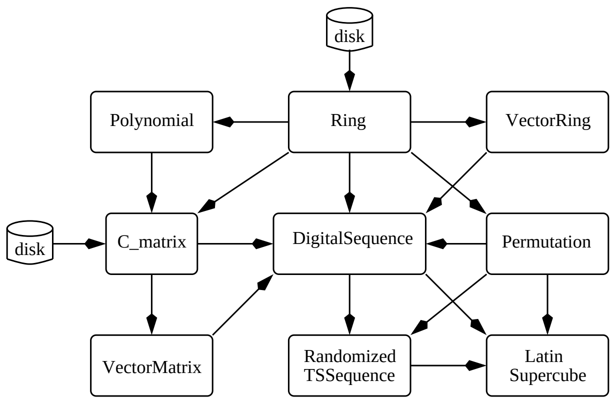

Figure 1: "Direct use of"-hierarchy

In this chapter the author describes the general structure and implementation details of the library `libseq`. The library was written using `C++`. All mathematical structures are represented as `C++` objects. Some of these data structures are defined by using other objects. This is illustrated by figure 1. It can be seen, that for example the object RandomizedTSSequence uses directly objects of the `DigitalSequence` and `Permutation` families, but it doesn't use the object `Ring` directly.

For defining all the sequence generators, a small and specialized computer algebra system was implemented. Most of the complexity is hidden from the users. With a bit experience in `C++` and some knowledge about numerical integration they are able to use the desired point sets.

The code was written using mostly 32-bit integers. At some points in the source code it was not possible to avoid using 64-bit integers. The library was developed under Linux `g++`, Open BSD `g++` and Irix `CC`. With minor changes it should also run on different UNIX-platforms.

0.1 Basic Classes and Types

0.1.1 `UL_int`, `LL_int` etc. (own_types.h)

For portability and strong typing, we introduce several new integer types in this package. The new types are declared in the file `own_types.h`. A short overview of these types provides the next table.

type	definition	purpose
`S_int`	16 bit signed integer	not used
`L_int`	32 bit signed integer	used in `FOR`-loops etc.
`LL_int`	64 bit signed integer	not used
`US_int`	16 bit unsigned integer	used in special situations
`UL_int`	32 bit unsigned integer	standard data type
`ULL_int`	64 bit unsigned integer	used in special situations
`R_Elem`	32 bit unsigned integer	ring elements
`FixPoint`	32 bit unsigned integer	fixed point reals
`RVector`	32 bit unsigned integer	small vectors over ring

0.1.1.1 Notes and Examples

Use the function `check_own_types()` to make sure, that your computer is using integer types of correct size - and change the definitions if necessary.

Example:

```
#include "own_types.h"
main()
{
   UL_int i;               // declare unsigned long integer
   R_Elem e;               // declare element of a ring

   check_own_types();      // general test, if types are OK
}
```

0.1.2 Class `Ring` (Ring.h)

The class `Ring` should hide all internal aspects of an, in our case commutative, ring $(R, +, \cdot)$. Functions for addition ("+"), additive inverse ("$-$") and multiplication ("$\cdot$") are built in as well as more complex ones. You may also find functions for loading and saving user defined rings via ASCII files from disk.

0.1.2.1 Data Structures and Functions

The neutral elements of $(R, +, \cdot)$ are always `(R_Elem)0` and `(R_Elem)1`. It is also assumed, that $R = \{0, 1, 2, \ldots, q-1\}$.

- `Ring(UL_INT` *q*`)`
  Constructs the commutative ring $\mathbb{Z}/q\mathbb{Z}$.

- `Ring(UL_INT p,UL_INT n)`
  Constructs the Galois field $\mathrm{GF}(p^n)$.

- `Ring(CHAR*` *fname*`)`
  Constructs the ring specified by $fname$ by loading it from disk.

- `R_ELEM add(R_ELEM` *a* `, R_ELEM` *b* `)`
  Returns $a +_R b$. Internally this operation is performed by a lookup from array `_add`.

- `R_ELEM mult(R_ELEM a, R_ELEM b)`
  Returns $a \cdot_R b$. This operation is performed by a lookup from array `_mult`.

- `R_ELEM minus(R_ELEM a)`
  Returns $-_R a$. This operation is performed by a lookup from the array `_a_inverse`.

- `R_ELEM invert(R_ELEM a)`
  Returns a^{-1}. If a^{-1} does not exist in R, the element `(R_Elem)0` is returned. The user is responsible for handling this case. The `invert()` function is slow, because the array `_mult` is scanned for the result. In average this takes $O(|R|/2)$ operations to perform. It should be easy to derive a class `Field` from ring if it is necessary to improve this.

- `R_ELEM power(R_ELEM a, L_INT n)`
  Returns a^n, where $n \in \{-2^{31}, \ldots, 2^{31}-1\}$. The algorithm uses fast binary powering. It takes $O(\log n)$ ring operations to compute the result.

- `R_ELEM times(R_ELEM a, L_INT n)`
  Returns $n \cdot a$. Not implemented. Just copy the function `power()` and change `mult()` to `add()` and `invert()` to `minus()`.

- `INT abelian(VOID)`
  Checks if R is an abelian ring. Not implemented.

- `UL_INT memory_used(VOID)`
  Returns an estimate for the main memory occupied by R in bytes.

- `CHAR* query_name(VOID)`
  Returns a pointer to the ring name, if exists.

- `CHAR* query_filename(VOID)`
  Returns a pointer to the file name of the ring, if exists.

- `VOID load(CHAR* fname)`
  Private function. Load ring data from the file specified by $fname$.

- `VOID save(CHAR* fname)`
  Private function. Save ring data to the file specified by $fname$.

With `load()` and `save()` it is possible to store and create special rings on disk. The file format is very simple and described in detail in section 0.10.1. The functions `load()` and `save()` are not intended for general usage! They are for rapid prototyping only. Please don't use these functions if it is not really necessary and try to use the constructor `Ring(char* fname)` instead. For distinction with future additional construction methods, the string `fname` has to start with "FILE:" for specifying a file on disk. (This is not true for `save()`.) The characters following "FILE:" form a normal UNIX-path. Because the author plans to introduce more algebraic objects, for instance groups, the internals of this class and the file format are subject to change in future versions.

0.1.2.2 Class VectorRing (VectorRing.h)

The class `VectorRing` is needed for implementing the optimizations of the generator `DigitalSequence` for small bases as described in section . For the users this object is transparent. Important functions and variables are:

- `VectorRing(RING* R , UL_INT t )`
  Creates the arithmetic of the vector space with t dimensions over R. This is the direct product of t rings.

- `UL_INT base`
  The base b of ring R.

- `UL_INT times`
  Stores the dimension t of the vector space.

- `UL_INT base_pow_times`
  Stores b^t.

- `RVECTOR add(RVECTOR va , RVECTOR vb )`
  Adds two vectors of the vector space.

- `RVECTOR mult(R_ELEM s , RVECTOR va )`
  Multiplies a vector $va \in R^t$ with a scalar $s \in R$.

- `RVECTOR minus(RVECTOR va )`
  Returns the additive inverse of a vector.

- `UL_INT memory_used(VOID)`
  Returns an estimate of the used memory in bytes.

0.1.2.3 Example

```
#include "Ring.h"

UL_int result;
Ring R(7);                              // ℤ/7ℤ
Ring S("FILE:rings/foo_ring.rng"); // loads 'foo_ring.rng'
                                        // from directory 'rings'

result=R.minus(R.add(3,R.mult(2,4))); // −7(3 +7 2 ·7 4)
printf("Result: %d\n",result);
```

0.1.3 Class Polynomial (Polynomial.h)

This class provides a dirty implementation of the basic algebraic type of polynomials $R[X]$ over a given ring R. The operators '+' and '*' are overloaded and some functions will calculate lists of irreducible monic polynomials.

0.1.3.1 Data and Functions

- `Polynomial()`
  Empty constructor. Sometimes internally needed.

- `Polynomial(RING* R)`
  Creates the zero polynomial over R.

- `POLYNOMIAL operator + (POLYNOMIAL&)`
  Adds two polynomial over the same ring R.

- `POLYNOMIAL operator * (POLYNOMIAL&)`
  Multiplies two polynomials over the same ring R.

- `UL_INT deg`
  The degree of the polynomial.

- `R_ELEM digit [MAX_POLY_LEN]`
  The coefficients of the monomials.

- `UL_INT monic_poly_to_ulong(const POLYNOMIAL& a)`
  Extern function. Bijection between monic polynomials and integers for the sieve.

- `POLYNOMIAL ulong_to_monic_poly(const UL_INT n, const RING* R)`
  Extern function. Inverse function to `monic_poly_to_ulong()`.

- `POLYNOMIAL* first_irr_polynomials(UL_INT n, RING* R)`
  Extern function. Computes the first n monic, irreducible polynomials over $R[X]$. This function uses a sieve for finding the irreducible polynomials. This algorithm is guaranteed to work only in case if R is an integral domain[1]. Adjust `TABLE_LENGTH` and `MAX_POLY_LEN` if needed.

- `VOID print_poly(const POLYNOMIAL& P)`
  Extern function. Outputs the polynomial (for debugging etc.) to screen.

- `POLYNOMIAL one(RING* R)`
  Extern function. Returns $1 \in R[X]$.

This class is not very beautiful. It is a standard implementation (not even a very fast one) for dealing with polynomials. These polynomials must have a fixed maximal degree, which can be specified by changing `MAX_POLY_LEN` in the file `options.h`.

The algorithm for construction of monic irreducible polynomials is a brute force method, similar to the construction of primes by the sieve of Eratosthenes.

Everything in class `Polynomial` in this implementation is public. So, every access to the coefficients `digit`, or the degree `deg` has to be direct. This is not a good solution. Please don't use this class excessively and wait for a better solution in a future version of this package! Class `Polynomial` is only used in a few member functions of class `C_matrix`, so these changes should have only a very local impact.

[1] Because R is a finite integral domain, it is also a field.

0.1.4 Class `C_matrix` (C_matrix.h)

For every dimension of digital sequence we have to construct a special generator matrix. (Or for the whole sequence a three dimensional array.) This array contains data, which is different for the sequences introduced by I. M. Sobol, H. Faure or H. Niederreiter. The class `C_matrix` offers not only access to these matrices, but it provides also special methods to generate the data or to load it from a ASCII-file.

0.1.4.1 Data and Functions

- `C_matrix(`RING* R, CHAR* ***name***, UL_INT ***dim***`)`
  Constructor creates a `C_matrix` specified by the method ***name*** in ***dim*** dimensions with ***NumDigits*** chosen big enough to guarantee at least 32 bits of accuracy. Valid choices for ***name*** are "Niederreiter"[2] and "FILE:unix-file". The strings "Sobol" and "Faure" are reserved for future use. Please note that the constructor will not copy the `Ring` R. It will store a reference only. So please don't delete or change R while a `C_matrix` is in use!

- `C_matrix(`RING* R, CHAR* *name*, UL_INT *dim*, UL_INT ***NumDigits***`)`
  The same as the last constructor, only the user has to specify its own ***NumDigits***.

- `C_matrix(`CHAR* ***filename***`)`
  Not implemented in this version. Constructor loads the `C_matrix` specified by ***filename*** from disk. It also tries to create the ring specified in the file.

- UL_INT `memory_used(`VOID`)`
  Functions returns an estimate for the amount of used memory in bytes.

- R_ELEM `query(`INT ***dim***, INT ***row***, INT ***column***`)`
  Queries the element $c^{dim+1}_{row+1,column}$.

- VOID `set(`INT ***dim***, INT ***row***, INT ***column***, R_ELEM ***val***`)`
  Sets the element $c^{dim+1}_{row+1,column}$.

- R_ELEM `operator[](`INT`)`
  Internal direct access of elements in the matrix. A range check is performed.

- VOID `get_Niederreiter(`VOID`)`
  Creates the data for the Niederreiter-`C_matrix`.

- VOID `get_Faure(`VOID`)`
  Not implemented in this version.

- VOID `get_Sobol(`VOID`)`
  Not implemented in this version.

- VOID `load(`CHAR* ***fname***`)`
  Loads the file specified by ***fname*** from disk. As in class `Ring`, `load()` and `save()` are not for general usage. They are also in an early state of testing/verification. So please use these functions carefully!

[2] Please use in this method as ring R only finite fields! (See also class `Polynomial`.)

- VOID `save`(CHAR* *fname*)
  Saves the C_matrix with file name *fname* to disk.

- CHAR* `query_name`(VOID)
  Returns a pointer to the name of the `C_matrix`.

- CHAR* `query_filename`(VOID)
  Returns a pointer to the file name of the `C_matrix`.

0.1.4.2 Class `VectorMatrix` (VectorMatrix.h)

The class `VectorMatrix` is needed for implementing the optimizations of the generator `DigitalSequence` for small bases as described in section . For the users this object is transparent. Important functions and variables are:

- `VectorMatrix`(C_MATRIX* *C*, UL_INT *t*)
  Transfers the entries of C into the vector space R^t.

- RVECTOR `query`(UL_INT *dim*, UL_INT *row*, UL_INT *column*)
  Queries a value.

- VOID `set`(UL_INT *dim*, UL_INT *row*, UL_INT *column*, RVECTOR *val*)
  Sets a value.

- UL_INT *dimension*
  The dimension of C.

- UL_INT *NumDigits*
  A copy of the $NumDigits$ of C.

- UL_INT *times*
  Stores the dimension t of the vector space.

- UL_INT *NumVectors*
  Stores the number of vectors, e.g. $ceiling(NumDigits/t)$.

0.1.4.3 Example

```
#include "Ring.h"
#include "C_matrix.h"

C_matrix *C;
Ring     *R;
UL_int akdim,row,column;

R=new Ring(5);                         // R = Z_{/5Z}
C=new C_matrix(R,"Niederreiter",dim);

for(akdim=0;akdim<dim;akdim++)         // print C to screen
{
   for(row=0;row<C->NumDigits;row++)
   {
      for(column=0;column<C->NumDigits;column++)
```

```
            printf("%3d",C->query(akdim,row,column));
        printf("\n");
    }
    printf("\n");
}

C.save("Niederreiter_matrix.dat"); // write matrix to file
```

0.1.5 `Class Permutation` (Permutation.h)

Bijections are very basic mathematical objects. Because only bijections between a set $D = \{0, 1, \ldots, k-1\}$ to itself are needed, we restrict our attention to permutations. Often these permutations have to be selected randomly, independent and equidistributed over $S_k = S_D$, the symmetric group with $k!$ elements. The class `Permutation` and its subclasses provide several efficient solutions for this problem.

0.1.5.1 `Class RandomPermutation` (Permutation.h)

This class implements random, independent and equidistributed drawn random permutations.

- `RandomPermutation(UL_INT` *b* `)`
  Constructs a permutation from S_b. The initial permutation after construction is the identity. To randomize this use `operator++()`.

0.1.5.2 `Class VectorRandomPermutation` (Permutation.h)

The class `VectorRandomPermutation` is needed for implementing the optimizations of the generator `DigitalSequence` for small bases as described in section . For the users this object is transparent.

- `VectorRandomPermutation(UL_INT` *b* `, UL_INT` *q* `)`
  Constructs q permutations from S_b. These permutations are then lifted to the vector space R^q. The initial permutation after construction is the identity. To randomize this use `operator++()`.

0.1.5.3 `Class LazyRandomPermutation` (Permutation.h)

The class `LazyRandomPermutation` is needed for implementing the optimizations of the generator `RandomizedTSSequence` for small bases as described in section . For the users this object is transparent.

- `LazyRandomPermutation(UL_INT` *b* `)`
  Constructs a lazy random permutation. This class can be interchanged with `RandomPermutation`. The only difference for the user is the time behavior. For computing complete permutations this class is slightly slower than `RandomPermutation`. But for non-surjective functions a speed-up can be expected. The initial permutation after construction is the identity. To randomize this use `operator++()`.

0.1.5.4 Common Functions

- `VOID operator++(VOID)`
  Computes a new permutation. If the class has a `Random` in its name, then the permutation after application of this operation is guaranteed to be random, independent and equidistributed.

- `R_ELEM operator[](UL_INT` i `)`
  Evaluates the permutation at position $0 \leq i < b$.

- `UL_INT Base(VOID)`
  If the permutation is from S_b then this function returns b.

- `UL_INT memory_used(VOID)`
  Returns an estimate for the occupied memory in bytes.

0.1.5.5 Example

Please note, that the class hierarchy of (random) permutations is not perfect and subject to future changes. The behavior, for instance that the first permutation after creation is always the identity, and method names, will remain with high probability.

Example:

```
#include "Permutation.h"

RandomPermutation        RP (10);
LazyRandomPermutation    LRP(10);
VectorRandomPermutation  VRP(10,2);

for(i=0;i<10;i++) printf("<%d>",RP[i]);
printf("\n");

++RP;      // choose a new, random permutation
for(i=0;i<10;i++) printf("<%d>",RP[i]);
printf("\n");
```

0.1.6 Class `Counter` (Counter.h)

This class implements functionality of an $b-$ary counter. Instances of class `Counter` are the heart of classes `DigitalSequence_classic`, `DigitalSequence_-medium_base` and `DigitalSequence_advanced`. This makes it a very sensitive object.

The `Counter` hides the equation

$$\psi_l(d_l(i)) \tag{1}$$

for $0 \leq l < NumDigits = M$ and $0 \leq i < b^M$ from these classes. This formula is the inner part of equation (**??**).

0.1.6.1 Data and Functions

- `Counter(RING* R ,UL_INT NumDigits , RANDOMPERMUTATION** psi )`
  Creates a counter over Ring R of $NumDigits$ digits length. The parameter psi allows the user to specify an array of pointers to random permutations. It is not valid to use `VectorRandomPermutations` here! The array psi should have at least `NumDigits` entries. The counter will then apply the corresponding permutation to each digit.

- `Counter(RING* R ,UL_INT NumDigits )`
  Creates a counter over the ring R of $NumDigits$ digits length. The permutations psi are chosen to be identity.

- `UL_INT query(VOID)`
  Returns the state of the counter. This does not reflect permutations used! In future this function might return `ULL_int`.

- `VOID set(UL_INT n )`
  Sets the state of the counter. This doesn't affect the used permutations. In future this function might have `ULL_int` as parameter. Don't use this function if you are not exactly sure what you are doing!

- `VOID reset(VOID)`
  Resets the counter to the initial state.

- `VOID operator++(VOID)`
  Counts up. This function calls `increment_digit(0)`.

- `VOID operator--(VOID)`
  Counts down. This function calls `decrement_digit(0)`.

- `VOID increment_digit(UL_INT l )`
  This function increments the counter starting with digit l. The digits 0 to $l-1$ are not affected.

- `VOID decrement_digit(UL_INT l )`
  This function decrements the counter starting with digit l. The digits 0 to $l-1$ are not affected.

- `UL_INT operator[](UL_INT l )`
  Queries the l-th digit of the counter, $l=0$ being the least significant.

- `UL_INT difference(UL_INT l )`
  Returns the difference of the l-th digit in the current state and the l-th digit of the previous state. The formula for this is $\psi_l(digit_{current}(l)) -_R \psi_l(digit_{previous}(l))$. Please compare with formula (**??**)! This function is very important for speeding up the digital sequence generators.

- `UL_INT memory_used(VOID)`
  Returns an estimate for the occupied memory in bytes.

- `UL_INT base`
  Variable stores the base b of the counter.

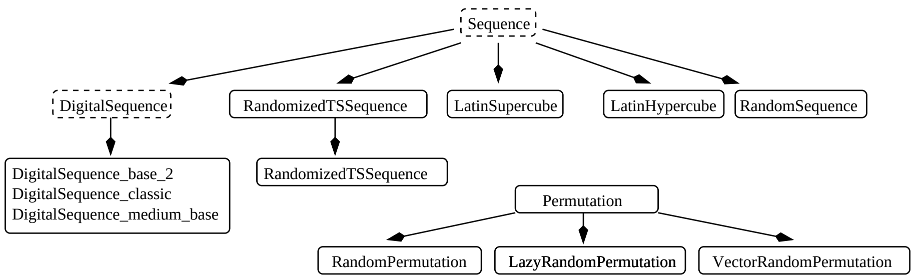

Figure 2: Inheritance of classes in Sequence.h and Permutation.h

- UL_INT *NumDigits*
  The number of digits of the counter.

- INT *LastChangingDigit*
  The most significant digit that changed during the last operation. This information is required for most speedup techniques in class `DigitalSequence`.

0.1.6.2 Notes and Examples

The `Counter` is a very sensitive object! It is used in class `DigitalSequence`, and several implicit assumptions are made. It is possible to change it to a gray code counter or any other counter - and `DigitalSequence_classic` will still work (but `DigitalSequence_advanced` probably not). But for this it is essential, that the variable `LastChangingDigit` and function `difference` are in the correct state.

Example:

```
#include "Ring.h"
#include "Counter.h"

Ring    R(7);
Counter N(&R,NumberOfDigits);

for(i=0;i<NumberOfDigits;i++) printf("<%d>",N[i]);
printf("\n");
// print all digits of the counter

++N;                    // increment counter
N.decrement_digit(2); // decrement third digit

for(i=0;i<N.LastChangingDigit;i++)
   printf("<%d>",N.difference(i));
// prints the difference to the previous state
```

0.2 Class Sequence (Sequence.h)

The goal for the development of this package, was to implement methods for generation of s-dimensional point sets for integration. The class `Sequence` is an abstract superclass for all of the following classes. It provides functionality, which can be used by all instances of its subclasses. We describe this functionality in here in this section, and omit redundant information in future.

0.2.1 Data and Functions

- `DOUBLE operator[](UL_INT dim)`
  Returns X_n^{dim}. The variable n is private and can be manipulated by the functions `operator++()` and `[random_]restart()`.

- `VOID operator++(VOID)`
  Increment n. This switches to the next vector in the sequence. It also sets $j := 0$.

- `DOUBLE get_next_dim(VOID)`
  Returns X_n^j and increments j. This function allows the limited simulation of the behavior of a scalar random number generator like `drand48()`.

- `VOID reset_next_dim(VOID)`
  Sets $j := 0$.

- `VOID restart(VOID)`
  Sets $n := n0$, $j := 0$. The permutations ψ and η remain unchanged. It is not possible to restart all sequences deterministically.

- `VOID random_restart(VOID)`
  Sets $n := n0$, $j := 0$. The sequence is randomized. It is not possible to restart every sequence randomly.

- `UL_INT memory_used(VOID)`
  Returns an estimate for the consumed memory in bytes.

- `UL_INT vector_number(VOID)`
  Returns $n - n0$. In most cases $n0 = 0$. In future versions might return `ULL_int`.

- `CHAR* query_name(VOID)`
  Returns a pointer to the name of the sequence, if exists.

- `CHAR* query_filename(VOID)`
  Returns a pointer to the file name of the sequence, if exists.

0.3 `Class RandomSequence` (RandomSequence.h)

Class `RandomSequence` is one of the most basic examples for a specialization of class `Sequence`. It implements an abstract view to the `drand48()` generator.

- `RandomSequence(UL_INT dim)`
  Constructs a sequence of random vectors. Internally `drand48()` is used.

- `RandomSequence(UL_INT dim, UL_INT len)`
  Constructs a sequence of random vectors. Internally `drand48()` is used. The first len vectors of the sequence are stored in an array to allow a deterministic `restart()`.

- `VOID random_restart(VOID)`
  This function should restart the generator randomly, equidistributed and independent.

- `VOID restart(VOID)`
  In case that the second constructor was used, it is possible to deterministically restart the sequence.

The following example shows how to solve with help of the Monte Carlo method the integral

$$\int_{[0,1)^{dim}} \prod_{i=0}^{dim-1} x_i \, dX.$$

```
#include "RandomSequence.h"
double f(Sequence *X)
{
  int i;
  double tmp=1.0;

  for(i=0;i<X->query_dimension();i++) tmp=tmp*X[i];
  return tmp;
}

RandomSequence X(dim);
double         sum=0.0;
int            N=1000;
for(i=0;i<N;i++)
{
   sum=sum+f(X); // evaluate f at X
   ++(*X);       // next vector in sequence
}
printf("Estimate for integral is %e\n",sum/N);
```

0.4 Class LatinHypercube (LatinHypercube.h)

Class `LatinHypercube` can be used in most cases in exchange for `RandomSequence`. One important difference is, that this sequence has a fixed number of vectors, which has to be specified in the constructor. The points generated are stratified. This reduces the variance in some cases significantly.

- `LatinHypercube(UL_INT dim, UL_INT len)`
  Constructs a Latin hypercube sample of dimension dim and length len.

- `VOID random_restart(VOID)`
  This function restarts with a new random, independent and equidistributed Latin hypercube.

- `VOID restart(VOID)`
  This function is not intended to work in this class.

0.5 Class HaltonSequence (HaltonSequence.h)

This class was originally written by Alexander Keller and adapted for this package by the author. It will generate the low discrepancy sequence introduced by Halton in [?]. No randomizing is offered in this version, so `random_restart()` will not work.

- `HaltonSequence(UL_INT dim, UL_INT n0=0, INT *Primes = NULL)`
  Creates a Halton sequence of dimension dim, starting at $n0$ with the primes in $Primes[dim]$ as bases. The default for $n0 = 0$ and for $Primes$ the first dim prime numbers in $\mathbb{N}$.

- `VOID random_restart(VOID)`
  This function is not intended to work within this class.

- `VOID restart(VOID)`
  This function deterministically restarts the specified Halton sequence.

0.6 Class DigitalSequence (DigitalSequence.h)

This is the base class for several different implementations of speedup techniques for constructing digital (t, s)-sequences over a ring R.

- `DigitalSequence(UL_INT dim)`
  This constructor forwards the dimension dim to class `Sequence`.

0.6.1 Class DigitalSequence_classic (DigitalSequence.h)

This class shows the traditional[3] approach for constructing digital (t, s)-sequences. It is a reference implementation only and is most likely not maintained in future by the author.

- `DigitalSequence_classic(C_MATRIX* C, UL_INT dim, UL_INT n0)`
  Creates a digital $(t, s)-$sequence with the given `C_matrix` C, dimension dim and counter starting at $n0$. Permutations ψ and η are chosen as identity. The dimension dim cannot be larger than the dimension of the `C_matrix`.

- `DigitalSequence_classic(C_MATRIX* C, UL_INT dim)`
  Creates a digital $(t, s)-$sequence with counter starting at zero.

- `DigitalSequence_classic(C_MATRIX* C)`
  Creates a digital $(t, s)-$sequence with dimension assumed the same as in `C_matrix`.

- `VOID random_restart(VOID)`
  Restarts the sequence at $n0$ and chooses new random permutations.

- `VOID restart(VOID)`
  Restarts the sequence deterministically at $n0$.

[3]The word *traditional* is missleading, because the traditional technique is the use of a Gray code counter. *Naive* would describe the technique better, but now the word is coined.

0.6.2 Class DigitalSequence_medium_base (DigitalSequence.h)

This class illustrates one of the buffering techniques which are used in class `DigitalSequence_advanced`. It is a reference implementation only and is most likely not maintained in future by the author.

- `DigitalSequence_medium_base(C_MATRIX* C , UL_INT dim , UL_INT n0 )`
  Creates a digital $(t, s)-$sequence with the given `C_matrix` C, dimension dim and counter starting at $n0$. Permutations ψ and η are chosen as identity. The dimension dim cannot be larger than the dimension of the `C_matrix`.

- `DigitalSequence_medium_base(C_MATRIX* C , UL_INT dim)`
  Creates a digital $(t, s)-$sequence with counter starting at zero.

- `DigitalSequence_medium_base(C_MATRIX* C)`
  Creates a digital $(t, s)-$sequence with dimension assumed the same as in `C_matrix`.

- `VOID random_restart(VOID)`
  Restarts the sequence at $n0$ and chooses new random permutations.

- `VOID restart(VOID)`
  Restarts the sequence deterministically at $n0$.

0.6.3 Class DigitalSequence_advanced (DigitalSequence.h)

This class provides the fastest implementation of this package for long sequences. The running time for short sequences should be similar to the other two techniques. This is the class most users want to use for bases $b \geq 3$.

- `DigitalSequence_advanced(C_MATRIX* C , UL_INT dim , UL_INT n0 )`
  Creates a digital $(t, s)-$sequence with the given `C_matrix` C , dimension dim and counter starting at $n0$. Permutations ψ and η are chosen as identity. The dimension dim cannot be larger than the dimension of the `C_matrix`.

- `DigitalSequence_advanced(C_MATRIX* C , UL_INT dim)`
  Creates a digital $(t, s)-$sequence with counter starting at zero.

- `DigitalSequence_advanced(C_MATRIX* C)`
  Creates a digital $(t, s)-$sequence with dimension assumed the same as in `C_matrix`.

- `VOID random_restart(VOID)`
  Restarts the sequence at $n0$ and chooses new random permutations.

- `VOID restart(VOID)`
  Restarts the sequence deterministically at $n0$.

0.6.4 Class `DigitalSequence_base_2` (DigitalSequence.h)

This class provides an efficient implementation for digital (t, s)-sequence of base $b = 2$.

- `DigitalSequence_base_2(C_MATRIX* C, UL_INT dim, UL_INT n0)`
  Creates a digital $(t, s)-$sequence with the given `C_matrix` C, dimension dim and counter starting at $n0$. Permutations ψ and η are chosen as identity. The dimension dim cannot be larger than the dimension of the `C_matrix`.

- `DigitalSequence_base_2(C_MATRIX* C, UL_INT dim)`
  Creates a digital $(t, s)-$sequence with counter starting at zero.

- `DigitalSequence_base_2(C_MATRIX* C)`
  Creates a digital $(t, s)-$sequence with dimension assumed the same as in `C_matrix`.

- `FIXPOINT query_bitmap(UL_INT dim)`
  Returns the sequence as fixed point reals, e.g. $2^{32}\cdot$`operator[](dim)`.

- `VOID random_restart(VOID)`
  Restarts the sequence at $n0$ and chooses new random permutations.

- `VOID restart(VOID)`
  Restarts the sequence deterministically at $n0$.

- `VOID calc_vector(UL_INT i)`
  Directly computes the vector X_i of the sequence, where $0 \leq i < 2^{32}$. This is slow and should not be used without a good reason.

- `VOID transfer_C(VOID)`
  Private function. The content of the `C_matrix` is copied into an internal format.

0.6.5 Notes and Examples

The constructor will not copy the Ring R nor C_matrix C. It will store a reference only. So please be careful when deleting R or C! No function of `DigitalSequence` will change R or C. So it is possible to reuse the ring and the matrix for further sequences.

Example:

```
#include "Ring.h"
#include "C_matrix.h"
#include "DigitalSequence.h"

Ring     *R;
C_matrix *C;
Sequence *Y;

R=new Ring(5);
C=new C_matrix(R,"Niederreiter",4);
```

```
Y=new DigitalSequence_advanced(C);

++(*Y);

delete(Y);
delete(C);
delete(R);

R=new Ring(2);
C=new C_matrix(R,"Niederreiter",8);
Y=new DigitalSequence_base_2(C);
```

0.7 Class RandomizedTSSequence (RandomizedTSSequence.h)

This class implements the randomization of (t, s)-sequences suggested by Owen as described in section . The input to this class should be a class (t,s)-sequence. But this is not realized here. Internally a digital sequence is created, which is then randomized.

0.7.1 Functions

- `RandomizedTSSequence(C_MATRIX* C, UL_INT dim, UL_INT len, UL_INT LSS_flag)`
  Creates a randomized (t, s)-sequence of dimension dim and length len. For that a `DigitalSequence` over the matrix C is constructed. If the LSS_flag is set, then the order of the vectors in the sequence is randomized. This is helpful for constructing Latin supercube samples. In most cases the users should use a length $len = b^k$ which is a power of the used base.

- `RandomizedTSSequence(C_MATRIX* C, UL_INT dim, UL_INT len)`
  The same as last constructor with $LSS\_flag = 0$.

- `RandomizedTSSequence(C_MATRIX* C, UL_INT len)`
  The same as last constructor with dimension dim taken from C.

- `VOID random_restart(VOID)`
  Creates a random, independent and equidistributed version of the sequence. This function is basically a call to `randomize_sequence()`.

- `VOID restart(VOID)`
  Deterministically restarts the sequence.

- `FIXPOINT** Seq`
  The sequence is stored in $Seq[dim][len]$ as fixed point integers.

- `UL_INT** P`
  The array $P[dim][len]$ is used to store the sorting permutation. In function `apply_permutation_for_latin_supercube()` pointers $Seq[i]$ and $P[j]$

are exchanged and the memory is used to store each others data. Thats why Seq and P must have the same type and size on the binary level.

- `UNSIGNED CHAR** ChangingDigit`
  The array $ChangingDigit[dim][len+1]$ is exactly the $d[i]$ in algorithm ??. As infinity symbol `(UNSIGNED CHAR)0` is used. For the used algorithm $ChangingDigit[i][len] := 0$. This will guarantee to terminate a while-loop.

- `VOID sort_sequence_and_generate_permutation(VOID)`
  Private function. It sorts every $Seq[i]$ to $Seq_{new}[i]$ and creates permutations in $P[i]$ such that $Seq_{old}[i][j] = Seq_{new}[i][P[i][j]]$.

- `VOID fixed_point_to_digits(FIXPOINT fp , UL_INT b , UL_INT* p )`
  Private function. It converts the fixed point number fp to its radix b representation in p. The array p should be at least of length 33. The number of digits of p is stored in $p[0]$. Rounding errors may appear.

- `FIXPOINT digits_to_fixed_point(UL_INT b , UL_INT* p )`
  Private function. It converts the radix b representation of a fixed point number to the 32 bit binary representation. Rounding errors may appear.

- `VOID stochastic_quicksort(UL_INT dim , UL_INT p , UL_INT r )`
  Private function. It sorts Seq and keeps track of P.

- `VOID apply_permutation_for_latin_supercube(VOID)`
  Private function. Randomizes the order of all vectors of the sequence. The same random permutation is applied to all dimensions.

- `VOID calc_changing_digits(VOID)`
  Private function. After sorting the sequence this function will compute the data in $ChangingDigit$.

- `VOID randomize_sequence(VOID)`
  Private function. Does main job for randomizing the sequence.

- `RandomizedTSSequence(UL_INT dim)`
  Don't use this function. It is used by class `RandomizedTSSequence_base_2` and forwards the dimension to class `Sequence`.

0.7.2 Class `RandomizedTSSequence_base_2` (RandomizedTSSequence.h)

This class provides an efficient implementation for randomizing digital (t,s)-sequence of base $b = 2$ as suggested by Owen.

0.7.2.1 Functions

- `RandomizedTSSequence_base_2(C_MATRIX* C , UL_INT dim , UL_INT len , UL_INT LSS_flag )`
  Creates a randomized (t,s)-sequence of dimension dim and length len and base $b = 2$. For that a `DigitalSequence_base_2` over the matrix C is constructed. If the LSS_flag is set, then the order of the vectors in the sequence is randomized. This is helpful for constructing Latin supercube

samples. In most cases the users should use a length $len = 2^k$ which is a power of two.

- `RandomizedTSSequence_base_2(C_MATRIX* C, UL_INT dim, UL_INT len)`
  The same as last constructor with $LSS\_flag = 0$.

- `RandomizedTSSequence_base_2(C_MATRIX* C, UL_INT len)`
  The same as last constructor with dimension dim taken from C.

- `VOID calc_changing_digits(VOID)`
  Private function. This is a version specialized for base $b = 2$.

- `VOID randomize_sequence(VOID)`
  Private function. This is a version specialized for base $b = 2$.

- `VOID random_restart(VOID)`
  Creates a random, independent and equidistributed version of the sequence.

- `VOID restart(VOID)`
  Deterministically restarts the sequence. This function is derived from the base class.

0.7.2.2 Notes and Examples

The constructor will not copy the Ring R nor C_matrix C. It will store a reference only. So please be careful when deleting R or C! No function of `DigitalSequence` will change R or C. So it is possible to reuse them for other sequences.

Example:

```
#include "Ring.h"
#include "C_matrix.h"
#include "RandomizedTSSequence.h"

Ring     *R;
C_matrix *C;
Sequence *X;

R=new Ring(5);
C=new C_matrix(R,"Niederreiter",8,5);
             // 8 dimensions, 5 digits accuracy
X=new RandomizedTSSequence(C,125);
             // first 125 vectors of the sequence
```

0.8 Class `LatinSupercube` (LatinSupercube.h)

This class implements Owen's Latin supercube randomization of randomized (t, s)-sequences as described in section . It is a very simple class, not much more then a convenient macro for using `RandomizedTSSequence`.

0.8.1 Functions

- `LatinSupercube(C_MATRIX* C , UL_INT dim , UL_INT* LD , UL_INT len )`
  Constructs a Latin supercube made of `RandomizedTSSequences`. For that the array LD specifies, how big the sub-dimensions are. It is required, that $\sum_i LD[i] = dim$. There is no variable specifying the size of array LD.

- `LatinSupercube(C_MATRIX* C , UL_INT dim1 , UL_INT dim2 , UL_INT len )`

  Constructs a Latin supercube made of two `RandomizedTSSequences` with dimensions $dim1$ and $dim2$. There are also similar constructors with more sub-dimensions.

- `VOID random_restart(VOID)`
  Creates a random, independent and equidistributed version of the sequence.

- `VOID restart(VOID)`
  Deterministically restarts the sequence.

- `RANDOMIZEDTSSEQUENCE** GenList`
  The array $GenList$ stores a pointer to a `RandomizedTSSequence` generator for every sub-dimension.

- `UL_INT* gen_at_dim`
  The array $gen\_at\_dim[dim]$ stores for every $0 \leq i < dim$ an index to the generator in $GenList$. This indirection is not necessary and might be removed in next version.

- `UL_INT* LocalDim`
  The array $LocalDim[dim]$ stores the local- or sub-dimension $LocalDim[i]$ for the generator at dimension $gen\_at\_dim[i]$.

0.9 Other Files

0.9.1 digit_gen.h

This file hides all calls to the random number generator `drand48()`. This makes it possible to exchange all calls to random number generators easily. It also provides an efficient way to introduce some kind of statistics and count the number of calls.

- `DOUBLE drand47()`
  Returns a random real number $0.0 \leq r < 1.0$. The number r should have at least 32 leading random bits.

- `R_ELEM random_Ring_Element(UL_INT base )`
  Returns a random ring element $0 \leq re < base$.

- `UL_INT get_31_random_bits()`
  Returns an unsigned long integer. This integer consists of 32 bits. The highest bit should be zero and all other bits are random.

- UL_INT get_32_random_bits()
  Returns an unsigned long integer. This integer consists of 32 random bits.
- VOID random_seed_for_drand48()
  Sets the used random number generator to a new state. Here the system time is taken to perform this. This function is not used inside of `libseq`.

0.9.2 options.h

In this file the user will find all options as `#define`s. So all changes can be made on a single point.

- `#define MAX_POLY_LEN` *128*
  The polynomials used have fixed maximal degree. The constant 128 should be safe for all applications.
- `#define TABLE_LENGTH` *1000*
  Defines the size of the table used in the sieve for finding monic irreducible polynomials. This number should be increased if dimension of the digital sequence is large.
- `#define EPSILON` *1e-20*
  Not implemented in this version. Defines a small positive number.
- `#define MAX_BUFFER_LEN` *100*
  Defines the maximum size of the buffers in `DigitalSequence_medium_base` and `DigitalSequence_advanced`.
- `#define USE_DRAND48` *1*
  Not implemented in this version. Specifies, if the random number generator `drand48()` should be used in `digit_gen.h`.
- `#define DEBUG`
  Several runtime checks are made and informations printed to screen id flag is set. Try this flag if you suspect the library or your program is broken.
- `#define RTSS_TEST`
  Additional tests in `RandomizedTSSequence`. Don't use this flag.
- `#define DESTRUCTOR`
  Several destructors will output there names on screen if flag is set.
- `#define RETURN_NO_NULLS`
  Not implemented in this version. If flag is set, no generator will ever return a zero. Instead `EPSILON` is returned.
- `#define WARNINGS_TO_SCREEN`
  Not implemented in this version. If flag is set, the library will output warnings to the screen.
- `#define WARNINGS_TO_FILE`
  Not implemented in this version. If flag is set, the library will output warnings to a file.
- `#define WARNINGS_FILE` *"/tmp/libseq_warning.txt"*
  Not implemented in this version. File name for `WARNINGS_TO_FILE`.

0.10 Implementing additional Digital Sequences

For constructing new digital sequences with different `C_matrix`, it is possible to go several ways. For permanent usage a function similar to `get_Niederreiter()` should be implemented. Whereas for experiments with new methods simply a `*.cmx` file with tools from outside of this library could be created and then loaded into a `C_matrix`.

0.10.1 Format of the `*.rng` and `*.cmx` Files

It is possible to save `Ring` and `C_matrix` files in an ASCII file to disk and load them again. This is useful for experiments with new algebraic objects.

The file format is very simple. Lines with a leading hash # are ignored as well as empty lines. All important variables and data structures of these objects are stored in the files. A BNF for the syntax wasn't defined yet. Nevertheless the syntax is easily understood, as the next subsection shows.

0.10.1.1 Examples

The ring $\mathbb{Z}/5\mathbb{Z}$ is saved as:

```
RING
# [abelian] ring, generated by Ring::save('Ring.rng')
# Ring has 5 elements
# Zero is '0' and One is '1'

NAME Z/qZ
METHOD 0
CARD 5

ADD
0 1 2 3 4 END_LINE
1 2 3 4 0 END_LINE
2 3 4 0 1 END_LINE
3 4 0 1 2 END_LINE
4 0 1 2 3 END_LINE
END_ADD

MULT
0 0 0 0 0 END_LINE
0 1 2 3 4 END_LINE
0 2 4 1 3 END_LINE
0 3 1 4 2 END_LINE
0 4 3 2 1 END_LINE
END_MULT
END_RING
```

The `C_matrix(`$\mathbb{Z}/5\mathbb{Z}$`,"Niederreiter",2,4)` is saved as:

```
C_MATRIX
C_MATRIX_NAME Niederreiter
```

```
C_MATRIX_RING_NAME Z/qZ
C_MATRIX_RING_CARD 5
DIGIT_ACCURACY 4
MAX_DIMENSION 2

# The indices of the following arrays are
# horizontal is l
# vertical is r
# Please compare with the documentation
# or Niederreiter'92

DIMENSION 0
1 0 0 0 END_LINE
0 1 0 0 END_LINE
0 0 1 0 END_LINE
0 0 0 1 END_LINE
END_DIMENSION

DIMENSION 1
1 4 1 4 END_LINE
0 1 3 3 END_LINE
0 0 1 2 END_LINE
0 0 0 1 END_LINE
END_DIMENSION

END_C_MATRIX
```