LaurentPoly Library Manual

version 0.0.1

R. Bernardini (bernardini@uniud.it)

This manual is for LaurentPoly Library.

Table of Contents

1 Overview

Let me briefly introduce you to this library

What this library is for?
: This is `C++` template library for working with multivariate Laurent polynomials with coefficients in a generic commutative ring, or, equivalenty, for working with finetly supported multidimensional signals whose samples assume values in a generic commutative ring.

. . . and this means. . . ?
: This means that you can do sum, differences, products (i.e. convolutions), interpolation, sampling, polyphase decomposition, . . . of multidimensional signals over complexes, reals, rationals, integers, finite fields, and so on. . .

Cool, anything else?
: Yes, there is also a set of functions for working with matrices with entries in a commutative ring. Beside the usual ones (sum, product, inverses, determinantes, submatrices, . . .) there is also some less-than-usual function such the Smith Normal Form for matrices with entries in an Euclidean ring. Do you need to compute the Smith Normal Form of a matrix whose entries are polynomial over a finite field? Well, with this library you can.

So, is this a signal processing library?
: Not really. Actually, you can use it to do signal processing stuff, but the library was not designed with that goal in mind. I wanted a library which was *very general* even if sometimes this had to be paid with some loss of efficiency.

  Do not get me wrong: the library is *not* slow and it can be used to process fairly large signals (such as large images). However, if your goal is to do some *heavy* number crunching with signals whose characteristic (sample field, number of dimensions, . . .) are well-known (so that you do not need the genericity of this library), I guess that this library is not the optimal choice.

Just out of curiosiy, why did you write it?
: I needed to do some computations with matrices with entries in the multivariate Laurent polynomials ring. After a long search for some off-the-shelf software, I decided to write my own library.

Why did you reinvent the wheel? Could not you use (your favorite SW here)?
: Actually, I searched for a while for some software which suited me, but I could not find any. I tried

  - Well-known generic symbolic manipulation software (such as Maple, Mathematica, Symbolic Toolbox, . . .). I guess this is the most obvious choice. Unfortunately, most of symbolic manipulation programs are not free (I wanted to be able to work at home) and, in a sense, they are maybe too powerful since they bring a whole symbolic engine, when I needed something much simpler.

    Do not believe that this is a minor point: if you give to a symbolic program an expression like $(2 + x) * (1 + x)$ you most probably will get the same

expression as an answer. If you want the product done, you must ask it explicetly (with something like `expand((2+x)*(1+x))`).

- Less well-known specialized software such as `Singular`. In this class I found something which was almost suited to my needs, unfortunately there was always some serious drawback, for example a very primitive set of I/O functions (I needed to read/write my signal from/to file).
- Already available `C` or `C++` libraries. Although in this class too there is some nice software, most of the tried libraries can handle only polynomials with positive powers and often only monovariate polynomials.
- and so on. . .

Can you show me an example of usage?

Sure, here it is how to compute the GCD of two polynomials

```
#include <iostream>
#include "signal.H"

using namespace std;

int main()
{
  Signal<double> a, b;
  Signal<double> alpha, beta, GCD;

  cout << "Gimme two polynomials\n";

  cin >> a;
  cin >> b;

  //
  // Find alpha, beta and GCD such that
  // GCD=(gcd of a and b) and GCD=alpha*a+beta*b
  //
  mcd(a, b, alpha, beta, GCD);
  cout << "alpha=" << alpha << "\n";
  cout << "beta=" << beta << "\n";
  cout << "GCD=" << GCD << "\n";
  cout << "alpha*a + beta*b=" << alpha*a + beta*b << endl;
}
```

If instead you want to compute the Smith Normal Form of a matrix whose entries are polynomials over an algebraic extension of the rational numbers obtained by adding to it the golden ratio, you can use

```
#include <iostream>
#include "signal.H"
#include <matrixring/matrixring.H>
#include <matrixring/smith.H>
```

```
using namespace std;
using namespace MRing;

//
// Coefficient field: rational numbers
// extended with the golden ratio.
//
typedef ExtendedField< Fractions<int> > Field;

int main()
{

  Field.set_base("x^2-x-1");  // Initialize the extended field

  Matrix< Signal<Field> > X, Left, Right, SNF;

  cin >> X;
  //
  // Get Left, SNF and Right such that
  //
  //        Left*X*Right = SNF
  //
  smith_normal_form(X, Left, SNF, Right);
  cout << SNF;
}
```

2 How to use this manual

This library can be used at several "levels" of complexity. At the simplest level you can be a `C++` programmer with little or no knowledge of algebraic concepts (such as rings, euclidean domains, Smith Normal Form and so on) which wants to manipulate multivariate polynomials by using the library off-the-shelf. At a greater level of complexity you have some knowledge of abstract algebra and you want to use some special features (e.g. Smith Normal Form). At a still greater level of complexity you have some knowledge of algebra and you want to extend the library by, say, adding a new ring.

In order to accomodate for the different needs of those classes of users, this manual is divided into three parts, each one suited to a specific class of users.

3 Class Signal<Ring>

3.1 Brief description

Class Signal<Ring> is a template class which represents multidimensional signals whose samples belong to a commutative ring Ring[1]. Ring must be a class which implements the *Generic Ring Interface* (see Appendix B [Generic Ring Interface], page 24). Do not be scared by those buzzy names, this library comes with some utility functions which allow you to use the usual C++ builtin types (short int, int, long int, float, double, complex<float> and complex<double>) as Ring.

In order to use the Signal class you must first include the template file with

```
#include<signals/signal.H>
```

Successively you can declare your signals as, for example,

```
Signal<double> x;
Signal< std::complex<float> > y;
Signal<int> z;
```

You can also have signals defined over a finite field

```
#include<rings/galois.H>
Signal< GF<64> > x;
```

3.2 Signal Constructors

`Signal<Ring> zero()`
: Returns the zero signal

`Signal<Ring> one()`
: Returns the delta function (i.e. the neutral element of convolution)

`Signal<Ring>(int x=0, int nd=0)`
: Returns an nd-dimensional signal whose value in the origin is x and is zero otherwise. Note that integer x is converted to an Ring value via the Ring(int) constructor that every ring satisying the *General Ring Interface* (see Appendix B [Generic Ring Interface], page 24) must have. A 0-dimensional signal is just a scalar.

`Signal<Ring>(Ring x, int nd)`
: Returns an nd-dimensional signal whose value in the origin is x and is zero otherwise.

`Signal<Ring>(R x, const Coords &pos)`
: Returns an nd-dimensional signal whose value in pos is x and zero otherwise.

`Signal<Ring> delta(R x, const Coords &pos)`
: Syntactic sugar. Equivalent to Signal<Ring>(R x, const Coords &pos)

[1] If you do not know what a ring is, I will tell you, very briefly, that is a set with a sum and a product which behave as the sum and the product between integers (they are commutative, distributive, ...). Some commonly used commutative rings are: integers, rationals, complex numbers, polynomials, finite fields (used in error correction codes) and so on.

`Signal<Ring> var(unsigned int idx)`
: Returns a signal which represents the `idx`-th variable.

`Signal<Ring>(string s)`
: Parse string `s` and return the corresponding signal, example

```
Signal<double> x("x + 3.2*y - 1.1e3 x^2 y^-1");
```

`Signal<R>(const Support &s)`
: Create a null signal with support `s`. This function can be convenient when the support is known since it avoids reallocations when samples are added.

3.3 Math operators

The library defines the usual operators `+`, `+=`, `-` (both unary and binary), `-=`, `*` (which represents signal convolution), `*=`, `==` and `!=` whose meaning and usage should be clear. Other mathematical functions are

`bool is_zero(const Signal<R> &x)`
: GRI function. Return true if `x` is zero. Semantically equivalent to `x == Signal<R>(0)` it can be more efficient.

`bool is_one(const Signal<R> &x)`
: GRI function. Return true if `x` is the delta sequence. Semantically equivalent to `x == Signal<R>(1)` it can be more efficient.

`bool is_unit(const Signal<R> &x)`
: GRI function. Return true if `x` is a unit (see Appendix H [Glossary], page 33), that is, if `x` has a multiplicative inverse. This is true if and only if `x` is a monomial whose only non-zero value is a unit of `R`.

`bool is_euclidean(const Signal<R> &x)`
: GRI function. Return tre if `x` belongs to an Euclidean ring (see Appendix H [Glossary], page 33). For a signal this is true if and only if `x` is monodimensional.

`bool is_field(const Signal<R> &x)`
: GRI function. Always return `false`.

`bool is_normed(const Signal<R> &x)`
: GRI function. Return true if `Signal<R>` is a normed ring (see Appendix H [Glossary], page 33). This is true if and only if `R` is a normed ring.

`unsigned int deg(const Signal<R> &x)`
: If `x` is one-dimensional, return the degree of `x`; if `x` is not one-dimensional throw exception `RingEx::InvalidOperation` (see Appendix C [Ring Exceptions], page 27).

`Signal<R> inv(Signal<R> b)`
: GRI function. Return, if it exists, the multiplicative inverse of `b`. If the multiplicative inverse of `b` does not exist (i.e., `is_unit(b)` returns `false`) throws exception `RingEx::NonUnitDivision` (see Appendix C [Ring Exceptions], page 27).

`Signal<R> approx_inv(const Signal<R> &x, double tol)`
: If `R` is a normed ring (see Appendix H [Glossary], page 33), compute an "approximate inverse" of `x`, otherwise throw exception `RingEx::InvalidOperation` (see Appendix C [Ring Exceptions], page 27). This function requires that `x` is "close" to a monomial, if it is not it throw exception `RingEx::Unimplemented` (see Appendix C [Ring Exceptions], page 27).

  The meaning of "approximate inverse" is as follows: if `y` is the returned signal and `x_inv` is the true inverse, then `y` is granted to be such that `norm_1(y-x_inv) <= tol`.

`void divmod(const Signal<R> &a, const Signal<R> &b, Signal<R> &quot, Signal<R> &rem)`
: GRI function. If `a` and `b` are one-dimensional signals, return `quot` and `rem` such that

```
a = b*q + r
```

  where `deg(rem) < deg(b)`. If `a` or `b` is not a one Signal,-dimensional signal throw `RingEx::InvalidOperation` (see Appendix C [Ring Exceptions], page 27). If `b` is zero, throw `RingEx::NonUnitDivision`.

`Signal<R> apply(R (fun)(const R&))`
: Return the signal obtained by applying function `fun` to each samples of the signal.

`Signal<R> apply(R (fun)(R))`
: Same as `Signal<R> apply(R (fun)(const R&))`, but now `fun` takes is parameter by value and not by `const`-reference.

`Signal<R> conj(const Signal<R> &b)`
: GRI function. Return the "conjugated" of signal `b`. In this context "conjugated" means the signal obtained by taking the `conj` of each sample of `b` (this is always possible since `conj` is a GRI function) and time-reversing the result. Although this definition of "conjugated" could seem strange, it is the most commonly used type of conjugation.

  If you just need to take the conjugate of each sample of `b`, use

```
b.apply(conj)
```

`void apply_dead_zone(double threshold)`
: If `R` is a normed ring (see Appendix H [Glossary], page 33), replace with zero any sample of `*this` whose `abs` is less than threshold, otherwise throw exception `RingEx::InvalidOperation` (see Appendix C [Ring Exceptions], page 27).

`double norm_1(const Signal<R> &y)`
: Return the 1-norm of `y`, defined as the sum of the `abs` of the samples of `y`. This is defined only if `R` is a normed ring, since in this case the GRI requires the existence of an `abs` function.

`double norm_2(const Signal<R> &y)`
: Return the 2-norm of `y`. The same remarks made for `norm_1` apply.

`double norm_inf(const Signal<R> &y)`
: Return the inf-norm of `y`, that is, the maximum among the `abs` of the samples of `y`. The same remarks made for `norm_1` apply.

`double norm_p(const Signal<R> &y)`
: Return the p-norm of `y`. The same remarks made for `norm_1` apply.

`double abs(const Signal<R> &x)`
: GRI function. Same as `norm_1`.

`template<typename V> Signal<R> eval(unsigned int idx, V val) const`
: Replace each istance of the `idx`-th variable with `val`. Type `V` must be such that there are two operators

  - `V operator*(V, V)`
  - `Signal<R> operator*(V, Signal<R>)`

  For example, to evaluate $x^2 + 3x - 1$ in $x = 4.2$, use

```
Signal<double> foo("x^2+3x-1");
double result=foo.eval(1, 4.2);
```

3.4 Signal processing functions

3.4.1 Signal movement

`Signal<R> time_reverse() const`
: Method. It returns a time-reversed version of the signal.

`Signal<R> translate(const Coords &delta) const`
: Method. It returns the signal translated of `delta`.

3.4.2 Subsampling

`Signal<R> subsample(const Signal<R> &x, const Coords &fact)`
: Function. It returns `x` sampled of a factor `fact[n]` along the `n`-th dimension.

`Signal<R> subsample(const Signal<R> &x, int fact)`
: Function. It returns `x` sampled of a factor `fact` along all the dimensions.

3.4.3 Interpolation

`Signal<R> interpolate(const Signal<R> &x, const Coords &fact)`
: Function. It returns `x` interpolated of a factor `fact[n]` along the `n`-th dimension.

`Signal<R> interpolate(const Signal<R> &x, vector<unsigned int> fact)`
: As `interpolate(const Signal<R> &, const Coords &)`, but accepts a vector of `unsigned int` rather than a `Coords`.

`Signal<R> interpolate(const Signal<R> &x, int fact)`
: Function. It returns `x` interpolated of a factor `fact` along all the dimensions.

3.4.4 Periodic repetition

`Signal<R> periodize(const Signal<R> &x, const vector<unsigned int> &period)`
Function. Return a period of the periodic repetition of `x`, repeated with step `period[n]` along the `n`-th dimension. If `period[n]=0` no periodic repetition is applied along the `n`-th dimension.

`Signal<R> periodize(const Signal<R> &x, const vector<int> &period)`
As `periodize(const Signal<R> &, const vector<unsigned int>&)`, but accepts a vector of `int` rather than a vector of `unsigned int`.

3.4.5 Polyphase transform

`vector< Signal<R> > polyphase(const Signal<R> &x, const Coords &fact, const vector<Coords> &base, bool signal_type)`
Return the polyphase transform of `x` with sampling factor `fact[n]` along the `n`-th dimensionand translations `base[n]`. Use the "signal convention" if `signal_type` is `true`, use the "filter convention" otherwise.

More precisely, the `n`-th element of the returned vector is obtained as follows

1. Translate `x` by
   - `base[n]` if `signal_type` is `true`
   - `-base[n]` if `signal_type` is `false`
2. Subsample the result according to `fact`

`vector< Signal<R> > polyphase(const Signal<R> &x, int fact, const vector<Coords> &base, bool signal_type)`
Same as `polyphase(const Signal<R> x, const Coords &, const vector<Coords> &, bool)`, but uses the same sampling factor `fact` along every dimension.

`vector< Signal<R> > polyphase(const Signal<R> &x, const Coords &fact, bool signal_type)`
Same as `polyphase(const Signal<R> x, const Coords &, const vector<Coords> &, bool)`, but uses a standard set of translations as `base`.

`vector< Signal<R> > polyphase(const Signal<R> &x, int fact, bool signal_type)`
Same as `polyphase(const Signal<R> x, const Coords &, const vector<Coords> &, bool)`, but uses the same sampling factor `fact` along every dimension and a standard set of translations as `base`.

The following functions are in `signals/polyphase_mtx.H`

`void polyphase_mtx(const vector< Signal<Ring> > &filters,const vector<unsigned int> &fact, MRing::Matrix<Signal<Ring> > &pphase, bool analysis=true)`
Compute the polyphase matrix associated with the filter bank whose impulse responses are stored in `filters`. The polyphase matrix is returned in `pphase`. If `analysis` is true, use the convention for analysis filter banks; otherwise the convention for synthesis filter banks. `fact` has the same meaning as in `polyphase`; the used set of translation is the standard one.

3.4.6 Inverse polyphase transform

`Signal<R> polyphase_inv(const vector< Signal<R> > &pphase, const Coords &fact, const vector<Coords> &base, bool signal_type)`

Returns the inverse polyphase transform of the signal vector `pphase` with sampling factor `fact[n]` along the `n`-th dimension and base point `base[m]` for component `pphase[m]`. Use the "signal convention" if `signal_type` is `true`, use the "filter convention" otherwise.

More precisely, the returned signal is computed as follows:

1. Interpolate each signal in `pphase` with factor `fact`
2. Translate the interpolated version of `pphase[n]` by
   - `-base[n]` if `signal_type` is `true`
   - `base[n]` if `signal_type` is `false`
3. Sum the translated versions.

`Signal<R> polyphase_inv(const vector< Signal<R> > &pphase, int fact, const vector<Coords> &base, bool signal_type)`

As `polyphase_inv(const vector< Signal<R> > &, const Coords &, const vector<Coords> &, bool signal_type)`, but use the same sampling factor `fact` along all the dimensions.

`Signal<R> polyphase_inv(const vector< Signal<R> > &pphase, const Coords &fact, bool signal_type)`

As `polyphase_inv(const vector< Signal<R> > &, const Coords &, const vector<Coords> &, bool signal_type)`, but use a standard set of translations as `base`.

`Signal<R> polyphase_inv(const vector< Signal<R> > &pphase, int fact, bool signal_type)`

As `polyphase_inv(const vector< Signal<R> > &, const Coords &, const vector<Coords> &, bool signal_type)`, but use the same sampling factor `fact` along all the dimensions and a standard set of translations as `base`.

The following functions are in `signals/polyphase_mtx.H`

`void polyphase_mtx_inv(const MRing::Matrix< Signal<Ring> > &pphase, const vector<unsigned int> &fact, vector< Signal<Ring> > &filters, bool analysis=true)`

Compute from the polyphase matrix `pphase` of a filter bank the impulse responses associated with each channel. The impulse response of the `n`-th channel is stored in `filters[n]`. If `analysis` is true, use the convention for analysis filter banks; otherwise the convention for synthesis filter banks. `fact` has the same meaning as in `polyphase`; the used set of translation is the standard one.

3.5 Access to sample values

`void put(const Coords &where, R what)`

Set the value of the sample in `where` equal to `what`. If needed, the signal support is automatically extended.

`R get(const Coords &where) const`
: Get the value of the sample in `where`. If `where` is outside the support this function returns `R(0)`. The signal support is unchanged

`R operator[](const Coords &where) const`
: Semantically equivalent to `get(const Coords &)`.

`R& operator[](const Coords &where)`
: Return a **reference** to the sample in `where`. If `where` is outside the signal support, the support is automatically extended. Because of this, this function can be slower than `get` since the latter never does reallocation.

3.6 Support-related functions

3.6.1 Reading support attributes

`unsigned int ndims() const`
: Return the number of dimensions of this signal.

`Support support() const`
: Return the support of this signal as an object of class `Support`.

`vector<int> size() const`
: Return the size of the support of this signal.

`unsigned int size(unsigned int idx) const`
: Return the size of the `idx`-th dimension of the support of this signal.

`Coords start() const`
: Return the "lowest point" of the support of this signal.

`int start(unsigned int idx) const`
: Return the `idx`-th coordinate of the "lowest point" of the support of this signal.

3.6.2 Modifying the support

`bool true_support(Coords &new_start, Coords &new_size) const`
: Return in `new_start` and `new_size` the lowest point and the size of the "minimal support" of the signal, i.e., the smallest multidimensional box which such that every sample outside the box is zero. This function returns `true` if the signal is not null and `false` otherwise.

`bool true_support(Coords &new_start, Coords &new_size, double thr) const`
: Same as `true_support(Coords &, Coords &) const`, but the samples whose norm is less than threshold `thr` are considered equal to zero. In other words, the `Support` identified by `new_start` and `new_size` is the smallest multidimensional box such that every sample outside the box has norm less or equal to `thr`. This function returns `false` if every sample has norm less or equal to `thr` and `true` otherwise.

`void squeeze_support()`
: Force the signal support to be equal to the support returned by `true_support(Coords&, Coords&)`.

`void squeeze_support(double thr)`
: Force the signal support to be equal to the support returned by `true_support(Coords&, Coords&, thr)`.

`void enlarge(int nd)`
: Increase the number of dimensions of this signal to `nd`. If `nd <= ndims()`, this is a `nop`.

`Signal<R> enlarge_ndim(const Signal<R> &x, int ndim)`
: Return signal `x` with the number of dimensions increased to `ndim`.

3.6.3 Iterations

It is possible to iterate over a support by means of the three methods `begin()`, `again()` and `next()` as shown in the following example

```
Support s;
Coords  point;

for(point=s.begin(); s.again(); point=s.next())
   {
      // do something with point...
   }
```

With the code above the *n*-th coordinate runs faster than the *n+1*-th. See [support-iteration], page 21 for details.

`Coords begin() const`
: Start an iteration over the signal support

`bool again() const`
: Check for a new iteration over the signal support

`Coords next() const`
: Go to the next iteration over the signal support

3.7 I/O functions

3.7.1 Low-level

`std::istream& operator>>(std::istream &str, Signal<R> &sig)`
: Read a signal from input stream `str`. The textual form of a signal is a polynomial in *x*, *y*, ... with coefficients in `R`. See Appendix G [Signal Syntax], page 32 for a formal description of the syntax.

`std::ostream& operator<<(std::ostream &str, Signal<R> sig)`

3.7.2 High-level

The following functions read/write signals from/to files in a format which is more convenient than the "polynomial format" used by operators `<<`, `>>`. In order to use these functions the file `<signals/load_signal.H>` must be `#included`.

`Signal<Ring> load_signal_from_datafile(const string &filename)`
: Read a signal from a "datafile." This is quite verbose a format for storing signals and filters. Although is very readable from humans it cannot be read in an

efficient way and it is suggest that it should be used only for "small" signals such as impulse responses. For long signals (such as audio signals or large images) is more convenient to use the "compact format." (See Appendix D [Datafile Signal Format], page 29 for more information about the datafile format).

`Signal<Ring> load_signal_compact_format(const string &filename)`
: Read a signal from `filename`. The signal is stored in "compact format" (see Appendix E [Compact Signal Format], page 30) which can be read more efficiently.

`Signal<Ring> load_signal(const string &filename)`
: Load a signal from `filename`. This function guesses from the first line if the file has the datafile or the compact format and calls the corresponding function.

`void load_filter_bank(const string &filename, vector< Signal<Ring> > &fbank, vector<unsigned int> &sampling, unsigned int &ndim, const string bank_type="analysis")`
: Load a filter bank from `filename`. The filter bank is stored according to the datafile format for filter banks (see Appendix F [Datafile Filterbank Format], page 31). After the call `fbank[n]` will contain the filter associated with the `n`-th channel, `sampling[m]` will be the sampling factor along the `m`-th dimension, `ndim` will be the number of dimensions of the filters (note that it will always be `sampling.size() == ndim`.

`void save_signal(const Signal<Ring> &signal, const string &filename, bool compact=true)`
: Save signal `signal` in file `filename` using the compact format if `compact` is `true`, the datafile format otherwise. Currently, only the compact format is implemented.

4 Matrix functions

4.1 Constructors

`Matrix()` Default constructur. Returns the empty matrix

`Matrix(const Matrix<R> &A)`
Copy constructor

`Matrix(size_type nrow, size_type ncol, const R& value = R())`
Constructor for constant matrices: all the entries are equal to `value`

`Matrix(size_type nrow, size_type ncol, const R* v)`
Constructor from array. The entries of `v` are used to initialize the matrix row-wise.

```
double v[]={1,2,3,4,5,6,7,8,9};

                          1 2 3
Matrix foo(3, 3, v); ⇒    4 5 6
                          7 8 9
```

`Matrix(size_type nrow, size_type ncol, const char *s)`
Constructor from strings. The matrix is read row-wise from the string used as an input stream.

```
                                                  1 2 3
Matrix foo(3, 3, "1 2 3 4 5 6 7 8 9"); ⇒          4 5 6
                                                  7 8 9
```

`eye(size_type M, size_type N=0, const Ring& val=Ring(1))`
Constructor for the identity matrix. `eye(M)`returns an `MxM` identity matrix. `eye(M, N)` returns an `MxN` matrix which is different from zero (and equal to `val`) only on the main diagonal.

`zeros(size_type M, size_type N=0)`
Constructor for the null matrix. `zeros(M)`returns an `MxM` null matrix. `zeros(M, N)` returns an `MxN` null matrix.

`ones(size_type M, size_type N=0)`
Constructor for the "all ones" matrix. `zeros(M, N)` returns an `MxN` matrix with each entries equal to one. By default `N==M`.

4.2 Shape-related functions

`Matrix<T>& newsize(size_type M, size_type N)`
Change the number of rows and columns equal to `M` and `N`. Any previous content is lost.

`Matrix<T>& reshape(size_type M, size_type N)`
Matlab-like reshape. Force the matrix dimensions to `M` rows and `N` columns, but preserve the matrix entries (like the `reshape` function of Matlab).

`size_type dim(size_type d) const`
: `A.dim(1)` returns the number of rows of `A`; `A.dim(2)` returns the number of columns.

`size_type size() const`
: Return the total number of entries in the matrix, i.e., the number of rows multiplied by the number of columns.

`size_type num_rows() const`
: `A.num_rows()` returns the number of rows of `A`.

`size_type num_cols() const`
: `A.num_cols()` returns the number of columns of `A`.

4.3 Access functions

This class has two different way to access matrix elements: with a FORTRAN-like syntax `A(r,c)`, where `r` and `c` start from 1, and with a C-like syntax `A[i][j]` where `i` and `j` start from 0.

`T &operator()(unsigned int r, unsigned int c)`
: `A(r,c)` returns a reference to the element of row `r` and column `c`. **Please note** that indexes `r` and `c` start from 1 (*a-la* Matlab, FORTRANT, ...) and **not** from 0.

`const T &operator() (unsigned int i, unsigned int j) const`
: As `operator()`, but returns a `const` reference.

`operator T**(), operator T**() const`
: Convert a `Matrix<T>` to a pointer-to-pointer-to-`T`. This allows to access matrix elements with a `C`-like syntax, i.e., `A[i][j]` (where now `i` and `j` start from 0).

4.4 Submatrix functions

`Matrix<T> operator()(Range i, Range j) const`
: `A(range_r, range_c)` returns the submatrix obtained by taking the elements of `A` with rows in `range_r` and columns in `range_c`.

`void rewrite(Range i, Range j, Matrix<T> x)`
: `A.rewrite(range_r, range_c, x)` assign the elements of `x` to the submatrix obtained by taking the elements of `A` with rows in `range_r` and columns in `range_c`. This is equivalent to `Matlab` code

```
A(range_r, range_c) = x;
```

`Matrix<T> submatrix(Matrix <T> &A, vector<int> rows, vector<int>cols, bool do_check=true)`
: Extract a submatrix

`Matrix<T> submatrix(Matrix <T> &A, int first_row, int last_row, int first_col, int last_col)`
: Extract a submatrix

4.5 Math functions

The library has the most common operators whose meaning should be clear.

```
Matrix<T>& operator=(const Matrix<T> &A)
bool operator==(const Matrix<T> &b) const
bool operator!=(const Matrix<T> &b) const
Matrix<T> operator+(const Matrix<T> &A, const Matrix<T> &B)
Matrix<T> operator+(const Matrix<T> &A, const T &B)
Matrix<T> operator+=(const Matrix<T> &A, const Matrix<T> &B)
Matrix<T> operator+=(const Matrix<T> &A, const T &B)
Matrix<T> operator-(const Matrix<T> &A, const Matrix<T> &B)
Matrix<T> operator-(const Matrix<T> &A, const &B)
Matrix<T> operator-=(const Matrix<T> &A, const Matrix<T> &B)
Matrix<T> operator-=(const Matrix<T> &A, const &B)
Matrix<T> operator-(const Matrix<T> &A)
Matrix<T> operator*(const Matrix<T> &A, const Matrix<T> &B)
Matrix<T> operator*(const Matrix<T> &A, const T B)
Matrix<T> operator*(const T B, const Matrix<T> &A)
```

Other operators are:

`Matrix<T> apply(Matrix<T> &A, T fun(T))`
: Return the matrix obtained by applying function `fun` to every element of `A`

`Matrix<T> mult_element(const Matrix<T> &A, const Matrix<T> &B)`
: Multiply `A` and `B` element-wise. This is equivalent to `Matlab` operator `.*`.

`Matrix<T> transpose(const Matrix<T> &A)`
: Returns the matrix whose `(i,j)`-th element is the `conj`-ugated of the `(j,i)`-th element of `A` (note that `conj` is a GRI function (see Appendix B [Generic Ring Interface], page 24)). This is equivalent to `Matlab` operator `'`. If you want to take the transpose without the `conj`-ugation, use `pure_transpose`.

`Matrix<T> pure_transpose(const Matrix<T> &A)`
: Returns the matrix whose `(i,j)`-th element is the `(j,i)`-th element of `A` . This is equivalent to `Matlab` operator `.'`. If you want the equivalent of `'` (transpose and `conj`-ugation) use `transpose`.

`Matrix<T> conj(const Matrix<T> &A)`
: Returns the matrix obtained by `conj`-ugating the elements of `A`. Note that `conj` is a GRI function (see Appendix B [Generic Ring Interface], page 24)

`Matrix<T> matmult(const Matrix<T> &A, const Matrix<T> &B)`
: Matrix product with function syntax (equivalent to `operator*`).

`int matmult(Matrix<T>& C, const Matrix<T> &A, const Matrix<T> &B)`
: Matrix product with three-argument function syntax. Matrix `C` is equal to `A*B`.

`Matrix<T> inv(Matrix<T> x)`
: Return the inverse of `x`. Throw `RingEx::InvalidOperation` if `x` is not square, and `RingEx::NonUnitDivision` if `x` is not invertible.

`T det(Matrix<T> x)`
: Return the determinant of `x`. Throw `RingEx::InvalidOperation` if `x` is not square.

4.6 I/O functions

`istream& operator>>(istream &s, Matrix<T> &A)`
: Read matrix `A` from stream `s`. It reads from `s`:

1. the number of rows of `A` (an integer)
2. the number of columns of `A` (an integer)
3. the elements of `A` row-wise

For example, if `s` contains

```
2 3  10 11 12 13 14 15
```

after `s >> A`, `A` will be matrix

```
10 11 12
13 14 15
```

`ostream& operator<<(ostream &s, const Matrix<T> &A)`
: Write matrix `A` on stream `s`.

4.7 Ranges of indices

4.7.1 Overview

Objects of class `Range` represent "MATLAB-like" ranges as `a:b:c` and vectors of indices. Range extremes can be integer values or expressions with term `Range::end()`. For example,

```
Range(2, Range::end()-3)
```

is a range which represents the MATLAB range `2:end-3` and

```
A = B(Range(1,3), Range(Range::end()-3, Range::end()));
```

assigns to `A` the upper-right 3x3 submatrix of `B`, while

```
A = B(Range(1, 2, Range::end()), Range::all());
```

assigns to `A` the submatrix of `B` made of the odd rows of `B`.

What can I do with a `Range`?

- Create it
- Set its "end" value
- Ask for its size (i.e. the number of indexes in it) with size()
- Loop over its entries
- Ask for the min/max value

4.7.2 Constructors

```
Range(RangeEntry n)
Range(int n)
Range(RangeEntry from, RangeEntry to)
Range(RangeEntry from, int delta, RangeEntry to)
Range(vector<idx_t> idx)
static inline Range all()
static inline RangeEntry end()
```

4.7.3 Iteration

It is possible to iterate over the entries of a `Range` with a code like

```
Range r(...);  // Initialize the range

for(unsigned int n=r.begin(); r.again(); n=r.next())
  {
     // do something...
  }
```

Since the value of "`end`" depends on the context where the range is used, the "`end`" value must be set by calling `set_end()` before using the `Range` in a loop, for example the following code

```
Range r(1, 2, Range::end()-3);  // Initialize the range

r.set_end(11)
for(unsigned int n=r.begin(); r.again(); n=r.next())
  {
     cout << r << endl;
  }
```

will print the numbers 1, 3, 5 and 7.

```
idx_t begin();
bool again();
idx_t next();
```

4.7.4 Other functions

`void set_end(int n)`
: Set the "end" of this range

`void ubound(idx_t n)`
: Syntactic sugar. Equivalent to `set_end()`

`idx_t size()`
: Number of entries in this range

`idx_t min()`
: Smallest index in this range

`idx_t max()`
: Largest index in this range

5 Builtin rings

5.1 Introduction

The library comes with some builtin rings which can be used as signal coefficients and matrix entries. Note that library has also the GRI functions (see Appendix B [Generic Ring Interface], page 24) which are necessary to use the following `C++` rings as signal coefficients and matrix entries

- `int`
- `long int`
- `float`
- `double`
- `std::complex<float>`
- `std::complex<double>`

The library has the following builtin rings

Finite fields
: There are three different implementations of Galois field (in order of increasing complexity and flexibility)

  `GF2<size>`
  : Template class for fields of size 2^n

  `GF<size>` Template class for fields of size p^n

  `Galois` Non-template class for fields of size p^n

Derivate rings
: The library has also some template classes which can be used to derive new rings from old ones

  `Fraction<Ring>`
  : Fraction field of ring `Ring`

  `Extended_Field<Field>`
  : Algebric extension of field `Field`

For example, one can define a matrix whose entries are polynomials whose coefficients are numbers of type $q + sqrt(2)r$, where q and r are rational, by using

```
typedef Extended_Field< Fraction<int> > Field;

Field.set_primitive("x^2-2");

Matrix< Signal <Field> > mtx;
```

As another example, the following code

```
Matrix< Fraction< Signal<double> > > mtx;
```

defines `mtx` as a matrix of rational functions.

6 Advanced Usage

Appendix A Other classes

The library has several auxiliary classes which are internally used by the class Signal. Although the user usually does not interact with those classes, sometimes it can happen.

A.1 Support class

A.1.1 Overview

The support of a signal is a (maybe multidimensional) "rectangle". It is represented by its "lowest" corner (i.e. the point x of the support such that $x_n \leq y_n$ for every point y) and the length of its edges.

Operations which can be done on a support

- Ask if a point is inside the support
- Ask for the lowest corner and the edge lengths
- Compute the union of two supports
- Iterate over the support points

A.1.2 Iteration over a support

It is possible to iterate over a support by means of the three methods `begin()`, `again()` and `next()` as shown in the following example

```
Support s;
Coords  point;

for(point=s.begin(); s.again(); point=s.next())
   {
      // do something with point...
   }
```

With the code above the *n*-th coordinate runs faster than the *n+1*-th. For example, with a support (0..2) x (-1..1) x (1..3) 'point' assumes the values

```
0 -1 1
1 -1 1
2 -1 1

0  0 1
1  0 1
2  0 1

0  1 1
1  1 1
2  1 1

0 -1 2
1 -1 2
2 -1 2
```

```
0  0 2
```

...and so on...

A.1.3 List of methods

A.1.3.1 Constructors

`Support()`
: Default constructor. Create an empty support.

`Support(const Coords &st, const vector<unsigned int> &sz)`
: Create a support whose lowest point is `st` and whose `n`-th side has size `sz[n]`.

`Support(const Coords &st, const vector<int> &sz)`
: As `Support(const Coords &, const vector<unsigned int> &)`, but the size vector is a vector of `int` rather than `unsigned int`.

`Support(const vector<int> &st, const vector<unsigned int> &sz)`
: As `Support(const Coords &, const vector<unsigned int> &)`, but the starting point is specified by a vector of `int`.

`Support(const vector<int> &st, const vector<int> &sz)`
: As `Support(const Coords &, const vector<unsigned int> &)`, but the size vector is a vector of `int` rather than `unsigned int` and the starting point is specified by a vector of `int`.

`Support(const Coords &st)`
: Create a support which contains only `st`

`Support(const vector<int> &st)`
: Create a support which contains only `st`

A.1.3.2 Membership functions

```
bool is_inside(Coords x) const
bool contains(const Support &x)
```

A.1.3.3 Support extension

```
int start(int n) const
Coords start() const
unsigned int size(int n) const
vector<unsigned int> size() const
int end(int n) const
```

A.1.3.4 Support operations

`Support common_support(Support a, Support b)`
: Returns the smallest support which includes a and b

`Support common_support(Support a, Coords b)`
: Returns the smallest support which includes a and b

`Support operator+(Support b) const`
: Returns the sum of two supports

`Support operator-() const`
: Returns the "time-reversed" version of this support.

A.2 Coords class

A.2.1 Overview

Class `Coords` represents the signal indexes. What can we do with a `Coords`?

- Construct one
- Add/subtract
- Scalar multiply
- Read/Write a single coordinate

A.2.2 Constructors

`Coords(unsigned int nd=0, int x=0)`
: Returns a point on the `nd`-dimensional diagonal (usually the origin)

`Coords(const vector<int> &x)`
: Returns the point whose coordinates are in `x`

`Coords(const vector<unsigned int> &x);`
: Returns the point whose coordinates are in `x`

Appendix B Generic Ring Interface

In order to allow to write code which works for generic rings, this library introduce the *Generic Ring Interface* (GRI) which describes which operator, functions and methods a class must have in order to be used as a `Ring` in `Signal<Ring>`.

Sometimes a ring can have some additional structure which allows for some operations which are not possible with other rings. For example, an Eucleadean ring (such as monovariate polynomials over a field) has a "degree" function which allows for an efficient computation of the greatest common divisor of two polynomials.

In order to accomodate for all the possible cases, the GRI has a "basic" set of functions (which every ring must have) and other "optional" sets of functions which can be implemented or not, depending on the additional structure of the rings. For example, if the ring is an Euclidean one, the GRI prescrives that functions `deg()` and `divmod()` must be present.

It is suggested that if for a ring an optional methods does not make sense (for example, `deg()` for a non-Euclidean ring) the method should throw `RingEx::InvalidOperation` (see Appendix C [Ring Exceptions], page 27).

B.1 Basic GRI

B.1.1 Constructors

There must be present at least one constructor

`Ring::Ring(int n)`

- If `n==0` returns the "zero" of `Ring`
- if `n==1` returns the "one" of `Ring`
- if `n==-1` returns the opposite of the "one" of `Ring`
- if `n>1` returns the "one" of `Ring` added to itself `n` times
- if `n<-1` returns the opposite of the "one" of `Ring` added to itself `n` times

B.1.2 Ring Structure Functions

The following functions return information about the ring structure. Note that the ring element given as parameter is usually not used, its presence is due to the necessity of overloading the function names.

`bool is_euclidean(Ring)`

Return true if the ring is an Euclidean one (see Appendix H [Glossary], page 33). If `is_euclidean` returns true, the function described in the Eusclidean subset of the GRI must be implemented.

`bool is_field(Ring)`

Return true if the ring is a field, i.e., every non-null element has a multiplicative inveres (see Appendix H [Glossary], page 33)

`bool is_normed(Ring)`

Return true if the ring is normed (see Appendix H [Glossary], page 33). If `is_euclidean` returns true, the function described in the Normed subset of the GRI must be implemented.

B.1.3 Operators

All the operators `+`, `-` (both binary and unary), `*`, `+=`, `-=`, `*=`, `==` and `!=` must be implemented. The meaning of each operator is the usual one and the usal ring properties must hold (e.g., `*` distributes with respect to `+`, unary `-` gives the additive inverse, `a-b` is equivalent to `a + (-b)`, and so on...)

Other functions:

`Ring inv(Ring x)`
: Return the multiplicative inverse of `x` if it exists, throw `RingEx::NonUnitDivision` otherwise.

`Ring conj(Ring)`
: Return the "coniugated" of `x`. The meaning of "coniugated" is left to the implementation, but this function must satisfy

```
conj(a+b) == conj(a)+conj(b)
conj(a*b) == conj(a)*conj(b)
conj(conj(a)) == a
```

B.1.4 Boolean functions

`bool is_one(Ring x)`
: Return true if `x` is one. This function is semantically equivalent to `x == Ring(1)`, but it can be more efficient.

`bool is_zero(Ring x)`
: Return true if `x` is zero. This function is semantically equivalent to `x == Ring(0)`, but it can be more efficient.

`bool is_unit(Ring x)`
: Return true if `x` has a multiplicative inverse.

B.1.5 I/O functions

`friend ostream& operator<<(ostream &str, Ring x)`
: Write a textual representation of `x` to the stream `str`

`friend istream& operator>>(istream &str, Ring &x)`
: Read `x` from the stream `str`

B.1.6 Other functions

`void regexp(string &re, int &idx, const Ring&)`
: Returns in `re` a regular expression describing the textual form of an element of `Ring`. In `idx` is returned the number of the sub-expression which describe the part to be parsed. Use `idx=0` if all the matched string is to be parsed. Note that the third parameter is usually ignored, but it is necessary in order to overload this function.

  Example:

```
void regexp(string &re, int &idx, const double&)
{
```

```
    static string double_re =
      "([+-])? *[0-9]+(\\.[0-9]*)?( *[eE] *[-+]? *[0-9]+)?";
    re = double_re;
    idx = 0;
  }
```

`string2val???`

B.2 GRI for Euclidean rings

If `is_euclidean(Ring)` returns `true` the following two functions must also be implemented

`unsigned int deg(Ring x)`
: Returns the "degree" of `x`. Function `deg()` must be such that for every `a` and `b != 0` in `Ring` there exist `quot` and `rem` such that

```
a = b*quot + rem
```

and `deg(rem) < deg(b)`.

`void divmod(const Ring &a, const Ring &b, Ring &quot, Ring &rem)`
: If `b != 0`, returns `quot` and `rem` such that `a = b*quot + rem` and `deg(rem) < deg(b)`; if `b == 0` throws `RingEx::NonUnitDivision`.

B.3 GRI for Normed rings

If `is_normed(Ring)` returns `true` the following function must also be implemented

`double abs(Ring x)`
: Returns the "norm" of `x`. Function `abs(x)` must be such that for every `a` and `b` the following inequalities hold

```
abs(a) >= 0
abs(a) == 0 if and only if a==0
abs(a+b) <= abs(a)+abs(b)
abs(a*b) <= abs(a)*abs(b)
abs(conj(a)) == abs(a) (???)
```

Appendix C Ring Exceptions

The library defines in `rings/rings_exceptions.H` a set of exceptions (in namespace `RingEx`) related to the most common errors that can happen when working with rings. A list of the exceptions follows.

`RingException`
: "Parent class" of every ring exception. Every other exception is derived from this one.

`NonUnitDivision : RingException`
: Attempt to compute the multiplicative inverse of a non-unit element. This is the ring equivalent of "division by zero"

`InvalidOperation : RingException`
: The ring does not support this operation.

  This exception is raised, for example, when calling `deg()` for an element which does not belong to an Euclidean ring.

`FailedConversion : RingException`
: Usually thrown when reading a ring element from an input stream.

`Unimplemented : RingException`
: This operation is unimplemented.

  The difference between this exception and `InvalidOperation` is that `InvalidOperation` is thrown when the requested operation does not make sense for a specific ring (i.e., asking the degree of an element which does not belong to an Euclidean ring), while `Unimplemented` is thrown when the operation makes sense, but it has not implemented yet.

  The constructor for class `Unimplemented` accept an *optional* string which gives more details about the error. Such a string can be accessed via field `reason`, for example

```
#include<rings/rings_exceptions.H>

try {
  throw RingEx::Unimplemented("foo");
} catch (RingEx::Unimplemented x) {
   cerr << x.reason << "\n";  // It will print "foo"
}
```

`Generic : RingException`
: Catch-all exception. It could happen that a specific ring can raise some exception which makes sense for that ring only. For example, when asking for GF(n), the finite field with n elements, an exception must be thrown if n is not the power of a prime number. In order to accomodate for these ring-specific exceptions, the Generic exception is provided. The string parameter of its only constructor can be used to specify the reason of the exception, for example,

```
#include<rings/rings_exceptions.H>
```

```
if (not is_prime_power(n))
  throw RingEx::Generic("Field size != prime^n");
```

The constructor for class `Generic` *requires* a string which gives more details about the error. Such a string can be accessed via the field `reason`, for example

```
#include<rings/rings_exceptions.H>

try {
  throw RingEx::Generic("foo");
} catch (RingEx::Generic x) {
   cerr << x.reason << "\n";  // It will print "foo"
}
```

Appendix D Datafile Signal Format

Appendix E Compact Signal Format

Appendix F Datafile Filterbank Format

Appendix G Formal Syntax of Signal Textual Form

The syntax of a polynomial is as follows

```
Start     := Poly
Poly      := Mono PM Poly | Mono
Mono      := ringVal | VarList | ringVal VarList
VarList   := SingleVar VarList | SingleVar
SingleVar := id | id '^' int
PM        := '+' | '-'
```

In the syntax above we used the following conventions

- Non-terminal symbols begin with an upper case letter, terminal symbol begins with a lower case one. The starting symbol is "Start." The meaning of terminal symbols is as follows

  id A single letter among "xyzuvrst"

  int A signed integer (matching the usual regular expression "[-+]?[0-9]+")

  empty The empty string

  ringVal A ring value. The regular expression for this type of terminal symbol depends on the ring itself and it is obtained by means of the function `regexp` required by the *Generic Ring Interface* (see Appendix B [Generic Ring Interface], page 24).

- Verbatim characters are between '..' (e.g., '^', '+', '-')
- '|' denotes alternatives, '[. . .]' denotes optional parts

According to the syntax a polynomial is a list of monomial separated by + or - signs. Each monomial can be (1) a single ring value, (2) a sequence of powers of variables or (3) a ring value followed by a sequence of powers of variables.

It is easy to see that the parser will first parse a Mono, then if there is a PM sign it will parse another Mono until no PM sign is found. This is important because if we want to put, for example, a polynomial and an integer on the same input line such as in

```
x^2 + 3 x + 1          5
x^3 + y^-1 + 2 x y    -3
```

In the first case the parsing will stop just after the "1" leaving the number "5" still to be read; but in the second case the parser will read also the "-3" because of the minus sign. It would be more convenient to separate the polynomial and the number with some character as in

```
x^2 + 3 x + 1 ,          5
x^3 + y^-1 + 2 x y ,    -3
```

Now in both cases parsing will stop at commas.

Appendix H Glossary

Indices

Constructor Index

Methods Index

N

P

S

Class Index

(Index is nonexistent)

Concept Index

D

F

S