrary to the general description, calling this routine with *test* $\neq 0$ will not perform an explicit test if we do not yet know, whether or not a is prime, since no efficient test is implemented. Instead, this will result in an error message. If you need to know, whether or not a `single_factor< alg_ideal >` is prime, call this function without parameter and if the result is not "prime", call the function `factor` and check the factorization.

Factorization Algorithms

If one tries to factor an integral ideal of an algebraic number field, the hard part is to find a factorization of the smallest strictly positive integer in this ideal. If this factorization has been succesfully computed, the remaining computations can be done in (probabilistic) polynomial time as described e.g. in [17], chapter 4.8.2 (for numbers not dividing the index) and [14] or [63] (for index divisors).

For fractional ideals, one needs to factor the denominator in addition, however, this can be handled separately.

Therefore the generic factorization functions are

`factorization< alg_ideal > finish (const alg_ideal & `*a*`, rational_factorization & `*f*`)`
: returns the factorization of the numerator of *a*. *f* must contain a factorization of the smallest strictly positive integer in this ideal.

`factorization< alg_ideal > `*a*`.finish (rational_factorization & `*f*`) const`
: returns `finish(`*a*`.base(), `*f*`)`.

To avoid copying objects, there are also the following variations of these calls:

`void finish (factorization< alg_ideal > & `*fact*`, const alg_ideal & `*a*`,`
`rational_factorization & `*f*`)`
: stores the factorization of the numerator of *a* to *fact*. *f* must contain a factorization of the smallest strictly positive integer in this ideal.

`void `*a*`.finish (factorization< alg_ideal > & `*fact*`, rational_factorization & `*f*`) const`
: stores the factorization of the numerator of *a*`.base()` to *fact*. *f* must contain a factorization of the smallest strictly positive integer in this ideal.

In general, our factorization routines use the following strategy: First, we compute the smallest strictly positive integer in an ideal. Then some function of the class `rational_factorization` is called to factor this integer. Finally, we call `finish` to obtain the factorization of the ideal.

To enable the user to use the full power of the factorization routines of class `rational_factorization`, we reuse the interface of this class. Thus we obtain the following functions:

`factorization< alg_ideal > `*a*`.factor (int `*upper_bound*` = 34) const`
: returns the factorization of *a*`.base()` using the function `factor` of class `rational_factorization` with the given parameter to compute the factorization of the smallest strictly positive integer in the ideal.

`factorization< alg_ideal > factor (const alg_ideal & `*a*`, int `*upper_bound*` = 34)`
: returns the factorization of *a* using the function `factor` of class `rational_factorization` with the given parameter to compute the factorization of the smallest strictly positive integer in the ideal.

`void `*a*`.factor (factorization< alg_ideal > & `*fact*`, int `*upper_bound*` = 34) const`
: stores the factorization of *a*`.base()` to *fact* using the function `factor` of class `rational_factorization` with the given parameter to compute the factorization of the smallest strictly positive integer in the ideal.

`void factor (factorization< alg_ideal > & `*fact*`, const alg_ideal & `*a*`, int `*upper_bound*` = 34)`
: stores the factorization of *a* to *fact* using the function `factor` of class `rational_factorization` with the given parameter to compute the factorization of the smallest strictly positive integer in the ideal.

`factorization< alg_ideal > a.trialdiv (unsigned int upper_bound = 1000000, unsigned int lower_bound = 1) const`

returns the factorization of `a.base()` using the function `trialdiv` of class `rational_factorization` with the given parameters to compute the factorization of the smallest strictly positive integer in the ideal.

`factorization< alg_ideal > trialdiv (const alg_ideal & a, unsigned int upper_bound = 1000000, unsigned int lower_bound = 1)`

returns the factorization of a using the function `trialdiv` of class `rational_factorization` with the given parameters to compute the factorization of the smallest strictly positive integer in the ideal.

`void a.trialdiv (factorization< alg_ideal > & fact, unsigned int upper_bound = 1000000, unsigned int lower_bound = 1) const`

stores the factorization of `a.base()` to *fact* using the function `trialdiv` of class `rational_factorization` with the given parameters to compute the factorization of the smallest strictly positive integer in the ideal.

`void trialdiv (factorization< alg_ideal > & fact, const alg_ideal & a, unsigned int upper_bound = 1000000, unsigned int lower_bound = 1)`

stores the factorization of a to *fact* using the function `trialdiv` of class `rational_factorization` with the given parameters to compute the factorization of the smallest strictly positive integer in the ideal.

`factorization< alg_ideal > a.ecm (int upper_bound = 34, int lower_bound = 6, int step = 3) const`

returns the factorization of `a.base()` using the function `ecm` of class `rational_factorization` with the given parameters to compute the factorization of the smallest strictly positive integer in the ideal.

`factorization< alg_ideal > ecm (const alg_ideal & a, int upper_bound = 34, int lower_bound = 6, int step = 3)`

returns the factorization of a using the function `ecm` of class `rational_factorization` with the given parameters to compute the factorization of the smallest strictly positive integer in the ideal.

`void a.ecm (factorization< alg_ideal > & fact, int upper_bound = 34, int lower_bound = 6, int step = 3) const`

stores the factorization of `a.base()` to *fact* using the function `ecm` of class `rational_factorization` with the given parameters to compute the factorization of the smallest strictly positive integer in the ideal.

`void ecm (factorization< alg_ideal > & fact, const alg_ideal & a, int upper_bound = 34, int lower_bound = 6, int step = 3)`

stores the factorization of a to *fact* using the function `ecm` of class `rational_factorization` with the given parameters to compute the factorization of the smallest strictly positive integer in the ideal.

`factorization< alg_ideal > a.mpqs () const`

returns the factorization of `a.base()` using the function `mpqs` of class `rational_factorization` to compute the factorization of the smallest strictly positive integer in the ideal.

`factorization< alg_ideal > mpqs (const alg_ideal & a)`

returns the factorization of a using the function `mpqs` of class `rational_factorization` to compute the factorization of the smallest strictly positive integer in the ideal.

`void a.mpqs (factorization< alg_ideal > & fact) const`

stores the factorization of `a.base()` to *fact* using the function `mpqs` of class `rational_factorization` to compute the factorization of the smallest strictly positive integer in the ideal.

`void mpqs (factorization< alg_ideal > & `*fact*`, const alg_ideal & `*a*`)`

stores the factorization of *a* to *fact* using the function `mpqs` of class `rational_factorization` to compute the factorization of the smallest strictly positive integer in the ideal.

See also

`factorization< T >`, `alg_ideal`, `prime_ideal`, `rational_factorization`

Warnings

Using `factor` or `mpqs` requires write permission in the `/tmp` directory or in the directory from which they are called. This is necessary because `mpqs` creates temporary files.

Examples

```
#include <LiDIA/alg_ideal.h>
#include <LiDIA/prime_ideal.h>
#include <LiDIA/factorization.h>

int main()
{

    alg_ideal f;

    factorization< alg_ideal > u;

    cout << "Please enter f : "; cin >> f ;

    u = factor(f);

    cout << "\nFactorization of f :\n" << u << endl;

}
```

Author

Stefan Neis, Anja Steih, Damian Weber

The LiDIA EC package

Chapter 19

Elliptic Curve Related Classes

elliptic_curve_flags

Name

`elliptic_curve_flags` .....class for providing elliptic curve related flags.

Abstract

The class `elliptic_curve_flags` collects the flags, that are necessary for computing with elliptic curves.

Description

`elliptic_curve_flags` is a class that simply collects all flags that can be used when computing with elliptic curves. The flags can be accessed by giving the prefix `elliptic_curve_flags::`, e.g.

```
elliptic_curve_flags::SHORT_W.
```

The flags of type `elliptic_curve_flags::curve_parametrization` are:

```
SHORT_W // Y^2 = X^3 + a4*X + a6 (affine)
        // Y^2 Z = X^3 + a4*X Z^2 + a6 Z^3 (projective)

LONG_W  // Y^2+a1*X*Y+a3*Y = X^3+a2*X^2+a4*X+a6 (affine)
        // Y^2 Z +a1*X*Y*Z+a3*Y*Z^2 =
        // X^3+a2*X^2*Z+a4*X*Z^2+a6 Z^3 (projective)

GF2N_F  // Y^2 +X*Y = X^3 + a2*X^2 + a6 (affine)
        // Y^2 Z + X*Y*Z = X^3 + a2*X^2*Z + a6 Z^3 (projective)
```

The flags of type `elliptic_curve_flags::curve_output_mode` are:

```
SHORT       // [a1,a2,a3,a4,a6,X] only, where X is either
            // "A" (for AFFINE) or "P" for (PROJECTIVE), respectively.
            // If ",X" is omitted the model is affine.
LONG        // as SHORT + b_i, c_j, delta,
            // and more in derived classes;
PRETTY      // equation form
TEX         // equation form in TeX format
```

The flags of type `elliptic_curve_flags::curve_model` are:

```
AFFINE      // affine model
PROJECTIVE  // projective model
```

See also

point< T >, elliptic_curve< T >

Notes

The elliptic curve package of LiDIA will be increased in future. Therefore the interface of the described functions might change in future.

Examples

```
#include <LiDIA/bigint.h>
#include <LiDIA/galois_field.h>
#include <LiDIA/gf_element.h>
#include <LiDIA/elliptic_curve.h>

int main()
{
    bigint p;
    cout << "\n Input prime characteristic : "; cin >> p;

    galois_field F(p);

    gf_element a4, a6;

    a4.assign_zero(F);
    a6.assign_zero(F);

    cout << "\n Coefficient a_4 : "; cin >> a4;
    cout << "\n Coefficient a_6 : "; cin >> a6;

    elliptic_curve< gf_element > e(a4, a6, elliptic_curve_flags::AFFINE);

    cout << "\n Group order is " << e.group_order();

    return 0;
}
```

Author

Birgit Henhapl, John Cremona, Markus Maurer, Volker Müller, Nigel Smart.

elliptic_curve

Name

`elliptic_curve< T >` ..... class for an elliptic curve defined over a field `T`.

Abstract

An elliptic curve is a curve of the form

$$Y^2 + a_1XY + a_3Y = X^3 + a_2X^2 + a_4X + a_6$$

which is non-singular. For some cases also the projective model

$$Y^2Z + a_1XYZ + a_3YZ^2 = X^3 + a_2X^2Z + a_4XZ^2 + a_6Z^3$$

is supported.

Description

`elliptic_curve< T >` is a class for working with elliptic curves defined over arbitrary fields. Therefore the template type `T` is assumed to represent a field. The current version of `elliptic_curve< T >` supports the template types `bigrational`, and `gf_element`. Instances for these classes are already included in the LiDIA library per default. Moreover a specialization for `elliptic_curves< bigint >` is also generated per default.

For the affine model, the class offers three different formats for the elliptic curves: long Weierstrass form as above, short Weierstrass form ($a_1 = a_2 = a_3 = 0$), if the characteristic of the field represented by the template type `T` is odd, and a format especially well suited for elliptic curves defined over fields of characteristic two ($a_1 = 1$ and $a_3 = a_4 = 0$). The currently used form is internally stored, such that points can use optimized algorithms for the group operations (see also `point< T >`). Moreover the curve coefficients a_i are also stored. The projective model is only available for curves over finite fields, i.e. `gf_element`, in short Weierstrass form.

The template class `elliptic_curve< T >` supports a different set of functions depending on the characteristic of the field represented by `T`. If `T` represents a finite field, i.e. `gf_element`, several functions like computing the order of the curve, computing the isomorphism type of the curve, etc. are available in addition to generally available functions.

The template class `elliptic_curve< T >` is closely linked with the template class `point< T >` which is used for holding points on elliptic curves. Therefore it is advised to read also the documentation of that class.

Constructors/Destructor

`ct elliptic_curve< T > ()`

constructs an invalid curve, i.e. all coefficients are set to zero.

`ct elliptic_curve< T > (const T & a, const T & b, elliptic_curve_flags::curve_model m = elliptic_curve_flags::AFFINE)`

If the characteristic of the field is not equal to two, then an elliptic curve in short Weierstrass form is constructed. If however the characteristic is two, then an elliptic curve in the special form for fields of characteristic two is generated (i.e. $a_1 = 1$, $a_2 = a$ and $a_6 = b$). If the discriminant of the elliptic curve is zero, then the `lidia_error_handler` is invoked. The model is affine by default, and projective if m is set to `elliptic_curve_flags::PROJECTIVE`.

`ct elliptic_curve< T > (const T & a1, const T & a2, const T & a3, const T & a4, const T & a6, elliptic_curve_flags::curve_model m = elliptic_curve_flags::AFFINE)`

constructs a curve in long Weierstrass form. If the discriminant of the elliptic curve is zero, then the `lidia_error_handler` is invoked. The model is affine by default, and this is the only supported model at the moment for long Weierstrass form.

`dt ~elliptic_curve< T > ()`

Assignments

The operator `=` is overloaded. In addition, there are the following functions for initializing a curve. Let C be an instance of type `elliptic_curve< T >`.

`void` C`.set_coefficients (const T & a, const T & b, elliptic_curve_flags::curve_model m = elliptic_curve_flags::AFFINE)`

If the characteristic of the field is not equal to two, then an elliptic curve in short Weierstrass form is constructed. If however the characteristic is two, then an elliptic curve in the special form for fields of characteristic two is generated (i.e. $a_1 = 1$, $a_2 = a$ and $a_6 = b$). If the discriminant of the elliptic curve is zero, then the `lidia_error_handler` is invoked. The model is affine by default, and projective if m is set to `elliptic_curve_flags::PROJECTIVE`.

`void` C`.set_coefficients (const T & a1, const T & a2, const T & a3, const T & a4, const T & a6, elliptic_curve_flags::curve_model m = elliptic_curve_flags::AFFINE)`

constructs a curve in long Weierstrass form. If the discriminant of the elliptic curve is zero, then the `lidia_error_handler` is invoked. The model is affine by default, and this is the only supported model at the moment for long Weierstrass form.

Access Methods

Let C be an instance of type `elliptic_curve< T >`.

`T` C`.discriminant () const`

returns the discriminant of C.

`T` C`.get_a1 () const`

returns the coefficient a_1.

`T` C`.get_a2 () const`

returns the coefficient a_2.

`T` C`.get_a3 () const`

returns the coefficient a_3.

`T` C`.get_a4 () const`
returns the coefficient a_4.

`T` C`.get_a6 () const`
returns the coefficient a_6.

`T` C`.get_b2 () const`
returns $b_2 = a_1^2 + 4a_2$.

`T` C`.get_b4 () const`
returns $b_4 = a_1a_3 + 2a_4$.

`T` C`.get_b6 () const`
returns $b_6 = a_3^2 + 4a_6$.

`T` C`.get_b8 () const`
returns $b_8 = (b_2b_6 - b_4^2)/4$.

`T` C`.get_c4 () const`
returns $c_4 = b_2^2 - 24b_4$.

`T` C`.get_c6 () const`
returns $c_6 = -b_2^3 + 36b_2b_4 - 216b_6$.

`void` C`.get_ai (T &` a_1`, T &` a_2`, T &` a_3`, T &` a_4`, T &` a_6`) const`
gets all the values of the a_i's at once.

`void` C`.get_bi (T &` b_2`, T &` b_4`, T &` b_6`, T &` b_8`) const`
gets all the values of the b_i's at once.

`void` C`.get_ci (T &` c_4`, T &` c_6`) const`
gets all the values of the c_i's at once.

High-Level Methods and Functions

Let C be an instance of `elliptic_curve< T >`.

`T` C`.j_invariant () const`
returns the j-invariant of C, namely c_4^3/Δ.

`void` C`.transform (const T &` r`, const T &` s`, const T &` t`)`
performs the change of variable $x = x' + r$ and $y = y' + sx' + t$.

`int` C`.is_null () const`
tests whether C has all its coefficients set to zero.

`int` C`.is_singular () const`
tests whether the discriminant is zero.

`void set_verbose (int` i`)`

sets the internal info variable to i. If this info variable is not zero, then the class outputs many information during computations; otherwise no output is given during computations. The default value of the info variable is zero.

`void verbose () const`

outputs the current value of the internal info variable.

In addition to the functions described above, there exist several functions valid only for `elliptic_curve< gf_element >`.

`bool` C`.is_supersingular ()`

returns `true` if and only if the curve C is supersingular.

`unsigned int` C`.degree_of_definition () const`

returns the extension degree of the field of definition of C over the corresponding prime field.

`point< T > random_point (const elliptic_curve< T > &` C`)`

returns a random point on the elliptic curve C.

`bigint` C`.group_order ()`

determines the order of the group of points on C over the finite field specified by the template type `T`. Note that this field might be bigger than the field of definition of C (e.g., in the case of `T` equals `gf2n`).

`void` C`.isomorphism_type (bigint &` m`, bigint &` n`)`

determines the isomorphism type of the group of points over the finite field specified by the template type `T`, i.e. determines integers m and n such that C is isomorphic to $\mathbb{Z}/m\mathbb{Z} \times \mathbb{Z}/n\mathbb{Z}$ and n divides m. The input variables m and n are set to these values.

`void` C`.isomorphism_type (bigint &` m`, bigint &` n`, point< T > &` P`, point< T > &` Q`)`

determines the isomorphism type of the group of points over the finite field specified by the template type `T`, i.e. determines integers m and n such that C is isomorphic to $\mathbb{Z}/m\mathbb{Z} \times \mathbb{Z}/n\mathbb{Z}$ and n divides m. The input variables m and n are set to these values. Moreover generators for the two subgroups are computed. The point P is set to a generator for the subgroup isomorphic to $\mathbb{Z}/m\mathbb{Z}$, Q is set to a generator for the $\mathbb{Z}/n\mathbb{Z}$ part.

`bool` C`.is_cyclic ()`

returns `true` if and only if the curve C is cyclic.

`bool` C`.probabilistic_is_order (const bigint &` r`, unsigned int` t `= 5)`

chooses t random points on the elliptic curve and tests whether r is a multiple of the order of all these points. If r satisfies all these tests, then `true` is returned, otherwise the function returns `false`.

Input/Output

Input and output for the projective model is not yet implemented.

There are four output modes for elliptic curves which can be used (with corresponding aliases):

(0) Short Format which outputs in the form "[a1,a2,a3,a4,a6]" (alias `SHORT`).

(1) Long Format which outputs all the information known about the curve (alias `LONG`).

(2) Pretty Format which outputs the equation (alias `PRETTY`).

(3) TeX Format which outputs the equation as TEX/LATEX source (alias `TEX`).

You can set the output mode using the aliases given above. The default value for the output mode is `SHORT`.

`void set_output_mode (long` i`)`
Sets the current output mode to i. Note that i should be one of the aliases described above.

The `ostream` operator `<<` has been overloaded and will produce output in the form specified by the current output mode. For convenience the following member functions are also defined.

`void` C`.output_short (ostream & out) const`
Send output in "short" format to the ostream `out`.

`void` C`.output_long (ostream & out) const`
Send output in "long" format to the ostream `out`.

`void` C`.output_tex (ostream & out) const`
Send output in "TEX" format to the ostream `out`.

`void` C`.output_pretty (ostream & out) const`
Send output in "pretty" format to the ostream `out`.

The `istream` operator `>>` has been overloaded and allows input in one of two forms $[a_1, a_2, a_3, a_4, a_6]$ or "$a_1 \quad a_2 \quad a_3 \quad a_4 \quad a_6$". In addition the following member function is also defined.

`void` C`.input (istream & in)`
Reads in coefficients for curve C from the istream `in`.

See also

`point< T >`, `elliptic_curve_flags`, `gf_element`

Notes

The elliptic curve package of LiDIA will be increased in future. Therefore the interface of the described functions might change in future.

Examples

```
#include <LiDIA/bigint.h>
#include <LiDIA/galois_field.h>
#include <LiDIA/gf_element.h>
#include <LiDIA/elliptic_curve.h>

int main()
{
```

```
    bigint p;
    cout << "\n Input prime characteristic : "; cin >> p;

    galois_field F(p);

    gf_element a4, a6;

    a4.assign_zero(F);
    a6.assign_zero(F);

    cout << "\n Coefficient a_4 : "; cin >> a4;
    cout << "\n Coefficient a_6 : "; cin >> a6;

    elliptic_curve< gf_element > e(a4, a6);

    cout << "\n Group order is " << e.group_order();

    return 0;
}
```

Author

Birgit Henhapl, John Cremona, Markus Maurer, Volker Müller, Nigel Smart.

point

Name

`point< T >` ..... class for holding a point on an elliptic curve over a field `T`.

Abstract

The class `point< T >` is used to store points on elliptic curves. A point on an elliptic curve over a field is held either in affine representation as $P = (x, y)$ or in projective representation as $P = (x, y, z)$. The representation depends on the model of the elliptic curve, the point is associated with.

Description

The class `point< T >` holds points on elliptic curves in either affine or projective representation. Therefore an instance of `point< T >` consists of two variables holding the x and y coordinate of this point, or of three variables holding the coordinates x, y, and z. In the projective model, the coordinates $(x : y : z)$ represent the equivalence class (k^2x, k^3y, kz) for k of type T.

Moreover it contains a boolean value which is `true` if and only if the point is the point at infinity, the zero of the corresponding group of points. Any point holds also a reference to the elliptic curve it sits on. It "knows" about the type of curve, i.e. whether the curve is in short or long Weierstrass form or a curve defined over $GF(2^n)$ and about the model of the curve, i.e., affine or projective. Relevant addition routines are then optimized for working with such curves.

In addition to functions for points defined over arbitrary fields the class offers special functions only for points defined over finite fields.

Constructors/Destructor

`ct point< T > ()`
constructs an invalid point.

`ct point< T > (const T & x, const T & y, const elliptic_curve< T > & E)`
constructs the point $P = (x, y), (x : y : z)$ on the curve E, depending on the model of E. The internal elliptic curve pointer for P is set to E. If $(x, y), (x : y : z)$, resp., does not satisfy the equation of E, the `lidia_error_handler` is invoked.

`ct point< T > (const T & x, const T & y, const T & z, const elliptic_curve< T > & E)`
constructs the point $P = (x : y : z)$ on the curve E. The internal elliptic curve pointer for P is set to E. If $(x : y : z)$ does not satisfy the equation of E or the model of the curve is affine, the `lidia_error_handler` is invoked.

`ct point< T > (const point< T > & P)`
constructs a new point and copies P to this new instance.

`ct point< T > (const elliptic_curve< T > & E)`
constructs the zero-point on the elliptic curve E.

`dt ~point< T > ()`

Assignments

The operator `=` is overloaded. However for efficiency reasons the following functions are also defined. Let P be of type `point< T >`.

`void P.assign_zero (const elliptic_curve< T > & E)`
assigns to P the point at infinity on the elliptic curve E.

`void P.assign (const point< T > & Q)`
assigns to P the point Q.

`void P.assign (const T & x, const T & y, const elliptic_curve< T > & E)`
assigns to P the point $(x, y), (x : y : 1)$, resp., on the curve E, depending on the model of E. The internal elliptic curve pointer for P is set to E. If $(x, y), (x : y : z)$ does not satisfy the equation of E, the `lidia_error_handler` is invoked.

`void P.assign (const T & x, const T & y, const T & z, const elliptic_curve< T > & E)`
assigns to P the point $(x : y : z)$ on the curve E. The internal elliptic curve pointer for P is set to E. If $(x : y : z)$ does not satisfy the equation of E or the model of the curve is affine, the `lidia_error_handler` is invoked.

If P is a point defined on a given curve, the following assignments exist which do not change the internal information about the curve. They should for this reason be used with care.

`void P.assign_zero ()`

`void P.assign (const T & x, const T & y)`
If $(x, y), (x : y : 1)$, resp., does not satisfy the equation of the curve P was previously defined on, the `lidia_error_handler` is invoked.

`void P.assign (const T & x, const T & y, const T & z)`
If $(x : y : z)$ does not satisfy the equation of the curve P was previously defined on or the model of the curve is affine, the `lidia_error_handler` is invoked.

Comparisons

The binary operators `==`, `!=` are overloaded. In the affine model, two points are considered equal if they lie on the same curve and their coordinates are equal. For the projective model, two points are considered equal if the lie on the same curve and their equivalence classes represented by the coordinates are equal. Let P be of type `point< T >`.

`bool P.on_curve () const`
returns `true` if and only if P is actually on its corresponding curve.

`bool` P`.is_zero () const`
returns `true` if and only if P is the zero-point.

`bool` P`.is_negative_of (const point< T > &` Q`) const`
returns `true` if and only if P is equal to $-Q$.

`bool` P`.is_affine_point () const`
returns `true` if and only if P is a point in affine representation.

`bool` P`.is_projective_point () const`
returns `true` if and only if P is a point in projective representation.

Access Methods

Let P be of type `point< T >`.

`T` P`.get_x () const`
returns the x-coordinate of P. The function will invoke the `lidia_error_handler`, if the model is affine and P is the point at infinity.

`T` P`.get_y () const`
returns the y-coordinate of P. The function will invoke the `lidia_error_handler` if the model is affine and P is the point at infinity.

`T` P`.get_z () const`
returns the z-coordinate of P. The function will invoke the `lidia_error_handler`, if the model is affine.

`const elliptic_curve< T > *` P`.get_curve () const`
OBSOLETE. Use the function below.

`elliptic_curve< T >` P`.get_curve () const`
returns the curve on which the point P sits.

Arithmetical Operations

The following operators are overloaded and can be used in exactly the same way as for machine types in C++ (e.g. `int`):

```
(unary) -
(binary) +, -
(binary with assignment) +=, -=
```

The `*` operator is overloaded as

`point< T > operator * (const bigint &` n`, const point< T > &` P`)`
returns $n \cdot P$.

For efficiency we also have implemented the following functions

`void negate (point< T > & `P`, const point< T > & `Q`)`
$P \leftarrow -Q$.

`void add (point< T > & `R`, const point< T > & `P`, const point< T > & `Q`)`
$R \leftarrow P + Q$.

`void subtract (point< T > & `R`, const point< T > & `P`, const point< T > & `Q`)`
$R \leftarrow P - Q$.

`void multiply_by_2 (point< T > & `P`, const point< T > & `Q`)`
$P \leftarrow 2 \cdot Q$.

`void multiply_by_4 (point< T > & `P`, const point< T > & `Q`)`
$P \leftarrow 4 \cdot Q$.

`point< T > `P`.twice () const`
returns $2 \cdot P$.

`void multiply (point< T > & `P`, const bigint & `n`, const point< T > & `Q`)`
$P \leftarrow n \cdot Q$.

Basic Methods and Functions

`void `P`.set_verbose (int `a`)`
sets the internal info variable to a. If the info variable is not zero, then some information is printed during computation. The default value of this info variable is zero.

High-Level Methods and Functions

If the template type `T` represents a finite field, i.e. `gf_element`, then the following additional functions are available. Let P be an instance of type `point< T >`.

`bool `P`.set_x_coordinate (const T & `xx`)`
tries to find a point with x-coordinate xx. If there exists such a point, it is assigned to P and `true` is returned. Otherwise P is set to the zero-point and `false` is returned.

`point< T > `P`.frobenius (unsigned int `d` = 1)`
evaluate the d-th power of the Frobenius endomorphism on the point P. Note that the Frobenius endomorphism is defined by the field of definition of the elliptic curve the point P is sitting on.

`point< T > frobenius (const point< T > & `P`, unsigned int `d` = 1)`
evaluate the d-th power of the Frobenius endomorphism on the point P and return the result. Note that the Frobenius endomorphism is defined by the field of definition of the elliptic curve the point P is sitting on.

`bigint order_point (const point< T > & `P`)`
returns the order of the point P.

`bigint order_point (const point< T > & P, rational_factorization & f)`

returns the order of the point P assuming that the integer represented by f is a multiple of the order of P. If f does not satisfy this condition, the `lidia_error_handler` is invoked. If necessary, the factorization f is refined during the order computation.

`rational_factorization rf_order_point (const point< T > & P)`

returns the order of the point P as `rational_factorization`.

`rational_factorization rf_order_point (const point< T > & P, rational_factorization & f)`

returns the order of the point P as `rational_factorization` assuming that the integer represented by f is a multiple of the order of P. If f does not satisfy this condition, the `lidia_error_handler` is invoked. If necessary, the factorization f is refined during the order computation.

`bigint bg_algorithm (const point< T > & P, const point< T > & Q, const bigint & l, const bigint & u)`

tries to find an integer $l \leq x \leq u$ such that $x \cdot P = Q$ with a Babystep Giantstep Algorithm. If such an integer does not exist, -1 is returned. If l is negative or $u < l$ or P and Q sit on different curves, the `lidia_error_handler` is invoked.

`bigint discrete_logarithm (const point< T > & P, const point< T > & Q, bool info = false)`

uses the Pohlig-Hellman algorithm to find the discrete logarithm of the point Q to base P, i.e. an integer $0 \leq x < \mathrm{ord}(P)$ such that $x \cdot P = Q$. If the discrete logarithm does not exist, -1 is returned. If *info* is `true` then the intermediate results are printed; otherwise no output is done.

Input/Output

Input and output is only implemented for the affine model.

The `ostream` and `istream` operators `<<` and `>>` have been overloaded. The operator `<<` outputs points in the form (x, y) or O for the point at infinity.

The operator `>>` allows input in one of three forms $[x, y]$ or (x, y) or $x \;\; y$. At present it is not possible to input the point at infinity. Note that the elliptic curve is not part of the input; it has to be set before a point can be read.

`void P.input (istream & in)`

input a point P from istream `in`.

Examples

The following program reads in an elliptic curves over `gf_element`'s plus a point and then tests the arithmetic routines on such a curve.

```
#include <LiDIA/galoes_field.h>
#include <LiDIA/gf_element.h>
#include <LiDIA/elliptic_curve.h>
#include <LiDIA/point.h>

int main ()
{
    bigint p;
```

```
int tests, i;

cout << "\n ELLIPTIC CURVES OVER FINITE PRIME FIELD\n\n";

cout << "\n\n Input  modulus : "; cin >> p;
p = next_prime(p-1);
cout << "\n Choosing next prime = "<<p<<"\n\n";

char model;
do {
    cout << "Affine or projective model ? Enter 'a' or 'p' : ";
    cin >> model;
} while (model != 'a' && model != 'p');

galois_field F(p);
gf_element a1, a2, a3, a4, a6;

a1.assign_zero(F);
a2.assign_zero(F);
a3.assign_zero(F);
a4.assign_zero(F);
a6.assign_zero(F);

elliptic_curve< gf_element > e;

if (model == 'a') {
    cout << "\n Input a1 : "; cin >> a1;
    cout << "\n Input a2 : "; cin >> a2;
    cout << "\n Input a3 : "; cin >> a3;
    cout << "\n Input a4 : "; cin >> a4;
    cout << "\n Input a6 : "; cin >> a6;
    cout << "\n # Tests : "; cin >> tests;

    cout << "Initialization ... " << flush;

    e.set_coefficients(a1, a2, a3, a4, a6);
}
else {
    cout << "\n Input a4 : "; cin >> a4;
    cout << "\n Input a6 : "; cin >> a6;
    cout << "\n # Tests : "; cin >> tests;

    cout << "Initialization ... " << flush;

    e.set_coefficients(a4, a6, elliptic_curve_flags::PROJECTIVE);
}

point< gf_element > P(e);
point< gf_element > Q(e), H(e);

cout << "done." << endl;
cout << "Starting tests " << flush;

for (i = 1; i <= tests; i++) {
    P = e.random_point();
    Q = e.random_point();
    H = e.random_point();
```

```
            assert(P.on_curve());
            assert(Q.on_curve());
            assert(H.on_curve());

            identitytest(P, Q, H);
            utiltest(P);
            cout << "*" << flush;
        }
        cout << "\n\n";

        return 0;
    }
```

See also

`elliptic_curve< T >`, `elliptic_curve_flags`.

Notes

Author

John Cremona, Birgit Henhapl, Markus Maurer, Volker Müller, Nigel Smart.

elliptic_curve< bigint >

Name

`elliptic_curve< bigint >` ..... class for holding the minimal model of a rational elliptic curve.

Abstract

An elliptic curve is a curve of the form

$$Y^2 + a_1 Y + a_3 XY = X^3 + a_2 X^2 + a_4 X + a_6$$

which is non-singular. The minimal model is an equation with integral coefficients and "minimal" discriminant.

Description

This class is a specialization of the template class `elliptic_curve< T >` and is for working with a minimal model of an elliptic curve defined over the rationales. This curve provides access to the reduction type at all primes, the conductor etc. This is extracted using Tate's algorithm after the curve has been reduced to a minimal model using Laska-Kraus-Connell reduction. The class also supports the computation of the torsion group and calculations using the complex lattice.

Constructors/Destructor

The constructors are the same as in the general template class `elliptic_curve< T >`, except the following difference:

`ct elliptic_curve (const bigint &` c_4`, const bigint &` c_6`)`
: constructs the elliptic curve with parameters c_4 and c_6. Note this is *not* the same as $[0, 0, 0, c_4, c_6]$ in PARI format.

Assignments

The operator `=` is overloaded. For efficiency we also have

`void init (const bigint &` a_1`, const bigint &` a_2`, const bigint &` a_3`, const bigint &` a_4`,`
`const bigint &` a_6`)`

`void init (const bigint &` c_4`, const bigint &` c_6`)`

Access Methods

These are the same as for the general template class `elliptic_curve< T >`.

High-Level Methods and Functions

These are almost the same as for the general template class `elliptic_curve< T >`, but exchanging the function for computation of the j-invariant and adding the following new functionalities.

`bigrational` C`.j_invariant () const`
: returns the j-invariant of C.

`bigint` C`.get_conductor () const`
: returns the conductor of the curve C.

`int` C`.get_ord_p_discriminant (const bigint &` p`) const`
: returns the valuation at p of the discriminant.

`int` C`.get_ord_p_N (const bigint &` p`) const`
: returns the valuation at p of the conductor.

`int` C`.get_ord_p_j_denom (const bigint &` p`) const`
: returns the valuation at p of the denominator of the j-invariant.

`int` C`.get_c_p (const bigint &` p`) const`
: returns the value of the Tamagawa number at p.

`int` C`.get_product_cp () const`
: returns the product of the Tamagawa numbers over all primes p.

`Kodaira_code` C`.get_Kodaira_code (const bigint &` p`) const`
: returns the Kodaira code of the curve at the prime p.

`int` C`.get_no_tors ()`
: computes the number of torsion elements in the Mordell-Weil group of C.

`base_vector< point< bigint > >` C`.get_torsion ()`
: returns the complete set of torsion points of C.

`bigcomplex` C`.get_omega_1 ()`
: returns the first period of the complex lattice associated to C.

`bigcomplex` C`.get_omega_2 ()`
: returns the second period of the complex lattice associated to C.

`complex_periods` C`.get_periods ()`
: returns the full data for the periods. Only really for expert use, if you want to know more see the file `LiDIA/complex_periods.h`.

`void` C`.get_xy_coords (bigcomplex &` x`, bigcomplex &` y`, const bigcomplex &` z`)`
: given the point z in the complex plain, this function returns the corresponding x and y coordinates on the elliptic curve.

`double` C`.get_height_constant ()`

returns the constant c such that $h(P) \leq \hat{h}(P) + c$ for all points P. The value computed is the best one returned by the formulae of Silverman or the algorithm of Siksek.

`sort_vector< bigint >` C`.get_bad_primes () const`

returns a sorted vector of the factors of the discriminant of C.

Note for efficiency many of the above functions only compute the results once and then store them in the internal data for the curve. This is why many of the higher functions are not declared "`const`".

Input/Output

As for `elliptic_curve< T >`, except that "long" format now produces even more data. In addition we have

`void` C`.output_torsion (ostream & out)`

output the torsion subgroup on the ostream `out`.

Examples

```
#include <LiDIA/elliptic_curve_bigint.h>
#include <LiDIA/point_bigint.h>

int main()
{
    cout << "Enter a_1,..,a_6 \n" << flush;
    bigint a1, a2, a3, a4, a6;
    cin >> a1 >> a2 >> a3 >> a4 >> a6;

    elliptic_curve< bigint > ER(a1, a2, a3, a4, a6);

    cout << ER << endl;
    cout << "Conductor = " << ER.conductor() << endl;

    long i;

    cout << "Torsion Group is given by \n";
    for (i = 0; i < ER.get_no_tors(); i++) {
        cout << i << " ) " << ER.get_torsion()[i] << endl;
    }

    cout << "Periods \n";
    cout << "Omega_1 = " << ER.get_omega_1() << endl;
    cout << "Omega_2 = " << ER.get_omega_2() << endl;

    cout << "Height Constant\n";
    cout << ER.get_height_constant() << endl;

    return 0;
}
```

See also

`elliptic_curve< T >`, `point< T >`, `point< bigint >`, `Kodaira_code`.

Author

John Cremona, Nigel Smart.

point< bigint >

Name

`point< bigint >` ..... class for holding a point on a minimal model of an elliptic curve over the rationals.

Abstract

A point on a minimal model of an elliptic curve is held in "projective" representation as $P = (x : y : z) = (x/z, y/z)$.

Description

This class is a specialization of the general template class `point< T >`. Any point holds a pointer to the curve it sits on, in addition the addition routines "know" about the type of curve, i.e. whether it is in short or long Weierstrass form. Algorithms are then optimized for working with such curves.

Constructors/Destructor

`ct point< bigint > ()`
constructs an invalid point.

`ct point< bigint > (const bigint & x, const bigint & y, const elliptic_curve< bigint > & E)`
constructs the point $P = (x : y : 1)$ on the curve E.

`ct point< bigint > (const bigint & x, const bigint & y, const bigint & z, const elliptic_curve< bigint > & E)`
constructs the point $P = (x : y : z)$ on the curve E.

`ct point< bigint > (const point< bigint > & P)`
Copy constructor for a point.

`ct point< bigint > (const elliptic_curve< bigint > & E)`
constructs the point at infinity on the elliptic curve E.

`dt ~point< bigint > ()`

Assignments

The operator = is overloaded. However for efficiency the following functions are also defined. Let P be of type `point< bigint >`.

`void` P`.assign_zero (const elliptic_curve< bigint > &` E`)`
assigns to P the point at infinity on the elliptic curve E.

`void` P`.assign (const point< bigint > &` Q`)`
assigns to P the point Q.

`void` P`.assign (const bigint &` x`, const bigint &` y`, const elliptic_curve< bigint > &` E`)`
assigns to P the point $(x : y : 1)$ on the curve E.

`void` P`.assign (const bigint &` x`, const bigint &` y`, const bigint &` z`,`
`const elliptic_curve< bigint > &` E`)`
assigns to P the point $(x : y : z)$ on the curve E.

If P is a point defined on a given curve, the following assignments exist which use the same curve. They should for this reason be used with care.

`void` P`.assign_zero ()`

`void` P`.assign (const bigint &` x`, const bigint &` y`)`

`void` P`.assign (const bigint &` x`, const bigint &` y`, const bigint &` z`)`

`void swap (point< bigint > &` P`, point < bigint > &` Q`)`
swaps the points P and Q.

Comparisons

The binary operators ==, != are overloaded. Two points are considered equal if they lie on the same curve and their coordinates are equal. Let P be of type `point< bigint >` then the following functions are also defined:

`bool` P`.on_curve () const`
returns `true` if and only if P is actually on the curve.

`bool` P`.is_zero () const`
returns `true` if and only if P is the point at infinity.

`bool` P`.is_integral () const`
returns `true` if and only if P is an integral point.

`bool` P`.is_negative_of (const point < bigint > &` Q`) const`
returns `true` if and only if P is equal to $-Q$.

`int eq (const point< bigint > &` P`, const point< bigint > &` Q`)`
returns `true` if $P = Q$, otherwise returns `false`.

Access Methods

Let P be of type `point< bigint >`.

`bigint` P`.get_xyz (bigint &` x`, bigint &` y`, bigint &` z`) const`
returns the x, y and z coordinates of P.

`const elliptic_curve< bigint > *` P`.get_curve () const`
OBSOLETE. Use the function below.

`elliptic_curve< bigint >` P`.get_curve () const`
returns the curve on which the point P sits.

Arithmetical Operations

The following operators are overloaded and can be used in exactly the same way as for machine types in C++ (e.g. `int`):

```
(unary) -
(binary) +, -
(binary with assignment) +=, -=
```

The `*` operator is overloaded by

`point< bigint > operator * (const bigint &` n`, const point< bigint > &` P`)`
returns $n \cdot P$.

For efficiency we also have the following functions

`void negate (point< bigint > &` P`, const point< bigint > &` Q`)`
$P \leftarrow -Q$.

`void add (point< bigint > &` R`, const point< bigint > &` P`, const point< bigint > &` Q`)`
$R \leftarrow P + Q$.

`void subtract (point< bigint > &` R`, const point< bigint > &` P`,`
`const point< bigint > &` Q`)`
$R \leftarrow P - Q$.

`void multiply_by_2 (point< bigint > &` P`, const point< bigint > &` Q`)`
$P \leftarrow 2 \cdot Q$.

`point< bigint >` P`.twice () const`
returns $2 \cdot P$.

`void multiply (point< bigint > &` P`, const bigint &` n`, const point< bigint > &` Q`)`
$P \leftarrow n \cdot Q$

Input/Output

Input/Output functions are exactly the same as in the template class `point< T >`.

High-Level Methods and Functions

Let P be of type `point< bigint >`.

`bigfloat` P`.get_height ()`
returns the canonical height of P.

`bigfloat` P`.get_naive_height ()`
returns the naive height of the point P.

`bigfloat` P`.get_pheight (const bigint &` p`)`
returns the local height of the point P at the prime p.

`bigfloat` P`.get_realheight ()`
returns the local height at infinity of the point P.

`int` P`.get_order ()`
returns the order of the point P in the Mordell-Weil group, this function returns -1 if the point has infinite order.

`int order (point< bigint > &` P`, base_vector< point< bigint > > &` *list*`)`
computes the order of the point P, if the point has finite order then *list* contains the list of all multiples of the point P.

Examples

```
#include <LiDIA/elliptic_curve_bigint.h>
#include <LiDIA/point_bigint.h>

int main()
{
    cout << "Enter a_1, ..., a_6" << endl;
    bigint a1, a2, a3, a4, a6;
    cin >> a1 >> a2 >> a3 >> a4 >> a6;

    elliptic_curve< bigint > ER(a1, a2, a3, a4, a6);

    cout << ER << endl;

    cout << "Enter point on this curve [x,y]" << endl;
    bigint x,y;
    cin >> x >> y;

    point< bigint > P(x,y,ER);
    point< bigint > Q(ER),H(ER);

    if (P.on_curve())
        cout << " The point is on curve\n" << endl;
    else
        lidia_error_handler("ec","point not on curve");

    for (i = 1; i <= 1000; i++) {
        add(Q, Q, P);
```

```
            if (!Q.on_curve()) {
                cerr << "\n Q = " << i << " * P = " << Q;
                cerr << " is not on curve any more\n";
                return 1;
            }
            multiply(H, i, P);

            if (Q != H) {
                cerr << "\n Q = " << i << " * P = " << Q;
                cerr << "\n H = " << H << " is different\n";
                return 1;
            }
            cout << i << "  " << Q << "\n     " << Q.get_height();
            cout << "\n     " << Q.get_naive_height();
            cout << endl;
        }
        cout<<"\n found no errors\n\n";

        return 0;
    }
```

See also

`elliptic_curve< T >`, `point< T >`, `elliptic_curve< bigint >`.

Author

John Cremona, Nigel Smart.

curve_isomorphism

Name

`curve_isomorphism< S, T >` .....class for holding isomorphisms between different elliptic curves.

Abstract

Assume that we have given two elliptic curves `elliptic_curve< S >` E_1 and `elliptic_curve< T >` E_2. This class constructs, if possible, an explicit isomorphism between them. This allows the user to map points from one curve to the other and back again.

Description

An isomorphism between two elliptic curves is a transformation of the variables of the form $x = u^2x' + r$ and $y = u^3y' + su^2x' + t$. The classes `S` and `T` require certain conditions to be met. To see what these are consult the file `LiDIA/curve_isomorphism.h`

Constructors/Destructor

`ct curve_isomorphism< S, T > (const elliptic_curve< S > & `E_1`, const elliptic_curve< T > & `E_2`)`
constructs an explicit isomorphism between the two elliptic curves E_1 and E_2. If the curves are not isomorphic then the `lidia_error_handler` is called.

`ct curve_isomorphism< S, T > (const curve_isomorphism< S, T > & `*iso*`)`
copy constructor.

`ct curve_isomorphism< S, T > (const elliptic_curve < S > & `E_1`, const elliptic_curve< T > & `E_2`, const S & `u`, const S & `r`, const S & `s`, const S & `t`)`
This constructor is used when the isomorphism is already known, however it does not check that it is valid.

`dt ~curve_isomorphism< S, T > ()`

Assignments

The operator `=` is overloaded. For efficiency we also have the function.

`void init (const elliptic_curve< S > & `E_1`, const elliptic_curve< T > & `E_2`, const S & `u`, const S & `r`, const S & `s`, const S & `t`)`
As the similar constructor above.

High-Level Methods and Functions

`int are_isomorphic (const elliptic_curve < S > & `E_1`, const elliptic_curve < T > & `E_2`)`
returns 1 if and only if the two curves are isomorphic.

`int are_isomorphic (const elliptic_curve< S > & `E_1`, const elliptic_curve< T > & `E_2`, curve_isomorphism< S, T > & iso)`
returns 1 if and only if the two curves are isomorphic, the explicit isomorphism is returned in *iso*.

`elliptic_curve< S > make_isomorphic_curve (const elliptic_curve< T > & `E_2`, S & `u`, S & `r`, S & `s`, S & `t`)`
constructs the elliptic curve E_1 over `S`, which is the isomorphic image of E_2 under the isomorphism $[u, r, s, t]$.

Let *iso* denote an object of type `curve_isomorphism< S, T >`.

`point< T > iso.map (const point< S > & `P`) const`
find the image of the point $P \in E_1$ on E_2.

`point < S > iso.inverse (const point< T > & `P`) const`
find the image of the point $P \in E_2$ on E_1.

`void iso.invert (S & `u`, S & `r`, S & `s`, S & `t`) const`
$[u, r, s, t]$ become the inverse isomorphism to that represented by *iso*.

`S iso.get_scale_factor () const`
returns u.

`int iso.is_unimodular () const`
tests whether u is equal to one.

`int iso.is_identity () const`
tests whether *iso* is the identity isomorphism.

`void minimal_model (elliptic_curve< bigint > & `E_{min}`, elliptic_curve< bigrational > & `E_R`, curve_isomorphism< bigrational, bigint > & iso)`
constructs the minimal model of the elliptic curve E_R and assigns it to E_{min}. The explicit isomorphism is returned via the variable *iso*. Note at present to use this function you need to include the file `LiDIA/minimal_model.h`.

Input/Output

Only the `ostream` operator `<<` has been overloaded.

Examples

```
#include <LiDIA/elliptic_curve_bigint.h>
#include <LiDIA/elliptic_curve.h>
#include <LiDIA/point_bigint.h>
#include <LiDIA/curve_isomorphism.h>
```

```
int main()
{
    elliptic_curve< bigrational > Er1(1,2,3,4,5);
    elliptic_curve< bigrational > Er2(6,0,24,16,320);
    elliptic_curve< bigint >      Ei(1,2,3,4,5);
    cout << Er1 << "  " << Er2 << "  " << Ei << endl;

    curve_isomorphism< bigrational, bigint > iso1(Er1,Ei);
    curve_isomorphism< bigrational, bigint > iso2(Er2,Ei);
    curve_isomorphism< bigrational, bigrational > iso3(Er1,Er2);

    cout << iso1 << "\n" << iso2 << "\n" << iso3 << "\n" << endl;

    point< bigint > Pi(2,3,Ei);

    cout << Pi << endl;

    point< bigrational > Pr1=iso1.inverse(Pi);
    point< bigrational > Pr2=iso2.inverse(Pi);

    cout << Pr1 << "  " << Pr1.on_curve() << endl;
    cout << Pr2 << "  " << Pr2.on_curve() << endl;

    point< bigint > twoPi=Pi+Pi;
    point< bigrational > twoPr1=Pr1+Pr1;
    point< bigrational > twoPr2=Pr2+Pr2;

    cout << "Test One\n";
    cout << twoPi << endl;
    cout << iso1.map(twoPr1) << endl;
    cout << iso2.map(twoPr2) << endl;

    cout << "Test Two\n";
    cout << twoPr1 << endl;
    cout << iso1.inverse(twoPi) << endl;
    cout << iso3.inverse(twoPr2) << endl;

    cout << "Test Three\n";
    cout << twoPr2 << endl;
    cout << iso2.inverse(twoPi) << endl;
    cout << iso3.map(twoPr1) << endl;

    return 0;
}
```

See also the file `LiDIA/src/templates/elliptic_curves/ec_rationals/minimal_model_appl.cc`.

See also

`elliptic_curve< T >`, `point< T >`, `elliptic_curve< bigint >`, `point< bigint >`.

Author

John Cremona, Nigel Smart.

quartic

Name

`quartic` ..... class for holding a curve of the form $y^2 = ax^4 + bx^3 + cx^2 + dx + e$.

Abstract

Curves of the form $y^2 = ax^4 + bx^3 + cx^2 + dx + e$, hereafter called quartics, arise in algorithms to find Mordell-Weil groups of elliptic curves.

Description

Constructors/Destructor

`ct quartic ()`
constructs an invalid quartic.

`ct quartic (const bigint & a, const bigint & b, const bigint & c, const bigint & d, const bigint & e)`
constructs the quartic $y^2 = ax^4 + bx^3 + cx^2 + dx + e$.

`ct quartic (const quartic & Q)`
copy constructor for a quartic.

`dt ~quartic ()`

Assignments

The operator `=` is overloaded. However for efficiency the following are also defined. Let Q be of type `quartic`.

`void Q.assign (const bigint & a, const bigint & b, const bigint & c, const bigint & d, const bigint & e)`
assigns Q the quartic $y^2 = ax^4 + bx^3 + cx^2 + dx + e$.

`void Q.assign (const quartic & R)`
assigns to Q the quartic R.

Access Methods

Let Q be of type `quartic`.

`bigint` Q`.get_a () const`
: returns the coefficient a.

`bigint` Q`.get_b () const`
: returns the coefficient b.

`bigint` Q`.get_c () const`
: returns the coefficient c.

`bigint` Q`.get_d () const`
: returns the coefficient d.

`bigint` Q`.get_e () const`
: returns the coefficient e.

`void` Q`.get_coeffs (bigint &` a`, bigint &` b`, bigint &` c`, bigint &` d`, bigint &` e`) const`
: sets a, b, c, d, e to be the requisite coefficients of Q.

High-Level Methods and Functions

Let Q be of type `quartic`.

`int` Q`.get_type () const`
: returns the type of the quartic.

`bigint` Q`.get_I () const`
: returns the I-invariant of Q.

`bigint` Q`.get_J () const`
: returns the J-invariant of Q.

`bigint` Q`.get_H () const`
: returns the semiinvariant H of Q.

`bigint` Q`.get_discriminant () const`
: returns the discriminant of Q.

`bigcomplex *` Q`.get_roots () const`
: returns the complex roots of $ax^4 + bx^3 + cx^2 + dx + e = 0$. Note the return type of this function WILL change in later releases once we have settled on how we are to use this function when implementing `mwrank`.

`void` Q`.doubleup ()`
: changes the curve by $b \leftarrow 2b$, $c \leftarrow 4c$, $d \leftarrow 8d$, and $e \leftarrow 16e$.

`int` Q`.trivial () const`

tests for a rational root, i.e. does Q represent the trivial element in the Selmer group.

`long` Q`.no_roots_mod (long` p`) const`

returns the number of roots of $ax^4 + bx^3 + cx^2 + dx + e$ modulo p, including those at infinity. This may seem like an obscure function but it is used in an equivalence check in `mwrank` based on an idea of Siksek.

`int` Q`.zp_soluble (const bigint &` p`, const bigint &` x_0`, long` ν`) const`

tests whether there is a solution to Q in the p-adic integers with $x \equiv x_0 \pmod{p^\nu}$.

`int` Q`.qp_soluble (const bigint &` p`) const`

tests whether Q is locally soluble at p. This uses the naive test for small values of p and the conjectured polynomial time test of Siksek for large p.

`int` Q`.locally_soluble (const sort_vector< bigint > &` *plist*`) const`

checks local solubility at infinity and at the set of primes in the list *plist*.

Input/Output

Only the `ostream` operator `<<` has been overloaded.

Examples

```
#include <LiDIA/quartic.h>

int main()
{
    long lidia_precision=40;
    bigfloat::precision(lidia_precision);

    // First a test of quartics...
    quartic q(-1,0,54,-256,487);
    cout << q << endl;
    cout << "Testing " << endl;
    cout << "2 : " << q.qp_soluble(2) << endl;
    cout << "3 : " << q.qp_soluble(3) << endl;
    cout << "11 : " << q.qp_soluble(11) << endl;
    cout << "941 : " << q.qp_soluble(941) << endl;

    q.assign(1,2,-9,-42,-43);
    cout << q << endl;
    cout << "Testing " << endl;
    cout << "2 : " << q.qp_soluble(2) << endl;
    cout << "3 : " << q.qp_soluble(3) << endl;
    cout << "11 : " << q.qp_soluble(11) << endl;
    cout << "941 : " << q.qp_soluble(941) << endl;

   return 0;
}
```

Author

John Cremona, Nigel Smart.

Kodaira_code

Name

`Kodaira_code< T >` ..... class for holding a Kodaira symbol.

Abstract

The Kodaira symbol of an elliptic curve defined over a number field classifies the reduction type of the curve at a prime. The possible reduction types are: I_n, I_n^*, II, II*, III, III*, IV, IV*, where $n \geq 0$.

Description

A Kodaira symbol is stored as an integer k according to the following scheme:

Symbol	k
I_m	$10m$
I_m^*	$10m+1$
$\mathrm{I}, \mathrm{II}, \mathrm{III}, \mathrm{IV}$	$1, 2, 3, 4$
$\mathrm{I}^*, \mathrm{II}^*, \mathrm{III}^*, \mathrm{IV}^*$	$5, 6, 7, 8$

Constructors/Destructor

`ct Kodaira_code (int` k`)`
constructs a symbol with code k.

Assignments

The operator `=` is overloaded; a `Kodaira_code` may be assigned to an integer or to another `Kodaira_code`.

Access Methods

This function should not really be needed by users: Let K be of class `Kodaira_code`.

`int` K`.get_code () const`
returns the value of the integer k.

Input/Output

Only the `ostream` operator `<<` has been overloaded. It prints a string denoting the Kodaira symbol.

See also

`elliptic_curve< bigint >`.

Notes

This class should not be needed by end-users. It is used by the class `elliptic_curve< bigint >` which contains an array of the `Kodaira_code`s for each prime of bad reduction.

The coding scheme was originally due to Richard Pinch.

Author

John Cremona, Nigel Smart.

The LiDIA ECO package

Chapter 20

Point counting of elliptic curves over $\mathrm{GF}(p)$ and $\mathrm{GF}(2^n)$

eco_prime

Name

`eco_prime` ..... class for holding information during point counting.

Abstract

The class `eco_prime` holds information needed in the point counting algorithm due to Atkin/Elkies. These information are for example the coefficients of the used elliptic curve, and the size and characteristic of the underlying field.

Description

`eco_prime` is a class for holding information during the point counting algorithm of Atkin and Elkies. The class uses a type `ff_element` which determines the type of the finite field used in the computation. At the moment the type `ff_element` is aliased to `bigmod`, so `eco_prime` can only be used for counting the number of points on elliptic curves defined over large prime fields, but this will change in future releases, where also arbitrary finite fields will be allowed to use. Note that the primality of the modulus of `bigmod` is not checked by the point counting code.

At the moment usage of the class `eco_prime` is considered only for a user who is familiar with details of the algorithm of Atkin/Elkies. A user not familiar with the subject should only use the following global function, defined in `LiDIA/eco_prime.h`, which solves the given problem:

`bigint compute_group_order (const gf_element & a, const gf_element & b)`
return the (probable) group order of the elliptic curve in short Weierstrass form $Y^2 = X^3 + aX + b$.

"Probable" group order denotes the fact that only a probabilistic correctness test of the computed result is done, no deterministic proof of correctness is performed.

From now on we assume that the reader is familiar with details of the algorithm of Atkin/Elkies. This is necessary since all the other functions of `eco_prime` do *not* test whether using the function makes sense at that specific moment, e.g. some access functions should only be used after the corresponding information has been determined. Details of the Atkin/Elkies algorithm can be found in [49].

The class `eco_prime` is directly linked to the two other classes used in this point counting implementation, the class `meq_prime` for managing equivalent polynomials, and the class `trace_list` for storing lists of possible values for the trace of the so called Frobenius endomorphism modulo different primes.

The class `eco_prime` stores several values of the used field (e.g., the size and the characteristic of the field) and used elliptic curve (e.g., the coefficients of the elliptic curve which is assumed to be in short Weierstrass form). These information are stored as static variables. Moreover there exist internal variables which store information about the group order of the elliptic curve modulo some prime l which can be set by the user. These values consist of a sorted vector for values of the trace of the Frobenius endomorphism modulo l, the number l itself and several values connected to the lth equivalent polynomial. Note that the size of l is restricted

since only a finite number of equivalent polynomials is precomputed and stored in a database (remember the general description of the point counting package). For a complete description of all private variables and the corresponding meaning, see the include file `LiDIA/eco_prime.h` and [49].

There exists the possibility to choose several strategies during computation. The default strategy is to determine for any prime l the complete splitting type of the lth modular polynomial. There is however also the opportunity to use different other strategies. These strategies are defined by the following constants:

- `COMPUTE_SP_DEGREE_AND_ROOTS`: compute the splitting degree of the modular polynomial and determine always a root in the base field of this modular polynomial, if such a root exists (the default value),
- `COMPUTE_SP_DEGREE`: compute the splitting degree of the modular polynomial in any case, but do not compute a root for polynomials of splitting type $(1 \ldots 1)$,
- `DONT_COMPUTE_SP_DEGREE`: determine whether a prime is an Atkin or Elkies prime, but do not compute the splitting degree of the modular polynomial,
- `COMPUTE_SP_DEGREE_IF_ATKIN`: determine whether a prime is an Atkin or Elkies prime, but do not compute the splitting degree of the modular polynomial in case it is an Elkies prime,
- `COMPUTE_SP_DEGREE_IF_ELKIES`: determine whether a prime is an Atkin or Elkies prime, but compute the splitting degree of the modular polynomial only in case it is an Elkies prime.

Another default strategy is, that once the Elkies polynomial is known, the eigenvalue of the Frobenius endomorphism is determined using division polynomials. This method is faster than using rational functions, but needs more storage (see [44]). These strategies are defined by the following constants:

- `EV_DIVISION_POLYNOMIAL` division polynomials are used to compute multiples of points (the default value),
- `EV_RATIONAL_FUNCTION` rational functions are used to compute multiple of points.

As said before, these values should only be changed by a user who is familiar with the details of the algorithm.

Constructors/Destructor

`ct eco_prime ()`
initializes an empty instance.

`dt ~eco_prime ()`

Assignments

The operator `=` is overloaded. Moreover there exists the following functions to set either static or non-static variables. Let e be an instance of `eco_prime`.

`void` e`.set_prime (udigit` l`)`
set the internal prime variable to l. l is assumed to be a prime, but this is not checked at the moment.

`void` e`.set_curve (const gf_element &` A`, const gf_element &` B`)`
A and B are the coefficients of the elliptic curve in short Weierstrass form. If the discriminant of the given elliptic curve is zero, the `lidia_error_handler` is invoked.

Access Methods

Let e be an instance of `eco_prime`.

`bool` e`.is_supersingular (bigint &` r`)`

returns `true` if and only if the elliptic curve set for e is supersingular. In this case r is set to the probable group order (according to the probabilistic correctness proof).

`bool check_j_0_1728 (bigint &` r`)`

returns `true` if and only if the elliptic curve set for e is isogen to a curve with j invariant 0 or 1728. In this case r is set to the probable group order (according to the probabilistic correctness proof).

`bool` e`.is_elkies ()`

returns `true` if and only if the actually used prime l is a so called Elkies prime, i.e. the characteristic polynomial of the Frobenius endomorphism splits modulo l. It is assumed that the splitting type of the corresponding l-th modular polynomial has already been computed.

`long` e`.get_prime () const`

returns the actually used prime l.

`base_vector< udigit > & get_relation () const`

returns a vector of all possible candidates for the trace of the Frobenius endomorphism modulo the actual used prime l. It is assumed that this information was already computed.

High-Level Methods and Functions

Let e be an instance of `eco_prime`.

`void` e`.compute_jinv (ff_element &` j`)`

sets j to the j-invariant of the elliptic curve of e. If the discriminant of this elliptic curve is zero, the `lidia_error_handler` is invoked.

`void` e`.compute_splitting_type ()`

computes the splitting type of the l-th equivalent polynomial where l is the internally stored prime. The splitting type and other relevant information (as roots of the equivalent polynomial) is stored internally.

`void` e`.set_strategy (char` m`)`

sets the strategy which is used for determining the splitting type of equivalent polynomials. The variable m must be one of the types described in the general description of this class, otherwise the `lidia_error_handler` is invoked.

`void` e`.set_schoof_bound (unsigned int` m`)`

set internal bound such that Schoof's algorithm is used for Atkin primes smaller m to determine exact trace of Frobenius. The default value for this bound is zero, i.e. Schoof's algorithm is never used.

`void` e`.set_ev_search_strategy (char` m`)`

sets the strategy which is used for computing point multiples when searching the eigenvalue in the Elkies case. The variable m must be one of the types described in the general description of this class, otherwise the `lidia_error_handler` is invoked.

`void` e`.compute_mod_2_power ()`

computes the exact value of the trace of the Frobenius endomorphism modulo some power of 2. l is set to the corresponding power of 2 and the trace is stored internally.

`void` e`.compute_trace_atkin ()`

computes possible values for the trace of the Frobenius endomorphism modulo the prime l and stores it internally. This function can be used for Atkin and Elkies primes.

`void` e`.compute_trace_elkies ()`

compute the exact value of the trace of the Frobenius endomorphism modulo the prime l and stores it internally. This function can only be used for Elkies primes.

`void` e`.schoof_algorithm ()`

compute the exact value of the trace of the Frobenius endomorphism modulo the prime l with Schoof' algorithm and store it internally. This function should only be used for "small" Atkin primes. It assumes that e`.compute_trace_atkin` has already be called, such that several internal values do exist.

`void` e`.set_info_mode (int` i `= 0)`

sets the info variable to i. If this info variable is not zero, then the class outputs many information during computations; otherwise no output is given during computations. The default value of the info variable is zero.

`bigint` e`.compute_group_order ()`

returns the (probable) group order of the elliptic curve stored in e.

Input/Output

Input/Output of instances of `eco_prime` is currently not possible.

See also

`meq_prime`, `trace_list`.

Notes

The Elliptic Curve Counting Package will be increased in future. This will probably also create changes in the class `eco_prime`. In future releases we will offer a more comfortable usage of this class, even for non experts.

Author

Markus Maurer, Volker Müller.

meq_prime

Name

`meq_prime` ..... class for input information from the precomputed database of modular polynomials.

Abstract

The class `meq_prime` is used for input of modular polynomials. These equations are currently stored in a database.

Description

`meq_prime` is a class the input of so called modular polynomials. This class is used from the functions of the class `eco_prime` to construct suitable polynomials for the elliptic curve counting algorithm of Atkin/Elkies.

For a non expert user it should not be necessary to use the class `meq_prime`, since all necessary computations are done automatically.

In the current version the class `meq_prime` invokes the `lidia_error_handler` if a modular polynomial can not be found in the precomputed database. This database is expected to be stored in the directory `$LIDIA_HOME/lib/LiDIA/MOD_EQ`. The modular equation corresponding to the prime l is stored in the file `meql`. In future releases of `meq_prime` it will be possible to recompute a missing modular polynomial and add it to the currently used database.

At the moment the input routine supports input files in *ascii* format, in *binary* format, in *gzipped ascii* format, and *gzipped binary* format. Note that of course the `gzip` program must be installed on your machine if you want to use the gzipped input versions. Furthermore it is important to note that binary formats differ on different architectures, such that we suggest to use either *ascii* or *gzipped ascii* format.

Constructors/Destructor

`ct meq_prime ()`
constructs an uninitialized instance.

`ct meq_prime (const meq_prime & m)`
copy constructor.

`ct meq_prime (udigit l)`
initializes index of modular polynomial with l, i.e. in a subsequent call to one of the polynomial building functions the l-th modular polynomial will be used. l is assumed to be an odd prime number. If this condition is not satisfied, the `lidia_error_handler` is invoked.

`dt ~meq_prime ()`

Assignments

The operator = is overloaded. Moreover there exists the following assignment function. Let m be an instance of meq_prime.

`void` m`.set_prime (udigit` l`)`
sets the index of the modular polynomial to be read in to l. l is assumed to be an odd prime number. If this condition is not satisfied, the lidia_error_handler is invoked.

Access Methods

Let m be an instance of meq_prime.

`udigit` m`.get_prime () const`
returns the prime index l of the currently used modular polynomial.

High-Level Methods and Functions

Let m be an instance of meq_prime. The following evaluation functions for the currently active modular polynomial can be used.

`void` m`.build_poly_in_X (Fp_polynomial &` f`, const bigmod &` y`)`
evaluate the modular polynomial of current index l at Y coordinate y. Set f to the result of this evaluation. If the l-th modular polynomial can't be found in the current database, the lidia_error_handler is invoked.

`void` m`.build_poly_in_Y (Fp_polynomial &` f`, const bigmod &` x`)`
evaluate the modular polynomial of current index l at X coordinate x. Set f to the result of this evaluation. If the l-th modular polynomial can't be found in the current database, the lidia_error_handler is invoked.

Input/Output

Input/Output of an instance of meq_prime is currently not possible.

See also

eco_prime.

Notes

At the moment it should not be necessary for a user to use this class. When the Elliptic Curve Counting Package will however be increased in future, this will probably also cause changes in the class meq_prime. Then we will offer a more comfortable usage of this class, especially for non experts. Moreover re-computation of missing information will be possible in future versions.

Author

Frank Lehmann, Markus Maurer, Volker Müller.

trace_mod

Name

`trace_mod` ..... class for storing the trace of the Frobenius endomorphism modulo a prime power

Abstract

The class `trace_mod` is used as a container to store information on the trace of the Frobenius endomorphism computed with the class `eco_prime`.

Description

`trace_mod` is a class for storing information about the trace of the Frobenius endomorphism computed with the help of `eco_prime`. The class offers essentially only administrative functions and simplifies handling these information.

The class stores a variable l and a vector which holds the different values of the trace of Frobenius modulo l. Moreover it determines a so called density which is defined as the size of this vector divided by l. This density is used in the class `trace_list` to sort different `trace_mod`s.

Constructors/Destructor

`ct trace_mod ()`
constructs an empty instance.

`dt ~trace_mod ()`

Assignments

The operator `=` is overloaded. Moreover there exists the following assignment function. Let t be an instance of `trace_mod`.

`void` m`.set_first_element (udigit` ll`, udigit` cc`)`
sets the internal value of l to ll and the first entry of the vector to cc. Moreover the size of the vector is set to one.

`void` m`.set_vector (udigit` ll`, const base_vector< udigit > &` cc`)`
sets the internal value of l to ll and the internal vector to cc.

Access Methods

Let m be an instance of `trace_mod`.

`lidia_size_t` m`.get_size_of_trace_mod () const`
: returns the size of the internal vector, i.e. the number of possible candidates for the trace of Frobenius modulo the current prime l.

`udigit` m`.get_modulus () const`
: returns the modulus l.

`udigit` m`.get_first_element () const`
: returns the first element of the internal vector, i.e. the element stored at index 0. If the vector is empty, the `lidia_error_handler` is invoked.

`udigit` m`.get_element (udigit` n`) const`
: returns the element of the internal vector stored at index n. If n is bigger than the size of the internal vector, the `lidia_error_handler` is invoked.

`void swap (trace_mod &` s`, trace_mod &` t`)`
: swaps the two instances.

Comparisons

The operators `==`, `<`, `<=`, `>=`, and `>` are overloaded. These operators compare the density of two instances of `trace_mod`.

`bool cmp (const trace_mod &` t`, const trace_mod &` s`)`
: returns -1, if the density of t is smaller than the density of s, 0 if the two densities are equal, and 1 otherwise.

Input/Output

The `ostream` operator `<<` and the `istream` operator `>>` have been overloaded. The `istream` operator first expects input of the prime l and then input of vector of possibilities for the trace of Frobenius modulo l.

See also

`eco_prime`, `trace_list`.

Notes

The Elliptic Curve Counting Package will change in future releases. This will also affect the class `trace_mod`.

Author

Markus Maurer, Volker Müller.

trace_list

Name

`trace_list` ..... class for storing a list of `trace_mod`s

Abstract

The class `trace_list` is used for storing information on the trace of the Frobenius endomorphism computed with the class `eco_prime` and stored in instances of the class `trace_mod`. Moreover `trace_list` offers a Babystep Giantstep algorithm to find a candidate for the group order.

Description

`trace_list` is a class for storing the information computed with the class `eco_prime`. These information is stored in several instances of the class `trace_mod`. Moreover there exist function to decide when enough information has been computed to start a combination step and to determine a candidate for the group order with a Babystep Giantstep type algorithm.

The class stores a list of `trace_mod` which contain information about traces modulo primes. In this list only these `trace_mod`s are stored, which do not contain the exact value of the trace modulo the corresponding prime l. In addition to this list an instance of `trace_list` contains two variables C_3 and M_3, which hold integers such that the trace of Frobenius is congruent to C_3 modulo M_3. Moreover some variable stores the size of the Hasse interval for the actual computation.

After the combination step two more moduli are set. The variables M_1, M_2 hold the moduli used for the Babystep, Giantstep list, respectively. A description of the combination step can be found in the master thesis of Markus Maurer [44].

Constructors/Destructor

`ct trace_list ()`

constructs an uninitialized instance.

`ct trace_list (const bigint & q)`

constructs an instance for a point computation over a field with q elements, i.e. the size of the Hasse interval is $4\sqrt{q}$ in this case.

`dt ~trace_list ()`

Assignments

Let t be an instance of `trace_list`.

`void` t`.clear ()`
deletes all internal data, except the size of the Hasse interval.

`void` t`.set_curve (const elliptic_curve< gf_element > &` e`)`
Initializes with curve e.

`static void set_info_mode (int` i `= 0)`
set the internal info variable to i. If this variable is not zero, then some information is printed during the Babystep Giantstep algorithm; otherwise no output is done. The default value for the info variable is zero.

Access Methods

Let t be an instance of `trace_list`.

`bigint` t`.get_M1 () const`
returns the modulus for the Babystep list.

`bigint` t`.get_M2 () const`
returns the modulus for the Giantstep list.

`bigint` t`.get_M3 () const`
returns the modulus for which we know the trace of Frobenius exact.

`bigint` t`.get_C3 () const`
returns the trace of Frobenius modulo C_3.

`const sort_vector< trace_mod >` t`.get_list () const`
returns the list of `trace_mod`s stored internally.

`bigint` t`.number_of_combinations ()`
returns the total number of possibilities which have to be checked. If the product of the three moduli is not bigger than the size of Hasse's interval, then -1 is returned.

High-Level Methods and Functions

Let t be an instance of `trace_list`.

`bool` t`.append (const trace_mod &` m`)`
appends m to the `trace_list` t. Then it checks whether the number of possible values is smaller than some predefined value and the product of all internal moduli is big enough. If these two conditions are fulfilled, then `true` is returned, otherwise `false` is returned.

`bool` t`.split_baby_giant (sort_vector< bigint > &` b`, sort_vector< bigint > &` g`)`
sets b to the list of Babystep indices, g to the list of Giantstep indices and returns `true`, if the number of possible candidates is smaller than some predefined bound. Otherwise both vectors are emptied and `false` is returned.

`bigint bg_algorithm (const point< ff_element > & `P`, sort_vector< bigint > & `b`, sort_vector< bigint > & `g`, const bigint & `q`)`

uses the two lists for a Babystep Giantstep algorithm to find a multiple of the order of the point P in the Hasse interval. This value is returned.

`bigint bg_search_for_order ()`

Generates a random point P and uses the trace information for a Babystep Giantstep algorithm to find a multiple of the order of the point P in the Hasse interval. This value is returned.

`bigint simple_search_for_order ()`

Generates a random point P, computes all possible traces, and checks for each trace candidate, whether the corresping group order candidate is a multiple of the order of the point P in the Hasse interval. This value is returned.

`bool baby_giant_lists_correct (const bigint & `n`)`

Uses the group order n to verify, that the baby step and the giant step list is correct, in the sense, that the correct trace can be found in the lists.

Input/Output

The `istream` operator `>>` and the `ostream` operator `<<` are overloaded.

See also

`eco_prime`, `trace_mod`.

Notes

The Elliptic Curve Counting Package will change in future releases. Then probably the interface of the class `trace_list` will also change.

Author

Markus Maurer, Volker Müller.

The LiDIA gec package

Chapter 21

Generating Elliptic Curves for Use in Cryptography

gec

Name

`gec` . . . . . virtual base class for generating elliptic curves for use in cryptography

Abstract

The class `gec` is a virtual base class for the generation of elliptic curves over finite fields suitable for use in cryptography.

Description

The class `gec` is a virtual base class for the generation of elliptic curves over finite fields suitable for use in cryptography. It holds the attributes which are common to all inheriting classes. In addition, it provides the corresponding accessors and mutators. Currently, `gec` serves as a virtual base class for the classes `gec_complex_multiplication`, `gec_point_counting_mod_p`, and `gec_point_counting_mod_2n`.

Let p be a prime and $q = p^m$ be a prime power. An elliptic curve over $\mathbb{F}_q$ is a tuple of the form $(a_1, a_2, a_3, a_4, a_6) \in \mathbb{F}_q^5$. We denote an elliptic curve by E. Associated to each tuple is an equation of the form

$$y^2 + a_1xy + a_3y \quad = \quad x^3 + a_2x^2 + a_4x + a_6 \; . \tag{21.1}$$

Equation (21.1) is called a *long Weierstrass equation*. The group of rational points of E over $\mathbb{F}_q$ is the set of solutions $(u, v) \in \mathbb{F}^2$ together with the point at infinity. We write $E(\mathbb{F}_q)$ for the group of rational points of E over $\mathbb{F}_q$. It is well known that $E(\mathbb{F}_q)$ carries a group structure with the point at infinity acting as the zero element.

Now let $p \geq 5$. It is easy to see that using affine linear maps the long Weierstrass equation (21.1) may be transformed to an equation of the form

$$y^2 \quad = \quad x^3 + a_4'x + a_6' \; . \tag{21.2}$$

Hence if $p \geq 5$ we simply write (a_4, a_6) for an elliptic curve over $\mathbb{F}_q$.

Next, let $p = 2$. It is a basic fact that any elliptic curve whose group of rational points is suitable for use in cryptography may be written as a Weierstrass equation of the form

$$y^2 + xy \quad = \quad x^3 + a_2'x^2 + a_6' \; . \tag{21.3}$$

Hence if $p = 2$ we simply write (a_2, a_6) for an elliptic curve over $\mathbb{F}_q$.

In order to be suitable for use in cryptography, an elliptic curve has to respect some requirements. For instance, if $E(\mathbb{F}_q)$ should be in conformance with the German Digital Signature Act we have to consider requirements published by the German Information Security Agency (GISA/BSI, see [27]). In case of $q = p$ [27] requires:

1. $|E(\mathbb{F}_p)| = r \cdot k$ with a prime $r > 2^{159}$ and a small positive integer k (e.g. $k \leq 4$).

2. The primes r and p are different.
3. The order of p in $\mathbb{F}_r^\times$ is at least $B := 10^4$.
4. The class number of the maximal order which contains End(E) is at least $h_0 := 200$.

In addition, in case of $q = 2^m$ [27] lists:

1. $|E(\mathbb{F}_{2^m})| = r \cdot k$ with a prime $r > 2^{159}$ and a small positive integer k (e.g. $k \leq 4$).
2. m is prime.
3. The j-invariant of E is not in $\mathbb{F}_2$.
4. The order of 2^m in $\mathbb{F}_r^\times$ is at least $B := 10^4$.
5. The class number of the maximal order which contains End(E) is at least $h_0 := 200$.

However, the choices B and h_0 may be set differently by using the mutator of the the boolean `according_to_bsi`. If it is set to `false` the last requirement is not taken into account in both cases. In addition, B has to be at least B_0 in this case, where B_0 is minimal with $r^{B_0} > 2^{1999}$.

A lower bound of the bitlength of r and an upper bound of k are stored in the attributes `lower_bound_bitlength_r` and `set_upper_bound_k`, respectively. Per default, we set their values to the requirements cited above. The default values are set in the file gec.cc in the source-directory. They may be changed using the corresponding mutators as explained below. If another security level is required, the user may invoke `set_lower_bound_bitlength_r` or `set_upper_bound_k` to set different bounds for r and k, respectively.

The private boolean VERBOSE may be used to get a lot of information during the computation. It may be accessed and changed using the corresponding accessor and mutator, respectively.

In addition, the private boolean `efficient_curve_parameters` may be used to generate curves with $a_4 = -3$, if a curve over $\mathbb{F}_p$ is searched. More precisely, if `efficient_curve_parameters` is set to true, a curve with $a_4 = -3$ will be output. Otherwise, a_4 is some element of $\mathbb{F}_p$. In case of the CM-method, the default value is `false`. In case of the point counting method, its default value is `true`. It may be accessed and changed using the corresponding accessor and mutator, respectively.

As `gec` is a virtual basic class the public available constructor and destructor are redundant.

Constructors/Destructor

`ct gec ()`
initializes an empty instance.

`ct gec (ostream & out)`
initializes an empty instance to write data to files.

`dt ~gec ()`

Assignments

Let I be an instance of `gec`.

`void` I`.set_field (const bigint &` l`)`
sets $p = l$. If l is not a prime ≥ 5, the `lidia_error_handler` is invoked.

`void` I`.set_degree (lidia_size_t` d`)`

sets degree of finite field $\mathbb{F}_q$ over its prime field to d. If d is not positive, the `lidia_error_handler` is invoked.

Mutators of security level.

`void` I`.set_lower_bound_bitlength_r (lidia_size_t` b_0`)`

sets lower bound of bitlength of r to b_0. The prime r will satisfy $r > 2^{b_0-1}$. If b_0 is not positive, the `lidia_error_handler` is invoked.

`void` I`.set_upper_bound_k (const bigint &` k_0`)`

sets upper bound of k to k_0. The cofactor k will respect $k \leq k_0$. If k_0 is not positive, the `lidia_error_handler` is invoked.

Mutators of default variables to set security level.

`void` I`.set_default_lower_bound_bitlength_r (lidia_size_t` b_0`)`

sets default lower bound of bitlength of r to b_0. The default value is 160. If b_0 is not positive, the `lidia_error_handler` is invoked.

`void` I`.set_default_upper_bound_k (const bigint &` k_0`)`

sets default upper bound of k to k_0. The default value is 4. If k_0 is not positive, the `lidia_error_handler` is invoked.

`void` I`.set_default_lower_bound_extension_bitlength (lidia_size_t` l`)`

sets default lower bound of the bitlength of finite fields in which the discrete logarithm problem is considered to be intractable, to l. The default value is 2000. If l is not positive, the `lidia_error_handler` is invoked.

Mutators to set variables in the context of the requirements of the German Information Security Agency (GISA, BSI).

`void` I`.set_according_to_BSI (bool` b`)`

if b is equal to `true`, the requirements 1 - 4 or 1 - 5 cited above are respected, respectively. If b is equal to `false`, requirement 4 or 5 is ignored, respectively. In addition, in requirement 3 or 4 we only ensure that the order of q in $\mathbb{F}_r^\times$ is at least B_0, where B_0 is minimal with $r^{B_0} > 2^{b_0-1}$. b_0 is equal to `default_lower_bound_extension_bitlength`.

`void` I`.set_BSI_lower_bound_h_field (lidia_size_t` h_0`)`

ensures that the maximal imaginary quadratic order corresponding to Δ has a class number at least h_0. The default value is 200 (see requirement 4). This function is only relevant if `according_to_BSI` is equal to `true`. If h_0 is not positive, the `lidia_error_handler` is invoked.

`void` I`.set_BSI_lower_bound_extension_degree (lidia_size_t` d_0`)`

ensures that the order of q in $\mathbb{F}_r^\times$ is at least d_0. The default value is 10000 (see requirement 3). This function is only relevant if `according_to_BSI`=`true`. If d_0 is not positive, the `lidia_error_handler` is invoked.

`void` I`.set_verbose_level (bool` v`)`

sets VERBOSE to v.

`void` I`.set_efficient_curve_parameters (bool` e`)`

sets `efficient_curve_parameters` to e.

Access Methods

Let I be an instance of `gec`. Most of the accessors are only meaningful if the method `generate` has been invoked before. Otherwise, the accessor returns 0.

Accessors for finite field, elliptic curve and its order.

`const bigint &` I`.get_p () const`

returns p. Either p has been set using `set_field` or the method `generate` has been invoked before. Otherwise, 0 is returned.

`const bigint &` I`.get_q () const`

returns the prime power $q = p^m$. The method `generate` has to be invoked before. Otherwise, 0 is returned.

`const lidia_size_t` I`.get_degree () const`

returns the degree m. Either m has been set using `set_degree` or the method `generate` has to be invoked before. Otherwise, 0 is returned.

`const bigint &` I`.get_r () const`

returns the prime r. The method `generate` has to be invoked before. Otherwise, 0 is returned.

`const bigint &` I`.get_k () const`

returns the cofactor k. The method `generate` has to be invoked before. Otherwise, 0 is returned.

`const gf_element &` I`.get_a2 () const`

returns the parameter a_2 of the curve E. The result is only meaningful if $p = 2$. The method `generate` has to be invoked before. Otherwise, 0 is returned.

`const gf_element &` I`.get_a4 () const`

returns the parameter a_4 of the curve E. The result is only meaningful if $p \geq 5$. The method `generate` has to be invoked before. Otherwise, 0 is returned.

`const gf_element &` I`.get_a6 () const`

returns the parameter a_6 of the curve E. The method `generate` has to be invoked before. Otherwise, 0 is returned.

`const point<gf_element> &` I`.get_G () const`

returns a point $G \in E(\mathbb{F}_q)$ of order r. The method `generate` has to be invoked before. Otherwise, 0 is returned.

Accessors for static variables.

`lidia_size_t` I`.get_default_lower_bound_bitlength_r () const`

returns the default lower bound of bitlength of r.

`lidia_size_t` I`.get_default_upper_bound_k () const`

returns the default upper bound of k.

`const bigint &` I`.get_default_lower_bound_extension_bitlength () const`

returns the default lower bound of the bitlength of finite fields in which the discrete logarithm problem is considered intractable. Only meaningful, if `according_to_BSI` is equal to `false`.

Accessors for non-static variables to set security level.

`lidia_size_t` I`.get_lower_bound_bitlength_r () const`

returns the lower bound of bitlength of r. If it is not set before, 0 is returned.

`const bigint &` I`.get_upper_bound_k () const`

returns the upper bound of k. If it is not set before, 0 is returned.

`const bigint &` I`.get_delta_field () const`

returns Δ_K. Only meaningful, if `according_to_BSI` is equal to `true`. If this is not the case or if the generate() is not invoked before, 0 is returned.

`lidia_size_t` I`.get_h_field () const`

returns $h(\Delta_K)$. Only meaningful in case of the complex multiplication approach and if `according_to_BSI` is equal to `true`. If this is not the case or if the generate() is not invoked before, 0 is returned.

`lidia_size_t` I`.get_lower_bound_h_field () const`

returns lower bound of $h(\Delta_K)$. Only meaningful if `according_to_BSI` is equal to `true`. If this is not the case or if the generate() is not invoked before, 0 is returned.

Accessors to get variables in the context of the requirements of the German Information Security Agency (GISA, BSI).

`bool` I`.get_according_to_BSI () const`

returns `true` if and only if the generated curve respects the requirements of GISA.

`lidia_size_t` I`.get_BSI_lower_bound_h_field () const`

returns the lower bound of the class number of the maximal order containing the endomorphism ring of the curve.

`lidia_size_t` I`.get_BSI_lower_bound_extension_degree () const`

returns the lower bound of the order of q in $\mathbb{F}_r^\times$.

`bool` I`.get_verbose_level () const`

returns the value of VERBOSE.

`bool` I`.get_efficient_curve_parameters () const`

returns the value of the attribute `efficient_curve_parameters`, that is if a curve with $a_4 = -3$ is searched or not.

Accessors which are specific to `gec_complex_multiplication`.

`const bigint &` I`.get_delta () const`

returns Δ. If it is not set or computed before, 0 is returned.

`lidia_size_t` I`.get_h () const`

returns $h(\Delta)$. If it is not set or computed before, 0 is returned.

Input/Output

Input/Output of instances of `gec` is currently not possible.

See also

`gec_complex_multiplication`, `gec_point_counting_mod_p`, `gec_point_counting_mod_2n`.

Notes

Author

Harald Baier

gec_complex_multiplication

Name

`gec_complex_multiplication`....class for generating elliptic curves for use in cryptography by the complex multiplication method

Abstract

The class `gec_complex_multiplication` generates elliptic curves over finite fields suitable for use in cryptography. It uses the theory of complex multiplication.

Description

`gec_complex_multiplication` is a class for generating elliptic curves over finite fields suitable for use in cryptography. The class implements algorithms which use the theory of complex multiplication. Currently, curves over finite prime fields and curves over Optimal Extension Fields may be generated using this package.

A central term in the theory of complex multiplication is an *imaginary quadratic discriminant*. We denote such a discriminant by Δ. An imaginary quadratic discriminant is simply a negative integer congruent 0 or 1 modulo 4. Associated to Δ is a class number, which we denote by $h(\Delta)$. In addition, for each imaginary quadratic discriminant Δ there exists a fundamental discriminant Δ_K; Δ_K is the largest imaginary quadratic discriminant dividing Δ.

Let p be a prime, and let m be either 1 or a prime. We set $q = p^m$. Let $E = (a_4, a_6)$ be an elliptic curve defined over $\mathbb{F}_q$. E is suitable for cryptographic purposes, if it respects the following conditions ([27]):

1. We have $|E(\mathbb{F}_{p^m})| = r \cdot k$ with a prime $r > 2^{159}$ and a positive integer $k \leq 4$.
2. The primes r and p are different.
3. The order of q in $\mathbb{F}_r^\times$ is at least $B := 10^4$.
4. The class number of the maximal order which contains $\mathrm{End}(E)$ is at least $h_0 := 200$.

However, the choices B and h_0 may be set differently be setting the boolean `according_to_bsi` to `false`. If `according_to_bsi=false`, the last requirement is not taken into account. In addition, B has to be at least B_0 in this case, where B_0 is minimal with $r^{B_0} > 2^{1999}$.

If another security level is required, the user may invoke `set_lower_bound_bitlength_r` or `set_upper_bound_k` to set different bounds for r and k, respectively. For example, if r has to be of bitlength at least 180, that is $r > 2^{179}$, one has to invoke `set_lower_bound_bitlength_r` with 180 as argument.

During the computation the algorithm computes a class polynomial, that is a polynomial with coefficients in $\mathbb{Z}$ generating the ring class field of the imaginary quadratic order of discriminant Δ over $\mathbb{Q}(\sqrt{\Delta})$. If $3 \nmid \Delta$, we make use of a polynomial G due to Atkin/Morain to generate the ring class field. If in addition $\Delta \equiv 1 \bmod 8$, the algorithm uses a polynomial W due to Yui/Zagier. If $3 \nmid \Delta$, the class member `generation_mode` is set to

3. If in addition $\Delta \equiv 1 \bmod 8$, `generation_mode` is set to 4. Furthermore, if `generation_mode` $= 1$, then the ring class polynomial H is used. The user may use the method `set_generation_mode` to change the attribute `generation_mode`. However, he has to ensure `generation_mode` $\in \{1, 3, 4\}$. Otherwise, it is set to 1.

In order to use the Fixed Field Approach, the user has to define the environment variable `LIDIA_DISCRIMINANTS`. It has to point to the directory, where the discriminants reside. For instance, if LiDIA is installed in /usr/local/LiDIA on a UNIX-like machine, the user has to set `LIDIA_DISCRIMINANTS`=/usr/local/LiDIA/share/LiDIA/Discriminants.

The boolean VERBOSE may be set to `true` to get a lot of information during the computation.

Constructors/Destructor

`ct gec_complex_multiplication ()`
initializes an empty instance.

`ct gec_complex_multiplication (const bigint & D)`
initializes an instance and sets $\Delta = D$.

`ct gec_complex_multiplication (lidia_size_t d)`
initializes an instance and sets the extension degree of the finite field $\mathbb{F}_q$ over $\mathbb{F}_p$ to d.

`ct gec_complex_multiplication (ostream & out)`
initializes an empty instance and writes some timings during the generation to the output stream out.

`dt ~gec_complex_multiplication ()`

Assignments

Let I be an instance of `gec_complex_multiplication`. We point to the methods of the base class `gec`.

`void` I`.set_delta (const bigint & D)`
sets $\Delta = D$. If D is positive or not $\equiv 0$ or 1 modulo 4, the `lidia_error_handler` is invoked.

`void` I`.set_generation_mode (unsigned int i)`
sets `generation_mode` $= i$. The user is responsible that i and Δ fit together.

`void` I`.set_complex_precision (long i)`
sets the floating point precision to compute the class polynomial corresponding to `generation_mode` to i. The user is responsible that the precision is sufficient to get a correct polynomial.

`void` I`.set_lower_bound_class_number (const long i)`
only relevant in case of the Fixed Field Approach or if an OEF is used. Sets the lower bound of the class number interval of available discriminants to i. The default value is 200. If $i < 200$ or $i \geq u$, where u denotes the current upper bound of the interval, the `lidia_error_handler` is invoked.

`void` I`.set_upper_bound_class_number (const long u)`
only relevant in case of the Fixed Field Approach or if an OEF is used. Sets the upper bound of the class number interval of available discriminants to u. The default value is 1000. If $u > 1000$ or $i \geq u$, where i denotes the current lower bound of the interval, the `lidia_error_handler` is invoked.

`void` *I*`.set_class_polynomial (const polynomial<bigint> &` *g*`)`

sets the class polynomial to g. The user is responsible, that g actually is a class polynomial for the order of discriminant Δ and that the chosen `generation_mode` fit.

`void` *I*`.set_class_polynomial (const bigint &` *D*`, const lidia_size_t` *h*`, char * input_file)`

sets $\Delta = D$ and the class number to h. The user is responsible that $h(\Delta) = h$. Let $C = \sum_{k=0}^{h} c_k X^k$ denote a class polynomial corresponding to Δ. The coefficients are read from the file h_<h>/<D> in the directory <input_file>, beginning with c_h.

`void` *I*`.assign (const gec_complex_multiplication & J)`

initializes I with J.

Access Methods

Accessors which are specific to `gec_complex_multiplication`. For further accessors we refer to the base class `gec`.

`const bigint &` *I*`.get_delta () const`

returns Δ.

`lidia_size_t` *I*`.get_h () const`

returns $h(\Delta)$.

`unsigned int` *I*`.get_generation_mode () const`

returns the generation mode.

`long` *I*`.get_complex_precision () const`

returns the floating point precision to compute the class polynomial corresponding to `generation_mode`.

`const polynomial<bigint>` *I*`.get_class_polynomial () const`

returns the class polynomial.

High-Level Methods and Functions

Let I be an instance of `gec_complex_multiplication`.

`void` *I*`.generate ()`

generates the field (if not already set) and the elliptic curve corresponding to the security level in use.

`void` *I*`.generate_oef (lidia_size_t` *n*`, lidia_size_t` *m*`, lidia_size_t` h_0`)`

generates a prime p of bitlength n such that p^m is cardinality of an Optimal Extension Field. The user has to ensure that m is prime. In addition, the function returns an elliptic curve of prime order r over this field. (Hence `upper_bound_k` is set to 1) In order to be applicable, the user has to take care that the lower bound of the bitlength of r is at most nm.

`void` *I*`.compute_class_polynomial ()`

computes the class polynomial corresponding to the chosen discriminant Δ and generation mode. The computation uses an efficient arithmetic basing on ideas of Karatsuba.

Input/Output

Input/Output of instances of `gec_complex_multiplication` is currently not possible.

See also

`gec`, `gec_point_counting_mod_p`, `gec_point_counting_mod_2n`.

Notes

Author

Harald Baier

Bibliography

[1] Alford, W. R., and Pomerance, C. Implementing the self initializing quadratic sieve on a distributed network. In *Number Theoretic and Algebraic Methods in Computer Science, NTAMCS '93* (1995), A. J. van der Poorten, Ed., World Scientific, pp. 163–174.

[2] Babai, L. On Lovász' lattice reduction and the nearest lattice point problem. *Combinatorica 6*, 1 (1986), 1–13.

[3] Bach, E. Improved approximations for Euler products. In *Number Theory, Halifax, Nova Scotia 1994* (1995), K. Dilcher, Ed., vol. 15 of *CMS Proceedings*, Canadian Mathematical Society, pp. 13–28.

[4] Berger, F.-D. ECM – Faktorisieren mit elliptischen Kurven. Master's thesis, Fachbereich Informatik, Universität des Saarlandes, 1993. German. `http://www.informatik.tu-darmstadt.de/TI/Veroeffentlichung/reports/berger.diplom.ps.gz`.

[5] Biehl, I., and Buchmann, J. Algorithms for quadratic orders. In *Mathematics of Computation 1943–1993: a half-century of computational mathematics, Vancouver 1993* (1995), W. Gautschi, Ed., vol. 48 of *Proceedings of Symposia in Applied Mathematics*, American Mathematical Society, pp. 425–449.

[6] Boehm, H.-J., and Weiser, M. Garbage collection in an uncooperative environment. *Software Practice and Experience 18*, 9 (1988), 807–820.

[7] Borwein, J. M., and Borwein, P. B. *Pi and the AGM. A study in analytic number theory and computational complexity.* Canadian Mathematical Society Series of Monographs and Advanced Texts. John Wiley & Sons, 1987.

[8] Brent, R. P. Fast multiple-precision evaluation of elementary functions. *Journal of the ACM 23*, 2 (1976), 242–251.

[9] Brent, R. P. Multiple-precision zero finding methods and the complexity of elementary function evaluation. In *Proceedings of the Symposium on Analytic Computational Complexity, Pittsburgh, Pennsylvania, 1975* (1976), J. F. Traub, Ed., Academic Press, pp. 151–176.

[10] Brent, R. P., and McMillan, E. M. Some new algorithms for high-precision computation of Euler's constant. *Mathematics of Computation 34*, 149 (1980), 305–312.

[11] Buchmann, J., and Eisenbrand, F. On factor refinement in number fields. *Mathematics of Computation 68*, 225 (1999), 345–350.

[12] Buchmann, J., Jacobson, Jr., M. J., and Teske, E. E. On some computational problems in finite abelian groups. *Mathematics of Computation 66*, 220 (1997), 1663–1687.

[13] Buchmann, J., and Kessler, V. Computing a lattice basis from a generating system. Unpublished. `http://www.informatik.tu-darmstadt.de/TI/Veroeffentlichung/reports/reduced_basis.ps.gz`, 1992.

[14] Buchmann, J., and Lenstra, Hendrik W., J. Computing maximal orders and decomposing primes in number fields. Unpublished.

[15] Buchmann, J., and Maurer, M. Approximate evaluation of $L(1, \chi_\Delta)$. Tech. Rep. TI-6/98, Technische Universität Darmstadt, Fachbereich Informatik, 1998. `http://www.informatik.tu-darmstadt.de/TI/Veroeffentlichung/TR/`.

[16] Buchmann, J., Thiel, C., and Williams, H. C. Short representation of quadratic integers. In *Computational Algebra and Number Theory, Sydney 1992* (1995), W. Bosma and A. J. van der Poorten, Eds., vol. 325 of *Mathematics and its Applications*, Kluwer Academic Publishers, pp. 159–185.

[17] Cohen, H. *A Course in Computational Algebraic Number Theory*, vol. 138 of *Graduate Texts in Mathematics*. Springer-Verlag, 1995.

[18] Cohen, H., Diaz y Diaz, F., and Olivier, M. Calculs de nombres de classes et de régulateurs de corps quadratiques en temps sous-exponentiel. In *Séminaire de Théorie des Nombres, Paris 1990–1991* (1993), S. David, Ed., vol. 108 of *Progress in Mathematics*, Birkhäuser, pp. 35–46. French.

[19] Cormen, T. H., Leiserson, C. E., and Rivest, R. L. *Introduction to Algorithms*. McGraw-Hill, 1990.

[20] de Weger, B. M. M. *Algorithms for Diophantine approximation*, vol. 65 of *CWI Tract*. Centrum voor Wiskunde en Informatica, 1989.

[21] Dekker, T. J. A floating-point technique for extending the available precision. *Numerische Mathematik 18* (1971/1972), 224–242.

[22] Denny, T. F. Faktorisieren mit dem Quadratischen Sieb. Master's thesis, Fachbereich Informatik, Universität des Saarlandes, 1993. German. `http://www.informatik.tu-darmstadt.de/TI/Veroeffentlichung/reports/denny.diplom.ps.gz`.

[23] Domich, P. D. Residual Hermite normal form computations. *ACM Transactions on Mathematical Software 15*, 3 (1989), 275–286.

[24] Düllmann, S. *Ein Algorithmus zur Bestimmung der Klassengruppe positiv definiter binärer quadratischer Formen*. PhD thesis, Fachbereich Informatik, Universität des Saarlandes, 1991. German. `http://www.informatik.tu-darmstadt.de/TI/Veroeffentlichung/reports/duell.diss.ps.gz`.

[25] Fincke, U., and Pohst, M. Improved methods for calculation vectors of short length in a lattice, including a complexity analysis. *Mathematics of Computation 44*, 170 (1985), 463–471.

[26] Fung, G. W., and Williams, H. C. Quadratic polynomials which have a high density of prime values. *Mathematics of Computation 55*, 191 (1990), 345–353.

[27] Geeignete kryptoalgorithmen, in erfüllung der anforderungen nach par. 17(1) sigg vom 16. mai 2001 in verbindung mit par. 17(2) sigv vom 22. oktober 1997, July 2001. Bundesanzeiger Nr. 158 - Seite 18 562 vom 24. August 2001.

[28] Hafner, J. L., and McCurley, K. S. A rigorous subexponential algorithm for computation of class groups. *Journal of the American Mathematical Society 2*, 4 (1989), 837–850.

[29] Hardy, G. H., and Littlewood, J. E. Partitio numerorum III: On the expression of a number as a sum of primes. *Acta Mathematica 44* (1923), 1–70.

[30] Havas, G., Holt, D. F., and Rees, S. Recognizing badly presented $\mathbb{Z}$-modules. *Linear Algebra and its Applications 192* (1993), 137–163.

[31] Havas, G., and Majewski, B. S. Integer matrix diagonalization. *Journal of Symbolic Computation 24*, 3–4 (1997), 399–408.

[32] Havas, G., and Sterling, L. S. Integer matrices and abelian groups. In *EUROSAM '79* (1979), vol. 72 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 431–451.

[33] IEEE. *754-1985. IEEE standard for binary floating-point arithmetic*, 1985.

[34] Jacobson, Jr., M. J. Experimental results on class groups of real quadratic fields (extended abstract). In *Algorithmic Number Theory, ANTS-III* (1998), J. P. Buhler, Ed., vol. 1423 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 463–474.

[35] Jacobson, Jr., M. J. Applying sieving to the computation of quadratic class groups. *Mathematics of Computation 68*, 226 (1999), 859–867.

[36] Jacobson, Jr., M. J. *Subexponential Class Group Computation in Quadratic Orders*. PhD thesis, Technische Universität Darmstadt, Fachbereich Informatik, Darmstadt, Germany, 1999.

[37] Jacobson, Jr., M. J., Lukes, R. F., and Williams, H. C. An investigation of bounds for the regulator of quadratic fields. *Experimental Mathematics 4*, 3 (1995), 211–225.

[38] Knuth, D. E. *The Art of Computer Programming — Seminumerical Algorithms*, 2nd ed., vol. 2. Addison-Wesley, 1981.

[39] Lenstra, A. K., Lenstra, Jr., H. W., and Lovász, L. Factoring polynomials with rational coefficients. *Mathematische Annalen 261*, 4 (1982), 515–534.

[40] Lenstra, H. W. Factoring integers with elliptic curves. *Annals of Mathematics 126*, 3 (1987), 649–673.

[41] Lenstra, Jr., H. W. On the calculation of regulators and class numbers of quadratic fields. In *Journees Arithmetiques, Exeter 1980*, J. V. Armitage, Ed., vol. 56 of *London Mathematical Society Lecture Notes Series*. Cambridge University Press, 1982, pp. 123–150.

[42] Linnainmaa, S. Software for doubled-precision floating-point computations. *ACM Transactions on Mathematical Software 7*, 3 (1981), 172–283.

[43] Littlewood, J. E. On the class number of the corpus $P(\sqrt{-k})$. In *Proceedings of the London Mathematical Society*, vol. 27 of *2nd series*. Cambridge University Press, 1928, pp. 358–372.

[44] Maurer, M. Eine Implementierung des Algorithmus von Atkin zur Berechnung der Punktanzahl elliptischer Kurven über endlichen Primkörpern der Characteristik größer drei. Master's thesis, Fachbereich Informatik, Universität des Saarlandes, 1994. German. `http://www.informatik.tu-darmstadt.de/TI/Veroeffentlichung/reports/maurer.diplom.ps.gz`.

[45] Maurer, M. *Regulator approximation and fundamental unit computation for real quadratic orders*. PhD thesis, Technische Universität Darmstadt, Fachbereich Informatik, 2000.

[46] Mollin, R. A., and Williams, H. C. Computation of the class number of a real quadratic field. *Utilitas Mathematica 41* (1992), 259–308.

[47] Müller, A. Effiziente Algorithmen für Probleme der linearen Algebra über $\mathbb{Z}$. Master's thesis, Fachbereich Informatik, Universität des Saarlandes, 1994. German. `http://www.informatik.tu-darmstadt.de/TI/Veroeffentlichung/reports/acmueller.diplom.ps.gz`.

[48] Müller, A. Eine fft-continuation für die elliptische kurven methode. Master's thesis, Fachbereich Informatik, Universität des Saarlandes, 1995. German. `http://www.informatik.tu-darmstadt.de/TI/Veroeffentlichung/reports/amueller.diplom.ps.gz`.

[49] Müller, V. *Ein Algorithmus zur Bestimmung der Punktanzahl elliptischer Kurven über endlichen Körpern der Charakteristik größer drei*. PhD thesis, Fachbereich Informatik, Universität des Saarlandes, 1995. German. `http://www.informatik.tu-darmstadt.de/TI/Veroeffentlichung/reports/vmueller.diss.ps.gz`.

[50] Pfahler, T. Polynomfaktorisierung über endlichen Körpern. Master's thesis, Technische Universität Darmstadt, Fachbereich Informatik, 1998. German. `http://www.informatik.tu-darmstadt.de/TI/Veroeffentlichung/reports/pfahler.diplom.ps.gz`.

[51] Pohst, M., and Zassenhaus, H. *Algorithmic algebraic number theory*, vol. 30 of *Encyclopedia of Mathematics and its Applications*. Cambridge University Press, 1989.

[52] Pomerance, C. Analysis and comparison of some integer factoring algorithms. In *Computational methods in number theory I*, H. W. Lenstra, Jr. and R. Tijdeman, Eds., vol. 154 of *Mathematical Centre Tracts*. Mathematisch Centrum, 1982, pp. 89–139.

[53] Priest, D. B. *On properties of floating point arithmetics: numerical stability and the cost of accurate computations*. PhD thesis, Berkeley, 1992.

[54] Riesel, H. *Prime Numbers and Computer Methods for Factorization*, 2nd ed., vol. 126 of *Progress in Mathematics*. Birkhäuser, 1994.

[55] Schnorr, C.-P., and Euchner, M. Lattice basis reduction: Improved practical algorithms and solving subset sum problems. *Mathematical Programming 66A*, 2 (1994), 181–199.

[56] Schoof, R. J. Quadratic fields and factorization. In *Computational methods in number theory II*, H. W. Lenstra, Jr. and R. Tijdeman, Eds., vol. 155 of *Mathematical Centre Tracts*. Mathematisch Centrum, 1982, pp. 235–286.

[57] Shanks, D. Class number, a theory of factorization and genera. In *1969 Number Theory Institute* (1971), D. J. Lewis, Ed., vol. 20 of *Proceedings of Simposia in Pure Mathematics*, American Mathematical Society, pp. 415–440.

[58] Shanks, D. Systematic examination of Littlewood's bounds on $L(1, \chi)$. In *Analytic number theory, St. Louis 1972* (1973), H. G. Diamond, Ed., vol. 24 of *Proceedings of Symposia in Pure Mathematics*, American Mathematical Society, pp. 267–283.

[59] Shoup, V. A new polynomial factorization algorithm and its implementation. *Journal of Symbolic Computation 20*, 4 (1995), 363–397.

[60] Sosnowski, T. Faktorisieren mit dem Quadratischen Sieb auf dem Hypercube. Master's thesis, Fachbereich Informatik, Universität des Saarlandes, 1994. German. `http://www.informatik.tu-darmstadt.de/TI/Veroeffentlichung/reports/sosnowski.diplom.ps.gz`.

[61] Spatscheck, O., O'Malley, S., Orman, H., and Schroeppel, R. Fast key exchange with elliptic curve systems. In *Advances in Cryptology — CRYPTO '95* (1995), D. Coppersmith, Ed., vol. 963 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 43–56.

[62] von zur Gathen, J., and Shoup, V. Computing Frobenius maps and factoring polynomials. *Computational Complexity 2*, 3 (1992), 187–224.

[63] Weber, D. Ein Algorithmus zur Zerlegung von Primzahlen in Primideale. Master's thesis, Fachbereich Informatik, Universität des Saarlandes, 1993. German. `http://www.informatik.tu-darmstadt.de/TI/Veroeffentlichung/reports/dweber.diplom.ps.gz`.

[64] Wetzel, S., and Backes, W. New results on lattice basis reduction in practice. In *Algorithmic Number Theory, ANTS-IV* (2000), W. Bosma, Ed., vol. 1838 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 135–152.