Aerobus

Generated by Doxygen 1.9.4

1 Concept Index 1

1.1 Concepts . . . 1

2 Class Index 3

2.1 Class List . . . 3

3 File Index 5

3.1 File List . . . 5

4 Concept Documentation 7

4.1 aerobus::IsEuclideanDomain Concept Reference . . . 7

4.1.1 Concept definition . . . 7

4.1.2 Detailed Description . . . 7

4.2 aerobus::IsField Concept Reference . . . 7

4.2.1 Concept definition . . . 7

4.2.2 Detailed Description . . . 8

4.3 aerobus::IsRing Concept Reference . . . 8

4.3.1 Concept definition . . . 8

4.3.2 Detailed Description . . . 8

5 Class Documentation 9

5.1 aerobus::bigint::add< I1, I2 > Struct Template Reference . . . 9

5.2 aerobus::bigint::add_low< I1, I2, I > Struct Template Reference . . . 9

5.3 aerobus::bigint::add_low< I1, I2, std::index_sequence< I... > > Struct Template Reference . . . 9

5.4 aerobus::bigint::add_low_helper< I1, I2, index > Struct Template Reference . . . 10

5.5 aerobus::bigint::add_low_helper< I1, I2, 0 > Struct Template Reference . . . 10

5.6 aerobus::bigint Struct Reference . . . 10

5.7 aerobus::polynomial< Ring, variable_name >::val< coeffN >::coeff_at< index, E > Struct Template Reference . . . 11

5.8 aerobus::polynomial< Ring, variable_name >::val< coeffN >::coeff_at< index, std::enable_if_↩ t<(index< 0||index > 0)> > Struct Template Reference . . . 11

5.9 aerobus::polynomial< Ring, variable_name >::val< coeffN >::coeff_at< index, std::enable_if_↩ t<(index==0)> > Struct Template Reference . . . 11

5.10 aerobus::ContinuedFraction< values > Struct Template Reference . . . 12

5.10.1 Detailed Description . . . 12

5.11 aerobus::ContinuedFraction< a0 > Struct Template Reference . . . 12

5.12 aerobus::ContinuedFraction< a0, rest... > Struct Template Reference . . . 12

5.13 aerobus::bigint::val< s, an, as >::digit_at< index, E > Struct Template Reference . . . 13

5.14 aerobus::bigint::val< s, a0 >::digit_at< index, E > Struct Template Reference . . . 13

5.15 aerobus::bigint::val< s, a0 >::digit_at< index, std::enable_if_t< index !=0 > > Struct Template Reference . . . 13

5.16 aerobus::bigint::val< s, a0 >::digit_at< index, std::enable_if_t< index==0 > > Struct Template Reference . . . 13

5.17 aerobus::bigint::val< s, an, as >::digit_at< index, std::enable_if_t<(index > sizeof...(as))> > Struct Template Reference . . . 14

5.18 aerobus::bigint::val< s, an, as >::digit_at< index, std::enable_if_t<(index<=sizeof...(as))> > Struct Template Reference 14
5.19 aerobus::i32 Struct Reference 14
5.19.1 Detailed Description 16
5.20 aerobus::i64 Struct Reference 16
5.20.1 Detailed Description 17
5.21 aerobus::polynomial< Ring, variable_name >::eval_helper< valueRing, P >::inner< index, stop > Struct Template Reference 17
5.22 aerobus::polynomial< Ring, variable_name >::eval_helper< valueRing, P >::inner< stop, stop > Struct Template Reference 18
5.23 aerobus::is_prime< n > Struct Template Reference 18
5.23.1 Detailed Description 18
5.24 aerobus::bigint::is_zero< I > Struct Template Reference 18
5.25 aerobus::polynomial< Ring, variable_name > Struct Template Reference 19
5.25.1 Detailed Description 20
5.25.2 Member Typedef Documentation 20
5.25.2.1 add_t 20
5.25.2.2 derive_t 21
5.25.2.3 div_t 21
5.25.2.4 eq_t 21
5.25.2.5 gcd_t 21
5.25.2.6 gt_t 22
5.25.2.7 lt_t 22
5.25.2.8 mod_t 22
5.25.2.9 monomial_t 23
5.25.2.10 mul_t 23
5.25.2.11 pos_t 23
5.25.2.12 simplify_t 24
5.25.2.13 sub_t 24
5.26 aerobus::type_list< Ts >::pop_front Struct Reference 25
5.27 aerobus::Quotient< Ring, X > Struct Template Reference 25
5.28 aerobus::type_list< Ts >::split< index > Struct Template Reference 26
5.29 aerobus::type_list< Ts > Struct Template Reference 26
5.29.1 Detailed Description 26
5.30 aerobus::type_list<> Struct Reference 27
5.31 aerobus::bigint::val< s, an, as > Struct Template Reference 27
5.32 aerobus::i32::val< x > Struct Template Reference 28
5.32.1 Detailed Description 28
5.32.2 Member Function Documentation 28
5.32.2.1 eval() 28
5.32.2.2 get() 29
5.33 aerobus::i64::val< x > Struct Template Reference 29
5.33.1 Detailed Description 30

5.33.2 Member Function Documentation . . . 30
5.33.2.1 eval() . . . 30
5.33.2.2 get() . . . 30
5.34 aerobus::polynomial< Ring, variable_name >::val< coeffN, coeffs > Struct Template Reference . . 31
5.34.1 Member Typedef Documentation . . . 31
5.34.1.1 coeff_at_t . . . 31
5.34.2 Member Function Documentation . . . 32
5.34.2.1 eval() . . . 32
5.34.2.2 to_string() . . . 32
5.35 aerobus::Quotient< Ring, X >::val< V > Struct Template Reference . . . 33
5.36 aerobus::zpz< p >::val< x > Struct Template Reference . . . 33
5.37 aerobus::polynomial< Ring, variable_name >::val< coeffN > Struct Template Reference . . . 33
5.38 aerobus::bigint::val< s, a0 > Struct Template Reference . . . 34
5.39 aerobus::zpz< p > Struct Template Reference . . . 35
5.39.1 Detailed Description . . . 35

6 File Documentation **37**
6.1 lib.h . . . 37

7 Example Documentation **65**
7.1 i32::template . . . 65
7.2 i64::template . . . 65
7.3 polynomial . . . 65
7.4 PI_fraction::val . . . 66
7.5 E_fraction::val . . . 66

Index **67**

Chapter 1

Concept Index

1.1 Concepts

Here is a list of all documented concepts with brief descriptions:

aerobus::IsEuclideanDomain
Concept to express R is an euclidean domain 7
aerobus::IsField
Concept to express R is a field 7
aerobus::IsRing
Concept to express R is a Ring (ordered) 8

Chapter 2

Class Index

2.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

aerobus::bigint::add< I1, I2 > 9

aerobus::bigint::add_low< I1, I2, I > 9

aerobus::bigint::add_low< I1, I2, std::index_sequence< I... > > 9

aerobus::bigint::add_low_helper< I1, I2, index > 10

aerobus::bigint::add_low_helper< I1, I2, 0 > 10

aerobus::bigint 10

aerobus::polynomial< Ring, variable_name >::val< coeffN >::coeff_at< index, E > 11

aerobus::polynomial< Ring, variable_name >::val< coeffN >::coeff_at< index, std::enable_if_t<(index< 0||index > 0)> > 11

aerobus::polynomial< Ring, variable_name >::val< coeffN >::coeff_at< index, std::enable_if_t<(index==0)> > 11

aerobus::ContinuedFraction< values >
Continued fraction a0 + 1/(a1 + 1/(...)) 12

aerobus::ContinuedFraction< a0 > 12

aerobus::ContinuedFraction< a0, rest... > 12

aerobus::bigint::val< s, an, as >::digit_at< index, E > 13

aerobus::bigint::val< s, a0 >::digit_at< index, E > 13

aerobus::bigint::val< s, a0 >::digit_at< index, std::enable_if_t< index !=0 > > 13

aerobus::bigint::val< s, a0 >::digit_at< index, std::enable_if_t< index==0 > > 13

aerobus::bigint::val< s, an, as >::digit_at< index, std::enable_if_t<(index > sizeof...(as))> > 14

aerobus::bigint::val< s, an, as >::digit_at< index, std::enable_if_t<(index<=sizeof...(as))> > 14

aerobus::i32
32 bits signed integers, seen as a algebraic ring with related operations 14

aerobus::i64
64 bits signed integers, seen as a algebraic ring with related operations 16

aerobus::polynomial< Ring, variable_name >::eval_helper< valueRing, P >::inner< index, stop > 17

aerobus::polynomial< Ring, variable_name >::eval_helper< valueRing, P >::inner< stop, stop > 18

aerobus::is_prime< n >
Checks if n is prime 18

aerobus::bigint::is_zero< I > 18

aerobus::polynomial< Ring, variable_name > 19

aerobus::type_list< Ts >::pop_front 25

aerobus::Quotient< Ring, X > 25

aerobus::type_list< Ts >::split< index > 26

aerobus::type_list< Ts >
Empty pure template struct to handle type list 26

aerobus::type_list<> 27
aerobus::bigint::val< s, an, as > 27
aerobus::i32::val< x >
Values in i32 28
aerobus::i64::val< x >
Values in i64 29
aerobus::polynomial< Ring, variable_name >::val< coeffN, coeffs > 31
aerobus::Quotient< Ring, X >::val< V > 33
aerobus::zpz< p >::val< x > 33
aerobus::polynomial< Ring, variable_name >::val< coeffN > 33
aerobus::bigint::val< s, a0 > 34
aerobus::zpz< p > 35

Chapter 3

File Index

3.1 File List

Here is a list of all documented files with brief descriptions:

src/lib.h 37

Chapter 4

Concept Documentation

4.1 aerobus::IsEuclideanDomain Concept Reference

Concept to express R is an euclidean domain.

`#include <lib.h>`

4.1.1 Concept definition

```
template<typename R>
concept aerobus::IsEuclideanDomain =  IsRing<R> && requires {
        typename R::template div_t<typename R::one, typename R::one>;
        typename R::template mod_t<typename R::one, typename R::one>;
        typename R::template gcd_t<typename R::one, typename R::one>;
        typename R::template eq_t<typename R::one, typename R::one>;
        typename R::template pos_t<typename R::one>;
        R::template pos_v<typename R::one> == true;

        R::is_euclidean_domain == true;
    }
```

4.1.2 Detailed Description

Concept to express R is an euclidean domain.

4.2 aerobus::IsField Concept Reference

Concept to express R is a field.

`#include <lib.h>`

4.2.1 Concept definition

```
template<typename R>
concept aerobus::IsField =  IsEuclideanDomain<R> && requires {
        R::is_field == true;
    }
```

4.2.2 Detailed Description

Concept to express R is a field.

4.3 aerobus::IsRing Concept Reference

Concept to express R is a Ring (ordered)

```
#include <lib.h>
```

4.3.1 Concept definition

```
template<typename R>
concept aerobus::IsRing =  requires {
        typename R::one;
        typename R::zero;
        typename R::template add_t<typename R::one, typename R::one>;
        typename R::template sub_t<typename R::one, typename R::one>;
        typename R::template mul_t<typename R::one, typename R::one>;
    }
```

4.3.2 Detailed Description

Concept to express R is a Ring (ordered)

Chapter 5

Class Documentation

5.1 aerobus::bigint::add< I1, I2 > Struct Template Reference

Public Types

- using **type** = simplify_t< typename add_low< I1, I2, typename internal::make_index_sequence_reverse< std::max(I1::digits, I2::digits)+1 > >::type >

The documentation for this struct was generated from the following file:

- src/lib.h

5.2 aerobus::bigint::add_low< I1, I2, I > Struct Template Reference

The documentation for this struct was generated from the following file:

- src/lib.h

5.3 aerobus::bigint::add_low< I1, I2, std::index_sequence< I... > > Struct Template Reference

Public Types

- using **type** = val< sign::positive, add_low_helper< I1, I2, I >::digit... >

The documentation for this struct was generated from the following file:

- src/lib.h

5.4 aerobus::bigint::add_low_helper< I1, I2, index > Struct Template Reference

Static Public Attributes

- static constexpr uint32_t **digit** = helper::value
- static constexpr uint8_t **carry_out** = helper::carry_out

The documentation for this struct was generated from the following file:

- src/lib.h

5.5 aerobus::bigint::add_low_helper< I1, I2, 0 > Struct Template Reference

Static Public Attributes

- static constexpr uint32_t **digit** = add_at_helper<I1, I2, 0, 0>::value
- static constexpr uint32_t **carry_out** = add_at_helper<I1, I2, 0, 0>::carry_out

The documentation for this struct was generated from the following file:

- src/lib.h

5.6 aerobus::bigint Struct Reference

Classes

- struct add
- struct add_low
- struct add_low< I1, I2, std::index_sequence< I... > >
- struct add_low_helper
- struct add_low_helper< I1, I2, 0 >
- struct is_zero
- struct val
- struct val< s, a0 >

Public Types

- enum **sign** { **positive** , **negative** }
- using **zero** = val< sign::positive, 0 >
- using **one** = val< sign::positive, 1 >
- template<typename I >
  using **simplify_t** = typename simplify< I >::type

Static Public Attributes

- template<typename I >
  static constexpr bool **is_zero_v** = is_zero<I>::value
- template<typename I1 , typename I2 , size_t index, uint8_t carry_in = 0>
  static constexpr uint32_t **add_at_digit** = add_at_helper<I1, I2, index, carry_in>::value
- template<typename I1 , typename I2 , size_t index, uint8_t carry_in = 0>
  static constexpr uint8_t **add_at_carry** = add_at_helper<I1, I2, index, carry_in>::carry_out

The documentation for this struct was generated from the following file:

- src/lib.h

5.7 aerobus::polynomial< Ring, variable_name >::val< coeffN >::coeff_at< index, E > Struct Template Reference

The documentation for this struct was generated from the following file:

- src/lib.h

5.8 aerobus::polynomial< Ring, variable_name >::val< coeffN >::coeff_at< index, std::enable_if_t<(index< 0||index > 0)> > Struct Template Reference

Public Types

- using **type** = typename Ring::zero

The documentation for this struct was generated from the following file:

- src/lib.h

5.9 aerobus::polynomial< Ring, variable_name >::val< coeffN >::coeff_at< index, std::enable_if_t<(index==0)> > Struct Template Reference

Public Types

- using **type** = aN

The documentation for this struct was generated from the following file:

- src/lib.h

5.10 aerobus::ContinuedFraction< values > Struct Template Reference

represents a continued fraction a0 + 1/(a1 + 1/(...))

```
#include <lib.h>
```

5.10.1 Detailed Description

template<int64_t... values>
struct aerobus::ContinuedFraction< values >

represents a continued fraction a0 + 1/(a1 + 1/(...))

Template Parameters

...values	

The documentation for this struct was generated from the following file:

- src/lib.h

5.11 aerobus::ContinuedFraction< a0 > Struct Template Reference

Public Types

- using **type** = typename q64::template inject_constant_t< a0 >

Static Public Attributes

- static constexpr double **val** = type::template get<double>()

The documentation for this struct was generated from the following file:

- src/lib.h

5.12 aerobus::ContinuedFraction< a0, rest... > Struct Template Reference

Public Types

- using **type** = q64::template add_t< typename q64::template inject_constant_t< a0 >, typename q64↩ ::template div_t< typename q64::one, typename ContinuedFraction< rest... >::type > >

Static Public Attributes

- static constexpr double **val** = type::template get<double>()

The documentation for this struct was generated from the following file:

- src/lib.h

5.13 aerobus::bigint::val< s, an, as >::digit_at< index, E > Struct Template Reference

The documentation for this struct was generated from the following file:

- src/lib.h

5.14 aerobus::bigint::val< s, a0 >::digit_at< index, E > Struct Template Reference

The documentation for this struct was generated from the following file:

- src/lib.h

5.15 aerobus::bigint::val< s, a0 >::digit_at< index, std::enable_if_t< index !=0 > > Struct Template Reference

Static Public Attributes

- static constexpr uint32_t **value** = 0

The documentation for this struct was generated from the following file:

- src/lib.h

5.16 aerobus::bigint::val< s, a0 >::digit_at< index, std::enable_if_t< index==0 > > Struct Template Reference

Static Public Attributes

- static constexpr uint32_t **value** = a0

The documentation for this struct was generated from the following file:

- src/lib.h

5.17 aerobus::bigint::val< s, an, as >::digit_at< index, std::enable_if_t<(index > sizeof...(as))> > Struct Template Reference

Static Public Attributes

- static constexpr uint32_t **value** = 0

The documentation for this struct was generated from the following file:

- src/lib.h

5.18 aerobus::bigint::val< s, an, as >::digit_at< index, std::enable_if_t<(index<=sizeof...(as))> > Struct Template Reference

Static Public Attributes

- static constexpr uint32_t **value** = internal::value_at<(sizeof...(as) - index), an, as...>::value

The documentation for this struct was generated from the following file:

- src/lib.h

5.19 aerobus::i32 Struct Reference

32 bits signed integers, seen as a algebraic ring with related operations

```
#include <lib.h>
```

Classes

- struct val

  values in i32

Public Types

- using **inner_type** = int32_t
- using **zero** = val< 0 >

  constant zero
- using **one** = val< 1 >

  constant one
- template<auto x>
  using **inject_constant_t** = val< static_cast< int32_t >(x)>
- template<typename v >
  using **inject_ring_t** = v
- template<typename v1 , typename v2 >
  using **add_t** = typename add< v1, v2 >::type

  addition operator
- template<typename v1 , typename v2 >
  using **sub_t** = typename sub< v1, v2 >::type

  substraction operator
- template<typename v1 , typename v2 >
  using **mul_t** = typename mul< v1, v2 >::type

  multiplication operator
- template<typename v1 , typename v2 >
  using **div_t** = typename div< v1, v2 >::type

  division operator
- template<typename v1 , typename v2 >
  using **mod_t** = typename remainder< v1, v2 >::type

  modulus operator
- template<typename v1 , typename v2 >
  using **gt_t** = typename gt< v1, v2 >::type

  strictly greater operator (v1 > v2)
- template<typename v1 , typename v2 >
  using **lt_t** = typename lt< v1, v2 >::type

  strict less operator (v1 < v2)
- template<typename v1 , typename v2 >
  using **eq_t** = typename eq< v1, v2 >::type

  equality operator
- template<typename v1 , typename v2 >
  using **gcd_t** = gcd_t< i32, v1, v2 >

  greatest common divisor
- template<typename v >
  using **pos_t** = typename pos< v >::type

  positivity (v > 0)

Static Public Attributes

- static constexpr bool **is_field** = false

  integers are not a field
- static constexpr bool **is_euclidean_domain** = true

  integers are an euclidean domain
- template<typename v >
  static constexpr bool **pos_v** = pos_t<v>::value

5.19.1 Detailed Description

32 bits signed integers, seen as a algebraic ring with related operations

The documentation for this struct was generated from the following file:

- src/lib.h

5.20 aerobus::i64 Struct Reference

64 bits signed integers, seen as a algebraic ring with related operations

```
#include <lib.h>
```

Classes

- struct val

  values in i64

Public Types

- using **inner_type** = int64_t
- template<auto x>
  using **inject_constant_t** = val< static_cast< int64_t >(x)>
- template<typename v >
  using **inject_ring_t** = v
- using **zero** = val< 0 >

  constant zero
- using **one** = val< 1 >

  constant one
- template<typename v1 , typename v2 >
  using **add_t** = typename add< v1, v2 >::type

  addition operator
- template<typename v1 , typename v2 >
  using **sub_t** = typename sub< v1, v2 >::type

  substraction operator
- template<typename v1 , typename v2 >
  using **mul_t** = typename mul< v1, v2 >::type

  multiplication operator
- template<typename v1 , typename v2 >
  using **div_t** = typename div< v1, v2 >::type

  division operator
- template<typename v1 , typename v2 >
  using **mod_t** = typename remainder< v1, v2 >::type

  modulus operator
- template<typename v1 , typename v2 >
  using **gt_t** = typename gt< v1, v2 >::type

  strictly greater operator (v1 > v2)

- template<typename v1 , typename v2 >
  using **lt_t** = typename lt< v1, v2 >::type

  strict less operator (v1 < v2)
- template<typename v1 , typename v2 >
  using **eq_t** = typename eq< v1, v2 >::type

  equality operator
- template<typename v1 , typename v2 >
  using **gcd_t** = gcd_t< i64, v1, v2 >

  greatest common divisor
- template<typename v >
  using **pos_t** = typename pos< v >::type

  is v posititive

Static Public Attributes

- static constexpr bool **is_field** = false

  integers are not a field
- static constexpr bool **is_euclidean_domain** = true

  integers are an euclidean domain
- template<typename v >
  static constexpr bool **pos_v** = pos_t<v>::value

5.20.1 Detailed Description

64 bits signed integers, seen as a algebraic ring with related operations

The documentation for this struct was generated from the following file:

- src/lib.h

5.21 aerobus::polynomial< Ring, variable_name >::eval_helper< valueRing, P >::inner< index, stop > Struct Template Reference

Static Public Member Functions

- static constexpr valueRing **func** (const valueRing &accum, const valueRing &x)

The documentation for this struct was generated from the following file:

- src/lib.h

5.22 aerobus::polynomial< Ring, variable_name >::eval_helper< valueRing, P >::inner< stop, stop > Struct Template Reference

Static Public Member Functions

- static constexpr valueRing **func** (const valueRing &accum, const valueRing &x)

The documentation for this struct was generated from the following file:

- src/lib.h

5.23 aerobus::is_prime< n > Struct Template Reference

checks if n is prime

```
#include <lib.h>
```

Static Public Attributes

- static constexpr bool **value** = internal::_is_prime<n, 5>::value
  true iff n is prime

5.23.1 Detailed Description

template<int32_t n>
struct aerobus::is_prime< n >

checks if n is prime

Template Parameters

n	

The documentation for this struct was generated from the following file:

- src/lib.h

5.24 aerobus::bigint::is_zero< I > Struct Template Reference

Static Public Attributes

- static constexpr bool **value** = I::digits == 1 && I::aN == 0

The documentation for this struct was generated from the following file:

- src/lib.h

5.25 aerobus::polynomial< Ring, variable_name > Struct Template Reference

```
#include <lib.h>
```

Classes

- struct val
- struct val< coeffN >

Public Types

- using **zero** = val< typename Ring::zero >

  constant zero
- using **one** = val< typename Ring::one >

  constant one
- using **X** = val< typename Ring::one, typename Ring::zero >

  generator
- template<typename P >
  using simplify_t = typename simplify< P >::type

  simplifies a polynomial (deletes highest degree if null, do nothing otherwise)
- template<typename v1 , typename v2 >
  using add_t = typename add< v1, v2 >::type

  adds two polynomials
- template<typename v1 , typename v2 >
  using sub_t = typename sub< v1, v2 >::type

  substraction of two polynomials
- template<typename v1 , typename v2 >
  using mul_t = typename mul< v1, v2 >::type

  multiplication of two polynomials
- template<typename v1 , typename v2 >
  using eq_t = typename eq_helper< v1, v2 >::type

  equality operator
- template<typename v1 , typename v2 >
  using lt_t = typename lt_helper< v1, v2 >::type

  strict less operator
- template<typename v1 , typename v2 >
  using gt_t = typename gt_helper< v1, v2 >::type

  strict greater operator
- template<typename v1 , typename v2 >
  using div_t = typename div< v1, v2 >::q_type

  division operator
- template<typename v1 , typename v2 >
  using mod_t = typename div_helper< v1, v2, zero, v1 >::mod_type

  modulo operator
- template<typename coeff , size_t deg>
  using monomial_t = typename monomial< coeff, deg >::type

  monomial : coeff X^deg
- template<typename v >
  using derive_t = typename derive_helper< v >::type

derivation operator

- template<typename v >
  using pos_t = typename Ring::template pos_t< typename v::aN >

  checks for positivity (an > 0)
- template<typename v1 , typename v2 >
  using gcd_t = std::conditional_t< Ring::is_euclidean_domain, typename make_unit< gcd_t< polynomial< Ring, variable_name >, v1, v2 > >::type, void >

  greatest common divisor of two polynomials
- template<auto x>
  using **inject_constant_t** = val< typename Ring::template inject_constant_t< x > >
- template<typename v >
  using **inject_ring_t** = val< v >

Static Public Attributes

- static constexpr bool **is_field** = false
- static constexpr bool **is_euclidean_domain** = Ring::is_euclidean_domain
- template<typename v >
  static constexpr bool **pos_v** = pos_t<v>::value

5.25.1 Detailed Description

template<typename Ring, char variable_name = 'x'>
requires IsEuclideanDomain<Ring>
struct aerobus::polynomial< Ring, variable_name >

polynomial with coefficients in Ring Ring must be an integral domain

5.25.2 Member Typedef Documentation

5.25.2.1 add_t

```
template<typename Ring , char variable_name = 'x'>
template<typename v1 , typename v2 >
using aerobus::polynomial< Ring, variable_name >::add_t = typename add<v1, v2>::type
```

adds two polynomials

Template Parameters

v1	
v2	

5.25.2.2 derive_t

```
template<typename Ring , char variable_name = 'x'>
template<typename v >
using aerobus::polynomial< Ring, variable_name >::derive_t = typename derive_helper<v>::type
```

derivation operator

Template Parameters

v	

5.25.2.3 div_t

```
template<typename Ring , char variable_name = 'x'>
template<typename v1 , typename v2 >
using aerobus::polynomial< Ring, variable_name >::div_t = typename div<v1, v2>::q_type
```

division operator

Template Parameters

v1	
v2	

5.25.2.4 eq_t

```
template<typename Ring , char variable_name = 'x'>
template<typename v1 , typename v2 >
using aerobus::polynomial< Ring, variable_name >::eq_t = typename eq_helper<v1, v2>::type
```

equality operator

Template Parameters

v1	
v2	

5.25.2.5 gcd_t

```
template<typename Ring , char variable_name = 'x'>
```

```
template<typename v1 , typename v2 >
using aerobus::polynomial< Ring, variable_name >::gcd_t = std::conditional_t< Ring::is_↩
euclidean_domain, typename make_unit<gcd_t<polynomial<Ring, variable_name>, v1, v2> >::type,
void>
```

greatest common divisor of two polynomials

Template Parameters

v1	
v2	

5.25.2.6 gt_t

```
template<typename Ring , char variable_name = 'x'>
template<typename v1 , typename v2 >
using aerobus::polynomial< Ring, variable_name >::gt_t = typename gt_helper<v1, v2>::type
```

strict greater operator

Template Parameters

v1	
v2	

5.25.2.7 lt_t

```
template<typename Ring , char variable_name = 'x'>
template<typename v1 , typename v2 >
using aerobus::polynomial< Ring, variable_name >::lt_t = typename lt_helper<v1, v2>::type
```

strict less operator

Template Parameters

v1	
v2	

5.25.2.8 mod_t

```
template<typename Ring , char variable_name = 'x'>
```

```
template<typename v1 , typename v2 >
using aerobus::polynomial< Ring, variable_name >::mod_t = typename div_helper<v1, v2, zero,
v1>::mod_type
```

modulo operator

Template Parameters

v1	
v2	

5.25.2.9 monomial_t

```
template<typename Ring , char variable_name = 'x'>
template<typename coeff , size_t deg>
using aerobus::polynomial< Ring, variable_name >::monomial_t = typename monomial<coeff, deg>↩
::type
```

monomial : coeff X^deg

Template Parameters

coeff	
deg	

5.25.2.10 mul_t

```
template<typename Ring , char variable_name = 'x'>
template<typename v1 , typename v2 >
using aerobus::polynomial< Ring, variable_name >::mul_t = typename mul<v1, v2>::type
```

multiplication of two polynomials

Template Parameters

v1	
v2	

5.25.2.11 pos_t

```
template<typename Ring , char variable_name = 'x'>
```

```
template<typename v >
using aerobus::polynomial< Ring, variable_name >::pos_t = typename Ring::template pos_t<typename
v::aN>
```

checks for positivity (an $>$ 0)

Template Parameters

v	

5.25.2.12 simplify_t

```
template<typename Ring , char variable_name = 'x'>
template<typename P >
using aerobus::polynomial< Ring, variable_name >::simplify_t = typename simplify<P>::type
```

simplifies a polynomial (deletes highest degree if null, do nothing otherwise)

Template Parameters

P	

5.25.2.13 sub_t

```
template<typename Ring , char variable_name = 'x'>
template<typename v1 , typename v2 >
using aerobus::polynomial< Ring, variable_name >::sub_t = typename sub<v1, v2>::type
```

substraction of two polynomials

Template Parameters

v1	
v2	

The documentation for this struct was generated from the following file:

- src/lib.h

5.26 aerobus::type_list< Ts >::pop_front Struct Reference

Public Types

- using **type** = typename internal::pop_front_h< Ts... >::head
- using **tail** = typename internal::pop_front_h< Ts... >::tail

The documentation for this struct was generated from the following file:

- src/lib.h

5.27 aerobus::Quotient< Ring, X > Struct Template Reference

Classes

- struct val

Public Types

- using **zero** = val< typename Ring::zero >
- using **one** = val< typename Ring::one >
- template<typename v1 , typename v2 >
  using **add_t** = val< typename Ring::template add_t< typename v1::type, typename v2::type > >
- template<typename v1 , typename v2 >
  using **mul_t** = val< typename Ring::template mul_t< typename v1::type, typename v2::type > >
- template<typename v1 , typename v2 >
  using **div_t** = val< typename Ring::template div_t< typename v1::type, typename v2::type > >
- template<typename v1 , typename v2 >
  using **mod_t** = val< typename Ring::template mod_t< typename v1::type, typename v2::type > >
- template<typename v1 , typename v2 >
  using **eq_t** = typename Ring::template eq_t< typename v1::type, typename v2::type >
- template<typename v1 >
  using **pos_t** = std::true_type
- template<auto x>
  using **inject_constant_t** = val< typename Ring::template inject_constant_t< x > >
- template<typename v >
  using **inject_ring_t** = val< v >

Static Public Attributes

- template<typename v >
  static constexpr bool **pos_v** = pos_t<v>::value
- static constexpr bool **is_euclidean_domain** = true

The documentation for this struct was generated from the following file:

- src/lib.h

5.28 aerobus::type_list< Ts >::split< index > Struct Template Reference

Public Types

- using **head** = typename inner::head
- using **tail** = typename inner::tail

The documentation for this struct was generated from the following file:

- src/lib.h

5.29 aerobus::type_list< Ts > Struct Template Reference

Empty pure template struct to handle type list.

Classes

- struct pop_front
- struct split

Public Types

- template<typename T >
  using **push_front** = type_list< T, Ts... >
- template<uint64_t index>
  using **at** = internal::type_at_t< index, Ts... >
- template<typename T >
  using **push_back** = type_list< Ts..., T >
- template<typename U >
  using **concat** = typename concat_h< U >::type
- template<uint64_t index, typename T >
  using **insert** = typename internal::insert_h< index, type_list< Ts... >, T >::type
- template<uint64_t index>
  using **remove** = typename internal::remove_h< index, type_list< Ts... > >::type

Static Public Attributes

- static constexpr size_t **length** = sizeof...(Ts)

5.29.1 Detailed Description

template<typename... Ts>
struct aerobus::type_list< Ts >

Empty pure template struct to handle type list.

The documentation for this struct was generated from the following file:

- src/lib.h

5.30 aerobus::type_list<> Struct Reference

Public Types

- template<typename T >
  using **push_front** = type_list< T >
- template<typename T >
  using **push_back** = type_list< T >
- template<typename U >
  using **concat** = U
- template<uint64_t index, typename T >
  using **insert** = type_list< T >

Static Public Attributes

- static constexpr size_t **length** = 0

The documentation for this struct was generated from the following file:

- src/lib.h

5.31 aerobus::bigint::val< s, an, as > Struct Template Reference

Classes

- struct digit_at
- struct digit_at< index, std::enable_if_t<(index > sizeof...(as))> >
- struct digit_at< index, std::enable_if_t<(index<=sizeof...(as))> >

Public Types

- using **strip** = val< s, as... >

Static Public Member Functions

- static std::string **to_string** ()

Static Public Attributes

- static constexpr bool **is_positive** = s != sign::negative
- static constexpr uint32_t **aN** = an
- static constexpr size_t **digits** = sizeof...(as) + 1

The documentation for this struct was generated from the following file:

- src/lib.h

5.32 aerobus::i32::val< x > Struct Template Reference

values in i32

```
#include <lib.h>
```

Public Types

- using **is_zero_t** = std::bool_constant< x==0 >

  is value zero

Static Public Member Functions

- template<typename valueType >
  static constexpr valueType get ()

  cast x into valueType
- static std::string **to_string** ()

  string representation of value
- template<typename valueRing >
  static constexpr valueRing eval (const valueRing &v)

  cast x into valueRing

Static Public Attributes

- static constexpr int32_t **v** = x

5.32.1 Detailed Description

template<int32_t x>
struct aerobus::i32::val< x >

values in i32

Template Parameters

x	an actual integer

5.32.2 Member Function Documentation

5.32.2.1 eval()

```
template<int32_t x>
template<typename valueRing >
```

```
static constexpr valueRing aerobus::i32::val< x >::eval (
              const valueRing & v )  [inline], [static], [constexpr]
```

cast x into valueRing

Template Parameters

valueRing	double for example

5.32.2.2 get()

```
template<int32_t x>
template<typename valueType >
static constexpr valueType aerobus::i32::val< x >::get ( )  [inline], [static], [constexpr]
```

cast x into valueType

Template Parameters

valueType	double for example

The documentation for this struct was generated from the following file:

- src/lib.h

5.33 aerobus::i64::val< x > Struct Template Reference

values in i64

```
#include <lib.h>
```

Public Types

- using **is_zero_t** = std::bool_constant< x==0 >

  is value zero

Static Public Member Functions

- template<typename valueType >
  static constexpr valueType get ()

  cast value in valueType
- static std::string **to_string** ()

  string representation
- template<typename valueRing >
  static constexpr valueRing eval (const valueRing &v)

  cast value in valueRing

Static Public Attributes

- static constexpr int64_t **v** = x

5.33.1 Detailed Description

template<int64_t x>
struct aerobus::i64::val< x >

values in i64

Template Parameters

x	an actual integer

5.33.2 Member Function Documentation

5.33.2.1 eval()

```
template<int64_t x>
template<typename valueRing >
static constexpr valueRing aerobus::i64::val< x >::eval (
              const valueRing & v )  [inline], [static], [constexpr]
```

cast value in valueRing

Template Parameters

valueRing	(double for example)

5.33.2.2 get()

```
template<int64_t x>
template<typename valueType >
static constexpr valueType aerobus::i64::val< x >::get ( )  [inline], [static], [constexpr]
```

cast value in valueType

Template Parameters

valueType	(double for example)

The documentation for this struct was generated from the following file:

- src/lib.h

5.34 aerobus::polynomial< Ring, variable_name >::val< coeffN, coeffs > Struct Template Reference

Public Types

- using **aN** = coeffN

  heavy weight coefficient (non zero)
- using **strip** = val< coeffs... >

  remove largest coefficient
- using **is_zero_t** = std::bool_constant<(degree==0) &&(aN::is_zero_t::value)>

  true if polynomial is constant zero
- template<size_t index>
  using coeff_at_t = typename coeff_at< index >::type

  coefficient at index

Static Public Member Functions

- static std::string to_string ()

  get a string representation of polynomial
- template<typename valueRing >
  static constexpr valueRing eval (const valueRing &x)

  evaluates polynomial seen as a function operating on ValueRing

Static Public Attributes

- static constexpr size_t **degree** = sizeof...(coeffs)

  degree of the polynomial

5.34.1 Member Typedef Documentation

5.34.1.1 coeff_at_t

```
template<typename Ring , char variable_name = 'x'>
template<typename coeffN , typename...  coeffs>
template<size_t index>
using aerobus::polynomial< Ring, variable_name >::val< coeffN, coeffs >::coeff_at_t = typename
coeff_at<index>::type
```

coefficient at index

Template Parameters

index	

5.34.2 Member Function Documentation

5.34.2.1 eval()

```
template<typename Ring , char variable_name = 'x'>
template<typename coeffN , typename... coeffs>
template<typename valueRing >
static constexpr valueRing aerobus::polynomial< Ring, variable_name >::val< coeffN, coeffs
>::eval (
            const valueRing & x ) [inline], [static], [constexpr]
```

evaluates polynomial seen as a function operating on ValueRing

Template Parameters

valueRing	usually float or double

Parameters

x	value

Returns

P(x)

5.34.2.2 to_string()

```
template<typename Ring , char variable_name = 'x'>
template<typename coeffN , typename... coeffs>
static std::string aerobus::polynomial< Ring, variable_name >::val< coeffN, coeffs >::to_←
string ( ) [inline], [static]
```

get a string representation of polynomial

Returns

something like a_n X^n + ... + a_1 X + a_0

The documentation for this struct was generated from the following file:

- src/lib.h

5.35 aerobus::Quotient< Ring, X >::val< V > Struct Template Reference

Public Types

- using **type** = std::conditional_t< Ring::template pos_v< tmp >, tmp, typename Ring::template sub_t< typename Ring::zero, tmp > >

The documentation for this struct was generated from the following file:

- src/lib.h

5.36 aerobus::zpz< p >::val< x > Struct Template Reference

Public Types

- using **is_zero_t** = std::bool_constant< x% p==0 >

Static Public Member Functions

- template<typename valueType >
  static constexpr valueType **get** ()
- static std::string **to_string** ()
- template<typename valueRing >
  static constexpr valueRing **eval** (const valueRing &v)

Static Public Attributes

- static constexpr int32_t **v** = x % p

The documentation for this struct was generated from the following file:

- src/lib.h

5.37 aerobus::polynomial< Ring, variable_name >::val< coeffN > Struct Template Reference

Classes

- struct coeff_at
- struct coeff_at< index, std::enable_if_t<(index< 0||index > 0)> >
- struct coeff_at< index, std::enable_if_t<(index==0)> >

Public Types

- using **aN** = coeffN
- using **strip** = val< coeffN >
- using **is_zero_t** = std::bool_constant< aN::is_zero_t::value >
- template<size_t index>
  using **coeff_at_t** = typename coeff_at< index >::type

Static Public Member Functions

- static std::string **to_string** ()
- template<typename valueRing >
  static constexpr valueRing **eval** (const valueRing &x)

Static Public Attributes

- static constexpr size_t **degree** = 0

The documentation for this struct was generated from the following file:

- src/lib.h

5.38 aerobus::bigint::val< s, a0 > Struct Template Reference

Classes

- struct digit_at
- struct digit_at< index, std::enable_if_t< index !=0 > >
- struct digit_at< index, std::enable_if_t< index==0 > >

Public Types

- using **strip** = val< s, a0 >

Static Public Member Functions

- static std::string **to_string** ()

Static Public Attributes

- static constexpr bool **is_positive** = s != sign::negative
- static constexpr uint32_t **aN** = a0
- static constexpr size_t **digits** = 1

The documentation for this struct was generated from the following file:

- src/lib.h

5.39 aerobus::zpz< p > Struct Template Reference

```
#include <lib.h>
```

Classes

- struct val

Public Types

- using **inner_type** = int32_t
- template<auto x>
  using **inject_constant_t** = val< static_cast< int32_t >(x)>
- using **zero** = val< 0 >
- using **one** = val< 1 >
- template<typename v1 , typename v2 >
  using **add_t** = typename add< v1, v2 >::type
- template<typename v1 , typename v2 >
  using **sub_t** = typename sub< v1, v2 >::type
- template<typename v1 , typename v2 >
  using **mul_t** = typename mul< v1, v2 >::type
- template<typename v1 , typename v2 >
  using **div_t** = typename div< v1, v2 >::type
- template<typename v1 , typename v2 >
  using **mod_t** = typename remainder< v1, v2 >::type
- template<typename v1 , typename v2 >
  using **gt_t** = typename gt< v1, v2 >::type
- template<typename v1 , typename v2 >
  using **lt_t** = typename lt< v1, v2 >::type
- template<typename v1 , typename v2 >
  using **eq_t** = typename eq< v1, v2 >::type
- template<typename v1 , typename v2 >
  using **gcd_t** = gcd_t< i32, v1, v2 >
- template<typename v1 >
  using **pos_t** = typename pos< v1 >::type

Static Public Attributes

- static constexpr bool **is_field** = is_prime<p>::value
- static constexpr bool **is_euclidean_domain** = true
- template<typename v >
  static constexpr bool **pos_v** = pos_t<v>::value

5.39.1 Detailed Description

template<int32_t p>
struct aerobus::zpz< p >

congruence classes of integers for a modulus if p is prime, zpz is a field, otherwise an integral domain with all related operations

The documentation for this struct was generated from the following file:

- src/lib.h

Chapter 6

File Documentation

6.1 lib.h

```
// -*- lsst-c++ -*-

#include <cstdint> // NOLINT(clang-diagnostic-pragma-pack)
#include <cstddef>
#include <cstring>
#include <type_traits>
#include <utility>
#include <algorithm>
#include <functional>
#include <string>
#include <concepts>
#include <array>


#ifdef _MSC_VER
#define ALIGNED(x) __declspec(align(x))
#define INLINED __forceinline
#else
#define ALIGNED(x) __attribute__((aligned(x)))
#define INLINED __attribute__((always_inline)) inline
#endif

// aligned allocation
namespace aerobus {
    template<typename T>
    T* aligned_malloc(size_t count, size_t alignment) {
#ifdef _MSC_VER
        return static_cast<T*>(_aligned_malloc(count * sizeof(T), alignment));
#else
        return static_cast<T*>(aligned_alloc(alignment, count * sizeof(T)));
#endif
    }

    constexpr std::array<int32_t, 1000> primes = { { 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43,
   47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103, 107, 109, 113, 127, 131, 137, 139, 149, 151,
   157, 163, 167, 173, 179, 181, 191, 193, 197, 199, 211, 223, 227, 229, 233, 239, 241, 251, 257, 263,
   269, 271, 277, 281, 283, 293, 307, 311, 313, 317, 331, 337, 347, 349, 353, 359, 367, 373, 379, 383,
   389, 397, 401, 409, 419, 421, 431, 433, 439, 443, 449, 457, 461, 463, 467, 479, 487, 491, 499, 503,
   509, 521, 523, 541, 547, 557, 563, 569, 571, 577, 587, 593, 599, 601, 607, 613, 617, 619, 631, 641,
   643, 647, 653, 659, 661, 673, 677, 683, 691, 701, 709, 719, 727, 733, 739, 743, 751, 757, 761, 769,
   773, 787, 797, 809, 811, 821, 823, 827, 829, 839, 853, 857, 859, 863, 877, 881, 883, 887, 907, 911,
   919, 929, 937, 941, 947, 953, 967, 971, 977, 983, 991, 997, 1009, 1013, 1019, 1021, 1031, 1033, 1039,
   1049, 1051, 1061, 1063, 1069, 1087, 1091, 1093, 1097, 1103, 1109, 1117, 1123, 1129, 1151, 1153, 1163,
   1171, 1181, 1187, 1193, 1201, 1213, 1217, 1223, 1229, 1231, 1237, 1249, 1259, 1277, 1279, 1283, 1289,
   1291, 1297, 1301, 1303, 1307, 1319, 1321, 1327, 1361, 1367, 1373, 1381, 1399, 1409, 1423, 1427, 1429,
   1433, 1439, 1447, 1451, 1453, 1459, 1471, 1481, 1483, 1487, 1489, 1493, 1499, 1511, 1523, 1531, 1543,
   1549, 1553, 1559, 1567, 1571, 1579, 1583, 1597, 1601, 1607, 1609, 1613, 1619, 1621, 1627, 1637, 1657,
   1663, 1667, 1669, 1693, 1697, 1699, 1709, 1721, 1723, 1733, 1741, 1747, 1753, 1759, 1777, 1783, 1787,
   1789, 1801, 1811, 1823, 1831, 1847, 1861, 1867, 1871, 1873, 1877, 1879, 1889, 1901, 1907, 1913, 1931,
   1933, 1949, 1951, 1973, 1979, 1987, 1993, 1997, 1999, 2003, 2011, 2017, 2027, 2029, 2039, 2053, 2063,
   2069, 2081, 2083, 2087, 2089, 2099, 2111, 2113, 2129, 2131, 2137, 2141, 2143, 2153, 2161, 2179, 2203,
   2207, 2213, 2221, 2237, 2239, 2243, 2251, 2267, 2269, 2273, 2281, 2287, 2293, 2297, 2309, 2311, 2333,
   2339, 2341, 2347, 2351, 2357, 2371, 2377, 2381, 2383, 2389, 2393, 2399, 2411, 2417, 2423, 2437, 2441,
   2447, 2459, 2467, 2473, 2477, 2503, 2521, 2531, 2539, 2543, 2549, 2551, 2557, 2579, 2591, 2593, 2609,
   2617, 2621, 2633, 2647, 2657, 2659, 2663, 2671, 2677, 2683, 2687, 2689, 2693, 2699, 2707, 2711, 2713,
   2719, 2729, 2731, 2741, 2749, 2753, 2767, 2777, 2789, 2791, 2797, 2801, 2803, 2819, 2833, 2837, 2843,
   2851, 2857, 2861, 2879, 2887, 2897, 2903, 2909, 2917, 2927, 2939, 2953, 2957, 2963, 2969, 2971, 2999,
   3001, 3011, 3019, 3023, 3037, 3041, 3049, 3061, 3067, 3079, 3083, 3089, 3109, 3119, 3121, 3137, 3163,
   3167, 3169, 3181, 3187, 3191, 3203, 3209, 3217, 3221, 3229, 3251, 3253, 3257, 3259, 3271, 3299, 3301,
   3307, 3313, 3319, 3323, 3329, 3331, 3343, 3347, 3359, 3361, 3371, 3373, 3389, 3391, 3407, 3413, 3433,
   3449, 3457, 3461, 3463, 3467, 3469, 3491, 3499, 3511, 3517, 3527, 3529, 3533, 3539, 3541, 3547, 3557,
   3559, 3571, 3581, 3583, 3593, 3607, 3613, 3617, 3623, 3631, 3637, 3643, 3659, 3671, 3673, 3677, 3691,
   3697, 3701, 3709, 3719, 3727, 3733, 3739, 3761, 3767, 3769, 3779, 3793, 3797, 3803, 3821, 3823, 3833,
   3847, 3851, 3853, 3863, 3877, 3881, 3889, 3907, 3911, 3917, 3919, 3923, 3929, 3931, 3943, 3947, 3967,
   3989, 4001, 4003, 4007, 4013, 4019, 4021, 4027, 4049, 4051, 4057, 4073, 4079, 4091, 4093, 4099, 4111,
   4127, 4129, 4133, 4139, 4153, 4157, 4159, 4177, 4201, 4211, 4217, 4219, 4229, 4231, 4241, 4243, 4253,
   4259, 4261, 4271, 4273, 4283, 4289, 4297, 4327, 4337, 4339, 4349, 4357, 4363, 4373, 4391, 4397, 4409,
   4421, 4423, 4441, 4447, 4451, 4457, 4463, 4481, 4483, 4493, 4507, 4513, 4517, 4519, 4523, 4547, 4549,
   4561, 4567, 4583, 4591, 4597, 4603, 4621, 4637, 4639, 4643, 4649, 4651, 4657, 4663, 4673, 4679, 4691,
   4703, 4721, 4723, 4729, 4733, 4751, 4759, 4783, 4787, 4789, 4793, 4799, 4801, 4813, 4817, 4831, 4861,
   4871, 4877, 4889, 4903, 4909, 4919, 4931, 4933, 4937, 4943, 4951, 4957, 4967, 4969, 4973, 4987, 4993,
   4999, 5003, 5009, 5011, 5021, 5023, 5039, 5051, 5059, 5077, 5081, 5087, 5099, 5101, 5107, 5113, 5119,
   5147, 5153, 5167, 5171, 5179, 5189, 5197, 5209, 5227, 5231, 5233, 5237, 5261, 5273, 5279, 5281, 5297,
   5303, 5309, 5323, 5333, 5347, 5351, 5381, 5387, 5393, 5399, 5407, 5413, 5417, 5419, 5431, 5437, 5441,
   5443, 5449, 5471, 5477, 5479, 5483, 5501, 5503, 5507, 5519, 5521, 5527, 5531, 5557, 5563, 5569, 5573,
   5581, 5591, 5623, 5639, 5641, 5647, 5651, 5653, 5657, 5659, 5669, 5683, 5689, 5693, 5701, 5711, 5717,
   5737, 5741, 5743, 5749, 5779, 5783, 5791, 5801, 5807, 5813, 5821, 5827, 5839, 5843, 5849, 5851, 5857,
   5861, 5867, 5869, 5879, 5881, 5897, 5903, 5923, 5927, 5939, 5953, 5981, 5987, 6007, 6011, 6029, 6037,
   6043, 6047, 6053, 6067, 6073, 6079, 6089, 6091, 6101, 6113, 6121, 6131, 6133, 6143, 6151, 6163, 6173,
   6197, 6199, 6203, 6211, 6217, 6221, 6229, 6247, 6257, 6263, 6269, 6271, 6277, 6287, 6299, 6301, 6311,
   6317, 6323, 6329, 6337, 6343, 6353, 6359, 6361, 6367, 6373, 6379, 6389, 6397, 6421, 6427, 6449, 6451,
   6469, 6473, 6481, 6491, 6521, 6529, 6547, 6551, 6553, 6563, 6569, 6571, 6577, 6581, 6599, 6607, 6619,
   6637, 6653, 6659, 6661, 6673, 6679, 6689, 6691, 6701, 6703, 6709, 6719, 6733, 6737, 6761, 6763, 6779,
   6781, 6791, 6793, 6803, 6823, 6827, 6829, 6833, 6841, 6857, 6863, 6869, 6871, 6883, 6899, 6907, 6911,
   6917, 6947, 6949, 6959, 6961, 6967, 6971, 6977, 6983, 6991, 6997, 7001, 7013, 7019, 7027, 7039, 7043,
   7057, 7069, 7079, 7103, 7109, 7121, 7127, 7129, 7151, 7159, 7177, 7187, 7193, 7207, 7211, 7213, 7219,
   7229, 7237, 7243, 7247, 7253, 7283, 7297, 7307, 7309, 7321, 7331, 7333, 7349, 7351, 7369, 7393, 7411,
   7417, 7433, 7451, 7457, 7459, 7477, 7481, 7487, 7489, 7499, 7507, 7517, 7523, 7529, 7537, 7541, 7547,
   7549, 7559, 7561, 7573, 7577, 7583, 7589, 7591, 7603, 7607, 7621, 7639, 7643, 7649, 7669, 7673, 7681,
   7687, 7691, 7699, 7703, 7717, 7723, 7727, 7741, 7753, 7757, 7759, 7789, 7793, 7817, 7823, 7829, 7841,
   7853, 7867, 7873, 7877, 7879, 7883, 7901, 7907, 7919 } };

    template<typename T, size_t N>
    constexpr bool contains(const std::array<T, N>& arr, const T& v) {
        for (const auto& vv :  arr) {
            if (v == vv) {
                return true;
            }
        }

        return false;
    }

}

// concepts
namespace aerobus
{
    template <typename R>
    concept IsRing = requires {
        typename R::one;
        typename R::zero;
        typename R::template add_t<typename R::one, typename R::one>;
        typename R::template sub_t<typename R::one, typename R::one>;
        typename R::template mul_t<typename R::one, typename R::one>;
    };

    template <typename R>
    concept IsEuclideanDomain = IsRing<R> && requires {
        typename R::template div_t<typename R::one, typename R::one>;
        typename R::template mod_t<typename R::one, typename R::one>;
        typename R::template gcd_t<typename R::one, typename R::one>;
        typename R::template eq_t<typename R::one, typename R::one>;
        typename R::template pos_t<typename R::one>;

        R::template pos_v<typename R::one> == true;
        //typename R::template gt_t<typename R::one, typename R::zero>;
        R::is_euclidean_domain == true;
    };

    template<typename R>
    concept IsField = IsEuclideanDomain<R> && requires {
        R::is_field == true;
    };
}

// utilities
namespace aerobus {
    namespace internal
    {
        template<template<typename...> typename TT, typename T>
        struct is_instantiation_of :  std::false_type { };

        template<template<typename...> typename TT, typename...  Ts>
        struct is_instantiation_of<TT, TT<Ts...» :  std::true_type { };

        template<template<typename...> typename TT, typename T>
        inline constexpr bool is_instantiation_of_v = is_instantiation_of<TT, T>::value;

        template <size_t i, typename T, typename...  Ts>
        struct type_at
        {
            static_assert(i < sizeof...(Ts) + 1, "index out of range");
            using type = typename type_at<i - 1, Ts...>::type;
        };

        template <typename T, typename...  Ts> struct type_at<0, T, Ts...> {
            using type = T;
        };

        template <size_t i, typename...  Ts>
        using type_at_t = typename type_at<i, Ts...>::type;

        template<size_t i, auto x, auto...  xs>
        struct value_at {
            static_assert(i < sizeof...(xs) + 1, "index out of range");
            static constexpr auto value = value_at<i-1, xs...>::value;
        };

        template<auto x, auto...  xs>
        struct value_at<0, x, xs...> {
            static constexpr auto value = x;
        };


        template<int32_t n, int32_t i, typename E = void>
        struct _is_prime {};

        // first 1000 primes are precomputed and stored in a table
        template<int32_t n, int32_t i>
        struct _is_prime<n, i, std::enable_if_t<(n < 7920) && (contains<int32_t, 1000>(primes, n))» : std::true_type {};

        // first 1000 primes are precomputed and stored in a table
        template<int32_t n, int32_t i>
        struct _is_prime<n, i, std::enable_if_t<(n < 7920) && (!contains<int32_t, 1000>(primes, n))» : std::false_type {};

        template<int32_t n, int32_t i>
        struct _is_prime<n, i, std::enable_if_t<
            (n >= 7920) &&
            (i >= 5 && i * i <= n) &&
            (n % i == 0 || n % (i + 2) == 0)» :  std::false_type {};


        template<int32_t n, int32_t i>
        struct _is_prime<n, i, std::enable_if_t<
            (n >= 7920) &&
            (i >= 5 && i * i <= n) &&
            (n % i != 0 && n % (i + 2) != 0)» {
            static constexpr bool value = _is_prime<n, i + 6>::value;
        };

        template<int32_t n, int32_t i>
        struct _is_prime<n, i, std::enable_if_t<
            (n >= 7920) &&
            (i >= 5 && i * i > n)» :  std::true_type {};
    }

    template<int32_t n>
    struct is_prime {
        static constexpr bool value = internal::_is_prime<n, 5>::value;
    };

    namespace internal {
        template <std::size_t...  Is>
        constexpr auto index_sequence_reverse(std::index_sequence<Is...> const&)
            -> decltype(std::index_sequence<sizeof...(Is) - 1U - Is...>{});

        template <std::size_t N>
        using make_index_sequence_reverse
            = decltype(index_sequence_reverse(std::make_index_sequence<N>{}));

        template<typename Ring, typename E = void>
        struct gcd;

        template<typename Ring>
        struct gcd<Ring, std::enable_if_t<Ring::is_euclidean_domain» {
            template<typename A, typename B, typename E = void>
            struct gcd_helper {};

            // B = 0, A > 0
            template<typename A, typename B>
            struct gcd_helper<A, B, std::enable_if_t<
                ((B::is_zero_t::value) &&
                    (Ring::template gt_t<A, typename Ring::zero>::value))»
            {
                using type = A;
            };

            // B = 0, A < 0
            template<typename A, typename B>
            struct gcd_helper<A, B, std::enable_if_t<
                ((B::is_zero_t::value) &&
                    !(Ring::template gt_t<A, typename Ring::zero>::value))»
            {
                using type = typename Ring::template sub_t<typename Ring::zero, A>;
            };

            // B != 0
            template<typename A, typename B>
            struct gcd_helper<A, B, std::enable_if_t<
                (!B::is_zero_t::value)
                » {
            private:
                // A / B
                using k = typename Ring::template div_t<A, B>;
                // A - (A/B)*B = A % B
                using m = typename Ring::template sub_t<A, typename Ring::template mul_t<k, B»;
            public:
                using type = typename gcd_helper<B, m>::type;
            };

            template<typename A, typename B>
            using type = typename gcd_helper<A, B>::type;
        };
    }

    template<typename T, typename A, typename B>
    using gcd_t = typename internal::gcd<T>::template type<A, B>;
}

// quotient ring by the principal ideal generated by X
namespace aerobus {
    template<typename Ring, typename X>
    requires IsRing<Ring>
    struct Quotient {
        template <typename V>
        struct val {
        private:
            using tmp = typename Ring::template mod_t<V, X>;
        public:
            using type = std::conditional_t<
                Ring::template pos_v<tmp>,
                tmp,
                typename Ring::template sub_t<typename Ring::zero, tmp>
            >;
        };

        using zero = val<typename Ring::zero>;
        using one = val<typename Ring::one>;

        template<typename v1, typename v2>
        using add_t = val<typename Ring::template add_t<typename v1::type, typename v2::type>>;
        template<typename v1, typename v2>
        using mul_t = val<typename Ring::template mul_t<typename v1::type, typename v2::type>>;
        template<typename v1, typename v2>
        using div_t = val<typename Ring::template div_t<typename v1::type, typename v2::type>>;
        template<typename v1, typename v2>
        using mod_t = val<typename Ring::template mod_t<typename v1::type, typename v2::type>>;
        template<typename v1, typename v2>
        using eq_t = typename Ring::template eq_t<typename v1::type, typename v2::type>;
        template<typename v1>
        using pos_t = std::true_type;

        template<typename v>
        static constexpr bool pos_v = pos_t<v>::value;

        static constexpr bool is_euclidean_domain = true;

        template<auto x>
        using inject_constant_t = val<typename Ring::template inject_constant_t<x>>;

        template<typename v>
        using inject_ring_t = val<v>;
    };
}

// type_list
```

```
namespace aerobus
{
    template <typename...  Ts>
    struct type_list;

    namespace internal
    {
        template <typename T, typename...  Us>
        struct pop_front_h
        {
            using tail = type_list<Us...>;
            using head = T;
        };

        template <uint64_t index, typename L1, typename L2>
        struct split_h
        {
        private:
            static_assert(index <= L2::length, "index ouf of bounds");
            using a = typename L2::pop_front::type;
            using b = typename L2::pop_front::tail;
            using c = typename L1::template push_back<a>;

        public:
            using head = typename split_h<index - 1, c, b>::head;
            using tail = typename split_h<index - 1, c, b>::tail;
        };

        template <typename L1, typename L2>
        struct split_h<0, L1, L2>
        {
            using head = L1;
            using tail = L2;
        };

        template <uint64_t index, typename L, typename T>
        struct insert_h
        {
            static_assert(index <= L::length, "index ouf of bounds");
            using s = typename L::template split<index>;
            using left = typename s::head;
            using right = typename s::tail;
            using ll = typename left::template push_back<T>;
            using type = typename ll::template concat<right>;
        };

        template <uint64_t index, typename L>
        struct remove_h
        {
            using s = typename L::template split<index>;
            using left = typename s::head;
            using right = typename s::tail;
            using rr = typename right::pop_front::tail;
            using type = typename left::template concat<rr>;
        };
    }

    template <typename...  Ts>
    struct type_list
    {
    private:
        template <typename T>
        struct concat_h;

        template <typename...  Us>
        struct concat_h<type_list<Us...»
        {
            using type = type_list<Ts..., Us...>;
        };

    public:
        static constexpr size_t length = sizeof...(Ts);

        template <typename T>
        using push_front = type_list<T, Ts...>;

        template <uint64_t index>
        using at = internal::type_at_t<index, Ts...>;

        struct pop_front
        {
            using type = typename internal::pop_front_h<Ts...>::head;
            using tail = typename internal::pop_front_h<Ts...>::tail;
        };

        template <typename T>
        using push_back = type_list<Ts..., T>;

        template <typename U>
        using concat = typename concat_h<U>::type;

        template <uint64_t index>
        struct split
        {
        private:
            using inner = internal::split_h<index, type_list<>, type_list<Ts...»;

        public:
            using head = typename inner::head;
            using tail = typename inner::tail;
        };

        template <uint64_t index, typename T>
        using insert = typename internal::insert_h<index, type_list<Ts...>, T>::type;

        template <uint64_t index>
        using remove = typename internal::remove_h<index, type_list<Ts...»::type;
    };

    template <>
    struct type_list<>
    {
        static constexpr size_t length = 0;

        template <typename T>
        using push_front = type_list<T>;

        template <typename T>
        using push_back = type_list<T>;

        template <typename U>
        using concat = U;

        // TODO: assert index == 0
        template <uint64_t index, typename T>
        using insert = type_list<T>;
    };
}

// i32
namespace aerobus {
    struct i32 {
        using inner_type = int32_t;
        template<int32_t x>
        struct val {
            static constexpr int32_t v = x;

            template<typename valueType>
            static constexpr valueType get() { return static_cast<valueType>(x); }

            using is_zero_t = std::bool_constant<x == 0>;

            static std::string to_string() {
                return std::to_string(x);
            }

            template<typename valueRing>
            static constexpr valueRing eval(const valueRing& v) {
                return static_cast<valueRing>(x);
            }
        };

        using zero = val<0>;
        using one = val<1>;
        static constexpr bool is_field = false;
        static constexpr bool is_euclidean_domain = true;
        template<auto x>
        using inject_constant_t = val<static_cast<int32_t>(x)>;

        template<typename v>
        using inject_ring_t = v;

    private:
        template<typename v1, typename v2>
        struct add {
            using type = val<v1::v + v2::v>;
        };

        template<typename v1, typename v2>
        struct sub {
            using type = val<v1::v - v2::v>;
        };

        template<typename v1, typename v2>
        struct mul {
            using type = val<v1::v* v2::v>;
        };

        template<typename v1, typename v2>
        struct div {
            using type = val<v1::v / v2::v>;
        };

        template<typename v1, typename v2>
        struct remainder {
            using type = val<v1::v % v2::v>;
        };

        template<typename v1, typename v2>
        struct gt {
            using type = std::conditional_t<(v1::v > v2::v), std::true_type, std::false_type>;
        };

        template<typename v1, typename v2>
        struct lt {
            using type = std::conditional_t<(v1::v < v2::v), std::true_type, std::false_type>;
        };

        template<typename v1, typename v2>
        struct eq {
            using type = std::conditional_t<(v1::v == v2::v), std::true_type, std::false_type>;
        };

        template<typename v1>
        struct pos {
            using type = std::bool_constant<(v1::v > 0)>;
        };

    public:
        template<typename v1, typename v2>
        using add_t = typename add<v1, v2>::type;

        template<typename v1, typename v2>
        using sub_t = typename sub<v1, v2>::type;

        template<typename v1, typename v2>
        using mul_t = typename mul<v1, v2>::type;

        template<typename v1, typename v2>
        using div_t = typename div<v1, v2>::type;

        template<typename v1, typename v2>
        using mod_t = typename remainder<v1, v2>::type;

        template<typename v1, typename v2>
        using gt_t = typename gt<v1, v2>::type;

        template<typename v1, typename v2>
        using lt_t = typename lt<v1, v2>::type;

        template<typename v1, typename v2>
        using eq_t = typename eq<v1, v2>::type;

        template<typename v1, typename v2>
        using gcd_t = gcd_t<i32, v1, v2>;

        template<typename v>
        using pos_t = typename pos<v>::type;

        template<typename v>
        static constexpr bool pos_v = pos_t<v>::value;
    };
}

// i64
namespace aerobus {
    struct i64 {
        using inner_type = int64_t;
        template<int64_t x>
        struct val {
            static constexpr int64_t v = x;

            template<typename valueType>
            static constexpr valueType get() { return static_cast<valueType>(x); }

            using is_zero_t = std::bool_constant<x == 0>;

            static std::string to_string() {
                return std::to_string(x);
            }

            template<typename valueRing>
            static constexpr valueRing eval(const valueRing& v) {
                return static_cast<valueRing>(x);
            }
        };

        template<auto x>
        using inject_constant_t = val<static_cast<int64_t>(x)>;

        template<typename v>
        using inject_ring_t = v;

        using zero = val<0>;
        using one = val<1>;
        static constexpr bool is_field = false;
        static constexpr bool is_euclidean_domain = true;

    private:
        template<typename v1, typename v2>
        struct add {
            using type = val<v1::v + v2::v>;
        };

        template<typename v1, typename v2>
        struct sub {
            using type = val<v1::v - v2::v>;
        };

        template<typename v1, typename v2>
        struct mul {
            using type = val<v1::v* v2::v>;
        };

        template<typename v1, typename v2>
        struct div {
            using type = val<v1::v / v2::v>;
        };

        template<typename v1, typename v2>
        struct remainder {
            using type = val<v1::v% v2::v>;
        };

        template<typename v1, typename v2>
        struct gt {
            using type = std::conditional_t<(v1::v > v2::v), std::true_type, std::false_type>;
        };

        template<typename v1, typename v2>
        struct lt {
            using type = std::conditional_t<(v1::v < v2::v), std::true_type, std::false_type>;
        };

        template<typename v1, typename v2>
        struct eq {
            using type = std::conditional_t<(v1::v == v2::v), std::true_type, std::false_type>;
        };

        template<typename v>
        struct pos {
            using type = std::bool_constant<(v::v > 0)>;
        };

    public:
        template<typename v1, typename v2>
        using add_t = typename add<v1, v2>::type;

        template<typename v1, typename v2>
        using sub_t = typename sub<v1, v2>::type;

        template<typename v1, typename v2>
        using mul_t = typename mul<v1, v2>::type;

        template<typename v1, typename v2>
        using div_t = typename div<v1, v2>::type;

        template<typename v1, typename v2>
        using mod_t = typename remainder<v1, v2>::type;

        template<typename v1, typename v2>
        using gt_t = typename gt<v1, v2>::type;

        template<typename v1, typename v2>
        using lt_t = typename lt<v1, v2>::type;

        template<typename v1, typename v2>
        using eq_t = typename eq<v1, v2>::type;

        template<typename v1, typename v2>
        using gcd_t = gcd_t<i64, v1, v2>;

        template<typename v>
        using pos_t = typename pos<v>::type;

        template<typename v>
        static constexpr bool pos_v = pos_t<v>::value;
    };
}

// z/pz
namespace aerobus {
    template<int32_t p>
    struct zpz {
        using inner_type = int32_t;
        template<int32_t x>
        struct val {
            static constexpr int32_t v = x % p;

            template<typename valueType>
            static constexpr valueType get() { return static_cast<valueType>(x % p); }

            using is_zero_t = std::bool_constant<x% p == 0>;
            static std::string to_string() {
                return std::to_string(x % p);
            }

            template<typename valueRing>
            static constexpr valueRing eval(const valueRing& v) {
                return static_cast<valueRing>(x % p);
            }
        };

        template<auto x>
        using inject_constant_t = val<static_cast<int32_t>(x)>;

        using zero = val<0>;
        using one = val<1>;
        static constexpr bool is_field = is_prime<p>::value;
        static constexpr bool is_euclidean_domain = true;

    private:
        template<typename v1, typename v2>
        struct add {
            using type = val<(v1::v + v2::v) % p>;
        };

        template<typename v1, typename v2>
        struct sub {
            using type = val<(v1::v - v2::v) % p>;
        };

        template<typename v1, typename v2>
        struct mul {
            using type = val<(v1::v* v2::v) % p>;
        };

        template<typename v1, typename v2>
        struct div {
            using type = val<(v1::v% p) / (v2::v % p)>;
        };

        template<typename v1, typename v2>
        struct remainder {
            using type = val<(v1::v% v2::v) % p>;
        };

        template<typename v1, typename v2>
        struct gt {
            using type = std::conditional_t<(v1::v% p > v2::v% p), std::true_type, std::false_type>;
        };

        template<typename v1, typename v2>
        struct lt {
            using type = std::conditional_t<(v1::v% p < v2::v% p), std::true_type, std::false_type>;
        };

        template<typename v1, typename v2>
        struct eq {
            using type = std::conditional_t<(v1::v% p == v2::v % p), std::true_type, std::false_type>;
        };

        template<typename v1>
        struct pos {
            using type = std::bool_constant<(v1::v > 0)>;
        };

    public:
        template<typename v1, typename v2>
        using add_t = typename add<v1, v2>::type;

        template<typename v1, typename v2>
        using sub_t = typename sub<v1, v2>::type;

        template<typename v1, typename v2>
        using mul_t = typename mul<v1, v2>::type;

        template<typename v1, typename v2>
        using div_t = typename div<v1, v2>::type;

        template<typename v1, typename v2>
        using mod_t = typename remainder<v1, v2>::type;

        template<typename v1, typename v2>
        using gt_t = typename gt<v1, v2>::type;

        template<typename v1, typename v2>
        using lt_t = typename lt<v1, v2>::type;

        template<typename v1, typename v2>
        using eq_t = typename eq<v1, v2>::type;

        template<typename v1, typename v2>
        using gcd_t = gcd_t<i32, v1, v2>;

        template<typename v1>
        using pos_t = typename pos<v1>::type;

        template<typename v>
        static constexpr bool pos_v = pos_t<v>::value;
    };
}

// polynomial
namespace aerobus {
    // coeffN x^N + ...
    template<typename Ring, char variable_name = ’x’>
    requires IsEuclideanDomain<Ring>
    struct polynomial {
        static constexpr bool is_field = false;
        static constexpr bool is_euclidean_domain = Ring::is_euclidean_domain;

        template<typename coeffN, typename...  coeffs>
        struct val {
            static constexpr size_t degree = sizeof...(coeffs);
            using aN = coeffN;
            using strip = val<coeffs...>;
            using is_zero_t = std::bool_constant<(degree == 0) && (aN::is_zero_t::value)>;

            private:
            template<size_t index, typename E = void>
            struct coeff_at {};

            template<size_t index>
            struct coeff_at<index, std::enable_if_t<(index >= 0 && index <= sizeof...(coeffs))» {
                using type = internal::type_at_t<sizeof...(coeffs) - index, coeffN, coeffs...>;
            };

            template<size_t index>
            struct coeff_at<index, std::enable_if_t<(index < 0 || index > sizeof...(coeffs))» {
                using type = typename Ring::zero;
            };

            public:
            template<size_t index>
            using coeff_at_t = typename coeff_at<index>::type;

            static std::string to_string() {
                return string_helper<coeffN, coeffs...>::func();
            }

            template<typename valueRing>
            static constexpr valueRing eval(const valueRing& x) {
                return eval_helper<valueRing, val>::template inner<0, degree + 1>::func(static_cast<valueRing>(0), x);
            }
        };

        // specialization for constants
        template<typename coeffN>
        struct val<coeffN> {
            static constexpr size_t degree = 0;
            using aN = coeffN;
            using strip = val<coeffN>;
            using is_zero_t = std::bool_constant<aN::is_zero_t::value>;

            template<size_t index, typename E = void>
            struct coeff_at {};

            template<size_t index>
            struct coeff_at<index, std::enable_if_t<(index == 0)» {
                using type = aN;
            };

            template<size_t index>
            struct coeff_at<index, std::enable_if_t<(index < 0 || index > 0)» {
                using type = typename Ring::zero;
            };

            template<size_t index>
            using coeff_at_t = typename coeff_at<index>::type;

            static std::string to_string() {
                return string_helper<coeffN>::func();
            }

            template<typename valueRing>
            static constexpr valueRing eval(const valueRing& x) {
                return static_cast<valueRing>(aN::template get<valueRing>());
            }
        };

        using zero = val<typename Ring::zero>;
        using one = val<typename Ring::one>;
        using X = val<typename Ring::one, typename Ring::zero>;

    private:
        template<typename P, typename E = void>
        struct simplify;

        template <typename P1, typename P2, typename I>
        struct add_low;

        template<typename P1, typename P2>
        struct add {
            using type = typename simplify<typename add_low<
            P1,
            P2,
            internal::make_index_sequence_reverse<
            std::max(P1::degree, P2::degree) + 1
            »::type>::type;
        };

        template <typename P1, typename P2, typename I>
        struct sub_low;

        template <typename P1, typename P2, typename I>
        struct mul_low;

        template<typename v1, typename v2>
        struct mul {
                using type = typename mul_low<
                    v1,
                    v2,
                    internal::make_index_sequence_reverse<
                    v1::degree + v2::degree + 1
                    »::type;
        };

        template<typename coeff, size_t deg>
        struct monomial;

        template<typename v, typename E = void>
        struct derive_helper {};

        template<typename v>
        struct derive_helper<v, std::enable_if_t<v::degree == 0» {
            using type = zero;
        };

        template<typename v>
        struct derive_helper<v, std::enable_if_t<v::degree != 0» {
            using type = typename add<
                typename derive_helper<typename simplify<typename v::strip>::type>::type,
                typename monomial<
                    typename Ring::template mul_t<
                        typename v::aN,
                        typename Ring::template inject_constant_t<(v::degree)>
                    >,
                    v::degree - 1
                >::type
            >::type;
        };

        template<typename v1, typename v2, typename E = void>
        struct eq_helper {};

        template<typename v1, typename v2>
        struct eq_helper<v1, v2, std::enable_if_t<v1::degree != v2::degree» {
            using type = std::false_type;
        };


        template<typename v1, typename v2>
        struct eq_helper<v1, v2, std::enable_if_t<
            v1::degree == v2::degree &&
            (v1::degree != 0 || v2::degree != 0) &&
            std::is_same<
            typename Ring::template eq_t<typename v1::aN, typename v2::aN>,
            std::false_type
            >::value
        >
        > {
            using type = std::false_type;
        };

        template<typename v1, typename v2>
        struct eq_helper<v1, v2, std::enable_if_t<
            v1::degree == v2::degree &&
            (v1::degree != 0 || v2::degree != 0) &&
            std::is_same<
            typename Ring::template eq_t<typename v1::aN, typename v2::aN>,
            std::true_type
            >::value
        » {
            using type = typename eq_helper<typename v1::strip, typename v2::strip>::type;
        };

        template<typename v1, typename v2>
        struct eq_helper<v1, v2, std::enable_if_t<
            v1::degree == v2::degree &&
            (v1::degree == 0)
        » {
            using type = typename Ring::template eq_t<typename v1::aN, typename v2::aN>;
        };

        template<typename v1, typename v2, typename E = void>
        struct lt_helper {};

        template<typename v1, typename v2>
        struct lt_helper<v1, v2, std::enable_if_t<(v1::degree < v2::degree)» {
            using type = std::true_type;
        };

        template<typename v1, typename v2>
        struct lt_helper<v1, v2, std::enable_if_t<(v1::degree == v2::degree)» {
            using type = typename Ring::template lt_t<typename v1::aN, typename v2::aN>;
        };

        template<typename v1, typename v2>
        struct lt_helper<v1, v2, std::enable_if_t<(v1::degree > v2::degree)» {
            using type = std::false_type;
        };

        template<typename v1, typename v2, typename E = void>
        struct gt_helper {};

        template<typename v1, typename v2>
        struct gt_helper<v1, v2, std::enable_if_t<(v1::degree > v2::degree)» {
            using type = std::true_type;
        };

        template<typename v1, typename v2>
        struct gt_helper<v1, v2, std::enable_if_t<(v1::degree == v2::degree)» {
            using type = std::false_type;
        };

        template<typename v1, typename v2>
        struct gt_helper<v1, v2, std::enable_if_t<(v1::degree < v2::degree)» {
            using type = std::false_type;
        };

        // when high power is zero :  strip
        template<typename P>
        struct simplify<P, std::enable_if_t<
            std::is_same<
            typename Ring::zero,
            typename P::aN
            >::value && (P::degree > 0)
        »
        {
            using type = typename simplify<typename P::strip>::type;
        };

        // otherwise :  do nothing
        template<typename P>
        struct simplify<P, std::enable_if_t<
            !std::is_same<
            typename Ring::zero,
            typename P::aN
            >::value && (P::degree > 0)
        »
        {
            using type = P;
        };

        // do not simplify constants
        template<typename P>
        struct simplify<P, std::enable_if_t<P::degree == 0» {
            using type = P;
        };

        // addition at
        template<typename P1, typename P2, size_t index>
        struct add_at {
            using type =
                typename Ring::template add_t<typename P1::template coeff_at_t<index>, typename P2::template coeff_at_t<index»;
        };

        template<typename P1, typename P2, size_t index>
        using add_at_t = typename add_at<P1, P2, index>::type;

        template<typename P1, typename P2, std::size_t...  I>
        struct add_low<P1, P2, std::index_sequence<I...» {
            using type = val<add_at_t<P1, P2, I>...>;
        };

        // substraction at
        template<typename P1, typename P2, size_t index>
        struct sub_at {
            using type =
                typename Ring::template sub_t<typename P1::template coeff_at_t<index>, typename P2::template coeff_at_t<index»;
        };

        template<typename P1, typename P2, size_t index>
        using sub_at_t = typename sub_at<P1, P2, index>::type;

        template<typename P1, typename P2, std::size_t...  I>
        struct sub_low<P1, P2, std::index_sequence<I...» {
            using type = val<sub_at_t<P1, P2, I>...>;
        };

        template<typename P1, typename P2>
        struct sub {
            using type = typename simplify<typename sub_low<
            P1,
            P2,
            internal::make_index_sequence_reverse<
            std::max(P1::degree, P2::degree) + 1
            »::type>::type;
        };

        // multiplication at
        template<typename v1, typename v2, size_t k, size_t index, size_t stop>
        struct mul_at_loop_helper {
            using type = typename Ring::template add_t<
                typename Ring::template mul_t<
                typename v1::template coeff_at_t<index>,
                typename v2::template coeff_at_t<k - index>
                >,
                typename mul_at_loop_helper<v1, v2, k, index + 1, stop>::type
            >;
        };

        template<typename v1, typename v2, size_t k, size_t stop>
        struct mul_at_loop_helper<v1, v2, k, stop, stop> {
            using type = typename Ring::template mul_t<typename v1::template coeff_at_t<stop>, typename v2::template coeff_at_t<0»;
        };

        template <typename v1, typename v2, size_t k, typename E = void>
        struct mul_at {};

        template<typename v1, typename v2, size_t k>
        struct mul_at<v1, v2, k, std::enable_if_t<(k < 0) || (k > v1::degree + v2::degree)» {
            using type = typename Ring::zero;
        };

        template<typename v1, typename v2, size_t k>
        struct mul_at<v1, v2, k, std::enable_if_t<(k >= 0) && (k <= v1::degree + v2::degree)» {
            using type = typename mul_at_loop_helper<v1, v2, k, 0, k>::type;
        };

        template<typename P1, typename P2, size_t index>
        using mul_at_t = typename mul_at<P1, P2, index>::type;

        template<typename P1, typename P2, std::size_t...  I>
        struct mul_low<P1, P2, std::index_sequence<I...» {
            using type = val<mul_at_t<P1, P2, I>...>;
        };

        // division helper
        template< typename A, typename B, typename Q, typename R, typename E = void>
        struct div_helper {};

        template<typename A, typename B, typename Q, typename R>
        struct div_helper<A, B, Q, R, std::enable_if_t<
            (R::degree < B::degree) ||
            (R::degree == 0 && std::is_same<typename R::aN, typename Ring::zero>::value)» {
            using q_type = Q;
            using mod_type = R;
            using gcd_type = B;
        };

        template<typename A, typename B, typename Q, typename R>
        struct div_helper<A, B, Q, R, std::enable_if_t<
            (R::degree >= B::degree) &&
            !(R::degree == 0 && std::is_same<typename R::aN, typename Ring::zero>::value)» {
        private:
            using rN = typename R::aN;
            using bN = typename B::aN;
            using pT = typename monomial<typename Ring::template div_t<rN, bN>, R::degree - B::degree>::type;
            using rr = typename sub<R, typename mul<pT, B>::type>::type;
            using qq = typename add<Q, pT>::type;

        public:
            using q_type = typename div_helper<A, B, qq, rr>::q_type;
            using mod_type = typename div_helper<A, B, qq, rr>::mod_type;
            using gcd_type = rr;
        };

        template<typename A, typename B>
        struct div {
            static_assert(Ring::is_euclidean_domain, "cannot divide in that type of Ring");
            using q_type = typename div_helper<A, B, zero, A>::q_type;
            using m_type = typename div_helper<A, B, zero, A>::mod_type;
        };


        template<typename P>
        struct make_unit {
            using type = typename div<P, val<typename P::aN»::q_type;
        };

        template<typename coeff, size_t deg>
        struct monomial {
            using type = typename mul<X, typename monomial<coeff, deg - 1>::type>::type;
        };

        template<typename coeff>
        struct monomial<coeff, 0> {
            using type = val<coeff>;
        };

        template<typename valueRing, typename P>
        struct eval_helper
        {
            template<size_t index, size_t stop>
            struct inner {
                static constexpr valueRing func(const valueRing& accum, const valueRing& x) {
                    constexpr valueRing coeff = static_cast<valueRing>(P::template coeff_at_t<P::degree - index>::template get<valueRing>());
                    return eval_helper<valueRing, P>::template inner<index + 1, stop>::func(x * accum + coeff, x);
                }
            };

            template<size_t stop>
            struct inner<stop, stop> {
                static constexpr valueRing func(const valueRing& accum, const valueRing& x) {
                    return accum;
                }
            };
        };

        template<typename coeff, typename...  coeffs>
        struct string_helper {
            static std::string func() {
                std::string tail = string_helper<coeffs...>::func();
                std::string result = "";
                if (Ring::template eq_t<coeff, typename Ring::zero>::value) {
                    return tail;
                }
                else if (Ring::template eq_t<coeff, typename Ring::one>::value) {
                    if (sizeof...(coeffs) == 1) {
                        result += std::string(1, variable_name);
                    }
                    else {
                        result += std::string(1, variable_name) + "^" + std::to_string(sizeof...(coeffs));
                    }
                }
                else {
                    if (sizeof...(coeffs) == 1) {
                        result += coeff::to_string() + " " + std::string(1, variable_name);
                    }
                    else {
                        result += coeff::to_string() + " " + std::string(1, variable_name) + "^" + std::to_string(sizeof...(coeffs));
                    }
                }

                if(!tail.empty()) {
                    result += " + " + tail;
                }

                return result;
            }
        };

        template<typename coeff>
        struct string_helper<coeff> {
            static std::string func() {
                if(!std::is_same<coeff, typename Ring::zero>::value) {
                    return coeff::to_string();
                } else {
                    return "";
                }
            }
        };

    public:
        template<typename P>
        using simplify_t = typename simplify<P>::type;

        template<typename v1, typename v2>
        using add_t = typename add<v1, v2>::type;

        template<typename v1, typename v2>
        using sub_t = typename sub<v1, v2>::type;

        template<typename v1, typename v2>
        using mul_t = typename mul<v1, v2>::type;

        template<typename v1, typename v2>
        using eq_t = typename eq_helper<v1, v2>::type;

        template<typename v1, typename v2>
        using lt_t = typename lt_helper<v1, v2>::type;

        template<typename v1, typename v2>
        using gt_t = typename gt_helper<v1, v2>::type;

        template<typename v1, typename v2>
        using div_t = typename div<v1, v2>::q_type;

        template<typename v1, typename v2>
        using mod_t = typename div_helper<v1, v2, zero, v1>::mod_type;

        template<typename coeff, size_t deg>
        using monomial_t = typename monomial<coeff, deg>::type;

        template<typename v>
        using derive_t = typename derive_helper<v>::type;

        template<typename v>
        using pos_t = typename Ring::template pos_t<typename v::aN>;

        template<typename v>
        static constexpr bool pos_v = pos_t<v>::value;

        template<typename v1, typename v2>
        using gcd_t = std::conditional_t<
            Ring::is_euclidean_domain,
            typename make_unit<gcd_t<polynomial<Ring, variable_name>, v1, v2»::type,
            void>;

        template<auto x>
        using inject_constant_t = val<typename Ring::template inject_constant_t<x>>;

        template<typename v>
        using inject_ring_t = val<v>;
    };
}

// big integers
namespace aerobus {
    struct bigint {
        enum sign {
            positive,
            negative
        };

        template<sign s, uint32_t an, uint32_t...  as>
        struct val {
            template<size_t index, typename E = void>
            struct digit_at {};

            template<size_t index>
            struct digit_at<index, std::enable_if_t<(index <= sizeof...(as))» {
                static constexpr uint32_t value = internal::value_at<(sizeof...(as) - index), an, as...>::value;
            };

            template<size_t index>
            struct digit_at<index, std::enable_if_t<(index > sizeof...(as))» {
                static constexpr uint32_t value = 0;
            };

            static constexpr bool is_positive = s != sign::negative;

            using strip = val<s, as...>;
            static constexpr uint32_t aN = an;
            static constexpr size_t digits = sizeof...(as) + 1;

            static std::string to_string() {
                return std::to_string(aN) + "B^" + std::to_string(digits-1) + " + " + strip::to_string();
            }
        };

        template<typename I>
        struct is_zero {
            static constexpr bool value = I::digits == 1 && I::aN == 0;
        };

        template<typename I>
        static constexpr bool is_zero_v = is_zero<I>::value;

        template<sign s, uint32_t a0>
        struct val<s, a0> {
            using strip = val<s, a0>;
            static constexpr bool is_positive = s != sign::negative;
            static constexpr uint32_t aN = a0;
            static constexpr size_t digits = 1;
            template<size_t index, typename E = void>
            struct digit_at {};
            template<size_t index>
            struct digit_at<index, std::enable_if_t<index == 0» {
                static constexpr uint32_t value = a0;
            };

            template<size_t index>
            struct digit_at<index, std::enable_if_t<index != 0» {
                static constexpr uint32_t value = 0;
            };

            static std::string to_string() {
                return std::to_string(a0);
            }
        };

        using zero = val<sign::positive, 0>;
        using one = val<sign::positive, 1>;

    private:
        template<uint32_t x, uint32_t y, uint8_t carry_in = 0>
        struct add_digit_helper {
        private:
            static constexpr uint64_t raw = ((uint64_t) x + (uint64_t) y + (uint64_t) carry_in);
        public:
            static constexpr uint32_t value = (uint32_t)(raw & 0xFFFF’FFFF);
            static constexpr uint8_t carry_out = (uint32_t) (raw » 32);
        };

        template<typename I1, typename I2, size_t index, uint16_t carry_in = 0>
        struct add_at_helper {
            static_assert(I1::is_positive, "always add positive values");
            static_assert(I2::is_positive, "always add positive values");
        private:
            static constexpr uint32_t d1 = I1::template digit_at<index>::value;
            static constexpr uint32_t d2 = I2::template digit_at<index>::value;
        public:
            static constexpr uint32_t value = add_digit_helper<d1, d2, carry_in>::value;
            static constexpr uint8_t carry_out = add_digit_helper<d1, d2, carry_in>::carry_out;
        };

        template<typename I, typename E = void>
        struct simplify {};

        template<typename I>
        struct simplify<I, std::enable_if_t<I::aN == 0» {
            using type = typename I::strip;
        };

        template<typename I>
        struct simplify<I, std::enable_if_t<I::aN != 0» {
            using type = I;
        };

    public:

        template<typename I>
        using simplify_t = typename simplify<I>::type;

        // exposed for testing -- DO NOT USE
        template<typename I1, typename I2, size_t index, uint8_t carry_in = 0>
        static constexpr uint32_t add_at_digit = add_at_helper<I1, I2, index, carry_in>::value;
        template<typename I1, typename I2, size_t index, uint8_t carry_in = 0>
        static constexpr uint8_t add_at_carry = add_at_helper<I1, I2, index, carry_in>::carry_out;

        // exposed for testing -- DO NOT USE
        template<typename I1, typename I2, size_t index>
        struct add_low_helper {
            private:
            using helper = add_at_helper<I1, I2, index, add_low_helper<I1, I2, index-1>::carry_out>;
            public:
            static constexpr uint32_t digit = helper::value;
            static constexpr uint8_t carry_out = helper::carry_out;
        };

        // exposed for testing -- DO NOT USE
        template<typename I1, typename I2>
        struct add_low_helper<I1, I2, 0> {
            static constexpr uint32_t digit = add_at_helper<I1, I2, 0, 0>::value;
            static constexpr uint32_t carry_out = add_at_helper<I1, I2, 0, 0>::carry_out;
        };

        template<typename I1, typename I2, typename I>
        struct add_low {};

        template<typename I1, typename I2, std::size_t...  I>
        struct add_low<I1, I2, std::index_sequence<I...» {
            static_assert(I1::is_positive, "add works on positive values");
            static_assert(I2::is_positive, "add works on positive values");
            using type = val<sign::positive, add_low_helper<I1, I2, I>::digit...>;
        };

        template<typename I1, typename I2>
        struct add {
            static_assert(I1::is_positive, "add works on positive values");
            static_assert(I2::is_positive, "add works on positive values");
            using type = simplify_t<
                typename add_low<
                        I1,
                        I2,
                        typename internal::make_index_sequence_reverse<std::max(I1::digits, I2::digits)
 + 1>
```

```
                    >::type>;
        };
    };
}

// fraction field
namespace aerobus {
    namespace internal {
        template<typename Ring, typename E = void>
        requires IsEuclideanDomain<Ring>
        struct _FractionField {};

        template<typename Ring>
        requires IsEuclideanDomain<Ring>
        struct _FractionField<Ring, std::enable_if_t<Ring::is_euclidean_domain»
        {
            static constexpr bool is_field = true;
            static constexpr bool is_euclidean_domain = true;

            private:
            template<typename val1, typename val2, typename E = void>
            struct to_string_helper {};

            template<typename val1, typename val2>
            struct to_string_helper <val1, val2,
                std::enable_if_t<
                Ring::template eq_t<
                val2, typename Ring::one
                >::value
                >
            > {
                static std::string func() {
                    return val1::to_string();
                }
            };

            template<typename val1, typename val2>
            struct to_string_helper<val1, val2,
                std::enable_if_t<
                !Ring::template eq_t<
                val2,
                typename Ring::one
                >::value
                >
            > {
                static std::string func() {
                    return "(" + val1::to_string() + ") / (" + val2::to_string() + ")";
                }
            };

            public:
            template<typename val1, typename val2>
            struct val {
                using x = val1;
                using y = val2;
                using is_zero_t = typename val1::is_zero_t;
                using ring_type = Ring;
                using field_type = _FractionField<Ring>;

                static constexpr bool is_integer = std::is_same<val2, typename Ring::one>::value;

                template<typename valueType>
                static constexpr valueType get() { return static_cast<valueType>(x::v) /
 static_cast<valueType>(y::v); }

                static std::string to_string() {
                    return to_string_helper<val1, val2>::func();
                }

                template<typename valueRing>
                static constexpr valueRing eval(const valueRing& v) {
                    return x::eval(v) / y::eval(v);
                }
            };

            using zero = val<typename Ring::zero, typename Ring::one>;
            using one = val<typename Ring::one, typename Ring::one>;

            template<typename v>
            using inject_t = val<v, typename Ring::one>;

            template<auto x>
            using inject_constant_t = val<typename Ring::template inject_constant_t<x>, typename
 Ring::one>;

            template<typename v>
            using inject_ring_t = val<typename Ring::template inject_ring_t<v>, typename Ring::one>;

            using ring_type = Ring;

        private:
            template<typename v, typename E = void>
            struct simplify {};

            // x = 0
            template<typename v>
            struct simplify<v, std::enable_if_t<v::x::is_zero_t::value» {
                using type = typename _FractionField<Ring>::zero;
            };

            // x != 0
            template<typename v>
            struct simplify<v, std::enable_if_t<!v::x::is_zero_t::value» {

            private:
                using _gcd = typename Ring::template gcd_t<typename v::x, typename v::y>;
                using newx = typename Ring::template div_t<typename v::x, _gcd>;
                using newy = typename Ring::template div_t<typename v::y, _gcd>;

                using posx = std::conditional_t<!Ring::template pos_v<newy>, typename Ring::template sub_t<typename Ring::zero, newx>, newx>;
                using posy = std::conditional_t<!Ring::template pos_v<newy>, typename Ring::template sub_t<typename Ring::zero, newy>, newy>;
            public:
                using type = typename _FractionField<Ring>::template val<posx, posy>;
            };

        public:
            template<typename v>
            using simplify_t = typename simplify<v>::type;

        private:

            template<typename v1, typename v2>
            struct add {
            private:
                using a = typename Ring::template mul_t<typename v1::x, typename v2::y>;
                using b = typename Ring::template mul_t<typename v1::y, typename v2::x>;
                using dividend = typename Ring::template add_t<a, b>;
                using diviser = typename Ring::template mul_t<typename v1::y, typename v2::y>;
                using g = typename Ring::template gcd_t<dividend, diviser>;

            public:
                using type = typename _FractionField<Ring>::template simplify_t<val<dividend, diviser»;
            };

            template<typename v>
            struct pos {
                using type = std::conditional_t<
                    (Ring::template pos_v<typename v::x> && Ring::template pos_v<typename v::y>) ||
                    (!Ring::template pos_v<typename v::x> && !Ring::template pos_v<typename v::y>),
                    std::true_type,
                    std::false_type>;

            };

            template<typename v1, typename v2>
            struct sub {
            private:
                using a = typename Ring::template mul_t<typename v1::x, typename v2::y>;
                using b = typename Ring::template mul_t<typename v1::y, typename v2::x>;
                using dividend = typename Ring::template sub_t<a, b>;
                using diviser = typename Ring::template mul_t<typename v1::y, typename v2::y>;
                using g = typename Ring::template gcd_t<dividend, diviser>;

            public:
                using type = typename _FractionField<Ring>::template simplify_t<val<dividend, diviser»;
            };

            template<typename v1, typename v2>
            struct mul {
            private:
                using a = typename Ring::template mul_t<typename v1::x, typename v2::x>;
                using b = typename Ring::template mul_t<typename v1::y, typename v2::y>;

            public:
                using type = typename _FractionField<Ring>::template simplify_t<val<a, b»;
            };

            template<typename v1, typename v2, typename E = void>
            struct div {};

            template<typename v1, typename v2>
            struct div<v1, v2, std::enable_if_t<!std::is_same<v2, typename
```

```
 _FractionField<Ring>::zero>::value»  {
            private:
                using a = typename Ring::template mul_t<typename v1::x, typename v2::y>;
                using b = typename Ring::template mul_t<typename v1::y, typename v2::x>;

            public:
                using type = typename _FractionField<Ring>::template simplify_t<val<a, b»;
            };

            template<typename v1, typename v2>
            struct div<v1, v2, std::enable_if_t<
                std::is_same<zero, v1>::value && std::is_same<v2, zero>::value»  {
                using type = one;
            };

            template<typename v1, typename v2>
            struct eq {
                using type = std::conditional_t<
                        std::is_same<typename simplify_t<v1>::x, typename simplify_t<v2>::x>::value &&
                        std::is_same<typename simplify_t<v1>::y, typename simplify_t<v2>::y>::value,
                    std::true_type,
                    std::false_type>;
            };

            template<typename TL, typename E = void>
            struct vadd {};

            template<typename TL>
            struct vadd<TL, std::enable_if_t<(TL::length > 1)» {
                using head = typename TL::pop_front::type;
                using tail = typename TL::pop_front::tail;
                using type = typename add<head, typename vadd<tail>::type>::type;
            };

            template<typename TL>
            struct vadd<TL, std::enable_if_t<(TL::length == 1)» {
                using type = typename TL::template at<0>;
            };

            template<typename...  vals>
            struct vmul {};

            template<typename v1, typename...  vals>
            struct vmul<v1, vals...> {
                using type = typename mul<v1, typename vmul<vals...>::type>::type;
            };

            template<typename v1>
            struct vmul<v1> {
                using type = v1;
            };


            template<typename v1, typename v2, typename E = void>
            struct gt;

            template<typename v1, typename v2>
            struct gt<v1, v2, std::enable_if_t<
                (eq<v1, v2>::type::value)
                » {
                using type = std::false_type;
            };

            template<typename v1, typename v2>
            struct gt<v1, v2, std::enable_if_t<
                (!eq<v1, v2>::type::value) &&
                (!pos<v1>::type::value) && (!pos<v2>::type::value)
                » {
                using type = typename gt<
                    typename sub<zero, v1>::type, typename sub<zero, v2>::type
                >::type;
            };

            template<typename v1, typename v2>
            struct gt<v1, v2, std::enable_if_t<
                (!eq<v1, v2>::type::value) &&
                (pos<v1>::type::value) && (!pos<v2>::type::value)
                » {
                using type = std::true_type;
            };

            template<typename v1, typename v2>
            struct gt<v1, v2, std::enable_if_t<
                (!eq<v1, v2>::type::value) &&
                (!pos<v1>::type::value) && (pos<v2>::type::value)
                » {
                using type = std::false_type;
            };

            template<typename v1, typename v2>
            struct gt<v1, v2, std::enable_if_t<
                (!eq<v1, v2>::type::value) &&
                (pos<v1>::type::value) && (pos<v2>::type::value)
                » {
                using type = typename Ring::template gt_t<
                    typename Ring::template mul_t<v1::x, v2::y>,
                    typename Ring::template mul_t<v2::y, v2::x>
                >;
            };


        public:
            template<typename v1, typename v2>
            using add_t = typename add<v1, v2>::type;
            template<typename v1, typename v2>
            using mod_t = zero;
            template<typename v1, typename v2>
            using gcd_t = v1;
            template<typename...  vs>
            using vadd_t = typename vadd<vs...>::type;
            template<typename...  vs>
            using vmul_t = typename vmul<vs...>::type;
            template<typename v1, typename v2>
            using sub_t = typename sub<v1, v2>::type;
            template<typename v1, typename v2>
            using mul_t = typename mul<v1, v2>::type;
            template<typename v1, typename v2>
            using div_t = typename div<v1, v2>::type;
            template<typename v1, typename v2>
            using eq_t = typename eq<v1, v2>::type;
            template<typename v1, typename v2>
            using gt_t = typename gt<v1, v2>::type;
            template<typename v1>
            using pos_t = typename pos<v1>::type;

            template<typename v>
            static constexpr bool pos_v = pos_t<v>::value;
        };

        template<typename Ring, typename E = void>
        requires IsEuclideanDomain<Ring>
        struct FractionFieldImpl {};

        // fraction field of a field is the field itself
        template<typename Field>
        requires IsEuclideanDomain<Field>
        struct FractionFieldImpl<Field, std::enable_if_t<Field::is_field» {
            using type = Field;
            template<typename v>
            using inject_t = v;
        };

        // fraction field of a ring is the actual fraction field
        template<typename Ring>
        requires IsEuclideanDomain<Ring>
        struct FractionFieldImpl<Ring, std::enable_if_t<!Ring::is_field» {
            using type = _FractionField<Ring>;
        };
    }

    template<typename Ring>
    requires IsEuclideanDomain<Ring>
    using FractionField = typename internal::FractionFieldImpl<Ring>::type;
}

// short names for common types
namespace aerobus {
    using q32 = FractionField<i32>;
    using fpq32 = FractionField<polynomial<q32»;
    using q64 = FractionField<i64>;
    using pi64 = polynomial<i64>;
    using fpq64 = FractionField<polynomial<q64»;
    template<typename Ring, typename v1, typename v2>
    using makefraction_t = typename FractionField<Ring>::template val<v1, v2>;

    template<typename Ring, typename v1, typename v2>
    using addfractions_t = typename FractionField<Ring>::template add_t<v1, v2>;
    template<typename Ring, typename v1, typename v2>
    using mulfractions_t = typename FractionField<Ring>::template mul_t<v1, v2>;
}

// taylor series and common integers (factorial, bernouilli...)  appearing in taylor coefficients
namespace aerobus {
    namespace internal {
        template<typename T, size_t x, typename E = void>
        struct factorial {};

        template<typename T, size_t x>
        struct factorial<T, x, std::enable_if_t<(x > 0)» {
        private:
            template<typename, size_t, typename>
            friend struct factorial;
        public:
            using type = typename T::template mul_t<typename T::template val<x>, typename factorial<T, x - 1>::type>;
            static constexpr typename T::inner_type value = type::template get<typename T::inner_type>();
        };

        template<typename T>
        struct factorial<T, 0> {
        public:
            using type = typename T::one;
            static constexpr typename T::inner_type value = type::template get<typename T::inner_type>();
        };
    }

    template<typename T, size_t i>
    using factorial_t = typename internal::factorial<T, i>::type;

    template<typename T, size_t i>
    inline constexpr typename T::inner_type factorial_v = internal::factorial<T, i>::value;

    namespace internal {
        template<typename T, size_t k, size_t n, typename E = void>
        struct combination_helper {};

        template<typename T, size_t k, size_t n>
        struct combination_helper<T, k, n, std::enable_if_t<(n >= 0 && k <= (n / 2) && k > 0)» {
            using type = typename FractionField<T>::template mul_t<
                typename combination_helper<T, k - 1, n - 1>::type,
                makefraction_t<T, typename T::template val<n>, typename T::template val<k»>;
        };

        template<typename T, size_t k, size_t n>
        struct combination_helper<T, k, n, std::enable_if_t<(n >= 0 && k > (n / 2) && k > 0)» {
            using type = typename combination_helper<T, n - k, n>::type;
        };

        template<typename T, size_t n>
        struct combination_helper<T, 0, n> {
            using type = typename FractionField<T>::one;
        };

        template<typename T, size_t k, size_t n>
        struct combination {
            using type = typename internal::combination_helper<T, k, n>::type::x;
            static constexpr typename T::inner_type value = internal::combination_helper<T, k, n>::type::template get<typename T::inner_type>();
        };
    }

    template<typename T, size_t k, size_t n>
    using combination_t = typename internal::combination<T, k, n>::type;

    template<typename T, size_t k, size_t n>
    inline constexpr typename T::inner_type combination_v = internal::combination<T, k, n>::value;

    namespace internal {
        template<typename T, size_t m>
        struct bernouilli;

        template<typename T, typename accum, size_t k, size_t m>
        struct bernouilli_helper {
            using type = typename bernouilli_helper<
                T,
                addfractions_t<T,
                    accum,
                    mulfractions_t<T,
                        makefraction_t<T,
                            combination_t<T, k, m + 1>,
                            typename T::one>,
                        typename bernouilli<T, k>::type
                    >
                >,
                k + 1,
                m>::type;
        };

        template<typename T, typename accum, size_t m>
        struct bernouilli_helper<T, accum, m, m>
        {
            using type = accum;
        };



        template<typename T, size_t m>
        struct bernouilli {
            using type = typename FractionField<T>::template mul_t<
                typename internal::bernouilli_helper<T, typename FractionField<T>::zero, 0, m>::type,
                makefraction_t<T,
                typename T::template val<static_cast<typename T::inner_type>(-1)>,
                typename T::template val<static_cast<typename T::inner_type>(m + 1)>
                >
            >;

            template<typename floatType>
            static constexpr floatType value = type::template get<floatType>();
        };

        template<typename T>
        struct bernouilli<T, 0> {
            using type = typename FractionField<T>::one;

            template<typename floatType>
            static constexpr floatType value = type::template get<floatType>();
        };
    }

    template<typename T, size_t n>
    using bernouilli_t = typename internal::bernouilli<T, n>::type;

    template<typename FloatType, typename T, size_t n >
    inline constexpr FloatType bernouilli_v = internal::bernouilli<T, n>::template value<FloatType>;

    namespace internal {
        template<typename T, int k, typename E = void>
        struct alternate {};

        template<typename T, int k>
        struct alternate<T, k, std::enable_if_t<k % 2 == 0» {
            using type = typename T::one;
            static constexpr typename T::inner_type value = type::template get<typename T::inner_type>();
        };

        template<typename T, int k>
        struct alternate<T, k, std::enable_if_t<k % 2 != 0» {
            using type = typename T::template sub_t<typename T::zero, typename T::one>;
            static constexpr typename T::inner_type value = type::template get<typename T::inner_type>();
        };
    }

    template<typename T, int k>
    using alternate_t = typename internal::alternate<T, k>::type;

    template<typename T, size_t k>
    inline constexpr typename T::inner_type alternate_v = internal::alternate<T, k>::value;

    // pow
    namespace internal {
        template<typename T, auto p, auto n>
        struct pow {
            using type = typename T::template mul_t<typename T::template val<p>, typename pow<T, p, n - 1>::type>;
        };

        template<typename T, auto p>
        struct pow<T, p, 0> { using type = typename T::one; };
    }

    template<typename T, auto p, auto n>
    using pow_t = typename internal::pow<T, p, n>::type;

    namespace internal {
        template<typename, template<typename, size_t> typename, class>
        struct make_taylor_impl;

        template<typename T, template<typename, size_t> typename coeff_at, size_t...  Is>
        struct make_taylor_impl<T, coeff_at, std::integer_sequence<size_t, Is...» {
            using type = typename polynomial<FractionField<T>»::template val<typename coeff_at<T, Is>::type...>;
        };
    }

    // generic taylor serie, depending on coefficients
    template<typename T, template<typename, size_t index> typename coeff_at, size_t deg>
    using taylor = typename internal::make_taylor_impl<T, coeff_at,
 internal::make_index_sequence_reverse<deg + 1>»::type;

    namespace internal {
        template<typename T, size_t i>
        struct exp_coeff {
            using type = makefraction_t<T, typename T::one, factorial_t<T, i»;
        };

        template<typename T, size_t i, typename E = void>
        struct sin_coeff_helper {};

        template<typename T, size_t i>
        struct sin_coeff_helper<T, i, std::enable_if_t<(i & 1) == 0» {
            using type = typename FractionField<T>::zero;
        };

        template<typename T, size_t i>
        struct sin_coeff_helper<T, i, std::enable_if_t<(i & 1) == 1» {
            using type = makefraction_t<T, alternate_t<T, i / 2>, factorial_t<T, i»;
        };

        template<typename T, size_t i>
        struct sin_coeff {
            using type = typename sin_coeff_helper<T, i>::type;
        };

        template<typename T, size_t i, typename E = void>
        struct sh_coeff_helper {};

        template<typename T, size_t i>
        struct sh_coeff_helper<T, i, std::enable_if_t<(i & 1) == 0» {
            using type = typename FractionField<T>::zero;
        };

        template<typename T, size_t i>
        struct sh_coeff_helper<T, i, std::enable_if_t<(i & 1) == 1» {
            using type = makefraction_t<T, typename T::one, factorial_t<T, i»;
        };

        template<typename T, size_t i>
        struct sh_coeff {
            using type = typename sh_coeff_helper<T, i>::type;
        };

        template<typename T, size_t i, typename E = void>
        struct cos_coeff_helper {};

        template<typename T, size_t i>
        struct cos_coeff_helper<T, i, std::enable_if_t<(i & 1) == 1» {
            using type = typename FractionField<T>::zero;
        };

        template<typename T, size_t i>
        struct cos_coeff_helper<T, i, std::enable_if_t<(i & 1) == 0» {
            using type = makefraction_t<T, alternate_t<T, i / 2>, factorial_t<T, i»;
        };

        template<typename T, size_t i>
        struct cos_coeff {
            using type = typename cos_coeff_helper<T, i>::type;
        };

        template<typename T, size_t i, typename E = void>
        struct cosh_coeff_helper {};

        template<typename T, size_t i>
        struct cosh_coeff_helper<T, i, std::enable_if_t<(i & 1) == 1» {
            using type = typename FractionField<T>::zero;
        };

        template<typename T, size_t i>
        struct cosh_coeff_helper<T, i, std::enable_if_t<(i & 1) == 0» {
            using type = makefraction_t<T, typename T::one, factorial_t<T, i»;
        };

        template<typename T, size_t i>
        struct cosh_coeff {
            using type = typename cosh_coeff_helper<T, i>::type;
        };

        template<typename T, size_t i>
        struct geom_coeff { using type = typename FractionField<T>::one; };


        template<typename T, size_t i, typename E = void>
        struct atan_coeff_helper;

        template<typename T, size_t i>
        struct atan_coeff_helper<T, i, std::enable_if_t<(i & 1) == 1» {
            using type = makefraction_t<T, alternate_t<T, i / 2>, typename T::template val<i»;
        };

        template<typename T, size_t i>
        struct atan_coeff_helper<T, i, std::enable_if_t<(i & 1) == 0» {
            using type = typename FractionField<T>::zero;
        };

        template<typename T, size_t i>
        struct atan_coeff { using type = typename atan_coeff_helper<T, i>::type; };

        template<typename T, size_t i, typename E = void>
        struct asin_coeff_helper;

        template<typename T, size_t i>
        struct asin_coeff_helper<T, i, std::enable_if_t<(i & 1) == 1»
        {
            using type = makefraction_t<T,
                factorial_t<T, i - 1>,
                typename T::template mul_t<
                    typename T::template val<i>,
                    T::template mul_t<
                        pow_t<T, 4, i / 2>,
                        pow<T, factorial<T, i / 2>::value, 2
                    >
                >
                »;
        };

        template<typename T, size_t i>
        struct asin_coeff_helper<T, i, std::enable_if_t<(i & 1) == 0»
        {
            using type = typename FractionField<T>::zero;
        };

        template<typename T, size_t i>
        struct asin_coeff {
            using type = typename asin_coeff_helper<T, i>::type;
        };

        template<typename T, size_t i>
        struct lnp1_coeff {
            using type = makefraction_t<T,
                alternate_t<T, i + 1>,
                typename T::template val<i»;
        };

        template<typename T>
        struct lnp1_coeff<T, 0> { using type = typename FractionField<T>::zero; };

        template<typename T, size_t i, typename E = void>
        struct asinh_coeff_helper;

        template<typename T, size_t i>
        struct asinh_coeff_helper<T, i, std::enable_if_t<(i & 1) == 1»
        {
            using type = makefraction_t<T,
                typename T::template mul_t<
                    alternate_t<T, i / 2>,
                    factorial_t<T, i - 1>
                >,
                typename T::template mul_t<
                    T::template mul_t<
                        typename T::template val<i>,
                        pow_t<T, (factorial<T, i / 2>::value), 2>
                    >,
                    pow_t<T, 4, i / 2>
                >
            >;
        };

        template<typename T, size_t i>
        struct asinh_coeff_helper<T, i, std::enable_if_t<(i & 1) == 0»
        {
            using type = typename FractionField<T>::zero;
        };

        template<typename T, size_t i>
        struct asinh_coeff {
            using type = typename asinh_coeff_helper<T, i>::type;
        };

        template<typename T, size_t i, typename E = void>
        struct atanh_coeff_helper;

        template<typename T, size_t i>
        struct atanh_coeff_helper<T, i, std::enable_if_t<(i & 1) == 1»
        {
            // 1/i
            using type = typename FractionField<T>::  template val<
                typename T::one,
                typename T::template val<static_cast<typename T::inner_type>(i)»;
        };

        template<typename T, size_t i>
        struct atanh_coeff_helper<T, i, std::enable_if_t<(i & 1) == 0»
        {
            using type = typename FractionField<T>::zero;
        };

        template<typename T, size_t i>
        struct atanh_coeff {
            using type = typename asinh_coeff_helper<T, i>::type;
        };

        template<typename T, size_t i, typename E = void>
        struct tan_coeff_helper;

        template<typename T, size_t i>
        struct tan_coeff_helper<T, i, std::enable_if_t<(i % 2) == 0» {
            using type = typename FractionField<T>::zero;
        };

        template<typename T, size_t i>
        struct tan_coeff_helper<T, i, std::enable_if_t<(i % 2) != 0» {
        private:
            // 4^((i+1)/2)
            using _4p = typename FractionField<T>::template inject_t<pow_t<T, 4, (i + 1) / 2»;
            // 4^((i+1)/2) - 1
            using _4pm1 = typename FractionField<T>::template sub_t<_4p, typename FractionField<T>::one>;
            // (-1)^((i-1)/2)
            using altp = typename FractionField<T>::template inject_t<alternate_t<T, (i - 1) / 2»;
            using dividend = typename FractionField<T>::template mul_t<
                altp,
                FractionField<T>::template mul_t<
                _4p,
                FractionField<T>::template mul_t<
                _4pm1,
                bernouilli_t<T, (i + 1)>
                >
                >
            >;
        public:
            using type = typename FractionField<T>::template div_t<dividend,
                typename FractionField<T>::template inject_t<factorial_t<T, i + 1»>;
        };

        template<typename T, size_t i>
        struct tan_coeff {
            using type = typename tan_coeff_helper<T, i>::type;
        };

        template<typename T, size_t i, typename E = void>
        struct tanh_coeff_helper;

        template<typename T, size_t i>
        struct tanh_coeff_helper<T, i, std::enable_if_t<(i % 2) == 0» {
            using type = typename FractionField<T>::zero;
        };

        template<typename T, size_t i>
        struct tanh_coeff_helper<T, i, std::enable_if_t<(i % 2) != 0» {
        private:
            using _4p = typename FractionField<T>::template inject_t<pow_t<T, 4, (i + 1) / 2»;
            using _4pm1 = typename FractionField<T>::template sub_t<_4p, typename FractionField<T>::one>;
            using dividend =
                typename FractionField<T>::template mul_t<
                _4p,
                typename FractionField<T>::template mul_t<
                _4pm1,
                bernouilli_t<T, (i + 1)>
                >
                >::type;
        public:
            using type = typename FractionField<T>::template div_t<dividend,
                FractionField<T>::template inject_t<factorial_t<T, i + 1»>;
        };

        template<typename T, size_t i>
        struct tanh_coeff {
            using type = typename tanh_coeff_helper<T, i>::type;
        };
    }

    template<typename T, size_t deg>
    using exp = taylor<T, internal::exp_coeff, deg>;

    template<typename T, size_t deg>
    using expm1 = typename polynomial<FractionField<T>>::template sub_t<
        exp<T, deg>,
        typename polynomial<FractionField<T>>::one>;

    template<typename T, size_t deg>
    using lnp1 = taylor<T, internal::lnp1_coeff, deg>;

    template<typename T, size_t deg>
    using atan = taylor<T, internal::atan_coeff, deg>;

    template<typename T, size_t deg>
    using sin = taylor<T, internal::sin_coeff, deg>;

    template<typename T, size_t deg>
    using sinh = taylor<T, internal::sh_coeff, deg>;

    template<typename T, size_t deg>
    using cosh = taylor<T, internal::cosh_coeff, deg>;

    template<typename T, size_t deg>
    using cos = taylor<T, internal::cos_coeff, deg>;

    template<typename T, size_t deg>
    using geometric_sum = taylor<T, internal::geom_coeff, deg>;

    template<typename T, size_t deg>
    using asin = taylor<T, internal::asin_coeff, deg>;

    template<typename T, size_t deg>
    using asinh = taylor<T, internal::asinh_coeff, deg>;

    template<typename T, size_t deg>
    using atanh = taylor<T, internal::atanh_coeff, deg>;

    template<typename T, size_t deg>
    using tan = taylor<T, internal::tan_coeff, deg>;

    template<typename T, size_t deg>
    using tanh = taylor<T, internal::tanh_coeff, deg>;
}

// continued fractions
namespace aerobus {
    template<int64_t...  values>
    struct ContinuedFraction {};

    template<int64_t a0>
    struct ContinuedFraction<a0> {
        using type = typename q64::template inject_constant_t<a0>;
        static constexpr double val = type::template get<double>();
    };

    template<int64_t a0, int64_t...  rest>
    struct ContinuedFraction<a0, rest...> {
        using type = q64::template add_t<
                typename q64::template inject_constant_t<a0>,
                typename q64::template div_t<
                    typename q64::one,
                    typename ContinuedFraction<rest...>::type
                >>;
        static constexpr double val = type::template get<double>();
    };

    using PI_fraction =
 ContinuedFraction<3, 7, 15, 1, 292, 1, 1, 1, 2, 1, 3, 1, 14, 2, 1, 1, 2, 2, 2, 2, 1>;
    using E_fraction =
 ContinuedFraction<2, 1, 2, 1, 1, 4, 1, 1, 6, 1, 1, 8, 1, 1, 10, 1, 1, 12, 1, 1, 14, 1, 1>;
    using SQRT2_fraction =
 ContinuedFraction<1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2>;
    using SQRT3_fraction =
 ContinuedFraction<1, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2>;
}

// known polynomials
namespace aerobus {
    namespace internal {
        template<int kind, int deg>
        struct chebyshev_helper {
            using type = typename pi64::template sub_t<
                typename pi64::template mul_t<
                    typename pi64::template mul_t<
                        pi64::inject_constant_t<2>,
                        typename pi64::X
                    >,
                    typename chebyshev_helper<kind, deg-1>::type
                >,
                typename chebyshev_helper<kind, deg-2>::type
            >;
        };

        template<>
        struct chebyshev_helper<1, 0> {
            using type = typename pi64::one;
        };

        template<>
        struct chebyshev_helper<1, 1> {
            using type = typename pi64::X;
        };

        template<>
        struct chebyshev_helper<2, 0> {
            using type = typename pi64::one;
        };

        template<>
        struct chebyshev_helper<2, 1> {
            using type = typename pi64::template mul_t<
                            typename pi64::inject_constant_t<2>,
                            typename pi64::X>;
        };
    }

    template<size_t deg>
    using chebyshev_T = typename internal::chebyshev_helper<1, deg>::type;

    template<size_t deg>
    using chebyshev_U = typename internal::chebyshev_helper<2, deg>::type;
}
```

Chapter 7

Example Documentation

7.1 i32::template

inject a native constant

inject a native constant

Template Parameters

x	inject_constant_2<2> -> i32::template val<2>

7.2 i64::template

injects constant as an i64 value

injects constant as an i64 value

Template Parameters

x	inject_constant_t<2>

7.3 polynomial

makes the constant (native type) polynomial a_0

makes the constant (native type) polynomial a_0

Template Parameters

x	<i32>::template inject_constant_t<2>

7.4 PI_fraction::val

representation of PI as a continued fraction -> 3.14...

7.5 E_fraction::val

approximation of e -> 2.718...

approximation of e -> 2.718...

Index

add_t
    aerobus::polynomial< Ring, variable_name >, 20
aerobus::bigint, 10
aerobus::bigint::add< I1, I2 >, 9
aerobus::bigint::add_low< I1, I2, I >, 9
aerobus::bigint::add_low< I1, I2, std::index_sequence< I... > >, 9
aerobus::bigint::add_low_helper< I1, I2, 0 >, 10
aerobus::bigint::add_low_helper< I1, I2, index >, 10
aerobus::bigint::is_zero< I >, 18
aerobus::bigint::val< s, a0 >, 34
aerobus::bigint::val< s, a0 >::digit_at< index, E >, 13
aerobus::bigint::val< s, a0 >::digit_at< index, std::enable_if_t< index !=0 > >, 13
aerobus::bigint::val< s, a0 >::digit_at< index, std::enable_if_t< index==0 > >, 13
aerobus::bigint::val< s, an, as >, 27
aerobus::bigint::val< s, an, as >::digit_at< index, E >, 13
aerobus::bigint::val< s, an, as >::digit_at< index, std::enable_if_t<(index > sizeof...(as))> >, 14
aerobus::bigint::val< s, an, as >::digit_at< index, std::enable_if_t<(index<=sizeof...(as))> >, 14
aerobus::ContinuedFraction< a0 >, 12
aerobus::ContinuedFraction< a0, rest... >, 12
aerobus::ContinuedFraction< values >, 12
aerobus::i32, 14
aerobus::i32::val< x >, 28
    eval, 28
    get, 29
aerobus::i64, 16
aerobus::i64::val< x >, 29
    eval, 30
    get, 30
aerobus::is_prime< n >, 18
aerobus::IsEuclideanDomain, 7
aerobus::IsField, 7
aerobus::IsRing, 8
aerobus::polynomial< Ring, variable_name >, 19
    add_t, 20
    derive_t, 20
    div_t, 21
    eq_t, 21
    gcd_t, 21
    gt_t, 22
    lt_t, 22
    mod_t, 22
    monomial_t, 23
    mul_t, 23
    pos_t, 23
    simplify_t, 24
    sub_t, 24
aerobus::polynomial< Ring, variable_name >::eval_helper< valueRing, P >::inner< index, stop >, 17
aerobus::polynomial< Ring, variable_name >::eval_helper< valueRing, P >::inner< stop, stop >, 18
aerobus::polynomial< Ring, variable_name >::val< coeffN >, 33
aerobus::polynomial< Ring, variable_name >::val< coeffN >::coeff_at< index, E >, 11
aerobus::polynomial< Ring, variable_name >::val< coeffN >::coeff_at< index, std::enable_if_t<(index< 0 | | index > 0)> >, 11
aerobus::polynomial< Ring, variable_name >::val< coeffN >::coeff_at< index, std::enable_if_t<(index==0)> >, 11
aerobus::polynomial< Ring, variable_name >::val< coeffN, coeffs >, 31
    coeff_at_t, 31
    eval, 32
    to_string, 32
aerobus::Quotient< Ring, X >, 25
aerobus::Quotient< Ring, X >::val< V >, 33
aerobus::type_list< Ts >, 26
aerobus::type_list< Ts >::pop_front, 25
aerobus::type_list< Ts >::split< index >, 26
aerobus::type_list<>, 27
aerobus::zpz< p >, 35
aerobus::zpz< p >::val< x >, 33

coeff_at_t
    aerobus::polynomial< Ring, variable_name >::val< coeffN, coeffs >, 31

derive_t
    aerobus::polynomial< Ring, variable_name >, 20
div_t
    aerobus::polynomial< Ring, variable_name >, 21

eq_t
    aerobus::polynomial< Ring, variable_name >, 21
eval
    aerobus::i32::val< x >, 28
    aerobus::i64::val< x >, 30
    aerobus::polynomial< Ring, variable_name >::val< coeffN, coeffs >, 32

gcd_t

aerobus::polynomial< Ring, variable_name >, 21
get
aerobus::i32::val< x >, 29
aerobus::i64::val< x >, 30
gt_t
aerobus::polynomial< Ring, variable_name >, 22

lt_t
aerobus::polynomial< Ring, variable_name >, 22

mod_t
aerobus::polynomial< Ring, variable_name >, 22
monomial_t
aerobus::polynomial< Ring, variable_name >, 23
mul_t
aerobus::polynomial< Ring, variable_name >, 23

pos_t
aerobus::polynomial< Ring, variable_name >, 23

simplify_t
aerobus::polynomial< Ring, variable_name >, 24
src/lib.h, 37
sub_t
aerobus::polynomial< Ring, variable_name >, 24

to_string
aerobus::polynomial< Ring, variable_name >::val< coeffN, coeffs >, 32