HIntLib Manual

Rudolf Schürer

17th February 2005

About This Document

To begin with, this document is a mess! It is supposed to document all the public interfaces provided by HIntLib. However, it is covering only about 15 % of them right now. It is also supposed to give examples of the usage of HIntLib, which—at the moment—it does not do at all.

This document refers to and is distributed with version 0.0.10 of HIntLib.

HIntLib is Copyright © 2002, 2003, 2004, 2005 by Rudolf Schürer under the GNU General Public License (GPL) (see Appendix B or the file COPYING in the source distribution). It is *not* available with the GNU Library General Public License (LGPL).

Contents

Chapter 1

Introduction

HINTLIB is a C++ library for high-dimensional numerical integration.

Chapter 2

Basic Types and Concepts

2.1 The Namespace HIntLib

All types, functions, and variables defined by HIntLib reside in the namespace `HIntLib`.

For reasons of conciseness, the namespace is never given explicitely in this document. However, the user should bear in mind that names like `u32` and `u64` in the next section are actually `HIntLib::u32` and `HIntLib::u64`, respectively.

One of the names that can be found in namespace `HIntLib` is another namespace: namespace `HIntLib::Private`. Names defined inside this namespace are meant to be used exclusively by the library. The user should not refer to any of these names directly.

All C++ preprocessor macros defined by HINTLIB start with the character sequence `HINTLIB_`. Names of this type should not be used for any other purpose.

2.2 Unsigned Numbers

Unsigned numbers play a major role in HINTLIB, since they are used for tasks as diverse as bit-maps or counting integrand evaluations.

The C++ language provides a number of unsigned types, however their size is not specified exactly. To ensure that the proper type is used in all situations, HINTLIB provides two typedefs: `u32` is the smallest unsigned integer type with at least 32 bits, `u64` the smallest type with at least 64 bits.

2.3 Counting Points

The type used by HINTLIB for counting points—for instance the number of allowed integrand evaluations, the number of abscissas of an interpolatory rule, or the current index in a low-discrepancy sequence—is `Index`.

`Index` is a typedef for either `u32` or `u64`, depending on the architecture. By default, HINTLIB uses 64 bits for `Index` if `unsigned long` has also at least 64 bits, assuming that the host computer has probably a 64-bit processor. If a different behavior is required, `Index` can be set to `u32` or `u64` by giving the options `--with-index=32` or `--with-index=64`, respectively, to `configure`. More information about configuring HINTLIB can be found in Appendix A.

2.4 Real Numbers

HINTLIB uses the type `real` for all real numbers.

By default, `real` is only a typedef for `double`. However, HINTLIB can be configured to use a different type to cope with increased precision or performance requirements. See Appendix A for details.

Unfortunately, the standard library header file `complex` defines a function with the name `real`. If this header file is used, care must be taken to avoid ambiguities.

2.5 Points in $\mathbb{R}^s$

Points $\boldsymbol{x} \in \mathbb{R}^s$ are represented by C-style arrays of `real`s, i. e., a point $\boldsymbol{x}$ is usually passed to a function or method as `const real*`. This design seems rather low-level, but provides the most efficient implementation of this crucial data structure.

If the user needs to allocate memory for a point with a dimension not known at compile time, the datatype `Point` should be used. `Point` is a typedef for `Array<real>`, therefore it does not cause any memory or performance penalties compared to `new[]`. However, it is exception save and ensures that the memory allocated is freed even in the case of an exception.

2.6 Hypercubes $C_s \subset \mathbb{R}^s$

All integration routines implemented in HINTLIB work on hyper-rectangular domains. The concept of such a region $C_s = [r_1, t_1] \times \cdots \times [r_s, t_s] \subset \mathbb{R}^s$ is available as class `Hypercube`:

```
class Hypercube
{
public:
  explicit Hypercube (unsigned s);
  Hypercube (unsigned s, real r, real t);
  Hypercube (unsigned s, const real r [], const real t []);
```

Constructors are provided to create all kinds of `Hypercube`s. To be precise, $[0,1]^s$, $[r,t]^s$, and $[r_i, t_i]^s$ can be created directly by the constructors given above.

```
  Hypercube (const Hypercube &);
  Hypercube& operator= (const Hypercube&);
```

`Hypercube`s have value semantic, both copy-constructor and assignment are provided. Assignment can not be used to change the dimension of a `Hypercube`.

```
  Hypercube (Hypercube &, unsigned dim);
```

This constructor creates a new `Hypercube` by splitting a given one by halving it along the specified dimension. The newly constructed `Hypercube` is initialized to one half of the old cube, the `Hypercube` passed in as argument is set to the other half.

```
  unsigned int getDimension() const;
  real getVolume () const;
  const real* getCenter() const;
  const real* getWidth () const;
        real  getCenter (unsigned) const;
        real  getWidth  (unsigned) const;
        real  getUpperBound (unsigned) const;
        real  getLowerBound (unsigned) const;
        real  getDiameter (unsigned) const;
```

A large number of methods for querying all kind of information about a `Hypercube` are provided. Constant-time inline implementations are available for all these methods, so no overhead is introduced when `Hypercube`s are used.

```
void set (unsigned dim, real a, real b);
```

Resets the lower and upper bound of the `Hypercube` for a certain dimension.

```
void move (unsigend dim, real distance);
```

Moves the cube a certain distance in the given direction.

```
void cutLeft  (unsigned dim);
void cutRight (unsigned dim);
```

Halves the `Hypercube` along a certain coordinate axis and sets it either to the left or to the right part of itself.

```
};
```

The following function can be used for `Hypercube`s.

```
bool operator== (const Hypercube &) const;
bool isUnitCube () const;
```

Comparing cubes.

```
bool isPointInside (const Hypercube &, const real[]);
Hypercube::Location whereIsPoint (const Hypercube &, const real[]);
```

These functions determine whether a given point is inside a `Hypercube`. `isPointInside()` returns `true` or `false`, without any special treatment for borderline cases. `whereIsPoint()` returns `INSIDE`, `OUTSIDE`, or `BORDER`, depending on the position of the point. These three values are defined in the enumeration `Hypercube::Location`.

```
std::ostream& operator<< (std::ostream &, const Hypercube &);
```

Prints a readable version (lower and upper bounds) of the cube into an `ostream`.

2.7 Integrand Functions

All integrand functions have to implement the following, self-explanatory interface. The user basically provides an implementation for `operator()`, the method that evaluates the integrand for a given point $\boldsymbol{x} \in \mathbb{R}^s$.

```
class Integrand
{
public:
  explicit Integrand (unsigned s);
  virtual ~Integrand ();
  unsigned getDimension() const;

  virtual real operator() (const real []) = 0;
  virtual real derivative (const real [], unsigned d);
};
```

The constructor expects the dimensionality of the integrand as parameter.

Some cubature rules base their calculations not only on function values, but also on first partial derivatives. If such a rule is used, an implementation for `derivative()` has to be provided in addition to `operator()()`—the default implementation in `Integrand` merely throws `DerivativeNotSupported`.

2.8 Integral Estimate with Estimated Error

Another concept that arises quite naturally during the implementation of numerical integration routines is the combination of two `real` values, the first one representing the estimation Qf of an integral, the second one representing the estimated error Ef of this approximation. An object `EstErr` is used for implementing this concept. Even though this type is rather simple and has only a small number of convenience methods, the use of this class simplifies many integration routines and avoids bugs.

```
class EstErr
{
public:
  EstErr ();
  EstErr (real newEst, real newErr);

  real getEstimate() const;
  real getError()    const;
  real getRelError() const;

  void set (real newEst, real newErr);
  void setNoErr (real newEst);
  EstErr& operator+= (const EstErr &);
  EstErr& operator-= (const EstErr &);

  void scale (real a);
};
std::ostream& operator<< (std::ostream &, const EstErr &);
```

2.9 Integration Routines

All integration routines are subclasses of `Integrator`. This class provides a common interface for calling integration routines and defines a number of possible status codes that inform the user about the outcome of the calculation.

```
class Integrator
{
public:
  enum Status {
    ABS_ERROR_REACHED, // Estimated error <= reqAbsError
    REL_ERROR_REACHED, // Estimated error <= reqRelError
    MAX_EVAL_REACHED,  // Max integ evaluations reached
    ERROR,             // Other reason
  };
```

An `Integrator` returns one of these status codes if it has been able to derive a result.

```
  Integrator ();
```

An `Integrator` has only a normal constructor. Copy constructor and assignment are private.

```
  virtual
  Status integrate (Integrand &,
                    const Hypercube &,
                    Index maxEval,
                    real reqRelError, real reqAbsError,
                    EstErr &) = 0;
```

This pure virtual function is overwritten by subclasses of `Integrator` to provide the actual implementation of the integration routine. The integral and the integration error are estimated for a given integrand and a given hyper-rectangular domain. A requested relative or absolute error, or a maximum number of integrand evaluations has to be specified. The routine terminates when at least one of these three criteria is met. The returned `Status` reports which condition terminated the routine.

```
real operator() (Integrand &, const Hypercube &,
                 Index maxEval,
                 real reqRelError, real reqAbsError);
```

This convenience function calls `integrate()`, discards `Status` and the estimated error, and returns the estimate for the integral directly.

```
};
```

Chapter 3

Integration Routines

Chapter 4

Pseudo Random Number Generators

All `Integrator`s based on Monte Carlo or randomized QMC techniques require some source of random numbers, which is provided by the pseudo random number generators (PRNGs) described in this chapter. If a Monte Carlo routine does not produce the expected result, there is a good chance that the PRNG in use has some deficiency which surfaces in this specific calculation. Therefore, a number of PRNGs are provided to allow the user to choose.

4.1 General Concepts

This section describes the design issues considered during the implementation of the PRNG interface. You will probably need this information if you design your own PRNG or implement new Monte Carlo algorithms using one of the provided PRNGs. If you only want to use one of the existing PRNGs in combination with one of the existing `Integrator`s, you can skip this section and continue with the description of the various available PRNGs. You can also go directly to Section 4.4 which describes the Mersenne Twister, one of the best PRNGs available today.

4.1.1 Interface of PRNGs

A PRNG in the sense of HINTLIB is some class that provides the following interface:

```
class PRNGName
{
public:
   PRNGName (unsigned start = someDefault);
   void init (unsigned seed);

   typedef someType ReturnType;

   ReturnType  getMax() const;
   const real& getResolution() const;
   const real& getRange() const;

   ReturnType operator() ();
   int operator() (int max);
   real getReal();

   size_t getStateSize () const;
   void saveState (void *) const;
   void restoreState (const void *);
};
```

The constructor takes an `unsigned` with is used by `init()` to seed the state of the PRNG. Initializing two PRNGs of identical type with equal start values results in identical generated sequences. Implementations should try to guarantee that the sequences generated by different start values are non-overlapping for a long time. However, if this can actually be achieved by a PRNG depends on the implementation.

`operator()()` is used to get the next value from the generator. The return type of this function depends on the PRNG. Usually, it is `long`, `u32`, or `u64`. This type is available as `ReturnType`. The number returned by `operator()` is always larger or equal to 0 and less or equal to $m =$ `getMax()`. For a manual conversion to a real number, it is often useful to have $m+1$ and $1/(m+1)$ available, with can easily be obtained by `getRange()` and `getResolution()`, respectively.

`operator()(int` max`)` returns a random integer from $\{0, 1, 2, \ldots, max - 1\}$. Syntax and semantics of this method are exactly what the C++ Standard Template Library expects as a Random Number Generator object.

`getReal()` returns a random sample from the uniform distribution on $(0, 1)$. Implementations have to ensure that 0 and 1 are never returned.

`saveState()` and `restoreState()` allow to save and restore the state of the PRNG to/from some buffer in memory. `getStateSize()` gives the required size of this buffer in `char`s.

4.2 Linear Congruential Generators

Linear Congruential Generators (LCGs) produce pseudo random numbers by the simple recurrence

$$x_{n+1} = ax_n + c \bmod m.$$

Parameters a, c, and m have to be chosen obeying a number of constraints in order to make the sequence appear to be random. In addition to these theoretical constraints, the multiplication modulo m can be implemented efficiently only for certain values of m and a.

Detailed information on the construction of powerful LCGs and a detailed discussion of the proper choice of m, a, and c can be found in [Knu98, Section 3.2.1].

4.2.1 LCGs with $m = 2^e$

LCGs with $m = 2^e$ can be implemented very efficiently on binary machines. However, the modulus leads to a number of caveats in the quality of the PRNG, most noteworthy that the lower order bits show a periodic pattern.

The choice of $c = 0$ leads to a slightly faster generator because the incrementation of x_n can be skipped. However, this marginal performance improvement is bought by restricting the period length to one fourth ($m/4$ instead of m) and invalidating half of the possible seed values (all x_n, and therefore x_0, have to be odd). An arbitrary odd value for c does not impose these restrictions. Therefore, the only two reasonable values for c are $c = 0$ (if efficiency is a premium) and $c = 1$, which allows a faster implementation than any other odd c and does not introduce any restrictions on the period length. The maximum period as described above is accomplished if and only if $a \bmod 4 = 1$ for c odd and $a \bmod 8 \in \{3, 5\}$ for $c = 0$ and $e > 3$.

Implementation

A special implementation making use of the optimizations possible for power-of-two modulus is provided as template `LCG_Pow2<>`:

```
template<typename T, T a, unsigned e, T c = T(1)>  // from lcg_pow2.h
class LCG_Pow2
{
```

```
public:
   LCG_Pow2 (unsigned start = 0, bool force = false);

   static const T A = a;
   static const T C = c;
   typedef T ReturnType;

   // other members common to all PRNGs
};
```

The first template parameter `typename T` must be an unsigned integral type, large enough to store e bits. The parameters a, e, and c of the LCG are provided as template parameters, allowing complete inlining and optimal code generation for all time-critical operations.

The constructor takes an optional *start* argument, specifying the initial value of the generator. x_0 is set to *start* if c is odd, otherwise to $2 \cdot start + 1$, ensuring that the maximal possible period is achieved. The boolean argument *force* determines, if the constructor should perform a consistency check on the values a, c, m as described above. Since all three values are available at compile time, this check can usually be optimized away by the compiler and does not introduce any runtime overhead. Setting $force = true$ allows the construction of a degenerated LCGs with a period length less than m and $m/4$ for c odd and $c = 0$, respectively.

`init()` can be used to reinitialize the generator.

`operator()()` returns the current value x_n from $\{0, 1, \ldots, m-1\}$ if c is odd. If $c = 0$, then $x_n \in \{1, 5, 9, \ldots, m-3\}$ or $x_n \in \{3, 7, 11, \ldots, m-1\}$, depending on the set x_0 belongs to. `operator()(`n`)` returns $\lfloor nx_n/m \rfloor \in \{0, 1, \ldots, n-1\}$. `getReal()` returns $(x_n + 1/2)/m$ for c odd, and x_n/m for c even. Both results are strictly between 0 and 1.

Predefined LCGs of this type

A number of LCGs of this type are provided as `typedef`s. All these generators, including the comments provided here are taken from the spectral test result list in [Knu98, Section 3.3.4]:

```
LCG_Pow2<u32,1103515245,32,12345> LCG_Pow2_Ansi_C;
```

The generator proposed in the ANSI C reference.

```
typedef LCG_Pow2<u64,1220703125,35,0> LCG_Pow2_Taussky_0;
typedef LCG_Pow2<u64,1220703125,35,1> LCG_Pow2_Taussky;
```

A reminder of the good old days when 35 bits was a common word size. It performs quite well in the spectral test, but requires 64 bit architecture. It is due to O. Taussky an is listed as line 11 of Knuth's spectral test result list.

```
typedef LCG_Pow2<u32,65539u,31,0> LCG_Pow2_RANDU_0
```

This is the infamous RANDU, one of the wort generators ever conceived. It has exceptionally bad values in the spectral test in dimensions 3, 4, 5, and 6 and therefore should have never be used. $a \bmod 4 = 3$, therefore this generator can only be used with $c = 0$, making it even worse. Line 12 in [Knu98].

```
typedef LCG_Pow2<u32,1812433253,32,0> LCG_Pow2_BoroshNiederreiter_0;
typedef LCG_Pow2<u32,1812433253,32,1> LCG_Pow2_BoroshNiederreiter;
```

Borosh–Niederreiter multiplier for $m = 2^{32}$. Line 13.

```
typedef LCG_Pow2<u32,1566083941,32,0> LCG_Pow2_Waterman_0;
typedef LCG_Pow2<u32,1566083941,32,1> LCG_Pow2_Waterman;
```

Due to A. Waterman. Line 14.

```
typedef LCG_Pow2<u32,69069,32,0> LCG_Pow2_69069_0;
typedef LCG_Pow2<u32,69069,32,1> LCG_Pow2_69069;
```

Here is another famous one, probably due to its nice looking a. However, it also performs well in the spectral test. Line 15.

```
typedef LCG_Pow2<u32,1664525,32,0> LCG_Pow2_LavauxJanssens32_0;
typedef LCG_Pow2<u32,1664525,32,1> LCG_Pow2_LavauxJanssens32;
typedef LCG_Pow2<u64,31167285ull,48,0> LCG_Pow2_LavauxJanssens48_0;
typedef LCG_Pow2<u64,31167285ull,48,1> LCG_Pow2_LavauxJanssens48;
```

These two multipliers have been found by M. Lavaux and F. Janssens in a computer search for spectrally good multipliers having a very high μ_2. Lines 16 and 23.

```
typedef LCG_Pow2<u64,44485709377909ull,48,0> LCG_Pow2_Cray_0;
typedef LCG_Pow2<u64,44485709377909ull,48,1> LCG_Pow2_Cray;
```

The one with $c = 0$ is used in the Cray X-MP library. Line 22.

```
typedef LCG_Pow2<u64,6364136223846793005ull,64,0> LCG_Pow2_Haynes_0;
typedef LCG_Pow2<u64,6364136223846793005ull,64,1> LCG_Pow2_Haynes;
```

Finally, an excellent one for 64 bit numbers due to C. E. Haynes. Line 26.

4.2.2 LCGs with m prime and special multiplier

LCGs with a prime number modulus m have advantageous theoretical properties. If the multiplier a is chosen to be a primitive element modulo m, a period length of $m - 1$ (cycling through all values except 0) is obtained even for $c = 0$. The low-order bits are as random as the most significant bit.

The drawback is that a prime-number LCG with m close to the word size requires double-word and therefore slow arithmetic to calculate $x_{n+1} = ax_n \bmod m$. However, for m fitting in a signed integer variable and certain well-chosen a, there is an algorithm that allows the calculation of x_{n+1} from x_n using only ordinary integer arithmetic. This algorithm requires two constants $q = \lfloor m/a \rfloor$ and $r = m \bmod a$. It works if and only if r turns out to be less than q, which is less likely for increasing a.

Implementation

```
template<typename T, T a, T m>     //  from lcg_prim.h
class LCG_Prime
{
public:
   LCG_Prime (unsigned start = 0);

   typedef T ReturnType;

   static const T    A = a;
   static const T    M = m;
   static const T    C = 0;

   typedef T ReturnType;
   // other members common to all PRNGs
};
```

The first template parameter `typename T` must be a signed integral type, large enough to store m. The parameters a and m of the LCG are provided as template parameters, allowing complete inlining of all time-critical operations.

The constructor takes an optional *start* argument, specifying the initial value of the generator to $x_0 = start + 1$. `init()` can be used to reinitialize the generator. The constructor contains code for doing some consistency checks on `T`, a and m. These tests can be done at compile time, so no additional run time overhead is introduced.

`getMax()` returns the value x_n from $\{1, 2, \ldots, m-1\}$. `operator(`n`)` return $\lfloor nx_n/m \rfloor \in \{0, 1, \ldots, n-1\}$. `getReal()` returns x_n/m which is strictly between 0 and 1.

Predefined LCGs of this type

A number of LCGs of this type are provided as `typedef`s. All these generators, including the comments provided here, are taken from the Spectral Test result list in [Knu98, Section 3.3.4]:

```
typedef LCG_Prime<long, 7*7*7*7*7, (1ul << 31) - 1> LCG_Prime_IMSL;
```

This generator is know is the "Minimum Standard Generator" and has been one of the main generators in the popular IMSL subroutine library since 1971. The multiplier $a = 7^5 = 16807$ is due to Lewis, Goodman, and Millers and the main reason for its continued use that a^2 is less than the modulus m, hence fast algorithms for calculating $ax \bmod m$ have been available for quite some time. However, the following multipliers perform better in the spectral test and allow for the same efficient implementation. Line 19 in [Knu98].

```
typedef LCG_Prime<long, 48271, (1ul << 31) - 1> LCG_Prime_Fishman;
```

The best multipliers for $m = 2^{31} - 1$ (a Mersenne Prime) allowing this implementation technique. Due to G. S. Fishman. Line 20.

```
typedef LCG_Prime<long, 40692, (1ul << 31) - 249> LCG_Prime_Lecuyer;
```

Another good one, due to P. L'Ecuyer, for a slightly smaller prime modulus. Line 21.

4.3 Combined LCGs

Two LCGs x_n and y_n can be combined to a new generator by simply using

$$z_n := x_n - y_n \bmod m$$

where m is the larger one of the moduli of x_n and y_n (see [Knu98, Section 3.2.2]).

Implementation

```
template<typename T, T a1, T m1, T a2, T m2>
class LCG_Combined
{
public:
   LCG_Combined (unsigned start = 0);
   typedef T ReturnType;
   // other members common to all PRNGs
};
```

The first template parameter `typename T` must be a signed integral type, large enough to store m_1 and m_2. The parameters a_1, a_2, m_1 and m_2 define the LCGs to be used.

Predefined LCGs of this type

A `typedef` exists for the following combined LCG: It is taken from the spectral test result list in [Knu98, Section 3.3.4]:

```
typedef LCG_Combined<long,
   48271, (1ul << 31) - 1,
   40692, (1ul << 31) - 249> LCG_Combined_Lecuyer;
```

This generator is based on two prime-modulus LCGs discussed in Section 4.2.2 and has a period length of $(2^{31} - 1)(2^{31} - 249)$. The spectral test results can be found in line 24 of Knuth's Spectral Test result list in [Knu98, Section 3.3.4]

4.4 Mersenne Twister

The Mersenne Twister PRNG was proposed by M. Matsumoto and T. Nishimura in [MN98]. It has a period of $2^{19937} - 1$ (a Mersenne Prime), produces a sequence that is 623-dimensionally equidistributed, has passed many stringent tests, including the *die-hard test* of G. Marsaglia and the *load-test* of P. Hellekalek and S. Wegenkittl, and has no known weaknesses. Therefore, it is considered to be one of the best PRNGs available today.

A number of open source implementations are publicly available, most noteworthy the original implementation of Takuji Nishimura in C, an optimized C version due to S. Cokus, and a Mersenne Twister source-forge project maintained by R. J. Wagner. The implementation contained in this library is compatible to each of them, including initialization. At least on my computer, it is also faster than any of the other implementations.

```
class MersenneTwister    // from mersennetwister.h
{
public:
   MersenneTwister (u32 start = 4357u);
   void init (unsigned = 4357u);
   void initCokus (unsigned start = 4357u);

   typedef u32 ReturnType;

   // other members common to all PRNGs
};
```

Three different initialization routines are available: `init()` is compatible with the original code of Nishimura, as well as Wagner's `MTRand` class. It used the LCG $x_0 = start$, $x_{n+1} = 69069x_n + 1$ to seed the state of the generator. Each 32-bit integer in `MersenneTwisters`'s state array is seeded by combining the most significant 16 bits of two values of this LCG.

`initCokus()` is compatible with Cokus' implementation. It uses the LCG $x_0 = start|1$, $x_{n+1} = 69069x_n$ to seed the state array. Therefore, $start$-values $2n$ and $2n + 1$ result in the same initialization. Use this initialization routine only for compatibility reasons.

The constructor uses `init()` to initialize the state array. Non of these initializers guarantees that the sequences resulting form two different start values are non-overlapping. If you know a method for seeding `MersenneTwister` in away that allows the construction of n different non-overlapping sequence with a length of at least m, please let me know!

4.5 The Built-In Random Number Generator

Every C and C++ implementation provides a built-in PRNG available through the C library functions `rand()` and `srand()`. The following class from `builtinprng.h` provides an interface to this facility that is compatible with HINTLIB.

```
class BuiltInPRNG      // from builtinprng.h
{
public:
   BuiltInPRNG (unsigned start = 0);
   ~BuiltInPRNG ();

   typedef int ReturnType;

   // other members common to all PRNGs
};
```

`operator()` is just a call to `rand()`, `seed()` a call to `srand()`. The implementation ensures that only one instance of `BuiltInPRNG` exists at the same time.

Chapter 5

Generator Matrices

A digital (t, m, s)-net over $\mathbb{F}_q$ is a (t, m, s)-net in base q defined by s $p \times m$-matrices $\boldsymbol{C}_0, \ldots, \boldsymbol{C}_{s-1}$ over $\mathbb{F}_q$. `GeneratorMatrix` and its subclasses provide all necessary facilities for handling these matrices in HINTLIB. The term "generator matrix" will always refer to the complete set of s matrices.

For $n = 0, \ldots, q^m - 1$ let

$$n = \sum_{r=0}^{m-1} a_k q^k$$

the q-adic representation of n in base q. Choose arbitrary bijections $\varphi : \{0, \ldots, q-1\} \to \mathbb{F}_q$ and $\psi : \mathbb{F}_q \to \{0, \ldots, q-1\}$. Let

$$\left(y_0^{(i)}(n), \ldots, y_{p-1}^{(i)}(n)\right)^T := \boldsymbol{C}_i \cdot (\varphi(a_0), \ldots, \varphi(a_{m-1}))^T$$

for $d = 0, \ldots, s-1$ and

$$\boldsymbol{x}_n := (x_n^{(0)}, \ldots, x_n^{(s-1)}) \qquad \text{with} \qquad x_n^{(i)} := \sum_{b=0}^{p-1} \frac{\psi(y_b^{(i)}(n))}{q^{b+1}}.$$

The the point set $\{\boldsymbol{x}_n \mid n = 0, \ldots, q^m - 1\}$ is the (t, m, s)-net in base b defined by the matrices $\boldsymbol{C}_0, \ldots, \boldsymbol{C}_{s-1}$.

5.1 Geometry of a Generator Matrix

There are four parameters determining the size and geometry of a generator matrix:

- The size of the base field $\mathbb{F}_q$, referred to as *base*.
- The number of matrices s, referred to as the *dimension* of the generator matrix.
- The number of columns in each matrix. This number, always denoted by m, determines the size of the point set: the resulting (t, m, s)-net has exactly q^m points.
- The number of rows in each matrix, denoted by *precision*. The precision determines how many digits of the coordinates of the resulting point set are controlled by the generator matrix.

  In the mathematical literature this value is always equal to m, so matrices considered are always $m \times m$ square matrices. This simplification is justifiable because a precision smaller than m can be extended to m by padding the matrices with 0s, whereas a precision larger than m can never affect the t-parameter.

  For a computer implementation, however, allowing smaller precisions is an advantage in memory usage as well as in execution time for some algorithms.

5.1.1 Notation

Throughout this section, the various parts of a generator matrix are referred to using the following notation:

- $\boldsymbol{C}_d$ denotes the one of the s $p \times m$-matrices for $0 \le d < s$.
- The element in row b and column r of $\boldsymbol{C}_d$ is denoted by $c_{b,r}^{(d)}$ for $0 \le b < p$ and $0 \le r < m$.
- The bth row of $\boldsymbol{C}_d$ is denoted by $\boldsymbol{c}_{b,*}^{(d)}$ for $0 \le b < p$.
- The rth column of $\boldsymbol{C}_d$ is denoted by $\boldsymbol{c}_{*,r}^{(d)}$ for $0 \le r < m$.

5.2 `GeneratorMatrix` – The Common Base Class

All representations of generator matrices in HINTLIB share the common abstract base class `GeneratorMatrix`.

```
class GeneratorMatrix
{
public:
   unsigned getBase() const;
   unsigned getDimension() const;
   unsigned getM() const;
   unsigned getPrec() const;
```

These four methods return information about the geometry of the generator matrix. The exact meaning of these four values is described in Section 5.1.

```
   virtual void     setDigit(unsigned d, unsigned r,
                             unsigned b, unsigned x);
   virtual unsigned getDigit(unsigned d, unsigned r,
                             unsigned b) const = 0;
```

`setDigit(`d`,`r`,`b`,`x`)` sets $c_{b,r}^{(d)}$ to x. `getDigit(`d`,`r`,`b`)` returns the current value of $c_{b,r}^{(d)}$.

```
   virtual u64  vGetPackedRowVector (
         unsigned d, unsigned b) const = 0;
   virtual void vSetPackedRowVector (
         unsigned d, unsigned b, u64 x);
```

Sets or returns the the row vector $\boldsymbol{c}_{b,*}^{(d)}$. The data of the vector is packed into a 64 bit integer using the formula

$$\sum_{r=0}^{m-1} c_{b,r}^{(d)} q^r.$$

```
   void print (std::ostream &) const;
   void printDimension    (std::ostream &, unsigned d) const;
   void printRowVector    (std::ostream &, unsigned d, unsigned b) const;
   void printColumnVector (std::ostream &, unsigned d, unsigned r) const;
```

These four methods can be used for printing (parts of) a generator matrix in a formated form to an output stream. `print()` prints the whole generator matrix, the other three methods prints $\boldsymbol{C}_d$, $\boldsymbol{c}_{b,*}^{(d)}$, and $\boldsymbol{c}_{*,r}^{(d)}$, respectively.

```
   void libSeqExport (std::ostream &) const;
   void binaryExport (std::ostream &) const;
```

These methods export the generator matrix in special formats.

```
};

bool operator== (const GeneratorMatrix &,
                 const GeneratorMatrix &);

void assign (const GeneratorMatrix &, unsigned,
                   GeneratorMatrix &, unsigned);
void assign (const GeneratorMatrix &,
                   GeneratorMatrix &);
```

`operator==(`G_1`,`G_2`)` compares two `GeneratorMatrix`es G_1 and G_2 and returns true if they are equal. They have to have the same geometry and the same entries.

`assign(`G_1`,`d_1`,`G_2`,`d_2`)` assigns $\boldsymbol{C}_{d_1}$ of G_1 to $\boldsymbol{C}_{d_2}$ of G_2. If G_1 has higher m or precision, $\boldsymbol{C}_{d_1}$ is truncated. If it has lower m or lower precision, an exception is thrown.

`assign(`G_1`,`G_2`)` assigns G_1 to G_2. Extra entries in the source matrix are discarded, missing entries result in an exception.

5.2.1 Default Geometry for Generator Matrices

Many subclasses of `GeneratorMatrix` allow the construction of matrices with user-defined sizes. The base and the dimension has to be specified explicitly, if it is not implied by some other parameters of the type of the object. For the other two parameters, m and p, most classes provide the following default values:

- If neither m nor p are given, the following default values are used: m is set to the largest value such that q^m is a) representable in an `Index` and b) less than 2^{48}. p is set to the smallest value such that q^{-p} is less or equal `std::numeric_limits<real>::epsilon()`.

  Since m is rather large, constructors of this type are best used for creating matrices of (t, s)-sequences with enough precision to control the full accuracy of a `real`.

- If only one value is given, this value is used for m as well as p. However, p is never larger than the default value for p described above.

  Constructors of this type are best used for (t, m, s)-nets with a certain, fixed m. The resulting generator matrix contains square matrices, which matches the usual mathematical concept.

- If both values are specified by the user, the given values are used.

5.3 Data Layout for Generator Matrices

At the moment HIntLib includes five subclasses of `GeneratorMatrix`. Each of them uses a different data layout, which is tailor-made for certain applications. Since each of these types has a copy constructor accepting a `GeneratorMatrix` argument, matrices of different type can easily be converted into each other.

The following five types are available:

1. `GeneratorMatrixGen<>`, optimized for fast point set generation
2. `GeneratorMatrixGenVec<>`, optimized for fast point set generation with additional vectorization
3. `GeneratorMatrixGenRow<>`, optimized for matrix operations dealing primarily with row vectors

4. `GeneratorMatrix2<>`, for $q = 2$ and optimized for point set generation

5. `GeneratorMatrix2Row<>`, for $q = 2$ and optimized for matrix operations dealing primarily with row vectors

5.3.1 GeneratorMatrixGen<>

```
template<typename T>
class GeneratorMatrixGen : public GeneratorMatrix
{
public:
   GeneratorMatrixGen (unsigned q, unsigned s);
   GeneratorMatrixGen (unsigned q, unsigned s, unsigned m);
   GeneratorMatrixGen
      (unsigned q, unsigned s, unsigned m, unsigned prec);

   GeneratorMatrixGen (const GeneratorMatrixGen<T> &);
   GeneratorMatrixGen (const GeneratorMatrix &);

   const T* getMatrix() const;
   const T* operator() (unsigned r) const;

   const T* operator() (unsigned d, unsigned r) const;
         T* operator() (unsigned d, unsigned r);
   void makeZeroColumnVector (unsigned d, unsigned r);
   u64  getPackedRowVector (unsigned d, unsigned b) const;
   void setPackedRowVector (unsigned d, unsigned b, u64 x);
   void makeZeroRowVector (unsigned d, unsigned b);
   T operator() (unsigned d, unsigned r, unsigned b) const;
   void setd (unsigned d, unsigned r, unsigned b, T x);
   void makeEquidistributedCoordinate (unsigned d);
   void makeIdentityMatrix (unsigned d);
   void makeZeroMatrix ();
   void makeZeroMatrix (unsigned d);
   void makeShiftNet (unsigned b);
   void makeShiftNet ();
};

GeneratorMatrixGen<unsigned char>*
loadLibSeq (std::istream &);
GeneratorMatrixGen<unsigned char>*
loadEdel   (std::istream &, unsigned);
GeneratorMatrixGen<unsigned char>*
loadBinary (std::istream &);
GeneratorMatrixGen<unsigned char>*
loadNiederreiterXing (unsigned dim);
```

5.3.2 GeneratorMatrixGenVec<>

5.3.3 GeneratorMatrixGenRow<>

`GeneratorMatrixGenRow<>` provides the most convenient data layout for implementing various algorithms for manipulating generator matrices. The matrices are laid out in memory in the

format

$$\boldsymbol{c}_{0,*}^{(0)}, \ldots, \boldsymbol{c}_{p,*}^{(0)}, \quad \ldots \quad , \boldsymbol{c}_{0,*}^{(s-1)}, \ldots, \boldsymbol{c}_{p-1,*}^{(s-1)},$$

i. e., the generator matrix is split up in its s matrices, which are again split up in its row vectors. Thus, each of the s matrices $\boldsymbol{C}_0, \ldots, \boldsymbol{C}_{s-1}$ is stored in a format that makes direct application of the linear algebra routines presented in Chapter 9 possible.

```
template<typename T>
class GeneratorMatrixGenRow : public GeneratorMatrix
{
public:
   LinearAlgebra& la() const;
```

Every `GeneratorMatrixGenRow<>` automatically creates its own instance of a `LinearAlgebra`. Therefore, all linear algebra routines (see Chapter 9) are readily available using the reference returned by `la()`. Creating and destroying the `LinearAlgebra` is taken care of by `GeneratorMatrix<>`.

```
   GeneratorMatrixGenRow (unsigned q, unsigned s);
   GeneratorMatrixGenRow (unsigned q, unsigned s, unsigned m);
   GeneratorMatrixGenRow
      (unsigned q, unsigned s, unsigned m, unsigned prec);
```

These constructors create a new generator matrix with dimension s over $\mathbb{F}_q$. The default values for m and precision are described in Section 5.2.1.

```
   GeneratorMatrixGenRow (const GeneratorMatrixGenRow<T> &);
   GeneratorMatrixGenRow (const GeneratorMatrix &);
```

Copy constructor for initializing a new `GeneratorMatrixRow<>` with data taken from another generator matrix.

```
   const T* getMatrix() const;
         T* getMatrix();
   const T* operator() (unsigned d) const;
         T* operator() (unsigned d);
```

These methods return pointers to certain parts of the generator matrix. `getMatrix()` returns a pointer to the complete data array; it can be used to access the generator matrix as one $sp \times m$-matrix. `operator()`(d) returns a pointer to the $p \times m$-matrix $\boldsymbol{C}_d$.

Both methods are available in a `const` and a `non-const` version. So the data can be modified in place

```
   const T* operator() (unsigned d, unsigned b) const;
         T* operator() (unsigned d, unsigned b);
   u64  getPackedRowVector (unsigned d, unsigned b) const;
   void setPackedRowVector (unsigned d, unsigned b, u64 x);
```

A number of methods are available for accessing row vectors. `operator()`(d,b) (available in a const and a non-const version) allows direct access to the data area used for storing $\boldsymbol{c}_{b,*}^{(d)}$. So each row vector can be used directly as an input vector as well as output vector for one of the linear algebra routines.

`getPackedRowVector`(d,b) and `setPackedRowVector`(d,b) are non-virtual counterparts to `vGetPackedRowVector()` and `vSetPackedRowVector()` available for all `GeneratorMatrix`es.

```
   T operator() (unsigned d, unsigned r, unsigned b) const;
   void setd (unsigned d, unsigned r, unsigned b, T x);
```

`operator(`d`,`r`,`b`)` returns $c_{b,r}^{(d)}$, `setd(`d`,`r`,`b`,`x`)` sets $c_{b,r}^{(d)}$ to x. These are the non-virtual counterparts to `getDigit()` and `setDigit()` available for all `GeneratorMatrix`es.

```
void makeZeroMatrix ();
void makeZeroMatrix (unsigned d);
void makeZeroRowVector (unsigned d, unsigned b);
void makeZeroColumnVector (unsigned d, unsigned r);
```

These four methods set all matrices, $\boldsymbol{C}_d$, $\boldsymbol{c}_{b,*}^{(d)}$, or $\boldsymbol{c}_{*,r}^{(d)}$ to zero.

```
void makeIdentityMatrix (unsigned d);
void makeEquidistributedCoordinate (unsigned d);
```

These methods set $\boldsymbol{C}_d$ to the $m \times m$-identity matrix or to a mirrored copy of this matrix, respectively. If $p > m$, rows $m, \ldots, p$ are filled with zeros; if $p < m$, the last $m - p$ rows of the identity matrix are discarded.

```
void makeShiftNet (unsigned b);
void makeShiftNet ();
```

`makeShiftNet(`b`)` takes $\boldsymbol{c}_{b,*}^{(0)}$ and sets $\boldsymbol{c}_{b,*}^{(1)}, \ldots, \boldsymbol{c}_{b,*}^{(1)}$ to cyclic shifts of this vector. `makeShiftNet()` performs the same operation for all rows b. Don't use if $s > m$!!!

```
};
```

5.3.4 GeneratorMatrix2<>

5.3.5 GeneratorMatrix2Row<>

5.4 Virtual `GeneratorMatrix`es

The `GeneratorMatrix` implementations discussed in the previous section use arrays of some kind for storing the matrix elements. HINTLIB contains a number of additional `GeneratorMatrix`es that do not rely on tables but implement the required interface by other means.

On the one hand there are `GenerorMatrix`es providing certain matrices that are so simple that its entries can be calculated easily for every call to `getDigit()`.

On the other hand, there are `GeneratorMatrix`es transforming a given generator matrix in some way. In many cases, these implementations simply store a pointer to the original matrix and implement `getDigit()` by appropriate calls to `getDigit()` on the source matrix.

5.4.1 ZeroMatrices

```
class ZeroMatrices : public GeneratorMatrix
{
public:
   ZeroMatrices (unsigned base, unsigned dim);
   ZeroMatrices (unsigned base, unsigned dim, unsigned m);
   ZeroMatrices (unsigned base, unsigned dim, unsigned m,
                 unsigned prec);
};
```

`ZeroMatrices` is a generator matrix which contains only 0s. No memory is allocated for the matrices, so this object can be constructed for almost arbitrary sizes.

`ZeroMatrices` supports all three types of constructors discussed in Section 5.2.1.

5.4.2 IdentityMatrices

```
class IdentityMatrices : public GeneratorMatrix
{
public:
   IdentityMatrices (unsigned base, unsigned dim);
   IdentityMatrices (unsigned base, unsigned dim, unsigned m);
   IdentityMatrices (unsigned base, unsigned dim, unsigned m,
                     unsigned prec);
};
```

`IdentityMatrices` is a generator matrix where C_d is the identity matrix for all $0 \leq d < s$. In other words,

$$c_{b,r}^{(d)} = \begin{cases} 1 & b = r \\ 0 & \text{otherwise} \end{cases}.$$

No memory is allocated for the matrices, so this object can be constructed for almost arbitrary sizes.

`IdentityMatrices` supports all three types of constructors discussed in Section 5.2.1.

5.4.3 AdjustPrec

```
class AdjustPrec : public GeneratorMatrix
{
public:
   AdjustPrec (int p, const GeneratorMatrix&);
};
```

`AdjustPrec` is a generator matrix adjusting the precision of another matrix to a given value. If the original matrix has a higher precision, trailing rows are discarded. Otherwise, new 0-filled rows are appended after the original rows.

`AdjustPrec` makes no deep copy of the original matrix. Therefore, the original matrix must not be destructed as long as the depending `AdjustPrec` is used.

5.4.4 AddLastRow

```
class AddLastRow : public GeneratorMatrix
{
public:
   AddLastRow (const GeneratorMatrix&);
};
```

`AddLastRow` is a generator matrix appending an additional row to a given generator matrix in the following way: if the original generator matrix has matrices $\boldsymbol{C}_i$ for $i = 0, \ldots, s-1$, the new generator matrix has matrices

$$\begin{pmatrix} \boldsymbol{C}_i \\ \boldsymbol{c}_{0,*}^{(i+1) \bmod s} \end{pmatrix}$$

for $i = 0, \ldots, s-1$. In other words, the first row vector of the (cyclically) next matrix is used as the last row vector for each new matrix.

If the original generator matrix is an OOA with depth $k-1$ and strength k, the new generator matrix has depth k and strength k, and therefore a net with strength k.

`AddLastRow` makes no deep copy of the original matrix. Therefore, the original matrix must not be destructed as long as the depending `AddLastRow` is used.

5.4.5 AdjustM

```
class AdjustM : public GeneratorMatrix
{
public:
   AdjustM (int m, const GeneratorMatrix&);
};
```

`AdjustM` is a generator matrix adding or removing columns to/from another matrix such that the new matrix has exactly m columns. If the original matrix has more columns, only the first m are chosen. This results in a net consisting of the first q^m points of the original net. If the original matrix is a (t, s)-sequence with $t \leq m$, the resulting net is a (t, m, s)-net.

If the original matrix has $m' < m$ columns, $m - m'$ additional 0-filled columns are appended and the resulting points set consists of $q^{m-m'}$ copies of the original point set. This construction corresponds to the propagation rule "Net Duplication".

`AdjustM` makes no deep copy of the original matrix. Therefore, the original matrix must not be destructed as long as the depending `AdjustM` is used.

5.4.6 MReduction

```
class MReduction : public GeneratorMatrix
{
public:
   MReduction (unsigned m, unsigned d = 0, const GeneratorMatrix&);
};
```

Based on a given generator matrix with $m' \geq m$ columns, a new generator matrix with only m columns is created using the following procedure:

1. The first $m' - m$ columns are removed from each matrix. This results in taking only every $q^{m'-m}$-th point of the original net.
2. The first $m' - m$ rows are removed from matrix number d and the same number of 0-filled rows are appended at the bottom of the matrix. This results in $x_i^{(d)} \rightarrow (q^{m'-m} x_i^{(d)}) \bmod 1$ for all points $\boldsymbol{x}_i = (x_i^{(0)}, \ldots, x_i^{(s-1)})$ of the point set.

If the original matrix has strength k, the first k rows of its dth matrix form the identity matrix, and $k \geq m' - m$, then the t-parameter of the resulting net is at least as good as the one of the original net. The presence of the identity matrix can be guaranteed by using a `WithIdentityMatrix` as input matrix.

This construction corresponds to the propagation rule "m-Reduction".

`MReduction` makes no deep copy of the original matrix. Therefore, the original matrix must not be destructed as long as the depending `MReduction` is used.

5.4.7 NetFromSequence

```
class NetFromSequence : public GeneratorMatrix
{
public:
   NetFromSequence (unsigned m, bool e = true,
                    const GeneratorMatrix&);
};
```

Based on a generator matrix with m' columns, a new generator matrix with $m \leq m'$ columns is created using the following procedure:

1. Only the first m columns of each matrix are used
2. If e is true, an additional matrix containing a mirrored identity matrix is added.

If the original generator matrix is (part of) a (t, s)-sequence with $t \leq m$, the resulting net is a (t, m, s)-net (or $(t, m, s+1)$-net if e is true). This construction corresponds to the propagation rule "Net from Sequence".

`NetFromSequence` makes no deep copy of the original matrix. Therefore, the original matrix must not be destructed as long as the depending `NetFromSequence` is used.

5.4.8 DiscardDimensions

```
class DiscardDimensions : public GeneratorMatrix
{
public:
   DiscardDimensions (int s, const GeneratorMatrix&);
};
```

Based on a generator matrix with dimension s', a new generator matrix with dimension $s \leq s'$ is created by discarding all but the first s matrices. This construction corresponds to the propagation rule "s-Reduction".

`DiscardDimensions` makes no deep copy of the original matrix. Therefore, the original matrix must not be destructed as long as the depending `DiscardDimensions` is used.

5.4.9 SelectDimensions

```
class SelectDimensions : public GeneratorMatrix
{
public:
   SelectDimensions (unsigned s, const GeneratorMatrix&);
   SelectDimensions (unsigned d1, unsigned d1,
                     const GeneratorMatrix&);
   SelectDimensions (const unsigned* p1, const unsigned* p2,
                     const GeneratorMatrix&);

   void selectDimension (unsigned d_result, unsigned d_original);
};
```

`SelectDimensions` creates a new generator matrix by selecting certain matrices from another generator matrix.

The first constructor selects the first s matrices, i.e. $\boldsymbol{C}_0, \ldots, \boldsymbol{C}_{s-1}$. The second constructor selects the $d_2 - d_1$ matrices $\boldsymbol{C}_{d_1}, \ldots, \boldsymbol{C}_{d_2-1}$. The third constructor selects matrices $\boldsymbol{C}_{d_1}, \boldsymbol{C}_{d_2}, \ldots, \boldsymbol{C}_{d_s}$, where $d_1, \ldots, d_s$ are the integers `*p1`,...,`*(p2-1)`.

Even after the construction the selection of matrices can be changed. `selectDimension(`d_{result}`,` d_{original}`)` makes $\boldsymbol{C}_{d_{\text{original}}}$ appear as matrix d_{result} of the new generator matrix.

This construction can be used for implementing the propagation rule "s-Reduction". However, `DiscardDimensions` should be used if selecting the first s matrices is sufficient.

`SelectDimensions` makes no deep copy of the original matrix. Therefore, the original matrix must not be destructed as long as the depending `SelectDimensions` is used.

5.4.10 BaseReduction

```
class BaseReduction : public GeneratorMatrix
{
public:
```

```
   BaseReduction (const GeneratorMatrix&);
};
```

Based on a net in base p^k with p prime, a new net in base p is constructed yielding the same point set as the original net.

This construction corresponds to the propagation rule "Base Reduction".

The original matrix can be destructed after the construction of `BaseReduction` is complete.

5.4.11 WithIdentityMatrix

```
class WithIdentityMatrix : public GeneratorMatrix
{
public:
   WithIdentityMatrix (unsigned d = 0, const GeneratorMatrix&);
};
```

Based on a given generator matrix, a new generator matrix yielding the same point set is created. However, the order of the points is changed such that the first k rows of $\boldsymbol{C}_d$ (where k is the number of initial linearly independent row vectors of $\boldsymbol{C}_d$) form the identity matrix.

This is accomplished by the following procedure:

1. Copy the first k row vectors of $\boldsymbol{C}_d$ into a new matrix $\boldsymbol{M}$
2. Supplement $\boldsymbol{M}$ to a regular square matrix
3. Find $\boldsymbol{M}^{-1}$, the inverse of $\boldsymbol{M}$
4. The new generator matrix is given by the matrices $\boldsymbol{C}_i\boldsymbol{M}^{-1}$ for $i = 0, \ldots, s-1$

Among other things, a `WithIdentityMatrix` can serve as an appropriate argument for `MReduction` in order to perform the propagation rule "m-Reduction".

The original matrix can be destructed after the construction of `WithIdentityMatrix` is complete.

5.5 Other Functions for Generator Matrices

```
void withIdentityMatrix  (GeneratorMatrixGenRow<unsigned char>&,
                          unsigned d = 0);
void withIdentityMatrix2 (GeneratorMatrixGenRow<unsigned char>&,
                          unsigned d = 0);
```

Multiplies all matrices with a regular matrix such that $\boldsymbol{C}_d$ becomes the identity matrix. See Section 5.4.11 for details.

The second version is slightly faster, but requires that $\boldsymbol{C}_d$ is a regular square matrix.

```
void makeRegular (
   GeneratorMatrixGenRow<unsigned char>&, unsigned d);
void fixOneDimensionalProjections (
   GeneratorMatrixGenRow<unsigned char>&);
```

`makeRegular()` converts $\boldsymbol{C}_d$ into a regular matrix. All initial linearly independent row vectors are kept. The remaining rows are replaced using a basis-supplement algorithm. Precision has to be at least as large as m.

`fixOneDimensionalProjections()` applies `makeRegular()` to all $\boldsymbol{C}_d$ for $d = 0, \ldots, s-1$.

```
int tParameter (const GeneratorMatrix&);
```

Determines the t-parameter of a given generator matrix.

```
bool confirmT (const GeneratorMatrix& gm, int t);
```

Determines whether the t-parameter of a given matrix is less or equal to t.

Chapter 6

Low Discrepancy Sequences

Chapter 7

Interpolatory Cubature Rules

Cubature rules are the basic building block of most adaptive integration routines. These algorithms use a certain basic rule

$$\mathrm{Q}_n f := \sum_{i=1}^{n} w_i f(\boldsymbol{x}_i)$$

with a fixed number n of abscissas which is applied to subregions of the initial integration domain. The final result is obtained by combining the transformed rules for each subregion to a copy rule for the whole integration domain.

A good cubature rule is crucial for the performance of every adaptive algorithm. Primarily, the cubature rule must provide two methods for the adaptive integration routine:

- The cubature rule must give good approximations $\mathrm{Q}_n f$ of $\mathrm{I} f$, while requiring a low number of integrand evaluations n.
- The cubature rules must support some method for estimating the integration error $|\mathrm{Q}_n f - \mathrm{I} f|$. The actual value of this estimation is not important. The required property is that a high estimated error is reported if and only if the actual integration error is large.

The second functionality (error estimation) is usually implemented based on the first one: two cubature rules $\mathrm{Q}^{(1)}_{n^{(1)}}$ and $\mathrm{Q}^{(2)}_{n^{(2)}}$ of different degree of accuracy are used. We assume that $\deg \mathrm{Q}^{(1)} > \deg \mathrm{Q}^{(2)}$, which will usually result in $n^{(2)}$ being significantly smaller than $n^{(1)}$. If these two rules are applied to the same integrand, the integration error $\left|\mathrm{Q}^{(1)} f - \mathrm{I} f\right|$ can be expected to be less than the difference $\mathrm{E} f := \left|\mathrm{Q}^{(1)} f - \mathrm{Q}^{(2)} f\right|$ of both estimations, i. e.

$$\left|\mathrm{Q}^{(1)} f - \mathrm{Q}^{(2)} f\right| \geq \left|\mathrm{Q}^{(1)} f - \mathrm{I} f\right| .$$

When $n^{(2)}$ is significantly smaller than $n^{(1)}$, this error estimation comes at little additional cost ($n^{(1)} + n^{(2)}$ instead of $n^{(1)}$ integrand evaluations). In some cases, the rule $\mathrm{Q}^{(2)}$ uses a subset of the abscissas of $\mathrm{Q}^{(1)}$. In this case, error estimation does not require any additional integrand evaluation, and $\mathrm{Q}^{(1)}$-$\mathrm{Q}^{(2)}$ is called an *Embedded Rule*.

One of the most complete collections of techniques, as well as theoretical results available for the construction of cubature rules is Stroud's monograph [Str71]. Even though this book is more than 30 years old, it has not been superseded by anything comparable.

7.1 Implementation

This section describes the objects that are used for implementing cubature rules.

7.1.1 Class `CubatureRule`

All implemented cubature rules are subclasses of a single base class `CubatureRule`. An object of this type is created for a given dimension s and can be used for estimating the integral of various integrands on various hyperrectangular regions with this dimension.

```
class CubatureRule
{
   CubatureRule();
   virtual ~CubatureRule();
```

The constructor of subclasses is primarily used for initializing all dimension-depending constants and to reserve extra memory that may be required for a fast evaluation of the rule. The destructor has to clean up this extra memory.

```
   virtual real eval (Integrand &, const Hypercube &) = 0;
```

This pure virtual function has to be implemented by subclasses of `CubatureRule` to provide the actual implementation of the cubature rule. The cubature rule is applied to a given integrand and a hyper-rectangular region.

```
   virtual unsigned getDimension()      const = 0;
   virtual Index    getNumPoints()      const = 0;
   virtual unsigned getDegree()         const = 0;
   virtual bool     isAllPointsInside() const = 0;
   virtual real     getSumAbsWeight()   const = 0;
};
```

These methods can be used by algorithms to query basic information about a rule. Knowing the number of abscissas of the rule (`getNumPoints()`) is particularly important, because it allows algorithms to decide how often they can apply the rule until the number of available integrand evaluations is exhausted.

`getDimension()` returns the dimension of the rule, `getDegree()` its polynomial degree. `isAllPointsInsied()` is `true` if and only if all abscissas are inside the `Hypercube` the rule is applied to. Finally, `getSumAbsWeight()` returns the sum of the absolute weights, i. e.

$$\sum_{i=1}^{n} |w_i| \,.$$

`CubatureRule`s cannot be assigned or copied.

7.1.2 Class `EmbeddedRule`

Class `EmbeddedRule` is a subclass of `CubatureRule` with the additional features of giving error estimations.

```
class EmbeddedRule : public CubatureRule
{
public:
   EmbeddedRule();

   virtual unsigned evalError
      (Integrand &, const Hypercube &, EstErr &ee) = 0;
```

This method is similar to `eval()` of `CubatureRule`. The difference is that, instead returning a `real`, an object of type `EstErr` (Section 2.8) is updated. The `unsigned` returned by `evalError()` suggests a split-direction for subsequent subdivision steps.

```
    virtual real eval (Integrand &, Hypercube &);
```

`EmbeddedRule` contains an implementation of the abstract method `CubatureRule::eval()`. It simply calls `evalError()`, discards the estimation of the error and returns the estimation for the integral as result.

```
};
```

7.1.3 Factories for Cubature Rules

A `CubatureRule` can only be applied to integration problems of a certain dimension which cannot be changed after the `CubatureRule` is created. Often one needs to specify a certain type of cubature rule, without determining the dimension. This can be accomplished by using a `CubatureRuleFactory`.

```
class CubatureRuleFactory
{
public:
   CubatureRuleFactory();
   virtual ~CubatureRuleFactory();
   virtual CubatureRule* create (unsigned) = 0;
   virtual CubatureRuleFactory* clone() const = 0;
};

class EmbeddedRuleFactory : public CubatureRuleFactory
{
public:
   EmbeddedRuleFactory();
   virtual ~EmbeddedRuleFactory();
   virtual EmbeddedRule* create (unsigned) = 0;
   virtual EmbeddedRuleFactory* clone() const = 0;
};
```

A `CubatureRuleFactory` creates `CubatureRule`s of a certain type. A call to `create()` returns a new `CubatureRule` for the specified dimension allocated on the free store. The user is responsible for `delete`ing it if it is not used anymore.

`CubatureRuleFactory`s cannot be copied or assigned. There is a `clone()` method which creates a copy, allocated on free store. In general, `CubatureRuleFactories` are always allocated on the free store.

`EmbeddedRuleFactory` is a subclass of `CubatureRuleFactory`, creating `EmbeddedRules`.

7.1.4 Pseudo Embedded Rules

For embedded rules, it is easy to implement `EmbeddedRule` directly. However, if a rule with error estimation is to be assembled from two unrelated rules, the following class can be used. Its constructor takes two `CubatureRuleFactories` in its constructor, and implements all other required methods by appropriately forwarding the calls.

```
class PseudoEmbeddedRule : public EmbeddedRule
{
public:
   PseudoEmbeddedRule (
      unsigned dimension,
      CubatureRuleFactory *fac1,
      CubatureRuleFactory *fac2);
```

```
    // All pure virtual functions inherited from EmbeddedRule
    // are implemented
};
```

The constructor of `PseudoEmbeddedRule` expects the dimension of the rule, together with two pointers to `CubatureRuleFactory`s, which are used for creating the `CubatureRules` the new `EmbeddedRule` is based upon.

```
class PseudoEmbeddedRuleFactory : public EmbeddedRuleFactory
{
public:
   PseudoEmbeddedRuleFactory (CubatureRuleFactory *fac1,
                              CubatureRuleFactory *fac2);

   virtual PseudoEmbeddedRuleFactory* clone() const;
   virtual PseudoEmbeddedRule* create (unsigned dim);
};
```

There is also an `EmbeddedRuleFactory` available which creates `PseudoEmbeddedRules`. Its constructor takes two pointers to `CubatureRuleFactories`. These factories are not `cloned` by the constructor, but `deleted` by the destructor of `PseudoEmbeddedRuleFactory`.

7.2 Implemented Cubature Rules

An abundant number of different cubature rules for multi-dimensional integration can be found in the literature. The most comprehensive collection of rules has been compiled by Stroud in [Str71], which covers most cubature rules known in 1971. This work was continued by Cools in [CR93], a collection of literature references published under the title *"Monomial Cubature Rules since* 'Stroud' *"*.

From all the cubature rules referenced in these collections, only those with the following properties have been implemented:

- Only rules for the hypercube C_s have been considered
- The article presenting the rule must contain all information for actually implementing the rule

Most cubature rules listed in [Str71] and [CR93] with the stated properties have been implemented. The following sections discuss these rules in detail.

The class name of the rules is always given in the section name. There is always a constructor of the form `classname(unsigned dim)`, and a `Cubature/EmbeddedRuleFactory` can be created using the static member function `classname::getFactory()`.

7.2.1 Midpoint Rule (`Rule1Midpoint`)

Degree 1, Fully Symmetric with $1 = O(1)$ point, equal-weight formula. For $s \geq 1$.

C_s:1-1 in [Str71].

Abscissas	w_i
$(0, \ldots, 0)$	V

This is the simplest possible integration formula. It consists of a single point in the center of C_s. The single weight w_1 is V, making this formula not only a positive-, but also an equal-weight formula. However, the degree is 1, so nothing but linear functions are integrated exactly.

7.2.2 Product Trapezoidal Rule (`Rule1Trapezoidal`)

Degree 1, Fully Symmetric with $2^s = O(2^s)$ points, equal-weight formula. For $s \geq 1$.
C_s:1-2 in [Str71].

Abscissas	w_i
$(\pm 1, \ldots, \pm 1)$	$\frac{1}{2^s} V$

This is the tensor product formula of the 1-dimensional trapezoidal rule

$$\int_{-1}^{1} f(x)\,dx \approx f(-1) + f(1).$$

7.2.3 "Simplex" Rule (`Rule2Simplex`) due to Stroud

Degree 2, Non-symmetric with $s + 1 = O(s)$ points, equal-weight formula. For $s \geq 1$.
[Str57] and C_s:2-1 in [Str71].

The integration nodes of this formula are the vertices of an s-dimensional regular simplex, lying on the surface of a sphere with radius $\sqrt{s/3}$. The tricky part is to rotate the simplex such that all vertices are inside the cube for any possible dimension s.

7.2.4 A Degree 2 Rule due to Thacher (`Rule2Thacher`)

Degree 2, Non-symmetric with $2s + 1 = O(s)$ points, $\frac{1}{V}\sum_{i=1}^{n} |w_i| = O(s)$. For $s \geq 1$.
[Tha64] and C_s:2-2 in [Str71].

Abscissas	w_i	
$(2r, \ldots, 2r)$	V	
$(1, r, \ldots, r)_{\mathrm{S}}$	$-rV$	with $r = \frac{\sqrt{3}}{6}$.
$(-1, r, \ldots, r)_{\mathrm{S}}$	rV	

7.2.5 A Degree 2 Rule due to Ionesu (`Rule2Ionescu`)

Degree 2, Non-symmetric with 4 integrand and 2 derivative evaluations. Only for $s = 2$.
[Ion62] and C_2:2-1 in [Str71].

Abscissas	w_i
$(-1, -1)$	$\frac{1}{4} V$
$(1, -1)$	$\frac{1}{12} V$
$(-1, 1)$	$\frac{1}{12} V$
$(1, 1)$	$\frac{7}{12} V$
$\frac{\partial}{\partial x}(1, 1)$	$-\frac{1}{3} V$
$\frac{\partial}{\partial y}(1, 1)$	$-\frac{1}{3} V$

In addition to integrand values (`Integrand::operator()()`), this rule uses derivatives of the integrand function to estimate the integral. Therefore, the integrand function has to define `Integrand::derivative()`.

7.2.6 "Octahedron" Rule due to Stroud (`Rule3Octahedron`)

Degree 3, Non-symmetric with $2s = O(s)$ points, equal-weight formula. For $s \geq 1$.
[Str57] and C_s:3-1 in [Str71].

The integration nodes of this formula are the vertices of an s-dimensional regular octahedron, with all its vertices lying on the surface of a sphere with radius $\sqrt{s/3}$, like the one used in `Rule3Cross` (Section 7.2.7). The tricky part is to rotate the octahedron such that it fits into the cube for any possible dimension s.

7.2.7 `Rule3Cross`

Degree 3, Fully Symmetric with $2s = O(s)$ points, equal-weight formula. For $s \geq 1$.
[Tyl53] and C_s:3-2 in [Str71]. Generalized in [Str57].

Abscissas	w_i
$(\sqrt{\frac{s}{3}}, 0, \ldots, 0)_{\mathrm{FS}}$	$\frac{1}{2s}V$

The relative position of the points is identical to `Rule3Octahedron` (Section 7.2.6). However, due to a different orientation, all abscissas are outside C_s for $s > 3$.

7.2.8 A Degree 3 Rule due to Tyler (`Rule3Tyler`)

Degree 3, Fully symmetric with $2s + 1 = O(s)$ points, $\frac{1}{V}\sum_{i=1}^{n} |w_i| = O(s)$. For $s \geq 1$.
[Tyl53] and C_s:2-3 in [Str71].

Abscissas	w_i	
$(0, \ldots, 0)$	W_1	
$(1, 0, \ldots, 0)_{\mathrm{FS}}$	W_2	with $W_1 = \frac{3-s}{3}V$ and $W_2 = \frac{1}{6}V$.

The weight W_1 is negative for $s > 3$. For $s = 3$ the weight W_1 becomes 0, thus, the rule has actually only 6 instead of 7 abscissas. However, this optimization is not implemented at the moment.

7.2.9 Product Gauss Rule of Degree 3 (`Rule3Gauss`)

Degree 3, Fully symmetric with $2^s = O(2^s)$ points, equal-weight formula. For $s \geq 1$.
C_s:2-4 in [Str71].

Abscissas	w_i	
$(\alpha, \ldots, \alpha)_{\mathrm{FS}}$	$\dfrac{1}{2^s}V$	with $\alpha = \frac{1}{\sqrt{3}}$.

This is the tensor product formula of the 1-dimensional 2-point degree 3 Gauss formula

$$\int_{-1}^{1} f(x)\,dx \approx f(-\frac{1}{\sqrt{3}}) + f(\frac{1}{\sqrt{3}}).$$

7.2.10 A Degree 3 Rule due to Ewing (`Rule3Ewing`)

Degree 3, Fully symmetric with $2^s + 1 = O(s^s)$ points, positive-weight formula. For $s \geq 1$.
[Ewi41] and C_s:3-5 in [Str71].

Abscissas	w_i
$(0,\dots,0)$	W_1
$(\pm1,\dots,\pm1)$	W_2

with $W_1 = \frac{2}{3}V$ and $W_2 = \frac{1}{3\cdot 2^s}V$.

Most points are at the border of C_s.

7.2.11 Product Simpson Rule (`Rule3Simpson`)

Degree 3, Fully Symmetric with $3^s = O(3^s)$ points, positive-weight formula. For $s \geq 1$.
C_s:3-6 in [Str71].
This is the tensor product formula of the 1-dimensional Simpson rule

$$\int_{-1}^{1} f(x)\,dx \approx \frac{f(-1)+4f(0)+f(1)}{3}.$$

7.2.12 A Degree 5 Rule due to Hammer and Stroud (`Rule5Hammer`)

Degree 5, Fully symmetric with $2s^2+1 = O(s^2)$ points, $\frac{1}{V}\sum_{i=1}^{n}|w_i| = O(s^2)$. For $s \geq 2$.
[HS58] and C_s:5-2 in [Str71].

Abscissas	w_i
$(0,\dots,0)$	W_0
$(\sqrt{\frac{3}{5}},0,\dots,0)_{\mathrm{FS}}$	$W_{\alpha,1}$
$(\sqrt{\frac{3}{5}},\sqrt{\frac{3}{5}},0,\dots,0)_{\mathrm{FS}}$	$W_{\alpha,2}$

7.2.13 A Degree 5 Rule due to Stroud (`Rule5Stroud`)

Degree 5, Symmetric with $3s^2+3s+1 = O(s^2)$ points, $\frac{1}{V}\sum_{i=1}^{n}|w_i| = O(s^2)$. For $s \geq 2$.
[Str68] and C_s:5-3 in [Str71].

Abscissas	w_i
$(0,\dots,0)$	W_0
$(\alpha,\alpha,0,\dots,0)_{\mathrm{S}}$	$W_{\alpha,2}$
$(-\alpha,-\alpha,0,\dots,0)_{\mathrm{S}}$	$W_{\alpha,2}$
$(\alpha,0,\dots,0)_{\mathrm{FS}}$	$W_{\alpha,1}$
$(\beta,-\gamma,0,\dots,0)_{\mathrm{S}}$	$W_{\beta+\gamma}$
$(-\beta,\gamma,0,\dots,0)_{\mathrm{S}}$	$W_{\beta+\gamma}$
$(\beta,0,\dots,0)_{\mathrm{FS}}$	$W_{\beta,\gamma}$
$(\gamma,0,\dots,0)_{\mathrm{FS}}$	$W_{\beta,\gamma}$

with $\alpha = \sqrt{\frac{7}{15}}$, $\beta = \sqrt{\frac{7+\sqrt{24}}{15}}$, and $\gamma = \sqrt{\frac{7-\sqrt{24}}{15}}$.

7.2.14 Product Gauss Rule of Degree 5 (`Rule5Gauss`)

Degree 5, Fully symmetric with $3^s = O(3^s)$ points, positive-weight formula. For $s \geq 1$.
C_s:5-9 in [Str71].
This is the tensor product formula of the 1-dimensional 3-point degree 5 Gauss formula

$$\int_{-1}^{1} f(x)\,dx \approx \frac{5f(-\sqrt{\frac{3}{5}})+8f(0)+5f(\sqrt{\frac{3}{5}})}{9}.$$

7.2.15 A Degree 7 Rule due to Phillips (`Rule7Phillips`)

Degree 7, Fully symmetric with $\frac{4}{3}s^3 - 2s^2 + \frac{14}{3}s + 1 = O(s^3)$ points, $\frac{1}{V}\sum_{i=1}^{n}|w_i| = O(s^3)$. For $s \geq 5$.

[Phi67, Dob70] and C_s:7-1 in [Str71].

Abscissas	w_i
$(0,\ldots,0)$	W_0
$(1,0,\ldots,0)_{\mathrm{FS}}$	W_α
$(\beta_s,0,\ldots,0)_{\mathrm{FS}}$	W_β
$(\sqrt{\frac{3}{5}},\sqrt{\frac{3}{5}},0,\ldots,0)_{\mathrm{FS}}$	W_γ
$(\sqrt{\frac{3}{5}},\sqrt{\frac{3}{5}},\sqrt{\frac{3}{5}},0,\ldots,0)_{\mathrm{FS}}$	W_δ

This formula is proposed in [Phi67]. In [Dob70] Dobrodeev proposes a 7th-degree formula with the same basic structure. He presents the formulas for abscissas and weights without showing how these results have been obtained. Even though the notation used by Dobrodeev is significantly different, it turns out that both formulas are actually identical. Due to the fact that Dobrodeev's paper does not contain a derivation and was published three years after Phillips' paper, this formula is referred to as Rule Phillips and not as Rule Dobrodeev in HINTLIB.

7.2.16 A Degree 9 Rule due to Stenger (`Rule9Stenger`)

Degree 9, Fully symmetric with $\frac{4}{3}s^4 - \frac{20}{3}s^3 + \frac{56}{3}s^2 - \frac{28}{3}s + 4 = O(s^4)$ points, $\frac{1}{V}\sum_{i=1}^{n}|w_i| = O(s^4)$. For $s \geq 4$.

[Ste63] and C_s:9-1 in [Str71].

Abscissas	w_i
$(0,\ldots,0)$	W_0
$(\alpha,0,\ldots,0)_{\mathrm{FS}}$	W_α
$(\beta,0,\ldots,0)_{\mathrm{FS}}$	W_β
$(\alpha,\alpha,0,\ldots,0)_{\mathrm{FS}}$	$W_{2\alpha}$
$(\beta,\beta,0,\ldots,0)_{\mathrm{FS}}$	$W_{2\beta}$
$(\alpha,\beta,0,\ldots,0)_{\mathrm{FS}}$	$W_{\alpha\beta}$
$(\beta,\beta,\beta,0,\ldots,0)_{\mathrm{FS}}$	$W_{3\beta}$
$(\alpha,\alpha,\alpha,\alpha,0,\ldots,0)_{\mathrm{FS}}$	$W_{4\alpha}$
$(\beta,\beta,\beta,\beta,0,\ldots,0)_{\mathrm{FS}}$	$W_{4\beta}$

with $\alpha = \frac{5}{9} + \sqrt{\frac{50}{567}}$, $\beta = \frac{5}{9} - \sqrt{\frac{50}{567}}$.

7.2.17 An Embedded Degree 7 Rule Due to Genz and Malik (`Rule75GenzMalik`)

Degree 7, Fully symmetric with $2^s + 2s^2 + 2s + 1 = O(2^s)$ points,
$\frac{1}{V}\sum_{i=1}^{n}|w_i| = O(s^2)$. For $s \geq 2$. [GM80].

Abscissas	w_i	w_i'
$(0,\ldots,0)$	W_0	W_0'
$(\alpha,0,\ldots,0)_{\mathrm{FS}}$	W_α	W_α'
$(\beta,0,\ldots,0)_{\mathrm{FS}}$	W_β	W_β'
$(\beta,\beta,0,\ldots,0)_{\mathrm{FS}}$	$W_{2\beta}$	$W_{2\beta}'$
$(\gamma,\ldots,\gamma)_{\mathrm{FS}}$	W_δ	

with $\alpha = \sqrt{\frac{9}{70}}$, $\beta = \sqrt{\frac{9}{10}}$, and $\gamma = \sqrt{\frac{9}{19}}$.

This cubature rule is proposed by Genz and Malik in [GM80]. It is special in various ways: To begin with, this rule contains an embedded degree 5 rule, which uses a subset of the abscissas of the degree 7 rule, so no additional integrand evaluations are required to evaluate it. Therefore, `Rule75GenzMalik` is a subclass not only of `CubatureRule`, but also of `EmbeddedRule`.

The coordinates of the abscissas of this rule do not depend on the dimension s, which simplifies the implementation of this rule. Even though the number of abscissas increases exponentially with the dimension s, the number of abscissas for $s \leq 11$ is small compared to other degree 7 rules. However, the probably most important feature of this rule is the low sum of its absolute weights, which makes its results extraordinarily stable.

This cubature rule is only a special case of a family of cubature rules developed by Genz and Malik in [GM83]. In this article a complete theory of a family of cubature rules $R^{(m,s)}$ of degree $2m + 1$ for arbitrary dimensions s and $1 \leq m \leq 6$ is developed, with the rule given here corresponding to $R^{(3,s)}$. This family is fully embedded, i. e., the abscissas of $R^{(m-1,s)}$ are a subset of the abscissas of $R^{(m,s)}$, allowing the construction of efficient embedded cubature rules of degree $3, 5, \ldots, 13$. Even though Cools and Haegemans [CH94] have tackled the resulting equations for weights and abscissas with a computer algebra systems and achieved significant simplifications, especially for the generation of the abscissa set, this general family of rules is not available in HINTLIB at the moment.

Chapter 8

Algebraic Structures

Digital nets and sequences, as well as other number theoretic constructions for low discrepancy sequences and pseudo random numbers are based on algebraic structures like finite rings and fields, as well as polynomials and vector spaces based thereupon.

This chapter describes the classes available in HIntLib for performing arithmetic and other operations in these structures. In Section 8.1 all member functions and types appearing in various kinds of algebraic structures are explained. Section 8.2 describes the classes that actually implement these concepts in the current version of HIntLib.

8.1 Algebraic Concepts

Algebraic structures share concepts, not interfaces, because the basic operations can be very simple and may require inlining to be executed sufficiently fast. Most algebraic structures implement one or more the the concepts discussed in this section.

8.1.1 Basic Features

An algebraic structure in the sense of HIntLib is a class encapsulating the features of some (number theoretical) algebraic structure. The methods provided by these classes perform the algebraic operations available in this structure (e. g. adding two values), while the data members contain all information necessary for actually performing these calculations. This data may consist of nothing (e. g. for normal integer arithmetic) up to tables used for implementing algebraic operations based on table lookups.

Algebraic structures have normal value semantic: they have a copy constructor, which creates a deep, independent copy of the original structure. On the other hand, copying algebraic structures is always cheap (i. e. they could be passed by value instead of by reference). Some algebraic structures may use techniques like reference counting to achieve this.

The following features are shared by all algebraic structures:

```
class A
{
public:
   typedef see text type;

   typedef see text algebra_category;
   typedef see text size_category;

   unsigned size() const;
   type element(unsigned) const;
```

```
    unsigned index(type) const;

    void print(std::ostream &, type) const;
    void printShort(std::ostream &, type) const;
    void printSuffix(std::ostream &) const;
};
```

Each algebraic structure A has a typedef `type`, specifying the type used for storing an element $x \in A$. It can be assumed that `operator==()`, `operator=()`, a copy constructor and a default constructor are defined for `type`. `type` must have value semantic, i. e. assigning it or passing it to a copy constructor results in a copy that is independent of the original.

`size()` returns $|A|$, the number of distinct elements in the structure—or 0 if A is infinite.

`element()` allows to enumerate the elements of A, while `index()` assigns an index number to each element $x \in A$. For finite structures, `element(0)`, ..., `element(`$|A|-1$`)` produce all elements of A. In this case `index()` and `element()` are bijections with `index(element(`i`))`=i for $i = 0, \ldots, |A| - 1$ and `element(index(`x`))`=x for all $x \in A$. If A is countable and infinite, `element()` and `index()` are still bijections with `index(element(`i`))`=i for all $i \in \mathbb{N}$ and `element(index(`x`))`=x for all $x \in A$. If A is uncountable, `element()` is injective with `index(element(`i`))`=i for all $i \in \mathbb{N}$, but it cannot be surjective anymore. For $x \in A$ with $x \notin$`element(`$\mathbb{N}$`)`, `index()` may return any number. However, `element(`$\mathbb{N}$`)` can be assumed to be dense in A (at least in some sense).

`print()` prints an element on an `ostream`. `printShort()` is similar, but it suppresses extra information like a modulus, which makes the result fit for use, for instance, as a coefficient of a polynomial. `printSuffix()` prints the information that is left out by `printShort()`.

Algebra Traits

An algebraic structure declares a number of types that can be used by templates to decide which features are available and what kind of algorithms can be used for implementing a certain task. A good example for how these types can be set to work is the class `PolynomialRing<>` (Section 8.2.3), which provides different and often optimized sets of methods depending on the algebraic structure that serves as coefficient ring or field.

`algebra_category` declares which operations can be performed in A. The following subsections discuss the member functions that are available for each of these types:

```
struct group_tag {};
 struct ringfield_tag
  struct ringdomain_tag   : public ringfield_tag {};
   struct ring_tag         : public ringdomain_tag {};
   struct domain_tag       : public ringdomain_tag {};
    struct ufd_tag          : public domain_tag {};
     struct euclidean_tag    : public ufd_tag {};
      struct integer_tag      : public euclidean_tag {};
      struct polyoverfield_tag: public euclidean_tag {};
  struct field_tag         : public ringfield_tag {};
   struct gf_tag           : public field_tag {};
    struct cyclic_tag       : public gf_tag {};
   struct realcomplex_tag    : public field_tag {};
    struct real_tag         : public realcomplex_tag {};
    struct complex_tag      : public realcomplex_tag {};
   struct numberfield_tag  : public field_tag {};
    struct rational_tag     : public numberfield_tag {};
   struct funfield_tag     : public field_tag {};
    struct ratfunfield_tag  : public funfield_tag {};
 struct vectorspace_tag  : public group_tag {};
```

`size_category` declares whether A is finite or infinite. It is a typedef for one of the following two types:

```
struct finite_tag   { static const bool finite = true;  };
struct infinite_tag { static const bool finite = false; };
```

`A::size_category::finite` can be used in an `if`-statement for determining whether A is finite at compile time.

Algebraic structures whose algebra category is a subclass of `ringfield_tag` define the following additional algebra traits: `polynomial_category`, `char_category`, and `zerodivisor_category`. These traits are described in Section 8.1.3.

Notes on the Interface Definitions

The signatures of the member functions of actual implementations may differ slightly form the signatures given here. The following list contains some examples of differences that may be encountered. However, the user should not assume that this list is complete—don't try to do anything too funny with the members of algebraic structures!

- Instead of `type`, an argument will usually be of type `const type &` if `type` has an expensive copy constructor.
- Member functions are usually declared `static` if this is possible.
- If a method returns a `const X&`, the reference is only valid during the lifetime of the algebraic structure. However, some implementation may return an `X` instead.
- Instead of returning `type`, some classes will return stub objects that only contain sufficient information to create the `type` when necessary. An automatic conversion to `type` is always defined in this case, so the user should not notice any difference in standard situations. However, the implementation can avoid unnecessary copying of `type`s using this approach.
  This kind of optimization is used, e. g., by `PolynomialRing<>` (Section 8.2.3).

Aliasing of Arguments

A number of methods of algebraic structure use non-const reference arguments in order to change one or more of their arguments. Care must be taken if such an argument aliases another argument passed by const or non-const reference.

Methods using one non-const and one const argument (prominent examples are `addTo()`, `subFrom()`, `mulBy()`, `divBy()`, `reduce()`, and `quotient()`) never allow aliasing. The expected result of these operations applied to identical arguments would either be 0 or 1, or some operation for which a dedicated method is provided (e. g. `times2()` or `square()`). Therefore, allowing aliasing in this situation would only complicate implementation without providing any advantage for the user.

Some other methods (e. g. `div()` of Euclidean domains, `isDivisor()`, `genGcd()` ...), however, often allow some kind of aliasing. In some cases, faster implementations can be used if aliasing is detected. In any case, information on the kind of aliasing allowed in a particular situation can be found in the description of the method.

8.1.2 Abelian Groups

The simplest algebraic concept used in HINTLIB is an Abelian group (or just group). It is shared by algebraic structures like rings, fields or vector spaces. A group has only a single operation $\oplus$, which is associative and commutative. There is also a neutral element denoted by "zero" or 0, and for each element $a \in A$, a negative (denoted by $-a$) can be found, such that $a \oplus -a = 0$.

Method	Equivalent code
`addTo(a,b)`	`a = add(a,b)`
`negate(a)`	`a = neg(a)`
`sub(a,b)`	`add(a,neg(b))`
`subFrom(a,b)`	`a = sub(a,b)`
`dbl(a)`	`add(a,a)`
`times2(a)`	`a = dbl(a)`

`times(a,k)`	`type t = type();`
	`for (unsigned i = 0; i<k; ++i) addTo(t, a);`
	`return t;`

`is0(a)`	`a == type()`

Table 8.1: Derived group operations

In addition to the basic features listed in Section 8.1.1, each group provides the following methods:

```
class A
{
public:
   typedef group_tag algebra_category;

   bool is0(type) const;

   type add  (type,  type) const;
   void addTo(type&, type) const;

   type neg   (type)  const;
   void negate(type&) const;

   type sub    (type,  type) const;
   void subFrom(type&, type) const;

   type dbl   (type)  const;
   void times2(type&) const;[1]

   type times(type, unsigned k) const;
   unsigned additiveOrder(type) const;
};
```

The neutral element of the group can be created by using the default constructor of `type`. It is also returned by `element(`0`)`.

`add(a,b)` returns $a \oplus b$ and `neg(`a`)` gives $-a$, the additive inverse of a in $(A, \oplus)$. `additiveOrder(`a`)` returns the order of a, denoted by $\mathrm{ord}_\oplus(a)$, in the group (i. e. the smallest n such that $\sum_{i=1}^{n} a = 0$), or 0 if $\mathrm{ord}_\oplus(a)$ is infinite. If $|A|$ is finite, $\mathrm{ord}_\oplus(a)$ is always a divisor of $|A|$.

All other methods could be implemented based on these four, according to Table 8.1. However, optimized versions can usually be provided.

8.1.3 `ringfield_tag`

No algebraic structure has an algebra category of `ringfield_tag`. This tag only serves as a common base class for `ringdomain_tag` (Section 8.1.4) and `field_tag` (Section 8.1.10).

[1] `times2` has to be used instead of the obvious `double` because `double` is a reserved word in C++.

Method	Equivalent code
`mulBy(a,b)`	`a = mul(a,b)`
`sqr(a)`	`mul(a,a)`
`square(a)`	`a = sqr(a)`

`power(a,k)`	`type t = one();`
	`for (unsigned i=0; i<k; ++i) mulBy(t, a);`
	`return t;`

`is1(a)`	`a == one()`

Table 8.2: Derived `ringfield_tag` operations

From a mathematical perspective, all algebraic structures with an algebra category being a subclass of `ringfield_tag` are commutative rings with identity, i. e. an algebraic structure A with two binary operations $\oplus$ and $\otimes$ such that $(A, \oplus)$ forms an Abelian group, $(A, \otimes)$ is a commutative semi-group with a neutral element denoted by 1 or "one", and the distributive laws hold.

In addition to the features of groups listed in Section 8.1.2, the following methods are provided:

```
class A
{
public:
   typedef ringfield_tag algebra_category; // never used
   typedef see text polynomial_category;
   typedef see text char_category;
   typedef see text zerodivisor_category;

   type one() const;

   bool is1(type) const;

   type mul  (type,  type) const;
   void mulBy(type&, type) const;

   type sqr   (type)  const;
   void square(type&) const;

   type power(type, unsigned k) const;
};
```

`one()` (as well as `element(`1`)`) returns the neutral element for the operation $\otimes$ and `mul(`a`,`b`)` returns $a \otimes b$. The other methods could be defined based on these two, according to Table 8.3. However, optimized implementations can usually be provided.

All algebraic structures with an algebra category subclass to `ringbase_tag` define three additional algebra traits (in addition to `algebra_category` and `size_category` described in Section 8.1.1):

`polynomial_category`

`polynomial_category` declares whether the elements of A are polynomials. It is defined as one of the following two types:

```
struct nopolynomial_tag {};
struct polynomial_tag {};
```

Section 8.1.16 discuesses additional methods available for polynomial rings, i. e. for algebraic structures with `polynomial_category` equal to `polynomial_tag`.

`char_category`

`char_category` provides information about the characteristic c of A. The characteristic is defined as the common additive order $\mathrm{ord}_\oplus(a)$ of all non-zero elements of A.

If no such number exists, the ring does not have a characteristic, and `char_category` is a typedef for `char_non`.

If c is the common additive order of all non-zero elements, it is either infinite (characteristic zero) or a prime number. In this case `char_zero` and `char_prime` are used as `char_category`, respectively. In some cases it may be known that $c = 2$ at compile time, which is signalized using the type `char_two`, a subclass of `char_prime`.

The four types `char_two`, `char_prime`, `char_zero` and `char_non` are part of the following class hierarchy:

```
struct char_any {};
 struct char_non : public char_any {};
 struct char_exists : public char_any {};
  struct char_zero : public char_exists {};
  struct char_prime : public char_exists {};
   struct char_two : public char_prime {};
```

The types `char_any` and `char_exists` are only used as base classes. No algebraic structure uses them directly as `char_category`.

If A has a characteristic, i. e. `char_category` is a subclass of `char_exists`, the algebraic structure provides the method

```
unsigned characteristic() const;
```

for determining the number c.

`zerodivisor_category`

`zerodivisor_category` declares whether the ring has zero divisors, i. e. whether the product of two non-zero elements can be zero. `zerodivisor_category` is a typedef for one of the following two types:

```
struct   zerodivisor_tag {};
struct nozerodivisor_tag {};
```

If A may have zero divisers, `zerodivisor_category` is a typedef for `zerodivisor_tag`.

If A is guaranteed to have no zero divisors, `zerodivisor_category` is set to `nozerodivisor_tag`. In this case, A also has a characteristic, i. e. `char_category` is a subclass of `char_exists`. In addition to that, structures without zero divisors define the following additional method:

```
unsigned order(type) const;
```

For non-zero elements `order(`a`)` returns the multiplicative order of a ($\mathrm{ord}_\otimes(a)$) (i. e. the smallest n such that $\prod_{i=1}^{n} a = 1$), or 0 if $\mathrm{ord}_\otimes(a)$ is infinite. If $|A|$ is finite, $\mathrm{ord}_\otimes(a)$ is always a divisor of $|A| - 1$. `order(`0`)` is undefined—it may throw `DivisionByZero`, trap, or show some other undefined bahaviour.

8.1.4 ringdomain_tag

No algebraic structure has an algebra category of ringdomain_tag. This tag only serves as a common base class for ring_tag (Section 8.1.5) and domain_tag (Section 8.1.6).

An element u of a ring A is called a unit, if it has a multiplicative inverse in A, i. e. there is an element $u^{-1} \in A$ such that $u^{-1} \otimes u = 1$. It follows immediately that the product of units is again a unit and that the set of all units of A, denoted by A^*, is the largest subset of A such that $(A^*, \otimes)$ is a group.

If all non-zero elements of A are units, the ring is a field, and algebraic structures with this property are described in Section 8.1.10. An algebraic structure with an algebra category subclass of ringdomain_tag is a commutative ring which is not necessarily a field, i. e. it may contain non-zero elements which are not units. ringdomain_tag structures provide the following operations in addition to the ones described in the previous section:

```
class A
{
public:
   typedef ringdomain_tag algebra_category;  // never used
   typedef see text unit_type;

   bool isUnit(type) const;
   unsigned numUnits() const;

   type fromUnit(unit_type) const;
   unit_type toUnit (type) const;

   unit_type unitRecip(unit_type) const;

   type mulUnit  (type,  unit_type) const;
   void mulByUnit(type&, unit_type) const;

   unit_type mulUnit  (unit_type,  unit_type) const;
   void      mulByUnit(unit_type&, unit_type) const;
};
```

isUnit(a) tests if a given element $a \in A$ is a unit. numUnits() returns the number of units in A, or 0 if this number is infinite. There is always at least one unit in A, namely the one-element of the ring.

Units can often be stored and manipulated in a simpler manner than arbitrary elements of A. Therefore a type unit_type is provided which is used for storing units. In some situations unit_type may be identical to type, but often it is not. unit_type provides operator==(), operator=() and a copy constructor. Two methods toUnit() and fromUnit() allow to convert from type to unit_type and vice versa. toUnit(a) is only defined if a is a unit.

Units have always a multiplicative inverse, which can be found by unitRecip(). A unit can be multiplied with another unit, resulting in a unit, as well as with an arbitrary element of A. These operations could be defined based on normal ring arithmetic, according to Table 8.3. However, optimized versions can usually be provided.

Some ringdomain_tag structures may provide the methods unitElement() and unitIndex() (required by UFDs, Section 8.1.7). However this is not always the case.

8.1.5 Rings

A ring with algebra category ring_tag is a ring as described in the previous section which (in general) may have non-units as well as zero divisors.

Method	Equivalent code
`mulUnit(unit_type a, unit_type b)`	`toUnit(mul( fromUnit(a),fromUnit(b)))`
`mulUnit(type a, unit_type b)`	`mul(a,fromUnit(b))`
`mulByUnit(unit_type& a,unit_type b)`	`a = mulUnit(a,b)`
`mulByUnit(type& a, unit_type b)`	`a = mulUnit(a,b)`

Table 8.3: Derived ring operations

```
class A
{
public:
   typedef ring_tag algebra_category;

   bool isNilpotent(type) const;
   unsigned numNilpotents() const;
};
```

`isNilpotent(`a`)` determines whether a is nilpotent, i. e. there is a $k \in \mathbb{N}$ such that $a^k = 0$. Every non-zero nilpotent element is a zero divisor. However, the opposite is not true in general. `numNilpotents()` returns the number of nilpotent elements in A, or 0 if this number is infinite. There is always at least one nilpotent element: the neutral element 0 of the ring.

8.1.6 Integral Domains

An integral domain is a ring as described in Section 8.1.4, which has no zero divisors, i. e. the product of two non-zero elements cannot be zero.

```
class A
{
public:
   typedef domain_tag algebra_category;

   typedef nozerodivisor_tag zerodivisor_category;

   bool isAssociate (type, type) const;
   bool isAssociate (type, type, unit_type&) const;

   bool isDivisor (type, type) const;
   bool isDivisor (type, type, type&) const;

   type div   (type,  type) const;
   void divBy (type&, type) const;
};
```

In a domain `zerodivisor_category` is always `nozerodivisor_tag`, and `char_category` is always a subclass of `char_exists`.

`isAssociate(`a`,`b`)` determines if a is associate to b (denoted by $a \sim b$), i. e. there is a unit $u \in A$ such that $u \otimes b = a$. If such a u exists, it is uniquely defined. The three-argument version of `isAssociate()` stores $u = a/b$ in the variable specified by the third argument, if $a \sim b$. If a is not associate to b, u is undefined after a call to `isAssociate(`a`,`b`,`u`)`. 0 is only associate to 0, and `isAssociate(`0`,`0`,`u`)` assigns 1 to u. The relation $\sim$ is an equivalence relation and defines a partition on A.

For $b \neq 0$, `isDivisor(`a`,`b`)` returns true if and only if b is a divisor of a (denoted by $b|a$), i. e. there is a $c \in A$ such that $c \otimes b = a$. If such a c exists, it is the unique result of the exact division of a and b, also denoted by a/b. The three-argument version of `isDivisor()` stores $c = a/b$ in the variable specified by the third argument, if $b|a$. If b does not divide a, c is undefined after a call to `isDivisor(`a`,`b`,`c`)`.[2]

If it is known beforehand that b divides a, `div(`a`,`b`)` and `divBy(`a`,`b`)` can be used to calculate a/b directly. If b is 0, the result of all these methods is undefined.

8.1.7 Unique Factorization Domains

A unique factorization domain (UFD) is an integral domain with the additional property that each element $a \in A$ can factored into prime elements in only one way (ignoring multiplication by units), i. e. the factorization of a is unique. In other words, each element $a \in A$ can be written as

$$a = u \cdot p_1^{k_1} \cdots p_n^{k_n}$$

with u a unit of A and p_i irreducible elements in A.

Consider the equivalence classes defined by the relation $\sim$. In a UFD a well-defined element $\bar{x} \in [x]$ (called the canonical form) can be chosen from each class $[x]$ in the following way: The canonical form of $[0]$ is 0. For $[1]$, the class containing all units, the canonical form is 1. For $[p]$ with p prime, choose an arbitrary element $\bar{p} \in [p]$. For all remaining classes $[a] = [u \cdot p_1^{k_1} \cdots p_n^{k_n}]$ define the canonical form as $\bar{a} := \bar{p_1}^{k_1} \cdots \bar{p_n}^{k_n}$. It follows that the product of two canonical forms is again a canonical form.

UFDs provide the following methods in addition to those for integral domains described in Section 8.1.6:

```
class A
{
public:
   typedef ufd_tag algebra_category;
   typedef see text primedetection_category;

   unit_type unitElement(unsigned) const;
   unsigned unitIndex(unit_type) const;

   bool isCanonical(type) const;
   unit_type makeCanonical(type &) const;

   // Depending on primedetection_category,
   // the following methods may be available

   bool isPrime(type) const;
   bool isComposit(type) const;
   typedef see text PrimeGenerator;

   typedef std::vector<std::pair<type,unsigned> > Factorization;
   unit_type factor(Factorization&, type);
};
```

`isCanonical(`x`)` determines whether $x = \bar{x}$. `makeCanonical(`x`)` replaces x by its canonical form $\bar{x}$ and returns a unit u such that $x = u\bar{x}$.

All units can be enumerated using `unitElement()`. If the number of units is finite, `element(1)`,..., `element(numUnits())` can also be used to retrieve all units. However if there

[2] Aliasing information missing!!!

are infinitely many units in A, `element()` will eventually start to produce non-unit elements, while `unitElemnt(`i`)` returns a new unit for each $i \in \mathbb{N}$. In addition to that, note that `unitElement()` returns a `unit_type`, while `element()` returns a `type`. `unitIndex()` is the reverse operation to `unitElement()`.

primedetection_category

In a UFD there are certain special elements p which are only divisible by 1 and itself (ignoring multiplication by units), i. e. p is neither 0 nor a unit, and for all elements $a, b \in A$ with $a \otimes b = p$, it follows that either a or b is a unit. Elements with this property are called primes.

Therefore, in a UFD each element $a \in A$ belongs to exactly one of the following four categories:

1. a is 0.
2. a is a unit.
3. a is prime.
4. a is composite, i. e. it can be factored into two or more primes.

In practice it may be difficult to determine into which category a certain element belongs. Especially the distinction between primes and composites is often hard. Some UFDs in HIntLib allow this distinction and provide the methods described in the remainder of the section.

Together with `is0()` and `isUnit()` (already defined for groups and rings in Sections 8.1.2 and 8.1.4, respectively) `isPrime()` and `isComposit()` allow to determine which of the four categories a given element a belongs to. For each $a \in A$, exactly one of these four methods returns true.

Even more important than enumerating all units is the enumeration of primes. However, in the case of primes it is usually much harder to provide a pair of functions like `unitElement()` / `unitIndex()`, therefore this approach has not been taken. Each UFD defines a type `PrimeGenerator` which is a class defining the following methods:

```
class A::PrimeGenerator
{
public:
   PrimeGenerator(const A&);
   A::type next();
};
```

The constructor creates a `PrimeGenerator` for a given UFD A. Each time the method `next()` is called, a new prime in canonical form is returned. `PrimeGenerator`s can neither be copied nor assigned and must be destructed before their algebraic structure.

Some UFDs provide a method `factor(`a`)` with calculates the complete factorization $a = u \cdot p_1^{d_1} \cdots p_n^{d_n}$ of a into a unit u and prime factors p_i in canonical form. The tuples (p_i, d_i) are stored in a data structure of type `Factorization`, the unit u is returned.

Which of these methods are available is determined by the type of `primedetection_category`. It is a typedef for one of the following types:

```
struct noprimedetection_tag {};
struct primedetection_tag : public noprimedetection_tag {};
struct factor_tag         : public   primedetection_tag {};
```

`factor_tag` signalized that all methods described above are available. `primedetection_tag` ensures that at least `isPrime()`, `isComposit()`, and `PrimeGenerator` are available.

8.1.8 Euclidean Domains

Euclidean domains are UFDs with the additional property that a division with remainder can be performed, such that the remainder is in some sense smaller than the divisor. To be more specific, there is a function $|\cdot| : A \to \mathbb{N}$, called a norm, with the following properties:[3] $|a| = 0$ if and only if $a = 0$, $|a| = 1$ if and only if a is a unit, and $|a| \leq |a \otimes b| \leq |a|\,|b|$ if $b \neq 0$. In addition to that, if the number of units in A is finite, we have $|a| \leq |b|$ whenever `index(`a`)`$\leq$ `index(`b`)`.

Most noteworthy, for all $a \in A$ and $b \in A \setminus \{0\}$, elements $q, r \in A$ can be found such that $a = (q \otimes b) \oplus r$ and $|r| < |b|$. In other words, the Euclidean algorithm can be performed: the remainder will eventually decrease to zero and the algorithm terminates.

Euclidean domains provide the following methods in addition to the ones for UFDs defined in Section 8.1.7:

```
class A
{
public:
   typedef euclidean_tag algebra_category;

   void div (type, type, type &, type &) const;
   type quot(type, type) const;
   type rem (type, type) const;

   void reduce  (type&, type) const;
   void quotient(type&, type) const;

   unsigned norm(type) const;
   unsigned numOfRemainders(type) const;
};
```

`div(`a`,`b`,`q`,`r`)` sets q and r as defined above. q as well as r are allowed to alias a as well as b. If the calculation of only either q or r is required, `quot()` or `rem()` can be used, respectively. `reduce(`a`,`b`)` and `quotient(`a`,`b`)` are equivalent to a`=rem(`a`,`b`)` and a`=quot(`a`,`b`)`, respectively. Providing an efficient implementation for `reduce()` is crucial, because it is a the basic building block for `genGcd()` and `powerMod()`.

`numOfRemainders(`b`)` returns the number of different remainders that can occur when dividing by b, i. e. the size of the factor ring A/bA. If this value is infinite, 0 is returned.

Most algebraic structures will throw `DivisionByZero` if b is 0 while calling any of these methods. However, some may trap or show some other undefined behavior.

Finally, `norm(`a`)` returns $|a|$.

Performing the Euclidean Algorithm

The header file `gcd.h` defines the following functions for calculating the greatest common divisor (GCD) of two elements in an Euclidean domain:

```
template<typename A>
A::type
genGcd (const A&, const A::type&, const A::type&, A::type&, A::type&);

template<typename A>
A::type
genGcd (const A&, const A::type&, const A::type&, A::type&);
```

[3] The concept of "norm" used here is weaker than what is commonly used in number theory. For instance, it does not hold in general that $|a \otimes b| = |a|\,|b|$. If $|\cdot|$ were defined in the usual way, the resulting values would often be too large for an `unsigned`.

```
template<typename A>
A::type
genGcd (const A&, const A::type&, const A::type&);
```

If a is an instance of an Euclidean domain, `genGcd(`a`,`u`,`v`,`m_u`,`m_v`)` returns the greatest common divisor $g = \gcd(u, v)$ of u and v and sets m_u and m_v such that $m_u u + m_v v = g$. If u or v is zero, $\gcd(u, v)$ is zero.

In general, g will not be in canonical form. If this is required, `makeCanonical()` has to be applied to g and m_u and m_v have to be multiplied with the reciprocal of the unit returned by `makeCanonical()`.

If no multiplier or only the multiplier m_u is required, one of the other versions of `genGcd()` can be used.

A far as aliasing is concerned, m_u is only allowed to alias u while m_v is only allowed to aliased v. This is the kind of aliasing that seems to be most useful in applications.

All versions of `genGcd()` perform most efficiently if $|u| \leq |v|$. So if the relative magnitude of u and v is known beforehand, the smaller one should be given before the larger one. Of course this is only possible for the versions of `genGcd()` determining both or no multiplier—if one multiplier is requested, the corresponding argument must be given first (in many applications where only one multiplier is needed, e. g. finding the reciprocal in factor rings and fields, the corresponding element will also be the smaller one).

In addition to `genGcd()`, two related functions are provided in `gcd.h`:

```
template<typename A>
A::type
genLcm (const A&, const A::type&, const A::type&);

template<typename A>
bool
genIsCoprime (const A&, const A::type&, const A::type&);
```

`genLcm(`a`,`u`,`v`)` determines the least common multiplier (LCM) of u and v. `genIsCoprime(`a`,`u`,`v`)` determines if u and v are coprime, i. e. if $\gcd(u, v)$ is a unit.

8.1.9 The Euclidean Ring of Integers

The set of integers $\mathbb{Z}$ with the common addition and multiplication is an Euclidean domain. Implementations of this structure use the `algebra_categoy` `integer_tag`, a subclass of `euclidean_tag`. The ring of integers has the following properties in addition to the ones discusses in Section 8.1.8 for Euclidean domains:

- `type` is either one of the built-in signed integer types, or a conversion operator to and from `int` is defined for `type`.
- `element()` enumerates $\mathbb{Z}$ in the order $0, 1, -1, 2, -2, \ldots$
- `unit_type` is `int`, and `unitElement()` enumerates the two units in the order $1, -1$.
- `norm(`a`)` returns $|a|$.
- The canonical form of a is $|a|$.
- `PrimeGenerator` enumerates all positive prime numbers in increasing order.

Method	Equivalent code
`reciprocal(a)`	`a = recip(a)`
`div(a,b)`	`mul(a,recip(b))`
`divBy(a,b)`	`a = div(a,b)`

Table 8.4: Derived field operations

8.1.10 Fields

A field is an arithmetic structure $(A, \oplus, \otimes)$ such that $(A, \oplus)$ as well as $(A \setminus \{0\}, \otimes)$ are Abelian groups, and the distributive laws hold. From a mathematical perspective a field is also an Euclidean Ring, with the additional properties that there are no primes and composites, and the remainder of divisions is always 0. Since most of the members introduced for Euclidean rings and UFDs would become dummies for fields, returning an identical result independent of their argument, `field_tag` is a direct subclass of `ringbase_tag` instead of `euclidean_tag`. So none of the superfluous members needs to be implemented.

In a field `zerodivisor_category` is always `nozerodivisor_tag`, and `char_category` is always a subclass of `char_exists`. Therefore, the methods `order()` and `characteristic()` are always defined. Fields provide the following methods in addition to the ones for `ringbase_tag` in Section 8.1.3.

```
class A
{
public:
   typedef field_tag algebra_category;

   type recip     (type)   const;
   void reciprocal(type &) const;

   type div  (type,   type) const;
   void divBy(type &, type) const;
};
```

`recip(`a`)` returns the multiplicative inverse of a. Many algebraic structures will throw `DivisionByZero` if a is 0. All other new methods can be defined based on `recip()` according to Table 8.4. However, optimized implementations can usually be provided.

8.1.11 The Field of Rational Numbers

The rational numbers $\mathbb{Q}$ form a field which can be recognized by its `algebra_category` `rational_tag`, a subclass of `field_tag`.

```
class A
{
public:
   typedef rational_tag algebra_category;

   typedef see text base_algebra;
   typedef typename base_algebra::type base_type;

   const base_algebra& getBaseAlgebra() const;

   type makeElement(base_type) const;
```

```
   type makeElement(base_type, base_type) const;
};
```

Rational numbers are based on the quotient of two integer numbers. The type of the algebraic structure which can be used for doing these integer calculations is available as `base_algebra`. `base_algebra::algebra_category` is always `integer_ring`. `getBaseAlgebra()` is used to get an instance of the `base_algebra` in use.

`base_algebra::type`, the type of each integer numerator and denominator, is available as `base_type`.

`makeElement(`a`)` returns the rational number $a \in \mathbb{Q}$ for $a \in \mathbb{Z}$. `makeElement(`a`,`b`)` returns $a/b \in \mathbb{Q}$ for $a, b \in \mathbb{Z}$ and $b \neq 0$.

8.1.12 The Field of Real Numbers

The set of real numbers $\mathbb{R}$, together with normal addition and multiplication, forms a field. It can be recognized by its `algebra_category` `real_tag`, a subclass of `field_tag`.

```
class A
{
public:
   typedef real_tag algebra_category;
   typedef see text complex_field;
   typedef typename complex_field::type complex_type;

   const complex_field& getComplexField() const;
};
```

`complex_field` is the type of an algebraic structure with algebra category `complex_tag` (see Section 8.1.13), which is used (or could be used alternatively) to perform the arithmetic operations in A. `getComplexField()` returns an instance of this structure.

`type` is either one of the built-in floating point types, or a conversion operator to and from `real` is defined for `type`. The following functions must be defined for `type`:

```
bool operator< (type a, type b);
type abs (type a);
```

In addition to `operator==()`, which is defined for all types used by algebraic structures, `operator<()` returns true if $a < b$. It must be consistent with `operator==()`, i. e. only one of these functions may return true for any given pair of numbers a and b. `abs(a)` returns the absolute value $|a|$ of a.

It should be noted that it is impossible to model the real numbers accurately on a computer. Therefore, each implementation of this concept will have some deficiency.

8.1.13 The Field of Complex Numbers

The complex numbers $\mathbb{C}$ form a field which can be recognized by its `algebra_category` `complex_tag`, a subclass of `field_tag`.

```
class A
{
public:
   typedef complex_tag algebra_category;
   typedef see text real_field;
   typedef typename real_field::type real_type;
```

```
   const real_field& getRealField() const;

   real_type re (type) const;
   real_type im (type) const;
};
```

`real_field` is the type of an algebraic structure with algebra category `real_tag` (see Section 8.1.12), which is used (or could be used alternatively) to perform the arithmetic operations in A. `getRealField()` returns an instance of this structure.

`re(`z`)` and `im(`z`)` return the real and the imaginary part of z.

`type` provides at least the following constructors in addition to default and copy constructor which are available for all `type`s. A conversion to `std::complex<real>` is also always defined.

```
class A::type
{
public:
   A::type (std::complex<real>);
   A::type (real);
   A::type (real, real);
   A::type (A::real_type);
   A::type (A::real_type, A::real_type);

   operator std::complex<real>() const;
};

real_type abs (type);
```

The function `abs(z)` returns the absolute value $|z|$ of z.

It should be noted that it is impossible to model the complex numbers accurately on a computer. Therefore, each implementation of this concept will have some deficiency.

8.1.14 Finite Fields

Finite fields are fields with a size $|A| = p^k < \infty$ with p prime. The following methods are available in addition to the ones defined for fields (Section 8.1.10).

```
class A
{
public:
   typedef gf_tag algebra_category;

   unsigned extensionDegree() const;

   bool isPrimitiveElement(type) const;
};
```

`extensionDegree()` returns the extension degree of the field, i. e. the number k in $|A| = p^k$ with p prime. `characteristic()` (which was already defined for domains in Section 8.1.3) returns p.

`isPrimitveElement(`a`)` returns true if a is a primitive element of the field, i. e. a is a generating element of the multiplicative group, i. e. $a \neq 0$ and $\mathrm{ord}_{\otimes}(a) = |A| - 1$ (see Section 8.1.3). If $A = \mathbb{Z}/(p)$ this is equivalent to a being a primitive root modulo p.

8.1.15 Fields with Cyclic Additive Groups

In general, the additive group of a finite field is not cyclic, i. e. the characteristic of the field is smaller than the size of the field. However, for finite fields of size p, with p prime, the whole set A can be generated by successively adding 1 (or any other non-zero element). This special situation is declared by setting `algebra_category` to `cyclic_tag`, a subclass of `gf_tag`.

```
class A
{
public:
   typedef cyclic_tag algebra_category;
};
```

There are no additional methods for fields with cyclic additive groups. However, due to the structure of $\mathbb{Z}_p$ it can be deduced that `characteristic()`=`size()` and `extensionDegree()`$= 1$.

8.1.16 Polynomial Rings

Polynomials p are terms of the form

$$a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

with a_i denoting coefficients from some ring R, and x being a formal parameter (indeterminant) that is not further defined. a_n is different from 0 and is called the leading coefficient of p, a_0 is called the constant term. n is called the degree of the polynomial ($\deg p$); if $p = 0$, $\deg p$ is defined as -1.

Addition and multiplication are defined as

$$(a_n x^n + \cdots + a_1 x + a_0) \oplus (b_m x^m + \cdots + b_1 x + b_0) := \sum_{i=0}^{\infty} (a_i + b_i) x^i$$

and

$$(a_n x^n + \cdots + a_1 x + a_0) \otimes (b_m x^m + \cdots + b_1 x + b_0) := \sum_{i=0}^{\infty} x^i \sum_{j=0}^{i} a_j b_{i-j}$$

with a_i and b_j assumed to be 0 for $i > n$ and $j > m$, respectively.

Polynomial rings are always rings, sometimes even domains, UFDs or Euclidean rings, and inherit all properties described in Sections 8.1.5, 8.1.6, 8.1.7, and 8.1.8, respectively. In addition they provides the following methods:

```
class A
{
public:
   typedef polynomial_tag polynomial_category;

   typedef see text coeff_algebra;
   typedef coeff_algebra::type coeff_type;
   typedef see text coeff_reference;
   typedef std::vector<std::pair<type,unsigned> > Factorization;

   const coeff_algebra& getCoeffAlgebra() const;

   type x(unsigned = 1) const;

   bool isMonic(type) const;
```

```
    type mul  (type,  coeff_type) const;
    void mulBy(type&, coeff_type) const;

    type derivative(type) const;
    coeff_type evaluate (type, coeff_type) const;

    // For polynomials over a field

    bool isSquarefree(type) const;
    unit_type squarefreeFactor(Factorization&, type) const;
};
```

`coeff_algebra` is the type of the ring used for the coefficients of the polynomials; `coeff_algebra::algebra_category` it is always as subclass of `ringfield_tag`. `getCoeffAlgebra()` returns an instance of the `coeff_algebra` in use.

`coeff_type` is the type of a single coefficient. `coeff_reference` is the type used to reference a single coefficient inside the polynomial, used for instance by `type::operator[]()`. If some kind of packed data layout is used for the polynomial, this type will not be `coeff_type&`!

`isMonic()` returns true if the leading coefficient is 1 or if $p = 0$. For polynomials over a field, this method is identical to `isCanonical()` available in all UFDs.

`x(`k`)` returns the polynomial x^k.

In addition to the versions of `mul()` and `mulBy()` defined for all rings and fields, polynomial rings provide additional versions for multiplying a polynomial with an element from the coefficient ring.

`evaluate(`p`,`a`)` returns $p(a)$, i. e. the ring element a is substituted for the indeterminant x in the polynomial, and the whole expression is evaluated using normal ring arithmetic. `derivative(`p`)` returns the derivative of p. If $p(x)$ is of the form $a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0$, the derivative p' of p is given as

$$p'(x) = \left(\sum_{i=1}^{n} a_n\right) x^{n-1} + \left(\sum_{i=1}^{n-1} a_{n-1}\right) x^{n-2} + \cdots + a_1.$$

If the coefficients a_i belong the field of real numbers, this definition coincides with the definition known from analysis.

If R is finite, `element()` enumerates A in the following order: Let $d_0, d_1, \ldots$ denote the $|R|$-adic digits of $n \in \mathbb{N}$, i. e.

$$n = \sum_{i=0}^{\infty} d_i \, |R|^i$$

with $d_i \in \{0, \ldots, |R| - 1\}$, then `element(`$n$`)` returns $d'_0 + d'_1 x + d'_2 x^2 + \cdots$, where d'_i stands for the d_i-th element in R, returned by R`::element()`.

Special Methods of Polynomial Rings forming a UFD[4]

For polynomials forming a UFD, `isSquarefree(`p`)` determines if p contains the square of a non-unit factor. `squarefreeFactor()` factors the polynomial. However, the resulting factors are not prime (as produced by `factor()`), but only coprime and square free. While `factor()` needs a special implementation for every coefficient field or UFD, and is therefore only available in special cases, `squareFreeFactor()` can easily be provided for all polynomials forming a UFD.

If R is finite, `PrimeGenerator` enumerates all monic irreducible polynomials ordered by degree.

[4] At the moment `PolynomialRing<>` in HINTLIB implements these methods only for polynomial rings over a field.

`type` of Polynomial Rings

In addition to the members of the polynomial ring itself, the type used to store the polynomials, `type`, is a class that is guaranteed to have the following members:

```
class P
{
public:
   typedef see text coeff_type;
   typedef see text coeff_reference;

   P();
   P(P);
   template<typename I> P(I i, I i);

   P& operator= (P);

   int degree() const;
   bool is0() const;
   bool isConstant() const;

   coeff_type      operator[] (unsigned) const;
   coeff_reference operator[] (unsigned);

   coeff_type      lc() const;
   coeff_reference lc();
   coeff_type      ct() const;
   coeff_reference ct();

   template<typename I> void toCoeff(I) const;

   P& mulByX(unsigned = 1);
   P& divByX(unsigned = 1);
};
```

`coeff_type` and `coeff_reference` are identical to A`::coeff_type` and A`::coeff_reference`.

`degree()` returns the degree of the polynomial. `is0()` is identical to A`::is0()`, `isConstant(`p`)` returns true if $\deg p$ is less or equal to 0.

In addition to the copy constructor and the default constructor creating the polynomial $p(x) = 0$ which are present for all `type`s used by algebraic structures, polynomials have a constructor creating the polynomial based on the coefficients specified by two bidirectional iterators, the first one pointing to a_0, the second one at the element past a_n.

Coefficient a_i of a polynomial p can be accessed using p`[`i`]` for $0 \leq i \leq \deg p$. There is a const version of `operator[]()` returning `coeff_type`, as well as a non-const version returning a `coeff_reference` which can be written to. p`.lc()` (leading coefficient) is equivalent to p`[`p`.degree()]`, `ct()` (constant term) to p`[0]`, again providing a const and a non-const version. Finally, `toCoeff()` writes the coefficients of the polynomial to an output iterator, starting with a_0.

`mulByX()` multiplies a polynomial by x, `divByX()` divides it by x. `mulByX(`k`)` multiplies it by x^k, while `divByX(`k`)` divides it by x^k. Any remainder appearing in these divisions is dropped. All four methods return a reference to the resulting polynomial.

8.1.17 Vector Spaces

A Vector Space V with dimension d over a ring or field A (denoted as $V = A^d$) is the set of all d-tuples $\boldsymbol{x} = (x_1, \ldots, x_d)$ with $x_i \in A$. x_i is called the i-th coordinate of $\boldsymbol{x}$. There is an addition $\oplus : V \times V \to V$, which is performed coordinate wise, making $(V, \oplus)$ an additive group with the neutral element $(0, \ldots, 0)$. In addition to that, there is a scalar multiplication $\otimes : A \times V \to V$, which is defined as $\lambda \otimes (x_1, \ldots, x_d) := (\lambda x_1, \ldots, \lambda x_d)$.

In addition to the operations for groups presented in Section 8.1.2, vector spaces provide the following methods and types:

```
class V
{
   typedef vectorspace_tag algebra_category;

   typedef see text scalar_algebra;
   typedef scalar_algebra::type scalar_type;
   typedef see text scalar_reference;

   const scalar_algebra& getScalarAlgebra() const;

   unsigned dimension() const;

   scalar_type      coord (type,   unsigned) const;
   scalar_reference coord (type &, unsigned) const;

   template<typename I> void toCoord   (type,   I) const;
   template<typename I> void fromCoord (type &, I) const;

   type mul  (type,   scalar_type) const;
   void scale(type &, scalar_type) const;
};
```

`scalar_algebra` defines the type of the algebraic structure A the vector space V is based on. `scalar_algebra::algebra_category` is always a subclass of `ring_tag`. `getScalarAlgebra()` returns an instance of the `scalar_algebra` in use

Coordinates and scalars are of the type `scalar_algebra::type`, which is also available directly as `scalar_type`. It is convenient to use a certain coordinate of a vector as an lvalue. Depending on the implementation of `type`, `scalar_type&` is not always appropriate to accomplish this. Therefore, `scalar_reference` is provided, which allows direct write access to coordinates.

`dimension()` returns the dimensionality of the vector space.

`mul()` and `scale()` provide multiplication with scalars in the obvious way, with `scale(`a`,`λ`)` equivalent to a `= mul(`a`,`λ`)`.

`coord()` is used to access a given coordinate. Two versions are available: the first one used for `type` or `const type &` arguments, returning `scalar_type` directly. The second one, requiring a non-const `type &` argument, returning a `scalar_reference`, which can be written to.

If access to more or all coordinates is required at the same time, `toCoord()` and `fromCoord()` can be used. They copy the coordinates to and from a output / input iterator, respectively.

If A is finite, `element()` enumerates V in the following order: Let $d_0, d_1, \ldots, d_{\dim V - 1}$ denote the $\#A$-adic digits of $0 \leq n < \#V$, i. e.

$$n = \sum_{i=0}^{\dim V - 1} d_i (\#A)^i$$

with $d_i \in \{0, \ldots, \#A - 1\}$, then `element(`$n$`)` returns $(d'_0, d'_1, \ldots, d'_{\dim V - 1})$, where d'_i stands for the d_i-th element in A, returned by A`::element()`.

8.2 Implemented Algebraic Structures

This section describes all algebraic structures which are available in HINTLIB.

All methods described in Section 8.1 of all implementations described in this Section are exercised and tested by the program `test_arithmetic` in the test suit.

8.2.1 Sets of Numbers

HINTLIB contains models for the sets $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$, $\mathbb{C}$, and number fields.

The Euclidean Ring of Integers

The current version of HINTLIB contains only one implementation of the concept of $\mathbb{Z}$, the ring of integers (see Section 8.1.9). It is based on the built-in signed integer types.

```
template<typename T = int>
class IntegerRing
{
public:
   typedef integer_tag algebra_category;
   typedef T type;

   IntegerRing();
};
```

All operations are performed using standard integer arithmetic based on `T` without any checks for overflow or division by zero. Therefore care has to be taken to avoid these situations. Primes are identified and enumerated based on the methods in class `Prime`.

In future versions of HINTLIB we plan to include an implementation of the ring of integers based on the `mpz_t` type of the GMP (GNU Multiple Precision Arithmetic Library).

Class `IntegerRing<int>` is declared in `integerring.h`.

The Field of Rational Numbers

The field of rational numbers, $\mathbb{Q}$, (see Section 8.1.11) is available by applying the template `QuotientField<>` (see Section 8.2.5) to a class implementing the ring of integers. `QuotientField<>` produces the `algebra_category` `rational_field` if the Euclidean ring it is based upon has the `algebra_category` `integer_ring`.

Therefore, creating the field of rational numbers could look like this:

```
IntegerRing<int> integers;
typedef QuotientField<IntegerRing<int> > Rationals;
Rationals rationals (integers);
```

In future versions of HINTLIB we plan to include an implementation of the field of rational numbers based on the `mpq_t` type of the GMP (GNU Multiple Precision Arithmetic Library).

Algebraic Numbers

Number fields $\mathbb{A}$ over $\mathbb{Q}$ can be created using appropriate factor fields (Section 8.2.2) of polynomials (Section 8.2.3) over $\mathbb{Q}$, i. e. using $\mathbb{Q}[x]/(p)$, where p is the defining polynomial of $\mathbb{A}$.

If `Rationals` has algebra category `rational_tag` and `rationals` if of type `Rationals`, (defined, e. g., as described in the previous paragraph) the field $\mathbb{Q}(\sqrt{2})$ can be created using

```
typedef PolynomialRing<Rationals> Polys;
Polys polys (rationals);
Polys::type poly = polys.x(2);
poly[0] = rationals.makeElement(-2);
FactorField<Polys> algebraics (polys, poly);
```

while $\mathbb{Q}(i)$ with $i = \sqrt{-1}$ is produced by an additional

```
Polys::type poly2 = polys.x(2);
poly2[0] = rationals.makeElement(1);
FactorField<Polys> algebraics2 (polys, poly2);
```

The Field of Real Numbers

The current version of HINTLIB contains only one implementation of the concept of the field of real numbers, $\mathbb{R}$, (see Section 8.1.12). It is based on a built-in floating point type, by default `real`.

```
template<typename T = real>
class RealField
{
public:
   typedef real_tag algebra_category;
   typedef Real<T> type;
   typedef ComplexField<T> complex_field;

   RealField();
};
```

The type `Real<T>` which is used as `type` is only a wraper for `T`. Essentially all operations are performed directly based on the built-in operations for `T`.

The main exception is that comparing `Real<>`s is done differently than for `T`: two `Real<>`s compare equal even if the floating point numbers differ slightly. This behavior can be found in `operator==(const Real<T>&, const Real<T>&)`, `is0()` and `is1()`. In addition to that a specialized version of `operator==()` for `Polynomial<Real<T> >` (see Section 8.2.3) is provided in order to allow polynomials of different degree to be compared as equal if the additional coefficients are very small. Without these precautions, many algorithms, e. g. the Euclidean algorithm in $\mathbb{R}[x]$ would not work due to numerical errors.

In future versions of HINTLIB we plan to include an implementation of the field of real numbers based on the `mpf_t` type of the GMP (GNU Multiple Precision Arithmetic Library).

Classes `RealField<real>` and `ComplexField<real>` are declared in `realfield.h`.

The Field of Complex Numbers

The template `ComplexField<T>` provides an implementation of the field of complex numbers based on the `std::complex<T>` type from the C++ standard library.

```
template<typename T = real>
class ComplexField
{
```

```
public:
   typedef complex_tag algebra_category;
   typedef Complex<T> type;
   typedef RealField<T> real_field;

   ComplexField();
};
```

The type `Complex<T>` which is used as `type` is a wraper for `std::complex<T>` for the same reason and with the same extensions as described for `Real<T>` in the previous paragraph.

`order()` uses a very naive algorithm which cannot detect nth roots of unity with $n > 100$.

Classes `RealField<real>` and `ComplexField<real>` are declared in `realfield.h`.

8.2.2 Modular Arithmetic, Factor Rings, and Factor Fields

Given an Euclidean ring R and an element $p \in R$, a new ring (the Factor Ring) $R/pR = R/(p)$ can be constructed, containing the elements of R modulo p. If p is prime in R, it can be shown that the resulting ring is actually a field. Since the compiler cannot determine if p is prime, the user has to specify if a field is expected by using the proper class.

```
template<class R>
class FactorRing
{
public:
   typedef ring_tag algebra_category;
   typedef R::type type;
   typedef R::type unit_type;

   FactorRing (const R& ring, const R::type& p);

   const type& modulus() const;
};

template<class R>
class FactorField
{
public:
   typedef see text algebra_category;
   typedef R::type type;

   FactorField (const R& ring, const R::type& p);

   const type& modulus() const;
};
```

The constructor expects an instance of the Euclidean ring R (which must have an algebra category of either `integer_tag` or `polyoverfield_tag`) and the element $p \in R$. `modulus()` returns the modulus p.

The algebra category of `FactorRing<>` is always `ring_tag` (Section 8.1.5). In the case of `FactorField<>` it is either `cyclic_tag` (Section 8.1.15) if R is $\mathbb{Z}$, or determined based on Table 8.5, if $R = F[x]$, the ring of polynomials over a certain field F.

Implementation Notes

- `FactorRing<`R`>` and `FactorField<`R`>` have R as a protected base class. All methods that are identical in R and `FactorRing/Field<`R`>` are imported from R with a `using`

Coefficient field F	Algebra category of `FactorField<`$F[x]$`>`
$\mathbb{F}_q$	`gf_tag`
$\mathbb{R}$ or $\mathbb{C}$	`realcomplex_tag`
A number field (including $\mathbb{Q}$)	`numberfield_tag`
A function field	`funfield_tag`

Table 8.5: Algebra category of `FactorField<>`s of Polynomials over a field

directive.

- If R is a ring of polynomials, the additive methods (`add()`, `neg()`, `times()`, `additiveOrder()`, `characteristic()`, ...) of R are used directly.
- `element()` enumerates the elements of $R/(p)$ in the following order: $[0], [1], \ldots, [p-1]$ if $R = \mathbb{Z}$; using R`::element()` if R is the ring of polynomials over a finite field; and using `unthreadn()` with $n = \deg p$ if R is the ring of polynomials over an infinite field.
- `FactorField<>::recip()` and `FactorRing<>::unitRecip()` are implemented using the extended Euclidean algorithm.
- `DivisionByZero` is thrown in all applicable situations.
- If the resulting field is finite, `isPrimitiveElement(`a`)` checks whether $a^{(|A|-1)/p} \neq 1$ for all prime divisors p of $|A| - 1$ (Algorithm 1.4.4 in [Coh93]).
- If the resulting field is finite, `order()` uses Algorithm 1.4.3 in [Coh93]. For $R = F[x]$ with F infinite, the algorithm is broken: we do a number of trial multiplications and return 0 if we do not reach 1 during this process.
- `isNilpotent(`a`)` of `FactorRing<>` tests if a is a multiple of the nilradical. The nilradical is calculated by the constructor of `FactorRing<>` using R`::factor()` if $R = \mathbb{Z}$ and R`::squarefreeFactor()` if $R = F[x]$.
- For $R = \mathbb{Z}$, `numUnits()` of `FactorRing<>` returns $\phi(p)$, which is calculated once during the construction of the ring based on R`::factor()`. If R is a polynomial ring over a finite field, R`::squarefreeFactor()` is used instead of R`::factor()`.
- If $R = \mathbb{Z}$, `FactorField<>::additiveOrder(`a`)` uses the formula $p/\gcd(a, p)$.

Specialization for Integer Modular Arithmetic with Small Modulus

Common modular arithmetic over the ring of integers, i. e. $\mathbb{Z}/(p)$, with p not too large could be implemented using `FactorRing<IntegerRing<T> >` and `FactorField<IntegerRing<T> >`. However, an optimized class providing modular arithmetic based on the built-in datatype `unsigned` is provided.

```
template<typename T>
class ModularArithmeticRing
{
public:
   typedef ring_tag algebra_category;
   typedef T type;
   typedef T unit_type;

   ModularArithmeticRing(T p);
```

R::algebra_category	algebra_category of the polynomial ring
ring_tag	ring_tag
domain_tag	domain_tag
ufd_tag	ufd_tag
field_tag and subclasses	euclidean_tag

Table 8.6: algebra_category of polynomial rings

```
   T modulus() const;
};

template<typename T>
class ModularArithmeticField
{
public:
   typedef cyclic_tag algebra_category;
   typedef T type;

   ModularArithmeticField(T p);

   T modulus() const;
};
```

T can either be unsigned char or unsigned short.

Additional implementations for $\mathbb{Z}/(p)$ for p being a very small prime number (not exceeding 256) can be found in Section 8.2.4.

Declarations and Definitions

FactorRing<> and FactorField<> are declared in factorring.h and definitions are available in factorring.tcc. The library contains instantiations for the following types:

```
IntegerRing<int>
PolynomialRing<ModularArithmeticField<unsigned char> >
PolynomialRing<ModularArithmeticField<unsigned short> >
PolynomialRing<QuotientField<IntegerRing<int> > >
PolynomialRing<QuotientField<PolynomialRing<GF2> > >
```

ModularArithmeticRing<> and ModularArithmeticField<> are declared in modulararithmetic.h.

8.2.3 Polynomials

Polynomial rings $R[x]$ over an arbitrary ring R can be constructed using the constructor PolynomialRing<R>(const R& R) of the template class PolynomialRing<>.

The algebra_category of the resulting polynomial ring is determined by R::algebra_category according to Table 8.6. isPrime() and related methods are available for polynomials over $\mathbb{C}$, $\mathbb{R}$, $\mathbb{Q}$ and $\mathbb{F}_q$. factor() is available for polynomials over $\mathbb{C}$, $\mathbb{R}$ and $\mathbb{F}_q$.

Declarations and Definitions

PolynomialRing<> is declared in polynomial.h.

Template definitions are available in the files polynomial.tcc, polynomial_ring.tcc, polynomial_field.tcc, polynomial_rational.tcc, polynomial_real.tcc and polynomial_gf.tcc.

Implementation Notes

- A special class `Polynomial<coeff_type>` is used for `type`. Internally, a `vector<coeff_type>` is used to store the coefficients of a polynomial, with the leading coefficient stored as the first element.
- No method in `PolynomialRing<>` returns a `type` (which would be expensive because it would require at least one additional copy construction and destruction of a `vector` for copying the local result out of the method). Instead, all such methods return stubs which contain just enough information to construct the resulting polynomial wherever it is required.
  `Polynomial<>` has constructors taking these stubs as arguments and creating the required polynomial. Since the original methods in `PolynomialRing<>` are all inline, the optimizer can usually dispose of them and the resulting stub data completely and call the correct constructor of `Polynomial<>` with the proper arguments directly.
- `additiveOrder()` of a polynomial over an arbitrary ring is calculated by taking the least common multiplier of the additive order of all coefficients.
- `unit_type` is equal to `coeff_type` if R is a field, equal to R`::unit_type` if R is a domain, and equal to `type` if R is only a ring.
- A non-zero polynomial over an arbitrary ring is a unit if and only if its constant term is a unit and all other coefficients are nilpotent.
- A polynomial over an arbitrary ring is nilpotent if and only if all its coefficients are nilpotent.
- If the polynomial p is a unit over an arbitrary ring, its reciprocal is found by writing $p = u - b$, with u a unit in R and b a nilpotent polynomial in $R[x]$, and returning $u^{-1}(1 + b + b^2 + b^3 + \cdots)$.
- `isUnit()`, `numUnits()`, and `unitReciprocal()` for polynomial rings over a domain or a field are implemented trivially based on the corresponding operations in R.
- `addTo()` and `subFrom()` are in place, at least when the first operand has a larger degree than the second.
- `sqr()` and `square()` require only about half of the R-multiplications than `mul()` and `mulBy()`. If the characteristic of R is 2, squaring is done in $\mathcal{O}(\deg p)$ steps.
- Optimized multiplication and square routines are used if R does not have zero-divisors, avoiding unnecessary checks for zero coefficients.
- `power()` uses a Left-to-Right binary algorithm (Algorithm 1.2.3 in [Coh93]) to avoid multiplication by polynomials of increasing degree.
- Division is implemented based on [Knu98], 2.6.1, Algorithm D and [Coh93], Algorithm 3.1.1. Optimized versions are provided for `quot()`, `rem()`, `div()`, `isDivisor()`, ...
- `reduce()` is in place. If r aliases a in `div(`a`,`b`,`q`,`r`)`, no temporary is introduced for r.
- `isPrime()` for polynomials over the rational numbers uses Kronecker's algorithm. See [LN97], Exercise 1.30 for details. Well, we know that this is not the fastest way of doing it ...
- `isPrime()` and `factor()` for polynomials over finite fields use Berlekamp's algorithm. See [Knu98], Section 4.6.2, [LN97], Section 4.1, or [Coh93], Section 3.4 for details.

  For `isPrime()` there is no need to calculate the complete factorization. The algorithm only builds the matrix $B - I$ and determines its rank.

- The identification and enumeration of irreducible polynomials over the real and complex numbers is trivial.

- `factor()` for polynomials over real and complex numbers is implemented using a numerical methods based on [Coh93], Algo. 3.6.6 after determining the squarefree factorization.

- Polynomials over finite fields provide an `isPrimitive()` method, which identifies primitive polynomials. See [LN97], Theorem 3.18 and [Knu98], Section 3.2.2 for details.

Polynomials over $\mathbb{F}_2$

For polynomials over $\mathbb{F}_2$ an optimized implementation storing the coefficients in bits of a computer word is available as `Polynomial2Ring<T>`, with `T` being either `u32` or `u64`. This implementation is much faster than the general class `PolynomialRing<>` presented above. All algorithms in `Polynomial2Ring<>` are at least as efficient as the corresponding routine in `PolynomialRing<GF2>`. However, care must be taken to avoid overflow.

This class is declared in `polynomial2.h` and definitions are available in `polynomial2.tcc`. The class is preinstantiated for types `u32` und `u64`.

8.2.4 Finite Fields

A large number of classes implementing finite fields are available in HINTLIB.

Field with Two Elements

An optimized version for the field with two elements, $\mathbb{F}_2$, is available as class `GF2` with constructor `GF2()`. All members (except of `print()`, `printShort()` and `printSuffix()`) are inline and are implemented based on bit or relational operations.

Direct Calculation in Arbitrary Finite Fields

A finite field F with $\#F = p^k$ elements, with p prime, has always the structure of the ring of polynomials over the field $\mathbb{Z}_p$, modulo an irreducible polynomial of degree k in this ring. Therefore it can be realized using `FactorField<PolynomialRing<ModularArithmeticField<`T`> > >` with T denoting an unsigned integer type. However, this is tedious because the user has to calculate an irreducible polynomial of the appropriate degree by hand. The class `GaloisField<>` takes care of the required setup.

```
template<typename T>
class GaloisField
   : public FactorField<PolynomialRing<ModularArithmeticField<T> > >
{
public:
   GaloisField (unsigned base, unsigned exponent);
   GaloisField (unsigned size);
};
```

In addition to all the members inherited from `FactorField<PolynomialRing<ModularArithmeticField<T> > >`, the constructors take care of setting up everything properly, given either the size $\#F = p^k$ of the field or the values p and k separately.

Field Arithmetic Using Table Lookups

If the finite field is small, is it most efficient to calculate the results of all basic operations once and store them in lookup tables. After this is done, all field operations can be performed based on table lookups.

HINTLIB provides three classes for doing this: `LookupGaloisFieldPow2<>` for $p = 2$, `LookupGaloisFieldPrime<>` for $k = 1$, and `LookupGaloisField<>` for the general case. `LookupGaloisField<>` uses tables for addition, additive inverse, doubling, multiplication, multiplicative inverse, squaring, and multiplicative order. The other two classes perform addition based on bit operations and by integer modular arithmetic, respectively. Therefore, they do not need the first three tables.

Each of these classes provides the same two constructors as `GaloisField<>`: one taking p and k as an argument, the other one the size $\#F = p^k$ of the field.

Lookup tables for $b \leq 50$ are precalculated at compile time and linked into the library. Therefore fields with less than 50 elements can be created instantaneously.

When `LookupField`s are copied, a reference counting algorithm is used to avoid actually copying the lookup tables.

8.2.5 Quotient Fields

Every domain R can be embedded in a field. The smallest of these fields is called the quotient field of R, denoted by $\mathrm{Quot}(R)$. It is the set of equivalence classes of pairs (a, b), with $a, b \in R$ and $b \neq 0$, and $[(a, b)] = [(c, d)]$ if and only if $ad = bc$. Based on these equivalence classes, addition can be defined as $[(a, b)]+[(c, d)] := [(ad+bc, bd)]$ and $[(a, b)]\cdot[(c, d)] := [(ac, bd)]$. The additive inverse of $[(a, b)]$ is given by $[(-a, b)]$, the multiplicative inverse by $[(b, a)]$. R is embedded in $\mathrm{Quot}(R)$ by $a \in R \mapsto [(a, 1)]$. An equivalence class $[(a, b)]$ is usually written as quotient a/b, therefore the name quotient ring.

The most important examples for quotient fields are the field of rational numbers, $\mathbb{Q} = \mathrm{Quot}(\mathbb{Z})$, and rational function fields over a field F, which have the form $\mathrm{Quot}(F[x])$. The set of (real valued) rational functions is given by $\mathrm{Quot}(\mathbb{R}[x])$.

If R is a UFD, the quotient a/b can be brought into a canonical form by dividing a and b by $\gcd(a, b)$ and by a unit of R, such that a and b are relatively prime and b assumes the canonical form of in R. However, determining the greatest common divisor of a and b efficiently is only possible if R is an Euclidean ring.

The template class `QuotientField<>` implements a quotient field based on an Euclidean ring R.

```
template<class R>
class QuotientField
{
public:
   typedef R base_algebra;
   typedef typename R::type base_type;

   QuotientField(const R&);

   base_algebra getBaseAlgebra() const;

   type makeElement(const base_type&) const;
   type makeElement(const base_type&, const base_type&) const;
};
```

It is constructed using `QuotientField(const R&` R`)`, where R is an Euclidean ring as defined in Section 8.1.8. The type and an instance of the Euclidean Ring R are available through `base_algebra` and `getBaseAlgebra()`. `base_algebra::type` is available as `base_type`.

`makeElement(`a`)` creates the quotient $[(a, 1)]$, the natural embedding of $a \in R$ in $\mathrm{Quot}(R)$. `makeElement(`a, b`)` returns the element $[(a, b)]$; a and b need not be relatively prime.

The `algebra_category` of a `QuotientField<>` is `ratfunfield_tag` if $R = F[x]$, and `integer_tag` if $R = \mathbb{Z}$.

8.2.6 Vector Spaces

A One-dimensional Vector Space

Given an arbitrary ring R, a one-dimensional vectorspace R^1 over R can be created using the template class `OneDimVectorSpace<`R`>`. All methods (except of `print()` and `printShort()`) are inline forwards to the appropriate methods in R and `type` is equal to R`::type`.

Vector Spaces over Small Finite Fields

If the field F is finite, so is the vectorspace F^n. If $\#F^n$ is sufficiently small, all vector space operations can be performed based on table lookups. The following two classes provide an implementation:

```
template<typename T, typename C>
class LookupVectorSpace
{
public:
   typedef T type;
   typedef C scalar_type;
   typedef LookupField<C> scalar_algebra;

   LookupVectorSpace (const scalar_algebra&, unsigned);
};

template<typename T, typename C>
class LookupVectorSpacePow2
{
public:
   typedef T type;
   typedef C scalar_type;
   typedef LookupFieldPow2<C> scalar_algebra;

   LookupVectorSpacePow2 (const scalar_algebra&, unsigned);
};
```

`LookupVectorSpacePow2<>` can be used if $\#F$ is a power of 2. In this case, all additive methods are implemented based on bit operations. `LookupVectorSpace<>` provides a general implementation which performs addition of vectors as well as multiplication by the scalar using lookup tables.

Both structures are created by passing an instance of the field as well as the dimension to the two-argument constructor.

Chapter 9

Linear Algebra

HINTLIB contains a number of basic linear algebra routines, which occur, e. g., in many algorithms dealing with generator matrices for digital nets.

Linear algebra methods deal with matrices over a field F or a ring R. Throughout this chapter, $\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}$ denote such matrices and n and m give the number of rows and columns, respectively, of an $n \times m$ matrix. Since row vectors play a much more prominent role than column vectors in the theory of generator matrices, $\boldsymbol{a}_i$ with $1 \le i \le n$ will refer to the ith row vector of $\boldsymbol{A}$, and $a_{i,j}$ with $1 \le i \le n, 1 \le j \le m$ denotes the element in row i and column j.

9.1 Three Different Implementations

Linear algebra methods in HINTLIB are available in (up to) three different versions. The first one can be used for arbitrary base rings and fields, while the second and the third one are specializations provided for small finite fields and packed vectors over $\mathbb{F}_2$, respectively.

9.1.1 General Implementation

All linear algebra functions are available as function templates parameterized by the type of an algebraic structure A (see Chapter 8). The first argument to these functions is always a const reference to the algebraic structure.

An $n \times m$-matrix $\boldsymbol{A}$ is stored as an array of nm elements of type `A::type` arranged in the order $a_{1,1}, \ldots, a_{1,m}, a_{2,1}, \ldots, a_{n-1,m}, a_{n,1}, \ldots, a_{n,m}$, i. e. as an array of row vectors. Matrices are passed to functions using pointers to the initial element, optionally followed the number of rows and the number of columns, if these values cannot be deduced from previous arguments.

Linear Algebra with Inexact Types

From a programmers point of view there is nothing wrong with using `RealField` or `ComplexField` as base fields. However, keep in mind that the linear algebra algorithms in HINTLIB were designed with finite fields in mind. Therefore, they do nothing to ensure numerical stability. Solving a small linear system over $\mathbb{R}$ is probably no problem, assuming that the system is not ill conditioned in the first place. For larger systems, however, special purpose implementations should be used.

Declarations and Definitions

All these functions are declared in `linearalgebragen.h`.

HINTLIB contains instantiations for the following types:

```
GF2
ModularArithmeticField<unsigned char>
ModularArithmeticField<unsigned short>
LookupField<unsigned char>
LookupFieldPrime<unsigned char>
LookupFieldPow2<unsigned char>
```

The definitions are available in `linearalgebragen.tcc` and can be instantiated using the macros `HINTLIB_INSTANTIATE_LINEARALGEBRAGEN(`A`)` and `HINTLIB_INSTANTIATE_LINEARALGEBRAGEN_T(`A`::type)` if other types are required.

9.1.2 Finite Fields with Size ≤ 256

The most widely used algebraic structures in HINTLIB are small finite fields. However, depending on the exact size b of the field, four alternative algebraic structures provide the most efficient implementations: `GF2` for $b = 2$, `LookupFieldPow2<unsigned char>` if $b = 2^k$, `LookupFieldPrime<unsigned char>` if b is prime, and `LookupField<unsigned char>` for the general case.

It is often crucial to perform the linear algebra operations as fast as possible (and therefore to use the appropriate algebraic structure). On the other hand, the routine calling linear algebra methods usually has no compile time knowledge whether b is prime or a power of two. To overcome this problem, the class `LinearAlgebra` provides a dispatch mechanism to the four different implementations mentioned above based on virtual function calls.

```
class LinearAlgebra
{
public:
   virtual ~LinearAlgebra() {}

   // member functions, see below

   static LinearAlgebra* make (unsigned b);
};
```

`LinearAlgebra` is an abstract base class with cannot be constructed directly. Instances are obtained by calling the static member `make(`b`)`, which returns a set of linear algebra routines optimized for the specific field size b. Every instance created by `make()` is allocated on the heap and has to be destructed using `delete`.

Since the `type` of all four possible algebraic structures is `unsigned char`, matrices have to be stored as arrays of nm bytes.

Declarations and Definitions

`LinearAlgebra` is declared in `linearalgebra.h`.

9.1.3 Linear Algebra for Packed Matrices over $\mathbb{F}_2$

Linear algebra over $\mathbb{F}_2$ can be done using either one of the methods discussed above. In both cases, however, the matrices are stored as arrays of bytes, where each entry is only allowed to have the values 0 or 1.

A more efficient approach represents the row vectors by computer words, with each bit corresponding to one coefficient. When m is less or equal to the word size, this leads to very efficient algorithms in terms of memory consumption as well as execution time.

A matrix $\boldsymbol{A}$ is passed to these functions by giving a bidirectional iterator to the first row vector and (optionally) to one past the last row vector, optionally followed by the number of columns. The coefficients of each row vector are supposed to reside in the lower order bits.

Packing and Unpacking

Two methods are provided for packing and unpacking a matrix over $\mathbb{F}_2$.

```
template<typename Bi>
void
packMatrix (const unsigned char* A, unsigned m, Bi B_begin, Bi B_end);

template<typename Bi>
void
unpackMatrix (Bi A_begin, Bi A_end, unsigned m, unsigned char* B);
```

Given a pointer to an unpacked $n \times m$ matrix $\boldsymbol{A}$, `packMatrix()` copies the content into the packed matrix $\boldsymbol{B}$.

Given a packed matrix $\boldsymbol{A}$ with n row vectors, `unpackMatrix()` copies the content into the unpacked $n \times m$ matrix $\boldsymbol{B}$.

Declarations and Definitions

All these functions are declared in `linearalgebra2.h`. The definitions are available in `linearalgebra2.tcc` and can be instantiated using the macro `HINTLIB_INSTANTIATE_LINEARALGEBRA2(`T`)`, where T is the type of a bidirectional iterator. Instantiations for `u32*` and `u64*` are included in the library.

9.2 Available Algorithms

The following sections lists all available linear algebra routines.

The section headers are augmented by a "G" if the general template routine and the member of `LinearAlgebra` is available. They are augmented by a "P" if the routine is available for packed matrices over $\mathbb{F}_2$.

The set of linear algebra functions provided by HINTLIB is in no way complete. It only contains those functions that are used by one of the implemented algorithms. Additions will be made when they are required.

9.2.1 Test for Zero Matrix (G, P)

```
template<class A>
bool
isZeroMatrix (const A&,
   const typename A::type* A, unsigned n, unsigned m);

bool
LinearAlgebra::isZeroMatrix (
   const unsigned char* A, unsigned n, unsigned m);

template<typename Bi>
bool
isZeroMatrix (Bi A_begin, Bi A_end);
```

Returns true if $\boldsymbol{A}$ is the zero matrix.

The general routines uses A`::is0()`. The `LinearAlgebra`-member is non-virtual.

9.2.2 Test for Identity Matrix (G, P)

```
template<class A>
bool
isIdentityMatrix (
   const A&, const typename A::type* A, unsigned n);

bool
LinearAlgebra::isIdentityMatrix (
   const unsigned char* A, unsigned n);

template<typename Bi>
bool
isIdentityMatrix (Bi A_begin, Bi A_end);
```

Returns true if the square matrix $\boldsymbol{A}$ is the identity matrix $\boldsymbol{E}$.

The general routines uses A`::is0()` and A`::is1()`. The `LinearAlgebra`-member is non-virtual.

9.2.3 Transpose of a Matrix (—)

```
template<typename T>
void
matrixTranspose (const T* A, unsigned n, unsigned m, T* B);
```

Sets the $m \times n$ matrix $\boldsymbol{B}$ to the transpose of the $n \times m$ matrix $\boldsymbol{A}$.

Since this routines does not make use of any members of the algebraic structure A, no member of `LinearAlgebra` is required.

9.2.4 Matrix Multiplication (G)

```
template<class A>
void
matrixMul (const A&,
   const typename A::type* A, const typename A::type* B,
   unsigned n, unsigned m, unsigned k,
   typename A::type* C);

virtual
void
LinearAlgebra::matrixMul (
   const unsigned char* A, const unsigned char* B,
   unsigned n, unsigned m, unsigned k,
   unsigned char* C);
```

Sets the $n \times k$ matrix $\boldsymbol{C}$ to the product of the $n \times m$ matrix $\boldsymbol{A}$ with the $m \times k$ matrix $\boldsymbol{B}$.

Used methods: A`::addTo()` and A`::mul()`. A straightforward implementation is used, requiring $n(m-1)k$ additions and nmk multiplications.

9.2.5 Multiplication for Square Matrices (G)

```
template<class A>
void
matrixMul (const A&,
   const typename A::type* A, const typename A::type* B,
```

```
   unsigned n,
   typename A::type* C
);

void
LinearAlgebra::matrixMul (
   const unsigned char* A, const unsigned char* B, unsigned n,
   unsigned char* C);
```

Sets the $n \times n$ matrix $\boldsymbol{C}$ to the product of the $n \times n$ matrices $\boldsymbol{A}$ and $\boldsymbol{B}$.

Both methods are inline and forward the call to the multiplication routine for general matrices. The Runtime is $\mathcal{O}(n^3)$.

9.2.6 Multiplication of a Matrix and a Vector (G, P)

```
template<class A>
void
matrixVectorMul (const A&,
   const typename A::type* A, const typename A::type* v,
   unsigned n, unsigned m, typename A::type* w);

virtual
void
LinearAlgebra::matrixVectorMul (
   const unsigned char* A, const unsigned char* v,
   unsigned n, unsigned m, unsigned char* w);

template<typename Bi>
typename std::iterator_traits<Bi>::value_type
matrixVectorMul (
   Bi A_begin, Bi A_end, std::iterator_traits<Bi>::value_type v);
```

Calculates the product of the $n \times m$ matrix $\boldsymbol{A}$ and the m-dimensional column vector $\boldsymbol{v}$. The result is the n-dimensional column vector $\boldsymbol{w} = \boldsymbol{A}\boldsymbol{v}$.

For the general as well as the packed version this operation requires nm steps.

9.2.7 Multiplication of a Vector and a Matrix (G, P)

```
template<class A>
void
vectorMatrixMul (const A&,
   const typename A::type* v, const typename A::type* A,
   unsigned n, unsigned m, typename A::type* w);

virtual
void
LinearAlgebra::matrixVectorMul (
   const unsigned char* v, const unsigned char* A,
   unsigned n, unsigned m, unsigned char* w);

template<typename Bi>
typename std::iterator_traits<Bi>::value_type
matrixVectorMul (
   std::iterator_traits<Bi>::value_type v,
   Bi A_begin, Bi A_end);
```

Calculates the product of the n-dimensional row vector $\boldsymbol{v}$ and the $n \times m$ matrix $\boldsymbol{A}$. The result is the m-dimensional row vector $\boldsymbol{w} = \boldsymbol{v}\boldsymbol{A}$.

While the general version requires mn steps, the version for packed $\mathbb{F}_2$-vectors requires only n steps.

9.2.8 Inverse of a Square Matrix (G)

```
template<class A>
bool
matrixInverse (const A&, typename A::type* A, unsigned n);

virtual
bool
LinearAlgebra::matrixInverse (unsigned char* A, unsigned n);
```

Tries to invert the $n \times n$ matrix $\boldsymbol{A}$. If $\boldsymbol{A}$ is regular, it is replaced by its inverse $\boldsymbol{A}^{-1}$ and true is returned. If $\boldsymbol{A}$ is not regular, false is returned and the content of $\boldsymbol{A}$ is undefined.

The algorithm is based on [PTVF92, Sect. 2.1] (without pivoting) and constructs $\boldsymbol{A}^{-1}$ in place.

9.2.9 Check Linear Independence (G, P)

```
template<class A>
bool
isLinearlyIndependent (const A&,
   typename A::type* A, unsigned n, unsigned m);

virtual
bool
LinearAlgebra::isLinearlyIndependent (
   unsigned char* A, unsigned n, unsigned m);

template<typename Bi>
bool
isLinearlyIndependent (Bi A_begin, Bi A_end);
```

Returns true if the n row vectors of the $n \times m$ matrix $\boldsymbol{A}$ are linearly independent. The matrix $\boldsymbol{A}$ is destroyed during the execution of this function.

If $n = m$, this function can be used to determine whether a square matrix is regular.

9.2.10 Rank of a Matrix (G, P)

```
template<class A>
unsigned
matrixRank (const A&,
   typename A::type* A, unsigned n, unsigned m);

virtual
unsigned
LinearAlgebra::matrixRank (
   unsigned char* A, unsigned n, unsigned m);

template<typename Bi>
unsigned
matrixRank (Bi A_begin, Bi A_end);
```

Determines the rank of the $n \times m$ matrix $\boldsymbol{A}$, i. e. the dimension of the subspace spanned by the row (or column) vectors. The matrix $\boldsymbol{A}$ is destroyed during the execution of this function.

The rank is a number between 0 and $\min\{n, m\}$. It is n if and only if the system of row vectors is linearly independent, and m if and only if the system of column vectors is linearly independent.

9.2.11 Number of Initial Linearly Independent Vectors (G)

```
template<class A>
unsigned
numLinearlyIndependentVectors (const A&,
   typename A::type* A, unsigned n, unsigned m);

virtual
unsigned
LinearAlgebra::numLinearlyIndependentVectors (
   unsigned char* A, unsigned n, unsigned m);
```

Given a $n \times m$ matrix $\boldsymbol{A}$, this function determines the largest integer k, $0 \leq k \leq n$, such that the system $\{\boldsymbol{a}_1, \ldots, \boldsymbol{a}_k\}$ is linearly independent. This number is always less or equal than the rank of $\boldsymbol{A}$.

The matrix $\boldsymbol{A}$ is destroyed during the execution of this function.

9.2.12 Null Space of a Matrix (G, P, P$^\top$)

```
template<class A>
unsigned
nullSpace (const A&,
   typename A::type* A, unsigned n, unsigned m,
   typename A::type* B);

virtual
unsigned
LinearAlgebra::nullSpace (
   unsigned char* A, unsigned n, unsigned m, unsigned char* B);

template<typename Bi>
unsigned
nullSpace (Bi A_begin, Bi A_end, unsigned m, Bi B_begin);

template<typename Bi>
unsigned
nullSpaceT (Bi A_begin, Bi A_end, Bi B_begin);
```

Calculates a basis of the null space (kernel) of the $n \times m$ matrix $\boldsymbol{A}$, i. e. the subspace of all vectors $\boldsymbol{x} \in F^m$ with $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{0}$. If the dimension of the null space is d, the resulting d basis vectors are stored as the first d row vectors of the $m \times m$ (sic!) matrix $\boldsymbol{B}$. The remaining $m - d$ vectors of $\boldsymbol{B}$ remain unchanged. Finally, the number d is returned. The matrix $\boldsymbol{A}$ is destroyed during this process.

For packed matrices over $\mathbb{F}_2$ an additional version `nullSpaceT()` is available, which calculates a basis of the subspace of all vectors $\boldsymbol{x}$ with $\boldsymbol{x}\boldsymbol{A} = \boldsymbol{0}$. In other words, the result is identical to `nullSpace()` applied to $\boldsymbol{A}^\top$ instead of $\boldsymbol{A}$.

The implementation of these routines is based on [Coh93, Algorithm 2.3.1], which is again based on [Knu98, Section 4.6.2, Alg. N].

9.2.13 Basis Supplement (G)

```
template<class A>
unsigned
basisSupplement (const A&,
   typename A::type* A, unsigned n, unsigned m = n);

virtual
unsigned
LinearAlgebra::basisSupplement (
   unsigned char* A, unsigned n, unsigned m);
```

Supplements the row vectors of $\boldsymbol{A}$ to a full basis of F^m.

In a first step this function determines the largest integer k, $0 \leq k \leq n$ such that the system $\{\boldsymbol{a}_1, \ldots, \boldsymbol{a}_k\}$ of row vectors of the $n \times m$ matrix $\boldsymbol{A}$ is linearly independent. This is the same number as determined by `numLinearlyIndependentVectors()`.

In a second step, the vectors $\boldsymbol{a}_{k+1}, \ldots, \boldsymbol{a}_m$ are set such that $\boldsymbol{a}_1, \ldots, \boldsymbol{a}_m$ is a full basis of F^m, i. e. the $m \times m$ matrix $\boldsymbol{A}$ is regular. The number k is returned.

There is a second version of the algorithm available, which determines a set of canonical unit vectors that can be used for supplementing $\boldsymbol{A}$ to a full basis of F^m.

```
template<class A>
unsigned
basisSupplement (const A&,
   typename A::type* A, unsigned n, unsigned m, unsigned* p);

virtual
unsigned
LinearAlgebra::basisSupplement (
   unsigned char* A, unsigned n, unsigned m, unsigned* p);
```

In a first step this function determines the largest integer k, $0 \leq k \leq n$ such that the system $\{\boldsymbol{a}_1, \ldots, \boldsymbol{a}_k\}$ of row vectors of the $n \times m$ matrix $\boldsymbol{A}$ is linearly independent. This step is the same as in the previous case.

In a second step, the indices of the missing canonical unit vectors $(0, \ldots, m-1)$ are stored at `p[`k`]`,...,`p[`$m-1$`]`. Finally, the number k is returned.

This algorithm is based on [Coh93, Algo. 2.3.6].

Appendix A

Installation

HINTLIB is installed using the common GNU installation procedure, which is

```
./configure
make
make install
```

optionally augmented by an additional

```
make check
```

`./configure` determines all properties of the host system that affect building HINTLIB. It also allows the user to specify additional configuration options or to override settings that are determined automatically by `configure` otherwise.

`make` compiles and links the library.

`make install` installs HINTLIB on the system. Usually this is done in /usr/local/, therefore you need root permissions while executing `make install`. If you do not have root permissions or want to install HINTLIB somewhere else for some other reason, see Section A.1.3.

Finally, `make check` runs the test suite. This can be done before or after installing the library with `make install`.

More general information about the GNU build process can be found in the file `INSTALL` in the HINTLIB source directory.

A.1 Configuring HINTLIB

The process of configuring HINTLIB is started by calling `./configure` in the top-level HINTLIB source directory (which is the directory that also contains this manual).

All the options discussed here are to be added to the `configure` command line.

A.1.1 Selecting the Compiler and Compiler Options

`configure` tries to determine the proper way of calling the C++ compiler. If it does not get it right or if you want to select between different compilers installed on your system, you can specify the correct compiler using "`CXX=`*`your_compiler`*".

Compiler options (for instance options for optimization or target specific options) can be added using "`CXXFLAGS=`*`your_options`*".

A.1.2 Selecting the Fortran Compiler

HINTLIB is written in C++ and requires only a C++ compiler to be built. However, the test suite contains a number of Fortran programs which are used to verify the correctness of HINTLIB by comparing it to well-tested software.

Usually, `configure` detects whether a Fortran 77 compiler is available and builds the affected test cases only if this is the case.

If you have a Fortran 77 compiler installed and `configure` does not find it, use "`F77=your_compiler`". If `configure` detects a Fortran 77 compiler even though non is installed, use "`F77=false`" to override the detection and skip the test cases requiring Fortran.

A.1.3 Installation Directory

By default HINTLIB is installed in `/usr/local/`. To be precise, the header files are stored in `/usr/local/include/HIntLib/`, the library in `/usr/local/lib/`, and the precalculated Niederreiter–Xing generator matrices in `/usr/local/share/HIntLib/`.

If you have root access to the system and can run `make install` as root, this default is probably the best solution.

If you want to install HINTLIB somewhere else (for instance in your home directory, because you do not have write access to `/usr/local`), you can use the "`--prefix=...`" option on the `configure` command line. `./configure --help` gives you more information on selecting the installation directory.

A.1.4 The MPI Header File

By default HINTLIB is built as a sequential as well as a parallel library using MPI (see [Mes95]).

If MPI is not installed on your system or if you do not want to build the parallel library, you can disable the MPI version of HINTLIB using "`--disable-MPI`".

If the MPI header file `mpi.h` is not in your default search path for header files, you can use "`MPI_HEADER_PATH=`*`directory_with_mpi.h`*". The path you specify will only be added to the header search path for building the parallel part of the library.

A.1.5 Building Static Libraries

If this is supported by the host system, HINTLIB is built as a shared library by default.

If you want to build a static library in addition to the shared library, add "`--enable-static`" to the `configure` command line.

If you want to build only a static library, add "`--enable-static --disable-shared`" to the `configure` command line.

A.1.6 Building HINTLIB Outside the Source Directory

HINTLIB can be built outside the source directory (given that the make program supports the VPATH variable). This is useful if you want to build HINTLIB multiple times, for instance to accommodate different platforms or different configuration settings.

Change to an empty directory which you want to use for the next build and call `configure` using *`path_to_HIntLib_top_level_source_dir`*`/configure`.

A.1.7 Directory Names with Spaces

The build environment used by HINTLIB (autoconf, automake, libtool) does not support directory names which contain spaces. Do not use such a directory for building HINTLIB.

A.1.8 Cross Compilation

It is not possible to cross compile HINTLIB, because a number of source files are created during the build-process by other C++ programs.

A.1.9 The Size of the Index Data Type

The datatype `HIntLib::Index` is an unsigned integer variable with either 32 or 64 bits precision (see Section 2.3). By default, `Index` has 64 bits if and only if `unsigned long int` has at least 64 bits.

You can force it to be 32 bits (to be precise: to be a typedef for `u32`) by adding "`--with-index=32`" to the `configure` command line, or to 64 bits (`u64`) by adding "`--with-index=64`". If you have a 64-bit processor, you probably want to use the "`--with-index=64`" option.

A.1.10 The Datatype for Real Numbers

By default, the datatype `HIntLib::real` is a typedef for `double` (see Section 2.4). It can be set to `float` or `long double` by adding "`--with-real=float`" or "`--with-real=long`" to the `configure` command line, respectively.

Using `long double` is useful if you need the extra precision. Using `float` could be faster on some systems, however I do not know about any modern system where this is actually the case.

A.2 Comments for Some Specific Architectures

This section contains some information that may be useful for building and installing HINTLIB on certain architectures.

A.2.1 Linux with GNU C++ Compiler

This should run out-of-the-box, for 2.95.x as well as for 3.x versions of the compiler.

- If you want to, you can disable debugging information using "`CXXFLAGS=-O2`", which replaces the "`CXXFLAGS=-g -O2`" default chosen by `configure` for the GNU C++ compiler.

A.2.2 Linux with Intel C++ Compiler

- Select the Intel C++ compiler by using "`CXX=icpc`".

A.2.3 Sun Solaris with GNU C++ Compiler

- Select the GNU C++ compiler by using "`CXX=g++`".
- If you want to, you can disable debugging information using "`CXXFLAGS=-O2`", which replaces the "`CXXFLAGS=-g -O2`" default chosen by `configure` for the GNU C++ compiler.
- If you have a 64-bit processor (Ultra SPARC), add "`--with-index=64`".
- If you want to build a 64-bit ABI version of HINTLIB, you have to use "`CXXFLAGS=-O2 -m64`". However, the only reason for doing so is to link HINTLIB to an executable that has to use the 64-bit ABI for some other reason. There is no advantage for HINTLIB using the 64-bit ABI.

A.2.4 SGI IRIX with Native C++ Compiler

The exception handling code in the C++ library that comes with version 7.4 of the MIPS Pro C++ compiler seems to be broken. You will see frequent crashes with the following error message:

```
Assertion failed in file "../../libC/lang_support/throw.cxx",
line 1618
```

At the moment I don't know about any solution to this problem, except of using version 7.3.x of the compiler which does not exhibit this failure.

- Add "`CXX=CC`" and "`CXXFLAGS=-O2`" to the `configure` command line.
- If you have a 64-bit processor, add "`--with-index=64`".
- If you want to build a 64-bit ABI version of HIntLib, you have to use "`CXXFLAGS=-O2 -64`". However, the only reason for doing so is to link HIntLib to an executable that has to use the 64-bit ABI for some other reason. There is no advantage for HIntLib using the 64-bit ABI.

A.2.5 MS Windows + Cygwin[1] with GNU C++

Cygwin does a pretty good job in emulating a UNIX-like environment on a MS Windows system. HIntLib can be built with version 2.95.x as well as version 3.x of the GCC[2]. When a static library is built, Cygwin behaves exactly like any other UNIX system. For shared libraries (available in the form of DLLs (Dynamic Link Libraries) on a Cygwin system) two potential problems have to be considered:

1. The Windows mechanism for linking an executable to a DLL is less flexible than the one usually used when linking to a static library or to a shared library on a UNIX system: Constant offsets to data members imported from a DLL are not allowed. Therefore, object files that are part of an executable linked against a DLL have to be compiled differently than object files part of an executable linked against a static library.
   This problem is described in more detail on the man-page of the GNU linker `ld`, as part of the discussion of the option `--enable-auto-import`.
2. With versions of Cygwin predating GCC 3.x it is impossible to propagate C++ exceptions from a DLL to the calling program. If an exception is thrown and not caught inside the DLL, the program aborts immediately, without giving the calling program a chance to catch the exception. This is a problem with Cygwin in general and has nothing to do with HIntLib.

With Cygwin, HIntLib can be built as a shared library (i. e. a Windows DLL) and also as a static library. However, the following points should be taken into account.

- If HIntLib is built either only as a DLL or only as a static library, the HIntLib header files are set up such that the proper client code is produced for constant offsets into data members imported from the library.
  If you build HIntLib as a DLL as well as a static library, there is no way for the compiler to know if you are going to link against the DLL or against the static library. By default, linking against the DLL is assumed. If you want to compile an object file which is fit for linking against the static library, the option "`-DHINTLIB_STATIC_LIB_ONLY`" has to be passed to the compiler. Another possible solution would be to use the "`--enable-runtime-pseudo-reloc`" switch of the GNU linker `ld`.

[1] `http://www.cygwin.com/`
[2] With some exceptions (e.g. GCC 3.3.3-3), see the end of this section.

- If you want to be able to propagate exception form HINTLIB to you application, you either have to use a recent version of Cygwin or build a static library using "`--enable-static --disable-shared`".
- To sum up, it can be said that on a Cygwin system HINTLIB should be built either only as a DLL (if you have GCC 3.x) or only as a static library.
- If the build directory is mounted via a network share, it may be possible that an executable or a library file is not available immediately after being written and therefore the following step in the build process, which may require this file, fails. If this happens, simply run `make` again.

The following points apply to any type of HINTLIB on Cygwin, static library as well as DLL:

- If you want to, you can disable debugging information using "`CXXFLAGS=-O2`", which replaces the "`CXXFLAGS=-g -O2`" default chosen by `configure` for the GNU C++ compiler.
- Since there is usually no MPI implementation available on Cygwin, you should use "`--disable-MPI`" to avoid warnings.
- At least GCC 3.3.3-3, which is the default GCC with Cygwin 1.5.12-1, is buggy because it does not produce code for some global constructors/destructors. GCC 3.4.1, which is available in the experimental branch of Cygwin 1.5.12.-1, fixes this problem.

Appendix B

The GNU General Public License

Version 2, June 1991

Copyright © 1989, 1991 Free Software Foundation, Inc.

59 Temple Place - Suite 330, Boston, MA 02111-1307, USA

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

   Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

   You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

   (a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

   (b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

   (c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

   These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

   Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

   In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

   (a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

   (b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

   (c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

   The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

   If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

   If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

   It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

   This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

   Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

Appendix: How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

> one line to give the program's name and a brief idea of what it does.
> Copyright (C) yyyy name of author
>
> This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.
>
> This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.
>
> You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

Also add information on how to contact you by electronic and paper mail.
If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

> Gnomovision version 69, Copyright (C) yyyy name of author
> Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type 'show w'.
> This is free software, and you are welcome to redistribute it under certain conditions; type 'show c' for details.

The hypothetical commands `show w` and `show c` should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than `show w` and `show c`; they could even be mouse-clicks or menu items—whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

> Yoyodyne, Inc., hereby disclaims all copyright interest in the program
> 'Gnomovision' (which makes passes at compilers) written by James Hacker.
> signature of Ty Coon, 1 April 1989
> Ty Coon, President of Vice

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.

List of Tables

Index

Bibliography

[CH94] Ronald Cools and Ann Haegemans, *Algorithm 35: An imbedded family of cubature formulae for n-dimensional product regions*, Journal of Computational and Applied Mathematics **51** (1994), 251–262.

[Coh93] Henri Cohen, *A course in computational algebraic number theory*, Graduate Texts in Mathematics, vol. 138, Springer-Verlag, Berlin, Heidelberg, New York, Tokyo, 1993.

[CR93] Ronald Cools and Philip Rabinowitz, *Monomial cubature rules since "Stroud": A compilation*, Journal of Computational and Applied Mathematics **48** (1993), 309–326.

[Dob70] L. N. Dobrodeev, *Cubature formulas of the seventh order of accuracy for a hypersphere and a hypercubus*, U. S. S. R. Computational Mathematics and Mathematical Physics **10** (1970), 252–253.

[Ewi41] G. M. Ewing, *On approximate cubature*, American Mathematical Monthly **48** (1941), 134–136.

[GM80] Alan C. Genz and A. A. Malik, *Remarks on algorithm 006: An adaptive algorithm for numerical integration over an n-dimensional rectangular region*, Journal of Computational and Applied Mathematics **6** (1980), no. 4, 295–302.

[GM83] ———, *An imbedded family of fully symmetric numerical integration rules*, SIAM Journal on Numerical Analysis **20** (1983), no. 3, 580–588.

[HS58] Preston C. Hammer and Arthur Howard Stroud, *Numerical evaluation of multiple integrals II*, Mathematical Tables and other Aids to Computation **12** (1958), no. 64, 272–280.

[Ion62] D. V. Ionescu, *Generalization of the quadrature formula of N. Obreschkoff for the double integral (in Romanian)*, Acad. R. P. Romine Fil. Cluj Stud. Cerc. Mat. **13** (1962), 35–86.

[Knu98] Donald Ervin Knuth, *Seminumerical algorithms*, third ed., The Art of Computer Programming, vol. 2, Addison-Wesley, Reading, MA, USA, 1998.

[LN97] Rudolf Lidl and Harald Niederreiter, *Finite fields*, second ed., Encyclopedia of Mathematics and its Applications, vol. 20, Cambridge University Press, Cambridge, UK, 1997.

[Mes95] Message Passing Interface Forum, *MPI: A message-passing interface standard*, June 1995.

[MN98] Makoto Matsumoto and Takuji Nishimura, *Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator*, ACM Transactions on Modeling and Computer Simulation **8** (1998), no. 1, 3–30.

[Phi67] G. M. Phillips, *Numerical integration over an n-dimensional rectangular region*, Computer Journal **10** (1967), 297–299.

[PTVF92] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery, *Numerical recipes in C*, second ed., Cambridge University Press, 1992.

[Ste63] Frank Stenger, *Numerical integration in n dimensions*, Master's thesis, University of Alberta, Canada, 1963.

[Str57] Arthur Howard Stroud, *Remarks on the disposition of points in numerical integration formulas*, Mathematical Tables and other Aids to Computation **11** (1957), 257–261.

[Str68] _____, *Extensions of symmetric integration formulas*, Mathematics of Computation **22** (1968), 271–274.

[Str71] _____, *Approximate calculation of multiple integrals*, Prentice-Hall, Englewood Cliffs, NJ, USA, 1971.

[Tha64] H. C. Thacher, *An efficient composite formula for multidimensional quadrature*, Communications of the ACM **7** (1964), 23–25.

[Tyl53] G. W. Tyler, *Numerical integration of functions of several variables*, Canadian Journal of Mathematics **5** (1953), 393–412.