Internal Math Classes (R) Documentation

December 4, 2020

Contents

Introduction

The Internal Math Classes are mainly used for our program internally for computation. In our program, there are two methods of computing linear systems, namely Armadillo and R. Here we illustrate the usage of the R classes, and compare it very briefly to Armadillo.

R can be used as a library for another C++ program, as such the below provides documentation on how it can be used.

What R stands for

R stands for Ring, the mathematical structure. Actually this is confusing, as R does computation only on some Euclidean domains and fields. Furthermore, R has some functions to compute linear operations, so it is not limited to operations on those rings.

Actually we haven't really discussed what we should call this. In our conversation we refer to this as *R* (from R.cpp, R class), and somehow this became the 'name' of the internal math operations.

Goals

Provide backbone for computation in some Euclidean domains and fields (1)

Implement linear operations functions, with steps recorded (2)

Handle display in console and to LaTeX (3)

Parse console display (4)

Files, headers, source

Each source file is placed along the header file of the same name. Here is the list of sources and headers.

```
math/double/Double.h / cpp
math/finite_field/ModField.h / cpp
math/fraction/Fraction.h / cpp
math/linear/LinearOperations.h / cpp
math/linear/CoupledOperations.h / cpp
math/linear/LinOpsSteps.h / cpp
math/long/Long.h / cpp
math/long/LongComplex.h / cpp
math/long/mpz_wrapper.h / cpp
math/polynomial/Polynomial.h / cpp
math/Field.h / cpp
math/Ring.h / cpp
math/R.h / cpp
math/tools.h / cpp
```

Used libraries

R uses the GNU GMP Bignum library, for internal computations of arbitrary large integers.

R vs Armadillo

Here is a brief comparison list of the pros and cons of R and Armadillo, inside our implementation

Supports variable of degree 1	(R)
Has steps included in linear operations	(R)
Has arbitrary precision of integers and rationals	(R)
Better support for double (numerical stability)	(Armadillo)
Much more linear operations	(Armadillo)

Basic usage

Include the header 'math/R.h'. Use R::parse_string to assign values to an R.

```
R a=R::parse_string("3+4/t"); //For polynomials the variable must be t
R b=R::parse_string("-5+(t2-1)/(t-1)");
cout << (a+b) << "\n";
if( !b.is_zero() ){
    cout << (a/b) << "\n";
}
```

The memory is automatically managed inside the R classes. Division throws a divide by zero error, but there might be memory leak internally, as dynamically allocated objects may not be removed. So it is required to check for zero before using / and % operators.

Type mixing errors

The double types and integer types cannot be mixed together. For example:

```
R a=R::parse_string("3.0"); //Double
R b=R::parse_string("4"); //Integer
cout << a.is_type_compatible(b) << "\n"; //False
cout << (a+b) << "\n";
```

throws an error. Also, some complex types and non-complex types cannot be mixed, for example

```
R a=R::parse_string("3+i"); //Complex Integers
R b=R::parse_string("4/3"); //Fraction over Integers
cout << a.is_type_compatible(b) << "\n"; //False
cout << (a+b) << "\n";
```

throws an error. It is best to check for type compatibility before doing operations, to avoid memory leak.

Intuitively the above multiplication should be possible. For a solution see here.

Type mixing success

R automatically checks for type promotion (if the types are compatible), for exaxmple

```
R a=R::parse_string("3+i"); //Complex Integers
R b=R::parse_string("(4t+i)/3"); //Polynomial over Fraction over Complex integers
cout << a.is_type_compatible(b) << "\n"; //True
cout << (a+b) << "\n"; //Success
```

There is no need to cast the values, R automatically does that. For a detailed explanation of types see below.

R wrapper and implementation

The R class is actually a wrapper around a Ring. The R constructor directly puts a given dynamically allocated Ring object internally. It will be destroyed in the destructor. For example:

```
R a=R::parse_string("3+i"); //Complex Integers
R a=R{ new LongComplex{3,1} }; //Equivalent
R coeff[3]={
        R{ new Fraction{ R{ new Long{2} } , R{ new Long{1} } } },
        R{ new Fraction{ R{ new Long{3} } , R{ new Long{2} } } },
        R{ new Fraction{ R{ new Long{4} } , R{ new Long{1} } } }
};
R b=R{ new Polynomial{coeff, 3} };
R b=R::parse_string("2+(3/2)t+4t2"); //Equivalent
```

Notice that Polynomial copies the array of coefficients (see details below). It is discouraged from directly using the subclasses of Ring (e.g. Fraction, Long, Polynomial) for computation.

Types

There are a few types in R, namely

Long, LongComplex, Double, DoubleComplex, ModField, Fraction, Polynomial

The Double class should not be used, since there may be problems in the linear operations regarding numerical stability. Long represents the integers $\mathbb{Z}$, LongComplex $\mathbb{Z}[i]$, ModField(n) $\mathbb{Z}_n$.

A polynomial type has to be created over some field F and the type would correspond to $F[t]$, the polynomials over F. A fraction type over R would be the field of fractions of R, which we denote $\text{Quot}(R)$.

Shallow type representation

The shallow types are represented by the enum RingType (in Ring.h). The values are

```
enum RingType{
    SPECIAL_ZERO,
    DOUBLE, LONG, FRACTION, POLYNOMIAL, COMPLEXIFY, MOD_FIELD
};
```

SPECIAL_ZERO is used by a universal zero element, that is type compatible to every type. This type is insufficient to determine the type of an element, for example

$$\mathbb{Z}_3[t], \quad \Big(\mathrm{Quot}(\mathbb{Z})\Big)[t], \quad \mathbb{Z}_5[t]$$

are of different types, but the outermost type is of RingType POLYNOMIAL.

Full type representation

Each R (or Ring) class comes with a full type representation. This is represented by a NestedRingType class (Ring.h), and by calling R::get_type(), Ring::get_type(). The R directly returns the type of the implementation Ring.

Layers

The NestedRingType class is a linked list, where the outermost element is the current type, and consequent NestedRingType elements would be the type the NestedRingType is over.

In each layer, a type has a RingType and extra_info (int). The current layer (shallow type) is equal iff both RingType and extra_info(int) are equal. The NestedRingType are deep_equals iff all the layers are equal.

Examples of full types

```
POLYNOMIAL(0) -> FRACTION(0) -> LONG(0)
POLYNOMIAL(0) -> FRACTION(0) -> COMPLEX(0) -> LONG(0)
POLYNOMIAL(0) -> MOD_FIELD(3)
POLYNOMIAL(0) -> MOD_FIELD(5)
FRACTION(0) -> COMPLEX(0) -> LONG(0)
LONG(0)
FRACTION(0) -> LONG(0)
COMPLEX(0) -> LONG(0)
```

The '->' denotes the linkage in the linked list NestedRingType.

Type inclusion, type compatibility

To make different types compatible in doing binary operations or comparison, the best we can do is to find, for each ring R_1, R_2, a common ring R' such that $R_1, R_2 \subset R$. In our R classes, use a condition that is sufficient for the above to happen, but may not be necessary.

Terminlogy Let R be a ring (NestedRingType). We have the following:

$$\mathrm{sub}(R) \text{ denotes the next element in the Linked list} \tag{1}$$
$$R_1 =_S R_2 \text{ denotes they are shallow equals} \tag{2}$$

Inclusion

Let R_1, R_2 be two ring (types, NestedRingType). Then R_1 includes R_2 iff one of the below conditions is satisfied:

$$R_2 \text{ is the wildcard zero type} \tag{1}$$
$$\text{sub}(R_1) \text{ includes } R_2 \tag{2}$$
$$R_1 =_S R_2 \text{ and sub}(R_1) \text{ includes sub}(R_2) \tag{3}$$
$$R_1 =_S R_2 \text{ and sub}(R_1) = \text{sub}(R_2) = \varnothing \tag{4}$$

The condition (4) means that no other subsequent elements in the linked list. Notice that the definition is recursive. The above condition may not be a necessary condition for $R_2 \subset R_1$ mathematically, but is sufficient.

Type compatibility

Two types are compatible if and only if $R_1 \subset R_2$ or $R_2 \subset R_1$ (in the above sense). It is possible to do computation/comparison with compatible types.

Extended complex number compatibility

One can deduce from the above rules that the following are incompatible:

```
COMPLEX(0) -> LONG(0)
FRACTION(0) -> LONG(0)
```

It is also quite clear that operations should be possible, we just need to do it in

```
FRACTION(0) -> COMPLEX(0) -> LONG(0)
```

To tackle this, use R::complexify_if_needed on two R objects a,b.

```
/**
* Tries to make a,b compatible by complexifying one of a,b. Returns true if
* it can be done and a,b will be replaced by the appropriate complexified
* versions. Returns false if it is not possible, and does not change a,b.
*/
static R::bool complexify_if_needed(R& a, R& b);
```

Automatic type promotions

When both types R_1, R_2 are compatible, internally R creates a new value if necessary, that is equal to the type of the larger ring R_1. Then the operation returns the result with the type of R_1.

```
R a=R{ new Fraction{new Long{1}, new Long{3}} }; //FRACTION(0) -> LONG(0)
R b=R{ new Long{2} }; //LONG(0)
R c=a+b; //Type of C is FRACTION(0) -> LONG(0)
//The types of a,b won't be changed.
```

Forced type promotions

In some occasions it is needed to force a type promotion, to match a type a.

```
R a= // some value...
R b= // some value...
R c=a.promote(b); //Promote b to the type of a, retaining the same value
R d=a.promote_one(); //Returns the element with value 1 with type of a
R zero=a.promote(R::ZERO); //Same thing
```

The above functions do not change the original values a, b. Note that the promote(const R&) function assumes that typeof(b) $\subset$ typeof(a), in the sense of R::is_type_subset. Check this before calling the promote function.

List of relevant functions

Type checking

```
const NestedRingType& R::get_type() const;

bool R::is_type_compatible(const R& other) const;

static bool Ring::is_type_compatible_shallow(const NestedRingType&, const NestedRingType&);

/**
 * Whether it is compatible with binary operations. Our implementation support
 * operations where either A includes B or B includes A. Note that SPECIAL_ZERO
 * is always compatible.
*/
static bool Ring::is_type_compatible(const NestedRingType&, const NestedRingType&);

/**
 * Returns -2 if they are not compatible, -1 if is_type_subset(b, a)
 * 0 if they are deep equals,
 * 1 if is_type_subset(a, b) (i.e. the ring of a includes the ring of b)
*/
static int Ring::compatibility(const NestedRingType& a, const NestedRingType& b);

/**
 * Whether the type supset contains the type subset. The inclusion need not be strict.
 * Note that subset==SPECIAL_ZERO always satisfies the inclusion.
*/
static bool Ring::is_type_subset(const NestedRingType& supset, const NestedRingType& subset);

/**
* For type outputting
*/
std::ostream& operator<< (std::ostream&, const NestedRingType&);

std::string NestedRingType::to_string() const;

/**
* Sees whether the whole array is compatible with each other.
*/
static bool is_type_compatible(const R* const& arr, int length);
```

Automatic type promotions

All binary operations of R and comparisons (that which require two elements) does automatic promotion internally.

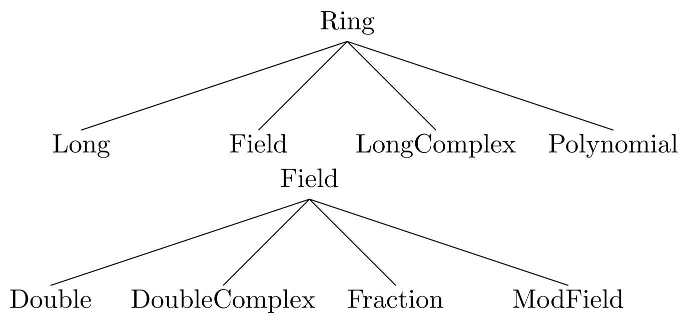
Forced type promotions

```
/**
 * Promotes other to match the type of this. Check
 * Ring::is_type_subset(this->get_type(),other.get_type())
 * before calling this function.
*/
R R::promote(const R& other) const;

/**
 * For internal use of Ring subclasses. Calls impl->promote(r)
*/
const Ring* R::promote_exp(const Ring* const& r) const;

R R::promote_one() const;

/**
* Tries to make a,b compatible by complexifying one of a,b. Returns true if
* it can be done and a,b will be replaced by the appropriate complexified
* versions. Returns false if it is not possible, and does not change a,b.
*/
static R::bool complexify_if_needed(R& a, R& b);

/**
* Promotes every element of the array to a field. Actually this wraps the element
* with a Fraction dividing by one, if the elements are not a field.
*/
static void promote_to_field(R* const& arr, int length);

/**
 * Promotes all types of the matrix to have exactly the same type.
 * Returns true if possible, false if not. This function complexifies
 * the matrix is needed.
*/
static bool RF::ensure_types_equal(RF* const* const matrix, int rows, int cols);

/**
 * Copies the matrix and promotes the type to make sure they are equal,
 * if possible. Returns a newly allocated copy of the matrix where the
 * types are exactly equal, returns nullptr if it is not possbile.
 * This function complexifies if needed.
*/
static RF** RF::copy_and_promote_if_compatible(const R* const* const matrix, int rows, int cols);

/**
 * Promotes all the types to a field. Assumes the types are already exactly equal
 * (see copy_and_promote_if_compatible and ensure_types_equal)
*/
static void promote_to_field(RF* const* const matrix, int rows, int cols);
```

The usage of RF classes will be stated below.

Ring, R, RF

Recall that each R is a wrapper containing a dynamically allocated Ring object. Notice that Ring is an abstract class, which declares the functions required for a Ring to function. Here is an hierarchy of Ring:



Long

This class represents the integers $\mathbb{Z}$. It uses GNU GMP to achieve arbitrary precision. Originally this class uses internally the C++ port of GMP (mpz_class, representing integers). However, we found out later that QT WebEngine requires MSVC on Windows, and GMP does not support MSVC compiler for C++. Instead, we use the C port of GMP, and used our own GMP wrapper mpz_wrapper (that functions similarly to mpz_class).

The Long class does computation analogous to int, long etc.

Instantiation

```
R a=R{ new Long{value} }; //Pass an int/long primitive type value
R b=R{ new Long{mpz_wrapper{"578249578295789425748295579824"}} };
//Pass a string value, can be arbitrarily large.
```

LongComplex

This class represents the Gaussian integers $\mathbb{Z}[i]$. Similar to Long, it uses GNU GMP. The computation of Long would be normal complex number operations. For division, it finds the remainder and quotient similar to that of the integers.

```
R a,b; //Assume they are COMPLEX(0) -> LONG(0) type.
R rem=a%b; //Remember to check for div by zero!
R quot=a/b;
//Types of c,d are still COMPLEX(0) -> LONG(0)
R e=a-(quot*b+rem); //Value is zero in e
```

Instantiation

```
R a=R{ new LongComplex{real, imaginary}};
R b=R{ new LongComplex{mpz_wrapper{"578249578295789425748295579824"}, 5} };
//Can mix int, mpz_wrapper. Actually uses implicit constructor of mpz_wrapper
```

Double

This represents the real numbers $\mathbb{R}$. It is discouraged from using this class as there may be numerical stability issues in the linear operations and polynomials.

Instantiation

```
R a=R{ new Double{value}};
R b=R{ new Double{1.0}};
```

DoubleComplex

Represents $\mathbb{C}$. Similar to Double and LongComplex.

Instantiation

```
R a=R{ new DoubleComplex{real, im}};
R b=R{ new DoubleComplex{1.0,2.0}};
```

ModField

Represents a finite field mod p, i.e. $\mathbb{Z}_p$. Note that this does not encompass all the finite fields, as not all finite fields are of the form $\mathbb{Z}_p$. This class uses the primitive int type, instead of GMP.

To make sure computations are safe, set $p < \frac{1}{2}\sqrt{\text{max int type}}$, and make sure p is a prime number. The operations are done in the field $\mathbb{Z}_p$.

Instantiation

```
R a=R{ new ModField{val, p}};
```

From instantiating a ModField, the value is internally stored as a positive integer n, where $n \equiv \text{val} \mod p$.

Fraction

Represents a fraction over some ring R, i.e. Quot(R). The fraction class internally does type checking and promotion, and reduces by the GCD of the two values.

Instantiation

```
R above,below,val; //Make sure they are compatible!
R frac=R{ new Fraction{above, below}}; //Does not change above, below
R val2=R{ new Fraction{val, val.promote_one()}};
```

Notice in the above the values val1, val2 are the same, but they are of different types. For example:

```
val: LONG(0), val2: FRACTION(0) -> LONG(0)
val: POLYNOMIAL(0) -> FRACTION(0) -> LONG(0)
||
val2: FRACTION(0) -> POLYNOMIAL(0) -> FRACTION(0) -> LONG(0)
```

Make sure that the types are compatible, and there is no divide by zero error before instantiation (check R::is_zero()).

Polynomial

Represents the polynomial ring over some field F, i.e. $F[t]$. The polynomial class stores an array of coefficients, and the length. The operations of polynomials would be standard operations on polynomials, while division and remainder is done through naive polynomial long division.

To make sure computations are safe, only pass values that are fields.

Instantiation

```
R coeff[length]; //Or a pointer works
R::promote_to_field(coeff, length); //This changes the array, if necessary.
if(R::is_type_compatible(coeff, length)){
        R val=R{ new Polynomial{coeff, length} };
}else{
  //Error
}
```

The polynomial copies the given array, in order to avoid strange things such as array of coefficients being deleted. Furthermore, the constructor automatically promotes the type to the same type internally, and removes all zeros, so the length may not be the same.

$$4 + 3t + 2t^2 + t^3 + 0t^4 + 0t^5 = 4 + 3t + 2t^2 + t^3$$

coeff[0] would mean the constant, coeff[1] the power 1 term and so on.

R class

The operators of this class follow from the implementations of the Rings above.

Default constructor

Creates an R with value equal zero, with type being the wildcard zero.

Equality checking

Notice that == and >, < operators does not check for the usual equality and comparison (how do you do < on complex numbers?). Instead, this compares the Euclidean function of the Ring R of the value.

Use bool R::exactly_equals(const R&) const, to check for equality of the value. This would be equality in the usual sense.

Display, conversion to string

R can be converted to a string using different methods. They can be classified:

$$\text{Normal string or LaTeX} \tag{1}$$

$$\text{Signed or unsigned} \tag{2}$$

$$\text{Normal or coefficient or leading coefficient} \tag{3}$$

Usual vs LaTeX Obviously means whether the display is usual or LaTeX.

Signed or unsigned Whether or not the string translation should forcibly add a sign, for example $+1$ vs 1.

Usual vs Coefficient vs Leading coefficient Coefficient would ensure that the display is enclosed in braces, if needed. For example $1, -1$ will be $+1, -1$, and $2+i$ will be $+(2+i)$.

Leading coefficient would not forcibly add a sign at the front of the string.

Functions The functions of conversion to string are:

```
std::string R::to_string() const;

std::string R::to_signed_string() const;

std::string R::to_latex() const;

std::string R::to_signed_latex() const;

std::string R::to_leading_coeff() const;

std::string R::to_coeff() const;

std::string R::to_latex_leading_coeff() const;

std::string R::to_latex_coeff() const;
```

Value from string

The from string methods are split into a modulo field type and normal type. They only support up to Fractions of polynomials of one variable, this means the largest types are $\mathrm{Quot}\Big(\mathrm{Quot}(\mathbb{Z}[i])[t]\Big)$, $\mathrm{Quot}\Big(\mathrm{Quot}(\mathbb{Z}_p[t]\Big)$ respectively. Both parsers tries to keep the type as small as possible.

```
/**
 * Parses using the univariate string/modulo parser (tools.h).
 * Throws exception when parsing fails.
*/
static R R::parse_string(const std::string& s);

static R R::parse_string_modulo(const std::string& s, int mod);
```

These functions are safe to handle using try-catch, without memory leak. Note that it is still required to make sure mod is a prime (refer to here).

The above parsers give the same value when the type corresponds (ModField or not), and when the string is obtained by any of the non-LaTeX conversion to string functions. However the type may not be the same. For example:

```
R val; //Type FRACTION(0) -> LONG(0), representing the value 5/1
R val2=parse_string(val.to_string()); //Type LONG(0), representing 5
```

Miscellaneous functions

```
bool R::is_zero() const;

bool R::is_one() const;
/**
* Whether the value is invertible.
*/
bool R::is_unit() const;
/**
* Whether the type is a field
*/
bool R::is_field() const;
/**
* Complexifies the R, creating a new R with same value, but of complex type.
*/
R R::complexify() const;

/**
 * Converts to a finite field version of this value. Throws an error if type
 * is complex, or if divide by zero. Does not check whether mod is prime.
 * See Fraction::to_finite_field for behaviour of fractions converting to finite field.
*/
R R::to_finite_field(int modulo) const;

/**
 * Returns the complex conjugate of the number. If the number does
 * not belong to a complex field/ring, returns itself.
*/
R R::conjugate() const;

/**
 * Dynamically allocates and copies the array.
*/
static R* R::array_copy(R* const& arr, int length);

/**
 * Dynamically allocates and copies the array. The new array is zero until index==shift.
 * The length of returned array will be length+shift. Notice that
 * array_copy(arr,len,0) is equivalent to array_copy(arr,len)
*/
static R* R::array_copy(R* const& arr, int length, int shift);
```

RF class

This class is a subclass of R. It is used for faster operations, omitting internal type checking and promotion. Therefore it is NOT SAFE to do binary operations/comparison with this class, if the types are not exactly equal (see here)! RF is useful for matrix/linear operations, where the types are initially ensured to be exactly equal.

The declarations are also in 'math/R.h'.

Conversion from R to RF

There is overloaded assignment operator, just use it. RF has no extra data stored internally, same as R.

Miscellaneous functions

```
/**
 * Copies the original array into a new array, shifting and adding zeros as needed
 * (the zero will be SPECIAL_ZERO type). For example
 * 1 2 3 4 5 copies into 0 0 0 1 2 3 4 5, with
 * len=2, shift=3
*/
static RF* array_copy(const RF* const& arr, int len, int shift);

/***
 *  Copies and returns the subarray, between min(inclusive) and max(exclusive).
*/
static RF* subarray_copy(const RF* const& arr, int min, int max);

/**
 * Promotes all types of the matrix to have exactly the same type.
 * Returns true if possible, false if not.
*/
static bool ensure_types_equal(RF* const* const matrix, int rows, int cols);

/**
 * Copies the matrix and promotes the type to make sure they are equal,
 * if possible. Returns a newly allocated copy of the matrix where the
 * types are exactly equal, returns nullptr if it is not possbile.
*/
static RF** copy_and_promote_if_compatible(const R* const* const matrix,
                                                  int rows, int cols);

/**
 * Copies the matrix.
*/
static RF** copy_matrix(const RF* const* const matrix, int rows, int cols);
```

```
/**
 * Promotes all the types to a field. Assumes the types are
 * already exactly equal (see copy_and_promote_if_compatible and ensure_types_equal)
*/
static void promote_to_field(RF* const* const matrix, int rows, int cols);

static void deallocate_matrix(RF **matrix, int rows);
```

Linear Operations

To make use of the LinearOperations which also produces steps, it is required to include 'math/linear/LinearOperations.h', and possibly 'steps/StepsHistory.h' and 'steps/Step.h'.

Calling the Linear Operations function

All operations are in the namespace LinearOperationsFunc. All functions are of the form

```
/**
* Given matrix, with the rows, cols of the matrix. Steps will be nullptr
* if the matrix type is not compatible. Otherwise, it will be dynamically
* allocated and the pointer is placed in the reference.
*/
void <function>(R** mat, int rows, int cols, StepsHistory* &steps);
```

List of functions

```
//Row echelon form
void row_reduce(R** mat, int rows, int cols, StepsHistory* &steps);

//Col echelon form
void col_reduce(R** mat, int rows, int cols, StepsHistory* &steps);

//Solve augmented matrix
void solve(R** mat, int rows, int cols, StepsHistory* &steps);

//Find matrix inverse/left/right inverse
void invert(R** mat, int rows, int cols, StepsHistory* &steps);

//Find determinant
void determinant(R** mat,int rows,int cols,StepsHistory*& steps);

//Characteristic polynomial
void char_poly(R** mat,int rows,int cols,StepsHistory*& steps);

//Gram-Schmidt process on rows
void orthogonalize(R** mat,int rows,int cols,StepsHistory*& steps);
```

Steps

The class for holding all the steps is a linked list StepHistory. The nodes are StepsNode, which in turn holds a pointer to a Step.

StepsNode is a class that is used internally in StepsHistory as a linked list. Step is an abstract class which contains the display information of the steps.

Step navigation and display

```
/**
 * Go to the next node
*/
void StepHistory::next_node();

/**
 * Go to the previous node
*/
void StepHistory::previous_node();

/**
 * Directly jump to the last node
*/
void StepHistory::last_node();

const Step& get_current_node() const;
```

The current step can then be displayed by using these functions:

```
virtual void Step::print_to_console() const=0;

virtual std::string Step::get_html_latex() const=0;
```

Answer setting and retrieval

```
/**
 * Directly saves the pointer internally to StepsHistory.
 * Does not perform copy. The previous answer, if exists, will be deallocated.
*/
void StepsHistory::setAnswer(RF** ans, int rows, int cols);

/**
 * Allocates and returns a new R** matrix respresenting the result,
 * and passes the rows and cols onto the references.result would be
 * nullptr if there is none. For example linear equations with no solution.
*/
void StepsHistory::getAnswer(R** &result, int& rows, int& cols);
```

How to add steps to StepsHistory

The steps have to be dynamically created and added using this function:

```
/**
* Adds a step, does not copy/allocate the step.
*/
void add_step(Step* s);
```

The currently available implementations of Step are StepText (Step.h), LinOpsSteps (LinOpSteps.h), MatrixSpaceStep(LinOpSteps.h). StepText saves simply a string for the console and LaTeX representations. LinOpsSteps saves a R** matrix, while MatrixSpaceStep saves one/two R** matrices for display. It is possible to create a custom class inheriting Step, for some other custom display.

LinOpsSteps is used for saving the state of the matrix after a row/col operation, and MatrixSpaceStep is used for displaying the answer, which usually involves two matrices (representing a particular solution, and the null space of some linear map).

More technical documentation is available in the header files.

Raw linear operations

It should be not required to use these classes. These classes directly operate on a given matrix, doing operations such as RREF or upper triangular diagonalization. There are currently two classes, LinearOperations, CoupledOperations.

Using these classes, make sure that the type of the matrix is compatible. Use RF::ensure_types_equal to ensure that (see documentation above).

The difference between them is just that CoupledOperations save a larger matrix in the steps in StepsHistory, while restricting operation to a submatrix. Notice that CoupledOperations inherit from LinearOperations. Available functions in both classes:

```
virtual void row_add(int from, int to, const RF& mult);
virtual void row_swap(int i, int j);
virtual void row_multiply(int row, const RF& mult);

virtual void col_add(int from, int to, const RF& mult);
virtual void col_swap(int i, int j);
virtual void col_multiply(int col, const RF& mult);

const int& get_rows() const;
const int& get_cols() const;
const int& get_num_swaps() const;

/**
 * Does row operations on the matrix, recording the steps to recorder.
 * Reduces the matrix to reduced row echelon form.
*/
void toRREF();
```

```
/**
 * Diagonalizes the matrix without doing row/col multiplication operations.
 * Virtual to let subclasses that disallow col operations to 'disable' this function
 * by throwing an error.
*/
virtual void diagonalize_no_mult(bool fully=false);

/**
 * Diagonalizes the matrix without doing row/col multiplication operations,
 * and also works for all Ring classes. Virtual to let subclasses that
 * disallow col operations to 'disable' this function by throwing an error.
*/
virtual void diagonalize_no_mult_no_div(bool fully=false);

/**
 * Do Gram-Schmidt process on the rows of the matrix, starting from the first row.
*/
void orthogonalize_rows();

/**
 * Checks whether the matrix now is diagonal, with entries equal to one.
*/
bool is_diagonally_one() const;
```

Initialization

To use these classes, not only it is required to check compatibility outside, a LinOpsRecorder (LinOpsSteps.h) has to accompany these classes in order to record the steps. For example,

```
StepsHistory *steps=new StepsHistory;
LinOpsRecorder rc{steps, mat, rows, cols};

rc.capture_initial(); //capture the initial state of the matrix.

LinearOperations o{mat, rows, cols, false, &rc};
o.toRREF(); //or some other operations

steps->setAnswer(...); //to some answer
```

Design and some algorithms

Interaction of R, RF, Ring

R, RF are the same, except for type checking (see above). The R stores a dynamically allocated Ring implementation. The constructor directly puts the allocated Ring inside R, and there are destructors and copy/assignment constrs/operators to handle the Ring. Here are the rules regarding the interactions

R,RF as a container of Ring

Each dynamically allocated Ring must be in at most 1 R or RF the same time.

Binary operations/comparison/promotion interaction

In here we will call RF as R since they are mostly the same.

R - dynamically allocates some Ring if necessary, to promote type (1)
R - saves the result of doing the operation in the Ring impl (2)
R - deletes any temporary dynamically allocated objects (3)
R - returns the result, by wrapping the result Ring by R (4)

Ring - retrieves another ring pointer from R (1)
Ring - can assume type of other is exactly_equals this (2)
Ring - does computation, implementation specific to the class (3)
Ring - deletes any temporary dynamically allocated objects (4)
Ring - always returns a newly allocated Ring (even if same value) (5)
Ring - always makes sure that the ring pointer from R is not deleted (6)

This is to make sure there are no access violations (invalid cast, pointer deleted in Ring, and try to delete in R again, etc..), and no memory leaks.

Unary operations/comparison/promotion interaction

R - Directly puts the result in R wrapper if return type is a ring (1)

Ring - does computation, implementation specific to the class (1)
Ring - deletes any temporary dynamically allocated objects (2)
Ring - always returns a newly allocated Ring (even if same value, type) (3)

GCD algorithms

Recall that the operator in R compares the Euclidean Function. This is handy for applications such as the GCD algorithm. There is no need to check or positive/negative, and can even be extended directly to other types such as polynomials. For the implementation see math/tools.cpp.

```
long long gcd(long long, long long);

/**
 * Notice that the input must be a,b>0. Returns c,d>0 such that ac+bd=gcd.
*/
void xgcd(const int& a, const int& b, int& gcd, int& c, int& d);

R gcd (R a, R b);

RF gcd_fast (RF a, RF b);
```

Parser

This parser (math/tools.h) in is different from the parser (parser/Parser.h). The parser/Parser.h parser is a more general parser made for the GUI application, which has features like saving variables. However that parser is not a core part of R, so its documentation is in a separate document.

We outline the workings of the parser in (math/tools.h). This is a recursive function, that aims to 'break down' a string into two parts, joined with a binary operation, similar to ASTs. The string will be broken down until they are 'atomic', which means it is a simple expression without binary operations. Then the atomic string can simply be parsed.

Note that this parser supports polynomials up to degree 1 only, with variable t.

Atomic string parsing

$$\begin{gathered}
\text{<Coefficient>}\ t\text{<degree>} \\
\text{<Coefficient>}\ t \\
\text{<Coefficient>}\ it\text{<degree>} \\
\text{<Coefficient>}\ it \\
\text{<Primitive value>} \\
\text{<Primitive value>}\ i
\end{gathered}$$

The above syntax are straightforward, t is the variable, i means complex or not.

General string parsing

To split up the string, we use the usual precedence of the operators. Brackets have highest precedence, then $\times, /$ then $+, -$. The parser reads from left to right. The operations are briefly:

If an open bracket (is encountered, the parser omits all characters until the close bracket) of the same rank (this means for how many open brackets opened, needs to wait for how many closed brackets closed).

Stores an index for the occurrence, and the type of operation of the binary operation with the lowest possible precedence. Note that implicit multiplication without symbol $(3)(4), (3)4, 3(4)$ is also detected.

Splits up the string according to the occurrence above (if exists), and parses the substrings recursively.

Design choice

Why not directly use Ring classes?

It is quite hard to overload operators with inheritance. Even if with multiple overloads, this creates a problem of having to write many overloads for the operators.

Why not use templates?

Some classes (like ModField) should be allowed to have their type determined at runtime (modulo which prime p). Moreover, template still has the problem of having to create many template specializations some operations.

Why use a type promotion system in R, not directly with inheritance?

It may be strange for some that the inheritance does not correspond to type inclusion. For example,

$$\mathbb{Z} \subset \mathbb{Z}[i] \subset \mathrm{Quot}(\mathbb{Z}[i]) \subset \Big(\mathrm{Quot}(\mathbb{Z}[i])\Big)[t] \subset \mathrm{Quot}\bigg(\Big(\mathrm{Quot}(\mathbb{Z}[i])\Big)[t]\bigg)$$

we could have created classes, where subclasses are of a larger type (like LongComplex extends Long). This is because some generic types like Polynomial and Fraction actually uses the same computation over any Field/Ring respectively. It would be redundant to create two classes,

$$\mathrm{Quot}(\mathbb{Z}[i]),\ \ \mathrm{Quot}\bigg(\Big(\mathrm{Quot}(\mathbb{Z}[i])\Big)[t]\bigg)$$

where the internal computation does the same thing. For templates this falls under problem addressed above.

Experimental / Nearly finished features

Note that it is possible to infinitely stack Polynomials and Fractions to represent multivariable polynomials. For example:

```
R a=R::parse_string("1/t"); //Type FRACTION(0) -> POLYNOMIAL(0) -> FRACTION(0) -> LONG(0)
R coeff[2]={a.promote_one(), a};
R val=R{ new Polynomial{coeff, 2} };
```

This creates a polynomial with coefficients, $1, \frac{1}{t}$, meaning a higher degree polynomial

$$p(x,t) = 1 + (\frac{1}{t})x$$

In the display of this polynomial, all the variables, although different, will be displayed with t, since currently to_string functions of polynomials do not check how many polynomials in total this value is wrapped in (although there is already a int no_polynomial saved in NestedRingType).

Also, there is no parser for these kinds of multivariate expression. It is possible to change the atomic parser to support those kinds of polynomials.

The computation of these higher degree polynomials should be correct, as they use the same principles of operating on a polynomial ring over a field and fractions over an Euclidean domain. However, they are not thoroughly tested and may be inefficient and slow in computation.